## 第二部分：入门 Getting Started

如果您开始使用Spring Boot或 Spring，请首先阅读本节。 它回答了基本的“What”，“How”和“Why”等问题。 它包括Spring Boot简介以及安装说明。 然后，我们将引导您构建您的第一个Spring Boot应用程序，并同时讨论一些核心原则。

### 8 SpringBoot介绍

Spring Boot可以轻松创建能独立运行的、生产级的、基于Spring的应用程序。 大多数Spring Boot应用程序只需要很少的Spring配置。

可以使用Spring Boot创建通过java -jar或传统的war部署启动的Java应用程序。 我们还提供了一个运行“spring脚本”的命令行工具。

我们的主要目标是：

为所有Spring开发提供从根本上更快且更容易接受的入门体验。

开箱即用。

提供大型项目通用的一系列非功能性功能（例如嵌入式服务器，安全性，度量标准，运行状况检查和外部化配置）

绝对没有代码生成，也不需要XML配置。

### 9 系统要求

Spring Boot 2.1.1.RELEASE需要Java 8，并且兼容Java 11。 还需要Spring Framework 5.1.3.RELEASE或更高版本。

为以下构建工具提供了显式构建支持：

Maven 3.3+；Gradle 4.4+

#### 9.1 Servlet容器

Spring Boot支持以下嵌入式servlet容器：

Tomcat 9.0 Servlet 4.0

Jetty 9.4 Servlet 3.1

Undertow 2.0 Servlet 4.0

也可以将Spring Boot应用程序部署到任何兼容Servlet 3.1+容器。

### 10 安装SpringBoot

#### 10.1 Maven方式安装

pom.xml通过parent继承：

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.1.RELEASE</version></parent>

spring-boot-starter-parent是一种使用Spring Boot的好方式，但它可能并不适用所有场景。 有时您可能需要从其他父POM继承，或者您可能不喜欢我们的默认设置。 在这些情况下，可以使用下面方式：

**<dependencyManagement>**

**<dependencies>**

**<dependency>**

*<!-- Import dependency management from Spring Boot -->*

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-dependencies**</artifactId>**

**<version>**2.1.1.RELEASE**</version>**

**<type>**pom**</type>**

**<scope>**import**</scope>  
</dependency>**

**</dependencies>  
</dependencyManagement>**

### 11 开发第一个SpringBoot应用

本节介绍如何开发一个简单的“Hello World！” Web应用程序，该应用程序突出了Spring Boot的一些主要功能。 我们使用Maven来构建这个项目，因为大多数IDE都支持它。

#### 11.1 创建pom

<?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
<modelVersion>**4.0.0**</modelVersion>  
<groupId>**com.example**</groupId>  
<artifactId>**myproject**</artifactId>  
<version>**0.0.1-SNAPSHOT**</version>  
<parent>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-parent**</artifactId>**

**<version>**2.1.1.RELEASE**</version>  
</parent>***<!-- Additional lines to be added here... -->***</project>**

#### 11.2 添加web依赖

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId></dependency>

</dependencies>

#### 11.3 编写代码

要完成我们的应用程序，我们需要创建一个Java文件。 默认情况下，Maven从src/main/java编译源代码，因此您需要创建该文件夹结构，然后添加名为src/main/java/Example.java的文件以包含以下代码：

**import** org.springframework.boot.\*;  
**import** org.springframework.boot.autoconfigure.\*;  
**import** org.springframework.web.bind.annotation.\*;

@RestController  
@EnableAutoConfiguration  
**public class** Example {

@RequestMapping("/")  
String home() {

**return *"Hello World!"***;  
}

**public static void** main(String[] args) **throws** Exception {

SpringApplication.run(Example.**class**, args);  
}

}

@RestController和@RequestMapping注解

@RequestMapping注释提供“路由”信息。 它告诉Spring，任何带有/ path的HTTP请求都应该映射到home方法。 @RestController注释告诉Spring将结果字符串直接渲染回调用者。

@EnableAutoConfiguration注解

第二个类级注释是@EnableAutoConfiguration。 这个注释告诉Spring Boot根据你添加的jar依赖关系“猜测”你想要如何配置Spring。 由于spring-boot-starter-web添加了Tomcat和Spring MVC，因此自动配置假定您正在开发Web应用程序并相应地设置Spring。

Starters和自动配置

自动配置旨在与“starter”配合使用，但这两个概念并不直接相关。 您可以starter之外自由选择jar依赖项。 Spring Boot仍然尽力自动配置您的应用程序。

starter只是封装了一组依赖关系。

main方法

main方法通过调用run来委托SpringBoot的SpringApplication类。 SpringApplication引导我们的应用程序，启动Spring，然后启动自动配置的Tomcat Web服务器。 我们需要将Example.class作为参数传递给run方法，以告诉SpringApplication谁是主要的Spring组件。 args数组也被传递以暴露任何命令行参数。

#### 11.4 运行

如果配置了SpringBoot Maven插件，则可以在项目根目录下执行命令：mvn spring-boot:run

然后访问localhost:8080

#### 11.5 创建一个可执行jar

我们通过创建一个完全自包含的可执行jar文件来完成我们的示例，我们可以在生产中运行它。 可执行jar（有时称为“fat jar”）是包含已编译类以及代码需要运行的所有jar依赖项的归档。

可执行jar和java

Java没有提供加载嵌套jar文件的标准方法（jar文件本身包含在jar中）。 如果您要分发自包含的应用程序，这可能会有问题。

为了解决这个问题，许多开发人员使用“超级”jar。 超级jar将所有应用程序依赖项中的所有类打包到一个存档中。 这种方法的问题在于很难看出应用程序中有哪些库。 如果在多个jar中使用相同的文件名（但具有不同的内容），也可能会有问题。

超级jar：将依赖的jar包内的class文件弄出来。

Spring Boot采用不同的方法，让你直接嵌套jar。

要创建可执行jar，我们需要将spring-boot-maven-plugin添加到我们的pom.xml中。 为此，请在依赖项部分下方插入以下行：

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId></plugin>

</plugins></build>

spring-boot-starter-parent POM包含<executions>配置以绑定repackage目标。 如果您不使用父POM，则需要自己声明此配置。

然后重新执行：mvn package

在target目录下,有一个maven生成的，被SpringBoot重新打包的.original文件，和一个可执行的jar文件。

通过java -jar 可执行jar则可运行。

## 第三部分 使用SpringBoot

本节详细介绍了如何使用Spring Boot。 涵盖了构建系统，自动配置以及如何运行应用程序等主题。 我们还介绍了一些Spring Boot最佳实践。 虽然Spring Boot没有什么特别之处（它只是你可以使用的另一个库），但是有一些建议，如果遵循这些建议，你的开发过程会更容易一些。

如果您刚开始使用Spring Boot，那么在深入了解本节之前，您应该阅读“入门指南”。

### 13 构建系统 Build System

强烈建议选择支持依赖关系管理且可以将工件发布到“Maven Central”存储库的构建系统。 如Maven或Gradle。 虽然也可以让Spring Boot与其他构建系统（例如Ant）一起使用，但它们并没有得到特别好的支持。

#### 13.1 依赖管理 Dependency Management

每个版本的Spring Boot都提供了它支持的依赖项列表。 实际上，您不需要为构建配置的任何依赖项提供版本，因为Spring Boot会为您管理这些依赖项。 当您升级Spring Boot时，这些依赖项也会以一致的方式升级。

如果需要，您仍然可以指定版本并覆盖Spring Boot的建议。

精选列表包含可以与Spring Boot一起使用的所有spring模块以及精制的第三方库。 该列表可作为标准的材料清单（spring-boot dependencies）提供，可与Maven和Gradle一起使用。

#### 13.2 Maven

Maven用户可以通过继承spring-boot-starter-parent项目以获得合理的默认值。 父项目提供以下功能：

Java 1.8作为默认编译器版本。

UTF-8源码编码方式

从Spring-boot-dependencies pom继承的Dependency Management部分，用于管理公共依赖项的版本。 此依赖关系管理允许您在自己的pom中使用时省略这些依赖项的<version>标记。

执行repackage目标。

合理的资源过滤

合理的插件配置（exec插件，Git提交ID等）。

application.properties和application.yml的合理资源过滤，包括特定profile配置文件（例如，application-dev.properties和application-dev.yml）

请注意，由于application.properties和application.yml文件接受Spring样式占位符（$ {...}），因此Maven过滤更改为使用@..@占位符。 （您可以通过设置名为resource.delimiter的Maven属性来覆盖它。）

**继承Starter Parent**

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.1.RELEASE</version></parent>

只需要在此依赖项上指定Spring Boot版本号。 如果导入其他starter，则可以安全地省略版本号。

通过该设置，您还可以通过覆盖自己项目中的属性来覆盖单个依赖项。 例如，要升级到另一个Spring Data版本系列，您需要将以下内容添加到pom.xml：

<properties>

<spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version></properties>

**不通过parent Pom引入SpringBoot**

不是每个人都喜欢继承spring-boot-starter-parent POM。 您可能拥有自己需要使用的公司标准parent，或者您可能更愿意明确声明所有Maven配置。

如果您不想使用spring-boot-starter-parent，您仍然可以通过使用scope = import依赖项来保持依赖项管理的好处，如下所示：

**<dependencyManagement>**

**<dependencies>**

**<dependency>**

*<!-- Import dependency management from Spring Boot -->*

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-dependencies**</artifactId>**

**<version>**2.1.1.RELEASE**</version>**

**<type>**pom**</type>**

**<scope>**import**</scope>  
</dependency>**

**</dependencies>  
</dependencyManagement>**

如上所述，前面的示例设置不允许您使用属性覆盖单个依赖项。 要获得相同的结果，您需要在spring-boot-dependencies条目之前在项目的dependencyManagement中添加一个条目。 例如，要升级到另一个Spring Data版本系列，可以将以下元素添加到pom.xml：

