Spring Boot 参考指南

作者

Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, Vedran Pavić, Jay Bryant, Madhura Bhave

2.1.1.RELEASE

目录

第一部分。Spring Boot 文件

本节简要概述了 Spring Boot 参考文档。它用作文档其余部分的映射。

1.关于文档

Spring Boot 参考指南可用作

- HTML
- PDF
- EPUB

最新的副本可在 docs.spring.io/spring-boot/docs/current/reference 上找到。

2.获得帮助

如果您在使用 Spring Boot 时遇到问题，我们很乐意为您提供帮助。

- 尝试使用方法文档。他们为最常见的问题提供解决方案。
- 了解 Spring 基础知识。Spring Boot 以许多其他 Spring 项目为基础。查看 spring.io 网站以获取大量参考文档。如果您从 Spring 开始，请尝试其中一个指南。
- 问一个问题。我们监控 stackoverflow.com 以获取标记的问题 `spring-boot`。
- 在 github.com/spring-projects/spring-boot/issues 上使用 Spring Boot 报告错误。

注意

所有 Spring Boot 都是开源的，包括文档。如果您发现文档存在问题或想要改进它们，请参与其中。

3.第一步

如果您开始使用 Spring Boot 或“Spring”，请从以下主题开始 ：

- **从头开始：** 概述 | 要求 | 安装

- **教程：** 第 1 部分 | 第 2 部分
- **运行你的例子：** 第 1 部分 | 第 2 部分

4.使用 Spring Boot

准备开始使用 Spring Boot？我们为您提供：

- **构建系统：** Maven | Gradle | Ant | Starters
- **最佳实践：** 代码结构 | @Configuration | @EnableAutoConfiguration | Beans 和依赖注入
- **运行代码** IDE | 打包 | Maven | Gradle
- **打包您的应用程序：** 生产罐
- **Spring Boot CLI：** 使用 CLI

5.了解 Spring Boot 功能

需要有关 Spring Boot 核心功能的更多详细信息？ 以下内容适合您：

- **核心功能：** SpringApplication | 外部配置 | 个人资料 | 记录
- **Web 应用程序：** MVC | 嵌入式容器
- **使用数据：** SQL | NO-SQL
- **消息：** 概述 | JMS
- **测试：** 概述 | 启动应用程序 | utils 的
- **扩展：** 自动配置 | @条件

6.转向生产

当您准备将 Spring Boot 应用程序推向生产时，我们有 一些您可能喜欢的技巧：

- **管理端点：** 概述 | 定制
- **连接选项：** HTTP | JMX
- **监控：** 指标 | 审计 | 追踪 | 处理

7.高级主题

最后，我们为更高级的用户提供了一些主题：

- **Spring Boot 应用程序部署：** 云部署 | OS 服务
- **构建工具插件：** Maven | Gradle
- **附录：** 应用程序属性 | 自动配置类 | 可执行的罐子

第二部分。入门

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“什么？”，“如何？”和“为什么？”的问题。它包括对 Spring Boot 的介绍以及安装说明。然后，我们将引导您构建您的第一个 Spring Boot 应用程序，并在我们讨论时讨论一些核心原则。

8.介绍 Spring Boot

Spring Boot 可以轻松创建可以运行的独立的，基于生产级 Spring 的应用程序。我们对 Spring 平台和第三方库进行了自以为是的观点，以便您可以尽量少开始。大多数 Spring Boot 应用程序需要非

常少的 Spring 配置。

您可以使用 Spring Boot 创建可以使用 `java -jar` 或更多传统战争部署启动的 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可广泛访问的入门体验。
- 开箱即用，但随着需求开始偏离默认值而迅速摆脱困境。
- 提供大型项目（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）通用的一系列非功能性功能。
- 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 并且与 Java 11 兼容（包括在内）。Spring 框架 5.1.3.RELEASE 或以上也是必需的。

为以下构建工具提供了显式构建支持：

构建工具	版
Maven	3.3+
Gradle	4.4+

9.1 Servlet 容器

Spring Boot 支持以下嵌入式 servlet 容器：

名称	Servlet 版本
Tomcat 9.0	4.0
Jetty 9.4	3.1
Undertow 2.0	4.0

您还可以将 Spring Boot 应用程序部署到任何 Servlet 3.1+兼容容器。

10.安装 Spring Boot

Spring Boot 可以与“经典”Java 开发工具一起使用，也可以作为命令行工具安装。无论哪种方式，您都需要 Java SDK v1.8 或更高版本。在开始之前，您应该使用以下命令检查当前的 Java 安装：

```
$ java -version
```

如果您不熟悉 Java 开发，或者想要试用 Spring Boot，则可能需要先尝试 Spring Boot CLI（命令行界面）。否则，请继续阅读“经典”安装说明。

10.1 Java Developer 的安装说明

您可以像使用任何标准 Java 库一样使用 Spring Boot。为此，请在类路径中包含相应的 `spring-boot-*.jar` 文件。Spring Boot 不需要任何特殊工具集成，因此您可以使用任何 IDE 或文本编辑器。此外，Spring Boot 应用程序没有什么特别之处，因此您可以像运行任何其他 Java 程序一样运行和调试 Spring 引导应用程序。

虽然您可以复制 Spring Boot 罐，但我们通常建议您使用支持依赖关系管理的构建工具（例如 Maven 或 Gradle）。

10.1.1 Maven 安装

Spring Boot 与 Apache Maven 3.3 或更高版本兼容。如果您尚未安装 Maven，则可以按照 maven.apache.org 上的说明进行操作。

小费

在许多操作系统上，Maven 可以与软件包管理器一起安装。如果您使用 OSX Homebrew，请尝试 `brew install maven`。Ubuntu 用户可以运行 `sudo apt-get install maven`。使用 Chocolatey 的 Windows 用户可以从提升（管理员）提示符运行 `choco install maven`。

Spring Boot 依赖项使用 `org.springframework.boot groupId`。通常，您的 Maven POM 文件继承自 `spring-boot-starter-parent` 项目，并声明对一个或多个“Starters”的依赖关系。Spring Boot 还提供了一个可选的 Maven 插件来创建可执行 jar。

以下清单显示了典型的 `pom.xml` 文件：

```
<?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>

        <!-- Inherit defaults from Spring Boot -->
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.1.1.RELEASE</version>
        </parent>

        <!-- Add typical dependencies for a web application -->
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>
        </dependencies>

        <!-- Package as an executable jar -->
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
```

```
            </plugins>
        </build>

</project>
```

小费

`spring-boot-starter-parent` 是使用 Spring Boot 的好方法，但它可能并不适合所有时间。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，请参见 第 13.2.2 节“使用不带父 POM 的 Spring Boot”作为使用 `import` 范围的替代解决方案。

10.1.2 Gradle 安装

Spring Boot 与 Gradle 4.4 及更高版本兼容。如果您尚未安装 Gradle，则可以按照 gradle.org 上的说明进行操作。

可以使用 `org.springframework.boot` `group` 声明 Spring Boot 依赖项。通常，您的项目会声明对一个或多个“Starters”的依赖关系 。Spring Boot 提供了一个有用的 Gradle 插件，可用于简化依赖声明和创建可执行 jar。

Gradle Wrapper

当您需要构建项目时，Gradle Wrapper 提供了一种“获取”Gradle 的好方法。它是一个小脚本和库，您可以与代码一起提交以引导构建过程。有关详细信息，请参阅 docs.gradle.org/4.2.1/userguide/gradle_wrapper.html。

有关 Spring Boot 和 Gradle 入门的更多详细信息 ，请参阅 Gradle 插件参考指南的“ 入门”部分。

10.2 安装 Spring Boot CLI

Spring Boot CLI（命令行界面）是一个命令行工具，可用于使用 Spring 快速原型。它允许您运行 Groovy 脚本，这意味着您拥有熟悉的类似 Java 的语法，而没有太多的样板代码。

您不需要使用 CLI 来使用 Spring Boot，但它绝对是实现 Spring 应用程序的最快方法。

10.2.1 手动安装

您可以从 Spring 软件存储库下载 Spring CLI 分发版：

- spring-boot-cli-2.1.1.RELEASE-bin.zip
- spring-boot-cli-2.1.1.RELEASE-bin.tar.gz

还提供最先进的 快照分发。

下载完成后，请按照 解压缩的存档中的 INSTALL.txt 说明进行操作。总之，在 `.zip` 文件的 `bin/` 目录中有一个 `spring` 脚本（Windows 为 `spring.bat`）。或者，您可以将 `java -jar` 与 `.jar` 文件一起使用（该脚本可帮助您确保正确设置类路径）。

10.2.2 使用 SDKMAN 安装！

SDKMAN！（软件开发工具包管理器）可用于管理各种二进制 SDK 的多个版本，包括 Groovy 和 Spring Boot CLI。获取 SDKMAN！从 sdkman.io 并使用以下命令安装 Spring Boot：

```
$ sdk install springboot
```

```
$ spring --version
Spring Boot v2.1.1.RELEASE
```

如果您为 CLI 开发功能并希望轻松访问您构建的版本，请使用以下命令：

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-
cli-2.1.1.RELEASE-bin/spring-2.1.1.RELEASE/
$ sdk default springboot dev
$ spring --version
Spring CLI v2.1.1.RELEASE
```

前面的说明安装了一个名为 `dev` 实例的 `spring` 本地实例。它指向您的目标构建位置，因此每次重建 Spring Boot 时，`spring` 都是最新的。

您可以通过运行以下命令来查看它：

```
$ sdk ls springboot

================================================================================
Available Springboot Versions
================================================================================
> + dev
* 2.1.1.RELEASE

================================================================================
+ - local version
* - installed
> - currently in use
================================================================================
```

10.2.3 OSX Homebrew 安装

如果您使用的是 Mac 并使用 Homebrew，则可以使用以下命令安装 Spring Boot CLI：

```
$ brew tap pivotal/tap
$ brew install springboot
```

Homebrew 将 `spring` 安装到`/usr/local/bin`。

注意

如果您没有看到公式，那么您的 brew 安装可能已过时。在这种情况下，运行 `brew update` 并再试一次。

10.2.4 MacPorts 安装

如果您使用的是 Mac 并使用 MacPorts，则可以使用以下命令安装 Spring Boot CLI：

```
$ sudo port install spring-boot-cli
```

10.2.5 命令行完成

Spring Boot CLI 包括为 BASH 和 zsh shell 提供命令完成的脚本 。您可以在任何 shell 中 `source` 脚本（也称为 `spring`）或将其放入您的个人或系统范围的 bash 完成初始化中。在 Debian 系统上，系统范围的脚本位于`/shell-completion/bash`，并且当新 shell 启动时，该目录中的所有脚本都将执行。例如，要使用 SDKMAN！安装，请手动运行脚本，请使用以下命令：

```
$ . ~/.sdkman/candidates/springboot/current/shell-completion/bash/spring
$ spring <HIT TAB HERE>
```

```
  grab  help  jar  run  test  version
```

注意

如果使用 Homebrew 或 MacPorts 安装 Spring Boot CLI，则命令行完成脚本将自动注册到 shell。

10.2.6 Windows Scoop 安装

如果您使用的是 Windows 并使用 Scoop，则可以使用以下命令安装 Spring Boot CLI：

```
> scoop bucket add extras
> scoop install springboot
```

Scoop 安装 `spring` 到`~/scoop/apps/springboot/current/bin`。

注意

如果您没有看到应用清单，则您的 scoop 安装可能已过时。在这种情况下，请运行 `scoop update` 并重试。

10.2.7 快速启动 Spring CLI 示例

您可以使用以下 Web 应用程序来测试您的安装。首先，创建一个名为 `app.groovy` 的文件，如下所示：

```
@RestController
class ThisWillActuallyRun {

        @RequestMapping("/")
        String home() {
                "Hello World!"
        }

}
```

然后从 shell 运行它，如下所示：

```
$ spring run app.groovy
```

注意

随着依赖项的下载，应用程序的第一次运行速度很慢。后续运行要快得多。

在您喜欢的网络浏览器中打开 `localhost:8080`。您应该看到以下输出：

```
Hello World!
```

10.3 从早期版本的 Spring Boot 升级

如果要从早期版本的 Spring Boot 升级，请查看 项目 Wiki 上的“迁移指南”， 其中提供了详细的升级说明。另请查看 “发行说明”，了解每个版本的“新的和值得注意的”功能列表。

升级到新功能版本时，某些属性可能已重命名或删除。Spring Boot 提供了一种在启动时分析应用程序环境和打印诊断的方法，还可以在运行时临时迁移属性。要启用该功能，请将以下依赖项添

加到项目中：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-properties-migrator</artifactId>
        <scope>runtime</scope>
</dependency>
```

警告

添加到环境后期的属性（例如使用`@PropertySource`时）将不会被考虑在内。

注意

完成迁移后，请确保从项目的依赖项中删除此模块。

要升级现有 CLI 安装，请使用相应的软件包管理器命令（例如，`brew upgrade`），或者，如果手动安装 CLI，请按照 标准说明操作，记住更新 `PATH` 环境变量以删除任何旧版本引用。

11.开发您的第一个 Spring Boot 应用程序

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。

小费

该 spring.io 网站包含了许多“入门” 指导在使用 Spring Boot。如果您需要解决特定问题，请先检查一下。

您可以通过转到 start.spring.io 并从依赖关系搜索器中选择“Web”启动器来快捷执行以下步骤。这样做会生成一个新的项目结构，以便您可以立即开始编码。查看 Spring Initializr 文档以获取更多详细信息。

在开始之前，打开终端并运行以下命令以确保您安装了有效版本的 Java 和 Maven：

```
$ java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)

$ mvn -v
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-17T14:33:14-
04:00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_102, vendor: Oracle Corporation
```

注意

此示例需要在其自己的文件夹中创建。后续说明假定您已创建合适的文件夹，并且它是您当前的目录。

11.1 创建 POM

我们需要先创建一个 Maven `pom.xml` 文件。`pom.xml` 是用于构建项目的配方。打开您喜欢的文

本编辑器并添加以下内容：

```
<?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>
```

上面的清单应该为您提供有效的构建。您可以通过运行 `mvn package` 来测试它（现在，您可以忽略“jar 将为空 - 没有内容被标记为包含！”警告）。

注意

此时，您可以将项目导入 IDE（大多数现代 Java IDE 包含对 Maven 的内置支持）。为简单起见，我们继续为此示例使用纯文本编辑器。

11.2 添加类路径依赖项

Spring Boot 提供了一些“Starters”，可让您将 jar 添加到类路径中。我们的示例应用程序已经在 POM 的 `parent` 部分使用了 `spring-boot-starter-parent`。`spring-boot-starter-parent` 是一个特殊的启动器，提供有用的 Maven 默认值。它还提供了一个 `dependency-management` 部分，以便您可以省略 `version` 标签中的“祝福”依赖项。

其他“Starters”提供了在开发特定类型的应用程序时可能需要的依赖项。由于我们正在开发 Web 应用程序，因此我们添加了 `spring-boot-starter-web` 依赖项。在此之前，我们可以通过运行以下命令来查看当前的内容：

```
$ mvn dependency:tree

[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT
```

`mvn dependency:tree` 命令打印项目依赖项的树表示。您可以看到 `spring-boot-starter-parent` 本身不提供依赖关系。要添加必要的依赖项，请编辑 `pom.xml` 并在 `parent` 部分下方添加 `spring-boot-starter-web` 依赖项：

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
</dependencies>
```

如果再次运行 `mvn dependency:tree`，您会发现现在有许多其他依赖项，包括 Tomcat Web 服务器和 Spring Boot 本身。

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 编译来自 `src/main/java` 的源，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);
        }

}
```

虽然这里的代码不多，但还是有很多代码。我们将在接下来的几节中逐步介绍重要部分。

11.3.1 @RestController 和@RequestMapping Annotations

`Example` 类的第一个注释是`@RestController`。这被称为 构造型注释。它为阅读代码的人提供了提示，并为 Spring 提供了该类扮演特定角色的提示。在这种情况下，我们的类是一个 web `@Controller`，所以 Spring 在处理传入的 Web 请求时会考虑它。

`@RequestMapping` 注释提供“路由”信息。它告诉 Spring 任何带有`/`路径的 HTTP 请求都应该映射到 `home` 方法。`@RestController` 注释告诉 Spring 将结果字符串直接呈现给调用者。

小费

`@RestController` 和`@RequestMapping` 注释是 Spring MVC 注释。（它们不是 Spring Boot 特有的。）有关详细信息，请参阅 Spring 参考文档中的 MVC 部分。

11.3.2 @EnableAutoConfiguration 注释

第二个类级别注释是`@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“Starters”配合使用，但这两个概念并不直接相关。您可以自由选择并在首发之外选择 jar 依赖项。Spring Boot 仍然尽力自动配置您的应用程序。

11.3.3“主要”方法

我们的应用程序的最后一部分是 `main` 方法。这只是遵循应用程序入口点的 Java 约定的标准方法。我们的主要方法是通过调用 `run` 来委托 Spring Boot 的 `SpringApplication` 类。`SpringApplication` 引导我们的应用程序，从 Spring 开始，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` 哪个是主要的 Spring 组件。还会传递 `args` 数组以公开任何命令行参数。

11.4 运行示例

此时，您的应用程序应该工作。由于您使用了 `spring-boot-starter-parent` POM，因此您可以使用有用的 `run` 目标来启动应用程序。从根项目目录中键入 `mvn spring-boot:run` 以启动应用程序。您应该看到类似于以下内容的输出：

```
$ mvn spring-boot:run

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v2.1.1.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

如果您打开 Web 浏览器到 localhost:8080，您应该看到以下输出：

```
Hello World!
```

要正常退出应用程序，请按 `ctrl-c`。

11.5 创建一个可执行的 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。

可执行的 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”罐子。uber 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。如果在多个罐子中使用相同的文件名（但具有不同的内容），也可能会有问题。

Spring Boot 采用不同的方法，让你直接嵌套罐子。

要创建可执行 jar，我们需要将 `spring-boot-maven-plugin` 添加到 `pom.xml`。为此，请在 `dependencies` 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>
```

注意

`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。有关详细信息，请参阅插件文档。

保存 `pom.xml` 并从命令行运行 `mvn package`，如下所示：

```
$ mvn package

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] .... ..
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---
[INFO] Building jar: /Users/developer/example/spring-boot-example/target/myproject-
0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.1.1.RELEASE:repackage (default) @ myproject ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

如果你查看 `target` 目录，你应该看到 `myproject-0.0.1-SNAPSHOT.jar`。该文件大小应为 10 MB 左右。如果要查看内部，可以使用 `jar tvf`，如下所示：

```
$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar
```

您还应该在 `target` 目录中看到一个名为 `myproject-0.0.1-SNAPSHOT.jar.original` 的小文件。这是 Maven 在 Spring Boot 重新打包之前创建的原始 jar 文件。

要运行该应用程序，请使用 `java -jar` 命令，如下所示：

```
$ java -jar target/myproject-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v2.1.1.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864)
```

和以前一样，要退出应用程序，请按 `ctrl-c`。

12.接下来要阅读的内容

希望本节提供了一些 Spring Boot 基础知识，让您开始编写自己的应用程序。如果您是面向任务的开发人员类型，您可能希望跳转到 spring.io 并查看一些 入门指南，这些指南解决了特定的“我如何使用 Spring？”问题。我们还有 Spring Boot 特定的“ 操作方法 ”参考文档。

该 Spring Boot 库也有 一堆样品可以运行。样本独立于其余代码（也就是说，您无需构建其余代码来运行或使用示例）。

否则，下一个逻辑步骤是阅读第 *III* 部分“使用 *Spring Boot*”。如果你真的很不耐烦，你也可以跳过去阅读 *Spring Boot* 功能。

第三部分。使用 Spring Boot

本节详细介绍了如何使用 Spring Boot。它涵盖了构建系统，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 最佳做法。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），但有一些建议，如果遵循这些建议，可以使您的开发过程更容易一些。

如果您从 Spring Boot 开始，在进入本节之前，您应该阅读“ 入门指南”。

13.构建系统

强烈建议您选择支持依赖关系管理且可以使用发布到“Maven Central”存储库的工件的构建系统 。我们建议您选择 Maven 或 Gradle。可以使 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

Spring Boot 的每个版本都提供了它支持的依赖项的策划列表。实际上，您不需要为构建配置中的任何这些依赖项提供版本，因为 Spring Boot 会为您管理。升级 Spring 引导时，这些依赖项也会以一致的方式升级。

注意

如果需要，您仍然可以指定版本并覆盖 Spring Boot 的建议。

精选列表包含您可以使用 Spring Boot 的所有 spring modules 以及精选的第三方库列表。该列表以标准 物料清单（`spring-boot-dependencies`）的形式提供 ，可与 Maven 和 Gradle 一起使用。

警告

Spring Boot 的每个版本都与 Spring 框架的基本版本相关联。我们**强烈**建议您不要指定其版本。

13.2 Maven

Maven 用户可以继承 `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 作为默认编译器级别。
- UTF-8 源编码。
- 一个依赖管理部分，从春天启动依赖性继承 POM，管理公共依赖的版本。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。
- 使用 `repackage` 执行 ID 执行 `repackage` 目标。
- 明智的 资源过滤。
- 明智的插件配置（exec 插件， Git 提交 ID 和 阴影）。
- `application.properties` 和 `application.yml` 的合理资源过滤，包括特定于配置文件的文件（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。）

13.2.1 继承 Starter Parent

要将项目配置为从 spring-boot-starter-parent 继承，请按以下方式设置 parent：

```
<!-- Inherit defaults from Spring Boot -->
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.1.RELEASE</version>
</parent>
```

注意

您应该只需要在此依赖项上指定 Spring Boot 版本号。如果导入其他启动器，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 数据发布列表，您可以将以下内容添加到 pom.xml：

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

小费

检查 spring-boot-dependencies pom 以获取支持的属性列表。

13.2.2 在没有父 POM 的情况下使用 Spring Boot

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父级，或者您可能希望明确声明所有 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（但不是插件管理）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 中添加条目。例如，要升级到另一个 Spring 数据发布列，您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>
                        <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>
```

注意

在前面的示例中，我们指定了 *BOM*，但是可以以相同的方式覆盖任何依赖关系类型。

13.2.3 使用 Spring Boot 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>
```

注意

如果您使用 Spring Boot 启动程序父 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 Gradle

要了解如何将 Spring Boot 与 Gradle 一起使用，请参阅 Spring Boot 的 Gradle 插件的文档：

- 参考（HTML 和 PDF）
- API

13.4 Ant

可以使用 Apache Ant + Ivy 构建一个 Spring Boot 项目。`spring-boot-antlib`“AntLib”模块也可用于帮助 Ant 创建可执行 jar。

要声明依赖项，典型的 `ivy.xml` 文件类似于以下示例：

```
<ivy-module version="2.0">
```

```
        <info organisation="org.springframework.boot" module="spring-boot-sample-ant"
/>
        <configurations>
                <conf name="compile" description="everything needed to compile this
module" />
                <conf name="runtime" extends="compile" description="everything needed
to run this module" />
        </configurations>
        <dependencies>
                <dependency org="org.springframework.boot" name="spring-boot-starter"
                        rev="${spring-boot.version}" conf="compile" />
        </dependencies>
</ivy-module>
```

典型的 `build.xml` 类似于以下示例：

```
<project
        xmlns:ivy="antlib:org.apache.ivy.ant"
        xmlns:spring-boot="antlib:org.springframework.boot.ant"
        name="myapp" default="build">

        <property name="spring-boot.version" value="2.1.1.RELEASE" />

        <target name="resolve" description="--> retrieve dependencies with ivy">
                <ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[revision].[ext]"
/>
        </target>

        <target name="classpaths" depends="resolve">
                <path id="compile.classpath">
                        <fileset dir="lib/compile" includes="*.jar" />
                </path>
        </target>

        <target name="init" depends="classpaths">
                <mkdir dir="build/classes" />
        </target>

        <target name="compile" depends="init" description="compile">
                <javac srcdir="src/main/java" destdir="build/classes"
classpathref="compile.classpath" />
        </target>

        <target name="build" depends="compile">
                <spring-boot:exejar destfile="build/myapp.jar"
classes="build/classes">
                        <spring-boot:lib>
                                <fileset dir="lib/runtime" />
                        </spring-boot:lib>
                </spring-boot:exejar>
        </target>
</project>
```

小费

如果您不想使用 `spring-boot-antlib` 模块，请参见 *第 91.9 节“从 Ant 构建可执行文件，而不使用 spring-boot-antlib”* “操作方法”。

13.5 Starters

Starters 是一组方便的依赖描述符，您可以在应用程序中包含这些描述符。您可以获得所需的所有 Spring 和相关技术的一站式服务，而无需搜索示例代码和复制粘贴依赖描述符的负载。例如，如果要开始使用 Spring 和 JPA 进行数据库访问，请在项目中包含 `spring-boot-starter-data-`

jpa 依赖项。

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

什么是名字

所有**官方**首发都遵循类似的命名模式; `spring-boot-starter-*`，其中`*`是一种特殊类型的应用程序。此命名结构旨在帮助您找到启动器。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，可以在 POM 编辑器中按 `ctrl-space` 并输入“spring-boot-starter”以获取完整列表。

正如“ 创建自己的初学者 ”部分所述，第三方启动者不应以 `spring-boot` 开头，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` 的第三方启动项目通常被命名为 `thirdpartyproject-spring-boot-starter`。

以下应用程序启动程序由 `org.springframework.boot` 组下的 Spring Boot 提供：

表 13.1。Spring Boot 应用程序启动器

名称	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 消息传递的入门者
`spring-boot-starter-amqp`	使用 Spring AMQP 和 Rabbit MQ 的入门者
`spring-boot-starter-aop`	使用 Spring AOP 和 AspectJ 进行面向方面编程的入门者
`spring-boot-starter-artemis`	使用 Apache Artemis 进行 JMS 消息传递的入门者
`spring-boot-starter-batch`	使用 Spring 批处理的初学者
`spring-boot-starter-cache`	使用 Spring Framework 的缓存支持的初学者
`spring-boot-starter-cloud-connectors`	使用 Spring 云连接器的初学者简化了与 Cloud Foundry 和 Heroku 等云平台中的服务的连接
`spring-boot-starter-data-cassandra`	使用 Cassandra 分布式数据库和 Spring 数据的初学者 Cassandra

名称	描述
spring-boot-starter-data-cassandra-reactive	使用 Cassandra 分布式数据库和 Spring 数据 Cassandra Reactive 的入门者
spring-boot-starter-data-couchbase	使用 Couchbase 面向文档的数据库和 Spring Data Couchbase 的入门者
spring-boot-starter-data-couchbase-reactive	使用 Couchbase 面向文档的数据库和 Spring Data Couchbase Reactive 的入门者
spring-boot-starter-data-elasticsearch	使用 Elasticsearch 搜索和分析引擎以及 Spring Data Elasticsearch 的初学者
spring-boot-starter-data-jdbc	使用 Spring 数据 JDBC 的入门者
spring-boot-starter-data-jpa	使用 Spring Data JPA 和 Hibernate 的初学者
spring-boot-starter-data-ldap	使用 Spring 数据 LDAP 的入门者
spring-boot-starter-data-mongodb	使用 MongoDB 面向文档的数据库和 Spring Data MongoDB 的初学者
spring-boot-starter-data-mongodb-reactive	使用 MongoDB 面向文档的数据库和 Spring Data MongoDB Reactive 的入门者
spring-boot-starter-data-neo4j	使用 Neo4j 图形数据库和 Spring 数据 Neo4j 的入门者
spring-boot-starter-data-redis	使用 Spring 数据 Redis 和 Lettuce 客户端使用 Redis 键值数据存储的入门者
spring-boot-starter-data-redis-reactive	使用 Redis 数据 Redis 被动和 Lettuce 客户端的 Redis 键值数据存储的入门者
spring-boot-starter-data-rest	使用 Spring Data REST 通过 REST 公开 Spring 数据存储库的入门者
spring-boot-starter-data-solr	使用带有 Spring Data Solr 的 Apache Solr 搜索平台的初学者
spring-boot-starter-freemarker	使用 FreeMarker 视图构建 MVC Web 应用程序的入门者
spring-boot-starter-groovy-templates	使用 Groovy 模板视图构建 MVC Web 应用程序的入门者
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

名称	描述
	的入门者
spring-boot-starter-integration	使用 Spring Integration 的入门者
spring-boot-starter-jdbc	将 JDBC 与 HikariCP 连接池一起使用的入门者
spring-boot-starter-jersey	使用 JAX-RS 和 Jersey 构建 RESTful Web 应用程序的入门者。替代 spring-boot-starter-web
spring-boot-starter-jooq	使用 jOOQ 访问 SQL 数据库的初学者。替代 spring-boot-starter-data-jpa 或 spring-boot-starter-jdbc
spring-boot-starter-json	阅读和写作 json 的初学者
spring-boot-starter-jta-atomikos	使用 Atomikos 进行 JTA 交易的入门者
spring-boot-starter-jta-bitronix	使用 Bitronix 进行 JTA 事务的入门者
spring-boot-starter-mail	使用 Java Mail 和 Spring Framework 的电子邮件发送支持的入门者
spring-boot-starter-mustache	使用 Mustache 视图构建 Web 应用程序的入门者
spring-boot-starter-oauth2-client	使用 Spring 安全性 OAuth2 / OpenID Connect 客户端功能的入门级产品
spring-boot-starter-oauth2-resource-server	使用 Spring 安全性 OAuth2 资源服务器功能的入门者
spring-boot-starter-quartz	使用 Quartz 调度程序的入门者
spring-boot-starter-security	使用 Spring 安全性的入门者
spring-boot-starter-test	用于测试包含 JUnit，Hamcrest 和 Mockito 等库的 Spring Boot 应用程序的入门者
spring-boot-starter-thymeleaf	使用 Thymeleaf 视图构建 MVC Web 应用程序的入门者

名称	描述
`spring-boot-starter-validation`	使用 Java Bean 验证与 Hibernate Validator 的初学者
`spring-boot-starter-web`	使用 Spring MVC 构建 Web（包括 RESTful）应用程序的入门者。使用 Tomcat 作为默认嵌入式容器
`spring-boot-starter-web-services`	使用 Spring Web 服务的入门者
`spring-boot-starter-webflux`	使用 Spring Framework 的 Reactive Web 支持构建 WebFlux 应用程序的初学者
`spring-boot-starter-websocket`	使用 Spring Framework 的 WebSocket 支持构建 WebSocket 应用程序的入门者

除应用程序启动器外，还可以使用以下启动器添加 生产就绪功能：

表 13.2。Spring Boot 制作首发

名称	描述	双响炮
`spring-boot-starter-actuator`	使用 Spring Boot 的 Actuator 的启动器，它提供生产就绪功能，帮助您监控和管理您的应用程序	双响炮

最后，Spring Boot 还包括以下可用于排除或交换特定技术方面的启动器：

表 13.3。Spring Boot 技术先发者

名称	描述	双响炮
`spring-boot-starter-jetty`	使用 Jetty 作为嵌入式 servlet 容器的入门。替代 `spring-boot-starter-tomcat`	双响炮
`spring-boot-starter-log4j2`	使用 Log4j2 进行日志记录的入门。替代 `spring-boot-starter-logging`	双响炮
`spring-boot-starter-logging`	使用 Logback 进行日志记录的入门。默认日志启动器	双响炮
`spring-boot-starter-reactor-netty`	使用 Reactor Netty 作为嵌入式响应式 HTTP 服务器的入门者。	双响炮
`spring-boot-starter-tomcat`	使用 Tomcat 作为嵌入式 servlet 容器的入门者。使用的默认 servlet 容器启动器	双响炮

名称	描述	双响炮
	spring-boot-starter-web	
spring-boot-starter-undertow	使用 Undertow 作为嵌入式 servlet 容器的入门者。替代 spring-boot-starter-tomcat	双响炮

小费

有关其他社区贡献的启动器的列表，请参阅 GitHub 上 spring-boot-starters 模块中的 README 文件。

14.构建您的代码

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 使用“默认”包

当一个类不包含 package 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。

小费

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 找到主应用程序类

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。的@SpringBootApplication 注释往往放在主类，它隐式地定义某些项目碱“的搜索包”。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。

小费

如果您不想使用@SpringBootApplication，它导入的@EnableAutoConfiguration 和@ComponentScan 注释会定义该行为，因此您也可以使用它。

以下清单显示了典型的布局：

```
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@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 源一起使用，但我们通常建议您的主要来源是单个@Configuration 类。通常，定义 main 方法的类是主要的@Configuration 候选者。

小费

许多 Spring 配置示例已在 Internet 上发布，使用 XML 配置。如果可能，请始终尝试使用等效的基于 Java 的配置。搜索 Enable*注释可能是一个很好的起点。

15.1 导入其他配置类

你不需要将所有@Configuration 放入一个班级。@Import 注释可用于导入其他配置类。或者，您可以使用@ComponentScan 自动选取所有 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 注释来加载 XML 配置文件。

16.自动配置

Spring Boot 自动配置尝试根据您添加的 jar 依赖项自动配置您的 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 beans，则 Spring Boot 会自动配置内存数据库。

您需要通过向@Configuration 类之一添加@EnableAutoConfiguration 或@SpringBootApplication 注释来选择加入自动配置。

小费

您应该只添加一个`@SpringBootApplication`或`@EnableAutoConfiguration`注释。我们通常建议您仅将一个或另一个添加到主`@Configuration`类。

16.1 逐步更换自动配置

自动配置是非侵入性的。在任何时候，您都可以开始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退回。

如果您需要了解当前正在应用的自动配置以及原因，请使用`--debug`开关启动您的应用程序。这样做可以为选择的核心记录器启用调试日志，并将条件报告记录到控制台。

16.2 禁用特定的自动配置类

如果发现正在应用您不需要的特定自动配置类，则可以使用`@EnableAutoConfiguration`的exclude属性禁用它们，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 {
}
```

如果类不在类路径上，则可以使用注释的`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。

小费

您可以在注释级别和使用属性定义排除项。

17. Spring Beans 和依赖注入

您可以自由使用任何标准Spring框架技术来定义beans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beans）和使用`@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（在根包中定位应用程序类），则可以添加`@ComponentScan`而不带任何参数。您的所有应用程序组件（`@Component`，`@Service`，`@Repository`，`@Controller`等）都会自动注册为Spring Beans。

以下示例显示了`@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 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;
        }

        // ...

}
```

小费

请注意使用构造函数注入如何将 `riskAssessor` 字段标记为 `final`，表示无法随后更改它。

18.使用@SpringBootApplication Annotation

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`：启用 Spring Boot 的自动配置机制
- `@ComponentScan`：对应用程序所在的软件包启用`@Component` 扫描（请参阅最佳实践）
- `@Configuration`：允许在上下文中注册额外的 beans 或导入其他配置类

`@SpringBootApplication` 注释等效于使用 `@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);
        }

}
```

注意

`@SpringBootApplication` 还提供了别名来自定义 `@EnableAutoConfiguration` 和`@ComponentScan` 的属性。

注意

这些功能中的 None 是强制性的，您可以选择使用它启用的任何功能替换此单个注释。例如，您可能不希望在应用程序中使用组件扫描：

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@EnableAutoConfiguration
@Import({ MyConfig.class, MyAnotherConfig.class })
public class Application {

        public static void main(String[] args) {
                        SpringApplication.run(Application.class, args);
        }

}
```

在此示例中，`Application` 与任何其他 Spring Boot 应用程序一样，除了 `@Component` - 未自动检测带注释的类，并且显式导入用户定义的 beans（请参阅 `@Import`） 。

19.运行您的应用程序

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，您可以像运行任何其他服务器一样运行应用程序。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。

注意

本节仅介绍基于罐子的包装。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 运行

您可以从 IDE 运行 Spring Boot 应用程序作为简单的 Java 应用程序。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 `File` 菜单中选择 `Import…→Existing Maven Projects`。

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 包括 Eclipse 和 IDEA 的插件 。Gradle 提供各种 IDE 的插件 。

小费

如果您不小心运行了两次 Web 应用程序，则会看到“端口已在使用中”

错误。STS 用户可以使用 Relaunch 按钮而不是 Run 按钮来确保关闭任何现有实例。

19.2 作为打包应用程序运行

如果您使用 Spring Boot Maven 或 Gradle 插件创建可执行 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
```

也可以运行启用了远程调试支持的打包应用程序。这样做可以将调试器附加到打包的应用程序，如以下示例所示：

```
$ java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n \
       -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

19.3 使用 Maven 插件

Spring Boot Maven 插件包含 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
```

19.4 使用 Gradle 插件

Spring Boot Gradle 插件还包含 bootRun 任务，可用于以爆炸形式运行您的应用程序。每当您应用 org.springframework.boot 和 java 插件时，都会添加 bootRun 任务，如以下示例所示：

```
$ gradle bootRun
```

您可能还想使用 JAVA_OPTS 操作系统环境变量，如以下示例所示：

```
$ export JAVA_OPTS=-Xmx1024m
```

19.5 热插拔

由于 Spring Boot 应用程序只是普通的 Java 应用程序，因此 JVM 热交换应该是开箱即用的。JVM 热交换在某种程度上受限于它可以替换的字节码。要获得更完整的解决方案， 可以使用 JRebel。

spring-boot-devtools 模块还包括对快速应用程序重启的支持。有关详细信息，请参阅本章后面的第 20 章“ 开发人员工具”部分和 热交换“操作方法”。

20.开发人员工具

Spring Boot 包括一组额外的工具，可以使应用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，以提供额外的开发时间功能。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 和 Gradle 列表所示：

Maven.

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

Gradle.

```
configurations {
        developmentOnly
        runtimeClasspath {
                extendsFrom developmentOnly
        }
}
dependencies {
        developmentOnly("org.springframework.boot:spring-boot-devtools")
}
```

注意

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，则它被视为“生产应用程序”。在 Maven 中将依赖项标记为可选，或在 Gradle 中使用 custom`developmentOnly`配置（如上所示）是防止 devtools 传递应用于使用项目的其他模块的最佳实践。

小费

重新打包的归档默认情况下不包含 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 插件均支持该属性。

20.1 Property 默认值

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-boot-devtools 默认禁用缓存选项。

缓存选项通常由 `application.properties` 文件中的设置配置。例如，Thymeleaf 提供 `spring.thymeleaf.cache` 财产。`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时配置。

由于在开发 Spring MVC 和 Spring WebFlux 应用程序时需要有关 Web 请求的更多信息，因此开发人员工具将为 `web` 日志记录组启用 `DEBUG` 日志记录。这将为您提供有关传入请求，处理程序正在处理它，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），您可以打开 `spring.http.log-request-details` 配置属性。

注意

如果您不希望应用属性默认值，则可以在

`application.properties` 中将 `spring.devtools.add-properties` 设置为 `false`。

小费

有关 devtools 应用的属性的完整列表，请参阅 DevToolsPropertyDefaultsPostProcessor。

20.2 自动重启

只要类路径上的文件发生更改，使用 `spring-boot-devtools` 的应用程序就会自动重新启动。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循环。默认情况下，将监视类路径上指向文件夹的任何条目的更改。请注意，某些资源（如静态资产和视图模板）无需重新启动应用程序。

触发重启

由于 DevTools 监视类路径资源，因此触发重新启动的唯一方法是更新类路径。导致更新类路径的方式取决于您使用的 IDE。在 Eclipse 中，保存修改后的文件会导致更新类路径并触发重新启动。在 IntelliJ IDEA 中，构建项目（`Build -> Build Project`）具有相同的效果。

注意

只要启用了分叉，您也可以使用支持的构建插件（Maven 和 Gradle）启动应用程序，因为 DevTools 需要一个独立的应用程序类加载器才能正常运行。默认情况下，Gradle 和 Maven 在类路径上检测到 DevTools 时会这样做。

小费

与 LiveReload 一起使用时，自动重启非常有效。 有关详细信息，请参阅 LiveReload 部分。如果使用 JRebel，则禁用自动重新启动以支持动态类重新加载。其他 devtools 功能（例如 LiveReload 和属性覆盖）仍然可以使用。

注意

DevTools 依赖于应用程序上下文的关闭钩子来在重启期间关闭它。如果禁用了关闭挂钩（`SpringApplication.setRegisterShutdownHook(false)`），它将无法正常工作。

注意

在确定类路径上的条目是否应在更改时触发重新启动时，DevTools 会自动忽略名为 `spring-boot`，`spring-boot-devtools`，`spring-boot-autoconfigure`，`spring-boot-actuator` 和 `spring-boot-starter` 的项目。

注意

DevTools 需要自定义 `ApplicationContext` 使用的

ResourceLoader。如果您的应用程序已经提供了一个，它将被包装。不支持直接覆盖 ApplicationContext 上的 getResource 方法。

重新启动 vs Reload

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。您正在积极开发的类将加载到重新启动的 类加载器中。重新启动应用程序时，将重新启动重新启动的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以使它们更适合重新加载。

20.2.1 记录条件评估中的变化

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。该报告显示在您进行更改（例如添加或删除 beans 和设置配置属性）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
```

20.2.2 不包括资源

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 中的资源不会触发重新启动，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/**
```

小费

如果要保留这些默认值并添加其他排除项，请改用 spring.devtools.restart.additional-exclude 属性。

20.2.3 查看其他路径

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.additional-paths 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。

20.2.4 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）。

如果您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 System 属性

设置为 `false`，如如下例所示：

```
public static void main(String[] args) {
        System.setProperty("spring.devtools.restart.enabled", "false");
        SpringApplication.run(MyApp.class, args);
}
```

20.2.5 使用触发文件

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。

小费

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

20.2.6 自定义重启类加载器

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` 文件都加载了“base”类加载器。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，您可以创建一个 `META-INF/spring-devtools.properties` 文件。

`spring-devtools.properties` 文件可以包含前缀为 `restart.exclude` 和 `restart.include` 的属性。`include` 元素是应该被拉入“restart”类加载器的项目，`exclude` 元素是应该被下推到“基础”类加载器中的项目。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
```

注意

所有属性键必须是唯一的。只要属性以 `restart.include.`或 `restart.exclude.`开头，就会被考虑。

小费

从类路径中加载所有 `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。

20.2.7 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。如果您需要反序列化数据，则可能需要将 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而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` 模块包括一个嵌入式 LiveReload 服务器，可用于在更改资源时触发浏览器刷新。LiveReload 浏览器扩展程序可从 livereload.com 免费用于 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`。

注意

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-boot-devtools.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 引导-devtools.properties。

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

注意

在 `.spring-boot-devtools.properties` 中激活的配置文件不会影响特定于配置文件的配置文件的加载。

20.5 远程应用程序

Spring Boot 开发人员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。远程支持是选择加入。要启用它，您需要确保重新打包的存档中包含 `devtools`，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，如以下示例所示：

```
spring.devtools.remote.secret=mysecret
```

警告

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。

20.5.1 运行远程客户端应用程序

远程客户端应用程序旨在从 IDE 中运行。您需要使用与连接到的远程项目相同的类路径运行 `org.springframework.boot.devtools.RemoteSpringApplication`。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，并且已经部署到 Cloud Foundry 的项目名为 `my-app`，则可以执行以下操作：

- 从 `Run` 菜单中选择 `Run Configurations…`。
- 创建一个新的 `Java Application`“启动配置”。
- 浏览 `my-app` 项目。
- 使用 `org.springframework.boot.devtools.RemoteSpringApplication` 作为主类。
- 将 `https://myapp.cfapps.io` 添加到 `Program arguments`（或任何远程 URL）。

正在运行的远程客户端可能类似于以下列表：

```
  .   ____          _                                              __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___                     _      \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_ ___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/ -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\___/\__\___|/ / / /
 =========|_|==============|___/=================================/_/_/_/
 :: Spring Boot Remote :: 2.1.1.RELEASE

2015-06-10 18:25:06.632  INFO 14938 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : Starting RemoteSpringApplication on pwmbp
with PID 14938 (/Users/pwebb/projects/spring-boot/code/spring-boot-
devtools/target/classes started by pwebb in /Users/pwebb/projects/spring-
boot/code/spring-boot-samples/spring-boot-sample-devtools)
2015-06-10 18:25:06.671  INFO 14938 --- [           main]
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@2a17b7b6:
startup date [Wed Jun 10 18:25:06 PDT 2015]; root of context hierarchy
2015-06-10 18:25:07.043  WARN 14938 --- [           main]
o.s.b.d.r.c.RemoteClientConfiguration    : The connection to http://localhost:8080 is
insecure. You should use a URL starting with 'https://'.
2015-06-10 18:25:07.074  INFO 14938 --- [           main]
o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2015-06-10 18:25:07.130  INFO 14938 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : Started RemoteSpringApplication in 0.74
seconds (JVM running for 1.105)
```

注意

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 `spring.devtools.remote.secret` 属性的读取方式并传递给服务器进行身份验证。

小费

始终建议使用 `https://`作为连接协议，以便加密流量并且不会截获密码。

小费

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。

20.5.2 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。

注意

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.包装您的生产应用程序

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。

对于其他“生产就绪”功能，例如运行状况，审计和度量标准 REST 或 JMX 端点，请考虑添加 `spring-boot-actuator`。有关详细信息，请参见 *第 V 部分“Spring Boot Actuator：生产就绪功能”*。

22.接下来要阅读的内容

您现在应该了解如何使用 Spring Boot 和一些您应该遵循的最佳实践。您现在可以继续深入了解特定的 *Spring Boot 功能*，或者您可以跳过并阅读 Spring Boot 的“ 生产就绪 ”方面。

第四部分。Spring Boot 功能

本节深入研究 Spring Boot 的细节。在这里，您可以了解您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，您可以委托静态 `SpringApplication.run` 方法，如以下示例所示：

```
public static void main(String[] args) {
        SpringApplication.run(MySpringConfiguration.class, args);
}
```

当您的应用程序启动时，您应该看到类似于以下输出的内容：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v2.1.1.RELEASE

2013-07-31 00:08:16.117  INFO 56603 --- [           main]
o.s.b.s.app.SampleApplication            : Starting SampleApplication v0.1.0 on
mycomputer with PID 56603 (/apps/myapp.jar started by pwebb)
2013-07-31 00:08:16.166  INFO 56603 --- [           main]
ationConfigServletWebServerApplicationContext : Refreshing
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicat
ionContext@6e5a8246: startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context
hierarchy
2014-03-04 13:09:54.912  INFO 41370 --- [           main]
.t.TomcatServletWebServerFactory : Server initialized with port: 8080
2014-03-04 13:09:56.501  INFO 41370 --- [           main]
o.s.b.s.app.SampleApplication            : Started SampleApplication in 2.992 seconds
(JVM running for 3.658)
```

默认情况下，会显示 INFO 日志记录消息，包括一些相关的启动详细信息，例如启动应用程序的用户。如果您需要 INFO 以外的日志级别，可以进行设置，如第 26.4 节“日志级别”中所述，

23.1 启动失败

如果您的应用程序无法启动，则已注册 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.
```

注意

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

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.ConditionEvaluationReportLoggingListener 启用 DEBUG 日志记录。

例如，如果使用 java -jar 运行应用程序，则可以启用 debug 属性，如下所示：

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

23.2 自定义横幅

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，可以更改启动时打印的横幅。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，并打印在任何文本横幅上方。

在 banner.txt 文件中，您可以使用以下任何占位符：

表 23.1。横幅变量

变量	描述
${application.version}	应用程序的版本号，如 MANIFEST.MF 中声明的那样。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（用括号括起来并以 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)。
${Ansi.NAME}（或${AnsiColor.NAME}，${AnsiBackground.NAME}，${AnsiStyle.NAME}）	其中 NAME 是 ANSI 转义码的名称。详情 AnsiPropertySource 请见。
${application.title}	申请的标题，如 MANIFEST.MF 中所述。例如 Implementation-Title: MyApp 打印为 MyApp。

小费

如果要以编程方式生成横幅，可以使用 SpringApplication.setBanner(…)方法。使用 org.springframework.boot.Banner 接口并实现您自己的 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）上打印横幅，发送到配置的记录器（log），或者根本不产生横幅（off）。

打印的横幅以下列名称注册为单身 bean：springBootBanner。

注意

YAML 将 `off` 映射到 `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` 默认值不符合您的口味，您可以改为创建本地实例并对其进行自定义。例如，要关闭横幅，您可以写：

```
public static void main(String[] args) {
        SpringApplication app = new SpringApplication(MySpringConfiguration.class);
        app.setBannerMode(Banner.Mode.OFF);
        app.run(args);
}
```

注意

传递给 `SpringApplication` 的构造函数参数是 Spring beans 的配置源。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 配置或应扫描的包的引用。

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。

有关配置选项的完整列表，请参阅 `SpringApplication` Javadoc。

23.4 Fluent Builder API

如果您需要构建 `ApplicationContext` 层次结构（具有父/子关系的多个上下文）或者您更喜欢使用“流畅”构建器 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 `parent` 和 `child` 方法，如以下示例所示：

```
new SpringApplicationBuilder()
                .sources(Parent.class)
                .child(Application.class)
                .bannerMode(Banner.Mode.OFF)
                .run(args);
```

注意

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，并且父/子上下文使用相同的 `Environment`。有关详细信息，请参阅 `SpringApplicationBuilder` Javadoc。

23.5 应用程序事件和监听器

除了通常的 Spring 框架事件之外，例如 `ContextRefreshedEvent`，`SpringApplication` 还会发送一些其他应用程序事件。

注意

某些事件实际上是在创建 ApplicationContext 之前触发的，因此您无法在@Bea
注册侦听器。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
org.springframework.context.ApplicationListener 键引用您的侦听器，
下所示：以下示例：

```
org.springframework.context.ApplicationListener=com.example.project.MyList
```

应用程序运行时，应按以下顺序发送应用程序事件：

1. 除了注册侦听器和初始化程序之外，在运行开始时但在任何处理之前发送 ApplicationStartingEvent。
2. 当在上下文中使用的 Environment 已知但在创建上下文之前，将发送 ApplicationEnvironmentPreparedEvent。
3. 在刷新开始之前但在加载 bean 定义之后发送 ApplicationPreparedEvent。
4. 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 ApplicationStartedEvent。
5. 在调用任何应用程序和命令行运行程序后发送 ApplicationReadyEvent。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，则会发送 ApplicationFailedEvent。

小费

您经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。

应用程序事件使用 Spring Framework 的事件发布机制发送。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 或者如果监听器是 bean，使用@Autowired 来注入上下文。

23.6 网络环境

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：

- 如果存在 Spring MVC，则使用 AnnotationConfigServletWebServerApplicationContext
- 如果 Spring MVC 不存在且存在 Spring WebFlux，则使用 AnnotationConfigReactiveWebServerApplicationContext
- 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 和来自 Spring WebFlux 的新 WebClient，默认情况下将使用 Spring MVC。您可以通过调用

setWebApplicationType(WebApplicationType)轻松覆盖它。

也可以通过调用 setApplicationContextClass(…)来完全控制使用的 ApplicationContext 类型。

小费

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

如果您需要访问传递给 SpringApplication.run(…)的应用程序参数，则可以注入 org.springframework.boot.ApplicationArguments bean。ApplicationArguments 接口提供对原始 String[]参数以及解析的 option 和 non-option 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"]
        }

}
```

小费

Spring Boot 还注册 CommandLinePropertySource 和 Spring Environment。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

如果您需要在 SpringApplication 启动后运行某些特定代码，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，作为简单的字符串数组，而 ApplicationRunner 使用前面讨论的 ApplicationArguments 接口。以下示例显示 CommandLineRunner 和 run 方法：

```
import org.springframework.boot.*;
import org.springframework.stereotype.*;

@Component
public class MyBean implements CommandLineRunner {

        public void run(String... args) {
```

```
                // Do something...
        }

}
```

如果定义了必须按特定顺序调用的多个 `CommandLineRunner` 或 `ApplicationRunner` beans，则可以另外实现 `org.springframework.core.Ordered` 接口或使用 `org.springframework.core.annotation.Order` 注释。

23.9 申请退出

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 Spring 生命周期回调（例如 `DisposableBean` 接口或 `@PreDestroy` 注释）。

此外，beans 如果希望在调用 `SpringApplication.exit()`时返回特定的退出代码，则可以实现 `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` 接口可以通过例外来实现。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 管理员功能

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。这暴露了 [SpringApplicationAdminMXBean] 平台 `MBeanServer`。您可以使用此功能远程管理您的 Spring Boot 应用程序。此功能对于任何服务包装器实现也很有用。

小费

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

警告

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 允许您外部化您的配置，以便您可以在不同的环境中使用相同的应用程序代码。您可以使用属性文件，YAML 文件，环境变量和命令行参数来外部化配置。Property 值可以通过使用 `@Value` 注释直接注入 beans，通过 Spring 的 `Environment` 抽象访问，或 通过 `@ConfigurationProperties` 绑定到结构化对象。

Spring Boot 使用非常特殊的 `PropertySource` 顺序，旨在允许合理地覆盖值。按以下顺序考虑属性：

1. Devtools 主目录上的全局设置属性（当 devtools 处于活动状态时`~/.spring-boot-devtools.properties`）。
2. `@TestPropertySource` 测试上的注释。
3. `properties` 属于您的测试。可 用于测试特定应用程序片段`@SpringBootTest` 的 测试注释。
4. 命令行参数。
5. 来自 `SPRING_APPLICATION_JSON` 的属性（嵌入在环境变量或系统属性中的内联 JSON）。
6. `ServletConfig` init 参数。
7. `ServletContext` init 参数。
8. JNDI 来自 `java:comp/env`。
9. Java 系统属性（`System.getProperties()`）。
10. OS 环境变量。
11. `RandomValuePropertySource` 仅在 `random.*`中具有属性。
12. 特定于配置文件的应用程序属性在打包的 jar 之外（`application-{profile}.properties` 和 YAML 变体）。
13. 打包在 jar 中的特定于配置文件的应用程序属性（`application-{profile}.properties` 和 YAML 变体）。
14. 打包 jar 之外的应用程序属性（`application.properties` 和 YAML 变体）。
15. 打包在 jar 中的应用程序属性（`application.properties` 和 YAML 变体）。
16. `@PropertySource` `@Configuration` 课程上的注释。
17. 默认属性（通过设置 `SpringApplication.setDefaultProperties` 指定）。

为了提供一个具体示例，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，为 `name` 提供合理的默认属性值。在新环境中运行时，可以在 jar 之外提供覆盖 `name` 的 `application.properties` 文件。对于一次性测试，您可以使用特定的命令行开关启动（例如，`java -jar app.jar --name="Spring"`）。

小费

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 `Environment` 中使用 `acme.name=test`。您还可以在 System 属性中将 JSON 提供为 `spring.application.json`，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（例如，进入秘密或测试用例）非常有用。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]}
```

`random.int*`语法为 `OPEN value (,max) CLOSE`，其中 `OPEN,CLOSE` 为任意字符，`value,max` 为整数。如果提供 `max`，那么 `value` 是最小值，`max` 是最大值（不包括）。

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 `property`，并将它们添加到 Spring `Environment`。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 `Environment`，则可以使用 `SpringApplication.setAddCommandLineProperties(false)`禁用它们。

24.3 应用程序 Property 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 `Environment`：

1. 当前目录的`/config` 子目录
2. 当前目录
3. classpath `/config` 包
4. 类路径根

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中定义的属性）。

注意

您还可以使用 YAML（'。mil'）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
```

警告

很早就使用 `spring.config.name` 和 `spring.config.location` 来确定必须加载哪些文件，因此必须将它们定义为环境属性（通常是 OS 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），则它们应以/结束（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，不支持特定于配置文件的变体，并且被任何特定于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，配置的位置为 `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:/`

使用 `spring.config.location` 配置自定义配置位置时，它们会替换默认位置。例如，如果 `spring.config.location` 配置了值 `classpath:/custom-config/,file:./custom-config/`，则搜索顺序将变为：

1. `file:./custom-config/`
2. `classpath:custom-config/`

或者，当使用 `spring.config.additional-location` 配置自定义配置位置时，除了默认位置外，还会使用它们。在默认位置之前搜索其他位置。例如，如果配置了 `classpath:/custom-config/,file:./custom-config/`的其他位置，则搜索顺序将变为以下内容：

1. `file:./custom-config/`
2. `classpath:custom-config/`
3. `file:./config/`
4. `file:./`
5. `classpath:/config/`
6. `classpath:/`

此搜索顺序允许您在一个配置文件中指定默认值，然后有选择地覆盖另一个配置文件中的值。您可以在 `application.properties`（或您使用 `spring.config.name` 选择的任何其他基本名称）中的某个默认位置为您的应用程序提供默认值。然后，可以在运行时使用位于其中一个自定义位置的不同文件覆盖这些默认值。

注意

如果使用环境变量而不是系统属性，则大多数操作系统都不允许使用句点分隔的键名称，但您可以使用下划线（例
如，`SPRING_CONFIG_NAME` 而不是 `spring.config.name`）。

注意

如果应用程序在容器中运行，则可以使用 JNDI 属性（在 `java:comp/env` 中）或 servlet 上下文初始化参数来代替环境变量或系统属性。

24.4 特定于配置文件的属性

除了 `application.properties` 文件之外，还可以使用以下命名约定来定义特定于配置文件的属性：`application-{profile}.properties`。`Environment` 有一组默认配置文件（默认情况下为`[default]`），如果未设置活动配置文件，则使用这些配置文件。换句话说，如果没有显式激活配置文件，则会加载 `application-default.properties` 中的属性。

特定于配置文件的属性从标准 `application.properties` 的相同位置加载，特定于配置文件的文件始终覆盖非特定文件，无论特定于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将在通过 `SpringApplication` API 配置的配置文件之后添加，因此优先。

注意

如果您在 `spring.config.location` 中指定了任何文件，则不会考虑这些文件的特定于配置文件的变体。如果您还想使用特定于配置文件的属性，请使用 `spring.config.location` 中的目录。

24.5 属性中的占位符

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

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

小费

您还可以使用此技术创建现有 Spring Boot 属性的“短”变体。有关详细信息，请参见第 *77.4* 节“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

Environment 中包含的值所必需的钩子点。EnvironmentPostProcessor 界面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，那么 Spring Cloud Vault 项目将支持在 HashiCorp Vault 中存储外部化配置 。

24.7 使用 YAML 而不是属性

YAML 是 JSON 的超集，因此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 作为属性的替代 。

注意

如果您使用“Starters”，则 spring-boot-starter 会自动提供 SnakeYAML。

24.7.1 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 列表表示为具有[index]解除引用的属性键。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 实用程序（这是@ConfigurationProperties 所做的）绑定到这样的属性，你需要在 java.util.List 类型的目标 bean 中拥有一个属性（或 Set）您需要提供一个 setter 或用可变值初始化它。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;
        }
}
```

24.7.2 在 Spring 环境中将 YAML 公开为属性

`YamlPropertySourceLoader` 类可用于在 Spring `Environment` 中将 YAML 公开为 `PropertySource`。这样做可以使用带有占位符语法的`@Value` 注释来访问 YAML 属性。

24.7.3 多轮廓 YAML 文档

您可以使用 `spring.profiles` 键在单个文件中指定多个特定于配置文件的 YAML 文档，以指示文档何时应用，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` 配置文件处于活动状态，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 `development`，`production` 和 `eu-central` 配置文件，则该属性的值为 `192.168.1.100`。

注意

因此，`spring.profiles` 可以包含简单的配置文件名称（例如 `production`）或配置文件表达式。概要表达式允许表达更复杂的概要逻辑，例如 `production & (eu-central | eu-west)`。有关详细信息，请查阅 参考指南。

如果在应用程序上下文启动时没有显式激活，则激活默认配置文件。因此，在以下 YAML 中，我们设置 `spring.security.user.password` 的值，该值**仅**在“默认”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配置文件，并且必须在必要时在所有其他配置文件中显式重置：

```
server:
```

```
  port: 8000
spring:
  security:
    user:
      password: weak
```

使用 spring.profiles 元素指定的 Spring 配置文件可以选择使用 ! 字符来否定。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以匹配。

24.7.4 YAML 缺点

无法使用@PropertySource 注释加载 YAML 文件。因此，如果您需要以这种方式加载值，则需要使用属性文件。

24.8 类型安全配置属性

使用@Value("${property}")注释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 beans 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`，其类型可以从 `String` 强制执行。
- `acme.security.username`，带有嵌套的“安全”对象，其名称由属性名称决定。特别是，那里根本没有使用返回类型，可能是 `SecurityProperties`。
- `acme.security.password`.
- `acme.security.roles`，收集 `String`。

注意

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：

- 映射，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变异。
- 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 `Security` 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 Project 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。

小费

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 {
}
```

注意

当`@ConfigurationProperties` bean 以这种方式注册时，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 名称为 `acme-com.example.AcmeProperties`。

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

```
@Component
@ConfigurationProperties(prefix="acme")
public class AcmeProperties {

        // ... see the preceding example

}
```

这种配置风格与 `SpringApplication` 外部 YAML 配置特别有效，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` beans，您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());
                // ...
        }

}
```

小费

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密钥提供自动完成功能。有关详细信息，请参阅 附录 B，配置元数据附录。

24.8.1 第三方配置

除了使用@ConfigurationProperties 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，请在其 bean 注册中添加@ConfigurationProperties，如以下示例所示：

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

使用 another 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 AnotherComponent bean。

24.8.2 松弛结合

Spring Boot 使用一些宽松的规则将 Environment 属性绑定到@ConfigurationProperties beans，因此不需要在 Environment 属性名称和 bean 属性之间进行精确匹配名称。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 port）。

例如，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，可以使用以下属性名称：

表 24.1。轻松绑定

Property	注意
acme.my-project.person.first-name	Kebab 案例，建议在.properties 和.yml 文件中使用。
acme.myProject.person.firstName	标准的驼峰案例语法。
acme.my_project.person.first_name	下划线表示法，这是在.properties 和.yml 文件中使用的替代格式。
ACME_MYPROJECT_PERSON_FIRSTNAME	大写格式，使用系统环境变量时建议使用。

注意

注释的 `prefix` 值必须为 kebab 大小写（小写并以-分隔，例如 `acme.my-project.person`）。

表 24.2。每个属性源放宽绑定规则

Property 来源	简单	名单
属性文件	骆驼案，烤肉串案例或下划线表示法	使用`[ ]`或逗号分隔值的标准列表语法
YAML 文件	骆驼案，烤肉串案例或下划线表示法	标准 YAML 列表语法或逗号分隔值
环境变量	大写格式，下划线作为分隔符。`_` 不应在属性名称中使用	由下划线包围的数字值，例如 `MY_ACME_1_OTHER = my.acme[1].other`
系统属性	骆驼案，烤肉串案例或下划线表示法	使用`[ ]`或逗号分隔值的标准列表语法

小费

我们建议，在可能的情况下，属性以小写烤肉串格式存储，例如 `my.property-name=acme`。

绑定到 `Map` 属性时，如果 `key` 包含除小写字母数字字符或`-`以外的任何内容，则需要使用括号表示法以保留原始值。如果密钥未被`[]`包围，则删除任何非字母数字或`-`的字符。例如，考虑将以下属性绑定到 `Map`：

```
acme:
  map:
    "[/key1]": value1
    "[/key2]": value2
    /key3: value3
```

上面的属性将绑定到 `Map`，其中`/key1`，`/key2` 和 `key3` 作为地图中的键。

24.8.3 合并复杂类型

当列表在多个位置配置时，覆盖通过替换整个列表来工作。

例如，假设一个 `MyPojo` 对象，其 `name` 和 `description` 属性默认为 `null`。以下示例公开了来自 `AcmeProperties` 的 `MyPojo` 对象的列表：

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

        private final List<MyPojo> list = new ArrayList<>();

        public List<MyPojo> getList() {
                return this.list;
        }
```

```
}
```

请考虑以下配置：

```
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，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 list 仍然 只包含一个条目（名称为 my another name 且描述为 null）。此配置不会向列表添加第二个 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（并且仅具有该优先级）的配置文件。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 配置文件处于活动状态，则 AcmeProperties.list 包含 一个 MyPojo 条目（名称为 my another name，描述为 null）。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;
        }

}
```

请考虑以下配置：

```
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 个人资料未激活，则 AcmeProperties.map 包含一个密钥为 key1 的条目（名称为 my name 1，描述为 my description 1）。但是，如果 dev 配置文件已启用，则 map 包含两个带有密钥 key1 的条目（名称为 dev name 1 且描述为 my description 1）和 key2（带有名称 dev name 2 和描述 dev description 2）。

注意

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。

24.8.4 属性转换

Spring Boot 在绑定到@ConfigurationProperties beans 时尝试将外部应用程序属性强制转换为正确的类型。如果您需要自定义类型转换，则可以提供 ConversionService bean（bean 名为 conversionService）或自定义属性编辑器（通过 CustomEditorConfigurer bean）或自定义 Converters（bean 定义注释为@ConfigurationPropertiesBinding）。

注意

由于在应用程序生命周期中很早就请求 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，您可能需要重命名自定义 ConversionService，并且只依赖于使用 @ConfigurationPropertiesBinding 限定的自定义转换器。

转换持续时间

Spring Boot 专门支持表达持续时间。如果公开 java.time.Duration 属性，则可以使用应用程序属性中的以下格式：

- 常规 long 表示（使用毫秒作为默认单位，除非指定了@DurationUnit）
- java.util.Duration 使用的标准 ISO-8601 格式
- 更可读的格式，其中值和单位耦合（例如 10s 表示 10 秒）

请考虑以下示例：

```
@ConfigurationProperties("app.system")
public class AppSystemProperties {

        @DurationUnit(ChronoUnit.SECONDS)
        private Duration sessionTimeout = Duration.ofSeconds(30);

        private Duration readTimeout = Duration.ofMillis(1000);

        public Duration getSessionTimeout() {
                return this.sessionTimeout;
        }

        public void setSessionTimeout(Duration sessionTimeout) {
                this.sessionTimeout = sessionTimeout;
```

```
        }

        public Duration getReadTimeout() {
                return this.readTimeout;
        }

        public void setReadTimeout(Duration readTimeout) {
                this.readTimeout = readTimeout;
        }

}
```

要指定 30 秒的会话超时，`30`，`PT30S` 和 `30s` 都是等效的。可以使用以下任何一种形式指定 500ms 的读取超时：`500`，`PT0.5S` 和 `500ms`。

您也可以使用任何支持的单位。这些是：

- 纳秒时间为 `ns`
- `us` 微秒
- `ms` 毫秒
- `s` 秒
- `m` 分钟
- `h` 几个小时
- `d` 几天

默认单位是毫秒，可以使用`@DurationUnit` 覆盖，如上面的示例所示。

小费

如果要从仅使用 `Long` 来表示持续时间的先前版本升级，请确保定义单位（使用`@DurationUnit`），如果它不是切换到 `Duration` 旁边的毫秒。这样做可以提供透明的升级路径，同时支持更丰富的格式。

转换数据大小

Spring Framework 有一个 `DataSize` 值类型，允许以字节为单位表示大小。如果公开 `DataSize` 属性，则可以使用应用程序属性中的以下格式：

- 常规 `long` 表示（使用字节作为默认单位，除非指定了`@DataSizeUnit`）
- 值和单元耦合的更易读的格式（例如 `10MB` 表示 10 兆字节）

请考虑以下示例：

```
@ConfigurationProperties("app.io")
public class AppIoProperties {

        @DataSizeUnit(DataUnit.MEGABYTES)
        private DataSize bufferSize = DataSize.ofMegabytes(2);

        private DataSize sizeThreshold = DataSize.ofBytes(512);

        public DataSize getBufferSize() {
                return this.bufferSize;
        }

        public void setBufferSize(DataSize bufferSize) {
                this.bufferSize = bufferSize;
        }

        public DataSize getSizeThreshold() {
```

```
                return this.sizeThreshold;
        }

        public void setSizeThreshold(DataSize sizeThreshold) {
                this.sizeThreshold = sizeThreshold;
        }

}
```

要指定 10 兆字节的缓冲区大小，`10` 和 `10MB` 是等效的。可以将大小阈值 256 字节指定为 `256` 或 `256B`。

您也可以使用任何支持的单位。这些是：

- `B` 表示字节
- 千字节 `KB`
- `MB` 表示兆字节
- `GB` 为千兆字节
- `TB` 表示太字节

默认单位是字节，可以使用`@DataSizeUnit` 覆盖，如上面的示例所示。

小费

如果要从仅使用 `Long` 来表示大小的先前版本升级，请确保定义单位（使用`@DataSizeUnit`），如果它不是切换到 `DataSize` 旁边的字节。这样做可以提供透明的升级路径，同时支持更丰富的格式。

24.8.5 @ConfigurationProperties 验证

Spring Boot 尝试使用 Spring `@Validated` 注释注释`@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

}
```

小费

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

虽然嵌套属性也会在绑定时进行验证，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`。应该声明`@Bean` 方法 `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` 方法声明为静态可以创建 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。

小费

`spring-boot-actuator` 模块包含一个暴露所有 `@ConfigurationProperties` beans 的端点。将 Web 浏览器指向`/actuator/configprops` 或使用等效的 JMX 端点。有关详细信息，请参阅“ 生产就绪功能 ”部分。

24.8.6 @ConfigurationProperties 与@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：

特征	`@ConfigurationProperties`	`@Value`
Relaxed binding	Yes	No
Meta-data support	Yes	No
`SpEL` evaluation	No	Yes

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带有 `@ConfigurationProperties` 注释的 POJO 中。您还应该知道，由于`@Value` 不支持宽松绑定，因此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中处理此类表达式。

25.简介

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

```
@Configuration
```

```
@Profile("production")
public class ProductionConfiguration {

        // ...

}
```

您可以使用 `spring.profiles.active` `Environment` 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 添加活动配置文件

`spring.profiles.active` 属性遵循与其他属性相同的排序规则：最高 `PropertySource` 获胜。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，将特定于配置文件的属性**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` 入口点还有一个用于设置其他配置文件的 Java API（即，在 `spring.profiles.active` 属性激活的配置文件之上）。请参阅 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
```

注意

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 以编程方式设置配置文件

您可以在应用程序运行之前通过调用 `SpringApplication.setAdditionalProfiles(…)`以编程方式设置活动配置文件。也可以使用 Spring 的 `ConfigurableEnvironment` 界面激活配置文件。

25.3 特定于配置文件的配置文件

`application.properties`（或 `application.yml`）的配置文件特定变体和通过 `@ConfigurationProperties` 引用的文件被视为文件并已加载。有关详细信息，请参见“ 第 24.4 节”“特定于配置文件的属性”。

26.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。

小费

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，并且 Spring Boot 默认值可以正常工作。

26。1 日志格式

Spring Boot 的默认日志输出类似于以下示例：

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main]
org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache
Tomcat/7.0.52
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].
[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1]
o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization
completed in 1358 ms
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1]
o.s.b.c.e.ServletRegistrationBean        : Mapping servlet: 'dispatcherServlet' to
[/]
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1]
o.s.b.c.embedded.FilterRegistrationBean  : Mapping filter: 'hiddenHttpMethodFilter'
to: [/*]
```

输出以下项目：

- 日期和时间：毫秒精度，易于排序。
- 日志级别：ERROR，WARN，INFO，DEBUG 或 TRACE。
- 进程 ID。
- 一个---分隔符，用于区分实际日志消息的开头。
- 线程名称：括在方括号中（可能会截断控制台输出）。
- 记录器名称：这通常是源类名称（通常缩写）。
- 日志消息。

注意

Logback 没有 FATAL 级别。它映射到 ERROR。

26.2 控制台输出

默认日志配置会在写入时将消息回显到控制台。默认情况下，会记录 ERROR - 级别，WARN - 级别和 INFO 级别的消息。您还可以通过使用--debug 标志启动应用程序来启用“调试”模式。

```
$ java -jar myapp.jar --debug
```

注意

您还可以在 `application.properties` 中指定 `debug=true`。

启用调试模式后，将选择一些核心记录器（嵌入式容器，Hibernate 和 Spring Boot）以输出更多信息。启用调试模式并没有将应用程序配置为记录与 `DEBUG` 级别的所有消息。

或者，您可以通过使用`--trace` 标志（或 `application.properties` 中的 `trace=true`）启动应用程序来启用“跟踪”模式。这样做可以为选择的核心记录器（嵌入式容器，Hibernate 模式生成和整个 Spring 组合）启用跟踪日志记录。

26.2.1 彩色编码输出

如果您的终端支持 ANSI，则使用颜色输出来提高可读性。您可以将 `spring.output.ansi.enabled` 设置为 支持的值以覆盖自动检测。

使用`%clr` 转换字配置颜色编码。在最简单的形式中，转换器根据日志级别为输出着色，如以下示例所示：

```
%clr(%5p)
```

下表描述了日志级别到颜色的映射：

水平	颜色
`FATAL`	Red
`ERROR`	Red
`WARN`	Yellow
`INFO`	Green
`DEBUG`	Green
`TRACE`	Green

或者，您可以通过将其作为转换选项指定应使用的颜色或样式。例如，要使文本变为黄色，请使用以下设置：

```
%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}
```

支持以下颜色和样式：

- `blue`
- `cyan`
- `faint`
- `green`
- `magenta`
- `red`
- `yellow`

26.3 文件输出

默认情况下，Spring Boot 仅记录到控制台，不会写入日志文件。如果除了控制台输出之外还要编写日志文件，则需要设置 `logging.file` 或 `logging.path` 属性（例如，在 `application.properties` 中）。

下表显示了 `logging.*`属性如何一起使用：

表 26.1。记录属性

`logging.file`	`logging.path`	例	描述
（没有）	（没有）		仅控制台记录。
具体文件	（没有）	`my.log`	写入指定的日志文件。名称可以是精确位置或相对于当前目录。
（没有）	具体目录	`/var/log`	将 `spring.log` 写入指定的目录。名称可以是精确位置或相对于当前目录。

日志文件在达到 10 MB 时会轮换，与控制台输出一样，默认情况下会记录 `ERROR` - 级别，`WARN` - 级别和 `INFO` 级别的消息。可以使用 `logging.file.max-size` 属性更改大小限制。除非已设置 `logging.file.max-history` 属性，否则以前轮换的文件将无限期归档。

注意

日志记录系统在应用程序生命周期的早期初始化。因此，在通过 `@PropertySource` 注释加载的属性文件中找不到日志记录属性。

小费

日志记录属性独立于实际的日志记录基础结构。因此，Spring Boot 不管理特定的配置密钥（例如用于 Logback 的 `logback.configurationFile`）。

26。4 日志级别

所有受支持的日志记录系统都可以使用 `logging.level.<logger-name>=<level>`在 Spring `Environment` 中设置记录器级别（例如，在 `application.properties` 中），其中 `level` 是 TRACE，DEBUG，INFO 之一，警告，错误，致命或关闭。可以使用 `logging.level.root` 配置 `root` 记录器。

以下示例显示 `application.properties` 中的潜在日志记录设置：

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

26。5 日志组

能够将相关记录器组合在一起以便可以同时配置它们通常很有用。例如，您通常可以更改所有

Tomcat 相关记录器的日志记录级别 ，但您无法轻松记住顶级软件包。

为此，Spring Boot 允许您在 Spring `Environment` 中定义日志记录组。例如，以下是如何通过将“tomcat”组添加到 `application.properties` 来定义它：

```
logging.group.tomcat=org.apache.catalina, org.apache.coyote, org.apache.tomcat
```

定义后，您可以使用一行更改组中所有记录器的级别：

```
logging.level.tomcat=TRACE
```

Spring Boot 包括以下可以开箱即用的预定义日志记录组：

名称	记录仪
web	`org.springframework.core.codec`, `org.springframework.http`, `org.springframework.web`
sql	`org.springframework.jdbc.core`, `org.hibernate.SQL`

26.6 自定义日志配置

可以通过在类路径中包含相应的库来激活各种日志记录系统，并且可以通过在类路径的根目录中或在以下 Spring `Environment` 属性指定的位置提供合适的配置文件来进一步自定义：`logging.config`。

您可以使用 `org.springframework.boot.logging.LoggingSystem` 系统属性强制 Spring Boot 使用特定的日志记录系统。该值应该是 `LoggingSystem` 实现的完全限定类名。您还可以使用值 `none` 完全禁用 Spring Boot 的日志记录配置。

注意

由于**在**创建 `ApplicationContext` **之前**初始化日志记录，因此无法控制 Spring `@Configuration` 文件中`@PropertySources` 的日志记录。更改日志记录系统或完全禁用它的唯一方法是通过系统属性。

根据您的日志记录系统，将加载以下文件：

记录系统	定制
Logback	`logback-spring.xml`, `logback-spring.groovy`, `logback.xml`, or `logback.groovy`
Log4j2	`log4j2-spring.xml` or `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

注意

如果可能，我们建议您使用`-spring` 变体进行日志记录配置（例如，`logback-spring.xml` 而不是 `logback.xml`）。如果使用标准

配置位置，Spring 无法完全控制日志初始化。

警告

Java Util Logging 存在已知的类加载问题，这些问题在从“可执行 jar”运行时会导致问题。如果可能的话，我们建议您在从“可执行 jar”运行时避免使用它。

为了帮助进行自定义，一些其他属性从 Spring `Environment` 传输到系统属性，如下表所述：

Spring 环境	系统 Property	评论
`logging.exception-conversion-word`	`LOG_EXCEPTION_CONVERSION_WORD`	The conversion word used when logging exceptions.
`logging.file`	`LOG_FILE`	If defined, it is used in the default log configuration.
`logging.file.max-size`	`LOG_FILE_MAX_SIZE`	Maximum log file size (if LOG_FILE enabled). (Only supported with the default Logback setup.)
`logging.file.max-history`	`LOG_FILE_MAX_HISTORY`	Maximum number of archive log files to keep (if LOG_FILE enabled). (Only supported with the default Logback setup.)
`logging.path`	`LOG_PATH`	If defined, it is used in the default log configuration.
`logging.pattern.console`	`CONSOLE_LOG_PATTERN`	The log pattern to use on the console (stdout). (Only supported with the default Logback setup.)
`logging.pattern.dateformat`	`LOG_DATEFORMAT_PATTERN`	Appender pattern for log date format. (Only supported with the default Logback setup.)
`logging.pattern.file`	`FILE_LOG_PATTERN`	The log pattern to use in a file (if `LOG_FILE` is enabled). (Only supported with the default Logback setup.)
`logging.pattern.level`	`LOG_LEVEL_PATTERN`	The format to use when rendering the log level (default `%5p`). (Only supported with the default Logback setup.)
`PID`	`PID`	The current process ID (discovered if possible and

Spring 环境	系统 Property	评论
		when not already defined as an OS environment variable).

所有受支持的日志记录系统在解析其配置文件时都可以参考系统属性。有关示例，请参阅 `spring-boot.jar` 中的默认配置：

- 的 logback
- Log4j 2
- Java Util 日志记录

小费

如果要在日志记录属性中使用占位符，则应使用 Spring Boot 的语法，而不是底层框架的语法。值得注意的是，如果使用 Logback，则应使用 `:` 作为属性名称与其默认值之间的分隔符，而不是使用 `:-`。

小费

您可以通过仅覆盖 `LOG_LEVEL_PATTERN`（或带有 Logback 的 `logging.pattern` MDC 和其他临时内容添加到日志行。例如，如果使用 `logging.pattern.level=%X{user} %5p`，则默认日志格式包含“user”的 MDC 条目（如果存在），如以下示

```
2015-09-30 12:30:04.031 user:someone INFO 22174 --- [  nio-8080-exec-0] de
Handling authenticated request
```

26.7 Logback Extensions

Spring Boot 包含许多 Logback 扩展，可以帮助进行高级配置。您可以在 `logback-spring.xml` 配置文件中使用这些扩展名。

注意

由于标准 `logback.xml` 配置文件加载过早，因此无法在其中使用扩展。您需要使用 `logback-spring.xml` 或定义 `logging.config` 属性。

警告

扩展不能与 Logback 的 配置扫描一起使用。如果尝试这样做，则更改配置文件会导致类似于以下记录之一的错误：

```
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for
[springProperty], current ElementPath is [[configuration][springProperty]]
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for
[springProfile], current ElementPath is [[configuration][springProfile]]
```

26.7.1 特定于配置文件的配置

`<springProfile>`标记允许您根据活动的 Spring 配置文件选择性地包含或排除配置部分。`<configuration>`元素中的任何位置都支持配置文件节。使用 `name` 属性指定哪个配置文件接受配置。`<springProfile>`标记可以包含简单的配置文件名称（例如 `staging`）或配置文

件表达式。概要表达式允许表达更复杂的概要逻辑，例如`production & (eu-central | eu-west)`。有关详细信息，请查阅 参考指南。以下清单显示了三个示例配置文件：

```
<springProfile name="staging">
        <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>

<springProfile name="dev | staging">
        <!-- configuration to be enabled when the "dev" or "staging" profiles are
active -->
</springProfile>

<springProfile name="!production">
        <!-- configuration to be enabled when the "production" profile is not active
-->
</springProfile>
```

26.7.2 环境属性

`<springProperty>`标记允许您公开 Spring `Environment` 中的属性，以便在 Logback 中使用。如果要在 Logback 配置中访问 `application.properties` 文件中的值，这样做非常有用。标签的工作方式与 Logback 的标准`<property>`标签类似。但是，不是指定直接 `value`，而是指定属性的 `source`（来自 `Environment`）。如果您需要将属性存储在 `local` 范围以外的其他位置，则可以使用 `scope` 属性。如果需要回退值（如果未在 `Environment` 中设置该属性），则可以使用 `defaultValue` 属性。以下示例显示如何公开在 Logback 中使用的属性：

```
<springProperty scope="context" name="fluentHost" source="myapp.fluentd.host"
                defaultValue="localhost"/>
<appender name="FLUENT" class="ch.qos.logback.more.appenders.DataFluentAppender">
        <remoteHost>${fluentHost}</remoteHost>
        ...
</appender>
```

注意

必须在烤肉串案例中指定 `source`（例如 `my.property-name`）。但是，可以使用宽松规则将属性添加到 `Environment`。

27. JSON

Spring Boot 提供了与三个 JSON 映射库的集成：

- GSON
- Jackson
- JSON-B

Jackson 是首选的默认库。

27.1 Jackson

提供了 Jackson 的自动配置，Jackson 是 `spring-boot-starter-json` 的一部分。当 Jackson 在类路径上时，会自动配置 `ObjectMapper` bean。提供了几个配置属性来自 定义 `ObjectMapper` 的配置。

27.2 Gson

提供 Gson 的自动配置。当 Gson 在类路径上时，会自动配置 `Gson` bean。提供了几个 `spring.gson.*`配置属性来自定义配置。为了获得更多控制，可以使用一个或多个 `GsonBuilderCustomizer` beans。

27.3 JSON-B

提供了 JSON-B 的自动配置。当 JSON-B API 和实现在类路径上时，将自动配置 `Jsonb` bean。首选的 JSON-B 实现是 Apache Johnzon，它提供了依赖关系管理。

28.开发 Web 应用程序

Spring Boot 非常适合 Web 应用程序开发。您可以使用嵌入式 Tomcat，Jetty，Undertow 或 Netty 创建自包含的 HTTP 服务器。大多数 Web 应用程序使用 `spring-boot-starter-web` 模块快速启动和运行。您还可以使用 `spring-boot-starter-webflux` 模块选择构建响应式 Web 应用程序。

如果您还没有开发 Spring Boot Web 应用程序，可以按照“Hello World！”进行操作。“ 入门”部分中的示例 。

28.1“Spring Web MVC 框架”

在 Spring Web 框架（通常简称为“Spring MVC”）是一种富含“模型视图控制器” Web 框架。Spring MVC 允许您创建特殊的`@Controller` 或`@RestController` beans 来处理传入的 HTTP 请求。控制器中的方法使用`@RequestMapping` 注释映射到 HTTP。

以下代码显示了为 JSON 数据提供服务的典型`@RestController`：

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

	@RequestMapping(value="/{user}", method=RequestMethod.GET)
	public User getUser(@PathVariable Long user) {
		// ...
	}

	@RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
	List<Customer> getUserCustomers(@PathVariable Long user) {
		// ...
	}

	@RequestMapping(value="/{user}", method=RequestMethod.DELETE)
	public User deleteUser(@PathVariable Long user) {
		// ...
	}

}
```

Spring MVC 是核心 Spring 框架的一部分，详细信息可在参考文档中找到。还有一些指南涵盖 Spring MVC，可在 spring.io/guides 上找到。

28.1.1 Spring MVC 自动配置

Spring Boot 为 Spring MVC 提供了自动配置，适用于大多数应用程序。

自动配置在 Spring 的默认值之上添加了以下功能：

- 包含 `ContentNegotiatingViewResolver` 和 `BeanNameViewResolver` beans。
- 支持提供静态资源，包括对 WebJars 的支持（ 本文档稍后介绍）） 。
- 自动注册 `Converter`，`GenericConverter` 和 `Formatter` beans。
- 支持 `HttpMessageConverters`（ 本文档稍后部分）。
- 自动注册 `MessageCodesResolver`（ 本文档后面部分）。
- 静态 `index.html` 支持。
- 自定义 `Favicon` 支持（本文档稍后介绍）。
- 自动使用 `ConfigurableWebBindingInitializer` bean（本文 后面会介绍）。

如果你想保留 Spring Boot MVC 功能，并且你想添加额外的 MVC 配置（拦截器，格式化程序，视图控制器和其他功能），你可以添加自己的`@Configuration` 类 `WebMvcConfigurer` 类但 **没有** `@EnableWebMvc`。如果您希望提供 `RequestMappingHandlerMapping`，`RequestMappingHandlerAdapter` 或 `ExceptionHandlerExceptionResolver` 的自定义实例，则可以声明 `WebMvcRegistrationsAdapter` 实例以提供此类组件。

如果您想完全控制 Spring MVC，可以添加自己的`@Configuration` 注释`@EnableWebMvc`。

28.1.2 HttpMessageConverters

Spring MVC 使用 `HttpMessageConverter` 接口转换 HTTP 请求和响应。明智的默认设置包含在开箱即用中。例如，对象可以自动转换为 JSON（通过使用 Jackson 库）或 XML（如果可用，使用 Jackson XML 扩展，或者如果 Jackson XML 扩展不是，则使用 JAXB 可用）。默认情况下，字符串以 `UTF-8` 编码。

如果您需要添加或自定义转换器，可以使用 Spring Boot 的 `HttpMessageConverters` 类，如下面的清单所示：

```
import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration
public class MyConfiguration {

        @Bean
        public HttpMessageConverters customConverters() {
                HttpMessageConverter<?> additional = ...
                HttpMessageConverter<?> another = ...
                return new HttpMessageConverters(additional, another);
        }

}
```

上下文中存在的任何 `HttpMessageConverter` bean 都将添加到转换器列表中。您也可以以相同的方式覆盖默认转换器。

28.1.3 自定义 JSON 序列化程序和反序列化程序

如果使用 Jackson 序列化和反序列化 JSON 数据，您可能需要编写自己的 `JsonSerializer` 和 `JsonDeserializer` 类。自定义序列化程序通常 通过模块注册 Jackson，但 Spring Boot 提供了另一种`@JsonComponent` 注释，可以更容易地直接注册 Spring Beans。

您可以直接在 `JsonSerializer` 或 `JsonDeserializer` 实现上使用`@JsonComponent` 注释。您还可以在包含序列化程序/反序列化程序作为内部类的类上使用它，如以下示例所示：

```
import java.io.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.springframework.boot.jackson.*;

@JsonComponent
public class Example {

        public static class Serializer extends JsonSerializer<SomeObject> {
                // ...
        }

        public static class Deserializer extends JsonDeserializer<SomeObject> {
                // ...
        }

}
```

`ApplicationContext` 中的所有`@JsonComponent` beans 都会自动注册到 Jackson。由于`@JsonComponent` 使用`@Component` 进行元注释，因此通常的组件扫描规则适用。

