Ausgabe 01/2021

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 01

Java aktuell

iJUG Verbund
www.ijug.eu

Java 15
Die Features im Überblick

Java-Libraries
fritz2, Vavr und Kotlin Arrow

Spring
Spring Data JPA, Spring HATEOAS und Spring mit React

WERKZEUGE
Tools & Frameworks

iJUG Verbund

Werden Sie Mitglied im iJUG!

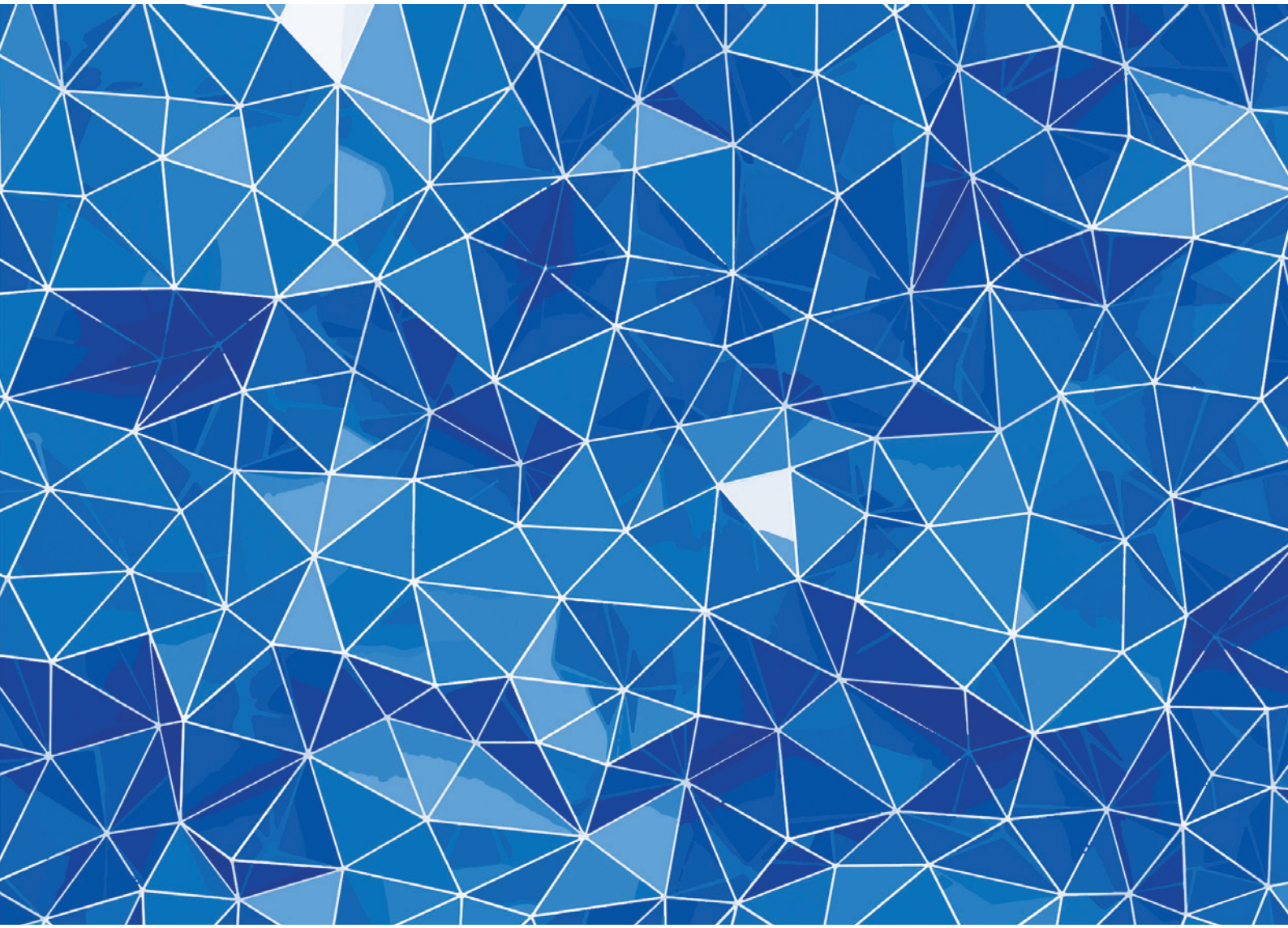
Ab 15,00 EUR im Jahr erhalten Sie

30 % Rabatt auf Tickets der JavaLand

Jahres-Abonnement der Java aktuell

Mitgliedschaft im Java Community Process

www.ijug.eu



Native Images mit GraalVM: Reflection

Bernd Müller, Ostfalia

In der Java aktuell 02/20 [1] haben wir begonnen, die Erzeugung nativer Images mit der GraalVM [2] [3] vorzustellen. Der Artikel hatte einführenden Charakter und beleuchtete auch die Hintergründe der Ahead-of-Time-Compilierung mit Graal beziehungsweise der GraalVM. Heute werden wir etwas weitergehen und zeigen, wie gut reflektive Programme bei der Erzeugung nativer Images unterstützt werden.

Was wir bereits können

Den einführenden Artikel haben wir mit einem Hello-World-Programm abgeschlossen, das zur Erinnerung noch einmal in *Listing 1* angegeben ist.

Es geht über ein einfaches Hello-World hinaus, da es ein wenig von Javas Reflection-API verwendet. Der Grund hierfür ist, dass wir zunächst verdeutlichen wollen, was bei der Native-Image-Erzeugung automatisch geschieht, um danach zu verdeutlichen, was und vor allem warum es nicht funktioniert. Das Reflection-API erlaubt uns, Java-Code zur Laufzeit zu untersuchen und sogar zu verändern, was bei nativem Code prinzipiell nicht möglich ist, zumindest nicht in