**<dependencyManagement>**

**<dependencies>**

*<!-- Override Spring Data release train provided by Spring Boot -->***<dependency>**

**<groupId>**org.springframework.data**</groupId>**

**<artifactId>**spring-data-releasetrain**</artifactId>**

**<version>**Fowler-SR2**</version>**

**<type>**pom**</type>**

**<scope>**import**</scope>  
</dependency>**

**<dependency>**

*<!-- Import dependency management from Spring Boot -->*

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-dependencies**</artifactId>**

**<version>**2.1.1.RELEASE**</version>**

**<type>**pom**</type>**

**<scope>**import**</scope>  
</dependency>**

**</dependencies>  
</dependencyManagement>**

**使用SpringBoot Maven插件**

Spring Boot包含一个Maven插件，可以将项目打包为可执行jar。 如果要使用插件，请将插件添加到<plugins>部分，如以下示例所示：

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin></plugins>

</build>

如果您使用Spring Boot starter parent pom，则只需添加插件。 除非您要更改父级中定义的设置，否则无需对其进行配置。

### 13.3 Starters

starts是一组的依赖关系描述符，您可以将其包含在应用程序中。 就可以获得所需的Spring和相关技术的一站式服务，而无需手动添加依赖。 例如，如果要开始使用Spring JPA进行数据库访问，请在项目中包含spring-boot-starter-data-jpa依赖项。

starters包含许多依赖项，这些依赖项是使项目快速启动和运行所需的依赖，以及一组受支持的托管传递依赖。

**starter的命名规范**

所有官方starter都遵循以下命名模式; spring-boot-starter- \*，其中\*是特定类型的应用程序。 此命名结构旨在帮助您找到starter。 许多IDE中的Maven集成允许您按名称搜索依赖项。 例如，安装了适当的Eclipse或STS插件后，您可以在POM编辑器中按ctrl-space并输入“spring-boot-starter”以获取完整列表。

第三方starter不应该以spring-boot命名，因为它是为官方Spring Boot工件保留的。 相反，第三方启动器通常以项目名称开头。 例如，名为thirdpartyproject的第三方starter通常被命名为thirdpartyproject-spring-boot-starter。

Spring Boot提供了以下starter,其group Id为org.springframework.boot：

|  |  |
| --- | --- |
| **名字** | **描述** |
| spring-boot-starter | 核心启动器，包括自动配置支持，日志记录和YAML |
| spring-boot-starter-activemq | 使用Apache ActiveMQ进行JMS消息传递 |
| spring-boot-starter-amqp | 使用Spring AMQP和Rabbit MQ |
| spring-boot-starter-aop | 使用Spring AOP和AspectJ进行面向切面编程 |
| spring-boot-starter-artemis | 使用Apache Artemis进行JMS消息传递 |
| spring-boot-starter-batch | 使用Spring Batch |
| spring-boot-starter-cache | 使用Spring Framework的缓存支持 |
| spring-boot-starter-cloud-connectors | 使用Spring Cloud Connectors，简化了Cloud Foundry和Heroku等云平台中的服务连接 |
| spring-boot-starter-data-cassandra | 使用Cassandra分布式数据库和Spring Data Cassandra |
| spring-boot-starter-data-cassandra-reactive | 使用Cassandra分布式数据库和Spring Data Cassandra Reactive |
| spring-boot-starter-data-couchbase | 使用Couchbase面向文档的数据库和Spring Data Couchbase |
| spring-boot-starter-data-couchbase-reactive | 使用Couchbase面向文档的数据库和Spring Data Couchbase Reactive |
| spring-boot-starter-data-elasticsearch | 使用Elasticsearch搜索和分析引擎以及Spring Data Elasticsearch |
| spring-boot-starter-data-jdbc | 使用Spring Data JDBC |
| spring-boot-starter-data-jpa | 将Spring Data JPA与Hibernate一起使用 |
| spring-boot-starter-data-ldap | 使用Spring Data LDAP |
| spring-boot-starter-data-mongodb | 使用MongoDB面向文档的数据库和Spring Data MongoDB |
| spring-boot-starter-data-mongodb-reactive | 使用MongoDB面向文档的数据库和Spring Data MongoDB Reactive |
| spring-boot-starter-data-neo4j | 使用Neo4j图形数据库和Spring Data Neo4j |
| spring-boot-starter-data-redis | 与Spring Data Redis和Lettuce客户端一起使用Redis keyvalue数据存储 |
| spring-boot-starter-data-redis-reactive | 使用Redis keyvalue数据存储与Spring Data Redis被动和Lettuce客户端 |
| spring-boot-starter-data-rest | 使用Spring Data REST通过REST公开Spring Data存储库 |
| spring-boot-starter-data-solr | 使用Apache Solr搜索平台和Spring Data Solr |
| spring-boot-starter-freemarker | 使用FreeMarker视图构建MVC Web应用程序 |
| spring-boot-starter-groovy-templates | 使用Groovy模板视图构建MVC Web应用程序 |
| spring-boot-starter-hateoas | 使用Spring MVC和Spring HATEOAS构建基于超媒体的RESTful Web应用程序 |
| spring-boot-starter-integration | 使用Spring Integration |
| spring-boot-starter-jdbc | 将JDBC与HikariCP连接池一起使用 |
| spring-boot-starter-jersey | 使用JAX-RS和Jersey构建RESTful Web应用程序；spring-boot-starter-web的替代品 |
| spring-boot-starter-jooq | 使用jOOQ访问SQL数据库。spring-bootstarter-data-jpa或spring-boot-starterjdbc的替代品 |
| spring-boot-starter-json | json读写 |
| spring-boot-starter-jta-atomikos | GTA交易个人的 |
| spring-boot-starter-mail | 使用Java Mail和Spring Framework的电子邮件发送支持 |
| spring-boot-starter-mustache | 使用Mustache视图构建Web应用程序 |
| spring-boot-starter-oauth2-client | 使用Spring Security的OAuth2 / OpenID Connect客户端功能 |
| spring-boot-starter-oauth2-resource-server | 使用Spring Security的OAuth2资源服务器功能 |
| spring-boot-starter-quartz | 使用Quartz调度程序 |
| spring-boot-starter-security | 使用Spring Security |
| spring-boot-starter-test | 使用JUnit，Hamcrest和Mockito等库来测试Spring Boot应用程序 |
| spring-boot-starter-thymeleaf | 使用Thymeleaf视图构建MVC Web应用程序 |
| spring-boot-starter-validation | 使用Java Bean Validation和Hibernate Validator |
| spring-boot-starter-web | 使用Spring MVC构建Web（包括RESTful）应用程序，Tomcat作为默认嵌入式容器 |
| spring-boot-starter-web-services | 使用Spring Web Services的 |
| spring-boot-starter-webflux | 使用Spring Framework的Reactive Web支持构建WebFlux应用程序 |
| spring-boot-starter-websocket | 使用Spring Framework的WebSocket支持构建WebSocket应用程序 |
| spring-boot-starter-actuator | 使用Spring Boot的Actuator，它提供生产就绪功能，帮助您监控和管理您的应用程序 |
| spring-boot-starter-jetty | 使用Jetty作为嵌入式servlet容器，spring-bootstarter-tomcat的替代品 |
| spring-boot-starter-log4j2 | 使用Log4j2进行日志记录， spring-boot-starterlogging的替代方案 |
| spring-boot-starter-logging | 使用Logback进行日志记录， 默认日志启动器 |
| spring-boot-starter-reactor-netty | 使用Reactor Netty作为嵌入式响应式HTTP服务器的 |
| spring-boot-starter-tomcat | 使用Tomcat作为嵌入式servlet容器，spring-bootstarter-web使用的默认servlet容器启动器 |
| spring-boot-starter-undertow | 使用Undertow作为嵌入式servlet容器，spring-boot-startertomcat的替代品 |

### 14 代码布局

Spring Boot不需要任何特定的代码布局。 但是，有一些最佳实践可以提供帮助。

#### 14.1 默认包

当类不包含包声明时，它被认为是在“默认包”中。 通常不鼓励使用“默认包”，应该避免使用。 对于使用@ComponentScan，@EntityScan或@SpringBootApplication注释的Spring Boot应用程序，它可能会导致特定问题，因为每个jar中的每个类都被读取。

我们建议您遵循Java推荐的包命名约定并使用反向域名（例如，com.example.project）。

#### 14.2 主应用程序类位置

我们通常建议您将主应用程序类放在其他类之上的根包中。 @SpringBootApplication注释通常放在您的主类上，它隐式地为某些项定义了一个基础“搜索包”。 例如，如果您正在编写JPA应用程序，则使用@SpringBootApplication带注释类的包来搜索@Entity项。 使用根包还允许组件扫描仅应用于您的项目。

如果您不想使用@SpringBootApplication，也可以使用@EnableAutoConfiguration和@ComponentScan注解替代它。

以下结构显示了典型的代码布局：

com

+- example

+- myapplication

+- Application.java

|

+- customer

| +- Customer.java

| +- CustomerController.java

| +- CustomerService.java

| +- CustomerRepository.java  
|  
+- order

+- Order.java

+- OrderController.java

+- OrderService.java

+- OrderRepository.java

Application.java文件将声明main方法以及基本的@SpringBootApplication，如下所示：

**package** com.example.myapplication;

**import** org.springframework.boot.SpringApplication;  
**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication  
**public class** Application {

**public static void** main(String[] args) {

SpringApplication.run(Application.**class**, args);  
}

}

### 15 配置类

Spring Boot支持基于Java的配置。 虽然可以将SpringApplication与XML一起使用，但我们通常建议您的主配置源是单个@Configuration类。 通常，定义main方法的类是主要@Configuration的良好候选者。

