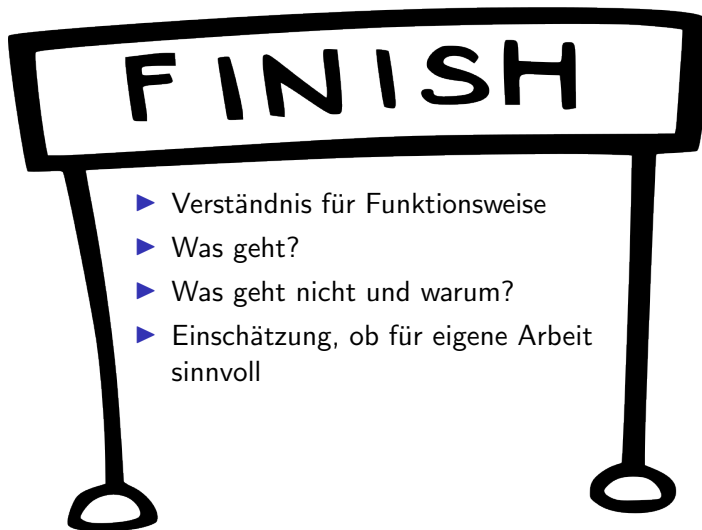


Was die **GraalVM**™ nativ schon so alles kann

Bernd Müller
Ostfalia







Vorstellung Referent

- ▶ Prof. Informatik (Ostfalia, HS Braunschweig/Wolfenbüttel)
- ▶ Buchautor (JSF, Seam, JPA, ...)



- ▶ Mitglied EGs JSR 344 (JSF 2.2) und JSR 338 (JPA 2.1)
- ▶ Geschäftsführer PMST GmbH
- ▶ JUG Ostfalen (Mitorganisator)
- ▶ bernd.mueller@ostfalia.de
- ▶  @berndmuller
- ▶  BerndMuller

Ist Java schnell ?



warum ?

