# GraalVM Native Image的理论与实践

分享者：王良 日期：2023-02-10

注：第一、二部分的大部分资料来自以下博文：

博文标题：《Java在云原生的破局利器——AoT(JIT与AoT)》

博文链接：[https://huaweicloud.csn.net/63311d00d3efff3090b52913.html](https://huaweicloud.csdn.net/63311d00d3efff3090b52913.html)

# 理论知识

# 什么是JIT、AoT？

开发语言在运行之前通常都需要编译，而**JIT（Just-in-Time，实时编译）与AoT（Ahead-of-Time，预编译）**则是最常见的两种编译模式。。

JIT 在运行时即时编译，在开发周期中使用，可以动态下发和执行代码，开发测试效率高，但运行速度和执行性能则会因为运行时即时编译受到影响。

AoT 即提前编译，可以生成被直接执行的二进制代码，运行速度快、执行性能表现好，但每次执行前都需要提前编译，开发测试效率低。

# AoT的优点与缺点（相对于JIT）

## AoT的优点

* 在程序运行前编译，可以避免在运行时的编译性能消耗和内存消耗；
* 可以在程序运行初期就达到最高性能，程序启动速度快；
* 运行产物只有机器码，打包体积小。

## AoT的缺点

* 由于是静态提前编译，不能根据硬件情况或程序运行情况择优选择机器指令序列，**部分场景**下的**理论峰值性能还不如JIT**；
* 没有动态能力；
* 同一份产物**不能跨平台运行。**

# Java-JIT的现状

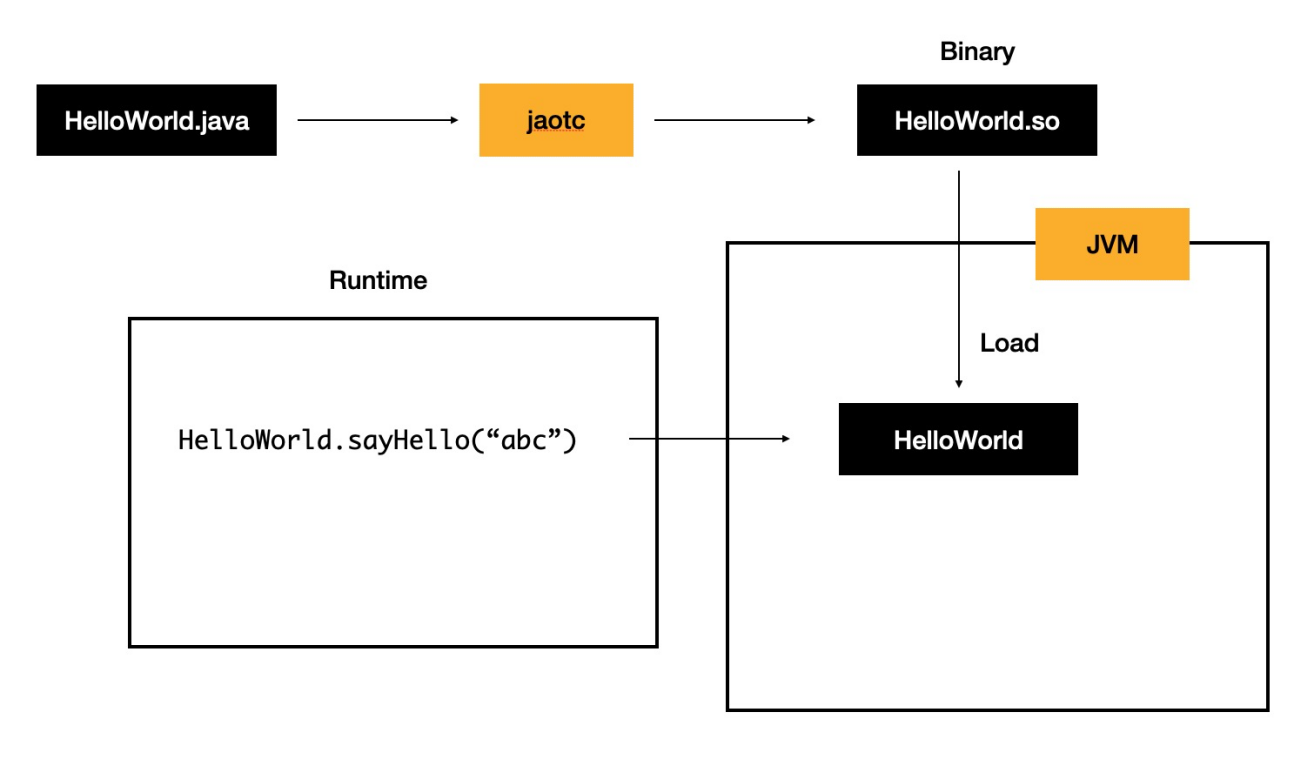
JIT一直是Java语言的灵魂特性之一。AoT似乎长久以来和Java语言都没有什么太大的关系。但是近年来随着Serverless、云原生等概念和技术的火爆，Java JVM和JIT的性能问题越来越多地被诟病，在Golang、Rust、NodeJS等新一代语言的包夹下，业界也不断出现 “云原生时代，Java已死” 的言论。

# Java-AoT的历史演进

## 在JVM中运行AoT的思路（jaotc、阿里-AppCDS）

### jaotc

2016年，OpenJDK的 [JEP 295](https://openjdk.org/jeps/295?login=from_csdn) 提案首次在Java中引入了AoT支持，在这一草案中，JDK团队提供了 jaotc 工具，用于将指定class文件中的方法逐个编译到native代码片段，再通过JVM加载某个类后替换方法的入口到AoT代码来实现启动加速的效果。



（jaotc原理图）

但jaotc存在很多问题：

* 首先，设计上没有考虑到Java的多Classloader场景；
* 此外，由于社区人手不足，缺乏调优和维护，jaotc的实际运行效果不尽人意，有时甚至会对应用的启动和运行速度带来反向优化，实装没多久之后就退化为实验特性，最终在JDK 16中被删除，结束了短暂的一生。

### 阿里-AppCDS

后来阿里AJDK团队自研的AppCDS（Class-Data-Share）技术继承了jatoc的思路，进行了大幅的优化和完善，目前也不失为一种Java AoT的选择，其本质思路和jaotc基本一致 ，这里就不再赘述了。

## 摒弃JVM的思路（GraalVM Native Image）

目前业界除了上面的方案，还有另外一种实现Java AoT的思路，那就是直接摒弃JVM，和C/C++ 一样通过编译器直接将代码编译成机器代码，然后运行。这无疑是一种直接颠覆Java语言设计的思路，不过还是被各路大佬们实现了，那就是GraalVM Native Image。

