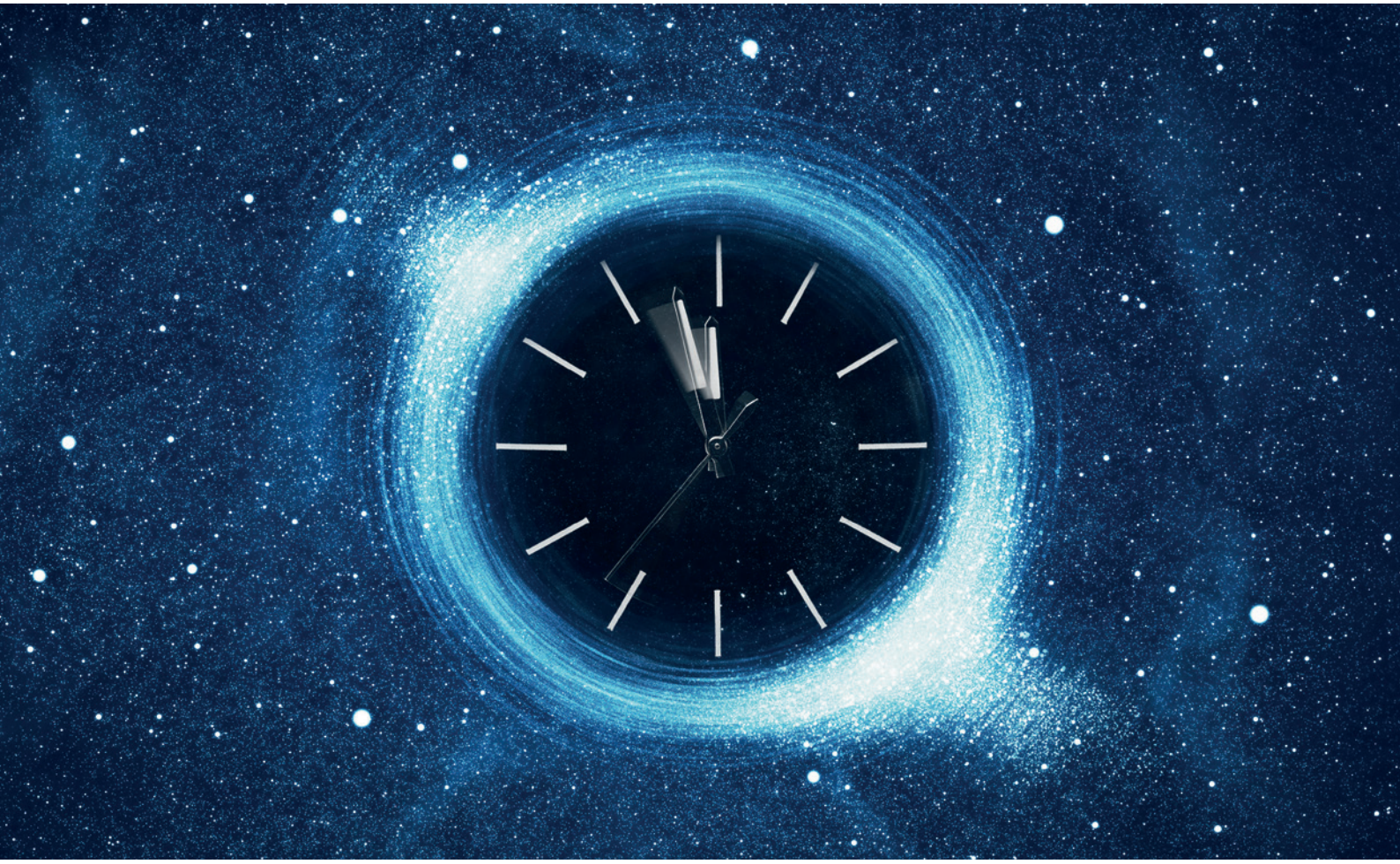

Unbekannte Kostbarkeiten des SDK Heute: Ahead-of-Time Compilation

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Unser heutiges Thema ist die AOT-Kompilierung, die mehrere Jahre Bestandteil des JDK war. Die AOT-Kompilierung ist ganz unzweifelhaft den unbekannten Kostbarkeiten zuzurechnen, da sie so selten verwendet wurde, dass sie nun wegen Nichtverwendung wieder aus dem JDK entfernt wird. Eventuell liegt es auch an wenig sinnvollen, dafür aber umso aufwendigeren Arten der Verwendung. Wir werden dies näher beleuchten.

