Spring Boot Reference Guide译：Spring Boot参考指南

本帮助文档是 觉得烦死 整理--QQ:654638585
声明：
中文文档都是由软件翻译,翻译内容未检查校对,文档内容仅供参考。
您可以任意转发，但请至保留作者&出处(http://bolg.fondme.cn)，请尊重作者劳动成果,谢谢！

Authors译: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

2.0.3.RELEASE

Part I. Spring Boot Documentation译:第一部分 . Spring Boot文档

本节简要介绍Spring Boot参考文档。 它作为文档其余部分的映射。

1. About the Documentation译：1.关于文档

Spring Boot参考指南可用

- HTML
- PDF
- EPUB

最新副本可在 docs.spring.io/spring-boot/docs/current/reference获得 。

2. Getting Help译：2.获得帮助

如果您在Spring Boot遇到问题，我们希望能提供帮助。

- Try the How-to documents. They provide solutions to the most common questions.
- Learn the Spring basics. Spring Boot builds on many other Spring projects. Check the spring.io web-site for a wealth of reference documentation. If you are starting out with Spring, try one of the guides.
- Ask a question. We monitor stackoverflow.com for questions tagged with `spring-boot`.
- Report bugs with Spring Boot at github.com/spring-projects/spring-boot/issues.

所有的Spring Boot都是开源的，包括文档。 如果您发现文档有问题，或者您想改进它们，请致电get involved 。

3. First Steps译：3.第一步

如果您正在开始使用Spring Boot或“Spring”，请从 the following topics开始：

- **From scratch:** 概述 | Requirements | Installation
- **Tutorial:** Part 1 | Part 2
- **Running your example:** Part 1 | Part 2

4. Working with Spring Boot译：4.使用 Spring Boot

准备好真正开始使用Spring Boot？ We have you covered ：

- **Build systems:** Maven | Gradle | Ant | Starters
- **Best practices:** Code Structure | @Configuration | @EnableAutoConfiguration | Beans and Dependency Injection
- **Running your code** IDE | Packaged | Maven | Gradle
- **Packaging your app:** Production jars
- **Spring Boot CLI:** Using the CLI

5. Learning about Spring Boot Features译：5.了解 Spring Boot特性

需要更多关于Spring Boot的核心特性的细节？ The following content is for you ：

- **Core Features:** SpringApplication | External Configuration | Profiles | Logging
- **Web Applications:** MVC | Embedded Containers
- **Working with data:** SQL | NO-SQL
- **Messaging:** 概述 | JMS
- **Testing:** 概述 | Boot Applications | Utils
- **Extending:** Auto-configuration | @Conditions

6. Moving to Production译：6.转向生产

当您准备将Spring Boot应用程序推向生产时，我们有 some tricks ，您可能会喜欢：

- **Management endpoints:** 概述 | Customization
- **Connection options:** HTTP | JMX
- **Monitoring:** Metrics | Auditing | Tracing | Process

7. Advanced Topics译：7.高级主题

最后，我们有更多高级用户的几个主题：

- **Spring Boot Applications Deployment:** Cloud Deployment | OS Service
- **Build tool plugins:** Maven | Gradle
- **Appendix:** Application Properties | Auto-configuration classes | Executable Jars

Part II. Getting Started译：第二部分．入门

如果您正在开始使用Spring Boot或“Spring”，请阅读本节。 它回答了基本的“什么”，“怎么做”以及“为什么”这些问题。 它包括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，并讨论一些核心原则。

8. Introducing Spring Boot译：8.介绍Spring Boot

Spring Boot可以轻松创建可以运行的独立的，生产级的基于Spring的应用程序。 我们对Spring平台和第三方库采取自己的看法，以便您尽可能少用大惊小怪。 大多数Spring Boot应用程序只需要很少的Spring配置。

您可以使用Spring Boot创建可通过使用`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“弹簧脚本”的命令行工具。

我们的主要目标是：

- Provide a radically faster and widely accessible getting-started experience for all Spring development.
- Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

9. System Requirements译：9.系统要求

Spring Boot 2.0.3.RELEASE需要Java 8 or 9和Spring Framework 5.0.7.RELEASE或以上。 为Maven 3.2+和Gradle 4提供了明确的构建支持。

9.1 Servlet Containers译：9.1 Servlet容器

Spring Boot支持以下嵌入式servlet容器：

Name	Servlet Version
Tomcat 8.5	3.1
码头9.4	3.1
Undertow 1.4	3.1

您也可以将Spring Boot应用程序部署到任何与Servlet 3.1+兼容的容器。

10. Installing Spring Boot译：10.安装Spring Boot

Spring Boot可以用于“经典”Java开发工具或作为命令行工具安装。 无论哪种方式，你需要Java SDK v1.8或更高。 在开始之前，您应该使用以下命令检查当前的Java安装：

```
$ java -version
```

如果您对Java开发还不熟悉 ，或者想要试验Spring Boot，则可能需要先尝试Spring Boot CLI（命令行界面）。 否则，请阅读“经典”安装说明。

10.1 Installation Instructions for the Java Developer译：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，请在您的类路径中包含相应的`spring-boot-*.jar`文件。 Spring Boot不需要任何特殊的工具集成，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，因此您可以像运行其他任何Java程序一样运行和调试Spring Boot应用程序。

虽然您 *可以*复制Spring Boot jar，但我们通常建议您使用支持依赖管理的构建工具（如Maven或Gradle）。

10.1.1 Maven Installation译：10.1.1 Maven安装

Spring Boot与Apache Maven 3.2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 。

在许多操作系统上，Maven可以与包管理器一起安装。 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` 。 具有Chocolatey的 Windows用户可以从提升（管理员）提示运行`choco install maven` 。

Spring Boot依赖使用`org.springframework.boot` `groupId` 。 通常，您的Maven POM文件从`spring-boot-starter-parent`项目继承，并将依赖项声明为一个或多个“Starters” 。 Spring Boot还提供了一个可选的Maven plugin来创建可执行的jar。

以下清单显示了一个典型的 `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>2.0.3.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>
```

`spring-boot-starter-parent`是使用Spring Boot的好方法，但它可能并不适合所有的时间。 有时您可能需要从不同的父POM继承，或者您可能不喜欢我们的默认设置。 在这些情况下，请参阅Section 13.2.2, “Using Spring Boot without the Parent POM”了解使用`import`范围的替代解决方案。

10.1.2 Gradle Installation译：10.1.2 Gradle安装

Spring Boot与Gradle 4兼容。如果您尚未安装Gradle，则可以按照 gradle.org的说明操作。

Spring引导依赖可以通过使用`org.springframework.boot` `group`来声明。 通常，您的项目将依赖关系声明为一个或多个“Starters” 。 Spring Boot提供了一个有用的Gradle plugin ，可用于简化依赖声明和创建可执行的jar。

Gradle包装

当您需要构建项目时，Gradle Wrapper提供了一种“获取”Gradle的好方法。 这是一个小脚本和库，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 。

以下示例显示了一个典型的 `build.gradle`文件：

```
plugins {
 id 'org.springframework.boot' version '2.0.3.RELEASE'
 id 'java'
}


jar {
 baseName = 'myproject'
 version =  '0.0.1-SNAPSHOT'
}

repositories {
 jcenter()
}

dependencies {
 compile("org.springframework.boot:spring-boot-starter-web")
 testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

10.2 Installing the Spring Boot CLI译：10.2安装 Spring Boot CLI

Spring Boot CLI（命令行界面）是一个命令行工具，您可以使用它来快速使用Spring进行原型开发。 它可以让你运行Groovy脚本，这意味着你有一个熟悉的类Java语法，没有太多的样板代码。

您不需要使用CLI来使用Spring Boot，但它绝对是让Spring应用程序实现最快的最快捷方式。

10.2.1 Manual Installation译：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-2.0.3.RELEASE-bin.zip
- spring-boot-cli-2.0.3.RELEASE-bin.tar.gz

切边 snapshot distributions也可用。

下载后，请按照解压缩归档中的INSTALL.txt说明操作。 总之，在`.zip`文件的`bin/`目录中有`spring`脚本（ `spring.bat`于Windows的`.zip` 。 或者，您可以使用`java -jar`和`.jar`文件（该脚本可帮助您确保类路径设置正确）。

10.2.2 Installation with SDKMAN! 译：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并通过使用以下命令安装Spring Boot：

```
$ sdk install springboot
$ spring --version
Spring Boot v2.0.3.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-2.0.3.RELEASE-bin/spring-2.0.3.RELEASE/
$ sdk default springboot dev
$ spring --version
Spring CLI v2.0.3.RELEASE
```

前面的说明安装本地实例`spring`称为`dev`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
* 2.0.3.RELEASE

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

10.2.3 OSX Homebrew Installation 译：10.2.3 OSX自制软件安装

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

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

Homebrew安装`spring`至`/usr/local/bin` 。

如果您没有看到该公式，那么您的brew的安装可能会过时。 在这种情况下，请运行`brew update`试。

10.2.4 MacPorts Installation 译：10.2.4 MacPorts安装

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

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

10.2.5 Command-line Completion 译：10.2.5命令行完成

Spring Boot CLI包含为BASH和zsh外壳提供命令完成的脚本。 您可以在任何shell `source`脚本（也称为`spring` ）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` ，当新shell启动时，该目录中的所有脚本都会执行。 例如，如果您使用SDKMAN！安装了手动运行脚本，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
```

如果您使用Homebrew或MacPorts安装Spring Boot CLI，则命令行完成脚本会自动在您的shell中注册。

10.2.6 Windows Scoop Installation 译：10.2.6 Windows Scoop安装

如果您在Windows上并使用 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`至`~/scoop/apps/springboot/current/bin` 。

如果您没有看到应用程序清单，那么您的安装瓢可能会过时。 在这种情况下，请运行`scoop update`试。

10.2.7 Quick-start Spring CLI Example译：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
```

由于下载依赖项，应用程序的第一次运行速度很慢。 后续运行速度更快。

在您最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 Upgrading from an Earlier Version of Spring Boot译：10.3从较早版本的Spring Boot升级

如果您正在从早期版本的Spring Boot进行升级，请检查提供详细升级说明的“migration guide” on the project wiki 。 还请检查“release notes”以获取每个版本的“新的和值得注意的”功能列表。

要升级现有的CLI安装，请使用相应的软件包管理器命令（例如， `brew upgrade` ），或者如果您手动安装了CLI，请按照 standard instructions ，记住更新您的 `PATH` 环境变量以删除任何较旧的引用。

11. Developing Your First Spring Boot Application译：11.开发你的第一个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，该应用程序强调了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。

该spring.io网站包含许多“guides”使用Spring的引导。 如果您需要解决特定问题，请先在那里查看。
您可以通过转到start.spring.io并从依赖关系搜索器中选择“Web”起始器来缩短以下步骤。 这样做会产生一个新的项目结构，以便您可以start coding right away 。 查看Spring Initializr documentation了解更多详情。

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

```
$ 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 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T16:41:47+00:00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_102, vendor: Oracle Corporation
```

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

11.1 Creating the POM译：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。 打开您最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>2.0.3.RELEASE</version>
 </parent>

 <!-- Additional lines to be added here... -->

</project>
```

前面的列表应该给你一个工作版本。 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）。

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）。 为了简单起见，我们在本例中继续使用纯文本编辑器。

11.2 Adding Classpath Dependencies译：11.2添加类路径依赖关系

Spring Boot提供了许多“启动器”，可让您将jar添加到类路径中。 我们的示例应用程序已经使用`spring-boot-starter-parent`在`parent`的POM的部分。 `spring-boot-starter-parent`是一个提供有用Maven默认设置的特别启动器。 它还提供了一个`dependency-management`部分，以便您可以省略“ `version`依赖项的`version`标记。

其他“启动器”提供了在开发特定类型的应用程序时可能需要的依赖关系。 由于我们正在开发一个Web应用程序，因此我们添加了一个`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我们目前的功能：

```
$ mvn dependency:tree

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

`mvn dependency:tree`命令打印项目相关性的树形表示。 你可以看到`spring-boot-starter-parent`本身不提供依赖关系。 要添加必要的依赖关系，请编辑`pom.xml`并在`parent`部分正下方添加`spring-boot-starter-web`依赖`parent`：

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

如果您再次运行`mvn dependency:tree`，则会看到现在有许多其他依赖项，包括Tomcat Web服务器和Spring Boot本身。

11.3 Writing the Code译：11.3编写代码

为了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`源`src/main/java`，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);
 }

}
```

虽然这里没有太多的代码，但还是有很多。 我们将在接下来的几节中介绍一些重要的部分。

11.3.1 The @RestController and @RequestMapping Annotations译：11.3.1 @RestController和 @RequestMapping注解

我们的`Example`课程的第一个注释是`@RestController`。 这被称为*刻板*印记。 它为阅读代码的人提供了线索，对于Spring来说，这个类扮演着特定的角色。 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`注释提供了“路由”信息。 它告诉Spring，任何具有`/`路径的HTTP请求都应映射到`home`方法。 `@RestController`注释告诉Spring将结果字符串直接呈现给调用者。

`@RestController`和`@RequestMapping`注释是Spring MVC注释。（它们并不特定于Spring Boot。）有关更多详细信息，请参见Spring参考手册中的MVC section。

11.3.2 The @EnableAutoConfiguration Annotation译：11.3.2 @EnableAutoConfiguration注释

第二级的注释是`@EnableAutoConfiguration`。 这个注释告诉Spring Boot基于你添加的jar依赖关系来“配置”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。

> **启动器和自动配置**
>
> 自动配置旨在与“启动器”配合使用，但这两个概念并不直接相关。 您可以自由选择初学者之外的jar依赖项。 Spring Boot仍然尽力自动配置您的应用程序。

11.3.3 The “main” Method译：11.3.3“main”方法

我们应用程序的最后一部分是`main`方法。 这只是一个遵循Java约定的应用程序入口点的标准方法。 我们的主要方法通过调用`run`委托Spring Boot的`SpringApplication`类。 `SpringApplication`启动我们的应用程序，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，以告知`SpringApplication`哪些是主要的Spring组件。 `args`数组也被传递以显示任何命令行参数。

11.4 Running the Example译：11.4运行示例

在这一点上，你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`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 ::  (v2.0.3.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

如果您打开一个网页浏览器到`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 Creating an Executable Jar译：11.5创建可执行jar

我们通过创建一个完全独立的可执行jar文件来完成我们的示例，该文件可以在生产环境中运行。 可执行jar（有时称为“fat jars”）是包含您的编译类以及您的代码需要运行的所有jar依赖项的归档文件。

> **可执行的jar和Java**
>
> Java没有提供加载嵌套jar文件的标准方式（本身包含在jar中的jar文件）。 如果您想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“瓶子”罐子。 一个超级jar将所有应用程序依赖项中的所有类打包到一个单独的存档中。 这种方法的问题是很难看到你的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（但是具有不同的内容），则它也可能是有问题的。
>
> Spring Boot需要 different approach，并且可以让您直接嵌入罐子。

要创建一个可执行的jar，我们需要将`spring-boot-maven-plugin`添加到我们的`pom.xml`。 为此，请在`dependencies`部分下方插入以下几行：

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

`spring-boot-starter-parent` POM包含`<executions>`配置以绑定`repackage`目标。 如果您不使用父POM，则需要自行声明此配置。 详情请参阅plugin documentation。

保存您的`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:2.0.3.RELEASE:repackage (default) @ myproject ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

如果您查看`target`目录，则应该看到`myproject-0.0.1-SNAPSHOT.jar`。 该文件的大小应该在10 MB左右。 如果你想偷看，你可以使用`jar tvf`，如下所示：

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

您还应该在`target`目录中看到名为`myproject-0.0.1-SNAPSHOT.jar.original`小得多的文件。 这是Maven在被Spring Boot重新包装之前创建的原始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 ::  (v2.0.3.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. What to Read Next译：12.下一步阅读什么

希望本节提供了一些Spring Boot的基础知识，并帮助您编写自己的应用程序。 如果您是面向任务的开发人员，则可能需要跳至spring.io，并查看一些解决特定“我该如何使用Spring”问题的getting started指南。 我们也有Spring Boot特定的“ How-to”参考文档。

Spring Boot repository也有一个bunch of samples你可以运行。 样本与代码的其余部分无关（也就是说，您不需要构建其余代码以运行或使用样本）。

否则，下一个逻辑步骤是阅读*Part III, “Using Spring Boot”*。 如果你真的不耐烦，你也可以跳到前面阅读*Spring Boot features* 。

Part III. Using Spring Boot译：第三部分．使用 Spring Boot

本节将详细介绍如何使用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的最佳实践。 尽管Spring Boot没有特别的特殊之处（它只是您可以使用的另一个库），但有一些建议，如果遵循这些建议，您的开发过程会更容易一些。

如果您 *刚*开始使用Spring Boot，那么在 *深入*阅读本节之前，您应该阅读 *Getting Started*指南。

13. Build Systems译：13.构建系统

强烈建议您选择一个支持*dependency management*的构建系统，并且可以使用发布到“Maven Central”存储库的构件。 我们建议您选择Maven或Gradle。 Spring Boot可以与其他构建系统（例如Ant）一起工作，但它们并没有得到特别好的支持。

13.1 Dependency Management译：13.1依赖管理

Spring Boot的每个发行版都提供了它支持的依赖关系的策略列表。 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为您管理这些版本。 当您升级Spring Boot本身时，这些依赖关系也会以一致的方式升级。

如果你需要的话，你仍然可以指定一个版本并覆盖Spring Boot的建议。

策划列表包含您可以在Spring Boot中使用的所有Spring模块以及第三方库的精炼列表。 该列表可用作Bills of Materials (`spring-boot-dependencies`)的标准，可用于Maven和Gradle 。

Spring Boot的每个版本都与Spring Framework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译：13.2 Maven

Maven用户可以从`spring-boot-starter-parent`项目继承以获得合理的默认值。 父项目提供以下功能：

- Java 1.8 as the default compiler level.
- UTF-8 source encoding.
- A Dependency Management section, inherited from the spring-boot-dependencies pom, that manages the versions of common dependencies. This dependency management lets you omit <version> tags for those dependencies when used in your own pom.
- Sensible resource filtering.
- Sensible plugin configuration (exec plugin, Git commit ID, and shade).
- Sensible resource filtering for `application.properties` and `application.yml` including profile-specific files (for example, `application-dev.properties` and `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属性来覆盖该属性。）

13.2.1 Inheriting the Starter Parent译：13.2.1继承初始父项

要将项目配置为从 `spring-boot-starter-parent`继承， `parent`按如下所示设置 `parent` ：

```
<!-- Inherit defaults from Spring Boot -->
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>2.0.3.RELEASE</version>
</parent>
```

您应该只需在此依赖项上指定Spring Boot版本号。 如果您导入额外的启动器，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各个依赖项。 例如，要升级到另一个Spring Data发行版，请将以下内容添加到`pom.xml` ：

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

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

13.2.2 Using Spring Boot without the Parent POM译：13.2.2在没有父POM的情况下使用 Spring Boot

不是每个人都喜欢从`spring-boot-starter-parent` POM继承。 您可能拥有自己的公司标准父项，或者您可能更愿意明确声明所有Maven配置。

如果您不想使用 `spring-boot-starter-parent` ，那么通过使用 `scope=import` 依赖关系，您仍然可以保持依赖关系管理的好处（但不是插件管理），如下所示：

```
<dependencyManagement>
  <dependencies>
  <dependency>
   <!-- Import dependency management from Spring Boot -->
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-dependencies</artifactId>
   <version>2.0.3.RELEASE</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

如上所述，上述示例设置不会让您使用属性重写个别依赖关系。 为了达到同样的效果，你需要添加的条目 `dependencyManagement` 的**前**项目的 `spring-boot-dependencies` 条目。 例如，要升级到另一个Spring Data发行版，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>2.0.3.RELEASE</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

在前面的例子中，我们指定了一个 *BOM* ，但是任何依赖类型都可以用相同的方式覆盖。

13.2.3 Using the Spring Boot Maven Plugin 译：13.2.3使用 Spring Boot Maven插件

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

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

如果您使用Spring Boot启动器父POM，则只需添加插件。 除非您想更改父级中定义的设置，否则无需对其进行配置。

13.3 Gradle 译：13.3 Gradle

要了解如何使用Gradle使用Spring Boot，请参阅Spring Boot的Gradle插件文档：

- Reference (HTML and PDF)
- API

13.4 Ant 译：13.4蚂蚁

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

为了声明依赖关系，典型的 `ivy.xml` 文件看起来如下例所示：

```
<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>
```