Spring Boot 还提供 `JsonObjectSerializer` 与 `JsonObjectDeserializer` 该给标准提供有用的替代基类 Jackson 版本序列化对象时。见 `JsonObjectSerializer` 和 `JsonObjectDeserializer` 在 Javadoc 了解详情。

28.1.4 MessageCodesResolver

Spring MVC 有一个生成错误代码的策略，用于从绑定错误中呈现错误消息：`MessageCodesResolver`。如果设置 `spring.mvc.message-codes-resolver.format` 属性 `PREFIX_ERROR_CODE` 或 `POSTFIX_ERROR_CODE`，则 Spring Boot 会为您创建一个（请参阅枚举 `DefaultMessageCodesResolver.Format`）。

28.1.5 静态内容

默认情况下，Spring Boot 从类路径中的`/static`（或`/public` 或`/resources` 或`/META-INF/resources`）目录或 `ServletContext` 的根目录中提供静态内容。它使用来自 Spring MVC 的 `ResourceHttpRequestHandler`，以便您可以通过添加自己的 `WebMvcConfigurer` 并覆盖 `addResourceHandlers` 方法来修改该行为。

在独立的 Web 应用程序中，容器中的默认 servlet 也会启用，并作为后备，如果 Spring 决定不处理它，则从 `ServletContext` 的根目录提供内容。大多数情况下，这不会发生（除非您修改默认的 MVC 配置），因为 Spring 始终可以通过 `DispatcherServlet` 处理请求。

默认情况下，资源映射到`/**`，但您可以使用 `spring.mvc.static-path-pattern` 属性对其进行调整。例如，将所有资源重新定位到`/resources/**`可以实现如下：

```
spring.mvc.static-path-pattern=/resources/**
```

您还可以使用 `spring.resources.static-locations` 属性自定义静态资源位置（将默认值替换为目录位置列表）。根 Servlet 上下文路径`"/"`也会自动添加为位置。

除了前面提到的“标准”静态资源位置之外，还为 Webjars 内容制作了一个特例。如果它们以 Webjars 格式打包，那么具有`/webjars/**`中路径的任何资源都将从 jar 文件中提供。

小费

如果您的应用程序打包为 jar，请不要使用 `src/main/webapp` 目录。

虽然这个目录是一个通用的标准，它的工作原理**只是**战争的包装，它是默默大多数构建工具忽略，如果你生成一个罐子。

Spring Boot 还支持 Spring MVC 提供的高级资源处理功能，允许使用缓存破坏静态资源等用例或使用与 Webjars 无关的 URL。

要为 Webjars 使用版本无关的 URL，请添加 `webjars-locator-core` 依赖项。然后声明你的 Webjar。以 jQuery 为例，添加`"/webjars/jquery/jquery.min.js"`会产生`"/webjars/jquery/x.y.z/jquery.min.js"`。其中 `x.y.z` 是 Webjar 版本。

注意

如果使用 JBoss，则需要声明 `webjars-locator-jboss-vfs` 依赖项而不是 `webjars-locator-core`。否则，所有 Webjars 都将解析为 `404`。

要使用缓存清除，以下配置会为所有静态资源配置缓存清除解决方案，从而在 URL 中有效添加内容哈希（例如`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`）：

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
```

注意

由于为 Thymeleaf 和 FreeMarker 自动配置了 `ResourceUrlEncodingFilter`，因此在运行时可以在模板中重写资源链接。您应该在使用 JSP 时手动声明此过滤器。其他模板引擎目前不是自动支持的，但可以使用自定义模板宏/帮助程序和使用 `ResourceUrlProvider`。

使用（例如）JavaScript 模块加载器动态加载资源时，不能重命名文件。这就是为什么其他策略也得到支持并可以合并的原因。“固定”策略在 URL 中添加静态版本字符串而不更改文件名，如以下示例所示：

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
spring.resources.chain.strategy.fixed.enabled=true
spring.resources.chain.strategy.fixed.paths=/js/lib/
spring.resources.chain.strategy.fixed.version=v12
```

使用此配置，位于`"/js/lib/"`下的 JavaScript 模块使用固定版本控制策略（`"/v12/js/lib/mymodule.js"`），而其他资源仍使用内容 1（`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`）。

有关 `ResourceProperties` 更多支持选项，请参阅

小费

此功能已在专门的博客文章和 Spring Framework 的 参考文档中进行了详细描述 。

28.1.6 欢迎页面

Spring Boot 支持静态和模板化的欢迎页面。它首先在配置的静态内容位置中查找 `index.html` 文件。如果找不到，则会查找 `index` 模板。如果找到任何一个，它将自动用作应用程序的欢迎页面。

28.1.7 自定义 Favicon

Spring Boot 在配置的静态内容位置和类路径的根（按此顺序）中查找 `favicon.ico`。如果存在这样的文件，它将自动用作应用程序的 favicon。

28.1.8 路径匹配和内容协商

Spring MVC 可以通过查看请求路径并将其与应用程序中定义的映射相匹配（例如，关于 Controller 方法的`@GetMapping` 注释），将传入的 HTTP 请求映射到处理程序。

Spring Boot 默认情况下选择禁用后缀模式匹配，这意味着像`"GET /projects/spring-boot.json"`这样的请求将不会与`@GetMapping("/projects/spring-boot")`映射匹配。这被认为是 Spring MVC 应用程序的 最佳实践。对于没有发送正确“接受”请求标头的 HTTP 客户端，此功能在过去主要有用; 我们需要确保将正确的内容类型发送给客户端。如今，内容协商更加可靠。

还有其他方法可以处理不一致发送正确“接受”请求标头的 HTTP 客户端。我们可以使用查询参数来确保`"GET /projects/spring-boot?format=json"`之类的请求映射到`@GetMapping("/projects/spring-boot")`，而不是使用后缀匹配：

```
spring.mvc.contentnegotiation.favor-parameter=true

# We can change the parameter name, which is "format" by default:
# spring.mvc.contentnegotiation.parameter-name=myparam

# We can also register additional file extensions/media types with:
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

如果您了解警告并仍希望您的应用程序使用后缀模式匹配，则需要以下配置：

```
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-suffix-pattern=true
```

或者，不是打开所有后缀模式，而是仅支持已注册的后缀模式更安全：

```
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-registered-suffix-pattern=true

# You can also register additional file extensions/media types with:
# spring.mvc.contentnegotiation.media-types.adoc=text/asciidoc
```

28.1.9 ConfigurableWebBindingInitializer

Spring MVC 使用 `WebBindingInitializer` 为特定请求初始化 `WebDataBinder`。如果您创建自己的 `ConfigurableWebBindingInitializer` `@Bean`，Spring Boot 会自动配置 Spring MVC 以使用它。

28.1.10 模板引擎

除 REST Web 服务外，您还可以使用 Spring MVC 来提供动态 HTML 内容。Spring MVC 支持各种模板技术，包括 Thymeleaf，FreeMarker 和 JSP。此外，许多其他模板引擎包括他们自己的 Spring MVC 集成。

Spring Boot 包括对以下模板引擎的自动配置支持：

- FreeMarker 的
- Groovy 的
- Thymeleaf
- 胡子

小费

如果可能，应该避免使用 JSP。将它们与嵌入式 servlet 容器一起使用时有几个 已知的限制。

当您使用其中一个模板引擎和默认配置时，您的模板将从 `src/main/resources/templates` 自动获取。

小费

根据您运行应用程序的方式，IntelliJ IDEA 以不同方式对类路径进行排序。从主方法在 IDE 中运行应用程序会产生与使用 Maven 或 Gradle 或其打包的 jar 运行应用程序时不同的顺序。这可能导致 Spring Boot 无法在类路径上找到模板。如果遇到此问题，可以在 IDE 中重新排序类路径，以便首先放置模块的类和资源。或者，您可以配置模板前缀以搜索类路径上的每个 `templates` 目录，如下所示：`classpath*:/templates/`。

28.1.11 错误处理

默认情况下，Spring Boot 提供`/error` 映射，以合理的方式处理所有错误，并在 servlet 容器中注册为“全局”错误页面。对于计算机客户端，它会生成一个 JSON 响应，其中包含错误，HTTP 状态和异常消息的详细信息。对于浏览器客户端，有一个“whitelabel”错误视图，以 HTML 格式呈现相同的数据（要自定义它，添加一个解析为 `error` 的 `View`）。要完全替换默认行为，您可以实现 `ErrorController` 并注册该类型的 bean 定义或添加 bean 类型 `ErrorAttributes` 以使用现有机制但替换内容。

小费

`BasicErrorController` 可以用作自定义 `ErrorController` 的基类。如果要为新内容类型添加处理程序，则此功能特别有用（默认情况下，专门处理 `text/html` 并为其他所有内容提供后备）。为此，请扩展 `BasicErrorController`，添加具有 `produces` 属性的 `@RequestMapping` 的公共方法，并创建新类型的 bean。

您还可以定义使用`@ControllerAdvice` 注释的类，以自定义要为特定控制器和/或异常类型返回的 JSON 文档，如以下示例所示：

```
@ControllerAdvice(basePackageClasses = AcmeController.class)
public class AcmeControllerAdvice extends ResponseEntityExceptionHandler {

        @ExceptionHandler(YourException.class)
        @ResponseBody
        ResponseEntity<?> handleControllerException(HttpServletRequest request,
Throwable ex) {
                HttpStatus status = getStatus(request);
                return new ResponseEntity<>(new CustomErrorType(status.value(),
ex.getMessage()), status);
        }
```

```
	private HttpStatus getStatus(HttpServletRequest request) {
		Integer statusCode = (Integer)
request.getAttribute("javax.servlet.error.status_code");
		if (statusCode == null) {
			return HttpStatus.INTERNAL_SERVER_ERROR;
		}
		return HttpStatus.valueOf(statusCode);
	}

}
```

在前面的示例中，如果 `YourException` 在与 `AcmeController` 相同的包中定义的控制器抛出，则使用 `CustomErrorType` POJO 的 JSON 表示而不是 `ErrorAttributes` 表示。

自定义错误页面

如果要显示给定状态代码的自定义 HTML 错误页面，可以将文件添加到`/error` 文件夹。错误页面可以是静态 HTML（即，添加到任何静态资源文件夹下），也可以使用模板构建。文件名应该是确切的状态代码或系列掩码。

例如，要将 `404` 映射到静态 HTML 文件，您的文件夹结构将如下所示：

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

要使用 FreeMarker 模板映射所有 `5xx` 错误，您的文件夹结构如下：

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftl
             +- <other templates>
```

对于更复杂的映射，您还可以添加实现 `ErrorViewResolver` 接口的 beans，如以下示例所示：

```
public class MyErrorViewResolver implements ErrorViewResolver {

	@Override
	public ModelAndView resolveErrorView(HttpServletRequest request,
			HttpStatus status, Map<String, Object> model) {
		// Use the request or status to optionally return a ModelAndView
		return ...
	}

}
```

您还可以使用常规的 Spring MVC 功能，例如 [`@ExceptionHandler` 方法]和 [`@ControllerAdvice`]。`ErrorController` 然后选择任何未处理的异常。

将错误页面映射到 Spring MVC 之外

对于不使用 Spring MVC 的应用程序，可以使用 `ErrorPageRegistrar` 接口直接注册 `ErrorPages`。这种抽象直接与底层嵌入式 servlet 容器一起工作，即使你没有 Spring MVC `DispatcherServlet` 也可以工作。

```
@Bean
public ErrorPageRegistrar errorPageRegistrar(){
        return new MyErrorPageRegistrar();
}

// ...

private static class MyErrorPageRegistrar implements ErrorPageRegistrar {

        @Override
        public void registerErrorPages(ErrorPageRegistry registry) {
                registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST,
"/400"));
        }

}
```

注意

如果你注册 `ErrorPage` 的路径最终由 `Filter` 处理（如某些非 Spring 网页框架，如 Jersey 和 Wicket），那么 `Filter` 必须显式注册为 `ERROR` 调度程序，如以下示例所示：

```
@Bean
public FilterRegistrationBean myFilter() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(new MyFilter());
        ...
        registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
        return registration;
}
```

请注意，默认值 `FilterRegistrationBean` 不包括 `ERROR` 调度程序类型。

小心：当部署到 servlet 容器时，Spring Boot 使用其错误页面过滤器将具有错误状态的请求转发到相应的错误页面。如果尚未提交响应，则只能将请求转发到正确的错误页面。缺省情况下，WebSphere Application Server 8.0 及更高版本在成功完成 servlet 的服务方法后提交响应。您应该通过将 `com.ibm.ws.webcontainer.invokeFlushAfterService` 设置为 `false` 来禁用此行为。

28.1.12 Spring HATEOAS

如果您开发使用超媒体的 RESTful API，Spring Boot 为 Spring HATEOAS 提供了适用于大多数应用程序的自动配置。自动配置取代了使用`@EnableHypermediaSupport` 并注册多个 beans 以简化基于超媒体的应用程序的需求，包括 `LinkDiscoverers`（用于客户端支持）和 `ObjectMapper` 配置为正确地将响应编组到所需的表示中。`ObjectMapper` 是通过设置各种 `spring.jackson.*`属性或（如果存在）`Jackson2ObjectMapperBuilder` bean 来自定义的。

您可以使用`@EnableHypermediaSupport` 控制 Spring HATEOAS 的配置。请注意，这样做会禁用前面描述的 `ObjectMapper` 自定义。

28.1.13 CORS 支持

跨源资源共享 （CORS）是大多数浏览器实现 的 W3C 规范，允许您以灵活的方式指定授权何种跨域请求，而不是使用一些不太安全且功能较弱的方法，如 IFRAME 或 JSONP。

从版本 4.2 开始，Spring MVC 支持 CORS。 在 Spring Boot 应用程序中使用带有 注释的控制器方法 CORS 配置`@CrossOrigin` 不需要任何特定配置。 可以通过使用自定义的 `addCorsMappings(CorsRegistry)`方法注册 `WebMvcConfigurer` bean 来定义全局 CORS 配置，如以下示例所示：

```
@Configuration
public class MyConfiguration {

        @Bean
        public WebMvcConfigurer corsConfigurer() {
                return new WebMvcConfigurer() {
                        @Override
                        public void addCorsMappings(CorsRegistry registry) {
                                registry.addMapping("/api/**");
                        }
                };
        }
}
```

28.2“Spring WebFlux 框架”

Spring WebFlux 是 Spring Framework 5.0 中引入的新的响应式 Web 框架。与 Spring MVC 不同，它不需要 Servlet API，完全异步且无阻塞，并 通过 Reactor 项目实现 Reactive Streams 规范。

Spring WebFlux 有两种版本：功能和注释。基于注释的注释非常接近 Spring MVC 模型，如以下示例所示：

```
@RestController
@RequestMapping("/users")
public class MyRestController {

        @GetMapping("/{user}")
        public Mono<User> getUser(@PathVariable Long user) {
                // ...
        }

        @GetMapping("/{user}/customers")
        public Flux<Customer> getUserCustomers(@PathVariable Long user) {
                // ...
        }

        @DeleteMapping("/{user}")
        public Mono<User> deleteUser(@PathVariable Long user) {
                // ...
        }

}
```

“WebFlux.fn”是功能变体，它将路由配置与请求的实际处理分开，如以下示例所示：

```
@Configuration
public class RoutingConfiguration {

        @Bean
        public RouterFunction<ServerResponse> monoRouterFunction(UserHandler
userHandler) {
                return route(GET("/{user}").and(accept(APPLICATION_JSON)),
userHandler::getUser)
```

```
                                .andRoute(GET("/
{user}/customers").and(accept(APPLICATION_JSON)), userHandler::getUserCustomers)
                                .andRoute(DELETE("/
{user}").and(accept(APPLICATION_JSON)), userHandler::deleteUser);
        }

}

@Component
public class UserHandler {

        public Mono<ServerResponse> getUser(ServerRequest request) {
                // ...
        }

        public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
                // ...
        }

        public Mono<ServerResponse> deleteUser(ServerRequest request) {
                // ...
        }
}
```

WebFlux 是 Spring 框架的一部分，详细信息可在其 参考文档中找到。

小费

您可以根据需要定义尽可能多的 `RouterFunction` beans 来模块化路由器的定义。如果您需要应用优先级，可以订购 Beans。

要开始使用，请将 `spring-boot-starter-webflux` 模块添加到您的应用程序中。

注意

在您的应用程序中添加 `spring-boot-starter-web` 和 `spring-boot-starter-webflux` 模块会导致 Spring Boot 自动配置 Spring MVC，而不是 WebFlux。选择此行为是因为许多 Spring 开发人员将 `spring-boot-starter-webflux` 添加到他们的 Spring MVC 应用程序以使用被动 `WebClient`。您仍然可以通过将所选应用程序类型设置为 `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`来强制执行您的选择。

28.2.1 Spring WebFlux 自动配置

Spring Boot 为 Spring WebFlux 提供自动配置，适用于大多数应用程序。

自动配置在 Spring 的默认值之上添加了以下功能：

- 为 `HttpMessageReader` 和 `HttpMessageWriter` 实例配置编解码器（ 本文档后面会介绍）。
- 支持提供静态资源，包括对 WebJars 的支持（ 本文档后面将介绍）。

如果你想保留 Spring Boot WebFlux 功能，并且想要添加额外的 WebFlux 配置，你可以添加自己的`@Configuration` 类 `WebFluxConfigurer` 但**没有** `@EnableWebFlux`。

如果您想完全控制 Spring WebFlux，可以使用`@EnableWebFlux` 添加自己的`@Configuration`

注释。

28.2.2 带有 HttpMessageReaders 和 HttpMessageWriters 的 HTTP 编解码器

Spring WebFlux 使用 `HttpMessageReader` 和 `HttpMessageWriter` 接口转换 HTTP 请求和响应。通过查看类路径中可用的库，它们配置为 `CodecConfigurer` 以具有合理的默认值。

Spring Boot 通过使用 `CodecCustomizer` 实例进一步自定义。例如，`spring.jackson.*`配置密钥应用于 Jackson 编解码器。

如果需要添加或自定义编解码器，可以创建自定义 `CodecCustomizer` 组件，如以下示例所示：

```
import org.springframework.boot.web.codec.CodecCustomizer;

@Configuration
public class MyConfiguration {

        @Bean
        public CodecCustomizer myCodecCustomizer() {
                return codecConfigurer -> {
                        // ...
                }
        }

}
```

您还可以利用 Boot 的自定义 JSON 序列化程序和反序列化程序。

28.2.3 静态内容

默认情况下，Spring Boot 从类路径中名为`/static`（或`/public` 或`/resources` 或`/META-INF/resources`）的目录中提供静态内容。它使用来自 Spring WebFlux 的 `ResourceWebHandler`，以便您可以通过添加自己的 `WebFluxConfigurer` 并覆盖 `addResourceHandlers` 方法来修改该行为。

默认情况下，资源映射到`/**`，但您可以通过设置 `spring.webflux.static-path-pattern` 属性来调整它。例如，将所有资源重新定位到`/resources/**`可以实现如下：

```
spring.webflux.static-path-pattern=/resources/**
```

您还可以使用 `spring.resources.static-locations` 自定义静态资源位置。这样做会将默认值替换为目录位置列表。如果这样做，默认的欢迎页面检测会切换到您的自定义位置。因此，如果您在启动时的任何位置都有 `index.html`，那么它就是应用程序的主页。

除了前面列出的“标准”静态资源位置之外，还为 Webjars 内容制作了一个特例。如果文件以 Webjars 格式打包，那么具有`/webjars/**`中路径的任何资源都将从 jar 文件中提供。

小费

Spring WebFlux 应用程序并不严格依赖于 Servlet API，因此它们不能作为 war 文件部署，也不能使用 `src/main/webapp` 目录。

28.2.4 模板引擎

除 REST Web 服务外，您还可以使用 Spring WebFlux 来提供动态 HTML 内容。Spring WebFlux 支持各种模板技术，包括 Thymeleaf，FreeMarker 和 Mustache。

Spring Boot 包括对以下模板引擎的自动配置支持：

- FreeMarker 的
- Thymeleaf
- 胡子

当您使用其中一个模板引擎和默认配置时，您的模板将从 `src/main/resources/templates` 自动获取。

28.2.5 错误处理

Spring Boot 提供 `WebExceptionHandler` 以合理的方式处理所有错误。它在处理顺序中的位置紧接在 WebFlux 提供的处理程序之前，这被认为是最后的。对于计算机客户端，它会生成一个 JSON 响应，其中包含错误，HTTP 状态和异常消息的详细信息。对于浏览器客户端，有一个“whitelabel”错误处理程序，它以 HTML 格式呈现相同的数据。您还可以提供自己的 HTML 模板来显示错误（请参阅 下一节）。

自定义此功能的第一步通常涉及使用现有机制，但替换或扩充错误内容。为此，您可以添加 bean 类型 `ErrorAttributes`。

要更改错误处理行为，您可以实现 `ErrorWebExceptionHandler` 并注册该类型的 bean 定义。因为 `WebExceptionHandler` 非常低级，Spring Boot 还提供了一个方便的 `AbstractErrorWebExceptionHandler` 来让你以 WebFlux 函数方式处理错误，如下例所示：

```
public class CustomErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler
{

        // Define constructor here

        @Override
        protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes
errorAttributes) {

                return RouterFunctions
                                .route(aPredicate, aHandler)
                                .andRoute(anotherPredicate, anotherHandler);
        }

}
```

要获得更完整的图片，您还可以直接继承 `DefaultErrorWebExceptionHandler` 并覆盖特定方法。

自定义错误页面

如果要显示给定状态代码的自定义 HTML 错误页面，可以将文件添加到`/error` 文件夹。错误页面可以是静态 HTML（即，添加到任何静态资源文件夹下）或使用模板构建。文件名应该是确切的状态代码或系列掩码。

例如，要将 `404` 映射到静态 HTML 文件，您的文件夹结构将如下所示：

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

要使用 Mustache 模板映射所有 5xx 错误，您的文件夹结构如下：

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.mustache
             +- <other templates>
```

28.2.6 Web 过滤器

Spring WebFlux 提供了一个 `WebFilter` 接口，可以实现过滤 HTTP 请求 - 响应交换。在应用程序上下文中找到的 `WebFilter` beans 将自动用于过滤每个交换。

如果过滤器的顺序很重要，则可以实现 `Ordered` 或使用`@Order` 进行注释。Spring Boot 自动配置可以为您配置 Web 过滤器。执行此操作时，将使用下表中显示的订单：

网络过滤器	订购
`MetricsWebFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`WebFilterChainProxy` (Spring Security)	`-100`
`HttpTraceWebFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

28.3 JAX-RS 和 Jersey

如果您更喜欢 REST 端点的 JAX-RS 编程模型，则可以使用其中一个可用的实现而不是 Spring MVC。Jersey 和 Apache CXF 开箱即用。CXF 要求您在应用程序上下文中将 `Servlet` 或 `Filter` 注册为`@Bean`。Jersey 具有一些本地 Spring 支持，因此我们还在 Spring Boot 中为其提供了自动配置支持以及启动器。

要开始使用 Jersey，请将 `spring-boot-starter-jersey` 作为依赖项包含在内，然后需要一个`@Bean` 类型 `ResourceConfig`，在其中注册所有端点，如以下示例所示：

```
@Component
public class JerseyConfig extends ResourceConfig {

        public JerseyConfig() {
                register(Endpoint.class);
        }

}
```

警告

Jersey 对扫描可执行档案的支持相当有限。例如，它无法扫描完全可执行 jar 文件中的包中的端点，也无法在运行可执行 war 文件时扫描 `WEB-INF/classes` 中的端点。为避免此限制，不应使用 `packages` 方法，并且应使用 `register` 方法单独注册端点，如上例所示。

对于更高级的自定义，您还可以注册实现 `ResourceConfigCustomizer` 的任意数量的 beans。

所有已注册的端点都应为@Components，并带有 HTTP 资源注释（@GET 和其他），如以下示例所示：

```
@Component
@Path("/hello")
public class Endpoint {

        @GET
        public String message() {
                return "Hello";
        }

}
```

由于 Endpoint 是 Spring @Component，其生命周期由 Spring 管理，您可以使用@Autowired 注释注入依赖项并使用@Value 注释注入外部组态。默认情况下，Jersey servlet 已注册并映射到/*。您可以通过将@ApplicationPath 添加到 ResourceConfig 来更改映射。

默认情况下，Jersey 设置为名为 jerseyServletRegistration 的 ServletRegistrationBean 类型的@Bean 中的 Servlet。默认情况下，servlet 是懒惰地初始化的，但您可以通过设置 spring.jersey.servlet.load-on-startup 来自定义该行为。您可以通过创建一个具有相同名称的自己来禁用或覆盖 bean。您也可以通过设置 spring.jersey.type=filter 来使用过滤器而不是 servlet（在这种情况下，要替换或覆盖的@Bean 为 jerseyFilterRegistration）。过滤器的@Order，您可以使用 spring.jersey.filter.order 进行设置。通过使用 spring.jersey.init.*指定属性映射，可以为 servlet 和过滤器注册提供 init 参数。

有一个 Jersey 样本，以便您可以看到如何设置。

28.4 嵌入式 Servlet 容器支持

Spring Boot 包括对嵌入式 Tomcat， Jetty 和 Undertow 服务器的支持。大多数开发人员使用适当的“Starter”来获取完全配置的实例。默认情况下，嵌入式服务器侦听端口 8080 上的 HTTP 请求。

警告

如果您选择在 CentOS 上使用 Tomcat ，请注意，默认情况下，临时目录用于存储已编译的 JSP，文件上载等。当您的应用程序运行时，tmpwatch 可能会删除此目录，从而导致失败。要避免此行为，您可能希望自定义 tmpwatch 配置，以便不删除 tomcat.*目录或配置 server.tomcat.basedir，以便嵌入式 Tomcat 使用不同的位置。

28.4.1 Servlet，过滤器和监听器

使用嵌入式 servlet 容器时，可以使用 Spring beans 或扫描 Servlet 组件，从 Servlet 规范中注册 servlet，过滤器和所有侦听器（例如 HttpSessionListener）。

注册 Servlet，过滤器和监听器 Spring Beans

在嵌入式容器中注册了 Spring bean 的任何 Servlet，Filter 或 servlet *Listener 实例。如果要在配置期间引用 application.properties 中的值，这可能特别方便。

默认情况下，如果上下文仅包含一个 Servlet，则它将映射到/。在多个 servlet beans 的情况下，bean 名称用作路径前缀。过滤器映射到/*。

如果基于约定的映射不够灵活，您可以使用 `ServletRegistrationBean`，`FilterRegistrationBean` 和 `ServletListenerRegistrationBean` 类进行完全控制。

Spring Boot 附带了许多可以定义 Filter beans 的自动配置。以下是过滤器及其各自顺序的一些示例（较低的顺序值表示较高的优先级）：

Servlet 过滤器	订购
`OrderedCharacterEncodingFilter`	`Ordered.HIGHEST_PRECEDENCE`
`WebMvcMetricsFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`ErrorPageFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`HttpTraceFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

将 Filter beans 无序放置通常是安全的。

如果需要特定订单，则应避免在 `Ordered.HIGHEST_PRECEDENCE` 处配置读取请求正文的筛选器，因为它可能违反应用程序的字符编码配置。如果 Servlet 过滤器包装请求，则应使用小于或等于 `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER` 的顺序进行配置。

28.4.2 Servlet 上下文初始化

嵌入式 servlet 容器不直接执行 Servlet 3.0+ `javax.servlet.ServletContainerInitializer` 接口或 Spring 的 `org.springframework.web.WebApplicationInitializer` 接口。这是一项有意的设计决策，旨在降低设计在战争中运行的第三方图书馆可能会破坏 Spring Boot 应用程序的风险。

如果需要在 Spring Boot 应用程序中执行 servlet 上下文初始化，则应注册实现 `org.springframework.boot.web.servlet.ServletContextInitializer` 接口的 bean。单 `onStartup` 方法提供对 `ServletContext` 的访问，如果需要，可以很容易地用作现有 `WebApplicationInitializer` 的适配器。

扫描 Servlet，过滤器和侦听器

使用嵌入式容器时，可以使用`@ServletComponentScan` 启用使用 `@WebServlet`，`@WebFilter` 和`@WebListener` 注释的类的自动注册。

小费

`@ServletComponentScan` 对独立容器没有影响，其中使用容器的内置发现机制。

28.4.3 ServletWebServerApplicationContext

在引擎盖下，Spring Boot 使用不同类型的 `ApplicationContext` 来支持嵌入式 servlet 容器。`ServletWebServerApplicationContext` 是 `WebApplicationContext` 的一种特殊类型，它通过搜索单个 `ServletWebServerFactory` bean 来引导自己。通常会自动配置 `TomcatServletWebServerFactory`，`JettyServletWebServerFactory` 或 `UndertowServletWebServerFactory`。

注意

您通常不需要了解这些实现类。大多数应用程序都是自动配置的，并且代表您创建了相应的 `ApplicationContext` 和 `ServletWebServerFactory`。

28.4.4 自定义嵌入式 Servlet 容器

可以使用 Spring `Environment` 属性配置公共 servlet 容器设置。通常，您将在 `application.properties` 文件中定义属性。

常用服务器设置包括：

- 网络设置：侦听传入 HTTP 请求的端口（`server.port`），绑定到 `server.address` 的接口地址，依此类推。
- 会话设置：会话是持久的（`server.servlet.session.persistence`），会话超时（`server.servlet.session.timeout`），会话数据的位置（`server.servlet.session.store-dir`）和会话 cookie 配置（`server.servlet.session.cookie.*`）。
- 错误管理：错误页面的位置（`server.error.path`）等。
- SSL
- HTTP 压缩

Spring Boot 尝试尽可能多地暴露常见设置，但这并非总是可行。对于这些情况，专用命名空间提供特定于服务器的自定义（请参阅 `server.tomcat` 和 `server.undertow`）。例如， 可以使用嵌入式 servlet 容器的特定功能配置访问日志。

小费

请参阅 `ServerProperties` 课程以获取完整列表。

程序化定制

如果需要以编程方式配置嵌入式 servlet 容器，可以注册实现 `WebServerFactoryCustomizer` 接口的 Spring bean。`WebServerFactoryCustomizer` 提供对 `ConfigurableServletWebServerFactory` 的访问，其中包括许多自定义 setter 方法。以下示例以编程方式设置端口：

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import
org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

        @Override
        public void customize(ConfigurableServletWebServerFactory server) {
                server.setPort(9000);
        }

}
```

注意

`TomcatServletWebServerFactory`，`JettyServletWebServerFactory` 和 `UndertowServletWebServerFactory` 是 `ConfigurableServletWebServerFactory` 的专用变体，分别为 Tomcat，Jetty 和 Undertow 提供了额外的自定义 setter 方法。

直接自定义 ConfigurableServletWebServerFactory

如果前面的自定义技术太有限，您可以自己注册 `TomcatServletWebServerFactory`，`JettyServletWebServerFactory` 或 `UndertowServletWebServerFactory` bean。

```
@Bean
public ConfigurableServletWebServerFactory webServerFactory() {
        TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory();
        factory.setPort(9000);
        factory.setSessionTimeout(10, TimeUnit.MINUTES);
        factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, "/notfound.html"));
        return factory;
}
```

为许多配置选项提供了 Setter。如果您需要做一些更具异国情调的事情，还会提供一些受保护的方法“挂钩”。有关详细信息，请参阅 源代码文档。

28.4.5 JSP 限制

运行使用嵌入式 servlet 容器的 Spring Boot 应用程序（并打包为可执行存档）时，JSP 支持存在一些限制。

- 使用 Jetty 和 Tomcat，如果使用 war 包装，它应该可以工作。使用 `java -jar` 启动时，可执行战争将起作用，并且还可以部署到任何标准容器。使用可执行 jar 时不支持 JSP。
- Undertow 不支持 JSP。
- 创建自定义 `error.jsp` 页面不会覆盖错误处理的默认视图 。 应该使用自定义错误页面。

有一个 JSP 示例，以便您可以看到如何设置。

28.5 嵌入式 Reactive Server 支持

Spring Boot 包括对以下嵌入式响应式 Web 服务器的支持：Reactor Netty，Tomcat，Jetty 和 Undertow。大多数开发人员使用适当的“Starter”来获取完全配置的实例。默认情况下，嵌入式服务器在端口 8080 上侦听 HTTP 请求。

28.6 Reactive Server 资源配置

在自动配置 Reactor Netty 或 Jetty 服务器时，Spring Boot 将创建特定的 beans，它将为服务器实例提供 HTTP 资源：`ReactorResourceFactory` 或 `JettyResourceFactory`。

默认情况下，这些资源也将与 Reactor Netty 和 Jetty 客户端共享以获得最佳性能，具体如下：

- 相同的技术用于服务器和客户端
- 客户端实例使用由 Spring Boot 自动配置的 `WebClient.Builder` bean 构建

开发人员可以通过提供自定义 `ReactorResourceFactory` 或 `JettyResourceFactory` bean 覆盖 Jetty 和 Reactor Netty 的资源配置 - 这将应用于客户端和服务器。

您可以在 WebClient Runtime 部分中了解有关客户端资源配置的更多信息 。

29.安全

如果 Spring 安全性在类路径上，则默认情况下 Web 应用程序是安全的。Spring Boot 依赖于 Spring 安全性的内容协商策略来确定是使用 `httpBasic` 还是 `formLogin`。要向 Web 应用程序添加方法级安全性，您还可以使用所需设置添加`@EnableGlobalMethodSecurity`。其他信息可在 Spring 安全参考指南中找到。

默认 `UserDetailsService` 只有一个用户。用户名为 `user`，密码是随机的，在应用程序启动时以 INFO 级别打印，如以下示例所示：

```
Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35
```

注意

如果您对日志记录配置进行微调，请确保将 `org.springframework.boot.autoconfigure.security` 类别设置为记录 `INFO` 级别的消息。否则，不会打印默认密码。

您可以通过提供 `spring.security.user.name` 和 `spring.security.user.password` 来更改用户名和密码。

您在 Web 应用程序中默认获得的基本功能包括：

- `UserDetailsService`（对于 WebFlux 应用程序，为 `ReactiveUserDetailsService`）bean 具有内存存储，单个用户具有生成的密码（请参阅 `SecurityProperties.User` 用户的属性）。
- 基于表单的登录或 HTTP 基本安全性（取决于 Content-Type），用于整个应用程序（如果执行器在类路径上，则包括执行器端点）。
- 用于发布身份验证事件的 `DefaultAuthenticationEventPublisher`。

您可以为其添加 bean 来提供不同的 `AuthenticationEventPublisher`。

29.1 MVC 安全性

默认安全配置在 `SecurityAutoConfiguration` 和 `UserDetailsServiceAutoConfiguration` 中实现。`SecurityAutoConfiguration` 导入用于 Web 安全的 `SpringBootWebSecurityConfiguration` 和用于配置身份验证的 `UserDetailsServiceAutoConfiguration`，这在非 Web 应用程序中也是相关的。要完全关闭默认 Web 应用程序安全配置，您可以添加 bean 类型 `WebSecurityConfigurerAdapter`（这样做不会禁用 `UserDetailsService` 配置或 Actuator 的安全性）。

要同时关闭 `UserDetailsService` 配置，您可以添加 bean 类型 `UserDetailsService`，`AuthenticationProvider` 或 `AuthenticationManager`。Spring Boot 示例中有几个安全应用程序可以帮助您开始使用常见用例。

可以通过添加自定义 `WebSecurityConfigurerAdapter` 来覆盖访问规则。Spring Boot 提供了便捷方法，可用于覆盖执行器端点和静态资源的访问规则。`EndpointRequest` 可用于创建基于 `management.endpoints.web.base-path` 属性的 `RequestMatcher`。`PathRequest` 可用于为常用位置的资源创建 `RequestMatcher`。

29.2 WebFlux 安全性

与 Spring MVC 应用程序类似，您可以通过添加 `spring-boot-starter-security` 依赖项来保护 WebFlux 应用程序。默认安全配置在 `ReactiveSecurityAutoConfiguration` 和 `UserDetailsServiceAutoConfiguration` 中实现。`ReactiveSecurityAutoConfiguration` 导入用于 Web 安全的 `WebFluxSecurityConfiguration` 和用于配置身份验证的 `UserDetailsServiceAutoConfiguration`，这在非 Web 应用程序中也是相关的。要完全关闭默认 Web 应用程序安全配置，您可以添加 bean 类型 `WebFilterChainProxy`（这样做不会禁用 `UserDetailsService` 配置或执行器的安全性）。

要同时关闭 `UserDetailsService` 配置，您可以添加 bean 类型 `ReactiveUserDetailsService` 或 `ReactiveAuthenticationManager`。

可以通过添加自定义 `SecurityWebFilterChain` 来配置访问规则。Spring Boot 提供了便捷方法，可用于覆盖执行器端点和静态资源的访问规则。`EndpointRequest` 可用于创建基于 `management.endpoints.web.base-path` 属性的 `ServerWebExchangeMatcher`。

`PathRequest` 可用于为常用位置的资源创建 `ServerWebExchangeMatcher`。

例如，您可以通过添加以下内容来自定义安全配置：

```
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        return http
                .authorizeExchange()

.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
                        .pathMatchers("/foo", "/bar")
                                .authenticated().and()
                        .formLogin().and()
                .build();
}
```

29.3 OAuth2

OAuth2 是 Spring 支持的广泛使用的授权框架。

29.3.1 客户

如果您的类路径上有 `spring-security-oauth2-client`，则可以利用一些自动配置来轻松设置 OAuth2 / Open ID Connect 客户端。此配置使用 `OAuth2ClientProperties` 下的属性。相同的属性适用于 servlet 和反应应用程序。

您可以在 `spring.security.oauth2.client` 前缀下注册多个 OAuth2 客户端和提供商，如以下示例所示：

```
spring.security.oauth2.client.registration.my-client-1.client-id=abcd
spring.security.oauth2.client.registration.my-client-1.client-secret=password
spring.security.oauth2.client.registration.my-client-1.client-name=Client for user
scope
spring.security.oauth2.client.registration.my-client-1.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-1.scope=user
spring.security.oauth2.client.registration.my-client-1.redirect-uri-
template=http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-1.client-authentication-
method=basic
spring.security.oauth2.client.registration.my-client-1.authorization-grant-
```

```
type=authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id=abcd
spring.security.oauth2.client.registration.my-client-2.client-secret=password
spring.security.oauth2.client.registration.my-client-2.client-name=Client for email
scope
spring.security.oauth2.client.registration.my-client-2.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-2.scope=email
spring.security.oauth2.client.registration.my-client-2.redirect-uri-
template=http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-2.client-authentication-
method=basic
spring.security.oauth2.client.registration.my-client-2.authorization-grant-
type=authorization_code

spring.security.oauth2.client.provider.my-oauth-provider.authorization-uri=http://my-
auth-server/oauth/authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-uri=http://my-auth-
server/oauth/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri=http://my-
auth-server/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.user-info-authentication-
method=header
spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-uri=http://my-auth-
server/token_keys
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
```

对于支持 OpenID Connect 发现的 OpenID Connect 提供程序，可以进一步简化配置。提供程序需要配置 `issuer-uri`，这是它声明为其颁发者标识符的 URI。例如，如果提供的 `issuer-uri` 为“https://example.com”，则会将 `OpenID Provider Configuration Request` 设置为“https://example.com/.well-known/openid-configuration”。结果预计为 `OpenID Provider Configuration Response`。以下示例显示如何使用 `issuer-uri` 配置 OpenID Connect Provider：

```
spring.security.oauth2.client.provider.oidc-provider.issuer-uri=https://dev-
123456.oktapreview.com/oauth2/default/
```

默认情况下，Spring 安全性 `OAuth2LoginAuthenticationFilter` 仅处理与`/login/oauth2/code/*`匹配的网址。如果要自定义 `redirect-uri` 以使用其他模式，则需要提供配置以处理该自定义模式。例如，对于 servlet 应用程序，您可以添加类似于以下内容的 `WebSecurityConfigurerAdapter`：

```
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .authorizeRequests()
                                .anyRequest().authenticated()
                                .and()
                        .oauth2Login()
                                .redirectionEndpoint()
                                        .baseUri("/custom-callback");
        }
}
```

OAuth2 共同提供者的客户注册

对于常见的 OAuth2 和 OpenID 提供商，包括 Google，Github，Facebook 和 Okta，我们提供了一组提供商默认值（分别为 `google`，`github`，`facebook` 和 `okta`）。

如果您不需要自定义这些提供程序，则可以将 `provider` 属性设置为您需要推断默认值的属性。此外，如果客户端注册的密钥与默认支持的提供者匹配，则 Spring Boot 也会推断出。

换句话说，以下示例中的两个配置使用 Google 提供程序：

```
spring.security.oauth2.client.registration.my-client.client-id=abcd
spring.security.oauth2.client.registration.my-client.client-secret=password
spring.security.oauth2.client.registration.my-client.provider=google

spring.security.oauth2.client.registration.google.client-id=abcd
spring.security.oauth2.client.registration.google.client-secret=password
```

29.3.2 资源服务器

如果类路径上有 `spring-security-oauth2-resource-server`，只要指定了 JWK Set URI 或 OIDC Issuer URI，Spring Boot 就可以设置 OAuth2 资源服务器，如以下示例所示：

```
spring.security.oauth2.resourceserver.jwt.jwk-set-
uri=https://example.com/oauth2/default/v1/keys

spring.security.oauth2.resourceserver.jwt.issuer-uri=https://dev-
123456.oktapreview.com/oauth2/default/
```

相同的属性适用于 servlet 和反应应用程序。

或者，您可以为 servlet 应用程序定义自己的 `JwtDecoder` bean，或者为响应式应用程序定义 `ReactiveJwtDecoder`。

29.3.3 授权服务器

目前，Spring 安全性不支持实施 OAuth 2.0 授权服务器。但是，此功能可从 Spring 安全 OAuth 项目获得，该项目最终将完全被 Spring 安全性取代。在此之前，您可以使用 `spring-security-oauth2-autoconfigure` 模块轻松设置 OAuth 2.0 授权服务器; 请参阅其文档以获取说明

29.4 执行器安全性

出于安全考虑，默认情况下禁用 `/health` 和 `/info` 以外的所有执行器。`management.endpoints.web.exposure.include` 属性可用于启用执行器。

如果 Spring 安全性在类路径上且没有其他 WebSecurityConfigurerAdapter 存在，则 `/health` 和 `/info` 以外的所有执行器都由 Spring Boot 自动配置保护。如果您定义自定义 `WebSecurityConfigurerAdapter`，则 Spring Boot 自动配置将退回，您将完全控制执行器访问规则。

注意

在设置 `management.endpoints.web.exposure.include` 之前，请确保暴露的执行器不包含敏感信息和/或通过将它们放在防火墙后面或通过 Spring 安全性等方式进行保护。

29.4.1 跨站点请求伪造保护

由于 Spring Boot 依赖于 Spring 安全性的默认值，因此默认情况下会启用 CSRF 保护。这意味着当使用默认安全配置时，需要 `POST`（关闭和记录器端点），`PUT` 或 `DELETE` 的执行器端点将获得 403 禁止错误。

注意

我们建议仅在创建非浏览器客户端使用的服务时才完全禁用 CSRF 保护。

有关 CSRF 保护的其他信息，请参阅 Spring 安全参考指南。

30.使用 SQL 数据库

在 Spring 框架提供用于使用 `JdbcTemplate` 完成“对象关系映射”的技术，如休眠使用 SQL 数据库，从直接 JDBC 访问广泛的支持。Spring 数据提供了更多级别的功能：直接从接口创建 `Repository` 实现，并使用约定从方法名称生成查询。

30.1 配置数据源

Java 的 `javax.sql.DataSource` 接口提供了一种使用数据库连接的标准方法。传统上，'DataSource'使用 `URL` 以及一些凭据来建立数据库连接。

小费

有关更多高级示例，请参阅“操作方法”部分，通常是为了完全控制 DataSource 的配置。

30.1.1 嵌入式数据库支持

通过使用内存中嵌入式数据库来开发应用程序通常很方便。显然，内存数据库不提供持久存储。您需要在应用程序启动时填充数据库，并准备在应用程序结束时丢弃数据。

小费

“操作方法”部分包含有关如何初始化数据库的部分。

Spring Boot 可以自动配置嵌入式 H2， HSQL 和 Derby 数据库。您无需提供任何连接 URL。您只需要包含要使用的嵌入式数据库的构建依赖项。

注意

如果您在测试中使用此功能，您可能会注意到整个测试套件都会重复使用相同的数据库，无论您使用的应用程序上下文的数量如何。如果要确保每个上下文都有一个单独的嵌入式数据库，则应将 `spring.datasource.generate-unique-name` 设置为 `true`。

例如，典型的 POM 依赖关系如下：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <scope>runtime</scope>
</dependency>
```

注意

您需要依赖 `spring-jdbc` 才能自动配置嵌入式数据库。在这个例子中，它通过 `spring-boot-starter-data-jpa` 传递。

小费

如果由于某种原因，您确实为嵌入式数据库配置了连接 URL，请注意确保禁用数据库的自动关闭。如果您使用 H2，则应使用 `DB_CLOSE_ON_EXIT=FALSE` 来执行此操作。如果使用 HSQLDB，则应确保未使用 `shutdown=true`。禁用数据库的自动关闭允许 Spring Boot 控制数据库何时关闭，从而确保在不再需要访问数据库时发生这种情况。

30.1.2 连接到生产数据库

也可以使用池 `DataSource` 自动配置生产数据库连接。Spring Boot 使用以下算法选择特定实现：

1. 我们更喜欢 HikariCP 的性能和并发性。如果 HikariCP 可用，我们总是选择它。
2. 否则，如果 Tomcat 池 `DataSource` 可用，我们将使用它。
3. 如果 HikariCP 和 Tomcat 池化数据源都不可用，并且 Commons DBCP2 可用，我们就会使用它。

如果您使用 `spring-boot-starter-jdbc` 或 `spring-boot-starter-data-jpa`“starters”，则会自动获得 `HikariCP` 的依赖关系。

注意

您可以完全绕过该算法，并通过设置 `spring.datasource.type` 属性指定要使用的连接池。如果您在 Tomcat 容器中运行应用程序，这一点尤为重要，因为默认情况下会提供 `tomcat-jdbc`。

小费

始终可以手动配置其他连接池。如果您定义自己的 `DataSource` bean，则不会进行自动配置。

DataSource 配置由 `spring.datasource.*`中的外部配置属性控制。例如，您可以在 `application.properties` 中声明以下部分：

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

注意

您至少应该通过设置 `spring.datasource.url` 属性来指定 URL。否则，Spring Boot 会尝试自动配置嵌入式数据库。

小费

您通常不需要指定 `driver-class-name`，因为 Spring Boot 可以从 `url` 中为大多数数据库推断出它。

注意

对于要创建的池 DataSource，我们需要能够验证有效的 Driver 类是否可用，因此我们在执行任何操作之前检查它。换句话说，如果设置 spring.datasource.driver-class-name=com.mysql.jdbc.Driver，那么该类必须是可加载的。

有关 DataSourceProperties 更多支持的选项，请参阅 。无论实际实施如何，这些都是标准选项。还可以使用各自的前缀（spring.datasource.hikari.*，spring.datasource.tomcat.*和 spring.datasource.dbcp2.*）来微调特定于实现的设置。有关更多详细信息，请参阅您正在使用的连接池实现的文档。

例如，如果使用 Tomcat 连接池，则可以自定义许多其他设置，如以下示例所示：

```
# Number of ms to wait before throwing an exception if no connection is available.
spring.datasource.tomcat.max-wait=10000

# Maximum number of active connections that can be allocated from this pool at the
same time.
spring.datasource.tomcat.max-active=50

# Validate the connection before borrowing it from the pool.
spring.datasource.tomcat.test-on-borrow=true
```

30.1.3 连接到 JNDI 数据源

如果将 Spring Boot 应用程序部署到 Application Server，则可能希望使用 Application Server 的内置功能配置和管理 DataSource，并使用 JNDI 访问它。

spring.datasource.jndi-name 属性可用作 spring.datasource.url，spring.datasource.username 和 spring.datasource.password 属性的替代，以从特定 JNDI 位置访问 DataSource。例如，application.properties 中的以下部分显示了如何访问定义的 DataSource JBoss AS：

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

30.2 使用 JdbcTemplate

Spring 的 JdbcTemplate 和 NamedParameterJdbcTemplate 类是自动配置的，您可以 @Autowire 直接将它们放入您自己的 beans 中，如以下示例所示：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

        private final JdbcTemplate jdbcTemplate;

        @Autowired
        public MyBean(JdbcTemplate jdbcTemplate) {
                this.jdbcTemplate = jdbcTemplate;
        }

        // ...
```

```
}
```

您可以使用 `spring.jdbc.template.*`属性自定义模板的某些属性，如以下示例所示：

```
spring.jdbc.template.max-rows=500
```

注意

`NamedParameterJdbcTemplate` 在幕后重用了相同的 `JdbcTemplate` 实例。如果定义了多个 `JdbcTemplate` 并且不存在主要候选者，则不会自动配置 `NamedParameterJdbcTemplate`。

30.3 JPA 和 Spring 数据 JPA

Java Persistence API 是一种标准技术，可让您将对象“映射”到关系数据库。`spring-boot-starter-data-jpa` POM 提供了一种快速入门方式。它提供以下关键依赖项：

- Hibernate：最受欢迎的 JPA 实现之一。
- Spring Data JPA：使实现基于 JPA 的存储库变得容易。
- Spring ORM：来自 Spring 框架的核心 ORM 支持。

小费

我们不会在这里详细介绍 JPA 或 Spring 数据。您可以按照“访问数据与 JPA” 从指导 spring.io 和读取 Spring 数据 JPA 和 Hibernate 的参考文档。

30.3.1 实体类

传统上，JPA“实体”类在 `persistence.xml` 文件中指定。使用 Spring Boot 时，不需要此文件，而是使用“实体扫描”。默认情况下，将搜索主配置类（注释为`@EnableAutoConfiguration` 或`@SpringBootApplication`）下的所有包。

任何注明`@Entity`，`@Embeddable` 或`@MappedSuperclass` 的类都会被考虑。典型的实体类类似于以下示例：

```
package com.example.myapp.domain;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class City implements Serializable {

        @Id
        @GeneratedValue
        private Long id;

        @Column(nullable = false)
        private String name;

        @Column(nullable = false)
        private String state;

        // ... additional members, often include @OneToMany mappings

        protected City() {
                // no-args constructor required by JPA spec
                // this one is protected since it shouldn't be used directly
```

```
        }

        public City(String name, String state) {
                this.name = name;
                this.state = state;
        }

        public String getName() {
                return this.name;
        }

        public String getState() {
                return this.state;
        }

        // ... etc

}
```

小费

您可以使用@EntityScan 注释自定义实体扫描位置。请参阅“ 第 84.4 节 ”，“Spring 配置”中的@Entity 定义“ ”操作方法。

30.3.2 Spring 数据 JPA 存储库

Spring 数据 JPA 存储库是您可以定义以访问数据的接口。JPA 查询是从您的方法名称自动创建的。例如，CityRepository 接口可能会声明 findAllByState(String state)方法来查找给定状态中的所有城市。

对于更复杂的查询，您可以使用 Spring Data 的 Query 注释来注释您的方法 。

Spring 数据存储库通常从 Repository 或 CrudRepository 接口扩展 。如果使用自动配置，则会从包含主配置类（使用@EnableAutoConfiguration 或@SpringBootApplication 注释的包）的包中搜索存储库。

以下示例显示了典型的 Spring 数据存储库接口定义：

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

        Page<City> findAll(Pageable pageable);

        City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

Spring 数据 JPA 存储库支持三种不同的引导模式：default，deferred 和 lazy。要启用延迟或延迟引导，请将 spring.data.jpa.repositories.bootstrap-mode 分别设置为 deferred 或 lazy。使用延迟或延迟引导时，自动配置的 EntityManagerFactoryBuilder 将使用上下文的异步任务执行程序（如果有）作为引导程序执行程序。

小费

我们几乎没有触及 Spring Data JPA 的表面。有关完整的详细信息，请参

阅 Spring Data JPA 参考文档。

30.3.3 创建和删除 JPA 数据库

默认情况下，**仅**当您使用嵌入式数据库（H2，HSQL 或 Derby）时，**才会**自动创建 JPA 数据库。您可以使用 `spring.jpa.*`属性显式配置 JPA 设置。例如，要创建和删除表，可以将以下行添加到 `application.properties`：

```
spring.jpa.hibernate.ddl-auto=create-drop
```

注意

Hibernate 自己的内部属性名称（如果你碰巧更好地记住它）是 `hibernate.hbm2ddl.auto`。您可以使用 `spring.jpa.properties.*`（在将它们添加到实体管理器之前删除前缀）来设置它以及其他 Hibernate 本机属性。以下行显示了为 Hibernate 设置 JPA 属性的示例：

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

前面示例中的行将 `hibernate.globally_quoted_identifiers` 属性的值 `true` 传递给 Hibernate 实体管理器。

默认情况下，DDL 执行（或验证）将延迟到 `ApplicationContext` 开始。还有一个 `spring.jpa.generate-ddl` 标志，但如果 Hibernate 自动配置处于活动状态，则不会使用它，因为 `ddl-auto` 设置更精细。

30.3.4 在 View 中打开 EntityManager

如果您正在运行 Web 应用程序，则默认情况下 Spring Boot 会注册 `OpenEntityManagerInViewInterceptor` 以应用“在视图中打开 EntityManager”模式，以允许在 Web 视图中进行延迟加载。如果您不想要此行为，则应在 `application.properties` 中将 `spring.jpa.open-in-view` 设置为 `false`。

30.4 Spring 数据 JDBC

Spring 数据包括 JDBC 的存储库支持，并将自动为 `CrudRepository` 上的方法生成 SQL。对于更高级的查询，提供了`@Query` 注释。

当必要的依赖项在类路径上时，Spring Boot 将自动配置 Spring 数据的 JDBC 存储库。可以使用 `spring-boot-starter-data-jdbc` 上的单个依赖项将它们添加到项目中。如有必要，您可以通过向应用程序添加`@EnableJdbcRepositories` 注释或 `JdbcConfiguration` 子类来控制 Spring Data JDBC 的配置。

小费

有关 Spring 数据 JDBC 的完整详细信息，请参阅 参考文档。

30.5 使用 H2 的 Web 控制台

该 H2 数据库提供了一个 基于浏览器的控制台是 Spring Boot 可以自动为您配置。满足以下条件

时，将自动配置控制台：

- 您正在开发基于 servlet 的 Web 应用程序。
- `com.h2database:h2` 在类路径上。
- 您正在使用 Spring Boot 的开发人员工具。

小费

如果您没有使用 Spring Boot 的开发人员工具但仍想使用 H2 的控制台，则可以使用值 `true` 配置 `spring.h2.console.enabled` 属性。

注意

H2 控制台仅用于开发期间，因此您应该注意确保生产中 `spring.h2.console.enabled` 未设置为 `true`。

30.5.1 更改 H2 控制台的路径

默认情况下，控制台位于`/h2-console`。您可以使用 `spring.h2.console.path` 属性自定义控制台的路径。

30.6 使用 jOOQ

Java 面向对象查询（jOOQ）是 Data Geekery 的一个流行产品， 它从您的数据库生成 Java 代码，并允许您通过其流畅的 API 构建类型安全的 SQL 查询。商业版和开源版都可以与 Spring Boot 一起使用。

30.6.1 代码生成

要使用 jOOQ 类型安全查询，您需要从数据库模式生成 Java 类。您可以按照 jOOQ 用户手册中的说明进行 操作。如果您使用 `jooq-codegen-maven` 插件并且还使用 `spring-boot-starter-parent`“父 POM”，则可以安全地省略插件的`<version>`标记。您还可以使用 Spring 引导定义的版本变量（例如 `h2.version`）来声明插件的数据库依赖性。以下清单显示了一个示例：

```
<plugin>
	<groupId>org.jooq</groupId>
	<artifactId>jooq-codegen-maven</artifactId>
	<executions>
		...
	</executions>
	<dependencies>
		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<version>${h2.version}</version>
		</dependency>
	</dependencies>
	<configuration>
		<jdbc>
			<driver>org.h2.Driver</driver>
			<url>jdbc:h2:~/yourdatabase</url>
		</jdbc>
		<generator>
			...
		</generator>
	</configuration>
</plugin>
```

30.6.2 使用 DSLContext

jOOQ 提供的流畅 API 通过 `org.jooq.DSLContext` 接口启动。Spring Boot 将 `DSLContext` 自动配置为 Spring Bean 并将其连接到您的应用 `DataSource`。要使用 `DSLContext`，您可以 `@Autowire`，如下例所示：

```
@Component
public class JooqExample implements CommandLineRunner {

        private final DSLContext create;

        @Autowired
        public JooqExample(DSLContext dslContext) {
                this.create = dslContext;
        }

}
```

小费

jOOQ 手册倾向于使用名为 `create` 的变量来保存 `DSLContext`。

然后，您可以使用 `DSLContext` 构建查询，如以下示例所示：

```
public List<GregorianCalendar> authorsBornAfter1980() {
        return this.create.selectFrom(AUTHOR)
                .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new GregorianCalendar(1980,
0, 1)))
                .fetch(AUTHOR.DATE_OF_BIRTH);
}
```

30.6.3 jOOQ SQL 方言

除非已配置 `spring.jooq.sql-dialect` 属性，否则 Spring Boot 将确定用于数据源的 SQL 方言。如果 Spring Boot 无法检测到方言，则使用 `DEFAULT`。

注意

Spring Boot 只能自动配置开源版本的 jOOQ 支持的方言。

30.6.4 自定义 jOOQ

通过定义自己的`@Bean` 定义可以实现更高级的自定义，这些定义在创建 jOOQ `Configuration` 时使用。您可以为以下 jOOQ 类型定义 beans：

- `ConnectionProvider`
- `ExecutorProvider`
- `TransactionProvider`
- `RecordMapperProvider`
- `RecordUnmapperProvider`
- `RecordListenerProvider`
- `ExecuteListenerProvider`
- `VisitListenerProvider`
- `TransactionListenerProvider`

如果您想完全控制 jOOQ 配置，也可以创建自己的 `org.jooq.Configuration` `@Bean`。

31.使用 NoSQL Technologies

Spring 数据提供了其他项目，可帮助您访问各种 NoSQL 技术，包括： MongoDB， Neo4J， Elasticsearch， Solr， Redis， Gemfire， Cassandra， Couchbase 和 LDAP。Spring Boot 为 Redis，MongoDB，Neo4j，Elasticsearch，Solr Cassandra，Couchbase 和 LDAP 提供自动配置。您可以使用其他项目，但必须自己配置它们。请参阅 projects.spring.io/spring-data 上的相应参考文档。

31.1 Redis

Redis 是一个缓存，消息代理和功能丰富的键值存储。Spring Boot 为 Lettuce 和 Jedis 客户端库提供了基本的自动配置， 并为 Spring 数据 Redis 提供了它们之外的抽象。

有一个 `spring-boot-starter-data-redis`“Starter”用于以方便的方式收集依赖项。默认情况下，它使用 Lettuce。该启动器处理传统和反应应用程序。

小费

我们还提供 `spring-boot-starter-data-redis-reactive`“Starter”以与其他具有反应支持的商店保持一致。

31.1.1 连接到 Redis

您可以像注射任何其他 Spring Bean 一样注入自动配置的 `RedisConnectionFactory`，`StringRedisTemplate` 或 vanilla `RedisTemplate` 实例。默认情况下，实例尝试在 `localhost:6379` 连接到 Redis 服务器。以下列表显示了此类 bean 的示例：

```
@Component
public class MyBean {

        private StringRedisTemplate template;

        @Autowired
        public MyBean(StringRedisTemplate template) {
                this.template = template;
        }

        // ...

}
```

小费

您还可以注册实现 `LettuceClientConfigurationBuilderCustomizer` 的任意数量的 beans 以进行更高级的自定义。如果您使用 Jedis，也可以使用 `JedisClientConfigurationBuilderCustomizer`。

如果您添加自己配置的任何类型的`@Bean`，它将替换默认值（`RedisTemplate` 除外，当排除基于 bean 名称时，`redisTemplate` ，而不是它的类型）。默认情况下，如果类路径上有 `commons-pool2`，则会出现池连接工厂。

31.2 MongoDB

MongoDB 是一个开源的 NoSQL 文档数据库，它使用类似 JSON 的模式而不是传统的基于表的关系数据。Spring Boot 提供了一些使用 MongoDB 的便利，包括 `spring-boot-starter-data-mongodb` 和 `spring-boot-starter-data-mongodb-reactive`“Starters”。

31.2.1 连接 MongoDB 数据库

要访问 Mongo 数据库，您可以注入自动配置的 `org.springframework.data.mongodb.MongoDbFactory`。默认情况下，实例尝试在 `mongodb://localhost/test` 连接到 MongoDB 服务器。以下示例显示如何连接到 MongoDB 数据库：

```
import org.springframework.data.mongodb.MongoDbFactory;
import com.mongodb.DB;

@Component
public class MyBean {

        private final MongoDbFactory mongo;

        @Autowired
        public MyBean(MongoDbFactory mongo) {
                this.mongo = mongo;
        }

        // ...

        public void example() {
                DB db = mongo.getDb();
                // ...
        }

}
```

您可以设置 `spring.data.mongodb.uri` 属性以更改 URL 并配置其他设置，例如副本集，如以下示例所示：

```
spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com:12345,mongo2.example
.com:23456/test
```

或者，只要您使用 Mongo 2.x，就可以指定 `host` / `port`。例如，您可以在 `application.properties` 中声明以下设置：

```
spring.data.mongodb.host=mongoserver
spring.data.mongodb.port=27017
```

如果您已经定义了自己的 `MongoClient`，它将用于自动配置合适的 `MongoDbFactory`。支持 `com.mongodb.MongoClient` 和 `com.mongodb.client.MongoClient`。

注意

如果您使用 Mongo 3.0 Java 驱动程序，则不支持 `spring.data.mongodb.host` 和 `spring.data.mongodb.port`。在这种情况下，`spring.data.mongodb.uri` 应该用于提供所有配置。

小费

如果未指定 `spring.data.mongodb.port`，则使用默认值 `27017`。您可以从前面显示的示例中删除此行。

小费

如果您不使用 Spring Data Mongo，则可以注入 `com.mongodb.MongoClient` beans 而不是 `MongoDbFactory`。如果您想完全控制建立 MongoDB 连接，您也可以声明自己的 `MongoDbFactory` 或 `MongoClient` bean。

注意

如果您使用的是反应式驱动程序，则 SSL 需要 Netty。如果 Netty 可用且自己的工厂尚未自定义，则自动配置会自动配置此工厂。

31.2.2 MongoTemplate

Spring 数据 MongoDB 提供了一个 `MongoTemplate` 与 Spring `JdbcTemplate` 设计非常相似的类。与 `JdbcTemplate` 一样，Spring Boot 为您自动配置 bean 以注入模板，如下所示：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

        private final MongoTemplate mongoTemplate;

        @Autowired
        public MyBean(MongoTemplate mongoTemplate) {
                this.mongoTemplate = mongoTemplate;
        }

        // ...

}
```

有关完整的详细信息，请参阅 `MongoOperations` Javadoc。

31.2.3 Spring 数据 MongoDB 存储库

Spring 数据包括 MongoDB 的存储库支持。与前面讨论的 JPA 存储库一样，基本原则是基于方法名称自动构造查询。

实际上，Spring Data JPA 和 Spring Data MongoDB 共享相同的公共基础结构。您可以从前面获取 JPA 示例，假设 `City` 现在是 Mongo 数据类而不是 JPA `@Entity`，它的工作方式相同，如下例所示：

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

        Page<City> findAll(Pageable pageable);

        City findByNameAndStateAllIgnoringCase(String name, String state);
```

```
}
```

小费

您可以使用`@EntityScan`注释自定义文档扫描位置。

小费

有关 Spring Data MongoDB 的完整详细信息，包括其丰富的对象映射技术，请参阅其参考文档。

31.2.4 嵌入式 Mongo

Spring Boot 为 Embedded Mongo 提供自动配置 。要在 Spring Boot 应用程序中使用它，请在`de.flapdoodle.embed:de.flapdoodle.embed.mongo`上添加依赖项。

可以通过设置`spring.data.mongodb.port`属性来配置 Mongo 侦听的端口。要使用随机分配的空闲端口，请使用值 0. `MongoAutoConfiguration`创建的`MongoClient`将自动配置为使用随机分配的端口。

注意

如果未配置自定义端口，则嵌入式支持默认使用随机端口（而不是 27017）。

如果类路径上有 SLF4J，则 Mongo 生成的输出会自动路由到名为`org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo`的记录器。

您可以声明自己的`IMongodConfig`和`IRuntimeConfig` beans 来控制 Mongo 实例的配置和日志路由。

31.3 Neo4j

Neo4j 是一个开源的 NoSQL 图形数据库，它使用由一级关系连接的节点的丰富数据模型，与传统的 RDBMS 方法相比，它更适合于连接的大数据。Spring Boot 为使用 Neo4j 提供了一些便利，包括`spring-boot-starter-data-neo4j`“Starter”。

31.3.1 连接到 Neo4j 数据库

要访问 Neo4j 服务器，您可以注入自动配置的`org.neo4j.ogm.session.Session`。默认情况下，实例尝试使用 Bolt 协议连接到`localhost:7687`的 Neo4j 服务器。以下示例显示了如何注入 Neo4j `Session`：

```
@Component
public class MyBean {

        private final Session session;

        @Autowired
        public MyBean(Session session) {
                this.session = session;
        }
```

```
        // ...

}
```

您可以通过设置 `spring.data.neo4j.*`属性来配置要使用的 URI 和凭据，如以下示例所示：

```
spring.data.neo4j.uri=bolt://my-server:7687
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=secret
```

您可以通过添加 `org.neo4j.ogm.config.Configuration` `@Bean` 来完全控制会话创建。此外，添加`@Bean` 类型 `SessionFactory` 会禁用自动配置并为您提供完全控制权。

31.3.2 使用嵌入模式

如果将 `org.neo4j:neo4j-ogm-embedded-driver` 添加到应用程序的依赖项中，Spring Boot 会自动配置 Neo4j 的进程内嵌入式实例，该应用程序在应用程序关闭时不会保留任何数据。

注意

由于嵌入式 Neo4j OGM 驱动程序本身不提供 Neo4j 内核，因此您必须自己声明 `org.neo4j:neo4j` 为依赖项。有关兼容版本的列表，请参阅 Neo4j OGM 文档。

当类路径上有多个驱动程序时，嵌入式驱动程序优先于其他驱动程序。您可以通过设置 `spring.data.neo4j.embedded.enabled=false` 来明确禁用嵌入模式。

如果嵌入式驱动程序和 Neo4j 内核如上所述位于类路径上，则数据 Neo4j 测试会自动使用嵌入式 Neo4j 实例。

注意

您可以通过在配置中提供数据库文件的路径来为嵌入模式启用持久性，例如 `spring.data.neo4j.uri=file://var/tmp/graph.db`。

31.3.3 Neo4jSession

默认情况下，如果您正在运行 Web 应用程序，则会话将绑定到该线程以进行整个请求处理（即，它使用“在视图中打开会话”模式）。如果您不想要此行为，请将以下行添加到您的 `application.properties` 文件中：

```
spring.data.neo4j.open-in-view=false
```

31.3.4 Spring 数据 Neo4j 存储库

Spring 数据包括 Neo4j 的存储库支持。

Spring 数据 Neo4j 与 Spring Data JPA 共享公共基础架构，正如许多其他 Spring 数据模块那样。您可以从之前的 JPA 示例中将 `City` 定义为 Neo4j OGM `@NodeEntity` 而不是 JPA `@Entity`，并且存储库抽象以相同的方式工作，如以下示例所示：

```
package com.example.myapp.domain;

import java.util.Optional;

import org.springframework.data.neo4j.repository.*;
```

```
public interface CityRepository extends Neo4jRepository<City, Long> {

        Optional<City> findOneByNameAndState(String name, String state);

}
```

`spring-boot-starter-data-neo4j`“Starter”启用存储库支持以及事务管理。您可以在`@Configuration` - bean 上分别使用`@EnableNeo4jRepositories`和`@EntityScan`来自定义位置以查找存储库和实体。

小费

有关 Spring Data Neo4j 的完整详细信息，包括其对象映射技术，请参阅参考文档。

31.4 Gemfire

Spring 数据 Gemfire 为访问 Pivotal Gemfire 数据管理平台提供了方便的 Spring 友好工具 。有一个`spring-boot-starter-data-gemfire`“Starter”用于以方便的方式收集依赖项。目前没有 Gemfire 的自动配置支持，但您可以使用单个注释启用 Spring 数据存储库 ：`@EnableGemfireRepositories`。

31.5 Solr

Apache Solr 是一个搜索引擎。Spring Boot 为 Solr 5 客户端库提供了基本的自动配置，并在 Spring Data Solr 提供了它上面的抽象。有一个`spring-boot-starter-data-solr`“Starter”用于以方便的方式收集依赖项。

31.5.1 连接到 Solr

您可以像注射任何其他 Spring bean 一样注入自动配置的`SolrClient`实例。默认情况下，实例尝试连接到`localhost:8983/solr`的服务器。以下示例显示了如何注入 Solr bean：

```
@Component
public class MyBean {

        private SolrClient solr;

        @Autowired
        public MyBean(SolrClient solr) {
                this.solr = solr;
        }

        // ...

}
```

如果您添加`SolrClient`类型的`@Bean`，它将替换默认值。

31.5.2 Spring 数据 Solr 存储库

Spring 数据包括 Apache Solr 的存储库支持。与前面讨论的 JPA 存储库一样，基本原则是根据方法名称自动构建查询。

实际上，Spring Data JPA 和 Spring Data Solr 共享相同的公共基础结构。您可以从前面获取 JPA 示例，假设`City`现在是`@SolrDocument`类而不是 JPA `@Entity`，它的工作方式相同。

小费

有关 Spring Data Solr 的完整详细信息，请参阅 参考文档。

31.6 Elasticsearch

Elasticsearch 是一个开源，分布式，RESTful 搜索和分析引擎。Spring Boot 为 Elasticsearch 提供基本的自动配置。

Spring Boot 支持多个 HTTP 客户端：

- 官方 Java“低级”和“高级”REST 客户端
- 笑话

Spring Data Elasticsearch 仍在使用传输客户端 ，您可以使用 `spring-boot-starter-data-elasticsearch`“Starter”开始使用它。

31.6.1 REST 客户端连接到 Elasticsearch

Elasticsearch 提供了 两个 可用于查询集群的 REST 客户端：“低级”客户端和“高级”客户端。

如果您对类路径具有 `org.elasticsearch.client:elasticsearch-rest-client` 依赖关系，Spring Boot 将自动配置并注册默认目标为 `localhost:9200` 的 `RestClient` bean。您可以进一步调整 `RestClient` 的配置方式，如以下示例所示：

```
spring.elasticsearch.rest.uris=http://search.example.com:9200
spring.elasticsearch.rest.username=user
spring.elasticsearch.rest.password=secret
```

您还可以注册实现 `RestClientBuilderCustomizer` 的任意数量的 beans 以进行更高级的自定义。要完全控制注册，请定义 `RestClient` bean。

如果您对类路径具有 `org.elasticsearch.client:elasticsearch-rest-high-level-client` 依赖关系，Spring Boot 将自动配置 `RestHighLevelClient`，其包装任何现有的 `RestClient` bean，重用其 HTTP 配置。

31.6.2 使用 Jest 连接到 Elasticsearch

如果类路径上有 `Jest`，则可以注入一个自动配置的 `JestClient`，默认情况下为 `localhost:9200`。您可以进一步调整客户端的配置方式，如以下示例所示：

```
spring.elasticsearch.jest.uris=http://search.example.com:9200
spring.elasticsearch.jest.read-timeout=10000
spring.elasticsearch.jest.username=user
spring.elasticsearch.jest.password=secret
```

您还可以注册实现 `HttpClientConfigBuilderCustomizer` 的任意数量的 beans 以进行更高级的自定义。以下示例调整其他 HTTP 设置：

```
static class HttpSettingsCustomizer implements HttpClientConfigBuilderCustomizer {

        @Override
        public void customize(HttpClientConfig.Builder builder) {
                builder.maxTotalConnection(100).defaultMaxTotalConnectionPerRoute(5);
        }
```

```
}
```

要完全控制注册，请定义 `JestClient` bean。

31.6.3 使用 Spring 数据连接到 Elasticsearch

要连接到 Elasticsearch，您必须提供一个或多个群集节点的地址。可以通过将 `spring.data.elasticsearch.cluster-nodes` 属性设置为逗号分隔的 `host:port` 列表来指定地址。有了这种配置，`ElasticsearchTemplate` 或 `TransportClient` 可以像任何其他 Spring bean 一样注入，如下例所示：

```
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

```
@Component
public class MyBean {

        private final ElasticsearchTemplate template;

        public MyBean(ElasticsearchTemplate template) {
                this.template = template;
        }

        // ...

}
```

如果您添加自己的 `ElasticsearchTemplate` 或 `TransportClient` `@Bean`，则会替换默认值。

31.6.4 Spring 数据 Elasticsearch 存储库

Spring 数据包括 Elasticsearch 的存储库支持。与前面讨论的 JPA 存储库一样，基本原则是根据方法名称自动为您构建查询。

事实上，Spring Data JPA 和 Spring Data Elasticsearch 共享相同的通用基础架构。您可以从之前获取 JPA 示例，假设 `City` 现在是 Elasticsearch `@Document` 类而不是 JPA `@Entity`，它的工作方式相同。

小费

有关 Spring Data Elasticsearch 的完整详细信息，请参阅 参考文档。

31.7 Cassandra

Cassandra 是一个开源的分布式数据库管理系统，旨在处理许多商用服务器上的大量数据。Spring Boot 提供 Cassandra 的自动配置以及 Spring 数据 Cassandra 提供的摘要。有一个 `spring-boot-starter-data-cassandra`“Starter”用于以方便的方式收集依赖项。

31.7.1 连接到 Cassandra

您可以像对待任何其他 Spring Bean 一样注入自动配置的 `CassandraTemplate` 或 Cassandra `Session` 实例。`spring.data.cassandra.*`属性可用于自定义连接。通常，您提供 `keyspace-name` 和 `contact-points` 属性，如以下示例所示：

```
spring.data.cassandra.keyspace-name=mykeyspace
```

```
spring.data.cassandra.contact-points=cassandrahost1,cassandrahost2
```

您还可以注册实现 `ClusterBuilderCustomizer` 的任意数量的 beans 以进行更高级的自定义。

以下代码清单显示了如何注入 Cassandra bean：

```
@Component
public class MyBean {

        private CassandraTemplate template;

        @Autowired
        public MyBean(CassandraTemplate template) {
                this.template = template;
        }

        // ...

}
```

如果您添加 `CassandraTemplate` 类型的`@Bean`，它将替换默认值。

31.7.2 Spring 数据 Cassandra 存储库

Spring 数据包括 Cassandra 的基本存储库支持。目前，这比前面讨论的 JPA 存储库更有限，需要使用`@Query` 来注释 finder 方法。

小费

有关 Spring 数据 Cassandra 的完整详细信息，请参阅 参考文档。

31.8 Couchbase

Couchbase 是一个开源的，分布式的，多模型的 NoSQL 面向文档的数据库，针对交互式应用程序进行了优化。Spring Boot 提供了 Couchbase 的自动配置以及 Spring Data Couchbase 提供的抽象 。有 `spring-boot-starter-data-couchbase` 和 `spring-boot-starter-data-couchbase-reactive`“Starters”用于以方便的方式收集依赖项。

31.8.1 连接 Couchbase

您可以通过添加 Couchbase SDK 和一些配置来获得 `Bucket` 和 `Cluster`。`spring.couchbase.*`属性可用于自定义连接。通常，您提供引导主机，存储桶名称和密码，如以下示例所示：

```
spring.couchbase.bootstrap-hosts=my-host-1,192.168.1.123
spring.couchbase.bucket.name=my-bucket
spring.couchbase.bucket.password=secret
```

小费

您需要至少提供引导主机，在这种情况下，存储桶名称为 `default`，密码为空字符串
或者，您可以定义自己的 `org.springframework.data.couchbase.config.CouchbaseConfigurer` `@Bean` 来控制整个配置。

也可以自定义一些 `CouchbaseEnvironment` 设置。例如，以下配置更改用于打开新 `Bucket` 的超时并启用 SSL 支持：

```
spring.couchbase.env.timeouts.connect=3000
spring.couchbase.env.ssl.key-store=/location/of/keystore.jks
spring.couchbase.env.ssl.key-store-password=secret
```

查看 `spring.couchbase.env.*`属性以获取更多详细信息。

31.8.2 Spring 数据 Couchbase 存储库

Spring 数据包括 Couchbase 的存储库支持。有关 Spring Data Couchbase 的完整详细信息，请参阅参考文档。

您可以像使用任何其他 Spring Bean 一样注入自动配置的 `CouchbaseTemplate` 实例，前提是默认 `CouchbaseConfigurer` 可用（当您启用 Couchbase 支持时会发生这种情况，如前所述）。

以下示例显示了如何注入 Couchbase bean：

```
@Component
public class MyBean {

        private final CouchbaseTemplate template;

        @Autowired
        public MyBean(CouchbaseTemplate template) {
                this.template = template;
        }

        // ...

}
```

您可以在自己的配置中定义一些 beans 来覆盖自动配置提供的内容：

- `CouchbaseTemplate` `@Bean`，名称为 `couchbaseTemplate`。
- `IndexManager` `@Bean`，名称为 `couchbaseIndexManager`。
- `CustomConversions` `@Bean`，名称为 `couchbaseCustomConversions`。

为避免在您自己的配置中对这些名称进行硬编码，您可以重用 Spring Data Couchbase 提供的 `BeanNames`。例如，您可以自定义要使用的转换器，如下所示：

```
@Configuration
public class SomeConfiguration {

        @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
        public CustomConversions myCustomConversions() {
                return new CustomConversions(...);
        }

        // ...

}
```

小费

如果您想完全绕过 Spring Data Couchbase 的自动配置，请提供您自己的 `org.springframework.data.couchbase.config.AbstractCouchbaseDa` 现。

31.9 LDAP

LDAP（轻量级目录访问协议）是一种开放的，与供应商无关的行业标准应用程序协议，用于通过 IP 网络访问和维护分布式目录信息服务。Spring Boot 为任何兼容的 LDAP 服务器提供自动配置，并为 UnboundID 支持嵌入式内存中 LDAP 服务器 。

LDAP 抽象由 Spring 数据 LDAP 提供。有一个 `spring-boot-starter-data-ldap`“Starter”用于以方便的方式收集依赖项。

31.9.1 连接 LDAP 服务器

要连接到 LDAP 服务器，请确保声明对 `spring-boot-starter-data-ldap`“Starter”或 `spring-ldap-core` 的依赖关系，然后在 application.properties 中声明服务器的 URL，如以下示例所示：

```
spring.ldap.urls=ldap://myserver:1235
spring.ldap.username=admin
spring.ldap.password=secret
```

如果需要自定义连接设置，可以使用 `spring.ldap.base` 和 `spring.ldap.base-environment` 属性。

根据这些设置自动配置 `LdapContextSource`。如果您需要自定义它，例如使用 `PooledContextSource`，您仍然可以注入自动配置的 `LdapContextSource`。请务必将自定义的 `ContextSource` 标记为`@Primary`，以便自动配置的 `LdapTemplate` 使用它。

31.9.2 Spring 数据 LDAP 存储库

Spring 数据包括 LDAP 的存储库支持。有关 Spring 数据 LDAP 的完整详细信息，请参阅 参考文档。

您也可以像对待任何其他 Spring Bean 一样注入自动配置的 `LdapTemplate` 实例，如以下示例所示：

```
@Component
public class MyBean {

        private final LdapTemplate template;

        @Autowired
        public MyBean(LdapTemplate template) {
                this.template = template;
        }

        // ...