#### 15.1 导入其他配置类

您无需将所有@Configuration放入单个类中。 @Import注释可用于导入其他配置类。 或者，您可以使用@ComponentScan自动获取所有Spring组件，包括@Configuration类。

#### 15.2 导入XML配置

如果您绝对必须使用基于XML的配置，我们建议您仍然使用@Configuration类。 然后，您可以使用@ImportResource注解来加载XML配置文件。

### 16 自动配置

Spring Boot会尝试根据您添加的jar依赖来自动配置Spring应用程序。 例如，如果HSQLDB在您的类路径上，并且您尚未手动配置任何数据库连接bean，则Spring Boot会自动配置内存数据库。

您需要通过将@EnableAutoConfiguration或@SpringBootApplication注释添加到其中一个@Configuration类来开启自动配置。

您应该只添加一个@SpringBootApplication或@EnableAutoConfiguration注解。通常建议您仅将其添加到主@Configuration类中。

#### 16.1替换自动配置

自动配置是非侵入性的。 在任何时候，您都可以定义自己的配置以替换自动配置的特定部分。 例如，如果添加自己的DataSource bean，则默认的嵌入式数据库支持会不生效。

如果您需要了解当前应用的自动配置，请使用--debug开关启动应用程序。 这样可以输出自动配置的相关日志。

#### 16.2 禁用特定的自动配置类

可以使用@EnableAutoConfiguration的exclude属性来排除不需要的自动配置类，如以下示例所示：

**import** org.springframework.boot.autoconfigure.\*;  
**import** org.springframework.boot.autoconfigure.jdbc.\*;  
**import** org.springframework.context.annotation.\*;  
@Configuration  
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})  
**public class** MyConfiguration {  
}

如果自动配置类不在类路径上，则可以使用注解的excludeName属性并指定完全限定名称。 最后，您还可以使用spring.autoconfigure.exclude属性控制要排除的自动配置类列表。

### 17 Spring Bean和依赖注入

您可以自由使用任何标准的Spring Framework技术来定义bean及其注入的依赖项。 为简单起见，我们经常发现使用@ComponentScan（找到你的bean）并使用@Autowired（做构造函数注入）效果很好。

如果按照上面的建议构建代码（应用程序类放置根包中），则可以添加不带任何参数的@ComponentScan。 所有应用程序组件（@ Component，@ Service，@ Repository，@ Controller等）都自动注册为Spring Beans。

以下示例显示了一个@Service Bean，它使用构造函数注入来获取所需的RiskAssessor bean：

**package** com.example.service;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.stereotype.Service;  
@Service  
**public class** DatabaseAccountService **implements** AccountService {

**private final** RiskAssessor riskAssessor;

@Autowired

**public** DatabaseAccountService(RiskAssessor riskAssessor) {

**this**.riskAssessor = riskAssessor;  
}  
*// ...*

}

如果bean只有一个构造函数，则可以省略@Autowired，如以下示例所示：

@Service  
**public class** DatabaseAccountService **implements** AccountService {

**private final** RiskAssessor riskAssessor;

**public** DatabaseAccountService(RiskAssessor riskAssessor) {

**this**.riskAssessor = riskAssessor;  
}  
*// ...*

}

使用构造函数注入可以将riskAssessor字段标记为final，表示无法随后更改它。

### 18 使用@SpringBootApplication注解

许多Spring Boot开发人员喜欢他们的应用程序使用自动配置，组件扫描，并能够在他们的“应用程序类”上定义额外的配置。 单个@SpringBootApplication注释可用于启用这三个功能，即：

@EnableAutoConfiguration:开启SpringBoot自动配置机制

@ComponentScan：在应用程序所在的包上启用@Component扫描

@Configuration：允许在上下文中注册额外的bean或导入其他配置类

@SpringBootApplication是一个组合注解，包含了@ Configuration，@ EnableAutoConfiguration和@ ComponentScan及其默认属性，如以下示例所示：

**package** com.example.myapplication;  
**import** org.springframework.boot.SpringApplication;  
**import** org.springframework.boot.autoconfigure.SpringBootApplication;  
@SpringBootApplication *// same as @Configuration @EnableAutoConfiguration @ComponentScan***public class** Application {

**public static void** main(String[] args) {

SpringApplication.run(Application.**class**, args);  
}

}

### 19 运行应用

将应用程序打包为jar并使用嵌入式HTTP服务器的最大优势之一是，可以像其他应用一样运行该应用程序。 调试Spring Boot应用程序也很容易。 您不需要任何特殊的IDE插件或扩展。

本节仅介绍基于jar的打包。 如果您选择将应用程序打包为war文件，则应参阅服务器和IDE文档。

#### 19.1 从IDE上运行

可以将Spring Boot应用作为简单的Java程序，在IDE上运行。 但是，您首先需要导入项目。 导入步骤因IDE和构建系统而异。 大多数IDE可以直接导入Maven项目。 例如，Eclipse用户可以从“文件”菜单中选择“导入...”→“现有Maven项目”。

如果无法将项目直接导入IDE，则可以使用构建插件生成IDE元数据。 Maven包含Eclipse和IDEA的插件。 Gradle提供各种IDE的插件。

#### 19.2 运行一个打包好的应用

如果是使用Spring Boot Maven或Gradle插件来创建可执行jar，则可以使用java -jar运行应用程序，如以下示例所示：

java -jar target/myapplication-0.0.1-SNAPSHOT.jar

#### 19.3 使用Maven插件运行

Spring Boot Maven插件包含一个run目标，可用于快速编译和运行您的应用程序。 应用程序以分解形式运行，就像在IDE中一样。 以下示例显示了运行Spring Boot应用程序的典型Maven命令：

mvn spring-boot:run

您可能还想使用MAVEN\_OPTS操作系统环境变量，如以下示例所示：

export MAVEN\_OPTS=-Xmx1024m

### 20 开发者工具

Spring Boot包含一组额外的工具，可以使程序开发体验更加愉快。 spring-boot-devtools模块可以包含在任何项目中，以提供额外的功能。 要包含devtools支持，请将模块依赖项添加到您的构建中，如以下Maven所示：

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional></dependency>

</dependencies>

运行完全打包的应用程序时会自动禁用开发人员工具。 如果您的应用程序是从java -jar启动的，或者它是从特殊的类加载器启动的，那么它将被视为“生产应用程序”。 在Maven中将依赖项标记为可选（如上所示）是最佳实践，可防止devtools传递应用于使用项目的其他模块。

重新打包的jar默认情况下不包含devtools。 如果要使用某个远程devtools功能，则需要禁用excludeDevtools构建属性以包含它。Maven和Gradle插件都支持该property。

#### 20.1 默认属性

Spring Boot支持的几个库使用缓存来提高性能。 例如，模板引擎缓存已编译的模板以避免重复解析模板文件。 此外，Spring MVC可以在提供静态资源时为响应添加HTTP缓存头。

虽然缓存在生产中非常有用，但在开发过程中可能会适得其反，从而使您无法看到刚刚在应用程序中进行的更改。 因此，spring-bootdevtools默认禁用缓存选项。

缓存选项通常由application.properties文件中的属性进行配置。 例如，Thymeleaf提供了spring.thymeleaf.cache属性。 spring-boot-devtools模块不需要手动设置这些属性，而是自动应用合理的开发配置。

因为在开发Spring MVC和Spring WebFlux应用程序时需要有关Web请求的更多信息，所以开发者工具将为Web启用DEBUG日志记录。 这将为您提供有关请求，处理，响应结果等的信息。如果您希望记录所有请求详细信息（包括可能的敏感信息），您可以配置spring.http.log-request-details属性

如果您不希望应用属性默认值，则可以在application.properties中将spring.devtools.addproperties设置为false。

有关devtools应用的完整属性列表，请参阅DevToolsPropertyDefaultsPostProcessor。

#### 20.2 自动重启

使用spring-boot-devtools的应用程序会在类路径上的文件发生更改时自动重启。 在IDE中工作时，这可能是一个有用的功能，因为它为代码更改提供了非常快速的反馈。 默认情况下，将监视类路径上指向文件夹的任何文件的更改。 请注意，某些资源（如静态资源和视图模板）无需重新启动应用程序。

Spring Boot提供的重启技术使用两个类加载器。 不更改的类（例如，来自第三方jar的类）将加载到基类加载器中。 正在开发的类将加载到restart类加载器中。 重新启动应用程序时，将重新启动restart类加载器并创建一个新的类加载器。 这种方法意味着应用程序重新启动通常比“冷启动”快得多，因为基本类加载器已经可用并已填充。

如果您发现重新启动对于您的应用程序来说不够快或遇到类加载问题，您可以考虑从ZeroTurnaround重新加载JRebel等技术。 这些工作通过在加载类时重写类使它们更适合重新加载。

**记录更改**

默认情况下，每次应用程序重新启动时，都会记录一个显示条件评估增量的报告。 该报告显示了在进行更改（例如添加或删除Bean以及设置配置属性）时对应用程序的自动配置所做的更改。

要禁用报告的日志记录，请设置以下属性：

spring.devtools.restart.log-condition-evaluation-delta=false

**排除资源**

某些资源在更改时不一定需要触发重启。 例如，可以就地编辑Thymeleaf模板。 默认情况下，更改/META-INF/maven，/META-INF/resources，/resources，/static，/public或/templates中的资源不会触发restart，但会触发实时重新加载。 如果要自定义这些排除项，可以使用spring.devtools.restart.exclude属性。 例如，要仅排除/static和/public，您需要设置以下属性：

spring.devtools.restart.exclude=static/\*\*,public/\*\*

如果要保留这些默认值并添加其他排除项，请改用属性：

spring.devtools.restart.additional-exclude

**监测其他路径**

当您对不在类路径中的文件进行更改时，您可能希望重新启动或重新加载应用程序。 为此，请使用spring.devtools.restart.additionalpaths属性配置其他路径以监视更改。 您可以使用前面描述的spring.devtools.restart.exclude属性来控制其他路径下的更改是触发完全重新启动还是实时重新加载。