一个典型的 `build.xml` 看起来像下面的例子：

```
<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="2.0.3.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>
```

如果您不想使用 `spring-boot-antlib` 模块，请参阅 *Section 87.9, "Build an Executable Archive from Ant without Using `spring-boot-antlib`"* How to"。

13.5 Starters 译：13.5起动器

启动器是一套方便的依赖描述符，可以包含在应用程序中。 您可以获得所需的所有Spring及相关技术的一站式商店，而无需查看示例代码并复制粘贴依赖描述符。 例如，如果您想开始使用Spring和JPA进行数据库访问，请将 `spring-boot-starter-data-jpa` 依赖项包含在您的项目中。

初学者包含很多依赖项，您需要快速启动并快速运行项目，并且需要一组支持的可传递依赖关系。

什么是名字

所有**官方**首发者都遵循类似的命名模式; `spring-boot-starter-*` ，其中 `*` 是特定类型的应用程序。 这种命名结构旨在帮助您在需要查找启动器时。 许多IDE中的Maven集成允许您按名称搜索依赖项。 例如，通过安装适当的Eclipse或STS插件，您可以在POM编辑器中按 `ctrl-space` ，然后键入“spring-boot-starter”以获取完整列表。

正如“ Creating Your Own Starter”部分所解释的那样，第三方首发不应该以 `spring-boot` ，因为它是为官方Spring Boot工件保留的。 相反，第三方初学者通常以项目名称开头。 例如，名为 `thirdpartyproject` 的第三方初始项目通常会被命名为 `thirdpartyproject-spring-boot-starter` 。

以下应用程序启动器由Spring Boot根据 `org.springframework.boot` 组提供：

表13.1。 Spring Boot应用程序启动器

Name	描述	Pom
`spring-boot-starter`	核心入门者，包括自动配置支持，日志记录和YAML	Pom
`spring-boot-starter-activemq`	使用Apache ActiveMQ启动JMS消息传递	Pom
`spring-boot-starter-amqp`	使用Spring AMQP和Rabbit MQ的入门者	Pom
`spring-boot-starter-aop`	使用Spring AOP和AspectJ进行面向方面编程的入门者	Pom
`spring-boot-starter-artemis`	使用Apache Artemis开始JMS消息传递	Pom
`spring-boot-starter-batch`	使用Spring Batch的入门者	Pom
`spring-boot-starter-cache`	Starter使用Spring Framework的缓存支持	Pom
`spring-boot-starter-cloud-connectors`	Starter使用Spring Cloud Connectors，可简化Cloud Foundry和Heroku等云平台中的服务连接	Pom
`spring-boot-starter-data-cassandra`	入门使用Cassandra分布式数据库和Spring Data Cassandra	Pom
`spring-boot-starter-data-cassandra-reactive`	使用Cassandra分布式数据库和Spring Data Cassandra Reactive的初学者	Pom
`spring-boot-starter-data-couchbase`	使用Couchbase面向文档的数据库和Spring Data Couchbase的初学者	Pom
`spring-boot-starter-data-couchbase-reactive`	初级用于使用Couchbase面向文档的数据库和Spring Data Couchbase Reactive	Pom
`spring-boot-starter-data-elasticsearch`	使用Elasticsearch搜索和分析引擎和Spring Data Elasticsearch的入门者	Pom
`spring-boot-starter-data-jpa`	使用Spring数据JPA与Hibernate的入门者	Pom
`spring-boot-starter-data-ldap`	使用Spring Data LDAP的入门者	Pom
`spring-boot-starter-data-mongodb`	入门使用MongoDB面向文档的数据库和Spring Data MongoDB	Pom
`spring-boot-starter-data-mongodb-reactive`	入门使用MongoDB面向文档的数据库和Spring Data MongoDB Reactive	Pom

Name	描述	Pom
`spring-boot-starter-data-neo4j`	初学者使用Neo4j图形数据库和Spring Data Neo4j	Pom
`spring-boot-starter-data-redis`	使用Spring Data Redis和Lettuce客户端使用Redis键值数据存储的入门者	Pom
`spring-boot-starter-data-redis-reactive`	初学者使用Redis键值数据存储以及Spring Data Redis反应器和Lettuce客户端	Pom
`spring-boot-starter-data-rest`	Starter使用Spring Data REST通过REST公开Spring Data存储库	Pom
`spring-boot-starter-data-solr`	启动Spring Data Solr使用Apache Solr搜索平台	Pom
`spring-boot-starter-freemarker`	使用FreeMarker视图构建MVC Web应用程序的入门者	Pom
`spring-boot-starter-groovy-templates`	使用Groovy模板视图构建MVC Web应用程序的入门者	Pom
`spring-boot-starter-hateoas`	使用Spring MVC和Spring HATEOAS构建基于超媒体的RESTful Web应用程序的入门者	Pom
`spring-boot-starter-integration`	使用Spring Integration的入门者	Pom
`spring-boot-starter-jdbc`	使用JDBC和HikariCP连接池的入门者	Pom
`spring-boot-starter-jersey`	使用JAX-RS和Jersey构建RESTful Web应用程序的入门者。 `spring-boot-starter-web`的替代品	Pom
`spring-boot-starter-jooq`	使用jOOQ访问SQL数据库的入门者。 替代`spring-boot-starter-data-jpa`或`spring-boot-starter-jdbc`	Pom
`spring-boot-starter-json`	用于阅读和编写json的初学者	Pom
`spring-boot-starter-jta-atomikos`	使用Atomikos启动JTA交易	Pom
`spring-boot-starter-jta-bitronix`	使用Bitronix启动JTA交易	Pom
`spring-boot-starter-jta-narayana`	使用Narayana进行JTA交易的首发	Pom
`spring-boot-starter-mail`	Starter使用Java Mail和Spring Framework的电子邮件发送支持	Pom
`spring-boot-starter-mustache`	使用Mustache视图构建Web应用程序的入门者	Pom
`spring-boot-starter-quartz`	使用Quartz调度器的入门者	Pom
`spring-boot-starter-security`	Starter使用Spring Security	Pom
`spring-boot-starter-test`	Starter用于测试包含JUnit，Hamcrest和Mockito等库的Spring Boot应用程序	Pom
`spring-boot-starter-thymeleaf`	使用Thymeleaf视图构建MVC Web应用程序的入门者	Pom
`spring-boot-starter-validation`	通过Hibernate Validator使用Java Bean验证的入门者	Pom
`spring-boot-starter-web`	使用Spring MVC构建Web的初学者，包括RESTful应用程序。 使用Tomcat作为默认的嵌入容器	Pom
`spring-boot-starter-web-services`	使用Spring Web Services的入门者	Pom
`spring-boot-starter-webflux`	使用Spring Framework的Reactive Web支持构建WebFlux应用程序的入门者	Pom
`spring-boot-starter-websocket`	使用Spring Framework的WebSocket支持构建WebSocket应用程序的入门者	Pom

除应用程序启动器外，还可以使用以下启动器来添加 *production ready*功能：

表13.2。 Spring Boot生产启动器

Name	描述	Pom
`spring-boot-starter-actuator`	Starter使用Spring Boot的Actuator，可提供生产就绪功能，帮助您监控和管理您的应用程序	Pom

最后，Spring Boot还包含以下启动器，如果您想要排除或交换特定技术方面，可以使用以下启动器：

表13.3。 Spring Boot技术首发

Name	描述	Pom
`spring-boot-starter-jetty`	将Jetty用作嵌入式servlet容器的入门者。 替代`spring-boot-starter-tomcat`	Pom
`spring-boot-starter-log4j2`	使用Log4j2进行日志记录的入门者。 `spring-boot-starter-logging`的替代品	Pom
`spring-boot-starter-logging`	启动器使用Logback进行日志记录。 默认日志启动器	Pom
`spring-boot-starter-reactor-netty`	入门使用Reactor Netty作为嵌入式反应式HTTP服务器。	Pom
`spring-boot-starter-tomcat`	使用Tomcat作为嵌入式servlet容器的入门。 默认的Servlet容器启动器由`spring-boot-starter-web`使用	Pom
`spring-boot-starter-undertow`	使用Undertow作为嵌入式servlet容器的入门者。 `spring-boot-starter-tomcat`的替代品	Pom

有关其他社区的列表贡献首先，看 README file中`spring-boot-starters`模块在GitHub上。

14. Structuring Your Code译：14构建你的代码

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

14.1 Using the “default” Package译：14.1使用“默认”软件包

当一个类不包含`package`声明时，它被认为是在“缺省包”中。 通常不鼓励使用“缺省包”，应该避免使用“缺省包”。 这可能会导致使用了Spring启动应用程序的特殊问题`@ComponentScan`， `@EntityScan`，或`@SpringBootApplication`注解，因为从每一个罐子每一个类被读取。

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

14.2 Locating the Main Application Class译：14.2查找主要应用程序类

我们通常建议您将主应用程序类定位在其他类上方的根包中。 `@SpringBootApplication` annotation通常放在你的主类上，它隐式地为某些项目定义了一个基本的“搜索包”。 例如，如果您正在编写JPA应用程序，则`@SpringBootApplication`注释的类的包将用于搜索`@Entity`项。 使用根包也允许组件扫描仅适用于您的项目。

如果您不想使用`@SpringBootApplication`，则`@EnableAutoConfiguration`和`@ComponentScan`注释将定义该行为，以便您也可以使用该注释。

下面的清单显示了一个典型的布局：

```
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. Configuration Classes译：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源是一个`@Configuration`类。 通常，定义`main`方法的类作为主要的`@Configuration`是一个很好的候选`@Configuration`。

许多使用XML配置的互联网上发布了Spring配置示例。 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 Importing Additional Configuration Classes译：15.1导入其他配置类

你不需要把所有的`@Configuration`放到一个班级中。 `@Import`注释可用于导入其他配置类。 或者，您可以使用`@ComponentScan`自动获取所有Spring组件，包括`@Configuration`类。

15.2 Importing XML Configuration译：15.2导入XML配置

如果您绝对必须使用基于XML的配置，我们建议您仍然从`@Configuration`类开始。 然后您可以使用`@ImportResource`注释来加载XML配置文件。

16. Auto-configuration译：16自动配置

Spring Boot自动配置会尝试根据您添加的jar依赖关系自动配置您的Spring应用程序。 例如，如果`HSQLDB`位于您的类路径中，并且您尚未手动配置任何数据库连接Bean，则Spring Boot会自动配置内存数据库。

您需要通过将`@EnableAutoConfiguration`或`@SpringBootApplication`注释添加到其中一个`@Configuration`类中来选择加入自动配置。

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您的主要`@Configuration`类。

16.1 Gradually Replacing Auto-configuration译：16.1逐渐更换自动配置

自动配置是非侵入式的。 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，如果您添加自己的`DataSource` bean， `DataSource`默认的嵌

入式数据库支持。

如果您需要了解当前正在应用的自动配置以及为什么使用`--debug`开关启动应用程序。 这样做可以为选定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 Disabling Specific Auto-configuration Classes 译: 16.2禁用特定的自动配置类

如果您发现不需要的特定自动配置类正在应用，则可以使用`@EnableAutoConfiguration`的exclude属性来禁用它们，如下例所示：

```
import org.springframework.boot.autoconfigure.*;
import org.springframework.boot.autoconfigure.jdbc.*;
import org.springframework.context.annotation.*;

@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

如果类不在类路径中，则可以使用注释的`excludeName`属性，并指定完全限定名称。 最后，您还可以通过使用`spring.autoconfigure.exclude`属性来控制要排除的自动配置类的列表。

您可以在注释级别和通过使用属性来定义排除。

17. Spring Beans and Dependency Injection 译: 17.春豆和依赖注入

您可以自由使用任何标准的Spring框架技术来定义您的bean及其注入的依赖关系。 为了简单起见，我们经常发现使用`@ComponentScan`（找到你的bean）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（在根包中查找应用程序类），则可以添加`@ComponentScan`而不带任何参数。 您的所有应用程序组件（的`@Component`， `@Service`，`@Repository`， `@Controller`等）自动注册为春豆。

下面的示例示出了 `@Service`豆使用构造子注入，以获得所需的 `RiskAssessor`豆：

```
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;
 }

 // ...

}
```

如果一个bean有一个构造函数，则可以省略 `@Autowired`，如下例所示：

```
@Service
public class DatabaseAccountService implements AccountService {

 private final RiskAssessor riskAssessor;

 public DatabaseAccountService(RiskAssessor riskAssessor) {
  this.riskAssessor = riskAssessor;
 }

 // ...

}
```

注意如何使用构造函数注入让 `riskAssessor`字段被标记为 `final`，表明它不能被随后改变。

18. Using the @SpringBootApplication Annotation 译: 18.使用 @SpringBootApplication注释

许多Spring Boot开发者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`: enable Spring Boot's auto-configuration mechanism
- `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)
- `@Configuration`: allow to register extra beans in the context or import additional configuration classes

所述 `@SpringBootApplication`注释是相当于使用 `@Configuration`， `@EnableAutoConfiguration`和 `@ComponentScan`与他们的默认属性，如显示在下面的例子：

```
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);
 }

}
```

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

这些功能都不是强制性的，您可以选择使用它启用的任何功能替换此单个注释。 例如，您可能不想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);
 }

}
```

在本例中， `Application`与其他Spring Boot应用程序相似，只是`@Component` -annotated类不会自动检测到，并且用户定义的bean明确导入（请参阅`@Import`）。

19. Running Your Application 译：19.运行你的应用程序

将应用程序打包为jar并使用嵌入式HTTP服务器的最大优点之一是，您可以像运行其他应用程序一样运行应用程序。 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。

本节仅涵盖基于jar的包装。 如果您选择将应用程序打包为war文件，则应参考您的服务器和IDE文档。

19.1 Running from an IDE 译：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，您首先需要导入您的项目。 导入步骤因您的IDE和构建系统而异。 大多数IDE可以直接导入Maven项目。 例如，Eclipse用户可以选择`Import…` â†’ `Existing Maven Projects`从`File`菜单。

如果您无法直接将您的项目导入IDE，则可以使用构建插件生成IDE元数据。 Maven包含Eclipse和IDEA的插件。 Gradle为various IDEs提供插件。

如果您意外运行了两次Web应用程序，则会看到一个“Port already in use”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 Running as a Packaged Application 译：19.2作为打包应用程序运行

如果使用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
```

也可以在启用远程调试支持的情况下运行打包的应用程序。 这样做可以让您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 Using the Maven Plugin 译：19.3使用Maven插件

Spring Boot Maven插件包含一个可用于快速编译和运行应用程序的`run`目标。 应用程序以分解形式运行，就像它们在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，如下例所示：

```
$ export MAVEN_OPTS=-Xmx1024m
```

19.4 Using the Gradle Plugin 译：19.4使用Gradle插件

Spring Boot Gradle插件还包含一个可用于以分解形式运行应用程序的`bootRun`任务。 每当您应用`org.springframework.boot`和`java`插件时，都会添加`bootRun`任

务，并显示在以下示例中：

```
$ gradle bootRun
```

您可能还想使用`JAVA_OPTS`操作系统环境变量，如下例所示：

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

19.5 Hot Swapping译：19.5热插拔

由于Spring Boot应用程序只是普通的Java应用程序，所以JVM热插拔应该可以开箱即用。 JVM热插拔在可以替换的字节码方面有所限制。 要获得更完整的解决方案，可以使用JRebel 。

`spring-boot-devtools`模块还包含对快速应用程序重新启动的支持。 有关详细信息，请参阅本章后面的Chapter 20, *Developer Tools*部分和Hot swapping “How-to” 。

20. Developer Tools译：20.开发工具

Spring Boot包含一组额外的工具，可以使应用程序开发体验更愉快。 `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Maven和Gradle的以下列表所示：

Maven的。

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

摇篮。

```
dependencies {
 compile("org.springframework.boot:spring-boot-devtools")
}
```

运行完整打包的应用程序时，开发人员工具会自动禁用 如果你的应用程序是从`java -jar`启动的，或者它是从一个特殊的类加载器启动的，那么它就被认为是一个“生产应用程序”。 在Maven中将依赖项标记为可选项或在Gradle中使用`compileOnly`是一种防止devtools被传递应用到其他使用项目的模块的最佳实践。

重新打包的档案在默认情况下不包含devtools。 如果您想使用certain remote devtools feature ，则需要禁用`excludeDevtools`构建属性才能包含它。 该属性支持Maven和Gradle插件。

20.1 Property Defaults译：20.1属性默认值

Spring Boot支持的一些库使用缓存来提高性能。 例如， template engines缓存已编译的模板以避免重复解析模板文件。 另外，Spring MVC可以在服务静态资源时将HTTP缓存头添加到响应中。

虽然缓存在生产中非常有用，但它在开发过程中会起到反作用，使您无法看到您在应用程序中所做的更改。 因此，spring-boot-devtools默认禁用缓存选项。

缓存选项通常由您的`application.properties`文件中的设置进行配置。 例如，Thymeleaf提供`spring.thymeleaf.cache`财产。 `spring-boot-devtools`模块`spring-boot-devtools`手动设置这些属性， `spring-boot-devtools`自动应用合理的开发时配置。

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

20.2 Automatic Restart译：20.2自动重启

每当类路径上的文件发生更改时，使用`spring-boot-devtools`应用程序`spring-boot-devtools`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，监视指向文件夹的类路径上的任何条目以进行更改。 请注意，某些资源（如静态资产和视图模板）为do not need to restart the application 。

触发重启

由于DevTools监控类路径资源，触发重启的唯一方法是更新类路径。 导致类路径更新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。

只要启用分叉，您也可以使用受支持的构建插件（Maven和Gradle）启动您的应用程序，因为DevTools需要隔离的应用程序类加载器才能正常运行。 默认情况下，当他们在类路径中检测到DevTools时，Gradle和Maven会这样做。

与LiveReload一起使用时，自动重启的效果非常好。 详情请参阅See the LiveReload section 。 如果您使用JRebel，则会禁用自动重新启动，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。

DevTools依靠应用程序上下文的关闭挂钩在重新启动期间关闭它。 如果您已禁用关机挂钩（`SpringApplication.setRegisterShutdownHook(false)`），则它不能正常工作。

当决定是否在类路径中的条目应该触发重新启动时，它的变化，DevTools自动忽略命名的项目`spring-boot`， `spring-boot-devtools`， `spring-boot-autoconfigure`， `spring-boot-actuator`，并`spring-boot-starter`。

DevTools需要自定义`ResourceLoader`使用的`ApplicationContext`。 如果你的应用程序已经提供了一个，它将被包装。 在直接覆盖`getResource`的方法`ApplicationContext`不支持。

重新启动vs重新加载

Spring Boot提供的重启技术通过使用两个类加载器来工作。 不改变的类（例如来自第三方jar的类）被加载到*基*类加载器中。 您正在开发的类将加载到*重启*类加载器中。 当应用程序重新启动时， *重新启动*类加载器将被丢弃并创建一个新类。 这种方法意味着应用程序重启通常比“冷启动”快得多，因为*基*类加载器已经可用并且已经被填充了。

如果您发现重启对于您的应用程序来说不够快或者遇到类加载问题，则可以考虑从ZeroTurnaround中重新加载技术，例如JRebel 。 这些工作通过在加载类时重写类，使它们更易于重新加载。

20.2.1 Logging changes in condition evaluation译: 20.2.1记录条件评估中的更改

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

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 Excluding Resources译: 20.2.2排除资源

某些资源不一定需要在更改时触发重新启动。 例如，可以就地编辑Thymeleaf模板。 默认情况下，在改变资源`/META-INF/maven`， `/META-INF/resources`， `/resources`， `/static`， `/public`，或`/templates`不会触发重启但并触发live reload 。 如果您想自定义这些排除项，可以使用`spring.devtools.restart.exclude`属性。 例如，要仅排除`/static`和`/public`您可以设置以下属性：

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

如果您想保留这些默认设置并 *添加*其他排除项，请改为使用`spring.devtools.restart.additional-exclude`属性。

20.2.3 Watching Additional Paths译: 20.2.3观察其他路径

您可能希望在更改不在类路径中的文件时重新启动或重新加载应用程序。 为此，请使用`spring.devtools.restart.additional-paths`属性来配置其他路径以监视更改。 您可以使用`spring.devtools.restart.exclude`属性described earlier来控制其他路径下的更改是否触发完全重新启动或live reload 。