### GraalVM Native Image

它通过C语言实现了一个超微缩的运行时组件 —— Substrate VM，基本实现了JVM的各种特性，但足够轻量、可以被轻松内嵌，这就让Java语言和工程摆脱JVM的限制，能够真正意义上实现和C/C++一样的AoT编译。这一方案在经过长时间的优化和积累后，已经拥有非常不错的效果，基本上成为Oracle官方首推的Java AoT解决方案，接下来我们会重点分析一下这项技术的原理和实际应用。

# Java-AoT的破局利器：GraalVM

## 什么是GraalVM？

GraalVM是Oracle在2019年推出的新一代UVM（通用虚拟机），它在HotSpotVM的基础上进行了大量的优化和改进，主要提供了两大特性：

* **Polyglot：多语言支持**，你可以在GraalVM中无缝运行多种语言，包括Java、JS、Ruby、Python甚至是Rust。更重要的是可以通过GraalVM的API来实现语言混编 —— 比如在一段Java代码中无缝引用并调用一个Python实现的模块。
* **HighPerformance：高性能**，首先它提供了一个高性能的JIT引擎，让Java语言在GraalVM上执行的时候效率更高速度更快 ；其次就是提供了SubstrateVM，通过Graal Compiler你可以将各种支持的语言（包括Java）编译成本地机器代码，获得更好的性能表现。

值得一提的是，SubstrateVM虽然名为VM，但并不是一个虚拟机，而是一个包含了 垃圾回收、线程管理 等功能的运行时组件（Runtime Library），就好比C++当中的stdlib一样。当Java程序被编译为Native Image运行的时候，并不是运行在SubstrateVM里，而是将SubstrateVM当作库来使用其提供的各种基础能力，以保障程序的正常运行。

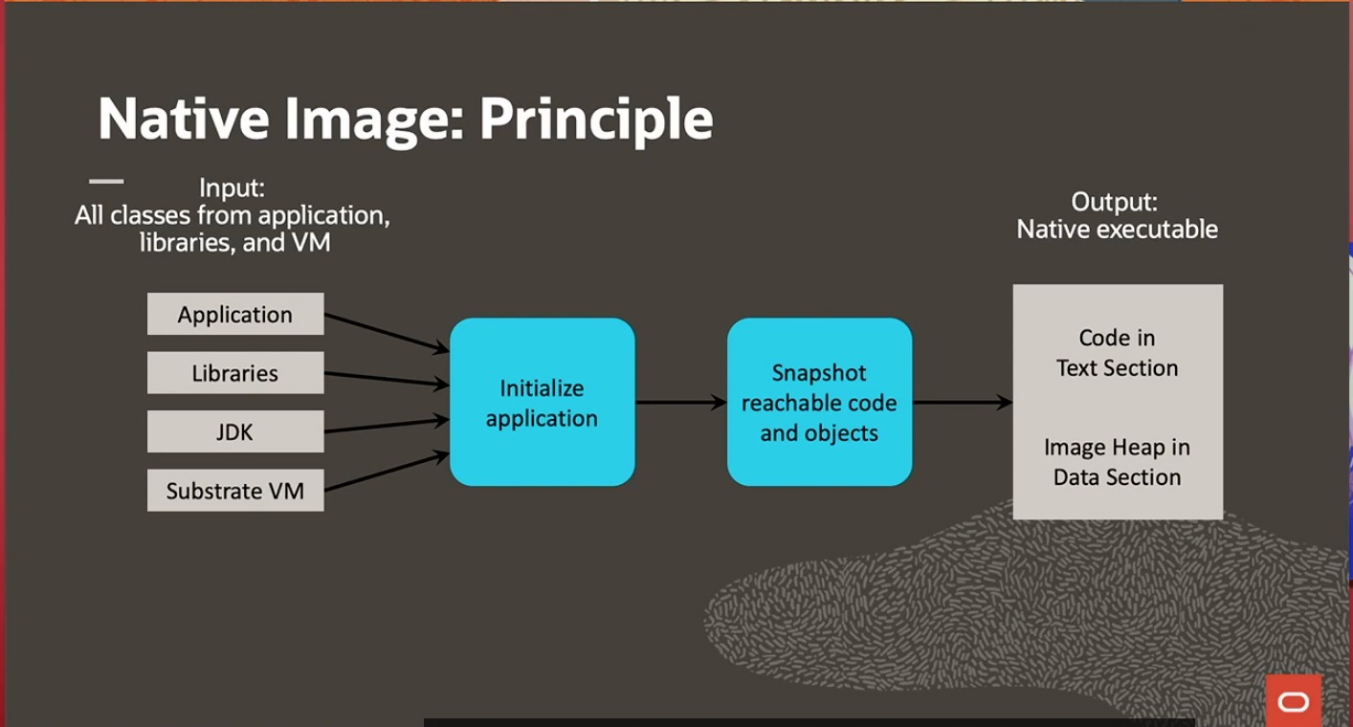
不难看出，GraalVM这个项目的野心是非常大的，可以说这个项目是Oracle抢占云原生市场的一个重要布局，随着官方的不断投入和社区的壮大，目前GraalVM已经日渐成熟，在高性能和跨语言支持方面都交出了令人满意的答卷。GraalVM本身是一个非常庞大的项目，有很多的细节点可以深挖，不过接下来我们还是重点研究一下它的AoT能力 —— Native Image。