**禁用重启**

如果您不想使用重新启动功能，可以使用spring.devtools.restart.enabled属性将其禁用。 在大多数情况下，您可以在application.properties中设置此属性（这样做仍会初始化重新启动的类加载器，但它不会监视文件更改）。

如果需要完全禁用重新启动支持（例如，因为它不能与特定库一起使用），则需要在调用SpringApplication.run(...)之前将spring.devtools.restart.enabled 系统属性设置为false，如图所示 在以下示例中：

**public static void** main(String[] args) {

System.setProperty(***"spring.devtools.restart.enabled"***, ***"false"***);

SpringApplication.run(MyApp.**class**, args);  
}

**使用触发器文件**

如果使用不断编译已更改文件的IDE，则可能更喜欢仅在特定时间触发重新启动。 为此，您可以使用“触发器文件”，这是一个特殊文件，当您想要实际触发重新启动检查时必须对其进行修改。 更改文件只会触发检查，只有在Devtools检测到必须执行某些操作时才会重新启动。 触发器文件可以手动更新，也可以使用IDE插件更新

要使用触发器文件，请将spring.devtools.restart.trigger-file属性设置为触发器文件的路径。

您可能希望将spring.devtools.restart.trigger文件设置为全局设置，以便所有项目的行为方式相同

**自定义restart类加载器**

如前面在Restart vs Reload部分中所述，使用两个类加载器实现了重启功能。 对于大多数应用程序，此方法运行良好。 但是，它有时会导致类加载问题。

默认情况下，IDE中的任何打开项目都使用“restart”类加载器加载，并且任何常规.jar文件都使用“base”类加载器加载。 如果您处理多模块项目，并且并非每个模块都导入到IDE中，则可能需要自定义内容。 为此，您可以创建META-INF/spring-devtools.properties文件。

spring-devtools.properties文件可以包含以restart.exclude和restart.include为前缀的属性。 include元素是应该被提取到“restart”类加载器中的项，而exclude元素是应该被下推到“base”类加载器中的项。 该属性的值是应用于类路径的正则表达式模式，如以下示例所示：

**restart.exclude.companycommonlibs**=/mycorp-common-[\\w-]+\.jar  
**restart.include.projectcommon**=/mycorp-myproj-[\\w-]+\.jar

所有属性键必须是唯一的。 只要属性以restart.include或者restart.exclude开头即可。

将加载类路径中的所有META-INF/spring-devtools.properties。 您可以将文件打包到项目中，也可以打包在项目使用的库中。

**已知限制**

对于使用标准ObjectInputStream反序列化的对象，重新启动功能不起作用。 如果需要反序列化数据，可能需要将Spring的ConfigurableObjectInputStream与Thread.currentThread().getContextClassLoader()结合使用。

不幸的是，几个第三方库反序列化时没有考虑上下文类加载器。 如果您发现此类问题，则需要向原始作者请求修复。

#### 20.3 LiveReload

spring-boot-devtools模块包含可以使用的嵌入式LiveReload服务器在更改资源时触发浏览器刷新。 LiveReload浏览器扩展是免费的，适用于Chrome，Firefox和Safari。

如果您不想在应用程序运行时启动LiveReload服务器，则可以将spring.devtools.livereload.enabled属性设置为false。

您一次只能运行一个LiveReload服务器。 在启动应用程序之前，请确保没有其他LiveReload服务器正在运行。 如果从IDE启动多个应用程序，则只有第一个具有LiveReload支持。

#### 20.4 全局设置

您可以通过将名为.spring-bootdevtools.properties的文件添加到$HOME文件夹来配置全局devtools设置（请注意，文件名以“.”开头）。 添加到此文件的任何属性都适用于计算机上使用devtools的所有Spring Boot应用程序。 例如，要将restart配置为始终使用触发器文件，您需要添加以下属性：

**~/.spring-boot-devtools.properties.**

**spring.devtools.reload.trigger-file**=.reloadtrigger

在.spring-boot-devtools.properties中激活的配置文件不会影响profile配置文件的加载。

#### 20.5 远程应用

Spring Boot开发者工具不仅限于本地开发。 远程运行应用程序时，您还可以使用多个功能。 远程支持需要配置的，要启用它，您需要确保devtools包含在repackage的jar中，如下面的清单所示：

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId><artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<excludeDevtools>false</excludeDevtools></configuration>

</plugin></plugins>

</build>

然后，您需要设置spring.devtools.remote.secret属性，如下所示：

**spring.devtools.remote.secret**=mysecret

在远程应用程序上启用spring-boot-devtools存在安全风险。 您永远不应该在生产部署上启用支持。

远程devtools支持由两部分组成：一个接受连接的服务器端端点和一个在IDE中运行的客户端应用程序。 设置spring.devtools.remote.secret属性后，将自动启用服务器组件。 必须手动启动客户端组件。

**运行客户端程序**

客户端应用程序旨在从IDE中运行。 您需要运行org.springframework.boot.devtools.RemoteSpringApplication，其类路径与您连接的远程项目相同。 应用程序的单个必需参数是它连接的远程URL。

例如，如果您使用的是Eclipse或STS，并且有一个名为my-app的项目已部署到Cloud Foundry，那么您将执行以下操作：

从“run”菜单中选择“run Configurations...”

创建一个新的Java应用程序“启动配置”。

浏览my-app项目。

使用org.springframework.boot.devtools.RemoteSpringApplication作为程序主类。

将远程Url：https://myapp.cfapps.io添加到Program参数

因为远程客户端使用与真实应用程序相同的类路径，所以它可以直接读取应用程序属性。 这是如何读取spring.devtools.remote.secret属性并将其传递给服务器进行身份验证的方法。

始终建议使用https：//作为连接协议，以便加密流量并且不会被截获密码。

如果需要使用代理来访问远程应用程序，请配置spring.devtools.remote.proxy.host和spring.devtools.remote.proxy.port属性。

**远程更新**

远程客户端以与本地重新启动相同的方式监视应用程序类路径以进行更改。 任何更新的资源都会被推送到远程应用程序，并且（如果需要）会触发重新启动。 如果您迭代使用本地没有的云服务的功能，这将非常有用。 通常，远程更新和重新启动比完全重建和部署周期快得多。

仅在远程客户端运行时监视文件。 如果在启动远程客户端之前更改文件，则不会将其推送到远程服务器。

### 21 打包应用部署到生成环境

可执行jar可用于生产部署。 由于它们是独立的，因此它们也非常适合基于云的部署。

## 第四部分：Spring Boot功能

本节深入介绍Spring Boot的详细信息。 在这里，您可以了解可能想要使用和自定义的主要功能。 如果您还没有这样做，您可能需要阅读“第二部分：入门”和“第三部分：使用Spring Boot”部分，以便您掌握基础知识。

### 23 SpringApplication

SpringApplication类提供了一种方便的方法来引导从main()方法启动的Spring应用程序。 在许多情况下，您可以委派给静态SpringApplication.run方法，如下例所示：

**public static void** main(String[] args) {

SpringApplication.run(MySpringConfiguration.**class**, args);  
}

默认情况下，会显示INFO日志消息，包括一些相关的启动详细信息，例如启动应用程序的用户。 如果需要INFO以外的日志级别，可以按照第26.4节“日志级别”中的说明进行设置。

#### 23.1 启动失败

如果您的应用程序无法启动，则已注册的FailureAnalyzers有机会提供专用错误消息和具体操作来解决问题。 例如，如果您在端口8080上启动Web应用程序并且该端口已在使用中，您应该会看到类似于以下消息的内容：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
APPLICATION FAILED TO START  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Description:  
Embedded servlet container failed to start. Port 8080 was already in use.  
Action:  
Identify and stop the process that's listening on port 8080 or configure this application to listen on  
another port.

Spring Boot提供了许多FailureAnalyzer实现，您可以添加自己的。

如果没有故障分析器能够处理异常，您仍然可以显示完整的条件报告，以便更好地了解出现了什么问题。 为此，您需要启用debug属性或为

org.springframework.boot.autoconfigure.logging.ConditionEvauationReportLoggingListener启用DEBUG日志记录。

例如，如果使用java -jar运行应用程序，则可以按如下方式启用debug属性：

java -jar myproject-0.0.1-SNAPSHOT.jar --debug

#### 23.2 自定义Banner

可以通过将banner.txt文件添加到类路径或将spring.banner.location属性设置为此类文件的位置来更改启动时打印的banner。 如果文件的编码不是UTF-8，则可以设置spring.banner.charset。 除了文本文件，您还可以将banner.gif，banner.jpg或banner.png图像文件添加到类路径或设置spring.banner.image.location属性。 图像将转换为ASCII艺术表示，并打印在任何文本banner上方。

在banner.txt文件中，您可以使用以下任何占位符：

|  |  |
| --- | --- |
| **变量名称** | **说明** |
| ${application.version} | 应用程序的版本号，在MANIFEST.MF中声明。 例如，Implementation-Version：1.0打印为1.0 |
| ${application.formatted-version} | 应用程序的版本号，在MANIFEST.MF中声明并格式化以显示（用括号括起并以v为前缀）。 例如（v1.0）。 |
| ${spring-boot.version} | 您正在使用的Spring Boot版本。 例如2.1.1.RELEASE。 |
| ${spring-boot.formatted-version} | 您正在使用的Spring Boot版本，格式化显示（用括号括起来并以v为前缀）。 例如（v2.1.1.RELEASE）。 |
| ${Ansi.NAME} (or ${AnsiColor.NAME}, ${AnsiBackground.NAME}, ${AnsiStyle.NAME}) | 其中NAME是ANSI转义码的名称。 有关详细信息，请参见AnsiPropertySource |
| ${application.title} | 应用程序的标题，如MANIFEST.MF中所声明的。 例如，Implementation-Title：MyApp打印为MyApp。 |