}
```

31.9.3 嵌入式内存 LDAP 服务器

出于测试目的，Spring Boot 支持从 UnboundID 自动配置内存中 LDAP 服务器。要配置服务器，请向 `com.unboundid:unboundid-ldapsdk` 添加依赖项并声明 `base-dn` 属性，如下所示：

```
spring.ldap.embedded.base-dn=dc=spring,dc=io
```

注意

可以定义多个 base-dn 值，但是，由于可分辨名称通常包含逗号，因此必须使用正确的符号来定义它们。

在 yaml 文件中，您可以使用 yaml 列表表示法：

```
spring.ldap.embedded.base-dn:
  - dc=spring,dc=io
  - dc=pivotal,dc=io
```

在属性文件中，必须包含索引作为属性名称的一部分：

```
spring.ldap.embedded.base-dn[0]=dc=spring,dc=io
spring.ldap.embedded.base-dn[1]=dc=pivotal,dc=io
```

默认情况下，服务器在随机端口上启动并触发常规 LDAP 支持。无需指定 `spring.ldap.urls` 属性。

如果类路径上有 `schema.ldif` 文件，则用于初始化服务器。如果要从其他资源加载初始化脚本，还可以使用 `spring.ldap.embedded.ldif` 属性。

默认情况下，标准模式用于验证 `LDIF` 文件。您可以通过设置 `spring.ldap.embedded.validation.enabled` 属性完全关闭验证。如果您有自定义属性，则可以使用 `spring.ldap.embedded.validation.schema` 来定义自定义属性类型或对象类。

31.10 InfluxDB

InfluxDB 是一个开源时间序列数据库，针对运营监控，应用程序指标，物联网传感器数据和实时分析等领域中的时间序列数据的快速，高可用性存储和检索进行了优化。

31.10.1 连接到 InfluxDB

Spring Boot 自动配置 `InfluxDB` 实例，前提是 `influxdb-java` 客户端在类路径上并且设置了数据库的 URL，如以下示例所示：

```
spring.influx.url=http://172.0.0.1:8086
```

如果与 InfluxDB 的连接需要用户和密码，则可以相应地设置 `spring.influx.user` 和 `spring.influx.password` 属性。

InfluxDB 依赖于 OkHttp。如果您需要在后台调整 http 客户端 `InfluxDB`，则可以注册 `InfluxDbOkHttpClientBuilderProvider` bean。

32.缓存

Spring 框架支持透明地向应用程序添加缓存。从本质上讲，抽象将缓存应用于方法，从而根据缓存中可用的信息减少执行次数。缓存逻辑应用透明，不会对调用者造成任何干扰。只要通过 `@EnableCaching` 注释启用了缓存支持，Spring Boot 就会自动配置缓存基础结构。

注意

有关详细信息，请查看 Spring 框架参考的相关部分。

简而言之，将缓存添加到服务操作就像在其方法中添加相关注释一样简单，如以下示例所示：

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class MathService {

        @Cacheable("piDecimals")
        public int computePiDecimal(int i) {
                // ...
        }

}
```

此示例演示了如何在可能代价高昂的操作上使用缓存。在调用 `computePiDecimal` 之前，抽象在 `piDecimals` 缓存中查找与 `i` 参数匹配的条目。如果找到条目，则缓存中的内容会立即返回给调用者，并且不会调用该方法。否则，将调用该方法，并在返回值之前更新缓存。

警告

您还可以透明地使用标准 JSR-107（JCache）注释（例如 `@CacheResult`）。但是，我们强烈建议您不要混用和匹配 Spring Cache 和 JCache 注释。

如果您不添加任何特定的缓存库，Spring Boot 会自动配置在内存中使用并发映射的 简单提供程序。当需要缓存时（例如前面示例中的 `piDecimals`），此提供程序会为您创建缓存。简单的提供程序并不是真正推荐用于生产用途，但它非常适合入门并确保您了解这些功能。当您决定使用缓存提供程序时，请务必阅读其文档以了解如何配置应用程序使用的缓存。几乎所有提供程序都要求您显式配置在应用程序中使用的每个缓存。有些提供了一种自定义 `spring.cache.cache-names` 属性定义的默认缓存的方法。

小费

还可以透明地 更新或 逐出缓存中的数据。

32.1 支持的缓存提供程序

缓存抽象不提供实际存储，并依赖于 `org.springframework.cache.Cache` 和 `org.springframework.cache.CacheManager` 接口实现的抽象。

如果您尚未定义类型为 `CacheManager` 的 bean 或名为 `cacheResolver` 的 `CacheResolver`（请参阅参考资料 `CachingConfigurer`），则 Spring Boot 会尝试检测以下提供程序（按指示顺序）：

1. 通用
2. JCache（JSR-107）（EhCache 3，Hazelcast，Infinispan 等）
3. EhCache 2.x
4. Hazelcast
5. Infinispan
6. Couchbase
7. Redis
8. Caffeine
9. 简单

小费

也可以通过设置 `spring.cache.type` 属性来强制特定的缓存提供程序。如果您需要在某些环境（例如测试）中完全禁用缓存，请使用此属性 。

小费

使用 `spring-boot-starter-cache`“Starter”快速添加基本缓存依赖项。首发进入 `spring-context-support`。如果手动添加依赖项，则必须包含 `spring-context-support` 才能使用 JCache，EhCache 2.x 或 Guava 支持。

如果 Wavefront 自动配置 `CacheManager`，则可以通过公开实现 `CacheManagerCustomizer` 接口的 bean 来完全初始化之前调整其配置。以下示例设置一个标志，表示应将 null 值传递给底层映射：

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
        return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {
                @Override
                public void customize(ConcurrentMapCacheManager cacheManager) {
                        cacheManager.setAllowNullValues(false);
                }
        };
}
```

注意

在前面的示例中，预计会自动配置 `ConcurrentMapCacheManager`。如果不是这种情况（您提供了自己的配置或自动配置了不同的缓存提供程序），则根本不会调用自定义程序。您可以拥有任意数量的自定义程序，也可以使用`@Order` 或 `Ordered` 订购它们。

32.1.1 通用

如果上下文定义至少一个 `org.springframework.cache.Cache` bean，则使用通用缓存。创建了包含该类型的所有 beans 的 `CacheManager`。

32.1.2 JCache（JSR-107）

JCache 通过类路径上的 `javax.cache.spi.CachingProvider` 进行自举（即类路径上存在符合 JSR-107 的缓存库），`spring-boot-starter-cache` 由 `spring-boot-starter-cache`“Starter”提供。可以使用各种兼容库，Spring Boot 为 Ehcache 3，Hazelcast 和 Infinispan 提供依赖关系管理。还可以添加任何其他兼容库。

可能会出现多个提供程序，在这种情况下必须明确指定提供程序。即使 JSR-107 标准没有强制使用标准化方法来定义配置文件的位置，Spring Boot 也会尽力适应使用实现细节设置缓存，如以下示例所示：

```
   # Only necessary if more than one provider is present
spring.cache.jcache.provider=com.acme.MyCachingProvider
spring.cache.jcache.config=classpath:acme.xml
```

注意

当缓存库同时提供本机实现和 JSR-107 支持时，Spring Boot 更喜欢 JSR-107 支持，因此如果切换到不同的 JSR-107 实现，则可以使用相同的功能。

小费

Spring Boot 普遍支持 Hazelcast。如果单个 `HazelcastInstance` 可用，它也会自动重用于 `CacheManager`，除非指定了 `spring.cache.jcache.config` 属性。

有两种方法可以自定义基础 `javax.cache.cacheManager`：

- 可以通过设置 `spring.cache.cache-names` 属性在启动时创建缓存。如果定义了自定义 `javax.cache.configuration.Configuration` bean，则会使用它来自定义它们。
- 使用 `CacheManager` 的引用调用 `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` beans 进行完全自定义。

小费

如果定义了标准 `javax.cache.CacheManager` bean，它将自动包装在抽象所需的 `org.springframework.cache.CacheManager` 实现中。没有进一步的自定义。

32.1.3 EhCache 2.x

如果可以在类路径的根目录中找到名为 `ehcache.xml` 的文件，则使用 EhCache 2.x. 如果找到 EhCache 2.x，则 `spring-boot-starter-cache`“Starter”提供的 `EhCacheCacheManager` 用于引导缓存管理器。还可以提供备用配置文件，如以下示例所示：

```
spring.cache.ehcache.config=classpath:config/another-config.xml
```

32.1.4 Hazelcast

Spring Boot 普遍支持 Hazelcast。如果已自动配置 `HazelcastInstance`，则会自动将其包装在 `CacheManager` 中。

32.1.5 Infinispan

Infinispan 没有默认配置文件位置，因此必须明确指定。否则，使用默认引导程序。

```
spring.cache.infinispan.config=infinispan.xml
```

可以通过设置 `spring.cache.cache-names` 属性在启动时创建缓存。如果定义了自定义 `ConfigurationBuilder` bean，则它用于自定义缓存。

注意

Spring Boot 中 Infinispan 的支持仅限于嵌入模式，并且非常基础。如果您想要更多选项，则应使用官方 Infinispan Spring Boot 启动器。有关更

多详细信息，请参阅 Infinispan 的文档。

32.1.6 Couchbase

如果 Couchbase Java 客户端和 couchbase-spring-cache 实现可用并且已配置 Couchbase，则会自动配置 CouchbaseCacheManager。通过设置 spring.cache.cache-names 属性，还可以在启动时创建其他缓存。这些缓存在自动配置的 Bucket 上运行。您可以还通过使用定制创建另一个 Bucket 额外的缓存。假设你需要两个缓存（cache1 和 cache2）在“主”Bucket 和一个（cache3）缓存上，自定义时间为 2 秒，“另一个”Bucket。您可以通过配置创建前两个缓存，如下所示：

```
spring.cache.cache-names=cache1,cache2
```

然后，您可以定义@Configuration 类来配置额外的 Bucket 和 cache3 缓存，如下所示：

```
@Configuration
public class CouchbaseCacheConfiguration {

        private final Cluster cluster;

        public CouchbaseCacheConfiguration(Cluster cluster) {
                this.cluster = cluster;
        }

        @Bean
        public Bucket anotherBucket() {
                return this.cluster.openBucket("another", "secret");
        }

        @Bean
        public CacheManagerCustomizer<CouchbaseCacheManager> cacheManagerCustomizer()
{
                return c -> {
                        c.prepareCache("cache3",
CacheBuilder.newInstance(anotherBucket())
                                        .withExpiration(2));
                };
        }

}
```

此示例配置重用通过自动配置创建的 Cluster。

32.1.7 Redis

如果 Redis 可用且已配置，则会自动配置 RedisCacheManager。通过设置 spring.cache.cache-names 属性可以在启动时创建其他缓存，并且可以使用 spring.cache.redis.*属性配置缓存默认值。例如，以下配置创建 cache1 和 cache2 缓存，生存时间为 10 分钟：

```
spring.cache.cache-names=cache1,cache2
spring.cache.redis.time-to-live=600000
```

注意

默认情况下，会添加一个键前缀，以便在两个单独的缓存使用相同的键时，Redis 没有重叠的键，并且不能返回无效值。如果您创建自己的 RedisCacheManager，我们强烈建议您启用此设置。

小费

您可以通过添加自己的 `RedisCacheConfiguration` `@Bean` 来完全控制配置。如果您正在寻找自定义序列化策略，这可能很有用。

32.1.8 Caffeine

Caffeine 是 Java 8 重写的 Guava 缓存，取代了对 Guava 的支持。如果存在 Caffeine，则自动配置 `CaffeineCacheManager`（由 `spring-boot-starter-cache`“Starter”提供）。可以通过设置 `spring.cache.cache-names` 属性在启动时创建缓存，并且可以通过以下之一（按指示的顺序）自定义缓存：

1. 由 `spring.cache.caffeine.spec` 定义的缓存规范
2. 定义了 `com.github.benmanes.caffeine.cache.CaffeineSpec` bean
3. 定义了 `com.github.benmanes.caffeine.cache.Caffeine` bean

例如，以下配置创建 `cache1` 和 `cache2` 缓存，最大大小为 500，生存时间为 10 分钟

```
spring.cache.cache-names=cache1,cache2
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

如果定义了 `com.github.benmanes.caffeine.cache.CacheLoader` bean，它将自动与 `CaffeineCacheManager` 相关联。由于 `CacheLoader` 将与缓存管理器管理的所有缓存关联，因此必须将其定义为 `CacheLoader<Object, Object>`。自动配置忽略任何其他泛型类型。

32.1.9 简单

如果找不到其他提供程序，则配置使用 `ConcurrentHashMap` 作为缓存存储的简单实现。如果您的应用程序中没有缓存库，则这是默认值。默认情况下，会根据需要创建缓存，但您可以通过设置 `cache-names` 属性来限制可用缓存列表。例如，如果您只想要 `cache1` 和 `cache2` 缓存，请按如下所示设置 `cache-names` 属性：

```
spring.cache.cache-names=cache1,cache2
```

如果这样做并且您的应用程序使用未列出的缓存，则在需要缓存时它会在运行时失败，但在启动时则不会。这类似于“真实”缓存提供程序在使用未声明的缓存时的行为方式。

32.1.10 None

当配置中存在`@EnableCaching` 时，也需要合适的缓存配置。如果需要在某些环境中完全禁用缓存，请将缓存类型强制为 `none` 以使用 no-op 实现，如以下示例所示：

```
spring.cache.type=none
```

33.消息传递

Spring 框架为与消息传递系统的集成提供了广泛的支持，从使用 `JmsTemplate` 的 JMS API 的简化使用到异步接收消息的完整基础结构。Spring AMQP 为高级消息队列协议提供了类似的功能集。Spring Boot 还为 `RabbitTemplate` 和 RabbitMQ 提供了自动配置选项。Spring WebSocket 本身包含对 STOMP 消息传递的支持，Spring Boot 通过启动器和少量自动配置支持。Spring Boot 也支持 Apache Kafka。

33.1 JMS

`javax.jms.ConnectionFactory` 接口提供了一种创建 `javax.jms.Connection` 的标准方法，用于与 JMS 代理进行交互。虽然 Spring 需要 `ConnectionFactory` 来使用 JMS，但您通常不需要自己直接使用它，而是可以依赖更高级别的消息传递抽象。（有关详细信息，请参阅 Spring 框架参考文档的 相关部分。）Spring Boot 还自动配置发送和接收消息所需的基础结构。

33.1.1 ActiveMQ 支持

当 ActiveMQ 在类路径上可用时，Spring Boot 也可以配置 `ConnectionFactory`。如果代理存在，则会自动启动并配置嵌入式代理（前提是未通过配置指定代理 URL）。

注意

如果使用 `spring-boot-starter-activemq`，则提供连接或嵌入 ActiveMQ 实例的必要依赖项，以及与 JMS 集成的 Spring 基础结构。

ActiveMQ 配置由 `spring.activemq.*`中的外部配置属性控制。例如，您可以在 `application.properties` 中声明以下部分：

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

默认情况下，`CachingConnectionFactory` 使用 `spring.jms.*`中的外部配置属性可以控制的合理设置包装本机 `ConnectionFactory`：

```
spring.jms.cache.session-cache-size=5
```

如果您更愿意使用本机池，则可以通过向 `org.messaginghub:pooled-jms` 添加依赖项并相应地配置 `JmsPoolConnectionFactory` 来实现，如以下示例所示：

```
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

小费

有关 `ActiveMQProperties` 更多支持的选项，请参阅 。您还可以注册实现 `ActiveMQConnectionFactoryCustomizer` 的任意数量的 beans 以进行更高级的自定义。

默认情况下，ActiveMQ 会创建一个目标（如果它尚不存在），以便根据提供的名称解析目标。

33.1.2 阿耳忒弥斯支持

Spring Boot 可以在检测到类路径上的 Artemis 可用时自动配置 `ConnectionFactory`。如果存在代理，则会自动启动并配置嵌入式代理（除非已明确设置 mode 属性）。支持的模式是 `embedded`（以明确表示需要嵌入式代理，如果代理路径在类路径上不可用则发生错误）和 `native`（使用{11 /连接到代理）传输协议）。配置后者时，Spring Boot 使用默认设置配置连接到本地计算机上运行的代理的 `ConnectionFactory`。

注意

如果使用 `spring-boot-starter-artemis`，则会提供连接到现有

Artemis 实例的必要依赖项，以及与 JMS 集成的 Spring 基础结构。将 `org.apache.activemq:artemis-jms-server` 添加到您的应用程序可让您使用嵌入模式。

Artemis 配置由 `spring.artemis.*`中的外部配置属性控制。例如，您可以在 `application.properties` 中声明以下部分：

```
spring.artemis.mode=native
spring.artemis.host=192.168.1.210
spring.artemis.port=9876
spring.artemis.user=admin
spring.artemis.password=secret
```

嵌入代理时，您可以选择是否要启用持久性并列出应该可用的目标。这些可以指定为逗号分隔列表以使用默认选项创建它们，或者您可以分别为高级队列和主题配置定义 `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` 或 `org.apache.activemq.artemis.jms.server.config.TopicConfiguration` 类型的 bean。

默认情况下，`CachingConnectionFactory` 使用 `spring.jms.*`中的外部配置属性可以控制的合理设置包装本机 `ConnectionFactory`：

```
spring.jms.cache.session-cache-size=5
```

如果您更愿意使用本机池，则可以通过向 `org.messaginghub:pooled-jms` 添加依赖关系并相应地配置 `JmsPoolConnectionFactory` 来实现，如以下示例所示：

```
spring.artemis.pool.enabled=true
spring.artemis.pool.max-connections=50
```

有关 `ArtemisProperties` 更多支持选项，请参阅

不使用 JNDI 查找，并使用 Artemis 配置中的 `name` 属性或通过配置提供的名称来解析目标名称。

33.1.3 使用 JNDI ConnectionFactory

如果您在应用程序服务器中运行应用程序，Spring Boot 会尝试使用 JNDI 找到 JMS `ConnectionFactory`。默认情况下，会检查 `java:/JmsXA` 和 `java:/XAConnectionFactory` 位置。如果需要指定备用位置，可以使用 `spring.jms.jndi-name` 属性，如以下示例所示：

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

33.1.4 发送消息

Spring 的 `JmsTemplate` 是自动配置的，您可以将其直接自动装入您自己的 beans，如以下示例所示：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

        private final JmsTemplate jmsTemplate;

        @Autowired
```

```
        public MyBean(JmsTemplate jmsTemplate) {
                this.jmsTemplate = jmsTemplate;
        }

        // ...

}
```

注意

JmsMessagingTemplate 可以以类似的方式注射。如果定义了 DestinationResolver 或 MessageConverter bean，则会自动将其与自动配置的 JmsTemplate 相关联。

33.1.5 接收消息

当存在 JMS 基础结构时，可以使用@JmsListener 注释任何 bean 以创建侦听器端点。如果未定义 JmsListenerContainerFactory，则会自动配置默认值。如果定义了 DestinationResolver 或 MessageConverter beans，它将自动关联到默认工厂。

默认情况下，默认工厂是事务性的。如果您在存在 JtaTransactionManager 的基础结构中运行，则默认情况下它与侦听器容器关联。如果不是，则启用 sessionTransacted 标志。在后一种情况下，您可以通过在侦听器方法（或其委托）上添加@Transactional，将本地数据存储事务与传入消息的处理相关联。这确保了在本地事务完成后确认传入消息。这还包括发送已在同一 JMS 会话上执行的响应消息。

以下组件在 someQueue 目标上创建一个侦听器端点：

```
@Component
public class MyBean {

        @JmsListener(destination = "someQueue")
        public void processMessage(String content) {
                // ...
        }

}
```

小费

有关 更多详细信息，请参阅@EnableJms 的 Javadoc。

如果您需要创建更多 JmsListenerContainerFactory 实例，或者如果要覆盖默认值，Spring Boot 提供 DefaultJmsListenerContainerFactoryConfigurer，您可以使用 DefaultJmsListenerContainerFactoryConfigurer 来初始化 DefaultJmsListenerContainerFactory，其设置与自动配置。

例如，以下示例公开了另一个使用特定 MessageConverter 的工厂：

```
@Configuration
static class JmsConfiguration {

        @Bean
        public DefaultJmsListenerContainerFactory myFactory(
                        DefaultJmsListenerContainerFactoryConfigurer configurer) {
                DefaultJmsListenerContainerFactory factory =
                                new DefaultJmsListenerContainerFactory();
                configurer.configure(factory, connectionFactory());
                factory.setMessageConverter(myMessageConverter());
```

```
                return factory;
        }

}
```

然后您可以在任何`@JmsListener`-注释方法中使用工厂，如下所示：

```
@Component
public class MyBean {

        @JmsListener(destination = "someQueue", containerFactory="myFactory")
        public void processMessage(String content) {
                // ...
        }

}
```

33.2 AMQP

高级消息队列协议（AMQP）是面向消息的中间件的平台中立的线级协议。Spring AMQP 项目将核心 Spring 概念应用于基于 AMQP 的消息传递解决方案的开发。Spring Boot 为通过 RabbitMQ 使用 AMQP 提供了一些便利，包括`spring-boot-starter-amqp`“Starter”。

33.2.1 RabbitMQ 支持

RabbitMQ 是一个基于 AMQP 协议的轻量级，可靠，可扩展且可移植的消息代理。Spring 使用`RabbitMQ`通过 AMQP 协议进行通信。

RabbitMQ 配置由`spring.rabbitmq.*`中的外部配置属性控制。例如，您可以在`application.properties`中声明以下部分：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

如果上下文中存在`ConnectionNameStrategy` bean，它将自动用于命名由自动配置的`ConnectionFactory`创建的连接。有关`RabbitProperties`更多支持的选项，请参阅 。

小费

有关详细信息，请参阅 了解 AMQP，RabbitMQ 使用的协议。

33.2.2 发送消息

Spring 的`AmqpTemplate`和`AmqpAdmin`是自动配置的，您可以将它们直接自动装入您自己的 beans，如以下示例所示：

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

        private final AmqpAdmin amqpAdmin;
        private final AmqpTemplate amqpTemplate;
```

```
        @Autowired
        public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
                this.amqpAdmin = amqpAdmin;
                this.amqpTemplate = amqpTemplate;
        }

        // ...

}
```

注意

RabbitMessagingTemplate 可以以类似的方式注射。如果定义了 MessageConverter bean，它将自动关联到自动配置的 AmqpTemplate。

如有必要，任何定义为 bean 的 org.springframework.amqp.core.Queue 都将自动用于在 RabbitMQ 实例上声明相应的队列。

要重试操作，可以在 AmqpTemplate 上启用重试（例如，在代理连接丢失的情况下）：

```
spring.rabbitmq.template.retry.enabled=true
spring.rabbitmq.template.retry.initial-interval=2s
```

默认情况下禁用重试。您还可以通过声明 RabbitRetryTemplateCustomizer bean 以编程方式自定义 RetryTemplate。

33.2.3 接收消息

当 Rabbit 基础结构存在时，任何 bean 都可以使用@RabbitListener 进行注释以创建侦听器端点。如果未定义 RabbitListenerContainerFactory，则会自动配置默认值 SimpleRabbitListenerContainerFactory，您可以使用 spring.rabbitmq.listener.type 属性切换到直接容器。如果定义了 MessageConverter 或 MessageRecoverer bean，它将自动与默认工厂关联。

以下示例组件在 someQueue 队列上创建一个侦听器端点：

```
@Component
public class MyBean {

        @RabbitListener(queues = "someQueue")
        public void processMessage(String content) {
                // ...
        }

}
```

小费

有关详细信息，请参阅@EnableRabbit 的 Javadoc。

如果您需要创建更多 RabbitListenerContainerFactory 个实例，或者如果要覆盖默认值，Spring Boot 会提供 SimpleRabbitListenerContainerFactoryConfigurer 和 DirectRabbitListenerContainerFactoryConfigurer，您可以使用它来初始化 SimpleRabbitListenerContainerFactory 和 DirectRabbitListenerContainerFactory 具有与自动配置使用的工厂相同的设置。

小费

您选择的容器类型无关紧要。这两个 beans 由自动配置公开。

例如，以下配置类公开了另一个使用特定 `MessageConverter` 的工厂：

```
@Configuration
static class RabbitConfiguration {

        @Bean
        public SimpleRabbitListenerContainerFactory myFactory(
                        SimpleRabbitListenerContainerFactoryConfigurer configurer) {
                SimpleRabbitListenerContainerFactory factory =
                                new SimpleRabbitListenerContainerFactory();
                configurer.configure(factory, connectionFactory);
                factory.setMessageConverter(myMessageConverter());
                return factory;
        }

}
```

然后你可以在任何`@RabbitListener` - 注释方法中使用工厂，如下所示：

```
@Component
public class MyBean {

        @RabbitListener(queues = "someQueue", containerFactory="myFactory")
        public void processMessage(String content) {
                // ...
        }

}
```

您可以启用重试来处理侦听器抛出异常的情况。默认情况下，使用 `RejectAndDontRequeueRecoverer`，但您可以定义自己的 `MessageRecoverer`。当重试耗尽时，如果代理配置了这样做，则拒绝该消息并将其丢弃或路由到死信交换。默认情况下，禁用重试。您还可以通过声明 `RabbitRetryTemplateCustomizer` bean 以编程方式自定义 `RetryTemplate`。

重要

默认情况下，如果禁用重试并且侦听器抛出异常，则会无限期地重试传递。您可以通过两种方式修改此行为：将 `defaultRequeueRejected` 属性设置为 `false`，以便尝试零重新传递或抛出 `AmqpRejectAndDontRequeueException` 以表示应拒绝该消息。后者是启用重试并且达到最大传递尝试次数时使用的机制。

33.3 Apache Kafka 支持

通过提供 `spring-kafka` 项目的自动配置来支持 Apache Kafka。

Kafka 配置由 `spring.kafka.*`中的外部配置属性控制。例如，您可以在 `application.properties` 中声明以下部分：

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

小费

要在启动时创建主题，请添加 bean 类型 `NewTopic`。如果主题已存在，则忽略 bean。

有关 `KafkaProperties` 更多支持选项，请参阅

33.3.1 发送消息

Spring 的 `KafkaTemplate` 是自动配置的，您可以直接在自己的 beans 中自动装配它，如下例所示：

```
@Component
public class MyBean {

        private final KafkaTemplate kafkaTemplate;

        @Autowired
        public MyBean(KafkaTemplate kafkaTemplate) {
                this.kafkaTemplate = kafkaTemplate;
        }

        // ...

}
```

注意

如果定义了属性 `spring.kafka.producer.transaction-id-prefix`，则会自动配置 `KafkaTransactionManager`。此外，如果定义了 `RecordMessageConverter` bean，它将自动与自动配置的 `KafkaTemplate` 相关联。

33.3.2 接收消息

当存在 Apache Kafka 基础结构时，可以使用`@KafkaListener` 注释任何 bean 以创建侦听器端点。如果未定义 `KafkaListenerContainerFactory`，则会使用 `spring.kafka.listener.*`中定义的键自动配置默认值。

以下组件在 `someTopic` 主题上创建一个侦听器端点：

```
@Component
public class MyBean {

        @KafkaListener(topics = "someTopic")
        public void processMessage(String content) {
                // ...
        }

}
```

如果定义了 `KafkaTransactionManager` bean，它将自动关联到容器工厂。同样，如果定义了 `RecordMessageConverter`，`ErrorHandler` 或 `AfterRollbackProcessor` bean，它将自动与默认工厂相关联。

小费

自定义 ChainedKafkaTransactionManager 必须标记为 @Primary，因为它通常会引用自动配置的 KafkaTransactionManager bean。

33.3.3 卡夫卡流

Apache Kafka 的 Spring 提供了一个工厂 bean 来创建一个 StreamsBuilder 对象并管理其流的生命周期。Spring Boot 只要 kafka-streams 在类路径上，并且通过@EnableKafkaStreams 注释启用 Kafka Streams，就会自动配置所需的 KafkaStreamsConfiguration bean。

启用 Kafka Streams 意味着必须设置应用程序 ID 和引导程序服务器。可以使用 spring.kafka.streams.application-id 配置前者，如果未设置，则默认为 spring.application.name。后者可以全局设置或专门为流而重写。

使用专用属性可以使用其他几个属性; 可以使用 spring.kafka.streams.properties 命名空间设置其他任意 Kafka 属性。有关更多信息，另请参见第 33.3.4 节“其他 Kafka 属性”。

要使用工厂 bean，只需将 StreamsBuilder 连接到@Bean，如下例所示：

```
@Configuration
@EnableKafkaStreams
static class KafkaStreamsExampleConfiguration {

        @Bean
        public KStream<Integer, String> kStream(StreamsBuilder streamsBuilder) {
                KStream<Integer, String> stream = streamsBuilder.stream("ks1In");
                stream.map((k, v) -> new KeyValue<>(k, v.toUpperCase())).to("ks1Out",
                                Produced.with(Serdes.Integer(), new JsonSerde<>()));
                return stream;
        }

}
```

默认情况下，由其创建的 StreamBuilder 对象管理的流将自动启动。您可以使用 spring.kafka.streams.auto-startup 属性自定义此行为。

33.3.4 附加 Kafka 属性

自动配置支持的属性显示在 附录 A，常见应用程序属性中。请注意，在大多数情况下，这些属性（连字符或 camelCase）直接映射到 Apache Kafka 点状属性。有关详细信息，请参阅 Apache Kafka 文档。

这些属性中的前几个适用于所有组件（生产者，使用者，管理员和流），但如果您希望使用不同的值，则可以在组件级别指定。Apache Kafka 指定重要性为 HIGH，MEDIUM 或 LOW 的属性。Spring Boot auto-configuration 支持所有 HIGH 重要性属性，一些选定的 MEDIUM 和 LOW 属性，以及任何没有默认值的属性。

只有 Kafka 支持的属性的一部分可以通过 KafkaProperties 类直接获得。如果您希望使用不直接支持的其他属性配置生产者或使用者，请使用以下属性：

```
spring.kafka.properties.prop.one=first
spring.kafka.admin.properties.prop.two=second
spring.kafka.consumer.properties.prop.three=third
spring.kafka.producer.properties.prop.four=fourth
spring.kafka.streams.properties.prop.five=fifth
```

这将常见的 prop.one Kafka 属性设置为 first（适用于生产者，消费者和管理员），将 prop.two admin 属性设置为 second，将 prop.three 使用者属性设置为 third，

`prop.four` 生产者属性为 `fourth`，`prop.five` 流属性为 `fifth`。

您还可以按如下方式配置 Spring Kafka `JsonDeserializer`：

```
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.value.default.type=com.example.Invoice
spring.kafka.consumer.properties.spring.json.trusted.packages=com.example,org.acme
```

同样，您可以禁用在标头中发送类型信息的 `JsonSerializer` 默认行为：

```
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.add.type.headers=false
```

重要

以这种方式设置的属性会覆盖 Spring Boot 明确支持的任何配置项。

34.使用 `RestTemplate` 调用 REST 服务

如果需要从应用程序调用远程 REST 服务，可以使用 Spring Framework 的 `RestTemplate` 类。由于 `RestTemplate` 实例在使用之前通常需要进行自定义，因此 Spring Boot 不提供任何单个自动配置 `RestTemplate` bean。但是，它会自动配置 `RestTemplateBuilder`，可在需要时用于创建 `RestTemplate` 实例。自动配置的 `RestTemplateBuilder` 确保将合理的 `HttpMessageConverters` 应用于 `RestTemplate` 实例。

以下代码显示了一个典型示例：

```
@Service
public class MyService {

        private final RestTemplate restTemplate;

        public MyService(RestTemplateBuilder restTemplateBuilder) {
                this.restTemplate = restTemplateBuilder.build();
        }

        public Details someRestCall(String name) {
                return this.restTemplate.getForObject("/{name}/details",
Details.class, name);
        }

}
```

小费

`RestTemplateBuilder` 包含许多可用于快速配置 `RestTemplate` 的有用方法。例如，要添加 BASIC auth 支持，可以使用 `builder.basicAuthentication("user", "password").build()`。

34.1 RestTemplate 自定义

`RestTemplate` 自定义有三种主要方法，具体取决于您希望自定义应用的广泛程度。

要使任何自定义的范围尽可能窄，请注入自动配置的 `RestTemplateBuilder`，然后根据需要调用其方法。每个方法调用都返回一个新的 `RestTemplateBuilder` 实例，因此自定义只会影响构建器的这种使用。

要进行应用程序范围的附加自定义，请使用 `RestTemplateCustomizer` bean。所有这些 beans 都会自动注册到自动配置的 `RestTemplateBuilder`，并应用于使用它构建的任何模板。

以下示例显示了一个自定义程序，它为除 `192.168.0.5` 之外的所有主机配置代理的使用：

```
static class ProxyCustomizer implements RestTemplateCustomizer {

        @Override
        public void customize(RestTemplate restTemplate) {
                HttpHost proxy = new HttpHost("proxy.example.com");
                HttpClient httpClient = HttpClientBuilder.create()
                                .setRoutePlanner(new DefaultProxyRoutePlanner(proxy)
{

                                        @Override
                                        public HttpHost determineProxy(HttpHost
target,
                                                        HttpRequest request,
HttpContext context)
                                                        throws HttpException {
                                                if
(target.getHostName().equals("192.168.0.5")) {
                                                        return null;
                                                }
                                                return super.determineProxy(target,
request, context);
                                        }

                                }).build();
                restTemplate.setRequestFactory(
                                new
HttpComponentsClientHttpRequestFactory(httpClient));
        }

}
```

最后，最极端（也很少使用）的选项是创建自己的 `RestTemplateBuilder` bean。这样做会关闭 `RestTemplateBuilder` 的自动配置，并阻止使用任何 `RestTemplateCustomizer` beans。

35.使用 `WebClient` 调用 REST 服务

如果您的类路径上有 Spring WebFlux，您还可以选择使用 `WebClient` 来调用远程 REST 服务。与 `RestTemplate` 相比，该客户端具有更多功能感并且完全被动。您可以在 Spring 框架文档的专用部分中了解有关 `WebClient` 的更多信息 。

Spring Boot 为您创建并预先配置 `WebClient.Builder`; 强烈建议将其注入您的组件并使用它来创建 `WebClient` 实例。Spring Boot 正在配置该构建器以共享 HTTP 资源，以与服务器相同的方式反映编解码器设置（请参阅 WebFlux HTTP 编解码器自动配置）等。

以下代码显示了一个典型示例：

```
@Service
public class MyService {

        private final WebClient webClient;
```

```
        public MyService(WebClient.Builder webClientBuilder) {
                this.webClient =
webClientBuilder.baseUrl("http://example.org").build();
        }

        public Mono<Details> someRestCall(String name) {
                return this.webClient.get().uri("/{name}/details", name)

.retrieve().bodyToMono(Details.class);
        }

}
```

35.1 WebClient 运行时

Spring Boot 将自动检测用于驱动 `WebClient` 的 `ClientHttpConnector`，具体取决于应用程序类路径上可用的库。目前，支持 Reactor Netty 和 Jetty RS 客户端。

`spring-boot-starter-webflux` 启动程序默认依赖于 `io.projectreactor.netty:reactor-netty`，它带来了服务器和客户端实现。如果您选择将 Jetty 用作反应式服务器，则应该在 Jetty Reactive HTTP 客户端库 `org.eclipse.jetty:jetty-reactive-httpclient` 上添加依赖项。对服务器和客户端使用相同的技术具有优势，因为它将自动在客户端和服务器之间共享 HTTP 资源。

开发人员可以通过提供自定义 `ReactorResourceFactory` 或 `JettyResourceFactory` bean 覆盖 Jetty 和 Reactor Netty 的资源配置 - 这将应用于客户端和服务器。

如果您希望覆盖客户端的该选项，您可以定义自己的 `ClientHttpConnector` bean 并完全控制客户端配置。

您可以在 Spring 框架参考文档中了解有关 `WebClient` 配置选项的更多信息 。

35.2 WebClient 自定义

`WebClient` 自定义有三种主要方法，具体取决于您希望自定义应用的广泛程度。

要使任何自定义的范围尽可能窄，请注入自动配置的 `WebClient.Builder`，然后根据需要调用其方法。`WebClient.Builder` 实例是有状态的：构建器上的任何更改都会反映在随后使用它创建的所有客户端中。如果要使用相同的构建器创建多个客户端，还可以考虑使用 `WebClient.Builder other = builder.clone();`克隆构建器。

要对所有 `WebClient.Builder` 实例进行应用程序范围的附加自定义，您可以声明 `WebClientCustomizer` beans 并在注入点本地更改 `WebClient.Builder`。

最后，您可以回退到原始 API 并使用 `WebClient.create()`。在这种情况下，不应用自动配置或 `WebClientCustomizer`。

36.验证

只要 JSR-303 实现（例如 Hibernate 验证器）在类路径上，Bean 验证 1.1 支持的方法验证功能就会自动启用。这使得 bean 方法可以使用 `javax.validation` 对其参数和/或返回值的约束进行注释。具有此类带注释方法的目标类需要在类型级别使用`@Validated` 注释进行注释，以便搜索其内联约束注释的方法。

例如，以下服务触发第一个参数的验证，确保其大小在 8 到 10 之间：

```
@Service
@Validated
public class MyBean {

        public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code,
                        Author author) {
                ...
        }

}
```

37.发送电子邮件

Spring 框架提供了使用 `JavaMailSender` 界面发送电子邮件的简单抽象，Spring Boot 为其提供了自动配置以及启动器模块。

小费

有关如何使用 `JavaMailSender` 的详细说明，请参阅参考文档。

如果 `spring.mail.host` 和相关库（由 `spring-boot-starter-mail` 定义）可用，则创建默认值 `JavaMailSender`（如果不存在）。可以通过 `spring.mail` 命名空间中的配置项进一步自定义发件人。有关 `MailProperties` 详细信息，请参阅

特别是，某些默认超时值是无限的，您可能希望更改它以避免线程被无响应的邮件服务器阻塞，如以下示例所示：

```
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=3000
spring.mail.properties.mail.smtp.writetimeout=5000
```

也可以使用 JNDI 中的现有 `Session` 配置 `JavaMailSender`：

```
spring.mail.jndi-name=mail/Session
```

设置 `jndi-name` 时，它优先于所有其他与会话相关的设置。

38.使用 JTA 的分布式事务

Spring Boot 通过使用 Atomikos 或 Bitronix 嵌入式事务管理器支持跨多个 XA 资源的分布式 JTA 事务。部署到合适的 Java EE Application Server 时，也支持 JTA 事务。

检测到 JTA 环境时，Spring 的 `JtaTransactionManager` 用于管理事务。自动配置的 JMS，DataSource 和 JPA beans 已升级为支持 XA 事务。您可以使用标准 Spring 惯用语（例如 `@Transactional`）来参与分布式事务。如果您在 JTA 环境中并且仍想使用本地事务，则可以将 `spring.jta.enabled` 属性设置为 `false` 以禁用 JTA 自动配置。

38.1 使用 Atomikos 事务管理器

Atomikos 是一个流行的开源事务管理器，可以嵌入到您的 Spring Boot 应用程序中。您可以使用 `spring-boot-starter-jta-atomikos` Starter 引入相应的 Atomikos 库。Spring Boot 自动配置 Atomikos 并确保将适当的 `depends-on` 设置应用于 Spring beans 以正确启动和关闭顺序。

默认情况下，Atomikos 事务日志将写入应用程序主目录（应用程序 jar 文件所在的目录）中的 `transaction-logs` 目录。您可以通过在 `application.properties` 文件中设置

`spring.jta.log-dir` 属性来自定义此目录的位置。以 `spring.jta.atomikos.properties` 开头的属性也可用于自定义 Atomikos `UserTransactionServiceImp`。有关 完整的详细信息，请参阅 [`AtomikosProperties` Javadoc]。

注意

为确保多个事务管理器可以安全地协调相同的资源管理器，必须为每个 Atomikos 实例配置唯一 ID。默认情况下，此 ID 是运行 Atomikos 的计算机的 IP 地址。要确保生产中的唯一性，应为应用程序的每个实例配置具有不同值的 `spring.jta.transaction-manager-id` 属性。

38.2 使用 Bitronix 事务管理器

Bitronix 是一种流行的开源 JTA 事务管理器实现。您可以使用 `spring-boot-starter-jta-bitronix` 启动程序将适当的 Bitronix 依赖项添加到项目中。与 Atomikos 一样，Spring Boot 自动配置 Bitronix 并对 beans 进行后处理，以确保启动和关闭顺序正确。

默认情况下，Bitronix 事务日志文件（`part1.btm` 和 `part2.btm`）将写入应用程序主目录中的 `transaction-logs` 目录。您可以通过设置 `spring.jta.log-dir` 属性来自定义此目录的位置。以 `spring.jta.bitronix.properties` 开头的属性也绑定到 `bitronix.tm.Configuration` bean，允许完全自定义。有关详细信息，请参阅 Bitronix 文档。

注意

为确保多个事务管理器可以安全地协调相同的资源管理器，必须为每个 Bitronix 实例配置唯一的 ID。默认情况下，此 ID 是运行 Bitronix 的计算机的 IP 地址。为了确保生产的唯一性，您应该为应用程序的每个实例配置不同值的 `spring.jta.transaction-manager-id` 属性。

38.3 使用 Java EE 托管事务管理器

如果将 Spring Boot 应用程序打包为 `war` 或 `ear` 文件并将其部署到 Java EE 应用程序服务器，则可以使用应用程序服务器的内置事务管理器。Spring Boot 尝试通过查看常见的 JNDI 位置（`java:comp/UserTransaction`，`java:comp/TransactionManager` 等）来自动配置事务管理器。如果使用应用程序服务器提供的事务服务，通常还需要确保所有资源都由服务器管理并通过 JNDI 公开。Spring Boot 尝试通过在 JNDI 路径（`java:/JmsXA` 或 `java:/XAConnectionFactory`）查找 `ConnectionFactory` 来自动配置 JMS，并且可以使用 `spring.datasource.jndi-name` 属性来配置 `DataSource`。`spring.datasource.jndi-name` 属性 配置{1801 } /}。

38.4 混合 XA 和非 XA JMS 连接

使用 JTA 时，主 JMS `ConnectionFactory` bean 可识别 XA 并参与分布式事务。在某些情况下，您可能希望使用非 XA `ConnectionFactory` 处理某些 JMS 消息。例如，您的 JMS 处理逻辑可能需要比 XA 超时更长的时间。

如果你想使用非 XA `ConnectionFactory`，你可以注入 `nonXaJmsConnectionFactory` bean 而不是`@Primary` `jmsConnectionFactory` bean。为了保持一致性，使用 bean 别名 `xaJmsConnectionFactory` 也提供了 `jmsConnectionFactory` bean。

以下示例显示如何注入 ConnectionFactory 实例：

```
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;

// Inject the XA aware ConnectionFactory (uses the alias and injects the same as
above)
@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;

// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

38.5 支持替代嵌入式事务管理器

该 XAConnectionFactoryWrapper 和 XADataSourceWrapper 接口可用于支持替代嵌入式事务经理。接口负责包装 XAConnectionFactory 和 XADataSource beans 并将它们公开为常规 ConnectionFactory 和 DataSource beans，它们透明地注册分布式事务。如果您在 ApplicationContext 中注册了 JtaTransactionManager bean 和相应的 XA 包装 beans，则 DataSource 和 JMS 自动配置将使用 JTA 变体。

该 BitronixXAConnectionFactoryWrapper 和 BitronixXADataSourceWrapper 提供了如何编写 XA 包装很好的例子。

39. Hazelcast

如果 Hazelcast 在类路径上并找到合适的配置，Spring Boot 会自动配置您可以在应用程序中注入的 HazelcastInstance。

如果你定义 com.hazelcast.config.Config bean，Spring Boot 使用它。如果您的配置定义了实例名称，Spring Boot 会尝试查找现有实例而不是创建新实例。

您还可以指定要通过配置使用的 hazelcast.xml 配置文件，如以下示例所示：

```
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

否则，Spring Boot 会尝试从默认位置找到 Hazelcast 配置：工作目录中的 hazelcast.xml 或类路径的根目录。我们还检查是否设置了 hazelcast.config 系统属性。有关更多详细信息，请参阅 Hazelcast 文档。

如果类路径中存在 hazelcast-client，则 Spring Boot 首先尝试通过检查以下配置选项来创建客户端：

- 存在 com.hazelcast.client.config.ClientConfig bean。
- 由 spring.hazelcast.config 属性定义的配置文件。
- hazelcast.client.config 系统属性的存在。
- 工作目录中的 hazelcast-client.xml 或类路径的根目录。

注意

Spring Boot 还为 Hazelcast 提供了 明确的缓存支持。如果启用了缓

存，HazelcastInstance 将自动包含在 CacheManager 实现中。

40. Quartz Scheduler

Spring Boot 为使用 Quartz 调度程序提供了一些便利 ，包括 spring-boot-starter-quartz“Starter”。如果 Quartz 可用，则自动配置 Scheduler（通过 SchedulerFactoryBean 抽象）。

自动选取以下类型的 Beans 并与 Scheduler 相关联：

- JobDetail：定义一个特定的工作。可以使用 JobBuilder API 构建 JobDetail 个实例。
- Calendar.
- Trigger：定义何时触发特定作业。

默认情况下，使用内存中的 JobStore。但是，如果应用程序中有 DataSource bean 并且相应地配置了 spring.quartz.job-store-type 属性，则可以配置基于 JDBC 的存储，如以下示例所示：

```
spring.quartz.job-store-type=jdbc
```

使用 JDBC 存储时，可以在启动时初始化架构，如以下示例所示：

```
spring.quartz.jdbc.initialize-schema=always
```

警告

默认情况下，使用 Quartz 库提供的标准脚本检测并初始化数据库。这些脚本删除现有表，在每次重启时删除所有触发器。也可以通过设置 spring.quartz.jdbc.schema 属性来提供自定义脚本。

要让 Quartz 使用 DataSource 而不是应用程序的主 DataSource，请声明 DataSource bean，用@QuartzDataSource 注释其@Bean 方法。这样做可确保 SchedulerFactoryBean 和架构初始化都使用特定于 Quartz 的 DataSource。

默认情况下，配置创建的作业不会覆盖已从永久性作业存储区读取的已注册作业。要启用覆盖现有作业定义，请设置 spring.quartz.overwrite-existing-jobs 属性。

可以使用 spring.quartz 属性和 SchedulerFactoryBeanCustomizer beans 自定义 Quartz Scheduler 配置，这允许程序化 SchedulerFactoryBean 自定义。可以使用 spring.quartz.properties.*自定义高级 Quartz 配置属性。

注意

特别是，Executor bean 与调度程序无关，因为 Quartz 提供了一种通过 spring.quartz.properties 配置调度程序的方法。如果需要自定义任务执行程序，请考虑实现 SchedulerFactoryBeanCustomizer。

作业可以定义 setter 以注入数据映射属性。常规 beans 也可以以类似的方式注入，如以下示例所示：

```
public class SampleJob extends QuartzJobBean {
```

```
        private MyService myService;

        private String name;

        // Inject "MyService" bean
        public void setMyService(MyService myService) { ... }

        // Inject the "name" job data property
        public void setName(String name) { ... }

        @Override
        protected void executeInternal(JobExecutionContext context)
                throws JobExecutionException {
            ...
        }

}
```

41.任务执行和调度

在上下文中没有 `TaskExecutor` bean 的情况下，Spring Boot 使用合理的默认值自动配置 `ThreadPoolTaskExecutor`，这些默认值可以自动与异步任务执行相关联（`@EnableAsync`）和 Spring MVC 异步请求处理。

线程池使用 8 个核心线程，可根据负载增长和缩小。可以使用 `spring.task.execution` 命名空间对这些默认设置进行微调，如以下示例所示：

```
spring.task.execution.pool.max-threads=16
spring.task.execution.pool.queue-capacity=100
spring.task.execution.pool.keep-alive=10s
```

这会将线程池更改为使用有界队列，以便在队列满（100 个任务）时，线程池增加到最多 16 个线程。当线程在闲置 10 秒（而不是默认为 60 秒）时回收线程时，池的收缩会更加激进。

如果需要与计划任务执行（`@EnableScheduling`）相关联，也可以自动配置 `ThreadPoolTaskScheduler`。默认情况下，线程池使用一个线程，并且可以使用 `spring.task.scheduling` 命名空间对这些设置进行微调。

如果需要创建自定义执行程序或调度程序，则在上下文中可以使用 `TaskExecutorBuilder` bean 和 `TaskSchedulerBuilder` bean。

42. Spring Integration

Spring Boot 为使用 Spring 集成提供了一些便利，包括 `spring-boot-starter-integration`“Starter”。Spring 集成提供了有关消息传递以及其他传输（如 HTTP，TCP 等）的抽象。如果类路径上有 Spring Integration，则通过`@EnableIntegration` 注释初始化它。

Spring Boot 还配置由附加 Spring Integration 模块的存在触发的一些功能。如果 `spring-integration-jmx` 也在类路径上，则通过 JMX 发布消息处理统计信息。如果 `spring-integration-jdbc` 可用，则可以在启动时创建默认数据库模式，如以下行所示：

```
spring.integration.jdbc.initialize-schema=always
```

有关 详细信息，请参阅 `IntegrationAutoConfiguration` 和 `IntegrationProperties` 类。

默认情况下，如果存在千分尺 `meterRegistry` bean，则千分尺将管理 Spring Integration 指标。如果您希望使用旧版 Spring Integration 指标，请在应用程序上下文中添加 `DefaultMetricsFactory` bean。

43. Spring Session

Spring Boot 为各种数据存储提供 Spring Session 自动配置。构建 Servlet Web 应用程序时，可以自动配置以下存储：

- JDBC
- Redis
- Hazelcast
- MongoDB 的

构建响应式 Web 应用程序时，可以自动配置以下存储：

- Redis
- MongoDB 的

如果类路径中存在单个 Spring Session 模块，则 Spring Boot 会自动使用该存储实现。如果您有多个实现，则必须选择 `StoreType` 要用于存储会话的实现。例如，要使用 JDBC 作为后端存储，您可以按如下方式配置应用程序：

```
spring.session.store-type=jdbc
```

小费

您可以通过将 `store-type` 设置为 `none` 来禁用 Spring Session。

每个商店都有特定的附加设置。例如，可以为 JDBC 存储定制表的名称，如以下示例所示：

```
spring.session.jdbc.table-name=SESSIONS
```

要设置会话超时，可以使用 `spring.session.timeout` 属性。如果未设置该属性，则自动配置将回退到 `server.servlet.session.timeout` 的值。

44.对 JMX 的监测和管理

Java Management Extensions（JMX）提供了一种监视和管理应用程序的标准机制。默认情况下，Spring Boot 会创建一个 ID 为 `mbeanServer` 的 `MBeanServer` bean，并公开使用 Spring JMX 注释（`@ManagedResource` 注释的任何 beans， `@ManagedAttribute` 或 `@ManagedOperation`）。

有关 `JmxAutoConfiguration` 详细信息，请参阅 课程。

45.测试

Spring Boot 提供了许多实用程序和注释来帮助您测试应用程序。测试支持由两个模块提供：`spring-boot-test` 包含核心项，`spring-boot-test-autoconfigure` 支持测试的自动配置。

大多数开发人员使用 `spring-boot-starter-test`“Starter”，它导入 Spring Boot 测试模块以及

JUnit，AssertJ，Hamcrest 和许多其他有用的库。

45.1 测试范围依赖性

`spring-boot-starter-test`“Starter”（在 `test` scope 中）包含以下提供的库：

- JUnit：单元测试 Java 应用程序的事实标准。
- Spring 测试和 Spring Boot 测试：Spring Boot 应用程序的实用程序和集成测试支持。
- AssertJ：一个流畅的断言库。
- Hamcrest：匹配器对象库（也称为约束或谓词）。
- Mockito：一个 Java 模拟框架。
- JSONassert：JSON 的断言库。
- JsonPath：JSON 的 XPath。

我们通常发现这些常用库在编写测试时很有用。如果这些库不适合您的需求，您可以添加自己的其他测试依赖项。

45.2 测试 Spring 应用程序

依赖注入的一个主要优点是它应该使您的代码更容易进行单元测试。您可以使用 `new` 运算符实例化对象，甚至不涉及 Spring。您还可以使用模拟对象而不是真正的依赖项。

通常，您需要超越单元测试并开始集成测试（使用 Spring `ApplicationContext`）。能够在不需要部署应用程序或需要连接到其他基础架构的情况下执行集成测试非常有用。

Spring 框架包括用于此类集成测试的专用测试模块。您可以直接向 `org.springframework:spring-test` 声明依赖关系，或使用 `spring-boot-starter-test`“Starter”将其传递给它。

如果您之前未使用过 `spring-test` 模块，则应首先阅读 Spring 框架参考文档的 相关部分。

45.3 测试 Spring Boot 应用程序

Spring Boot 应用程序是 Spring `ApplicationContext`，因此除了通常使用 vanilla Spring 上下文所做的测试之外，没有什么特别的要做。

注意

仅当您使用 `SpringApplication` 创建外部属性，日志记录和 Spring Boot 的其他功能时，才会默认安装在上下文中。

Spring Boot 提供了`@SpringBootTest` 注释，当您需要 Spring 引导功能时，可以将其用作标准 `spring-test` `@ContextConfiguration` 注释的替代。注释的工作原理是 通过 `SpringApplication` 创建测试中使用的 `ApplicationContext`。除了`@SpringBootTest` 之外，还提供了许多其他注释来 测试应用程序的更具体的切片。

小费

如果您使用的是 JUnit 4，请不要忘记在测试中添加 `@RunWith(SpringRunner.class)`，否则注释将被忽略。如果您正在使用 JUnit 5，则无需将等效的`@ExtendWith(SpringExtension)` 添加为`@SpringBootTest`，而其他`@…Test` 注释已经使用它进行注

释。

默认情况下，@SpringBootTest 将无法启动服务器。您可以使用@SpringBootTest 的 webEnvironment 属性来进一步优化测试的运行方式：

- MOCK（默认）：加载网络 ApplicationContext 并提供模拟网络环境。使用此批注时，不会启动嵌入式服务器。如果您的类路径上没有 Web 环境，则此模式将透明地回退到创建常规非 Web ApplicationContext。它可以与 @AutoConfigureMockMvc 或 @AutoConfigureWebTestClient 一起用于基于模拟的 Web 应用程序测试。
- RANDOM_PORT：加载 WebServerApplicationContext 并提供真实的网络环境。嵌入式服务器启动并在随机端口上侦听。
- DEFINED_PORT：加载 WebServerApplicationContext 并提供真实的网络环境。嵌入式服务器启动并侦听定义的端口（来自您的 application.properties）或默认端口 8080。
- NONE：使用 SpringApplication 加载 ApplicationContext 但不提供 任何网络环境（模拟或其他）。

注意

如果您的测试是@Transactional，则默认情况下会在每个测试方法的末尾回滚事务。但是，当使用 RANDOM_PORT 或 DEFINED_PORT 的这种安排隐式地提供真正的 servlet 环境时，HTTP 客户端和服务器在单独的线程中运行，因此在单独的事务中运行。在这种情况下，在服务器上启动的任何事务都不会回滚。

注意

如果您的应用程序使用不同的管理服务器端口，@SpringBootTest 和 webEnvironment = WebEnvironment.RANDOM_PORT 也将在单独的随机端口上启动管理服务器。

45.3.1 检测 Web 应用程序类型

如果 Spring MVC 可用，则配置基于 MVC 的常规应用程序上下文。如果您只有 Spring WebFlux，我们将检测到并配置基于 WebFlux 的应用程序上下文。

如果两者都存在，Spring MVC 优先。如果要在此方案中测试响应式 Web 应用程序，则必须设置 spring.main.web-application-type 属性：

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.main.web-application-type=reactive")
public class MyWebFluxTests { ... }
```

45.3.2 检测测试配置

如果您熟悉 Spring 测试框架，则可能习惯使用@ContextConfiguration(classes=…)来指定要加载的 Spring @Configuration。或者，您可能经常在测试中使用嵌套的@Configuration 类。

在测试 Spring Boot 应用程序时，通常不需要这样做。只要您没有明确定义一个，Spring Boot 的@*Test 注释就会自动搜索您的主要配置。

搜索算法从包含测试的包开始工作，直到找到使用@SpringBootApplication 或 @SpringBootConfiguration 注释的类。只要您以合理的方式构建代码，通常就会找到主要

配置。

注意

如果使用 测试批注来测试应用程序的更具体的片段，则应避免在 main 方法的应用程序类中添加特定于特定区域的配置设置 。

@SpringBootApplication 的基础组件扫描配置定义了排除过滤器，用于确保切片按预期工作。如果您在@SpringBootApplication - 带注释的类上使用明确的@ComponentScan 指令，请注意这些过滤器将被禁用。如果您正在使用切片，则应再次定义它们。

如果要自定义主要配置，可以使用嵌套的@TestConfiguration 类。与嵌套的 @Configuration 类不同，它将用于代替应用程序的主要配置，除了应用程序的主要配置之外，还使用嵌套的@TestConfiguration 类。

注意

Spring 的测试框架在测试之间缓存应用程序上下文。因此，只要您的测试共享相同的配置（无论如何发现），加载上下文的潜在耗时过程只发生一次。

45.3.3 排除测试配置

如果您的应用程序使用组件扫描（例如，如果您使用@SpringBootApplication 或 @ComponentScan），您可能会发现仅为特定测试创建的顶级配置类会意外地在任何地方进行检索。

如前所述，@TestConfiguration 可用于测试的内部类以自定义主要配置。前面所看到的，1941 年{/}可以在一个内部类的测试的用于定制的主配置。当放置在顶级类时，@TestConfiguration 表示不应通过扫描拾取 src/test/java 中的类。然后，您可以在需要的位置显式导入该类，如以下示例所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest
@Import(MyTestsConfiguration.class)
public class MyTests {

        @Test
        public void exampleTest() {
                ...
        }

}
```

注意

如果您直接使用@ComponentScan（即不通过 @SpringBootApplication），则需要使用 TypeExcludeFilter 注册。有关详细信息，请参阅 Javadoc。

45.3.4 使用模拟环境进行测试

默认情况下，@SpringBootTest 无法启动服务器。如果您要针对此模拟环境测试 Web 端点，则可以另外进行配置 MockMvc，如以下示例所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class MockMvcExampleTests {

        @Autowired
        private MockMvc mvc;

        @Test
        public void exampleTest() throws Exception {
                this.mvc.perform(get("/")).andExpect(status().isOk())
                                .andExpect(content().string("Hello World"));
        }

}
```

小费

如果您只想关注网络层而不是开始一个完整的 ApplicationContext，请考虑 使用@WebMvcTest。

或者，您可以配置 a WebTestClient，如以下示例所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWebTestClient
public class MockWebTestClientExampleTests {

        @Autowired
        private WebTestClient webClient;

        @Test
        public void exampleTest() {
                this.webClient.get().uri("/").exchange().expectStatus().isOk()
                                .expectBody(String.class).isEqualTo("Hello World");
        }

}
```

45.3.5 使用正在运行的服务器进行测试

如果您需要启动完整运行的服务器，我们建议您使用随机端口。如果使用`@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`，则每次测试运行时随机选择一个可用端口。

`@LocalServerPort` 注释可用于 注入测试中使用的实际端口。为方便起见，需要对启动的服务器进行 REST 调用的测试还可以`@Autowire` a `WebTestClient`，它解析了与正在运行的服务器的相对链接，并附带了用于验证响应的专用 API，如以下示例所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortWebTestClientExampleTests {

        @Autowired
        private WebTestClient webClient;

        @Test
        public void exampleTest() {
                this.webClient.get().uri("/").exchange().expectStatus().isOk()
                                .expectBody(String.class).isEqualTo("Hello World");
        }

}
```

此设置在类路径上需要`spring-webflux`。如果您不能或不会添加 webflux，Spring Boot 还提供`TestRestTemplate`设施：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortTestRestTemplateExampleTests {

        @Autowired
        private TestRestTemplate restTemplate;

        @Test
        public void exampleTest() {
                String body = this.restTemplate.getForObject("/", String.class);
                assertThat(body).isEqualTo("Hello World");
        }

}
```

45.3.6 使用 JMX

当测试上下文框架缓存上下文时，默认情况下禁用 JMX 以防止相同的组件在同一域上注册。如果此类测试需要访问 `MBeanServer`，请考虑将其标记为脏：

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
public class SampleJmxTests {

        @Autowired
        private MBeanServer mBeanServer;

        @Test
        public void exampleTest() {
                // ...
        }

}
```

45.3.7 嘲弄和间谍活动 Beans

运行测试时，有时需要在应用程序上下文中模拟某些组件。例如，您可能拥有在开发期间不可用的某些远程服务的外观。当您想要模拟在真实环境中可能难以触发的故障时，模拟也很有用。

Spring Boot 包含`@MockBean` 注释，可用于为 `ApplicationContext` 内的 bean 定义 Mockito 模拟。您可以使用注释添加新的 beans 或替换单个现有的 bean 定义。注释可以直接用于测试类，测试中的字段或`@Configuration` 类和字段。在字段上使用时，也会注入创建的模拟的实例。模拟 beans 在每种测试方法后自动重置。

注意

如果您的测试使用 Spring Boot 的测试注释之一（例如 `@SpringBootTest`），则会自动启用此功能。要以不同的排列方式使用此功能，必须显式添加侦听器，如以下示例所示：

```
@TestExecutionListeners(MockitoTestExecutionListener.class)
```

以下示例使用模拟实现替换现有的 `RemoteService` bean：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

        @MockBean
        private RemoteService remoteService;

        @Autowired
        private Reverser reverser;

        @Test
        public void exampleTest() {
```

```
                // RemoteService has been injected into the reverser bean
                given(this.remoteService.someCall()).willReturn("mock");
                String reverse = reverser.reverseSomeCall();
                assertThat(reverse).isEqualTo("kcom");
        }

}
```

此外，您可以使用@SpyBean 将任何现有的 bean 与 Mockito spy 包装在一起。有关详细信息，请参阅 Javadoc。

注意

虽然 Spring 的测试框架在测试之间缓存应用程序上下文并重用共享相同配置的测试的上下文，但使用@MockBean 或@SpyBean 会影响缓存密钥，这很可能会增加缓存密钥的数量。上下文。

小费

如果您使用@SpyBean 监视 bean 并使用@Cacheable 方法按名称引用参数，则必须使用-parameters 编译应用程序。这可以确保在 bean 被监视后，参数名称可用于缓存基础结构。

45.3.8 自动配置的测试

Spring Boot 的自动配置系统适用于应用程序，但有时对于测试来说有点太多了。通常，只需加载测试应用程序“切片”所需的配置部分。例如，您可能希望测试 Spring MVC 控制器是否正确映射 URL，并且您不希望在这些测试中涉及数据库调用，或者您可能想要测试 JPA 实体，并且您对 Web 不感兴趣这些测试运行时的图层。

spring-boot-test-autoconfigure 模块包括许多可用于自动配置这种“切片”的注释。它们中的每一个都以类似的方式工作，提供@…Test 注释，用于加载 ApplicationContext 和一个或多个@AutoConfigure…注释，可用于自定义自动配置设置。

注意

每个切片都将组件扫描限制为适当的组件，并加载一组非常有限的自动配置类。如果您需要排除其中一个，则大多数@…Test 注释都会提供 excludeAutoConfiguration 属性。或者，您可以使用 @ImportAutoConfiguration#exclude。

注意

不支持在一次测试中使用多个@…Test 注释包含多个“切片”。如果您需要多个“切片”，请选择@…Test 注释之一并手动包含其他“切片”的 @AutoConfigure…注释。

小费

也可以将@AutoConfigure…注释与标准@SpringBootTest 注释一起使用。如果您对“切片”应用程序不感兴趣，但想要一些自动配置的测试 beans，则可以使用此组合。

45.3.9 自动配置的 JSON 测试

要测试该对象 JSON 序列化和反序列化是否按预期工作，您可以使用`@JsonTest`注释。`@JsonTest`自动配置可用的受支持的 JSON 映射器，它可以是以下库之一：

- Jackson `ObjectMapper`，任何`@JsonComponent` beans 和任何 Jackson `Module` s
- `Gson`
- `Jsonb`

小费

可以在附录中找到 `@JsonTest` 启用的自动配置列表 。

如果需要配置自动配置的元素，可以使用`@AutoConfigureJsonTesters`注释。

Spring Boot 包括基于 AssertJ 的助手，它们与 JSONAssert 和 JsonPath 库一起使用，以检查 JSON 是否按预期显示。`JacksonTester`，`GsonTester`，`JsonbTester` 和 `BasicJsonTester` 类可分别用于 Jackson，Gson，Jsonb 和 Strings。使用`@JsonTest`时，测试类上的任何辅助字段都可以是`@Autowired`。以下示例显示了 Jackson 的测试类：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

        @Autowired
        private JacksonTester<VehicleDetails> json;

        @Test
        public void testSerialize() throws Exception {
                VehicleDetails details = new VehicleDetails("Honda", "Civic");
                // Assert against a `.json` file in the same package as the test
                assertThat(this.json.write(details)).isEqualToJson("expected.json");
                // Or use JSON path based assertions

assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");

assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make")
                                .isEqualTo("Honda");
        }

        @Test
        public void testDeserialize() throws Exception {
                String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
                assertThat(this.json.parse(content))
                                .isEqualTo(new VehicleDetails("Ford", "Focus"));

assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
        }

}
```

注意

JSON 帮助程序类也可以直接用于标准单元测试。为此，如果不使用`@JsonTest`，请在`@Before`方法中调用助手的`initFields`方法。

45.3.10 自动配置的 Spring MVC 测试

要测试 Spring MVC 控制器是否按预期工作，请使用`@WebMvcTest`注释。`@WebMvcTest`自动配置 Spring MVC 基础设施并将扫描 beans 限制为`@Controller`，`@ControllerAdvice`，`@JsonComponent`，`Converter`，`GenericConverter`，`Filter`，`WebMvcConfigurer`和`HandlerMethodArgumentResolver`。使用此注释时，不会扫描常规`@Component` beans。

小费

可以在附录中找到 `@WebMvcTest`启用的自动配置设置列表 。

小费

如果您需要注册额外的组件，例如 Jackson `Module`，则可以在测试中使用`@Import`导入其他配置类。

通常，`@WebMvcTest`仅限于一个控制器，并与`@MockBean`结合使用，为所需的协作者提供模拟实现。

`@WebMvcTest`也自动配置`MockMvc`。Mock MVC 提供了一种快速测试 MVC 控制器的强大方法，无需启动完整的 HTTP 服务器。

小费

您还可以使用`@AutoConfigureMockMvc`对其进行注释，以非`@WebMvcTest`（例如`@SpringBootTest`）自动配置`MockMvc`。以下示例使用`MockMvc`：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

        @Autowired
        private MockMvc mvc;

        @MockBean
        private UserVehicleService userVehicleService;

        @Test
        public void testExample() throws Exception {
                given(this.userVehicleService.getVehicleDetails("sboot"))
```

```
                        .willReturn(new VehicleDetails("Honda", "Civic"));
                this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))

.andExpect(status().isOk()).andExpect(content().string("Honda Civic"));
        }

}
```

小费

如果需要配置自动配置的元素（例如，应该应用 servlet 过滤器时），可以使用`@AutoConfigureMockMvc`注释中的属性。

如果您使用 HtmlUnit 或 Selenium，则自动配置还会提供 HTMLUnit `WebClient` bean 和/或 `WebDriver` bean。以下示例使用 HtmlUnit：

```
import com.gargoylesoftware.htmlunit.*;
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

        @Autowired
        private WebClient webClient;

        @MockBean
        private UserVehicleService userVehicleService;

        @Test
        public void testExample() throws Exception {
                given(this.userVehicleService.getVehicleDetails("sboot"))
                                .willReturn(new VehicleDetails("Honda", "Civic"));
                HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
                assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
        }

}
```

注意

默认情况下，Spring Boot 将`WebDriver` beans 置于特殊的“范围”中，以确保驱动程序在每次测试后退出并注入新实例。如果您不想要此行为，可以将`@Scope("singleton")`添加到`WebDriver` `@Bean`定义中。

警告

Spring Boot 创建的`webDriver`范围将替换任何用户定义的同名范围。如果您定义自己的`webDriver`范围，则在使用`@WebMvcTest`时可能会发现它停止工作。

如果您在类路径上拥有 Spring 安全性，`@WebMvcTest`也会扫描`WebSecurityConfigurer` beans。您可以使用 Spring 安全性测试支持，而不是完全禁用此类测试的安全性。有关如何使用

Spring 安全性 MockMvc 支持的更多详细信息，请参阅本章 *80*，使用 *Spring* 安全性操作方法部分进行测试。

小费

有时写 Spring MVC 测试是不够的; Spring Boot 可以帮助您使用实际服务器运行 完整的端到端测试。

45.3.11 自动配置 Spring WebFlux 测试

要测试 Spring WebFlux 控制器是否按预期工作，您可以使用@WebFluxTest 注释。@WebFluxTest 自动配置 Spring WebFlux 基础架构，并将扫描的 beans 限制为 @Controller，@ControllerAdvice，@JsonComponent，Converter，GenericConverter 和 WebFluxConfigurer。使用@WebFluxTest 注释时，不会扫描常规@Component beans。

小费

可以在附录中找到 @WebFluxTest 启用的自动配置列表 。

小费

如果需要注册额外的组件，例如 Jackson Module，则可以在测试中使用@Import 导入其他配置类。

通常，@WebFluxTest 仅限于单个控制器，并与@MockBean 注释结合使用，为所需的协作者提供模拟实现。

@WebFluxTest 也是自动配置 WebTestClient，它提供了一种快速测试 WebFlux 控制器的强大方法，无需启动完整的 HTTP 服务器。

小费

您还可以通过使用@AutoConfigureWebTestClient 对其进行注释，在非@WebFluxTest（例如@SpringBootTest）中自动配置 WebTestClient。以下示例显示了同时使用@WebFluxTest 和 WebTestClient 的类：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@WebFluxTest(UserVehicleController.class)
public class MyControllerTests {

        @Autowired
        private WebTestClient webClient;

        @MockBean
        private UserVehicleService userVehicleService;
```

```
        @Test
        public void testExample() throws Exception {
                given(this.userVehicleService.getVehicleDetails("sboot"))
                                .willReturn(new VehicleDetails("Honda", "Civic"));

this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN)
                                .exchange()
                                .expectStatus().isOk()
                                .expectBody(String.class).isEqualTo("Honda Civic");
        }

}
```

小费

此设置仅由 WebFlux 应用程序支持，因为在模拟的 Web 应用程序中使用 `WebTestClient` 目前仅适用于 WebFlux。

注意

`@WebFluxTest` 无法检测通过功能 Web 框架注册的路由。要在上下文中测试 `RouterFunction` beans，请考虑通过`@Import` 或使用 `@SpringBootTest` 自行导入 `RouterFunction`。

小费

有时写 Spring WebFlux 测试是不够的; Spring Boot 可以帮助您使用实际服务器运行 完整的端到端测试。

45.3.12 自动配置的数据 JPA 测试

您可以使用`@DataJpaTest` 注释来测试 JPA 应用程序。默认情况下，它配置内存中的嵌入式数据库，扫描`@Entity` 类，并配置 Spring Data JPA 存储库。常规`@Component` beans 未加载到 `ApplicationContext`。

小费

可以在附录中找到 `@DataJpaTest` 启用的自动配置设置列表 。

默认情况下，数据 JPA 测试是事务性的，并在每次测试结束时回滚。有关 更多详细信息，请参阅 Spring 框架参考文档中的相关部分。如果这不是您想要的，您可以为测试或整个类禁用事务管理，如下所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

数据 JPA 测试也可以注入 `TestEntityManager` bean，它提供了专门为测试设计的标准 JPA

EntityManager 的替代方案。如果要在@DataJpaTest 实例之外使用 TestEntityManager，还可以使用@AutoConfigureTestEntityManager 注释。如果您需要，也可以使用 JdbcTemplate。以下示例显示正在使用的@DataJpaTest 注释：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

        @Autowired
        private TestEntityManager entityManager;

        @Autowired
        private UserRepository repository;

        @Test
        public void testExample() throws Exception {
                this.entityManager.persist(new User("sboot", "1234"));
                User user = this.repository.findByUsername("sboot");
                assertThat(user.getUsername()).isEqualTo("sboot");
                assertThat(user.getVin()).isEqualTo("1234");
        }

}
```

内存中嵌入式数据库通常适用于测试，因为它们速度快且不需要任何安装。但是，如果您更喜欢对真实数据库运行测试，则可以使用@AutoConfigureTestDatabase 注释，如以下示例所示：

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class ExampleRepositoryTests {

        // ...

}
```

45.3.13 自动配置的 JDBC 测试

@JdbcTest 类似于@DataJpaTest，但适用于仅需要 DataSource 并且不使用 Spring 数据 JDBC 的测试。默认情况下，它配置内存中嵌入式数据库和 JdbcTemplate。常规@Component beans 未加载到 ApplicationContext。

小费

可以在附录中找到 @JdbcTest 启用的自动配置列表 。

默认情况下，JDBC 测试是事务性的，并在每次测试结束时回滚。有关更多详细信息，请参阅 Spring 框架参考文档中的 相关部分。如果这不是您想要的，您可以禁用测试或整个类的事务管理，如下所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
```

```
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

如果您希望测试针对真实数据库运行，则可以使用@AutoConfigureTestDatabase 注释，方法与 DataJpaTest 相同。（参见“ 第 45.3.12 节 ”，“自动配置的数据 JPA 测试”。）

45.3.14 自动配置的数据 JDBC 测试

@DataJdbcTest 类似于@JdbcTest，但适用于使用 Spring 数据 JDBC 存储库的测试。默认情况下，它配置内存中的嵌入式数据库，JdbcTemplate 和 Spring 数据 JDBC 存储库。常规 @Component beans 未加载到 ApplicationContext。

小费

可以在附录中找到 @DataJdbcTest 启用的自动配置列表 。

默认情况下，数据 JDBC 测试是事务性的，并在每次测试结束时回滚。有关 更多详细信息，请参阅 Spring 框架参考文档中的相关部分。如果这不是您想要的，您可以禁用测试或整个测试类的事务管理，如 JDBC 示例中所示。

如果您希望测试针对真实数据库运行，则可以使用@AutoConfigureTestDatabase 注释，方法与 DataJpaTest 相同。（参见“ 第 45.3.12 节 ”，“自动配置的数据 JPA 测试”。）

45.3.15 自动配置的 jOOQ 测试

您可以使用与@JdbcTest 类似的方式使用@JooqTest，但是可以使用与 jOOQ 相关的测试。由于 jOOQ 严重依赖于与数据库模式相对应的基于 Java 的模式，因此使用现有的 DataSource。如果要将其替换为内存数据库，可以使用@AutoConfigureTestDatabase 覆盖这些设置。（有关在 Spring Boot 中使用 jOOQ 的更多信息，请参阅本章前面的“ 第 30.6 节 ”，“使用 jOOQ”。）常规@Component beans 未加载到 ApplicationContext。

小费

可以在附录中找到 @JooqTest 启用的自动配置列表 。

@JooqTest 配置 DSLContext。常规@Component beans 未加载到 ApplicationContext。以下示例显示正在使用的@JooqTest 注释：

```
import org.jooq.DSLContext;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@JooqTest
public class ExampleJooqTests {

        @Autowired
        private DSLContext dslContext;
}
```

JOOQ 测试是事务性的，默认情况下在每个测试结束时回滚。如果这不是您想要的，您可以禁用测试或整个测试类的事务管理，如 JDBC 示例中所示。

45.3.16 自动配置的数据 MongoDB 测试

您可以使用`@DataMongoTest` 来测试 MongoDB 应用程序。默认情况下，它配置内存中嵌入的 MongoDB（如果可用），配置 `MongoTemplate`，扫描`@Document` 类，并配置 Spring Data MongoDB 存储库。常规`@Component` beans 未加载到 `ApplicationContext`。（有关将 MongoDB 与 Spring Boot 一起使用的更多信息，请参阅本章前面的“ 第 31.2 节”，“MongoDB”。）

小费

可以在附录中找到 `@DataMongoTest` 启用的自动配置设置列表 。

以下类显示正在使用的`@DataMongoTest` 注释：

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest
public class ExampleDataMongoTests {

        @Autowired
        private MongoTemplate mongoTemplate;

        //
}
```

内存中嵌入式 MongoDB 通常适用于测试，因为它速度快，不需要任何开发人员安装。但是，如果您更喜欢对真正的 MongoDB 服务器运行测试，则应排除嵌入式 MongoDB 自动配置，如以下示例所示：

```
import org.junit.runner.RunWith;
 import
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)
public class ExampleDataMongoNonEmbeddedTests {

}
```

45.3.17 自动配置的数据 Neo4j 测试

您可以使用`@DataNeo4jTest` 来测试 Neo4j 应用程序。默认情况下，它使用内存中嵌入式 Neo4j（如果嵌入式驱动程序可用），扫描`@NodeEntity` 类，并配置 Spring Data Neo4j 存储库。常规`@Component` beans 未加载到 `ApplicationContext`。（有关使用带有 Spring Boot 的 Neo4J 的更多信息，请参阅本章前面的“ 第 31.3 节”，“Neo4j”。）

小费

可以在附录中找到 `@DataNeo4jTest` 启用的自动配置设置列表 。

以下示例显示了在 Spring Boot 中使用 Neo4J 测试的典型设置：

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataNeo4jTest
public class ExampleDataNeo4jTests {

        @Autowired
        private YourRepository repository;

        //
}
```

默认情况下，Data Neo4j 测试是事务性的，并在每次测试结束时回滚。有关更多详细信息，请参阅 Spring 框架参考文档中的相关部分。如果这不是您想要的，您可以禁用测试或整个类的事务管理，如下所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

45.3.18 自动配置的数据 Redis 测试

您可以使用`@DataRedisTest`来测试 Redis 应用程序。默认情况下，它会扫描`@RedisHash`类并配置 Spring Data Redis 存储库。常规`@Component` beans 未加载到`ApplicationContext`。（有关将 Redis 与 Spring Boot 一起使用的更多信息，请参阅本章前面的“ 第 31.1 节”，“37 /}”。

小费

可以在附录中找到 `@DataRedisTest`启用的自动配置设置列表 。

以下示例显示正在使用的`@DataRedisTest`注释：

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataRedisTest
public class ExampleDataRedisTests {

        @Autowired
        private YourRepository repository;

        //
```

```
}
```

45.3.19 自动配置的数据 LDAP 测试

您可以使用`@DataLdapTest`来测试 LDAP 应用程序。默认情况下，它配置内存中嵌入式 LDAP（如果可用），配置`LdapTemplate`，扫描`@Entry`类，并配置 Spring 数据 LDAP 存储库。常规`@Component` beans 未加载到`ApplicationContext`。（有关将 LDAP 与 Spring Boot 一起使用的更多信息，请参阅本章前面的“ 第 31.9 节”，“LDAP”。）

小费

可以在附录中找到 `@DataLdapTest`启用的自动配置设置列表 。

以下示例显示正在使用的`@DataLdapTest`注释：

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest
public class ExampleDataLdapTests {

        @Autowired
        private LdapTemplate ldapTemplate;

        //
}
```

内存中嵌入式 LDAP 通常适用于测试，因为它速度快，不需要任何开发人员安装。但是，如果您希望针对真实 LDAP 服务器运行测试，则应排除嵌入式 LDAP 自动配置，如以下示例所示：

```
import org.junit.runner.RunWith;
import
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
public class ExampleDataLdapNonEmbeddedTests {

}
```

45.3.20 自动配置的 REST 客户端

您可以使用`@RestClientTest`注释来测试 REST 客户端。默认情况下，它会自动配置 Jackson，GSON 和 Jsonb 支持，配置`RestTemplateBuilder`，并添加对`MockRestServiceServer`的支持。常规`@Component` beans 未加载到`ApplicationContext`。

小费

可以在附录中找到 `@RestClientTest`启用的自动配置设置列表。

应使用`@RestClientTest`的`value`或`components`属性指定要测试的特定 beans，如以下示例所示：

```
@RunWith(SpringRunner.class)
@RestClientTest(RemoteVehicleDetailsService.class)
public class ExampleRestClientTest {

	@Autowired
	private RemoteVehicleDetailsService service;

	@Autowired
	private MockRestServiceServer server;

	@Test
	public void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails()
			throws Exception {
		this.server.expect(requestTo("/greet/details"))
				.andRespond(withSuccess("hello",
MediaType.TEXT_PLAIN));
		String greeting = this.service.callRestService();
		assertThat(greeting).isEqualTo("hello");
	}

}
```

45.3.21 自动配置的 Spring REST 文档测试

您可以使用`@AutoConfigureRestDocs`注释在 Mock MVC，REST Assured 或 WebTestClient 的测试中使用 Spring REST Docs。它消除了对 Spring REST Docs 中 JUnit 规则的需求。

`@AutoConfigureRestDocs`可用于覆盖默认输出目录（如果您使用 Maven，则为`target/generated-snippets`;如果您使用 Gradle，则为`build/generated-snippets`）。它还可用于配置出现在任何已记录的 URI 中的主机，方案和端口。

使用 Mock MVC 自动配置 Spring REST 文档测试

`@AutoConfigureRestDocs`自定义`MockMvc` bean 以使用 Spring REST 文档。您可以使用`@Autowired`注入它并在测试中使用它，就像使用 Mock MVC 和 Spring REST Docs 时一样，如下例所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
public class UserDocumentationTests {

	@Autowired
	private MockMvc mvc;

	@Test
	public void listUsers() throws Exception {
		this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
				.andExpect(status().isOk())
				.andDo(document("list-users"));
	}
```

```
}
```

如果您需要对 Spring REST Docs 配置的更多控制，而不是`@AutoConfigureRestDocs`属性提供的控制，则可以使用`RestDocsMockMvcConfigurationCustomizer` bean，如以下示例所示：

```
@TestConfiguration
static class CustomizationConfiguration
                implements RestDocsMockMvcConfigurationCustomizer {

        @Override
        public void customize(MockMvcRestDocumentationConfigurer configurer) {
                configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
        }

}
```

如果要对参数化输出目录使用 Spring REST Docs 支持，可以创建`RestDocumentationResultHandler` bean。自动配置使用此结果处理程序调用`alwaysDo`，从而导致每个`MockMvc`调用自动生成默认代码段。以下示例显示正在定义的`RestDocumentationResultHandler`：

```
@TestConfiguration
static class ResultHandlerConfiguration {

        @Bean
        public RestDocumentationResultHandler restDocumentation() {
                return MockMvcRestDocumentation.document("{method-name}");
        }

}
```

使用 REST Assured 自动配置 Spring REST 文档测试

`@AutoConfigureRestDocs`生成`RequestSpecification` bean，预先配置为使用 Spring REST 文档，可用于您的测试。您可以使用`@Autowired`注入它并在测试中使用它，就像使用 REST Assured 和 Spring REST Docs 时一样，如下例所示：

```
import io.restassured.specification.RequestSpecification;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.junit4.SpringRunner;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;
import static
org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.document;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
public class UserDocumentationTests {

        @LocalServerPort
        private int port;

        @Autowired
```

```
        private RequestSpecification documentationSpec;

        @Test
        public void listUsers() {
                given(this.documentationSpec).filter(document("list-users")).when()

.port(this.port).get("/").then().assertThat().statusCode(is(200));
        }

}
```

如果您需要对 Spring REST Docs 配置进行更多控制，而不是`@AutoConfigureRestDocs`属性提供的控制，则可以使用`RestDocsRestAssuredConfigurationCustomizer` bean，如以下示例所示：

```
@TestConfiguration
public static class CustomizationConfiguration
                implements RestDocsRestAssuredConfigurationCustomizer {

        @Override
        public void customize(RestAssuredRestDocumentationConfigurer configurer) {
                configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
        }

}
```

45.3.22 附加自动配置和切片

每个切片提供一个或多个`@AutoConfigure…`注释，即定义应作为切片的一部分包括的自动配置。可以通过创建自定义`@AutoConfigure…`注释或仅通过向测试添加`@ImportAutoConfiguration`来添加其他自动配置，如以下示例所示：

```
@RunWith(SpringRunner.class)
@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration.class)
public class ExampleJdbcTests {

}
```

注意

确保不使用常规`@Import`注释来导入自动配置，因为 Spring Boot 以特定方式处理它们。

45.3.23 用户配置和切片

如果以合理的方式构造代码，默认情况下会使用 `@SpringBootApplication`类 作为测试的配置。

然后，重要的是不要使用特定于其功能的特定区域的配置设置来丢弃应用程序的主类。

假设您正在使用 Spring Batch，并依赖于它的自动配置。您可以按如下方式定义`@SpringBootApplication`：

```
@SpringBootApplication
@EnableBatchProcessing
public class SampleApplication { ... }
```

因为此类是测试的源配置，所以任何切片测试实际上都会尝试启动 Spring Batch，这绝对不是您想要做的。建议的方法是将特定于区域的配置移动到与应用程序相同级别的单独`@Configuration`

类，如以下示例所示：

```
@Configuration
@EnableBatchProcessing
public class BatchConfiguration { ... }
```

注意

根据应用程序的复杂程度，您可能只有一个`@Configuration`类用于自定义，或者每个域区域有一个类。后一种方法允许您在必要时使用`@Import`注释在其中一个测试中启用它。

混淆的另一个原因是类路径扫描。假设您以合理的方式构建代码，则需要扫描其他包。您的应用程序可能类似于以下代码：

```
@SpringBootApplication
@ComponentScan({ "com.example.app", "org.acme.another" })
public class SampleApplication { ... }
```

这样做会有效地覆盖默认的组件扫描指令，无论您选择哪个切片，都会扫描这两个包。例如，`@DataJpaTest`似乎突然扫描应用程序的组件和用户配置。同样，将自定义指令移动到单独的类是解决此问题的好方法。

小费

如果这不是您的选项，您可以在测试的层次结构中的某处创建一个`@SpringBootConfiguration`，以便使用它。或者，您可以为测试指定源，这会禁用查找默认源的行为。

45.3.24 使用 Spock 测试 Spring Boot 应用程序

如果您希望使用 Spock 来测试 Spring Boot 应用程序，您应该将 Spock 的`spock-spring`模块的依赖项添加到您的应用程序的构建中。`spock-spring`将 Spring 的测试框架集成到 Spock 中。建议您使用 Spock 1.2 或更高版本从 Spock 的 Spring 框架和 Spring Boot 集成的许多改进中受益。有关更多详细信息，请参阅 Spock 的 Spring 模块的文档。

45.4 测试实用程序

测试应用程序时通常有用的一些测试实用程序类打包为`spring-boot`的一部分。

45.4.1 ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer`是`ApplicationContextInitializer`，您可以将其应用于测试以加载 Spring Boot `application.properties`个文件。当您不需要`@SpringBootTest`提供的全部功能时，可以使用它，如以下示例所示：

```
@ContextConfiguration(classes = Config.class,
        initializers = ConfigFileApplicationContextInitializer.class)
```

注意

仅使用`ConfigFileApplicationContextInitializer`不能为`@Value("${…}")`注射提供支持。它唯一的工作是确保将

application.properties 个文件加载到 Spring 的 Environment 中。对于@Value 支持，您需要另外配置 PropertySourcesPlaceholderConfigurer 或使用 @SpringBootTest，它会为您自动配置一个。

45.4.2 TestPropertyValues

TestPropertyValues 可让您快速向 ConfigurableEnvironment 或 ConfigurableApplicationContext 添加属性。您可以使用 key=value 字符串调用它，如下所示：

```
TestPropertyValues.of("org=Spring", "name=Boot").applyTo(env);
```

45.4.3 OutputCapture

OutputCapture 是一个 JUnit Rule，可用于捕获 System.out 和 System.err 输出。您可以将捕获声明为@Rule，然后使用 toString()进行断言，如下所示：

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.rule.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {

        @Rule
        public OutputCapture capture = new OutputCapture();

        @Test
        public void testName() throws Exception {
                System.out.println("Hello World!");
                assertThat(capture.toString(), containsString("World"));
        }

}
```

45.4.4 TestRestTemplate

小费

Spring Framework 5.0 提供了一个新的 WebTestClient，适用于 WebFlux 集成测试以及 WebFlux 和 MVC 端到端测试。与 TestRestTemplate 不同，它为断言提供了流畅的 API。

TestRestTemplate 是 Spring RestTemplate 的便利替代品，可用于集成测试。您可以获得一个 vanilla 模板或一个发送基本 HTTP 身份验证（使用用户名和密码）的模板。在任何一种情况下，模板都以一种测试友好的方式运行，不会在服务器端错误上抛出异常。建议（但不是强制性的）使用 Apache HTTP Client（版本 4.3.2 或更高版本）。如果您在类路径上有这个，那么 TestRestTemplate 通过适当地配置客户端来响应。如果您确实使用 Apache 的 HTTP 客户端，则启用一些其他测试友好功能：

- 不遵循重定向（因此您可以断言响应位置）。
- Cookie 被忽略（因此模板是无状态的）。

TestRestTemplate 可以直接在集成测试中实例化，如以下示例所示：

```
public class MyTest {

        private TestRestTemplate template = new TestRestTemplate();

        @Test
        public void testRequest() throws Exception {
                HttpHeaders headers = this.template.getForEntity(
                                        "http://myhost.example.com/example",
String.class).getHeaders();
                assertThat(headers.getLocation()).hasHost("other.example.com");
        }

}
```

或者，如果您将`@SpringBootTest`注释与`WebEnvironment.RANDOM_PORT`或`WebEnvironment.DEFINED_PORT`一起使用，则可以注入完全配置的`TestRestTemplate`并开始使用它。如有必要，可以通过`RestTemplateBuilder` bean 应用其他自定义设置。任何未指定主机和端口的 URL 都会自动连接到嵌入式服务器，如以下示例所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class SampleWebClientTests {

        @Autowired
        private TestRestTemplate template;

        @Test
        public void testRequest() {
                HttpHeaders headers = this.template.getForEntity("/example",
String.class)
                                        .getHeaders();
                assertThat(headers.getLocation()).hasHost("other.example.com");
        }

        @TestConfiguration
        static class Config {

                @Bean
                public RestTemplateBuilder restTemplateBuilder() {
                        return new
RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                                                .setReadTimeout(Duration.ofSeconds(1));
                }

        }

}
```

46. WebSockets

Spring Boot 为嵌入式 Tomcat，Jetty 和 Undertow 提供 WebSockets 自动配置。如果将 war 文件部署到独立容器，则 Spring Boot 假定容器负责其 WebSocket 支持的配置。

Spring Framework 为 MVC Web 应用程序提供了丰富的 WebSocket 支持，可以通过`spring-boot-starter-websocket`模块轻松访问。

WebSocket 支持也可用于 响应式 Web 应用程序，并且需要在`spring-boot-starter-webflux`旁边包含 WebSocket API：

```
<dependency>
        <groupId>javax.websocket</groupId>
        <artifactId>javax.websocket-api</artifactId>
</dependency>
```

47.网络服务

Spring Boot 提供 Web 服务自动配置，因此您必须做的就是定义 `Endpoints`。

该 Spring Web 服务功能，可以与 `spring-boot-starter-webservices` 模块可以轻松访问。

可以分别为您的 WSDL 和 XSD 自动创建 `SimpleWsdl11Definition` 和 `SimpleXsdSchema` beans。为此，请配置其位置，如以下示例所示：

```
spring.webservices.wsdl-locations=classpath:/wsdl
```

48.使用 `WebServiceTemplate` 调用 Web 服务

如果需要从应用程序调用远程 Web 服务，则可以使用 `WebServiceTemplate` 该类。由于 `WebServiceTemplate` 实例在使用之前通常需要进行自定义，因此 Spring Boot 不提供任何单个自动配置的 `WebServiceTemplate` bean。但是，它会自动配置 `WebServiceTemplateBuilder`，可用于在需要时创建 `WebServiceTemplate` 实例。

以下代码显示了一个典型示例：

```
@Service
public class MyService {

        private final WebServiceTemplate webServiceTemplate;

        public MyService(WebServiceTemplateBuilder webServiceTemplateBuilder) {
                this.webServiceTemplate = webServiceTemplateBuilder.build();
        }

        public DetailsResp someWsCall(DetailsReq detailsReq) {
                 return (DetailsResp)
this.webServiceTemplate.marshalSendAndReceive(detailsReq, new
SoapActionCallback(ACTION));

        }

}
```

默认情况下，`WebServiceTemplateBuilder` 使用类路径上的可用 HTTP 客户端库检测到合适的基于 HTTP 的 `WebServiceMessageSender`。您还可以按如下方式自定义读取和连接超时：

```
@Bean
public WebServiceTemplate webServiceTemplate(WebServiceTemplateBuilder builder) {
        return builder.messageSenders(new HttpWebServiceMessageSenderBuilder()

.setConnectTimeout(5000).setReadTimeout(2000).build()).build();
}
```

49.创建自己的自动配置

如果您在开发共享库的公司工作，或者您在开源或商业库中工作，则可能需要开发自己的自动配置。自动配置类可以捆绑在外部 jar 中，仍然可以通过 Spring Boot 获取。

自动配置可以与“启动器”相关联，该启动器提供自动配置代码以及您将使用它的典型库。我们首先介绍了构建自己的自动配置需要了解的内容，然后我们将继续介绍创建自定义启动器所需的典型步骤。

小费

可以使用演示项目来展示如何逐步创建启动器。

49.1 了解自动配置 Beans

在引擎盖下，自动配置使用标准@Configuration 类实现。额外的@Conditional 注释用于约束何时应用自动配置。通常，自动配置类使用@ConditionalOnClass 和 @ConditionalOnMissingBean 注释。这可确保仅在找到相关类时以及未声明自己的 @Configuration 时才应用自动配置。

您可以浏览源代码 spring-boot-autoconfigure 以查看 Spring 提供的@Configuration 类（请参阅 META-INF/spring.factories 文件）。

49.2 找到自动配置候选者

Spring Boot 检查已发布 jar 中是否存在 META-INF/spring.factories 文件。该文件应在 EnableAutoConfiguration 键下列出您的配置类，如以下示例所示：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

如果需要按特定顺序应用配置，则可以使用 @AutoConfigureAfter 或 @AutoConfigureBefore 注释。例如，如果您提供特定于 Web 的配置，则可能需要在 WebMvcAutoConfiguration 之后应用您的类。

如果您想订购某些不应该彼此直接了解的自动配置，您也可以使用@AutoConfigureOrder。该注释与常规@Order 注释具有相同的语义，但为自动配置类提供了专用顺序。

注意

自动配置，必须加载这种方式只。确保它们是在特定的包空间中定义的，特别是它们永远不是组件扫描的目标。

49.3 条件 Annotations

您几乎总是希望在自动配置类中包含一个或多个@Conditional 注释。@ConditionalOnMissingBean 注释是一个常见示例，用于允许开发人员在您的默认值不满意时覆盖自动配置。

Spring Boot 包含许多@Conditional 注释，您可以通过注释@Configuration 类或单独的 @Bean 方法在您自己的代码中重用这些注释。这些注释包括：

- 第 49.3.1 节“类别条件”
- 第 49.3.2 节，“Bean 条件”
- 第 49.3.3 节，“Property 条件”
- 第 49.3.4 节“资源条件”
- 第 49.3.5 节“Web 应用程序条件”
- 第 49.3.6 节“SpEL 表达条件”

49.3.1 等级条件

@ConditionalOnClass 和@ConditionalOnMissingClass 注释允许根据特定类的存在与否来包含配置。由于使用 ASM 解析注释元数据这一事实，您可以使用 value 属性来引用真实类，即使该类实际上可能不会出现在正在运行的应用程序类路径中。如果您希望使用 String 值指定类名，也可以使用 name 属性。

小费

如果您使用@ConditionalOnClass 或 @ConditionalOnMissingClass 作为元注释的一部分来编写自己的组合注释，则必须使用 name 作为引用该类的情况，在这种情况下不会处理。

49.3.2 Bean 条件

@ConditionalOnBean 和@ConditionalOnMissingBean 注释允许根据特定 beans 的存在与否来包括 bean。您可以使用 value 属性按类型指定 beans，或使用 name 按名称指定 beans。search 属性允许您限制搜索 beans 时应考虑的 ApplicationContext 层次结构。

置于@Bean 方法时，目标类型默认为方法的返回类型，如以下示例所示：