20.2.4 Disabling Restart译: 20.2.4禁用重新启动

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

如果您需要 *完全*禁用重启的支持（例如，因为它不与特定库的工作），你需要设置`spring.devtools.restart.enabled` `System`属性为`false`调用之前`SpringApplication.run(… )`，如下面的例子：

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

20.2.5 Using a Trigger File译: 20.2.5使用触发文件

如果您使用持续编译更改文件的IDE，则可能只希望在特定时间触发重新启动。 为此，您可以使用“触发文件”，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。 只更改文件会触发检查，只有在Devtools检测到必须执行某些操作时才会重新启动。 触发文件可以手动更新或使用IDE插件更新。

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

您可能需要将`spring.devtools.restart.trigger-file`设置为 global setting ，以便所有项目的行为方式都相同。

20.2.6 Customizing the Restart Classloader译: 20.2.6自定义重启类加载器

如前面Restart vs Reload部分所述，重启功能通过使用两个类加载器来实现。 对于大多数应用程序，这种方法运作良好 但是，它有时会导致类加载问题。

默认情况下，IDE中任何打开的项目都会加载“precart”类加载器，并且任何常规的`.jar`文件都将加载“base”类加载器。 如果您使用多模块项目，并且不是每个模块都导入到IDE中，则可能需要自定义。 为此，您可以创建一个`META-INF/spring-devtools.properties`文件。

`spring-devtools.properties`文件可以包含以`restart.exclude`和`restart.include`为前缀的属性。 `include`元素是应该拉入“重新启动”类加载器的项目，`exclude`元素是应该下推到“基本”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
```

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。

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

20.2.7 Known Limitations 译：20.2.7已知限制

对于使用标准`ObjectInputStream`进行反序列化的对象，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，有些第三方库反序列化而没有考虑上下文类加载器。 如果您发现这样的问题，您需要向原作者请求修复。

20.3 LiveReload 译：20.3 LiveReload

`spring-boot-devtools`模块包含一个嵌入式LiveReload服务器，可用于在更改资源时触发浏览器刷新。 LiveReload浏览器扩展程序免费提供来自livereload.com的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`。

一次只能运行一个LiveReload服务器。 在开始您的应用程序之前，请确保没有其他LiveReload服务器正在运行。 如果您从IDE启动多个应用程序，则只有第一个应用程序支持LiveReload。

20.4 Global Settings 译：20.4全局设置

您可以通过向`$HOME`文件夹添加一个名为`.spring-boot-devtools.properties`的文件来配置全局devtools设置（请注意文件名以“。”开头）。 添加到此文件的任何属性*都*适用于使用devtools的计算机上的*所有*Spring Boot应用程序。 例如，要将重新启动配置为始终使用trigger file，您需要添加以下属性：

~/ .spring引导-devtools.properties。

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

20.5 Remote Applications 译：20.5远程应用程序

Spring Boot开发人员工具不限于本地开发。 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归档文件中包含`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
```

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

远程devtools支持分两部分提供：接受连接的服务器端端点以及您在IDE中运行的客户端应用程序。 `spring.devtools.remote.secret`属性设置后，服务器组件将自动启用。 客户端组件必须手动启动。

20.5.1 Running the Remote Client Application 译：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您的IDE中运行。 您需要使用与您连接的远程项目相同的类路径运行`org.springframework.boot.devtools.RemoteSpringApplication`。 该应用程序的单个必需参数是它所连接的远程URL。

例如，如果您使用的是Eclipse或STS，并且您已将一个名为`my-app`的项目部署到Cloud Foundry，则可以执行以下操作：

- Select `Run Configurations…` from the `Run` menu.
- Create a new `Java Application` “launch configuration”.
- Browse for the `my-app` project.
- Use `org.springframework.boot.devtools.RemoteSpringApplication` as the main class.
- Add `https://myapp.cfapps.io` to the `Program arguments` (or whatever your remote URL is).

正在运行的远程客户端可能类似于以下列表：

```
  .   ____          _                                              __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___              _      \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_ ___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/ -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\__/\__\___|/ / / /
 =========|_|==============|___/=================================/_/_/_/
 :: Spring Boot Remote :: 2.0.3.RELEASE

2015-06-10 18:25:06.632  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringApplication   : Starting RemoteSpringApplication on pwmbp with P
2015-06-10 18:25:06.671  INFO 14938 --- [           main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotatic
2015-06-10 18:25:07.043  WARN 14938 --- [           main] o.s.b.d.r.c.RemoteClientConfiguration    : The connection to http://localhost:8080 is insec
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
```

由于远程客户端使用与真实应用程序相同的类路径，它可以直接读取应用程序属性。 这是如何读取`spring.devtools.remote.secret`属性并将其传递到服务器以进行身份验证。

始终建议使用`https://`作为连接协议，以便流量加密并且密码不会被拦截。

如果您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 Remote Update译：20.5.2远程更新

远程客户端以与local restart相同的方式监视应用程序类路径的更改。 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。 如果您对使用本地没有的云服务的功能进行迭代，这会很有帮助。 通常，远程更新和重新启动比完整的重建和部署周期快得多。

仅在远程客户端运行时才监视文件。 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. Packaging Your Application for Production译：21.包装您的生产申请

可执行的罐子可用于生产部署。 由于它们是独立的，它们也非常适合基于云的部署。

对于其他“生产就绪”功能（如健康，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，请参阅*Part V, “Spring Boot Actuator: Production-ready features”*。

22. What to Read Next译：22.下一步阅读什么

您现在应该了解如何使用Spring Boot以及您应遵循的一些最佳实践。 您现在可以深入了解特定的*Spring Boot features*，或者您可以跳过并阅读关于Spring Boot的“production ready”方面的内容。

Part IV. Spring Boot features译：第四部分。 Spring Boot功能

本节将介绍Spring Boot的细节。 在这里，您可以了解您可能想要使用和定制的关键功能。 如果您尚未这样做，则可能需要阅读“ Part II, “Getting Started” ”和“ Part III, “Using Spring Boot” ”部分，以便您具备良好的基础知识。

23. SpringApplication译：23. SpringApplication

`SpringApplication`类提供了一种便捷方式来引导从`main()`方法启动的Spring应用程序。 在很多情况下，您可以委派到静态`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 ::   v2.0.3.RELEASE

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplication            : Starting SampleApplication v0.1.0 on mycomputer
2013-07-31 00:08:16.166  INFO 56603 --- [           main] ationConfigServletWebServerApplicationContext : Refreshing org.springframework.boot.web.ser
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
```

默认情况下，会显示`INFO`日志消息，其中包括一些相关的启动详细信息，例如启动应用程序的用户。 如果您需要的日志级别不是`INFO`，则可以按照Section 26.4, “Log Levels”中所述对其进行设置，

23.1 Startup Failure译：23.1启动失败

如果您的应用程序无法启动，注册`FailureAnalyzers`将有机会提供专门的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被使用，则应该看到与以下消息类似的内容：

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Embedded servlet container failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that's listening on port 8080 or configure this application to listen on another port.
```

Spring Boot提供了许多`FailureAnalyzer`实现，并且您可以 add your own 。

如果没有故障分析仪能够处理异常情况，您仍然可以显示完整的情况报告以更好地了解问题所在。 为此，您需要enable the `debug` property或enable `DEBUG` logging `org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener` 。

例如，如果您使用`java -jar`运行应用程序，则可以按如下所示启用`debug`属性：

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

23.2 Customizing the Banner译：23.2自定义横幅

启动时打印的横幅可以通过将`banner.txt`文件添加到类路径中或通过将`spring.banner.location`属性设置为此类文件的位置来更改。 如果文件的编码不是UTF-8，则可以设置`spring.banner.charset` 。 除了一个文本文件，你还可以将添加`banner.gif` ， `banner.jpg` ，或`banner.png`图像文件到类路径或设置`spring.banner.image.location`属性。 图像被转换成ASCII艺术表现形式并打印在任何文字横幅上方。

在您的`banner.txt`文件中，您可以使用以下任何占位符：

表23.1。 横幅变量

Variable	描述
`${application.version}`	您的应用程序的版本号，如`MANIFEST.MF`所声明的。 例如，`Implementation-Version: 1.0`被打印为`1.0` 。
`${application.formatted-version}`	您的应用程序的版本号，如`MANIFEST.MF`所声明的并且已格式化以显示（用括号括起来并以`v`为前缀）。 例如`(v1.0)` 。
`${spring-boot.version}`	您正在使用的Spring Boot版本。 例如`2.0.3.RELEASE` 。
`${spring-boot.formatted-version}`	您正在使用的Spring Boot版本，已格式化显示（用括号括起来，前缀为`v` ）。 例如`(v2.0.3.RELEASE)` 。
`${Ansi.NAME}` （或`${AnsiColor.NAME}` ，`${AnsiBackground.NAME}` ，`${AnsiStyle.NAME}` ）	其中`NAME`是ANSI转义代码的名称。 详情请参阅`AnsiPropertySource` 。
`${application.title}`	您的申请的标题，如`MANIFEST.MF`所声明的。 例如`Implementation-Title: MyApp`被打印为`MyApp` 。

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`界面并实现您自己的`printBanner()`方法。

您还可以使用`spring.main.banner-mode`属性来确定横幅是否必须在`System.out` （`console` ）上打印，发送到配置的记录器（`log` ），还是根本不生产（`off` ）。

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

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 Customizing SpringApplication译：23.3定制SpringApplication

如果`SpringApplication`默认值与您的口味不符，则可以创建本地实例并对其进行自定义。 例如，要关闭横幅，你可以写：

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

传递给`SpringApplication`的构造函数参数是Spring bean的配置源。 在大多数情况下，这些类都是对`@Configuration`类的引用，但它们也可能是对XML配置或应扫描的包的引用。

也可以配置`SpringApplication`通过使用`application.properties`文件。 有关详细信息，请参阅*Chapter 24, Externalized Configuration* 。

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

23.4 Fluent Builder API 译：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），或者如果您更愿意使用“流利”构建器API，则可以使用 `SpringApplicationBuilder` 。

`SpringApplicationBuilder`可让您将多个方法调用链接在一起，并包含 `parent`和 `child`方法，可让您创建层次结构，如下例所示：

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

创建`ApplicationContext`层次结构时有一些限制。 例如，Web组件必须包含在子上下文中，并且父代和子代上下文都使用相同的`Environment` 。 有关完整的详细信息，请参阅`SpringApplicationBuilder` Javadoc 。

23.5 Application Events and Listeners 译：23.5应用程序事件和监听器

除了通常的Spring Framework事件（例如 `ContextRefreshedEvent`）之外， `SpringApplication`发送一些其他应用程序事件。

有些事件实际上是在创建`ApplicationContext`之前触发的，因此您无法将这些侦听器注册为`@Bean` 。 您可以使用`SpringApplication.addListeners(… )`方法或`SpringApplicationBuilder.listeners(… )`方法注册它们。
如果您希望自动注册这些侦听器，而不管创建应用程序的方式如何，则可以将 `META-INF/spring.factories`文件添加到项目中，并使用`org.springframework.context.ApplicationListener`密钥引用侦听器，如以下示例所示：

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

随着您的应用程序运行，应用程序事件按以下顺序发送：

1. An `ApplicationStartingEvent` is sent at the start of a run but before any processing, except for the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known but before the context is created.
3. An `ApplicationPreparedEvent` is sent just before the refresh is started but after bean definitions have been loaded.
4. An `ApplicationStartedEvent` is sent after the context has been refreshed but before any application and command-line runners have been called.
5. An `ApplicationReadyEvent` is sent after any application and command-line runners have been called. It indicates that the application is ready to service requests.
6. An `ApplicationFailedEvent` is sent if there is an exception on startup.

您通常不需要使用应用程序事件，但可以方便地知道它们存在。 在内部，Spring Boot使用事件来处理各种任务。

应用程序事件通过使用Spring Framework的事件发布机制发送。 该机制的一部分确保发布给子上下文中侦听器的事件也发布给任何祖先上下文中的侦听器。 因此，如果您的应用程序使用`SpringApplication`实例的层次结构， `SpringApplication`侦听器可能会收到同一类型应用程序事件的多个实例。

为了让你的监听器区分它的上下文事件和后代上下文事件，它应该请求它的应用上下文被注入，然后比较注入的上下文和事件的上下文。 上下文可以通过实现`ApplicationContextAware`注入，或者如果侦听器是bean，则可以通过使用`@Autowired` 。

23.6 Web Environment 译：23.6 Web环境

`SpringApplication`尝试以您的名义创建正确类型的`ApplicationContext` 。 用于确定`WebApplicationType`的算法非常简单：

- If Spring MVC is present, an `AnnotationConfigServletWebServerApplicationContext` is used
- If Spring MVC is not present and Spring WebFlux is present, an `AnnotationConfigReactiveWebServerApplicationContext` is used
- Otherwise, `AnnotationConfigApplicationContext` is used

这意味着如果您在同一个应用程序中使用Spring MVC和Spring WebFlux中的新`WebClient` ，则默认情况下会使用Spring MVC。 你可以通过调用`setWebApplicationType(WebApplicationType)`轻松地覆盖它。

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

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

23.7 Accessing Application Arguments 译：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，则可以注入一个`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`参数的访问权限，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"]
 }

}
```

春季启动也注册了一个`CommandLinePropertySource`同春`Environment` 。 这使您可以通过使用`@Value`注释来注入单个应用程序参数。

23.8 Using the ApplicationRunner or CommandLineRunner译：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。 两个接口都以相同的方式工作，并提供一个`run`方法，该方法在`SpringApplication.run(… )`完成之前`SpringApplication.run(… )` 。

`CommandLineRunner`接口提供对应用程序参数的访问，作为一个简单的字符串数组，而`ApplicationRunner`使用前面讨论的`ApplicationArguments`接口。 以下示例使用`run`方法显示了`CommandLineRunner` ：

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

@Component
public class MyBean implements CommandLineRunner {

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

}
```

如果定义了几个必须以特定顺序调用的`CommandLineRunner`或`ApplicationRunner` Bean，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 Application Exit译：23.9应用程序退出

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

此外，豆类可以实现`org.springframework.boot.ExitCodeGenerator` ，如果他们想什么时候返回一个特定的退出代码接口`SpringApplication.exit()`被调用。 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)));
 }

}
```

另外， `ExitCodeGenerator`接口可能由例外实现。 遇到这样的异常时，Spring Boot会返回执行的`getExitCode()`方法提供的退出代码。

23.10 Admin Features译：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`平台上`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可能有用。

如果您想知道应用程序在哪个HTTP端口上运行，请使用密钥`local.server.port`获取该属性。

Caution

启用此功能时要小心，因为MBean公开了关闭应用程序的方法。

24. Externalized Configuration译：24.外部化配置

Spring Boot允许您将配置外部化，以便您可以在不同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。 通过使用`@Value`注释，可以通过Spring的`Environment`抽象或bound to structured objects至`@ConfigurationProperties`访问属性值，从而将属性值直接注入到bean中。

Spring Boot使用非常特殊的`PropertySource`命令，旨在允许明智的覆盖值。 属性按以下顺序考虑：

1. Devtools global settings properties on your home directory (`~/.spring-boot-devtools.properties` when devtools is active).
2. `@TestPropertySource` annotations on your tests.
3. `@SpringBootTest#properties` annotation attribute on your tests.
4. Command line arguments.
5. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
6. `ServletConfig` init parameters.
7. `ServletContext` init parameters.
8. JNDI attributes from `java:comp/env`.
9. Java System properties (`System.getProperties()`).
10. OS environment variables.
11. A `RandomValuePropertySource` that has properties only in `random.*`.
12. Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` and YAML variants).
13. Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants).
14. Application properties outside of your packaged jar (`application.properties` and YAML variants).
15. Application properties packaged inside your jar (`application.properties` and YAML variants).
16. `@PropertySource` annotations on your `@Configuration` classes.
17. Default properties (specified by setting `SpringApplication.setDefaultProperties`).

为了提供一个具体的例子，假设你开发一个 `@Component` 一个使用 `name` 属性，见下面的例子：

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

@Component
public class MyBean {

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

    // ...

}
```

在您的应用程序类路径中（例如，在您的jar中），您可以使用`application.properties`文件为`name`提供合理的默认属性值。 在新环境中运行时，可以在您的jar外部提供`application.properties`文件来覆盖`name`。 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"` ）。

可以使用环境变量在命令行上提供`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
```

在上面的例子中，你最终`acme.name=test`在春节`Environment`。 您还可以在System属性中将JSON作为`spring.application.json`提供，如以下示例中所示：

```
$ 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 Configuring Random Values译：24.1配置随机值

`RandomValuePropertySource`对于注入随机值（例如，注入秘密或测试用例）非常有用。 它可以产生整数，长整数，uuids或字符串，如下例所示：

```
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*`语法是`OPEN value (,max) CLOSE`，其中`OPEN,CLOSE`是任何字符， `value,max`是整数。 如果提供`max`，则`value`是最小值， `max`是最大值（不包括）。

24.2 Accessing Command Line Properties译：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（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 Application Property Files译：24.3应用程序属性文件

`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. A `/config` subdirectory of the current directory
2. The current directory
3. A classpath `/config` package
4. The classpath root

该列表按优先顺序排列（在列表中较高的位置定义的属性会覆盖在较低位置定义的属性）。

您也可以将 use YAML ('.yml') files替代为'.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（这是逗号分隔的目录位置或文件路径列表）引用显式位置。 以下示例显示如何指定不同的文件名称：

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

以下示例显示如何指定两个位置：

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

`spring.config.name`和`spring.config.location`很早就用于确定哪些文件必须加载，因此它们必须定义为环境属性（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`/`（并且在运行`spring.config.name`加载在加载之前从`spring.config.name`生成的名称，包括配置文件特定的文件名）。 在`spring.config.location`中指定的文件按`spring.config.location`使用，不支持特定于配置文件的变体，并且被特定于配置文件的特性覆盖。

配置位置按相反顺序搜索。 默认情况下，配置的位置是`classpath:/,classpath:/config/,file:./,file:./config/`。 结果搜索顺序如下：

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

当通过使用`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/`，搜索顺序变为以下内容：

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

或者，使用`spring.config.additional-location`配置自定义配置位置时，除了默认位置以外，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/`其他位置，则搜索顺序变为以下内容：

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

此搜索顺序可让您在一个配置文件中指定默认值，然后在另一配置文件中选择性地覆盖这些值。 您可以在其中一个默认位置为您的应用程序提供默认值`application.properties`（或您选择的其他基本名称，`spring.config.name`）。 这些默认值可以在运行时被置于其中一个自定义位置的不同文件覆盖。

如果使用环境变量而非系统属性，则大多数操作系统不允许使用句点分隔的键名称，但可以使用下划线（例如，`SPRING_CONFIG_NAME`而不是`spring.config.name`）。

如果您的应用程序在容器中运行，那么可以使用JNDI属性（`java:comp/env`）或servlet上下文初始化参数，而不是使用环境变量或系统属性。

24.4 Profile-specific Properties译：24.4特定于配置文件的属性

除`application.properties`文件外，还可以使用以下命名约定来定义配置文件特定的属性：`application-{profile}.properties`。`Environment`有一组默认配置文件（默认为`[default]`），如果未设置活动配置文件，则使用该配置文件。 换句话说，如果没有显式激活配置文件，则加载`application-default.properties`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的文件，而不管配置文件特定的文件是否位于打包的jar内部或外部。

如果指定了多个配置文件，则应用最后赢取策略。 例如，由`spring.profiles.active`属性指定的配置文件将添加到通过`SpringApplication`API配置的配置文件之后，因此优先。

如果您在`spring.config.location`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。 如果您还想使用配置文件特定的属性，请使用`spring.config.location`目录。

24.5 Placeholders in Properties译：24.5属性中的占位符

中的值`application.properties`通过现有过滤`Environment`使用它们时，这样就可以返回参考先前定义值（例如，从系统性能）。

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

您也可以使用这种技术来创建现有Spring Boot属性的“short”变体。 有关详细信息，请参阅*Section 74.4, “Use ‘Short’ Command Line Arguments”操作指南*。

24.6 Using YAML Instead of Properties译：24.6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利格式。`SpringApplication`类自动支持YAML作为属性的替代方法，只要您的类路径中包含SnakeYAML库。

如果您使用“启动器”，则SnakeYAML将自动由`spring-boot-starter`提供。

24.6.1 Loading YAML译：24.6.1加载YAML

Spring框架提供了两个方便的类，可以用来加载YAML文档。 `YamlPropertiesFactoryBean`将YAML加载为`Properties`，将`YamlMapFactoryBean`加载YAML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列表以`[index]`解引用表示为属性键。 例如，请考虑以下YAML：

