

课程内容：



- 1、GraalVM介绍与基本使用
- 2、Spring Boot 3.0新特性介绍与实战
- 3、Docker SpringBoot3.0 新特性实战
- 4、RuntimeHints介绍与实战
- 5、Spring AOT作用与核心原理源码分析

有道云链接：<https://note.youdao.com/s/3bUHzs15>

GraalVM体验

下载压缩包

打开<https://github.com/graalvm/graalvm-ce-builds/releases>，按JDK版本下载GraalVM对应的压缩包，请下载**Java 17**对应的版本，不然后面运行SpringBoot3可能会有问题。

Platform	Java 11	Java 17	Java 19	
Linux (amd64)	 download	 download	 download	<i>instructions</i>
Linux (aarch64)	 download	 download	 download	<i>instructions</i>
macOS (amd64) †	 download	 download	 download	<i>instructions</i>
macOS (aarch64) †	 download	 download	 download	<i>instructions</i>
Windows (amd64)	 download	 download	 download	<i>instructions</i>

下载完后，就解压，

配置环境变量

新开一个cmd测试：

安装Visual Studio Build Tools

因为需要C语言环境，所以需要安装Visual Studio Build Tools。

打开visualstudio.microsoft.com，下载Visual Studio Installer。

选择C++桌面开发，和Windows 11 SDK，然后进行下载和安装，安装后重启操作系统。

要使用GraalVM，不能使用普通的windows自带的命令行窗口，得使用VS提供的 **x64 Native Tools Command Prompt for VS 2019**，如果没有可以执行C:\Program Files (x86)\Microsoft Visual Studio\2019\BuildTools\VC\Auxiliary\Build\vcvars64.bat脚本来安装。

安装完之后其实就可以在 **x64 Native Tools Command Prompt for VS 2019**中去使用native-image命令去进行编译了。

但是，如果后续在编译过程中编译失败了，出现以下错误：

那么可以执行cl.exe，如果是中文，那就得修改为英文。

通过Visual Studio Installer来修改，比如：

可能一开始只选择了中文，手动选择英文，去掉中文，然后安装即可。

再次检查

这样就可以正常的编译了。

Hello World实战

新建一个简单的Java工程：

我们可以直接把graalvm当作普通的jdk的使用

我们也可以利用native-image命令来将字节码编译为二进制可执行文件。

打开**x64 Native Tools Command Prompt for VS 2019**，进入工程目录下，并利用javac将java文件编译为class文件：`javac -d . src/com/zhouyu/App.java`

此时的class文件因为有main方法，所以用java命令可以运行

我们也可以利用native-image来编译：

编译需要一些些。。。。。。时间。

编译完了之后就会在当前目录生成一个exe文件：

我们可以直接运行这个exe文件：

并且**运行这个exe文件是不需要操作系统上安装了JDK环境的。**

我们可以使用-o参数来指定exe文件的名字：

```
1 native-image com.zhouyu.App -o app
```

GraalVM的限制

GraalVM在编译成二进制可执行文件时，需要确定该应用到底用到了哪些类、哪些方法、哪些属性，从而把这些代码编译为机器指令（也就是exe文件）。但是我们一个应用中某些类可能是动态生成的，也就是应用运行后才生成的，为了解决这个问题，GraalVM提供了配置的方式，可以让我们在编译时告诉GraalVM哪些类会动态生成类，比如我们可以通过proxy-config.json、reflect-config.json来进行配置。

SpringBoot 3.0实战

然后新建一个Maven工程，添加SpringBoot依赖

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-parent</artifactId>
4   <version>3.0.0</version>
5 </parent>
6
7 <dependencies>
8   <dependency>
9       <groupId>org.springframework.boot</groupId>
10      <artifactId>spring-boot-starter-web</artifactId>
11   </dependency>
12 </dependencies>
```

以及SpringBoot的插件

```
1 <build>
2   <plugins>
3       <plugin>
4           <groupId>org.graalvm.buildtools</groupId>
5           <artifactId>native-maven-plugin</artifactId>
6       </plugin>
7       <plugin>
8           <groupId>org.springframework.boot</groupId>
9           <artifactId>spring-boot-maven-plugin</artifactId>
10      </plugin>
11   </plugins>
12 </build>
```

以及一些代码