如果要以编程方式生成banner，可以使用SpringApplication.setBanner(...)方法。 使用org.springframework.boot.Banner接口并实现自己的printBanner（）方法。

您还可以使用spring.main.banner-mode属性来确定是否必须在System.out（console）上打印banner，发送到配置的日志记录器（log），或者根本不生成横幅（off）。

打印的banner的类会注册成名称为：springBootBanner的单例bean。

YAML映射为false，因此如果要在应用程序中禁用横幅，请务必添加引号，如以下示例所示：

**spring**:

**main**:

**banner-mode**: ***"off"***

#### 23.3 自定义SpringApplication

如果SpringApplication默认值不符合您的要求，您可以改为创建本地实例并对其进行自定义。 例如，要关闭banner，您可以写：

**public static void** main(String[] args) {

SpringApplication app = **new** SpringApplication(MyConfiguration.**class**);

app.setBannerMode(Banner.Mode.OFF);

app.run(args);  
}

传递给SpringApplication的构造参数是Spring bean的配置类。 在大多数情况下，这些是对@Configuration类的引用，但它们也可以是对XML配置或扫描的包的引用。

也可以使用application.properties文件配置SpringApplication。 有关详细信息，请参见第24章，外部化配置。

完整的配置选项，请参阅SpringApplication Javadoc。

#### 23.4 Fluent Builder API

如果需要构建ApplicationContext层次结构（具有父/子关系的多个上下文），或者如果您更喜欢使用Fluent Builder API，则可以使用SpringApplicationBuilder。

SpringApplicationBuilder允许您将多个方法调用链接在一起，并包含允许您创建层次结构的父方法和子方法，如以下示例所示：

**new** SpringApplicationBuilder()

.sources(Parent.**class**)

.child(Application.**class**)

.bannerMode(Banner.Mode.OFF)

.run(args);

创建ApplicationContext层次结构时存在一些限制。 例如，Web组件必须包含在子上下文中，并且相同的Environment用于父上下文和子上下文。 有关完整详细信息，请参阅SpringApplicationBuilder Javadoc。

#### 23.5 程序事件和监听器

除了通常的Spring Framework事件，例如ContextRefreshedEvent，SpringApplication还会发送一些额外的应用程序事件。

某些事件实际上是在创建ApplicationContext之前触发的，因此您无法将这些事件的侦听器注册为@Bean。 您可以使用SpringApplication.addListeners（...）方法或SpringApplicationBuilder.listeners（...）方法注册它们。

如果您希望自动注册这些侦听器，无论应用程序的创建方式如何，您都可以将META-INF/spring.factories文件添加到项目中，并使用键

org.springframework.contex.ApplicationListener，如以下示例所示：

org.springframework.context.ApplicationListener=com.example.project.MyListener

应用程序运行时，应按以下顺序发送应用程序事件：

1. ApplicationStartingEvent在运行开始时但在任何处理之前发送，但监听器和初始化程序的注册除外。

2.ApplicationEnvironmentPreparedEvent在上下文创建之前，使用Environment时将发送。

3. ApplicationPreparedEvent在刷新开始之前但在加载bean定义之后发送。

4.ApplicationStartedEvent在刷新上下文之后但在调用任何应用程序和命令行运行程序之前发送。

5. ApplicationReadyEvent在调用任何应用程序和命令行运行程序之后发送。 它表示应用程序已准备好为请求提供服务。

6. 如果启动时发生异常，则发送ApplicationFailedEvent

您通常不需要使用应用程序事件，但知道它们存在可能很方便。 在内部，Spring Boot使用事件来处理各种任务。

使用Spring Framework的事件发布机制发送应用程序事件。 此机制的一部分确保在子上下文中发布给侦听器的事件也会在任何祖先上下文中发布给侦听器。 因此，如果您的应用程序使用SpringApplication实例的层次结构，则侦听器可能会收到相同类型的应用程序事件的多个实例。

为了允许侦听器区分其上下文的事件和后代上下文的事件，它应该请求注入其应用程序上下文，然后将注入的上下文与事件的上下文进行比较。 可以通过实现ApplicationContextAware来注入上下文，或者，如果侦听器是bean，则使用@Autowired注入上下文。

#### 23.6 Web环境

SpringApplication试图代表您创建正确类型的ApplicationContext。用于确定WebApplicationType的算法非常简单：

1. 如果存在Spring MVC，则使用AnnotationConfigServletWebServerApplicationContext

2. 如果Spring MVC不存在且Spring WebFlux存在，那么使用AnnotationConfigReactiveWebServerApplicationContext

3. 否则，使用AnnotationConfigApplicationContext

这意味着如果您在同一个应用程序中使用Spring MVC和Spring WebFlux中的新WebClient，默认情况下将使用Spring MVC。 您可以通过调用setWebApplicationType（WebApplicationType）来覆盖它。

也可以完全控制调用setApplicationContextClass（...），传递ApplicationContext类型。

在JUnit测试中使用SpringApplication时，通常需要调用setWebApplicationType（WebApplicationType.NONE）。

#### 23.7 访问应用程序参数

如果需要访问传递给SpringApplication.run（...）的应用程序参数，可以注入org.springframework.boot.ApplicationArguments bean。 ApplicationArguments接口提供对原始String []参数以及解析选项和非选项参数的访问，如以下示例所示：

**import** org.springframework.boot.\*;  
**import** org.springframework.beans.factory.annotation.\*;  
**import** org.springframework.stereotype.\*;  
@Component  
**public class** MyBean {

@Autowired  
**public** MyBean(ApplicationArguments args) {

**boolean** debug = args.containsOption(***"debug"***);

List<String> files = args.getNonOptionArgs();  
*// if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]*

}

}

Spring Boot还在Spring Environment中注册了一个CommandLinePropertySource。 这使您还可以使用@Value注释注入单个应用程序参数。

#### 23.8 使用ApplicationRunner或CommandLineRunner

如果在SpringApplication启动后需要运行某些特定代码，则可以实现ApplicationRunner或CommandLineRunner接口。 两个接口以相同的方式工作，并提供单个run方法，该方法在SpringApplication.run（...）完成之前调用。

CommandLineRunner接口提供对应用程序参数的访问（String[] args），而ApplicationRunner使用前面讨论的ApplicationArguments接口。 以下示例显示了带有run方法的CommandLineRunner：

**import** org.springframework.boot.\*;  
**import** org.springframework.stereotype.\*;  
@Component  
**public class** MyBean **implements** CommandLineRunner {

**public void** run(String... args) {  
*// Do something...*}

}

如果定义了必须以特定顺序调用的多个CommandLineRunner或ApplicationRunner bean，则还可以实现org.springframework.core.Ordered接口或使用org.springframework.core.annotation.Order注释。

#### 23.9 程序退出

每个SpringApplication都会向JVM注册一个关闭钩子，以确保ApplicationContext在退出时正常关闭。 可以使用所有标准的Spring生命周期回调（例如DisposableBean接口或@PreDestroy注释）。

此外，如果bean希望在调用SpringApplication.exit（）时返回特定的退出代码，则bean可以实现org.springframework.boot.ExitCodeGenerator接口。 然后可以将此退出代码传递给System.exit（）以将其作为状态代码返回，如以下示例所示：

@SpringBootApplication  
**public class** ExitCodeApplication {

@Bean  
**public** ExitCodeGenerator exitCodeGenerator() {

**return** () -> 42;  
}  
**public static void** main(String[] args) {

System.exit(SpringApplication.exit(SpringApplication.run(ExitCodeApplication.**class**,

args)));

}  
}

此外，ExitCodeGenerator接口可以通过异常实现。 遇到这样的异常时，Spring Boot返回实现的getExitCode（）方法提供的退出代码。

#### 23.10 管理功能

通过指定spring.application.admin.enabled属性，可以为应用程序启用与管理相关的功能。 这会在平台MBeanServer上公开SpringApplicationAdminMXBean。 您可以使用此功能远程管理Spring Boot应用程序。 此功能对于任何服务包装器实现也很有用。

如果您想知道应用程序正在运行的HTTP端口，请使用local.server.port键获取该属性。

启用此功能时要小心，因为MBean公开了一种关闭应用程序的方法。

### 24 外部配置

Spring Boot允许您外部化配置，以便您可以在不同的环境中使用相同的应用程序代码。 您可以使用properties文件，YAML文件，环境变量和命令行参数来外部化配置。 可以使用@Value注释将属性值直接注入到bean中，通过Spring的Environment抽象访问，或者通过@ConfigurationProperties绑定到结构化对象。

Spring Boot使用一个非常特殊的PropertySource命令，旨在允许合理地覆盖配置。 按以下顺序生效配置属性：

1. 在主目录上开发者工具全局设置属性（当devtools处于活动状态时，〜/.spring-bootdevtools.properties）。

2. 测试中的@TestPropertySource注释

3. 测试中的属性。 可在@SpringBootTest上使用，以及用于测试应用程序特定片段的测试注释。

4. 命令行参数。

5. SPRING\_APPLICATION\_JSON中的属性（嵌入在环境变量或系统属性中的内联JSON）。

6. ServletConfig初始化参数

7. ServletContext 初始化参数

8. 来自java:comp/env的JNDI属性

9. Java系统属性(System.getProperties())

10. 系统环境变量

11. RandomValuePropertySource，仅对属性random.\*有效

12 jar包之外的profile配置（application- {profile} .properties和YAML变量）

13 jar包内的profile配置（application- {profile} .properties和YAML变量）

14 jar包之外的配置文件（application.properties和YAML变量）

15 jar包之内的配置文件（application.properties和YAML变量）

16 @Configuration类上的@PropertySource注释

17 Default属性（通过设置SpringApplication.setDefaultProperties指定）

假设您开发了一个使用name属性的@Component，如以下示例所示：

**import** org.springframework.stereotype.\*;  
**import** org.springframework.beans.factory.annotation.\*;

@Component  
**public class** MyBean {

@Value("${name}")  
**private** String name;  
*// ...*

}

在应用程序类路径上（例如，在jar中），您可以拥有一个application.properties文件，该文件为name提供合理的默认属性值。 在新环境中运行时，可以在jar外部提供覆盖name的application.properties文件。 对于一次性测试，您可以使用命令行开关启动（例如，java -jar app.jar --name =“Spring”）。