```
my:
servers:
 - dev.example.com
 - another.example.com
```

前面的例子将被转换成这些属性：

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

要通过使用Spring Boot的`Binder`实用程序（这是`@ConfigurationProperties`所做的）绑定到类似属性，需要在`java.util.List`（或`Set`）类型的目标Bean中拥有一个属性，并且您需要提供setter或initialize它具有可变的价值。 例如，以下示例绑定到以前显示的属性：

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

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

 public List<String> getServers() {
  return this.servers;
 }
}
```

24.6.2 Exposing YAML as Properties in the Spring Environment 译：24.6.2在 Spring环境 中将 YAML作为属性公开

该`YamlPropertySourceLoader`类可用于暴露YAML为`PropertySource`在春节`Environment`。 这样做可让您使用带占位符语法的`@Value`注释来访问YAML属性。

24.6.3 Multi-profile YAML Documents 译：24.6.3多配置文件YAML文档

您可以通过使用`spring.profiles`键指定文档适用的时间，在单个文件中指定多个特定于配置文件的YAML文档，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
server:
 address: 192.168.1.120
```

在上例中，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`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`和`production`配置文件，则该属性的值为`192.168.1.100`。

如果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，在下面的YAML中，我们为`spring.security.user.password`设置了仅在“默认”配置文件中可用的值：

```
server:
  port: 8000
---
spring:
  profiles: default
  security:
    user:
      password: weak
```

而在以下示例中，由于密码未附加到任何配置文件，因此始终设置密码，必须根据需要在所有其他配置文件中明确重置该密码：

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

通过使用`spring.profiles`元素指定的弹簧配置文件可以通过使用`!`字符来有选择地取消。 如果为单个文档指定了否定配置文件和非否定配置文件，则至少有一个非否

定配置文件必须匹配，且不存在否定配置文件可能匹配。

24.6.4 YAML Shortcomings译：24.6.4 YAML缺点

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

24.7 Type-safe Configuration Properties译：24.7类型安全的配置属性

使用`@Value("${property}")`注释来注入配置属性有时会很麻烦，尤其是在您使用多个属性或者您的数据具有分层性质的情况下。 Spring Boot提供了另一种使用属性的方法，可以让强类型bean管理和验证应用程序的配置，如以下示例所示：

```
package com.example;

import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("acme")
public class AcmeProperties {

 private boolean enabled;

 private InetAddress remoteAddress;

 private final Security security = new Security();

 public boolean isEnabled() { ... }

 public void setEnabled(boolean enabled) { ... }

 public InetAddress getRemoteAddress() { ... }

 public void setRemoteAddress(InetAddress remoteAddress) { ... }

 public Security getSecurity() { ... }

 public static class Security {

  private String username;

  private String password;

  private List<String> roles = new ArrayList<>(Collections.singleton("USER"));

  public String getUsername() { ... }

  public void setUsername(String username) { ... }

  public String getPassword() { ... }

  public void setPassword(String password) { ... }

  public List<String> getRoles() { ... }

  public void setRoles(List<String> roles) { ... }

 }
}
```

前面的POJO定义了以下属性：

- `acme.enabled`, with a value of `false` by default.
- `acme.remote-address`, with a type that can be coerced from `String`.
- `acme.security.username`, with a nested "security" object whose name is determined by the name of the property. In particular, the return type is not used at all there and could have been `SecurityProperties`.
- `acme.security.password`.
- `acme.security.roles`, with a collection of `String`.

getters和setter通常是强制性的，因为绑定是通过标准的Java Beans属性描述符来完成的，就像在Spring MVC中一样。 在以下情况下可能会忽略setter：

- Maps, as long as they are initialized, need a getter but not necessarily a setter, since they can be mutated by the binder.
- Collections and arrays can be accessed either through an index (typically with YAML) or by using a single comma-separated value (properties). In the latter case, a setter is mandatory. We recommend to always add a setter for such types. If you initialize a collection, make sure it is not immutable (as in the preceding example).
- If nested POJO properties are initialized (like the `Security` field in the preceding example), a setter is not required. If you want the binder to create the instance on the fly by using its default constructor, you need a setter.

有些人使用Project Lombok来自动添加getter和setter。 确保Lombok不会为这种类型生成任何特定的构造函数，因为它被容器自动使用来实例化对象。
最后，只考虑标准的Java Bean属性，并且不支持在静态属性上进行绑定。

另见 differences between `@Value` and `@ConfigurationProperties` 。

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

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

当以这种方式注册`@ConfigurationProperties` bean时，该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。 如果注释没有提供任何前缀，则只使用bean的完全限定名称。
上例中的bean名称是 `acme-com.example.AcmeProperties` 。

即使前面的配置为`AcmeProperties`创建了一个常规bean，我们也建议`@ConfigurationProperties`只处理环境，特别是不会从上下文中注入其他bean。 话虽如此，`@EnableConfigurationProperties`注释*也会*自动应用到您的项目中，以便从`Environment`配置任何*现有的*使用`@ConfigurationProperties`注释的`Environment` 。 通过确保`AcmeProperties`已经是bean，可以快捷键`MyConfiguration` ，如以下示例所示：

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

 // ... see the preceding example

}
```

这种配置方式对 `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` bean，可以像使用其他bean一样注入它们，如以下示例所示：

```
@Service
public class MyService {

 private final AcmeProperties properties;

 @Autowired
 public MyService(AcmeProperties properties) {
     this.properties = properties;
 }

  //...

 @PostConstruct
 public void openConnection() {
  Server server = new Server(this.properties.getRemoteAddress());
  // ...
 }

}
```

使用`@ConfigurationProperties`还可以生成元数据文件，IDE可以使用这些文件为自己的密钥提供自动完成功能。 详情请参阅Appendix B, *Configuration Metadata*附录。

24.7.1 Third-party Configuration译：24.7.1第三方配置

除了使用`@ConfigurationProperties`来注释一个类，您还可以在公共方法`@Bean`上使用它。 如果要将属性绑定到不在您控制之外的第三方组件，则这样做会特别有用。

要从 `Environment`属性配置一个bean，请将 `@ConfigurationProperties`添加到其注册的bean中，如以下示例所示：

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

任何以 `another`前缀定义的属性都 `AnotherComponent`类似于前面的 `AcmeProperties`示例的方式映射到该 `AnotherComponent` bean上。

24.7.2 Relaxed Binding译：24.7.2松绑定

Spring Boot使用一些宽松的规则将`Environment`属性绑定到`@ConfigurationProperties` bean，因此不需要`Environment`属性名称和bean属性名称之间的精确匹配。 常用的例子包括破折号分隔的环境属性（例如， `context-path`绑定到`contextPath` ）和大写的环境属性（例如， `PORT`绑定到`port` ）。

例如，请考虑以下 `@ConfigurationProperties`类：

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

 private String firstName;

 public String getFirstName() {
  return this.firstName;
 }

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

}
```

在前面的示例中，可以使用以下属性名称：

表24.1。 松绑定

Property	Note
`acme.my-project.person.first-name`	Kebab案例，建议在`.properties`和`.yml`文件中使用。
`acme.myProject.person.firstName`	标准骆驼大小写语法。
`acme.my_project.person.first_name`	下划线符号，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。

注释的`prefix`值 *必须*位于`-`大小写（小写字母并用`-`，例如`acme.my-project.person`）。

表24.2。 放宽每个财产来源的绑定规则

Property Source	Simple	List
属性文件	骆驼案，烤肉串案或下划线标记	使用`[ ]`或逗号分隔值的标准列表语法
YAML文件	骆驼案，烤肉串案或下划线标记	标准YAML列表语法或逗号分隔值
环境变量	大写格式，下划线为分隔符。 `_`不应在属性名称中使用	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，烤肉串案或下划线标记	使用`[ ]`或逗号分隔值的标准列表语法

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

24.7.3 Merging Complex Types译：24.7.3合并复杂类型

当列表配置在多个地方时，通过替换整个列表来覆盖作品。

例如，假设`MyPojo`对象的`name`和`description`属性默认为`null`。 下面的示例公开的列表`MyPojo`从对象`AcmeProperties`：

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

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

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

}
```

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`配置文件未处于活动状态，则`AcmeProperties.list`包含一个`MyPojo`条目，如前所述。 但是，如果`dev`配置文件已启用，则`list` *仍*只包含一个条目（名称为`my another name`，描述为`null`）。 此配置*不会*将第二个`MyPojo`实例添加到列表中，并且不会合并这些项目。

当在多个配置文件中指定`List`，将使用具有最高优先级（仅限那一个）的配置文件。 考虑下面的例子：

```
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，可以使用逗号分隔列表和YAML列表完全覆盖列表的内容。

对于`Map`属性，可以使用从多个来源绘制的属性值进行绑定。 但是，对于多个来源中的相同属性，将使用具有最高优先级的属性。 下面的示例公开了一个`Map<String, MyPojo>`从`AcmeProperties`：

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

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

 public Map<String, MyPojo> getMap() {
  return this.map;
 }

}
```

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`配置文件未处于活动状态，则`AcmeProperties.map`包含一个包含密钥`key1`（名称为`my name 1`，描述为`my description 1`）的`my description 1`。 但是，如果`dev`配置文件已启用，则`map`包含两个包含密钥`key1`（名称为`dev name 1`和描述`my description 1`）和`key2`（名称为`dev name 2`和描述为`dev description 2`）的`dev description 2`。

上述合并规则适用于来自所有属性源的属性，而不仅仅是YAML文件。

24.7.4 Properties Conversion译：24.7.4属性转换

当Spring Boot绑定到`@ConfigurationProperties`bean时，会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（具有名为`conversionService`的bean）或定制属性编辑器（通过`CustomEditorConfigurer`bean）或定制`Converters`（具有注释为`@ConfigurationPropertiesBinding`bean定义）。

由于此bean在应用程序生命周期中很早被请求，因此请确保限制您的`ConversionService`正在使用的依赖项。 通常，您需要的任何依赖项可能在创建时未完全初始化。 您可能需要重命名您的自定义`ConversionService`如果不需要配置键强制它只有依靠合格定制的转换器`@ConfigurationPropertiesBinding`。

Converting durations译:转换持续时间

Spring Boot有专门的支持来表达持续时间。 如果您公开`java.time.Duration`属性，则应用程序属性中的以下格式可用：

- A regular `long` representation (using milliseconds as the default unit unless a `@DurationUnit` has been specified)
- The standard ISO-8601 format used by `java.util.Duration`
- A more readable format where the value and the unit are coupled (e.g. `10s` means 10 seconds)

考虑下面的例子：

```
@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，会话超时`30`， `PT30S`和`30s`都是等价的。 的500ms的读超时可以以任何形式如下指定： `500`， `PT0.5S`和`500ms`。

您也可以使用任何支持的单位。 这些是：

- `ns` for nanoseconds
- `ms` for milliseconds
- `s` for seconds
- `m` for minutes
- `h` for hours
- `d` for days

默认单位是毫秒，可以使用`@DurationUnit`覆盖， `@DurationUnit`例所示。

如果您正在从以前的版本进行升级（仅使用`Long`来表示持续时间），请确保定义单位（使用`@DurationUnit`），如果它不是切换到`Duration`毫秒`Duration`。 这样做提供了一个透明的升级途径，同时支持更丰富的格式。

24.7.5 @ConfigurationProperties Validation译：24.7.5 @ConfigurationProperties验证

Spring Boot尝试验证`@ConfigurationProperties`类，只要它们使用Spring的`@Validated`注释进行注释。 您可以直接在配置类上使用JSR-303 `javax.validation`约束注释。 为此，请确保您的类路径上包含一个兼容的JSR-303实现，然后将约束注释添加到您的字段中，如以下示例所示：

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

 @NotNull
 private InetAddress remoteAddress;

 // ... getters and setters

}
```

您还可以通过注释`@Bean`方法触发验证，该方法使用`@Validated`创建配置属性。

尽管嵌套属性在绑定时也会被验证，但最好还是将相关字段注释为`@Valid`。 这确保即使未找到嵌套属性也会触发验证。 以下示例建立在前面的`AcmeProperties`示例上：

```
@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

 }

}
```

您还可以添加自定义的春天`Validator`通过创建称为bean定义`configurationPropertiesValidator`。 `@Bean`方法应声明为`static`。 配置属性验证器是在应用程序生命周期的早期创建的，并且声明`@Bean`方法为静态方法，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property validation sample显示如何设置。

`spring-boot-actuator`模块包含一个暴露所有`@ConfigurationProperties` bean的端点。 将您的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“ Production ready features ”部分。

24.7.6 @ConfigurationProperties vs. @Value 译：24.7.6 @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：

Feature	`@ConfigurationProperties`	`@Value`
Relaxed binding	是	没有
Meta-data support	是	没有
`SpEL`评估	没有	是

如果您为自己的组件定义了一组配置密钥，我们建议您将它们分组在POJO中，注释为`@ConfigurationProperties` 。 您还应该意识到，由于`@Value`不支持放宽绑定，因此如果您需要使用环境变量提供值，则不是一个好的选择。

最后，虽然可以在`@Value`编写`SpEL`表达式， `@Value`这些表达式不会从 application property files处理。

25. Profiles 译：25.配置文件

Spring Profiles提供了一种分离部分应用程序配置的方法，并使其仅在特定环境中可用。 任何`@Component`或`@Configuration`都可以用`@Profile`标记以限制其加载时间，如以下示例所示：

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

 // ...

}
```

您可以使用`spring.profiles.active` `Environment`属性来指定哪些配置文件处于活动状态。 您可以用本章前面所述的任何方式指定属性。 例如，您可以将其包含在您的`application.properties` ，如以下示例所示：

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

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

25.1 Adding Active Profiles 译：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`指定活动配置文件，然后使用命令行开关**替换**它们。

有时，将具有配置文件的属性**添加**到活动配置文件而不是替换它们会很有用。 可以使用`spring.profiles.include`属性无条件添加活动配置文件。
`SpringApplication`入口点还具有用于设置其他配置文件（即，在`spring.profiles.active`属性激活的配置文件之上）的Java API。
见`setAdditionalProfiles()`的方法SpringApplication 。

例如，当具有以下属性的应用程序是通过使用开关，运行 `--spring.profiles.active=prod` ，在 `proddb`个 `prodmq`轮廓也被激活：

```
---
my.property: fromyamlfile
---
spring.profiles: prod
spring.profiles.include:
  - proddb
  - prodmq
```

请记住，可以在YAML文档中定义`spring.profiles`属性，以确定此特定文档何时包含在配置中。 有关更多详细信息，请参阅Section 74.7, “Change Configuration Depending on the Environment” 。

25.2 Programmatically Setting Profiles 译：25.2以编程方式设置配置文件

在应用程序运行之前，您可以通过调用`SpringApplication.setAdditionalProfiles(… )`以编程方式设置活动配置文件。 通过使用Spring的`ConfigurableEnvironment`接口也可以激活配置文件。

25.3 Profile-specific Configuration Files 译：25.3配置文件特定的配置文件

`application.properties` （或`application.yml` ）和通过`@ConfigurationProperties`引用的文件的特定于配置文件的变体均视为文件并加载。 有关详细信息，请参阅“ Section 24.4, “Profile-specific Properties” ”。

26. Logging 译：26.记录

Spring Boot将Commons Logging用于所有内部日志记录，但将底层日志实现保留为打开状态。 提供用于默认配置Java Util Logging ， Log4J2 ，和Logback 。 在每种情况下，记录器都预先配置为使用控制台输出，并提供可选的文件输出。

默认情况下，如果使用“启动器”，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日志记录，Commons日志记录，Log4J或SLF4J的依赖

库全部正常工作。

Java有很多可用的日志框架。 如果上面的列表看起来很混乱，请不要担心。 一般来说，你不需要改变你的日志依赖性，Spring Boot的默认工作就可以。

26.1 Log Format 译：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 WebApplicationConte
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization compl
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: [/*
```

以下项目被输出：

- Date and Time: Millisecond precision and easily sortable.
- Log Level: `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`.
- Process ID.
- A `---` separator to distinguish the start of actual log messages.
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Logger name: This is usually the source class name (often abbreviated).
- The log message.

Logback没有`FATAL`级别。 它被映射到`ERROR` 。

26.2 Console Output 译：26.2控制台输出

默认日志配置会在写入消息时将消息回传给控制台。 默认情况下，会记录`ERROR`级别， `WARN`级别和`INFO`级别的消息。 您还可以通过以`--debug`标志启动应用程序来启用“调试”模式。

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

您还可以指定`debug=true`在`application.properties` 。

当启用调试模式时，将选择核心记录器（嵌入式容器，Hibernate和Spring Boot）配置为输出更多信息。 启用调试模式*不会*配置您的应用程序记录所有消息`DEBUG`水平。

或者，您可以启用“跟踪”模式，方法是使用`--trace`标志（或`trace=true`的`application.properties` ）启动应用程序。 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产品组合）启用跟踪记录。

26.2.1 Color-coded Output 译：26.2.1颜色编码的输出

如果您的终端支持ANSI，则会使用彩色输出来提高可读性。 您可以将`spring.output.ansi.enabled`设置为supported value以覆盖自动检测。

颜色编码通过使用`%clr`转换字进行配置。 最简单的形式是，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：

Level	Color
`FATAL`	红
`ERROR`	红
`WARN`	黄色
`INFO`	绿色
`DEBUG`	绿色
`TRACE`	绿色

或者，您可以通过将其作为选项提供给转换来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 File Output译：26.3文件输出

默认情况下，Spring Boot仅记录到控制台，不写入日志文件。 如果除了输出控制台之外还想写日志文件，则需要设置`logging.file`或`logging.path`属性（例如，在您的`application.properties`）。

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`	Example	描述
（没有）	*（没有）*		仅限控制台日志。
具体文件	*（没有）*	`my.log`	写入指定的日志文件。 名称可以是确切的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。 名称可以是确切的位置或相对于当前目录。

日志文件在达到10 MB时进行旋转，并且与控制台输出一样，缺省情况下会记录`ERROR`级别，`WARN`级别和`INFO`级别的消息。 大小限制可以使用`logging.file.max-size`属性进行更改。 除非设置了`logging.file.max-history`属性，否则之前旋转的文件将无限期地归档。

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。

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

26.4 Log Levels译：26.4日志级别

所有支持的日志记录系统都可以使用`logging.level.<logger-name>=<level>`其中`level`是TRACE，DEBUG，INFO，WARN，ERROR，FATAL或OFF之一）在Spring `Environment`（例如`application.properties`）中设置记录器级别。 `root`记录器可以通过使用`logging.level.root`进行配置。

以下示例显示了`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 Custom Log Configuration译：26.5自定义日志配置

可以通过在类路径中包含适当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进行自定义：`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称。 您还可以完全禁用Spring Boot的日志记录配置，使用值为`none`。

由于在创建`ApplicationContext`之前初始化日志记录，因此无法从`@Configuration`文件中控制`@PropertySources`日志。 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，加载以下文件：

Logging System	Customization
的logback	`logback-spring.xml`，`logback-spring.groovy`，`logback.xml`，或`logback.groovy`
Log4j2	`log4j2-spring.xml`或`log4j2.xml`
JDK（Java Util日志记录）	`logging.properties`

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`）。 如果您使用标准配置位置，则Spring无法完全控制日志初始化。

Java Util Logging存在已知的类加载问题，当从“可执行jar”运行时会导致问题。 如果可能的话，我们建议您从“可执行的jar”运行时避免它。

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 Environment	System Property	Comments
`logging.exception-conversion-word`	`LOG_EXCEPTION_CONVERSION_WORD`	记录异常时使用的转换字。
`logging.file`	`LOG_FILE`	如果已定义，则用于默认的日志配置。
`logging.file.max-size`	`LOG_FILE_MAX_SIZE`	最大日志文件大小（如果启用LOG_FILE）。（仅支持默认的Logback设置。）

