Quick contact: +49 89 954 5718 10 office@javatraining.de

GraalVM – Java und JavaScript Anwendungen als Windows Binary

Veröffentlicht von Michael Schaffler-Gloessl beim Juni 19, 2020 Tags Kategorien

Jeder Java Programmierer kennt die Java Virtual Machine (JVM) und benötigt sie, um Java Programme laufen lassen zu können. Seit Anbeginn gibt es unterschiedliche Implementierungen von Sun, IBM, HP, Google, BEA, um nur einige zu nennen. In letzter Zeit macht jedoch eine weitere Implementierung von sich reden, deren Wurzeln sogar in Österreich liegen: die GraalVM. Insbesondere die Ahead-Of-Time-Compilation (AOT) rückt hier in das Interesse der Entwickler. Kann die native Compilierung in eine Binary die Erwartungen erfüllen?

Die Ideen einer alternativen JVM wurde schon zu Zeiten gelegt in denen Java noch das bekannteste Aushängeschild von Sun Microsystems war. Das Ziel war ursprünglich, eine vollständig in Java implementierte JVM zu programmieren anstatt diese wie bisher in C/C++ zu realisieren.

Wie die Johannes Keppler Universität in Linz (JKU) berichtet, wurde die Basis der heutigen GraalVM im Jahr 2010 durch die Doktoranden Thomas Würthinger und Lukas Stadler gelegt.

Das ursprüngliche Ziel war die Implementierung eines Just-In-Time-Compilers, der für eine effiziente Interaktion zwischen unterschiedlichen Programmiersprachen innerhalb der JVM sorgt. Diese Forschungsarbeiten mündeten schließlich in einer Kooperation mit Sun / Oracle, die bis heute andauert. Oracle betreibt dazu ein Forschungslabor direkt an der JKU in Linz.

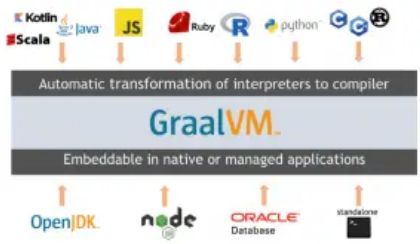
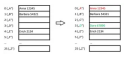


Abbildung 1 Schematische Darstellung GraalVm (c) Oracle

Moment, wie war das? Unterschiedliche Programmiersprachen innerhalb der Java Virtual Machine?

Dazu muss man wissen, dass die Java Virtual Machine nicht zwangsweise an Java gebunden ist. Sie exekutiert eigentlich den sogenannten Bytecode (welchen der Programmierer als Class File kennt). Dieser Bytecode wird durch den Java Compiler (javac) erzeugt und wird bei der Ausführung eines

Development Tool Suites
Dezember 4, 2020


Hashing
November 19, 2020


Android UsageStatsManager
September 15, 2020

JPA Java Persistence API HIBERNATE
Dynamische Query mit Spring und JPA Specification
August 30, 2020

Dynamische Formulare mit Angular Reactive Forms
August 11, 2020

CSS verstehen: z-index genauer betrachtet
Juli 22, 2020

Programms zunächst von der JVM interpretiert, bis er schließlich von dem in der JVM integrierten Compiler von Bytecode in nativen Maschinencode übersetzt wird. Warum? Richtig, Performance! In der Sun JVM, später Oracle JVM trägt dieser Compiler den Namen Hotspot Compiler. Er übersetzt die „Hotspots“, also die kritischen Codestellen des Programms in optimierten Code, den die CPU direkt ausführen kann. Damit erreicht die JVM ähnliche, manchmal sogar bessere Ausführungszeiten wie ein C Programm.

Ebendieser Compiler wurde in der GraalVM neu implementiert. Desweiteren hat die GraalVM Vorteile, wenn man andere Sprachen auf der JVM ausführen möchte. Warum möchte man das tun? Nun ja, wenn man eine Programmiersprache auf der JVM ausführen kann, ergeben sich die folgenden Vorteile:

- Man kann das Programm überall dort ausführen, wo eine JVM verfügbar ist
- Man kann darunterliegende JVM Funktionalität nutzen, wie z.B. Multithreading
- Man kann aus seinem Programm auf vorhandene Java Bibliotheken zugreifen und damit auf das wohl größte Software Archiv der IT (eine potenzielle Nachfolgesprache von Java wird daher mit Sicherheit auf der JVM ausführbar sein müssen, siehe Kotlin)

Ein Compiler könnte also jede beliebige Sprache in Bytecode übersetzen oder aber es kann auch ein Interpreter für Scriptsprachen als Bytecode auf der JVM betrieben werden.

Inwiefern unterstützt die GraalVM unterschiedliche Programmiersprachen nun besser?

Steve Yegg beschreibt ein wichtiges Problem, das man bei dynamischen Skriptsprachen, die auf einer JVM laufen, lösen möchte: Wie ruft man in einer VM aus einer Scriptsprache eine Funktion einer anderen Scriptsprache auf? Steve Yegg, der die noch aus Netscape Zeiten stammende Rhino JavaScript Engine auf die JVM portierte, löste elegant, wie aus Java heraus JavaScript Funktionen aufgerufen werden können und wie umgekehrt aus JavaScript heraus Java Klassen instanziiert und aufgerufen werden können. Ja selbst die Implementierungen von Java Interfaces als JavaScript Objekte wurden in Rhino gelöst. Letztendlich ist Rhino ein Parser und Interpreter, der auf der JVM ausgeführt wird.