```
1 @RestController
2 public class ZhouyuController {
3
4     @Autowired
5     private UserService userService;
6
7     @GetMapping("/demo")
8     public String test() {
9         return userService.test();
10    }
11
12 }
```

```
1 package com.zhouyu;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class UserService {
7
8     public String test(){
9         return "hello zhouyu";
10    }
11 }
12
```

```
1 package com.zhouyu;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
```

```
6 @SpringBootApplication
7 public class MyApplication {
8     public static void main(String[] args) {
9         SpringApplication.run(MyApplication.class, args);
10    }
11 }
12
```

这本身就是一个普通的SpringBoot工程，所以可以使用我们之前的方式使用，同时也支持利用native-image命令把整个SpringBoot工程编译成为一个exe文件。

同样在 **x64 Native Tools Command Prompt for VS 2019**中，进入到工程目录下，执行`mvn -Pnative native:compile`进行编译就可以了，就能在target下生成对应的exe文件，后续只要运行exe文件就能启动应用了。

在执行命令之前，请确保环境变量中设置的`graalvm`的路径。

编译完成截图：

这样，我们就能够直接运行这个exe来启动我们的SpringBoot项目了。

Docker SpringBoot3.0 实战

我们可以直接把SpringBoot应用对应的本地可执行文件构建为一个Docker镜像，这样就能跨操作系统运行了。

Buildpacks，类似Dockerfile的镜像构建技术

注意要安装docker，并启动docker

注意这种方式并不要求你机器上安装了GraalVM，会由SpringBoot插件利用/paketo-buildpacks/native-image来生成本地可执行文件，然后打入到容器中

Docker镜像名字中不能有大写字母，我们可以配置镜像的名字：

```
1 <properties>
2   <maven.compiler.source>17</maven.compiler.source>
3   <maven.compiler.target>17</maven.compiler.target>
4   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
5   <spring-boot.build-image.imageName>springboot3demo</spring-boot.build-
    image.imageName>
6 </properties>
```

然后执行：

```
1 mvn -Pnative spring-boot:build-image
```

来生成Docker镜像，成功截图：

执行完之后，就能看到docker镜像了：

然后就可以运行容器了：

```
1 docker run --rm -p 8080:8080 springboot3demo
```

如果要传参数，可以通过-e

```
1 docker run --rm -p 8080:8080 -e methodName=test springboot3demo
```

不过代码中，得通过以下代码获取：

```
1 String methodName = System.getenv("methodName")
```

建议工作中直接使用Environment来获取参数：

RuntimeHints

假如应用中有如下代码：

```
1  /**
2   * 作者：周瑜大都督
3   */
4  public class ZhouyuService {
5
6      public String test(){
7          return "zhouyu";
8      }
9  }
```

```
1  @Component
2  public class UserService {
3
4      public String test(){
5
6          String result = "";
7          try {
8              Method test = ZhouyuService.class.getMethod("test", null);
9              result = (String) test.invoke(ZhouyuService.class.newInstance(), null);
10         } catch (NoSuchMethodException e) {
11             throw new RuntimeException(e);
12         } catch (InvocationTargetException e) {
13             throw new RuntimeException(e);
14         } catch (IllegalAccessException e) {
15             throw new RuntimeException(e);
16         } catch (InstantiationException e) {
17             throw new RuntimeException(e);
18         }
19
20         return result;
21     }
22
23 }
```


在UserService中，通过反射的方式使用到了ZhouyuService的无参构造方法

(ZhouyuService.class.newInstance())，如果我们不做任何处理，那么打成二进制可执行文件后是运行不了的，可执行文件中是没有ZhouyuService的无参构造方法的，会报如下错误：

我们可以通过Spring提供的Runtime Hints机制来间接的配置reflect-config.json。

方式一：RuntimeHintsRegistrar

提供一个RuntimeHintsRegistrar接口的实现类，并导入到Spring容器中就可以了：