Spring Environment	System Property	Comments
`logging.file.max-history`	`LOG_FILE_MAX_HISTORY`	保留的归档日志文件的最大数量（如果启用LOG_FILE）。（仅支持默认的Logback设置。）
`logging.path`	`LOG_PATH`	如果已定义，则用于默认的日志配置。
`logging.pattern.console`	`CONSOLE_LOG_PATTERN`	在控制台上使用的日志模式（stdout）。（仅支持默认的Logback设置。）
`logging.pattern.dateformat`	`LOG_DATEFORMAT_PATTERN`	日志日期格式的Appender模式。（仅支持默认的Logback设置。）
`logging.pattern.file`	`FILE_LOG_PATTERN`	在文件中使用的日志模式（如果启用了`LOG_FILE`）。（仅支持默认的Logback设置。）
`logging.pattern.level`	`LOG_LEVEL_PATTERN`	呈现日志级别时使用的格式（默认为`%5p`）。（仅支持默认的Logback设置。）
`PID`	`PID`	当前进程ID（如果可能，还没有定义为OS环境变量时发现）。

所有支持的日志记录系统在分析其配置文件时都可以查阅系统属性。 有关示例，请参阅`spring-boot.jar`的默认配置：

- Logback
- Log4j 2
- Java Util logging

如果您想在日志记录属性中使用占位符，则应该使用Spring Boot's syntax而不是基础框架的语法。 值得注意的是，如果您使用Logback，则应该使用`:`作为属性名称与其默认值之间的分隔符，而不使用`:-`。

您可以添加MDC和其他临时内容仅重写记录行`LOG_LEVEL_PATTERN`（或`logging.pattern.level`用的logback）。 例如，如果使用`logging.pattern.level=user:%X{user} %5p`，则默认日志格式包含“user”的MDC条目（如果存在），如以下示例所示。

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

26.6 Logback Extensions 译：26.6 Logback扩展

Spring Boot包含许多可用于高级配置的Logback扩展。 您可以在`logback-spring.xml`配置文件中使用这些扩展。

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

这些扩展名不能与Logback的configuration scanning一起使用。 如果您尝试这样做，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][springPr
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProfile], current ElementPath is [[configuration][springPro
```

26.6.1 Profile-specific Configuration 译：26.6.1配置文件特定的配置

`<springProfile>`标签可让您根据活动的Spring配置文件选择性地包含或排除配置部分。 配置文件部分在`<configuration>`元素的任何位置都受支持。 使用`name`属性指定哪个配置文件接受配置。 多个配置文件可以用逗号分隔的列表指定。 以下清单显示了三个样本配置文件：

```
<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>
```

26.6.2 Environment Properties 译：26.6.2环境属性

使用`<springProperty>`标签可以显示Spring `Environment`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，则这样做会很有用。 该标签的工作方式与Logback的标准`<property>`标签类似。 然而，而不是指定一个直接`value`，指定`source`的财产（从`Environment`）。 如果您需要将属性存储在`local`范围以外的其他`local`，则可以使用`scope`属性。 如果您需要回退值（如果该属性未在`Environment`设置），则可以使用`defaultValue`属性。 以下示例显示如何公开要在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>
```

> 所述`source`必须在串的情况下（如指定`my.property-name`）。 但是，通过使用宽松的规则，可以将属性添加到`Environment`。

27. Developing Web Applications译：27.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应用程序都使用`spring-boot-starter-web`模块快速启动并运行。 您还可以选择使用`spring-boot-starter-webflux`模块构建反应性Web应用程序。

如果您尚未开发Spring Boot Web应用程序，则可以按照“Hello World！”进行操作。 例如*Getting started*部分。

27.1 The “Spring Web MVC Framework”译：27.1“Spring Web MVC框架”

Spring Web MVC framework （通常简称为“Spring MVC”）是一个丰富的“模型视图控制器”Web框架。 Spring MVC允许您创建特殊的`@Controller`或`@RestController` bean来处理传入的HTTP请求。 您的控制器中的方法通过使用`@RequestMapping`注释映射到HTTP。

以下代码显示了提供JSON数据的典型`@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 Framework的一部分，详细信息请参阅reference documentation 。 还有几个指南，涵盖了spring.io/guides提供的Spring MVC。

27.1.1 Spring MVC Auto-configuration译：27.1.1 Spring MVC自动配置

Spring Boot为Spring MVC提供了自动配置，可与大多数应用程序配合使用。

自动配置在Spring的默认设置之上添加了以下功能：

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document)).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered later in this document).
- Automatic registration of `MessageCodesResolver` (covered later in this document).
- Static `index.html` support.
- Custom `Favicon` support (covered later in this document).
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

如果你想保持弹簧引导MVC功能和您要添加其他MVC configuration （拦截器，格式化，视图控制器，以及其他功能），你可以添加自己`@Configuration`类型的类`WebMvcConfigurer`但**没有** `@EnableWebMvc`。 如果您希望提供`RequestMappingHandlerAdapter`或`ExceptionHandlerExceptionResolver`自定义实例，`RequestMappingHandlerMapping`可以声明`WebMvcRegistrationsAdapter`实例来提供此类组件。

如果你想完全控制Spring MVC，你可以添加你自己的`@Configuration`注释`@EnableWebMvc`。

27.1.2 HttpMessageConverters译：27.1.2 HttpMessageConverters

Spring MVC使用`HttpMessageConverter`接口来转换HTTP请求和响应。 包装盒中包含明智的默认设置。 例如，可以将对象自动转换为JSON（通过使用Jackson库）或XML（通过使用Jackson XML扩展（如果可用），或者如果Jackson XML扩展不可用，则通过使用JAXB）。 默认情况下，字符串编码为`UTF-8`。

如果您需要添加或自定义转换器，则可以使用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);
 }

}
```

任何存在于上下文中的`HttpMessageConverter` bean都会添加到转换器列表中。 您也可以用相同的方式覆盖默认转换器。

27.1.3 Custom JSON Serializers and Deserializers译：27.1.3自定义JSON序列化器和反序列化器

如果您使用Jackson对JSON数据进行序列化和反序列化，则可能需要编写自己的`JsonSerializer`和`JsonDeserializer`类。 自定义序列化器通常为registered with Jackson through a module ，但Spring Boot提供了备选`@JsonComponent`注释，可以更容易地直接注册Spring Bean。

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，如下例所示：

```
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> {
  // ...
 }

}
```

所有`@JsonComponent`豆在`ApplicationContext`会自动与杰克逊注册。 由于`@JsonComponent`进行元注释， `@Component`适用通常的组件扫描规则。

Spring Boot还提供了`JsonObjectSerializer`和`JsonObjectDeserializer`基类，它们在序列化对象时为标准杰克逊版本提供了有用的替代方法。 有关详细信息，请参阅Javadoc中的`JsonObjectSerializer`和`JsonObjectDeserializer` 。

27.1.4 MessageCodesResolver 译：27.1.4 MessageCodesResolver

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

27.1.5 Static Content 译：27.1.5静态内容

默认情况下，春季引导服务从目录称为静态内容`/static` （或`/public`或者`/resources`或者`/META-INF/resources`在classpath或从根） `ServletContext` 。 它使用Spring MVC中的`ResourceHttpRequestHandler` ，以便您可以通过添加自己的`WebMvcConfigurer`并重写`addResourceHandlers`方法来修改该行为。

在独立的Web应用程序中，如果Spring决定不处理它，则容器中的默认Servlet也将启用并作为回退，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总是可以处理通过`DispatcherServlet`请求。

默认情况下，资源映射到`/**` ，但您可以使用`spring.mvc.static-path-pattern`属性调整该资源。 例如，将所有资源重新分配到`/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（使用目录位置列表替换缺省值）来自定义静态资源位置。 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”静态资源位置外， Webjars content还有一个特例。 `/webjars/**`中包含路径的任何资源如果以Webjars格式打包，则会从jar文件提供。

如果应用程序打包为jar，请勿使用`src/main/webapp`目录。 虽然这个目录是一个通用的标准，它**仅适用**于战争的包装，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，允许使用例如缓存清除静态资源或使用Webjars的版本不可知URL。

要为Webjar使用版本不可知的URL，请添加`webjars-locator-core`依赖项。 然后声明你的Webjar。 以jQuery为例，在`"/webjars/jquery/x.y.z/jquery.min.js"`添加`"/webjars/jquery/jquery.min.js"`结果。 其中`x.y.z`是Webjar版本。

如果您使用JBoss，则需要声明`webjars-locator-jboss-vfs`依赖项而不是`webjars-locator-core` 。 否则，所有Webjars解决为`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中有效地添加内容散列（如`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>` ）：

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

链接资源在运行时被重写模板，得益于`ResourceUrlEncodingFilter`被自动设定Thymeleaf和FreeMarker的。 使用JSP时，应该手动声明此过滤器。 其他模板引擎当前不会自动支持，但可以使用自定义模板宏/助手和`ResourceUrlProvider`的使用。

例如，在使用JavaScript模块加载程序动态加载资源时，重命名文件不是一种选择。 这就是为什么其他战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
```

通过此配置，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（`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>` ）。

有关更多支持选项，请参阅`ResourceProperties` 。

这个特性已经在专用的 blog post和Spring Framework的 reference documentation中进行了详细描述。

27.1.6 Welcome Page译：27.1.6欢迎页面

Spring Boot支持静态和模板欢迎页面。 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。

27.1.7 Custom Favicon译：27.1.7自定义Favicon

Spring Boot `favicon.ico`在配置的静态内容位置和类路径的根目录（ `favicon.ico`中查找`favicon.ico` 。 如果存在这样的文件，它会自动用作应用程序的图标。

27.1.8 Path Matching and Content Negotiation译：27.1.8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（例如，Controller方法中的`@GetMapping`注释）匹配，将传入的HTTP请求映射到处理程序。

Spring Boot选择默认禁用后缀模式匹配，这意味着像`"GET /projects/spring-boot.json"`这样的请求不会与`@GetMapping("/projects/spring-boot")`映射匹配。 这被认为是best practice for Spring MVC applications 。 此功能在过去对于没有发送正确的“Accept”请求标头的HTTP客户端来说非常有用; 我们需要确保将正确的内容类型发送到客户端。 如今，内容协商更可靠。

还有其他方法可以处理HTTP客户端，它们不会始终发送正确的“接受”请求标头。 我们可以使用查询参数来确保类似`"GET /projects/spring-boot?format=json"`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
```

如果您了解注意事项并仍然希望应用程序使用后缀模式匹配，则需要进行以下配置：

```
spring.mvc.contentnegotiation.favor-path-extension=true

# You can also restrict that feature to known extensions only
# spring.mvc.pathmatch.use-registered-suffix-pattern=true

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

27.1.9 ConfigurableWebBindingInitializer译：27.1.9 ConfigurableWebBindingInitializer

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

27.1.10 Template Engines译：27.1.10模板引擎

除了REST Web服务，您还可以使用Spring MVC为动态HTML内容提供服务。 Spring MVC支持各种模板技术，包括Thymeleaf，FreeMarker和JSP。 另外，许多其他模板引擎还包括他们自己的Spring MVC集成。

Spring Boot包含以下模板引擎的自动配置支持：

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

如果可能的话，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known limitations 。

当您使用默认配置的模板引擎之一时，您的模板将从`src/main/resources/templates`自动`src/main/resources/templates` 。

根据您运行应用程序的方式，IntelliJ IDEA以不同的方式排序类路径。 使用主方法在IDE中运行应用程序会导致与使用Maven或Gradle或从其打包的jar运行应用程序时不同的顺序。 这可能会导致Spring Boot无法在类路径中找到模板。 如果你有这个问题，你可以在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/` 。

27.1.11 Error Handling译：27.1.11错误处理

默认情况下，Spring Boot提供了一个`/error`映射，以合理的方式处理所有错误，并将其注册为servlet容器中的“全局”错误页面。 对于机器客户端，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，也呈现在HTML格式的相同数据（定制它，加上A A A€œwhitelabelâ€错误观点`View`解析为`error` ）。 要完全替换默认行为，可以实现`ErrorController`并注册该类型的bean定义或添加类型为`ErrorAttributes`的bean以使用现有机制，但替换内容。

`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（默认情况下专门处理`text/html`并为其他所有内容提供回退），此功能特别有用。 为此，请扩展`BasicErrorController` ，添加一个具有`produces`属性的`@RequestMapping`的公共方法，并创建一个新类型的bean。

您还可以定义一个用`@ControllerAdvice`注释的类来定制JSON文档以返回特定的控制器和/或异常类型，如下例所示：

```
@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` 由与 `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。

Custom Error Pages 译:自定义错误页面

如果要为给定的状态代码显示自定义HTML错误页面，则可以将文件添加到 `/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。 文件的名称应该是确切的状态码或系列掩码。

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所示：

```
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bean，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` methods和 `@ControllerAdvice` 。 `ErrorController` 然后拿起任何未处理的异常。

Mapping Error Pages outside of Spring MVC 译:在Spring MVC之外映射错误页面

对于不使用Spring MVC的应用程序，可以使用 `ErrorPageRegistrar` 接口直接注册 `ErrorPages` 。 这个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` 。

```
@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"));
 }

}
```

如果注册 `ErrorPage` 的路径最终由 `Filter` 处理（如同一些非Spring Web框架（例如Jersey和Wicket）常见的情况），那么必须将 `Filter` 明确注册为 `ERROR` 调度程序，如下面的例子：

```
@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 `FilterRegistrationBean` 不包括 `ERROR` 调度程序类型。

注意：部署到servlet容器时，Spring Boot会使用其错误页面过滤器将具有错误状态的请求转发到适当的错误页面。 如果响应尚未提交，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和更高版本在成功完成servlet的服务方法后提交响应。 你应该通过设置禁用此行为 `com.ibm.ws.webcontainer.invokeFlushAfterService` 至 `false` 。

27.1.12 Spring HATEOAS 译：27.1.12 Spring HATEOAS

如果您开发了一个使用超媒体的RESTful API，Spring Boot为Spring HATEOAS提供了自动配置，可与大多数应用程序配合使用。 自动配置取代了使用 `@EnableHypermediaSupport` 并注册大量bean的需求，以简化构建基于超媒体的应用程序，包括 `LinkDiscoverers` （用于客户端支持）和 `ObjectMapper` 配置为将响应正确编组为所需的表示形式。 `ObjectMapper` 是通过设置各种 `spring.jackson.*` 属性或 `Jackson2ObjectMapperBuilder` bean（如果存在）来 `Jackson2ObjectMapperBuilder` 。

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

27.1.13 CORS Support 译：27.1.13 CORS支持

Cross-origin resource sharing （CORS）是 W3C specification通过实施 most browsers ，可以让你在被授权什么样的跨域请求的灵活方式指定，而不是使用一些不太安全的，并不太强大的方法，如IFRAME或JSONP。

截至4.2版本，Spring MVC supports CORS 。 在Spring Boot应用程序中使用controller method CORS configuration和 `@CrossOrigin` 注释不需要任何特定配置。 Global CORS configuration可以通过使用自定义的 `addCorsMappings(CorsRegistry)` 方法注册 `WebMvcConfigurer` bean来定义，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/**");
   }
  };
 }
}
```

27.2 The “Spring WebFlux Framework” 译：27.2“Spring WebFlux框架”

Spring WebFlux是Spring Framework 5.0中引入的新的反应式Web框架。 与Spring MVC不同，它不需要Servlet API，完全异步和非阻塞，并通过the Reactor project实现Reactive Streams规范。

Spring WebFlux有两种风格：基于功能和基于注释的。 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，详细信息请 参阅reference documentation 。

您可以根据需要定义许多`RouterFunction` bean，以模块化路由器的定义。 如果您需要应用优先级，则可以订购豆类。

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

在应用程序中添加`spring-boot-starter-web`和`spring-boot-starter-webflux`模块将导致Spring Boot自动配置Spring MVC，而不是WebFlux。 之所以选择此行为是因为许多Spring开发人员将`spring-boot-starter-webflux`添加到其Spring MVC应用程序中以使用反应`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。

27.2.1 Spring WebFlux Auto-configuration 译：27.2.1 Spring WebFlux自动配置

Spring Boot为Spring WebFlux提供了自动配置，可与大多数应用程序配合使用。

自动配置在Spring的默认设置之上添加了以下功能：

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described later in this document).
- Support for serving static resources, including support for WebJars (described later in this document).

如果您想保留Spring Boot WebFlux的功能，并且想要添加额外的 WebFlux configuration ，则可以添加自己的 `@Configuration`类 `WebFluxConfigurer`但 不 `@EnableWebFlux` 。

如果你想完全控制Spring WebFlux，你可以添加自己的 `@Configuration`注释 `@EnableWebFlux` 。

27.2.2 HTTP Codecs with HttpMessageReaders and HttpMessageWriters 译：27.2.2使用 HttpMessageReaders和 HttpMessageWriters的 HTTP编解码器

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

Spring Boot通过使用`CodecCustomizer`实例进一步定制。 例如， `spring.jackson.*`配置密钥适用于Jackson编解码器。

如果您需要添加或自定义编解码器，则可以创建自定义 `CodecCustomizer`组件，如下例所示：

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

@Configuration
public class MyConfiguration {

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

}
```

您还可以利用 Boot's custom JSON serializers and deserializers 。

27.2.3 Static Content 译：27.2.3静态内容

默认情况下，Spring Boot会在类路径中提供`/static` （或`/public`或`/resources`或`/META-INF/resources` ）目录中的静态内容。 它使用`ResourceWebHandler`从春WebFlux，让您可以通过添加自己的修改行为`WebFluxConfigurer`并重写`addResourceHandlers`方法。

默认情况下，资源映射到`/**` ，但可以通过设置`spring.webflux.static-path-pattern`属性来调整资源。 例如，将所有资源重新分配到`/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。 这样做会用目录位置列表替换默认值。 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，如果您的任何位置在启动时都有`index.html`，则它是应用程序的主页。

除了前面列出的“标准”静态资源位置之外，还有一个特例是Webjars content 。 在路径的任何资源`/webjars/**`从jar文件送达，如果他们被包装在Webjars格式。

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

27.2.4 Template Engines译：27.2.4模板引擎

除了REST Web服务外，您还可以使用Spring WebFlux提供动态HTML内容。 Spring WebFlux支持各种模板技术，包括Thymeleaf，FreeMarker和小胡子。

Spring Boot包含以下模板引擎的自动配置支持：

- FreeMarker
- Thymeleaf
- Mustache

当您使用默认配置的模板引擎之一时，您的模板会从`src/main/resources/templates`自动`src/main/resources/templates`。

27.2.5 Error Handling译：27.2.5错误处理

Spring Boot提供了一个以合理的方式处理所有错误的`WebExceptionHandler`。 它在处理顺序中的位置就在WebFlux提供的处理程序之前，这被认为是最后一个处理程序。 对于机器客户端，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“白色标签”错误处理程序以HTML格式呈现相同的数据。 您也可以提供自己的HTML模板来显示错误（请参阅next section ）。

定制此功能的第一步通常涉及使用现有机制，但替换或增加错误内容。 为此，您可以添加一个类型为`ErrorAttributes`的bean。

要更改错误处理行为，可以实现`ErrorWebExceptionHandler`并注册该类型的bean定义。 因为`WebExceptionHandler`是相当低级的，所以Spring Boot还提供了一个方便的`AbstractErrorWebExceptionHandler`，让您以WebFlux功能的方式处理错误，如以下示例所示：

```
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，您也可以直接`DefaultErrorWebExceptionHandler`并覆盖特定的方法。

Custom Error Pages译：自定义错误页面

如果要为给定的状态代码显示自定义HTML错误页面，则可以将文件添加到`/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。 文件的名称应该是确切的状态码或系列掩码。

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所示：

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

27.2.6 Web Filters译：27.2.6Web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` bean将被自动用于过滤每个交换。

如果过滤器的顺序很重要，可以执行`Ordered`或用`@Order`注释。 Spring Boot自动配置可以为您配置网页过滤器。 当它这样做时，将使用下表中显示的订单：

Web Filter	Order
`MetricsWebFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`WebFilterChainProxy`（Spring Security）	`-100`

Web Filter	Order
`HttpTraceWebFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

27.3 JAX-RS and Jersey 译：27.3 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实现，而不是Spring MVC。 Jersey如果您在应用程序上下文`@Bean`其`Servlet`或`Filter`注册为`@Bean`， 则 1.x和Apache CXF的开箱即可正常工作。 Jersey 2.x具有一些原生Spring支持，因此我们还在Spring Boot中为它提供了自动配置支持以及一个启动器。