```
@Configuration
public class MyAutoConfiguration {

        @Bean
        @ConditionalOnMissingBean
        public MyService myService() { ... }

}
```

在前面的示例中，如果 ApplicationContext 中未包含类型 MyService 的 bean，则将创建 myService bean。

小费

您需要非常小心添加 bean 定义的顺序，因为这些条件是根据到目前为止已处理的内容进行评估的。因此，我们建议仅对自动配置类使用 @ConditionalOnBean 和@ConditionalOnMissingBean 注释（因为这些注释保证在添加任何用户定义的 bean 定义后加载）。

注意

@ConditionalOnBean 和@ConditionalOnMissingBean 不会阻止创建@Configuration 类。在类级别使用这些条件和使用注释标记每个包含@Bean 方法的唯一区别是，如果条件不匹配，前者会阻止将 @Configuration 类注册为 bean。

49.3.3 Property 条件

@ConditionalOnProperty 注释允许基于 Spring Environment 属性包含配置。使用 prefix 和 name 属性指定应检查的属性。默认情况下，匹配存在且不等于 false 的任何属性。您还可以使用 havingValue 和 matchIfMissing 属性创建更高级的检查。

49.3.4 资源条件

@ConditionalOnResource 注释仅在存在特定资源时才允许配置。可以使用通常的 Spring 约定来指定资源，如以下示例所示：file:/home/user/test.dat。

49.3.5 Web 应用程序条件

@ConditionalOnWebApplication 和@ConditionalOnNotWebApplication 注释允许配置，具体取决于应用程序是否为“Web 应用程序”。Web 应用程序是使用 Spring WebApplicationContext，定义 session 范围或具有 StandardServletEnvironment 的任何应用程序。

49.3.6 SpEL 表达条件

@ConditionalOnExpression 注释允许根据 SpEL 表达式的结果包含配置。

49.4 测试自动配置

自动配置可能受许多因素的影响：用户配置（@Bean 定义和 Environment 自定义），条件评估（存在特定库）等。具体而言，每个测试都应创建一个定义良好的 ApplicationContext，它代表这些自定义的组合。ApplicationContextRunner 提供了实现这一目标的好方法。

ApplicationContextRunner 通常被定义为测试类的一个字段，用于收集基本的通用配置。以下示例确保始终调用 UserServiceAutoConfiguration：

```
private final ApplicationContextRunner contextRunner = new ApplicationContextRunner()

.withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class));
```

小费

如果必须定义多个自动配置，则无需按照与运行应用程序时完全相同的顺序调用它们的声明。

每个测试都可以使用运行器来表示特定的用例。例如，下面的示例调用用户配置（UserConfiguration）并检查自动配置是否正确退回。调用 run 提供了一个可以与 Assert4J 一起使用的回调上下文。

```
@Test
public void defaultServiceBacksOff() {
        this.contextRunner.withUserConfiguration(UserConfiguration.class)
                        .run((context) -> {
                                assertThat(context).hasSingleBean(UserService.class);

assertThat(context.getBean(UserService.class)).isSameAs(

context.getBean(UserConfiguration.class).myUserService());
                        });
}

@Configuration
static class UserConfiguration {

        @Bean
        public UserService myUserService() {
                return new UserService("mine");
        }

}
```

也可以轻松自定义 Environment，如以下示例所示：

```
@Test
public void serviceNameCanBeConfigured() {
        this.contextRunner.withPropertyValues("user.name=test123").run((context) -> {
                assertThat(context).hasSingleBean(UserService.class);

assertThat(context.getBean(UserService.class).getName()).isEqualTo("test123");
        });
}
```

跑步者也可以用来显示 ConditionEvaluationReport。该报告可以 INFO 或 DEBUG 级别打印。以下示例显示如何使用 ConditionEvaluationReportLoggingListener 在自动配置测试中打印报表。

```
@Test
public void autoConfigTest {
        ConditionEvaluationReportLoggingListener initializer = new
ConditionEvaluationReportLoggingListener(
                        LogLevel.INFO);
        ApplicationContextRunner contextRunner = new ApplicationContextRunner()
                        .withInitializer(initializer).run((context) -> {
                                        // Do something...
                        });
}
```

49.4.1 模拟 Web 上下文

如果需要测试仅在 Servlet 或 Reactive Web 应用程序上下文中运行的自动配置，请分别使用 WebApplicationContextRunner 或 ReactiveWebApplicationContextRunner。

49.4.2 覆盖 Classpath

还可以测试在运行时不存在特定类和/或包时发生的情况。Spring Boot 装有 FilteredClassLoader，可以很容易地被跑步者使用。在以下示例中，我们声明如果不存在 UserService，则会正确禁用自动配置：

```
@Test
public void serviceIsIgnoredIfLibraryIsNotPresent() {
        this.contextRunner.withClassLoader(new
FilteredClassLoader(UserService.class))
                        .run((context) ->
assertThat(context).doesNotHaveBean("userService"));
}
```

49.5 创建自己的初学者

库的完整 Spring Boot 启动器可能包含以下组件：

- 包含自动配置代码的 autoconfigure 模块。
- starter 模块，它提供对 autoconfigure 模块以及库的依赖关系以及通常有用的任何其他依赖项。简而言之，添加启动器应该提供开始使用该库所需的一切。

小费

如果您不需要将这两个问题分开，则可以将自动配置代码和依赖关系管理组合在一个模块中。

49.5.1 命名

您应该确保为您的启动器提供适当的命名空间。即使您使用不同的 Maven `groupId`，也不要使用 `spring-boot` 启动模块名称。我们可能会为您将来自动配置的内容提供官方支持。

根据经验，您应该在启动后命名组合模块。例如，假设您正在为“acme”创建启动器，并且您将自动配置模块 `acme-spring-boot-autoconfigure` 和启动器 `acme-spring-boot-starter` 命名为。如果您只有一个组合两者的模块，请将其命名为 `acme-spring-boot-starter`。

此外，如果您的启动器提供配置密钥，请为它们使用唯一的命名空间。特别是，不要将密钥包含在 Spring Boot 使用的名称空间中（例如 `server`，`management`，`spring` 等）。如果您使用相同的命名空间，我们将来可能会以破坏您的模块的方式修改这些命名空间。

确保 触发元数据生成，以便为您的密钥提供 IDE 帮助。您可能需要查看生成的元数据（`META-INF/spring-configuration-metadata.json`）以确保正确记录您的密钥。

49.5.2 `autoconfigure` 模块

`autoconfigure` 模块包含开始使用库所需的所有内容。它还可能包含配置键定义（例如 `@ConfigurationProperties`）以及可用于进一步自定义组件初始化方式的任何回调接口。

小费

您应该将库的依赖项标记为可选，以便您可以更轻松地在项目中包含 `autoconfigure` 模块。如果您这样做，则不提供库，默认情况下，Spring Boot 会退出。

Spring Boot 使用注释处理器来收集元数据文件（`META-INF/spring-autoconfigure-metadata.properties`）中自动配置的条件。如果该文件存在，则用于热切过滤不匹配的自动配置，这将缩短启动时间。建议在包含自动配置的模块中添加以下依赖项：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure-processor</artifactId>
        <optional>true</optional>
</dependency>
```

对于 Gradle 4.5 及更早版本，依赖项应在 `compileOnly` 配置中声明，如以下示例所示：

```
dependencies {
        compileOnly "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

对于 Gradle 4.6 及更高版本，应在 `annotationProcessor` 配置中声明依赖项，如以下示例所示：

```
dependencies {
        annotationProcessor "org.springframework.boot:spring-boot-autoconfigure-
processor"
}
```

49.5.3 启动器模块

起动器真的是一个空罐子。它的唯一目的是提供必要的依赖项来使用库。您可以将其视为对入门所需内容的一种看法。

不要对添加启动器的项目做出假设。如果您自动配置的库通常需要其他启动器，请同时提及它

们。如果可选依赖项的数量很高，则提供一组适当的默认依赖项可能很难，因为您应该避免包含对典型库的使用不必要的依赖项。换句话说，您不应该包含可选的依赖项。

注意

无论哪种方式，您的启动器必须直接或间接引用核心 Spring Boot 启动器（`spring-boot-starter`）（即如果您的启动器依赖于另一个启动器，则无需添加它）。如果仅使用自定义启动程序创建项目，则核心启动程序将支持 Spring 引导的核心功能。

50. Kotlin 的支持

Kotlin 是一种针对 JVM（和其他平台）的静态类型语言，它允许编写简洁而优雅的代码，同时提供 与 Java 编写的现有库的互操作性。

Spring Boot 通过利用 Spring 框架，Spring 数据和反应堆等其他 Spring 项目的支持，提供 Kotlin 支持。有关 更多信息，请参阅 Spring Framework Kotlin 支持文档。

从 Spring Boot 和 Kotlin 开始的最简单方法是遵循 这个全面的教程。您可以通过 start.spring.io 创建新的 Kotlin 项目 。如果您需要支持，请随意加入 Kotlin Slack 的#spring 频道或在 Stack Overflow 上使用 `spring` 和 `kotlin` 标签提问。

50.1 要求

Spring Boot 支持 Kotlin 1.2.x. 要使用 Kotlin，类路径上必须存在 `org.jetbrains.kotlin:kotlin-stdlib` 和 `org.jetbrains.kotlin:kotlin-reflect`。也可以使用 `kotlin-stdlib` 变体 `kotlin-stdlib-jdk7` 和 `kotlin-stdlib-jdk8`。

由于 Kotlin 类默认为 final，因此您可能需要配置 kotlin-spring 插件以自动打开 Spring - 带注释的类，以便可以代理它们。

在 Kotlin 中序列化/反序列化 JSON 数据需要 Jackson 的 Kotlin 模块。在类路径中找到它时会自动注册。如果 Jackson 和 Kotlin 存在但 Jackson Kotlin 模块不存在，则会记录警告消息。

小费

如果在 start.spring.io 上引导 Kotlin 项目，则默认提供这些依赖项和插件。

50.2 无安全性

Kotlin 的一个关键特性是零安全性。它在编译时处理 `null` 值，而不是将问题推迟到运行时并遇到 `NullPointerException`。这有助于消除常见的错误来源，而无需支付 `Optional` 等包装器的成本。Kotlin 还允许使用具有可空值的功能构造，如本 Kotlin 中关于零安全性的综合指南中所述。

虽然 Java 不允许在其类型系统中表示 null 安全性，但 Spring Framework，Spring Data 和 Reactor 现在通过工具友好的注释提供其 API 的空安全性。默认情况下，Kotlin 中使用的 Java API 类型被识别为 放宽空检查的平台类型。 Kotlin 对 JSR 305 注释的支持与可空性注释相结合，为 Kotlin 中相关的 Spring API 提供了空的安全性。

可以通过使用以下选项添加`-Xjsr305`编译器标志来配置 JSR 305 检查：`-Xjsr305={strict|warn|ignore}`。默认行为与`-Xjsr305=warn`相同。`strict`值需要在从 Spring API 推断的 Kotlin 类型中考虑空安全性，但应该使用 Spring API 可空性声明甚至可以在次要版本和更多检查之间发展的知识可能会在将来添加）。

警告

尚不支持泛型类型参数，varargs 和数组元素可空性。有关最新信息，请参见 SPR-15942。另请注意，Spring Boot 自己的 API 尚未注释。

50.3 Kotlin API

50.3.1 runApplication

Spring Boot 提供了使用`runApplication<MyApplication>(*args)`运行应用程序的惯用方法，如以下示例所示：

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
        runApplication<MyApplication>(*args)
}
```

这是`SpringApplication.run(MyApplication::class.java, *args)`的替代品。它还允许自定义应用程序，如以下示例所示：

```
runApplication<MyApplication>(*args) {
        setBannerMode(OFF)
}
```

50.3.2 扩展

Kotlin 扩展提供了使用附加功能扩展现有类的能力。Spring Boot Kotlin API 利用这些扩展为现有 API 添加新的 Kotlin 特定便利。

`TestRestTemplate`扩展类似于 Spring 框架中 Caffeine 框架 Caffeine 提供的扩展。除此之外，扩展使得可以利用 Kotlin 具体类型参数。

50.4 依赖管理

为了避免在类路径上混合使用不同版本的 Kotlin 依赖项，提供了以下 Kotlin 依赖项的依赖项管理：

- `kotlin-reflect`
- `kotlin-runtime`
- `kotlin-stdlib`
- `kotlin-stdlib-jdk7`
- `kotlin-stdlib-jdk8`
- `kotlin-stdlib-jre7`
- `kotlin-stdlib-jre8`

使用 Maven，可以通过`kotlin.version`属性自定义 Kotlin 版本，并为`kotlin-maven-`

`plugin` 提供插件管理。使用 Gradle，Spring Boot 插件会自动将 `kotlin.version` 与 Kotlin 插件的版本对齐。

50.5 `@ConfigurationProperties`

`@ConfigurationProperties` 目前仅适用于 `lateinit` 或可空 `var` 属性（建议使用前者），因为尚不支持由构造函数初始化的不可变类。

```
@ConfigurationProperties("example.kotlin")
class KotlinExampleProperties {

        lateinit var name: String

        lateinit var description: String

        val myService = MyService()

        class MyService {

                lateinit var apiToken: String

                lateinit var uri: URI

        }

}
```

小费

要使用注释处理器生成 您自己的元数据，`kapt` 应配置 `spring-boot-configuration-processor` 依赖项。

50.6 测试

虽然可以使用 JUnit 4（由 `spring-boot-starter-test` 提供的默认值）来测试 Kotlin 代码，但建议使用 JUnit 5。JUnit 5 使测试类能够实例化一次并重用于所有类的测试。这使得可以在非静态方法上使用`@BeforeAll` 和`@AfterAll` 注释，这非常适合 Kotlin。

要使用 JUnit 5，请从 `spring-boot-starter-test` 中排除 `junit:junit` 依赖项，添加 JUnit 5 依赖项，并相应地配置 Maven 或 Gradle 插件。有关更多详细信息，请参阅 JUnit 5 文档。您还需要将 测试实例生命周期切换为“每个类”。

50.7 资源

50.7.1 进一步阅读

- Kotlin 语言参考
- Kotlin Slack（带有专用的#spring 频道）
- 具有 `spring` 和 `kotlin` 标记的 Stackoverflow
- 在浏览器中尝试 Kotlin
- 科特林博客
- 令人敬畏的 Kotlin
- 教程：使用 Spring Boot 和 Kotlin 构建 Web 应用程序
- 使用 Kotlin 开发 Spring Boot 应用程序
- 使用 Kotlin，Spring Boot 和 PostgreSQL 的地理空间信使
- 在 Spring Framework 5.0 中引入 Kotlin 支持

- Spring 框架 5 Kotlin API，功能方式

50.7.2 例子

- spring-boot-kotlin-demo：常规 Spring Boot + Spring Data JPA 项目
- mixit：Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB
- spring-kotlin-fullstack：WebFlux Kotlin fullstack 示例，其中 Kotlin2js 用于前端而不是 JavaScript 或 TypeScript
- spring-petclinic-kotlin：Spring PetClinic 样本应用程序的 Kotlin 版本
- spring-kotlin-deepdive：将 Boot 1.0 + Java 逐步迁移到 Boot 2.0 + Kotlin

51.接下来要阅读的内容

如果您想了解本节中讨论的任何类的更多信息，可以查看 Spring Boot API 文档，也可以直接浏览源代码。如果您有具体问题，请查看 操作方法部分。

如果您对 Spring Boot 的核心功能感到满意，可以继续阅读有关生产就绪功能的内容。

第五部分。Spring Boot Actuator：生产就绪功能

Spring Boot 包含许多其他功能，可帮助您在将应用程序推送到生产环境时监控和管理应用程序。您可以选择使用 HTTP 端点或 JMX 来管理和监视应用程序。审核，运行状况和指标收集也可以自动应用于您的应用程序。

52.启用生产就绪功能

该 `spring-boot-actuator` 模块提供了所有 Spring Boot 的生产就绪功能。启用这些功能的最简单方法是为 `spring-boot-starter-actuator`'Starter'添加依赖项。

执行器的定义

致动器是制造术语，指的是用于移动或控制某物的机械装置。执行器可以通过微小的变化产生大量的运动。

要将执行器添加到基于 Maven 的项目，请添加以下“Starter”依赖项：

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
</dependencies>
```

对于 Gradle，请使用以下声明：

```
dependencies {
        compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

53.终点

通过执行器端点，您可以监控应用程序并与之交互。Spring Boot 包含许多内置端点，允许您添加自己的端点。例如，`health` 端点提供基本的应用程序运行状况信息。

可以启用或禁用每个单独的端点。它控制是否在应用程序上下文中创建端点并且其 bean 存在。要远程访问，还必须通过 JMX 或 HTTP 公开端点 。大多数应用程序选择 HTTP，其中端点的 ID 以及`/actuator` 的前缀映射到 URL。例如，默认情况下，`health` 端点映射到`/actuator/health`。

可以使用以下与技术无关的端点：

ID	描述	默认情况下启用
`auditevents`	公开当前应用程序的审核事件信息。	Yes
`beans`	显示应用程序中所有 Spring beans 的完整列表。	Yes
`caches`	暴露可用的缓存。	Yes
`conditions`	显示在配置和自动配置类上评估的条件以及它们匹配或不匹配的原因。	Yes
`configprops`	显示所有`@ConfigurationProperties` 的整理列表。	Yes
`env`	公开 Spring `ConfigurableEnvironment` 的财产。	Yes
`flyway`	显示已应用的任何 Flyway 数据库迁移。	Yes
`health`	显示应用健康信息。	Yes
`httptrace`	显示 HTTP 跟踪信息（默认情况下，最后 100 个 HTTP 请求 - 响应交换）。	Yes
`info`	显示任意应用信息。	Yes
`integrationgraph`	显示 Spring Integration 图表。	Yes
`loggers`	显示和修改应用程序中记录器的配置。	Yes
`liquibase`	显示已应用的任何 Liquibase 数据库迁移。	Yes
`metrics`	显示当前应用程序的“指标”信息。	Yes
`mappings`	显示所有`@RequestMapping` 路径的整理列表。	Yes
`scheduledtasks`	显示应用程序中的计划任务。	Yes
`sessions`	允许从支持 Spring Session 的会话存储中检索和删除用户会话。使用 Spring Session 对响应式 Web 应用程序的支持时不可用。	Yes
`shutdown`	允许应用程序正常关闭。	No
`threaddump`	执行线程转储。	Yes

如果您的应用程序是 Web 应用程序（Spring MVC，Spring WebFlux 或 Jersey），则可以使用以下

附加端点：

ID	描述	默认情况下启用
`heapdump`	返回 `hprof` 堆转储文件。	Yes
`jolokia`	通过 HTTP 公开 JMX beans（当 Jolokia 在类路径上时，不适用于 WebFlux）。	Yes
`logfile`	返回日志文件的内容（如果已设置 `logging.file` 或 `logging.path` 属性）。支持使用 HTTP `Range` 标头来检索部分日志文件的内容。	Yes
`prometheus`	以可由 Prometheus 服务器抓取的格式公开指标。	Yes

要了解有关 Actuator 端点及其请求和响应格式的更多信息，请参阅单独的 API 文档（HTML 或 PDF）。

53.1 启用端点

默认情况下，启用除 `shutdown` 之外的所有端点。要配置端点的启用，请使用其 `management.endpoint.<id>.enabled` 属性。以下示例启用 `shutdown` 端点：

```
management.endpoint.shutdown.enabled=true
```

如果您希望端点启用是选择加入而不是选择退出，请将 `management.endpoints.enabled-by-default` 属性设置为 `false` 并使用单个端点 `enabled` 属性重新加入。以下示例启用 `info` endpoint 并禁用所有其他端点：

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

注意

已完全从应用程序上下文中删除已禁用的端点。如果您只想更改端点所暴露的技术，请改用 `include` 和 `exclude` 属性 。

53.2 公开端点

由于端点可能包含敏感信息，因此应仔细考虑何时公开它们。下表显示了内置端点的默认曝光：

ID	JMX	卷筒纸
`auditevents`	Yes	No
`beans`	Yes	No
`caches`	Yes	No
`conditions`	Yes	No
`configprops`	Yes	No

ID	JMX	卷筒纸
env	Yes	No
flyway	Yes	No
health	Yes	Yes
heapdump	N/A	No
httptrace	Yes	No
info	Yes	Yes
integrationgraph	Yes	No
jolokia	N/A	No
logfile	N/A	No
loggers	Yes	No
liquibase	Yes	No
metrics	Yes	No
mappings	Yes	No
prometheus	N/A	No
scheduledtasks	Yes	No
sessions	Yes	No
shutdown	Yes	No
threaddump	Yes	No

要更改公开的端点，请使用以下特定于技术的 `include` 和 `exclude` 属性：

Property	默认
management.endpoints.jmx.exposure.exclude	
management.endpoints.jmx.exposure.include	*
management.endpoints.web.exposure.exclude	
management.endpoints.web.exposure.include	info, health

`include` 属性列出了公开的端点的 ID。`exclude` 属性列出了不应公开的端点的 ID。`exclude`

属性优先于 include 属性。include 和 exclude 属性都可以配置端点 ID 列表。

例如，要停止通过 JMX 公开所有端点并仅显示 health 和 info 端点，请使用以下属性：

```
management.endpoints.jmx.exposure.include=health,info
```

*可用于选择所有端点。例如，要通过 HTTP 公开除 env 和 beans 端点之外的所有内容，请使用以下属性：

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans
```

注意

*在 YAML 中具有特殊含义，因此如果要包含（或排除）所有端点，请务必添加引号，如以下示例所示：

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

注意

如果您的申请是公开的，我们强烈建议您 保护您的终端。

小费

如果您想在暴露端点时实施自己的策略，可以注册 EndpointFilter bean。

53.3 保护 HTTP 端点

您应该像使用任何其他敏感 URL 一样注意保护 HTTP 端点。如果存在 Spring 安全性，则默认使用 Spring 安全性内容协商策略来保护端点。例如，如果您希望为 HTTP 端点配置自定义安全性，仅允许具有特定角色的用户访问它们，Spring Boot 提供了一些方便的 RequestMatcher 对象，可以与 Spring 安全性结合使用。

典型的 Spring 安全配置可能类似于以下示例：

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {

http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests()
                                    .anyRequest().hasRole("ENDPOINT_ADMIN")
                                    .and()
                        .httpBasic();
        }

}
```

上面的示例使用 EndpointRequest.toAnyEndpoint()将请求与任何端点进行匹配，然后确保所有端点都具有 ENDPOINT_ADMIN 角色。EndpointRequest 也提供了其他几种匹配方法。

有关详细信息，请参阅 API 文档（HTML 或 PDF）。

如果在防火墙后部署应用程序，您可能希望无需身份验证即可访问所有执行器端点。您可以通过更改 `management.endpoints.web.exposure.include` 属性来执行此操作，如下所示：

application.properties。

```
management.endpoints.web.exposure.include=*
```

此外，如果存在 Spring 安全性，则需要添加自定义安全性配置，以允许对端点进行未经身份验证的访问，如以下示例所示：

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {

http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests()
                        .anyRequest().permitAll();
        }

}
```

53.4 配置端点

端点自动缓存对不带任何参数的读取操作的响应。要配置端点缓存响应的时间量，请使用其 `cache.time-to-live` 属性。以下示例将 `beans` 端点缓存的生存时间设置为 10 秒：

application.properties。

```
management.endpoint.beans.cache.time-to-live=10s
```

注意

前缀 `management.endpoint.<name>`用于唯一标识正在配置的端点。

注意

在进行经过身份验证的 HTTP 请求时，`Principal` 被视为端点的输入，因此不会缓存响应。

53.5 用于执行器 Web 端点的超媒体

添加了“发现页面”，其中包含指向所有端点的链接。默认情况下，`/actuator` 上提供了“发现页面”。

配置自定义管理上下文路径后，“发现页面”会自动从`/actuator` 移动到管理上下文的根目录。例如，如果管理上下文路径为`/management`，则可以从`/management` 获取发现页面。当管理上下文路径设置为`/`时，将禁用发现页面以防止与其他映射冲突的可能性。

53.6 CORS 支持

跨源资源共享 （CORS）是一种 W3C 规范，允许您以灵活的方式指定授权的跨域请求类型。如

果您使用 Spring MVC 或 Spring WebFlux，可以配置 Actuator 的 Web 端点以支持此类方案。

默认情况下禁用 CORS 支持，仅在设置了 `management.endpoints.web.cors.allowed-origins` 属性后才启用 CORS 支持。以下配置允许来自 `example.com` 域的 `GET` 和 `POST` 来电：

```
management.endpoints.web.cors.allowed-origins=http://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

小费

有关 选项的完整列表，请参阅 CorsEndpointProperties。

53.7 实现自定义端点

如果添加注释为`@Endpoint` 的`@Bean`，则使用`@ReadOperation`，`@WriteOperation` 或`@DeleteOperation` 注释的任何方法都会通过 JMX 自动公开，并且在 Web 应用程序中也会通过 HTTP 自动公开。可以使用 Jersey，Spring MVC 或 Spring WebFlux 通过 HTTP 公开端点。

您还可以使用`@JmxEndpoint` 或`@WebEndpoint` 编写特定于技术的端点。这些端点仅限于各自的技术。例如，`@WebEndpoint` 仅通过 HTTP 而不是通过 JMX 公开。

您可以使用`@EndpointWebExtension` 和`@EndpointJmxExtension` 编写特定于技术的扩展。通过这些注释，您可以提供特定于技术的操作来扩充现有端点。

最后，如果您需要访问特定于 Web 框架的功能，您可以实现 Servlet 或 Spring `@Controller` 和`@RestController` 端点，但代价是它们无法通过 JMX 或使用不同的 Web 框架。

53.7.1 接收输入

端点上的操作通过其参数接收输入。通过 Web 公开时，这些参数的值取自 URL 的查询参数和 JSON 请求体。通过 JMX 公开时，参数将映射到 MBean 操作的参数。默认情况下需要参数。可以通过使用`@org.springframework.lang.Nullable` 注释它们来使它们成为可选项。

JSON 请求正文中的每个根属性都可以映射到端点的参数。请考虑以下 JSON 请求正文：

```
{
        "name": "test",
        "counter": 42
}
```

这可用于调用带有 `String name` 和 `int counter` 参数的写操作。

小费

由于端点与技术无关，因此只能在方法签名中指定简单类型。特别是不支持使用定义 `name` 和 `counter` 属性的自定义类型声明单个参数。

注意

要允许输入映射到操作方法的参数，实现端点的 Java 代码应使用`-parameters` 进行编译，实现端点的 Kotlin 代码应使用`-java-parameters` 进行编译。如果您使用的是 Spring Boot 的 Gradle 插件，或者您使用的是 Maven 和 `spring-boot-starter-parent`，则会自

动执行此操作。

输入类型转换

如有必要，传递给端点操作方法的参数将自动转换为所需类型。在调用操作方法之前，通过 JMX 或 HTTP 请求接收的输入将使用 `ApplicationConversionService` 的实例转换为所需类型。

53.7.2 自定义 Web 端点

`@Endpoint`，`@WebEndpoint` 或`@EndpointWebExtension` 上的操作将使用 Jersey，Spring MVC 或 Spring WebFlux 通过 HTTP 自动公开。

Web 端点请求谓词

为 Web 暴露的端点上的每个操作自动生成请求谓词。

路径

谓词的路径由端点的 ID 和 Web 暴露的端点的基本路径确定。默认基本路径为`/actuator`。例如，ID 为 `sessions` 的端点将使用`/actuator/sessions` 作为谓词中的路径。

可以通过使用`@Selector` 注释操作方法的一个或多个参数来进一步定制路径。这样的参数作为路径变量添加到路径谓词中。调用端点操作时，将变量的值传递给操作方法。

HTTP 方法

谓词的 HTTP 方法由操作类型决定，如下表所示：

手术	HTTP 方法
`@ReadOperation`	`GET`
`@WriteOperation`	`POST`
`@DeleteOperation`	`DELETE`

消费

对于使用请求主体的`@WriteOperation`（HTTP `POST`），谓词的 consumemes 子句为 `application/vnd.spring-boot.actuator.v2+json, application/json`。对于所有其他操作，consumemes 子句为空。

产生

谓词的 produce 子句可以由`@DeleteOperation`，`@ReadOperation` 和`@WriteOperation` 注释的 `produces` 属性确定。该属性是可选的。如果未使用，则自动确定 produce 子句。

如果操作方法返回 `void` 或 `Void`，则 produce 子句为空。如果操作方法返回 `org.springframework.core.io.Resource`，则 produce 子句为 `application/octet-stream`。对于所有其他操作，produce 子句是 `application/vnd.spring-boot.actuator.v2+json, application/json`。

Web 端点响应状态

端点操作的默认响应状态取决于操作类型（读取，写入或删除）以及操作返回的内容（如果

有）。

@ReadOperation 返回一个值，响应状态为 200（OK）。如果它未返回值，则响应状态将为 404（未找到）。

如果@WriteOperation 或@DeleteOperation 返回值，则响应状态将为 200（OK）。如果它没有返回值，则响应状态将为 204（无内容）。

如果在没有必需参数的情况下调用操作，或者使用无法转换为所需类型的参数，则不会调用操作方法，并且响应状态将为 400（错误请求）。

Web 端点范围请求

HTTP 范围请求可用于请求 HTTP 资源的一部分。使用 Spring MVC 或 Spring Web Flux 时，返回 org.springframework.core.io.Resource 的操作会自动支持范围请求。

注意

使用 Jersey 时不支持范围请求。

Web 端点安全

Web 端点或特定于 Web 的端点扩展上的操作可以接收当前 java.security.Principal 或 org.springframework.boot.actuate.endpoint.SecurityContext 作为方法参数。前者通常与@Nullable 结合使用，为经过身份验证和未经身份验证的用户提供不同的行为。后者通常用于使用 isUserInRole(String)方法执行授权检查。

53.7.3 Servlet 端点

通过实现一个注释为@ServletEndpoint 且同时实现 Supplier<EndpointServlet>的类，可以将 Servlet 公开为端点。Servlet 端点提供与 Servlet 容器更深层次的集成，但代价是可移植性。它们旨在用于将现有的 Servlet 作为端点公开。对于新端点，应尽可能优先选择 @Endpoint 和@WebEndpoint 注释。

53.7.4 控制器端点

@ControllerEndpoint 和@RestControllerEndpoint 可用于实现仅由 Spring MVC 或 Spring WebFlux 公开的端点。使用 Spring MVC 和 Spring WebFlux 的标准注释（例如 @RequestMapping 和@GetMapping）映射方法，并将端点的 ID 用作路径的前缀。控制器端点提供与 Spring Web 框架的更深层次集成，但代价是可移植性。应尽可能优先考虑@Endpoint 和 @WebEndpoint 注释。

53.8 健康信息

您可以使用运行状况信息来检查正在运行的应用程序的状态。监视软件经常使用它来在生产系统出现故障时向某人发出警报。health 端点公开的信息取决于 management.endpoint.health.show-details 属性，该属性可以使用以下值之一进行配置：

名称	描述
never	细节永远不会显示。

名称	描述
`when-authorized`	详细信息仅向授权用户显示。可以使用 `management.endpoint.health.roles` 配置授权角色。
`always`	详细信息显示给所有用户。

默认值为 `never`。当用户处于一个或多个端点的角色时，将被视为已获得授权。如果端点没有配置角色（默认值），则认为所有经过身份验证的用户都已获得授权。可以使用 `management.endpoint.health.roles` 属性配置角色。

注意

如果您已保护应用程序并希望使用 `always`，则您的安全配置必须允许对经过身份验证和未经身份验证的用户访问运行状况终结点。

健康信息是从 a 的内容中收集的 （默认情况下，`ApplicationContext` 中定义的所有 实例。Spring Boot 包括一些自动配置的 `HealthIndicators`，您也可以自己编写。默认情况下，最终系统状态由 `HealthAggregator` 导出，它根据状态的有序列表对每个 `HealthIndicator` 的状态进行排序。排序列表中的第一个状态用作整体健康状态。如果没有 `HealthIndicator` 返回 `HealthAggregator` 已知的状态，使用 `UNKNOWN` 状态。 _ `HealthIndicatorRegistryHealthIndicator`

小费

`HealthIndicatorRegistry` 可用于在运行时注册和取消注册健康指示器。

53.8.1 自动配置的 HealthIndicators

适当时，Spring Boot 会自动配置以下 `HealthIndicators`：

名称	描述
`CassandraHealthIndicator`	检查 Cassandra 数据库是否已启动。
`CouchbaseHealthIndicator`	检查 Couchbase 群集是否已启动。
`DiskSpaceHealthIndicator`	检查磁盘空间不足。
`DataSourceHealthIndicator`	检查是否可以获得与 `DataSource` 的连接。
`ElasticsearchHealthIndicator`	检查 Elasticsearch 集群是否已启动。
`InfluxDbHealthIndicator`	检查 InfluxDB 服务器是否已启动。
`JmsHealthIndicator`	检查 JMS 代理是否已启动。
`MailHealthIndicator`	检查邮件服务器是否已启动。

名称	描述
MongoHealthIndicator	检查 Mongo 数据库是否已启动。
Neo4jHealthIndicator	检查 Neo4j 服务器是否已启动。
RabbitHealthIndicator	检查 Rabbit 服务器是否已启动。
RedisHealthIndicator	检查 Redis 服务器是否已启动。
SolrHealthIndicator	检查 Solr 服务器是否已启动。

小费

您可以通过设置 `management.health.defaults.enabled` 属性来禁用它们。

53.8.2 编写自定义 HealthIndicators

要提供自定义健康信息，您可以注册实现该 `HealthIndicator` 界面的 Spring beans 。您需要提供 `health()`方法的实现并返回 `Health` 响应。`Health` 响应应包含状态，并可选择包含要显示的其他详细信息。以下代码显示了一个示例 `HealthIndicator` 实现：

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

        @Override
        public Health health() {
                int errorCode = check(); // perform some specific health check
                if (errorCode != 0) {
                        return Health.down().withDetail("Error Code",
errorCode).build();
                }
                return Health.up().build();
        }

}
```

注意

给定 `HealthIndicator` 的标识符是没有 `HealthIndicator` 后缀的 bean 的名称（如果存在）。在前面的示例中，健康信息在名为 `my` 的条目中可用。

除了 Spring Boot 的预定义 `Status` 类型之外，`Health` 还可以返回表示新系统状态的自定义 `Status`。在这种情况下，`HealthAggregator` 还需要提供接口的自定义实现 ，或者必须使用 `management.health.status.order` 配置属性配置默认实现。

例如，假设在 `HealthIndicator` 实现之一中使用了代码为 `FATAL` 的新 `Status`。要配置严重性顺序，请将以下属性添加到应用程序属性：

```
management.health.status.order=FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

响应中的 HTTP 状态代码反映了整体运行状况（例如，UP 映射到 200，而 OUT_OF_SERVICE 和 DOWN 映射到 503）。如果通过 HTTP 访问运行状况端点，则可能还需要注册自定义状态映射。例如，以下属性将 FATAL 映射到 503（服务不可用）：

```
management.health.status.http-mapping.FATAL=503
```

小费

如果您需要更多控制权，可以定义自己的 `HealthStatusHttpMapper` bean。

下表显示了内置状态的默认状态映射：

状态	制图
DOWN	SERVICE_UNAVAILABLE (503)
OUT_OF_SERVICE	SERVICE_UNAVAILABLE (503)
UP	No mapping by default, so http status is 200
UNKNOWN	No mapping by default, so http status is 200

53.8.3 反应性健康指标

对于反应性应用程序，例如那些使用 Spring WebFlux 的应用程序，`ReactiveHealthIndicator` 提供了一个非阻塞的合同来获取应用程序运行状况。与传统的 `HealthIndicator` 类似，健康信息是从 a 的内容中收集的 （默认情况下，在 `ApplicationContext` 中定义的所有 和 实例。不检查反应 API 的常规 `HealthIndicator` 是在弹性上执行的调度。 `ReactiveHealthIndicatorRegistryHealthIndicator` `ReactiveHealthIndicator`

小费

在响应式应用程序中，`ReactiveHealthIndicatorRegistry` 可用于在运行时注册和取消注册运行状况指示器。

要从反应式 API 提供自定义运行状况信息，您可以注册实现该 `ReactiveHealthIndicator` 接口的 Spring beans 。以下代码显示了一个示例 `ReactiveHealthIndicator` 实现：

```
@Component
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {

        @Override
        public Mono<Health> health() {
                return doHealthCheck() //perform some specific health check that
returns a Mono<Health>
                        .onErrorResume(ex -> Mono.just(new
Health.Builder().down(ex).build()));
        }

}
```

小费

要自动处理错误，请考虑从 `AbstractReactiveHealthIndicator` 扩展。

53.8.4 自动配置的 ReactiveHealthIndicators

适当时，Spring Boot 会自动配置以下 `ReactiveHealthIndicators`：

名称	描述
`CassandraReactiveHealthIndicator`	检查 Cassandra 数据库是否已启动。
`CouchbaseReactiveHealthIndicator`	检查 Couchbase 群集是否已启动。
`MongoReactiveHealthIndicator`	检查 Mongo 数据库是否已启动。
`RedisReactiveHealthIndicator`	检查 Redis 服务器是否已启动。

小费

必要时，反应指标取代常规指标。此外，任何未明确处理的 `HealthIndicator` 都会自动换行。

53.9 申请信息

应用程序信息公开了从 `InfoContributor`ApplicationContext 中定义的所有 beans 收集的各种信息 。Spring Boot 包含一些自动配置的 `InfoContributor` beans，您可以自己编写。

53.9.1 自动配置的 InfoContributors

在适当的情况下，Spring Boot 会自动配置以下 `InfoContributor` beans：

名称	描述
`EnvironmentInfoContributor`	在 `info` 键下显示 `Environment` 中的任意键。
`GitInfoContributor`	如果 `git.properties` 文件可用，则公开 git 信息。
`BuildInfoContributor`	如果 `META-INF/build-info.properties` 文件可用，则公开构建信息。

小费

可以通过设置 `management.info.defaults.enabled` 属性来禁用它们。

53.9.2 自定义应用程序信息

您可以通过设置 `info.*` Spring 属性来自定义 `info` 端点公开的数据。`info` 键下的所有 `Environment` 属性都会自动显示。例如，您可以将以下设置添加到 `application.properties` 文件中：

```
info.app.encoding=UTF-8
```

```
info.app.java.source=1.8
info.app.java.target=1.8
```

小费

您可以在构建时扩展信息属性，而不是对这些值进行硬编码 。

假设您使用 Maven，您可以按如下方式重写前面的示例：

```
info.app.encoding=@project.build.sourceEncoding@
info.app.java.source=@java.version@
info.app.java.target=@java.version@
```

53.9.3 Git 提交信息

`info` 端点的另一个有用功能是它能够在构建项目时发布有关 `git` 源代码存储库状态的信息。如果 `GitProperties` bean 可用，则会公开 `git.branch`，`git.commit.id` 和 `git.commit.time` 属性。

小费

如果类路径的根目录中有 `git.properties` 文件，则自动配置 `GitProperties` bean。有关更多详细信息，请参阅“ 生成 git 信息 ”。

如果要显示完整的 git 信息（即 `git.properties` 的完整内容），请使用 `management.info.git.mode` 属性，如下所示：

```
management.info.git.mode=full
```

53.9.4 构建信息

如果 `BuildProperties` bean 可用，`info` 端点也可以发布有关您的构建的信息。如果类路径中有 `META-INF/build-info.properties` 文件，则会发生这种情况。

小费

Maven 和 Gradle 插件都可以生成该文件。有关更多详细信息，请参阅“ 生成构建信息 ”。

53.9.5 编写自定义 InfoContributors

要提供自定义应用程序信息，您可以注册实现该 `InfoContributor` 接口的 Spring beans 。

以下示例使用单个值提供 `example` 条目：

```
import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

        @Override
        public void contribute(Info.Builder builder) {
```

```
                builder.withDetail("example",
                                Collections.singletonMap("key", "value"));
        }

}
```

如果到达 `info` 端点，您应该看到包含以下附加条目的响应：

```
{
        "example": {
                "key" : "value"
        }
}
```

54.通过 HTTP 进行监控和管理

如果您正在开发 Web 应用程序，Spring Boot Actuator 会自动配置所有已启用的端点以通过 HTTP 公开。默认约定是使用端点 `id` 作为 URL 路径，前缀为`/actuator`。例如，`health` 暴露为`/actuator/health`。提示：Actuator 本身支持 Spring MVC，Spring WebFlux 和 Jersey。

54.1 自定义管理端点路径

有时，自定义管理端点的前缀很有用。例如，您的应用程序可能已将`/actuator` 用于其他目的。您可以使用 `management.endpoints.web.base-path` 属性更改管理端点的前缀，如以下示例所示：

```
management.endpoints.web.base-path=/manage
```

前面的 `application.properties` 示例将端点从`/actuator/{id}`更改为`/manage/{id}`（例如，`/manage/info`）。

注意

除非已将管理端口配置为使用其他 HTTP 端口公开端点，否则 `management.endpoints.web.base-path` 相对于 `server.servlet.context-path`。如果配置了 `management.server.port`，则 `management.endpoints.web.base-path` 相对于 `management.server.servlet.context-path`。

如果要将端点映射到其他路径，可以使用 `management.endpoints.web.path-mapping` 属性。

以下示例将`/actuator/health` 重新映射为`/healthcheck`：

application.properties。

```
management.endpoints.web.base-path=/
management.endpoints.web.path-mapping.health=healthcheck
```

54.2 自定义 Management Server 端口

使用默认 HTTP 端口公开管理端点是基于云的部署的明智选择。但是，如果您的应用程序在您自己的数据中心内运行，您可能更喜欢使用不同的 HTTP 端口公开端点。

您可以设置 management.server.port 属性以更改 HTTP 端口，如以下示例所示：

```
management.server.port=8081
```

54.3 配置管理特定的 SSL

配置为使用自定义端口时，还可以使用各种 management.server.ssl.*属性为管理服务器配置自己的 SSL。例如，这样做可以让主应用程序使用 HTTPS 时管理服务器通过 HTTP 可用，如以下属性设置所示：

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:store.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=false
```

或者，主服务器和管理服务器都可以使用 SSL 但具有不同的密钥库，如下所示：

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:main.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=true
management.server.ssl.key-store=classpath:management.jks
management.server.ssl.key-password=secret
```

54.4 自定义管理服务器地址

您可以通过设置 management.server.address 属性来自定义管理端点可用的地址。如果您只想在内部或面向运行的网络上侦听或仅侦听来自 localhost 的连接，这样做非常有用。

注意

仅当端口与主服务器端口不同时，才能侦听不同的地址。

以下示例 application.properties 不允许远程管理连接：

```
management.server.port=8081
management.server.address=127.0.0.1
```

54.5 禁用 HTTP 端点

如果您不想通过 HTTP 公开端点，可以将管理端口设置为-1，如以下示例所示：

```
management.server.port=-1
```

这也可以使用 management.endpoints.web.exposure.exclude 属性来实现，如以下示例所示：

```
management.endpoints.web.exposure.exclude=*
```

55.对 JMX 的监测和管理

Java Management Extensions（JMX）提供了一种监视和管理应用程序的标准机制。默认情况

下，Spring Boot 将管理端点公开为 `org.springframework.boot` 域下的 JMX MBean。

55.1 自定义 MBean 名称

MBean 的名称通常是从端点的 `id` 生成的。例如，`health` 端点公开为 `org.springframework.boot:type=Endpoint,name=Health`。

如果您的应用程序包含多个 Spring `ApplicationContext`，您可能会发现名称发生冲突。要解决此问题，可以将 `spring.jmx.unique-names` 属性设置为 `true`，以便 MBean 名称始终是唯一的。

您还可以自定义公开端点的 JMX 域。以下设置显示了在 `application.properties` 中执行此操作的示例：

```
spring.jmx.unique-names=true
management.endpoints.jmx.domain=com.example.myapp
```

55.2 禁用 JMX 端点

如果您不想通过 JMX 公开端点，可以将 `management.endpoints.jmx.exposure.exclude` 属性设置为`*`，如以下示例所示：

```
management.endpoints.jmx.exposure.exclude=*
```

55.3 通过 HTTP 使用 Jolokia for JMX

Jolokia 是一个 JMX-HTTP 桥，它提供了一种访问 JMX beans 的替代方法。要使用 Jolokia，请在 `org.jolokia:jolokia-core` 中包含依赖项。例如，使用 Maven，您将添加以下依赖项：

```
<dependency>
        <groupId>org.jolokia</groupId>
        <artifactId>jolokia-core</artifactId>
</dependency>
```

然后，可以通过向 `management.endpoints.web.exposure.include` 属性添加 `jolokia` 或`*`来公开 Jolokia 端点。然后，您可以在管理 HTTP 服务器上使用`/actuator/jolokia` 来访问它。

55.3.1 自定义 Jolokia

Jolokia 有许多设置，您可以通过设置 servlet 参数来进行传统配置。使用 Spring Boot，您可以使用 `application.properties` 文件。为此，请在参数前加 `management.endpoint.jolokia.config.`，如以下示例所示：

```
management.endpoint.jolokia.config.debug=true
```

55.3.2 禁用 Jolokia

如果您使用 Jolokia 但不希望 Spring Boot 配置它，请将 `management.endpoint.jolokia.enabled` 属性设置为 `false`，如下所示：

```
management.endpoint.jolokia.enabled=false
```

56.记录器

Spring Boot Actuator 包括在运行时查看和配置应用程序日志级别的功能。您可以查看整个列表或单个记录器的配置，该配置由显式配置的日志记录级别以及日志记录框架为其提供的有效日志记录级别组成。这些级别可以是以下之一：

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `OFF`
- `null`

`null` 表示没有明确的配置。

56.1 配置记录器

要配置给定的记录器，`POST` 是资源 URI 的部分实体，如以下示例所示：

```
{
        "configuredLevel": "DEBUG"
}
```

小费

要“重置”记录器的特定级别（并使用默认配置），您可以传递 `null` 的值 `configuredLevel`。

57.度量标准

Spring Boot Actuator 为 Micrometer 提供依赖关系管理和自动配置， Micrometer 是一个支持众多监控系统的应用程序指标外观，包括：

- AppOptics
- Atlas
- Datadog
- Dynatrace
- Elastic
- Ganglia
- Graphite
- Humio
- Influx
- JMX
- KairosDB
- New Relic
- Prometheus
- SignalFx
- 简单（内存中）
- StatsD
- Wavefront

小费

要了解有关 Micrometer 功能的更多信息，请参阅其 参考文档，特别是概念部分。

57.1 入门

Spring Boot 自动配置组合 `MeterRegistry`，并为组合路径中找到的每个受支持的实现添加一个注册表。在运行时类路径中依赖 `micrometer-registry-{system}`足以使 Spring Boot 配置注册表。

大多数注册管理机构都有共同点 例如，即使 Micrometer 注册表实现位于类路径上，您也可以禁用特定的注册表。例如，要禁用 Datadog：

```
management.metrics.export.datadog.enabled=false
```

Spring Boot 还会将任何自动配置的注册表添加到 `Metrics` 类的全局静态复合注册表中，除非您明确告诉它不要：

```
management.metrics.use-global-registry=false
```

在注册表中注册任何仪表之前，您可以注册任意数量的 `MeterRegistryCustomizer` beans 以进一步配置注册表，例如应用通用标签：

```
@Bean
MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
        return registry -> registry.config().commonTags("region", "us-east-1");
}
```

您可以通过更具体地说明泛型类型，将自定义应用于特定的注册表实现：

```
@Bean
MeterRegistryCustomizer<GraphiteMeterRegistry> graphiteMetricsNamingConvention() {
        return registry -> registry.config().namingConvention(MY_CUSTOM_CONVENTION);
}
```

使用该设置，您可以在组件中注入 `MeterRegistry` 并注册指标：

```
@Component
public class SampleBean {

        private final Counter counter;

        public SampleBean(MeterRegistry registry) {
                this.counter = registry.counter("received.messages");
        }

        public void handleMessage(String message) {
                this.counter.increment();
                // handle message implementation
        }

}
```

Spring Boot 还配置了 可以通过配置或专用注释标记控制的内置检测（即 `MeterBinder` 实现）。

57.2 支持的监控系统

57.2.1 AppOptics

默认情况下，AppOptics 注册表会定期将指标推送到 api.appoptics.com/v1/measurements。要将指标导出到 SaaS AppOptics，必须提供您的 API 令牌：

```
management.metrics.export.appoptics.api-token=YOUR_TOKEN
```

57.2.2 Atlas

默认情况下，度量标准导出到 本地计算机上运行的 Atlas。可以使用以下方式提供要使用的 Atlas 服务器的位置 ：

```
management.metrics.export.atlas.uri=http://atlas.example.com:7101/api/v1/publish
```

57.2.3 Datadog

Datadog 注册表定期将指标推送到 datadoghq。要将指标导出到 Datadog，必须提供您的 API 密钥：

```
management.metrics.export.datadog.api-key=YOUR_KEY
```

您还可以更改度量标准发送到 Datadog 的时间间隔：

```
management.metrics.export.datadog.step=30s
```

57.2.4 Dynatrace

Dynatrace 注册表定期将指标推送到配置的 URI。要将指标导出到 Dynatrace，必须提供您的 API 令牌，设备 ID 和 URI：

```
management.metrics.export.dynatrace.api-token=YOUR_TOKEN
management.metrics.export.dynatrace.device-id=YOUR_DEVICE_ID
management.metrics.export.dynatrace.uri=YOUR_URI
```

您还可以更改指标发送到 Dynatrace 的时间间隔：

```
management.metrics.export.dynatrace.step=30s
```

57.2.5 Elastic

默认情况下，指标会导出到 本地计算机上运行的 Elastic。可以使用以下属性提供要使用的 Elastic 服务器的位置：

```
management.metrics.export.elastic.host=http://elastic.example.com:8086
```

57.2.6 Ganglia

默认情况下，度量标准将导出到 本地计算机上运行的 Ganglia。可以使用以下命令提供要使用的 Ganglia 服务器主机和端口：

```
management.metrics.export.ganglia.host=ganglia.example.com
management.metrics.export.ganglia.port=9649
```

57.2.7 Graphite

默认情况下，度量标准将导出到 本地计算机上运行的 Graphite。可以使用以下命令提供要使用的 Graphite 服务器主机和端口：

```
management.metrics.export.graphite.host=graphite.example.com
management.metrics.export.graphite.port=9004
```

千分尺提供默认值 `HierarchicalNameMapper`，用于控制尺寸计 id 如何映射到平面分层名称。

小费

要控制此行为，请定义 `GraphiteMeterRegistry` 并提供您自己的 `HierarchicalNameMapper`。除非您自己定义，否则会提供自动配置的 `GraphiteConfig` 和 `Clock` beans：

```
@Bean
public GraphiteMeterRegistry graphiteMeterRegistry(GraphiteConfig config, Clock
clock) {
        return new GraphiteMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

57.2.8 Humio

默认情况下，Humio 注册表会定期将指标推送到 cloud.humio.com。要将指标导出到 SaaS Humio，必须提供您的 API 令牌：

```
management.metrics.export.humio.api-token=YOUR_TOKEN
```

您还应配置一个或多个标记，以标识要推送指标的数据源：

```
management.metrics.export.humio.tags.alpha=a
management.metrics.export.humio.tags.bravo=b
```

57.2.9 Influx

默认情况下，度量标准将导出到 本地计算机上运行的 Influx。可以使用以下方式提供要使用的 Influx 服务器的位置：

```
management.metrics.export.influx.uri=http://influx.example.com:8086
```

57.2.10 JMX

Micrometer 提供了与 JMX 的分层映射 ，主要是作为在本地查看指标的便宜且可移植的方式。默认情况下，度量标准将导出到 `metrics` JMX 域。可以使用以下方式提供要使用的域：

```
management.metrics.export.jmx.domain=com.example.app.metrics
```

千分尺提供默认值 `HierarchicalNameMapper`，用于控制尺寸计 id 如何映射到平面分层名称。

小费

要控制此行为，请定义 `JmxMeterRegistry` 并提供您自己的 `HierarchicalNameMapper`。除非您自己定义，否则会提供自动配置

的 `JmxConfig` 和 `Clock` beans：

```
@Bean
public JmxMeterRegistry jmxMeterRegistry(JmxConfig config, Clock clock) {
        return new JmxMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

57.2.11 KairosDB

默认情况下，度量标准将导出到 本地计算机上运行的 KairosDB。可以使用以下方式提供要使用的 KairosDB 服务器的位置：

```
management.metrics.export.kairos.uri=http://kairosdb.example.com:8080/api/v1/datapoints
```

57.2.12 New Relic

New Relic 注册表会定期将指标推送到 New Relic。要将指标导出到 New Relic，必须提供您的 API 密钥和帐户 ID：

```
management.metrics.export.newrelic.api-key=YOUR_KEY
management.metrics.export.newrelic.account-id=YOUR_ACCOUNT_ID
```

您还可以更改指标发送到 New Relic 的时间间隔：

```
management.metrics.export.newrelic.step=30s
```

57.2.13 Prometheus

Prometheus 期望抓取或轮询各个应用实例以获取指标。Spring Boot 提供了`/actuator/prometheus` 处可用的执行器端点，以提供具有适当格式的 Prometheus 刮擦。

小费

默认情况下端点不可用，必须公开，请参阅 公开端点以获取更多详细信息。

以下是添加到 `prometheus.yml` 的示例 `scrape_config`：

```
scrape_configs:
  - job_name: 'spring'
        metrics_path: '/actuator/prometheus'
        static_configs:
          - targets: ['HOST:PORT']
```

57.2.14 SignalFx

SignalFx 注册表 定期将指标推送到 SignalFx。要将指标导出到 SignalFx，必须提供您的访问令牌：

```
management.metrics.export.signalfx.access-token=YOUR_ACCESS_TOKEN
```

您还可以更改指标发送到 SignalFx 的时间间隔：

```
management.metrics.export.signalfx.step=30s
```

57.2.15 简单

Micrometer 附带一个简单的内存后端，如果没有配置其他注册表，它将自动用作后备。这使您可以查看度量标准终结点中收集的度量标准。

只要您使用任何其他可用后端，内存后端就会自动禁用。您也可以显式禁用它：

```
management.metrics.export.simple.enabled=false
```

57.2.16 StatsD

StatsD 注册表急需将 UDP 上的指标推送到 StatsD 代理。默认情况下，度量标准将导出到本地计算机上运行的 StatsD 代理程序。可以使用以下方式提供要使用的 StatsD 代理主机和端口：

```
management.metrics.export.statsd.host=statsd.example.com
management.metrics.export.statsd.port=9125
```

您还可以更改要使用的 StatsD 行协议（默认为 Datadog）：

```
management.metrics.export.statsd.flavor=etsy
```

57.2.17 Wavefront

Wavefront 注册表会定期将指标推送到 Wavefront。如果您要将指标直接导出到 Wavefront，则必须提供您的 API 令牌：

```
management.metrics.export.wavefront.api-token=YOUR_API_TOKEN
```

或者，您可以在您的环境中使用 Wavefront 边车或内部代理设置，将指标数据转发到 Wavefront API 主机：

```
management.metrics.export.wavefront.uri=proxy://localhost:2878
```

小费

如果将度量标准发布到 Wavefront 代理（如文档中 所述），则主机必须采用 `proxy://HOST:PORT` 格式。

您还可以更改指标发送到 Wavefront 的时间间隔：

```
management.metrics.export.wavefront.step=30s
```

57.3 支持的度量标准

Spring Boot 在适用时注册以下核心指标：

- JVM 指标，报告利用率：
  - 各种内存和缓冲池
  - 与垃圾收集有关的统计
  - 线程利用率
  - 加载/卸载的类数
- CPU 指标
- 文件描述符指标
- 卡夫卡消费者指标
- Log4j2 指标：记录每个级别记录到 Log4j2 的事件数

- Logback 指标：记录每个级别记录到 Logback 的事件数
- 正常运行时间指标：报告正常运行时间表和表示应用程序绝对启动时间的固定计量表
- Tomcat 指标
- Spring Integration 指标

57.3.1 Spring MVC 指标

自动配置可以对 Spring MVC 处理的请求进行检测。当 `management.metrics.web.server.auto-time-requests` 为 `true` 时，将对所有请求进行此检测。或者，当设置为 `false` 时，您可以通过将`@Timed` 添加到请求处理方法来启用检测：

```
@RestController
@Timed
public class MyController {

        @GetMapping("/api/people")
        @Timed(extraTags = { "region", "us-east-1" })
        @Timed(value = "all.people", longTask = true)
        public List<Person> listPeople() { ... }

}
```

一个控制器类，用于在控制器中的每个请求处理程序上启用计时。

一种启用单个端点的方法。如果您在类上拥有它，则不需要这样做，但可以用于进一步自定义此特定端点的计时器。

使用 `longTask = true` 的方法为该方法启用长任务计时器。长任务计时器需要单独的度量标准名称，并且可以使用短任务计时器进行堆叠。

默认情况下，使用名称 `http.server.requests` 生成指标。可以通过设置 `management.metrics.web.server.requests-metric-name` 属性来自定义名称。

默认情况下，Spring 与 MVC 相关的指标标记有以下信息：

标签	描述
`exception`	处理请求时抛出的任何异常的简单类名。
`method`	请求的方法（例如，`GET` 或 `POST`）
`outcome`	根据响应的状态代码请求结果。1xx 是 `INFORMATIONAL`，2xx 是 `SUCCESS`，3xx 是 `REDIRECTION`，4xx `CLIENT_ERROR`，5xx 是 `SERVER_ERROR`
`status`	响应的 HTTP 状态代码（例如，`200` 或 `500`）
`uri`	如果可能，在变量替换之前请求 URI 模板（例如，`/api/person/{id}`）

要自定义标记，请提供实现 `WebMvcTagsProvider` 的`@Bean`。

57.3.2 Spring WebFlux 度量标准

自动配置支持 WebFlux 控制器和功能处理程序处理的所有请求的检测。

默认情况下，会生成名称为 `http.server.requests` 的指标。您可以通过设置 `management.metrics.web.server.requests-metric-name` 属性来自定义名称。

默认情况下，与 WebFlux 相关的指标标记有以下信息：

标签	描述
`exception`	处理请求时抛出的任何异常的简单类名。
`method`	请求的方法（例如，`GET` 或 `POST`）
`outcome`	根据响应的状态代码请求结果。1xx 是 `INFORMATIONAL`，2xx 是 `SUCCESS`，3xx 是 `REDIRECTION`，4xx `CLIENT_ERROR`，5xx 是 `SERVER_ERROR`
`status`	响应的 HTTP 状态代码（例如，`200` 或 `500`）
`uri`	如果可能，在变量替换之前请求 URI 模板（例如，`/api/person/{id}`）

要自定义标记，请提供实现 `WebFluxTagsProvider` 的`@Bean`。

57.3.3 Jersey 服务器度量标准

自动配置支持对 Jersey JAX-RS 实现处理的请求进行检测。当 `management.metrics.web.server.auto-time-requests` 为 `true` 时，此检测将针对所有请求进行。或者，当设置为 `false` 时，您可以通过将`@Timed` 添加到请求处理方法来启用检测：

```
@Component
@Path("/api/people")
@Timed
public class Endpoint {
    @GET
    @Timed(extraTags = { "region", "us-east-1" })
    @Timed(value = "all.people", longTask = true)
    public List<Person> listPeople() { ... }
}
```

在资源类上，为资源中的每个请求处理程序启用计时。

在启用单个端点的方法上。如果您在类上拥有它，则不需要这样做，但可以用于进一步自定义此特定端点的计时器。

在使用 `longTask = true` 的方法上为该方法启用长任务计时器。长任务计时器需要单独的度量标准名称，并且可以使用短任务计时器进行堆叠。

默认情况下，使用名称 `http.server.requests` 生成度量标准。可以通过设置

management.metrics.web.server.requests-metric-name 属性来自定义名称。

默认情况下，Jersey 服务器指标标记有以下信息：

标签	描述
exception	处理请求时抛出的任何异常的简单类名。
method	请求的方法（例如，GET 或 POST）
outcome	根据响应的状态代码请求结果。1xx 是 INFORMATIONAL，2xx 是 SUCCESS，3xx 是 REDIRECTION，4xx CLIENT_ERROR，5xx 是 SERVER_ERROR
status	响应的 HTTP 状态代码（例如，200 或 500）
uri	如果可能，在变量替换之前请求 URI 模板（例如，/api/person/{id}）

要自定义标记，请提供实现 JerseyTagsProvider 的@Bean。

57.3.4 HTTP 客户端度量标准

Spring Boot Actuator 管理 RestTemplate 和 WebClient 的工具。为此，您必须注入一个自动配置的构建器并使用它来创建实例：

- RestTemplateBuilder RestTemplate
- WebClient.Builder WebClient

也可以手动应用负责此仪器的定制器，即 MetricsRestTemplateCustomizer 和 MetricsWebClientCustomizer。

默认情况下，使用名称 http.client.requests 生成指标。可以通过设置 management.metrics.web.client.requests-metric-name 属性来自定义名称。

默认情况下，已检测客户端生成的度量标准使用以下信息进行标记：

- method，请求的方法（例如，GET 或 POST）。
- uri，变量替换之前的请求 URI 模板（如果可能）（例如，/api/person/{id}）。
- status，响应的 HTTP 状态代码（例如，200 或 500）。
- clientName，URI 的主机部分。

要自定义标记，并根据您选择的客户端，您可以提供@Bean 来实现 RestTemplateExchangeTagsProvider 或 WebClientExchangeTagsProvider。RestTemplateExchangeTags 和 WebClientExchangeTags 中有便利的静态函数。

57.3.5 缓存度量标准

自动配置允许在启动时使用前缀为 cache 的度量标准检测所有可用的 Cache。缓存检测针对一组基本指标进行了标准化。此外，还提供了特定于缓存的指标。

支持以下缓存库：

- Caffeine

- EhCache 2
- Hazelcast
- 任何兼容的 JCache（JSR-107）实现

度量标准由缓存的名称和从 bean 名称派生的 `CacheManager` 的名称标记。

注意

只有启动时可用的缓存才会绑定到注册表。对于在启动阶段之后即时或以编程方式创建的缓存，需要显式注册。`CacheMetricsRegistrar` bean 可用于简化此过程。

57.3.6 数据源度量标准

自动配置使用名为 `jdbc` 的度量标准启用所有可用 `DataSource` 对象的检测。数据源检测会生成表示池中当前活动，最大允许和最小允许连接的计量器。这些仪表中的每一个都有一个以 `jdbc` 为前缀的名称。

度量标准也由基于 bean 名称计算的 `DataSource` 的名称标记。

小费

默认情况下，Spring Boot 为所有支持的数据源提供元数据; 如果您不喜欢自己喜欢的数据源，则可以添加额外的 `DataSourcePoolMetadataProvider` beans。有关示例，请参阅 `DataSourcePoolMetadataProvidersConfiguration`。

此外，Hikari 特定的指标以 `hikaricp` 前缀公开。每个度量标准都由池名称标记（可以使用 `spring.datasource.name` 控制）。

57.3.7 Hibernate 度量标准

自动配置允许使用名为 `hibernate` 的度量标准启用统计信息的所有可用 Hibernate `EntityManagerFactory` 实例的检测。

度量标准也由 bean 名称派生的 `EntityManagerFactory` 名称标记。

要启用统计信息，标准 JPA 属性 `hibernate.generate_statistics` 必须设置为 `true`。您可以在自动配置的 `EntityManagerFactory` 上启用它，如以下示例所示：

```
spring.jpa.properties.hibernate.generate_statistics=true
```

57.3.8 RabbitMQ 指标

自动配置将使用名为 `rabbitmq` 的度量标准启用所有可用 RabbitMQ 连接工厂的检测。

57.4 注册自定义指标

要注册自定义指标，请将 `MeterRegistry` 注入组件，如以下示例所示：

```
class Dictionary {

        private final List<String> words = new CopyOnWriteArrayList<>();
```

```
        Dictionary(MeterRegistry registry) {
                registry.gaugeCollectionSize("dictionary.size", Tags.empty(),
this.words);
        }

        // …

}
```

如果您发现跨组件或应用程序重复检测一套度量标准，则可以将此套件封装在 `MeterBinder` 实现中。默认情况下，所有 `MeterBinder` beans 的指标都会自动绑定到 Spring - 托管 `MeterRegistry`。

57.5 自定义各个指标

如果需要将自定义应用于特定的 `Meter` 实例，可以使用 `io.micrometer.core.instrument.config.MeterFilter` 接口。默认情况下，所有 `MeterFilter` beans 都将自动应用于千分尺 `MeterRegistry.Config`。

例如，如果要将 `mytag.region` 标记重命名为 `mytag.area` 以查找以 `com.example` 开头的所有仪表 ID，您可以执行以下操作：

```
@Bean
public MeterFilter renameRegionTagMeterFilter() {
        return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");
}
```

57.5.1 通用标签

通用标签通常用于操作环境中的维度向下钻取，如主机，实例，区域，堆栈等。共用标签应用于所有仪表，并且可以按以下示例所示进行配置：

```
management.metrics.tags.region=us-east-1
management.metrics.tags.stack=prod
```

上面的示例将 `region` 和 `stack` 标记添加到所有计量表，其值分别为 `us-east-1` 和 `prod`。

注意

如果您使用 Graphite，则常用标记的顺序很重要。由于使用此方法无法保证常用标记的顺序，因此建议 Graphite 用户定义自定义 `MeterFilter`。

57.5.2 Per-meter 属性

除了 `MeterFilter` beans 之外，还可以使用属性在每米的基础上应用一组有限的自定义。Per-meter 自定义适用于以给定名称开头的所有仪表 ID。例如，以下内容将禁用任何 ID 为 `example.remote` 的计量表

```
management.metrics.enable.example.remote=false
```

以下属性允许每米定制：

表 57.1。Per-meter 自定义

Property	描述
`management.metrics.enable`	是否拒绝米发布任何指标。
`management.metrics.distribution.percentiles-histogram`	是否发布适合于计算可聚合（跨维度）百分位近似的直方图。
`management.metrics.distribution.minimum-expected-value`, `management.metrics.distribution.maximum-expected-value`	通过限制预期值的范围来发布更少的直方图桶。
`management.metrics.distribution.percentiles`	发布在您的应用程序中计算的百分位数值
`management.metrics.distribution.sla`	使用 SLA 定义的存储桶发布累积直方图。

有关 `percentiles-histogram`，`percentiles` 和 `sla` 背后概念的更多详细信息，请参阅千分尺文档的“直方图和百分位数”部分。

57.6 度量标准端点

Spring Boot 提供了一个 `metrics` 端点，可以在诊断上用于检查应用程序收集的指标。默认情况下端点不可用，必须公开，请参阅公开端点以获取更多详细信息。

导航到`/actuator/metrics` 会显示可用的仪表名称列表。您可以深入查看有关特定仪表的信息，方法是将其名称作为选择器，例如`/actuator/metrics/jvm.memory.max`。

小费

您在此处使用的名称应与代码中使用的名称相匹配，而不是在命名之后的名称 - 为其运送到的监视系统规范化的约定。换句话说，如果 `jvm.memory.max` 由于其蛇案例命名约定在 Prometheus 中显示为 `jvm_memory_max`，则在检查 `metrics` 端点中的仪表时仍应使用 `jvm.memory.max` 作为选择器。

您还可以在 URL 的末尾添加任意数量的 `tag=KEY:VALUE` 查询参数，以便按尺寸向下钻取仪表，例如`/actuator/metrics/jvm.memory.max?tag=area:nonheap`。

小费

报告的测量值是与仪表名称和已应用的任何标签匹配的所有仪表的统计数据的总和。因此，在上面的示例中，返回的“Value”统计信息是堆的“Code Cache”，“Compressed Class Space”和“Metaspace”区域的最大内存占用量的总和。如果您只想查看“Metaspace”的最大大小，可以添加额外的 `tag=id:Metaspace`，即`/actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace`。

58.审计

一旦 Spring 安全性发挥作用，Spring Boot Actuator 具有灵活的审计框架，可以发布事件（默认情况下，“身份验证成功”，“失败”和“访问被拒绝”例外）。此功能对于报告和基于身份验证失败实施锁定策略非常有用。要自定义已发布的安全事件，您可以提供自己的 `AbstractAuthenticationAuditListener` 和 `AbstractAuthorizationAuditListener` 实现。

您还可以将审计服务用于您自己的业务事件。为此，请将现有的 `AuditEventRepository` 注入您自己的组件并直接使用它或使用 Spring `ApplicationEventPublisher` 发布 `AuditApplicationEvent`（通过实施 `ApplicationEventPublisherAware`）。

59. HTTP 跟踪

将自动为所有 HTTP 请求启用跟踪。您可以查看 `httptrace` 端点并获取有关最近 100 次请求 - 响应交换的基本信息。

59.1 自定义 HTTP 跟踪

要自定义每个跟踪中包含的项目，请使用 `management.trace.http.include` 配置属性。要进行高级自定义，请考虑注册您自己的 `HttpExchangeTracer` 实现。

默认情况下，使用存储最后 100 个请求 - 响应交换的跟踪的 `InMemoryHttpTraceRepository`。如果您需要扩展容量，可以定义自己的 `InMemoryHttpTraceRepository` bean 实例。您还可以创建自己的替代 `HttpTraceRepository` 实现。

60.过程监测

在 `spring-boot` 模块中，您可以找到两个类来创建通常对进程监视有用的文件：

- `ApplicationPidFileWriter` 创建一个包含应用程序 PID 的文件（默认情况下，在应用程序目录中，文件名为 `application.pid`）。
- `WebServerPortFileWriter` 创建一个包含正在运行的 Web 服务器端口的文件（默认情况下，在文件名为 `application.port` 的应用程序目录中）。

默认情况下，这些编写器未激活，但您可以启用：

- 通过扩展配置
- 第 60.2 节“以编程方式”

60.1 扩展配置

在 `META-INF/spring.factories` 文件中，您可以激活写入 PID 文件的侦听器，如以下示例所示：

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.context.ApplicationPidFileWriter,\
org.springframework.boot.web.context.WebServerPortFileWriter
```

60.2 以编程方式

您还可以通过调用 `SpringApplication.addListeners(…)`方法并传递相应的 `Writer` 对象来激活侦听器。此方法还允许您在 `Writer` 构造函数中自定义文件名和路径。

61. Cloud Foundry 支持

Spring Boot 的执行器模块包括在部署到兼容的 Cloud Foundry 实例时激活的其他支持。`/cloudfoundryapplication` 路径为所有`@Endpoint` beans 提供了另一条安全路线。

通过扩展支持，可以使用 Spring Boot 执行器信息扩充 Cloud Foundry 管理 UI（例如可用于查看已部署应用程序的 Web 应用程序）。例如，应用程序状态页面可以包括完整的健康信息，而不是典型的“运行”或“停止”状态。

注意

常规用户无法直接访问`/cloudfoundryapplication` 路径。为了使用端点，必须与请求一起传递有效的 UAA 令牌。

61.1 禁用 Extended Cloud Foundry Actuator 支持

如果要完全禁用`/cloudfoundryapplication` 端点，可以将以下设置添加到 `application.properties` 文件中：

application.properties。

```
management.cloudfoundry.enabled=false
```

61.2 Cloud Foundry 自签名证书

默认情况下，`/cloudfoundryapplication` 端点的安全验证会对各种 Cloud Foundry 服务进行 SSL 调用。如果您的 Cloud Foundry UAA 或 Cloud Controller 服务使用自签名证书，则需要设置以下属性：

application.properties。

```
management.cloudfoundry.skip-ssl-validation=true
```

61.3 自定义上下文路径

如果服务器的上下文路径已配置为`/`以外的任何其他内容，则 Cloud Foundry 端点将不会在应用程序的根目录中可用。例如，如果 `server.servlet.context-path=/app`，Cloud Foundry 端点将在`/app/cloudfoundryapplication/*`处可用。

如果您希望 Cloud Foundry 端点始终在`/cloudfoundryapplication/*`处可用，则无论服务器的上下文路径如何，您都需要在应用程序中明确配置它。配置将根据使用的 Web 服务器而有所不同。对于 Tomcat，可以添加以下配置：

```
@Bean
public TomcatServletWebServerFactory servletWebServerFactory() {
        return new TomcatServletWebServerFactory() {

                @Override
                protected void prepareContext(Host host,
                                ServletContextInitializer[] initializers) {
                        super.prepareContext(host, initializers);
                        StandardContext child = new StandardContext();
                        child.addLifecycleListener(new Tomcat.FixContextListener());
                        child.setPath("/cloudfoundryapplication");
                        ServletContainerInitializer initializer =
getServletContextInitializer(
                                        getContextPath());
                        child.addServletContainerInitializer(initializer,
Collections.emptySet());
                        child.setCrossContext(true);
                        host.addChild(child);
                }

        };
}

private ServletContainerInitializer getServletContextInitializer(String contextPath)
{
        return (c, context) -> {
                Servlet servlet = new GenericServlet() {

                        @Override
                        public void service(ServletRequest req, ServletResponse res)
                                        throws ServletException, IOException {
                                ServletContext context = req.getServletContext()
                                                .getContext(contextPath);

context.getRequestDispatcher("/cloudfoundryapplication").forward(req,
                                                res);
                        }

                };
                context.addServlet("cloudfoundry", servlet).addMapping("/*");
        };
}
```

62.接下来要阅读的内容

如果您想探索本章中讨论的一些概念，您可以查看执行器示例应用程序。您还可以阅读有关图形工具的信息，例如 Graphite。

否则，您可以继续阅读有关“部署选项”的信息，或者继续阅读有关 Spring Boot 构建工具插件的一些深入信息 。

第六部分。部署 Spring Boot 应用程序

在部署应用程序时，Spring Boot 灵活的打包选项提供了大量选择。您可以将 Spring Boot 应用程序部署到各种云平台，容器映像（例如 Docker）或虚拟/真实计算机。

本节介绍一些更常见的部署方案。

63.部署到云端

Spring Boot 的可执行 jar 是现成的，适用于大多数流行的云 PaaS（平台即服务）提供商。这些提供商往往要求您“自带容器”。它们管理应用程序进程（而不是 Java 应用程序），因此它们需要一个中间层，使您的应用程序 适应云的运行过程概念。

两个流行的云提供商 Heroku 和 Cloud Foundry 采用“buildpack”方法。buildpack 将您部署的代码包装在启动应用程序所需的任何内容中。它可能是 JDK 和对 `java` 的调用，嵌入式 Web 服务器或完整的应用程序服务器。buildpack 是可插拔的，但理想情况下，您应该能够尽可能少地进行自定义。这减少了不受您控制的功能的占用空间。它最大限度地减少了开发和生产环境之间的差异。

理想情况下，您的应用程序（如 Spring Boot 可执行 jar）具有在其中运行打包所需的所有内容。

在本节中，我们将了解如何在“入门”部分中开发并在云中运行的 简单应用程序。

63.1 Cloud Foundry

如果未指定其他 buildpack，Cloud Foundry 将提供默认的构建包。Cloud Foundry Java buildpack 对 Spring 应用程序提供了出色的支持，包括 Spring Boot。您可以部署独立的可执行 jar 应用程序以及传统的 `.war` 打包应用程序。

构建应用程序（例如，使用 `mvn clean package`）并安装了 `cf` 命令行工具后，使用 `cf push` 命令部署应用程序，将路径替换为已编译的{12}。 /}。安装了 `cf` 命令行工具后，使用 `cf push` 命令部署应用程序，将路径替换为已编译的{2759} /}。在推送应用程序之前，请务必 使用 `cf` 命令行客户端登录。以下行显示使用 `cf push` 命令部署应用程序：

```
$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

注意

在前面的示例中，我们将 `acloudyspringtime` 替换为您提供的任何值 `cf` 作为应用程序的名称。

有关更多选项，请参阅 `cf push` 文档。如果 `manifest.yml` 同一目录中存在 Cloud Foundry 文件，则会考虑该文件。

此时，`cf` 开始上传您的应用程序，生成类似于以下示例的输出：

```
Uploading acloudyspringtime... OK
Preparing to start acloudyspringtime... OK
-----> Downloaded app package (8.9M)
-----> Java Buildpack Version: v3.12 (offline) |
https://github.com/cloudfoundry/java-buildpack.git#6f25b7e
-----> Downloading Open Jdk JRE 1.8.0_121 from https://java-
buildpack.cloudfoundry.org/openjdk/trusty/x86_64/openjdk-1.8.0_121.tar.gz (found in
cache)
       Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.6s)
-----> Downloading Open JDK Like Memory Calculator 2.0.2_RELEASE from https://java-
buildpack.cloudfoundry.org/memory-calculator/trusty/x86_64/memory-calculator-
2.0.2_RELEASE.tar.gz (found in cache)
       Memory Settings: -Xss349K -Xmx681574K -XX:MaxMetaspaceSize=104857K -Xms681574K
-XX:MetaspaceSize=104857K
-----> Downloading Container Certificate Trust Store 1.0.0_RELEASE from https://java-
buildpack.cloudfoundry.org/container-certificate-trust-store/container-certificate-
trust-store-1.0.0_RELEASE.jar (found in cache)
       Adding certificates to .java-
buildpack/container_certificate_trust_store/truststore.jks (0.6s)
-----> Downloading Spring Auto Reconfiguration 1.10.0_RELEASE from https://java-
```

```
buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-
1.10.0_RELEASE.jar (found in cache)
Checking status of app 'acloudyspringtime'...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  1 of 1 instances running (1 running)

App started
```

恭喜！该应用程序现已上线！

应用程序运行后，您可以使用 `cf apps` 命令验证已部署应用程序的状态，如以下示例所示：

```
$ cf apps
Getting applications in ...
OK

name                requested state   instances   memory   disk   urls
...
acloudyspringtime   started           1/1         512M     1G
acloudyspringtime.cfapps.io
...
```

一旦 Cloud Foundry 确认您的应用程序已部署，您应该能够在给定的 URI 处找到该应用程序。在前面的示例中，您可以在 `http://acloudyspringtime.cfapps.io/` 找到它。

63.1.1 绑定到服务

默认情况下，有关正在运行的应用程序的元数据以及服务连接信息将作为环境变量公开给应用程序（例如：`$VCAP_SERVICES`）。此体系结构决策归功于 Cloud Foundry 的多语言（任何语言和平台都可以作为 buildpack 支持）。进程范围的环境变量与语言无关。

环境变量并不总是适用于最简单的 API，因此 Spring Boot 会自动提取它们并将数据展平为可通过 Spring 的 `Environment` 抽象访问的属性，如以下示例所示：

```
@Component
class MyBean implements EnvironmentAware {

        private String instanceId;

        @Override
        public void setEnvironment(Environment environment) {
                this.instanceId =
environment.getProperty("vcap.application.instance_id");
        }

        // ...

}
```

所有 Cloud Foundry 属性都以 `vcap` 为前缀。您可以使用 `vcap` 属性来访问应用程序信息（例如应用程序的公共 URL）和服务信息（例如数据库凭据）。有关 完整的详细信息，请参阅 “CloudFoundryVcapEnvironmentPostProcessor” Javadoc。

小费

该 Spring 云连接器项目是任务，如配置数据源更适合。Spring Boot 包括

自动配置支持和 `spring-boot-starter-cloud-connectors` 启动器。

63.2 Heroku

Heroku 是另一个流行的 PaaS 平台。要自定义 Heroku 构建，请提供 `Procfile`，它提供部署应用程序所需的咒语。Heroku 为要使用的 Java 应用程序分配 `port`，然后确保路由到外部 URI 工作。

您必须将应用程序配置为侦听正确的端口。以下示例显示了我们的入门 REST 应用程序的 `Procfile`：

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

Spring Boot 使`-D` 个参数可用作可从 Spring `Environment` 实例访问的属性。`server.port` 配置属性被馈送到嵌入式 Tomcat，Jetty 或 Undertow 实例，然后在启动时使用该端口。`$PORT` 环境变量由 Heroku PaaS 分配给我们。

这应该是你需要的一切。Heroku 部署最常见的部署工作流程是 `git push` 生产代码，如以下示例所示：

```
$ git push heroku master

Initializing repository, done.
Counting objects: 95, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (78/78), done.
Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, done.
Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected
-----> Installing OpenJDK 1.8... done
-----> Installing Maven 3.3.1... done
-----> Installing settings.xml... done
-----> Executing: mvn -B -DskipTests=true clean install

       [INFO] Scanning for projects...
       Downloading: https://repo.spring.io/...
       Downloaded: https://repo.spring.io/... (818 B at 1.8 KB/sec)
                .....
       Downloaded: http://s3pository.heroku.com/jvm/... (152 KB at 595.3 KB/sec)
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/target/...
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ...
       [INFO]
------------------------------------------------------------------------
       [INFO] BUILD SUCCESS
       [INFO]
------------------------------------------------------------------------
       [INFO] Total time: 59.358s
       [INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014
       [INFO] Final Memory: 20M/493M
       [INFO]
------------------------------------------------------------------------

-----> Discovering process types
       Procfile declares types -> web

-----> Compressing... done, 70.4MB
-----> Launching... done, v6
       http://agile-sierra-1405.herokuapp.com/ deployed to Heroku

To git@heroku.com:agile-sierra-1405.git
 * [new branch]      master -> master
```

您的应用程序现在应该在 Heroku 上启动并运行。

63.3 OpenShift

OpenShift 是 Kubernetes 容器编排平台的 Red Hat 公共（和企业）扩展。与 Kubernetes 类似，OpenShift 有许多选项可用于安装基于 Spring Boot 的应用程序。

OpenShift 有许多资源描述如何部署 Spring Boot 应用程序，包括：

- 使用 S2I 构建器
- 建筑指南
- 在 Wildfly 上作为传统 Web 应用程序运行
- OpenShift Commons 简报

63.4 亚马逊网络服务（AWS）

Amazon Web Services 提供了多种方法来安装基于 Spring Boot 的应用程序，可以是传统的 Web 应用程序（war），也可以是带有嵌入式 Web 服务器的可执行 jar 文件。选项包括：

- AWS Elastic Beanstalk
- AWS Code Deploy
- AWS OPS Works
- AWS Cloud Formation
- AWS 容器注册表

每个都有不同的功能和定价模型。在本文档中，我们仅描述了最简单的选项：AWS Elastic Beanstalk。

63.4.1 AWS Elastic Beanstalk

正如官方的 Elastic Beanstalk Java 指南中所述，部署 Java 应用程序有两个主要选项。您可以使用“Tomcat 平台”或“Java SE 平台”。

使用 Tomcat 平台

此选项适用于生成 war 文件的 Spring Boot 项目。无需特殊配置。您只需遵循官方指南即可。

使用 Java SE 平台

此选项适用于生成 jar 文件并运行嵌入式 Web 容器的 Spring Boot 项目。Elastic Beanstalk 环境在端口 80 上运行 nginx 实例以代理在端口 5000 上运行的实际应用程序。要配置它，请将以下行添加到 `application.properties` 文件中：

```
server.port=5000
```

上传二进制文件而不是源代码

默认情况下，Elastic Beanstalk 上传源并在 AWS 中编译它们。但是，最好上传二进制文件。为此，请在 `.elasticbeanstalk/config.yml` 文件中添加与以下内容类似的行：

```
deploy:
        artifact: target/demo-0.0.1-SNAPSHOT.jar
```

通过设置环境类型来降低成本

默认情况下，Elastic Beanstalk 环境是负载平衡的。负载平衡器具有显着的成本。要避免此成本，请将环境类型设置为“Single instance”，如 Amazon 文档中所述。您还可以使用 CLI 和以下命令创建单实例环境：

```
eb create -s
```

63.4.2 摘要

这是访问 AWS 的最简单方法之一，但还有更多内容需要涉及，例如如何将 Elastic Beanstalk 集成到任何 CI / CD 工具中，使用 Elastic Beanstalk Maven 插件代替 CLI 和其他人。有一篇 博文更详细地介绍了这些主题。

63.5 Boxfuse 和亚马逊网络服务

Boxfuse 的工作原理是将您的 Spring Boot 可执行 jar 或 war 转换为可以在 VirtualBox 或 AWS 上无需部署的最小 VM 映像。Boxfuse 为 Spring Boot 提供深度集成，并使用 Spring Boot 配置文件中的信息自动配置端口和运行状况检查 URL。Boxfuse 利用这些信息来处理它产生的图像以及它提供的所有资源（实例，安全组，弹性负载平衡器等）。

创建 Boxfuse 帐户后，将其连接到您的 AWS 账户，安装最新版本的 Boxfuse 客户端，并确保该应用程序是由 Maven 或 Gradle 构建的（例如，使用 `mvn clean package`），您可以使用类似于以下内容的命令将您的 Spring Boot 应用程序部署到 AWS：

```
$ boxfuse run myapp-1.0.jar -env=prod
```

有关更多选项，请参阅 `boxfuse run` 文档。如果 `boxfuse.conf` 当前目录中存在文件，则会考虑该文件。

小费

默认情况下，Boxfuse 在启动时激活名为 `boxfuse` 的 Spring 个人资料。如果您的可执行 jar 或 war 包含一个 `application-boxfuse.properties` 文件，Boxfuse 将其配置基于它包含的属性。

此时，`boxfuse` 为您的应用程序创建一个映像，上传它，并在 AWS 上配置和启动必要的资源，从而产生类似于以下示例的输出：

```
Fusing Image for myapp-1.0.jar ...
Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0
Creating axelfontaine/myapp ...
Pushing axelfontaine/myapp:1.0 ...
Verifying axelfontaine/myapp:1.0 ...
Creating Elastic IP ...
Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...
Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in eu-central-1 (this may
take up to 50 seconds) ...
AMI created in 00:23.557s -> ami-d23f38cf
Creating security group boxfuse-sg_axelfontaine/myapp:1.0 ...
Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f38cf) in eu-central-
1 ...
Instance launched in 00:30.306s -> i-92ef9f53
Waiting for AWS to boot Instance i-92ef9f53 and Payload to start at
http://52.28.235.61/ ...
Payload started in 00:29.266s -> http://52.28.235.61/
Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...
Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...
```

```
Deployment completed successfully. axelfontaine/myapp:1.0 is up and running at
http://myapp-axelfontaine.boxfuse.io/
```

您的应用程序现在应该在 AWS 上启动并运行。

请参阅有关在 EC2 上部署 Spring Boot 应用程序的博客文章以及 Boxfuse Spring 启动集成的 文档，以开始使用 Maven 构建来运行应用程序。

63.6 Google Cloud

Google Cloud 有几个选项可用于启动 Spring Boot 应用程序。最容易上手的可能是 App Engine，但您也可以找到在带有 Container Engine 的容器中运行 Spring Boot 或在带有 Compute Engine 的虚拟机上运行的方法。