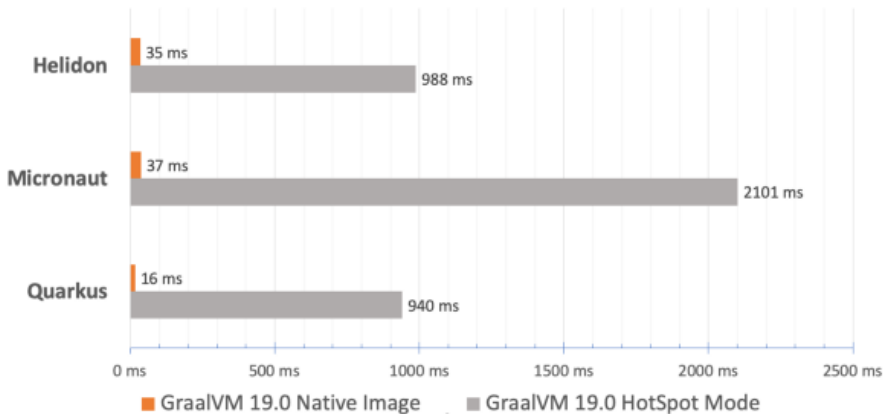
Die Anforderungen ändern sich



Motivation: Startup Time

Java Microservice: Startup Time

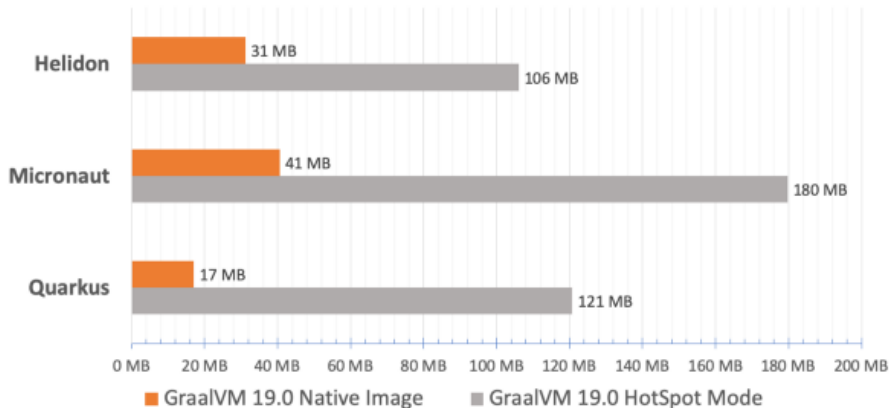
~50x faster



Motivation: Memory Footprint

Java Microservice: Memory Footprint

~5x lower



Die GraalVM im Überblick

- ▶ Laufzeitumgebung für Java, JavaScript, Ruby, Python, R, WebAssembly ... und LLVM-Sprachen
- ▶ Graal (Geschichte und Gegenwart)
 - ▶ JIT- und AOT-Compiler (in Java implementiert)
 - ▶ Als evtl. Ersatz für C2 in Hotspot-VM gedacht
 - ▶ JEP 243: Java-Level JVM Compiler Interface (10/2014)
 - ▶ JEP 295: Ahead-of-Time Compilation (9/2016)
 - ▶ JEP 317: Experimental Java-Based JIT Compiler (10/2017)
 - ▶ Wurde aber nichts draus ...
Removal of experimental features AOT and Graal JIT (10/2020)

Die GraalVM im Überblick (cont'd)

- ▶ Truffle
 - ▶ Implementierungshilfe für beliebige Sprachen
 - ▶ Interpreter für AST (high level)
- ▶ SubstrateVM
 - ▶ Kleine VM (in Java), die mit in natives Executable compilieren wird
 - ▶ Enthält GC, Thread-Scheduling, Code Caches, ...
 - ▶ Damit zwei Optionsklassen
 - ▶ Hosted Options (-H:): Konfiguriert Boot-Image-Erzeugung
 - ▶ Runtime Options (-R: -XX): Initiale Werte während Boot-Image-Erzeugung und Laufzeit

Wie funktioniert Native-Image-Erzeugung ?

- ▶ Erzeugt ELF oder Mach-O (Windows experimentell)
- ▶ Analysiert alle Klassen der Applikation plus Abhängigkeiten (SDK, Bibliotheken)
- ▶ Also **statische** Analyse, um zu bestimmen, welche Klassen und Methoden bei Programma**usführung** verwendet werden
- ▶ Dieser – und nur dieser – Code wird AOT in natives Image kompiliert
- ▶ Man spricht auch von *Closed World Assumption*

Details ...

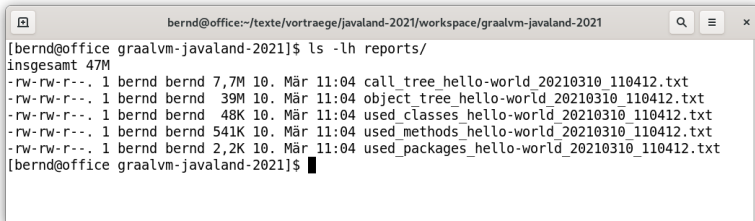
- ▶ Points-to-Analyse
Findet erreichbare Klassen, Methoden und Fields
- ▶ Zwei Ergebnisse
 - ▶ Call-Tree
 - ▶ Image-Object-Tree
- ▶ Call-Tree
Graph von Methodenaufrufen. Damit nicht erreichbare Blöcke bekannt. Werden nicht kompiliert.
- ▶ Image-Object-Tree
Objekte im nativen Image-Heap. Wurzeln z.B. static Fields oder Graph von Methoden, die Konstanten enthalten

Hello World mit GraalVMs native-image

1. `$ javac HelloWorld.java`
2. `$ native-image HelloWorld hello-world`
3. `./hello-world`
4. Größe dynamisch gelinkt 7,7 MB
5. Größe statisch gelinkt 9,6 MB

Überprüfung der Points-to Analysis

- ▶ Call-Tree: Graph von Methodenaufrufen
Option zur Ausgabe: `-H:+PrintAnalysisCallTree`
- ▶ gibts auch für Image-Object-Tree `-H:+PrintImageObjectTree`
- ▶ erzeugt recht große Dateien:



```
bernd@office:~/texte/vortraege/javaland-2021/workspace/graalvm-javaland-2021
[bernd@office graalvm-javaland-2021]$ ls -lh reports/
insgesamt 47M
-rw-rw-r--. 1 bernd bernd 7,7M 10. Mär 11:04 call_tree_hello-world_20210310_110412.txt
-rw-rw-r--. 1 bernd bernd 39M 10. Mär 11:04 object_tree_hello-world_20210310_110412.txt
-rw-rw-r--. 1 bernd bernd 48K 10. Mär 11:04 used_classes_hello-world_20210310_110412.txt
-rw-rw-r--. 1 bernd bernd 541K 10. Mär 11:04 used_methods_hello-world_20210310_110412.txt
-rw-rw-r--. 1 bernd bernd 2,2K 10. Mär 11:04 used_packages_hello-world_20210310_110412.txt
[bernd@office graalvm-javaland-2021]$
```

Aber da war doch noch was ...