Zurück zur GraalVM. GraalVM unterstützt also eine Integration unterschiedlicher Sprachen innerhalb der JVM und verfügt über einen ByteCode Compiler, der selbst in Java geschrieben und in der Lage ist mittels Ahead-Of-Time-Compilation gewöhnlichen Java Code in eine Binary (z.b. Windows exe) zu übersetzen.

Diese Kern-Funktionalitäten wollen wir nun testen.

Installation der GraalVm

Zunächst installieren wir die Graalvm mit JDK 11 von github. Am leichtesten findet man die Binaries über http://www.graalvm.org/download. **WICHTIG: Bitte Version ab 20.2 installieren** (ggf. nightly build), davor funktioniert der native Build Prozess unter Windows nicht.

Die Windows Version ist als zip Datei verfügbar. Nach dem Entpacken gehen wir in das bin Verzeichnis des graalvm jdk und prüfen die Version:

```
C:\graalvm\bin>java -version
openjdk version "11.0.7" 2020-04-14
OpenJDK Runtime Environment GraalVM CE 20.2.0-dev (build 11.0.7+10-jvmci-20.1-
b02)
OpenJDK 64-Bit Server VM GraalVM CE 20.2.0-dev (build 11.0.7+10-jvmci-20.1-b02,
mixed mode, sharing)
```

Ein erster Test des integrierten JavaScript Interpreters ist auf der Kommandozeile möglich:

```
C:\graalvm\bin>js
> 1 + 1
2
> var a = 1 + 1;
```

```
> a
2
> var o = {name:"Michael", email:"michael@java.at"};
> JSON.stringify(o);
{"name":"Michael","email":"michael@java.at"}
>
```

Nun wollen wir testen, ob wir einen JavaScript Code aus Java heraus ausführen können:

```
package at.ciit;

import javax.naming.Context;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestGraalJs {
    public static void main(String[] args) throws ScriptException {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("javascript");

        ScriptContext context = engine.getContext();

        Object eval = engine.eval("var a = -5 + 3; Math.abs(a);");
        engine.put("b","Hello JavaScript World;");
        engine.eval("function square(i){return i*i;}");
        engine.eval("var c = square(a)");

        System.out.println("JavaScript Implementation: " +
engine.getClass().getName()); //GraalJSScriptEngine
        System.out.println("Result of type: " + eval.getClass().getName());
        System.out.println("Result: " + eval);
        System.out.println("Variable a in engine: " + engine.get("a"));
        System.out.println("Variable b in engine: " + engine.get("b"));
System.out.println("Variable c in engine: " + engine.get("c"));
    }
}
```

Obenstehender Code funktioniert auch schon in Java 8. Dort wird allerdings die Nashorn JavaScript Engine verwendet, die mit JDK 11 deprecated ist und mit JDK 15 entfernt wird. Die GraalVM ist hier wohl als Ersatz gedacht.

Ich nächsten Schritt entwickeln wir uns einen kleinen Performance Benchmark und versuchen diesen in eine native Windows exe zu übersetzen. Zu diesem Zweck kann man sich in seiner Lieblings IDE einfach das neu installierte JDK mit GraalVM als Projekt JDK konfigurieren.

Unser Code sieht wie folgt aus:

```
package at.ciit;

import javax.swing.plaf.basic.BasicInternalFrameTitlePane;
import java.beans.XMLEncoder;
import java.time.Clock;
import java.time.Instant;
import java.time.LocalDate;
import java.util.Base64;
import java.util.Comparator;
import java.util.List;
import java.util.Random;
import java.util.function.Function;
```

```
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class TestGraalVm {

    //java -Xmx5g -Xms5g at.ciit.TestGraalVm 50000000
    public static void main(String[] args) {
        for (int j = 0; j < 2; j++) {


            long start = System.currentTimeMillis();

            int sum = new Random().ints(Integer.valueOf(args[0]))
                    .mapToObj(randomNumber -> new
SomeDataClass(String.valueOf(randomNumber), randomNumber,
LocalDate.ofEpochDay(randomNumber)))
                    .sorted(Comparator.comparing(SomeDataClass::getDate))
                    .map(d -> d.toString())
                    .map(s -> s.length())
                    .mapToInt(i -> i)
                    .sum();
            System.out.println("Some useless number: " + sum);

            long duration = System.currentTimeMillis() - start;
            System.out.println("Duration in ms: " + duration);
        }
    }
}
class SomeDataClass {
    String string; double number; LocalDate date;

    public SomeDataClass(String string, double number, LocalDate date) {
        this.string = string; this.number = number; this.date = date;
    }

    public String getString() { return string; }
    public double getNumber() { return number; }
    public LocalDate getDate() { return date; }
    public String toJson(){return new
StringBuilder().append(string).append(",").append(number).append(",").append(date).t

}
```