Ein wenig Geschichte

Das JEP 295 – *Ahead-of-Time Compilation* *[1]* liest sich in der Zusammenfassung als „*compile Java classes to native code prior to launching the virtual machine*". Als explizites Ziel ist die Verringerung der Startzeit sowohl von kleinen als auch von großen Anwendungen ohne spürbare Performanzeinbußen genannt. Zusätzlich sollen Entwicklungsworkflows möglichst nicht geändert werden. Das JEP 295 datiert vom 15. September 2016. Die AOT-Kompilierung wurde im JDK 9 in der Linux-Variante realisiert, die im September 2017 veröffentlicht wurde. Die Windows-Variante folgte mit JDK 10.

Die AOT-Kompilierung basiert intern auf dem Graal-Compiler, der innerhalb des Graal-Projekts *[2]* entwickelt wurde. Graal ist ein JIT-Compiler, der allerdings so flexibel ist, dass er auch ahead of time verwendet werden kann. Durch das JEP 317 – *Experimental Java-based JIT Compiler* *[3]* gelangte Graal als JIT-Compiler in das JDK und durch das

JEP 243 – *Java-Level JVM Compiler Interface* [4] wurde eine Schnittstelle geschaffen, diesen und andere Compiler als Alternative zu den althergebrachten C1- und C2-Compilern des JDK verwenden zu können. Graal dürfte mittlerweile allen Java-Entwicklern namentlich bekannt sein, da er in der GraalVM sowohl als AOT- als auch als JIT-Compiler seinen Dienst verrichtet. Wir haben diese Zusammenhänge bereits beschrieben [5] [6] und auf mehreren Konferenzen, beispielsweise auf dem Java Forum Stuttgart 2018, vorgestellt.

Wie geht AOT-Kompilierung?

Doch wie kann man nun mit einem Standard-JDK der Version 9 (Windows-Version 10) oder höher Klassen oder Anwendungen ahead of time kompilieren? Das `bin`-Verzeichnis des JDK enthält den AOT-Compiler als `jaotc`. Dieser Befehl erwartet neben einigen Optionen eine Liste von Klassen- oder Modulnamen, JAR-Dateien oder Verzeichnisse mit Class-Dateien. Die ersten Gehversuche auf Hello-World-Niveau sind einfach und von uns auf einem Fedora-System durchgeführt worden. Zunächst wird die Klasse kompiliert, da `jaotc` auf Bytecode arbeitet. Dann wird der AOT-Compiler aufgerufen *(siehe Listing 1)*.

Die standardmäßig erzeugte Shared-Object(SO)-Datei im ELF-Format trägt den Namen `unnamed.so`, sodass sich die Vergabe eines sinnvolleren Namens, wie im Beispiel mit `--output` demonstriert, anbietet. Um dann mit der Java Virtual Machine diese SO-Datei zu verwenden, wird sie mit der Option `AOTLibrary` angegeben *(siehe Listing 2)*.

Wenn Sie die Beispiele selbst nachvollziehen wollen, raten wir beim Befehl `jaotc` die Option `--verbose` und beim Befehl `java` die Option `-XX:+PrintAOT` zusätzlich zu verwenden. Die erste Option gestattet einen Einblick in die verschiedenen Teilbereiche des Kompilierens, die zweite Option erlaubt zu verifizieren, dass tatsächlich die SO-Datei geladen und verwendet wird.

Versionen

Im Vorgriff auf das noch näher zu erläuternde JEP 410, das die Entfernung des AOT-Compilers zum Ziel hat, hier eine kleine Versionsanalyse, falls Sie die Beispiele nachvollziehen wollen. Das JEP 410 plant die Entfernung für Java 17. Auf dem Fedora-System des Autors funktionierten die beiden oben genannten Aufrufe von `jaotc` und `java` in den OpenJDK-Versionen 9 bis 13 wie beschrieben. In den Versionen 14 und 15 funktioniert das Kompilieren, allerdings wird die Verwendung mit der JVM mit einer Fehlermeldung quittiert, da die Verwendung der Option `AOTLibrary` als ein experimentelles Feature erklärt ist und der Aufruf daher mit der Option `-XX:+UnlockExperimentalVMOptions` zu erfolgen hat.