```
1  @Component
2  @ImportRuntimeHints(UserService.ZhouyuServiceRuntimeHints.class)
3  public class UserService {
4
5      public String test(){
6
7          String result = "";
8          try {
9              Method test = ZhouyuService.class.getMethod("test", null);
10             result = (String) test.invoke(ZhouyuService.class.newInstance(), null);
11         } catch (NoSuchMethodException e) {
12             throw new RuntimeException(e);
13         } catch (InvocationTargetException e) {
14             throw new RuntimeException(e);
15         } catch (IllegalAccessException e) {
16             throw new RuntimeException(e);
17         } catch (InstantiationException e) {
18             throw new RuntimeException(e);
19         }
20
21
22         return result;
23     }
24 }
```

```

25     static class ZhouyuServiceRuntimeHints implements RuntimeHintsRegistrar {
26
27         @Override
28         public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
29             try {
30
31 hints.reflection().registerConstructor(ZhouyuService.class.getConstructor(),
32 ExecutableMode.INVOKE);
33
34             } catch (NoSuchMethodException e) {
35                 throw new RuntimeException(e);
36             }
37         }
38     }
39 }

```

方式二：@RegisterReflectionForBinding

```

1  @RegisterReflectionForBinding(ZhouyuService.class)
2  public String test(){
3
4      String result = "";
5      try {
6          Method test = ZhouyuService.class.getMethod("test", null);
7          result = (String) test.invoke(ZhouyuService.class.newInstance(), null);
8      } catch (NoSuchMethodException e) {
9          throw new RuntimeException(e);
10     } catch (InvocationTargetException e) {
11         throw new RuntimeException(e);
12     } catch (IllegalAccessException e) {
13         throw new RuntimeException(e);
14     } catch (InstantiationException e) {
15         throw new RuntimeException(e);
16     }
17
18
19     return result;
20 }

```

注意

如果代码中的methodName是通过参数获取的，那么GraalVM在编译时就不能知道到底会使用到哪个方法，那么test方法也要利用RuntimeHints来进行配置。

```
1  @Component
2  @ImportRuntimeHints(UserService.ZhouyuServiceRuntimeHints.class)
3  public class UserService {
4
5      public String test(){
6
7          String methodName = System.getProperty("methodName");
8
9          String result = "";
10         try {
11             Method test = ZhouyuService.class.getMethod(methodName, null);
12             result = (String) test.invoke(ZhouyuService.class.newInstance(), null);
13         } catch (NoSuchMethodException e) {
14             throw new RuntimeException(e);
15         } catch (InvocationTargetException e) {
16             throw new RuntimeException(e);
17         } catch (IllegalAccessException e) {
18             throw new RuntimeException(e);
19         } catch (InstantiationException e) {
20             throw new RuntimeException(e);
21         }
22
23
24         return result;
25     }
26
27     static class ZhouyuServiceRuntimeHints implements RuntimeHintsRegistrar {
28
29         @Override
30         public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
31             try {
```

```

32 hints.reflection().registerConstructor(ZhouyuService.class.getConstructor(),
ExecutableMode.INVOKE);

33 hints.reflection().registerMethod(ZhouyuService.class.getMethod("test"),
ExecutableMode.INVOKE);

34         } catch (NoSuchMethodException e) {
35             throw new RuntimeException(e);
36         }
37     }
38 }
39 }

```

或者使用了JDK动态代理：

```

1 public String test() throws ClassNotFoundException {
2
3     String className = System.getProperty("className");
4     Class<?> aClass = Class.forName(className);
5
6     Object o = Proxy.newProxyInstance(UserService.class.getClassLoader(), new
Class[]{aClass}, new InvocationHandler() {
7         @Override
8         public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
9             return method.getName();
10        }
11    });
12
13    return o.toString();
14 }

```

那么也可以利用RuntimeHints来进行配置要代理的接口：

```

1 public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
2     hints.proxies().registerJdkProxy(UserInterface.class);
3 }

```

方式三：@Reflective

对于反射用到的地方，我们可以直接加一个@Reflective，前提是ZhouyuService得是一个Bean：