Der Code soll nur ein einfaches Beispiel geben, um einen ersten Eindruck zur Performance der unterschiedlichen JVMs zu geben. Wenn wir das Programm nachher im „normalen“ JDK, also mit dem Hotspot JIT Compiler laufen lassen, müssen wir um ein Warmlaufen zu gewährleisten, den Code in einer Schleife 2 mal durchlaufen. Tatsächlich laufen hier das Adopt Open JDK und die GraalVM mit gleicher Performance. Beide benötigen für die Verarbeitung der 5 Millionen Datensätze in der 2ten Iteration auf dem Testrechner in etwa 5 Sekunden. Damit hat der in Java geschriebene JIT Compiler der GraalVM schon einmal eine Hürde genommen. Er produziert zumindest in unserem einfachen Test ein gleichwertiges Ergebnis wir der Hotspot Compiler.

```
Some useless number: 149334773
Duration in ms: 5940
Some useless number: 149332948
Duration in ms: 5021
```

Nun wollen wir aber eine Ahead-Of-Time-Complilation testen. Da dies von den Entwicklungsumgebungen derzeit noch nicht unterstützt wird, gehen wir auf die Kommandozeile und installieren in der GraalVM zunächst das Native Image:

```
C: \graalvm\bin>gu install native-image
Downloading: Component catalog from www.graalvm.org
Processing Component: Native Image
Downloading: Component native-image: Native Image  from github.com
Installing new component: Native Image (org.graalvm.native-image, version
20.1.0)
```

Wir fügen zunächst das bin Verzeichnis der GraalVM zum Windows Pfad hinzu:

```
C: \graalvm\bin> set PATH=C:\graalvm\bin;%PATH%
```

Nun gehen wir in unser Projektverzeichnis:

```
C: \graalvm\bin> cd \Users\michael\Documents\work\graalvmtest\src
```

Wir kompilieren unser Java Programm zunächst mit dem javac Compiler:

```
javac at\ciit\TestGraalVm.java
```

Und testen das Programm nochmals als Java Programm:

```
java at.ciit.TestGraalVm 5000000

Some useless number: 149334771

Duration in ms: 6419
```

Wenn wir nun versuchen, das Programm in nativen Code umzuwandeln bekommen wir auf einem reinen Java Entwicklungsrechner das folgende Resultat:

```
native-image at.ciit.TestGraalVm
[at.ciit.testgraalvm:14844]    classlist:   1,604.33 ms,  0.96 GB
[at.ciit.testgraalvm:14844]        setup:     950.64 ms,  0.96 GB
Error: Default native-compiler executable 'cl.exe' not found via environment
variable PATH
Error: Use -H:+ReportExceptionStackTraces to print stacktrace of underlying
exception
Error: Image build request failed with exit status 1
```

Was ist cl.exe? Ein Build Tool, das in VisualStudio enthalten ist und den C++ Compiler und Linker ansteuert. Also müssen wir noch Microsoft Visual Studio installieren. Wir laden und installieren die MS Visual Studio Community Edition von https://visualstudio.microsoft.com/de/downloads/ und installieren sie.

Nun müssen wir die Umgebungsvariablen der Build Tool Chain von Visual Studio setzen:

```
"C:\Program Files (x86)\Microsoft Visual
Studio\2019\Community\VC\Auxiliary\Build\vcvars64.bat"

Wir versuchen es nochmals:

native-image at.ciit.TestGraalVm
```

...

Der Übersetzungsvorgang klappt, unser Windows Programm steht im aktuellen Working-Directory unter „at.ciit.testgraalvm.exe“. Wir können also die Früchte unserer Arbeit ernten:

```
at.ciit.testgraalvm.exe 5000000
Some useless number: 165000000
Duration in ms: 7522
Some useless number: 165000000
Duration in ms: 7088
```

Was ist das? Um gute 40% langsamer als der gute alte Sun Hotspot Compiler?

Nun es bestätigt sich hier vielleicht, was viele Compiler Experten sagen:

Die Optimierung des Codes durch einen JIT (Just In Time) Compilers während der Laufzeit hat wesentliche Vorteile gegenüber der statischen Compilierung durch einen AOT (Ahead Of Time) Compiler. Es ist wie mit dem Märchen vom schnellen C Code und dem langsamen Java Code, das sich nach genauerer Betrachtung bei den modernen Umgebungen als falsch herausgestellt hat.

Vielleicht müssen wir aber auch nur etwas warten, bis die GraalVM ein bisschen ausgereifter ist.

Michael Schaffler-Glößl

09.06.2020

Unsere Schulung zum Themenkreis:

- GraalVM – Einführung und Einsatz

Zusammenhängende Posts

August 24, 2021

CIIT GmbH im Wirtschaftstalk

Weiterlesen

Juni 7, 2021

Individuelle Softwareentwicklung für APF – Agentur für Passagier- und Fahrgastrechte

Weiterlesen

März 20, 2021

Zuordnen einer einzelnen Entität zu mehreren Tabellen

Weiterlesen

Kontakt und Impressum