In der Version 16 des OpenJDK fehlt das Programm `jaotc` komplett. In der Fedora-Version von Java 16, gebaut von Red Hat, existiert das Programm. Die JVM muss mit der Option des experimentellen Features gestartet werden. Ebenfalls anzumerken ist der Umstand, dass der erzeugte Binärcode nicht rückwärtskompatibel ist, wie wir das von Java gewohnt sind. Die SO-Erzeugung mit `jaotc` in einer Version n kann nicht mit einer JVM-Version größer n verwendet werden. Es erfolgt in diesem Fall ein Rückgriff auf die Bytecode-Variante der Klasse beziehungsweise – falls diese nicht existiert – ein Abbruch des JVM-Aufrufs.

Verwendungen über dem Hello-World-Niveau

Verwendet die Klasse `HelloWorld` eine zweite Klasse, muss diese explizit auch kompiliert werden. Ein transitiver Verwendungsgraph, wie ihn etwa die Native-Image-Erzeugung der GraalVM nutzt, wird nicht erstellt und verwendet.

Die Kompilierung eines ganzen Moduls ist ebenfalls möglich und durch die Option `--module` realisiert. Um etwa das ganze Base-Modul zu kompilieren, führt man den in *Listing 3* gezeigten Befehl aus.

Die so erzeugte Datei hat eine Größe von 318 MB. Selbst die gestripte Version hat noch 194 MB. Bei der Verwendung müssen dann beide SO-Bibliotheken angegeben werden *(siehe Listing 4)*.

Die Verwendung der Option `-XX:+PrintAOT` zeigt hierbei recht eindrucksvoll, wie viele Methoden für ein einfaches Hello-World verwendet werden.

High-End-Features

Die Just-in-Time-Kompilierung der JVM erfolgt gestuft [7]. Die Levels 0 bis 5 decken die reine Interpretation des Bytecodes bis hin zu dem durch den C2-Compiler hochoptimierten Maschinen-Code ab. Damit einher gehen verschiedene Stufen des Profilings der Ausführung des Codes, auf dessen Ergebnisse die Optimierungen basieren. Die `jaotc`-Option `--compile-for-tiered` fügt dem Kompilat Code hinzu, der ein Profiling wie durch den C1-Compiler erzeugten Code des Tier 2 durchführt. Falls der Schwellenwert erreicht wird, wird zur Laufzeit erneut durch C1 mit Tier 3 und dadurch mit dem maximalen Profiling kompiliert, um die spätere Kompilierung durch C2 zu ermöglichen.

Der `jaotc`-Befehl versteht noch eine Reihe weiterer Optionen sowie die Möglichkeit, über eine Textdatei weitere Details der Kompilierung festzulegen. Wir gehen hierauf jedoch nicht weiter ein und

```
jaotc --output helloworld.so  HelloWorld
```

Listing 1

```
java -XX:AOTLibrary=./helloworld.so HelloWorld
```

Listing 2

```
jaotc --output libjavabase.so --module java.base
```

Listing 3

```
java -XX:AOTLibrary=./helloworld.so,./libjavabase.so HelloWorld
```

Listing 4

verweisen den Leser auf einen sehr interessanten Artikel von Ludovic Henry zum Verhältnis von AOT und JIT sowie Details über die Tiered Compilation *[8]*.

Rest in Peace AOT-Kompilierung

Das JEP 410 – *Remove the Experimental AOT and JIT Compiler [7]* hat die Entfernung des AOT- und JIT-Compilers basierend auf Graal zum Ziel. Das JEP formuliert explizit: *„This compiler has seen little use since its introduction and the effort required to maintain it is significant"*. Der Grund ist also schlicht und einfach die geringe Verwendungshäufigkeit und ein sehr hoher Wartungsaufwand. Im Titel dieses Artikels haben wir die JIT-Kompilierung unterschlagen, da diese ja weiterhin durch C1 und C2 vorhanden ist. Der Wegfall der AOT-Kompilierung ist schwerwiegender, sodass wir den Titel entsprechend formulierten.