要在 App Engine 中运行，您可以首先在 UI 中创建项目，该项目为您设置唯一标识符并设置 HTTP 路由。将 Java 应用程序添加到项目中并将其留空，然后使用 Google Cloud SDK 将 Spring Boot 应用程序从命令行或 CI 构建推送到该插槽。

App Engine Standard 要求您使用 WAR 包装。按照 以下步骤 将 App Engine Standard 应用程序部署到 Google Cloud。

或者，App Engine Flex 要求您创建一个 `app.yaml` 文件来描述您的应用所需的资源。通常，您将此文件放在 `src/main/appengine` 中，它应该类似于以下文件：

```
service: default

runtime: java
env: flex

runtime_config:
  jdk: openjdk8

handlers:
- url: /.*
  script: this field is required, but ignored

manual_scaling:
  instances: 1

health_check:
  enable_health_check: False

env_variables:
  ENCRYPT_KEY: your_encryption_key_here
```

您可以通过将项目 ID 添加到构建配置来部署应用程序（例如，使用 Maven 插件），如以下示例所示：

```
<plugin>
        <groupId>com.google.cloud.tools</groupId>
        <artifactId>appengine-maven-plugin</artifactId>
        <version>1.3.0</version>
        <configuration>
                <project>myproject</project>
        </configuration>
</plugin>
```

然后使用 `mvn appengine:deploy` 进行部署（如果需要先进行身份验证，则构建失败）。

64.安装 Spring Boot 应用程序

除了使用 `java -jar` 运行 Spring Boot 应用程序之外，还可以为 Unix 系统创建完全可执行的应用程序。完全可执行的 jar 可以像任何其他可执行二进制文件一样执行，也可以 使用 `init.d` 或 `systemd` 注册。这使得在常见生产环境中安装和管理 Spring Boot 应用程序变得非常容易。

警告

完全可执行的 jar 通过在文件的前面嵌入额外的脚本来工作。目前，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` 可能无法提取已完全可执行的 jar 或战争。建议您只有在打算直接执行 jar 或 war 时才能使 jar 或 war 完全可执行，而不是使用 `java -jar` 运行它或将其部署到 servlet 容器。

要使用 Maven 创建“完全可执行”jar，请使用以下插件配置：

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

以下示例显示了等效的 Gradle 配置：

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

然后，您可以通过键入`./my-application.jar`（其中`my-application`是您的工件的名称）来运行您的应用程序。包含 jar 的目录用作应用程序的工作目录。

64.1 支持的操作系统

默认脚本支持大多数 Linux 发行版，并在 CentOS 和 Ubuntu 上进行测试。其他平台，如 OS X 和 FreeBSD，需要使用自定义 `embeddedLaunchScript`。

64.2 Unix / Linux 服务

Spring Boot 应用程序可以使用 `init.d` 或 `systemd` 轻松启动为 Unix / Linux 服务。

64.2.1 作为 `init.d` 服务安装（系统 V）

如果您配置了 Spring Boot 的 Maven 或 Gradle 插件来生成完全可执行的 jar，并且您没有使用自定义 `embeddedLaunchScript`，那么您的应用程序可以用作 `init.d` 服务。为此，请将 jar 符号链接到 `init.d` 以支持标准 `start`，`stop`，`restart` 和 `status` 命令。

该脚本支持以下功能：

- 以拥有 jar 文件的用户身份启动服务
- 使用`/var/run/<appname>/<appname>.pid`跟踪应用程序的 PID
- 将控制台日志写入`/var/log/<appname>.log`

假设您在`/var/myapp`中安装了 Spring Boot 应用程序，要将 Spring Boot 应用程序安装为

`init.d` 服务，请创建一个符号链接，如下所示：

```
$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp
```

安装后，您可以按常规方式启动和停止服务。例如，在基于 Debian 的系统上，您可以使用以下命令启动它：

```
$ service myapp start
```

小费

如果您的应用程序无法启动，请检查写入`/var/log/<appname>.log`的日志文件是否有错误。

您还可以使用标准操作系统工具标记应用程序以自动启动。例如，在 Debian 上，您可以使用以下命令：

```
$ update-rc.d myapp defaults <priority>
```

保护 `init.d` 服务

注意

以下是一组有关如何保护作为 init.d 服务运行的 Spring Boot 应用程序的指南。它并不是为了强化应用程序及其运行环境而应该做的所有事情的详尽列表。

当以 root 身份执行时，就像 root 用于启动 init.d 服务的情况一样，默认可执行脚本以拥有 jar 文件的用户身份运行应用程序。您永远不应该以 `root` 运行 Spring Boot 应用程序，因此您的应用程序的 jar 文件永远不应该由 root 拥有。相反，创建一个特定用户来运行您的应用程序并使用 `chown` 使其成为 jar 文件的所有者，如以下示例所示：

```
$ chown bootapp:bootapp your-app.jar
```

在这种情况下，默认可执行脚本以 `bootapp` 用户身份运行应用程序。

小费

为了减少应用程序用户帐户遭到入侵的可能性，您应该考虑阻止它使用登录 shell。例如，您可以将帐户的 shell 设置为`/usr/sbin/nologin`。

您还应该采取措施来防止修改应用程序的 jar 文件。首先，配置其权限，使其无法写入，只能由其所有者读取或执行，如以下示例所示：

```
$ chmod 500 your-app.jar
```

其次，如果您的应用程序或运行它的帐户受到损害，您还应该采取措施限制损害。如果攻击者确实获得了访问权限，他们可以使 jar 文件可写并更改其内容。防止这种情况的一种方法是使用 `chattr` 使其不可变，如以下示例所示：

```
$ sudo chattr +i your-app.jar
```

这将阻止任何用户（包括 root）修改 jar。

如果 root 用于控制应用程序的服务，并且您 使用 .conf 文件来自定义其启动，则 root 用户将读取并评估 .conf 文件。它应该得到相应的保护。使用 chmod 以便文件只能由所有者读取并使用 chown 使 root 成为所有者，如以下示例所示：

```
$ chmod 400 your-app.conf
$ sudo chown root:root your-app.conf
```

64.2.2 作为 systemd 服务安装

systemd 是 System V init 系统的后继者，现在被许多现代 Linux 发行版使用。虽然您可以继续将 init.d 脚本与 systemd 一起使用，但也可以使用 systemd'service'脚本启动 Spring Boot 应用程序。

假设您在/var/myapp 中安装了 Spring Boot 应用程序，要将 Spring Boot 应用程序安装为 systemd 服务，请创建名为 myapp.service 的脚本并将其放在/etc/systemd/system 目录中。以下脚本提供了一个示例：

```
[Unit]
Description=myapp
After=syslog.target

[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

重要

请务必更改应用的 Description，User 和 ExecStart 字段。

注意

ExecStart 字段未声明脚本操作命令，这意味着默认情况下使用 run 命令。

请注意，与作为 init.d 服务运行时不同，运行应用程序的用户，PID 文件和控制台日志文件由 systemd 本身管理，因此必须使用"中的相应字段进行配置。服务'脚本。有关更多详细信息，请参阅 服务单元配置手册页。

要将应用程序标记为在系统引导时自动启动，请使用以下命令：

```
$ systemctl enable myapp.service
```

有关详细信息，请参阅 man systemctl。

64.2.3 自定义启动脚本

Maven 或 Gradle 插件编写的默认嵌入式启动脚本可以通过多种方式进行自定义。对于大多数人来说，使用默认脚本和一些自定义通常就足够了。如果您发现无法自定义所需内容，请使用 embeddedLaunchScript 选项完全编写自己的文件。

写入时自定义启动脚本

在将脚本写入 jar 文件时自定义启动脚本的元素通常是有意义的。例如，init.d 脚本可以提供“描述”。由于您事先了解了描述（并且不需要更改），因此您可以在生成 jar 时提供它。

要自定义书面元素，请使用 Spring Boot Maven 插件的 `embeddedLaunchScriptProperties` 选项或 Spring Boot Gradle 插件 `launchScript` 的 `properties` 属性。

默认脚本支持以下属性替换：

名称	描述	Gradle 默认	Maven 默
`mode`	脚本模式。	`auto`	`auto`
`initInfoProvides`	“INIT INFO”的 `Provides` 部分	`${task.baseName}`	`${proje`
`initInfoRequiredStart`	{INI INFO“`Required-Start` 部分。	`$remote_fs $syslog $network`	`$remote $networ`
`initInfoRequiredStop`	`Required-Stop`“INIT INFO”部分。	`$remote_fs $syslog $network`	`$remote $networ`
`initInfoDefaultStart`	{INI INFO“`Default-Start` 部分。	`2 3 4 5`	`2 3 4 5`
`initInfoDefaultStop`	“INIT INFO”的 `Default-Stop` 部分。	`0 1 6`	`0 1 6`
`initInfoShortDescription`	{INI INFO“`Short-Description` 部分。	Single-line version of `${project.description}` (falling back to `${task.baseName}`)	`${proje`
`initInfoDescription`	“INIT INFO”的 `Description` 部分。	`${project.description}` (falling back to `${task.baseName}`)	`${proje` (falling ba `{projec`
`initInfoChkconfig`	{INI INFO“`chkconfig` 部分	`2345 99 01`	`2345 99`
`confFolder`	`CONF_FOLDER` 的默认值	Folder containing the jar	Folder cor
`inlinedConfScript`	引用应在默认启动脚本中内联的文件脚本。在加载任何外部配置文件之前，这可用于设置环境变量，例如 `JAVA_OPTS`		
`logFolder`	`LOG_FOLDER` 的默认值。仅适用于 `init.d` 服务		

名称	描述	Gradle 默认	Maven 默
logFilename	LOG_FILENAME 的默认值。仅对 init.d 服务有效		
pidFolder	PID_FOLDER 的默认值。仅适用于 init.d 服务		
pidFilename	PID_FOLDER 中 PID 文件名称的默认值。仅适用于 init.d 服务		
useStartStopDaemon	start-stop-daemon 命令是否可用，应该用于控制进程	true	true
stopWaitTime	STOP_WAIT_TIME 的默认值（以秒为单位）。仅对 init.d 服务有效	60	60

它运行时自定义脚本

对于在编写 jar 后需要自定义的脚本项，可以使用环境变量或配置文件。

默认脚本支持以下环境属性：

变量	描述
MODE	操作的“模式”。默认值取决于 jar 的构建方式，但通常是 auto（意味着它通过检查它是否是名为 init.d 的目录中的符号链接来尝试猜测它是否是 init 脚本）。如果要在前台运行脚本，可以将 stop|start|status|restart 命令显式设置为 service 或 run。
USE_START_STOP_DAEMON	start-stop-daemon 命令是否可用，应该用于控制进程。默认为 true。
PID_FOLDER	pid 文件夹的根名称（默认为/var/run）。
LOG_FOLDER	放置日志文件的文件夹的名称（默认为/var/log）。
CONF_FOLDER	从中读取.conf 文件的文件夹的名称（默认情况下与 jar 文件相同）。
LOG_FILENAME	LOG_FOLDER 中的日志文件名称（默认为<appname>.log）。
APP_NAME	应用程序的名称。如果 jar 从符号链接运行，则脚本会猜测应用程序名称。如果它不是符号链接或您想要显式设置应用程序名称，这可能很有用。
RUN_ARGS	传递给程序的参数（Spring Boot 应用程序）。
JAVA_HOME	默认情况下使用 PATH 发现 java 可执行文件的位置，但如果

变量	描述
	`$JAVA_HOME/bin/java` 处有可执行文件，则可以显式设置它。
`JAVA_OPTS`	启动时传递给 JVM 的选项。
`JARFILE`	jar 文件的显式位置，以防脚本用于启动实际上未嵌入的 jar。
`DEBUG`	如果不为空，则在 shell 进程上设置 `-x` 标志，以便于在脚本中查看逻辑。
`STOP_WAIT_TIME`	在强制关闭之前停止应用程序的等待时间（默认为 `60`）。

注意

`PID_FOLDER`，`LOG_FOLDER` 和 `LOG_FILENAME` 变量仅对 `init.d` 服务有效。对于 `systemd`，使用“service”脚本进行等效的自定义。有关更多详细信息，请参阅 服务单元配置手册页。

除 `JARFILE` 和 `APP_NAME` 之外，可以使用 `.conf` 文件配置上一节中列出的设置。该文件应该位于 jar 文件的旁边，并且具有相同的名称，但后缀为 `.conf` 而不是 `.jar`。例如，名为 `/var/myapp/myapp.jar` 的 jar 使用名为 `/var/myapp/myapp.conf` 的配置文件，如以下示例所示：

myapp.conf。

```
JAVA_OPTS=-Xmx1024M
LOG_FOLDER=/custom/log/folder
```

小费

如果您不喜欢在 jar 文件旁边有配置文件，可以设置 `CONF_FOLDER` 环境变量来自定义配置文件的位置。

要了解如何正确保护此文件，请参阅 保护 init.d 服务的准则。

64.3 Microsoft Windows 服务

可以使用启动 Spring Boot 应用程序作为 Windows 服务 `winsw`。

A（单独维护的示例）逐步介绍了如何为 Spring Boot 应用程序创建 Windows 服务。

65.接下来要阅读的内容

查看 Cloud Foundry， Heroku，OpenShift 和 Boxfuse 网站，了解有关 PaaS 可提供的各种功能的更多信息。这些只是四个最受欢迎的 Java PaaS 提供商。由于 Spring Boot 非常适合基于云的部署，因此您也可以自由地考虑其他提供商。

接下来的部分将介绍 *Spring Boot CLI*，或者您可以继续阅读有关 构建工具插件的内容。

第七部分。Spring Boot CLI

Spring Boot CLI 是一个命令行工具，如果您想快速开发 Spring 应用程序，可以使用它。它允许您运行 Groovy 脚本，这意味着您拥有熟悉的类似 Java 的语法，而没有太多的样板代码。您还可以引导新项目或为其编写自己的命令。

66.安装 CLI

可以使用 SDKMAN 手动安装 Spring Boot CLI（命令行界面）！（SDK Manager）或使用 Homebrew 或 MacPorts（如果您是 OSX 用户）。有关全面的安装说明，请参见 “入门”一节中的第 *10.2* 节“安装 *Spring Boot CLI*”。

67.使用 CLI

安装 CLI 后，可以通过键入 `spring` 并在命令行按 Enter 键来运行它。如果在没有任何参数的情况下运行 `spring`，将显示一个简单的帮助屏幕，如下所示：

```
$ spring
usage: spring [--help] [--version]
       <command> [<args>]

Available commands are:

  run [options] <files> [--] [args]
    Run a spring groovy script

  ... more command help is shown here
```

您可以键入 `spring help` 以获取有关任何支持的命令的更多详细信息，如以下示例所示：

```
$ spring help run
spring run - Run a spring groovy script

usage: spring run [options] <files> [--] [args]

Option                     Description
------                     -----------
--autoconfigure [Boolean]  Add autoconfigure compiler
                             transformations (default: true)
--classpath, -cp           Additional classpath entries
-e, --edit                 Open the file with the default system
                             editor
--no-guess-dependencies    Do not attempt to guess dependencies
--no-guess-imports         Do not attempt to guess imports
-q, --quiet                Quiet logging
-v, --verbose              Verbose logging of dependency
                             resolution
--watch                    Watch the specified file for changes
```

`version` 命令提供了一种快速检查您正在使用的 Spring Boot 版本的方法，如下所示：

```
$ spring version
Spring CLI v2.1.1.RELEASE
```

67.1 使用 CLI 运行应用程序

您可以使用 `run` 命令编译和运行 Groovy 源代码。Spring Boot CLI 是完全独立的，因此您不需要任何外部 Groovy 安装。

以下示例显示了使用 Groovy 编写的“hello world”Web 应用程序：

hello.groovy。

```
@RestController
class WebApplication {

        @RequestMapping("/")
        String home() {
                "Hello World!"
        }

}
```

要编译并运行该应用程序，请键入以下命令：

```
$ spring run hello.groovy
```

要将命令行参数传递给应用程序，请使用--将命令与“spring”命令参数分开，如以下示例所示：

```
$ spring run hello.groovy -- --server.port=9000
```

要设置 JVM 命令行参数，可以使用 JAVA_OPTS 环境变量，如以下示例所示：

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

注意

在 Microsoft Windows 上设置 JAVA_OPTS 时，请确保引用整个指令，例如 `set "JAVA_OPTS=-Xms256m -Xmx2048m"`。这样做可确保将值正确传递给流程。

67.1.1 扣除“抓住”依赖关系

标准 Groovy 包含一个@Grab 注释，它允许您声明对第三方库的依赖性。这个有用的技术让 Groovy 以与 Maven 或 Gradle 相同的方式下载 jar，但不需要你使用构建工具。

Spring Boot 进一步扩展了这种技术，并尝试根据您的代码推断出“抓取”哪些库。例如，由于前面显示的 WebApplication 代码使用@RestController 注释，Spring Boot 获取“Tomcat”和“Spring MVC”。

以下项目用作“抓取提示”：

项目	抓斗
JdbcTemplate, NamedParameterJdbcTemplate, DataSource	JDBC Application.
@EnableJms	JMS Application.
@EnableCaching	Caching abstraction.
@Test	JUnit.
@EnableRabbit	RabbitMQ.

项目	抓斗
extends `Specification`	Spock test.
`@EnableBatchProcessing`	Spring Batch.
`@MessageEndpoint` `@EnableIntegration`	Spring Integration.
`@Controller` `@RestController` `@EnableWebMvc`	Spring MVC + Embedded Tomcat.
`@EnableWebSecurity`	Spring Security.
`@EnableTransactionManagement`	Spring Transaction Management.

小费

请参阅 `CompilerAutoConfiguration` Spring Boot CLI 源代码中的子类， 以准确了解自定义的应用方式。

67.1.2 扣除“抓住”坐标

Spring Boot 通过允许您指定没有组或版本的依赖项（例如，`@Grab('freemarker')`）来扩展 Groovy 的标准`@Grab` 支持。这样做可以参考 Spring Boot 的默认依赖关系元数据来推断工件的组和版本。

注意

默认元数据与您使用的 CLI 版本相关联。只有当您移动到新版本的 CLI 时，它才会更改，让您可以控制依赖项版本何时更改。可以在附录中找到显示默认元数据中包含的依赖关系及其版本的表。

67.1.3 默认导入语句

为了帮助减小 Groovy 代码的大小，自动包含多个 `import` 语句。请注意前面的示例如何引用 `@Component`，`@RestController` 和`@RequestMapping`，而无需使用完全限定名称或 `import` 语句。

小费

许多 Spring 注释在不使用 `import` 语句的情况下工作。尝试运行应用程序以在添加导入之前查看失败的内容。

67.1.4 自动主方法

与等效的 Java 应用程序不同，您不需要在 `Groovy` 脚本中包含 `public static void main(String[] args)`方法。自动创建 `SpringApplication`，编译的代码充当 `source`。

67.1.5 自定义依赖关系管理

默认情况下，CLI 在解析`@Grab` 依赖项时使用 `spring-boot-dependencies` 中声明的依赖关

系管理。可以使用@DependencyManagementBom 注释配置覆盖默认依赖关系管理的其他依赖关系管理。注释的值应指定一个或多个 Maven BOM 的坐标（groupId:artifactId:version）。

例如，请考虑以下声明：

```
@DependencyManagementBom("com.example.custom-bom:1.0.0")
```

前面的声明在 com/example/custom-versions/1.0.0/下的 Maven 存储库中获取 custom-bom-1.0.0.pom。

指定多个 BOM 时，它们将按您声明的顺序应用，如以下示例所示：

```
@DependencyManagementBom(["com.example.custom-bom:1.0.0",
                "com.example.another-bom:1.0.0"])
```

上面的示例表明 another-bom 中的依赖关系管理会覆盖 custom-bom 中的依赖关系管理。

您可以在任何可以使用@Grab 的地方使用@DependencyManagementBom。但是，为了确保依赖关系管理的一致排序，您最多可以在应用程序中使用@DependencyManagementBom。一个有用的依赖关系管理源（它是 Spring Boot 的依赖关系管理的超集）是 Spring IO 平台，您可以在其中包含以下行：

```
@DependencyManagementBom('io.spring.platform:platform-bom:1.1.2.RELEASE')
```

67.2 具有多个源文件的应用程序

您可以对所有接受文件输入的命令使用“shell globbing”。这样做可以让您使用单个目录中的多个文件，如以下示例所示：

```
$ spring run *.groovy
```

67.3 打包您的应用程序

您可以使用 jar 命令将应用程序打包到一个自包含的可执行 jar 文件中，如以下示例所示：

```
$ spring jar my-app.jar *.groovy
```

生成的 jar 包含通过编译应用程序和所有应用程序的依赖项生成的类，以便可以使用 java -jar 运行它。jar 文件还包含应用程序类路径中的条目。您可以使用--include 和--exclude 添加和删除 jar 的显式路径。两者都以逗号分隔，并且都以“+”和“ -”的形式接受前缀，以表示它们应该从默认值中删除。默认包括如下：

```
public/**, resources/**, static/**, templates/**, META-INF/**, *
```

默认排除如下：

```
.*, repository/**, build/**, target/**, **/*.jar, **/*.groovy
```

在命令行上键入 spring help jar 以获取更多信息。

67.4 初始化新项目

init 命令允许您在不离开 shell 的情况下使用 start.spring.io 创建新项目，如以下示例所示：

```
$ spring init --dependencies=web,data-jpa my-project
Using service at https://start.spring.io
Project extracted to '/Users/developer/example/my-project'
```

上面的示例创建了一个 `my-project` 目录，其中包含基于 Maven 的项目，该项目使用 `spring-boot-starter-web` 和 `spring-boot-starter-data-jpa`。您可以使用`--list` 标志列出服务的功能，如以下示例所示：

```
$ spring init --list
=======================================
Capabilities of https://start.spring.io
=======================================

Available dependencies:
-----------------------
actuator - Actuator: Production ready features to help you monitor and manage your
application
...
web - Web: Support for full-stack web development, including Tomcat and spring-webmvc
websocket - Websocket: Support for WebSocket development
ws - WS: Support for Spring Web Services

Available project types:
------------------------
gradle-build -  Gradle Config [format:build, build:gradle]
gradle-project -  Gradle Project [format:project, build:gradle]
maven-build -  Maven POM [format:build, build:maven]
maven-project -  Maven Project [format:project, build:maven] (default)

...
```

`init` 命令支持许多选项。有关详细信息，请参阅 `help` 输出。例如，以下命令创建一个使用 Java 8 和 `war` 打包的 Gradle 项目：

```
$ spring init --build=gradle --java-version=1.8 --dependencies=websocket
--packaging=war sample-app.zip
Using service at https://start.spring.io
Content saved to 'sample-app.zip'
```

67.5 使用嵌入式 Shell

Spring Boot 包括 BASH 和 zsh shell 的命令行完成脚本。如果您不使用这些 shell 中的任何一个（可能是 Windows 用户），则可以使用 `shell` 命令启动集成 shell，如以下示例所示：

```
$ spring shell
Spring Boot (v2.1.1.RELEASE)
Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.
```

在嵌入式 shell 中，您可以直接运行其他命令：

```
$ version
Spring CLI v2.1.1.RELEASE
```

嵌入式 shell 支持 ANSI 颜色输出以及 `tab` 完成。如果需要运行本机命令，可以使用`!`前缀。要退出嵌入式 shell，请按 `ctrl-c`。

67.6 在 CLI 中添加扩展

您可以使用 `install` 命令向 CLI 添加扩展。该命令采用 `group:artifact:version` 格式的一组或多组工件坐标，如以下示例所示：

```
$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

除了安装由您提供的坐标标识的工件外，还会安装所有工件的依赖项。

要卸载依赖项，请使用 `uninstall` 命令。与 `install` 命令一样，它采用 `group:artifact:version` 格式的一组或多组工件坐标，如以下示例所示：

```
$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

它会卸载由您提供的坐标及其依赖项标识的工件。

要卸载所有其他依赖项，可以使用`--all` 选项，如以下示例所示：

```
$ spring uninstall --all
```

68.使用 Groovy Beans DSL 开发应用程序

Spring Framework 4.0 本身支持 `beans{}`“DSL”（从 Grails 借用 ），您可以使用相同的格式在您的 Groovy 应用程序脚本中嵌入 bean 定义。这有时是包含中间件声明等外部功能的好方法，如以下示例所示：

```
@Configuration
class Application implements CommandLineRunner {

        @Autowired
        SharedService service

        @Override
        void run(String... args) {
                println service.message
        }

}

import my.company.SharedService

beans {
        service(SharedService) {
                message = "Hello World"
        }
}
```

您可以将类声明与 `beans{}`混合在同一个文件中，只要它们保持在顶层，或者，如果您愿意，可以将 beans DSL 放在单独的文件中。

69.使用 `settings.xml` 配置 CLI

Spring Boot CLI 使用 Aether，Maven 的依赖性解析引擎来解析依赖关系。CLI 使用 `~/.m2/settings.xml` 中的 Maven 配置来配置 Aether。CLI 支持以下配置设置：

- 离线
- 镜子
- 服务器
- 代理
- 简介
    - 激活
    - 库

- 活动档案

有关详细信息，请参阅 Maven 的设置文档。

70.接下来要阅读的内容

GitHub 存储库中提供了一些示例 groovy 脚本，您可以使用它们来尝试 Spring Boot CLI。整个源代码中还有广泛的 Javadoc 。

如果您发现自己达到了 CLI 工具的限制，那么您可能希望将应用程序转换为完整的 Gradle 或 Maven 构建的“Groovy 项目”。下一节将介绍 Spring Boot 的“ 构建工具插件 ”，您可以将其与 Gradle 或 Maven 一起使用。

第八部分。构建工具插件

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，包括可执行 jar 的包装。本节提供了有关这两个插件的更多详细信息，以及在需要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 III 部分 ”中的“ 第 13 章，构建系统 ”，然后再使用 Spring Boot“ ”部分。

71. Spring Boot Maven 插件

该 Spring Boot Maven 插件提供了 Maven，让你打包可执行的 JAR 或战争档案和运行“就地”的应用程序 Spring Boot 支持。要使用它，您必须使用 Maven 3.2（或更高版本）。

注意

有关完整的插件文档，请参阅 Spring Boot Maven 插件站点。

71.1 包括插件

要使用 Spring Boot Maven 插件，请在 `pom.xml` 的 `plugins` 部分中包含相应的 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
http://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>
                                <version>2.1.1.RELEASE</version>
                                <executions>
                                        <execution>
                                                <goals>
                                                        <goal>repackage</goal>
                                                </goals>
                                        </execution>
                                </executions>
```

```
                        </plugin>
                </plugins>
        </build>
</project>
```

上述配置重新打包在 Maven 生命周期的 package 阶段构建的 jar 或 war。以下示例显示了重新打包的 jar 以及 target 目录中的原始 jar：

```
$ mvn package
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

如果您不包含<execution/>配置，如前面的示例所示，您可以单独运行插件（但仅在使用包目标时），如以下示例所示：

```
$ mvn package spring-boot:repackage
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

如果您使用里程碑或快照版本，则还需要添加相应的 pluginRepository 元素，如下面的清单所示：

```
<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>
```

71.2 打包可执行文件夹和 War 文件

一旦 pom.xml 中包含 spring-boot-maven-plugin，它就会自动尝试使用 spring-boot:repackage 目标重写存档以使其可执行。您应该使用通常的 packaging 元素将项目配置为构建 jar 或 war（视情况而定），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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <!-- ... -->
        <packaging>jar</packaging>
        <!-- ... -->
</project>
```

在 package 阶段，Spring Boot 增强了您现有的存档。您可以通过使用配置选项或通过以常规方式向清单添加 Main-Class 属性来指定要启动的主类。如果未指定主类，则插件将使用 public static void main(String[] args)方法搜索类。

要构建和运行项目工件，可以键入以下内容：

```
$ mvn package
$ java -jar target/mymodule-0.0.1-SNAPSHOT.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“已提供”，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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <!-- ... -->
        <packaging>war</packaging>
        <!-- ... -->
        <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>
</project>
```

小费

有关如何创建可部署的 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。

插件信息页面中提供了高级配置选项和示例 。

72. Spring Boot Gradle 插件

Spring Boot Gradle 插件在 Gradle 中提供 Spring Boot 支持，允许您打包可执行 jar 或 war 档案，运行 Spring Boot 应用程序，并使用 `spring-boot-dependencies` 提供的依赖关系管理。它需要 Gradle 4.4 或更高版本。请参阅插件的文档以了解更多信息：

- 参考（HTML 和 PDF）
- API

73. Spring Boot AntLib 模块

Spring Boot AntLib 模块为 Apache Ant 提供基本的 Spring Boot 支持。您可以使用该模块创建可执行 jar。要使用该模块，您需要在 `build.xml` 中声明一个额外的 `spring-boot` 命名空间，如以下示例所示：

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
        xmlns:spring-boot="antlib:org.springframework.boot.ant"
        name="myapp" default="build">
        ...
</project>
```

您需要记住使用 `-lib` 选项启动 Ant，如以下示例所示：

```
$ ant -lib <folder containing spring-boot-antlib-2.1.1.RELEASE.jar>
```

小费

“使用 Spring Boot”部分包含使用 Apache Ant 和 `spring-boot-antlib`

的更完整示例 。

73.1 Spring Boot Ant 任务

声明 `spring-boot-antlib` 命名空间后，可以使用以下附加任务：

- 第 73.1.1 节，“`spring-boot:exejar`”
- 第 73.2 节，“`spring-boot:findmainclass`”

73.1.1 `spring-boot:exejar`

您可以使用 `exejar` 任务创建一个 Spring Boot 可执行 jar。任务支持以下属性：

属性	描述	需要
`destfile`	要创建的目标 jar 文件	Yes
`classes`	Java 类文件的根目录	Yes
`start-class`	要运行的主要应用程序类	No *(the default is the first class found that declares a `main` method)*

以下嵌套元素可与任务一起使用：

元件	描述
`resources`	一个或多个资源集合，描述应添加到创建的 jar 文件内容中的一组资源。
`lib`	应该添加到构成应用程序的运行时依赖性类路径的 jar 库集的一个或多个资源集合。

73.1.2 例子

本节显示了 Ant 任务的两个示例。

指定 start-class。

```
<spring-boot:exejar destfile="target/my-application.jar"
                    classes="target/classes" start-class="com.example.MyApplication">
        <resources>
                <fileset dir="src/main/resources" />
        </resources>
        <lib>
                <fileset dir="lib" />
        </lib>
</spring-boot:exejar>
```

检测开始级别。

```
<exejar destfile="target/my-application.jar" classes="target/classes">
        <lib>
                <fileset dir="lib" />
        </lib>
</exejar>
```

73.2 spring-boot:findmainclass

exejar 内部使用 findmainclass 任务来查找声明 main 的类。如有必要，您还可以直接在构建中使用此任务。支持以下属性：

属性	描述	需要
classesroot	Java 类文件的根目录	Yes *(unless `mainclass` is specified)*
mainclass	可用于短路 main 类搜索	No
property	应该使用结果设置的 Ant 属性	No *(result will be logged if unspecified)*

73.2.1 例子

本节包含使用 findmainclass 的三个示例。

查找并记录。

```
<findmainclass classesroot="target/classes" />
```

查找并设置。

```
<findmainclass classesroot="target/classes" property="main-class" />
```

覆盖并设置。

```
<findmainclass mainclass="com.example.MainClass" property="main-class" />
```

74.支持其他构建系统

如果您想使用 Maven，Gradle 或 Ant 以外的构建工具，则可能需要开发自己的插件。可执行 jar 需要遵循特定的格式，并且某些条目需要以未压缩的形式编写（有关详细信息，请参阅附录中的“可执行 jar 格式”部分）。

Spring Boot Maven 和 Gradle 插件都使用 spring-boot-loader-tools 来实际生成 jar。如果需要，您可以直接使用此库。

74.1 重新包装档案

要重新打包现有存档以使其成为自包含的可执行存档，请使用 org.springframework.boot.loader.tools.Repackager。Repackager 类采用一个构造函数参数，该参数引用现有的 jar 或 war 存档。使用两种可用的 repackage()方法之一替换原始文件或写入新目标。在重新打包程序运行之前，还可以配置各种设置。

74.2 嵌套库

重新打包存档时，可以使用 org.springframework.boot.loader.tools.Libraries 接口包含对依赖项文件的引用。我们在这里没有提供 Libraries 的任何具体实现，因为它们通常是特定于构建系统的。

如果您的存档已包含库，则可以使用 Libraries.NONE。

74.3 寻找主要类别

如果不使用 `Repackager.setMainClass()`指定主类，则重新打包程序使用 ASM 读取类文件并尝试使用 `public static void main(String[] args)`方法查找合适的类。如果找到多个候选项，则抛出异常。

74.4 重新包装实施示例

以下示例显示了典型的重新打包实现：

```
Repackager repackager = new Repackager(sourceJarFile);
repackager.setBackupSource(false);
repackager.repackage(new Libraries() {
                        @Override
                        public void doWithLibraries(LibraryCallback callback) throws
IOException {
                                // Build system specific implementation, callback for
each dependency
                                // callback.library(new Library(nestedFile,
LibraryScope.COMPILE));
                        }
                });
```

75.接下来要阅读的内容

如果您对构建工具插件的工作方式感兴趣，可以 `spring-boot-tools` 在 GitHub 上查看该模块。附录中介绍了可执行 jar 格式的更多技术细节 。

如果您有与构建相关的特定问题，可以查看“操作方法 ”指南。

第九部分。'如何'指南

本节提供了使用 Spring Boot 时经常出现的一些常见'我该怎么做.....'的问题的答案。它的报道并不详尽，但确实涵盖了很多。

如果您遇到我们未在此处讨论的特定问题，您可能需要查看 stackoverflow.com 以查看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“操作方法”，请向我们发送拉取请求。

76. Spring Boot 申请

本节包括与 Spring Boot 应用程序直接相关的主题。

76.1 创建自己的失败分析器

`FailureAnalyzer` 是一种在启动时拦截异常并将其转换为人类可读消息的好方法，包含在 `FailureAnalysis`。Spring Boot 为应用程序上下文相关异常，JSR-303 验证等提供了这样的分析器。您也可以创建自己的。

`AbstractFailureAnalyzer` 是 `FailureAnalyzer` 的方便扩展，用于检查要处理的异常中是否存在指定的异常类型。您可以从中进行扩展，以便您的实现只有在实际存在时才有机会处理异常。如果由于某种原因，您无法处理异常，请返回 `null` 以使另一个实现有机会处理异常。

FailureAnalyzer 实施必须在 META-INF/spring.factories 中注册。以下示例注册 ProjectConstraintViolationFailureAnalyzer：

```
org.springframework.boot.diagnostics.FailureAnalyzer=\
com.example.ProjectConstraintViolationFailureAnalyzer
```

注意

如果您需要访问 BeanFactory 或 Environment，则您的 FailureAnalyzer 可以分别实现 BeanFactoryAware 或 EnvironmentAware。

76.2 自动配置故障排除

Spring Boot 自动配置尽力“做正确的事”，但有时事情会失败，而且很难说清楚原因。

任何 Spring Boot ApplicationContext 都有一个非常有用的 ConditionEvaluationReport。如果启用 DEBUG 日志记录输出，则可以看到它。如果使用 spring-boot-actuator（参见 Actuator 章节），还有一个 conditions 端点以 JSON 格式呈现报表。使用该端点调试应用程序，并在运行时查看 Spring Boot 添加了哪些功能（以及哪些尚未添加）。

通过查看源代码和 Javadoc 可以回答更多问题。阅读代码时，请记住以下经验法则：

- 查找名为*AutoConfiguration 的类并阅读其来源。请特别注意@Conditional*注释，以了解它们启用哪些功能以及何时启用。将--debug 添加到命令行或系统属性-Ddebug，以在控制台上记录应用程序中所做的所有自动配置决策。在正在运行的 Actuator 应用程序中，查看 conditions 端点（/actuator/conditions 或 JMX 等效物）以获取相同的信息。
- 查找@ConfigurationProperties（例如 ServerProperties）的类， 并从那里读取可用的外部配置选项。@ConfigurationProperties 注释具有 name 属性，该属性充当外部属性的前缀。因此，ServerProperties 具有 prefix="server"，其配置属性为 server.port，server.address 等。在正在运行的 Actuator 应用程序中，查看 configprops 端点。
- 查看 Binder 上 bind 方法的使用，以轻松的方式从 Environment 中明确地提取配置值。它通常与前缀一起使用。
- 查找直接绑定到 Environment 的@Value 注释。
- 查找用于响应 SpEL 表达式打开和关闭功能的@ConditionalOnExpression 注释，通常使用从 Environment 解析的占位符进行评估。

76.3 在开始之前自定义环境或 ApplicationContext

SpringApplication 具有 ApplicationListeners 和 ApplicationContextInitializers，用于将自定义应用于上下文或环境。Spring Boot 加载了许多此类自定义项，以便在 META-INF/spring.factories 内部使用。注册其他自定义项的方法不止一种：

- 按程序编程，在运行之前调用 SpringApplication 上的 addListeners 和 addInitializers 方法。
- 声明地，根据应用程序，通过设置 context.initializer.classes 或 context.listener.classes 属性。
- 声明性地，对于所有应用程序，通过添加 META-INF/spring.factories 并打包应用程序全部用作库的 jar 文件。

SpringApplication 向侦听器发送一些特殊的 ApplicationEvents（有些甚至在创建上下文之前），然后为 ApplicationContext 发布的事件注册侦听器。有关完整列表，请参阅“Spring Boot 功能”部分中的“ 第 23.5 节”“应用程序事件和监听器”。

在使用 EnvironmentPostProcessor 刷新应用程序上下文之前，还可以自定义 Environment。每个实现都应在 META-INF/spring.factories 中注册，如以下示例所示：

```
org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPost
Processor
```

该实现可以加载任意文件并将它们添加到 Environment。例如，以下示例从类路径加载 YAML 配置文件：

```
public class EnvironmentPostProcessorExample implements EnvironmentPostProcessor {

        private final YamlPropertySourceLoader loader = new
YamlPropertySourceLoader();

        @Override
        public void postProcessEnvironment(ConfigurableEnvironment environment,
                        SpringApplication application) {
                Resource path = new
ClassPathResource("com/example/myapp/config.yml");
                PropertySource<?> propertySource = loadYaml(path);
                environment.getPropertySources().addLast(propertySource);
        }

        private PropertySource<?> loadYaml(Resource path) {
                if (!path.exists()) {
                        throw new IllegalArgumentException("Resource " + path + "
does not exist");
                }
                try {
                        return this.loader.load("custom-resource", path).get(0);
                }
                catch (IOException ex) {
                        throw new IllegalStateException(
                                        "Failed to load yaml configuration from " +
path, ex);
                }
        }

}
```

小费

Environment 已经准备好了 Spring Boot 默认加载的所有常用属性源。因此，可以从环境中获取文件的位置。前面的示例在列表末尾添加了 custom-resource 属性源，以便在任何其他常用位置中定义的键优先。自定义实现可以定义另一个订单。

警告

虽然在@SpringBootApplication 上使用@PropertySource 似乎是在 Environment 中加载自定义资源的一种方便而简单的方法，但我们不建议这样做，因为 Spring Boot 准备了 Environment 在刷新 ApplicationContext 之前。使用@PropertySource 定义的任何键加载太晚，不会对自动配置产生任何影响。

76.4 构建 ApplicationContext 层次结构（添加父或根上下文）

您可以使用 `ApplicationBuilder` 类创建父/子 `ApplicationContext` 层次结构。有关详细信息，请参阅“ Spring Boot 功能”部分中的“ 第 23.4 节 ”，“Fluent Builder API”。

76.5 创建非 Web 应用程序

并非所有 Spring 应用程序都必须是 Web 应用程序（或 Web 服务）。如果要在 `main` 方法中执行某些代码，而且还要引导 Spring 应用程序来设置要使用的基础结构，则可以使用 Spring Boot 的 `SpringApplication` 功能。`SpringApplication` 更改其 `ApplicationContext` 类，具体取决于它是否认为它需要 Web 应用程序。您可以做的第一件事就是将与服务器相关的依赖项（例如 servlet API）从类路径中删除。如果您不能这样做（例如，您从相同的代码库运行两个应用程序），那么您可以在 `SpringApplication` 实例上显式调用 `setWebApplicationType(WebApplicationType.NONE)`或设置 `applicationContextClass` 属性（通过 Java API 或与外部属性）。您希望作为业务逻辑运行的应用程序代码可以实现为 `CommandLineRunner` 并作为`@Bean` 定义放入上下文中。

77.属性和配置

本节包括有关设置和读取属性和配置设置及其与 Spring Boot 应用程序交互的主题。

77.1 在构建时自动展开属性

您可以使用现有的构建配置自动扩展它们，而不是硬编码在项目的构建配置中也指定的某些属性。Maven 和 Gradle 都可以实现这一点。

77.1.1 使用 Maven 自动 Property 扩展

您可以使用资源过滤自动从 Maven 项目扩展属性。如果您使用 `spring-boot-starter-parent`，则可以使用`@..@`占位符引用 Maven'项目属性'，如以下示例所示：

```
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

注意

只有生产配置以这种方式过滤（换句话说，在 `src/test/resources` 上不应用过滤）。

小费

如果启用 `addResources` 标志，则 `spring-boot:run` 目标可以将 `src/main/resources` 直接添加到类路径（用于热重新加载）。这样做可以绕过资源过滤和此功能。相反，您可以使用 `exec:java` 目标或自定义插件的配置。有关详细信息，请参阅 插件使用情况页面。

如果您不使用 starter 父级，则需要在 `pom.xml` 的`<build/>`元素中包含以下元素：

```
<resources>
        <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
        </resource>
```

```
</resources>
```

您还需要在<plugins/>中包含以下元素：

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-resources-plugin</artifactId>
        <version>2.7</version>
        <configuration>
                <delimiters>
                        <delimiter>@</delimiter>
                </delimiters>
                <useDefaultDelimiters>false</useDefaultDelimiters>
        </configuration>
</plugin>
```

注意

如果在配置中使用标准 Spring 占位符（例如${placeholder}），则 useDefaultDelimiters 属性非常重要。如果该属性未设置为 false，则可以通过构建扩展这些属性。

77.1.2 使用 Gradle 自动 Property 扩展

您可以通过配置 Java 插件的 processResources 任务来自动扩展 Gradle 项目中的属性，如以下示例所示：

```
processResources {
        expand(project.properties)
}
```

然后，您可以使用占位符来引用您的 Gradle 项目的属性，如以下示例所示：

```
app.name=${name}
app.description=${description}
```

注意

Gradle 的 expand 方法使用 Groovy 的 SimpleTemplateEngine，它会转换${..}令牌。${..}样式与 Spring 自己的属性占位符机制冲突。要将 Spring 属性占位符与自动扩展一起使用，请按如下方式转义 Spring 属性占位符：\${..}。

77.2 外部化 SpringApplication 的配置

SpringApplication 具有 bean 属性（主要是 setter），因此您可以在创建应用程序时使用其 Java API 来修改其行为。或者，您可以通过在 spring.main.*中设置属性来外部化配置。例如，在 application.properties 中，您可能具有以下设置：

```
spring.main.web-application-type=none
spring.main.banner-mode=off
```

然后在启动时不打印 Spring Boot 横幅，并且应用程序未启动嵌入式 Web 服务器。

外部配置中定义的属性会覆盖使用 Java API 指定的值，但用于创建 ApplicationContext 的源的明显例外。考虑以下应用程序：

```
new SpringApplicationBuilder()
        .bannerMode(Banner.Mode.OFF)
        .sources(demo.MyApp.class)
        .run(args);
```

现在考虑以下配置：

```
spring.main.sources=com.acme.Config,com.acme.ExtraConfig
spring.main.banner-mode=console
```

实际应用程序现在显示横幅（由配置覆盖）并使用 ApplicationContext 的三个来源（按以下顺序）：demo.MyApp，com.acme.Config 和 com.acme.ExtraConfig。

77.3 更改应用程序外部属性的位置

默认情况下，来自不同来源的属性会按照定义的顺序添加到 Spring Environment（有关确切的顺序，请参阅“ Spring Boot 功能”部分中的“ 第 24 章，外部化配置 ”）。

增加和修改此排序的一种好方法是向应用程序源添加@PropertySource 注释。传递给 SpringApplication 静态便捷方法的类和使用 setSources()添加的类将被检查以查看它们是否具有@PropertySources。如果他们这样做，那么这些属性会尽早添加到 Environment，以便在 ApplicationContext 生命周期的所有阶段中使用。以这种方式添加的属性的优先级低于使用默认位置（例如 application.properties），系统属性，环境变量或命令行添加的属性。

您还可以提供以下系统属性（或环境变量）来更改行为：

- spring.config.name（SPRING_CONFIG_NAME）：默认为 application 作为文件名的根。
- spring.config.location（SPRING_CONFIG_LOCATION）：要加载的文件（例如类路径资源或 URL）。为此文档设置了单独的 Environment 属性源，它可以被系统属性，环境变量或命令行覆盖。

无论您在环境中设置什么，Spring Boot 总是如上所述加载 application.properties。默认情况下，如果使用 YAML，则扩展名为“.yml”的文件也会添加到列表中。

Spring Boot 记录在 DEBUG 级别加载的配置文件以及在 TRACE 级别找不到的候选项。

有关 ConfigFileApplicationListener 详细信息，请参阅

77.4 使用“短”命令行参数

有些人喜欢使用（例如）--port=9000 而不是--server.port=9000 来在命令行上设置配置属性。您可以使用 application.properties 中的占位符启用此行为，如以下示例所示：

```
server.port=${port:8080}
```

小费

如果您从 spring-boot-starter-parent POM 继承，则 maven-resources-plugins 的默认过滤器令牌已从${*}更改为@（即@maven.token@而不是${maven.token}）防止与 Spring 样式占位符发生冲突。如果您已直接为 application.properties 启用

Maven 过滤，则可能还需要更改默认过滤器令牌以使用 其他分隔符。

注意

在这种特定情况下，端口绑定可在 Paoku 环境（如 Heroku 或 Cloud Foundry）中运行。在这两个平台中，`PORT` 环境变量自动设置，Spring 可以绑定到 `Environment` 属性的大写同义词。

77.5 将 YAML 用于外部属性

YAML 是 JSON 的超集，因此，它是以分层格式存储外部属性的便捷语法，如以下示例所示：

```
spring:
        application:
                name: cruncher
        datasource:
                driverClassName: com.mysql.jdbc.Driver
                url: jdbc:mysql://localhost/test
server:
        port: 9000
```

创建一个名为 `application.yml` 的文件并将其放在类路径的根目录中。然后将 `snakeyaml` 添加到您的依赖项（Maven 坐标 `org.yaml:snakeyaml`，如果您使用 `spring-boot-starter`，则已包含在内。将 YAML 文件解析为 Java `Map<String,Object>`（类似于 JSON 对象），并且 Spring Boot 将地图展平，使其深度为一级并具有句点分隔的键，因为很多人习惯使用 {11 Java 中的/}文件。

上面的示例 YAML 对应于以下 `application.properties` 文件：

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

有关 YAML 的更多信息，请参见“ Spring Boot 功能”部分中的“ 第 24.7 节 ”，“使用 YAML 而不是属性”。

77.6 设置活动 Spring Profiles

Spring `Environment` 有一个 API，但您通常会设置一个 System 属性（`spring.profiles.active`）或一个 OS 环境变量（`SPRING_PROFILES_ACTIVE`）。此外，您可以使用 `-D` 参数启动应用程序（请记住将其放在主类或 jar 存档之前），如下所示：

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

在 Spring Boot 中，您还可以在 `application.properties` 中设置活动配置文件，如以下示例所示：

```
spring.profiles.active=production
```

以这种方式设置的值将由 System 属性或环境变量设置替换，但不会由 `SpringApplicationBuilder.profiles()`方法替换。因此，后一个 Java API 可用于扩充配置文件而不更改默认值。

有关详细信息，请参阅“ Spring Boot 功能”部分中的“ 第 25 章，配置文件 ”。

77.7 根据环境更改配置

YAML 文件实际上是由`---`行分隔的文档序列，每个文档分别解析为展平的地图。

如果 YAML 文档包含 `spring.profiles` 键，则配置文件值（以逗号分隔的配置文件列表）将输入 Spring `Environment.acceptsProfiles()`方法。如果这些配置文件中的任何一个处于活动状态，那么该文档将包含在最终合并中（否则，它不会），如以下示例所示：

```
server:
        port: 9000
---

spring:
        profiles: development
server:
        port: 9001

---

spring:
        profiles: production
server:
        port: 0
```

在前面的示例中，默认端口为 9000.但是，如果名为“development”的 Spring 配置文件处于活动状态，则端口为 9001.如果“production”处于活动状态，则端口为 0。

注意

YAML 文档按它们遇到的顺序合并。以后的值会覆盖以前的值。

要对属性文件执行相同操作，可以使用 `application-${profile}.properties` 指定特定于配置文件的值。

77.8 发现外部属性的内置选项

Spring Boot 在运行时将 `application.properties`（或 `.yml` 文件和其他位置）的外部属性绑定到应用程序中。没有（并且在技术上不可能）单个位置中所有受支持属性的详尽列表，因为贡献可以来自类路径上的其他 jar 文件。

具有 Actuator 功能的正在运行的应用程序具有 `configprops` 端点，该端点显示通过 `@ConfigurationProperties` 可用的所有绑定和可绑定属性。

附录中包含一个 `application.properties` 示例，其中列出了 Spring Boot 支持的最常见属性。最终列表来自于搜索`@ConfigurationProperties` 和`@Value` 注释的源代码以及偶尔使用 `Binder`。有关加载属性的确切顺序的更多信息，请参见“ 第 24 章，外部化配置 ”。

78.嵌入式 Web 服务器

每个 Spring Boot Web 应用程序都包含一个嵌入式 Web 服务器。此功能会导致许多操作方法问题，包括如何更改嵌入式服务器以及如何配置嵌入式服务器。本节回答了这些问题。

78.1 使用其他 Web 服务器

许多 Spring Boot 启动器包括默认嵌入式容器。

- 对于 servlet 堆栈应用程序，`spring-boot-starter-web` 包括 `spring-boot-starter-tomcat` 包含 Tomcat，但您可以使用 `spring-boot-starter-jetty` 或 `spring-boot-starter-undertow`。
- 对于反应堆栈应用程序，`spring-boot-starter-webflux` 包括 `spring-boot-starter-reactor-netty` 包含 Reactor Netty，但您可以使用 `spring-boot-starter-tomcat`，`spring-boot-starter-jetty` 或 `spring-boot-starter-undertow`。

切换到其他 HTTP 服务器时，除了包含所需的依赖项外，还需要排除默认依赖项。Spring Boot 为 HTTP 服务器提供单独的启动程序，以帮助使此过程尽可能简单。

以下 Maven 示例显示了如何排除 Tomcat 并为 Spring MVC 包含 Jetty：

```
<properties>
        <servlet-api.version>3.1.0</servlet-api.version>
</properties>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
                <!-- Exclude the Tomcat dependency -->
                <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<!-- Use Jetty instead -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

注意

Servlet API 的版本已被覆盖，与 Tomcat 9 和 Undertow 2.0 不同，Jetty 9.4 不支持 Servlet 4.0。

以下 Gradle 示例显示了如何排除 Netty 并为 Spring WebFlux 包含 Undertow：

```
configurations {
        // exclude Reactor Netty
        compile.exclude module: 'spring-boot-starter-reactor-netty'
}

dependencies {
        compile 'org.springframework.boot:spring-boot-starter-webflux'
        // Use Undertow instead
        compile 'org.springframework.boot:spring-boot-starter-undertow'
        // ...
}
```

注意

使用 `WebClient` 类需要 `spring-boot-starter-reactor-netty`，因此即使需要包含不同的 HTTP 服务器，也可能需要依赖 Netty。

78.2 禁用 Web 服务器

如果您的类路径包含启动 Web 服务器所需的位，Spring Boot 将自动启动它。要禁用此行为，请在 `application.properties` 中配置 `WebApplicationType`，如以下示例所示：

```
spring.main.web-application-type=none
```

78.3 更改 HTTP 端口

在独立应用程序中，主 HTTP 端口默认为 `8080`，但可以使用 `server.port` 设置（例如，在 `application.properties` 或作为系统属性）。由于轻松绑定 `Environment` 值，您还可以使用 `SERVER_PORT`（例如，作为 OS 环境变量）。

要完全关闭 HTTP 端点但仍然创建 `WebApplicationContext`，请使用 `server.port=-1`。（这样做有时对测试很有用。）

有关更多详细信息，请参阅“ Spring Boot 功能”部分中的“ 第 28.4.4 节 ”，“自定义嵌入式 Servlet 容器”或 `ServerProperties` 源代码。

78.4 使用随机未分配的 HTTP 端口

要扫描空闲端口（使用 OS 本机来防止冲突），请使用 `server.port=0`。

78.5 在运行时发现 HTTP 端口

您可以从日志输出或 `ServletWebServerApplicationContext` 到 `WebServer` 访问服务器运行的端口。获得它并确保它已被初始化的最佳方法是添加`@Bean` 类型 `ApplicationListener<ServletWebServerInitializedEvent>`并在发布时将容器拉出事件。

使用`@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`的测试也可以使用`@LocalServerPort` 注释将实际端口注入字段，如以下示例所示：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

        @Autowired
        ServletWebServerApplicationContext server;

        @LocalServerPort
        int port;

        // ...

}
```

注意

`@LocalServerPort` 是`@Value("${local.server.port}")`的元注释。不要尝试在常规应用程序中注入端口。正如我们刚刚看到的那样，只有在容器初始化之后才设置该值。与测试相反，应用程序代码回调会尽早处理（在值实际可用之前）。

78.6 启用 HTTP 响应压缩

Jetty，Tomcat 和 Undertow 支持 HTTP 响应压缩。它可以在 `application.properties` 中启用，如下所示：

```
server.compression.enabled=true
```

默认情况下，响应必须至少为 2048 字节，才能执行压缩。您可以通过设置 `server.compression.min-response-size` 属性来配置此行为。

默认情况下，只有在内容类型为以下内容之一时才会压缩响应：

- `text/html`
- `text/xml`
- `text/plain`
- `text/css`
- `text/javascript`
- `application/javascript`
- `application/json`
- `application/xml`

您可以通过设置 `server.compression.mime-types` 属性来配置此行为。

78.7 配置 SSL

可以通过设置各种 `server.ssl.*`属性以声明方式配置 SSL，通常在 `application.properties` 或 `application.yml`。以下示例显示了在 `application.properties` 中设置 SSL 属性：

```
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=secret
server.ssl.key-password=another-secret
```

有关 `Ssl` 所有支持的属性的详细信息，请参阅。

使用上述示例之类的配置意味着应用程序不再支持端口 8080 上的普通 HTTP 连接器。Spring Boot 不支持通过 `application.properties` 配置 HTTP 连接器和 HTTPS 连接器。如果要同时使用这两者，则需要以编程方式配置其中一个。我们建议使用 `application.properties` 配置 HTTPS，因为 HTTP 连接器更容易以编程方式配置。有关 `spring-boot-sample-tomcat-multi-connectors` 示例，请参阅 示例项目。

78.8 配置 HTTP / 2

您可以使用 `server.http2.enabled` 配置属性在 Spring Boot 应用程序中启用 HTTP / 2 支持。此支持取决于所选的 Web 服务器和应用程序环境，因为 JDK8 不支持该协议。

注意

Spring Boot 不支持 `h2c`，即 HTTP / 2 协议的明文版本。因此，您必须先配置 SSL。

78.8.1 带有 Undertow 的 HTTP / 2

从 Undertow 1.4.0+开始，支持 HTTP / 2，对 JDK8 没有任何额外要求。

78.8.2 带 Jetty 的 HTTP / 2

从 Jetty 9.4.8 开始，Conscrypt 库也支持 HTTP / 2 。要启用该支持，您的应用程序需要有两个额外的依赖项：`org.eclipse.jetty:jetty-alpn-conscrypt-server` 和 `org.eclipse.jetty.http2:http2-server`。

78.8.3 使用 Tomcat 的 HTTP / 2

Spring Boot 默认使用 Tomcat 9.0.x，它在使用 JDK 9 或更高版本时支持 HTTP / 2 开箱即用。或者，如果在主机操作系统上安装了 `libtcnative` 库及其依赖项，则可以在 JDK 8 上使用 HTTP / 2。

必须使库文件夹（如果尚未可用）到 JVM 库路径。您可以使用 `-Djava.library.path=/usr/local/opt/tomcat-native/lib` 等 JVM 参数执行此操作。有关 Tomcat 官方文档的更多 信息。

在没有该本机支持的情况下在 JDK 8 上启动 Tomcat 9.0.x 会记录以下错误：

```
ERROR 8787 --- [           main] o.a.coyote.http11.Http11NioProtocol      : The
upgrade handler [org.apache.coyote.http2.Http2Protocol] for [h2] only supports
upgrade via ALPN but has been configured for the ["https-jsse-nio-8443"] connector
that does not support ALPN.
```

此错误不是致命的，应用程序仍然以 HTTP / 1.1 SSL 支持启动。

78.8.4 带有 Reactor Netty 的 HTTP / 2

`spring-boot-webflux-starter` 默认使用 Reactor Netty 作为服务器。可以使用 JDK 9 或更高版本的 JDK 支持为 Reactor Netty 配置 HTTP / 2。对于 JDK 8 环境或最佳运行时性能，此服务器还支持具有本机库的 HTTP / 2。要启用它，您的应用程序需要具有其他依赖项。

Spring Boot 管理 `io.netty:netty-tcnative-boringssl-static`“超级 jar”的版本，包含所有平台的本机库。开发人员可以选择使用分类器仅导入所需的依赖项（请参阅 Netty 官方文档）。

78.9 配置 Web 服务器

通常，您应首先考虑使用众多可用配置键中的一个，并通过在 `application.properties`（或 `application.yml` 或环境等）中添加新条目来自定义您的 Web 服务器。请参阅“ 第 77.8 节”，“发现内置”在外部属性选项中“ ”）。`server.*`命名空间在这里非常有用，它包括 `server.tomcat.*`，`server.jetty.*`等名称空间，用于特定于服务器的功能。请参阅附录 A，常见应用程序属性列表。

前面的部分涵盖了许多常见用例，例如压缩，SSL 或 HTTP / 2。但是，如果您的用例不存在配置密钥，则应该查看 `WebServerFactoryCustomizer`。您可以声明这样的组件并获得与您选择的服务器工厂相关的访问权限：您应该为所选服务器（Tomcat，Jetty，Reactor Netty，Undertow）和所选 Web 堆栈（Servlet 或 Reactive）选择变体。

以下示例适用于具有 `spring-boot-starter-web`（Servlet 堆栈）的 Tomcat：

```
@Component
```

```
public class MyTomcatWebServerCustomizer
                implements WebServerFactoryCustomizer<TomcatServletWebServerFactory>
{

        @Override
        public void customize(TomcatServletWebServerFactory factory) {
                // customize the factory here
        }
}
```

此外，Spring Boot 提供：

服务器	Servlet 堆栈	反应堆栈
Tomcat	`TomcatServletWebServerFactory`	`TomcatReactiveWebServerFactory`
Jetty	`JettyServletWebServerFactory`	`JettyReactiveWebServerFactory`
Undertow	`UndertowServletWebServerFactory`	`UndertowReactiveWebServerFactory`
Reactor	N/A	`NettyReactiveWebServerFactory`

一旦您有权访问 `WebServerFactory`，您通常可以向其添加定制器以配置特定部件，例如连接器，服务器资源或服务器本身 - 所有这些都使用特定于服务器的 API。

作为最后的手段，您还可以声明自己的 `WebServerFactory` 组件，该组件将覆盖 Spring Boot 提供的组件。在这种情况下，您不能再依赖 `server` 命名空间中的配置属性。

78.10 向应用程序添加 Servlet，过滤器或监听器

在 servlet 堆栈应用程序中，即使用 `spring-boot-starter-web`，有两种方法可以将 `Servlet`，`Filter`，`ServletContextListener` 以及 Servlet API 支持的其他侦听器添加到应用程序中：

- 第 78.10.1 节“使用 Spring Bean 添加 Servlet，过滤器或监听器”
- 部分 78.10.2，“使用类路径扫描添加 Servlet，过滤器和监听器”

78.10.1 使用 Spring Bean 添加 Servlet，过滤器或监听器

要使用 Spring bean 添加 `Servlet`，`Filter` 或 Servlet `*Listener`，您必须为其提供`@Bean` 定义。当您想要注入配置或依赖项时，这样做非常有用。但是，您必须非常小心，它们不会导致太多其他 beans 的初始化，因为它们必须在应用程序生命周期的早期安装在容器中。（例如，让它们依赖于您的 `DataSource` 或 JPA 配置并不是一个好主意。）您可以通过在首次使用而不是初始化时懒惰地初始化 beans 来解决此类限制。

对于 `Filters` 和 `Servlets`，您还可以通过添加 `FilterRegistrationBean` 或 `ServletRegistrationBean` 来代替或添加基础组件来添加映射和初始化参数。

注意

如果在过滤器注册中未指定 `dispatcherType`，则使用 `REQUEST`。这与 Servlet 规范的默认调度程序类型一致。

与任何其他 Spring bean 一样，您可以定义 Servlet 过滤器 beans 的顺序; 请务必查看“ 名为”注册 Servlet，过滤器和监听器的部分为 Spring Beans“ ”部分。

禁用 Servlet 或过滤器的注册

正如前面所述，任何 `Servlet` 或 `Filter` beans 与自动 servlet 容器登记。要禁用特定 `Filter` 或 `Servlet` bean 的注册，请为其创建注册 bean 并将其标记为已禁用，如以下示例所示：

```
@Bean
public FilterRegistrationBean registration(MyFilter filter) {
        FilterRegistrationBean registration = new FilterRegistrationBean(filter);
        registration.setEnabled(false);
        return registration;
}
```

78.10.2 使用类路径扫描添加 Servlet，过滤器和监听器

通过使用`@ServletComponentScan` 注释`@Configuration` 类并指定包含组件的包，可以使用嵌入式 servlet 容器自动注册`@WebServlet`，`@WebFilter` 和`@WebListener` 带注释的类你想注册。默认情况下，`@ServletComponentScan` 从带注释的类的包中进行扫描。

78.11 配置访问日志记录

可以通过各自的命名空间为 Tomcat，Undertow 和 Jetty 配置访问日志。

例如，以下设置使用自定义模式在 Tomcat 上记录访问权限 。

```
server.tomcat.basedir=my-tomcat
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.pattern=%t %a "%r" %s (%D ms)
```

注意

日志的默认位置是相对于 Tomcat 基目录的 `logs` 目录。默认情况下，`logs` 目录是临时目录，因此您可能希望修复 Tomcat 的基目录或使用日志的绝对路径。在前面的示例中，日志在 `my-tomcat/logs` 中相对于应用程序的工作目录可用。

可以以类似的方式配置 Undertow 的访问日志记录，如以下示例所示：

```
server.undertow.accesslog.enabled=true
server.undertow.accesslog.pattern=%t %a "%r" %s (%D ms)
```

日志存储在相对于应用程序工作目录的 `logs` 目录中。您可以通过设置 `server.undertow.accesslog.directory` 属性来自定义此位置。

最后，Jetty 的访问日志记录也可以配置如下：

```
server.jetty.accesslog.enabled=true
server.jetty.accesslog.filename=/var/log/jetty-access.log
```

默认情况下，日志会重定向到 `System.err`。有关更多详细信息，请参阅 Jetty 文档。

78.12 在前端代理服务器后面运行

您的应用程序可能需要发送 `302` 重定向或使用绝对链接将内容呈现回自身。在代理后面运行时，调用者需要指向代理的链接，而不是托管应用程序的计算机的物理地址。通常，这种情况是通过与代理的合同来处理的，代理会添加标题以告诉后端如何构建自身链接。

如果代理添加了传统的 X-Forwarded-For 和 X-Forwarded-Proto 标头（大多数代理服务器都这样做），则应该正确呈现绝对链接，前提是 application.properties 中的 server.use-forward-headers 设置为 application.properties。

注意

如果您的应用程序在 Cloud Foundry 或 Heroku 中运行，则 server.use-forward-headers 属性默认为 true。在所有其他情况下，默认为 false。

78.12.1 自定义 Tomcat 的代理配置

如果使用 Tomcat，还可以配置用于携带“转发”信息的标头名称，如以下示例所示：

```
server.tomcat.remote-ip-header=x-your-remote-ip-header
server.tomcat.protocol-header=x-your-protocol-header
```

Tomcat 还配置了一个默认正则表达式，该表达式匹配要信任的内部代理。默认情况下，10/8，192.168/16，169.254/16 和 127/8 中的 IP 地址是可信的。您可以通过向 application.properties 添加条目来自定义阀门的配置，如以下示例所示：

```
server.tomcat.internal-proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}
```

注意

仅当使用属性文件进行配置时才需要双反斜杠。如果使用 YAML，则单个反斜杠就足够了，并且与前面示例中显示的值相等的值为 192\.168\.\d{1,3}\.\d{1,3}。

注意

您可以通过将 internal-proxies 设置为空（但在生产中不这样做）来信任所有代理。

您可以通过关闭自动关闭（为此，设置 server.use-forward-headers=false）并在 TomcatServletWebServerFactory bean 中添加新的阀门实例来完全控制 Tomcat RemoteIpValve 的配置。

78.13 使用 Tomcat 启用多个连接器

您可以将 org.apache.catalina.connector.Connector 添加到 TomcatServletWebServerFactory，这可以允许多个连接器，包括 HTTP 和 HTTPS 连接器，如以下示例所示：

```
@Bean
public ServletWebServerFactory servletContainer() {
        TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory();
        tomcat.addAdditionalTomcatConnectors(createSslConnector());
        return tomcat;
}

private Connector createSslConnector() {
        Connector connector = new
Connector("org.apache.coyote.http11.Http11NioProtocol");
        Http11NioProtocol protocol = (Http11NioProtocol)
connector.getProtocolHandler();
```

```
        try {
                File keystore = new ClassPathResource("keystore").getFile();
                File truststore = new ClassPathResource("keystore").getFile();
                connector.setScheme("https");
                connector.setSecure(true);
                connector.setPort(8443);
                protocol.setSSLEnabled(true);
                protocol.setKeystoreFile(keystore.getAbsolutePath());
                protocol.setKeystorePass("changeit");
                protocol.setTruststoreFile(truststore.getAbsolutePath());
                protocol.setTruststorePass("changeit");
                protocol.setKeyAlias("apitester");
                return connector;
        }
        catch (IOException ex) {
                throw new IllegalStateException("can't access keystore: [" +
"keystore"
                                + "] or truststore: [" + "keystore" + "]", ex);
        }
}
```

78.14 使用 Tomcat 的 LegacyCookieProcessor

默认情况下，Spring Boot 使用的嵌入式 Tomcat 不支持 Cookie 格式的“Version 0”，因此您可能会看到以下错误：

```
java.lang.IllegalArgumentException: An invalid character [32] was present in the
Cookie value
```

如果可能的话，您应该考虑将代码更新为仅存储符合以后 Cookie 规范的值。但是，如果您无法更改 cookie 的编写方式，则可以将 Tomcat 配置为使用 `LegacyCookieProcessor`。要切换到 `LegacyCookieProcessor`，请使用添加 `TomcatContextCustomizer` 的 `WebServerFactoryCustomizer` bean，如以下示例所示：

```
@Bean
public WebServerFactoryCustomizer<TomcatServletWebServerFactory>
cookieProcessorCustomizer() {
        return (factory) -> factory.addContextCustomizers(
                        (context) -> context.setCookieProcessor(new
LegacyCookieProcessor()));
}
```

78.15 使用 Undertow 启用多个侦听器

将 `UndertowBuilderCustomizer` 添加到 `UndertowServletWebServerFactory` 并向 `Builder` 添加一个侦听器，如以下示例所示：

```
@Bean
public UndertowServletWebServerFactory servletWebServerFactory() {
        UndertowServletWebServerFactory factory = new
UndertowServletWebServerFactory();
        factory.addBuilderCustomizers(new UndertowBuilderCustomizer() {

                @Override
                public void customize(Builder builder) {
                        builder.addHttpListener(8080, "0.0.0.0");
                }

        });
        return factory;
}
```

78.16 使用@ServerEndpoint 创建 WebSocket 端点

如果要在使用嵌入式容器的 Spring Boot 应用程序中使用`@ServerEndpoint`，则必须声明单个`ServerEndpointExporter` `@Bean`，如以下示例所示：

```
@Bean
public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
}
```

前面示例中显示的 bean 使用基础 WebSocket 容器注册任何`@ServerEndpoint` 带注释的 beans。当部署到独立的 servlet 容器时，此角色由 servlet 容器初始化程序执行，而不需要`ServerEndpointExporter` bean。

79. Spring MVC

Spring Boot 有许多包括 Spring MVC 的先发球员。请注意，某些启动器包含对 Spring MVC 的依赖，而不是直接包含它。本节回答有关 Spring MVC 和 Spring Boot 的常见问题。

79.1 编写 JSON REST 服务

只要 Jackson2 在类路径中，Spring Boot 应用程序中的任何 Spring `@RestController` 都应默认呈现 JSON 响应，如以下示例所示：

```
@RestController
public class MyController {

        @RequestMapping("/thing")
        public MyThing thing() {
                        return new MyThing();
        }

}
```

只要`MyThing` 可以被 Jackson2 序列化（对于普通的 POJO 或 Groovy 对象来说是真的），那么`localhost:8080/thing` 默认为它提供 JSON 表示。请注意，在浏览器中，您有时可能会看到 XML 响应，因为浏览器倾向于发送更喜欢 XML 的接受标头。

79.2 编写 XML REST 服务

如果类路径上有 Jackson XML 扩展（`jackson-dataformat-xml`），则可以使用它来呈现 XML 响应。我们用于 JSON 的前一个示例可以使用。要使用 Jackson XML 渲染器，请将以下依赖项添加到项目中：

```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

如果 Jackson 的 XML 扩展不可用，则使用 JAXB（默认情况下在 JDK 中提供），并附加要求将`MyThing` 注释为`@XmlRootElement`，如以下示例所示：

```
@XmlRootElement
public class MyThing {
        private String name;
        // .. getters and setters
}
```

要使服务器呈现 XML 而不是 JSON，您可能必须发送 `Accept: text/xml` 标头（或使用浏览器）。

79.3 自定义 Jackson ObjectMapper

Spring MVC（客户端和服务器端）使用 `HttpMessageConverters` 协商 HTTP 交换中的内容转换。如果 Jackson 在类路径上，则您已获得 `Jackson2ObjectMapperBuilder` 提供的默认转换器，其实例将自动为您配置。

`ObjectMapper`（或 Jackson XML 转换器的 `XmlMapper`）实例（默认创建）具有以下自定义属性：

- `MapperFeature.DEFAULT_VIEW_INCLUSION` 已停用
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 已停用
- `SerializationFeature.WRITE_DATES_AS_TIMESTAMPS` 已停用

Spring Boot 还具有一些功能，可以更轻松地自定义此行为。

您可以使用环境配置 `ObjectMapper` 和 `XmlMapper` 实例。Jackson 提供了一套广泛的简单开/关功能，可用于配置其处理的各个方面。这些功能在六个枚举中描述（在 Jackson 中），映射到环境中的属性：

枚举	Property
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jackson.dese e>`
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jackson.gene`
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jackson.mapp`
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jackson.pars`
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jackson.seri`
`com.fasterxml.jackson.annotation.JsonInclude.Include`	`spring.jackson.defa`

例如，要启用漂亮打印，请设置 `spring.jackson.serialization.indent_output=true`。注意，由于使用了[松弛绑定](#)，`indent_output` 的情况不必与相应的枚举常量（`INDENT_OUTPUT`}的情况相匹配。

此基于环境的配置应用于自动配置的 Jackson2ObjectMapperBuilder bean，并应用于使用构建器创建的任何映射器，包括自动配置的 ObjectMapper bean。

上下文的 Jackson2ObjectMapperBuilder 可以通过一个或多个 Jackson2ObjectMapperBuilderCustomizer beans 进行自定义。可以订购这样的定制器 beans（Boot 自己的定制器的顺序为 0），允许在 Boot 定制之前和之后应用其他定制。

任何 beans 类型 com.fasterxml.jackson.databind.Module 都会自动注册自动配置的 Jackson2ObjectMapperBuilder，并应用于它创建的任何 ObjectMapper 个实例。这为您在应用程序中添加新功能时提供了一种全局机制，用于提供自定义模块。

如果要完全替换默认的 ObjectMapper，请定义该类型的@Bean 并将其标记为@Primary，或者，如果您更喜欢基于构建器的方法，请定义 Jackson2ObjectMapperBuilder @Bean。请注意，在任何一种情况下，这样做都会禁用 ObjectMapper 的所有自动配置。

如果您提供 MappingJackson2HttpMessageConverter 类型的任何@Beans，它们将替换 MVC 配置中的默认值。此外，还提供了 HttpMessageConverters 类型的便利 bean（如果使用默认的 MVC 配置，则始终可用）。它有一些有用的方法来访问默认和用户增强的消息转换器。

有关详细信息，请参阅“ 第 79.4 节 ”，“自定义@ResponseBody 渲染 ”部分和 WebMvcAutoConfiguration 源代码。

79.4 自定义@ResponseBody 渲染

Spring 使用 HttpMessageConverters 渲染@ResponseBody（或来自@RestController 的回复）。您可以通过在 Spring Boot 上下文中添加适当类型的 beans 来提供其他转换器。如果你添加的 bean 是默认包含的类型（例如 MappingJackson2HttpMessageConverter 用于 JSON 转换），它将替换默认值。提供了 HttpMessageConverters 类型的便利 bean，如果您使用默认的 MVC 配置，它始终可用。它有一些有用的方法来访问默认和用户增强的消息转换器（例如，如果要将它们手动注入自定义 RestTemplate，它会很有用）。

与正常的 MVC 使用情况一样，您提供的任何 WebMvcConfigurer beans 也可以通过覆盖 configureMessageConverters 方法来提供转换器。但是，与普通的 MVC 不同，您只能提供所需的其他转换器（因为 Spring Boot 使用相同的机制来提供其默认值）。最后，如果您通过提供自己的@EnableWebMvc 配置选择退出 Spring Boot 默认 MVC 配置，则可以完全控制并使用 WebMvcConfigurationSupport 中的 getMessageConverters 手动完成所有操作。

有关 WebMvcAutoConfiguration 更多详细信息，请参阅 源代码。

79.5 处理多部分文件上载

Spring Boot 包含 Servlet 3 javax.servlet.http.Part API 以支持上传文件。默认情况下，Spring Boot 配置 Spring MVC，每个文件的最大大小为 1MB，单个请求中的文件数据最大为 10MB。您可以覆盖这些值，中间数据的存储位置（例如，到/tmp 目录），以及使用 MultipartProperties 类中公开的属性将数据刷新到磁盘的阈值。例如，如果要指定文件不受限制，请将 spring.servlet.multipart.max-file-size 属性设置为-1。

当您希望在 Spring MVC 控制器处理程序方法中接收多部分编码文件数据作为@RequestParam -带注释的 MultipartFile 类参数时，多部分支持非常有用。

有关 MultipartAutoConfiguration 详细信息，请参阅 源代码。

注意

建议使用容器的内置支持进行分段上传，而不是引入其他依赖项，例如 Apache Commons File Upload。

79.6 关闭 Spring MVC DispatcherServlet

默认情况下，所有内容都是从应用程序的根目录（/）提供的。如果您希望映射到其他路径，可以按如下方式配置：

```
spring.mvc.servlet.path=/acme
```

如果你有额外的 servlet，你可以声明@Bean 类型为 Servlet 或 ServletRegistrationBean，Spring Boot 将透明地注册到容器。因为 servlet 是以这种方式注册的，所以可以将它们映射到 DispatcherServlet 的子上下文而不调用它。

自己配置 DispatcherServlet 是不寻常的，但如果你真的需要这样做，还必须提供 DispatcherServletPath 类型 DispatcherServletPath，以提供自定义 DispatcherServlet 的路径。

79.7 关闭默认 MVC 配置

完全控制 MVC 配置的最简单方法是为您自己的@Configuration 提供@EnableWebMvc 注释。这样做会将所有 MVC 配置留在您的手中。

79.8 自定义 ViewResolvers

ViewResolver 是 Spring MVC 的核心组件，将@Controller 中的视图名称转换为实际的 View 实现。请注意，ViewResolvers 主要用于 UI 应用程序，而不是 REST 样式的服务（View 不用于呈现@ResponseBody）。ViewResolver 有很多实现可供选择，而 Spring 本身并不反对你应该使用哪些。另一方面，Spring Boot 为您安装一个或两个，具体取决于它在类路径和应用程序上下文中找到的内容。DispatcherServlet 使用它在应用程序上下文中找到的所有解析器，依次尝试每个解析器直到得到结果，因此，如果你添加自己的解析器，你必须知道顺序以及你的解析器在哪个位置添加。

WebMvcAutoConfiguration 在您的上下文中添加以下 ViewResolvers：

- InternalResourceViewResolver 名为'defaultViewResolver'。这个可以通过使用 DefaultServlet（包括静态资源和 JSP 页面，如果您使用它们）来查找可以呈现的物理资源。它将前缀和后缀应用于视图名称，然后在 servlet 上下文中查找具有该路径的物理资源（默认值为空，但可通过 spring.mvc.view.prefix 和 spring.mvc.view.suffix 进行外部配置）。您可以通过提供相同类型的 bean 来覆盖它。
- 一个名为'beanNameViewResolver'的 BeanNameViewResolver。这是视图解析器链中一个有用的成员，并选择与正在解析的 View 同名的任何 beans。不必覆盖或替换它。
- A ContentNegotiatingViewResolver 名为'的 ViewResolver'只如果有添加**是** 实际上 beans 类型的 View 本。这是一个“主”解析器，委托给所有其他人，并尝试找到与客户端发送的“Accept”HTTP 头匹配的内容。有一个关于 ContentNegotiatingViewResolver 的有用 博客，您可能希望学习以了解更多信息，您也可以查看源代码以获取详细信息。您可以通过定义名为“viewResolver”的 bean 来关闭自动配置的 ContentNegotiatingViewResolver。
- 如果您使用 Thymeleaf，您还有 ThymeleafViewResolver 名

为'thymeleafViewResolver'。它通过使用前缀和后缀包围视图名称来查找资源。前缀为`spring.thymeleaf.prefix`，后缀为`spring.thymeleaf.suffix`。前缀和后缀的值分别默认为“classpath：/ templates /”和“.html”。您可以通过提供相同名称的 bean 来覆盖`ThymeleafViewResolver`。
- 如果您使用 FreeMarker，您还有`FreeMarkerViewResolver`名为'freeMarkerViewResolver'。它通过用前缀和后缀包围视图名称来查找加载器路径中的资源（外部化为`spring.freemarker.templateLoaderPath`并且默认值为'classpath：/ templates /'）。前缀外部化为`spring.freemarker.prefix`，后缀外部化为`spring.freemarker.suffix`。前缀和后缀的默认值分别为空和“.ftl”。您可以通过提供相同名称的 bean 来覆盖`FreeMarkerViewResolver`。
- 如果您使用 Groovy 模板（实际上，如果`groovy-templates`在您的类路径上），您还有`GroovyMarkupViewResolver`名为'groovyMarkupViewResolver'。它通过用前缀和后缀（外部化为`spring.groovy.template.prefix`和`spring.groovy.template.suffix`）包围视图名称来查找加载器路径中的资源。前缀和后缀分别具有“classpath：/ templates /”和“.tpl”的默认值。您可以通过提供相同名称的 bean 来覆盖`GroovyMarkupViewResolver`。

有关更多详细信息，请参阅以下部分：

- `WebMvcAutoConfiguration`
- `ThymeleafAutoConfiguration`
- `FreeMarkerAutoConfiguration`
- `GroovyTemplateAutoConfiguration`

80.使用 Spring 安全性进行测试

Spring 安全性支持以特定用户身份运行测试。例如，下面代码段中的测试将使用具有`ADMIN`角色的经过身份验证的用户运行。

```
@Test
@WithMockUser(roles="ADMIN")
public void requestProtectedUrlWithUser() throws Exception {
        mvc
                .perform(get("/"))
                ...
}
```

Spring 安全性提供与 Spring MVC 测试的全面集成，这也可以在使用`@WebMvcTest`切片和`MockMvc`测试控制器时使用。

有关 Spring 安全性测试支持的其他详细信息，请参阅 Spring 安全性 参考文档）。

81. Jersey

81.1 使用 Spring 安全性保护 Jersey 端点

Spring 安全性可用于保护基于 Jersey 的 Web 应用程序，其方式与用于保护基于 Spring MVC 的 Web 应用程序的方式非常相似。但是，如果要将 Spring 安全性的方法级安全性与 Jersey 一起使用，则必须将 Jersey 配置为使用`setStatus(int)`而不是`sendError(int)`。这可以防止 Jersey 在 Spring 安全性有机会向客户端报告身份验证或授权失败之前提交响应。

`jersey.config.server.response.setStatusOverSendError`属性必须在应用程序的`ResourceConfig` bean 上设置为`true`，如以下示例所示：

```
@Component
public class JerseyConfig extends ResourceConfig {

        public JerseyConfig() {
                register(Endpoint.class);
                setProperties(Collections.singletonMap(

"jersey.config.server.response.setStatusOverSendError", true));
        }

}
```

82. HTTP 客户端

Spring Boot 提供了许多与 HTTP 客户端配合使用的启动器。本节回答与使用它们相关的问题。

82.1 配置 RestTemplate 以使用代理

如第 34.1 节“RestTemplate Customization”中所述，您可以使用 `RestTemplateCustomizer` 和 `RestTemplateBuilder` 来构建自定义的 `RestTemplate`。这是创建配置为使用代理的 `RestTemplate` 的推荐方法。

代理配置的确切详细信息取决于正在使用的基础客户端请求工厂。以下示例使用 `HttpClient` 配置 `HttpComponentsClientRequestFactory`，该代理使用除 `192.168.0.5` 以外的所有主机的代理：

```
static class ProxyCustomizer implements RestTemplateCustomizer {

        @Override
        public void customize(RestTemplate restTemplate) {
                HttpHost proxy = new HttpHost("proxy.example.com");
                HttpClient httpClient = HttpClientBuilder.create()
                                .setRoutePlanner(new DefaultProxyRoutePlanner(proxy)
{

                                        @Override
                                        public HttpHost determineProxy(HttpHost
target,
                                                        HttpRequest request,
HttpContext context)
                                                        throws HttpException {
                                                if
(target.getHostName().equals("192.168.0.5")) {
                                                        return null;
                                                }
                                                return super.determineProxy(target,
request, context);
                                        }

                                }).build();
                restTemplate.setRequestFactory(
                                new
HttpComponentsClientHttpRequestFactory(httpClient));
        }

}
```

83.记录

Spring Boot 没有强制日志记录依赖，但 Commons Logging API 除外，它通常由 Spring Framework 的 `spring-jcl` 模块提供。要使用 Logback，您需要在类路径中包含它和 `spring-jcl`。最简

单的方法是通过首发，这些都取决于 spring-boot-starter-logging。对于 Web 应用程序，您只需要 spring-boot-starter-web，因为它依赖于日志记录启动器。如果您使用 Maven，则以下依赖项会为您添加日志记录：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Boot 具有 LoggingSystem 抽象，尝试根据类路径的内容配置日志记录。如果 Logback 可用，则它是第一选择。

如果您需要对日志记录进行的唯一更改是设置各种记录器的级别，则可以使用“logging.level”前缀在 application.properties 中执行此操作，如以下示例所示：

```
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

您还可以使用“logging.file”设置要写入日志的文件的位置（除控制台外）。

要配置日志记录系统的更细粒度设置，您需要使用相关 LoggingSystem 支持的本机配置格式。默认情况下，Spring Boot 从系统的默认位置（例如用于 Logback 的 classpath:logback.xml）中选择本机配置，但您可以使用“logging.config”属性设置配置文件的位置。

83.1 配置日志记录的日志记录

如果在类路径的根目录中放置 logback.xml，则从那里（或从 logback-spring.xml 中选取它）以利用 Boot 提供的模板功能。Spring Boot 提供了一个默认的基本配置，如果要设置级别，可以包括该配置，如以下示例所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
        <include resource="org/springframework/boot/logging/logback/base.xml"/>
        <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

如果你看看 spring-boot jar 中的 base.xml，你可以看到它使用 LoggingSystem 为你创建的一些有用的系统属性：

- ${PID}：当前进程 ID。
- ${LOG_FILE}：是否在 Boot 的外部配置中设置了 logging.file。
- ${LOG_PATH}：是否在引导的外部配置中设置了 logging.path（表示日志文件所在的目录）。
- ${LOG_EXCEPTION_CONVERSION_WORD}：是否在 Boot 的外部配置中设置了 logging.exception-conversion-word。

Spring Boot 还通过使用自定义 Logback 转换器在控制台上（但不在日志文件中）提供了一些漂亮的 ANSI 颜色终端输出。有关详细信息，请参阅默认的 base.xml 配置。

如果 Groovy 在类路径上，您应该能够使用 logback.groovy 配置 Logback。如果存在，则优先考虑此设置。

83.1.1 配置仅文件输出的回溯

如果要禁用控制台日志记录并仅将输出写入文件，则需要导入 file-appender.xml 而不是

`console-appender.xml` 的自定义 `logback-spring.xml`，如以下示例所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
        <include resource="org/springframework/boot/logging/logback/defaults.xml" />
        <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-$
{java.io.tmpdir:-/tmp}}/}spring.log}"/>
        <include resource="org/springframework/boot/logging/logback/file-
appender.xml" />
        <root level="INFO">
                <appender-ref ref="FILE" />
        </root>
</configuration>
```

您还需要将 `logging.file` 添加到 `application.properties`，如以下示例所示：

```
logging.file=myapplication.log
```

83.2 配置 Log4j 进行日志记录

Spring Boot 支持 Log4j 2 进行日志记录配置（如果它在类路径上）。如果使用启动器来组装依赖项，则必须排除 Logback，然后包含 log4j 2。如果您不使用启动器，除了 Log4j 2 之外，您还需要提供（至少）`spring-jcl`。

最简单的路径可能是通过初学者，即使它需要一些与排除的摇晃。以下示例显示如何在 Maven 中设置启动器：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <exclusions>
                <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-logging</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

以下示例显示了在 Gradle 中设置启动器的一种方法：

```
dependencies {
        compile 'org.springframework.boot:spring-boot-starter-web'
        compile 'org.springframework.boot:spring-boot-starter-log4j2'
}

configurations {
        all {
                exclude group: 'org.springframework.boot', module: 'spring-boot-
starter-logging'
        }
}
```

注意

Log4j 启动器将常见日志记录要求的依赖关系聚集在一起（例如让 Tomcat 使用 `java.util.logging` 但使用 Log4j 2 配置输出）。有关详细信息，请参阅 Actuator Log4j 2 样品并查看其实际操作。

注意

要确保使用 `java.util.logging` 执行的调试日志记录路由到 Log4j 2，请通过将 `java.util.logging.manager` 系统属性设置为 `org.apache.logging.log4j.jul.LogManager` 来配置其 JDK 日志记录适配器。

83.2.1 使用 YAML 或 JSON 配置 Log4j 2

除了默认的 XML 配置格式外，Log4j 2 还支持 YAML 和 JSON 配置文件。要将 Log4j 2 配置为使用备用配置文件格式，请将相应的依赖项添加到类路径，并将配置文件命名为与所选文件格式匹配，如以下示例所示：

格式	依赖	文件名
YAML	`com.fasterxml.jackson.core:jackson-databind` `com.fasterxml.jackson.dataformat:jackson-dataformat-yaml`	`log4j2.yaml` `log4j2.yml`
JSON	`com.fasterxml.jackson.core:jackson-databind`	`log4j2.json` `log4j2.jsn`

84.数据访问

Spring Boot 包括一些处理数据源的初学者。本节回答与此相关的问题。

84.1 配置自定义数据源

要配置您自己的 `DataSource`，请在配置中定义该类型的`@Bean`。Spring Boot 在任何需要的地方重用 `DataSource`，包括数据库初始化。如果需要外部化某些设置，可以将 `DataSource` 绑定到环境中（请参阅“ 第 24.8.1 节”“第三方配置 ”）。

以下示例显示如何在 bean 中定义数据源：