## Native Image的原理与限制

官网资料：<https://www.graalvm.org/latest/reference-manual/native-image/>

### Native Image的原理

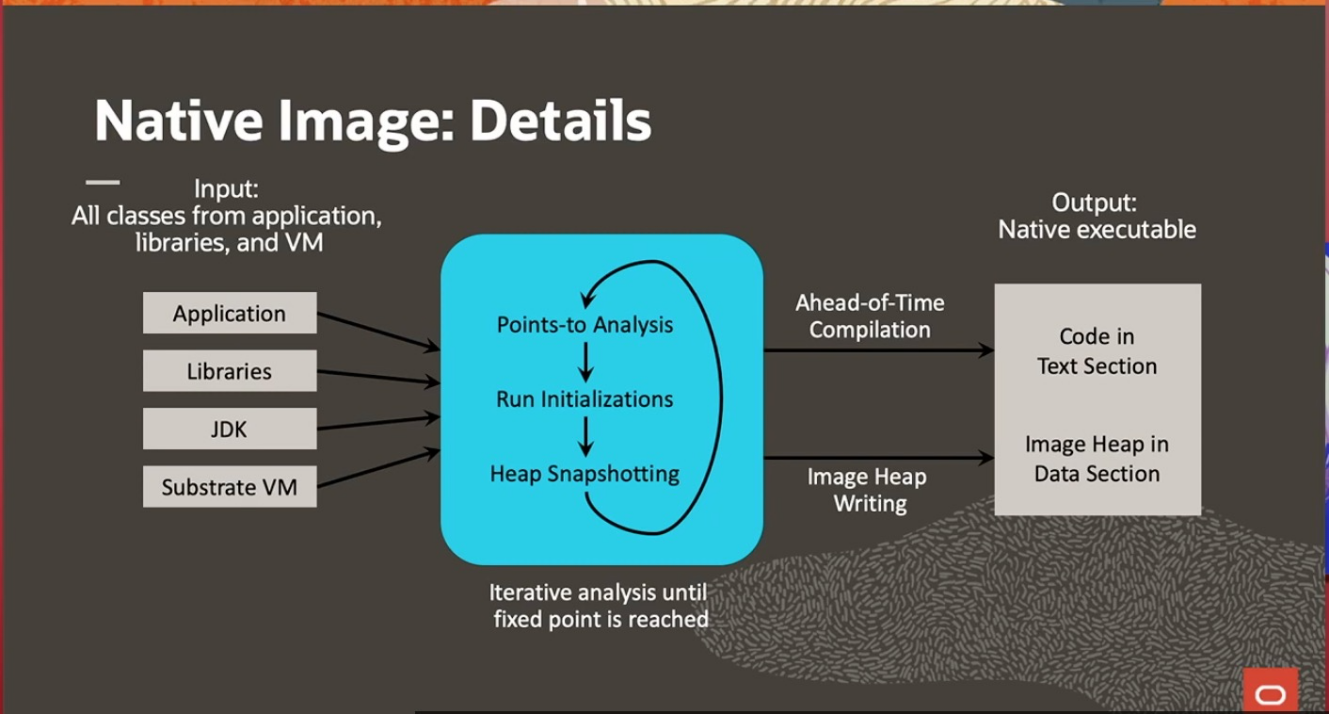
一个Java程序究竟是如何被编译成静态可执行文件的？我们先来看一下Native Image的原理。



（Native Image原理图）

如上图所示，Native Image的输入是整个应用的所有组件，包括：应用本身的代码、各种依赖的库、JDK库、以及SVM。首先会进行整个应用的初始化，也就是代码的静态分析，这个分析过程有点类似GC中的“可达性分析”，会将程序运行过程中将所有可达的代码、变量、对象生成一个快照，最终打包成一个可执行的Native Image，而“不可到达”的代码将被丢弃。

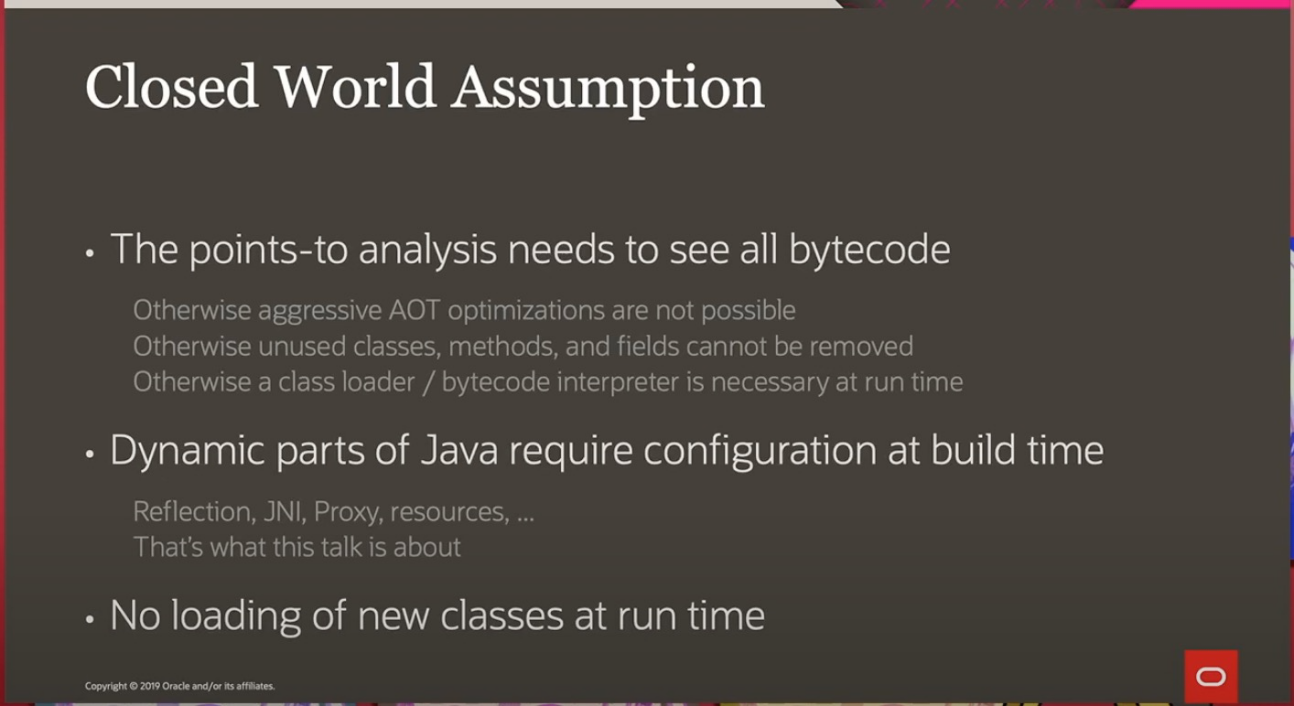
一个完整的Native Image包含两个部分，一部分称为 Text Section，即用户代码编译成的机器代码；另一部分称为 Data Section，存储了应用启动后堆区内存中各种对象的快照。



这个静态分析的过程（官方称之为 Pionts-to Analysis）是非常复杂且耗时的，整个分析过程会以递归的方式进行，最终得到两个树形结构Call Tree（包含所有可达的方法）以及Object Tree（包含所有可达的对象），Call Tree中所有可达的方法会被AoT编译为机器码，成为Native Image的Text Section，而Object Tree中所有可达的对象及变量则会被保存下来，写入Native Image的Data Setion。

### Native Image的限制（Closed World）

你一定会好奇JVM的动态特性，例如反射、代理，要如何进行静态分析呢？很显然，这两者之间是存在冲突的，因此Native Image设置了一个名为 “Closed World Assumption”（封闭世界假设），作为静态分析的基本前提。