der Form, wie wir es bei Java gewohnt sind. Der Code in *Listing 1* wird vom `native-image`-Befehl problemlos in nativen Code kompiliert und kann danach ausgeführt werden. Warum? Sowohl die `main()`-Methode als auch die über Reflection aufgerufene Methode `helloWorld()` befinden sich in derselben Klasse beziehungsweise dem kompilierten Byte-Code. Die Methoden `getDeclaredMethode()` sowie `invoke()` sind gewöhnliche Methoden, die eventuell auch vom JIT-Compiler in nativen Code übersetzt werden würden. Warum sollte das nicht auch durch den AOT-Compiler realisiert werden können?

Problematisch: Nicht verwendete Klassen

Das Problem des Native-Image-Compilers ist seine statische Programmanalyse. Um nicht alle Klassen im Klassenpfad kompilieren und linken zu müssen, führt der Compiler eine daten- und kontrollflussbasierte Analyse auf Basis einer Closed-World-Assumption durch: Alles was erreichbar ist, wird auch verwendet, also kompiliert. Damit wird alles, was nicht erreichbar ist, nicht kompiliert. Ausgangspunkt ist die Main-Methode. Alle verwendeten Klassen und deren transitive Hülle wird kompiliert und gelinkt. Die Analyse findet jedoch nicht nur auf Klassenebene, sondern sogar auf Methodenebene statt. Nicht verwendete Methoden finden nicht den Weg in das Kompilat. Um dies besser zu verstehen, dient die Klasse `ReflectiveInteger` in *Listing 2* als Beispiel.

Die Klasse `ReflectiveInteger` ruft in der `main()`-Methode alternativ eine von zwei Methoden auf, die jeweils eine Instanz der Klasse `Integer` erzeugen und zwar durch den reflektiven Aufruf des entsprechenden Konstruktors der Klasse. Der Konstruktor ist zwar mittlerweile deprecated (veraltet), was jedoch für das Beispiel ohne Belang ist. Auch der weitere Code der beiden praktisch identischen Methoden spielt keine Rolle. Lediglich der Parameter der Methode `forName()` ist von Interesse.