要开始使用2.X新泽西州，包括 `spring-boot-starter-jersey`的依赖，那么你就需要一个 `@Bean`类型 `ResourceConfig`其中注册您的所有端点，如下面的例子：

```
@Component
public class JerseyConfig extends ResourceConfig {

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

}
```

Jersey对扫描可执行档案的支持相当有限。 例如，运行可执行文件时，它无法扫描`WEB-INF/classes`找到的包中的端点。 为避免此限制，不应使用`packages`方法，并应使用`register`方法单独注册端点，如上例所示。

对于更高级的自定义，您还可以注册任意数量的实现 `ResourceConfigCustomizer`的bean。

所有注册的端点应为 `@Components`并带有HTTP资源注释（ `@GET`及其他），如以下示例所示：

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

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

}
```

由于`Endpoint`是春天`@Component`，它的生命周期由Spring管理，你可以使用`@Autowired`注释注入依赖和使用`@Value`注释注入外部配置。 默认情况下，Jersey servlet已注册并映射到`/*`。 您可以通过添加改变映射`@ApplicationPath`您`ResourceConfig`。

默认情况下，Jersey被设置为名为`jerseyServletRegistration`的`@Bean`类型的`ServletRegistrationBean` `jerseyServletRegistration`。 默认情况下，servlet会被懒惰地初始化，但您可以通过设置`spring.jersey.servlet.load-on-startup`定义该行为。 您可以通过使用相同名称创建自己的一个来禁用或覆盖该bean。 也可以通过设置使用一个过滤器，而不是一个servlet `spring.jersey.type=filter`（在这种情况下， `@Bean`替换或重写是`jerseyFilterRegistration`）。 该过滤器有一个`@Order`，您可以使用`spring.jersey.filter.order`进行设置。 通过使用`spring.jersey.init.*`来指定属性映射，servlet和过滤器注册都可以被赋予init参数。

有一个Jersey sample，这样你就可以看到如何设置。 还有一个Jersey 1.x sample 。 请注意，在Jersey 1.x示例中，spring-boot maven插件已被配置为解压缩一些Jersey瓶，以便它们可以通过JAX-RS实现进行扫描（因为样本要求在`Filter`扫描它们注册）。 如果您的任何JAX-RS资源打包为嵌套罐，您可能需要执行相同的操作。

27.4 Embedded Servlet Container Support 译：27.4嵌入式Servlet容器支持

春季启动包括嵌入式支持Tomcat， Jetty，并Undertow服务器。 大多数开发人员使用适当的“启动器”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。

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。 当您的应用程序正在运行时，该目录可能会被`tmpwatch`删除，从而导致失败。 为了避免这种情况，您可能需要定制`tmpwatch`配置，以便`tomcat.*`目录不会被删除或配置`server.tomcat.basedir`，以便嵌入式Tomcat使用不同的位置。

27.4.1 Servlets, Filters, and listeners 译：27.4.1 Servlet，过滤器和监听器

使用嵌入式servlet容器时，可以通过使用Spring bean或通过扫描Servlet组件来注册Servlet规范中的Servlet，过滤器和所有侦听器（例如 `HttpSessionListener`）。

Registering Servlets, Filters, and Listeners as Spring Beans 译：将 Spring Servlet，过滤器和监听器注册为 Spring Bean

任何`Servlet`， `Filter`，或servlet `*Listener`实例，它是一个Spring bean与嵌入的容器中注册。 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 bean的情况下，bean名称被用作路径前缀。 过滤器映射到`/*`。

如果以公约为基础测绘不够灵活，你可以使用 `ServletRegistrationBean`， `FilterRegistrationBean`，并 `ServletListenerRegistrationBean`类的完全控制。

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几个示例（低位值意味着更高的优先级）：

Servlet Filter	Order
`OrderedCharacterEncodingFilter`	`Ordered.HIGHEST_PRECEDENCE`
`WebMvcMetricsFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`ErrorPageFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`

Servlet Filter	Order
`HttpTraceFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

离开Filter beans无序通常是安全的。

如果需要特定顺序，您应该避免配置一个读取请求主体的过滤器（`Ordered.HIGHEST_PRECEDENCE`，因为它可能违背应用程序的字符编码配置。 如果一个Servlet过滤器包装请求，它应该配置一个小于或等于`FilterRegistrationBean.REQUEST_WRAPPER_FILTER_MAX_ORDER`。

27.4.2 Servlet Context Initialization译：27.4.2 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。 这是一个有意的设计决策，旨在降低设计在战争中运行的第三方库可能破坏Spring Boot应用程序的风险。

如果您需要在Spring Boot应用程序中执行Servlet上下文初始化，则应该注册一个实现`org.springframework.boot.web.servlet.ServletContextInitializer`接口的bean。 单个`onStartup`方法提供对`ServletContext`访问权限，并且如有必要，可以轻松用作现有`WebApplicationInitializer`的适配器。

Scanning for Servlets, Filters, and listeners译：扫描Servlet，筛选器和侦听器

当使用嵌入式容器中，类自动登记注解为`@WebServlet`， `@WebFilter`和`@WebListener`可通过使用启用`@ServletComponentScan`。

`@ServletComponentScan`在独立容器中不起作用，而独立容器使用容器的内置发现机制。

27.4.3 The ServletWebServerApplicationContext译：27.4.3 ServletWebServerApplicationContext

在引擎盖下，Spring Boot为嵌入式servlet容器支持使用了不同类型的`ApplicationContext`。 `ServletWebServerApplicationContext`是一种特殊类型的`WebApplicationContext`，通过搜索单个`ServletWebServerFactory` bean来引导自身。 通常一个`TomcatServletWebServerFactory`，`JettyServletWebServerFactory`，或`UndertowServletWebServerFactory`已经自动配置。

您通常不需要知道这些实现类。 大多数应用程序都是自动配置的，并且以您的名义创建适当的`ApplicationContext`和`ServletWebServerFactory`。

27.4.4 Customizing Embedded Servlet Containers译：27.4.4定制嵌入式Servlet容器

通用的servlet容器设置可以使用Spring `Environment`属性进行配置。 通常，您可以在`application.properties`文件中定义属性。

通用服务器设置包括：

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, and so on.
- Session settings: Whether the session is persistent (`server.servlet.session.persistence`), session timeout (`server.servlet.session.timeout`), location of session data (`server.servlet.session.store-dir`), and session-cookie configuration (`server.servlet.session.cookie.*`).
- Error management: Location of the error page (`server.error.path`) and so on.
- SSL
- HTTP compression

Spring Boot尽可能地尝试暴露常见设置，但这并非总是可行。 对于这些情况，专用名称空间提供特定于服务器的定制（请参阅`server.tomcat`和`server.undertow`）。 例如， access logs可以配置嵌入式servlet容器的特定功能。

请参阅`ServerProperties`类以获取完整列表。

Programmatic Customization译：程序化定制

如果你需要以编程方式配置你的嵌入式servlet容器，你可以注册一个实现了`WebServerFactoryCustomizer`接口的Spring bean。 `WebServerFactoryCustomizer`提供对`ConfigurableServletWebServerFactory`访问权限，其中包括许多定制设置方法。 以下示例以编程方式显示设置端口：

```
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);
 }

}
```

`TomcatServletWebServerFactory`， `JettyServletWebServerFactory`和`UndertowServletWebServerFactory`是专用变体`ConfigurableServletWebServerFactory`具有用于分别的Tomcat，码头和暗流额外定制setter方法。

Customizing ConfigurableServletWebServerFactory Directly译：直接自定义ConfigurableServletWebServerFactory

如果上述定制技术过于有限，则可以`TomcatServletWebServerFactory`注册`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安装程序。 如果您需要做更奇特的事情，还会提供一些受保护的方法“挂钩”。 有关详细信息，请参阅source code documentation 。

27.4.5 JSP Limitations 译：27.4.5 JSP限制

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

- With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when launched with `java -jar`, and will also be deployable to any standard container. JSPs are not supported when using an executable jar.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for error handling. Custom error pages should be used instead.

有一个 JSP sample，这样你就可以看到如何设置。

28. Security 译：28.安全

如果Spring Security位于类路径中，则默认情况下会保护Web应用程序。 Spring Boot依靠Spring Security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别的安全性，您还可以添加`@EnableGlobalMethodSecurity`和所需的设置。 其他信息可以在Spring Security Reference Guide找到。

默认`UserDetailsService`有一个用户。 用户名是`user` ，密码是随机的，并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
```

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

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是：

- A `UserDetailsService` (or `ReactiveUserDetailsService` in case of a WebFlux application) bean with in-memory store and a single user with a generated password (see `SecurityProperties.User` for the properties of the user).
- Form-based login or HTTP Basic security (depending on Content-Type) for the entire application (including actuator endpoints if actuator is on the classpath).
- A `DefaultAuthenticationEventPublisher` for publishing authentication events.

您可以通过 `AuthenticationEventPublisher`添加一个bean来提供不同的 `AuthenticationEventPublisher` 。

28.1 MVC Security 译：28.1 MVC安全性

默认安全配置在`SecurityAutoConfiguration`和`UserDetailsServiceAutoConfiguration` 。 `SecurityAutoConfiguration`进口`SpringBootWebSecurityConfiguration`的Web安全和`UserDetailsServiceAutoConfiguration`配置认证，这也是非Web应用程序相关。 要完全关闭默认Web应用程序安全配置，您可以添加类型为`WebSecurityConfigurerAdapter`的bean（这样做不会禁用`UserDetailsService`配置或Actuator的安全性）。

为了还关闭`UserDetailsService`配置，您可以添加类型的豆`UserDetailsService` ， `AuthenticationProvider` ，或`AuthenticationManager` 。 Spring Boot samples中有几个安全的应用程序让您开始使用常见用例。

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

28.2 WebFlux Security 译：28.2 WebFlux安全

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` 。 `ReactiveSecurityAutoConfiguration`进口`WebFluxSecurityConfiguration`的Web安全和`UserDetailsServiceAutoConfiguration`配置认证，这也是非Web应用程序相关。 要完全关闭默认Web应用程序安全配置，您可以添加类型为`WebFilterChainProxy`的Bean（这样做不会禁用`UserDetailsService`配置或Actuator的安全性）。

要关闭 `UserDetailsService`配置，可以添加类型为 `ReactiveUserDetailsService`或 `ReactiveAuthenticationManager`的bean。

访问规则可以通过添加自定义`SecurityWebFilterChain`进行配置。 Spring Boot提供了可用于覆盖执行器端点和静态资源的访问规则的便捷方法。 `EndpointRequest`可用于创建`ServerWebExchangeMatcher`那是基于`management.endpoints.web.base-path`财产。

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

例如，您可以通过添加如下内容来自定义您的安全配置：

```
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
 return http
  .authorizeExchange()
   .matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
   .pathMatchers("/foo", "/bar")
    .authenticated().and()
   .formLogin().and()
  .build();
}
```

28.3 OAuth2 译：28.3 OAuth2

OAuth2是Spring支持的广泛使用的授权框架。

28.3.1 Client 译：28.3.1客户端

如果您的类路径中有`spring-security-oauth2-client`，则可以利用一些自动配置来轻松设置OAuth2客户端。 该配置使用`OAuth2ClientProperties`下的属性。

您可以在`spring.security.oauth2.client`前缀下注册多个OAuth2客户端和提供者，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.jwk-set-uri=http://my-auth-server/token_keys
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
```

默认情况下，Spring Security的`OAuth2LoginAuthenticationFilter`只处理匹配`/login/oauth2/code/*`URL。 如果您想自定义`redirect-uri-template`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，您可以添加自己的`WebSecurityConfigurerAdapter`，类似于以下内容：

```
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

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

对于常见的OAuth2和OpenID提供商，包括谷歌，Github上，Facebook和1563，我们提供了一组供应商默认值（中`google`，`github`，`facebook`，并`okta`，分别）。

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`provider`属性。 另外，如果您的客户端的ID与默认支持的提供者相匹配，那么Spring Boot也会推断这一点。

换句话说，以下示例中的两种配置使用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
```

28.3.2 Server 译：28.3.2服务器

目前，Spring Security不支持实现OAuth 2.0授权服务器或资源服务器。 但是，该功能可从Spring Security OAuth项目获得，该项目最终将完全由Spring Security取代。 在此之前，您可以使用`spring-security-oauth2-autoconfigure`模块轻松设置OAuth 2.0服务器; 请参阅其documentation的说明。

28.4 Actuator Security 译：28.4执行器安全

出于安全考虑，`/health`和`/info`之外的所有执行器默认都是禁用的。 `management.endpoints.web.exposure.include`属性可用于启用执行器。

如果Spring Security位于类路径中，并且没有其他WebSecurityConfigurerAdapter存在，则执行器通过Spring Boot auto-config进行保护。 如果您定义了自定义`WebSecurityConfigurerAdapter`，Spring Boot自动配置将退出，您将完全控制执行器访问规则。

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/或通过将其放置在防火墙或Spring Security等类似设备上进行安全保护。

28.4.1 Cross Site Request Forgery Protection 译：28.4.1跨站请求伪造保护

由于Spring Boot依赖于Spring Security的默认设置，默认情况下，CSRF保护功能处于打开状态。 这意味着当默认安全配置正在使用时，需要`POST`（关机和记录器端点），`PUT`或`DELETE`的执行器端点将获得403禁止的错误。

我们建议只有在创建非浏览器客户端使用的服务时才能完全禁用CSRF保护。

有关CSRF保护的更多信息，请参阅 Spring Security Reference Guide 。

29. Working with SQL Databases译：29.使用SQL数据库

Spring Framework为使用SQL数据库提供了广泛的支持，从使用`JdbcTemplate`直接JDBC访问完成“对象关系映射”技术（如Hibernate）。 Spring Data提供了额外的功能级别：直接从接口创建`Repository`实现，并使用约定从您的方法名称生成查询。

29.1 Configure a DataSource译：29.1配置数据源

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

有关更高级的示例，请参见 the “How-to” section ，通常用于完全控制DataSource的配置。

29.1.1 Embedded Database Support译：29.1.1嵌入式数据库支持

通过使用内存中的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。

“How to to”部分包含 section on how to initialize a database 。

春天开机即可自动配置嵌入式H2， HSQL ，并Derby数据库。 您无需提供任何连接网址。 您只需要包含对要使用的嵌入式数据库的构建依赖关系。

如果您在测试中使用此功能，则可能会注意到无论您使用的应用程序上下文的数量如何，整个测试套件都会重复使用相同的数据库。 如果要确保每个上下文都有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依赖关系如下所示：

```
<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>
```

您需要依赖`spring-jdbc`来自动配置嵌入式数据库。 在这个例子中，它被传递到`spring-boot-starter-data-jpa` 。

如果出于某种原因，您配置了嵌入式数据库的连接URL，请注意确保数据库的自动关闭已禁用。 如果你使用H2，你应该使用`DB_CLOSE_ON_EXIT=FALSE`这样做。 如果您使用HSQLDB，则应确保不使用`shutdown=true` 。 禁用数据库的自动关闭功能可在数据库关闭时进行Spring Boot控制，从而确保在不再需要访问数据库时发生这种情况。

29.1.2 Connection to a Production Database译：29.1.2连接到生产数据库

生产数据库连接也可以通过使用池`DataSource`自动配置。 Spring Boot使用以下算法来选择特定的实现：

1. We prefer HikariCP for its performance and concurrency. If HikariCP is available, we always choose it.
2. Otherwise, if the Tomcat pooling `DataSource` is available, we use it.
3. If neither HikariCP nor the Tomcat pooling datasource are available and if Commons DBCP2 is available, we use it.

如果使用 `spring-boot-starter-jdbc`或 `spring-boot-starter-data-jpa`启动器”，则会自动获得 `HikariCP`的依赖关系。

您可以完全绕过该算法，并通过设置`spring.datasource.type`属性来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，因为默认情况下为`tomcat-jdbc` 。

其他连接池始终可以手动配置。 如果您定义了自己的`DataSource` bean，则不会发生自动配置。

数据源配置由`spring.datasource.*`外部配置属性`spring.datasource.*` 。 例如，您可以在`application.properties`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
```

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

您通常不需要指定 `driver-class-name` ，因为Spring Boot可以从 `url`大多数数据库中推导出它。

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，因此我们在做任何事之前检查它。 换句话说，如果你设置了`spring.datasource.driver-class-name=com.mysql.jdbc.Driver`，那么这个类必须是可加载的。

有关更多支持的选项，请参阅`DataSourceProperties`。 无论实际实施情况如何，这些都是标准选项。 它也有可能微调实现特定的设置，使用各自的前缀（`spring.datasource.hikari.*`，`spring.datasource.tomcat.*`，并`spring.datasource.dbcp2.*`）。 请参阅您正在使用的连接池实现的文档以获取更多详细信息。

例如，如果您使用 Tomcat connection pool，则可以自定义许多其他设置，如以下示例中所示：

```
# 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
```

29.1.3 Connection to a JNDI DataSource译：29.1.3连接到JNDI数据源

如果您将Spring Boot应用程序部署到应用程序服务器，则可能需要使用Application Server的内置功能配置和管理您的DataSource，并使用JNDI访问它。

所述`spring.datasource.jndi-name`属性可以被用作一个替代`spring.datasource.url`，`spring.datasource.username`和`spring.datasource.password`属性访问`DataSource`从特定JNDI位置。 例如，`application.properties`中的以下部分显示了如何访问定义为`DataSource`的JBoss AS：

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

29.2 Using JdbcTemplate译：29.2使用JdbcTemplate

Spring的`JdbcTemplate`和`NamedParameterJdbcTemplate`类是自动配置的，你可以直接将它们`@Autowire`放入你自己的bean中，如下例所示：

```
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`spring.jdbc.template.*`属性，如下例所示：

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

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

29.3 JPA and “Spring Data”译：29.3 JPA和“Spring Data”

Java持久性API是一种标准技术，可让您将“映射”对象转换为关系数据库。 `spring-boot-starter-data-jpa` POM提供了一种快速开始的方式。 它提供了以下关键依赖关系：

- Hibernate: One of the most popular JPA implementations.
- Spring Data JPA: Makes it easy to implement JPA-based repositories.
- Spring ORMs: Core ORM support from the Spring Framework.

我们在这里没有涉及太多的JPA或Spring Data的细节。 您可以按照“Accessing Data with JPA”指南从spring.io和阅读Spring Data JPA和Hibernate参考文档。

29.3.1 Entity Classes译：29.3.1实体类

传统上，JPA“Entity”类在`persistence.xml`文件中指定。 使用Spring Boot，这个文件不是必需的，而是使用“实体扫描”。 默认情况下，将搜索主配置类下的所有软件包（使用`@EnableAutoConfiguration`或`@SpringBootApplication`注释的`@SpringBootApplication`）。

带注释的所有类`@Entity`，`@Embeddable`，或`@MappedSuperclass`被考虑。 典型的实体类与以下示例类似：