```
1  @Component
2  public class ZhouyuService {
3
4      @Reflective
5      public ZhouyuService() {
6      }
7
8      @Reflective
9      public String test(){
10         return "zhouyu";
11     }
12 }
```

以上Spring6提供的RuntimeHints机制，我们可以使用该机制更方便的告诉GraalVM我们额外用到了哪些类、接口、方法等信息，最终Spring会生成对应的reflect-config.json、proxy-config.json中的内容，GraalVM就知道了。

Spring AOT的源码实现

流程图：<https://www.processon.com/view/link/63edeea8440e433d3d6a88b2>

SpringBoot 3.0插件实现原理

上面的SpringBoot3.0实战过程中，我们在利用image-native编译的时候，target目录下会生成一个spring-aot文件夹：

这个spring-aot文件夹是编译的时候spring boot3.0的插件生成的，resources/META-INF/native-image文件夹中的存放的就是graalvm的配置文件。

当我们执行`mvn -Pnative native:compile`时，实际上执行的是插件native-maven-plugin的逻辑。我们可以执行`mvn help:describe -Dplugin=org.graalvm.buildtools:native-maven-plugin -Ddetail`

来查看这个插件的详细信息。

发现native:compile命令对应的实现类为NativeCompileMojo，并且会先执行package这个命令，从而会执行process-aot命令，因为spring-boot-maven-plugin插件中有如下配置：

```
我们可以执行mvn help:describe -Dplugin=org.springframework.boot:spring-boot-maven-plugin -Ddetail
```

发现对应的phase为：prepare-package，所以会在打包之前执行ProcessAotMojo。

所以，我们在运行`mvn -Pnative native:compile`时，会先编译我们自己的java代码，然后执行`executeAot()`方法（会生成一些Java文件并编译成class文件，以及GraalVM的配置文件），然后才执行利用GraalVM打包出二进制可执行文件。

对应的源码实现：

maven插件在编译的时候，就会调用到`executeAot()`这个方法，这个方法会：

1. 先执行`org.springframework.boot.SpringApplicationAotProcessor`的`main`方法
2. 从而执行`SpringApplicationAotProcessor`的`process()`
3. 从而执行`ContextAotProcessor`的`doProcess()`，从而会生成一些Java类并放在`spring-aot/main/sources`目录下，详情看后文
4. 然后把生成在`spring-aot/main/sources`目录下的Java类进行编译，并把对应class文件放在项目的编译目录下`target/classes`
5. 然后把`spring-aot/main/resources`目录下的`graalvm`配置文件复制到`target/classes`
6. 然后把`spring-aot/main/classes`目录下生成的class文件复制到`target/classes`

Spring AOT核心原理

以下只是一些关键源码，详细内容请看直播视频。

prepareApplicationContext会直接启动我们的SpringBoot，并在触发contextLoaded事件后，返回所创建的Spring对象，注意此时还没有扫描Bean。

```
1  protected ClassName performAotProcessing(GenericApplicationContext applicationContext)
   {
2      FileSystemGeneratedFiles generatedFiles = createFileSystemGeneratedFiles();
3
4      DefaultGenerationContext generationContext = new
        DefaultGenerationContext(createClassNameGenerator(), generatedFiles);
5
6      ApplicationContextAotGenerator generator = new ApplicationContextAotGenerator();
7
8      // 会进行扫描，并且根据扫描得到的BeanDefinition生成对应的Xx_BeanDefinitions.java文件
9      // 并返回com.zhouyu.MyApplication__ApplicationContextInitializer
10     ClassName generatedInitializerClassName =
        generator.processAheadOfTime(applicationContext, generationContext);
11
12     // 因为后续要通过反射调用com.zhouyu.MyApplication__ApplicationContextInitializer的构造方法
13     // 所以将相关信息添加到reflect-config.json对应的RuntimeHints中去
14     registerEntryPointHint(generationContext, generatedInitializerClassName);
15
16     // 生成source目录下的Java文件
17     generationContext.writeGeneratedContent();
18
19     // 将RuntimeHints中的内容写入resource目录下的Graalvm的各个配置文件中
20     writeHints(generationContext.getRuntimeHints());
21     writeNativeImageProperties(getDefaultNativeImageArguments(getApplicationClass().getName()));
22
23     return generatedInitializerClassName;
24 }
```

```
1  public ClassName processAheadOfTime(GenericApplicationContext applicationContext,
```