In der ersten der beiden Methoden ist der Parameter ein String-Literal. Durch die Analyse wird erkannt, dass die Klasse `java.lang.Integer` verwendet wird. Sie wird ebenfalls kompiliert und

```
public class HelloWorld {
  public static void main(String[] args) throws Exception {
     Method method = HelloWorld.class
        .getDeclaredMethod("helloWorld", new Class[] {});
        method.invoke(null, new Object[] {});
  }

  private static void helloWorld() {
     System.out.println("Hello World");
  }
}
```

Listing 1: Hello-World mit Reflection

```
public class ReflectiveInteger {

  public static void main(String[] args) throws Exception {
     createInteger();
     //createInteger("java.lang.Integer");
  }

  private static void createInteger() throws Exception {
     Class<?> clazz = Class.forName("java.lang.Integer");
     Constructor<?> constructor = clazz.getConstructor(new Class[] { String.class });
     Object instance = constructor.newInstance(new Object[] { "42" });
     System.out.println("Mit createInteger() erzeugt: " + instance);
}


  private static void createInteger(String javaLangInteger) throws Exception {
     Class<?> clazz = Class.forName(javaLangInteger);
     Constructor<?> constructor = clazz.getConstructor(new Class[] { String.class });
     Object instance = constructor.newInstance(new Object[] { "42" });
     System.out.println("Mit createInteger(String) erzeugt: " + instance);
  }

}
```

Listing 2: Die Klasse ReflectiveInteger

```
[{
  "name" : "java.lang.Integer",
  "allDeclaredConstructors" : true,
  "allPublicConstructors" : true,
  "allDeclaredMethods" : true,
  "allPublicMethods" : true
}]
```

Listing 3: Konfigurationsdatei für java.lang.Integer

```
[{
  "name":"java.lang.Integer",
  "methods":[{"name":"<init>","parameterTypes":["java.lang.String"] }]
}]
```

Listing 4: Konfigurationsdatei nur mit Konstruktor

```
public class ReflectiveSomeClass {

  public static void main(String[] args) throws Exception {
     useSomeClass("de.pdbm.reflection.SomeClass");
  }

  private static void useSomeClass(String someClass) throws Exception {
     Class<?> clazz = Class.forName(someClass);
     Constructor<?> constructor = clazz.getConstructor(new Class[] { });
     Object instance = constructor.newInstance(new Object[] { });
     Method method = clazz.getMethod("sayHello", String.class);
     System.out.println("Antwort von ‚SomeClass': " + method.invoke(instance, "world"));
  }

}
```

Listing 5: Laden und Verwenden einer Anwendungsklasse

```
public class SomeClass {

  public String sayHello(String arg) {
     return "Hello " + arg;
  }
}
```

Listing 6: Einfache Anwendungsklasse

gelinkt. Das Programm ist ohne weiteren Aufwand mit dem Befehl `native-image` kompilierbar, das Kompilat ausführbar.

Das Ganze ändert sich, wenn in der Main-Methode die erste Methode auskommentiert und stattdessen die zweite verwendet wird. Während des Kompilierens wird eine Warnung ausgegeben, die explizit erwähnt, dass `forName()` aufgerufen und daher kein Stand-alone-Executable erzeugt wird. Stattdessen wird ein Executable erzeugt, dass als Fall-Back ein installiertes JDK benötigt. Das Executable kann aber ausgeführt werden und verhält sich wie erwartet. Im Hintergrund wird hierbei von der Fall-Back-JVM die Klasse `java.lang.Integer` geladen.

Eine weitere Warnung gibt den Hinweis, dass die Fall-Back-Lösung verhindert werden kann, indem der Befehl `native-image` mit der Option `-no-fallback` aufgerufen wird. Das so erzeugte Executable bricht bei der Ausführung dann mit einer `ClassNotFoundException` für `java.lang.Integer` ab.