（Closed World Assumption内容图）

如上图所示，这个基本前提包含三个要求，对应的也就是目前Native Image存在的三个限制：

* **Points-to分析的时候，需要接受完整的字节码作为输入（即项目中所有用到的class的字节码都需要获取的到）。**

=> 在运行期动态生成或者是动态获取字节码的程序，无法构建成 Native Image。

* **Java的动态特性，目前包括5种：反射、JNI、JDK动态代理、resources、JDK序列化，都需要通过配置文件在构建前事先声明好。**

=> 无法提前声明动态特性使用范围的程序，无法构建成Native Image （例如，根据用户输入的一个参数反射去调用某个方法）。

* **在整个运行过程中，程序不会再加载任何新的class。**

=> 在运行期间执行动态编译，或者是自定义Classloader动态装载类的程序，无法构建成Native Image。

# 二、GraalVM Native Image的实践

# 环境安装

GraalVM GitHub发布地址：

<https://github.com/graalvm/graalvm-ce-builds/releases/>

GraalVM官网下载地址：

<https://www.graalvm.org/downloads/>

下载GraalVM JDK后，你可以直接把它当作JAVA JDK来配置环境变量。

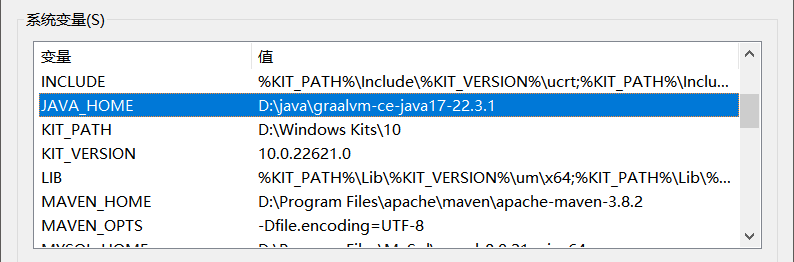
这里用graalvm-ce-java17-windows-amd64-22.3.1.zip来说明：

## 下载graalvm后解压

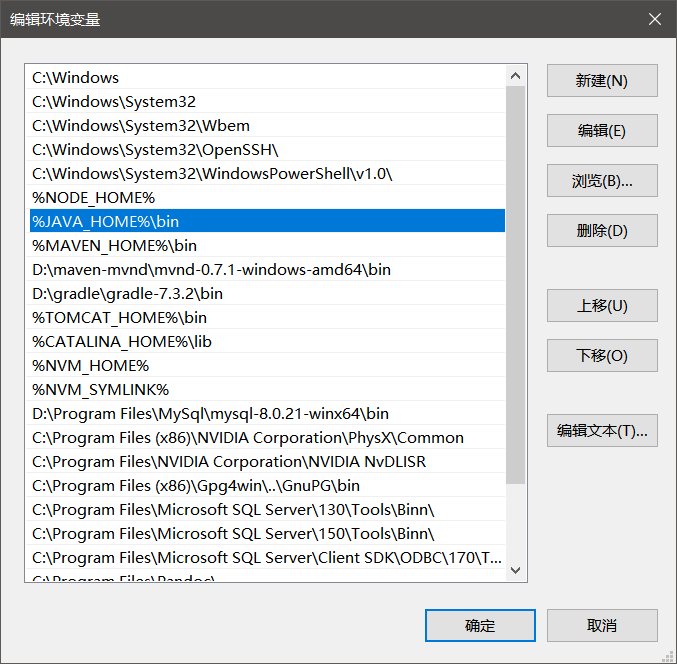


## 配置环境变量

添加环境变量 JAVA\_HOME



在环境变量 Path 里添加 %JAVA\_HOME%\bin;



确认下环境配置是否成功：

|  |
| --- |
| C:\Users\new>java -version  openjdk version "17.0.6" 2023-01-17  OpenJDK Runtime Environment GraalVM CE 22.3.1 (build 17.0.6+10-jvmci-22.3-b13)  OpenJDK 64-Bit Server VM GraalVM CE 22.3.1 (build 17.0.6+10-jvmci-22.3-b13, mixed mode, sharing) |

## 安装native-image

|  |
| --- |
| > gu install native-image |

`gu list` 命令确认下安装是否成功：

|  |
| --- |
| C:\Users\new> gu list  ComponentId Version Component name Stability Origin  -----------------------------------------------------------------------------------  graalvm 22.3.1 GraalVM Core Supported  native-image 22.3.1 Native Image Early adopter github.com |

## windows环境下需安装VS并配置C库的环境变量

该步骤可以参考我的博文：

《Windows环境下打包Native Image》

[https://easyj.icu/blog/#/native-image/native-image-windows?id=\_13%e3%80%81%e5%ae%89%e8%a3%85visual-studio](https://easyj.icu/blog/" \l "/native-image/native-image-windows?id=_13%e3%80%81%e5%ae%89%e8%a3%85visual-studio)

## linux环境下需安装gcc

|  |
| --- |
| $ yum install gcc |

# 简单实践

介绍了Native Image的基本原理和限制后，让我们来实际实践看看这项技术到底能够带给我们什么。

## 编写代码

这里我们先给出一个非常基础的DEMO代码：HelloWorld.java

|  |
| --- |
| public class HelloWorld {   private static final String *CONST* = "this-is-a constant var";   private String name;   public HelloWorld(String name) {  this.name = name;  }   public void sayHello() {  System.*out*.println("hello, " + name);  }   public static void main(String[] args) throws Exception {  long t0 = System.*nanoTime*();   System.*out*.println(*CONST*);  HelloWorld h1 = new HelloWorld("lumin");  HelloWorld h2 = new HelloWorld(args[0]);  h1.sayHello();  h2.sayHello();   System.*out*.println("time: " + (System.*nanoTime*() - t0));   Thread.*sleep*(3000);  } } |