Wie kann eine so dynamische Sprache wie Java
vor Programmstart compiliert werden ?

Kann ja nicht funktionieren ...

- ▶ Closed World Assumption trifft für meisten Java-Programme nicht zu
- ▶ Damit muss alles, was dynamisch geladen wird, von uns ! explizit angegeben werden
- ▶ Oder wir verwenden Hilfswerkzeuge

Einfache Reflection (geht!)

```
private static void createInteger() {  
    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 Reflection erzeugt: " + instance);  
}
```

Einfache Reflection 2 (geht mittlerweile)

```
private static void createInteger2() throws Exception {  
    String javaLangInteger = "java.lang.Integer";  
    Class<?> clazz = Class.forName(javaLangInteger);  
    Constructor<?> constructor =  
        clazz.getConstructor(new Class[] { String.class });  
    Object instance = constructor.newInstance(new Object[] { "42" });  
    System.out.println("Mit Reflection erzeugt: " + instance);  
}
```

Etwas kompliziertere Reflection (geht nicht!)

```
...
createInteger("java.lang.Integer");
...

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

Nicht ganz wahr

- ▶ geht, aber ...

```
Warning: Reflection method java.lang.Class.forName invoked at
    de.pdbm.graalvm.ReflectiveInteger.createInteger(ReflectiveInteger.
Warning: Reflection method java.lang.Class.getConstructor invoked
    at de.pdbm.graalvm.ReflectiveInteger.createInteger(ReflectiveInteg
Warning: Aborting stand-alone image build due to reflection
    use without configuration.
Warning: Image 'reflective-integer' is a fallback image that
    requires a JDK for execution (use --no-fallback to suppress
    fallback image generation
```

- ▶ Ohne Fallback: kompiliert, zur Laufzeit CNFE

Also nachhelfen ...

► `native-image -H:ReflectionConfigurationFiles=<file>`

mit Datei

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

Ein Werkzeug: Der Tracing Agent

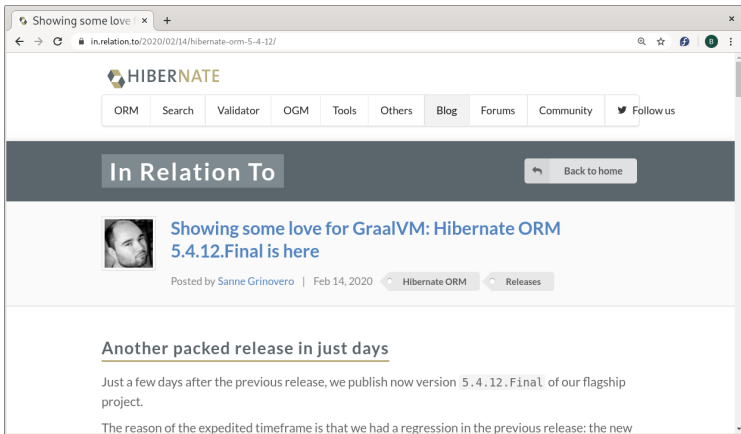
- ▶ Tracing Agent beobachtet laufendes Programm in normaler VM und erkennt reflektive Zugriffe
- ▶ Erzeugt daraus JSON-Konfiguration, die dann für native-image verwendet werden kann
- ▶ Aufruf:
`java -agentlib:native-image-agent=config-output-dir=...`
- ▶ Letztendlich: wenn *alle* Ausführungspfade bei Testläufen durchlaufen, dann alles verwendete auch kompiliert

Wer verwendet javac ?

Z.B. verwendet Quarkus ...

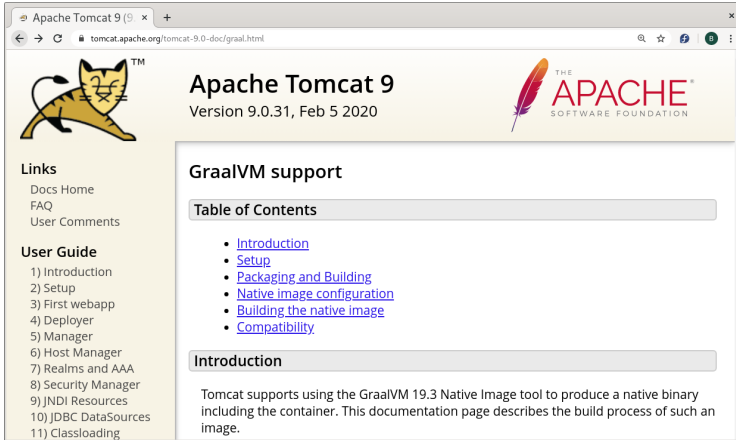
- ▶ CDI
- ▶ Hibernate
- ▶ ...
- ▶ Und es geht trotzdem
- ▶ Zusätzlicher Build-Schritt: *Build-time Augmentation*

Jüngste Entwicklungen: Hibernate



Quelle

Jüngste Entwicklungen: Tomcat



The screenshot shows a web browser window with the URL `tomcat.apache.org/tomcat-9.0-doc/graal.html`. The page header features the Tomcat logo (a yellow cat), the text "Apache Tomcat 9 Version 9.0.31, Feb 5 2020", and the Apache Software Foundation logo. On the left, there are navigation links: "Links" (Docs Home, FAQ, User Comments), "User Guide" (1) Introduction, 2) Setup, 3) First webapp, 4) Deployer, 5) Manager, 6) Host Manager, 7) Realms and AAA, 8) Security Manager, 9) JNDI Resources, 10) JDBC DataSources, 11) Classloading), and "GraulVM support". The main content area is titled "GraulVM support" and contains a "Table of Contents" with links to "Introduction", "Setup", "Packaging and Building", "Native image configuration", "Building the native image", and "Compatibility". Below this is an "Introduction" section that states: "Tomcat supports using the GraalVM 19.3 Native Image tool to produce a native binary including the container. This documentation page describes the build process of such an image."

