Spring Boot Maven Plugin Documentation

Stephane Nicoll, Andy Wilkinson, Scott Frederick

Table of Contents

Chapter 1. 介绍

Spring Boot Maven 插件在 Apache Maven 中提供了对 Spring Boot 支持.
它允许您打包可执行的 jar 或 war 文件,运行 Spring Boot 应用程序
,生成构建信息以及在运行集成测试之前启动 Spring Boot 应用程序.

Chapter 2. 入门

要使用 Spring Boot Maven 插件,请在 pom.xml 的 plugins 部分中包含适当的 XML,如以下示例所示:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!-- ... -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

如果使用里程碑或快照版本,则还需要添加适当的 pluginRepository 元素,如下所示:

```
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <url>https://repo.spring.io/snapshot</url>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <url>https://repo.spring.io/milestone</url>
    </pluginRepository>
</pluginRepositories>
```

Chapter 3. 使用插件

Maven 用户可以继承 spring-boot-starter-parent 项目以获取合适的默认值,父项目提供了以下功能:

- Java 1.8 作为默认编译器.
- 源代码使用 UTF-8 编码.
- using-boot-dependency-management,依赖管理部分, 继承自 spring-boot-dependencies 的 POM,允许您省略常见依赖的 <version> 标签.
- 执行 repackage id 重新执行 repackage goal
- 合理的 资源过滤.
- 合适的插件配置 (exec plugin, Git commit ID, and shade).
- 对 application.properties 和 application.yml 资源的合理过滤,包括特定 profile 的文件(例如 , application-dev.properties 和 application-dev.yml)

注意: 由于 application.properties 和 application.yml 文件接受 Spring 风格的占位符 (${…}) ,因此 Maven 改为使用 @..@ 占位符 (您可以使用 Maven 的 resource.delimiter 属性重写它)

3.1. 继承 Starter Parent

配置项目继承 spring-boot-starter-parent,只需要按以下方式设置 parent:

```
<!-- Inherit defaults from Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.5</version>
</parent>
```

您只需要在此依赖上指定 Spring Boot 的版本号. 如果您要导入其它 starter,则可以放心地省略版本号.

通过该设置,您还可以重写自己项目中的配置属性来覆盖个别依赖. 例如,要升级到另一个 SLF4J 库和 Spring Data 发行版本,您需要将以下内容添加到 pom.xml 文件中.

```
<properties>
    <slf4j.version>1.7.30</slf4j.version>
    <spring-data-releasetrain.version>Moore-SR6</spring-data-
releasetrain.version>
</properties>
```

查看 Dependency versions Appendix 获取等多的版本依赖关系.

3.2. 不使用父 POM

不是每个人都喜欢从 spring-boot-starter-parent 继承 POM.
您可能需要使用自己公司标准的父 POM,或者您可能只是希望明确地声明所有 Maven 配置.

如果您不想使用 spring-boot-starter-parent,则仍然可以通过使用 `import`scope 依赖来获得依赖管理 (但不是插件管理) 的好处:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <!-- Import dependency management from Spring Boot -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.4.5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

如上所述,上述示例设置不会让您使用属性来覆盖个别依赖. 要达到相同的目的,需要在 spring-boot-dependencies 项之前在项目的 dependencyManagement 中添加一项. 例如,要升级到另一个 SLF4J 库和 Spring Data 发行版,您可以将以下元素添加到 pom.xml 中:

```
<dependencyManagement>
    <dependencies>
        <!-- Override SLF4J provided by Spring Boot -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>
        <!-- Override Spring Data release train provided by Spring Boot -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-releasetrain</artifactId>
            <version>2020.0.0-SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.4.5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

3.3. 通过命令行重写配置

使用 `spring-boot`,该插件提供了许多用户属性,可让您从命令行自定义配置.

例如,您可以调整配置文件以在运行应用程序时启用,如下所示:

```
$ mvn spring-boot:run -Dspring-boot.run.profiles=dev,local
```

如果您希望同时拥有一个默认值,同时允许在命令行上覆盖它,那么您应该结合使用用户提供的项目属性和 `MOJO` 配置.

```
<project>
    <properties>
        <app.profiles>local,dev</app.profiles>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <profiles>${app.profiles}</profiles>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

以上内容确保默认情况下启用了 `local` 和 `dev`. 也可以在命令行中覆盖它:

```
$ mvn spring-boot:run -Dapp.profiles=test
```

Chapter 4. 目标

Spring Boot Plugin 有以下目标:

Goal	Description
spring-boot:build-image	使用 buildpack 将应用程序打包到 OCI 镜像中.
spring-boot:build-info	根据当前 MavenProject 的内容生成 build-info.properties 文件.
spring-boot:help	在 spring-boot-maven-plugin 上显示帮助信息. 使用 mvn spring-boot:help -Ddetail=true -Dgoal=<goal-name> 以显示参数详细信息..
spring-boot:repackage	重新打包现有的 JAR 和 WAR 归档文件, 以便可以使用 java -jar 在命令行中执行它们. 使用 layout=NONE 也可以简单地用于打包具有嵌套依赖关系的 JAR (没有主类不能执行) .
spring-boot:run	运行应用程序.
spring-boot:start	启动 spring 应用程序. 与 run goal 相反, 这不会阻止并允许其他 goal 在应用程序上运行. 此 goal 通常用于集成测试方案中, 在该方案中, 应用程序在测试套件之前启动, 而在测试套件之后停止.
spring-boot:stop	停止已通过 "start" 目标启动的应用程序. 通常在测试套件完成后调用.

Chapter 5. 打包可执行文件

该插件可以创建包含应用程序所有依赖的可执行文件（jar 文件和 war 文件），然后可以使用 java -jar 运行。

打包可执行文件是由重 repackage 目标执行的，如以下示例所示：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

如果您使用的是 spring-boot-starter-parent，则已经使用 repackage 执行 ID 预配置了此类执行，因此只需要添加插件定义。

上面的示例重新打包了在 Maven 生命周期的打包阶段构建的 jar 或 war，包括在项目中定义的 provided 所有依赖。如果其中一些依赖需要排除，则可以使用 exclude 选项之一；有关更多详细信息，请参见 依赖排除 。

默认情况下，原始（即不可执行）artifact 被重命名为 .original，但是也可以使用自定义 classifier 保留原始 artifact。

当前不支持 maven-war-plugin 的 outputFileNameMapping 功能。

默认情况下，会自动排除 Devtools（您可以使用 excludeDevtools 属性进行控制）。为了使它与 war 打包一起使用，必须将 spring-boot-devtools 依赖设置为`optional`或具有 provided 的作用域。

该插件将重写清单,尤其是它管理 Main-Class 和 Start-Class. 如果默认设置不起作用,则必须在 Spring Boot 插件中配置值,而不是在 jar 插件中配置值. 清单中的 Main-Class 由 Spring Boot 插件的 layout 属性控制,如以下示例所示:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <mainClass>${start.class}</mainClass>
                <layout>ZIP</layout>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

layout 属性默认为由 archive 类型(jar 或 war)确定的值.以下是可用的布局:(jar or war). 以下布局可用:

- JAR: 常规的可执行 JAR 布局.
- WAR: 可执行的 WAR 布局. provided 依赖放在 WEB-INF/lib-provided 中,以避免在将 war 部署到 servlet 容器中时发生任何冲突.
- ZIP (又名 DIR): 与使用 PropertiesLauncher 的 JAR 布局相似.
- NONE: 捆绑所有依赖和项目资源.不捆绑引导加载程序.

5.1. Jars 分层

重新打包的 jar 分别在 BOOT-INF/classes 和 BOOT-INF/lib 中包含应用程序的类和依赖. 对于需要从 jar 的内容中构建 docker 镜像的情况,能够进一步分隔这些目录以便将它们写入不同的层中很有用.

分层的 jar 使用与常规重新打包的 jar 相同的布局,
但是包括了描述每个层的附加元数据文件.