## 编译代码

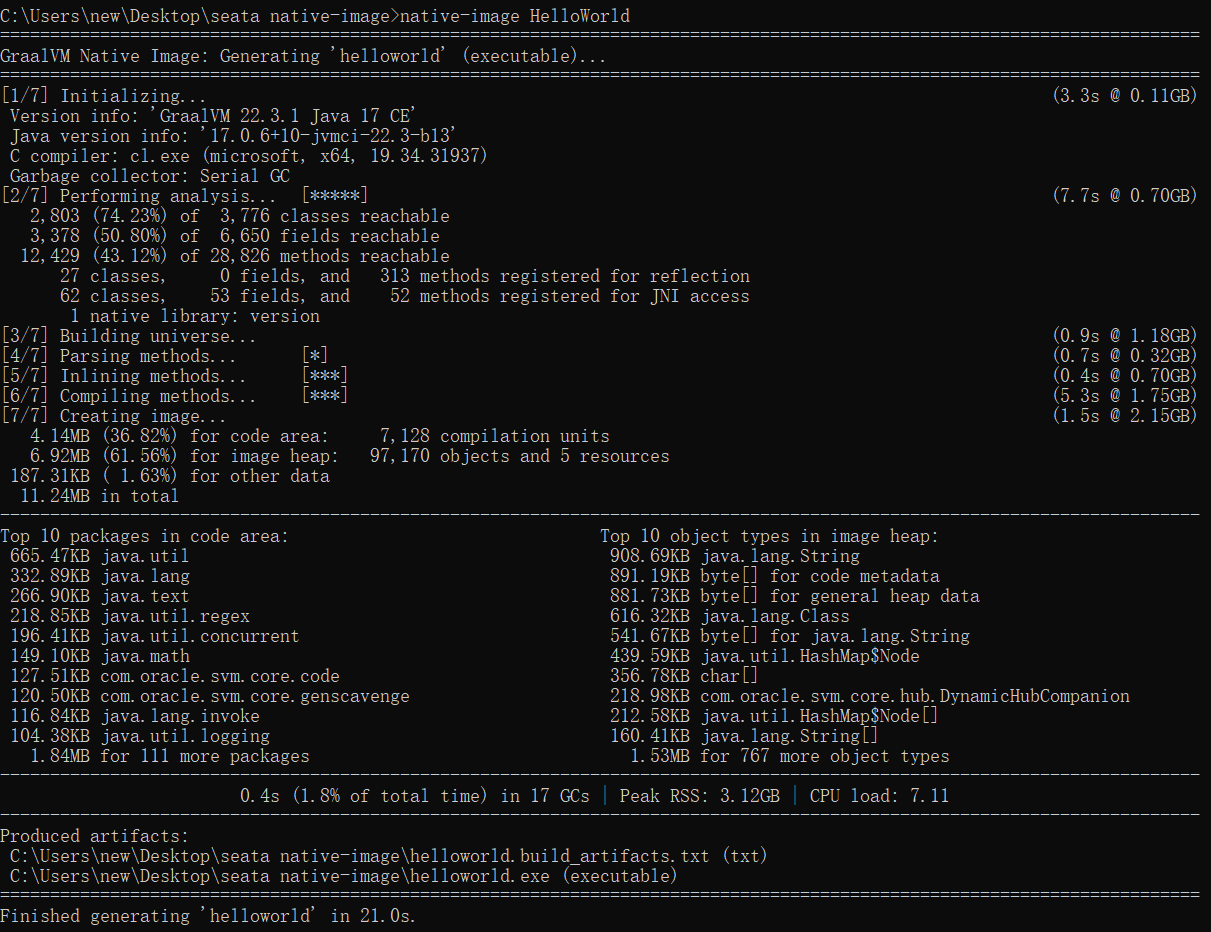
如何将这段代码构建成Native Image呢？下载并配置好graalvm后，首先使用javac将代码编译成字节码文件：HelloWorld.class

|  |
| --- |
| > javac HelloWorld.java |

## 打包native-image

接下来执行Native Image Build，开始打包native-image：

|  |
| --- |
| > native-image HelloWorld |



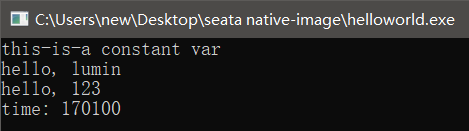
整个构建过程会执行比较长的一段时间，主要是执行Points-Analysis过程较长（大约三分多钟），最终的产物就是一个二进制文件：helloworld.exe

## 打包native-image的参数

我们可以加上 -H:+PrintImageObjectTree -H:+ExhaustiveHeapScan -H:+PrintAnalysisCallTree的参数再进行一次build，这样可以将整个Points-to Analysis的详细过程（Object Tree和Call Tree）打印到文件以供分析。

## 运行native-image

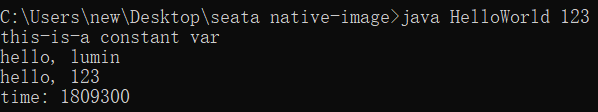
双击或执行 `start helloworld.exe 123` 运行native-image文件helloworld.exe，效果如下：