```
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.country = country;
 }

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

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

 // ... etc

}
```

您可以使用`@EntityScan`批注自定义实体扫描位置。 请参阅“ Section 80.4, “Separate @Entity Definitions from Spring Configuration” ”的使用说明。

29.3.2 Spring Data JPA Repositories 译：29.3.2 Spring数据JPA存储库

Spring Data JPA存储库是可以定义用于访问数据的接口。 JPA查询是从您的方法名称自动创建的。 例如，一个`CityRepository`接口可能会声明一个`findAllByState(String state)`方法来查找给定状态下的所有城市。

对于更复杂的查询，你可以注释与Spring数据A€™的你的方法`Query`注释。

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

以下示例显示了一个典型的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 findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

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 reference documentation 。

29.3.3 Creating and Dropping JPA Databases 译：29.3.3创建和删除JPA数据库

默认情况下， **只有**在使用嵌入式数据库（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`application.properties`添加到`application.properties`：

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

Hibernate自己的内部属性名称（如果你碰巧记得更好）是`hibernate.hbm2ddl.auto`。 您可以使用`spring.jpa.properties.*`（其前缀在将其添加到实体管理器之前剥离）设置它以及其他Hibernate本机属性。 以下一行显示了为Hibernate设置JPA属性的示例：

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

上例中的`true`传递给Hibernate实体管理器。

默认情况下，DDL执行（或验证）被推迟到`ApplicationContext`启动。 还有一个`spring.jpa.generate-ddl`标志，但如果Hibernate自动配置处于活动状态，则不使用该标志，因为`ddl-auto`设置更加细化。

29.3.4 Open EntityManager in View译：29.3.4在View中打开EntityManager

如果您运行的是Web应用程序，默认情况下Spring Boot会注册`OpenEntityManagerInViewInterceptor`以在View中应用“Open EntityManager”模式，以允许在Web视图中进行延迟加载。 如果您不想要这种行为，则应在`spring.jpa.open-in-view` `false`设置为`application.properties` 。

29.4 Using H2’s Web Console译：29.4使用H2的Web控制台

H2 database提供了browser-based console ，Spring Boot可以为您自动配置。 满足以下条件时，控制台会自动配置：

- You are developing a servlet-based web application.
- `com.h2database:h2` is on the classpath.
- You are using Spring Boot’s developer tools.

如果您没有使用Spring Boot的开发工具，但仍希望使用H2的控制台，则可以使用值为`true`的`spring.h2.console.enabled`属性进行配置。

该H2控制台只能用于开发过程中使用，所以你应该小心，以确保`spring.h2.console.enabled`未设置为`true`的生产。

29.4.1 Changing the H2 Console’s Path译：29.4.1更改H2 Console的路径

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

29.5 Using jOOQ译：29.5使用jOOQ

面向Java对象查询（ jOOQ ）是一个广受欢迎的产品Data Geekery从数据库中生成的Java代码，让您通过构建其流畅的API类型安全的SQL查询。 商业和开源版本都可以与Spring Boot一起使用。

29.5.1 Code Generation译：29.5.1代码生成

为了使用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 user manual中的说明进行操作 。 如果您使用`jooq-codegen-maven`插件，您还可以使用`spring-boot-starter-parent` â€œparentPOMâ€，你可以放心地忽略pluginâ€™的`<version>`标签。 您也可以使用Spring Boot定义的版本变量（如`h2.version` ）来声明插件的数据库依赖关系。 以下列表显示了一个示例：

```
<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>
```

29.5.2 Using DSLContext译：29.5.2使用DSLContext

JOOQ提供的流畅的API是通过`org.jooq.DSLContext`接口启动的。 Spring Boot将`DSLContext`自动配置为Spring Bean，并将其连接到您的应用程序`DataSource` 。 要使用`DSLContext` ，您可以`@Autowire`它，如以下示例所示：

```
@Component
public class JooqExample implements CommandLineRunner {

 private final DSLContext create;

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

}
```

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);
}
```

29.5.3 jOOQ SQL Dialect译：29.5.3 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`。

Spring Boot只能自动配置jOOQ的开源版本支持的方言。

29.5.4 Customizing jOOQ译：29.5.4定制jOOQ

更高级的自定义可以通过定义自己的`@Bean`定义来实现，该定义在创建jOOQ `Configuration`时使用。 您可以为以下jOOQ类型定义bean：

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

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

30. Working with NoSQL Technologies译：30.与NoSQL技术合作

春天的数据提供了帮助您使用各种不同的NoSQL技术，包括增发项目： 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，但您必须自己配置它们。 请参阅相应的参考文档projects.spring.io/spring-data。

30.1 Redis译：30.1 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，并为其提供了Spring Data Redis以上的抽象。

有一个`spring-boot-starter-data-redis` Starter”以便捷的方式收集依赖关系。 默认情况下，它使用Lettuce。 该入门者可以处理传统和反应式应用程序。

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

30.1.1 Connecting to Redis译：30.1.1连接到Redis

你可以注入的自动配置`RedisConnectionFactory`， `StringRedisTemplate`，或香草`RedisTemplate`，就像任何其他的Spring Bean实例。 默认情况下，该实例尝试连接到位于`localhost:6379`的Redis服务器。 下面的列表显示了这样一个bean的例子：

```
@Component
public class MyBean {

 private StringRedisTemplate template;

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

 // ...

}
```

您还可以注册任意数量的实现更高级自定义的`LettuceClientConfigurationBuilderCustomizer`的bean。 如果你使用Jedis，`JedisClientConfigurationBuilderCustomizer`也可用。

如果您添加自己的`@Bean`的任何自动配置类型，它将替换默认值（ `RedisTemplate`的情况`RedisTemplate`，此时排除基于bean名称`redisTemplate`，而不是其类型）。 默认情况下，如果`commons-pool2`位于类路径中，则会得到一个池连接工厂。

30.2 MongoDB译：30.2 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格的关系数据。 Spring Boot为MongoDB提供了一些便利，包括`spring-boot-starter-data-mongodb`和`spring-boot-starter-data-mongodb-reactive`启动器”。

30.2.1 Connecting to a MongoDB Database译：30.2.1连接到MongoDB数据库

要访问Mongo数据库，可以注入一个自动配置的`org.springframework.data.mongodb.MongoDbFactory`。 默认情况下，实例尝试连接到`mongodb://localhost/test`的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`属性来更改URL并配置其他设置（如 *副本集*），如下例所示：

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

另外，只要您使用蒙戈2.x中，你可以指定一个`host`/`port`。 例如，您可以在`application.properties`声明以下设置：

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

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，应该使用`spring.data.mongodb.uri`来提供所有配置。

如果`spring.data.mongodb.port`没有指定，默认`27017`使用。 您可以从前面的示例中删除此行。

如果您不使用Spring Data Mongo，则可以注入`com.mongodb.MongoClient`豆而不是使用`MongoDbFactory`。 如果你想完全控制建立MongoDB连接，你也可以声明你自己的`MongoDbFactory`或`MongoClient` bean。

如果您使用的是反应驱动程序，则Netty是SSL所必需的。 如果Netty可用且工厂使用尚未定制，则自动配置会自动配置该工厂。

30.2.2 MongoTemplate 译：30.2.2 MongoTemplate

Spring Data MongoDB提供了一个`MongoTemplate`类，在其设计Springâ€™的非常相似`JdbcTemplate`。 和`JdbcTemplate`，Spring Boot会自动配置一个bean来注入模板，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。

30.2.3 Spring Data MongoDB Repositories 译：30.2.3 Spring Data MongoDB存储库

Spring Data包含对MongoDB的存储库支持。 与前面讨论的JPA存储库一样，基本原则是查询是基于方法名称自动构建的。

实际上，Spring Data JPA和Spring Data MongoDB共享相同的通用基础结构。 你可以从之前的JPA例子中，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 findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

您可以使用`@EntityScan`注释来自定义文档扫描位置。

有关Spring Data MongoDB的完整详细信息，包括丰富的对象映射技术，请参阅其 reference documentation 。

30.2.4 Embedded Mongo译：30.2.4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`de.flapdoodle.embed:de.flapdoodle.embed.mongo` 。

Mongo监听的端口可以通过设置`spring.data.mongodb.port`属性进行配置。 要使用随机分配的空闲端口，请使用值0.由`MongoClient`创建的`MongoAutoConfiguration`会自动配置为使用随机分配的端口。

如果您未配置自定义端口，则默认情况下嵌入式支持使用随机端口（而不是27017）。

如果在类路径中有SLF4J，则Mongo生成的输出将自动路由到名为`org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo`的记录器。

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

30.3 Neo4j译：30.3 Neo4j

Neo4j是一个开放源码的NoSQL图形数据库，它使用由一级关系关联的节点的丰富数据模型，它比传统的rdbms方法更适合于连接大数据。 Spring Boot为使用Neo4j提供了一些便利，包括`spring-boot-starter-data-neo4j` Starter”。

30.3.1 Connecting to a Neo4j Database译：30.3.1连接到Neo4j数据库

你可以注入的自动配置`Neo4jSession` ， `Session` ，或`Neo4jOperations`实例，就像任何其他的Spring bean。 默认情况下，该实例尝试连接到`localhost:7474`的Neo4j服务器。 以下示例显示如何注入Neo4j bean：

```
@Component
public class MyBean {

 private final Neo4jTemplate neo4jTemplate;

 @Autowired
 public MyBean(Neo4jTemplate neo4jTemplate) {
  this.neo4jTemplate = neo4jTemplate;
 }

 // ...

}
```

您可以通过添加自己的`org.neo4j.ogm.config.Configuration` `@Bean`来完全控制配置。 此外，加入`@Bean`类型的`Neo4jOperations`禁用自动配置。

您可以通过设置`spring.data.neo4j.*`属性来配置要使用的用户和凭据，如下例所示：

```
spring.data.neo4j.uri=http://my-server:7474
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=secret
```

30.3.2 Using the Embedded Mode译：30.3.2使用嵌入模式

如果您将`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。 您可以通过设置`spring.data.neo4j.embedded.enabled=false`来明确禁用该模式。 您还可以通过提供数据库文件的路径来为嵌入模式启用持久性，如以下示例所示：

```
 spring.data.neo4j.uri=file://var/tmp/graph.db
```

Neo4j OGM嵌入式驱动程序不提供Neo4j内核。 用户需要手动提供这种依赖关系。 有关更多详细信息，请参阅the documentation 。

30.3.3 Neo4jSession译：30.3.3 Neo4j Session

默认情况下，如果您正在运行Web应用程序，则该会话将绑定到整个请求处理的线程（即，它使用“在会话中打开会话”模式）。 如果您不想要这种行为，请将以下行添加到`application.properties`文件中：

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

30.3.4 Spring Data Neo4j Repositories 译：30.3.4 Spring Data Neo4j存储库

Spring Data包含Neo4j的存储库支持。

事实上，Spring Data JPA和Spring Data Neo4j共享相同的通用基础架构。 您可以从之前的JPA示例中获得，并假设`City`现在是Neo4j OGM `@NodeEntity`而不是JPA `@Entity`，它的工作方式相同。

您可以使用`@EntityScan`批注自定义实体扫描位置。

要启用存储库支持（并可选择支持`@Transactional`），请将以下两个注释添加到您的Spring配置中：

```
@EnableNeo4jRepositories(basePackages = "com.example.myapp.repository")
@EnableTransactionManagement
```

30.3.5 Repository Example 译：30.3.5存储库示例

以下示例显示了Neo4j存储库的接口定义：

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

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

public interface CityRepository extends GraphRepository<City> {

 Page<City> findAll(Pageable pageable);

 City findByNameAndCountry(String name, String country);

}
```

有关Spring Data Neo4j的完整详细信息，包括丰富的对象映射技术，请参阅 reference documentation 。

30.4 Gemfire 译：30.4 Gemfire

Spring Data 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”以方便的方式收集依赖关系。 目前Gemfire没有自动配置支持，但您可以使用single annotation: `@EnableGemfireRepositories`启用Spring Data Repositories。

30.5 Solr 译：30.5 Solr

Apache Solr是一个搜索引擎。 Spring Boot为Solr 5客户端库以及Spring Data Solr提供的抽象类提供了基本的自动配置。 有一个`spring-boot-starter-data-solr` Starter”以便捷的方式收集依赖关系。

30.5.1 Connecting to Solr 译：30.5.1连接到Solr

您可以注入一个自动配置的`SolrClient`实例，就像其他任何Spring bean一样。 默认情况下，该实例尝试连接到服务器`localhost:8983/solr`。 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;
 }

 // ...

}
```

如果您添加自己`@Bean`型`SolrClient`，它取代了默认。

30.5.2 Spring Data Solr Repositories 译：30.5.2 Spring Data Solr存储库

Spring Data包含Apache Solr的存储库支持。 和前面讨论的JPA库一样，基本原则是查询是根据方法名自动为\ you构建的。

事实上，Spring Data JPA和Spring Data Solr都共享相同的通用基础结构。 你可以从之前的JPA例子中，假设`City`现在是`@SolrDocument`类而不是JPA `@Entity`，它的工作方式是相同的。

有关Spring Data Solr的完整细节，请参阅 reference documentation 。

30.6 Elasticsearch 译：30.6 Elasticsearch

Elasticsearch是一个开源的，分布式的实时搜索和分析引擎。 Spring Boot为Elasticsearch提供基本的自动配置，并为其提供Spring Data Elasticsearch的抽象。 有一个`spring-boot-starter-data-elasticsearch` Starter”以便捷的方式收集依赖关系。 Spring Boot还支持Jest 。

30.6.1 Connecting to Elasticsearch by Using Jest 译：30.6.1使用Jest连接到Elasticsearch

如果您有`Jest`在类路径中，你可以注入的自动配置`JestClient`，通过默认目标`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
```

您还可以注册任意数量的实现更高级自定义的`HttpClientConfigBuilderCustomizer`的bean。 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，请定义一个`JestClient` bean。

30.6.2 Connecting to Elasticsearch by Using Spring Data译：30.6.2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`属性设置为以逗号分隔的`host:port`列表来指定地址。 使用此配置，可以像其他任何Spring bean一样注入`ElasticsearchTemplate`或`TransportClient`，如以下示例所示：

```
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`或`@Bean`，它将替换默认值。

30.6.3 Spring Data Elasticsearch Repositories译：30.6.3 Spring数据Elasticsearch存储库

Spring Data包含Elasticsearch的存储库支持。 与前面讨论的JPA存储库一样，基本原则是查询是基于方法名称自动为您构建的。

实际上，Spring Data JPA和Spring Data Elasticsearch共享相同的通用基础结构。 你可以从之前的JPA例子中，假设`City`现在是一个Elasticsearch `@Document`类而不是JPA `@Entity`，它的工作原理是一样的。

有关Spring Data Elasticsearch的完整详细信息，请参阅 reference documentation 。

30.7 Cassandra译：30.7卡桑德拉

Cassandra是一个开源的分布式数据库管理系统，旨在处理大量商品服务器上的大量数据。 Spring Boot为Cassandra提供自动配置，并为其提供Spring Data Cassandra的抽象。 有一个`spring-boot-starter-data-cassandra` Starter”以便捷的方式收集依赖关系。

30.7.1 Connecting to Cassandra译：30.7.1连接到Cassandra

您可以像使用其他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，如以下示例中所示：

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

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;
 }

 // ...

}
```

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

30.7.2 Spring Data Cassandra Repositories译：30.7.2 Spring Data Cassandra存储库

Spring Data包含对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`注释查找方法。

有关Spring Data Cassandra的完整详细信息，请参阅 reference documentation 。

30.8 Couchbase 译：30.8 Couchbase

Couchbase是一款面向交互式应用程序优化的开源，分布式，多模型NoSQL面向文档的数据库。 Spring Boot提供了Couchbase的自动配置和Spring Data Couchbase提供的抽象 。 有`spring-boot-starter-data-couchbase`和`spring-boot-starter-data-couchbase-reactive`启动器”以便捷的方式收集依赖关系。

30.8.1 Connecting to Couchbase 译：30.8.1连接到Couchbase

您可以通过添加Couchbase SDK和某些配置来获得`Bucket`和`Cluster` 。 `spring.couchbase.*`属性可用于自定义连接。 通常，您提供引导程序主机，存储区名称和密码，如以下示例中所示：

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

您*至少*需要提供引导主机，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`并启用SSL支持的超时时间：

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

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

30.8.2 Spring Data Couchbase Repositories 译：30.8.2 Spring Data Couchbase存储库

Spring Data包含Couchbase的存储库支持。 有关Spring Data Couchbase的完整详细信息，请参阅reference documentation 。

您可以像使用其他Spring Bean一样注入一个自动配置的`CouchbaseTemplate`实例，前提是 *默认的* `CouchbaseConfigurer`可用（当您启用Couchbase支持时会发生，如前所述）。

以下示例显示如何注入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几个bean，以覆盖由自动配置提供的那些bean：

- A `CouchbaseTemplate` `@Bean` with a name of `couchbaseTemplate`.
- An `IndexManager` `@Bean` with a name of `couchbaseIndexManager`.
- A `CustomConversions` `@Bean` with a name of `couchbaseCustomConversions`.

为避免在自己的配置中对这些名称进行硬编码，可以重复使用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(...);
 }

 // ...

}
```

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

30.9 LDAP 译：30.9 LDAP

LDAP （轻量级目录访问协议）是一种开放的，厂商中立的行业标准应用协议，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，并支持来自UnboundID的嵌入式内存LDAP服务器。

LDAP抽象由Spring Data LDAP提供。 有一个`spring-boot-starter-data-ldap` Starter”以方便的方式收集依赖关系。

30.9.1 Connecting to an LDAP Server 译：30.9.1连接到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
```

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

30.9.2 Spring Data LDAP Repositories 译：30.9.2 Spring数据LDAP存储库

Spring Data包含对LDAP的存储库支持。 有关Spring Data LDAP的完整详细信息，请参阅reference documentation 。

您也可以像使用其他Spring Bean一样注入一个自动配置的 `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;
 }

 // ...

}
```

30.9.3 Embedded In-memory LDAP Server 译：30.9.3嵌入式内存中LDAP服务器

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

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

可以定义多个base-dn值，但是，由于专有名称通常包含逗号，因此必须使用正确的记号来定义它们。
在yaml文件中，您可以使用yaml列表符号：

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

在属性文件中，您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。 没有必要指定 `spring.ldap.urls` 属性。

如果您的类路径中有 `schema.ldif` 文件，它将用于初始化服务器。 如果要从其他资源加载初始化脚本，则还可以使用 `spring.ldap.embedded.ldif` 属性。

默认情况下，使用标准架构来验证 `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。

30.10 InfluxDB 译：30.10 InfluxDB

InfluxDB是一款开源时间序列数据库，针对操作监控，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。

30.10.1 Connecting to InfluxDB 译：30.10.1连接到InfluxDB

Spring Boot会自动配置 `InfluxDB` 实例，前提是 `influxdb-java` 客户端位于类路径上并设置了数据库的URL，如以下示例所示：

```
spring.influx.url=http://172.0.0.1:8086
```

如果与InfluxDB的连接需要用户名和密码，则可以相应地设置 `spring.influx.user` 和 `spring.influx.password` 属性。

InfluxDB依赖于OkHttp。 如果您需要调整 `InfluxDB` 使用的http客户端 `InfluxDB` ，则可以注册一个 `OkHttpClient.Builder` bean。

31. Caching 译：31.缓存

Spring框架提供了对应用程序透明地添加缓存的支持。 其核心是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。 缓存逻辑是透明应用的，不会对调用者产生任何干扰。 只要通过 `@EnableCaching` 批注启用了缓存支持，Spring Boot就会自动配置缓存基础结构。

查看Spring Framework参考的 relevant section以获取更多详细信息。

简而言之，将缓存添加到服务的操作中非常简单，只需将相关注释添加到其方法中即可，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) {
  // ...
 }

}
```

这个例子演示了如何在一个潜在的昂贵操作上使用缓存。 在调用`computePiDecimal`之前，抽象查找`piDecimals`缓存中与`i`参数匹配的`i`。 如果找到条目，则缓存中的内容立即返回给调用者，并且不调用该方法。 否则，将调用该方法，并在返回值之前更新缓存。

> **Caution**
>
> 您也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合使用Spring Cache和JCache注释。

如果您不添加任何特定的缓存库，Spring Boot将自动配置在内存中使用并发映射的simple provider。 当需要缓存时（例如上例中的`piDecimals`），此提供程序会为您创建该缓存。 简单的提供者并不是真正推荐用于生产用途，但它对于入门和确保理解这些功能非常有用。 当您决定使用缓存提供程序时，请确保阅读其文档以了解如何配置应用程序使用的缓存。 几乎所有的提供程序都要求您明确配置您在应用程序中使用的每个缓存。 有些提供了一种自定义由`spring.cache.cache-names`属性定义的默认缓存的`spring.cache.cache-names`。

> 透明地缓存 update或 evict数据也是可能的。

31.1 Supported Cache Providers译：31.1支持的缓存提供程序

缓存抽象不提供实际的存储，并依赖于`org.springframework.cache.Cache`和`org.springframework.cache.CacheManager`接口实现的抽象。

如果你还没有定义类型的豆`CacheManager`或者`CacheResolver`命名为`cacheResolver`（见`CachingConfigurer`），弹簧引导尝试检测以下提供商（在指定的顺序）：

1. Generic
2. JCache (JSR-107) (EhCache 3, Hazelcast, Infinispan, and others)
3. EhCache 2.x
4. Hazelcast
5. Infinispan
6. Couchbase
7. Redis
8. Caffeine
9. Simple

> 也可以通过设置`spring.cache.type`属性来*强制*某个缓存提供者。 如果您在某些环境（如测试）中需要disable caching altogether，请使用此属性。

> 使用`spring-boot-starter-cache`启动器"快速添加基本的缓存依赖关系。 首发`spring-context-support`。 如果您手动添加依赖项，则必须包含`spring-context-support`才能使用JCache，EhCache 2.x或Guava支持。

如果`CacheManager`是Spring Boot自动配置的，那么可以通过公开一个实现了`CacheManagerCustomizer`接口的bean来完全初始化它的配置。 以下示例将一个标志设置为空值应传递给底层映射：

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
 return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {
  @Override
  public void customize(ConcurrentMapCacheManager cacheManager) {
   cacheManager.setAllowNullValues(false);
  }
 };
}
```

> 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者自动配置了不同的缓存提供程序），则根本不调用定制程序。 您可以根据需要拥有尽可能多的定制器，也可以使用`@Order`或`Ordered`来`@Order` `Ordered`。

31.1.1 Generic译：31.1.1通用

如果上下文定义*了至少一个*`org.springframework.cache.Cache` bean，则使用通用缓存。 包装该类型的所有bean的`CacheManager`被创建。

31.1.2 JCache (JSR-107)译：31.1.2 JCache（JSR-107）

JCache通过类路径上存在的`javax.cache.spi.CachingProvider`（即，类路径中存在符合JSR-107的高速缓存库）进行`JCacheCacheManager`，`JCacheCacheManager`由`spring-boot-starter-cache` Starter"提供。 各种兼容的库可用，Spring Boot为Ehcache 3，Hazelcast和Infinispan提供依赖管理。 任何其他兼容库也可以添加。

可能会出现多个提供者存在的情况，在这种情况下，必须明确指定提供者。 即使JSR-107标准没有强制规定配置文件位置的标准方式，Spring Boot也会尽力为缓存设置实现细节，如下例所示：

```
  # Only necessary if more than one provider is present
spring.cache.jcache.provider=com.acme.MyCachingProvider
spring.cache.jcache.config=classpath:acme.xml
```

当缓存库提供本机实现和JSR-107支持时，Spring Boot更倾向于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。

Spring Boot有general support for Hazelcast 。 如果有一个`HazelcastInstance`可用，则它也会自动重新用于`CacheManager`，除非指定了`spring.cache.jcache.config`属性。

有两种方法可以自定义底层`javax.cache.cacheManager`：

- Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `javax.cache.configuration.Configuration` bean is defined, it is used to customize them.
- `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` beans are invoked with the reference of the `CacheManager` for full customization.