Quelle

Jüngste Entwicklungen: Spring Native

The screenshot shows a web browser window with the URL `spring.io/blog/2021/03/11/announcing-spring-native-beta`. The page features the Spring logo and a navigation menu with links like 'Why Spring', 'Learn', 'Projects', 'Training', 'Support', and 'Community'. Below the navigation is a 'Spring Blog' section with filters for 'All Posts', 'Engineering', 'Releases', and 'News and Events'. The main article is titled 'Announcing Spring Native Beta!' and is categorized under 'ENGINEERING'. It is authored by Sébastien Deleuze and dated March 11, 2021, with 17 comments. The article text states: 'Today, after one year and half of work, I am pleased to announce that we're launching the beta release of [Spring Native](#) and its availability on [start.spring.io](#)! In practice, that means that in addition to the regular Java Virtual Machine supported by Spring since its inception, we are adding beta support for compiling Spring applications to [native images](#) with [GraalVM](#) in order to provide a new way to deploy Spring applications. Java and Kotlin are supported. Those native Spring applications can be deployed as a standalone executable (no JVM installation required) and offer interesting characteristics including almost instant startup (typically < 100ms), instant peak performance and lower memory consumption at the cost of longer build times and fewer runtime optimizations than the JVM.'

On the right side of the article, there is a 'Get the Spring newsletter' section. It includes a text input field for 'Email Address', a checkbox labeled 'Yes, I would like to be contacted by The Spring Team and VMware for newsletters, promotions and events', and a 'SUBSCRIBE' button.

Quelle

Substrate VM Java Limitations

Dynamic Class Loading / Unloading	Not supported
Reflection	Supported (req. Conf.)
Dynamic Proxy	Supported (req. Conf.)
Java Native Interface (JNI)	Mostly supported
Unsafe Memory Access	Mostly supported
Static Initializers	Partially supported
InvokeDynamic Bytecode and Method Handles	Not supported
Lambda Expressions	Supported
Synchronized, wait, and notify	Supported
Finalizers	Not supported
References	Mostly supported
Threads	Supported
Identity Hash Code	Supported
Security Manager	Not supported
JVMTI, JMX, other native VM interfaces	Not supported
JCA Security Services	Supported

Quelle: Substrate VM Java Limitations, deprecated

Es gibt noch viel mehr ...

- ▶ Profile-guided Optimizations (PGO, nur in EE-Version): JIT schon AOT
- ▶ G1 basierter Garbage Collector
- ▶ Klasseninitialisierung zur Compile- oder Laufzeit
- ▶ Protokolle hinzunehmen
- ▶ Service-Loader wird unterstützt
- ▶ VisualVM
- ▶ Maven
- ▶ ...sehr aktive Entwicklung

Fazit

- ▶ Hello World geht ;-)
- ▶ Größere Systeme ohne Werkzeugunterstützung nicht praktikabel
- ▶ Unterm Strich: wenn Werkzeug gut ist, merkt man den Mehraufwand gar nicht (außer in der Compile-Zeit)

Fragen und Anmerkungen



Vortrag und Code

<https://github.com/BerndMuller/graalvm-javaland-2021>