Um ein funktionsfähiges Stand-alone-Executable zu erhalten, muss dem Compiler mitgeteilt werden, dass die Klasse `java.lang.Integer` zu kompilieren und zu linken ist. Dies wird mit der Option `-H:ReflectionConfigurationFiles` erreicht, die eine Liste von JSON-Dateien übergeben bekommt. Die Konfigurationsdatei in *Listing 3* sorgt dafür, dass die Klasse `java.lang.Integer` komplett, das heißt mit allen Konstruktoren und Methoden kompiliert wird.

Ein genauer Blick in *Listing 2* zeigt aber, dass lediglich der Konstruktor mit String-Parameter verwendet wird. Man kann also deutlich restriktiver konfigurieren. Das *Listing 4* zeigt eine Konfigurationsdatei, die lediglich diesen Konstruktor verwendet.

Beide Executables funktionieren wir erwartet und unterscheiden sich lediglich in ihrer Größe. Die Version mit der kompletten Integer-Klasse ist 73 kB größer, was allerdings bei einer Gesamtgröße von zirka 10 MB nicht ins Gewicht fällt.

Werkzeugunterstützung

Das Schreiben der JSON-Datei ist nicht besonders attraktiv. Wir könnten für alle benötigten Klassen immer die erste Alternative mit allen Konstruktoren und Methoden verwenden, was aber bei vielen Klassen dann tatsächlich die Größe des Executable spürbar erhöht. Die zweite Alternative händisch zu erstellen, macht bei größeren Klassen mit mehreren verwendeten Methoden definitiv keinen Spaß. Glücklicherweise unterstützt uns die GraalVM beim Erstellen der Konfigurationsdatei. Der Agent `native-image-agent` erzeugt neben der Konfigurationsdatei für Reflection auch noch entsprechende Dateien für JNI-, Proxy- und Ressourcen-Zugriffe.

Noch interessanter wird das Ganze, wenn man diese Konfigurationsdateien in das Verzeichnis `/META-INF/native-image` schreibt. Beim Erzeugen des Executable schaut der Compiler automatisch in dieses Verzeichnis und verwendet die zuvor geschriebenen Dateien mit den Namen `jni-config.json`, `proxy-config.json`, `reflect-config.json` und `resource-config.json` automatisch.

Die Klasse `java.lang.Integer` gehört zum JDK. Können auch beliebige Anwendungsklassen analog verwendet werden? Ja! Es gibt keinen Unterschied zwischen JDK- und Anwendungsklassen. Das *Listing 5* zeigt ein einfaches Beispiel für das Laden und Verwenden einer Anwendungsklasse. *Listing 6* zeigt die recht einfach gehaltene Anwendungsklasse selbst.

Die beiden Klassen in den Listings 5 und 6 gleichen dem einführenden Beispiel mit `java.lang.Integer` und hätten nicht unbedingt dargestellt werden müssen. Sie dienen uns aber auch als Beispiel für eine weitere Alternative, wie zusätzlich zu kompilierende und zu linkende Klassen angegeben werden können, nämlich programmatisch.

```
@AutomaticFeature
public class RuntimeReflectionRegistrationFeature implements Feature {

  @Override
  public void beforeAnalysis(BeforeAnalysisAccess access) {
     try {
        RuntimeReflection.register(SomeClass.class);
        RuntimeReflection.register(SomeClass.class.getConstructor());
        RuntimeReflection.register(SomeClass.class.getDeclaredMethod("sayHello", String.class));
     } catch (NoSuchMethodException e) {
        System.out.println(e.getMessage());
     }
  }

}
```

Listing 7: Mit @AutomaticFeature annotierte Klasse

```
[{
  "name":"de.pdbm.reflection.SomeClass",
  "methods":[
     {"name":"<init>","parameterTypes":[] },
     {"name":"sayHello","parameterTypes":["java.lang.String"] }
  ]
}]
```