默认情况下,定义了以下层:

- dependencies: 包含所有的依赖,但不包括 SNAPSHOT 版本的依赖.
- spring-boot-loader: 用于加载 jar.
- snapshot-dependencies: 包含所有的 SNAPSHOT 版本依赖 .
- application: 应用程序类和资源.

通过查看当前构建的所有模块来确定模块依赖关系. 如果一个模块依赖只能被解析,
因为它已经被安装到 Maven 的本地缓存中, 并且它不是当前构建的一部分,
那么它将被标识为常规依赖.

层的顺序很重要,因为它确定了部分应用程序更改时可以缓存先前的层的可能性. 默认顺序是 dependencies, spring-boot-loader, snapshot-dependencies, application.
应该首先添加最不可能更改的内容,然后添加有可能更改的层.

重新打包 jar 默认包含 layers.idx 文件. 禁用该特性的方法如下:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <layers>
                        <enabled>false</enabled>
                    </layers>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

5.1.1. 自定义层的配置

根据您的应用程序,您可能想要调整层的创建方式和添加新层的方式.可以使用一个单独的配置文件来完成此操作,该文件应如下所示进行注册:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <layers>
                        <enabled>true</enabled>

<configuration>${project.basedir}/src/layers.xml</configuration>
                    </layers>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

配置文件描述了如何将 `jar` 分为几层,以及这些层的顺序
.下面的示例显示如何显式定义上述默认顺序:

```
<layers xmlns="http://www.springframework.org/schema/boot/layers"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/boot/layers
                      https://www.springframework.org/schema/boot/layers/layers-
{spring-boot-xsd-version}.xsd">
    <application>
        <into layer="spring-boot-loader">
            <include>org/springframework/boot/loader/**</include>
        </into>
        <into layer="application" />
    </application>
    <dependencies>
        <into layer="application">
            <includeModuleDependencies />
        </into>
        <into layer="snapshot-dependencies">
            <include>*:*:*SNAPSHOT</include>
        </into>
        <into layer="dependencies" />
    </dependencies>
    <layerOrder>
        <layer>dependencies</layer>
        <layer>spring-boot-loader</layer>
        <layer>snapshot-dependencies</layer>
        <layer>application</layer>
    </layerOrder>
</layers>
```

`layers` XML 分为三部分:

- `<application>` 定义应如何对应用程序类和资源进行分层.
- `<dependencies>` 定义应层之间的依赖关系.
- `<layerOrder>` 定义应写入层的顺序.

嵌套的 `<into>` 用于 `<application>` 和 `<dependencies>` 中,以声明层的内容.
从上到下,按照定义的顺序评估. 较早的块未声明的任何内容仍然可供后续块使用.

`<into>` 块使用嵌套的 `<include>` 和 `<exclude>` 元素声明内容. `<application>` 对 include/exclude 表达式使用 Ant 风格的匹配模式. `<dependencies>` 使用 `group:artifact[:version]` 模式.它还提供了 `<includeModuleDependencies />` 和 `<excludeModuleDependencies />` 元素, 可以用来包含或排除本地模块依赖.

如果未定义 `<include>`,则将考虑所有内容（较早的块未声明）。

如果未定义 `<exclude>`,则不应用任何排除项。

查看上面的 `<dependencies>` 示例,我们可以看到第一个 `<into>` 将声明所有的项目依赖,第二个 `<into>` 将为 `snapshot-dependencies` 声明所有快照依赖。最后的 `<into>` 将声明 `dependencies` 层的所有剩余内容（在这种情况下,不是 项目 或 SNAPSHOT 的任何依赖）。

`<application>` 块具有相似的规则。首先声明 `spring-boot-loader` 层的 `org/springframework/boot/loader/**` 内容。然后为 `application` 层声明所有剩余的类和资源。

`<into>` 块的定义顺序通常与层的写入顺序不同。因此,必须始终包含 `<layerOrder>` 元素,并且必须覆盖 `<into>` 块引用的所有层。

5.2. spring-boot:repackage

`org.springframework.boot:spring-boot-maven-plugin:2.4.5`

重新打包现有的 JAR 和 WAR 归档文件，以便可以使用 `java -jar` 在命令行中执行它们。使用 `layout=NONE` 也可以简单地用于打包具有嵌套依赖关系的 JAR（没有主类不能执行）。

5.2.1. Required parameters

Name	Type	Default
outputDirectory	File	${project.build.directory}

5.2.2. Optional parameters

Name	Type	Default
attach	boolean	true
classifier	String	
embeddedLaunchScript	File	

Name	Type	Default
embeddedLaunchScriptProperties	Properties	
excludeDevtools	boolean	true
excludeGroupIds	String	
excludes	List	
executable	boolean	false
includeSystemScope	boolean	false
includes	List	
layers	Layers	
layout	LayoutType	
layoutFactory	LayoutFactory	
mainClass	String	
outputTimestamp	String	${project.build.outputTimestamp}
requiresUnpack	List	
skip	boolean	false

5.2.3. Parameter details

attach

Attach the repackaged archive to be installed into your local Maven repository or deployed to a remote repository. If no classifier has been configured, it will replace the normal jar. If a classifier has been configured such that the normal jar and the repackaged jar are different, it will be attached alongside the normal jar. When the property is set to false, the repackaged archive will not be installed or deployed.

Name	attach

Type	boolean
Default value	true
User property	
Since	1.4.0

classifier

Classifier to add to the repackaged archive. If not given, the main artifact will be replaced by the repackaged archive. If given, the classifier will also be used to determine the source archive to repackage: if an artifact with that classifier already exists, it will be used as source and replaced. If no such artifact exists, the main artifact will be used as source and the repackaged archive will be attached as a supplemental artifact with that classifier. Attaching the artifact allows to deploy it alongside to the original one, see $1[$2].

Name	classifier
Type	java.lang.String
Default value	
User property	
Since	1.0.0

embeddedLaunchScript

The embedded launch script to prepend to the front of the jar if it is fully executable. If not specified the 'Spring Boot' default script

will be used.

Name	embeddedLaunchScript
Type	java.io.File
Default value	
User property	
Since	1.3.0

embeddedLaunchScriptProperties

Properties that should be expanded in the embedded launch script.

Name	embeddedLaunchScriptProperties
Type	java.util.Properties
Default value	
User property	
Since	1.3.0

excludeDevtools

Exclude Spring Boot devtools from the repackaged archive.

Name	excludeDevtools
Type	boolean

Default value	true
User property	spring-boot.repackage.excludeDevtools
Since	1.3.0

excludeGroupIds

Comma separated list of groupId names to exclude (exact match).

Name	excludeGroupIds
Type	java.lang.String
Default value	
User property	spring-boot.excludeGroupIds
Since	1.1.0

excludes

Collection of artifact definitions to exclude. The Exclude element defines mandatory groupId and artifactId properties and an optional classifier property.

Name	excludes
Type	java.util.List
Default value	

User property	spring-boot.excludes
Since	1.1.0

executable

Make a fully executable jar for *nix machines by prepending a launch script to the jar. <p> Currently, some tools do not accept this format so you may not always be able to use this technique. For example, jar -xf may silently fail to extract a jar or war that has been made fully-executable. It is recommended that you only enable this option if you intend to execute it directly, rather than running it with java -jar or deploying it to a servlet container.

Name	executable
Type	boolean
Default value	false
User property	
Since	1.3.0

includeSystemScope

Include system scoped dependencies.

Name	includeSystemScope
Type	boolean
Default value	false

User property	
Since	1.4.0

includes

Collection of artifact definitions to include. The Include element defines mandatory groupId and artifactId properties and an optional mandatory groupId and artifactId properties and an optional classifier property.

Name	includes
Type	java.util.List
Default value	
User property	spring-boot.includes
Since	1.2.0

layers

Layer configuration with options to disable layer creation, exclude layer tools jar, and provide a custom layers configuration file.

Name	layers
Type	org.springframework.boot.maven.Layers
Default value	

User property	
Since	2.3.0

layout

The type of archive (which corresponds to how the dependencies are laid out inside it). Possible values are JAR, WAR, ZIP, DIR, NONE. Defaults to a guess based on the archive type.

Name	layout
Type	org.springframework.boot.maven.AbstractPackagerMojo$LayoutType
Default value	
User property	spring-boot.repackage.layout
Since	1.0.0

layoutFactory

The layout factory that will be used to create the executable archive if no explicit layout is set. Alternative layouts implementations can be provided by 3rd parties.

Name	layoutFactory
Type	org.springframework.boot.loader.tools.LayoutFactory
Default value	

User property	
Since	1.5.0

mainClass

The name of the main class. If not specified the first compiled class found that contains a `main` method will be used.

Name	mainClass
Type	java.lang.String
Default value	
User property	
Since	1.0.0

outputDirectory

Directory containing the generated archive.

Name	outputDirectory
Type	java.io.File
Default value	${project.build.directory}
User property	
Since	1.0.0

outputTimestamp

Timestamp for reproducible output archive entries, either formatted as ISO 8601 (yyyy-MM-dd'T'HH:mm:ssXXX) or an int representing seconds since the epoch. Not supported with war packaging.

Name	outputTimestamp
Type	java.lang.String
Default value	${project.build.outputTimestamp}
User property	
Since	2.3.0

requiresUnpack

A list of the libraries that must be unpacked from fat jars in order to run. Specify each library as a <dependency> with a <groupId> and a <artifactId> and they will be unpacked at runtime.

Name	requiresUnpack
Type	java.util.List
Default value	
User property	
Since	1.1.0

skip

Skip the execution.

Name	skip
Type	boolean
Default value	false
User property	spring-boot.repackage.skip
Since	1.2.0

5.3. 示例

5.3.1. 自定义 Classifier

默认情况下,repackage 目标将原始 artifact 替换为重新打包的 artifact.
对于表示应用程序的模块来说,这是理智的行为,但是如果您的模块用作另一个模块的依赖,则需要为重新打包的模块提供 classifier. 这样做的原因是应用程序类打包在 BOOT-INF/classes 中,因此从属模块无法加载经过重新打包的 jar 的类.

如果是这种情况,或者您希望保留原始 artifact,然后将重新打包的 artifact 与其他 classifier 相连,请配置插件,如以下示例所示:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <classifier>exec</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

如果您使用的是 `spring-boot-starter-parent`,则 `repackage` 目标将在 `ID` 为 `repackage` 的执行中自动执行.在该设置中,只需要指定配置,如以下示例所示:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <configuration>
                            <classifier>exec</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

这种配置将生成两个 artifacts : 原始 artifacts
和由重新打包目标产生的重新打包的对应零件. 两者都将透明安装/部署.

如果要以替换主要 artifact 的相同方式重新打包辅助 artifact,也可以使用相同的配置.
以下配置使用重新打包的应用程序安装/部署单个 task 分类的 artifact:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>jar</goal>
                        </goals>
                        <phase>package</phase>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

由于 maven-jar-plugin 和 spring-boot-maven-plugin 都在同一阶段运行

,因此必须首先定义 jar 插件（以便在重新打包目标之前运行），这一点很重要。同样，如果您使用的是 spring-boot-starter-parent，可以将其简化如下：

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <executions>
                    <execution>
                        <id>default-jar</id>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

5.3.2. 自定义名称

如果您需要重新打包的 jar 的本地名称与项目的 artifactId 属性定义的本地名称不同，请使用标准的 finalName，如以下示例所示：

```
<project>
    <build>
        <finalName>my-app</finalName>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

此配置将在 `target/my-app.jar` 中生成重新打包的文件.

5.3.3. 本地 Repackaged Artifact

默认情况下,`repackage` 目标将原始 `artifact` 替换为可执行文件. 如果您只需要部署原始 `jar`,但仍能够使用常规文件名运行您的应用,请按以下步骤配置插件:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <attach>false</attach>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

这种配置生成两个 artifacts ：原始 artifacts 和由 repackage 目标产生的可执行计数器部分.仅原始的将被安装/部署.

5.3.4. 自定义布局

Spring Boot 使用附加 jar 文件中定义的自定义布局工厂对该项目的 jar 文件重新打包,该工厂作为对构建插件的依赖关系提供:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <layoutFactory
implementation="com.example.CustomLayoutFactory">
                                <customProperty>value</customProperty>
                            </layoutFactory>
                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.example</groupId>
                        <artifactId>custom-layout</artifactId>
                        <version>0.0.1.BUILD-SNAPSHOT</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>
</project>
```

布局工厂是作为 `pom` 中明确指定的 `LayoutFactory` 的实现（来自 `spring-boot-loader-tools`）提供的. 如果插件的类路径上只有一个自定义 `LayoutFactory`,并且已在 `META-INF/spring.factories` 中列出,则无需在插件配置中显式设置它.

如果设置了 `layout` ,则总是会忽略布局工厂.

5.3.5. Dependency 排除

默认情况下,`repackage` 和 `run` 目标都将包括项目中定义的所有提供的依赖. Spring Boot 项目应将 `provided` 的依赖视为运行应用程序所需的 "container" 依赖.

这些依赖中的某些可能根本不需要,应该从可执行 jar 中排除. 为了保持一致,在运行应用程序时也不应该显示它们.

有两种方法可以在运行时打包/使用中排除依赖:

- 排除由 groupId 和 artifactId 标识的组件, 如有需要,可以使用可选的 classifier.
- 排除所有 groupId 的组件.

以下示例排除 com.foo:bar,只排除此 artifact:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>com.foo</groupId>
                            <artifactId>bar</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

此示例排除了属于 com.foo 组的所有组件:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludeGroupIds>com.foo</excludeGroupIds>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

5.3.6. Layered Jar Tools

创建分层 jar 时,spring-boot-jarmode-layertools jar 将作为依赖添加到 jar 中. 将此 jar 放在类路径上,您可以在特殊模式下启动应用程序
,该模式允许引导代码运行与您的应用程序完全不同的内容,例如,提取层的内容.
如果要排除此依赖关系,可以按以下方式进行:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <layers>
                        <enabled>true</enabled>
                        <includeLayerTools>false</includeLayerTools>
                    </layers>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

5.3.7. Custom Layers Configuration

默认设置将依赖分为快照和非快照,但是,您可能具有更复杂的规则. 例如
,您可能希望在专用层中隔离项目的公司特定依赖性. 以下 layers.xml

配置显示了一种这样的设置：

```
<layers xmlns="http://www.springframework.org/schema/boot/layers"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/boot/layers
                      https://www.springframework.org/schema/boot/layers/layers-
{spring-boot-xsd-version}.xsd">
    <application>
        <into layer="spring-boot-loader">
            <include>org/springframework/boot/loader/**</include>
        </into>
        <into layer="application" />
    </application>
    <dependencies>
        <into layer="snapshot-dependencies">
            <include>*:*:*SNAPSHOT</include>
        </into>
        <into layer="company-dependencies">
            <include>com.acme:*</include>
        </into>
        <into layer="dependencies"/>
    </dependencies>
    <layerOrder>
        <layer>dependencies</layer>
        <layer>spring-boot-loader</layer>
        <layer>snapshot-dependencies</layer>
        <layer>company-dependencies</layer>
        <layer>application</layer>
    </layerOrder>
</layers>
```

上面的配置使用所有带有 `com.acme` `groupId` 的库创建一个附加的 `company-dependencies` 层.

Chapter 6. 打包 OCI 镜像

该插件可以使用 Cloud Native Buildpacks (CNB) 从可执行 jar 文件创建 OCI image.可以使用 build-image 目标来构建镜像.

出于安全原因,镜像以非 root 用户身份构建和运行.有关更多详细信息,请参见 CNB specification.

build-image 任务不支持 打包成 war.

最简单的入门方法是在项目上调用 mvn spring-boot:build-image. 每当调用 package 阶段时,都可以自动创建镜像,如以下示例所示:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>build-image</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

尽管 buildpack 是从 executable archive 运行的,但不必先执行 repackage 目标,因为必要时会自动创建可执行存档.
当构建镜像重新打包应用程序时,它将应用与重新打包目标相同的设置,即可以使用 exclude 选项之一排除依赖,并且默认情况下会自动排除 Devtools (您可以使用 excludeDevtools 属性进行控制) .

6.1. Docker Daemon

build-image 任务需要访问 Docker 守护程序. 默认情况下,它将通过本地连接与 Docker 守护程序通信. 无需配置即可在所有支持的平台上使用 Docker Engine.

可以设置环境变量以将 build-image 任务配置为使用 minikube 提供的 Docker 守护程序. 下表显示了环境变量及其值:

Environment variable	Description
DOCKER_HOST	URL containing the host and port for the Docker daemon - e.g. tcp://192.168.99.100:2376
DOCKER_TLS_VERIFY	Enable secure HTTPS protocol when set to 1 (optional)
DOCKER_CERT_PATH	Path to certificate and key files for HTTPS (required if DOCKER_TLS_VERIFY=1, ignored otherwise)

在 Linux 和 macOS 上,启动 minikube 之后,可以使用命令 eval $(minikube docker-env) 设置这些环境变量.

也可以使用插件配置中的 docker 参数提供 Docker 守护进程连接信息.
下表汇总了可用参数:

Parameter	Description
host	Docker daemon 主机和端口 URL - e.g. tcp://192.168.99.100:2376
tlsVerify	当设置为 true 时,开启 HTTPS 协议(可选)
certPath	HTTPS 的证书和密钥文件的路径 (如果 tlsVerify 为 true 则需要, 否则将忽略)

更多信息, 请查看 examples.

6.2. Docker Registry

如果由 builder 或 runImage 参数指定的 Docker 镜像存储在需要身份验证的私有 Docker image registry 中，可以使用 docker.builderRegistry 参数提供身份验证凭据.

如果生成的 Docker 镜像要发布到 Docker image registry 中，可以使用 docker.publishRegistry 参数提供身份验证凭据.

为用户身份验证或身份令牌验证提供了参数. 关于支持的身份验证方法的更多信息，请参考 Docker registry 的文档.

下表总结了 docker.builderRegistry 和 docker.publishRegistry 的可用参数:

Parameter	Description
username	Docker image registry 的 username. Required for user authentication.
password	Password for the Docker image registry user. Required for user authentication.
url	Address of the Docker image registry. Optional for user authentication.
email	E-mail address for the Docker image registry user. Optional for user authentication.
token	Identity token for the Docker image registry user. Required for token authentication.

更多信息，请查看 examples.

6.3. Image Customizations

该插件调用一个 builder 来协助镜像的生成. 该构建器包括多个 buildpacks ,可以检查应用程序以影响生成的镜像. 默认情况下,插件选择一个生成镜像.
生成的镜像名称是从项目属性推导出的.

image 属性可用于配置构建器应如何在项目上运行.下表总结了可用的属性及其默认值:

Parameter	Description	User property	Default value
builder	Name of the Builder image to use.	spring-boot.build-image.builder	paketobuildpacks/builder:base
runImage	Name of the run image to use.	spring-boot.build-image.runImage	No default value, indicating the run image specified in Builder metadata should be used.
name	Image name for the generated image.	spring-boot.build-image.imageName	docker.io/library/${project.artifactId}:${project.version}
pullPolicy	Policy used to determine when to pull the builder and run images from the registry. Acceptable values are ALWAYS, NEVER, and IF_NOT_PRESENT.	spring-boot.build-image.pullPolicy	ALWAYS

Parameter	Description	User property	Default value
env	Environment variables that should be passed to the builder.		
cleanCache	Whether to clean the cache before building.	spring-boot.build-image.cleanCache	false
verboseLogging	Enables verbose logging of builder operations.		false
publish	Whether to publish the generated image to a Docker registry.	spring-boot.build-image.publish	false

该插件使用编译器的插件配置或 maven.compiler.target 属性检测项目的目标 Java 兼容性. 当使用默认的 Paketo 构建器和 buildpacks 时,插件会指示 buildpacks 安装相同的 Java 版本. 您可以覆盖此行为,如 builder configuration 示例中所示.

有关更多详细信息,另请参见 examples.

6.4. spring-boot:build-image

org.springframework.boot:spring-boot-maven-plugin:2.4.5

使用 buildpack 将应用程序打包到 OCI 镜像中

6.4.1. Required parameters

Name	Type	Default
sourceDirectory	File	${project.build.directory}

6.4.2. Optional parameters

Name	Type	Default
classifier	String	
docker	Docker	
excludeDevtools	boolean	true
excludeGroupIds	String	
excludes	List	
image	Image	
includeSystemScope	boolean	false
includes	List	
layers	Layers	
mainClass	String	
skip	boolean	false

6.4.3. Parameter details

classifier

查找 source archive 时使用的 Classifier.

Name	classifier
Type	java.lang.String
Default value	

User property	
Since	2.3.0

docker

Docker configuration options.

Name	docker
Type	org.springframework.boot.maven.Docker
Default value	
User property	
Since	2.4.0

excludeDevtools

从重新打包的 archive 中排除 Spring Boot devtools

Name	excludeDevtools
Type	boolean
Default value	true
User property	spring-boot.repackage.excludeDevtools
Since	1.3.0

excludeGroupIds

用逗号分隔的要排除的 groupId 名称列表（完全匹配）.

Name	excludeGroupIds
Type	java.lang.String
Default value	
User property	spring-boot.excludeGroupIds
Since	1.1.0

excludes

要排除的组件定义的集合. Exclude 元素必需定义的 groupId 和 artifactId 属性以及可选的 classifier 属性.

Name	excludes
Type	java.util.List
Default value	
User property	spring-boot.excludes
Since	1.1.0

image

镜像配置,包括 builder, runImage, name, env, cleanCache ,verboseLogging, pullPolicy 和 publish 选项.

Name	image
Type	org.springframework.boot.maven.Image
Default value	
User property	
Since	2.3.0

includeSystemScope

Include system scoped dependencies.

Name	includeSystemScope
Type	boolean
Default value	false
User property	
Since	1.4.0

includes

要包括的组件定义的集合. Exclude 元素必需定义的 groupId 和 artifactId 属性以及可选的 classifier 属性.

Name	includes
Type	java.util.List

Default value	
User property	spring-boot.includes
Since	1.2.0

layers

具有禁用层创建选项的层配置,排除分层工具的 jar,并提供自定义层配置文件.

Name	layers
Type	org.springframework.boot.maven.Layers
Default value	
User property	
Since	2.3.0

mainClass

主类的名称.如果未指定,将使用找到的第一个包含 main 方法的类.

Name	mainClass
Type	java.lang.String
Default value	

User property	
Since	1.0.0

skip

跳过 execution.

Name	skip
Type	boolean
Default value	false
User property	spring-boot.build-image.skip
Since	2.3.0

sourceDirectory

包含 source archive 的目录.

Name	sourceDirectory
Type	java.io.File
Default value	${project.build.directory}
User property	
Since	2.3.0

6.5. 示例

6.5.1. Custom Image Builder

如果需要自定义用于创建镜像的 Builder 或用于启动生成的镜像的运行镜像,请使用如下配置插件:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <builder>mine/java-cnb-builder</builder>
                        <runImage>mine/java-cnb-run</runImage>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

此配置将使用名称为 mine/java-cnb-builder 的构建器镜像的最新版本(latest tag),以及名为 mine/java-cnb-run 的运行镜像的最新版本(latest tag).

构建器和运行镜像也可以在命令行上指定,如本示例所示:

```
$ mvn spring-boot:build-image -Dspring-boot.build-image.builder=mine/java-cnb
-builder -Dspring-boot.build-image.runImage=mine/java-cnb-run
```

6.5.2. 构建器配置

如果构建器使用环境变量暴露配置选项,则可以使用 env 属性进行设置.

以下是配置在构建时由 Paketo Java buildpack configuring the JVM version 的示例:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <env>
                            <BP_JVM_VERSION>8.*</BP_JVM_VERSION>
                        </env>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

以类似的方式,Paketo Java buildpacks 支持 配置 JVM 运行时行为. 请参阅 Paketo 文档 以获取 Paketo Java buildpacks 支持的其他配置选项.

如果构建器在其运行的 Docker 守护程序与构建打包下载 artifacts 的网络位置之间存在网络代理,则需要将构建器配置为使用代理. 使用 Paketo 构建器时,可以通过设置 HTTPS_PROXY and/or HTTP_PROXY 环境变量来实现,如以下示例所示:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <env>
                            <HTTP_PROXY>http://proxy.example.com</HTTP_PROXY>
                            <HTTPS_PROXY>https://proxy.example.com</HTTPS_PROXY>
                        </env>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

6.5.3. Runtime JVM Configuration

Paketo Java buildpacks 通过设置环境变量 JAVA_TOOL_OPTIONS 来 配置 JVM 运行时环境.

当在容器中启动应用程序镜像时, 可以修改 buildpack 提供的 JAVA_TOOL_OPTIONS 值以自定义JVM运行时行为.

可以如 Paketo documentation 中所述在镜像中配置并应用于每个部署的环境变量修改, 并在以下示例中显示:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <env>
                            <BPE_DELIM_JAVA_TOOL_OPTIONS xml:space="preserve">
</BPE_DELIM_JAVA_TOOL_OPTIONS>
                            <BPE_APPEND_JAVA_TOOL_OPTIONS>-
XX:+HeapDumpOnOutOfMemoryError</BPE_APPEND_JAVA_TOOL_OPTIONS>
                        </env>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

6.5.4. 自定义镜像名称

默认情况下,镜像名称是根据 artifactId 和项目版本来推断的,例如 docker.io/library/${project.artifactId}:${project.version}.您可以控制名称 ,如以下示例所示:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <name>example.com/library/${project.artifactId}</name>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

此配置未显式提供标签,因此使用最新的.也可以使用 `${project.version}`,构建中可用的任何属性或硬编码版本来指定标签.

镜像名称也可以在命令行上指定,如以下示例所示:

```
$ mvn spring-boot:build-image -Dspring-boot.build
-image.imageName=example.com/library/my-app:v1
```

6.5.5. Image Publishing

生成的镜像可以发布到 Docker registry , 方法是启用 `publish` 选项并使用 `docker.publishRegistry` 配置 registry 的身份验证.

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>

<name>docker.example.com/library/${project.artifactId}</name>
                        <publish>true</publish>
                    </image>
                    <docker>
                        <publishRegistry>
                            <username>user</username>
                            <password>secret</password>
                            <url>https://docker.example.com/v1/</url>
                            <email>user@example.com</email>
                        </publishRegistry>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

也可以在命令行上指定 `publish` 选项，如下例所示：

```
$ mvn spring-boot:build-image -Dspring-boot.build
-image.imageName=docker.example.com/library/my-app:v1 -Dspring-boot.build
-image.publish=true
```

6.5.6. Docker 配置

如果你需要插件使用远程连接而不是默认的本地连接来与 Docker 守护进程通信，可以使用 `docker` 属性提供连接细节，如下所示：

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <docker>
                        <host>tcp://192.168.99.100:2376</host>
                        <tlsVerify>true</tlsVerify>
                        <certPath>/home/user/.minikube/certs</certPath>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

如果构建器或运行镜像存储在支持用户身份验证的私有 Docker registry 中，则可以使用 `docker.builderRegistry` 参数提供身份验证细节。如下所示：

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <docker>
                        <builderRegistry>
                            <username>user</username>
                            <password>secret</password>
                            <url>https://docker.example.com/v1/</url>
                            <email>user@example.com</email>
                        </builderRegistry>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

如果构建器或运行映像存储在支持令牌身份验证的私有 Docker registry 中，则可以使用 `docker.builderRegistry` 参数提供令牌值。如下所示：

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <docker>
                        <builderRegistry>
                            <token>9cbaf023786cd7...</token>
                        </builderRegistry>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Chapter 7. 使用 Maven 运行你的应用程序

该插件包括一个 run 目标,可用于从命令行启动您的应用程序,如以下示例所示:

```
$ mvn spring-boot:run
```

可以使用 arguments 参数指定应用程序参数,有关更多详细信息,请参见 使用应用程序参数.

默认情况下,应用程序是在 forked 进程中执行的,并且在命令行上设置属性不会影响应用程序. 如果需要指定一些 JVM 参数 (即用于调试目的) ,则可以使用 jvmArguments 参数 ,有关更多详细信息,请参见 调试应用程序. 还明确支持 系统属性 和 环境变量.

由于启用配置文件非常普遍,因此有一个专用的 profiles 属性,它提供了 -Dspring -boot.run.jvmArguments="-Dspring.profiles.active=dev" 的快捷方式,请参阅 指定激活 profiles.

尽管不建议这样做,但是可以通过禁用 fork 属性直接从 Maven JVM 执行应用程序. 这样做意味着将忽略 jvmArguments,systemPropertyVariables ,environmentVariables 和 agent 选项.

Spring Boot devtools 是一个模块,用于改善在使用 Spring Boot 应用程序时的开发时间体验. 要启用它,只需将以下依赖添加到您的项目中:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

当 devtools 运行时,它会在重新编译应用程序时检测到更改并自动刷新它. 这不仅适用于资源,而且适用于代码. 它还提供了 LiveReload 服务器 ,以便它可以在发生任何变化时自动触发浏览器刷新.

还可以将 Devtools 配置为仅在静态资源发生更改时刷新浏览器 (并忽略代码中的任何更改)

. 只需在项目中包括以下属性:

```
spring.devtools.remote.restart.enabled=false
```

在 devtools 之前,该插件默认情况下支持资源的热刷新,现在已禁用它,以支持上述解决方案.您可以随时通过配置项目来还原它:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <addResources>true</addResources>
            </configuration>
        </plugin>
    </plugins>
</build>
```

启用 addResources 时,在运行应用程序时,所有 src/main/resources 目录都将添加到应用程序类路径中,并且将删除在 target/classes 中发现的所有重复项.这样可以热刷新资源,这在开发 Web 应用程序时非常有用. 例如,您可以处理 HTML,CSS 或 JavaScript 文件,并且无需重新编译应用程序即可立即查看更改.
这也是允许您的前端开发人员进行工作而无需下载和安装 Java IDE 的一种有用方法.

使用此功能的缺点是在构建时无法进行资源过滤.

为了与 repackage 目标保持一致,run 目标以如下方式构建类路径:
将插件配置中排除的任何依赖也从类路径中排除. 有关更多详细信息,请参见the dedicated example.

有时在运行应用程序时包括测试依赖很有用. 例如
,如果要在使用存根类的测试模式下运行应用程序. 如果您希望这样做,可以将 useTestClasspath 参数设置为 true.

这仅在运行应用程序时适用: repackage 目标不会将测试依赖添加到生成的 JAR/WAR 中.

7.1. spring-boot:run

org.springframework.boot:spring-boot-maven-plugin:2.4.5

运行应用程序.

7.1.1. Required parameters

Name	Type	Default
classesDirectory	File	${project.build.outputDirectory}

7.1.2. Optional parameters

Name	Type	Default
addResources	boolean	false
agents	File[]	
arguments	String[]	
commandlineArguments	String	
directories	String[]	
environmentVariables	Map	
excludeGroupIds	String	
excludes	List	
folders	String[]	
fork	boolean	true
includes	List	
jvmArguments	String	
mainClass	String	
noverify	boolean	
optimizedLaunch	boolean	true

Name	Type	Default
profiles	String[]	
skip	boolean	false
systemPropertyVariables	Map	
useTestClasspath	Boolean	false
workingDirectory	File	

7.1.3. Parameter details

addResources

Add maven resources to the classpath directly, this allows live in-place editing of resources. Duplicate resources are removed from target/classes to prevent them to appear twice if ClassLoader.getResources() is called. Please consider adding spring-boot-devtools to your project instead as it provides this feature and many more.

Name	addResources
Type	boolean
Default value	false
User property	spring-boot.run.addResources
Since	1.0.0

agents

Path to agent jars. NOTE: a forked process is required to use this feature.

Name	agents
Type	java.io.File[]
Default value	
User property	spring-boot.run.agents
Since	2.2.0

arguments

Arguments that should be passed to the application.

Name	arguments
Type	java.lang.String[]
Default value	
User property	
Since	1.0.0

classesDirectory

Directory containing the classes and resource files that should be packaged into the archive.

Name	classesDirectory
Type	java.io.File

Default value	${project.build.outputDirectory}
User property	
Since	1.0.0

commandlineArguments

Arguments from the command line that should be passed to the application. Use spaces to separate multiple arguments and make sure to wrap multiple values between quotes. When specified, takes precedence over #arguments.

Name	commandlineArguments
Type	java.lang.String
Default value	
User property	spring-boot.run.arguments
Since	2.2.3

directories

Additional directories besides the classes directory that should be added to the classpath.

Name	directories
Type	java.lang.String[]

Default value	
User property	spring-boot.run.directories
Since	1.0.0

environmentVariables

List of Environment variables that should be associated with the forked process used to run the application. NOTE: a forked process is required to use this feature.

Name	environmentVariables
Type	java.util.Map
Default value	
User property	
Since	2.1.0

excludeGroupIds

Comma separated list of groupId names to exclude (exact match).

Name	excludeGroupIds
Type	java.lang.String
Default value	

User property	spring-boot.excludeGroupIds
Since	1.1.0

excludes

Collection of artifact definitions to exclude. The Exclude element defines mandatory groupId and artifactId properties and an optional classifier property.

Name	excludes
Type	java.util.List
Default value	
User property	spring-boot.excludes
Since	1.1.0

folders

Additional directories besides the classes directory that should be added to the classpath.

Name	folders
Type	java.lang.String[]
Default value	

User property	spring-boot.run.folders
Since	1.0.0

fork

Flag to indicate if the run processes should be forked. Disabling forking will disable some features such as an agent, custom JVM arguments, devtools or specifying the working directory to use.

Name	fork
Type	boolean
Default value	true
User property	spring-boot.run.fork
Since	1.2.0

includes

Collection of artifact definitions to include. The Include element defines mandatory groupId and artifactId properties and an optional mandatory groupId and artifactId properties and an optional classifier property.

Name	includes
Type	java.util.List
Default value	

User property	spring-boot.includes
Since	1.2.0

jvmArguments

JVM arguments that should be associated with the forked process used to run the application. On command line, make sure to wrap multiple values between quotes. NOTE: a forked process is required to use this feature.

Name	jvmArguments
Type	java.lang.String
Default value	
User property	spring-boot.run.jvmArguments
Since	1.1.0

mainClass

The name of the main class. If not specified the first compiled class found that contains a 'main' method will be used.

Name	mainClass
Type	java.lang.String
Default value	

User property	spring-boot.run.main-class
Since	1.0.0

noverify

Flag to say that the agent requires -noverify.

Name	noverify
Type	boolean
Default value	
User property	spring-boot.run.noverify
Since	1.0.0

optimizedLaunch

Whether the JVM's launch should be optimized.

Name	optimizedLaunch
Type	boolean
Default value	true
User property	spring-boot.run.optimizedLaunch
Since	2.2.0

profiles

The spring profiles to activate. Convenience shortcut of specifying the 'spring.profiles.active' argument. On command line use commas to separate multiple profiles.

Name	profiles
Type	java.lang.String[]
Default value	
User property	spring-boot.run.profiles
Since	1.3.0

skip

Skip the execution.

Name	skip
Type	boolean
Default value	false
User property	spring-boot.run.skip
Since	1.3.2

systemPropertyVariables

List of JVM system properties to pass to the process. NOTE: a forked process is required to use this feature.

Name	systemPropertyVariables
Type	java.util.Map
Default value	
User property	
Since	2.1.0

useTestClasspath

Flag to include the test classpath when running.

Name	useTestClasspath
Type	java.lang.Boolean
Default value	false
User property	spring-boot.run.useTestClasspath
Since	1.3.0

workingDirectory

Current working directory to use for the application. If not specified, basedir will be used. NOTE: a forked process is required to use this feature.

Name	workingDirectory
Type	java.io.File

Default value	
User property	spring-boot.run.workingDirectory
Since	1.5.0

7.2. 示例

7.2.1. 调试应用程序

默认情况下,run 目标在 forkd 进程中运行您的应用程序. 如果需要调试,则应添加必要的 JVM 参数以启用远程调试. 以下配置将挂起该进程,直到调试器在端口 5005 上加入为止:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <jvmArguments>
                        -Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
                    </jvmArguments>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

这些参数也可以在命令行上指定,请确保将其正确包装,即:

```
$ mvn spring-boot:run -Dspring-boot.run.jvmArguments="-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005"
```

7.2.2. 使用系统属性

可以使用 systemPropertyVariables 属性指定系统属性.以下示例将 property1 设置为 test 并将 property2 设置为 42:

```
<project>
    <build>
        <properties>
            <my.value>42</my.value>
        </properties>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <systemPropertyVariables>
                        <property1>test</property1>
                        <property2>${my.value}</property2>
                    </systemPropertyVariables>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

如果该值为空或未定义（例如 <my-property/>），则将系统属性设置为空的字符串. 因为 Maven 对 pom 中指定的值进行了 trim,因此无法通过此机制指定需要以空格开头或结尾的 System 属性: 考虑改用 jvmArguments.

任何 String 类型的 Maven 变量都可以作为系统属性传递. 任何传递其他 Maven 变量类型（例如 List 或 URL 变量）的尝试都将导致变量表达式按字面值传递（未评估）.

jvmArguments 参数优先于上述机制定义的系统属性. 在以下示例中,property1 的值被覆盖:

```
$ mvn spring-boot:run -Dspring-boot.run.jvmArguments="-Dproperty1=overridden"
```

7.2.3. 使用环境变量

可以使用 environmentVariables 属性指定环境变量.以下示例设置了 'ENV1', 'ENV2',

'ENV3', 'ENV4' env变量:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <environmentVariables>
                        <ENV1>5000</ENV1>
                        <ENV2>Some Text</ENV2>
                        <ENV3/>
                        <ENV4></ENV4>
                    </environmentVariables>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

如果该值为空或未定义（例如 <MY_ENV/>），则将 env 变量设置为空的字符串。因为 Maven 对 pom 中指定的值进行了 trim,因此无法指定需要以空格开头或结尾的 env 变量。

任何 String 类型的 Maven 变量都可以作为系统属性传递。任何传递其他 Maven 变量类型（例如 List 或 URL 变量）的尝试都将导致变量表达式按字面值传递（未评估）。

用这种方法定义的环境变量优先于现有值。

7.2.4. 使用应用程序参数

可以使用 arguments 属性指定应用程序参数.下面的示例设置两个参数: property1 和 property2=42:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <arguments>
                        <argument>property1</argument>
                        <argument>property2=${my.value}</argument>
                    </arguments>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

在命令行上,参数用空格分隔,与 `jvmArguments` 相同.如果参数包含空格,请使用引号 .在以下示例中,有两个参数可用: `property1` 和 `property2=Hello World`:

```
$ mvn spring-boot:run -Dspring-boot.run.arguments="property1 'property2=Hello
World'"
```

7.2.5. 指定激活的 Profiles

可以使用 `profiles` 参数指定.

下面的配置启用了 `foo` 和 `bar` 配置文件:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <profiles>
                        <profile>foo</profile>
                        <profile>bar</profile>
                    </profiles>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

也可以在命令行上指定要启用的配置文件,请确保用逗号分隔它们,如以下示例所示:

```
$ mvn spring-boot:run -Dspring-boot.run.profiles=foo,bar
```

Chapter 8. 在集成测试中运行

尽管您可以从测试（或测试套件）本身非常轻松地启动 Spring Boot 应用程序，但可能需要在构建本身中进行处理。为了确保围绕集成测试正确管理 Spring Boot 应用程序的生命周期，可以使用 start 和 stop 目标，如以下示例所示：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>pre-integration-test</id>
                    <goals>
                        <goal>start</goal>
                    </goals>
                </execution>
                <execution>
                    <id>post-integration-test</id>
                    <goals>
                        <goal>stop</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

现在，这样的设置可以使用 failsafe-plugin 按预期运行集成测试。

默认情况下，应用程序是在单独的进程中启动的，并且使用 JMX 与应用程序进行通信。如果需要配置 JMX 端口，请参见 the dedicated example.

您还可以配置更高级的设置，以在设置了特定属性时跳过集成测试，请参见 the dedicated example.

8.1. 在没有 Spring Boot 的 Parent POM 的情况下使用故障保护

Spring Boot's Parent POM, spring-boot-starter-parent, 将 Failsafe 的 <classesDirectory> 设置为 ${project.build.outputDirectory}.
如果没有此配置, 这将导致 Failsafe 使用已编译的类而不是重新包装的 jar, Failsafe 无法加载您的应用程序的类. 如果您不使用 parent POM, 则应以相同的方式配置故障保护:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <configuration>
        <classesDirectory>${project.build.outputDirectory}</classesDirectory>
    </configuration>
</plugin>
```

8.2. spring-boot:start

org.springframework.boot:spring-boot-maven-plugin:2.4.5

启动 spring 应用程序. 与 run 目标相反,这不会阻止并允许其他目标在应用程序上运行. 此目标通常用于集成测试方案中,在该方案中,应用程序在测试套件之前启动,然后在测试套件之后停止.

8.2.1. 必要参数

Name	Type	Default
classesDirectory	File	${project.build.outputDirectory}

8.2.2. 可选参数

Name	Type	Default
addResources	boolean	false
agents	File[]	

Name	Type	Default
arguments	String[]	
commandlineArguments	String	
directories	String[]	
environmentVariables	Map	
excludeGroupIds	String	
excludes	List	
folders	String[]	
fork	boolean	true
includes	List	
jmxName	String	
jmxPort	int	
jvmArguments	String	
mainClass	String	
maxAttempts	int	
noverify	boolean	
profiles	String[]	
skip	boolean	false
systemPropertyVariables	Map	
useTestClasspath	Boolean	false
wait	long	
workingDirectory	File	

8.2.3. Parameter details

`addResources`

将 Maven 资源直接添加到类路径,这允许实时就地编辑资源. 如果调用 `ClassLoader.getResources()`,则从 `target/classes` 中删除重复的资源

,以防止它们出现两次. 请考虑将 spring-boot-devtools 添加到您的项目中,因为它提供了此功能以及更多其他功能.

Name	addResources
Type	boolean
Default value	false
User property	spring-boot.run.addResources
Since	1.0.0

agents

代理 jars 的路径.注意: 要使用此功能,需要使用 forked process .

Name	agents
Type	java.io.File[]
Default value	
User property	spring-boot.run.agents
Since	2.2.0

arguments

传递给应用程序的参数.

Name	arguments
Type	java.lang.String[]

Default value	
User property	
Since	1.0.0

classesDirectory

包含应打包到 `archive` 文件中的类和资源文件的目录.

Name	classesDirectory
Type	java.io.File
Default value	${project.build.outputDirectory}
User property	
Since	1.0.0

commandlineArguments

命令行中应传递给应用程序的参数.使用空格分隔多个参数,并确保在引号之间包含多个值.指定后,优先于 `#arguments`.

Name	commandlineArguments
Type	java.lang.String
Default value	

User property	spring-boot.run.arguments
Since	2.2.3

directories

除了 classes 目录之外的其他目录,应添加到类路径中.

Name	directories
Type	java.lang.String[]
Default value	
User property	spring-boot.run.directories
Since	1.0.0

environmentVariables

用于运行应用程序的 forked process 相关联的环境变量列表.注意: 要使用此功能,需要使用 forked process.

Name	environmentVariables
Type	java.util.Map
Default value	
User property	
Since	2.1.0

excludeGroupIds

要排除的 groupId 名称列表（完全匹配）,使用逗号分隔.

Name	excludeGroupIds
Type	java.lang.String
Default value	
User property	spring-boot.excludeGroupIds
Since	1.1.0

excludes

要排除的组件的集合. Exclude 元素必需定义 groupId 和 artifactId 属性以及可选的 classifier 属性.

Name	excludes
Type	java.util.List
Default value	
User property	spring-boot.excludes
Since	1.1.0

folders

除了 classes 目录之外的其他目录,应添加到类路径中.

Name	folders

Type	java.lang.String[]
Default value	
User property	spring-boot.run.folders
Since	1.0.0

fork

是否使用 forked 的标志.如果禁用 forked 将禁用某些功能,例如代理,自定义 JVM 参数,devtools 或指定要使用的工作目录.

Name	fork
Type	boolean
Default value	true
User property	spring-boot.run.fork
Since	1.2.0

includes

要包含的组件的集合. Include 元素必需定义 groupId 和 artifactId 属性以及可选的 classifier 属性.

Collection of artifact definitions to include. The Include element defines mandatory groupId and artifactId properties and an optional mandatory groupId and artifactId properties and an optional classifier property.

Name	includes
Type	java.util.List
Default value	
User property	spring-boot.includes
Since	1.2.0

jmxName

自动部署的 MBean 的 JMX 名称,用于管理 Spring 应用程序的生命周期.

Name	jmxName
Type	java.lang.String
Default value	
User property	
Since	

jmxPort

如果应用程序是 forked 的,则用于暴露平台 MBeanServer 的端口.

Name	jmxPort
Type	int
Default value	

User property	
Since	

jvmArguments

用于运行应用程序的 forked 进程相关联的 JVM 参数.在命令行上 ,请确保在引号之间包含多个值.注意: 要使用此功能,需要使用 forked 进程.

Name	jvmArguments
Type	java.lang.String
Default value	
User property	spring-boot.run.jvmArguments
Since	1.1.0

mainClass

主类的名称.如果未指定,将使用找到的第一个包含 'main' 方法的类.

Name	mainClass
Type	java.lang.String
Default value	
User property	spring-boot.run.main-class
Since	1.0.0

maxAttempts

检查 spring 应用程序是否准备就绪的最大尝试次数. 结合 "wait" 参数
,这给出了一个全局超时值 (默认为 30 秒)

Name	maxAttempts
Type	int
Default value	
User property	
Since	

noverify

标记该代理是否需要 -noverify.

Name	noverify
Type	boolean
Default value	
User property	spring-boot.run.noverify
Since	1.0.0

profiles

spring profiles 激活. 指定 'spring.profiles.active' 参数的简洁方式.
在命令行上使用逗号分隔多个配置文件.

Name	profiles
Type	java.lang.String[]
Default value	
User property	spring-boot.run.profiles
Since	1.3.0

skip

跳过执行.

Name	skip
Type	boolean
Default value	false
User property	spring-boot.run.skip
Since	1.3.2

systemPropertyVariables

传递给进程的 JVM 系统属性列表.注意: 要使用此功能,需要使用 forked 进程.

Name	systemPropertyVariables
Type	java.util.Map
Default value	

User property	
Since	2.1.0

useTestClasspath

运行时是否包括测试类路径.

Name	useTestClasspath
Type	java.lang.Boolean
Default value	false
User property	spring-boot.run.useTestClasspath
Since	1.3.0

wait

每次尝试检查 `spring` 应用程序是否准备就绪之间要等待的毫秒数.

Name	wait
Type	long
Default value	
User property	
Since	

workingDirectory

当前工作目录.如果未指定,将使用 basedir.注意: 要使用此功能,需要使用 forked 进程.

Name	workingDirectory
Type	java.io.File
Default value	
User property	spring-boot.run.workingDirectory
Since	1.5.0

8.3. spring-boot:stop

org.springframework.boot:spring-boot-maven-plugin:2.4.5

停止已通过 "start" 目标启动的应用程序.通常在测试套件完成后调用.

8.3.1. 可选参数

Name	Type	Default
fork	Boolean	
jmxName	String	
jmxPort	int	
skip	boolean	false

8.3.2. 参数细节

fork

指示停止过程是否是 forked 的标志. 默认情况下,该值是从 MavenProject 继承的.
如果已设置,则必须与 StartMojo start 过程中使用的值匹配.

Name	fork
Type	java.lang.Boolean
Default value	
User property	spring-boot.stop.fork
Since	1.3.0

jmxName

自动部署的 MBean 的 JMX 名称,用于管理 Spring 应用程序的生命周期.

Name	jmxName
Type	java.lang.String
Default value	
User property	
Since	

jmxPort

如果应用程序是 forked 的,则用于暴露平台 MBeanServer 的端口.

Name	jmxPort
Type	int
Default value	

User property	
Since	

`skip`

跳过执行.

Name	`skip`
Type	`boolean`
Default value	`false`
User property	`spring-boot.stop.skip`
Since	`1.3.2`

8.4. 示例

8.4.1. 使用随机端口进行集成测试

Spring Boot 测试集成的一个不错的功能是它可以为 Web 应用程序分配一个空闲端口.
当使用插件的 `start` 目标时,Spring Boot 应用程序将单独启动
,因此很难将实际端口传递给集成测试本身.

下面的示例展示了如何使用 Build Helper Maven 插件 实现相同的功能:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>build-helper-maven-plugin</artifactId>
                <executions>
```

```
                    <execution>
                        <id>reserve-tomcat-port</id>
                        <goals>
                            <goal>reserve-network-port</goal>
                        </goals>
                        <phase>process-resources</phase>
                        <configuration>
                            <portNames>
                                <portName>tomcat.http.port</portName>
                            </portNames>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>pre-integration-test</id>
                        <goals>
                            <goal>start</goal>
                        </goals>
                        <configuration>
                            <arguments>
                                <argument>--
server.port=${tomcat.http.port}</argument>
                            </arguments>
                        </configuration>
                    </execution>
                    <execution>
                        <id>post-integration-test</id>
                        <goals>
                            <goal>stop</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <configuration>
                    <systemPropertyVariables>
                        <test.server.port>${tomcat.http.port}</test.server.port>
                    </systemPropertyVariables>
                </configuration>
            </plugin>
        </plugins>
```

```
    </build>
</project>
```

现在,您可以在任何集成测试中搜索 `test.server.port` 系统属性,以创建指向服务器的正确 URL.

8.4.2. 自定义 JMX 端口

`jmxPort` 属性允许自定义插件用于与 Spring Boot 应用程序通信的端口.

本示例显示了在已经使用 `9001` 的情况下如何自定义端口:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <jmxPort>9009</jmxPort>
                </configuration>
                <executions>
                    <execution>
                        <id>pre-integration-test</id>
                        <goals>
                            <goal>start</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>post-integration-test</id>
                        <goals>
                            <goal>stop</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

如果需要配置 JMX 端口,请确保在上述全局配置中进行配置,以便两个目标都可以共享它.

8.4.3. 跳过集成测试

`skip` 属性允许完全跳过 Spring Boot maven 插件的执行.

此示例说明如何跳过带有命令行属性的集成测试,并仍然确保 `repackage` 目标运行:

```
<project>
    <properties>
        <skip.it>false</skip.it>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>pre-integration-test</id>
                        <goals>
                            <goal>start</goal>
                        </goals>
                        <configuration>
                            <skip>${skip.it}</skip>
                        </configuration>
                    </execution>
                    <execution>
                        <id>post-integration-test</id>
                        <goals>
                            <goal>stop</goal>
                        </goals>
                        <configuration>
                            <skip>${skip.it}</skip>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <configuration>
                    <skip>${skip.it}</skip>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

默认情况下,将运行集成测试,但是此设置使您可以在命令行上轻松禁用它们,如下所示:

```
$ mvn verify -Dskip.it=true
```

Chapter 9. 集成 Actuator

如果存在 META-INF/build-info.properties 文件,Spring Boot Actuator 将显示与构建相关的信息. build-info 目标将生 成带有项目坐标和构建时间的文件. 它还允许您添加任意数量的其他属性,如以下示例所示:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>build-info</goal>
                        </goals>
                        <configuration>
                            <additionalProperties>
                                <encoding.source>UTF-8</encoding.source>
                                <encoding.reporting>UTF-8</encoding.reporting>

<java.source>${maven.compiler.source}</java.source>

<java.target>${maven.compiler.target}</java.target>
                                </additionalProperties>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

此配置将在预期位置生成带有 4 个 additional keys 的 build-info.properties.

maven.compiler.source 和 maven.compiler.target 应该是项目中可用的常规属性. 它们将按照您的期望进行插值..

9.1. spring-boot:build-info

org.springframework.boot:spring-boot-maven-plugin:2.4.5

根据 MavenProject 生成 build-info.properties 文件.

9.1.1. Optional parameters

Name	Type	Default
additionalProperties	Map	
outputFile	File	${project.build.outputDirectory}/META-INF/build-info.properties
time	String	

9.1.2. Parameter details

additionalProperties

build-info.properties 文件中的其他属性. 每个条目在 build-info.properties 以 build. 为前缀.

Name	additionalProperties
Type	java.util.Map
Default value	
User property	
Since	

outputFile

生成 build-info.properties 文件的位置.

Name	outputFile
Type	java.io.File
Default value	${project.build.outputDirectory}/META-INF/build-info.properties
User property	
Since	

time

build.time 属性的值, 格式为 Instant#parse(CharSequence). 默认为 session.request.startTime. 要完全禁用 build.time 属性, 请使用 'off'.

Name	time
Type	java.lang.String
Default value	
User property	
Since	2.2.0

Chapter 10. Help 信息

help 目标是一个标准目标,显示有关插件功能的信息.

10.1. spring-boot:help

org.springframework.boot:spring-boot-maven-plugin:2.4.5

在 spring-boot-maven-plugin 上显示帮助信息. 使用 mvn spring-boot:help -Ddetail=true -Dgoal=<goal-name> 以显示参数详细信息.

10.1.1. Optional parameters

Name	Type	Default
detail	boolean	false
goal	String	
indentSize	int	2
lineLength	int	80

10.1.2. Parameter details

detail

如果为 true, 则显示每个 goal 的所有可设置属性

Name	detail
Type	boolean
Default value	false
User property	detail

Since	

goal

要显示帮助的 goal 的名称. 如果未指定, 将显示所有 goals.

Name	goal
Type	java.lang.String
Default value	
User property	goal
Since	

indentSize

缩进的空格数, 应为正数.

Name	indentSize
Type	int
Default value	2
User property	indentSize
Since	

lineLength

显示行的最大长度, 应为正数.

Name	lineLength
Type	int
Default value	80
User property	lineLength
Since	