耗时：170,100纳秒。

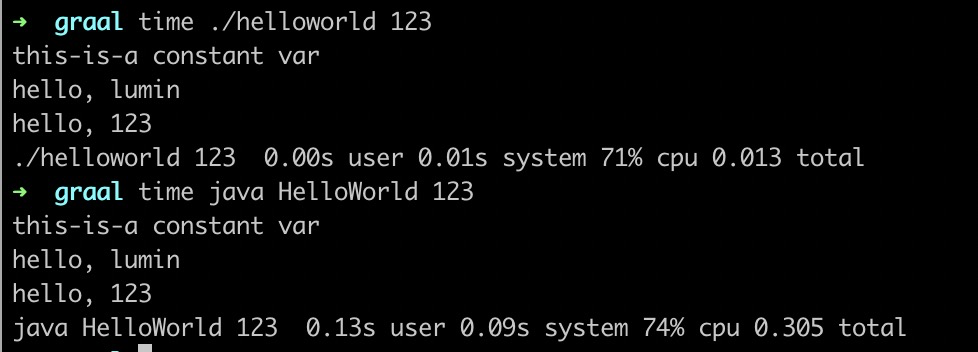
## 与JVM运行的比较

执行 `java HelloWorld 123` 运行HelloWorld.class，效果如下：



耗时：1,809,300纳秒，启动速度耗时是native-image的10倍以上。

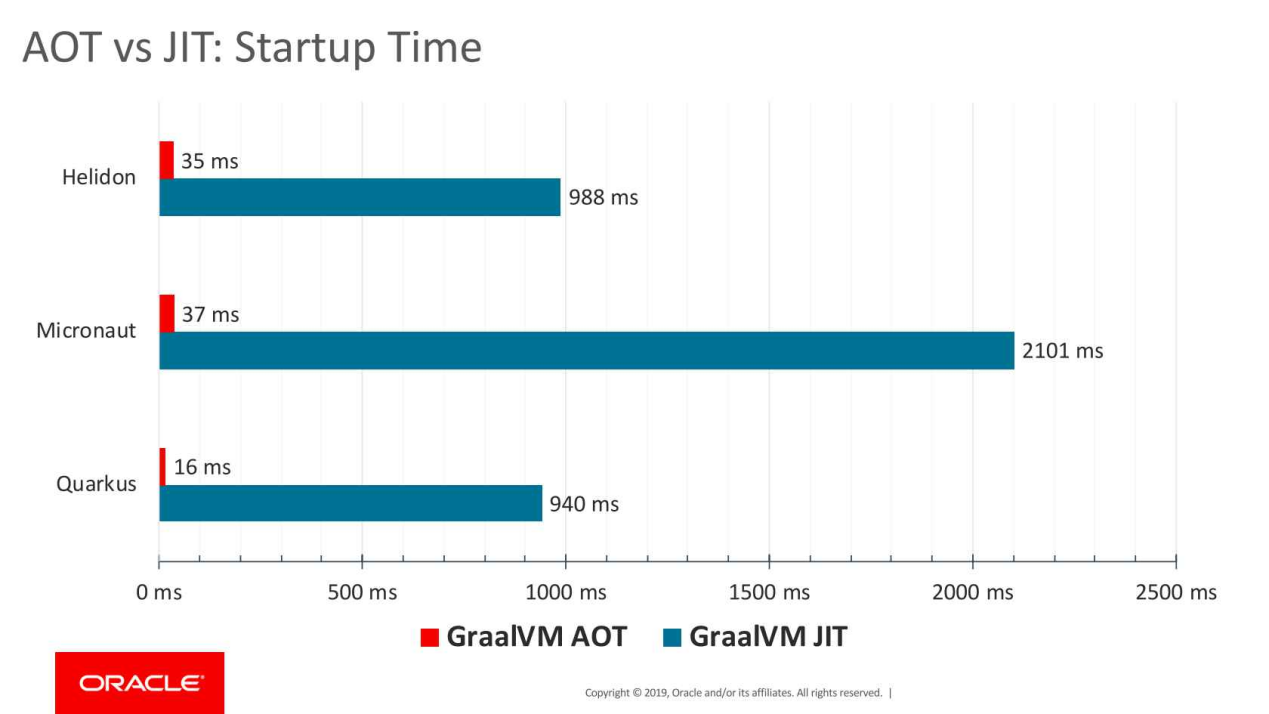
再看博文中在linux下的运行效果：



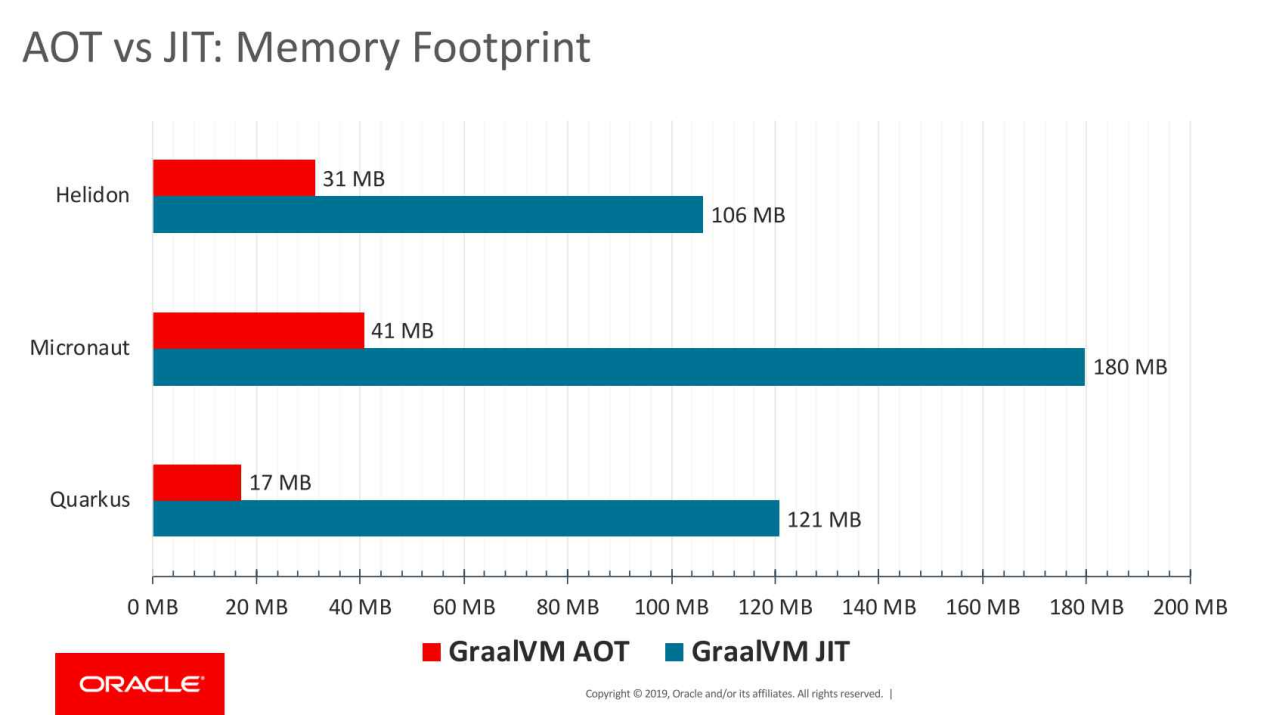
可以看到，相比于使用JVM运行，Native Image的速度要快上不少，cpu占用也更低一些。

## 其他实验数据

从官方提供的各类实验数据也可以看出Native Image对于**启动速度**和**内存占用**带来的提升是非常显著的：



（三组**启动时间**对比图）



（三组**内存占用**对比图）

# 可达性元数据

## 对可达性的疑惑

在讲解下面的内容之前，我们先看几个问题：

问题1）Native Image到底能够分析出哪些可达性元素？哪些又是分析不出来的？

问题2）Java中，所有使用动态语言特性执行的代码，都是不可达的吗？

带着这两个问题，我们再继续看以下内容。

## 可达性元数据（Reachability Metadata）

在GraalVM Native Image中，除了经过分析代码获取可达性元素以外，其他分析不出来的需到达元素则必须通过配置告知Native Image，官方称这些配置内容为可达性元数据（Reachability Metadata）（下面简称为元数据）。

**官网资料：**<https://www.graalvm.org/22.3/reference-manual/native-image/metadata/>

每个需要元数据的Java动态特性都有一个名为`<feature>-config.json`的JSON文件。这些JSON文件由一些条目组成，以此来告诉Native Image要包含的元素。单个条目的基本结构如下：

|  |
| --- |
| {  "condition": {  "typeReachable": "<fully-qualified-class-name>"  },  <metadata-entry> } |

`condition`是该条目的生效条件，以此来避免不需要的条目也生效，导致生成的二进制文件太大。

官网资料显示，目前`condition`暂时只支持 `typeReachable` 一种，允许配置一个类的完全限定名。

目前有以下两种方式可以提供元数据文件给Native Image：

* 通过命令引用，格式如：-H:XxxxxxConfigurationFiles=./<feature>-config.json具体的命令参数名见[下一章节](#_Java动态语言特性)。
* 存放在jar包中的 `/META-INF/native-image/<groupId>/<artifactId>/<feature>-config.json`

# Java动态语言特性及其配置

## 反射（Reflection）

在编译的目录下新建一个 `reflect-config.json`，格式内容如下：

|  |
| --- |
| [  {  "condition": {  "typeReachable": "<condition-class>"  },  "name": "<class>",  "methods": [  {"name": "<methodName>", "parameterTypes": ["<param-one-type>"]}  ],  "queriedMethods": [  {"name": "<methodName>", "parameterTypes": ["<param-one-type>"]}  ],  "fields": [  {"name": "<fieldName>"}  ],  "allDeclaredMethods": true,  "allDeclaredFields": true,  "allDeclaredConstructors": true,  "allPublicMethods": true,  "allPublicFields": true,  "allPublicConstructors": true,  "queryAllDeclaredMethods": true,  "queryAllDeclaredConstructors": true,  "queryAllPublicMethods": true,  "queryAllPublicConstructors": true,  "unsafeAllocated": true  } ] |