```

2      GenerationType generationContext) {
3      return withCglibClassHandler(new CglibClassHandler(generationContext), () -> {
4
5          // 会进行扫描，并找到beanType是代理类的请求，把代理类信息设置到RuntimeHints
中
6
7          applicationContext.refreshForAotProcessing(generationContext.getRuntimeHints());
8
9          // 拿出Bean工厂，扫描得到的BeanDefinition对象在里面
10
11          DefaultListableBeanFactory beanFactory =
12          applicationContext.getDefaultListableBeanFactory();
13
14          ApplicationContextInitializationCodeGenerator codeGenerator =
15          new
16          ApplicationContextInitializationCodeGenerator(generationContext);
17
18          // 核心
19          new
20          BeanFactoryInitializationAotContributions(beanFactory).applyTo(generationContext,
21          codeGenerator);
22
23          return codeGenerator.getGeneratedClass().getName();
24      });
25  }

```

```

1  BeanFactoryInitializationAotContributions(DefaultListableBeanFactory beanFactory) {
2      // 把aot.factories文件的加载器以及BeanFactory，封装成为一个Loader对象，然后传入
3      this(beanFactory, AotServices.factoriesAndBeans(beanFactory));
4  }

```

```

1  BeanFactoryInitializationAotContributions(DefaultListableBeanFactory beanFactory,
2
3      AotServices.Loader loader) {
4
5      // getProcessors()中会从aot.factories以及beanfactory中拿出
6      BeanFactoryInitializationAotProcessor类型的Bean对象
7
8      // 同时还会添加一个RuntimeHintsBeanFactoryInitializationAotProcessor

```



```
6  this.contributions = getContributions(beanFactory, getProcessors(loader));
7  }
```

```
1  private List<BeanFactoryInitializationAotContribution> getContributions(
2      DefaultListableBeanFactory beanFactory,
3      List<BeanFactoryInitializationAotProcessor> processors) {
4
5      List<BeanFactoryInitializationAotContribution> contributions = new ArrayList<>();
6
7      // 逐个调用BeanFactoryInitializationAotProcessor的processAheadOfTime()开始处理
8      for (BeanFactoryInitializationAotProcessor processor : processors) {
9          BeanFactoryInitializationAotContribution contribution =
processor.processAheadOfTime(beanFactory);
10         if (contribution != null) {
11             contributions.add(contribution);
12         }
13     }
14     return Collections.unmodifiableList(contributions);
15 }
```

总结一下，在SpringBoot项目编译时，最终会通过BeanFactoryInitializationAotProcessor来生成Java文件，或者设置RuntimeHints，后续会把写入Java文件到磁盘，将RuntimeHints中的内容写入GraalVM的配置文件，再后面会编译Java文件，再后面就会基于生成出来的GraalVM配置文件打包出二进制可执行文件了。

所以我们要看Java文件怎么生成的，RuntimeHints如何收集的就看具体的BeanFactoryInitializationAotProcessor就行了。

比如:

1. 有一个BeanRegistrationsAotProcessor，它就会负责生成Xx_BeanDefinition.java以及Xx_ApplicationContextInitializer.java、Xx_BeanFactoryRegistrations.java中的内容
2. 还有一个RuntimeHintsBeanFactoryInitializationAotProcessor，它负责从aot.factories文件以及BeanFactory中获取RuntimeHintsRegistrar类型的对象，以及会找到@ImportRuntimeHints所导入的RuntimeHintsRegistrar对象，最终就是从这些RuntimeHintsRegistrar中设置RuntimeHints。

Spring Boot3.0启动流程

在run()方法中，SpringBoot会创建一个Spring容器，但是SpringBoot3.0中创建容器逻辑为：

```
1 private ConfigurableApplicationContext createContext() {  
2     if (!AotDetector.useGeneratedArtifacts()) {  
3         return new AnnotationConfigServletWebServerApplicationContext();  
4     }  
5     return new ServletWebServerApplicationContext();  
6 }
```

如果没有使用AOT，那么就会创建AnnotationConfigServletWebServerApplicationContext，它里面会添加ConfigurationClassPostProcessor，从而会解析配置类，从而会扫描。

而如果使用了AOT，则会创建ServletWebServerApplicationContext，它就是一个空容器，它里面没有ConfigurationClassPostProcessor，所以后续不会触发扫描了。

创建完容器后，就会找到MyApplication_ApplicationContextInitializer，开始向容器中注册BeanDefinition。

后续就是创建Bean对象了。