可以在命令行上使用环境变量提供SPRING\_APPLICATION\_JSON属性。 例如，您可以在UN \* X shell中使用以下行：

SPRING\_APPLICATION\_JSON='{"acme":{"name":"test"}}' java -jar myapp.jar

在前面的示例中，您最终在Spring环境中使用了acme.name = test。 您还可以在System属性：spring.application.json提供相关配置，如以下示例所示：

java -Dspring.application.json='{"name":"test"}' -jar myapp.jar

您还可以使用命令行参数提供JSON，如以下示例所示：

java -jar myapp.jar --spring.application.json='{"name":"test"}'

您还可以将JSON作为JNDI变量提供，如下所示：java:comp/env/ spring.application.json。

#### 24.1 配置随机值

RandomValuePropertySource对于注入随机值非常有用（例如，注入密钥或测试用例）。 它可以生成整数，长整数，uuids或字符串，如以下示例所示：

**my.secret**=${random.value}  
**my.number**=${random.int}  
**my.bignumber**=${random.long}  
**my.uuid**=${random.uuid}  
**my.number.less.than.ten**=${random.int(10)}  
**my.number.in.range**=${random.int[1024,65536]}

#### 24.2访问命令行属性

默认情况下，SpringApplication将任何命令行选项参数（即以 -- 开头的参数，例如--server.port = 9000）转换为属性，并将它们添加到Spring环境中。 如前所述，命令行属性始终优先于其他属性源。

如果您不希望将命令行属性添加到Spring环境中，可以使用SpringApplication.setAddCommandLineProperties（false）禁用它们。

#### 24.3 properties文件

SpringApplication从以下位置的application.properties文件加载属性，并将它们添加到Spring环境中：

1. 当前目录的/config子目录
2. 当前目录
3. 类路径的/config目录
4. 类路径

上面的列表按优先级排序（在列表中较高位置定义的属性将覆盖在较低位置定义的属性）

您还可以使用YAML文件替代'.properties'

如果您不喜欢application.properties作为配置文件名，则可以通过指定spring.config.name环境属性来切换到另一个文件名。 您还可以使用spring.config.location环境属性（以逗号分隔的目录位置或文件路径列表）来引用显式位置。 以下示例显示如何指定其他文件名：

java -jar myproject.jar --spring.config.name=myproject

以下示例显示如何指定两个位置：

java -jar myproject.jar --spring.config.location=classpath:/default.properties,classpath:/  
override.properties

spring.config.name和spring.config.location很早就用来确定哪些文件必须加载，因此必须将它们定义为环境属性（通常是系统环境变量，系统属性或命令行参数）。

如果spring.config.location包含目录（而不是文件），则它们应该以/结尾（并且在运行时，在加载之前附加从spring.config.name生成的名称，包括特定于配置文件的文件名）。 spring.config.location中指定的文件按原样使用，会被profile配置文件的属性覆盖。

以相反的顺序搜索配置位置。 默认情况下，配置的位置是classpath:/, classpath:/ config/,file:./,file:./config/。 生成的搜索顺序如下：

1. file:./config/
2. file:./
3. classpath:/config/
4. classpath:/

#### 24.4 profile属性

除application.properties文件外，还可以使用以下命名约定定义profile配置文件的属性：application-{profile}.properties。 环境具有一组默认配置文件（默认情况下为[default]），如果未设置活动profile，则使用这些配置文件。 换句话说，如果未显式激活任何配置文件，则会加载application-default.properties中的属性。

profile配置文件的属性从标准application.properties相同的位置加载，profile配置文件的文件始终覆盖非profile文件，无论profile配置文件的文件是在打包的jar内部还是外部。

如果指定了多个配置文件，则应用last-wins策略。 例如，spring.profiles.active属性指定的配置文件是在通过SpringApplication API配置的配置文件之后添加的，因此优先。

如果在spring.config.location中指定了任何文件，则不考虑这些文件的profile配置文件的变量。 如果您还想使用profile配置文件的属性，请使用spring.config.location指定目录。

#### 24.5 属性中的占位符

application.properties中的值在使用时通过现有环境进行过滤，因此您可以返回先前定义的值（例如，从系统属性）。

**app.name**=MyApp  
**app.description**=${app.name} is a Spring Boot application

您还可以使用此技术创建现有Spring Boot属性的“简短”变量。 有关details.properties的信息，请参见部分77.4，“使用'短'命令行参数”方法。

#### 24.6 加密属性

Spring Boot没有为加密属性值提供任何内置支持，但是，它确实提供了修改Spring环境中包含的值所必需的钩子点。 EnvironmentPostProcessor接口允许您在应用程序启动之前操作Environment。 有关详细信息，请参见第76.3节“在开始之前自定义环境或ApplicationContext”。

如果您正在寻找一种存储凭据和密码的安全方法，Spring Cloud Vault项目支持在HashiCorp Vault中存储外部化配置。

#### 24.7 使用YAML文件替换properties

YAML是JSON的超集，是用于指定分层配置数据的便捷格式。 只要在类路径上有SnakeYAML库，SpringApplication类就会自动支持YAML作为properties的替代。

如果使用“Starters”，则Spring-boot-starter会自动提供SnakeYAML。

**加载YAML**

Spring Framework提供了两个方便的类，可用于加载YAML文档。 YamlPropertiesFactoryBean将YAML加载为Properties，YamlMapFactoryBean将YAML加载为Map。

例如，请考虑以下YAML文档：

**environments**:

**dev**:

**url**: <http://dev.example.com>

**name**: Developer Setup

**prod**:

**url**: <http://another.example.com>

**name**: My Cool App

前面的示例将转换为以下属性：

**environments.dev.url**=http://dev.example.com  
**environments.dev.name**=Developer Setup  
**environments.prod.url**=http://another.example.com  
**environments.prod.name**=My Cool App

YAML列表被表示为带有[index]引用的属性键。 例如，考虑以下YAML：

**my**:

**servers**:

- dev.example.com

- another.example.com

前面的示例将转换为这些属性：

**my.servers[0]**=dev.example.com  
**my.servers[1]**=another.example.com

要使用Spring Boot的Binder工具（这是@ConfigurationProperties所做的）绑定上面的属性，你需要在目标bean有一个java.util.List（或Set）类型的属性，你需要提供一个setter 或者使用可变值初始化它。例如，以下示例绑定到前面显示的属性：

@ConfigurationProperties(prefix="my")  
**public class** Config {

**private** List<String> servers = **new** ArrayList<String>();

**public** List<String> getServers() {

**return this**.servers;  
}

}

**在Spring环境中将YAML转换为Properties**

YamlPropertySourceLoader类可用于在Spring环境中将YAML转换为PropertySource。 这样做可以让您使用带占位符语法的@Value注释访问YAML属性。

**多个profile YAML文档**

您可以使用spring.profiles键在同个文件中指定多个profile配置，以指示该文档何时应用，如以下示例所示：

**server**:

**address**: 192.168.1.100  
*---***spring**:  
 **profiles**: development  
**server**:  
 **address**: 127.0.0.1  
*---***spring**:  
 **profiles**: production & eu-central  
**server**:  
 **address**: 192.168.1.120

在前面的示例中，如果development处于活动状态，则server.address属性为127.0.0.1。 同样，如果production和eu-central配置文件处于活动状态，则server.address属性为192.168.1.120。 如果未启用开发，生产和eu-central配置文件，则该属性的值为192.168.1.100。

如果在应用程序上下文启动时没有显式激活profile，则激活默认profile。 因此，在以下YAML中，我们为spring.security.user.password设置了一个值，该值仅在“default”配置文件中可用

**server**:

**port**: 8000  
*---***spring**:  
 **profiles**: default  
 **security**:  
 **user**:  
 **password**: weak

然而，在以下示例中，始终设置密码，因为它未附加到任何profile文件，并且必须在必要时在所有其他profile文件中显式重置：

**server**:

**port**: 8000  
*---***spring**:  
 **security**:  
 **user**:  
 **password**: weak

使用spring.profiles元素指定不使用profile时，通过！字符。 如果为单个文档指定了否定和非否定的配置文件，则至少一个非否定的配置文件必须匹配，并且没有否定的配置文件可能匹配

**YAML的缺点**

无法使用@PropertySource注解加载YAML文件。 因此，如果您需要以这种方式加载值，则需要使用properties文件。

#### 24.8 类型安全的配置属性

使用@Value(“$ {property}”)注解来注入配置属性有时会很麻烦，特别是如果您正在使用多个属性或者您的数据本质上是分层的。 Spring Boot提供了一种使用属性的替代方法，该方法允许强类型bean管理和验证应用程序的配置，如以下示例所示：

**package** com.example;  
**import** java.net.InetAddress;  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.List;  
**import** org.springframework.boot.context.properties.ConfigurationProperties;  
@ConfigurationProperties("acme")  
**public class** AcmeProperties {

**private boolean** enabled;  
**private** InetAddress remoteAddress;  
**private final** Security security = **new** Security();  
**public boolean** isEnabled() { ... }  
**public void** setEnabled(**boolean** enabled) { ... }  
**public** InetAddress getRemoteAddress() { ... }  
**public void** setRemoteAddress(InetAddress remoteAddress) { ... }  
**public** Security getSecurity() { ... }  
**public static class** Security {

**private** String username;

**private** String password;

**private** List<String> roles = **new** ArrayList<>(Collections.singleton(***"USER"***));  
**public** String getUsername() { ... }  
**public void** setUsername(String username) { ... }  
**public** String getPassword() { ... }  
**public void** setPassword(String password) { ... }  
**public** List<String> getRoles() { ... }  
**public void** setRoles(List<String> roles) { ... }  
}

}

前面的POJO定义了以下属性：

acme.enabled，默认值为false。

acme.remote-address，具有可以从String强制转换的类型。

acme.security.username，带有嵌套的“security”对象，其名称由属性名称决定。 特别是，返回类型根本没有使用，可能是SecurityProperties

acme.security.password

acme.security.roles，带有String集合

getter和setter通常是必需的，因为绑定是通过标准的Java Beans属性描述符，就像在Spring MVC中一样。 在下列情况下可以省略setter：

map，只要它们被初始化，就需要一个getter但不一定是setter，因为它们可以被绑定器使用。

可以通过索引（通常使用YAML）或使用单个逗号分隔值（属性）访问集合和数组。 在后一种情况下，必须使用setter。

我们建议始终为此类型添加setter。 如果初始化集合，请确保它不是不可变的（如上例所示）。

如果初始化嵌套的POJO属性（如前面示例中的“安全性”字段），则不需要setter。 如果您希望绑定器通过使用其默认构造函数动态创建实例，则需要一个setter。

有些人使用Lombok自动添加getter和setter。 确保Lombok不为此类型生成任何特定构造函数，因为容器会自动使用它来实例化对象。

最后，仅考虑标准Java Bean属性，并且不支持对静态属性的绑定。

另请参阅@Value和@ConfigurationProperties之间的差异。

您还需要列出要在@EnableConfigurationProperties注释中注册的属性类，如以下示例所示：

@Configuration  
@EnableConfigurationProperties(AcmeProperties.class)  
**public class** MyConfiguration {  
}

当以这种方式注册@ConfigurationProperties bean时，bean具有常规名称：<prefix> - <fqn>，其中<prefix>是@ConfigurationProperties注释中指定的环境键前缀，<fqn>是bean的完全限定名称。 如果注释未提供任何前缀，则仅使用bean的完全限定名称。

上例中的bean名称是acme-com.example.AcmeProperties。

即使前面的配置为AcmeProperties创建了一个常规bean，我们也建议@ConfigurationProperties只处理环境，特别是不从上下文中注入其他bean。 话虽如此，@ EnableConfigurationProperties注释也会自动应用于您的项目，以便从Environment配置任何使用@ConfigurationProperties注释的现有bean。 您可以通过确保AcmeProperties已经是一个bean来快捷方式MyConfiguration，如以下示例所示：

@Component  
@ConfigurationProperties(prefix="acme")  
**public class** AcmeProperties {  
*// ... see the preceding example*}

这种配置风格特别适用于SpringApplication外部YAML配置，如以下示例所示：

*# application.yml***acme**:  
 **remote-address**: 192.168.1.1  
 **security**:  
 **username**: admin  
 **roles**:  
 - USER  
 - ADMIN  
*# additional configuration as required*

要使用@ConfigurationProperties bean，可以使用与任何其他bean相同的方式注入它们，如以下示例所示：

@Service  
**public class** MyService {

**private final** AcmeProperties properties;  
@Autowired  
**public** MyService(AcmeProperties properties) {

**this**.properties = properties;  
}  
*//...*@PostConstruct  
**public void** openConnection() {

Server server = **new** Server(**this**.properties.getRemoteAddress());  
*// ...*}

}

使用@ConfigurationProperties还可以生成元数据文件，IDE可以使用这些文件为您自己的键提供自动完成功能。 有关详细信息，请参阅附录B，配置元数据附录。

**第三方配置**

除了使用@ConfigurationProperties来注释类之外，您还可以在公共@Bean方法上使用它。 当您想要将属性绑定到控件之外的第三方组件时，这样做会特别有用。

要从Environment属性配置bean，请将@ConfigurationProperties添加到其bean注册中，如以下示例所示

@ConfigurationProperties(prefix = "another")  
@Bean  
**public** AnotherComponent anotherComponent() {  
...  
}

使用另一个前缀定义的任何属性都以与前面的AcmeProperties示例类似的方式映射到该AnotherComponent bean。

**轻松绑定**

Spring Boot使用一些宽松的规则将Environment属性绑定到@ConfigurationProperties bean，因此不需要在Environment属性名和bean属性名之间进行精确匹配。 这有用的常见示例包括破折号分隔的环境属性（例如，context-path绑定到contextPath）和大写环境属性（例如，PORT绑定到端口）。

例如，请考虑以下@ConfigurationProperties类：

@ConfigurationProperties(prefix="acme.my-project.person")  
**public class** OwnerProperties {

**private** String firstName;

**public** String getFirstName() {

**return this**.firstName;  
}  
**public void** setFirstName(String firstName) {

**this**.firstName = firstName;  
}

}

在前面的示例中，可以匹配到以下属性名称：

acme.my-project.person.first-name

acme.my-project.person.firstName

acme.my-project.person.first\_name

ACME\_MYPROJECT\_PERSON\_FIRSTNAME

注释的前缀值必须是kebab大小写（小写并用 - 分隔，例如acme.my-project.person）。

**合并复杂类型**

当列表在多个位置配置时，后面定义的列表会覆盖前面定义的列表。

例如，假设具有名称和描述属性的MyPojo对象默认为null。 以下示例公开了AcmeProperties中的MyPojo对象列表：

@ConfigurationProperties("acme")  
**public class** AcmeProperties {

**private final** List<MyPojo> list = **new** ArrayList<>();  
**public** List<MyPojo> getList() {

**return this**.list;  
}

}

考虑下面的配置：

**acme**:

**list**:

**- name**: my name

**description**: my description

*---*

**spring**:

**profiles**: dev

**acme**:

**list**:

**- name**: my another name

如果dev配置文件未激活，则AcmeProperties.list包含一个MyPojo条目，如前所述。 但是，如果启用了dev配置文件，则列表仍然只包含一个条目。 此配置不会向列表中添加第二个MyPojo实例，也不会合并项目

在多个配置文件中指定List时，将使用具有最高优先级（并且只有一个）的列表。 请考虑以下示例：

**acme**:

**list**:

**- name**: my name

**description**: my description

**- name**: another name

**description**: another description  
*---***spring**:

**profiles**: dev  
**acme**:

**list**:

**- name**: my another name

在前面的示例中，如果dev配置文件处于激活状态，则AcmeProperties.list包含一个MyPojo条目。 对于YAML，逗号分隔列表和YAML列表都可用于完全覆盖列表的内容。

对于Map属性，您可以绑定从多个源中提取的属性值。 但是，对于多个源中的相同属性，使用具有最高优先级的属性。 以下示例从AcmeProperties公开Map <String，MyPojo>：

@ConfigurationProperties("acme")  
**public class** AcmeProperties {

**private final** Map<String, MyPojo> map = **new** HashMap<>();

**public** Map<String, MyPojo> getMap() {

**return this**.map;  
}

}

考虑下面的配置：

**acme**:

**map**:

**key1**:

**name**: my name 1

**description**: my description 1  
*---***spring**:

**profiles**: dev  
**acme**:

**map**:

**key1**:

**name**: dev name 1

**key2**:

**name**: dev name 2

**description**: dev description 2

如果dev配置文件未激活，则AcmeProperties.map包含一个带键key1的属性。 但是，如果启用了dev配置文件，则map包含两个条目，其中键key1（名称为dev name 1和我的描述1的描述）和key2（名称为dev name 2和dev description 2的描述）

前面的合并规则适用于所有属性源的属性，而不仅仅是YAML文件。

**属性转换**

当Spring绑定到@ConfigurationProperties bean时，Spring Boot会尝试将外部应用程序属性强制转换为正确的类型。如果需要自定义类型转换，可以提供ConversionService bean（带有名为conversionService的bean）或自定义属性编辑器（通过CustomEditorConfigurer bean）或自定义转换器（带有注释为@ConfigurationPropertiesBinding的bean定义）。

由于在应用程序生命周期中很早就请求了此bean，因此请确保限制ConversionService正在使用的依赖项。 通常，您在创建时可能无法完全初始化所需的任何依赖项。 如果配置密钥强制不需要，您可能希望重命名自定义ConversionService，并且只依赖于使用@ConfigurationPropertiesBinding限定的自定义转换器。

**@ConfigurationProperties验证**

只要使用Spring的@Validated注释注释，Spring Boot就会尝试验证@ConfigurationProperties类。 您可以直接在配置类上使用JSR-303 javax.validation约束注释。 为此，请确保符合条件的JSR-303实现位于类路径上，然后将约束注释添加到字段中，如以下示例所示：

@ConfigurationProperties(prefix="acme")

@Validated  
**public class** AcmeProperties {

@NotNull

**private** InetAddress remoteAddress;  
*// ... getters and setters*

}

您还可以通过注释使用@Validated创建配置属性的@Bean方法来触发验证。

虽然绑定时也会验证嵌套属性，但最好还是将关联字段注释为@Valid。 这可确保即使未找到嵌套属性也会触发验证。 以下示例基于前面的AcmeProperties示例构建：

@ConfigurationProperties(prefix="acme")  
@Validated  
**public class** AcmeProperties {