```
@Bean
@ConfigurationProperties(prefix="app.datasource")
public DataSource dataSource() {
        return new FancyDataSource();
}
```

以下示例显示如何通过设置属性来定义数据源：

```
app.datasource.url=jdbc:h2:mem:mydb
app.datasource.username=sa
app.datasource.pool-size=30
```

假设您的 `FancyDataSource` 具有 URL 的常规 JavaBean 属性，用户名和池大小，这些设置在 `DataSource` 可用于其他组件之前自动绑定。也会发生常规 数据库初始化（因此 `spring.datasource.*`的相关子集仍可用于您的自定义配置）。

Spring Boot 还提供了一个名为 `DataSourceBuilder` 的实用程序构建器类，可用于创建其中一个标准数据源（如果它位于类路径中）。构建器可以根据类路径上的可用内容检测要使用的那个。它还会根据 JDBC URL 自动检测驱动程序。

以下示例显示如何使用 `DataSourceBuilder` 创建数据源：

```
@Bean
@ConfigurationProperties("app.datasource")
public DataSource dataSource() {
        return DataSourceBuilder.create().build();
}
```

要使用 `DataSource` 运行应用程序，您只需要连接信息。还可以提供特定于池的设置。检查将在运行时使用的实现以获取更多详细信息。

以下示例说明如何通过设置属性来定义 JDBC 数据源：

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

然而，有一个问题。由于未公开连接池的实际类型，因此您的自定义 `DataSource` 的元数据中不会生成任何键，并且 IDE 中没有可用的完成（因为 `DataSource` 接口不公开任何属性）。此外，如果你碰巧在类路径上有 Hikari，这个基本设置不起作用，因为 Hikari 没有 `url` 属性（但确实有 `jdbcUrl` 属性）。在这种情况下，您必须按如下方式重写配置：

```
app.datasource.jdbc-url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.maximum-pool-size=30
```

您可以通过强制连接池使用并返回专用实现而不是 `DataSource` 来解决此问题。您无法在运行时更改实现，但选项列表将是显式的。

以下示例显示如何使用 `DataSourceBuilder` 创建 `HikariDataSource`：

```
@Bean
@ConfigurationProperties("app.datasource")
public HikariDataSource dataSource() {
        return DataSourceBuilder.create().type(HikariDataSource.class).build();
}
```

您甚至可以通过利用 `DataSourceProperties` 为您做的事情来进一步发展 - 也就是说，如果没有提供 URL，则通过提供具有合理用户名和密码的默认嵌入式数据库。您可以从任何 `DataSourceProperties` 对象的状态轻松初始化 `DataSourceBuilder`，因此您还可以注入 Spring Boot 自动创建的 DataSource。但是，这会将您的配置拆分为两个名称空间：`spring.datasource` 上的 `url`，`username`，`password`，`type` 和 `driver` 以及您自定义名称空间上的其余名称空间（`app.datasource`）。为避免这种情况，您可以在自定义命名空间上重新定义自定义 `DataSourceProperties`，如以下示例所示：

```
@Bean
@Primary
@ConfigurationProperties("app.datasource")
public DataSourceProperties dataSourceProperties() {
        return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource.configuration")
```

```
public HikariDataSource dataSource(DataSourceProperties properties) {
	return properties.initializeDataSourceBuilder().type(HikariDataSource.class)
			.build();
}
```

默认情况下，此设置使您与 Spring Boot 为您执行的操作保持同步，除了选择了专用连接池（在代码中）并且其设置在 `app.datasource.configuration` 子命名空间中公开。因为 `DataSourceProperties` 正在为您处理 `url` / `jdbcUrl` 翻译，您可以按如下方式对其进行配置：

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.configuration.maximum-pool-size=30
```

小费

Spring Boot 会将 Hikari 特定的设置暴露给 `spring.datasource.hikari`。此示例使用更通用的 `configuration` 子命名空间，因为该示例不支持多个数据源实现。

注意

由于您的自定义配置选择使用 Hikari，`app.datasource.type` 无效。在实践中，构建器初始化为您可能在那里设置的任何值，然后通过调用 `.type()` 覆盖。

有关 更多详细信息，请参阅“ Spring Boot 功能”部分中的“ 部分 30.1，”配置数据源“ ”和 `DataSourceAutoConfiguration` 类。

84.2 配置两个数据源

如果需要配置多个数据源，可以应用上一节中描述的相同技巧。但是，您必须将 `DataSource` 个实例中的一个标记为`@Primary`，因为随后的各种自动配置希望能够按类型获得。

如果您创建自己的 `DataSource`，则自动配置会退回。在下面的例子中，我们提供的确切相同的功能集自动配置提供主数据源：

```
@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
	return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first.configuration")
public HikariDataSource firstDataSource() {
	return firstDataSourceProperties().initializeDataSourceBuilder()
			.type(HikariDataSource.class).build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public BasicDataSource secondDataSource() {
	return DataSourceBuilder.create().type(BasicDataSource.class).build();
}
```

小费

`firstDataSourceProperties` 必须标记为`@Primary`，以便数据库初始化程序功能使用您的副本（如果您使用初始化程序）。

这两个数据源也绑定了高级自定义。例如，您可以按如下方式配置它们：

```
app.datasource.first.url=jdbc:mysql://localhost/first
app.datasource.first.username=dbuser
app.datasource.first.password=dbpass
app.datasource.first.configuration.maximum-pool-size=30

app.datasource.second.url=jdbc:mysql://localhost/second
app.datasource.second.username=dbuser
app.datasource.second.password=dbpass
app.datasource.second.max-total=30
```

您也可以将相同的概念应用于辅助 `DataSource`，如以下示例所示：

```
@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
        return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first.configuration")
public HikariDataSource firstDataSource() {
        return firstDataSourceProperties().initializeDataSourceBuilder()
                        .type(HikariDataSource.class).build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public DataSourceProperties secondDataSourceProperties() {
        return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource.second.configuration")
public BasicDataSource secondDataSource() {
        return secondDataSourceProperties().initializeDataSourceBuilder()
                        .type(BasicDataSource.class).build();
}
```

上面的示例使用与自动配置中使用的 Spring Boot 相同的逻辑在自定义命名空间上配置两个数据源。请注意，每个 `configuration` 子命名空间都根据所选实现提供高级设置。

84.3 使用 Spring 数据存储库

Spring 数据可以创建各种风格的`@Repository` 接口的实现。只要那些`@Repositories` 包含在`@EnableAutoConfiguration` 类的同一个包（或子包）中，Spring Boot 就会处理所有这些内容。

对于许多应用程序，您只需要在类路径上放置正确的 Spring 数据依赖项（JPA 有 `spring-boot-starter-data-jpa`，Mongodb 有 `spring-boot-starter-data-mongodb`）并创建一些存储库接口来处理您的`@Entity` 对象。例子在 JPA 样本和 Mongodb 样本中。

Spring Boot 根据找到的`@EnableAutoConfiguration` 尝试猜测`@Repository` 定义的位置。要

获得更多控制，请使用`@EnableJpaRepositories`注释（来自 Spring Data JPA）。

有关 Spring 数据的更多信息，请参阅 Spring 数据项目页面。

84.4 从 Spring 配置中分离@Entity 定义

Spring Boot 根据找到的`@EnableAutoConfiguration`尝试猜测`@Entity`定义的位置。要获得更多控制，可以使用`@EntityScan`注释，如以下示例所示：

```
@Configuration
@EnableAutoConfiguration
@EntityScan(basePackageClasses=City.class)
public class Application {

        //...

}
```

84.5 配置 JPA 属性

Spring 数据 JPA 已经提供了一些独立于供应商的配置选项（例如用于 SQL 日志记录的配置选项），Spring Boot 公开了这些选项以及 Hibernate 的一些选项作为外部配置属性。根据上下文自动检测其中一些，因此您不必设置它们。

`spring.jpa.hibernate.ddl-auto`是一种特殊情况，因为根据运行时条件，它具有不同的默认值。如果使用嵌入式数据库且没有架构管理器（例如 Liquibase 或 Flyway）正在处理`DataSource`，则默认为`create-drop`。在所有其他情况下，默认为`none`。

使用的方言也会根据当前`DataSource`自动检测，但如果您想要明确并且在启动时绕过该检查，您可以自己设置`spring.jpa.database`。

注意

指定`database`会导致配置明确定义的 Hibernate 方言。有几个数据库有多个`Dialect`，这可能不适合您的需求。在这种情况下，您可以将`spring.jpa.database`设置为`default`以让 Hibernate 解决问题或通过设置`spring.jpa.database-platform`属性来设置方言。

以下示例中显示了最常用的设置选项：

```
spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhysicalNamingStrategy
spring.jpa.show-sql=true
```

此外，`spring.jpa.properties.*`中的所有属性在创建本地`EntityManagerFactory`时作为普通 JPA 属性（带有前缀剥离）传递。

小费

如果您需要对 Hibernate 属性应用高级自定义，请考虑注册将在创建`EntityManagerFactory`之前调用的`HibernatePropertiesCustomizer` bean。这优先于自动配置应用的任何内容。

84.6 配置 Hibernate 命名策略

Hibernate 使用两种不同的命名策略将名称从对象模型映射到相应的数据库名称。可以通过分别设置 `spring.jpa.hibernate.naming.physical-strategy` 和 `spring.jpa.hibernate.naming.implicit-strategy` 属性来配置物理策略实现和隐式策略实现的完全限定类名。或者，如果应用程序上下文中有 `ImplicitNamingStrategy` 或 `PhysicalNamingStrategy` beans，则 Hibernate 将自动配置为使用它们。

默认情况下，Spring Boot 使用 `SpringPhysicalNamingStrategy` 配置物理命名策略。这个实现提供了与 Hibernate 4 相同的表结构：所有点都被下划线替换，并且驼峰外壳也被下划线替换。默认情况下，所有表名都以小写形式生成，但如果您的架构需要，则可以覆盖该标志。

例如，`TelephoneNumber` 实体映射到 `telephone_number` 表。

如果您更喜欢使用 Hibernate 5 的默认设置，请设置以下属性：

```
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

或者，您可以配置以下 bean：

```
@Bean
public PhysicalNamingStrategy physicalNamingStrategy() {
        return new PhysicalNamingStrategyStandardImpl();
}
```

见 `HibernateJpaAutoConfiguration` 和 `JpaBaseConfiguration` 更多的细节。

84.7 配置 Hibernate 二级缓存

可以为一系列缓存提供程序配置 Hibernate 二级缓存。不是将 Hibernate 配置为再次查找缓存提供程序，最好尽可能提供上下文中可用的那个。

如果你正在使用 JCache，这很容易。首先，确保类路径上有 `org.hibernate:hibernate-jcache` 可用。然后，添加 `HibernatePropertiesCustomizer` bean，如以下示例所示：

```
@Configuration
public class HibernateSecondLevelCacheExample {

        @Bean
        public HibernatePropertiesCustomizer hibernateSecondLevelCacheCustomizer(
                        JCacheCacheManager cacheManager) {
                return (properties) -> properties.put(ConfigSettings.CACHE_MANAGER,
                                cacheManager.getCacheManager());

        }

}
```

此自定义程序将配置 Hibernate 使用与应用程序使用的 `CacheManager` 相同的 `CacheManager`。也可以使用单独的 `CacheManager` 实例。有关详细信息，请参阅 Hibernate 用户指南。

84.8 在 Hibernate 组件中使用依赖注入

默认情况下，Spring Boot 注册使用 `BeanFactory` 的 `BeanContainer` 实现，以便转换器和实体侦听器可以使用常规依赖项注入。

您可以通过注册删除或更改 `hibernate.resource.beans.container` 属性的 `HibernatePropertiesCustomizer` 来禁用或调整此行为。

84.9 使用自定义 EntityManagerFactory

要完全控制 `EntityManagerFactory` 的配置，您需要添加名为'entityManagerFactory'的 `@Bean`。Spring Boot auto-configuration 在存在 bean 类型的情况下关闭其实体管理器。

84.10 使用两个 EntityManagers

即使默认 `EntityManagerFactory` 工作正常，您也需要定义一个新的。否则，该类型的第二个 bean 的存在将关闭默认值。为方便起见，您可以使用 Spring Boot 提供的方便 `EntityManagerBuilder`。或者，您可以直接从 Spring ORM 获取 `LocalContainerEntityManagerFactoryBean`，如以下示例所示：

```
// add two data sources configured as above

@Bean
public LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(
                EntityManagerFactoryBuilder builder) {
        return builder
                        .dataSource(customerDataSource())
                        .packages(Customer.class)
                        .persistenceUnit("customers")
                        .build();
}

@Bean
public LocalContainerEntityManagerFactoryBean orderEntityManagerFactory(
                EntityManagerFactoryBuilder builder) {
        return builder
                        .dataSource(orderDataSource())
                        .packages(Order.class)
                        .persistenceUnit("orders")
                        .build();
}
```

上面的配置几乎可以单独使用。要完成图片，您还需要为两个 `EntityManagers` 配置 `TransactionManagers`。如果您将其中一个标记为`@Primary`，则可以通过 Spring Boot 中的默认 `JpaTransactionManager` 来获取它。另一个必须明确地注入新实例。或者，您可以使用跨越两者的 JTA 事务管理器。

如果您使用 Spring 数据，则需要相应地配置`@EnableJpaRepositories`，如以下示例所示：

```
@Configuration
@EnableJpaRepositories(basePackageClasses = Customer.class,
                entityManagerFactoryRef = "customerEntityManagerFactory")
public class CustomerConfiguration {
        ...
}

@Configuration
@EnableJpaRepositories(basePackageClasses = Order.class,
                entityManagerFactoryRef = "orderEntityManagerFactory")
public class OrderConfiguration {
        ...
}
```

84.11 使用繁体 `persistence.xml` 文件

Spring Boot 默认情况下不会搜索或使用 `META-INF/persistence.xml`。如果您更喜欢使用传统的 `persistence.xml`，则需要定义自己的 `LocalEntityManagerFactoryBean` 类型 `LocalEntityManagerFactoryBean`（ID 为“entityManagerFactory”）并在其中设置持久性单元名称。

请参阅 `JpaBaseConfiguration` 默认设置。

84.12 使用 Spring 数据 JPA 和 Mongo 存储库

Spring 数据 JPA 和 Spring Data Mongo 都可以自动为您创建 `Repository` 实现。如果它们都存在于类路径中，则可能需要执行一些额外配置以告知 Spring Boot 要创建哪些存储库。最明确的方法是使用标准的 Spring 数据`@EnableJpaRepositories` 和`@EnableMongoRepositories` 注释，并提供 `Repository` 接口的位置。

还有标志（`spring.data.*.repositories.enabled` 和 `spring.data.*.repositories.type`）可用于在外部配置中打开和关闭自动配置的存储库。这样做很有用，例如，如果您想关闭 Mongo 存储库并仍然使用自动配置的 `MongoTemplate`。

其他自动配置的 Spring 数据存储库类型（Elasticsearch，Solr 等）存在相同的障碍和相同的功能。要使用它们，请相应地更改注释和标志的名称。

84.13 自定义 Spring 数据的 Web 支持

Spring 数据提供 Web 支持，简化了 Web 应用程序中 Spring 数据存储库的使用。Spring Boot 提供 `spring.data.web` 命名空间中的属性以自定义其配置。请注意，如果您使用的是 Spring Data REST，则必须使用 `spring.data.rest` 命名空间中的属性。

84.14 将 Spring 数据存储库公开为 REST 端点

Spring 数据 REST 可以为您提供 `Repository` 实现作为 REST 端点，前提是已为应用程序启用了 Spring MVC。

Spring Boot 公开了一组自定义的有用属性（来自 `spring.data.rest` 命名空间）`RepositoryRestConfiguration`。如果您需要提供其他自定义，则应使用 `RepositoryRestConfigurer` bean。

注意

如果您未在自定义 `RepositoryRestConfigurer` 上指定任何订单，则会在内部使用 Spring Boot 之后运行。如果需要指定订单，请确保订单高于 0。

84.15 配置 JPA 使用的组件

如果要配置 JPA 使用的组件，则需要确保在 JPA 之前初始化组件。组件自动配置后，Spring Boot 会为您处理此问题。例如，当 Flyway 自动配置时，Hibernate 配置为依赖 Flyway，以便 Flyway 有机会在 Hibernate 尝试使用它之前初始化数据库。

如果您自己配置组件，则可以使用 `EntityManagerFactoryDependsOnPostProcessor` 子类作为设置必要依赖项的便捷方法。例如，如果您将 Hibernate Search 与 Elasticsearch 一起用作其索引管理器，则必须将任何 `EntityManagerFactory` beans 配置为依赖于 `elasticsearchClient` bean，如以下示例所示：

```
/**
 * {@link EntityManagerFactoryDependsOnPostProcessor} that ensures that
 * {@link EntityManagerFactory} beans depend on the {@code elasticsearchClient} bean.
 */
@Configuration
static class ElasticsearchJpaDependencyConfiguration
                extends EntityManagerFactoryDependsOnPostProcessor {

        ElasticsearchJpaDependencyConfiguration() {
                super("elasticsearchClient");
        }

}
```

84.16 使用两个 DataSource 配置 jOOQ

如果您需要将 jOOQ 用于多个数据源，则应为每个数据源创建自己的 `DSLContext`。 有关更多详细信息，请参阅 JooqAutoConfiguration。

小费

特别是，`JooqExceptionTranslator` 和 `SpringTransactionProvider` 可以重复使用，以提供与单个 `DataSource` 自动配置相似的功能。

85.数据库初始化

可以使用不同的方式初始化 SQL 数据库，具体取决于堆栈的内容。当然，如果数据库是一个单独的进程，您也可以手动执行此操作。

85.1 使用 JPA 初始化数据库

JPA 具有 DDL 生成功能，可以将这些功能设置为在数据库启动时运行。这是通过两个外部属性控制的：

- `spring.jpa.generate-ddl`（布尔值）打开和关闭该功能，与供应商无关。
- `spring.jpa.hibernate.ddl-auto`（枚举）是一种 Hibernate 功能，它以更细粒度的方式控制行为。本指南后面将详细介绍此功能。

85.2 使用 Hibernate 初始化数据库

您可以显式设置 `spring.jpa.hibernate.ddl-auto`，标准 Hibernate 属性值为 `none`，`validate`，`update`，`create` 和 `create-drop`。Spring Boot 根据是否认为您的数据库是嵌入式的，为您选择默认值。如果未检测到架构管理器，则默认为 `create-drop`;在所有其他情况下，默认为 `none`。通过查看 `Connection` 类型来检测嵌入式数据库。`hsqldb`，`h2` 和 `derby` 是嵌入的，而其他则不是。从内存切换到“真实”数据库时要小心，不要假设新平台中存在表和数据。您必须显式设置 `ddl-auto` 或使用其他机制之一来初始化数据库。

注意

您可以通过启用 `org.hibernate.SQL` 记录器来输出架构创建。如果启用调试模式，则会自动完成此 操作。

此外，如果 Hibernate 从头开始创建模式（即，如果 `ddl-auto` 属性设置为 `create` 或 `create-drop`，则会在启动时执行类路径根目录中名为 `import.sql` 的文件）。这对于演示和测试很有用，如果你小心，但可能不是你想要在生产中的类路径上。它是一个 Hibernate 功能（与 Spring 无关）。

85.3 初始化数据库

Spring Boot 可以自动创建 `DataSource` 的模式（DDL 脚本）并初始化它（DML 脚本）。它从标准根类路径位置加载 SQL：`schema.sql` 和 `data.sql`。此外，Spring Boot 处理 `schema-${platform}.sql` 和 `data-${platform}.sql` 文件（如果存在），其中 `platform` 是 `spring.datasource.platform` 的值。这允许您在必要时切换到特定于数据库的脚本。例如，您可以选择将其设置为数据库的供应商名称（`hsqldb`，`h2`，`oracle`，`mysql`，`postgresql` 等等）。

注意

Spring Boot 自动创建嵌入式 `DataSource` 的架构。可以使用 `spring.datasource.initialization-mode` 属性自定义此行为。例如，如果您想要始终初始化 `DataSource`，无论其类型如何：

```
spring.datasource.initialization-mode=always
```

默认情况下，Spring Boot 启用 Spring JDBC 初始化程序的快速失败功能。这意味着，如果脚本导致异常，则应用程序无法启动。您可以通过设置 `spring.datasource.continue-on-error` 来调整该行为。

注意

在基于 JPA 的应用程序中，您可以选择让 Hibernate 创建架构或使用 `schema.sql`，但您不能同时执行这两项操作。如果您使用 `schema.sql`，请务必禁用 `spring.jpa.hibernate.ddl-auto`。

85.4 初始化 Spring 批处理数据库

如果您使用 Spring Batch，它会预先打包用于大多数流行数据库平台的 SQL 初始化脚本。Spring Boot 可以检测您的数据库类型并在启动时执行这些脚本。如果使用嵌入式数据库，默认情况下会发生这种情况。您还可以为任何数据库类型启用它，如以下示例所示：

```
spring.batch.initialize-schema=always
```

您还可以通过设置 `spring.batch.initialize-schema=never` 显式关闭初始化。

85.5 使用更高级别的数据库迁移工具

Spring Boot 支持两种更高级别的迁移工具：迁飞 和 Liquibase。

85.5.1 在启动时执行 Flyway 数据库迁移

要在启动时自动运行 Flyway 数据库迁移，请将 `org.flywaydb:flyway-core` 添加到类路径中。

迁移是 `V<VERSION>__<NAME>.sql` 形式的脚本（`<VERSION>`是下划线分隔的版本，例如'1'或'2_1'）。默认情况下，它们位于名为 `classpath:db/migration` 的文件夹中，但您可以通过设置 `spring.flyway.locations` 来修改该位置。这是一个以逗号分隔的列表，其中包含一个或多个 `classpath:`或 `filesystem:`位置。例如，以下配置将在默认类路径位置和`/opt/migration` 目录中搜索脚本：

```
spring.flyway.locations=classpath:db/migration,filesystem:/opt/migration
```

您还可以添加特殊的`{vendor}`占位符以使用特定于供应商的脚本。假设如下：

```
spring.flyway.locations=classpath:db/migration/{vendor}
```

前面的配置不是使用 `db/migration`，而是根据数据库的类型设置要使用的文件夹（例如 MySQL 的 `db/migration/mysql`）。支持的数据库列表可在以下位置找到 `DatabaseDriver`。

`FlywayProperties` 提供大部分 Flyway 的设置和一小组其他属性，可用于禁用迁移或关闭位置检查。如果您需要更多控制配置，请考虑注册 `FlywayConfigurationCustomizer` bean。

Spring Boot 调用 `Flyway.migrate()`来执行数据库迁移。如果您想要更多控制权，请提供实施的`@Bean` `FlywayMigrationStrategy`。

Flyway 支持 SQL 和 Java 回调。要使用基于 SQL 的回调，请将回调脚本放在 `classpath:db/migration` 文件夹中。要使用基于 Java 的回调，请创建一个或多个实现 `Callback` 的 beans。任何此类 beans 都会自动注册 `Flyway`。可以使用`@Order` 或实施 `Ordered` 订购。也可以检测到实现已弃用的 `FlywayCallback` 接口的 Beans，但它们不能与 `Callback` beans 一起使用。

默认情况下，Flyway 会在您的上下文中自动装配（`@Primary`）`DataSource` 并将其用于迁移。如果您想使用其他 `DataSource`，则可以创建一个`@Bean` 并将其标记为 `@FlywayDataSource`。如果您这样做并想要两个数据源，请记住创建另一个数据源并将其标记为`@Primary`。或者，您可以通过在外部属性中设置 `spring.flyway.[url,user,password]`来使用 Flyway 的原生 `DataSource`。设置 `spring.flyway.url` 或 `spring.flyway.user` 足以使 Flyway 使用自己的 `DataSource`。如果未设置三个属性中的任何一个，将使用其等效 `spring.datasource` 属性的值。

有一个 Flyway 示例，您可以看到如何设置。

您还可以使用 Flyway 为特定方案提供数据。例如，您可以在 `src/test/resources` 中放置特定于测试的迁移，并且只有在应用程序启动测试时才会运行它们。此外，您可以使用特定于配置文件的配置来自定义 `spring.flyway.locations`，以便某些迁移仅在特定配置文件处于活动状态时运行。例如，在 `application-dev.properties` 中，您可以指定以下设置：

```
spring.flyway.locations=classpath:/db/migration,classpath:/dev/db/migration
```

使用该设置，`dev/db/migration` 中的迁移仅在 `dev` 配置文件处于活动状态时运行。

85.5.2 在启动时执行 Liquibase 数据库迁移

要在启动时自动运行 Liquibase 数据库迁移，请将 `org.liquibase:liquibase-core` 添加到

类路径中。

默认情况下，主更改日志是从 `db/changelog/db.changelog-master.yaml` 读取的，但您可以通过设置 `spring.liquibase.change-log` 来更改位置。除了 YAML，Liquibase 还支持 JSON，XML 和 SQL 更改日志格式。

默认情况下，Liquibase 会在您的上下文中自动装配（`@Primary`）`DataSource` 并将其用于迁移。如果您需要使用其他 `DataSource`，则可以创建一个`@Bean` 并将其标记为 `@LiquibaseDataSource`。如果您这样做并且想要两个数据源，请记住创建另一个数据源并将其标记为`@Primary`。或者，您可以通过在外部属性中设置 `spring.liquibase.[url,user,password]`来使用 Liquibase 的本机 `DataSource`。设置 `spring.liquibase.url` 或 `spring.liquibase.user` 足以使 Liquibase 使用自己的 `DataSource`。如果未设置三个属性中的任何一个，将使用其等效 `spring.datasource` 属性的值。

`LiquibaseProperties` 有关可用设置的详细信息，请参阅 上下文，默认架构等。

有一个 Liquibase 样本，以便您可以看到如何设置。

86.消息传递

Spring Boot 提供了许多包含消息传递的初学者。本节回答使用 Spring Boot 消息传递产生的问题。

86.1 禁用事务处理的 JMS 会话

如果您的 JMS 代理不支持事务会话，则必须完全禁用事务支持。如果您创建自己的 `JmsListenerContainerFactory`，则无需执行任何操作，因为默认情况下无法进行交易。如果要使用 `DefaultJmsListenerContainerFactoryConfigurer` 重用 Spring Boot 的默认值，可以禁用事务处理会话，如下所示：

```
@Bean
public DefaultJmsListenerContainerFactory jmsListenerContainerFactory(
                ConnectionFactory connectionFactory,
                DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory listenerFactory =
                        new DefaultJmsListenerContainerFactory();
        configurer.configure(listenerFactory, connectionFactory);
        listenerFactory.setTransactionManager(null);
        listenerFactory.setSessionTransacted(false);
        return listenerFactory;
}
```

上面的示例将覆盖默认工厂，并且应该应用于应用程序定义的任何其他工厂（如果有）。

87.批量申请

本节回答了使用 Spring 批处理 Spring Boot 时出现的问题。

注意

默认情况下，批处理应用程序需要 `DataSource` 来存储作业详细信息。如果你想偏离它，你需要实现 `BatchConfigurer`。有关详细信息，请参阅 `@EnableBatchProcessing` 的 Javadoc。

有关 Spring Batch 的更多信息，请参阅 Spring 批处理项目页面。

87.1 在启动时执行 Spring 批处理作业

Spring 通过在上下文中的某处添加@EnableBatchProcessing（来自 Spring 批处理）来启用批量自动配置。

默认情况下，它在启动时在应用程序上下文中执行**所有** Jobs（ 有关详细信息，请参阅 JobLauncherCommandLineRunner）。您可以通过指定 spring.batch.job.names（以逗号分隔的作业名称模式列表）缩小到特定作业或作业的范围。

在命令行上指定作业参数

与在 Environment 中设置属性的命令行选项参数 （即以--开头，例如--my-property=value）不同，必须在命令行上指定作业参数而不使用破折号（例如 jobParam=value ）。

如果应用程序上下文包含 JobRegistry，则 spring.batch.job.names 中的作业将在注册表中查找，而不是从上下文中自动装配。这是一个具有更复杂系统的常见模式，其中多个作业在子上下文中定义并集中注册。

有关 更多详细信息，请参阅 BatchAutoConfiguration 和 @EnableBatchProcessing。

88.执行器

Spring Boot 包括 Spring Boot Actuator。本节回答了其使用中经常出现的问题。

88.1 更改执行器端点的 HTTP 端口或地址

在独立应用程序中，Actuator HTTP 端口默认与主 HTTP 端口相同。要使应用程序侦听其他端口，请设置外部属性：management.server.port。要侦听完全不同的网络地址（例如，当您有用于管理的内部网络和用于用户应用程序的外部网络时），您还可以将 management.server.address 设置为服务器能够绑定的有效 IP 地址。

有关更多详细信息，请参阅“生产就绪功能”部分中的 ManagementServerProperties 源代码和“ 第 54.2 节 ”，“自定义管理服务器端口”“。

88.2 自定义'whitelabel'错误页面

Spring Boot 如果您遇到服务器错误，则会在浏览器客户端中安装“whitelabel”错误页面（使用 JSON 和其他媒体类型的计算机客户端应该看到具有正确错误代码的合理响应）。

注意

设置 server.error.whitelabel.enabled=false 以关闭默认错误页面。这样做会恢复您正在使用的 servlet 容器的默认值。请注意，Spring Boot 仍尝试解决错误视图，因此您应该添加自己的错误页面，而不是完全禁用它。

使用您自己的错误页面覆盖错误页面取决于您使用的模板技术。例如，如果您使用 Thymeleaf，则可以添加 error.html 模板。如果您使用 FreeMarker，则可以添加 error.ftl 模板。通常，

您需要View，其名称为error或@Controller处理/error路径。除非您替换了某些默认配置，否则您应该在ApplicationContext中找到BeanNameViewResolver，因此名为error的@Bean将是一种简单的方法。有关ErrorMvcAutoConfiguration更多选项，请参阅

有关如何在servlet容器中注册处理程序的详细信息，另请参阅“ 错误处理 ”一节。

88.3 消除敏感价值

env和configprops端点返回的信息可能有些敏感，因此默认情况下将匹配特定模式的密钥进行清理（即它们的值由******替换）。

Spring Boot使用这些密钥的合理默认值：例如，任何以“password”，“secret”，“key”或“token”结尾的密钥都将被清理。也可以使用正则表达式，例如*credentials.*来清理任何将单词credentials作为键的一部分的键。

可以使用management.endpoint.env.keys-to-sanitize和management.endpoint.configprops.keys-to-sanitize分别自定义要使用的模式。

89.安全

本节介绍使用Spring Boot时有关安全性的问题，包括使用Spring安全性Spring Boot时出现的问题。

有关Spring安全性的更多信息，请参阅Spring安全项目页面。

89.1 关闭Spring Boot安全配置

如果您在应用程序中使用WebSecurityConfigurerAdapter定义@Configuration，则会关闭Spring Boot中的默认Webapp安全设置。

89.2 更改UserDetailsService和添加用户帐户

如果您提供的@Bean类型为AuthenticationManager，AuthenticationProvider或UserDetailsService，则不会创建InMemoryUserDetailsManager的默认@Bean，因此您拥有{的完整功能集15 /}安全可用（例如各种身份验证选项）。各种身份验证选项）。

添加用户帐户的最简单方法是提供您自己的UserDetailsService bean。

89.3 在代理服务器后面运行时启用HTTPS

确保所有主要端点仅通过HTTPS可用是任何应用程序的重要工作。如果你使用Tomcat作为servlet容器，那么Spring Boot会自动添加Tomcat自己的RemoteIpValve，如果它检测到一些环境设置，你应该能够依靠HttpServletRequest来报告它是否安全（甚至是处理真正SSL终止的代理服务器的下游）。标准行为取决于某些请求标头（x-forwarded-for和x-forwarded-proto）的存在与否，其名称是常规的，因此它应该适用于大多数前端代理。您可以通过向application.properties添加一些条目来打开阀门，如以下示例所示：

```
server.tomcat.remote-ip-header=x-forwarded-for
server.tomcat.protocol-header=x-forwarded-proto
```

（这些属性中的任何一个的存在都会打开阀门。或者，您可以通过添加

TomcatServletWebServerFactory bean 来添加 RemoteIpValve。）

要将 Spring 安全性配置为要求所有（或某些）请求的安全通道，请考虑添加自己的 WebSecurityConfigurerAdapter，以添加以下 HttpSecurity 配置：

```
@Configuration
public class SslWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                // Customize the application security
                http.requiresChannel().anyRequest().requiresSecure();
        }

}
```

90.热插拔

Spring Boot 支持热交换。本节回答有关其工作原理的问题。

90.1 重新加载静态内容

热重新加载有几种选择。推荐的方法是使用 spring-boot-devtools，因为它提供了额外的开发时间功能，例如支持快速应用程序重启和 LiveReload 以及合理的开发时配置（例如模板缓存）。Devtools 通过监视类路径进行更改来工作。这意味着必须“构建”静态资源更改才能使更改生效。默认情况下，当您保存更改时，这会在 Eclipse 中自动发生。在 IntelliJ IDEA 中，Make Project 命令会触发必要的构建。由于 默认的重新启动排除，对静态资源的更改不会触发应用程序的重新启动。但是，它们会触发实时重新加载。

或者，在 IDE 中运行（特别是在调试时）是进行开发的好方法（所有现代 IDE 都允许重新加载静态资源，并且通常还允许热插拔 Java 类更改）。

最后，可以配置 Maven 和 Gradle 插件（请参阅 addResources 属性）以支持从命令行运行，并直接从源重新加载静态文件。如果您使用更高级别的工具编写该代码，则可以将其与外部 css / js 编译器进程一起使用。

90.2 重新加载模板而不重新启动容器

Spring Boot 支持的大多数模板技术都包括禁用缓存的配置选项（本文档后面会介绍）。如果使用 spring-boot-devtools 模块， 则会在开发时自动为您配置这些属性 。

90.2.1 Thymeleaf 模板

如果您使用 Thymeleaf，请将 spring.thymeleaf.cache 设置为 false。查看 ThymeleafAutoConfiguration 其他 Thymeleaf 自定义选项。

90.2.2 FreeMarker 模板

如果您使用 FreeMarker，请将 spring.freemarker.cache 设置为 false。有关 FreeMarkerAutoConfiguration 其他 FreeMarker 自定义选项，请参阅 。

90.2.3 Groovy 模板

如果您使用 Groovy 模板，请将 spring.groovy.template.cache 设置为 false。请参阅

`GroovyTemplateAutoConfiguration` 其他 Groovy 自定义选项。

90.3 快速应用程序重新启动

`spring-boot-devtools` 模块包括对自动应用程序重启的支持。虽然没有像 JRebel 这样的技术那么快， 但它通常比“冷启动”快得多。在调查本文档后面讨论的一些更复杂的重载选项之前，您应该尝试一下。

有关更多详细信息，请参阅第 20 章“ 开发人员工具”部分。

90.4 重新加载 Java 类而不重新启动容器

许多现代 IDE（Eclipse，IDEA 和其他）支持字节码的热交换。因此，如果您进行的更改不会影响类或方法签名，则应该干净地重新加载，不会产生任何副作用。

91.建立

Spring Boot 包括 Maven 和 Gradle 的构建插件。本节回答有关这些插件的常见问题。

91.1 生成构建信息

Maven 插件和 Gradle 插件都允许生成包含项目的坐标，名称和版本的构建信息。插件还可以配置为通过配置添加其他属性。当存在这样的文件时，Spring Boot 会自动配置 `BuildProperties` bean。

要使用 Maven 生成构建信息，请为 `build-info` 目标添加执行，如以下示例所示：

```
<build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                        <version>2.1.1.RELEASE</version>
                        <executions>
                                <execution>
                                        <goals>
                                                <goal>build-info</goal>
                                        </goals>
                                </execution>
                        </executions>
                </plugin>
        </plugins>
</build>
```

小费

有关详细信息，请参阅 Spring Boot Maven 插件文档 。

以下示例对 Gradle 执行相同的操作：

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

小费

有关详细信息，请参阅 Spring Boot Gradle 插件文档。

91.2 生成 Git 信息

Maven 和 Gradle 都允许生成 `git.properties` 文件，其中包含有关项目构建时 `git` 源代码存储库状态的信息。

对于 Maven 用户，`spring-boot-starter-parent` POM 包含一个预先配置的插件，用于生成 `git.properties` 文件。要使用它，请将以下声明添加到 POM：

```
<build>
        <plugins>
                <plugin>
                        <groupId>pl.project13.maven</groupId>
                        <artifactId>git-commit-id-plugin</artifactId>
                </plugin>
        </plugins>
</build>
```

Gradle 用户可以通过使用 `gradle-git-properties` 插件获得相同的结果 ，如以下示例所示：

```
plugins {
        id "com.gorylenko.gradle-git-properties" version "1.5.1"
}
```

小费

`git.properties` 中的提交时间应符合以下格式：`yyyy-MM-dd'T'HH:mm:ssZ`。这是上面列出的两个插件的默认格式。使用此格式可以将时间解析为 `Date` 及其格式，在序列化为 JSON 时，由 Jackson 的日期序列化配置设置控制。

91.3 自定义依赖版本

如果您使用直接或间接从 `spring-boot-dependencies` 继承的 Maven 版本（例如，`spring-boot-starter-parent`）但您想要覆盖特定的第三方依赖项，则可以添加适当的 `<properties>`元素。浏览 `spring-boot-dependencies` POM 以获取完整的属性列表。例如，要选择不同的 `slf4j` 版本，您需要添加以下属性：

```
<properties>
        <slf4j.version>1.7.5<slf4j.version>
</properties>
```

注意

这样做只适用于 Maven 项目（直接或间接）从 `spring-boot-dependencies` 继承的情况。如果您使用`<scope>import</scope>` 在自己的 `dependencyManagement` 部分添加了 `spring-boot-dependencies`，则必须自行重新定义工件，而不是覆盖该属性。

警告

每个 Spring Boot 版本都是针对这组特定的第三方依赖项进行设计和测试的。覆盖版本可能会导致兼容性问题。

要覆盖 Gradle 中的依赖项版本，请参阅 Gradle 插件文档的此部分。

91.4 使用 Maven 创建可执行 JAR

`spring-boot-maven-plugin` 可用于创建可执行的“胖”JAR。如果你使用 `spring-boot-starter-parent` POM，你可以声明插件，你的罐子重新包装如下：

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

如果您不使用父 POM，您仍然可以使用该插件。但是，您必须另外添加`<executions>`部分，如下所示：

```
<build>
	<plugins>
		<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>
	</plugins>
</build>
```

有关完整的使用详细信息，请参阅插件文档。

91.5 使用 Spring Boot 应用程序作为依赖关系

与 war 文件一样，Spring Boot 应用程序不能用作依赖项。如果您的应用程序包含要与其他项目共享的类，则建议的方法是将该代码移动到单独的模块中。然后，您的应用程序和其他项目可以依赖单独的模块。

如果您无法按照上面的建议重新安排代码，则必须配置 Spring Boot 的 Maven 和 Gradle 插件以生成适合用作依赖项的单独工件。可执行存档不能用作依赖项，因为 可执行 jar 格式在 `BOOT-INF/classes` 中打包应用程序类。这意味着当可执行 jar 用作依赖项时，无法找到它们。

要生成两个可以用作依赖项的工件和一个可执行的工件，必须指定一个分类器。此分类器应用于可执行归档的名称，保留默认归档以用作依赖项。

要在 Maven 中配置 `exec` 的分类器，您可以使用以下配置：

```
<build>
	<plugins>
		<plugin>
```

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

91.6 可执行 jar 运行时提取特定库

可执行 jar 中的大多数嵌套库不需要解压缩才能运行。但是，某些库可能存在问题。例如，JRuby 包含自己的嵌套 jar 支持，它假设 `jruby-complete.jar` 总是直接作为文件直接使用。

要处理任何有问题的库，您可以标记在可执行 jar 首次运行时应该自动解压缩特定的嵌套 jar。这些嵌套的 jar 被写在 `java.io.tmpdir` 系统属性标识的临时目录下。

警告

应注意确保您的操作系统已配置，以便在应用程序仍在运行时不会删除已解压缩到临时目录的 jar。

例如，要指示应使用 Maven 插件标记 JRuby 以进行解包，您将添加以下配置：

```
<build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                        <configuration>
                                <requiresUnpack>
                                        <dependency>
                                                <groupId>org.jruby</groupId>
                                                <artifactId>jruby-
complete</artifactId>
                                        </dependency>
                                </requiresUnpack>
                        </configuration>
                </plugin>
        </plugins>
</build>
```

91.7 使用排除项创建不可执行的 JAR

通常，如果您将可执行文件和非可执行 jar 作为两个单独的构建产品，则可执行版本具有库 jar 中不需要的其他配置文件。例如，`application.yml` 配置文件可能会从非可执行 JAR 中排除。

在 Maven 中，可执行 jar 必须是主工件，您可以为库添加一个分类 jar，如下所示：

```
<build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
                <plugin>
                        <artifactId>maven-jar-plugin</artifactId>
                        <executions>
                                <execution>
                                        <id>lib</id>
                                        <phase>package</phase>
```

```
                                        <goals>
                                                <goal>jar</goal>
                                        </goals>
                                        <configuration>
                                                <classifier>lib</classifier>
                                                <excludes>

<exclude>application.yml</exclude>
                                                </excludes>
                                        </configuration>
                                </execution>
                        </executions>
                </plugin>
        </plugins>
</build>
```

91.8 远程调试 Spring Boot 应用程序以 Maven 开头

要将远程调试器附加到以 Maven 启动的 Spring Boot 应用程序，您可以使用 maven 插件的 `jvmArguments` 属性。

有关详细信息，请参阅此示例。

91.9 从 Ant 构建可执行文件而不使用 `spring-boot-antlib`

要使用 Ant 构建，您需要获取依赖项，编译，然后创建 jar 或 war 存档。要使其可执行，您可以使用 `spring-boot-antlib` 模块，也可以按照以下说明操作：

1. 如果要构建 jar，请将应用程序的类和资源打包到嵌套的 `BOOT-INF/classes` 目录中。如果要构建 war，请像往常一样将应用程序的类打包在嵌套的 `WEB-INF/classes` 目录中。
2. 在 jar 的嵌套 `BOOT-INF/lib` 目录中添加运行时依赖项，或在战争中添加 `WEB-INF/lib`。切记**不要**压缩存档中的条目。
3. 将 `provided`（嵌入式容器）依赖项添加到 jar 的嵌套 `BOOT-INF/lib` 目录或战争的 `WEB-INF/lib-provided`。切记**不要**压缩存档中的条目。
4. 在归档的根目录中添加 `spring-boot-loader` 类（以便 `Main-Class` 可用）。
5. 使用适当的启动程序（例如 `JarLauncher` 作为 jar 文件）作为清单中的 `Main-Class` 属性，并指定其作为清单条目所需的其他属性 - 主要是通过设置 `Start-Class` 属性。

以下示例显示如何使用 Ant 构建可执行存档：

```
<target name="build" depends="compile">
        <jar destfile="target/${ant.project.name}-${spring-boot.version}.jar"
compress="false">
                <mappedresources>
                        <fileset dir="target/classes" />
                        <globmapper from="*" to="BOOT-INF/classes/*"/>
                </mappedresources>
                <mappedresources>
                        <fileset dir="src/main/resources" erroronmissingdir="false"/>
                        <globmapper from="*" to="BOOT-INF/classes/*"/>
                </mappedresources>
                <mappedresources>
                        <fileset dir="${lib.dir}/runtime" />
                        <globmapper from="*" to="BOOT-INF/lib/*"/>
                </mappedresources>
                <zipfileset src="${lib.dir}/loader/spring-boot-loader-jar-${spring-
boot.version}.jar" />
                <manifest>
                        <attribute name="Main-Class"
value="org.springframework.boot.loader.JarLauncher" />
```

```
                        <attribute name="Start-Class" value="${start-class}" />
                </manifest>
        </jar>
</target>
```

该 Ant 样品具有 `build.xml` 用 `manual` 任务，如果你用下面的命令运行它应该工作文件：

```
$ ant -lib <folder containing ivy-2.2.jar> clean manual
```

然后，您可以使用以下命令运行该应用程序：

```
$ java -jar target/*.jar
```

92.传统部署

Spring Boot 支持传统部署以及更现代的部署形式。本节回答有关传统部署的常见问题。

92.1 创建可部署的 War 文件

警告

因为 Spring WebFlux 并不严格依赖于 Servlet API，并且默认情况下在嵌入式 Reactor Netty 服务器上部署了应用程序，所以 WebFlux 应用程序不支持 War 部署。

生成可部署 war 文件的第一步是提供 `SpringBootServletInitializer` 子类并覆盖其 `configure` 方法。这样做可以利用 Spring Framework 的 Servlet 3.0 支持，并允许您在 servlet 容器启动时配置应用程序。通常，您应该更新应用程序的主类以扩展 `SpringBootServletInitializer`，如以下示例所示：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

        @Override
        protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
                return application.sources(Application.class);
        }

        public static void main(String[] args) throws Exception {
                SpringApplication.run(Application.class, args);
        }

}
```

下一步是更新构建配置，以便项目生成 war 文件而不是 jar 文件。如果您使用 Maven 和 `spring-boot-starter-parent`（为您配置 Maven 的 war 插件），您需要做的就是修改 `pom.xml` 以将包装更改为 war，如下所示：

```
<packaging>war</packaging>
```

如果您使用 Gradle，则需要修改 `build.gradle` 以将 war 插件应用于项目，如下所示：

```
apply plugin: 'war'
```

该过程的最后一步是确保嵌入式 servlet 容器不会干扰部署 war 文件的 servlet 容器。为此，您需要将嵌入式 servlet 容器依赖项标记为已提供。

如果使用 Maven，则以下示例将 servlet 容器（在本例中为 Tomcat）标记为提供：

```
<dependencies>
        <!-- … -->
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
                <scope>provided</scope>
        </dependency>
        <!-- … -->
</dependencies>
```

如果使用 Gradle，则以下示例将 servlet 容器（在本例中为 Tomcat）标记为提供：

```
dependencies {
        // …
        providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
        // …
}
```

小费

`providedRuntime` 优于 Gradle 的 `compileOnly` 配置。除了其他限制之外，`compileOnly` 依赖关系不在测试类路径上，因此任何基于 Web 的集成测试都会失败。

如果使用 Spring Boot 构建工具，则标记所提供的嵌入式 servlet 容器依赖项会生成一个可执行的 war 文件，其中提供的依赖项打包在 `lib-provided` 目录中。这意味着，除了可部署到 servlet 容器之外，还可以在命令行上使用 `java -jar` 运行应用程序。

小费

查看 Spring Boot 的示例应用程序，了解基于 Maven 的先前描述的配置示例。

92.2 将现有应用程序转换为 Spring Boot

对于非 Web 应用程序，应该很容易将现有的 Spring 应用程序转换为 Spring Boot 应用程序。为此，请丢弃创建 `ApplicationContext` 的代码，并将其替换为 `SpringApplication` 或 `SpringApplicationBuilder`。Spring MVC Web 应用程序通常可以首先创建可部署的 war 应用程序，然后再将其迁移到可执行的 war 或 jar。请参阅将 jar 转换为战争的入门指南。

要通过扩展 `SpringBootServletInitializer`（例如，在名为 `Application` 的类中）并添加 Spring Boot `@SpringBootApplication` 注释来创建可部署的战争，请使用类似于以下示例中所示的代码：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

        @Override
        protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
                // Customize the application or call application.sources(...) to add
sources
                // Since our example is itself a @Configuration class (via
@SpringBootApplication)
                // we actually don't need to override this method.
                return application;
        }
```

```
}
```

请记住，无论你在 sources 中放置的是 Spring ApplicationContext。通常，任何已经有效的东西都应该在这里工作。可能有一些 beans 你可以在以后删除，让 Spring Boot 为它们提供自己的默认值，但应该可以在你需要之前获得一些工作。

静态资源可以移动到类路径根目录中的/public（或/static 或/resources 或/META-INF/resources）。这同样适用于 messages.properties（Spring Boot 在类路径的根中自动检测到）。

使用 Spring DispatcherServlet 和 Spring 安全性的香草应该不需要进一步更改。如果应用程序中有其他功能（例如，使用其他 servlet 或过滤器），则可能需要在 Application 上下文中添加一些配置，方法是将这些元素替换为 web.xml，如下所示：

- Servlet 类型 Servlet 或 ServletRegistrationBean 将 bean 安装在容器中，就好像它是<servlet/>和 web.xml web.xml 一样。
- 类型为 Filter 或 FilterRegistrationBean 的@Bean 表现相似（作为<filter/>和<filter-mapping/>）。
- 可以通过 Application 中的@ImportResource 添加 XML 文件中的 ApplicationContext。或者，已经大量使用注释配置的简单情况可以作为@Bean 定义在几行中重新创建。

一旦 war 文件正常工作，您可以通过向 Application 添加 main 方法使其可执行，如以下示例所示：

```
public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
}
```

注意

如果您打算将应用程序作为战争或可执行应用程序启动，则需要以 SpringBootSer
main 方法可用的方法共享构建器的自定义项。类似如下：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

        @Override
        protected SpringApplicationBuilder configure(SpringApplicationBuil
                return configureApplication(builder);
        }

        public static void main(String[] args) {
                configureApplication(new SpringApplicationBuilder()).run(a
        }

        private static SpringApplicationBuilder configureApplication(Sprin
                return builder.sources(Application.class).bannerMode(Banne
        }

}
```

应用程序可以分为多个类别：

- 没有 web.xml 的 Servlet 3.0+应用程序。
- web.xml 的应用程序。
- 具有上下文层次结构的应用

- 没有上下文层次结构的应用

所有这些都应该适合翻译，但每种都可能需要稍微不同的技术。

如果 Servlet 3.0+应用程序已经使用 Spring Servlet 3.0+初始化程序支持类，则它们可能很容易翻译。通常，现有 `WebApplicationInitializer` 中的所有代码都可以移动到 `SpringBootServletInitializer`。如果您的现有应用程序有多个 `ApplicationContext`（例如，如果它使用 `AbstractDispatcherServletInitializer`），那么您可以将所有上下文源合并为一个 `SpringApplication`。您可能遇到的主要复杂问题是，如果组合不起作用，您需要维护上下文层次结构。有关示例，请参阅构建层次结构的 条目。通常需要拆分包含特定于 Web 的功能的现有父上下文，以便所有 `ServletContextAware` 组件都在子上下文中。

尚未 Spring 应用程序的应用程序可以转换为 Spring 启动应用程序，前面提到的指南可能有所帮助。但是，您可能会遇到问题。在这种情况下，我们建议 使用标签 `spring-boot` 向 Stack Overflow 提问。

92.3 将 WAR 部署到 WebLogic

要将 Spring Boot 应用程序部署到 WebLogic，必须确保 servlet 初始化程序**直接**实现 `WebApplicationInitializer`（即使从已经实现它的基类扩展）。

WebLogic 的典型初始化程序应类似于以下示例：

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements
WebApplicationInitializer {

}
```

如果使用 Logback，则还需要告知 WebLogic 更喜欢打包版本而不是预先安装在服务器上的版本。您可以通过添加包含以下内容的 `WEB-INF/weblogic.xml` 文件来执行此操作：

```
<?xml version="1.0" encoding="UTF-8"?>
<wls:weblogic-web-app
        xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
                http://xmlns.oracle.com/weblogic/weblogic-web-app
                http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-
app.xsd">
        <wls:container-descriptor>
                <wls:prefer-application-packages>
                        <wls:package-name>org.slf4j</wls:package-name>
                </wls:prefer-application-packages>
        </wls:container-descriptor>
</wls:weblogic-web-app>
```

92.4 使用 Jedis 代替 Lettuce

默认情况下，Spring Boot 启动器（`spring-boot-starter-data-redis`）使用 Lettuce。您需要排除该依赖项并改为包含 Jedis。Spring Boot 管理这些依赖项，以帮助使这个过程尽可能简单。

以下示例显示了如何在 Maven 中执行此操作：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
        <exclusions>
                <exclusion>
                        <groupId>io.lettuce</groupId>
                        <artifactId>lettuce-core</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
</dependency>
```

以下示例显示了如何在 Gradle 中执行此操作：

```
configurations {
        compile.exclude module: "lettuce"
}

dependencies {
        compile("redis.clients:jedis")
        // ...
}
```

第十部分附录

附录 A.常用应用程序属性

可以在 `application.properties` 文件内，`application.yml` 文件内或命令行开关中指定各种属性。本附录提供了常用 Spring Boot 属性的列表以及对使用它们的基础类的引用。

小费

Spring Boot 提供了各种具有高级值格式的转换机制，请务必查看属性转换部分。

注意

Property 贡献可以来自类路径上的其他 jar 文件，因此您不应将此视为详尽的列表。此外，您可以定义自己的属性。

警告

此示例文件仅供参考。千万**不能**复制和粘贴的全部内容到应用程序中。相反，只选择您需要的属性。

```
# ===================================================================
# COMMON SPRING BOOT PROPERTIES
#
# This sample file is provided as a guideline. Do NOT copy it in its
# entirety to your own application.               ^^^
# ===================================================================
```

```
# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------
debug=false # Enable debug logs.
trace=false # Enable trace logs.

# LOGGING
logging.config= # Location of the logging configuration file. For instance,
`classpath:logback.xml` for Logback.
logging.exception-conversion-word=%wEx # Conversion word used when logging
exceptions.
logging.file= # Log file name (for instance, `myapp.log`). Names can be an exact
location or relative to the current directory.
logging.file.max-history=0 # Maximum of archive log files to keep. Only supported
with the default logback setup.
logging.file.max-size=10MB # Maximum log file size. Only supported with the default
logback setup.
logging.group.*= # Log groups to quickly change multiple loggers at the same time.
For instance, `logging.level.db=org.hibernate,org.springframework.jdbc`.
logging.level.*= # Log levels severity mapping. For instance,
`logging.level.org.springframework=DEBUG`.
logging.path= # Location of the log file. For instance, `/var/log`.
logging.pattern.console= # Appender pattern for output to the console. Supported only
with the default Logback setup.
logging.pattern.dateformat=yyyy-MM-dd HH:mm:ss.SSS # Appender pattern for log date
format. Supported only with the default Logback setup.
logging.pattern.file= # Appender pattern for output to a file. Supported only with
the default Logback setup.
logging.pattern.level=%5p # Appender pattern for log level. Supported only with the
default Logback setup.
logging.register-shutdown-hook=false # Register a shutdown hook for the logging
system when it is initialized.

# AOP
spring.aop.auto=true # Add @EnableAspectJAutoProxy.
spring.aop.proxy-target-class=true # Whether subclass-based (CGLIB) proxies are to be
created (true), as opposed to standard Java interface-based proxies (false).

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.name= # Application name.

# ADMIN (SpringApplicationAdminJmxAutoConfiguration)
spring.application.admin.enabled=false # Whether to enable admin features for the
application.
spring.application.admin.jmx-
name=org.springframework.boot:type=Admin,name=SpringApplication # JMX name of the
application admin MBean.

# AUTO-CONFIGURATION
spring.autoconfigure.exclude= # Auto-configuration classes to exclude.

# BANNER
spring.banner.charset=UTF-8 # Banner file encoding.
spring.banner.location=classpath:banner.txt # Banner text resource location.
spring.banner.image.location=classpath:banner.gif # Banner image file location (jpg
or png can also be used).
spring.banner.image.width=76 # Width of the banner image in chars.
spring.banner.image.height= # Height of the banner image in chars (default based on
image height).
spring.banner.image.margin=2 # Left hand image margin in chars.
spring.banner.image.invert=false # Whether images should be inverted for dark
terminal themes.

# SPRING CORE
spring.beaninfo.ignore=true # Whether to skip search of BeanInfo classes.

# SPRING CACHE (CacheProperties)
spring.cache.cache-names= # Comma-separated list of cache names to create if
supported by the underlying cache manager.
```

```
spring.cache.caffeine.spec= # The spec to use to create caches. See CaffeineSpec for
more details on the spec format.
spring.cache.couchbase.expiration= # Entry expiration. By default the entries never
expire. Note that this value is ultimately converted to seconds.
spring.cache.ehcache.config= # The location of the configuration file to use to
initialize EhCache.
spring.cache.infinispan.config= # The location of the configuration file to use to
initialize Infinispan.
spring.cache.jcache.config= # The location of the configuration file to use to
initialize the cache manager.
spring.cache.jcache.provider= # Fully qualified name of the CachingProvider
implementation to use to retrieve the JSR-107 compliant cache manager. Needed only if
more than one JSR-107 implementation is available on the classpath.
spring.cache.redis.cache-null-values=true # Allow caching null values.
spring.cache.redis.key-prefix= # Key prefix.
spring.cache.redis.time-to-live= # Entry expiration. By default the entries never
expire.
spring.cache.redis.use-key-prefix=true # Whether to use the key prefix when writing
to Redis.
spring.cache.type= # Cache type. By default, auto-detected according to the
environment.

# SPRING CONFIG - using environment property only (ConfigFileApplicationListener)
spring.config.additional-location= # Config file locations used in addition to the
defaults.
spring.config.location= # Config file locations that replace the defaults.
spring.config.name=application # Config file name.

# HAZELCAST (HazelcastProperties)
spring.hazelcast.config= # The location of the configuration file to use to
initialize Hazelcast.

# PROJECT INFORMATION (ProjectInfoProperties)
spring.info.build.encoding=UTF-8 # File encoding.
spring.info.build.location=classpath:META-INF/build-info.properties # Location of the
generated build-info.properties file.
spring.info.git.encoding=UTF-8 # File encoding.
spring.info.git.location=classpath:git.properties # Location of the generated
git.properties file.

# JMX
spring.jmx.default-domain= # JMX domain name.
spring.jmx.enabled=true # Expose management beans to the JMX domain.
spring.jmx.server=mbeanServer # MBeanServer bean name.
spring.jmx.unique-names=false # Whether unique runtime object names should be
ensured.

# Email (MailProperties)
spring.mail.default-encoding=UTF-8 # Default MimeMessage encoding.
spring.mail.host= # SMTP server host. For instance, `smtp.example.com`.
spring.mail.jndi-name= # Session JNDI name. When set, takes precedence over other
Session settings.
spring.mail.password= # Login password of the SMTP server.
spring.mail.port= # SMTP server port.
spring.mail.properties.*= # Additional JavaMail Session properties.
spring.mail.protocol=smtp # Protocol used by the SMTP server.
spring.mail.test-connection=false # Whether to test that the mail server is available
on startup.
spring.mail.username= # Login user of the SMTP server.

# APPLICATION SETTINGS (SpringApplication)
spring.main.allow-bean-definition-overriding=false # Whether bean definition
overriding, by registering a definition with the same name as an existing definition,
is allowed.
spring.main.banner-mode=console # Mode used to display the banner when the
application runs.
spring.main.sources= # Sources (class names, package names, or XML resource
locations) to include in the ApplicationContext.
spring.main.web-application-type= # Flag to explicitly request a specific type of web
```

```
application. If not set, auto-detected based on the classpath.

# FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding= # Expected character encoding the application must
use.

# INTERNATIONALIZATION (MessageSourceProperties)
spring.messages.always-use-message-format=false # Whether to always apply the
MessageFormat rules, parsing even messages without arguments.
spring.messages.basename=messages # Comma-separated list of basenames (essentially a
fully-qualified classpath location), each following the ResourceBundle convention
with relaxed support for slash based locations.
spring.messages.cache-duration= # Loaded resource bundle files cache duration. When
not set, bundles are cached forever. If a duration suffix is not specified, seconds
will be used.
spring.messages.encoding=UTF-8 # Message bundles encoding.
spring.messages.fallback-to-system-locale=true # Whether to fall back to the system
Locale if no files for a specific Locale have been found.
spring.messages.use-code-as-default-message=false # Whether to use the message code
as the default message instead of throwing a "NoSuchMessageException". Recommended
during development only.

# OUTPUT
spring.output.ansi.enabled=detect # Configures the ANSI output.

# PID FILE (ApplicationPidFileWriter)
spring.pid.fail-on-write-error= # Fails if ApplicationPidFileWriter is used but it
cannot write the PID file.
spring.pid.file= # Location of the PID file to write (if ApplicationPidFileWriter is
used).

# PROFILES
spring.profiles.active= # Comma-separated list of active profiles. Can be overridden
by a command line switch.
spring.profiles.include= # Unconditionally activate the specified comma-separated
list of profiles (or list of profiles if using YAML).

# QUARTZ SCHEDULER (QuartzProperties)
spring.quartz.auto-startup=true # Whether to automatically start the scheduler after
initialization.
spring.quartz.jdbc.comment-prefix=-- # Prefix for single-line comments in SQL
initialization scripts.
spring.quartz.jdbc.initialize-schema=embedded # Database schema initialization mode.
spring.quartz.jdbc.schema=classpath:org/quartz/impl/jdbcjobstore/tables_@@platform@@.
sql # Path to the SQL file to use to initialize the database schema.
spring.quartz.job-store-type=memory # Quartz job store type.
spring.quartz.overwrite-existing-jobs=false # Whether configured jobs should
overwrite existing job definitions.
spring.quartz.properties.*= # Additional Quartz Scheduler properties.
spring.quartz.scheduler-name=quartzScheduler # Name of the scheduler.
spring.quartz.startup-delay=0s # Delay after which the scheduler is started once
initialization completes.
spring.quartz.wait-for-jobs-to-complete-on-shutdown=false # Whether to wait for
running jobs to complete on shutdown.

# REACTOR (ReactorCoreProperties)
spring.reactor.stacktrace-mode.enabled=false # Whether Reactor should collect
stacktrace information at runtime.

# SENDGRID (SendGridAutoConfiguration)
spring.sendgrid.api-key= # SendGrid API key.
spring.sendgrid.proxy.host= # SendGrid proxy host.
spring.sendgrid.proxy.port= # SendGrid proxy port.

# TASK EXECUTION  (TaskExecutionProperties)
spring.task.execution.pool.allow-core-thread-timeout=true # Whether core threads are
allowed to time out. This enables dynamic growing and shrinking of the pool.
spring.task.execution.pool.core-size=8 # Core number of threads.
spring.task.execution.pool.keep-alive=60s # Time limit for which threads may remain
```

```
idle before being terminated.
spring.task.execution.pool.max-size= # Maximum allowed number of threads. If tasks
are filling up the queue, the pool can expand up to that size to accommodate the
load. Ignored if the queue is unbounded.
spring.task.execution.pool.queue-capacity= # Queue capacity. An unbounded capacity
does not increase the pool and therefore ignores the "max-size" property.
spring.task.execution.thread-name-prefix=task- # Prefix to use for the names of newly
created threads.

# TASK SCHEDULING  (TaskSchedulingProperties)
spring.task.scheduling.pool.size=1 # Maximum allowed number of threads.
spring.task.scheduling.thread-name-prefix=scheduling- # Prefix to use for the names
of newly created threads.

# ----------------------------------------
# WEB PROPERTIES
# ----------------------------------------

# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.address= # Network address to which the server should bind.
server.compression.enabled=false # Whether response compression is enabled.
server.compression.excluded-user-agents= # Comma-separated list of user agents for
which responses should not be compressed.
server.compression.mime-
types=text/html,text/xml,text/plain,text/css,text/javascript,application/javascript,a
pplication/json,application/xml # Comma-separated list of MIME types that should be
compressed.
server.compression.min-response-size=2KB # Minimum "Content-Length" value that is
required for compression to be performed.
server.connection-timeout= # Time that connectors wait for another HTTP request
before closing the connection. When not set, the connector's container-specific
default is used. Use a value of -1 to indicate no (that is, an infinite) timeout.
server.error.include-exception=false # Include the "exception" attribute.
server.error.include-stacktrace=never # When to include a "stacktrace" attribute.
server.error.path=/error # Path of the error controller.
server.error.whitelabel.enabled=true # Whether to enable the default error page
displayed in browsers in case of a server error.
server.http2.enabled=false # Whether to enable HTTP/2 support, if the current
environment supports it.
server.jetty.acceptors=-1 # Number of acceptor threads to use. When the value is -1,
the default, the number of acceptors is derived from the operating environment.
server.jetty.accesslog.append=false # Append to log.
server.jetty.accesslog.date-format=dd/MMM/yyyy:HH:mm:ss Z # Timestamp format of the
request log.
server.jetty.accesslog.enabled=false # Enable access log.
server.jetty.accesslog.extended-format=false # Enable extended NCSA format.
server.jetty.accesslog.file-date-format= # Date format to place in log file name.
server.jetty.accesslog.filename= # Log filename. If not specified, logs redirect to
"System.err".
server.jetty.accesslog.locale= # Locale of the request log.
server.jetty.accesslog.log-cookies=false # Enable logging of the request cookies.
server.jetty.accesslog.log-latency=false # Enable logging of request processing time.
server.jetty.accesslog.log-server=false # Enable logging of the request hostname.
server.jetty.accesslog.retention-period=31 # Number of days before rotated log files
are deleted.
server.jetty.accesslog.time-zone=GMT # Timezone of the request log.
server.jetty.max-http-post-size=200000B # Maximum size of the HTTP post or put
content.
server.jetty.selectors=-1 # Number of selector threads to use. When the value is -1,
the default, the number of selectors is derived from the operating environment.
server.max-http-header-size=8KB # Maximum size of the HTTP message header.
server.port=8080 # Server HTTP port.
server.server-header= # Value to use for the Server response header (if empty, no
header is sent).
server.use-forward-headers= # Whether X-Forwarded-* headers should be applied to the
HttpRequest.
server.servlet.context-parameters.*= # Servlet context init parameters.
server.servlet.context-path= # Context path of the application.
server.servlet.application-display-name=application # Display name of the
```

```
application.
server.servlet.jsp.class-name=org.apache.jasper.servlet.JspServlet # Class name of
the servlet to use for JSPs.
server.servlet.jsp.init-parameters.*= # Init parameters used to configure the JSP
servlet.
server.servlet.jsp.registered=true # Whether the JSP servlet is registered.
server.servlet.session.cookie.comment= # Comment for the session cookie.
server.servlet.session.cookie.domain= # Domain for the session cookie.
server.servlet.session.cookie.http-only= # Whether to use "HttpOnly" cookies for
session cookies.
server.servlet.session.cookie.max-age= # Maximum age of the session cookie. If a
duration suffix is not specified, seconds will be used.
server.servlet.session.cookie.name= # Session cookie name.
server.servlet.session.cookie.path= # Path of the session cookie.
server.servlet.session.cookie.secure= # Whether to always mark the session cookie as
secure.
server.servlet.session.persistent=false # Whether to persist session data between
restarts.
server.servlet.session.store-dir= # Directory used to store session data.
server.servlet.session.timeout=30m # Session timeout. If a duration suffix is not
specified, seconds will be used.
server.servlet.session.tracking-modes= # Session tracking modes.
server.ssl.ciphers= # Supported SSL ciphers.
server.ssl.client-auth= # Client authentication mode.
server.ssl.enabled=true # Whether to enable SSL support.
server.ssl.enabled-protocols= # Enabled SSL protocols.
server.ssl.key-alias= # Alias that identifies the key in the key store.
server.ssl.key-password= # Password used to access the key in the key store.
server.ssl.key-store= # Path to the key store that holds the SSL certificate
(typically a jks file).
server.ssl.key-store-password= # Password used to access the key store.
server.ssl.key-store-provider= # Provider for the key store.
server.ssl.key-store-type= # Type of the key store.
server.ssl.protocol=TLS # SSL protocol to use.
server.ssl.trust-store= # Trust store that holds SSL certificates.
server.ssl.trust-store-password= # Password used to access the trust store.
server.ssl.trust-store-provider= # Provider for the trust store.
server.ssl.trust-store-type= # Type of the trust store.
server.tomcat.accept-count=100 # Maximum queue length for incoming connection
requests when all possible request processing threads are in use.
server.tomcat.accesslog.buffered=true # Whether to buffer output such that it is
flushed only periodically.
server.tomcat.accesslog.directory=logs # Directory in which log files are created.
Can be absolute or relative to the Tomcat base dir.
server.tomcat.accesslog.enabled=false # Enable access log.
server.tomcat.accesslog.file-date-format=.yyyy-MM-dd # Date format to place in the
log file name.
server.tomcat.accesslog.pattern=common # Format pattern for access logs.
server.tomcat.accesslog.prefix=access_log # Log file name prefix.
server.tomcat.accesslog.rename-on-rotate=false # Whether to defer inclusion of the
date stamp in the file name until rotate time.
server.tomcat.accesslog.request-attributes-enabled=false # Set request attributes for
the IP address, Hostname, protocol, and port used for the request.
server.tomcat.accesslog.rotate=true # Whether to enable access log rotation.
server.tomcat.accesslog.suffix=.log # Log file name suffix.
server.tomcat.additional-tld-skip-patterns= # Comma-separated list of additional
patterns that match jars to ignore for TLD scanning.
server.tomcat.background-processor-delay=10s # Delay between the invocation of
backgroundProcess methods. If a duration suffix is not specified, seconds will be
used.
server.tomcat.basedir= # Tomcat base directory. If not specified, a temporary
directory is used.
server.tomcat.internal-proxies=10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\
                192\\.168\\.\\d{1,3}\\.\\d{1,3}|\\
                169\\.254\\.\\d{1,3}\\.\\d{1,3}|\\
                127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\
                172\\.1[6-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
                172\\.2[0-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
                172\\.3[0-1]{1}\\.\\d{1,3}\\.\\d{1,3}\\
```

```
                0:0:0:0:0:0:0:1\\
                ::1 # Regular expression that matches proxies that are to be trusted.
server.tomcat.max-connections=10000 # Maximum number of connections that the server
accepts and processes at any given time.
server.tomcat.max-http-post-size=2MB # Maximum size of the HTTP post content.
server.tomcat.max-swallow-size=2MB # Maximum amount of request body to swallow.
server.tomcat.max-threads=200 # Maximum amount of worker threads.
server.tomcat.min-spare-threads=10 # Minimum amount of worker threads.
server.tomcat.port-header=X-Forwarded-Port # Name of the HTTP header used to override
the original port value.
server.tomcat.protocol-header= # Header that holds the incoming protocol, usually
named "X-Forwarded-Proto".
server.tomcat.protocol-header-https-value=https # Value of the protocol header
indicating whether the incoming request uses SSL.
server.tomcat.redirect-context-root=true # Whether requests to the context root
should be redirected by appending a / to the path.
server.tomcat.remote-ip-header= # Name of the HTTP header from which the remote IP is
extracted. For instance, `X-FORWARDED-FOR`.
server.tomcat.resource.allow-caching=true # Whether static resource caching is
permitted for this web application.
server.tomcat.resource.cache-ttl= # Time-to-live of the static resource cache.
server.tomcat.uri-encoding=UTF-8 # Character encoding to use to decode the URI.
server.tomcat.use-relative-redirects= # Whether HTTP 1.1 and later location headers
generated by a call to sendRedirect will use relative or absolute redirects.
server.undertow.accesslog.dir= # Undertow access log directory.
server.undertow.accesslog.enabled=false # Whether to enable the access log.
server.undertow.accesslog.pattern=common # Format pattern for access logs.
server.undertow.accesslog.prefix=access_log. # Log file name prefix.
server.undertow.accesslog.rotate=true # Whether to enable access log rotation.
server.undertow.accesslog.suffix=log # Log file name suffix.
server.undertow.buffer-size= # Size of each buffer.
server.undertow.direct-buffers= # Whether to allocate buffers outside the Java heap.
The default is derived from the maximum amount of memory that is available to the
JVM.
server.undertow.eager-filter-init=true # Whether servlet filters should be
initialized on startup.
server.undertow.io-threads= # Number of I/O threads to create for the worker. The
default is derived from the number of available processors.
server.undertow.max-http-post-size=-1B # Maximum size of the HTTP post content. When
the value is -1, the default, the size is unlimited.
server.undertow.worker-threads= # Number of worker threads. The default is 8 times
the number of I/O threads.

# FREEMARKER (FreeMarkerProperties)
spring.freemarker.allow-request-override=false # Whether HttpServletRequest
attributes are allowed to override (hide) controller generated model attributes of
the same name.
spring.freemarker.allow-session-override=false # Whether HttpSession attributes are
allowed to override (hide) controller generated model attributes of the same name.
spring.freemarker.cache=false # Whether to enable template caching.
spring.freemarker.charset=UTF-8 # Template encoding.
spring.freemarker.check-template-location=true # Whether to check that the templates
location exists.
spring.freemarker.content-type=text/html # Content-Type value.
spring.freemarker.enabled=true # Whether to enable MVC view resolution for this
technology.
spring.freemarker.expose-request-attributes=false # Whether all request attributes
should be added to the model prior to merging with the template.
spring.freemarker.expose-session-attributes=false # Whether all HttpSession
attributes should be added to the model prior to merging with the template.
spring.freemarker.expose-spring-macro-helpers=true # Whether to expose a
RequestContext for use by Spring's macro library, under the name
"springMacroRequestContext".
spring.freemarker.prefer-file-system-access=true # Whether to prefer file system
access for template loading. File system access enables hot detection of template
changes.
spring.freemarker.prefix= # Prefix that gets prepended to view names when building a
URL.
spring.freemarker.request-context-attribute= # Name of the RequestContext attribute
```

```
for all views.
spring.freemarker.settings.*= # Well-known FreeMarker keys which are passed to
FreeMarker's Configuration.
spring.freemarker.suffix=.ftl # Suffix that gets appended to view names when building
a URL.
spring.freemarker.template-loader-path=classpath:/templates/ # Comma-separated list
of template paths.
spring.freemarker.view-names= # White list of view names that can be resolved.

# GROOVY TEMPLATES (GroovyTemplateProperties)
spring.groovy.template.allow-request-override=false # Whether HttpServletRequest
attributes are allowed to override (hide) controller generated model attributes of
the same name.
spring.groovy.template.allow-session-override=false # Whether HttpSession attributes
are allowed to override (hide) controller generated model attributes of the same
name.
spring.groovy.template.cache=false # Whether to enable template caching.
spring.groovy.template.charset=UTF-8 # Template encoding.
spring.groovy.template.check-template-location=true # Whether to check that the
templates location exists.
spring.groovy.template.configuration.*= # See GroovyMarkupConfigurer
spring.groovy.template.content-type=text/html # Content-Type value.
spring.groovy.template.enabled=true # Whether to enable MVC view resolution for this
technology.
spring.groovy.template.expose-request-attributes=false # Whether all request
attributes should be added to the model prior to merging with the template.
spring.groovy.template.expose-session-attributes=false # Whether all HttpSession
attributes should be added to the model prior to merging with the template.
spring.groovy.template.expose-spring-macro-helpers=true # Whether to expose a
RequestContext for use by Spring's macro library, under the name
"springMacroRequestContext".
spring.groovy.template.prefix= # Prefix that gets prepended to view names when
building a URL.
spring.groovy.template.request-context-attribute= # Name of the RequestContext
attribute for all views.
spring.groovy.template.resource-loader-path=classpath:/templates/ # Template path.
spring.groovy.template.suffix=.tpl # Suffix that gets appended to view names when
building a URL.
spring.groovy.template.view-names= # White list of view names that can be resolved.

# SPRING HATEOAS (HateoasProperties)
spring.hateoas.use-hal-as-default-json-media-type=true # Whether application/hal+json
responses should be sent to requests that accept application/json.

# HTTP (HttpProperties)
spring.http.converters.preferred-json-mapper= # Preferred JSON mapper to use for HTTP
message conversion. By default, auto-detected according to the environment.
spring.http.encoding.charset=UTF-8 # Charset of HTTP requests and responses. Added to
the "Content-Type" header if not set explicitly.
spring.http.encoding.enabled=true # Whether to enable http encoding support.
spring.http.encoding.force= # Whether to force the encoding to the configured charset
on HTTP requests and responses.
spring.http.encoding.force-request= # Whether to force the encoding to the configured
charset on HTTP requests. Defaults to true when "force" has not been specified.
spring.http.encoding.force-response= # Whether to force the encoding to the
configured charset on HTTP responses.
spring.http.encoding.mapping= # Locale in which to encode mapping.
spring.http.log-request-details=false # Whether logging of (potentially sensitive)
request details at DEBUG and TRACE level is allowed.

# MULTIPART (MultipartProperties)
spring.servlet.multipart.enabled=true # Whether to enable support of multipart
uploads.
spring.servlet.multipart.file-size-threshold=0B # Threshold after which files are
written to disk.
spring.servlet.multipart.location= # Intermediate location of uploaded files.
spring.servlet.multipart.max-file-size=1MB # Max file size.
spring.servlet.multipart.max-request-size=10MB # Max request size.
spring.servlet.multipart.resolve-lazily=false # Whether to resolve the multipart
```

```
request lazily at the time of file or parameter access.

# JACKSON (JacksonProperties)
spring.jackson.date-format= # Date format string or a fully-qualified date format
class name. For instance, `yyyy-MM-dd HH:mm:ss`.
spring.jackson.default-property-inclusion= # Controls the inclusion of properties
during serialization. Configured with one of the values in Jackson's
JsonInclude.Include enumeration.
spring.jackson.deserialization.*= # Jackson on/off features that affect the way Java
objects are deserialized.
spring.jackson.generator.*= # Jackson on/off features for generators.
spring.jackson.joda-date-time-format= # Joda date time format string. If not
configured, "date-format" is used as a fallback if it is configured with a format
string.
spring.jackson.locale= # Locale used for formatting.
spring.jackson.mapper.*= # Jackson general purpose on/off features.
spring.jackson.parser.*= # Jackson on/off features for parsers.
spring.jackson.property-naming-strategy= # One of the constants on Jackson's
PropertyNamingStrategy. Can also be a fully-qualified class name of a
PropertyNamingStrategy subclass.
spring.jackson.serialization.*= # Jackson on/off features that affect the way Java
objects are serialized.
spring.jackson.time-zone= #  Time zone used when formatting dates. For instance,
"America/Los_Angeles" or "GMT+10".
spring.jackson.visibility.*= # Jackson visibility thresholds that can be used to
limit which methods (and fields) are auto-detected.

# GSON (GsonProperties)
spring.gson.date-format= # Format to use when serializing Date objects.
spring.gson.disable-html-escaping= # Whether to disable the escaping of HTML
characters such as '<', '>', etc.
spring.gson.disable-inner-class-serialization= # Whether to exclude inner classes
during serialization.
spring.gson.enable-complex-map-key-serialization= # Whether to enable serialization
of complex map keys (i.e. non-primitives).
spring.gson.exclude-fields-without-expose-annotation= # Whether to exclude all fields
from consideration for serialization or deserialization that do not have the "Expose"
annotation.
spring.gson.field-naming-policy= # Naming policy that should be applied to an
object's field during serialization and deserialization.
spring.gson.generate-non-executable-json= # Whether to generate non executable JSON
by prefixing the output with some special text.
spring.gson.lenient= # Whether to be lenient about parsing JSON that doesn't conform
to RFC 4627.
spring.gson.long-serialization-policy= # Serialization policy for Long and long
types.
spring.gson.pretty-printing= # Whether to output serialized JSON that fits in a page
for pretty printing.
spring.gson.serialize-nulls= # Whether to serialize null fields.

# JERSEY (JerseyProperties)
spring.jersey.application-path= # Path that serves as the base URI for the
application. If specified, overrides the value of "@ApplicationPath".
spring.jersey.filter.order=0 # Jersey filter chain order.
spring.jersey.init.*= # Init parameters to pass to Jersey through the servlet or
filter.
spring.jersey.servlet.load-on-startup=-1 # Load on startup priority of the Jersey
servlet.
spring.jersey.type=servlet # Jersey integration type.

# SPRING LDAP (LdapProperties)
spring.ldap.anonymous-read-only=false # Whether read-only operations should use an
anonymous environment.
spring.ldap.base= # Base suffix from which all operations should originate.
spring.ldap.base-environment.*= # LDAP specification settings.
spring.ldap.password= # Login password of the server.
spring.ldap.urls= # LDAP URLs of the server.
spring.ldap.username= # Login username of the server.
```

```
# EMBEDDED LDAP (EmbeddedLdapProperties)
spring.ldap.embedded.base-dn= # List of base DNs.
spring.ldap.embedded.credential.username= # Embedded LDAP username.
spring.ldap.embedded.credential.password= # Embedded LDAP password.
spring.ldap.embedded.ldif=classpath:schema.ldif # Schema (LDIF) script resource
reference.
spring.ldap.embedded.port=0 # Embedded LDAP port.
spring.ldap.embedded.validation.enabled=true # Whether to enable LDAP schema
validation.
spring.ldap.embedded.validation.schema= # Path to the custom schema.

# MUSTACHE TEMPLATES (MustacheAutoConfiguration)
spring.mustache.allow-request-override=false # Whether HttpServletRequest attributes
are allowed to override (hide) controller generated model attributes of the same
name.
spring.mustache.allow-session-override=false # Whether HttpSession attributes are
allowed to override (hide) controller generated model attributes of the same name.
spring.mustache.cache=false # Whether to enable template caching.
spring.mustache.charset=UTF-8 # Template encoding.
spring.mustache.check-template-location=true # Whether to check that the templates
location exists.
spring.mustache.content-type=text/html # Content-Type value.
spring.mustache.enabled=true # Whether to enable MVC view resolution for this
technology.
spring.mustache.expose-request-attributes=false # Whether all request attributes
should be added to the model prior to merging with the template.
spring.mustache.expose-session-attributes=false # Whether all HttpSession attributes
should be added to the model prior to merging with the template.
spring.mustache.expose-spring-macro-helpers=true # Whether to expose a RequestContext
for use by Spring's macro library, under the name "springMacroRequestContext".
spring.mustache.prefix=classpath:/templates/ # Prefix to apply to template names.
spring.mustache.request-context-attribute= # Name of the RequestContext attribute for
all views.
spring.mustache.suffix=.mustache # Suffix to apply to template names.
spring.mustache.view-names= # White list of view names that can be resolved.

# SPRING MVC (WebMvcProperties)
spring.mvc.async.request-timeout= # Amount of time before asynchronous request
handling times out.
spring.mvc.contentnegotiation.favor-parameter=false # Whether a request parameter
("format" by default) should be used to determine the requested media type.
spring.mvc.contentnegotiation.favor-path-extension=false # Whether the path extension
in the URL path should be used to determine the requested media type.
spring.mvc.contentnegotiation.media-types.*= # Map file extensions to media types for
content negotiation. For instance, yml to text/yaml.
spring.mvc.contentnegotiation.parameter-name= # Query parameter name to use when
"favor-parameter" is enabled.
spring.mvc.date-format= # Date format to use. For instance, `dd/MM/yyyy`.
spring.mvc.dispatch-trace-request=false # Whether to dispatch TRACE requests to the
FrameworkServlet doService method.
spring.mvc.dispatch-options-request=true # Whether to dispatch OPTIONS requests to
the FrameworkServlet doService method.
spring.mvc.favicon.enabled=true # Whether to enable resolution of favicon.ico.
spring.mvc.formcontent.filter.enabled=true # Whether to enable Spring's
FormContentFilter.
spring.mvc.hiddenmethod.filter.enabled=true # Whether to enable Spring's
HiddenHttpMethodFilter.
spring.mvc.ignore-default-model-on-redirect=true # Whether the content of the
"default" model should be ignored during redirect scenarios.
spring.mvc.locale= # Locale to use. By default, this locale is overridden by the
"Accept-Language" header.
spring.mvc.locale-resolver=accept-header # Define how the locale should be resolved.
spring.mvc.log-resolved-exception=false # Whether to enable warn logging of
exceptions resolved by a "HandlerExceptionResolver", except for
"DefaultHandlerExceptionResolver".
spring.mvc.message-codes-resolver-format= # Formatting strategy for message codes.
For instance, `PREFIX_ERROR_CODE`.
spring.mvc.pathmatch.use-registered-suffix-pattern=false # Whether suffix pattern
matching should work only against extensions registered with
```

"spring.mvc.contentnegotiation.media-types.*".
spring.mvc.pathmatch.use-suffix-pattern=false # Whether to use suffix pattern match (".*") when matching patterns to requests.
spring.mvc.servlet.load-on-startup=-1 # Load on startup priority of the dispatcher servlet.
spring.mvc.servlet.path=/ # Path of the dispatcher servlet.
spring.mvc.static-path-pattern=/** # Path pattern used for static resources.
spring.mvc.throw-exception-if-no-handler-found=false # Whether a "NoHandlerFoundException" should be thrown if no Handler was found to process a request.
spring.mvc.view.prefix= # Spring MVC view prefix.
spring.mvc.view.suffix= # Spring MVC view suffix.

SPRING RESOURCES HANDLING (ResourceProperties)
spring.resources.add-mappings=true # Whether to enable default resource handling.
spring.resources.cache.cachecontrol.cache-private= # Indicate that the response message is intended for a single user and must not be stored by a shared cache.
spring.resources.cache.cachecontrol.cache-public= # Indicate that any cache may store the response.
spring.resources.cache.cachecontrol.max-age= # Maximum time the response should be cached, in seconds if no duration suffix is not specified.
spring.resources.cache.cachecontrol.must-revalidate= # Indicate that once it has become stale, a cache must not use the response without re-validating it with the server.
spring.resources.cache.cachecontrol.no-cache= # Indicate that the cached response can be reused only if re-validated with the server.
spring.resources.cache.cachecontrol.no-store= # Indicate to not cache the response in any case.
spring.resources.cache.cachecontrol.no-transform= # Indicate intermediaries (caches and others) that they should not transform the response content.
spring.resources.cache.cachecontrol.proxy-revalidate= # Same meaning as the "must-revalidate" directive, except that it does not apply to private caches.
spring.resources.cache.cachecontrol.s-max-age= # Maximum time the response should be cached by shared caches, in seconds if no duration suffix is not specified.
spring.resources.cache.cachecontrol.stale-if-error= # Maximum time the response may be used when errors are encountered, in seconds if no duration suffix is not specified.
spring.resources.cache.cachecontrol.stale-while-revalidate= # Maximum time the response can be served after it becomes stale, in seconds if no duration suffix is not specified.
spring.resources.cache.period= # Cache period for the resources served by the resource handler. If a duration suffix is not specified, seconds will be used.
spring.resources.chain.cache=true # Whether to enable caching in the Resource chain.
spring.resources.chain.compressed=false # Whether to enable resolution of already compressed resources (gzip, brotli).
spring.resources.chain.enabled= # Whether to enable the Spring Resource Handling chain. By default, disabled unless at least one strategy has been enabled.
spring.resources.chain.html-application-cache=false # Whether to enable HTML5 application cache manifest rewriting.
spring.resources.chain.strategy.content.enabled=false # Whether to enable the content Version Strategy.
spring.resources.chain.strategy.content.paths=/** # Comma-separated list of patterns to apply to the content Version Strategy.
spring.resources.chain.strategy.fixed.enabled=false # Whether to enable the fixed Version Strategy.
spring.resources.chain.strategy.fixed.paths=/** # Comma-separated list of patterns to apply to the fixed Version Strategy.
spring.resources.chain.strategy.fixed.version= # Version string to use for the fixed Version Strategy.
spring.resources.static-locations=classpath:/META-INF/resources/,classpath:/resources/,classpath:/static/,classpath:/public/ # Locations of static resources.