Listing 8: Konfiguration für Klasse SomeClass

Automatische Features

Das Interface-Feature erlaubt es, die Native-Image-Erzeugung zu „intercepten" und Custom-Code einzubauen. Dies ist allerdings eher für Framework-Entwickelnde gedacht. Die Annotation `@AutomaticFeature` kann jedoch verwendet werden, um Klassen, die Feature implementieren, automatisch über die entsprechenden `Feature`-Methoden aufzurufen. Das *Listing 7* zeigt die Klasse `RuntimeReflectionRegistrationFeature`, die mit `@AutomaticFeature` annotiert ist.

Die Methode `beforeAnalysis()` ist eine Call-Back-Methode des Native-Image-Compilers und wird vor der Code-Analyse aufgerufen. In ihr werden die Klasse `SomeClass` selbst sowie deren Konstruktor und die `sayHello()`-Methode programmatisch für das Kompilieren registriert. Sie ist damit das Äquivalent zur JSON-Konfigurationsdatei, die in *Listing 8* abgebildet ist.

Wer soll das alles machen (wollen)?

Wir haben gesehen, dass das dynamische Laden einer Java-Klasse zur Laufzeit bei der Erzeugung und Verwendung nativer Images nicht funktioniert und prinzipiell nicht funktionieren kann. Ist jedoch zur Compile-Zeit bekannt, welche Klassen zur Laufzeit geladen werden sollen, so können diese kompiliert, gelinkt und später auch genutzt werden. Leider ist es recht aufwendig, die Klassen zu bestimmen. Soll die Code-Erzeugung auf tatsächlich verwendete Methoden beschränkt werden, so ist dies noch aufwendiger, da sämtliche Verwendungen explizit aufzuzählen sind. Glücklicherweise kommt die GraalVM mit einem Agenten, der uns diese Arbeit abnimmt und die entsprechenden Konfigurationsdateien für uns erzeugt.

Trotzdem bleibt ein nicht zu unterschätzender Aufwand, der bei einer so dynamischen Sprache wie Java geleistet werden muss, wenn sie Ahead-of-Time kompiliert werden soll. Wenn wir aber aufmerksam sind und das Java-Ökosystem anschauen, sind wir auf dem richtigen Weg. So wie wir heutzutage Java-Anwendungen nicht mit `javac` kompilieren, sondern IDEs, Maven und Gradle verwenden, so werden wir auch nicht die Konfigurationsdateien zur Native-Image-Erzeugung erstellen, sondern sie uns von entsprechenden Werkzeugen automatisch generieren lassen. Wir müssen uns nur Quarkus anschauen, um zu sehen, wie gut das bereits funktioniert.

Zusammenfassung

Die Native-Image-Erzeugung mit der GraalVM ist bereits sehr weit fortgeschritten. Klassen, die bei der statischen Code-Analyse des Compilers jedoch nicht nachweislich verwendet werden, finden nicht den Weg in das Executable. Sie müssen explizit über verschiedene Alternativen als zu kompilierend gekennzeichnet werden. Die dazu nötigen Konfigurationsdateien können über einen Java-Agenten erzeugt werden. Eine programmatische Registrierung dieser Klassen ist ebenfalls möglich.

Weiterführende Links

[1] Bernd Müller: „Native Images mit GraalVM", Java aktuell 02/2020, Seite 25 *https://backoffice.doag.org/formes/pubfiles/12161133/docs/Publikationen/Java-Aktuell/2020/02-2020/02-2020-Java-aktuell-WEB.pdf*

[2] *https://www.graalvm.org*

[3] *https://github.com/oracle/graal/*

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.

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu

JavaLand

16. - 18. März 2021
in Brühl bei Köln

Programm jetzt online

Hybride Veranstaltung

Was die JavaLand als Plattform für Wissentransfer und Networking ausmacht, kannst du vor Ort im Phantasialand oder online erleben. Als Teilnehmer entscheidest du selbst!

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu