Spring Boot Gradle Plugin Reference Guide

Andy Wilkinson

Table of Contents

Chapter 1. 简介

Spring Boot Gradle 插件在 Gradle 中提供 Spring Boot 支持，允许您打包可执行的 jar 或 war 文件，运行 Spring Boot 应用程序，并使用 spring-boot-dependencies 提供的依赖管理。Spring Boot 的 Gradle 插件需要 6（6.3 更高版本）（支持 Gradle 5.6 版本，但在未来会移除）的 Gradle。当使用 Gradle 6.7 或更高版本时，支持 Gradle 的 configuration cache。

除了此用户指南，还提供了 API 文档。

Chapter 2. 入门

想要开始插件，首先需要把下面的代码加入你的程序中

使用这个插件可以让减少项目的改动。而且，会检测其他插件的使用或者更新。比如，当应用了 java 插件，会自动在构建时打包成可执行的 jar。一个经典的 Spring Boot 项目至少会使用 groovy，java 或 org.jetbrains.kotlin.jvm 和 io.spring.dependency-management 插件

Groovy

```
apply plugin: 'java'
apply plugin: 'io.spring.dependency-management'
```

Kotlin

```
plugins {
    java
    id("org.springframework.boot") version "2.4.5"
}

apply(plugin = "io.spring.dependency-management")
```

在使用其他插件的时候，Spring Boot 插件会有怎样的行为？要了解有关在应用其他插件时 Spring Boot 插件的行为的更多信息，请参见 reacting to other plugins.

Chapter 3. 依赖管理

要管理 Spring Boot 应用程序中的依赖关系,可以应用 io.spring.dependency-management 插件,或者,如果您使用的是 Gradle 6 或更高版本,请使用 Gradle 的本地 bom 支持. 前者的主要优点是它提供了基于属性的托管版本的的自定义
,而使用后者则会加快构建速度.

3.1. Managing Dependencies with the Dependency Management Plugin

当使用了 io.spring.dependency-management 插件, Spring Boot
的插件会自动地从你使用的 Spring Boot 版本里导入 import the spring-boot-dependencies bom. Maven用户使用起来感觉区别不大. 比如,
允许你在声明依赖的时候忽略掉版本号, 使用这项功能, 只需要正常的声明依赖,
不用写版本号就可以了.

Groovy

```
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-web')
    implementation('org.springframework.boot:spring-boot-starter-data-jpa')
}
```

Kotlin

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
    implementation("org.springframework.boot:spring-boot-starter-data-jpa")
}
```

3.2. 自定义管理版本号

当应用了依赖管理插件, spring-boot-dependencies bom 会被自动导入,
使用属性控制它管理的依赖的版本号. 点击 bom 查看完整的属性列表.

如果要自定义管理版本, 设置它的扩展属性. 比如, 自定义被管理的 SLF4J 的版本号, 设置 slf4j.version 属性:

Groovy

```
ext['slf4j.version'] = '1.7.20'
```

Kotlin

```
extra["slf4j.version"] = "1.7.20"
```

每个 Spring Boot 版本都与制定的第三方依赖设计和测试好了.
覆盖版本可能会造成兼容问题, 所以自定义的时候小心一点

3.3. 单独使用 Spring Boot 的依赖管理

我们不用将 Spring Boot 的插件应用于项目, 也可以在项目中使用 Spring Boot 的依赖管理. SpringBootPlugin 类提供一个 BOM_COORDINATES 常量, 可用于导入 Bom, 而不必知道其 artifact ID, group ID 或 version 号.

首先, 将项目配置为依赖于 Spring Boot 插件, 但不要应用它:

Spring Boot
插件对依赖管理插件的依赖意味着您可以使用依赖管理插件而不必声明对它的依赖.
这也意味着您将自动使用与 Spring Boot 使用的版本相同的依赖管理插件.

应用依赖管理插件, 然后将其配置为导入 Spring Boot 的 Bom:

Groovy

```
apply plugin: 'io.spring.dependency-management'

dependencyManagement {
    imports {
        mavenBom
org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
    }
}
```

Kotlin

```
apply(plugin = "io.spring.dependency-management")

the<DependencyManagementExtension>().apply {
    imports {

mavenBom(org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
)
    }
}
```

上面的Kotlin代码有点尴尬. 那是因为我们使用命令式方式来应用依赖管理插件.

我们可以通过应用根父项目中的插件, 或者像在使用 Spring Boot 插件一样使用 plugins 块, 来减少代码的尴尬. 该方法的缺点是它迫使我们指定依赖管理插件的版本:

```
plugins {
    java
    id("org.springframework.boot") version "2.4.5" apply false
    id("io.spring.dependency-management") version "{dependency-management-
plugin-version}"
}

dependencyManagement {
    imports {

mavenBom(org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
)
    }
}
```

3.4. 更多

要了解有关依赖管理插件功能的更多信息, 请参阅其 documentation.

3.5. 使用 Gradle 的 Bom 支持依赖管理

Gradle 通过将 Bom 声明为 platform 或 enforcedPlatform 依赖
,可用于管理项目的版本. platform 依赖将 bom 中的版本视为推荐版本
,这可能会导致项目中的其他版本或者依赖关系中的其他版本与此 bom 声明的版本不同.

enforcedPlatform 依赖将 bom 中的版本视为要求
,它们将覆盖依赖关系中找到的任何其他版本.

SpringBootPlugin 类提供 BOM_COORDINATES 常量,可用于声明对 Spring Boot 的 bom 的依赖关系,而不必知道其 group ID, artifact ID 或 version,如以下示例所示:

Groovy

```
dependencies {
    implementation
platform(org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
)
}
```

Kotlin

```
dependencies {

implementation(platform(org.springframework.boot.gradle.plugin.SpringBootPlugin.
BOM_COORDINATES))
}
```

platform 或 enforced platform 将只在声明它的配置中具有约束性
,或者在声明它的配置的扩展的地址.因此,可能需要在多个配置中声明相同的依赖.

3.5.1. 自定义版本管理

使用 Gradle 的 bom 支持时,您不能使用 spring-boot-dependencies
中的属性来控制它管理的依赖的版本. 相反,您必须使用 Gradle 提供的一种机制.
一种这样的机制是解决策略. SLF4J 的模块全部在 org.slf4j 组中
,因此可以通过将该组中的每个依赖配置为使用特定版本来控制其版本,如以下示例所示:

Groovy

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.slf4j') {
            details.useVersion '1.7.20'
        }
    }
}
```

Kotlin

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.group == "org.slf4j") {
            useVersion("1.7.20")
        }
    }
}
```

每个 Spring Boot 版本都是针对一组特定的第三方依赖进行设计和测试的.覆盖版本可能会引起兼容性问题,因此应格外小心.

Chapter 4. 打包可执行文件

插件可以创建可执行文件(jar 文件或者 war 文件), 里面包含所有应用的依赖并且可以使用 java -jar 执行.

4.1. 打包可执行 jars

可以使用 bootJar 任务构建可执行的 jars. 如果应用了 java 插件,
这个任务会自动的创建, 并且是 BootJar 的实例. assemble 任务会自动配置依赖于 bootJar 任务, 所以运行 assemble (or build) 也会运行 bootJar 任务.

4.2. 打包可执行 wars

可以使用 bootWar 任务构建可执行的 wars.如果应用了 war 插件, 这个任务会自动的创建, 并且是 BootWar 的实例. assemble 任务会自动配置依赖于 bootWar 任务, 所以运行 assemble (or build) 也会运行 bootWar 任务.

4.2.1. 打包可执行和可部署的 wars

可以打包 war 文件, 以便可以使用 java -jar 执行该文件并将其部署到外部容器中. 为此, 应将嵌入式 Servlet 容器依赖添加到提供的 Runtime 配置中, 例如:

Groovy

```
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-web')
    providedRuntime('org.springframework.boot:spring-boot-starter-tomcat')
}
```

Kotlin

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
    providedRuntime("org.springframework.boot:spring-boot-starter-tomcat")
}
```

这样可以确保在打包后的 war 文件中的 WEB-INF/lib-provided
提供的目录中不会与外部容器自己的类冲突.

providedRuntime 优先于 Gradle 的 compileOnly 配置，
因为除其他限制外，compileOnly 依赖不在测试类路径上，
因此任何基于 Web 的集成测试都将失败.

4.3. 打包可执行和普通文件

默认的，当 bootJar 或者 bootWar 任务配置了，jar 或者 war 任务会被禁用掉.
但是可以同时构建一个可执行和普通的文件通过设置 jar 或者 war 任务可用

Groovy

```
jar {
    enabled = true
}
```

Kotlin

```
tasks.getByName<Jar>("jar") {
    enabled = true
}
```

为了避免可执行文件和普通文件生成在同一个目录，其中一个应该使用不同的位置.
一种方法就是配置一个 classifier:

Groovy

```
bootJar {
    classifier = 'boot'
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    classifier = "boot"
}
```

4.4. 配置可执行文件打包

BootJar 和 BootWar

BootJar 和 BootWar 任务分别是 Gradle 的 jar 或者 war 任务的子类. 所以, 所有的在打包 jar 或者 war 时用的标准配置都对可执行的 jar 或者 war 可用. 当然也有对可执行 jars 或者 wars 的特殊配置.

4.4.1. 配置主类

默认的, 可执行文件的主类会自动的在任务的类路径目录里查找一个带有 public static void main(String[]) 的方法. 主类也可以显示的使用 mainClass 设置:

Groovy

```
bootJar {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    mainClass.set("com.example.ExampleApplication")
}
```

或者, 可以使用 Spring Boot DSL 的 mainClass 属性在项目范围内配置主类名称:

Groovy

```
springBoot {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
springBoot {
    mainClass.set("com.example.ExampleApplication")
}
```

另外, 如果应用了 application plugin 可以这么设置 mainClass 属性:

Groovy

```
application {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
application {
    mainClass.set("com.example.ExampleApplication")
}
```

最后, `Start-Class` 属性可以在任务的 `manifest` 里配置:

Groovy

```
bootJar {
    manifest {
        attributes 'Start-Class': 'com.example.ExampleApplication'
    }
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    manifest {
        attributes("Start-Class" to "com.example.ExampleApplication")
    }
}
```

如果主类是用 `Kotlin` 编写的,则应使用生成的 `Java` 类的名称. 默认情况下,这是添加了 `Kt` 后缀的 `Kotlin` 类的名称. 例如 ,`ExampleApplication` 变为`ExampleApplicationKt`. 如果使用 `@JvmName` 定义了另一个名称,则应使用该名称.

4.4.2. Including Development-only Dependencies

默认情况下,`developmentOnly` 配置中声明的所有依赖将从可执行 `jar` 或 `war` 中排除.

如果要在归档中包括在 `developmentOnly` 配置中声明的依赖

,请配置其任务的类路径以包括该配置,如 `bootWar` 任务的以下示例所示:

Groovy

```
bootWar {
    classpath configurations.developmentOnly
}
```

Kotlin

```
tasks.getByName<BootWar>("bootWar") {
    classpath(configurations["developmentOnly"])
}
```

4.4.3. 配置需要移除的库

大多数库可以直接嵌套进可执行的文件里, 但是某些库可能有问题. 比如,JRuby 引入了它自己的内嵌 jar 支持, 比如 `jruby-complete.jar` 总是直接在文件系统上可用.

处理这个问题库, 可执行文件可以配置在运行(run)的时候移除指定的嵌套 jars 到临时目录. 库可以使用 Ant-style 模式匹配源 jar 文件的绝对路径移除需要的包:

Groovy

```
bootJar {
    requiresUnpack '**/jruby-complete-*.jar'
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    requiresUnpack("**/jruby-complete-*.jar")
}
```

为了更好地控制, 也可以使用 closure. 该 closure 传递了一个 `FileTreeElement`, 并且应返回一个 `boolean`, 指示是否需要打包.

4.4.4. 让文件完全可执行

Spring Boot 对完全可执行文件提供支持. 通过已知的如何启动应用的预 shell 脚步来制造完全可执行. 在类 Unix 平台上,
这个启动脚本运行文件可以像任何可执行文件直接运行或者作为服务安装.

当前,某些工具不接受此格式,因此您可能无法始终使用此技术. 例如,jar -xf 可能在无提示的情况下无法提取出已成为完全可执行的 jar 或 war. 建议仅在打算直接执行此选项时才启用此选项,而不是使用 java -jar 运行它,将其部署到 servlet 容器或将其包含在 OCI 镜像中.

启用这个特性, launchScript 必须启用:

Groovy

```
bootJar {
    launchScript()
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    launchScript()
}
```

这将添加 Spring Boot 的默认脚本到文件里.
默认的启动脚本包含若干个属性并且设置了合适的默认值, 当然也可以通过 properties 属性自定义默认值:

Groovy

```
bootJar {
    launchScript {
        properties 'logFilename': 'example-app.log'
    }
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    launchScript {
        properties(mapOf("logFilename" to "example-app.log"))
    }
}
```

如果启动脚本没有你想要的，`script` 属性可以提供一个自定义的启动脚本：

Groovy

```
bootJar {
    launchScript {
        script = file('src/custom.script')
    }
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    launchScript {
        script = file("src/custom.script")
    }
}
```

4.4.5. 使用 PropertiesLauncher

要想使用 `PropertiesLauncher` 启动可执行 `jar` 或者 `war`，配置任务的 `manifest` 来设置 `Main-Class` 属性：

Groovy

```
bootWar {
    manifest {
        attributes 'Main-Class':
'org.springframework.boot.loader.PropertiesLauncher'
    }
}
```

Kotlin

```
tasks.getByName<BootWar>("bootWar") {
    manifest {
        attributes("Main-Class" to
"org.springframework.boot.loader.PropertiesLauncher")
    }
}
```

4.4.6. 打包分层的 Jars

默认情况下,bootJar 任务会构建一个 archive 文件,其中包含应用程序的类和依赖关系,分别位于 BOOT-INF/classes 和 BOOT-INF/lib 中. 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. 要使用此功能,必须启用分层功能:

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.

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.

要禁用此功能, 您可以通过以下方式执行此操作:

Groovy

```
bootJar {
    layered {
        enabled = false
    }
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    layered {
        isEnabled = false
    }
}
```

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:

Groovy

```
bootJar {
    layered {
        includeLayerTools = false
    }
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    layered {
        isIncludeLayerTools = false
    }
}
```

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.

这可以通过使用配置来完成,该配置描述了如何将 `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:

Groovy

```
bootJar {
    layered {
        application {
            intoLayer("spring-boot-loader") {
                include "org/springframework/boot/loader/**"
            }
            intoLayer("application")
        }
        dependencies {
            intoLayer("application") {
                includeProjectDependencies()
            }
            intoLayer("snapshot-dependencies") {
                include "*:*:*SNAPSHOT"
            }
            intoLayer("dependencies")
        }
        layerOrder = ["dependencies", "spring-boot-loader", "snapshot-
dependencies", "application"]
    }
}
```

Kotlin

```
tasks.getByName<BootJar>("bootJar") {
    layered {
        application {
            intoLayer("spring-boot-loader") {
                include("org/springframework/boot/loader/**")
            }
            intoLayer("application")
        }
        dependencies {
            intoLayer("snapshot-dependencies") {
                include("*:*:*SNAPSHOT")
            }
            intoLayer("dependencies")
        }
        layerOrder = listOf("dependencies", "spring-boot-loader", "snapshot-
dependencies", "application")
    }
}
```

`layered` DSL 包含三部分：

- `<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 `intoLayer` 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.

`intoLayer` 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 `includeProjectDependencies()` 和 `excludeProjectDependencies()` 元素, 可以用来包含或排除项目依赖.

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 `intoLayer` 将声明所有的项目依赖,
第二个 `intoLayer` 将声明上又有 `snapshot-dependencies` 层的 SNAPSHOT 依赖.
最后的 `intoLayer` 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.

`intoLayer` 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 `intoLayer` 块引用的所有层.

Chapter 5. 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 bootBuildImage 任务来构建镜像.

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.

应用 Java 插件时,该任务会自动创建,并且是 BootBuildImage 的实例.

bootBuildImage 任务不支持 打包成 war.

bootBuildImage 任务不能与包含启动脚本的 fully executable Spring Boot archive 一起使用.在构建旨在与 bootBuildImage 一起使用的 jar 文件时,请在 bootJar 任务中禁用启动脚本配置.

5.1. Docker Daemon

bootBuildImage 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 bootBuildImage 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)

Environment variable	Description
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:

Property	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.

5.2. 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:

Property	Description
username	Username for the Docker image registry user. 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.

5.3. 自定义镜像

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.

任务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:

Property	Command-line option	Description	Default value
builder	--builder	Name of the Builder image to use.	paketobuildpacks/builder:base

Property	Command-line option	Description	Default value
runImage	--runImage	Name of the run image to use.	No default value, indicating the run image specified in Builder metadata should be used.
imageName	--imageName	Image name for the generated image.	docker.io/library/${project.name}:${project.version}
pullPolicy	--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.	ALWAYS
environment		Environment variables that should be passed to the builder.	
cleanCache	--cleanCache	Whether to clean the cache before building.	false
verboseLogging		Enables verbose logging of builder operations.	false

Property	Command-line option	Description	Default value
publish	--publishImage	Whether to publish the generated image to a Docker registry.	false

该插件使用 JavaPlugin 的 targetCompatibility 属性检测目标项目的 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.

5.4. 示例

5.4.1. Custom Image Builder and Run Image

如果需要自定义用于创建镜像的构建器或用于启动生成的镜像的运行镜像,请配置任务,如以下示例所示:

Groovy

```
bootBuildImage {
    builder = "mine/java-cnb-builder"
    runImage = "mine/java-cnb-run"
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    builder = "mine/java-cnb-builder"
    runImage = "mine/java-cnb-run"
}
```

此配置将使用名称为 mine/java-cnb-builder 的构建器镜像和最新的标签,以及名为 mine/java-cnb-run 的运行镜像和最新的标签.

生成器和运行镜像也可以在命令行上指定,如本示例所示:

```
$ gradle bootBuildImage --builder=mine/java-cnb-builder --runImage=mine/java-cnb
-run
```

5.4.2. Builder Configuration

如果构建器公开了配置选项,则可以使用环境属性进行设置.

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

Groovy

```
bootBuildImage {
    environment = ["BP_JVM_VERSION" : "8.*"]
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    environment = mapOf("BP_JVM_VERSION" to "8.*")
}
```

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 和/或 HTTP_PROXY 环境变量来实现,如以下示例所示:

Groovy

```
bootBuildImage {
    environment = [
            "HTTP_PROXY" : "http://proxy.example.com",
            "HTTPS_PROXY": "https://proxy.example.com"
    ]
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    environment = mapOf("HTTP_PROXY" to "http://proxy.example.com",
                        "HTTPS_PROXY" to "https://proxy.example.com")
}
```

5.4.3. 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:

Groovy

```
bootBuildImage {
    environment = [
            "BPE_DELIM_JAVA_TOOL_OPTIONS" : " ",
            "BPE_APPEND_JAVA_TOOL_OPTIONS" : "-XX:+HeapDumpOnOutOfMemoryError"
    ]
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    environment = mapOf(
        "BPE_DELIM_JAVA_TOOL_OPTIONS" to " ",
        "BPE_APPEND_JAVA_TOOL_OPTIONS" to "-XX:+HeapDumpOnOutOfMemoryError"
    )
}
```

5.4.4. 自定义镜像名称

默认情况下,镜像名称是根据项目的 name 和 version 来推断的,例如 docker.io/library/${project.name}:${project.version}.

您可以通过设置任务属性来控制名称,如以下示例所示:

Groovy

```
bootBuildImage {
    imageName = "example.com/library/${project.name}"
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    imageName = "example.com/library/${project.name}"
}
```

请注意,此配置未提供明确的标记,因此使用了最新的标记. 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:

```
$ gradle bootBuildImage --imageName=example.com/library/my-app:v1
```

5.4.5. 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.

Groovy

```
bootBuildImage {
    imageName = "docker.example.com/library/${project.name}"
    publish = true
    docker {
        publishRegistry {
            username = "user"
            password = "secret"
            url = "https://docker.example.com/v1/"
            email = "user@example.com"
        }
    }
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    imageName = "docker.example.com/library/${project.name}"
    isPublish = true
    docker {
        publishRegistry {
            username = "user"
            password = "secret"
            url = "https://docker.example.com/v1/"
            email = "user@example.com"
        }
    }
}
```

也可以在命令行上指定 `publish` 选项，如下例所示：

```
$ gradle bootBuildImage --imageName=docker.example.com/library/my-app:v1
--publishImage
```

5.4.6. 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：

Groovy

```
bootBuildImage {
    docker {
        host = "tcp://192.168.99.100:2376"
        tlsVerify = true
        certPath = "/home/users/.minikube/certs"
    }
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    docker {
        host = "tcp://192.168.99.100:2376"
        isTlsVerify = true
        certPath = "/home/users/.minikube/certs"
    }
}
```

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：

Groovy

```
bootBuildImage {
    docker {
        builderRegistry {
            username = "user"
            password = "secret"
            url = "https://docker.example.com/v1/"
            email = "user@example.com"
        }
    }
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    docker {
        builderRegistry {
            username = "user"
            password = "secret"
            url = "https://docker.example.com/v1/"
            email = "user@example.com"
        }
    }
}
```

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：

Groovy

```
bootBuildImage {
    docker {
        builderRegistry {
            token = "9cbaf023786cd7..."
        }
    }
}
```

Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    docker {
        builderRegistry {
            token = "9cbaf023786cd7..."
        }
    }
}
```

Chapter 6. 发布应用

6.1. 使用 maven-publish 插件发布

使用 MavenPublication 的 artifact 方法发布你的 Spring Boot jar 或者 war. 将任务传给到你想要发布的 artifact 方法上. 比如, 通过默认的 bootJar 任务发布 artifact:

Groovy

```
publishing {
    publications {
        bootJava(MavenPublication) {
            artifact bootJar
        }
    }
    repositories {
        maven {
            url 'https://repo.example.com'
        }
    }
}
```

Kotlin

```
publishing {
    publications {
        create<MavenPublication>("bootJava") {
            artifact(tasks.getByName("bootJar"))
        }
    }
    repositories {
        maven {
            url = uri("https://repo.example.com")
        }
    }
}
```

6.2. 使用 maven 插件发布应用

应用了 maven plugin ,将自动创建名为 uploadBootArchives 的 bootArchives

配置的 Upload 任务. 默认的, bootArchives 配置包含了 bootJar 或者 bootWar 任务产生文件. uploadBootArchives 任务可以这么配置来发布到 Maven repository 里:

Groovy

```
uploadBootArchives {
    repositories {
        mavenDeployer {
            repository url: 'https://repo.example.com'
        }
    }
}
```

Kotlin

```
tasks.getByName<Upload>("uploadBootArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            "repository"("url" to "https://repo.example.com")
        }
    }
}
```

6.3. 使用 application 插件

当使用了 application plugin , 将创建一个名为 boot 的 distribution. 这个 distribution 包含了通过 bootJar 或者 bootWar 任务生成的文件和在 Unix 和 Windows 上启动的脚本. 可以分别通过 bootDistZip 和 bootDistTar 任务来构建 Zip 和 tar distributions. 要使用 application 插件,
必须使用应用程序主类的名称配置其 mainClassName 属性.

Chapter 7. 使用 Gradle 运行程序

使用 bootRun 任务来运行程序，不需要产生文件：

```
$ ./gradlew bootRun
```

bootRun 任务是 JavaExec 子类的 BootRun 的实例. 所以，所有的 usual configuration options 在 Gradle 里执行 Java 处理都可用.
任务会使用主源文件的运行期类路径自动配置.

默认的，通过查找任务的类路径下的 public static void main(String[]) 会自动配置.

主类也可以显示的配置：

Groovy

```
bootRun {
    main = 'com.example.ExampleApplication'
}
```

Kotlin

```
tasks.getByName<BootRun>("bootRun") {
    main = "com.example.ExampleApplication"
}
```

或者，可以使用 Spring Boot DSL 的 mainClass 属性在项目范围内配置主类名称：

Groovy

```
springBoot {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
springBoot {
    mainClass.set("com.example.ExampleApplication")
}
```

默认情况下，bootRun 将配置 JVM 以优化其启动，以在开发过程中加快启动速度。可以通过使用 optimizedLaunch 属性来禁用此行为，如以下示例所示：

Groovy

```
bootRun {
    optimizedLaunch = false
}
```

Kotlin

```
tasks.getByName<BootRun>("bootRun") {
    isOptimizedLaunch = false
}
```

另外，如果使用了 application plugin，mainClass 项目属性可以被使用：

Groovy

```
application {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
application {
    mainClass.set("com.example.ExampleApplication")
}
```

7.1. 将参数传递给您的应用程序

与所有 JavaExec 任务一样，在使用 Gradle 4.9 或更高版本时，可以使用 --args='<arguments>' 从命令行将参数传递到 bootRun 中。例如，要使用名为 dev 环境运行您的应用程序，可以使用以下命令：

```
$ ./gradlew bootRun --args='--spring.profiles.active=dev'
```

有关更多详细信息，请参见 JavaExec.setArgsString 的 javadoc

7.2. Passing System properties to your application

由于 bootRun 是标准的 JavaExec 任务
,因此可以通过在构建脚本中指定系统属性来将它们传递给应用程序的 JVM.
要使系统属性的值可配置,请使用 project property 设置其值. 若要使项目属性为可选
,请使用 findProperty . 这样做还允许使用 ?: 运算符提供默认值,如以下示例所示:

Groovy

```
bootRun {
    systemProperty 'com.example.property', findProperty('example') ?: 'default'
}
```

Kotlin

```
tasks.getByName<BootRun>("bootRun") {
    systemProperty("com.example.property", findProperty("example") ?: "default")
}
```

前面的示例将 com.example.property 系统属性设置为 example 属性的值. 如果未设置
example 属性,则系统属性的值为默认值.

Gradle 允许以多种方式设置项目属性,可以使用 -P
标志将这些值参数化并作为命令行上的属性传递.

```
$ ./gradlew bootRun -Pexample=custom
```

前面的示例将 example 属性的值设置为 custom.然后,bootRun 将使用它作为
com.example.property 系统属性的值.

7.3. 重加载资源

如果项目里加入了 devtools, 它会自动的监控你的程序变化. 另外, 如果你配置了 bootRun
比如静态资源, 应用程序的静态资源会从源位置被加载:

Groovy

```
bootRun {
    sourceResources sourceSets.main
}
```

Kotlin

```
tasks.getByName<BootRun>("bootRun") {
    sourceResources(sourceSets["main"])
}
```

这在开发时很有用.

Chapter 8. 整合 Actuator

8.1. 生成构建信息

Spring Boot Actuator 的 `info` endpoint 自动发布 `META-INF/build-info.properties` 文件里的信息. `BuildInfo` 任务可以用来生成这个文件.
最简单的方式是通过插件的 DSL 使用任务:

Groovy

```
springBoot {
    buildInfo()
}
```

Kotlin

```
springBoot {
    buildInfo()
}
```

这会配置一个 bootBuildInfo 的任务 `BuildInfo`, 并且如果他存在, 会让 Java 插件的 `classes` 任务基于它. 任务的目标目录将会是 `META-INF` 在主源设置的资源的输出目录 (通常是 `build/resources/main`).

默认的, 生成的信息是从项目里扩展 出来的:

Property	Default value
`build.artifact`	`bootJar` 或者 `bootWar` 任务的 base name, 如果没有就是 `unspecified`
`build.group`	The group of the project
`build.name`	The name of the project
`build.version`	The version of the project
`build.time`	The time at which the project is being built

可以使用 DSL 自定义属性:

Groovy

```
springBoot {
    buildInfo {
        properties {
            artifact = 'example-app'
            version = '1.2.3'
            group = 'com.example'
            name = 'Example application'
        }
    }
}
```

Kotlin

```
springBoot {
    buildInfo {
        properties {
            artifact = "example-app"
            version = "1.2.3"
            group = "com.example"
            name = "Example application"
        }
    }
}
```

`build.time` 的默认值为构建项目的瞬间. 这样做的副作用是该任务永远不会是最新的. 结果, 构建将花费更长的时间, 因为必须执行更多的任务, 包括项目的测试.
另一个副作用是任务的输出将始终更改, 因此构建将不会真正可重复.
如果您对构建性能或可重复性的重视程度高于 `build.time` 属性的准确性, 则将 `time` 设置为 `null` 或固定值.

可以添加额外的构建信息:

Groovy

```
springBoot {
    buildInfo {
        properties {
            additional = [
                'a': 'alpha',
                'b': 'bravo'
            ]
        }
    }
}
```

Kotlin

```
springBoot {
    buildInfo {
        properties {
            additional = mapOf(
                "a" to "alpha",
                "b" to "bravo"
            )
        }
    }
}
```

Chapter 9. 其他插件行为

当应用另一个插件时，Spring Boot
插件会通过对项目配置进行各种更改来做出反应。本节介绍这些更改.

9.1. Java 插件行为

当项目里使用了 java plugin,Spring Boot 插件会:

1. C创建一个名为 bootJar 的 BootJar 任务，它会为项目创建一个可执行的 fat jar.jar会包含所有 main source set 下运行时的类路径; 类会被打包进 BOOT-INF/classes 里，jars 会被打包进 BOOT-INF/lib 里.
2. 配置一个依赖于 bootJar 任务的 assemble 任务.
3. 禁止 jar 任务
4. 创建一个名为 bootBuildImage 的 BootBuildImage 任务,该任务将使用 buildpack 创建一个 OCI 镜像..
5. 创建一个名为 bootRun 的 BootRun 任务，用来运行你的程序.
6. 创建一个名为 bootArchives 的配置包含通过 bootJar 产生的 artifact.
7. 创建一个名为 developmentOnly 的配置,为仅在开发时需要的依赖（例如 Spring Boot 的 Devtools）,不应将其打包在可执行的 jar 和 wars 中.
8. 无需配置，所有的 JavaCompile 任务都使用 UTF-8.
9. 无需配置，所有的 JavaCompile 任务都使用 -parameters 编译器参数.

9.2. Kotlin 插件行为

当项目里使用了 Kotlin’s Gradle plugin ,Spring Boot 插件会:

1. 将 Spring Boot 的依赖管理中使用的 Kotlin 版本与该插件的版本保持一致.这可以通过将 kotlin.version 属性设置为与 Kotlin 插件的版本匹配的值来实现.This is achieved by setting the kotlin.version property with a value that matches the version of the Kotlin plugin.

2. 将任何 KotlinCompile 任务配置为使用 -java-parameters 编译器参数.

9.3. war 插件行为

当项目里使用了 war plugin ,Spring Boot 插件会:

1. 创建一个名为 bootWar 的 BootWar 任务, 用来为项目创建可执行的, far war. 另外对于标准的打包, 所以是 providedRuntime 的配置都会打包进 WEB-INF/lib-provided 里;
2. 配置一个依赖于 bootWar 任务的 assemble 任务;
3. 禁止 war 任务;
4. 配置 bootArchives 任务包含通过 bootWar 任务产生的 artifact.

9.4. 依赖管理插件行为

当项目里使用了 io.spring.dependency-management plugin ,Spring Boot 插件会自动导入 spring-boot-dependencies bom

9.5. application 插件行为

当项目里使用了 application plugin ,Spring Boot 插件会:

1. 创建一个名为 bootStartScripts 的 CreateStartScripts 的任务, 它会创建一个脚本用来使用 java -jar 命令启动在 bootArchives 配置里的 artifact. 该任务使用配置的 applicationDefaultJvmArgs 属性作为其 defaultJvmOpts 属性.
2. 创建一个名为 boot 的新的 distribution , 并将其配置为在其 lib 目录的 bootArchives 配置中包含 artifact, 在其 bin 目录中包含启动脚本.
3. 使用 mainClassName 作为 main 属性配置 bootRun 任务.
4. 使用 applicationDefaultJvmArgs 参数作为 jvmArgs 属性配置 bootRun 任务.
5. 使用 mainClassName 属性在 mainifest 作为 Start-Class 入口配置 bootJar 任务.

6. 使用 mainClassName 作为 mainifest 的 Start-Class 入口配置 bootWar 任务.

9.6. Maven 插件行为

当使用了 Gradle 的 maven plugin, Spring Boot 插件会配置一个 uploadBootArchives Upload 任务保证在它产生的 pom 里没有依赖被声明.