@NotNull

**private** InetAddress remoteAddress;

@Valid

**private final** Security security = **new** Security();

*// ... getters and setters*

**public static class** Security {

@NotEmpty

**public** String username;

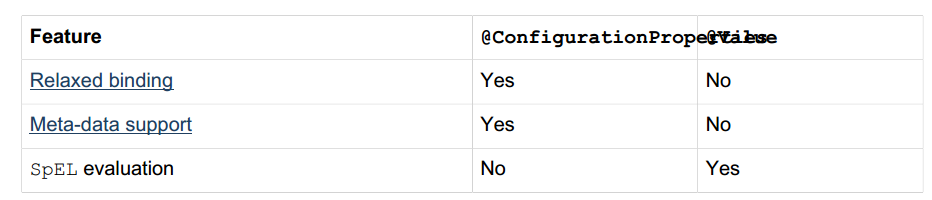
*// ... getters and setters*}

}

您还可以通过创建名为configurationPropertiesValidator的bean定义来添加自定义Spring Validator。 应该将@Bean方法声明为static。 配置属性验证器是在应用程序生命周期的早期创建的，并且将@Bean方法声明为static可以创建bean而无需实例化@Configuration类。 这样做可以避免早期实例化可能导致的任何问题。 有一个属性验证示例，显示如何设置。

**@ConfigurationProperties和@Value**

@Value注释是核心容器功能，它不提供与类型安全配置属性相同的功能。 下表总结了@ConfigurationProperties和@Value支持的功能：



如果为自己的组件定义一组配置键，我们建议您将它们分组到使用@ConfigurationProperties注释的POJO中。 您还应该知道，因为@Value不支持宽松绑定，所以如果您需要使用环境变量来提供值，则它不是一个好的候选者

最后，虽然您可以在@Value中编写SpEL表达式，但不会从应用程序属性文件处理此类表达式。

### 25 Profiles

Spring Profiles提供了一种隔离应用程序配置并使其仅在特定环境中可用的方法。 可以使用@Profile标记任何@Component或@Configuration以限制何时加载它，如以下示例所示：

@Configuration  
@Profile("production")  
**public class** ProductionConfiguration {  
*// ...*}

您可以使用spring.profiles.active 环境属性指定哪些配置文件处于活动状态。 您可以使用本章前面介绍的任何方法指定属性。 例如，您可以将其包含在application.properties中，如以下示例所示：

**spring.profiles.active**=dev,hsqldb

您还可以使用以下开关在命令行上指定它： --spring.profiles.active = dev,hsqldb。

#### 25.1 添加额外的profile

spring.profiles.active属性遵循与其他属性相同的排序规则：优先级最高的为PropertySource。 这意味着您可以在application.properties中指定活动配置文件，然后使用命令行开关替换它们。

有时，将特定profile的属性添加到活动配置文件而不是替换它们是有用的。 spring.profiles.include属性可用于无条件地添加活动配置文件。 SpringApplication入口点还有一个Java API，用于设置其他配置文件（即，在spring.profiles.active属性激活的配置文件之上）。 见SpringApplication中的setAdditionalProfiles（）方法。

例如，当使用开关--spring.profiles.active = prod运行具有以下属性的应用程序时，proddb和prodmq配置文件也会被激活：

*---***my.property**: fromyamlfile  
*---***spring.profiles**: prod  
**spring.profiles.include**:

- proddb

- prodmq

请记住，可以在YAML文档中定义spring.profiles属性，以确定此特定文档何时包含在配置中。 有关更多详细信息，请参见第77.7节“根据环境更改配置”。

#### 25.2 编程方式设置profile

您可以通过编程方式设置profile：SpringApplication.setAdditionalProfiles（...）。 也可以使用Spring的ConfigurableEnvironment接口激活配置文件。

#### 25.3 特定profile配置

application.properties（或application.yml）的特定于配置文件的变量和通过@ConfigurationProperties引用的文件被视为配置文件并加载。 有关详细信息，请参见“第24.4节：特定于配置文件的属性”。

### 26 日志记录

Spring Boot使用Commons Logging进行所有内部日志记录，但保留底层日志实现。 为Java Util Logging，Log4J2和Logback提供了默认配置。 在每种情况下，记录器都预先配置为使用控制台输出，并且还提供可选的文件输出。

默认情况下，如果使用“Starters”，则使用Logback进行日志记录。 还包括适当的Logback路由，以确保使用Java Util Logging，Commons Logging，Log4J或SLF4J的依赖库都能正常工作。

Java有很多日志框架可供使用。 如果以上列表看起来令人困惑，请不要担心。 通常，您不需要更改日志记录依赖项和Spring Boot默认能正常工作。

#### 26.1 日志格式

## 第五部分：SpringBoot Actuator 可用于生产的功能

Spring Boot包含许多其他功能，可帮助您在将应用程序部署到生产时监视和管理应用程序。 您可以选择使用HTTP端点或JMX来管理和监视应用程序。 审计，运行状况和指标收集会自动应用于您的应用程序。

### 52 开启生产就绪特性

spring-boot-actuator模块提供Spring Boot的所有生产就绪功能。 启用这些功能的最简单方法是将依赖项：spring-boot-starter-actuator添加到项目中：

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency></dependencies>

### 53 端点 Endpoints

通过Actuator端点，可以监控应用程序并与之交互。 Spring Boot包含许多内置端点，并且允许添加自己的端点。 例如：health端点提供应用程序基本的运行状况信息。

每个端点都可以单独的启用或禁用。控制该端点是否被创建并且其对应的bean是否存在于应用程序上下文中。 要远程访问，必须通过JMX或HTTP公开端点。 大多数应用程序选择HTTP，其中端点的ID会被映射成访问URL。 例如，默认情况下，health端点映射到/health。

默认情况下，以下端点是可用的：

|  |  |  |
| --- | --- | --- |
| **ID** | **描述** | **默认开启** |
| auditevents | 暴露当前应用程序的审核事件信息 | true |
| beans | 显示应用程序中所有Spring bean的完整列表。 | true |
| configprops | 显示所有@ConfigurationProperties的整理列表 | true |
| env | 公开Spring的ConfigurableEnvironment中的属性。 | true |
| health | 显示应用健康信息 | true |
| info | 显示info配置的信息 | true |
| loggers | 显示应用程序中Logger的配置。 | true |
| metrics | 显示应用的指标信息，如内存等 | true |
| mappings | 显示所有@RequestMapping路径的整理列表。 | true |
| shutdown | 允许应用程序正常关闭。 | No |

#### 53.1 启用端点

默认情况下，除了shutdown端点，其他端点都是启用的。 要配置端点的启用，请使用其endpoints.<id> .enabled属性。 以下示例启用shutdown端点：

**endpoints**.**shutdown**.**enabled**=**true**

想默认关闭所有端点：**endpoints.enabled=false**

## 第八部分：构建工具插件(Build Tool Plugins)

Spring Boot为Maven和Gradle提供构建工具插件。 这些插件提供了各种功能，包括打包成可执行jar。 本节提供了有关这两个插件的更多详细信息，以及扩展不受支持的构建系统时的一些帮助。 如果您刚刚开始，可能需要先阅读“第三部分 使用Spring Boot”中的“第13章，构建系统”。

### 71 Spring Boot Maven插件

Spring Boot Maven插件在Maven中提供Spring Boot支持，允许您打包可执行jar或war并“就地”运行应用程序。 要使用它，您必须使用Maven 3.2（或更高版本）。

#### 71.1 集成插件

要使用Spring Boot Maven插件，需要在pom.xml的plugins部分中包含相应的XML片段，如以下示例所示：

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<version>2.1.1.RELEASE</version>

<executions>

<execution>

<goals>

<goal>repackage</goal></goals>

</execution></executions>

</plugin>

remark：2.1.1.RELEASE版本最低要求需要Maven 2.0、JDK 1.8

上述配置会重新打包在Maven生命周期的打包阶段构建的jar或war。

如果不包含<execution />配置，如前面的示例所示，您可以单独运行插件（但仅在package阶段），如以下示例所示：

mvn package spring-boot:repackage

如果需要使用SpringBoot Maven插件的里程碑或快照版本，则还需要添加相应的pluginRepository元素：

<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>

#### 71.2打包成可执行的jar或war文件

一旦spring-boot-maven-plugin包含在你的pom.xml中，它就会自动尝试使用spring-boot：repackage目标重写生成文件以使它们可执行。 您应该使用<packaging>元素将项目配置为构建jar或war如以下示例所示：

<packaging>jar</packaging>

在package阶段，Spring Boot会增强现有jar/war。 可以通过使用配置选项或向manifest清单添加Main-Class属性来指定要启动的主类。 如果未指定主类，则插件将搜索具有public static void main（String [] args）方法的类。

可以通过下面的命令来构建和执行一个项目构件：

mvn package

java –jar target/mymodule-version.jar

要构建可执行且可部署到外部容器的war文件，需要将嵌入式容器依赖项标记为“provided”，如以所示：

**<dependencies>**

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-web**</artifactId>  
</dependency>  
<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-tomcat**</artifactId>**

**<scope>**provided**</scope>  
</dependency>***<!-- ... -->*

**</dependencies>**

有关如何创建可部署war文件的更多详细信息，请参阅“第92.1节 创建可部署的war文件”部分。

更多的高级配置选项和示例请参见[插件的信息页面](https://docs.spring.io/spring-boot/docs/2.1.1.RELEASE/maven-plugin)

#### 71.3 目标概述 Goals Overview

SpringBoot插件拥有下面的目标：

spring-boot:run 运行SpringBoot应用

spring-boot:repackage 重新打包jar/war，使其可执行

spring-boot:start/spring-boot:stop 管理SpringBoot应用程序的生命周期

spring-boot:build-info 生成可由Actuator使用的构建信息

## 第九部分：How to指南

本节提供了一些使用Spring Boot时经常出现的 “我该怎么做......”问题的答案。 没有进行全面覆盖，但确实涵盖了很多。

如果您遇到未在此处讨论的特定问题，您可能需要查看stackoverflow.com是否有人已提供答案。 这也是提出新问题的好地方（请使用spring-boot标签）。

我们也非常乐意扩展这一部分。 如果您想添加“how to”，请向我们发送拉取请求。