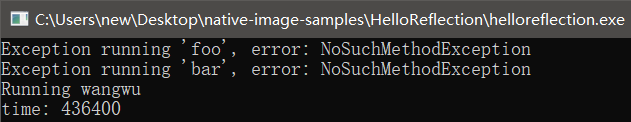
以下是一个简单的反射代码：

|  |
| --- |
| public class HelloReflection {   public static void foo() {  System.*out*.println("Running foo");  }   public static void bar() {  System.*out*.println("Running bar");  }   public static void wangwu() {  System.*out*.println("Running wangwu");  }   public static void main(String[] args) throws Exception {  for (String arg : args) {  try {  HelloReflection.class.getMethod(arg).invoke(null);  } catch (ReflectiveOperationException ex) {  System.*out*.println("Exception running '" + arg + "', error: "  + ex.getClass().getSimpleName());  }  }  HelloReflection.class.getMethod("wangwu").invoke(null);   Thread.*sleep*(3000);  } } |

这里我们需要加上 --no-fallback参数来构建，否则Native Image检测到这个程序使用了未配置的反射时，会把产物自动降级成JVM运行。打包命令如下：

|  |
| --- |
| > javac HelloReflection.java  > native-image HelloReflection --no-fallback |

然后运行命令 `start helloreflection.exe for bar`



可以看到，运行foo和bar方法都提示 NoSuchMethodException，而wangwu方法却成功运行了。

这是因为在编译时我们无法知道用户输入的会是哪个方法，因此静态编译的时候就不会把foo、bar这两个方法认为是“可达的”，而wangwu方法是直接通过编码写入的，Native Image可通过分析知道该方法是“可达的”，这证明了[3.1章节中提到的问题2](#_对可达性的疑惑)的回答是否定的。

然后，我们来看下foo、bar方法要配置怎样的元数据：

这里，我们可以配置如下：

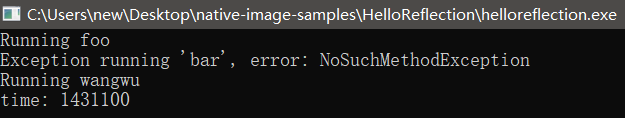
|  |
| --- |
| [  {  "name": "HelloReflection",  "methods": [  { "name": "foo", "parameterTypes": [] }  ]  } ] |

先只声明foo方法，来看一下效果。

添加参数引用配置文件：

-H:ReflectionConfigurationFiles=./reflect-config.json

重新打包运行一次：



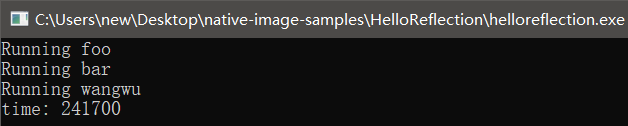
发现foo方法已经可以通过反射调用到，而bar方法依然提示NoSuchMethodException，

说明上面的配置生效了。

再将bar方法也配置进去：

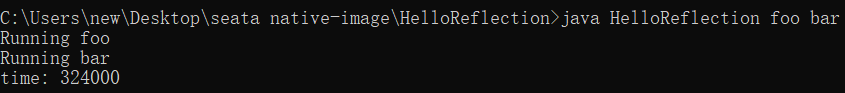
|  |
| --- |
| [  {  "name": "HelloReflection",  "methods": [  { "name": "foo", "parameterTypes": [] },  { "name": "bar", "parameterTypes": [] }  ]  } ] |

再重新打包运行一次：



这次foo和bar两个方法都通过反射调用到了。

然后我们再看JVM下运行的耗时：



虽然没有上面简单例子差距那么大，但JVM的耗时依然是native-image的1.5倍。

## JNI（Java Native Interface）

在编译的目录下新建一个 `jni-config.json`，格式与反射的配置文件格式是一致的。*请查看上一章节，这里就不重复展示了。*

添加参数引用配置文件：

-H:JNIConfigurationFiles=./jni-config.json

## 资源文件（Resources）

在编译的目录下新建一个 `resource-config.json`，格式内容如下：

|  |
| --- |
| {  "resources": {  "includes": [  {  "condition": {  "typeReachable": "<condition-class>"  },  "pattern": ".\*\\.txt"  }  ],  "excludes": [  {  "condition": {  "typeReachable": "<condition-class>"  },  "pattern": ".\*\\.txt"  }  ]  },  "bundles": [  {  "condition": {  "typeReachable": "<condition-class>"  },  "name": "fully.qualified.bundle.name",  "locales": ["en", "de", "sk"]  }  ] } |

例子：

|  |
| --- |
| {  "resources": {  "includes": [  {  "pattern": "\\Qapplication\\E.\*\\Q.yml\\E"  },  {  "pattern": "\\QMETA-INF\/spring.factories\\E"  },  {  "pattern": "\\QMETA-INF\/services\/io.seata.\\E.\*"  },  {  "pattern": "\\QMETA-INF\/services\\E"  },  {  "pattern": "\\QMETA-INF\\E"  }  ]  } } |

添加参数引用配置文件：

-H:ResourceConfigurationFiles=./resource-config.json

## JDK动态代理（DynamicProxy）

目前，在native-image中，只支持JDK代理，不支持其他代理。

在编译的目录下新建一个 `proxy-config.json`，格式内容如下：

*注：这里只能配置接口，因为JDK代理只能代理接口。*

|  |
| --- |
| [  {  "condition": {  "typeReachable": "<condition-class>"  },  "interfaces": [  "<InterfaceA>",  "<InterfaceB>"  ]  } ] |

添加参数引用配置文件：

-H:DynamicProxyConfigurationFiles=./proxy-config.json

## JDK自带序列化（Serialization）

在编译的目录下新建一个 `serialization-config.json`，格式内容如下：

|  |
| --- |
| {  "types": [  {  "condition": {  "typeReachable": "<condition-class>"  },  "name": "<fully-qualified-class-name>",  "customTargetConstructorClass": "<custom-target-constructor-class>"  }  ],  "lambdaCapturingTypes": [  {  "condition": {  "typeReachable": "<condition-class>"  },  "name": "<fully-qualified-class-name>",  "customTargetConstructorClass": "<custom-target-constructor-class>"  }  ] } |

注意，这里注册的类，必须是实现了接口 `java.io.Serializable` 接口的类，因为只有这些类才能序列化和反序列化。

添加参数引用配置文件：

-H:SerializationConfigurationFiles=./serialization-config.json

## 其他

### Predefined Classes（预定义类）

<https://www.graalvm.org/22.3/reference-manual/native-image/metadata/#predefined-classes>