SPRING SESSION (SessionProperties)
spring.session.store-type= # Session store type.
spring.session.timeout= # Session timeout. If a duration suffix is not specified, seconds will be used.
spring.session.servlet.filter-order=-2147483598 # Session repository filter order.
spring.session.servlet.filter-dispatcher-types=async,error,request # Session

```
repository filter dispatcher types.

# SPRING SESSION HAZELCAST (HazelcastSessionProperties)
spring.session.hazelcast.flush-mode=on-save # Sessions flush mode.
spring.session.hazelcast.map-name=spring:session:sessions # Name of the map used to
store sessions.

# SPRING SESSION JDBC (JdbcSessionProperties)
spring.session.jdbc.cleanup-cron=0 * * * * * # Cron expression for expired session
cleanup job.
spring.session.jdbc.initialize-schema=embedded # Database schema initialization mode.
spring.session.jdbc.schema=classpath:org/springframework/session/jdbc/schema-
@@platform@@.sql # Path to the SQL file to use to initialize the database schema.
spring.session.jdbc.table-name=SPRING_SESSION # Name of the database table used to
store sessions.

# SPRING SESSION MONGODB (MongoSessionProperties)
spring.session.mongodb.collection-name=sessions # Collection name used to store
sessions.

# SPRING SESSION REDIS (RedisSessionProperties)
spring.session.redis.cleanup-cron=0 * * * * * # Cron expression for expired session
cleanup job.
spring.session.redis.flush-mode=on-save # Sessions flush mode.
spring.session.redis.namespace=spring:session # Namespace for keys used to store
sessions.

# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.cache=true # Whether to enable template caching.
spring.thymeleaf.check-template=true # Whether to check that the template exists
before rendering it.
spring.thymeleaf.check-template-location=true # Whether to check that the templates
location exists.
spring.thymeleaf.enabled=true # Whether to enable Thymeleaf view resolution for Web
frameworks.
spring.thymeleaf.enable-spring-el-compiler=false # Enable the SpringEL compiler in
SpringEL expressions.
spring.thymeleaf.encoding=UTF-8 # Template files encoding.
spring.thymeleaf.excluded-view-names= # Comma-separated list of view names (patterns
allowed) that should be excluded from resolution.
spring.thymeleaf.mode=HTML # Template mode to be applied to templates. See also
Thymeleaf's TemplateMode enum.
spring.thymeleaf.prefix=classpath:/templates/ # Prefix that gets prepended to view
names when building a URL.
spring.thymeleaf.reactive.chunked-mode-view-names= # Comma-separated list of view
names (patterns allowed) that should be the only ones executed in CHUNKED mode when a
max chunk size is set.
spring.thymeleaf.reactive.full-mode-view-names= # Comma-separated list of view names
(patterns allowed) that should be executed in FULL mode even if a max chunk size is
set.
spring.thymeleaf.reactive.max-chunk-size=0B # Maximum size of data buffers used for
writing to the response.
spring.thymeleaf.reactive.media-types= # Media types supported by the view
technology.
spring.thymeleaf.render-hidden-markers-before-checkboxes=false # Whether hidden form
inputs acting as markers for checkboxes should be rendered before the checkbox
element itself.
spring.thymeleaf.servlet.content-type=text/html # Content-Type value written to HTTP
responses.
spring.thymeleaf.servlet.produce-partial-output-while-processing=true # Whether
Thymeleaf should start writing partial output as soon as possible or buffer until
template processing is finished.
spring.thymeleaf.suffix=.html # Suffix that gets appended to view names when building
a URL.
spring.thymeleaf.template-resolver-order= # Order of the template resolver in the
chain.
spring.thymeleaf.view-names= # Comma-separated list of view names (patterns allowed)
that can be resolved.
```

```
# SPRING WEBFLUX (WebFluxProperties)
spring.webflux.date-format= # Date format to use. For instance, `dd/MM/yyyy`.
spring.webflux.hiddenmethod.filter.enabled=true # Whether to enable Spring's
HiddenHttpMethodFilter.
spring.webflux.static-path-pattern=/** # Path pattern used for static resources.

# SPRING WEB SERVICES (WebServicesProperties)
spring.webservices.path=/services # Path that serves as the base URI for the
services.
spring.webservices.servlet.init= # Servlet init parameters to pass to Spring Web
Services.
spring.webservices.servlet.load-on-startup=-1 # Load on startup priority of the
Spring Web Services servlet.
spring.webservices.wsdl-locations= # Comma-separated list of locations of WSDLs and
accompanying XSDs to be exposed as beans.


# ----------------------------------------
# SECURITY PROPERTIES
# ----------------------------------------
# SECURITY (SecurityProperties)
spring.security.filter.order=-100 # Security filter chain order.
spring.security.filter.dispatcher-types=async,error,request # Security filter chain
dispatcher types.
spring.security.user.name=user # Default user name.
spring.security.user.password= # Password for the default user name.
spring.security.user.roles= # Granted roles for the default user name.

# SECURITY OAUTH2 CLIENT (OAuth2ClientProperties)
spring.security.oauth2.client.provider.*= # OAuth provider details.
spring.security.oauth2.client.registration.*= # OAuth client registrations.

# SECURITY OAUTH2 RESOURCE SERVER (OAuth2ResourceServerProperties)
spring.security.oauth2.resourceserver.jwt.jwk-set-uri= # JSON Web Key URI to use to
verify the JWT token.
   spring.security.oauth2.resourceserver.jwt.issuer-uri= # URI that an OpenID Connect
Provider asserts as its Issuer Identifier.

# ----------------------------------------
# DATA PROPERTIES
# ----------------------------------------

# FLYWAY (FlywayProperties)
spring.flyway.baseline-description=<< Flyway Baseline >> # Description to tag an
existing schema with when applying a baseline.
spring.flyway.baseline-on-migrate=false # Whether to automatically call baseline when
migrating a non-empty schema.
spring.flyway.baseline-version=1 # Version to tag an existing schema with when
executing baseline.
spring.flyway.check-location=true # Whether to check that migration scripts location
exists.
spring.flyway.clean-disabled=false # Whether to disable cleaning of the database.
spring.flyway.clean-on-validation-error=false # Whether to automatically call clean
when a validation error occurs.
spring.flyway.connect-retries=0 # Maximum number of retries when attempting to
connect to the database.
spring.flyway.enabled=true # Whether to enable flyway.
spring.flyway.encoding=UTF-8 # Encoding of SQL migrations.
spring.flyway.group=false # Whether to group all pending migrations together in the
same transaction when applying them.
spring.flyway.ignore-future-migrations=true # Whether to ignore future migrations
when reading the schema history table.
spring.flyway.ignore-ignored-migrations=false # Whether to ignore ignored migrations
when reading the schema history table.
spring.flyway.ignore-missing-migrations=false # Whether to ignore missing migrations
when reading the schema history table.
spring.flyway.ignore-pending-migrations=false # Whether to ignore pending migrations
when reading the schema history table.
spring.flyway.init-sqls= # SQL statements to execute to initialize a connection
```

```
immediately after obtaining it.
spring.flyway.installed-by= # Username recorded in the schema history table as having
applied the migration.
spring.flyway.locations=classpath:db/migration # Locations of migrations scripts. Can
contain the special "{vendor}" placeholder to use vendor-specific locations.
spring.flyway.mixed=false # Whether to allow mixing transactional and non-
transactional statements within the same migration.
spring.flyway.out-of-order=false # Whether to allow migrations to be run out of
order.
spring.flyway.password= # Login password of the database to migrate.
spring.flyway.placeholder-prefix=${ # Prefix of placeholders in migration scripts.
spring.flyway.placeholder-replacement=true # Perform placeholder replacement in
migration scripts.
spring.flyway.placeholder-suffix=} # Suffix of placeholders in migration scripts.
spring.flyway.placeholders= # Placeholders and their replacements to apply to sql
migration scripts.
spring.flyway.repeatable-sql-migration-prefix=R # File name prefix for repeatable SQL
migrations.
spring.flyway.schemas= # Scheme names managed by Flyway (case-sensitive).
spring.flyway.skip-default-callbacks=false # Whether to skip default callbacks. If
true, only custom callbacks are used.
spring.flyway.skip-default-resolvers=false # Whether to skip default resolvers. If
true, only custom resolvers are used.
spring.flyway.sql-migration-prefix=V # File name prefix for SQL migrations.
spring.flyway.sql-migration-separator=__ # File name separator for SQL migrations.
spring.flyway.sql-migration-suffixes=.sql # File name suffix for SQL migrations.
spring.flyway.table=flyway_schema_history # Name of the schema schema history table
that will be used by Flyway.
spring.flyway.target= # Target version up to which migrations should be considered.
spring.flyway.url= # JDBC url of the database to migrate. If not set, the primary
configured data source is used.
spring.flyway.user= # Login user of the database to migrate.
spring.flyway.validate-on-migrate=true # Whether to automatically call validate when
performing a migration.

# LIQUIBASE (LiquibaseProperties)
spring.liquibase.change-log=classpath:/db/changelog/db.changelog-master.yaml # Change
log configuration path.
spring.liquibase.check-change-log-location=true # Whether to check that the change
log location exists.
spring.liquibase.contexts= # Comma-separated list of runtime contexts to use.
spring.liquibase.database-change-log-lock-table=DATABASECHANGELOGLOCK # Name of table
to use for tracking concurrent Liquibase usage.
spring.liquibase.database-change-log-table=DATABASECHANGELOG # Name of table to use
for tracking change history.
spring.liquibase.default-schema= # Default database schema.
spring.liquibase.drop-first=false # Whether to first drop the database schema.
spring.liquibase.enabled=true # Whether to enable Liquibase support.
spring.liquibase.labels= # Comma-separated list of runtime labels to use.
spring.liquibase.liquibase-schema= # Schema to use for Liquibase objects.
spring.liquibase.liquibase-tablespace= # Tablespace to use for Liquibase objects.
spring.liquibase.parameters.*= # Change log parameters.
spring.liquibase.password= # Login password of the database to migrate.
spring.liquibase.rollback-file= # File to which rollback SQL is written when an
update is performed.
spring.liquibase.test-rollback-on-update=false # Whether rollback should be tested
before update is performed.
spring.liquibase.url= # JDBC URL of the database to migrate. If not set, the primary
configured data source is used.
spring.liquibase.user= # Login user of the database to migrate.

# COUCHBASE (CouchbaseProperties)
spring.couchbase.bootstrap-hosts= # Couchbase nodes (host or IP address) to bootstrap
from.
spring.couchbase.bucket.name=default # Name of the bucket to connect to.
spring.couchbase.bucket.password=  # Password of the bucket.
spring.couchbase.env.endpoints.key-value=1 # Number of sockets per node against the
key/value service.
spring.couchbase.env.endpoints.queryservice.min-endpoints=1 # Minimum number of
```

```
sockets per node.
spring.couchbase.env.endpoints.queryservice.max-endpoints=1 # Maximum number of
sockets per node.
spring.couchbase.env.endpoints.viewservice.min-endpoints=1 # Minimum number of
sockets per node.
spring.couchbase.env.endpoints.viewservice.max-endpoints=1 # Maximum number of
sockets per node.
spring.couchbase.env.ssl.enabled= # Whether to enable SSL support. Enabled
automatically if a "keyStore" is provided unless specified otherwise.
spring.couchbase.env.ssl.key-store= # Path to the JVM key store that holds the
certificates.
spring.couchbase.env.ssl.key-store-password= # Password used to access the key store.
spring.couchbase.env.timeouts.connect=5000ms # Bucket connections timeouts.
spring.couchbase.env.timeouts.key-value=2500ms # Blocking operations performed on a
specific key timeout.
spring.couchbase.env.timeouts.query=7500ms # N1QL query operations timeout.
spring.couchbase.env.timeouts.socket-connect=1000ms # Socket connect connections
timeout.
spring.couchbase.env.timeouts.view=7500ms # Regular and geospatial view operations
timeout.

# DAO (PersistenceExceptionTranslationAutoConfiguration)
spring.dao.exceptiontranslation.enabled=true # Whether to enable the
PersistenceExceptionTranslationPostProcessor.

# CASSANDRA (CassandraProperties)
spring.data.cassandra.cluster-name= # Name of the Cassandra cluster.
spring.data.cassandra.compression=none # Compression supported by the Cassandra
binary protocol.
spring.data.cassandra.connect-timeout= # Socket option: connection time out.
spring.data.cassandra.consistency-level= # Queries consistency level.
spring.data.cassandra.contact-points=localhost # Cluster node addresses.
spring.data.cassandra.fetch-size= # Queries default fetch size.
spring.data.cassandra.jmx-enabled=false # Whether to enable JMX reporting.
spring.data.cassandra.keyspace-name= # Keyspace name to use.
spring.data.cassandra.port= # Port of the Cassandra server.
spring.data.cassandra.password= # Login password of the server.
spring.data.cassandra.pool.heartbeat-interval=30s # Heartbeat interval after which a
message is sent on an idle connection to make sure it's still alive. If a duration
suffix is not specified, seconds will be used.
spring.data.cassandra.pool.idle-timeout=120s # Idle timeout before an idle connection
is removed. If a duration suffix is not specified, seconds will be used.
spring.data.cassandra.pool.max-queue-size=256 # Maximum number of requests that get
queued if no connection is available.
spring.data.cassandra.pool.pool-timeout=5000ms # Pool timeout when trying to acquire
a connection from a host's pool.
spring.data.cassandra.read-timeout= # Socket option: read time out.
spring.data.cassandra.repositories.type=auto # Type of Cassandra repositories to
enable.
spring.data.cassandra.serial-consistency-level= # Queries serial consistency level.
spring.data.cassandra.schema-action=none # Schema action to take at startup.
spring.data.cassandra.ssl=false # Enable SSL support.
spring.data.cassandra.username= # Login user of the server.

# DATA COUCHBASE (CouchbaseDataProperties)
spring.data.couchbase.auto-index=false # Automatically create views and indexes.
spring.data.couchbase.consistency=read-your-own-writes # Consistency to apply by
default on generated queries.
spring.data.couchbase.repositories.type=auto # Type of Couchbase repositories to
enable.

# ELASTICSEARCH (ElasticsearchProperties)
spring.data.elasticsearch.cluster-name=elasticsearch # Elasticsearch cluster name.
spring.data.elasticsearch.cluster-nodes= # Comma-separated list of cluster node
addresses.
spring.data.elasticsearch.properties.*= # Additional properties used to configure the
client.
spring.data.elasticsearch.repositories.enabled=true # Whether to enable Elasticsearch
repositories.
```

```
# DATA JDBC
spring.data.jdbc.repositories.enabled=true # Whether to enable JDBC repositories.

# DATA LDAP
spring.data.ldap.repositories.enabled=true # Whether to enable LDAP repositories.

# MONGODB (MongoProperties)
spring.data.mongodb.authentication-database= # Authentication database name.
spring.data.mongodb.database= # Database name.
spring.data.mongodb.field-naming-strategy= # Fully qualified name of the
FieldNamingStrategy to use.
spring.data.mongodb.grid-fs-database= # GridFS database name.
spring.data.mongodb.host= # Mongo server host. Cannot be set with URI.
spring.data.mongodb.password= # Login password of the mongo server. Cannot be set
with URI.
spring.data.mongodb.port= # Mongo server port. Cannot be set with URI.
spring.data.mongodb.repositories.type=auto # Type of Mongo repositories to enable.
spring.data.mongodb.uri=mongodb://localhost/test # Mongo database URI. Cannot be set
with host, port and credentials.
spring.data.mongodb.username= # Login user of the mongo server. Cannot be set with
URI.

# DATA REDIS
spring.data.redis.repositories.enabled=true # Whether to enable Redis repositories.

# NEO4J (Neo4jProperties)
spring.data.neo4j.auto-index=none # Auto index mode.
spring.data.neo4j.embedded.enabled=true # Whether to enable embedded mode if the
embedded driver is available.
spring.data.neo4j.open-in-view=true # Register OpenSessionInViewInterceptor. Binds a
Neo4j Session to the thread for the entire processing of the request.
spring.data.neo4j.password= # Login password of the server.
spring.data.neo4j.repositories.enabled=true # Whether to enable Neo4j repositories.
spring.data.neo4j.uri= # URI used by the driver. Auto-detected by default.
spring.data.neo4j.username= # Login user of the server.

# DATA REST (RepositoryRestProperties)
spring.data.rest.base-path= # Base path to be used by Spring Data REST to expose
repository resources.
spring.data.rest.default-media-type= # Content type to use as a default when none is
specified.
spring.data.rest.default-page-size= # Default size of pages.
spring.data.rest.detection-strategy=default # Strategy to use to determine which
repositories get exposed.
spring.data.rest.enable-enum-translation= # Whether to enable enum value translation
through the Spring Data REST default resource bundle.
spring.data.rest.limit-param-name= # Name of the URL query string parameter that
indicates how many results to return at once.
spring.data.rest.max-page-size= # Maximum size of pages.
spring.data.rest.page-param-name= # Name of the URL query string parameter that
indicates what page to return.
spring.data.rest.return-body-on-create= # Whether to return a response body after
creating an entity.
spring.data.rest.return-body-on-update= # Whether to return a response body after
updating an entity.
spring.data.rest.sort-param-name= # Name of the URL query string parameter that
indicates what direction to sort results.

# SOLR (SolrProperties)
spring.data.solr.host=http://127.0.0.1:8983/solr # Solr host. Ignored if "zk-host" is
set.
spring.data.solr.repositories.enabled=true # Whether to enable Solr repositories.
spring.data.solr.zk-host= # ZooKeeper host address in the form HOST:PORT.

# DATA WEB (SpringDataWebProperties)
spring.data.web.pageable.default-page-size=20 # Default page size.
spring.data.web.pageable.max-page-size=2000 # Maximum page size to be accepted.
spring.data.web.pageable.one-indexed-parameters=false # Whether to expose and assume
```

```
1-based page number indexes.
spring.data.web.pageable.page-parameter=page # Page index parameter name.
spring.data.web.pageable.prefix= # General prefix to be prepended to the page number
and page size parameters.
spring.data.web.pageable.qualifier-delimiter=_ # Delimiter to be used between the
qualifier and the actual page number and size properties.
spring.data.web.pageable.size-parameter=size # Page size parameter name.
spring.data.web.sort.sort-parameter=sort # Sort parameter name.

# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.continue-on-error=false # Whether to stop if an error occurs while
initializing the database.
spring.datasource.data= # Data (DML) script resource references.
spring.datasource.data-username= # Username of the database to execute DML scripts
(if different).
spring.datasource.data-password= # Password of the database to execute DML scripts
(if different).
spring.datasource.dbcp2.*= # Commons DBCP2 specific settings
spring.datasource.driver-class-name= # Fully qualified name of the JDBC driver. Auto-
detected based on the URL by default.
spring.datasource.generate-unique-name=false # Whether to generate a random
datasource name.
spring.datasource.hikari.*= # Hikari specific settings
spring.datasource.initialization-mode=embedded # Initialize the datasource with
available DDL and DML scripts.
spring.datasource.jmx-enabled=false # Whether to enable JMX support (if provided by
the underlying pool).
spring.datasource.jndi-name= # JNDI location of the datasource. Class, url, username
& password are ignored when set.
spring.datasource.name= # Name of the datasource. Default to "testdb" when using an
embedded database.
spring.datasource.password= # Login password of the database.
spring.datasource.platform=all # Platform to use in the DDL or DML scripts (such as
schema-${platform}.sql or data-${platform}.sql).
spring.datasource.schema= # Schema (DDL) script resource references.
spring.datasource.schema-username= # Username of the database to execute DDL scripts
(if different).
spring.datasource.schema-password= # Password of the database to execute DDL scripts
(if different).
spring.datasource.separator=; # Statement separator in SQL initialization scripts.
spring.datasource.sql-script-encoding= # SQL scripts encoding.
spring.datasource.tomcat.*= # Tomcat datasource specific settings
spring.datasource.type= # Fully qualified name of the connection pool implementation
to use. By default, it is auto-detected from the classpath.
spring.datasource.url= # JDBC URL of the database.
spring.datasource.username= # Login username of the database.
spring.datasource.xa.data-source-class-name= # XA datasource fully qualified name.
spring.datasource.xa.properties= # Properties to pass to the XA data source.

# JEST (Elasticsearch HTTP client) (JestProperties)
spring.elasticsearch.jest.connection-timeout=3s # Connection timeout.
spring.elasticsearch.jest.multi-threaded=true # Whether to enable connection requests
from multiple execution threads.
spring.elasticsearch.jest.password= # Login password.
spring.elasticsearch.jest.proxy.host= # Proxy host the HTTP client should use.
spring.elasticsearch.jest.proxy.port= # Proxy port the HTTP client should use.
spring.elasticsearch.jest.read-timeout=3s # Read timeout.
spring.elasticsearch.jest.uris=http://localhost:9200 # Comma-separated list of the
Elasticsearch instances to use.
spring.elasticsearch.jest.username= # Login username.

# Elasticsearch REST clients (RestClientProperties)
spring.elasticsearch.rest.password= # Credentials password.
   spring.elasticsearch.rest.uris=http://localhost:9200 # Comma-separated list of the
Elasticsearch instances to use.
   spring.elasticsearch.rest.username= # Credentials username.

# H2 Web Console (H2ConsoleProperties)
spring.h2.console.enabled=false # Whether to enable the console.
```

```
spring.h2.console.path=/h2-console # Path at which the console is available.
spring.h2.console.settings.trace=false # Whether to enable trace output.
spring.h2.console.settings.web-allow-others=false # Whether to enable remote access.

# InfluxDB (InfluxDbProperties)
spring.influx.password= # Login password.
spring.influx.url= # URL of the InfluxDB instance to which to connect.
spring.influx.user= # Login user.

# JOOQ (JooqProperties)
spring.jooq.sql-dialect= # SQL dialect to use. Auto-detected by default.

# JDBC (JdbcProperties)
spring.jdbc.template.fetch-size=-1 # Number of rows that should be fetched from the
database when more rows are needed.
spring.jdbc.template.max-rows=-1 # Maximum number of rows.
spring.jdbc.template.query-timeout= # Query timeout. Default is to use the JDBC
driver's default configuration. If a duration suffix is not specified, seconds will
be used.

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.data.jpa.repositories.bootstrap-mode=default # Bootstrap mode for JPA
repositories.
spring.data.jpa.repositories.enabled=true # Whether to enable JPA repositories.
spring.jpa.database= # Target database to operate on, auto-detected by default. Can
be alternatively set using the "databasePlatform" property.
spring.jpa.database-platform= # Name of the target database to operate on, auto-
detected by default. Can be alternatively set using the "Database" enum.
spring.jpa.generate-ddl=false # Whether to initialize the schema on startup.
spring.jpa.hibernate.ddl-auto= # DDL mode. This is actually a shortcut for the
"hibernate.hbm2ddl.auto" property. Defaults to "create-drop" when using an embedded
database and no schema manager was detected. Otherwise, defaults to "none".
spring.jpa.hibernate.naming.implicit-strategy= # Fully qualified name of the implicit
naming strategy.
spring.jpa.hibernate.naming.physical-strategy= # Fully qualified name of the physical
naming strategy.
spring.jpa.hibernate.use-new-id-generator-mappings= # Whether to use Hibernate's
newer IdentifierGenerator for AUTO, TABLE and SEQUENCE.
spring.jpa.mapping-resources= # Mapping resources (equivalent to "mapping-file"
entries in persistence.xml).
spring.jpa.open-in-view=true # Register OpenEntityManagerInViewInterceptor. Binds a
JPA EntityManager to the thread for the entire processing of the request.
spring.jpa.properties.*= # Additional native properties to set on the JPA provider.
spring.jpa.show-sql=false # Whether to enable logging of SQL statements.

# JTA (JtaAutoConfiguration)
spring.jta.enabled=true # Whether to enable JTA support.
spring.jta.log-dir= # Transaction logs directory.
spring.jta.transaction-manager-id= # Transaction manager unique identifier.

# ATOMIKOS (AtomikosProperties)
spring.jta.atomikos.connectionfactory.borrow-connection-timeout=30 # Timeout, in
seconds, for borrowing connections from the pool.
spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag=true # Whether
to ignore the transacted flag when creating session.
spring.jta.atomikos.connectionfactory.local-transaction-mode=false # Whether local
transactions are desired.
spring.jta.atomikos.connectionfactory.maintenance-interval=60 # The time, in seconds,
between runs of the pool's maintenance thread.
spring.jta.atomikos.connectionfactory.max-idle-time=60 # The time, in seconds, after
which connections are cleaned up from the pool.
spring.jta.atomikos.connectionfactory.max-lifetime=0 # The time, in seconds, that a
connection can be pooled for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.max-pool-size=1 # The maximum size of the pool.
spring.jta.atomikos.connectionfactory.min-pool-size=1 # The minimum size of the pool.
spring.jta.atomikos.connectionfactory.reap-timeout=0 # The reap timeout, in seconds,
for borrowed connections. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.unique-resource-name=jmsConnectionFactory # The
unique name used to identify the resource during recovery.
```

```
spring.jta.atomikos.connectionfactory.xa-connection-factory-class-name= # Vendor-specific implementation of XAConnectionFactory.
spring.jta.atomikos.connectionfactory.xa-properties= # Vendor-specific XA properties.
spring.jta.atomikos.datasource.borrow-connection-timeout=30 # Timeout, in seconds, for borrowing connections from the pool.
spring.jta.atomikos.datasource.concurrent-connection-validation= # Whether to use concurrent connection validation.
spring.jta.atomikos.datasource.default-isolation-level= # Default isolation level of connections provided by the pool.
spring.jta.atomikos.datasource.login-timeout= # Timeout, in seconds, for establishing a database connection.
spring.jta.atomikos.datasource.maintenance-interval=60 # The time, in seconds, between runs of the pool's maintenance thread.
spring.jta.atomikos.datasource.max-idle-time=60 # The time, in seconds, after which connections are cleaned up from the pool.
spring.jta.atomikos.datasource.max-lifetime=0 # The time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.datasource.max-pool-size=1 # The maximum size of the pool.
spring.jta.atomikos.datasource.min-pool-size=1 # The minimum size of the pool.
spring.jta.atomikos.datasource.reap-timeout=0 # The reap timeout, in seconds, for borrowed connections. 0 denotes no limit.
spring.jta.atomikos.datasource.test-query= # SQL query or statement used to validate a connection before returning it.
spring.jta.atomikos.datasource.unique-resource-name=dataSource # The unique name used to identify the resource during recovery.
spring.jta.atomikos.datasource.xa-data-source-class-name= # Vendor-specific implementation of XAConnectionFactory.
spring.jta.atomikos.datasource.xa-properties= # Vendor-specific XA properties.
spring.jta.atomikos.properties.allow-sub-transactions=true # Specify whether sub-transactions are allowed.
spring.jta.atomikos.properties.checkpoint-interval=500 # Interval between checkpoints, expressed as the number of log writes between two checkpoints.
spring.jta.atomikos.properties.default-jta-timeout=10000ms # Default timeout for JTA transactions.
spring.jta.atomikos.properties.default-max-wait-time-on-shutdown=9223372036854775807 # How long should normal shutdown (no-force) wait for transactions to complete.
spring.jta.atomikos.properties.enable-logging=true # Whether to enable disk logging.
spring.jta.atomikos.properties.force-shutdown-on-vm-exit=false # Whether a VM shutdown should trigger forced shutdown of the transaction core.
spring.jta.atomikos.properties.log-base-dir= # Directory in which the log files should be stored.
spring.jta.atomikos.properties.log-base-name=tmlog # Transactions log file base name.
spring.jta.atomikos.properties.max-actives=50 # Maximum number of active transactions.
spring.jta.atomikos.properties.max-timeout=300000ms # Maximum timeout that can be allowed for transactions.
spring.jta.atomikos.properties.recovery.delay=10000ms # Delay between two recovery scans.
spring.jta.atomikos.properties.recovery.forget-orphaned-log-entries-delay=86400000ms # Delay after which recovery can cleanup pending ('orphaned') log entries.
spring.jta.atomikos.properties.recovery.max-retries=5 # Number of retry attempts to commit the transaction before throwing an exception.
spring.jta.atomikos.properties.recovery.retry-interval=10000ms # Delay between retry attempts.
spring.jta.atomikos.properties.serial-jta-transactions=true # Whether sub-transactions should be joined when possible.
spring.jta.atomikos.properties.service= # Transaction manager implementation that should be started.
spring.jta.atomikos.properties.threaded-two-phase-commit=false # Whether to use different (and concurrent) threads for two-phase commit on the participating resources.
spring.jta.atomikos.properties.transaction-manager-unique-name= # The transaction manager's unique name.

# BITRONIX
spring.jta.bitronix.connectionfactory.acquire-increment=1 # Number of connections to create when growing the pool.
spring.jta.bitronix.connectionfactory.acquisition-interval=1 # Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was
```

```
acquired.
spring.jta.bitronix.connectionfactory.acquisition-timeout=30 # Timeout, in seconds, for acquiring connections from the pool.
spring.jta.bitronix.connectionfactory.allow-local-transactions=true # Whether the transaction manager should allow mixing XA and non-XA transactions.
spring.jta.bitronix.connectionfactory.apply-transaction-timeout=false # Whether the transaction timeout should be set on the XAResource when it is enlisted.
spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled=true # Whether resources should be enlisted and delisted automatically.
spring.jta.bitronix.connectionfactory.cache-producers-consumers=true # Whether producers and consumers should be cached.
spring.jta.bitronix.connectionfactory.class-name= # Underlying implementation class name of the XA resource.
spring.jta.bitronix.connectionfactory.defer-connection-release=true # Whether the provider can run many transactions on the same connection and supports transaction interleaving.
spring.jta.bitronix.connectionfactory.disabled= # Whether this resource is disabled, meaning it's temporarily forbidden to acquire a connection from its pool.
spring.jta.bitronix.connectionfactory.driver-properties= # Properties that should be set on the underlying implementation.
spring.jta.bitronix.connectionfactory.failed= # Mark this resource producer as failed.
spring.jta.bitronix.connectionfactory.ignore-recovery-failures=false # Whether recovery failures should be ignored.
spring.jta.bitronix.connectionfactory.max-idle-time=60 # The time, in seconds, after which connections are cleaned up from the pool.
spring.jta.bitronix.connectionfactory.max-pool-size=10 # The maximum size of the pool. 0 denotes no limit.
spring.jta.bitronix.connectionfactory.min-pool-size=0 # The minimum size of the pool.
spring.jta.bitronix.connectionfactory.password= # The password to use to connect to the JMS provider.
spring.jta.bitronix.connectionfactory.share-transaction-connections=false #  Whether connections in the ACCESSIBLE state can be shared within the context of a transaction.
spring.jta.bitronix.connectionfactory.test-connections=true # Whether connections should be tested when acquired from the pool.
spring.jta.bitronix.connectionfactory.two-pc-ordering-position=1 # The position that this resource should take during two-phase commit (always first is Integer.MIN_VALUE, always last is Integer.MAX_VALUE).
spring.jta.bitronix.connectionfactory.unique-name=jmsConnectionFactory # The unique name used to identify the resource during recovery.
spring.jta.bitronix.connectionfactory.use-tm-join=true # Whether TMJOIN should be used when starting XAResources.
spring.jta.bitronix.connectionfactory.user= # The user to use to connect to the JMS provider.
spring.jta.bitronix.datasource.acquire-increment=1 # Number of connections to create when growing the pool.
spring.jta.bitronix.datasource.acquisition-interval=1 # Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.
spring.jta.bitronix.datasource.acquisition-timeout=30 # Timeout, in seconds, for acquiring connections from the pool.
spring.jta.bitronix.datasource.allow-local-transactions=true # Whether the transaction manager should allow mixing XA and non-XA transactions.
spring.jta.bitronix.datasource.apply-transaction-timeout=false # Whether the transaction timeout should be set on the XAResource when it is enlisted.
spring.jta.bitronix.datasource.automatic-enlisting-enabled=true # Whether resources should be enlisted and delisted automatically.
spring.jta.bitronix.datasource.class-name= # Underlying implementation class name of the XA resource.
spring.jta.bitronix.datasource.cursor-holdability= # The default cursor holdability for connections.
spring.jta.bitronix.datasource.defer-connection-release=true # Whether the database can run many transactions on the same connection and supports transaction interleaving.
spring.jta.bitronix.datasource.disabled= # Whether this resource is disabled, meaning it's temporarily forbidden to acquire a connection from its pool.
spring.jta.bitronix.datasource.driver-properties= # Properties that should be set on the underlying implementation.
spring.jta.bitronix.datasource.enable-jdbc4-connection-test= # Whether
```

```
Connection.isValid() is called when acquiring a connection from the pool.
spring.jta.bitronix.datasource.failed= # Mark this resource producer as failed.
spring.jta.bitronix.datasource.ignore-recovery-failures=false # Whether recovery
failures should be ignored.
spring.jta.bitronix.datasource.isolation-level= # The default isolation level for
connections.
spring.jta.bitronix.datasource.local-auto-commit= # The default auto-commit mode for
local transactions.
spring.jta.bitronix.datasource.login-timeout= # Timeout, in seconds, for establishing
a database connection.
spring.jta.bitronix.datasource.max-idle-time=60 # The time, in seconds, after which
connections are cleaned up from the pool.
spring.jta.bitronix.datasource.max-pool-size=10 # The maximum size of the pool. 0
denotes no limit.
spring.jta.bitronix.datasource.min-pool-size=0 # The minimum size of the pool.
spring.jta.bitronix.datasource.prepared-statement-cache-size=0 # The target size of
the prepared statement cache. 0 disables the cache.
spring.jta.bitronix.datasource.share-transaction-connections=false #  Whether
connections in the ACCESSIBLE state can be shared within the context of a
transaction.
spring.jta.bitronix.datasource.test-query= # SQL query or statement used to validate
a connection before returning it.
spring.jta.bitronix.datasource.two-pc-ordering-position=1 # The position that this
resource should take during two-phase commit (always first is Integer.MIN_VALUE, and
always last is Integer.MAX_VALUE).
spring.jta.bitronix.datasource.unique-name=dataSource # The unique name used to
identify the resource during recovery.
spring.jta.bitronix.datasource.use-tm-join=true # Whether TMJOIN should be used when
starting XAResources.
spring.jta.bitronix.properties.allow-multiple-lrc=false # Whether to allow multiple
LRC resources to be enlisted into the same transaction.
spring.jta.bitronix.properties.asynchronous2-pc=false # Whether to enable
asynchronously execution of two phase commit.
spring.jta.bitronix.properties.background-recovery-interval-seconds=60 # Interval in
seconds at which to run the recovery process in the background.
spring.jta.bitronix.properties.current-node-only-recovery=true # Whether to recover
only the current node.
spring.jta.bitronix.properties.debug-zero-resource-transaction=false # Whether to log
the creation and commit call stacks of transactions executed without a single
enlisted resource.
spring.jta.bitronix.properties.default-transaction-timeout=60 # Default transaction
timeout, in seconds.
spring.jta.bitronix.properties.disable-jmx=false # Whether to enable JMX support.
spring.jta.bitronix.properties.exception-analyzer= # Set the fully qualified name of
the exception analyzer implementation to use.
spring.jta.bitronix.properties.filter-log-status=false # Whether to enable filtering
of logs so that only mandatory logs are written.
spring.jta.bitronix.properties.force-batching-enabled=true #  Whether disk forces are
batched.
spring.jta.bitronix.properties.forced-write-enabled=true # Whether logs are forced to
disk.
spring.jta.bitronix.properties.graceful-shutdown-interval=60 # Maximum amount of
seconds the TM waits for transactions to get done before aborting them at shutdown
time.
spring.jta.bitronix.properties.jndi-transaction-synchronization-registry-name= # JNDI
name of the TransactionSynchronizationRegistry.
spring.jta.bitronix.properties.jndi-user-transaction-name= # JNDI name of the
UserTransaction.
spring.jta.bitronix.properties.journal=disk # Name of the journal. Can be 'disk',
'null', or a class name.
spring.jta.bitronix.properties.log-part1-filename=btm1.tlog # Name of the first
fragment of the journal.
spring.jta.bitronix.properties.log-part2-filename=btm2.tlog # Name of the second
fragment of the journal.
spring.jta.bitronix.properties.max-log-size-in-mb=2 # Maximum size in megabytes of
the journal fragments.
spring.jta.bitronix.properties.resource-configuration-filename= # ResourceLoader
configuration file name.
spring.jta.bitronix.properties.server-id= # ASCII ID that must uniquely identify this
```

```
TM instance. Defaults to the machine's IP address.
spring.jta.bitronix.properties.skip-corrupted-logs=false # Skip corrupted
transactions log entries.
spring.jta.bitronix.properties.warn-about-zero-resource-transaction=true # Whether to
log a warning for transactions executed without a single enlisted resource.

# EMBEDDED MONGODB (EmbeddedMongoProperties)
spring.mongodb.embedded.features=sync_delay # Comma-separated list of features to
enable.
spring.mongodb.embedded.storage.database-dir= # Directory used for data storage.
spring.mongodb.embedded.storage.oplog-size= # Maximum size of the oplog.
spring.mongodb.embedded.storage.repl-set-name= # Name of the replica set.
spring.mongodb.embedded.version=3.5.5 # Version of Mongo to use.

# REDIS (RedisProperties)
spring.redis.cluster.max-redirects= # Maximum number of redirects to follow when
executing commands across the cluster.
spring.redis.cluster.nodes= # Comma-separated list of "host:port" pairs to bootstrap
from.
spring.redis.database=0 # Database index used by the connection factory.
spring.redis.url= # Connection URL. Overrides host, port, and password. User is
ignored. Example: redis://user:password@example.com:6379
spring.redis.host=localhost # Redis server host.
spring.redis.jedis.pool.max-active=8 # Maximum number of connections that can be
allocated by the pool at a given time. Use a negative value for no limit.
spring.redis.jedis.pool.max-idle=8 # Maximum number of "idle" connections in the
pool. Use a negative value to indicate an unlimited number of idle connections.
spring.redis.jedis.pool.max-wait=-1ms # Maximum amount of time a connection
allocation should block before throwing an exception when the pool is exhausted. Use
a negative value to block indefinitely.
spring.redis.jedis.pool.min-idle=0 # Target for the minimum number of idle
connections to maintain in the pool. This setting only has an effect if it is
positive.
spring.redis.lettuce.pool.max-active=8 # Maximum number of connections that can be
allocated by the pool at a given time. Use a negative value for no limit.
spring.redis.lettuce.pool.max-idle=8 # Maximum number of "idle" connections in the
pool. Use a negative value to indicate an unlimited number of idle connections.
spring.redis.lettuce.pool.max-wait=-1ms # Maximum amount of time a connection
allocation should block before throwing an exception when the pool is exhausted. Use
a negative value to block indefinitely.
spring.redis.lettuce.pool.min-idle=0 # Target for the minimum number of idle
connections to maintain in the pool. This setting only has an effect if it is
positive.
spring.redis.lettuce.shutdown-timeout=100ms # Shutdown timeout.
spring.redis.password= # Login password of the redis server.
spring.redis.port=6379 # Redis server port.
spring.redis.sentinel.master= # Name of the Redis server.
spring.redis.sentinel.nodes= # Comma-separated list of "host:port" pairs.
spring.redis.ssl=false # Whether to enable SSL support.
spring.redis.timeout= # Connection timeout.

# TRANSACTION (TransactionProperties)
spring.transaction.default-timeout= # Default transaction timeout. If a duration
suffix is not specified, seconds will be used.
spring.transaction.rollback-on-commit-failure= # Whether to roll back on commit
failures.



# ----------------------------------------
# INTEGRATION PROPERTIES
# ----------------------------------------

# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url= # URL of the ActiveMQ broker. Auto-generated by default.
spring.activemq.close-timeout=15s # Time to wait before considering a close complete.
spring.activemq.in-memory=true # Whether the default broker URL should be in memory.
Ignored if an explicit broker has been specified.
spring.activemq.non-blocking-redelivery=false # Whether to stop message delivery
```

```
before re-delivering messages from a rolled back transaction. This implies that
message order is not preserved when this is enabled.
spring.activemq.password= # Login password of the broker.
spring.activemq.send-timeout=0ms # Time to wait on message sends for a response. Set
it to 0 to wait forever.
spring.activemq.user= # Login user of the broker.
spring.activemq.packages.trust-all= # Whether to trust all packages.
spring.activemq.packages.trusted= # Comma-separated list of specific packages to
trust (when not trusting all packages).
spring.activemq.pool.block-if-full=true # Whether to block when a connection is
requested and the pool is full. Set it to false to throw a "JMSException" instead.
spring.activemq.pool.block-if-full-timeout=-1ms # Blocking period before throwing an
exception if the pool is still full.
spring.activemq.pool.enabled=false # Whether a JmsPoolConnectionFactory should be
created, instead of a regular ConnectionFactory.
spring.activemq.pool.idle-timeout=30s # Connection idle timeout.
spring.activemq.pool.max-connections=1 # Maximum number of pooled connections.
spring.activemq.pool.max-sessions-per-connection=500 # Maximum number of pooled
sessions per connection in the pool.
spring.activemq.pool.time-between-expiration-check=-1ms # Time to sleep between runs
of the idle connection eviction thread. When negative, no idle connection eviction
thread runs.
spring.activemq.pool.use-anonymous-producers=true # Whether to use only one anonymous
"MessageProducer" instance. Set it to false to create one "MessageProducer" every
time one is required.

# ARTEMIS (ArtemisProperties)
spring.artemis.embedded.cluster-password= # Cluster password. Randomly generated on
startup by default.
spring.artemis.embedded.data-directory= # Journal file directory. Not necessary if
persistence is turned off.
spring.artemis.embedded.enabled=true # Whether to enable embedded mode if the Artemis
server APIs are available.
spring.artemis.embedded.persistent=false # Whether to enable persistent store.
spring.artemis.embedded.queues= # Comma-separated list of queues to create on
startup.
spring.artemis.embedded.server-id= # Server ID. By default, an auto-incremented
counter is used.
spring.artemis.embedded.topics= # Comma-separated list of topics to create on
startup.
spring.artemis.host=localhost # Artemis broker host.
spring.artemis.mode= # Artemis deployment mode, auto-detected by default.
spring.artemis.password= # Login password of the broker.
spring.artemis.pool.block-if-full=true # Whether to block when a connection is
requested and the pool is full. Set it to false to throw a "JMSException" instead.
spring.artemis.pool.block-if-full-timeout=-1ms # Blocking period before throwing an
exception if the pool is still full.
spring.artemis.pool.enabled=false # Whether a JmsPoolConnectionFactory should be
created, instead of a regular ConnectionFactory.
spring.artemis.pool.idle-timeout=30s # Connection idle timeout.
spring.artemis.pool.max-connections=1 # Maximum number of pooled connections.
spring.artemis.pool.max-sessions-per-connection=500 # Maximum number of pooled
sessions per connection in the pool.
spring.artemis.pool.time-between-expiration-check=-1ms # Time to sleep between runs
of the idle connection eviction thread. When negative, no idle connection eviction
thread runs.
spring.artemis.pool.use-anonymous-producers=true # Whether to use only one anonymous
"MessageProducer" instance. Set it to false to create one "MessageProducer" every
time one is required.
spring.artemis.port=61616 # Artemis broker port.
spring.artemis.user= # Login user of the broker.

# SPRING BATCH (BatchProperties)
spring.batch.initialize-schema=embedded # Database schema initialization mode.
spring.batch.job.enabled=true # Execute all Spring Batch jobs in the context on
startup.
spring.batch.job.names= # Comma-separated list of job names to execute on startup
(for instance, `job1,job2`). By default, all Jobs found in the context are executed.
spring.batch.schema=classpath:org/springframework/batch/core/schema-@@platform@@.sql
```

```
# Path to the SQL file to use to initialize the database schema.
spring.batch.table-prefix= # Table prefix for all the batch meta-data tables.

# SPRING INTEGRATION (IntegrationProperties)
spring.integration.jdbc.initialize-schema=embedded # Database schema initialization
mode.
spring.integration.jdbc.schema=classpath:org/springframework/integration/jdbc/schema-
@@platform@@.sql # Path to the SQL file to use to initialize the database schema.

# JMS (JmsProperties)
spring.jms.cache.consumers=false # Whether to cache message consumers.
spring.jms.cache.enabled=true # Whether to cache sessions.
spring.jms.cache.producers=true # Whether to cache message producers.
spring.jms.cache.session-cache-size=1 # Size of the session cache (per JMS Session
type).
spring.jms.jndi-name= # Connection factory JNDI name. When set, takes precedence to
others connection factory auto-configurations.
spring.jms.listener.acknowledge-mode= # Acknowledge mode of the container. By
default, the listener is transacted with automatic acknowledgment.
spring.jms.listener.auto-startup=true # Start the container automatically on startup.
spring.jms.listener.concurrency= # Minimum number of concurrent consumers.
spring.jms.listener.max-concurrency= # Maximum number of concurrent consumers.
spring.jms.pub-sub-domain=false # Whether the default destination type is topic.
spring.jms.template.default-destination= # Default destination to use on send and
receive operations that do not have a destination parameter.
spring.jms.template.delivery-delay= # Delivery delay to use for send calls.
spring.jms.template.delivery-mode= # Delivery mode. Enables QoS (Quality of Service)
when set.
spring.jms.template.priority= # Priority of a message when sending. Enables QoS
(Quality of Service) when set.
spring.jms.template.qos-enabled= # Whether to enable explicit QoS (Quality of
Service) when sending a message.
spring.jms.template.receive-timeout= # Timeout to use for receive calls.
spring.jms.template.time-to-live= # Time-to-live of a message when sending. Enables
QoS (Quality of Service) when set.

# APACHE KAFKA (KafkaProperties)
spring.kafka.admin.client-id= # ID to pass to the server when making requests. Used
for server-side logging.
spring.kafka.admin.fail-fast=false # Whether to fail fast if the broker is not
available on startup.
spring.kafka.admin.properties.*= # Additional admin-specific properties used to
configure the client.
spring.kafka.admin.ssl.key-password= # Password of the private key in the key store
file.
spring.kafka.admin.ssl.key-store-location= # Location of the key store file.
spring.kafka.admin.ssl.key-store-password= # Store password for the key store file.
spring.kafka.admin.ssl.key-store-type= # Type of the key store.
spring.kafka.admin.ssl.protocol= # SSL protocol to use.
spring.kafka.admin.ssl.trust-store-location= # Location of the trust store file.
spring.kafka.admin.ssl.trust-store-password= # Store password for the trust store
file.
spring.kafka.admin.ssl.trust-store-type= # Type of the trust store.
spring.kafka.bootstrap-servers= # Comma-delimited list of host:port pairs to use for
establishing the initial connections to the Kafka cluster. Applies to all components
unless overridden.
spring.kafka.client-id= # ID to pass to the server when making requests. Used for
server-side logging.
spring.kafka.consumer.auto-commit-interval= # Frequency with which the consumer
offsets are auto-committed to Kafka if 'enable.auto.commit' is set to true.
spring.kafka.consumer.auto-offset-reset= # What to do when there is no initial offset
in Kafka or if the current offset no longer exists on the server.
spring.kafka.consumer.bootstrap-servers= # Comma-delimited list of host:port pairs to
use for establishing the initial connections to the Kafka cluster. Overrides the
global property, for consumers.
spring.kafka.consumer.client-id= # ID to pass to the server when making requests.
Used for server-side logging.
spring.kafka.consumer.enable-auto-commit= # Whether the consumer's offset is
periodically committed in the background.
```

```
spring.kafka.consumer.fetch-max-wait= # Maximum amount of time the server blocks
before answering the fetch request if there isn't sufficient data to immediately
satisfy the requirement given by "fetch-min-size".
spring.kafka.consumer.fetch-min-size= # Minimum amount of data the server should
return for a fetch request.
spring.kafka.consumer.group-id= # Unique string that identifies the consumer group to
which this consumer belongs.
spring.kafka.consumer.heartbeat-interval= # Expected time between heartbeats to the
consumer coordinator.
spring.kafka.consumer.key-deserializer= # Deserializer class for keys.
spring.kafka.consumer.max-poll-records= # Maximum number of records returned in a
single call to poll().
spring.kafka.consumer.properties.*= # Additional consumer-specific properties used to
configure the client.
spring.kafka.consumer.ssl.key-password= # Password of the private key in the key
store file.
spring.kafka.consumer.ssl.key-store-location= # Location of the key store file.
spring.kafka.consumer.ssl.key-store-password= # Store password for the key store
file.
spring.kafka.consumer.ssl.key-store-type= # Type of the key store.
spring.kafka.consumer.ssl.protocol= # SSL protocol to use.
spring.kafka.consumer.ssl.trust-store-location= # Location of the trust store file.
spring.kafka.consumer.ssl.trust-store-password= # Store password for the trust store
file.
spring.kafka.consumer.ssl.trust-store-type= # Type of the trust store.
spring.kafka.consumer.value-deserializer= # Deserializer class for values.
spring.kafka.jaas.control-flag=required # Control flag for login configuration.
spring.kafka.jaas.enabled=false # Whether to enable JAAS configuration.
spring.kafka.jaas.login-module=com.sun.security.auth.module.Krb5LoginModule # Login
module.
spring.kafka.jaas.options= # Additional JAAS options.
spring.kafka.listener.ack-count= # Number of records between offset commits when
ackMode is "COUNT" or "COUNT_TIME".
spring.kafka.listener.ack-mode= # Listener AckMode. See the spring-kafka
documentation.
spring.kafka.listener.ack-time= # Time between offset commits when ackMode is "TIME"
or "COUNT_TIME".
spring.kafka.listener.client-id= # Prefix for the listener's consumer client.id
property.
spring.kafka.listener.concurrency= # Number of threads to run in the listener
containers.
spring.kafka.listener.idle-event-interval= # Time between publishing idle consumer
events (no data received).
spring.kafka.listener.log-container-config= # Whether to log the container
configuration during initialization (INFO level).
spring.kafka.listener.monitor-interval= # Time between checks for non-responsive
consumers. If a duration suffix is not specified, seconds will be used.
spring.kafka.listener.no-poll-threshold= # Multiplier applied to "pollTimeout" to
determine if a consumer is non-responsive.
spring.kafka.listener.poll-timeout= # Timeout to use when polling the consumer.
spring.kafka.listener.type=single # Listener type.
spring.kafka.producer.acks= # Number of acknowledgments the producer requires the
leader to have received before considering a request complete.
spring.kafka.producer.batch-size= # Default batch size.
spring.kafka.producer.bootstrap-servers= # Comma-delimited list of host:port pairs to
use for establishing the initial connections to the Kafka cluster. Overrides the
global property, for producers.
spring.kafka.producer.buffer-memory= # Total memory size the producer can use to
buffer records waiting to be sent to the server.
spring.kafka.producer.client-id= # ID to pass to the server when making requests.
Used for server-side logging.
spring.kafka.producer.compression-type= # Compression type for all data generated by
the producer.
spring.kafka.producer.key-serializer= # Serializer class for keys.
spring.kafka.producer.properties.*= # Additional producer-specific properties used to
configure the client.
spring.kafka.producer.retries= # When greater than zero, enables retrying of failed
sends.
spring.kafka.producer.ssl.key-password= # Password of the private key in the key
```

```
store file.
spring.kafka.producer.ssl.key-store-location= # Location of the key store file.
spring.kafka.producer.ssl.key-store-password= # Store password for the key store
file.
spring.kafka.producer.ssl.key-store-type= # Type of the key store.
spring.kafka.producer.ssl.protocol= # SSL protocol to use.
spring.kafka.producer.ssl.trust-store-location= # Location of the trust store file.
spring.kafka.producer.ssl.trust-store-password= # Store password for the trust store
file.
spring.kafka.producer.ssl.trust-store-type= # Type of the trust store.
spring.kafka.producer.transaction-id-prefix= # When non empty, enables transaction
support for producer.
spring.kafka.producer.value-serializer= # Serializer class for values.
spring.kafka.properties.*= # Additional properties, common to producers and
consumers, used to configure the client.
spring.kafka.ssl.key-password= # Password of the private key in the key store file.
spring.kafka.ssl.key-store-location= # Location of the key store file.
spring.kafka.ssl.key-store-password= # Store password for the key store file.
spring.kafka.ssl.key-store-type= # Type of the key store.
spring.kafka.ssl.protocol= # SSL protocol to use.
spring.kafka.ssl.trust-store-location= # Location of the trust store file.
spring.kafka.ssl.trust-store-password= # Store password for the trust store file.
spring.kafka.ssl.trust-store-type= # Type of the trust store.
spring.kafka.streams.application-id= # Kafka streams application.id property; default
spring.application.name.
spring.kafka.streams.auto-startup=true # Whether or not to auto-start the streams
factory bean.
spring.kafka.streams.bootstrap-servers= # Comma-delimited list of host:port pairs to
use for establishing the initial connections to the Kafka cluster. Overrides the
global property, for streams.
spring.kafka.streams.cache-max-size-buffering= # Maximum memory size to be used for
buffering across all threads.
spring.kafka.streams.client-id= # ID to pass to the server when making requests. Used
for server-side logging.
spring.kafka.streams.properties.*= # Additional Kafka properties used to configure
the streams.
spring.kafka.streams.replication-factor= # The replication factor for change log
topics and repartition topics created by the stream processing application.
spring.kafka.streams.ssl.key-password= # Password of the private key in the key store
file.
spring.kafka.streams.ssl.key-store-location= # Location of the key store file.
spring.kafka.streams.ssl.key-store-password= # Store password for the key store file.
spring.kafka.streams.ssl.key-store-type= # Type of the key store.
spring.kafka.streams.ssl.protocol= # SSL protocol to use.
spring.kafka.streams.ssl.trust-store-location= # Location of the trust store file.
spring.kafka.streams.ssl.trust-store-password= # Store password for the trust store
file.
spring.kafka.streams.ssl.trust-store-type= # Type of the trust store.
spring.kafka.streams.state-dir= # Directory location for the state store.
spring.kafka.template.default-topic= # Default topic to which messages are sent.

# RABBIT (RabbitProperties)
spring.rabbitmq.addresses= # Comma-separated list of addresses to which the client
should connect.
spring.rabbitmq.cache.channel.checkout-timeout= # Duration to wait to obtain a
channel if the cache size has been reached.
spring.rabbitmq.cache.channel.size= # Number of channels to retain in the cache.
spring.rabbitmq.cache.connection.mode=channel # Connection factory cache mode.
spring.rabbitmq.cache.connection.size= # Number of connections to cache.
spring.rabbitmq.connection-timeout= # Connection timeout. Set it to zero to wait
forever.
spring.rabbitmq.dynamic=true # Whether to create an AmqpAdmin bean.
spring.rabbitmq.host=localhost # RabbitMQ host.
spring.rabbitmq.listener.direct.acknowledge-mode= # Acknowledge mode of container.
spring.rabbitmq.listener.direct.auto-startup=true # Whether to start the container
automatically on startup.
spring.rabbitmq.listener.direct.consumers-per-queue= # Number of consumers per queue.
spring.rabbitmq.listener.direct.default-requeue-rejected= # Whether rejected
deliveries are re-queued by default.
```

```
spring.rabbitmq.listener.direct.idle-event-interval= # How often idle container events should be published.
spring.rabbitmq.listener.direct.missing-queues-fatal=false # Whether to fail if the queues declared by the container are not available on the broker.
spring.rabbitmq.listener.direct.prefetch= # Maximum number of unacknowledged messages that can be outstanding at each consumer.
spring.rabbitmq.listener.direct.retry.enabled=false # Whether publishing retries are enabled.
spring.rabbitmq.listener.direct.retry.initial-interval=1000ms # Duration between the first and second attempt to deliver a message.
spring.rabbitmq.listener.direct.retry.max-attempts=3 # Maximum number of attempts to deliver a message.
spring.rabbitmq.listener.direct.retry.max-interval=10000ms # Maximum duration between attempts.
spring.rabbitmq.listener.direct.retry.multiplier=1 # Multiplier to apply to the previous retry interval.
spring.rabbitmq.listener.direct.retry.stateless=true # Whether retries are stateless or stateful.
spring.rabbitmq.listener.simple.acknowledge-mode= # Acknowledge mode of container.
spring.rabbitmq.listener.simple.auto-startup=true # Whether to start the container automatically on startup.
spring.rabbitmq.listener.simple.concurrency= # Minimum number of listener invoker threads.
spring.rabbitmq.listener.simple.default-requeue-rejected= # Whether rejected deliveries are re-queued by default.
spring.rabbitmq.listener.simple.idle-event-interval= # How often idle container events should be published.
spring.rabbitmq.listener.simple.max-concurrency= # Maximum number of listener invoker threads.
spring.rabbitmq.listener.simple.missing-queues-fatal=true # Whether to fail if the queues declared by the container are not available on the broker and/or whether to stop the container if one or more queues are deleted at runtime.
spring.rabbitmq.listener.simple.prefetch= # Maximum number of unacknowledged messages that can be outstanding at each consumer.
spring.rabbitmq.listener.simple.retry.enabled=false # Whether publishing retries are enabled.
spring.rabbitmq.listener.simple.retry.initial-interval=1000ms # Duration between the first and second attempt to deliver a message.
spring.rabbitmq.listener.simple.retry.max-attempts=3 # Maximum number of attempts to deliver a message.
spring.rabbitmq.listener.simple.retry.max-interval=10000ms # Maximum duration between attempts.
spring.rabbitmq.listener.simple.retry.multiplier=1 # Multiplier to apply to the previous retry interval.
spring.rabbitmq.listener.simple.retry.stateless=true # Whether retries are stateless or stateful.
spring.rabbitmq.listener.simple.transaction-size= # Number of messages to be processed between acks when the acknowledge mode is AUTO. If larger than prefetch, prefetch will be increased to this value.
spring.rabbitmq.listener.type=simple # Listener container type.
spring.rabbitmq.password=guest # Login to authenticate against the broker.
spring.rabbitmq.port=5672 # RabbitMQ port.
spring.rabbitmq.publisher-confirms=false # Whether to enable publisher confirms.
spring.rabbitmq.publisher-returns=false # Whether to enable publisher returns.
spring.rabbitmq.requested-heartbeat= # Requested heartbeat timeout; zero for none. If a duration suffix is not specified, seconds will be used.
spring.rabbitmq.ssl.algorithm= # SSL algorithm to use. By default, configured by the Rabbit client library.
spring.rabbitmq.ssl.enabled=false # Whether to enable SSL support.
spring.rabbitmq.ssl.key-store= # Path to the key store that holds the SSL certificate.
spring.rabbitmq.ssl.key-store-password= # Password used to access the key store.
spring.rabbitmq.ssl.key-store-type=PKCS12 # Key store type.
spring.rabbitmq.ssl.trust-store= # Trust store that holds SSL certificates.
spring.rabbitmq.ssl.trust-store-password= # Password used to access the trust store.
spring.rabbitmq.ssl.trust-store-type=JKS # Trust store type.
spring.rabbitmq.ssl.validate-server-certificate=true # Whether to enable server side certificate validation.
spring.rabbitmq.ssl.verify-hostname=true # Whether to enable hostname verification.
```

```
spring.rabbitmq.template.default-receive-queue= # Name of the default queue to
receive messages from when none is specified explicitly.
spring.rabbitmq.template.exchange= # Name of the default exchange to use for send
operations.
spring.rabbitmq.template.mandatory= # Whether to enable mandatory messages.
spring.rabbitmq.template.receive-timeout= # Timeout for `receive()` operations.
spring.rabbitmq.template.reply-timeout= # Timeout for `sendAndReceive()` operations.
spring.rabbitmq.template.retry.enabled=false # Whether publishing retries are
enabled.
spring.rabbitmq.template.retry.initial-interval=1000ms # Duration between the first
and second attempt to deliver a message.
spring.rabbitmq.template.retry.max-attempts=3 # Maximum number of attempts to deliver
a message.
spring.rabbitmq.template.retry.max-interval=10000ms # Maximum duration between
attempts.
spring.rabbitmq.template.retry.multiplier=1 # Multiplier to apply to the previous
retry interval.
spring.rabbitmq.template.routing-key= # Value of a default routing key to use for
send operations.
spring.rabbitmq.username=guest # Login user to authenticate to the broker.
spring.rabbitmq.virtual-host= # Virtual host to use when connecting to the broker.


# ----------------------------------------
# ACTUATOR PROPERTIES
# ----------------------------------------

# MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.server.add-application-context-header=false # Add the "X-Application-
Context" HTTP header in each response.
management.server.address= # Network address to which the management endpoints should
bind. Requires a custom management.server.port.
management.server.port= # Management endpoint HTTP port (uses the same port as the
application by default). Configure a different port to use management-specific SSL.
management.server.servlet.context-path= # Management endpoint context-path (for
instance, `/management`). Requires a custom management.server.port.
management.server.ssl.ciphers= # Supported SSL ciphers.
management.server.ssl.client-auth= # Client authentication mode.
management.server.ssl.enabled=true # Whether to enable SSL support.
management.server.ssl.enabled-protocols= # Enabled SSL protocols.
management.server.ssl.key-alias= # Alias that identifies the key in the key store.
management.server.ssl.key-password= # Password used to access the key in the key
store.
management.server.ssl.key-store= # Path to the key store that holds the SSL
certificate (typically a jks file).
management.server.ssl.key-store-password= # Password used to access the key store.
management.server.ssl.key-store-provider= # Provider for the key store.
management.server.ssl.key-store-type= # Type of the key store.
management.server.ssl.protocol=TLS # SSL protocol to use.
management.server.ssl.trust-store= # Trust store that holds SSL certificates.
management.server.ssl.trust-store-password= # Password used to access the trust
store.
management.server.ssl.trust-store-provider= # Provider for the trust store.
management.server.ssl.trust-store-type= # Type of the trust store.

# CLOUDFOUNDRY
management.cloudfoundry.enabled=true # Whether to enable extended Cloud Foundry
actuator endpoints.
management.cloudfoundry.skip-ssl-validation=false # Whether to skip SSL verification
for Cloud Foundry actuator endpoint security calls.

# ENDPOINTS GENERAL CONFIGURATION
management.endpoints.enabled-by-default= # Whether to enable or disable all endpoints
by default.

# ENDPOINTS JMX CONFIGURATION (JmxEndpointProperties)
management.endpoints.jmx.domain=org.springframework.boot # Endpoints JMX domain name.
Fallback to 'spring.jmx.default-domain' if set.
management.endpoints.jmx.exposure.include=* # Endpoint IDs that should be included or
```

```
'*' for all.
management.endpoints.jmx.exposure.exclude= # Endpoint IDs that should be excluded or
'*' for all.
management.endpoints.jmx.static-names= # Additional static properties to append to
all ObjectNames of MBeans representing Endpoints.

# ENDPOINTS WEB CONFIGURATION (WebEndpointProperties)
management.endpoints.web.exposure.include=health,info # Endpoint IDs that should be
included or '*' for all.
management.endpoints.web.exposure.exclude= # Endpoint IDs that should be excluded or
'*' for all.
management.endpoints.web.base-path=/actuator # Base path for Web endpoints. Relative
to server.servlet.context-path or management.server.servlet.context-path if
management.server.port is configured.
management.endpoints.web.path-mapping= # Mapping between endpoint IDs and the path
that should expose them.

# ENDPOINTS CORS CONFIGURATION (CorsEndpointProperties)
management.endpoints.web.cors.allow-credentials= # Whether credentials are supported.
When not set, credentials are not supported.
management.endpoints.web.cors.allowed-headers= # Comma-separated list of headers to
allow in a request. '*' allows all headers.
management.endpoints.web.cors.allowed-methods= # Comma-separated list of methods to
allow. '*' allows all methods. When not set, defaults to GET.
management.endpoints.web.cors.allowed-origins= # Comma-separated list of origins to
allow. '*' allows all origins. When not set, CORS support is disabled.
management.endpoints.web.cors.exposed-headers= # Comma-separated list of headers to
include in a response.
management.endpoints.web.cors.max-age=1800s # How long the response from a pre-flight
request can be cached by clients. If a duration suffix is not specified, seconds will
be used.

# AUDIT EVENTS ENDPOINT (AuditEventsEndpoint)
management.endpoint.auditevents.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.auditevents.enabled=true # Whether to enable the auditevents
endpoint.

# BEANS ENDPOINT (BeansEndpoint)
management.endpoint.beans.cache.time-to-live=0ms # Maximum time that a response can
be cached.
management.endpoint.beans.enabled=true # Whether to enable the beans endpoint.

# CACHES ENDPOINT (CachesEndpoint)
management.endpoint.caches.cache.time-to-live=0ms # Maximum time that a response can
be cached.
management.endpoint.caches.enabled=true # Whether to enable the caches endpoint.

# CONDITIONS REPORT ENDPOINT (ConditionsReportEndpoint)
management.endpoint.conditions.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.conditions.enabled=true # Whether to enable the conditions
endpoint.

# CONFIGURATION PROPERTIES REPORT ENDPOINT (ConfigurationPropertiesReportEndpoint,
ConfigurationPropertiesReportEndpointProperties)
management.endpoint.configprops.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.configprops.enabled=true # Whether to enable the configprops
endpoint.
management.endpoint.configprops.keys-to-
sanitize=password,secret,key,token,.*credentials.*,vcap_services,sun.java.command #
Keys that should be sanitized. Keys can be simple strings that the property ends with
or regular expressions.

# ENVIRONMENT ENDPOINT (EnvironmentEndpoint, EnvironmentEndpointProperties)
management.endpoint.env.cache.time-to-live=0ms # Maximum time that a response can be
cached.
management.endpoint.env.enabled=true # Whether to enable the env endpoint.
```

```
management.endpoint.env.keys-to-
sanitize=password,secret,key,token,.*credentials.*,vcap_services,sun.java.command #
Keys that should be sanitized. Keys can be simple strings that the property ends with
or regular expressions.

# FLYWAY ENDPOINT (FlywayEndpoint)
management.endpoint.flyway.cache.time-to-live=0ms # Maximum time that a response can
be cached.
management.endpoint.flyway.enabled=true # Whether to enable the flyway endpoint.

# HEALTH ENDPOINT (HealthEndpoint, HealthEndpointProperties)
management.endpoint.health.cache.time-to-live=0ms # Maximum time that a response can
be cached.
management.endpoint.health.enabled=true # Whether to enable the health endpoint.
management.endpoint.health.roles= # Roles used to determine whether or not a user is
authorized to be shown details. When empty, all authenticated users are authorized.
management.endpoint.health.show-details=never # When to show full health details.

# HEAP DUMP ENDPOINT (HeapDumpWebEndpoint)
management.endpoint.heapdump.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.heapdump.enabled=true # Whether to enable the heapdump endpoint.

# HTTP TRACE ENDPOINT (HttpTraceEndpoint)
management.endpoint.httptrace.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.httptrace.enabled=true # Whether to enable the httptrace
endpoint.

# INFO ENDPOINT (InfoEndpoint)
info= # Arbitrary properties to add to the info endpoint.
management.endpoint.info.cache.time-to-live=0ms # Maximum time that a response can be
cached.
management.endpoint.info.enabled=true # Whether to enable the info endpoint.

# INTEGRATION GRAPH ENDPOINT (IntegrationGraphEndpoint)
management.endpoint.integrationgraph.cache.time-to-live=0ms # Maximum time that a
response can be cached.
management.endpoint.integrationgraph.enabled=true # Whether to enable the
integrationgraph endpoint.

# JOLOKIA ENDPOINT (JolokiaProperties)
management.endpoint.jolokia.config.*= # Jolokia settings. Refer to the documentation
of Jolokia for more details.
management.endpoint.jolokia.enabled=true # Whether to enable the jolokia endpoint.

# LIQUIBASE ENDPOINT (LiquibaseEndpoint)
management.endpoint.liquibase.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.liquibase.enabled=true # Whether to enable the liquibase
endpoint.

# LOG FILE ENDPOINT (LogFileWebEndpoint, LogFileWebEndpointProperties)
management.endpoint.logfile.cache.time-to-live=0ms # Maximum time that a response can
be cached.
management.endpoint.logfile.enabled=true # Whether to enable the logfile endpoint.
management.endpoint.logfile.external-file= # External Logfile to be accessed. Can be
used if the logfile is written by output redirect and not by the logging system
itself.

# LOGGERS ENDPOINT (LoggersEndpoint)
management.endpoint.loggers.cache.time-to-live=0ms # Maximum time that a response can
be cached.
management.endpoint.loggers.enabled=true # Whether to enable the loggers endpoint.

# REQUEST MAPPING ENDPOINT  (MappingsEndpoint)
management.endpoint.mappings.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.mappings.enabled=true # Whether to enable the mappings endpoint.
```

```
# METRICS ENDPOINT (MetricsEndpoint)
management.endpoint.metrics.cache.time-to-live=0ms # Maximum time that a response can
be cached.
management.endpoint.metrics.enabled=true # Whether to enable the metrics endpoint.

# PROMETHEUS ENDPOINT (PrometheusScrapeEndpoint)
management.endpoint.prometheus.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.prometheus.enabled=true # Whether to enable the prometheus
endpoint.

# SCHEDULED TASKS ENDPOINT (ScheduledTasksEndpoint)
management.endpoint.scheduledtasks.cache.time-to-live=0ms # Maximum time that a
response can be cached.
management.endpoint.scheduledtasks.enabled=true # Whether to enable the
scheduledtasks endpoint.

# SESSIONS ENDPOINT (SessionsEndpoint)
management.endpoint.sessions.enabled=true # Whether to enable the sessions endpoint.

# SHUTDOWN ENDPOINT (ShutdownEndpoint)
management.endpoint.shutdown.enabled=false # Whether to enable the shutdown endpoint.

# THREAD DUMP ENDPOINT (ThreadDumpEndpoint)
management.endpoint.threaddump.cache.time-to-live=0ms # Maximum time that a response
can be cached.
management.endpoint.threaddump.enabled=true # Whether to enable the threaddump
endpoint.

# HEALTH INDICATORS
management.health.db.enabled=true # Whether to enable database health check.
management.health.cassandra.enabled=true # Whether to enable Cassandra health check.
management.health.couchbase.enabled=true # Whether to enable Couchbase health check.
management.health.defaults.enabled=true # Whether to enable default health
indicators.
management.health.diskspace.enabled=true # Whether to enable disk space health check.
management.health.diskspace.path= # Path used to compute the available disk space.
management.health.diskspace.threshold=10MB # Minimum disk space that should be
available.
management.health.elasticsearch.enabled=true # Whether to enable Elasticsearch health
check.
management.health.elasticsearch.indices= # Comma-separated index names.
management.health.elasticsearch.response-timeout=100ms # Time to wait for a response
from the cluster.
management.health.influxdb.enabled=true # Whether to enable InfluxDB health check.
management.health.jms.enabled=true # Whether to enable JMS health check.
management.health.ldap.enabled=true # Whether to enable LDAP health check.
management.health.mail.enabled=true # Whether to enable Mail health check.
management.health.mongo.enabled=true # Whether to enable MongoDB health check.
management.health.neo4j.enabled=true # Whether to enable Neo4j health check.
management.health.rabbit.enabled=true # Whether to enable RabbitMQ health check.
management.health.redis.enabled=true # Whether to enable Redis health check.
management.health.solr.enabled=true # Whether to enable Solr health check.
management.health.status.http-mapping= # Mapping of health statuses to HTTP status
codes. By default, registered health statuses map to sensible defaults (for example,
UP maps to 200).
management.health.status.order=DOWN,OUT_OF_SERVICE,UP,UNKNOWN # Comma-separated list
of health statuses in order of severity.

# HTTP TRACING (HttpTraceProperties)
management.trace.http.enabled=true # Whether to enable HTTP request-response tracing.
management.trace.http.include=request-headers,response-headers,cookies,errors # Items
to be included in the trace.

# INFO CONTRIBUTORS (InfoContributorProperties)
management.info.build.enabled=true # Whether to enable build info.
management.info.defaults.enabled=true # Whether to enable default info contributors.
management.info.env.enabled=true # Whether to enable environment info.
```

```
management.info.git.enabled=true # Whether to enable git info.
management.info.git.mode=simple # Mode to use to expose git information.

# METRICS
management.metrics.distribution.maximum-expected-value.*= # Maximum value that meter
IDs starting-with the specified name are expected to observe.
management.metrics.distribution.minimum-expected-value.*= # Minimum value that meter
IDs starting-with the specified name are expected to observe.
management.metrics.distribution.percentiles.*= # Specific computed non-aggregable
percentiles to ship to the backend for meter IDs starting-with the specified name.
management.metrics.distribution.percentiles-histogram.*= # Whether meter IDs starting
with the specified name should publish percentile histograms.
management.metrics.distribution.sla.*= # Specific SLA boundaries for meter IDs
starting-with the specified name. The longest match wins.
management.metrics.enable.*= # Whether meter IDs starting-with the specified name
should be enabled. The longest match wins, the key `all` can also be used to
configure all meters.
management.metrics.export.appoptics.api-token= # AppOptics API token.
management.metrics.export.appoptics.batch-size=500 # Number of measurements per
request to use for this backend. If more measurements are found, then multiple
requests will be made.
management.metrics.export.appoptics.connect-timeout=5s # Connection timeout for
requests to this backend.
management.metrics.export.appoptics.enabled=true # Whether exporting of metrics to
this backend is enabled.
management.metrics.export.appoptics.host-tag=instance # Tag that will be mapped to
"@host" when shipping metrics to AppOptics.
management.metrics.export.appoptics.num-threads=2 # Number of threads to use with the
metrics publishing scheduler.
management.metrics.export.appoptics.read-timeout=10s # Read timeout for requests to
this backend.
management.metrics.export.appoptics.step=1m # Step size (i.e. reporting frequency) to
use.
management.metrics.export.appoptics.uri=https://api.appoptics.com/v1/measurements #
URI to ship metrics to.
management.metrics.export.atlas.batch-size=10000 # Number of measurements per request
to use for this backend. If more measurements are found, then multiple requests will
be made.
management.metrics.export.atlas.config-refresh-frequency=10s # Frequency for
refreshing config settings from the LWC service.
management.metrics.export.atlas.config-time-to-live=150s # Time to live for
subscriptions from the LWC service.
management.metrics.export.atlas.config-
uri=http://localhost:7101/lwc/api/v1/expressions/local-dev # URI for the Atlas LWC
endpoint to retrieve current subscriptions.
management.metrics.export.atlas.connect-timeout=1s # Connection timeout for requests
to this backend.
management.metrics.export.atlas.enabled=true # Whether exporting of metrics to this
backend is enabled.
management.metrics.export.atlas.eval-uri=http://localhost:7101/lwc/api/v1/evaluate #
URI for the Atlas LWC endpoint to evaluate the data for a subscription.
management.metrics.export.atlas.lwc-enabled=false # Whether to enable streaming to
Atlas LWC.
management.metrics.export.atlas.meter-time-to-live=15m # Time to live for meters that
do not have any activity. After this period the meter will be considered expired and
will not get reported.
management.metrics.export.atlas.num-threads=2 # Number of threads to use with the
metrics publishing scheduler.
management.metrics.export.atlas.read-timeout=10s # Read timeout for requests to this
backend.
management.metrics.export.atlas.step=1m # Step size (i.e. reporting frequency) to
use.
management.metrics.export.atlas.uri=http://localhost:7101/api/v1/publish # URI of the
Atlas server.
management.metrics.export.datadog.api-key= # Datadog API key.
management.metrics.export.datadog.application-key= # Datadog application key. Not
strictly required, but improves the Datadog experience by sending meter descriptions,
types, and base units to Datadog.
management.metrics.export.datadog.batch-size=10000 # Number of measurements per
```

request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.datadog.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.datadog.descriptions=true # Whether to publish descriptions metadata to Datadog. Turn this off to minimize the amount of metadata sent.
management.metrics.export.datadog.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.datadog.host-tag=instance # Tag that will be mapped to "host" when shipping metrics to Datadog.
management.metrics.export.datadog.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.datadog.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.datadog.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.datadog.uri=https://app.datadoghq.com # URI to ship metrics to. If you need to publish metrics to an internal proxy en-route to Datadog, you can define the location of the proxy with this.
management.metrics.export.dynatrace.api-token= # Dynatrace authentication token.
management.metrics.export.dynatrace.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.dynatrace.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.dynatrace.device-id= # ID of the custom device that is exporting metrics to Dynatrace.
management.metrics.export.dynatrace.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.dynatrace.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.dynatrace.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.dynatrace.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.dynatrace.technology-type=java # Technology type for exported metrics. Used to group metrics under a logical technology name in the Dynatrace UI.
management.metrics.export.dynatrace.uri= # URI to ship metrics to. Should be used for SaaS, self managed instances or to en-route through an internal proxy.
management.metrics.export.elastic.auto-create-index=true # Whether to create the index automatically if it does not exist.
management.metrics.export.elastic.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.elastic.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.elastic.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.elastic.host=http://localhost:9200 # Host to export metrics to.
management.metrics.export.elastic.index=metrics # Index to export metrics to.
management.metrics.export.elastic.index-date-format=yyyy-MM # Index date format used for rolling indices. Appended to the index name, preceded by a '-'.
management.metrics.export.elastic.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.elastic.password= # Login password of the Elastic server.
management.metrics.export.elastic.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.elastic.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.elastic.timestamp-field-name=@timestamp # Name of the timestamp field.
management.metrics.export.elastic.user-name= # Login user of the Elastic server.
management.metrics.export.ganglia.addressing-mode=multicast # UDP addressing mode, either unicast or multicast.
management.metrics.export.ganglia.duration-units=milliseconds # Base time unit used to report durations.
management.metrics.export.ganglia.enabled=true # Whether exporting of metrics to

Ganglia is enabled.
management.metrics.export.ganglia.host=localhost # Host of the Ganglia server to receive exported metrics.
management.metrics.export.ganglia.port=8649 # Port of the Ganglia server to receive exported metrics.
management.metrics.export.ganglia.protocol-version=3.1 # Ganglia protocol version. Must be either 3.1 or 3.0.
management.metrics.export.ganglia.rate-units=seconds # Base time unit used to report rates.
management.metrics.export.ganglia.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.ganglia.time-to-live=1 # Time to live for metrics on Ganglia. Set the multi-cast Time-To-Live to be one greater than the number of hops (routers) between the hosts.
management.metrics.export.graphite.duration-units=milliseconds # Base time unit used to report durations.
management.metrics.export.graphite.enabled=true # Whether exporting of metrics to Graphite is enabled.
management.metrics.export.graphite.host=localhost # Host of the Graphite server to receive exported metrics.
management.metrics.export.graphite.port=2004 # Port of the Graphite server to receive exported metrics.
management.metrics.export.graphite.protocol=pickled # Protocol to use while shipping data to Graphite.
management.metrics.export.graphite.rate-units=seconds # Base time unit used to report rates.
management.metrics.export.graphite.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.graphite.tags-as-prefix= # For the default naming convention, turn the specified tag keys into part of the metric prefix.
management.metrics.export.humio.api-token= # Humio API token.
management.metrics.export.humio.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.humio.connect-timeout=5s # Connection timeout for requests to this backend.
management.metrics.export.humio.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.humio.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.humio.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.humio.repository=sandbox # Name of the repository to publish metrics to.
management.metrics.export.humio.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.humio.tags.*= # Humio tags describing the data source in which metrics will be stored. Humio tags are a distinct concept from Micrometer's tags. Micrometer's tags are used to divide metrics along dimensional boundaries.
management.metrics.export.humio.uri=https://cloud.humio.com # URI to ship metrics to. If you need to publish metrics to an internal proxy en-route to Humio, you can define the location of the proxy with this.
management.metrics.export.influx.auto-create-db=true # Whether to create the Influx database if it does not exist before attempting to publish metrics to it.
management.metrics.export.influx.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.influx.compressed=true # Whether to enable GZIP compression of metrics batches published to Influx.
management.metrics.export.influx.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.influx.consistency=one # Write consistency for each point.
management.metrics.export.influx.db=mydb # Tag that will be mapped to "host" when shipping metrics to Influx.
management.metrics.export.influx.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.influx.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.influx.password= # Login password of the Influx server.

management.metrics.export.influx.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.influx.retention-duration= # Time period for which Influx should retain data in the current database.
management.metrics.export.influx.retention-shard-duration= # Time range covered by a shard group.
management.metrics.export.influx.retention-policy= # Retention policy to use (Influx writes to the DEFAULT retention policy if one is not specified).
management.metrics.export.influx.retention-replication-factor= # How many copies of the data are stored in the cluster.
management.metrics.export.influx.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.influx.uri=http://localhost:8086 # URI of the Influx server.
management.metrics.export.influx.user-name= # Login user of the Influx server.
management.metrics.export.jmx.domain=metrics # Metrics JMX domain name.
management.metrics.export.jmx.enabled=true # Whether exporting of metrics to JMX is enabled.
management.metrics.export.jmx.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.kairos.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.kairos.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.kairos.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.kairos.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.kairos.password= # Login password of the KairosDB server.
management.metrics.export.kairos.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.kairos.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.kairos.uri= localhost:8080/api/v1/datapoints # URI of the KairosDB server.
management.metrics.export.kairos.user-name= # Login user of the KairosDB server.
management.metrics.export.newrelic.account-id= # New Relic account ID.
management.metrics.export.newrelic.api-key= # New Relic API key.
management.metrics.export.newrelic.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.newrelic.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.newrelic.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.newrelic.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.newrelic.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.newrelic.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.newrelic.uri=https://insights-collector.newrelic.com # URI to ship metrics to.
management.metrics.export.prometheus.descriptions=true # Whether to enable publishing descriptions as part of the scrape payload to Prometheus. Turn this off to minimize the amount of data sent on each scrape.
management.metrics.export.prometheus.enabled=true # Whether exporting of metrics to Prometheus is enabled.
management.metrics.export.prometheus.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.prometheus.pushgateway.base-url=localhost:9091 # Base URL for the Pushgateway.
management.metrics.export.prometheus.pushgateway.enabled=false # Enable publishing via a Prometheus Pushgateway.
management.metrics.export.prometheus.pushgateway.grouping-key= # Grouping key for the pushed metrics.
management.metrics.export.prometheus.pushgateway.job= # Job identifier for this application instance.
management.metrics.export.prometheus.pushgateway.push-rate=1m # Frequency with which

to push metrics.
management.metrics.export.prometheus.pushgateway.shutdown-operation= # Operation that should be performed on shutdown.
management.metrics.export.signalfx.access-token= # SignalFX access token.
management.metrics.export.signalfx.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.signalfx.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.signalfx.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.signalfx.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.signalfx.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.signalfx.source= # Uniquely identifies the app instance that is publishing metrics to SignalFx. Defaults to the local host name.
management.metrics.export.signalfx.step=10s # Step size (i.e. reporting frequency) to use.
management.metrics.export.signalfx.uri=https://ingest.signalfx.com # URI to ship metrics to.
management.metrics.export.simple.enabled=true # Whether, in the absence of any other exporter, exporting of metrics to an in-memory backend is enabled.
management.metrics.export.simple.mode=cumulative # Counting mode.
management.metrics.export.simple.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.statsd.enabled=true # Whether exporting of metrics to StatsD is enabled.
management.metrics.export.statsd.flavor=datadog # StatsD line protocol to use.
management.metrics.export.statsd.host=localhost # Host of the StatsD server to receive exported metrics.
management.metrics.export.statsd.max-packet-length=1400 # Total length of a single payload should be kept within your network's MTU.
management.metrics.export.statsd.polling-frequency=10s # How often gauges will be polled. When a gauge is polled, its value is recalculated and if the value has changed (or publishUnchangedMeters is true), it is sent to the StatsD server.
management.metrics.export.statsd.port=8125 # Port of the StatsD server to receive exported metrics.
management.metrics.export.statsd.publish-unchanged-meters=true # Whether to send unchanged meters to the StatsD server.
management.metrics.export.wavefront.api-token= # API token used when publishing metrics directly to the Wavefront API host.
management.metrics.export.wavefront.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.wavefront.connect-timeout=1s # Connection timeout for requests to this backend.
management.metrics.export.wavefront.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.wavefront.global-prefix= # Global prefix to separate metrics originating from this app's white box instrumentation from those originating from other Wavefront integrations when viewed in the Wavefront UI.
management.metrics.export.wavefront.num-threads=2 # Number of threads to use with the metrics publishing scheduler.
management.metrics.export.wavefront.read-timeout=10s # Read timeout for requests to this backend.
management.metrics.export.wavefront.source= # Unique identifier for the app instance that is the source of metrics being published to Wavefront. Defaults to the local host name.
management.metrics.export.wavefront.step=10s # Step size (i.e. reporting frequency) to use.
management.metrics.export.wavefront.uri=https://longboard.wavefront.com # URI to ship metrics to.
management.metrics.use-global-registry=true # Whether auto-configured MeterRegistry implementations should be bound to the global static registry on Metrics.
management.metrics.tags.*= # Common tags that are applied to every meter.
management.metrics.web.client.max-uri-tags=100 # Maximum number of unique URI tag values allowed. After the max number of tag values is reached, metrics with additional tag values are denied by filter.

```
management.metrics.web.client.requests-metric-name=http.client.requests # Name of the
metric for sent requests.
management.metrics.web.server.auto-time-requests=true # Whether requests handled by
Spring MVC, WebFlux or Jersey should be automatically timed.
management.metrics.web.server.max-uri-tags=100 # Maximum number of unique URI tag
values allowed. After the max number of tag values is reached, metrics with
additional tag values are denied by filter.
management.metrics.web.server.requests-metric-name=http.server.requests # Name of the
metric for received requests.


# ----------------------------------------
# DEVTOOLS PROPERTIES
# ----------------------------------------

# DEVTOOLS (DevToolsProperties)
spring.devtools.add-properties=true # Whether to enable development property
defaults.
spring.devtools.livereload.enabled=true # Whether to enable a livereload.com-
compatible server.
spring.devtools.livereload.port=35729 # Server port.
spring.devtools.restart.additional-exclude= # Additional patterns that should be
excluded from triggering a full restart.
spring.devtools.restart.additional-paths= # Additional paths to watch for changes.
spring.devtools.restart.enabled=true # Whether to enable automatic restart.
spring.devtools.restart.exclude=META-INF/maven/**,META-
INF/resources/**,resources/**,static/**,public/**,templates/**,**/*Test.class,**/*Tes
ts.class,git.properties,META-INF/build-info.properties # Patterns that should be
excluded from triggering a full restart.
spring.devtools.restart.log-condition-evaluation-delta=true # Whether to log the
condition evaluation delta upon restart.
spring.devtools.restart.poll-interval=1s # Amount of time to wait between polling for
classpath changes.
spring.devtools.restart.quiet-period=400ms # Amount of quiet time required without
any classpath changes before a restart is triggered.
spring.devtools.restart.trigger-file= # Name of a specific file that, when changed,
triggers the restart check. If not specified, any classpath file change triggers the
restart.

# REMOTE DEVTOOLS (RemoteDevToolsProperties)
spring.devtools.remote.context-path=/.~~spring-boot!~ # Context path used to handle
the remote connection.
spring.devtools.remote.proxy.host= # The host of the proxy to use to connect to the
remote application.
spring.devtools.remote.proxy.port= # The port of the proxy to use to connect to the
remote application.
spring.devtools.remote.restart.enabled=true # Whether to enable remote restart.
spring.devtools.remote.secret= # A shared secret required to establish a connection
(required to enable remote support).
spring.devtools.remote.secret-header-name=X-AUTH-TOKEN # HTTP header used to transfer
the shared secret.


# ----------------------------------------
# TESTING PROPERTIES
# ----------------------------------------

spring.test.database.replace=any # Type of existing DataSource to replace.
spring.test.mockmvc.print=default # MVC Print option.
```

附录 B.配置元数据

Spring Boot jar 包括元数据文件，提供所有支持的配置属性的详细信息。这些文件旨在让 IDE 开发人员在用户使用 `application.properties` 或 `application.yml` 文件时提供上下文帮助和“代码完成”。

通过处理使用@ConfigurationProperties 注释的所有项目，在编译时自动生成大部分元数据文件。但是，可以 针对极端情况或更高级的用例手动编写部分元数据。

B.1 元数据格式

配置元数据文件位于 META-INF/spring-configuration-metadata.json 下的 jar 中。它们使用简单的 JSON 格式，其中的项目分类为“groups”或“properties”，其他值提示分类在“hints”下，如以下示例所示：

```
{"groups": [
        {
                "name": "server",
                "type":
"org.springframework.boot.autoconfigure.web.ServerProperties",
                "sourceType":
"org.springframework.boot.autoconfigure.web.ServerProperties"
        },
        {
                "name": "spring.jpa.hibernate",
                "type":
"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate",
                "sourceType":
"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
                "sourceMethod": "getHibernate()"
        }
        ...
],"properties": [
        {
                "name": "server.port",
                "type": "java.lang.Integer",
                "sourceType":
"org.springframework.boot.autoconfigure.web.ServerProperties"
        },
        {
                "name": "server.address",
                "type": "java.net.InetAddress",
                "sourceType":
"org.springframework.boot.autoconfigure.web.ServerProperties"
        },
        {
                  "name": "spring.jpa.hibernate.ddl-auto",
                  "type": "java.lang.String",
                  "description": "DDL mode. This is actually a shortcut for
the \"hibernate.hbm2ddl.auto\" property.",
                  "sourceType":
"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"
        }
        ...
],"hints": [
        {
                "name": "spring.jpa.hibernate.ddl-auto",
                "values": [
                        {
                                "value": "none",
                                "description": "Disable DDL handling."
                        },
                        {
                                "value": "validate",
                                "description": "Validate the schema, make no changes
to the database."
                        },
                        {
                                "value": "update",
                                "description": "Update the schema if necessary."
                        },
                        {
```

```
                                "value": "create",
                                "description": "Create the schema and destroy
previous data."
                        },
                        {
                                "value": "create-drop",
                                "description": "Create and then destroy the schema at
the end of the session."
                        }
                ]
        }
]}
```

每个“属性”是用户使用给定值指定的配置项。例如，`server.port` 和 `server.address` 可能在 `application.properties` 中指定，如下所示：

```
server.port=9090
server.address=127.0.0.1
```

“组”是更高级别的项目，它们本身不指定值，而是为属性提供上下文分组。例如，`server.port` 和 `server.address` 属性是 `server` 组的一部分。

注意

并不要求每个“财产”都有“团体”。某些属性可能本身就存在。

最后，“提示”是用于帮助用户配置给定属性的附加信息。例如，当开发人员配置 `spring.jpa.hibernate.ddl-auto` 属性时，工具可以使用提示为 `none`，`validate`，`update`，`create` 提供一些自动完成帮助，和 `create-drop` 值。

B.1.1 组属性

`groups` 数组中包含的 JSON 对象可以包含下表中显示的属性：

名称	类型	目的
`name`	String	The full name of the group. This attribute is mandatory.
`type`	String	The class name of the data type of the group. For example, if the group were based on a class annotated with `@ConfigurationProperties`, the attribute would contain the fully qualified name of that class. If it were based on a `@Bean` method, it would be the return type of that method. If the type is not known, the attribute may be omitted.
`description`	String	A short description of the group that can be displayed to users. If not description is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this group. For example, if the group were based on a `@Bean` method annotated with `@ConfigurationProperties`, this attribute would contain the fully qualified name of the `@Configuration` class that contains the method. If the source type is not known, the attribute may be omitted.
`sourceMethod`	String	The full name of the method (include parenthesis and argument types) that

名称	类型	目的
		contributed this group (for example, the name of a `@ConfigurationProperties` annotated `@Bean` method). If the source method is not known, it may be omitted.