Nachdem mittlerweile alle bekannten REST-Frameworks (Quarkus, Micronaut, Helidon, Spring) und andere Systeme die Erzeugung nativer Images mithilfe der Graal-VM unterstützen, erscheint es zunächst schwer verständlich, warum die Möglichkeiten der Erzeugung nativen Codes direkt mit dem OpenJDK zu Grabe getragen wird. Ein zweiter Blick macht aber neben dem im JEP 410 genannten hohen Wartungsaufwand das Handling als Grund aus. Wie wir im Beispiel des JDK-Basis-Moduls gesehen haben, übersetzt `jaotc` sämtliche Methoden des Moduls. Bei der GraalVM wird ein transitiver Verwendungs- beziehungsweise Abhängigkeitsgraph erzeugt und nur die tatsächlich verwendeten Klassen, sogar bis auf Methodenebene hinunter, kompiliert. Das erzeugte Kompilat ist damit deutlich kleiner. Außerdem muss nicht der Anwender die zu kompilierenden Klassen angeben, sondern die Frameworks beziehungsweise deren Build-Werkzeuge übernehmen dies. Eine deutliche Vereinfachung und ein ganz offensichtlicher Grund für die geringe Verwendungshäufigkeit der `jaotc`-Kompilierung.

JEP-Prozess

Das JDK Enhancement Proposal (kurz JEP) *[9]* legt einen Prozess zur Weiterentwicklung von Java fest. Ohne das Proposal im Detail ganz gelesen zu haben, scheint uns beim JEP 410 *[7]* nicht alles so gelaufen zu sein, wie man sich das als Außenstehender vorstellt. Das JEP 410 definiert die Entfernung der beiden Compiler für Java 17 als Ziel. Unter der Überschrift *Motivation* liest man dann im JEP 410: *„We have seen little use of these experimental features since they were introduced, and the effort required to maintain and enhance them is significant. These features were not included in the JDK 16 builds published by Oracle, and no one complained"*. Das hört sich unserer Meinung nach etwas willkürlich an nach dem Motto „Wir lassen es mal weg und schauen, ob es jemand merkt" – nicht ganz die feine englische Art. Die Compiler sind allerdings noch im OpenJDK-Repo enthalten und wer sich berufen fühlt, kann sie weiterhin in seinen JDK-Build einbauen, wie es etwa Red Hat praktiziert.

Zusammenfassung

Mit dem JEP 295 wurde mit Java 9 ein Ahead-of-Time-Compiler verfügbar. Mit dem JEP 317 wurde mit Java 10 ein in Java geschriebener Just-in-Time-Compiler verfügbar. Beide Compiler basieren auf Graal, dem Compiler, der auch in der GraalVM verwendet wird und für sie namensgebend war. Während der AOT-Compiler durch ein Programm im `bin`-Verzeichnis des JDK realisiert wird, wird der JIT-Compiler durch eine Option beim Aufruf der JVM aktiviert. Beide Compiler gehören zu den unbekannten Kostbarkeiten und wurden in der Praxis von uns, den Entwicklern, so selten verwendet, dass sie mit dem JEP 410 mit Java 17 wieder aus dem JDK entfernt werden. Da Java 11 als LTS sicher noch einige Jahre Verwendung finden wird, können der AOT- und der JIT-Compiler noch weiter genutzt werden. Das JEP 243, die JVM-Schnittstelle zur Verwendung eines externen JIT-Compilers, wird im JEP 410 explizit von der Löschung ausgenommen, sodass es sein könnte, dass irgendwann in der Zukunft der Graal-Compiler wieder über diese Schnittstelle der JVM zur Verfügung steht.

Referenzen

[1] JEP 295: Ahead-of-Time Compilation, *https://openjdk.java.net/jeps/295.*

[2] Graal Project, *https://openjdk.java.net/projects/graal*

[3] JEP 317: Experimental Java-Based JIT Compiler, *https://openjdk.java.net/jeps/317*

[4] JEP 243: Java-Level JVM Compiler Interface, *https://openjdk.java.net/jeps/243*

[5] Bernd Müller, Native Images mit GraalVM, Java aktuell 02/2020

[6] Bernd Müller, Unbekannte Kostbarkeiten des SDK – Heute: Just-in-Time Compilation, Java aktuell 04/2019

[7] JEP 410: Remove the Experimental AOT and JIT Compiler, *https://openjdk.java.net/jeps/410*

[8] Ludovic Henry. AOT Compilation in HotSpot: Introduction, *https://devblogs.microsoft.com/java/aot-compilation-in-hotspot-introduction/*

[9] JEP draft: JEP 2.0, draft 2, *https://cr.openjdk.java.net/~mr/jep/jep-2.0-02.html*

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.