据官网介绍，可以预先将已生成为字节码的类配置到 `predefined-classes-config.json` 文件中：

|  |
| --- |
| [  {  "type": "agent-extracted",  "classes": [  {  "hash": "<class-bytecodes-hash>",  "nameInfo": "<class-name>"  }  ]  } ] |

在运行时，如果试图加载与跟踪期间遇到的某个类具有相同名称和字节码的类，则将向应用程序提供预定义的类。

### Java Agent（暂不支持）

非常遗憾的是目前GraalVM Native Image暂不支持使用Java Agent。

持续关注issue：<https://github.com/oracle/graal/issues/1065>

这将导致一些产品无法在native-image中使用，如：SkyWalking

### Charset.forName(String charsetName)

要想正常使用该功能，需要在Native Image的打包参数里，添加上 `-H:+AddAllCharsets`。

# 实际应用的阻碍

在了解了上面的原理知识后，大家是不是觉得，这东西也太难在实际应用中使用了吧？的确，实际应用过程中，根据系统复杂度的不同，会存在很多问题：

* 引用的第三方组件不兼容native-image；

**从维护方角度来看：**如果添加了太多元数据配置，打出来的包会变大；

**从使用方角度来看：**不了解三方组件的实现原理，只能实际打包出来去测试，碰到报错，加个配置，重新打包再测试，直到没有报错为止，如果引用的第三方组件太多，会非常耗时，打包native-image非常耗时。

现阶段，这的确是一个头痛的问题。但随着时间的推移，会有越来越多的三方组件去兼容native-image，到时候，使用方需要配置的元数据也就会越来越少了。

* 有些方法通过反射读取了几十上百个类，运行时会用到，但Native Image分析阶段该方法被判断为“不可达”，那是不是只能一个个全部配置起来？

答：其实只要将这个方法配置起来就可以了。Native Image会对该方法做进一步的分析。

* CGLIB动态代理不能用了，Spring项目怎么办？

这个问题，下一章节会讲到。

# 三、Spring6和SpringBoot3对Native Image的支持

了解了Native Image的原理后，知道了我们需要做的就是把用到动态特性的元素配置起来就可以了。Spring6就提供了 `RuntimeHintsRegistrar` 和 `BeanRegistrationAotProcessor` 两个接口，让我们能够快速的扩展，来注册需要的配置。

# RuntimeHintsRegistrar

这个接口只有一个方法，方法上有一个 `RuntimeHints` 类型的参数 `hints`，`RuntimeHints`类里针对上面提到的[7种中的前5种动态特性](#_动态特性的实践)分别提供了5个接口，来注册对应的配置。

|  |
| --- |
| package org.springframework.aot.hint;  import org.springframework.lang.Nullable;  @FunctionalInterface public interface RuntimeHintsRegistrar {  void registerHints(RuntimeHints hints, @Nullable ClassLoader classLoader);  } |

|  |
| --- |
| package org.springframework.aot.hint;  public class RuntimeHints {  public ReflectionHints reflection();  public ResourceHints resources();  public SerializationHints serialization();  public ProxyHints proxies();  public ReflectionHints jni();  } |

我们的应用可以实现 `RuntimeHintsRegistrar` 接口，spring会负责调用`RuntimeHintsRegistrar.registerHints(...)` 方法，我们只要调用hints的5个接口即可。

然后将开发好的实现类配置到 `/META-INF/spring/aot.factories` 中：

|  |
| --- |
| org.springframework.aot.hint.RuntimeHintsRegistrar=\  io.seata.spring.aot.SeataCoreRuntimeHints |

# BeanRegistrationAotProcessor

该接口允许我们在spring应用中获取注册进来的Bean，并对其进行解析，判断是否需要注册相关配置，如果需要，则实现接口 `BeanRegistrationAotContribution` 并返回其实例来告诉spring注册逻辑。

|  |
| --- |
| package org.springframework.beans.factory.aot;  import org.springframework.beans.factory.support.RegisteredBean; import org.springframework.lang.Nullable;  @FunctionalInterface public interface BeanRegistrationAotProcessor {   @Nullable  BeanRegistrationAotContribution processAheadOfTime(RegisteredBean registeredBean);   ...... } |

|  |
| --- |
| package org.springframework.beans.factory.aot;  import org.springframework.aot.generate.GenerationContext;  @FunctionalInterface public interface BeanRegistrationAotContribution {   ......   void applyTo(GenerationContext generationContext, BeanRegistrationCode beanRegistrationCode);  ......  } |

同样将开发好的实现类配置到 `/META-INF/spring/aot.factories` 中：

|  |
| --- |
| org.springframework.beans.factory.aot.BeanRegistrationAotProcessor=\ io.seata.spring.aot.SeataLocalTCCBeanRegistrationAotProcessor |

# Spring针对无法使用CGLIB代理的解决方式

Spring为了能够继续使用CGLIB代理，通过一个非常取巧的方式来实现的：生成目标类的子类（即：代理类）的class文件，并将其反射信息配置到reflect-config.json中，然后再通过反射实例化该代理类来代替目标类使用。

这样就很好的规避了CGLIB代理无法在native-image中使用的问题了。

# Native Image所需的maven插件

## org.graalvm.buildtools:native-maven-plugin

## spring-boot-maven-plugin

springboot3的maven插件中，增加了 `spring-aot` 插件，该插件会预运行spring-boot应用，分析可能会注册的bean，并通知到所有 [BeanRegistrationAotProcessor](#_BeanRegistrationAotProcessor) 的实现类，对bean进行解析并注册所需的信息。

另外，还会运行所有 [RuntimeHintsRegistrar](#_RuntimeHintsRegistrar) 的实现类，运行配置注册逻辑。

# 在spring-boot:2.7.x中的区别

由于在2.7.x的时候，spring-native项目还处于实验阶段，项目地址：

<https://github.com/spring-projects-experimental/spring-native/>

**该项目主要提供了一个依赖：**org.springframework.experimental:spring-native:0.12.2，它提供了上面提到的多个Hints接口。

**还提供了一个maven插件（当然gradle插件也有）:**

org.springframework.experimental:spring-aot-maven-plugin:0.12.2，该插件提供了一个名为 `generate` 的goal，其实就是上面提到的`spring-aot`。该项目并未发布到central上，而是发布在了Spring Releases仓库中。所以需要添加该仓库才能拉取到它们。

该实验性项目的代码，最终在Spring6和SpringBoot3项目中经过一翻改造和优化，最终发布到了central上。

目前，该实验性项目在发布了0.12.2后，已将近两周没维护了。上次在一个issue里看到他们说过即将停止更新，估计已经停止维护了。

# 复杂应用踩坑记录

请查阅 <https://easyj.icu/blog/#/native-image/treading-pit-log>

也欢迎正在使用Native Image的朋友，提交更多踩坑记录，谢谢。