B.1.2 Property 属性

`properties` 数组中包含的 JSON 对象可以包含下表中描述的属性：

名称	类型	目的
`name`	String	The full name of the property. Names are in lower-case period-separated form (for example, `server.address`). This attribute is mandatory.
`type`	String	The full signature of the data type of the property (for example, `java.lang.String`) but also a full generic type (such as `java.util.Map<java.util.String,acme.MyEnum>`). You can use this attribute to guide the user as to the types of values that they can enter. For consistency, the type of a primitive is specified by using its wrapper counterpart (for example, `boolean` becomes `java.lang.Boolean`). Note that this class may be a complex type that gets converted from a `String` as values are bound. If the type is not known, it may be omitted.
`description`	String	A short description of the group that can be displayed to users. If no description is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this property. For example, if the property were from a class annotated with `@ConfigurationProperties`, this attribute would contain the fully qualified name of that class. If the source type is unknown, it may be omitted.
`defaultValue`	Object	The default value, which is used if the property is not specified. If the type of the property is an array, it can be an array of value(s). If the default value is unknown, it may be omitted.
`deprecation`	Deprecation	Specify whether the property is deprecated. If the field is not deprecated or if that information is not known, it may be omitted. The next table offers more detail about the `deprecation` attribute.

每个 `properties` 元素的 `deprecation` 属性中包含的 JSON 对象可以包含以下属性：

名称	类型	目的
`level`	String	The level of deprecation, which can be either `warning` (the default) or `error`. When a property has a `warning` deprecation level, it should still be bound in the environment. However, when it has an `error` deprecation level, the property is no longer managed and is not bound.

名称	类型	目的
reason	String	A short description of the reason why the property was deprecated. If no reason is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.).
replacement	String	The full name of the property that *replaces* this deprecated property. If there is no replacement for this property, it may be omitted.

注意

在 Spring Boot 1.3 之前，可以使用单个 deprecated 布尔属性来代替 deprecation 元素。仍以不推荐的方式支持此功能，不应再使用它。如果没有可用的原因和替换，则应设置空的 deprecation 对象。

通过将@DeprecatedConfigurationProperty 注释添加到公开不推荐使用的属性的 getter 中，也可以在代码中以声明方式指定弃用。例如，假设 app.acme.target 属性令人困惑，并重命名为 app.acme.name。以下示例显示了如何处理这种情况：

```
@ConfigurationProperties("app.acme")
public class AcmeProperties {

        private String name;

        public String getName() { ... }

        public void setName(String name) { ... }

        @DeprecatedConfigurationProperty(replacement = "app.acme.name")
        @Deprecated
        public String getTarget() {
                return getName();
        }

        @Deprecated
        public void setTarget(String target) {
                setName(target);
        }
}
```

注意

没有办法设置 level。始终假设 warning，因为代码仍在处理属性。

前面的代码确保 deprecated 属性仍然有效（委托给幕后的 name 属性）。一旦可以从公共 API 中删除 getTarget 和 setTarget 方法，元数据中的自动弃用提示也会消失。如果要保留提示，添加具有 error 弃用级别的手动元数据可确保用户仍然了解该属性。当提供 replacement 时，这样做特别有用。

B.1.3 提示属性

hints 数组中包含的 JSON 对象可以包含下表中显示的属性：

名称	类型	目的
name	String	The full name of the property to which this hint refers. Names are in

名称	类型	目的
		lower-case period-separated form (such as `spring.mvc.servlet.path`). If the property refers to a map (such as `system.contexts`), the hint either applies to the *keys* of the map (`system.context.keys`) or the *values* (`system.context.values`) of the map. This attribute is mandatory.
`values`	ValueHint[]	A list of valid values as defined by the `ValueHint` object (described in the next table). Each entry defines the value and may have a description.
`providers`	ValueProvider[]	A list of providers as defined by the `ValueProvider` object (described later in this document). Each entry defines the name of the provider and its parameters, if any.

每个 `hint` 元素的 `values` 属性中包含的 JSON 对象可以包含下表中描述的属性：

名称	类型	目的
`value`	Object	A valid value for the element to which the hint refers. If the type of the property is an array, it can also be an array of value(s). This attribute is mandatory.
`description`	String	A short description of the value that can be displayed to users. If no description is available, it may be omitted . It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).

每个 `hint` 元素的 `providers` 属性中包含的 JSON 对象可以包含下表中描述的属性：

名称	类型	目的
`name`	String	The name of the provider to use to offer additional content assistance for the element to which the hint refers.
`parameters`	JSON object	Any additional parameter that the provider supports (check the documentation of the provider for more details).

B.1.4 重复的元数据项

具有相同“属性”和“组”名称的对象可以在元数据文件中多次出现。例如，您可以将两个单独的类绑定到同一个前缀，每个类都有可能重叠的属性名称。虽然多次出现在元数据中的相同名称不应该是常见的，但元数据的使用者应该注意确保他们支持它。

B.2 提供手动提示

要改善用户体验并进一步帮助用户配置给定属性，您可以提供以下其他元数据：

- 描述属性的潜在值列表。
- 关联提供者，将明确定义的语义附加到属性，以便工具可以根据项目的上下文发现潜在值列表。

B.2.1 价值提示

每个提示的 `name` 属性指的是属性的 `name`。在前面显示的 初始示例中，我们为 `spring.jpa.hibernate.ddl-auto` 属性提供了五个值：`none`，`validate`，`update`，`create` 和 `create-drop`。每个值也可以有描述。

如果您的属性类型为 `Map`，则可以提供键和值的提示（但不能提供地图本身的提示）。特殊的 `.keys` 和 `.values` 后缀必须分别引用键和值。

假设 `sample.contexts` 将 magic `String` 值映射为整数，如以下示例所示：

```
@ConfigurationProperties("sample")
public class SampleProperties {

        private Map<String,Integer> contexts;
        // getters and setters
}
```

神奇的值（在这个例子中）是 `sample1` 和 `sample2`。为了为密钥提供额外的内容帮助，您可以将以下 JSON 添加到 模块的手动元数据中：

```
{"hints": [
        {
                "name": "sample.contexts.keys",
                "values": [
                        {
                                "value": "sample1"
                        },
                        {
                                "value": "sample2"
                        }
                ]
        }
]}
```

小费

我们建议您使用 `Enum` 代替这两个值。如果您的 IDE 支持它，这是迄今为止最有效的自动完成方法。

B.2.2 价值提供者

提供程序是将语义附加到属性的强大方法。在本节中，我们定义了可用于您自己的提示的官方提供程序。但是，您最喜欢的 IDE 可能会实现其中一些或不实现。此外，它最终可以提供自己的。

注意

由于这是一项新功能，因此 IDE 供应商必须了解其工作原理。采用时间自然会有所不同。

下表总结了支持的提供程序列表：

名称	描述
`any`	允许提供任何附加值。
`class-reference`	自动完成项目中可用的类。通常由 `target` 参数指定的基类约

名称	描述
	束。
`handle-as`	处理属性，就好像它是由强制 `target` 参数定义的类型定义的一样。
`logger-name`	自动完成有效的记录器名称和 记录器组。通常，当前项目中可用的包名和类名可以自动完成，也可以自定义组。
`spring-bean-reference`	自动完成当前项目中可用的 bean 名称。通常由 `target` 参数指定的基类约束。
`spring-profile-name`	自动完成项目中可用的 Spring 配置文件名称。

小费

对于给定的属性，只有一个提供程序可以处于活动状态，但如果它们都可以以某种方式管理属性，则可以指定多个提供程序。确保首先放置最强大的提供程序，因为 IDE 必须使用它可以处理的 JSON 部分中的第一个。如果不支持给定属性的提供者，则也不提供特殊内容帮助。

任何

特殊的**任何**提供者值都允许提供任何其他值。如果支持，则应应用基于属性类型的常规值验证。

如果您有值列表并且任何额外值仍应被视为有效，则通常使用此提供程序。

以下示例提供 `on` 和 `off` 作为 `system.state` 的自动完成值：

```
{"hints": [
        {
                "name": "system.state",
                "values": [
                        {
                                "value": "on"
                        },
                        {
                                "value": "off"
                        }
                ],
                "providers": [
                        {
                                "name": "any"
                        }
                ]
        }
]}
```

请注意，在前面的示例中，还允许任何其他值。

课程参考

在**类引用**提供商自动完成项目中可用的类。此提供程序支持以下参数：

参数	类型	默认值	描述
`target`	`String`	*none*	应分配给所选值的类的完全限定名称。通常用于过滤非候选

参数	类型	默认值	描述
	(Class)		类。请注意，通过公开具有适当上限的类，可以通过类型本身提供此信息。
concrete	boolean	true	指定是否仅将具体类视为有效候选。

以下元数据片段对应于标准要使用的 JspServlet 类名称的标准 server.servlet.jsp.class-name 属性：

```
{"hints": [
        {
                "name": "server.servlet.jsp.class-name",
                "providers": [
                        {
                                "name": "class-reference",
                                "parameters": {
                                        "target": "javax.servlet.http.HttpServlet"
                                }
                        }
                ]
        }
]}
```

处理为

该**手柄的**供应商，您可以替代属性的类型到一个更高层次的类型。当属性具有 java.lang.String 类型时，通常会发生这种情况，因为您不希望配置类依赖于可能不在类路径上的类。此提供程序支持以下参数：

参数	类型	默认值	描述
target	String (Class)	*none*	要为该属性考虑的类型的完全限定名称。此参数是必需的。

可以使用以下类型：

- 任何 java.lang.Enum：列出属性的可能值。（我们建议使用 Enum 类型定义属性，因为 IDE 不需要进一步提示来自动完成值。）
- java.nio.charset.Charset：支持自动完成字符集/编码值（例如 UTF-8）
- java.util.Locale：自动完成语言环境（例如 en_US）
- org.springframework.util.MimeType：支持自动完成内容类型值（例如 text/plain）
- org.springframework.core.io.Resource：支持自动完成 Spring 的资源抽象，以引用文件系统或类路径上的文件。（例如 classpath:/sample.properties）

小费

如果可以提供多个值，请使用 Collection 或 *Array* 类型向 IDE 讲授它。

以下元数据片段对应于标准 spring.liquibase.change-log 属性，该属性定义要使用的更改日志的路径。它实际上在内部用作 org.springframework.core.io.Resource 但不能这样暴露，因为我们需要保留原始的 String 值以将其传递给 Liquibase API。

```
{"hints": [
        {
                "name": "spring.liquibase.change-log",
```

```
                "providers": [
                        {
                                "name": "handle-as",
                                "parameters": {
                                        "target":
"org.springframework.core.io.Resource"
                                }
                        }
                ]
        }
]}
```

记录器名称

该**记录器名**提供商自动完成有效的记录名称和 记录器组。通常，可以自动完成当前项目中可用的包名和类名。如果启用了组（默认），并且在配置中标识了自定义记录器组，则应提供自动完成组。特定框架可能还有额外的魔术记录器名称，也可以支持。

此提供程序支持以下参数：

参数	类型	默认值	描述
`group`	`boolean`	`true`	指定是否应考虑已知组。

由于记录器名称可以是任意名称，因此此提供程序应允许任何值，但可以突出显示项目类路径中不可用的有效包名和类名。

以下元数据片段对应于标准 `logging.level` 属性。键是记录器名称，值对应于标准日志级别或任何自定义级别。由于 Spring Boot 定义了几个开箱即用的记录器组，因此为这些记录器添加了专用值提示。

```
{"hints": [
        {
                "name": "logging.level.keys",
                "values": [
                        {
                                "value": "root",
                                "description": "Root logger used to assign the
default logging level."
                        },
                        {
                                "value": "sql",
                                "description": "SQL logging group including Hibernate
SQL logger."
                        },
                        {
                                "value": "web",
                                "description": "Web logging group including codecs."
                        }
                ],
                "providers": [
                        {
                                "name": "logger-name"
                        }
                ]
        },
        {
                "name": "logging.level.values",
                "values": [
                        {
                                "value": "trace"
                        },
                        {
                                "value": "debug"
```

```
                        },
                        {
                                "value": "info"
                        },
                        {
                                "value": "warn"
                        },
                        {
                                "value": "error"
                        },
                        {
                                "value": "fatal"
                        },
                        {
                                "value": "off"
                        }

                ],
                "providers": [
                        {
                                "name": "any"
                        }
                ]
        }
]}
```

Spring Bean 参考

的 **spring-bean-reference** 提供商自动完成在当前项目中的配置中定义的 beans。此提供程序支持以下参数：

参数	类型	默认值	描述
`target`	`String` (`Class`)	*none*	应分配给候选人的 bean 类的完全限定名称。通常用于过滤掉非候选 beans。

以下元数据片段对应于标准 `spring.jmx.server` 属性，该属性定义要使用的 `MBeanServer` bean 的名称：

```
{"hints": [
        {
                "name": "spring.jmx.server",
                "providers": [
                        {
                                "name": "spring-bean-reference",
                                "parameters": {
                                        "target": "javax.management.MBeanServer"
                                }
                        }
                ]
        }
]}
```

注意

活页夹不知道元数据。如果您提供该提示，则仍需要将 bean 名称转换为 `ApplicationContext` 使用的实际 Bean 引用。

Spring 个人资料名称

的 **spring-profile-name** 提供商自动完成 Spring 在当前项目中的配置中定义的配置文件。

以下元数据片段对应于标准 `spring.profiles.active` 属性，该属性定义要启用的 Spring 配置文件的名称：

```
{"hints": [
        {
                "name": "spring.profiles.active",
                "providers": [
                        {
                                "name": "spring-profile-name"
                        }
                ]
        }
]}
```

B.3 使用注释处理器生成自己的元数据

您可以使用 `spring-boot-configuration-processor` jar 从 `@ConfigurationProperties` 注释的项目轻松生成自己的配置元数据文件。jar 包含一个 Java 注释处理器，在您编译项目时调用该处理器。要使用处理器，请在 `spring-boot-configuration-processor` 上包含依赖项。

使用 Maven 时，依赖项应声明为可选，如以下示例所示：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-configuration-processor</artifactId>
        <optional>true</optional>
</dependency>
```

对于 Gradle 4.5 及更早版本，应在 `compileOnly` 配置中声明依赖项，如以下示例所示：

```
dependencies {
        compileOnly "org.springframework.boot:spring-boot-configuration-processor"
}
```

对于 Gradle 4.6 及更高版本，应在 `annotationProcessor` 配置中声明依赖项，如以下示例所示：

```
dependencies {
        annotationProcessor "org.springframework.boot:spring-boot-configuration-
processor"
}
```

如果您使用的是 `additional-spring-configuration-metadata.json` 文件，则应将 `compileJava` 任务配置为依赖于 `processResources` 任务，如以下示例所示：

```
compileJava.dependsOn(processResources)
```

此依赖关系确保在编译期间注释处理器运行时可以使用其他元数据。

处理器选择使用`@ConfigurationProperties` 注释的类和方法。配置类中的字段值的 Javadoc 用于填充 `description` 属性。

注意

您应该只使用`@ConfigurationProperties` 字段 Javadoc 的简单文本，因为它们在添加到 JSON 之前不会被处理。

通过存在标准的 getter 和 setter 来发现属性，这些 getter 和 setter 具有对集合类型的特殊处理（即使只有 getter 存在也会检测到）。注释处理器还支持使用`@Data`，`@Getter` 和`@Setter` lombok 注释。

注意

如果在项目中使用 AspectJ，则需要确保注释处理器仅运行一次。有几种方法可以做到这一点。使用 Maven，您可以显式配置 `maven-apt-plugin` 并仅在那里将依赖项添加到注释处理器。您还可以让 AspectJ 插件在 `maven-compiler-plugin` 配置中运行所有处理并禁用注释处理，如下所示：

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
                <proc>none</proc>
        </configuration>
</plugin>
```

B.3.1 嵌套属性

注释处理器自动将内部类视为嵌套属性。考虑以下课程：

```
@ConfigurationProperties(prefix="server")
public class ServerProperties {

        private String name;

        private Host host;

        // ... getter and setters

        public static class Host {

                private String ip;

                private int port;

                // ... getter and setters

        }

}
```

上面的示例为 `server.name`，`server.host.ip` 和 `server.host.port` 属性生成元数据信息。您可以在字段上使用`@NestedConfigurationProperty` 注释来指示应将常规（非内部）类视为嵌套。

小费

这对集合和映射没有影响，因为这些类型是自动标识的，并且为每个类型生成单个元数据属性。

B.3.2 添加附加元数据

Spring Boot 的配置文件处理非常灵活，通常情况下可能存在未绑定到 `@ConfigurationProperties` bean 的属性。您可能还需要调整现有密钥的某些属性。为支持此类情况并允许您提供自定义“提示”，注释处理器会自动将 `META-INF/additional-`

`spring-configuration-metadata.json` 中的项目合并到主元数据文件中。

如果引用自动检测到的属性，则会覆盖描述，默认值和弃用信息（如果已指定）。如果未在当前模块中标识手动属性声明，则将其添加为新属性。

`additional-spring-configuration-metadata.json` 文件的格式与常规 `spring-configuration-metadata.json` 完全相同。附加属性文件是可选的。如果您没有任何其他属性，请不要添加该文件。

附录 C.自动配置类

以下是 Spring Boot 提供的所有自动配置类的列表，其中包含文档和源代码的链接。请记住还要查看应用程序中的条件报告，以获取有关打开哪些功能的更多详细信息。（为此，请使用`--debug` 或`-Ddebug` 启动应用程序，或者在 Actuator 应用程序中使用 `conditions` 端点）。

C.1 来自“spring-boot-autoconfigure”模块

以下自动配置类来自 `spring-boot-autoconfigure` 模块：

配置类	链接
`ActiveMQAutoConfiguration`	javadoc
`AopAutoConfiguration`	javadoc
`ArtemisAutoConfiguration`	javadoc
`BatchAutoConfiguration`	javadoc
`CacheAutoConfiguration`	javadoc
`CassandraAutoConfiguration`	javadoc
`CassandraDataAutoConfiguration`	javadoc
`CassandraReactiveDataAutoConfiguration`	javadoc
`CassandraReactiveRepositoriesAutoConfiguration`	javadoc
`CassandraRepositoriesAutoConfiguration`	javadoc
`ClientHttpConnectorAutoConfiguration`	javadoc
`CloudServiceConnectorsAutoConfiguration`	javadoc
`CodecsAutoConfiguration`	javadoc
`ConfigurationPropertiesAutoConfiguration`	javadoc
`CouchbaseAutoConfiguration`	javadoc
`CouchbaseDataAutoConfiguration`	javadoc

配置类	链接
CouchbaseReactiveDataAutoConfiguration	javadoc
CouchbaseReactiveRepositoriesAutoConfiguration	javadoc
CouchbaseRepositoriesAutoConfiguration	javadoc
DataSourceAutoConfiguration	javadoc
DataSourceTransactionManagerAutoConfiguration	javadoc
DispatcherServletAutoConfiguration	javadoc
ElasticsearchAutoConfiguration	javadoc
ElasticsearchDataAutoConfiguration	javadoc
ElasticsearchRepositoriesAutoConfiguration	javadoc
EmbeddedLdapAutoConfiguration	javadoc
EmbeddedMongoAutoConfiguration	javadoc
EmbeddedWebServerFactoryCustomizerAutoConfiguration	javadoc
ErrorMvcAutoConfiguration	javadoc
ErrorWebFluxAutoConfiguration	javadoc
FlywayAutoConfiguration	javadoc
FreeMarkerAutoConfiguration	javadoc
GroovyTemplateAutoConfiguration	javadoc
GsonAutoConfiguration	javadoc
H2ConsoleAutoConfiguration	javadoc
HazelcastAutoConfiguration	javadoc
HazelcastJpaDependencyAutoConfiguration	javadoc
HibernateJpaAutoConfiguration	javadoc
HttpEncodingAutoConfiguration	javadoc
HttpHandlerAutoConfiguration	javadoc
HttpMessageConvertersAutoConfiguration	javadoc

配置类	链接
HypermediaAutoConfiguration	javadoc
InfluxDbAutoConfiguration	javadoc
IntegrationAutoConfiguration	javadoc
JacksonAutoConfiguration	javadoc
JdbcRepositoriesAutoConfiguration	javadoc
JdbcTemplateAutoConfiguration	javadoc
JerseyAutoConfiguration	javadoc
JestAutoConfiguration	javadoc
JmsAutoConfiguration	javadoc
JmxAutoConfiguration	javadoc
JndiConnectionFactoryAutoConfiguration	javadoc
JndiDataSourceAutoConfiguration	javadoc
JooqAutoConfiguration	javadoc
JpaRepositoriesAutoConfiguration	javadoc
JsonbAutoConfiguration	javadoc
JtaAutoConfiguration	javadoc
KafkaAutoConfiguration	javadoc
LdapAutoConfiguration	javadoc
LdapRepositoriesAutoConfiguration	javadoc
LiquibaseAutoConfiguration	javadoc
MailSenderAutoConfiguration	javadoc
MailSenderValidatorAutoConfiguration	javadoc
MessageSourceAutoConfiguration	javadoc
MongoAutoConfiguration	javadoc
MongoDataAutoConfiguration	javadoc

配置类	链接
MongoReactiveAutoConfiguration	javadoc
MongoReactiveDataAutoConfiguration	javadoc
MongoReactiveRepositoriesAutoConfiguration	javadoc
MongoRepositoriesAutoConfiguration	javadoc
MultipartAutoConfiguration	javadoc
MustacheAutoConfiguration	javadoc
Neo4jDataAutoConfiguration	javadoc
Neo4jRepositoriesAutoConfiguration	javadoc
OAuth2ClientAutoConfiguration	javadoc
OAuth2ResourceServerAutoConfiguration	javadoc
PersistenceExceptionTranslationAutoConfiguration	javadoc
ProjectInfoAutoConfiguration	javadoc
PropertyPlaceholderAutoConfiguration	javadoc
QuartzAutoConfiguration	javadoc
RabbitAutoConfiguration	javadoc
ReactiveOAuth2ClientAutoConfiguration	javadoc
ReactiveOAuth2ResourceServerAutoConfiguration	javadoc
ReactiveSecurityAutoConfiguration	javadoc
ReactiveUserDetailsServiceAutoConfiguration	javadoc
ReactiveWebServerFactoryAutoConfiguration	javadoc
ReactorCoreAutoConfiguration	javadoc
RedisAutoConfiguration	javadoc
RedisReactiveAutoConfiguration	javadoc
RedisRepositoriesAutoConfiguration	javadoc
RepositoryRestMvcAutoConfiguration	javadoc

配置类	链接
RestClientAutoConfiguration	javadoc
RestTemplateAutoConfiguration	javadoc
SecurityAutoConfiguration	javadoc
SecurityFilterAutoConfiguration	javadoc
SecurityRequestMatcherProviderAutoConfiguration	javadoc
SendGridAutoConfiguration	javadoc
ServletWebServerFactoryAutoConfiguration	javadoc
SessionAutoConfiguration	javadoc
SolrAutoConfiguration	javadoc
SolrRepositoriesAutoConfiguration	javadoc
SpringApplicationAdminJmxAutoConfiguration	javadoc
SpringDataWebAutoConfiguration	javadoc
TaskExecutionAutoConfiguration	javadoc
TaskSchedulingAutoConfiguration	javadoc
ThymeleafAutoConfiguration	javadoc
TransactionAutoConfiguration	javadoc
UserDetailsServiceAutoConfiguration	javadoc
ValidationAutoConfiguration	javadoc
WebClientAutoConfiguration	javadoc
WebFluxAutoConfiguration	javadoc
WebMvcAutoConfiguration	javadoc
WebServiceTemplateAutoConfiguration	javadoc
WebServicesAutoConfiguration	javadoc
WebSocketMessagingAutoConfiguration	javadoc
WebSocketReactiveAutoConfiguration	javadoc

配置类	链接
WebSocketServletAutoConfiguration	javadoc
XADataSourceAutoConfiguration	javadoc

C.2 来自“spring-boot-actuator-autoconfigure”模块

以下自动配置类来自 `spring-boot-actuator-autoconfigure` 模块：

配置类	链接
AppOpticsMetricsExportAutoConfiguration	javadoc
AtlasMetricsExportAutoConfiguration	javadoc
AuditAutoConfiguration	javadoc
AuditEventsEndpointAutoConfiguration	javadoc
BeansEndpointAutoConfiguration	javadoc
CacheMetricsAutoConfiguration	javadoc
CachesEndpointAutoConfiguration	javadoc
CassandraHealthIndicatorAutoConfiguration	javadoc
CassandraReactiveHealthIndicatorAutoConfiguration	javadoc
CloudFoundryActuatorAutoConfiguration	javadoc
CompositeMeterRegistryAutoConfiguration	javadoc
ConditionsReportEndpointAutoConfiguration	javadoc
ConfigurationPropertiesReportEndpointAutoConfiguration	javadoc
CouchbaseHealthIndicatorAutoConfiguration	javadoc
CouchbaseReactiveHealthIndicatorAutoConfiguration	javadoc
DataSourceHealthIndicatorAutoConfiguration	javadoc
DataSourcePoolMetricsAutoConfiguration	javadoc
DatadogMetricsExportAutoConfiguration	javadoc
DiskSpaceHealthIndicatorAutoConfiguration	javadoc

配置类	链接
DynatraceMetricsExportAutoConfiguration	javadoc
ElasticMetricsExportAutoConfiguration	javadoc
ElasticSearchClientHealthIndicatorAutoConfiguration	javadoc
ElasticSearchJestHealthIndicatorAutoConfiguration	javadoc
ElasticSearchRestHealthIndicatorAutoConfiguration	javadoc
EndpointAutoConfiguration	javadoc
EnvironmentEndpointAutoConfiguration	javadoc
FlywayEndpointAutoConfiguration	javadoc
GangliaMetricsExportAutoConfiguration	javadoc
GraphiteMetricsExportAutoConfiguration	javadoc
HealthEndpointAutoConfiguration	javadoc
HealthIndicatorAutoConfiguration	javadoc
HeapDumpWebEndpointAutoConfiguration	javadoc
HibernateMetricsAutoConfiguration	javadoc
HttpClientMetricsAutoConfiguration	javadoc
HttpTraceAutoConfiguration	javadoc
HttpTraceEndpointAutoConfiguration	javadoc
HumioMetricsExportAutoConfiguration	javadoc
InfluxDbHealthIndicatorAutoConfiguration	javadoc
InfluxMetricsExportAutoConfiguration	javadoc
InfoContributorAutoConfiguration	javadoc
InfoEndpointAutoConfiguration	javadoc
IntegrationGraphEndpointAutoConfiguration	javadoc
JerseyServerMetricsAutoConfiguration	javadoc
JettyMetricsAutoConfiguration	javadoc
JmsHealthIndicatorAutoConfiguration	javadoc

配置类	链接
JmxEndpointAutoConfiguration	javadoc
JmxMetricsExportAutoConfiguration	javadoc
JolokiaEndpointAutoConfiguration	javadoc
JvmMetricsAutoConfiguration	javadoc
KafkaMetricsAutoConfiguration	javadoc
KairosMetricsExportAutoConfiguration	javadoc
LdapHealthIndicatorAutoConfiguration	javadoc
LiquibaseEndpointAutoConfiguration	javadoc
Log4J2MetricsAutoConfiguration	javadoc
LogFileWebEndpointAutoConfiguration	javadoc
LogbackMetricsAutoConfiguration	javadoc
LoggersEndpointAutoConfiguration	javadoc
MailHealthIndicatorAutoConfiguration	javadoc
ManagementContextAutoConfiguration	javadoc
ManagementWebSecurityAutoConfiguration	javadoc
MappingsEndpointAutoConfiguration	javadoc
MetricsAutoConfiguration	javadoc
MetricsEndpointAutoConfiguration	javadoc
MongoHealthIndicatorAutoConfiguration	javadoc
MongoReactiveHealthIndicatorAutoConfiguration	javadoc
Neo4jHealthIndicatorAutoConfiguration	javadoc
NewRelicMetricsExportAutoConfiguration	javadoc
PrometheusMetricsExportAutoConfiguration	javadoc
RabbitHealthIndicatorAutoConfiguration	javadoc
RabbitMetricsAutoConfiguration	javadoc

配置类	链接
ReactiveCloudFoundryActuatorAutoConfiguration	javadoc
ReactiveManagementContextAutoConfiguration	javadoc
ReactiveManagementWebSecurityAutoConfiguration	javadoc
RedisHealthIndicatorAutoConfiguration	javadoc
RedisReactiveHealthIndicatorAutoConfiguration	javadoc
ScheduledTasksEndpointAutoConfiguration	javadoc
ServletManagementContextAutoConfiguration	javadoc
SessionsEndpointAutoConfiguration	javadoc
ShutdownEndpointAutoConfiguration	javadoc
SignalFxMetricsExportAutoConfiguration	javadoc
SimpleMetricsExportAutoConfiguration	javadoc
SolrHealthIndicatorAutoConfiguration	javadoc
StatsdMetricsExportAutoConfiguration	javadoc
SystemMetricsAutoConfiguration	javadoc
ThreadDumpEndpointAutoConfiguration	javadoc
TomcatMetricsAutoConfiguration	javadoc
WavefrontMetricsExportAutoConfiguration	javadoc
WebEndpointAutoConfiguration	javadoc
WebFluxMetricsAutoConfiguration	javadoc
WebMvcMetricsAutoConfiguration	javadoc

附录 D.测试自动配置注释

下表列出了可用于测试应用程序片段的各种@…Test 注释以及默认情况下导入的自动配置：

测试切片	导入的自动配置
@DataJdbcTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.data.jdbc.JdbcRepos

测试切片	导入的自动配置
	org.springframework.boot.autoconfigure.flyway.FlywayAutoCo org.springframework.boot.autoconfigure.jdbc.DataSourceAuto org.springframework.boot.autoconfigure.jdbc.DataSourceTrans org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAu org.springframework.boot.autoconfigure.liquibase.LiquibaseA org.springframework.boot.autoconfigure.transaction.Transact org.springframework.boot.test.autoconfigure.jdbc.TestDataba
@DataJpaTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.data.jpa.JpaReposito org.springframework.boot.autoconfigure.flyway.FlywayAutoCo org.springframework.boot.autoconfigure.jdbc.DataSourceAuto org.springframework.boot.autoconfigure.jdbc.DataSourceTrans org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAu org.springframework.boot.autoconfigure.liquibase.LiquibaseA org.springframework.boot.autoconfigure.orm.jpa.HibernateJpa org.springframework.boot.autoconfigure.transaction.Transact org.springframework.boot.test.autoconfigure.jdbc.TestDataba org.springframework.boot.test.autoconfigure.orm.jpa.TestEnt
@DataLdapTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.data.ldap.LdapReposi org.springframework.boot.autoconfigure.ldap.LdapAutoConfigu org.springframework.boot.autoconfigure.ldap.embedded.Embedd
@DataMongoTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.data.mongo.MongoData org.springframework.boot.autoconfigure.data.mongo.MongoReac org.springframework.boot.autoconfigure.data.mongo.MongoReac org.springframework.boot.autoconfigure.data.mongo.MongoRepo org.springframework.boot.autoconfigure.mongo.MongoAutoConfi org.springframework.boot.autoconfigure.mongo.MongoReactiveA org.springframework.boot.autoconfigure.mongo.embedded.Embed
@DataNeo4jTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.data.neo4j.Neo4jData org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepo org.springframework.boot.autoconfigure.transaction.Transact
@DataRedisTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.data.redis.RedisAuto org.springframework.boot.autoconfigure.data.redis.RedisRepo
@JdbcTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.flyway.FlywayAutoCo org.springframework.boot.autoconfigure.jdbc.DataSourceAuto org.springframework.boot.autoconfigure.jdbc.DataSourceTrans org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAu org.springframework.boot.autoconfigure.liquibase.LiquibaseA org.springframework.boot.autoconfigure.transaction.Transact

测试切片	导入的自动配置
	org.springframework.boot.test.autoconfigure.jdbc.TestDataba
@JooqTest	org.springframework.boot.autoconfigure.flyway.FlywayAutoCon org.springframework.boot.autoconfigure.jdbc.DataSourceAutoC org.springframework.boot.autoconfigure.jdbc.DataSourceTrans org.springframework.boot.autoconfigure.jooq.JooqAutoConfigu org.springframework.boot.autoconfigure.liquibase.LiquibaseA org.springframework.boot.autoconfigure.transaction.Transact
@JsonTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.gson.GsonAutoConfigu org.springframework.boot.autoconfigure.jackson.JacksonAutoC org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfi org.springframework.boot.test.autoconfigure.json.JsonTester
@RestClientTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.gson.GsonAutoConfigu org.springframework.boot.autoconfigure.http.HttpMessageConv org.springframework.boot.autoconfigure.http.codec.CodecsAut org.springframework.boot.autoconfigure.jackson.JacksonAutoC org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfi org.springframework.boot.autoconfigure.web.client.RestTempl org.springframework.boot.autoconfigure.web.reactive.functio org.springframework.boot.test.autoconfigure.web.client.Mock org.springframework.boot.test.autoconfigure.web.client.WebC
@WebFluxTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.context.MessageSourc org.springframework.boot.autoconfigure.gson.GsonAutoConfigu org.springframework.boot.autoconfigure.http.codec.CodecsAut org.springframework.boot.autoconfigure.jackson.JacksonAutoC org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfi org.springframework.boot.autoconfigure.security.reactive.Re org.springframework.boot.autoconfigure.security.reactive.Re org.springframework.boot.autoconfigure.validation.Validatio org.springframework.boot.autoconfigure.web.reactive.WebFlux org.springframework.boot.test.autoconfigure.web.reactive.We
@WebMvcTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfi org.springframework.boot.autoconfigure.context.MessageSourc org.springframework.boot.autoconfigure.data.web.SpringDataW org.springframework.boot.autoconfigure.freemarker.FreeMarke org.springframework.boot.autoconfigure.groovy.template.Groo org.springframework.boot.autoconfigure.gson.GsonAutoConfig org.springframework.boot.autoconfigure.hateoas.HypermediaA org.springframework.boot.autoconfigure.http.HttpMessageCon org.springframework.boot.autoconfigure.jackson.JacksonAutoC org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfi org.springframework.boot.autoconfigure.mustache.MustacheAu org.springframework.boot.autoconfigure.task.TaskExecutionA org.springframework.boot.autoconfigure.thymeleaf.ThymeleafA

测试切片	导入的自动配置

	`org.springframework.boot.autoconfigure.validation.Validati`
	`org.springframework.boot.autoconfigure.web.servlet.WebMvcA`
	`org.springframework.boot.autoconfigure.web.servlet.error.E`
	`org.springframework.boot.test.autoconfigure.web.servlet.Mo`
	`org.springframework.boot.test.autoconfigure.web.servlet.Mo`
	`org.springframework.boot.test.autoconfigure.web.servlet.Mo`
	`org.springframework.boot.test.autoconfigure.web.servlet.Mo`

附录 E.可执行的 Jar 格式

`spring-boot-loader` 模块允许 Spring Boot 支持可执行 jar 和 war 文件。如果您使用 Maven 插件或 Gradle 插件，则会自动生成可执行 jar，并且您通常不需要知道它们如何工作的详细信息。

如果您需要从不同的构建系统创建可执行 jar，或者如果您只是对底层技术感到好奇，本节提供了一些背景知识。

E.1 嵌套 JAR

Java 没有提供任何标准方法来加载嵌套的 jar 文件（即 jar 文件本身包含在 jar 中）。如果您需要分发可以从命令行运行而不解压缩的自包含应用程序，则可能会出现问题。

为了解决这个问题，许多开发人员使用“阴影”罐子。一个带阴影的罐子将所有类别的所有类别打包成一个“超级罐子”。阴影罐的问题在于很难看出哪些库实际上在您的应用程序中。如果在多个罐子中使用相同的文件名（但具有不同的内容），也可能会有问题。Spring Boot 采用不同的方法，让你直接嵌套罐子。

E.1.1 可执行 Jar 文件结构

Spring Boot 与 Loader 兼容的 jar 文件应按以下方式构建：

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-BOOT-INF
    +-classes
    |  +-mycompany
    |     +-project
    |        +-YourClasses.class
    +-lib
       +-dependency1.jar
       +-dependency2.jar
```

应用程序类应放在嵌套的 `BOOT-INF/classes` 目录中。依赖项应放在嵌套的 `BOOT-INF/lib` 目录中。

E.1.2 可执行文件 War 文件结构

Spring Boot 与 Loader 兼容的 war 文件应按以下方式构建：

```
example.war
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |     +-mycompany
    |        +-project
    |           +-YourClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
       +-servlet-api.jar
       +-dependency3.jar
```

依赖项应放在嵌套的 `WEB-INF/lib` 目录中。运行嵌入式时所需的任何依赖项，但在部署到传统 Web 容器时不需要，应放在 `WEB-INF/lib-provided` 中。

E.2 Spring Boot 的“JarFile”类

用于支持加载嵌套 jar 的核心类是 `org.springframework.boot.loader.jar.JarFile`。它允许您从标准 jar 文件或嵌套子 jar 数据中加载 jar 内容。首次加载时，每个 `JarEntry` 的位置将映射到外部 jar 的物理文件偏移量，如以下示例所示：

```
myapp.jar
+-------------------+-------------------------+
| /BOOT-INF/classes | /BOOT-INF/lib/mylib.jar |
|+-----------------+||+-----------+----------+|
||     A.class      |||  B.class  |  C.class ||
|+-----------------+||+-----------+----------+|
+-------------------+-------------------------+
 ^                   ^           ^
 0063                3452        3980
```

上面的示例显示如何在位于 `0063` 的 `myapp.jar` `/BOOT-INF/classes` 中找到 `A.class`。来自嵌套 jar 的 `B.class` 实际上可以在位置 `3452` 的 `myapp.jar` 中找到，而 `C.class` 位于 `3980` 位置。

有了这些信息，我们可以通过寻找外部 jar 的适当部分来加载特定的嵌套条目。我们不需要解压缩归档文件，也不需要将所有条目数据读入内存。

E.2.1 与标准 Java“JarFile”的兼容性

Spring Boot Loader 努力保持与现有代码和库的兼容性。`org.springframework.boot.loader.jar.JarFile` 从 `java.util.jar.JarFile` 延伸，应该作为替代品。`getURL()`方法返回 `URL`，打开与 `java.net.JarURLConnection` 兼容的连接，并可与 Java 的 `URLClassLoader` 一起使用。

E.3 启动可执行的 Jars

`org.springframework.boot.loader.Launcher` 类是一个特殊的引导类，用作可执行 jar 的主要入口点。这是您的 jar 文件中的实际 `Main-Class`，它用于设置适当的 `URLClassLoader`

并最终调用您的 main()方法。

有三个启动器子类（JarLauncher，WarLauncher 和 PropertiesLauncher）。它们的目的是从嵌套的 jar 文件或目录中的 war 文件加载资源（.class 文件等。）（而不是在类路径上显式的那些文件）。在 JarLauncher 和 WarLauncher 的情况下，嵌套路径是固定的。JarLauncher 查看 BOOT-INF/lib/，WarLauncher 查看 WEB-INF/lib/和 WEB-INF/lib-provided/。如果您需要更多，可以在这些位置添加额外的罐子。默认情况下，PropertiesLauncher 会在您的应用程序存档中查找 BOOT-INF/lib/，但您可以通过在 loader.properties 中设置名为 LOADER_PATH 或 loader.path 的环境变量来添加其他位置（这是一个逗号 - 存档中的目录，存档或目录的分隔列表）。

E.3.1 发射器清单

您需要指定适当的 Launcher 作为 META-INF/MANIFEST.MF 的 Main-Class 属性。应在 Start-Class 属性中指定要启动的实际类（即包含 main 方法的类）。

以下示例显示了可执行 jar 文件的典型 MANIFEST.MF：

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

对于 war 文件，它将如下：

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

注意

您无需在清单文件中指定 Class-Path 条目。类路径是从嵌套的 jar 中推导出来的。

E.3.2 分解档案

某些 PaaS 实现可能会选择在运行之前解压缩档案。例如，Cloud Foundry 就是这样运作的。您可以通过启动相应的启动程序来运行解压缩的存档，如下所示：

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```

E.4 PropertiesLauncher 特点

PropertiesLauncher 具有一些可以使用外部属性启用的特殊功能（系统属性，环境变量，清单条目或 loader.properties）。下表描述了这些属性：

键	目的
loader.path	Comma-separated Classpath, such as lib,${HOME}/app/lib. Earlier entries take precedence, like a regular -classpath on the javac command line.
loader.home	Used to resolve relative paths in loader.path. For example, given loader.path=lib, then ${loader.home}/lib is a classpath location (along with all jar files in that directory). This property is also used to locate a loader.properties file, as in the following

键	目的
	example /opt/app It defaults to ${user.dir}.
loader.args	Default arguments for the main method (space separated).
loader.main	Name of main class to launch (for example, com.app.Application).
loader.config.name	Name of properties file (for example, launcher) It defaults to loader.
loader.config.location	Path to properties file (for example, classpath:loader.properties). It defaults to loader.properties.
loader.system	Boolean flag to indicate that all properties should be added to System properties It defaults to false.

指定为环境变量或清单条目时，应使用以下名称：

键	清单入口	环境变量
loader.path	Loader-Path	LOADER_PATH
loader.home	Loader-Home	LOADER_HOME
loader.args	Loader-Args	LOADER_ARGS
loader.main	Start-Class	LOADER_MAIN
loader.config.location	Loader-Config-Location	LOADER_CONFIG_LOCATION
loader.system	Loader-System	LOADER_SYSTEM

小费

构建胖罐时，构建插件会自动将 Main-Class 属性移动到 Start-Class。如果您使用它，请使用 Main-Class 属性指定要启动的类的名称，并省略 Start-Class。

以下规则适用于使用 PropertiesLauncher：

- loader.properties 在 loader.home 中搜索，然后在类路径的根目录中搜索，然后在 classpath:/BOOT-INF/classes 中搜索。使用具有该名称的文件的第一个位置。
- loader.home 是仅在未指定 loader.config.location 时附加属性文件的目录位置（覆盖默认值）。
- loader.path 可以包含目录（以递归方式扫描 jar 和 zip 文件），存档路径，存档中扫描 jar 文件的目录（例如，dependencies.jar!/lib）或通配符模式（默认情况下） JVM 行为）。存档路径可以相对于 loader.home 或文件系统中具有 jar:file:前缀的任何位置。
- loader.path（如果为空）默认为 BOOT-INF/lib（表示本地目录或嵌套的目录，如果从存档运行）。因此，PropertiesLauncher 在没有提供其他配置时与 JarLauncher

的行为相同。
- `loader.path` 不能用于配置 `loader.properties` 的位置（用于搜索后者的类路径是启动 `PropertiesLauncher` 时的 JVM 类路径）。
- 占位符替换是在使用前从系统和环境变量以及属性文件本身在所有值上完成的。
- 属性的搜索顺序（在多个位置查看的位置）是环境变量，系统属性，`loader.properties`，展开的存档清单和存档清单。

E.5 可执行的 Jar 限制

使用 Spring Boot Loader 打包应用程序时，需要考虑以下限制：

- Zip 条目压缩：必须使用 `ZipEntry.STORED` 方法保存嵌套 jar 的 `ZipEntry`。这是必需的，以便我们可以直接寻找嵌套 jar 中的单个内容。嵌套的 jar 文件本身的内容仍然可以被压缩，外部 jar 中的任何其他条目也是如此。

- System classLoader：启动的应用程序在加载类时应该使用 `Thread.getContextClassLoader()`（大多数库和框架默认都这样做）。尝试使用 `ClassLoader.getSystemClassLoader()`加载嵌套的 jar 类失败。`java.util.Logging` 始终使用系统类加载器。因此，您应该考虑使用不同的日志记录实现。

E.6 替代单罐解决方案

如果前面的限制意味着您不能使用 Spring Boot Loader，请考虑以下备选方案：

- Maven Shade 插件
- 的 JarClassLoader
- OneJar
- Gradle Shadow Plugin

附录 F.依赖版本

下表提供了 Spring Boot 在其 CLI（命令行界面），Maven 依赖关系管理和 Gradle 插件中提供的所有依赖关系版本的详细信息。在未声明版本的情况下声明对其中一个工件的依赖关系时，将使用表中列出的版本。

组 ID	工件 ID
antlr	antlr
ch.qos.logback	logback-access
ch.qos.logback	logback-classic
ch.qos.logback	logback-core
com.atomikos	transactions-jdbc
com.atomikos	transactions-jms
com.atomikos	transactions-jta

组 ID	工件 ID
com.couchbase.client	couchbase-spring-cache
com.couchbase.client	java-client
com.datastax.cassandra	cassandra-driver-core
com.datastax.cassandra	cassandra-driver-mapping
com.fasterxml	classmate
com.fasterxml.jackson.core	jackson-annotations
com.fasterxml.jackson.core	jackson-core
com.fasterxml.jackson.core	jackson-databind
com.fasterxml.jackson.dataformat	jackson-dataformat-avro
com.fasterxml.jackson.dataformat	jackson-dataformat-cbor
com.fasterxml.jackson.dataformat	jackson-dataformat-csv
com.fasterxml.jackson.dataformat	jackson-dataformat-ion
com.fasterxml.jackson.dataformat	jackson-dataformat-properti
com.fasterxml.jackson.dataformat	jackson-dataformat-protobuf
com.fasterxml.jackson.dataformat	jackson-dataformat-smile
com.fasterxml.jackson.dataformat	jackson-dataformat-xml
com.fasterxml.jackson.dataformat	jackson-dataformat-yaml
com.fasterxml.jackson.datatype	jackson-datatype-guava
com.fasterxml.jackson.datatype	jackson-datatype-hibernate3
com.fasterxml.jackson.datatype	jackson-datatype-hibernate4
com.fasterxml.jackson.datatype	jackson-datatype-hibernate5
com.fasterxml.jackson.datatype	jackson-datatype-hppc
com.fasterxml.jackson.datatype	jackson-datatype-jaxrs
com.fasterxml.jackson.datatype	jackson-datatype-jdk8
com.fasterxml.jackson.datatype	jackson-datatype-joda
com.fasterxml.jackson.datatype	jackson-datatype-json-org

组 ID	工件 ID
com.fasterxml.jackson.datatype	jackson-datatype-jsr310
com.fasterxml.jackson.datatype	jackson-datatype-jsr353
com.fasterxml.jackson.datatype	jackson-datatype-pcollectio
com.fasterxml.jackson.jaxrs	jackson-jaxrs-base
com.fasterxml.jackson.jaxrs	jackson-jaxrs-cbor-provider
com.fasterxml.jackson.jaxrs	jackson-jaxrs-json-provider
com.fasterxml.jackson.jaxrs	jackson-jaxrs-smile-provide
com.fasterxml.jackson.jaxrs	jackson-jaxrs-xml-provider
com.fasterxml.jackson.jaxrs	jackson-jaxrs-yaml-provider
com.fasterxml.jackson.jr	jackson-jr-all
com.fasterxml.jackson.jr	jackson-jr-objects
com.fasterxml.jackson.jr	jackson-jr-retrofit2
com.fasterxml.jackson.jr	jackson-jr-stree
com.fasterxml.jackson.module	jackson-module-afterburner
com.fasterxml.jackson.module	jackson-module-guice
com.fasterxml.jackson.module	jackson-module-jaxb-annotat
com.fasterxml.jackson.module	jackson-module-jsonSchema
com.fasterxml.jackson.module	jackson-module-kotlin
com.fasterxml.jackson.module	jackson-module-mrbean
com.fasterxml.jackson.module	jackson-module-osgi
com.fasterxml.jackson.module	jackson-module-parameter-na
com.fasterxml.jackson.module	jackson-module-paranamer
com.fasterxml.jackson.module	jackson-module-scala_2.10
com.fasterxml.jackson.module	jackson-module-scala_2.11
com.fasterxml.jackson.module	jackson-module-scala_2.12

组 ID	工件 ID
com.github.ben-manes.caffeine	caffeine
com.github.ben-manes.caffeine	guava
com.github.ben-manes.caffeine	jcache
com.github.ben-manes.caffeine	simulator
com.github.mxab.thymeleaf.extras	thymeleaf-extras-data-attri
com.google.appengine	appengine-api-1.0-sdk
com.google.code.gson	gson
com.h2database	h2
com.hazelcast	hazelcast
com.hazelcast	hazelcast-client
com.hazelcast	hazelcast-hibernate52
com.hazelcast	hazelcast-spring
com.jayway.jsonpath	json-path
com.jayway.jsonpath	json-path-assert
com.microsoft.sqlserver	mssql-jdbc
com.querydsl	querydsl-apt
com.querydsl	querydsl-collections
com.querydsl	querydsl-core
com.querydsl	querydsl-jpa
com.querydsl	querydsl-mongodb
com.rabbitmq	amqp-client
com.samskivert	jmustache
com.sendgrid	sendgrid-java
com.sun.mail	javax.mail
com.sun.xml.messaging.saaj	saaj-impl

组 ID	工件 ID
com.timgroup	java-statsd-client
com.unboundid	unboundid-ldapsdk
com.zaxxer	HikariCP
commons-codec	commons-codec
commons-pool	commons-pool
de.flapdoodle.embed	de.flapdoodle.embed.mongo
dom4j	dom4j
io.dropwizard.metrics	metrics-annotation
io.dropwizard.metrics	metrics-core
io.dropwizard.metrics	metrics-ehcache
io.dropwizard.metrics	metrics-graphite
io.dropwizard.metrics	metrics-healthchecks
io.dropwizard.metrics	metrics-httpasyncclient
io.dropwizard.metrics	metrics-jdbi
io.dropwizard.metrics	metrics-jersey
io.dropwizard.metrics	metrics-jersey2
io.dropwizard.metrics	metrics-jetty8
io.dropwizard.metrics	metrics-jetty9
io.dropwizard.metrics	metrics-jetty9-legacy
io.dropwizard.metrics	metrics-jmx
io.dropwizard.metrics	metrics-json
io.dropwizard.metrics	metrics-jvm
io.dropwizard.metrics	metrics-log4j
io.dropwizard.metrics	metrics-log4j2
io.dropwizard.metrics	metrics-logback

组 ID	工件 ID
io.dropwizard.metrics	metrics-servlet
io.dropwizard.metrics	metrics-servlets
io.lettuce	lettuce-core
io.micrometer	micrometer-core
io.micrometer	micrometer-jersey2
io.micrometer	micrometer-registry-appopti
io.micrometer	micrometer-registry-atlas
io.micrometer	micrometer-registry-azure-m
io.micrometer	micrometer-registry-cloudwa
io.micrometer	micrometer-registry-datadog
io.micrometer	micrometer-registry-dynatra
io.micrometer	micrometer-registry-elastic
io.micrometer	micrometer-registry-ganglia
io.micrometer	micrometer-registry-graphit
io.micrometer	micrometer-registry-humio
io.micrometer	micrometer-registry-influx
io.micrometer	micrometer-registry-jmx
io.micrometer	micrometer-registry-kairos
io.micrometer	micrometer-registry-new-rel
io.micrometer	micrometer-registry-prometh
io.micrometer	micrometer-registry-signalf
io.micrometer	micrometer-registry-statsd
io.micrometer	micrometer-registry-wavefro
io.micrometer	micrometer-test
io.netty	netty-all

组 ID	工件 ID
io.netty	netty-buffer
io.netty	netty-codec
io.netty	netty-codec-dns
io.netty	netty-codec-haproxy
io.netty	netty-codec-http
io.netty	netty-codec-http2
io.netty	netty-codec-memcache
io.netty	netty-codec-mqtt
io.netty	netty-codec-redis
io.netty	netty-codec-smtp
io.netty	netty-codec-socks
io.netty	netty-codec-stomp
io.netty	netty-codec-xml
io.netty	netty-common
io.netty	netty-dev-tools
io.netty	netty-example
io.netty	netty-handler
io.netty	netty-handler-proxy
io.netty	netty-resolver
io.netty	netty-resolver-dns
io.netty	netty-tcnative-boringssl-st
io.netty	netty-transport
io.netty	netty-transport-native-epol
io.netty	netty-transport-native-kque
io.netty	netty-transport-native-unix

组 ID	工件 ID
io.netty	netty-transport-rxtx
io.netty	netty-transport-sctp
io.netty	netty-transport-udt
io.projectreactor	reactor-core
io.projectreactor	reactor-test
io.projectreactor.addons	reactor-adapter
io.projectreactor.addons	reactor-extra
io.projectreactor.addons	reactor-logback
io.projectreactor.kafka	reactor-kafka
io.projectreactor.netty	reactor-netty
io.prometheus	simpleclient_pushgateway
io.reactivex	rxjava
io.reactivex	rxjava-reactive-streams
io.reactivex.rxjava2	rxjava
io.rest-assured	json-path
io.rest-assured	json-schema-validator
io.rest-assured	rest-assured
io.rest-assured	scala-support
io.rest-assured	spring-mock-mvc
io.rest-assured	xml-path
io.searchbox	jest
io.undertow	undertow-core
io.undertow	undertow-servlet
io.undertow	undertow-websockets-jsr
javax.activation	javax.activation-api

组 ID	工件 ID
javax.annotation	javax.annotation-api
javax.cache	cache-api
javax.jms	javax.jms-api
javax.json	javax.json-api
javax.json.bind	javax.json.bind-api
javax.mail	javax.mail-api
javax.money	money-api
javax.persistence	javax.persistence-api
javax.servlet	javax.servlet-api
javax.servlet	jstl
javax.transaction	javax.transaction-api
javax.validation	validation-api
javax.websocket	javax.websocket-api
javax.xml.bind	jaxb-api
javax.xml.ws	jaxws-api
jaxen	jaxen
joda-time	joda-time
junit	junit
mysql	mysql-connector-java
net.bytebuddy	byte-buddy
net.bytebuddy	byte-buddy-agent
net.java.dev.jna	jna
net.java.dev.jna	jna-platform
net.sf.ehcache	ehcache
net.sourceforge.htmlunit	htmlunit

组 ID	工件 ID
net.sourceforge.jtds	jtds
net.sourceforge.nekohtml	nekohtml
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect
org.apache.activemq	activemq-amqp
org.apache.activemq	activemq-blueprint
org.apache.activemq	activemq-broker
org.apache.activemq	activemq-camel
org.apache.activemq	activemq-client
org.apache.activemq	activemq-console
org.apache.activemq	activemq-http
org.apache.activemq	activemq-jaas
org.apache.activemq	activemq-jdbc-store
org.apache.activemq	activemq-jms-pool
org.apache.activemq	activemq-kahadb-store
org.apache.activemq	activemq-karaf
org.apache.activemq	activemq-leveldb-store
org.apache.activemq	activemq-log4j-appender
org.apache.activemq	activemq-mqtt
org.apache.activemq	activemq-openwire-generator
org.apache.activemq	activemq-openwire-legacy
org.apache.activemq	activemq-osgi
org.apache.activemq	activemq-partition
org.apache.activemq	activemq-pool
org.apache.activemq	activemq-ra
org.apache.activemq	activemq-run

组 ID	工件 ID
org.apache.activemq	activemq-runtime-config
org.apache.activemq	activemq-shiro
org.apache.activemq	activemq-spring
org.apache.activemq	activemq-stomp
org.apache.activemq	activemq-web
org.apache.activemq	artemis-amqp-protocol
org.apache.activemq	artemis-commons
org.apache.activemq	artemis-core-client
org.apache.activemq	artemis-jms-client
org.apache.activemq	artemis-jms-server
org.apache.activemq	artemis-journal
org.apache.activemq	artemis-native
org.apache.activemq	artemis-selector
org.apache.activemq	artemis-server
org.apache.activemq	artemis-service-extensions
org.apache.commons	commons-dbcp2
org.apache.commons	commons-lang3
org.apache.commons	commons-pool2
org.apache.derby	derby
org.apache.httpcomponents	fluent-hc
org.apache.httpcomponents	httpasyncclient
org.apache.httpcomponents	httpclient
org.apache.httpcomponents	httpclient-cache
org.apache.httpcomponents	httpclient-osgi
org.apache.httpcomponents	httpclient-win

组 ID	工件 ID
org.apache.httpcomponents	httpcore
org.apache.httpcomponents	httpcore-nio
org.apache.httpcomponents	httpmime
org.apache.johnzon	johnzon-core
org.apache.johnzon	johnzon-jaxrs
org.apache.johnzon	johnzon-jsonb
org.apache.johnzon	johnzon-jsonb-extras
org.apache.johnzon	johnzon-jsonschema
org.apache.johnzon	johnzon-mapper
org.apache.johnzon	johnzon-websocket
org.apache.kafka	connect-api
org.apache.kafka	connect-file
org.apache.kafka	connect-json
org.apache.kafka	connect-runtime
org.apache.kafka	connect-transforms
org.apache.kafka	kafka_2.11
org.apache.kafka	kafka_2.12
org.apache.kafka	kafka-clients
org.apache.kafka	kafka-log4j-appender
org.apache.kafka	kafka-streams
org.apache.kafka	kafka-tools
org.apache.logging.log4j	log4j-1.2-api
org.apache.logging.log4j	log4j-api
org.apache.logging.log4j	log4j-cassandra
org.apache.logging.log4j	log4j-core

组 ID	工件 ID
org.apache.logging.log4j	log4j-couchdb
org.apache.logging.log4j	log4j-flume-ng
org.apache.logging.log4j	log4j-iostreams
org.apache.logging.log4j	log4j-jcl
org.apache.logging.log4j	log4j-jmx-gui
org.apache.logging.log4j	log4j-jpa
org.apache.logging.log4j	log4j-jul
org.apache.logging.log4j	log4j-liquibase
org.apache.logging.log4j	log4j-mongodb2
org.apache.logging.log4j	log4j-mongodb3
org.apache.logging.log4j	log4j-slf4j-impl
org.apache.logging.log4j	log4j-taglib
org.apache.logging.log4j	log4j-to-slf4j
org.apache.logging.log4j	log4j-web
org.apache.solr	solr-analysis-extras
org.apache.solr	solr-analytics
org.apache.solr	solr-cell
org.apache.solr	solr-clustering
org.apache.solr	solr-core
org.apache.solr	solr-dataimporthandler
org.apache.solr	solr-dataimporthandler-extr
org.apache.solr	solr-langid
org.apache.solr	solr-ltr
org.apache.solr	solr-solrj
org.apache.solr	solr-test-framework

组 ID	工件 ID
org.apache.solr	solr-uima
org.apache.solr	solr-velocity
org.apache.tomcat	tomcat-annotations-api
org.apache.tomcat	tomcat-catalina-jmx-remote
org.apache.tomcat	tomcat-jdbc
org.apache.tomcat	tomcat-jsp-api
org.apache.tomcat.embed	tomcat-embed-core
org.apache.tomcat.embed	tomcat-embed-el
org.apache.tomcat.embed	tomcat-embed-jasper
org.apache.tomcat.embed	tomcat-embed-websocket
org.aspectj	aspectjrt
org.aspectj	aspectjtools
org.aspectj	aspectjweaver
org.assertj	assertj-core
org.codehaus.btm	btm
org.codehaus.groovy	groovy
org.codehaus.groovy	groovy-ant
org.codehaus.groovy	groovy-bsf
org.codehaus.groovy	groovy-console
org.codehaus.groovy	groovy-docgenerator
org.codehaus.groovy	groovy-groovydoc
org.codehaus.groovy	groovy-groovysh
org.codehaus.groovy	groovy-jmx
org.codehaus.groovy	groovy-json
org.codehaus.groovy	groovy-jsr223

组 ID	工件 ID
org.codehaus.groovy	groovy-nio
org.codehaus.groovy	groovy-servlet
org.codehaus.groovy	groovy-sql
org.codehaus.groovy	groovy-swing
org.codehaus.groovy	groovy-templates
org.codehaus.groovy	groovy-test
org.codehaus.groovy	groovy-testng
org.codehaus.groovy	groovy-xml
org.codehaus.janino	janino
org.eclipse.jetty	apache-jsp
org.eclipse.jetty	apache-jstl
org.eclipse.jetty	jetty-alpn-client
org.eclipse.jetty	jetty-alpn-conscrypt-client
org.eclipse.jetty	jetty-alpn-conscrypt-server
org.eclipse.jetty	jetty-alpn-java-client
org.eclipse.jetty	jetty-alpn-java-server
org.eclipse.jetty	jetty-alpn-openjdk8-client
org.eclipse.jetty	jetty-alpn-openjdk8-server
org.eclipse.jetty	jetty-alpn-server
org.eclipse.jetty	jetty-annotations

组 ID	工件 ID
org.eclipse.jetty	jetty-ant
org.eclipse.jetty	jetty-client
org.eclipse.jetty	jetty-continuation
org.eclipse.jetty	jetty-deploy
org.eclipse.jetty	jetty-distribution
org.eclipse.jetty	jetty-hazelcast
org.eclipse.jetty	jetty-home
org.eclipse.jetty	jetty-http
org.eclipse.jetty	jetty-http-spi
org.eclipse.jetty	jetty-infinispan
org.eclipse.jetty	jetty-io
org.eclipse.jetty	jetty-jaas
org.eclipse.jetty	jetty-jaspi
org.eclipse.jetty	jetty-jmx
org.eclipse.jetty	jetty-jndi
org.eclipse.jetty	jetty-nosql
org.eclipse.jetty	jetty-plus

组 ID	工件 ID
org.eclipse.jetty	jetty-proxy
org.eclipse.jetty	jetty-quickstart
org.eclipse.jetty	jetty-reactive-httpclient
org.eclipse.jetty	jetty-rewrite
org.eclipse.jetty	jetty-security
org.eclipse.jetty	jetty-server
org.eclipse.jetty	jetty-servlet
org.eclipse.jetty	jetty-servlets
org.eclipse.jetty	jetty-spring
org.eclipse.jetty	jetty-unixsocket
org.eclipse.jetty	jetty-util
org.eclipse.jetty	jetty-util-ajax
org.eclipse.jetty	jetty-webapp
org.eclipse.jetty	jetty-xml
org.eclipse.jetty.cdi	cdi-core
org.eclipse.jetty.cdi	cdi-servlet
org.eclipse.jetty.fcgi	fcgi-client

组 ID	工件 ID
org.eclipse.jetty.fcgi	fcgi-server
org.eclipse.jetty.gcloud	jetty-gcloud-session-manage
org.eclipse.jetty.http2	http2-client
org.eclipse.jetty.http2	http2-common
org.eclipse.jetty.http2	http2-hpack
org.eclipse.jetty.http2	http2-http-client-transport
org.eclipse.jetty.http2	http2-server
org.eclipse.jetty.memcached	jetty-memcached-sessions
org.eclipse.jetty.orbit	javax.servlet.jsp
org.eclipse.jetty.osgi	jetty-httpservice
org.eclipse.jetty.osgi	jetty-osgi-boot
org.eclipse.jetty.osgi	jetty-osgi-boot-jsp
org.eclipse.jetty.osgi	jetty-osgi-boot-warurl
org.eclipse.jetty.websocket	javax-websocket-client-impl
org.eclipse.jetty.websocket	javax-websocket-server-impl
org.eclipse.jetty.websocket	websocket-api
org.eclipse.jetty.websocket	websocket-client
org.eclipse.jetty.websocket	websocket-common

组 ID	工件 ID
org.eclipse.jetty.websocket	websocket-server
org.eclipse.jetty.websocket	websocket-servlet
org.ehcache	ehcache
org.ehcache	ehcache-clustered
org.ehcache	ehcache-transactions
org.elasticsearch	elasticsearch
org.elasticsearch.client	elasticsearch-rest-client
org.elasticsearch.client	elasticsearch-rest-high-lev
org.elasticsearch.client	transport
org.elasticsearch.distribution.integ-test-zip	elasticsearch
org.elasticsearch.plugin	transport-netty4-client
org.firebirdsql.jdbc	jaybird-jdk17
org.firebirdsql.jdbc	jaybird-jdk18
org.flywaydb	flyway-core
org.freemarker	freemarker
org.glassfish	javax.el
org.glassfish.jaxb	jaxb-runtime
org.glassfish.jersey.bundles	jaxrs-ri
org.glassfish.jersey.connectors	jersey-apache-connector
org.glassfish.jersey.connectors	jersey-grizzly-connector
org.glassfish.jersey.connectors	jersey-jdk-connector
org.glassfish.jersey.connectors	jersey-jetty-connector
org.glassfish.jersey.connectors	jersey-netty-connector
org.glassfish.jersey.containers	jersey-container-grizzly2-h

组 ID	工件 ID
org.glassfish.jersey.containers	jersey-container-grizzly2-s
org.glassfish.jersey.containers	jersey-container-jdk-http
org.glassfish.jersey.containers	jersey-container-jetty-http
org.glassfish.jersey.containers	jersey-container-jetty-serv
org.glassfish.jersey.containers	jersey-container-netty-http
org.glassfish.jersey.containers	jersey-container-servlet
org.glassfish.jersey.containers	jersey-container-servlet-co
org.glassfish.jersey.containers	jersey-container-simple-htt
org.glassfish.jersey.containers.glassfish	jersey-gf-ejb
org.glassfish.jersey.core	jersey-client
org.glassfish.jersey.core	jersey-common
org.glassfish.jersey.core	jersey-server
org.glassfish.jersey.ext	jersey-bean-validation
org.glassfish.jersey.ext	jersey-declarative-linking
org.glassfish.jersey.ext	jersey-entity-filtering
org.glassfish.jersey.ext	jersey-metainf-services
org.glassfish.jersey.ext	jersey-mvc
org.glassfish.jersey.ext	jersey-mvc-bean-validation
org.glassfish.jersey.ext	jersey-mvc-freemarker
org.glassfish.jersey.ext	jersey-mvc-jsp
org.glassfish.jersey.ext	jersey-mvc-mustache
org.glassfish.jersey.ext	jersey-proxy-client
org.glassfish.jersey.ext	jersey-servlet-portability
org.glassfish.jersey.ext	jersey-spring4
org.glassfish.jersey.ext	jersey-wadl-doclet

组 ID	工件 ID
org.glassfish.jersey.ext.cdi	jersey-cdi1x
org.glassfish.jersey.ext.cdi	jersey-cdi1x-ban-custom-hk2
org.glassfish.jersey.ext.cdi	jersey-cdi1x-servlet
org.glassfish.jersey.ext.cdi	jersey-cdi1x-transaction
org.glassfish.jersey.ext.cdi	jersey-cdi1x-validation
org.glassfish.jersey.ext.cdi	jersey-weld2-se
org.glassfish.jersey.ext.rx	jersey-rx-client-guava
org.glassfish.jersey.ext.rx	jersey-rx-client-rxjava
org.glassfish.jersey.ext.rx	jersey-rx-client-rxjava2
org.glassfish.jersey.inject	jersey-cdi2-se
org.glassfish.jersey.inject	jersey-hk2
org.glassfish.jersey.media	jersey-media-jaxb
org.glassfish.jersey.media	jersey-media-json-binding
org.glassfish.jersey.media	jersey-media-json-jackson
org.glassfish.jersey.media	jersey-media-json-jackson1
org.glassfish.jersey.media	jersey-media-json-jettison
org.glassfish.jersey.media	jersey-media-json-processin
org.glassfish.jersey.media	jersey-media-kryo
org.glassfish.jersey.media	jersey-media-moxy
org.glassfish.jersey.media	jersey-media-multipart
org.glassfish.jersey.media	jersey-media-sse
org.glassfish.jersey.security	oauth1-client
org.glassfish.jersey.security	oauth1-server
org.glassfish.jersey.security	oauth1-signature
org.glassfish.jersey.security	oauth2-client

组 ID	工件 ID
org.glassfish.jersey.test-framework	jersey-test-framework-core
org.glassfish.jersey.test-framework	jersey-test-framework-util
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provi
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provi
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provi
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provi
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provi
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provi
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provi
org.hamcrest	hamcrest-core
org.hamcrest	hamcrest-library
org.hibernate	hibernate-c3p0
org.hibernate	hibernate-core
org.hibernate	hibernate-ehcache
org.hibernate	hibernate-entitymanager
org.hibernate	hibernate-envers
org.hibernate	hibernate-hikaricp
org.hibernate	hibernate-java8
org.hibernate	hibernate-jcache
org.hibernate	hibernate-jpamodelgen
org.hibernate	hibernate-proxool
org.hibernate	hibernate-spatial
org.hibernate	hibernate-testing
org.hibernate.validator	hibernate-validator
org.hibernate.validator	hibernate-validator-annotat

组 ID	工件 ID
org.hsqldb	hsqldb
org.infinispan	infinispan-cachestore-jdbc
org.infinispan	infinispan-cachestore-jpa
org.infinispan	infinispan-cachestore-level
org.infinispan	infinispan-cachestore-remot
org.infinispan	infinispan-cachestore-rest
org.infinispan	infinispan-cachestore-rocks
org.infinispan	infinispan-cdi-common
org.infinispan	infinispan-cdi-embedded
org.infinispan	infinispan-cdi-remote
org.infinispan	infinispan-client-hotrod
org.infinispan	infinispan-cloud
org.infinispan	infinispan-clustered-counte
org.infinispan	infinispan-clustered-lock
org.infinispan	infinispan-commons
org.infinispan	infinispan-core
org.infinispan	infinispan-directory-provid
org.infinispan	infinispan-hibernate-cache-
org.infinispan	infinispan-jcache
org.infinispan	infinispan-jcache-commons
org.infinispan	infinispan-jcache-remote
org.infinispan	infinispan-lucene-directory
org.infinispan	infinispan-objectfilter
org.infinispan	infinispan-osgi
org.infinispan	infinispan-persistence-cli

组 ID	工件 ID
org.infinispan	infinispan-persistence-soft
org.infinispan	infinispan-query
org.infinispan	infinispan-query-dsl
org.infinispan	infinispan-remote-query-cli
org.infinispan	infinispan-remote-query-ser
org.infinispan	infinispan-scripting
org.infinispan	infinispan-server-core
org.infinispan	infinispan-server-hotrod
org.infinispan	infinispan-server-memcached
org.infinispan	infinispan-server-router
org.infinispan	infinispan-spring4-common
org.infinispan	infinispan-spring4-embedded
org.infinispan	infinispan-spring4-remote
org.infinispan	infinispan-tasks
org.infinispan	infinispan-tasks-api
org.infinispan	infinispan-tools
org.infinispan	infinispan-tree
org.influxdb	influxdb-java
org.jboss	jboss-transaction-spi
org.jboss.logging	jboss-logging
org.jdom	jdom2
org.jetbrains.kotlin	kotlin-reflect
org.jetbrains.kotlin	kotlin-runtime
org.jetbrains.kotlin	kotlin-stdlib
org.jetbrains.kotlin	kotlin-stdlib-jdk7

组 ID	工件 ID
org.jetbrains.kotlin	kotlin-stdlib-jdk8
org.jetbrains.kotlin	kotlin-stdlib-jre7
org.jetbrains.kotlin	kotlin-stdlib-jre8
org.jolokia	jolokia-core
org.jooq	jooq
org.jooq	jooq-codegen
org.jooq	jooq-meta
org.junit.jupiter	junit-jupiter-api
org.junit.jupiter	junit-jupiter-engine
org.junit.jupiter	junit-jupiter-migrationsupp
org.junit.jupiter	junit-jupiter-params
org.junit.platform	junit-platform-commons
org.junit.platform	junit-platform-console
org.junit.platform	junit-platform-engine
org.junit.platform	junit-platform-launcher
org.junit.platform	junit-platform-runner
org.junit.platform	junit-platform-suite-api
org.junit.platform	junit-platform-surefire-pro
org.junit.vintage	junit-vintage-engine
org.jvnet.mimepull	mimepull
org.liquibase	liquibase-core
org.mariadb.jdbc	mariadb-java-client
org.messaginghub	pooled-jms
org.mockito	mockito-core
org.mockito	mockito-inline

组 ID	工件 ID
org.mockito	mockito-junit-jupiter
org.mongodb	bson
org.mongodb	mongodb-driver
org.mongodb	mongodb-driver-async
org.mongodb	mongodb-driver-core
org.mongodb	mongodb-driver-reactivestre
org.mongodb	mongo-java-driver
org.mortbay.jasper	apache-el
org.neo4j	neo4j-ogm-api
org.neo4j	neo4j-ogm-bolt-driver
org.neo4j	neo4j-ogm-core
org.neo4j	neo4j-ogm-embedded-driver
org.neo4j	neo4j-ogm-http-driver
org.postgresql	postgresql
org.projectlombok	lombok
org.quartz-scheduler	quartz
org.quartz-scheduler	quartz-jobs
org.reactivestreams	reactive-streams
org.seleniumhq.selenium	htmlunit-driver
org.seleniumhq.selenium	selenium-api
org.seleniumhq.selenium	selenium-chrome-driver
org.seleniumhq.selenium	selenium-edge-driver
org.seleniumhq.selenium	selenium-firefox-driver
org.seleniumhq.selenium	selenium-ie-driver
org.seleniumhq.selenium	selenium-java

组 ID	工件 ID
org.seleniumhq.selenium	selenium-opera-driver
org.seleniumhq.selenium	selenium-remote-driver
org.seleniumhq.selenium	selenium-safari-driver
org.seleniumhq.selenium	selenium-support
org.skyscreamer	jsonassert
org.slf4j	jcl-over-slf4j
org.slf4j	jul-to-slf4j
org.slf4j	log4j-over-slf4j
org.slf4j	slf4j-api
org.slf4j	slf4j-ext
org.slf4j	slf4j-jcl
org.slf4j	slf4j-jdk14
org.slf4j	slf4j-log4j12
org.slf4j	slf4j-nop
org.slf4j	slf4j-simple
org.springframework	spring-aop
org.springframework	spring-aspects
org.springframework	spring-beans
org.springframework	spring-context
org.springframework	spring-context-indexer
org.springframework	spring-context-support
org.springframework	spring-core
org.springframework	spring-expression
org.springframework	spring-instrument
org.springframework	spring-jcl

组 ID	工件 ID
org.springframework	spring-jdbc
org.springframework	spring-jms
org.springframework	spring-messaging
org.springframework	spring-orm
org.springframework	spring-oxm
org.springframework	spring-test
org.springframework	spring-tx
org.springframework	spring-web
org.springframework	spring-webflux
org.springframework	spring-webmvc
org.springframework	spring-websocket
org.springframework.amqp	spring-amqp
org.springframework.amqp	spring-rabbit
org.springframework.amqp	spring-rabbit-junit
org.springframework.amqp	spring-rabbit-test
org.springframework.batch	spring-batch-core
org.springframework.batch	spring-batch-infrastructure
org.springframework.batch	spring-batch-integration
org.springframework.batch	spring-batch-test
org.springframework.boot	spring-boot
org.springframework.boot	spring-boot-actuator
org.springframework.boot	spring-boot-actuator-autoco
org.springframework.boot	spring-boot-autoconfigure
org.springframework.boot	spring-boot-autoconfigure-p
org.springframework.boot	spring-boot-configuration-m

组 ID	工件 ID
org.springframework.boot	spring-boot-configuration-p
org.springframework.boot	spring-boot-devtools
org.springframework.boot	spring-boot-loader
org.springframework.boot	spring-boot-loader-tools
org.springframework.boot	spring-boot-properties-migr
org.springframework.boot	spring-boot-starter
org.springframework.boot	spring-boot-starter-activem
org.springframework.boot	spring-boot-starter-actuato
org.springframework.boot	spring-boot-starter-amqp
org.springframework.boot	spring-boot-starter-aop
org.springframework.boot	spring-boot-starter-artemis
org.springframework.boot	spring-boot-starter-batch
org.springframework.boot	spring-boot-starter-cache
org.springframework.boot	spring-boot-starter-cloud-c
org.springframework.boot	spring-boot-starter-data-ca
org.springframework.boot	spring-boot-starter-data-ca
org.springframework.boot	spring-boot-starter-data-co
org.springframework.boot	spring-boot-starter-data-co
org.springframework.boot	spring-boot-starter-data-el
org.springframework.boot	spring-boot-starter-data-jd
org.springframework.boot	spring-boot-starter-data-jp
org.springframework.boot	spring-boot-starter-data-ld
org.springframework.boot	spring-boot-starter-data-mo
org.springframework.boot	spring-boot-starter-data-mo
org.springframework.boot	spring-boot-starter-data-ne

组 ID	工件 ID
org.springframework.boot	spring-boot-starter-data-re
org.springframework.boot	spring-boot-starter-data-re
org.springframework.boot	spring-boot-starter-data-re
org.springframework.boot	spring-boot-starter-data-so
org.springframework.boot	spring-boot-starter-freemar
org.springframework.boot	spring-boot-starter-groovy-
org.springframework.boot	spring-boot-starter-hateoas
org.springframework.boot	spring-boot-starter-integra
org.springframework.boot	spring-boot-starter-jdbc
org.springframework.boot	spring-boot-starter-jersey
org.springframework.boot	spring-boot-starter-jetty
org.springframework.boot	spring-boot-starter-jooq
org.springframework.boot	spring-boot-starter-json
org.springframework.boot	spring-boot-starter-jta-ato
org.springframework.boot	spring-boot-starter-jta-bit
org.springframework.boot	spring-boot-starter-log4j2
org.springframework.boot	spring-boot-starter-logging
org.springframework.boot	spring-boot-starter-mail
org.springframework.boot	spring-boot-starter-mustach
org.springframework.boot	spring-boot-starter-oauth2-
org.springframework.boot	spring-boot-starter-oauth2-
org.springframework.boot	spring-boot-starter-quartz
org.springframework.boot	spring-boot-starter-reactor
org.springframework.boot	spring-boot-starter-securit
org.springframework.boot	spring-boot-starter-test

组 ID	工件 ID
org.springframework.boot	spring-boot-starter-thymele
org.springframework.boot	spring-boot-starter-tomcat
org.springframework.boot	spring-boot-starter-underto
org.springframework.boot	spring-boot-starter-validat
org.springframework.boot	spring-boot-starter-web
org.springframework.boot	spring-boot-starter-webflux
org.springframework.boot	spring-boot-starter-web-ser
org.springframework.boot	spring-boot-starter-websock
org.springframework.boot	spring-boot-test
org.springframework.boot	spring-boot-test-autoconfig
org.springframework.cloud	spring-cloud-cloudfoundry-c
org.springframework.cloud	spring-cloud-connectors-cor
org.springframework.cloud	spring-cloud-heroku-connect
org.springframework.cloud	spring-cloud-localconfig-co
org.springframework.cloud	spring-cloud-spring-service
org.springframework.data	spring-data-cassandra
org.springframework.data	spring-data-commons
org.springframework.data	spring-data-couchbase
org.springframework.data	spring-data-elasticsearch
org.springframework.data	spring-data-envers
org.springframework.data	spring-data-gemfire
org.springframework.data	spring-data-geode
org.springframework.data	spring-data-jdbc
org.springframework.data	spring-data-jpa
org.springframework.data	spring-data-keyvalue

组 ID	工件 ID
org.springframework.data	spring-data-ldap
org.springframework.data	spring-data-mongodb
org.springframework.data	spring-data-mongodb-cross-s
org.springframework.data	spring-data-neo4j
org.springframework.data	spring-data-redis
org.springframework.data	spring-data-rest-core
org.springframework.data	spring-data-rest-hal-browse
org.springframework.data	spring-data-rest-webmvc
org.springframework.data	spring-data-solr
org.springframework.hateoas	spring-hateoas
org.springframework.integration	spring-integration-amqp
org.springframework.integration	spring-integration-core
org.springframework.integration	spring-integration-event
org.springframework.integration	spring-integration-feed
org.springframework.integration	spring-integration-file
org.springframework.integration	spring-integration-ftp
org.springframework.integration	spring-integration-gemfire
org.springframework.integration	spring-integration-groovy
org.springframework.integration	spring-integration-http
org.springframework.integration	spring-integration-ip
org.springframework.integration	spring-integration-jdbc
org.springframework.integration	spring-integration-jms
org.springframework.integration	spring-integration-jmx
org.springframework.integration	spring-integration-jpa

组 ID	工件 ID
org.springframework.integration	spring-integration-mail
org.springframework.integration	spring-integration-mongodb
org.springframework.integration	spring-integration-mqtt
org.springframework.integration	spring-integration-redis
org.springframework.integration	spring-integration-rmi
org.springframework.integration	spring-integration-scriptin
org.springframework.integration	spring-integration-security
org.springframework.integration	spring-integration-sftp
org.springframework.integration	spring-integration-stomp
org.springframework.integration	spring-integration-stream
org.springframework.integration	spring-integration-syslog
org.springframework.integration	spring-integration-test
org.springframework.integration	spring-integration-test-sup
org.springframework.integration	spring-integration-webflux
org.springframework.integration	spring-integration-websocke
org.springframework.integration	spring-integration-ws
org.springframework.integration	spring-integration-xml
org.springframework.integration	spring-integration-xmpp
org.springframework.integration	spring-integration-zookeepe
org.springframework.kafka	spring-kafka
org.springframework.kafka	spring-kafka-test
org.springframework.ldap	spring-ldap-core
org.springframework.ldap	spring-ldap-core-tiger
org.springframework.ldap	spring-ldap-ldif-batch
org.springframework.ldap	spring-ldap-ldif-core
org.springframework.ldap	spring-ldap-odm

组 ID	工件 ID
org.springframework.ldap	spring-ldap-test
org.springframework.plugin	spring-plugin-core
org.springframework.plugin	spring-plugin-metadata
org.springframework.restdocs	spring-restdocs-asciidoctor
org.springframework.restdocs	spring-restdocs-core
org.springframework.restdocs	spring-restdocs-mockmvc
org.springframework.restdocs	spring-restdocs-restassured
org.springframework.restdocs	spring-restdocs-webtestclie
org.springframework.retry	spring-retry
org.springframework.security	spring-security-acl
org.springframework.security	spring-security-aspects
org.springframework.security	spring-security-cas
org.springframework.security	spring-security-config
org.springframework.security	spring-security-core
org.springframework.security	spring-security-crypto
org.springframework.security	spring-security-data
org.springframework.security	spring-security-ldap
org.springframework.security	spring-security-messaging
org.springframework.security	spring-security-oauth2-clie
org.springframework.security	spring-security-oauth2-core
org.springframework.security	spring-security-oauth2-jose
org.springframework.security	spring-security-oauth2-reso
org.springframework.security	spring-security-openid
org.springframework.security	spring-security-remoting
org.springframework.security	spring-security-taglibs

组 ID	工件 ID
org.springframework.security	spring-security-test
org.springframework.security	spring-security-web
org.springframework.session	spring-session-core
org.springframework.session	spring-session-data-gemfire
org.springframework.session	spring-session-data-geode
org.springframework.session	spring-session-data-mongodb
org.springframework.session	spring-session-data-redis
org.springframework.session	spring-session-hazelcast
org.springframework.session	spring-session-jdbc
org.springframework.ws	spring-ws-core
org.springframework.ws	spring-ws-security
org.springframework.ws	spring-ws-support
org.springframework.ws	spring-ws-test
org.springframework.ws	spring-xml
org.synchronoss.cloud	nio-multipart-parser
org.thymeleaf	thymeleaf
org.thymeleaf	thymeleaf-spring5
org.thymeleaf.extras	thymeleaf-extras-java8time
org.thymeleaf.extras	thymeleaf-extras-springsecu
org.webjars	hal-browser
org.webjars	webjars-locator-core
org.xerial	sqlite-jdbc
org.xmlunit	xmlunit-core
org.xmlunit	xmlunit-legacy

组 ID	工件 ID
org.xmlunit	xmlunit-matchers
org.yaml	snakeyaml
redis.clients	jedis
wsdl4j	wsdl4j
xml-apis	xml-apis

原文

提供更好的翻译建议