如果定义了标准的`javax.cache.CacheManager` bean，它将自动包装在抽象预期的`org.springframework.cache.CacheManager`实现中。 没有进一步的定制应用于它。

31.1.3 EhCache 2.x译：31.1.3 EhCache 2.x

EhCache如果可以在类路径的根目录找到名为`ehcache.xml`的文件，则使用`ehcache.xml` 如果找到EhCache 2.x，则使用`EhCacheCacheManager`提供的`spring-boot-starter-cache`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
```

31.1.4 Hazelcast译：31.1.4 Hazelcast

Spring Boot有general support for Hazelcast 。 如果`HazelcastInstance`已被自动配置，则会自动将其包装在`CacheManager`。

31.1.5 Infinispan译：31.1.5 Infinispan

Infinispan没有默认配置文件位置，因此必须明确指定。 否则，使用默认的引导程序。

```
spring.cache.infinispan.config=infinispan.xml
```

通过设置`spring.cache.cache-names`属性，可以在启动时创建缓存。 如果定义了定制的`ConfigurationBuilder` bean，它将用于定制缓存。

Infinispan在Spring Boot中的支持仅限于嵌入式模式，并且非常基础。 如果你想要更多的选择，你应该使用官方的Infinispan Spring Boot启动器。 有关更多详细信息，请参阅Infinispan's documentation 。

31.1.6 Couchbase译：31.1.6 Couchbase

如果Couchbase Java客户端和`couchbase-spring-cache`实现可用，且Couchbase为configured ，则自动配置`CouchbaseCacheManager` 。 通过设置`spring.cache.cache-names`属性，也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。 您还可以创建另一个附加的高速缓存`Bucket`通过使用定制。 假设你需要两个缓存（ `cache1`和`cache2`的"主"） `Bucket`和一个（ `cache3` ）高速缓存使用自定义的时间在A€œanotherâ€住2秒`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，您可以定义一个`@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` 。

31.1.7 Redis译：31.1.7 Redis

如果Redis可用且已配置，则自动配置`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`高速缓存， *生存时间为* 10分钟：

```
spring.cache.cache-names=cache1,cache2
spring.cache.redis.time-to-live=600000
```

默认情况下，会添加一个键前缀，这样，如果两个单独的缓存使用相同的键，则Redis不会有重叠的键并且无法返回无效值。 我们强烈建议您在创建自己的`RedisCacheManager`保持启用此设置。

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。

31.1.8 Caffeine译：31.1.8咖啡因

Caffeine是一个Java 8重写的Guava缓存，取代了对Guava的支持。 如果存在咖啡因，则会自动配置`CaffeineCacheManager` （由“`spring-boot-starter-cache` spring-boot-starter-cache提供）。 通过设置`spring.cache.cache-names`属性，可以在启动时创建缓存，并可以通过以下任一项（按指定顺序）进行自定义：

1. A cache spec defined by `spring.cache.caffeine.spec`
2. A `com.github.benmanes.caffeine.cache.CaffeineSpec` bean is defined
3. A `com.github.benmanes.caffeine.cache.Caffeine` bean is defined

例如，以下配置会创建最大大小为500的`cache1`和`cache2`高速缓存，并且 *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，它将自动关联到`CaffeineCacheManager`。 由于`CacheLoader`将与缓存管理器管理的 *所有* 缓存关联，因此它必须定义为`CacheLoader<Object, Object>`。 自动配置忽略任何其他泛型类型。

31.1.9 Simple译：31.1.9简单

如果找不到任何其他提供者，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，根据需要创建缓存，但可以通过设置`cache-names`属性来限制可用缓存的列表。 例如，如果只需要`cache1`和`cache2`高速缓存，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，那么它在运行时需要缓存时会失败，但不会在启动时缓存。 这与“真实”缓存提供程序在使用未声明的缓存时的行为方式类似。

31.1.10 None译：31.1.10无

当您的配置中存在`@EnableCaching`，预计也会有合适的缓存配置。 如果您需要在某些环境中完全禁用缓存，则强制缓存类型为`none`以使用no-op实现，如下例所示：

```
spring.cache.type=none
```

32. Messaging译：32.消息

Spring框架为集成消息传递系统提供了广泛的支持，从简化使用JMS API（使用`JmsTemplate`到完整的基础结构以异步接收消息。 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本身就包含对STOMP消息传递的支持，Spring Boot通过启动器和少量的自动配置提供支持。 Spring Boot也支持Apache Kafka。

32.1 JMS译：32.1 JMS

`javax.jms.ConnectionFactory`界面提供了创建用于与JMS代理进行交互的`javax.jms.Connection`的标准方法。 虽然Spring需要`ConnectionFactory`才能使用JMS，但您通常不需要直接使用它，而是可以依赖更高级别的消息抽象。 （有关详细信息，请参阅Spring Framework参考文档的relevant section。 ）Spring Boot还自动配置发送和接收消息的必要基础结构。

32.1.1 ActiveMQ Support译：32.1.1 ActiveMQ支持

当类路径中有ActiveMQ可用时，Spring Boot还可以配置`ConnectionFactory`。 如果代理存在，则会自动启动并配置嵌入式代理（如果未通过配置指定代理URL）。

如果使用`spring-boot-starter-activemq`，则提供连接或嵌入ActiveMQ实例所需的依赖关系，就像Spring基础结构要与JMS集成一样。

ActiveMQ配置由`spring.activemq.*`外部配置属性`spring.activemq.*`。 例如，您可以在`application.properties`声明以下部分：

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

您还可以通过将相关性添加到`org.apache.activemq:activemq-pool`并相应地配置`PooledConnectionFactory`来池化JMS资源，如以下示例所示：

```
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

有关更多受支持的选项，请参阅`ActiveMQProperties`。 您还可以注册任意数量的实现更高级定制的`ActiveMQConnectionFactoryCustomizer`的bean。

默认情况下，如果ActiveMQ尚不存在，ActiveMQ将创建一个目标，以便按照其提供的名称解析目标。

32.1.2 Artemis Support 译：32.1.2 Artemis支持

当Spring Boot检测到Artemis在类路径中可用时，它可以自动配置`ConnectionFactory`。 如果代理存在，将自动启动并配置嵌入式代理（除非已明确设置mode属性）。 支持的模式为`embedded`（为了明确说明需要嵌入式代理，并且在代理不在类路径中可用时发生错误）和`native`（使用`netty`传输协议连接到代理）。 当配置后者时，Spring Boot会配置一个`ConnectionFactory`，它使用默认设置连接到本地计算机上运行的代理。

如果使用`spring-boot-starter-artemis`，则必须提供连接到现有Artemis实例的必要依赖关系，并提供Spring基础结构以与JMS集成。 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.*`外部配置属性`spring.artemis.*`。 例如，您可以在`application.properties`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，可以选择是否启用持久性并列出应该可用的目标。 可以将它们指定为以逗号分隔的列表以使用默认选项创建它们，也可以分别为高级队列和主题配置定义类型
为`org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration`或`org.apache.activemq.artemis.jms.server.config.TopicConfiguration` bean。

有关更多支持的选项，请参阅`ArtemisProperties`。

不涉及JNDI查找，并且使用Artemis配置中的`name`属性或通过配置提供的名称，针对其名称解析目标。

32.1.3 Using a JNDI ConnectionFactory 译：32.1.3使用JNDI ConnectionFactory

如果您正在应用程序服务器中运行应用程序，Spring Boot会尝试使用JNDI查找JMS `ConnectionFactory`。 默认情况下，检查`java:/JmsXA`和`java:/XAConnectionFactory`位置。 如果您需要指定替代位置，则可以使用`spring.jms.jndi-name`属性，如下例所示：

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

32.1.4 Sending a Message 译：32.1.4发送消息

Spring的`JmsTemplate`是自动配置的，您可以直接将它自动装载到您自己的bean中，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;
 }

 // ...

}
```

`JmsMessagingTemplate`可以以类似的方式注入。 如果定义了`DestinationResolver`或`MessageConverter` bean，它将自动关联到自动配置的`JmsTemplate`。

32.1.5 Receiving a Message 译：32.1.5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。 如果没有定义`JmsListenerContainerFactory`，则会自动配置默认值。 如果定义了`DestinationResolver`或`MessageConverter` bean，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，则默认情况下它将与侦听器容器关联。 如果不是，则`sessionTransacted`标志被启用。 在后一种情况下，可以通过在侦听器方法（或其`@Transactional`上添加`@Transactional`，将本地数据存储事务与传入消息的处理关联起来。 这确保了一旦本地事务完成，传入的消息就会被确认。 这还包括发送已在相同的JMS会话上执行的响应消息。

以下组件在`someQueue`目标上创建侦听器端点：

```
@Component
public class MyBean {

 @JmsListener(destination = "someQueue")
 public void processMessage(String content) {
  // ...
 }

}
```

详情请参阅 the Javadoc of `@EnableJms`。

如果您需要创建更多`JmsListenerContainerFactory`实例，或者您想要覆盖默认值，则Spring Boot会提供`DefaultJmsListenerContainerFactoryConfigurer`，您可以使用它来使用与自动配置相同的设置初始化`DefaultJmsListenerContainerFactory`。

例如，以下示例公开了使用特定 `MessageConverter` 另一个工厂：

```
@Configuration
static class JmsConfiguration {

 @Bean
 public DefaultJmsListenerContainerFactory myFactory(
   DefaultJmsListenerContainerFactoryConfigurer configurer) {
  DefaultJmsListenerContainerFactory factory =
    new DefaultJmsListenerContainerFactory();
  configurer.configure(factory, connectionFactory());
  factory.setMessageConverter(myMessageConverter());
  return factory;
 }

}
```

然后你可以在任何 `@JmsListener` 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) {
  // ...
 }

}
```

32.2 AMQP 译：32.2 AMQP

高级消息队列协议（AMQP）是面向消息中间件的平台中立的有线协议。 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通过RabbitMQ为AMQP提供了一些便利，包括 `spring-boot-starter-amqp` Starter”。

32.2.1 RabbitMQ support 译：32.2.1 RabbitMQ支持

RabbitMQ是一款基于AMQP协议的轻量级，可靠，可扩展，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.*` 外部配置属性 `spring.rabbitmq.*` 。 例如，您可以在 `application.properties` 声明以下部分：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

如果上下文中存在一个 `ConnectionNameStrategy` bean，它将自动用于命名由自动配置的 `ConnectionFactory` 创建的连接。 有关更多受支持的选项，请参见 `RabbitProperties` 。

有关更多详细信息，请参阅 Understanding AMQP, the protocol used by RabbitMQ 。

32.2.2 Sending a Message 译：32.2.2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载到您自己的bean中，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;
 }

 // ...

}
```

`RabbitMessagingTemplate` 可以以类似的方式注入。 如果定义了 `MessageConverter` bean，它将自动关联到自动配置的 `AmqpTemplate` 。

如有必要，定义为bean的任何 `org.springframework.amqp.core.Queue` 将自动用于在RabbitMQ实例上声明相应的队列。

要重试操作，您可以在 `AmqpTemplate` 上启用重试（例如，在代理连接丢失的情况下）。 重试是默认禁用的。

32.2.3 Receiving a Message 译：32.2.3接收消息

当存在Rabbit基础结构时，可以使用 `@RabbitListener` 注释任何bean以创建侦听器端点。 如果没有 `RabbitListenerContainerFactory` 已经定义，默认 `SimpleRabbitListenerContainerFactory` 自动配置，您可以使用切换到直接容器 `spring.rabbitmq.listener.type` 财产。 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` 队列上创建侦听器端点：

```
@Component
public class MyBean {

 @RabbitListener(queues = "someQueue")
 public void processMessage(String content) {
  // ...
 }

}
```

有关更多详细信息，请参阅 the Javadoc of `@EnableRabbit` 。

如果您需要创建更多 `RabbitListenerContainerFactory` 实例或者想要覆盖默认值，Spring Boot提供了 `SimpleRabbitListenerContainerFactoryConfigurer` 和 `DirectRabbitListenerContainerFactoryConfigurer` ，您可以使用它们来初始化 `SimpleRabbitListenerContainerFactory` 和 `DirectRabbitListenerContainerFactory` ，其设置与自动配置使用的工厂相同。

您选择哪种容器类型并不重要。 这两个bean通过自动配置暴露出来。

例如，以下配置类公开了使用特定 `MessageConverter` 另一个工厂：

```
@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，您可以使用任何 `@RabbitListener` -annotated方法中的工厂，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引发异常的情况。 默认情况下，使用 `RejectAndDontRequeueRecoverer` ，但您可以定义自己的 `MessageRecoverer` 。 当重试耗尽时，如果代理被配置为这样，则该消息被拒绝并丢弃或路由到死信交换。 默认情况下，重试被禁用。

Important

默认情况下，如果重试被禁用并且侦听器引发异常，则传递将无限期地重试。 你可以用两种方法修改此行为：设置 `defaultRequeueRejected` 属性为 `false` ，使零再交货企图或抛出一个 `AmqpRejectAndDontRequeueException` 信号的消息应该被拒绝。 后者是启用重试并达到最大传送尝试次数时使用的机制。

32.3 Apache Kafka Support 译：32.3 Apache Kafka支持

Apache Kafka通过提供的自动配置支持 `spring-kafka` 项目。

卡夫卡配置由外部配置属性 `spring.kafka.*` 。 例如，您可以在 `application.properties` 声明以下部分：

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

要在启动时创建主题，请添加类型为 `NewTopic` 的bean。 如果该主题已经存在，则该bean将被忽略。

有关更多支持的选项，请参阅 `KafkaProperties` 。

32.3.1 Sending a Message 译：32.3.1发送消息

Spring的 `KafkaTemplate` 是自动配置的，您可以直接在自己的bean中自动配置它，如以下示例所示：

```
@Component
public class MyBean {

 private final KafkaTemplate kafkaTemplate;

 @Autowired
 public MyBean(KafkaTemplate kafkaTemplate) {
  this.kafkaTemplate = kafkaTemplate;
 }

 // ...

}
```

如果定义了一个 `RecordMessageConverter` bean，它将自动关联到自动配置的 `KafkaTemplate` 。

32.3.2 Receiving a Message译：32.3.2接收消息

当存在Apache Kafka基础结构时，可以使用`@KafkaListener`来注释任何bean以创建侦听器端点。 如果没有`KafkaListenerContainerFactory`已定义，一个默认自动地与在定义的键配置`spring.kafka.listener.*` 。 另外，如果定义了`RecordMessageConverter` bean，它将自动关联到默认工厂。

以下组件在 `someTopic` 主题上创建侦听器端点：

```
@Component
public class MyBean {

 @KafkaListener(topics = "someTopic")
 public void processMessage(String content) {
  // ...
 }

}
```

32.3.3 Additional Kafka Properties译：32.3.3其他卡夫卡属性

自动配置支持的属性显示在Appendix A, *Common application properties*中 。 请注意，大多数情况下，这些属性（连字符或camelCase）直接映射到Apache Kafka虚线属性。 有关详细信息，请参阅Apache Kafka文档。

前几个属性适用于生产者和消费者，但如果您希望为每个生产者和消费者使用不同的值，可以在生产者或消费者级别指定。 Apache Kafka指定具有HIGH，MEDIUM或LOW重要性的属性。 Spring Boot自动配置支持所有HIGH重要属性，一些选定的MEDIUM和LOW属性以及任何没有默认值的属性。

通过`KafkaProperties`类只能获得Kafka支持的属性子集。 如果您希望使用不直接支持的其他属性来配置生产者或消费者，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
```

这台共同 `prop.one`卡夫卡属性为 `first`的（适用于生产者，消费者和管理员）， `prop.two`管理财产 `second` ，该 `prop.three`消费属性 `third`和 `prop.four`制片属性 `fourth` 。

您也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
```

Important

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目。

33. Calling REST Services with `RestTemplate`译：33.使用 `RestTemplate`调用REST服务

如果您需要从应用程序调用远程REST服务，则可以使用Spring Framework的`RestTemplate`类。 由于`RestTemplate`实例经常需要在使用之前进行定制，因此Spring Boot不提供任何单一的自动配置的`RestTemplate` bean。 但它会自动配置`RestTemplateBuilder` ，可用于在需要时创建`RestTemplate`实例。 自动配置的`RestTemplateBuilder`可确保将合理的`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);
 }

}
```

`RestTemplateBuilder`包含许多可用于快速配置`RestTemplate`的有用方法。 例如，要添加BASIC认证支持，您可以使用`builder.basicAuthorization("user", "password").build()`。

33.1 RestTemplate Customization 译：33.1 RestTemplate自定义

有`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范围。

为了尽可能缩小任何自定义的范围，请注入自动配置的`RestTemplateBuilder`，然后根据需要调用其方法。 每个方法调用都会返回一个新的`RestTemplateBuilder`实例，所以自定义仅影响构建器的这种使用。

要进行应用程序范围的附加定制，请使用`RestTemplateCustomizer` bean。 所有这些bean都会自动注册到自动配置的`RestTemplateBuilder`，并应用于任何使用它构建的模板。

以下示例显示了一个定制程序，该定制程序为除`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，最极端的（也是很少使用的）选项是创建自己的`RestTemplateBuilder` bean。 这样做会关闭`RestTemplateBuilder`的自动配置，并防止使用任何`RestTemplateCustomizer` Bean。

34. Calling REST Services with `WebClient` 译：34.使用 `WebClient` 调用REST服务

如果您的类路径中包含Spring WebFlux，则还可以选择使用`WebClient`来调用远程REST服务。 与`RestTemplate`相比，该客户端具有更多的功能，并且完全被动。 您可以使用构建器`WebClient.create()`创建您自己的客户端实例。 请参阅relevant section on WebClient 。

Spring Boot为您创建并预配置这样的构建器。 例如，客户端HTTP编解码器的配置方式与服务器的相同（请参阅WebFlux HTTP codecs auto-configuration ）。

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().url("/{name}/details", name)
      .retrieve().bodyToMono(Details.class);
 }

}
```

34.1 WebClient Customization 译：34.1 WebClient自定义

定制`WebClient`有三种主要方法，具体取决于您希望自定义应用的范围。

为了尽可能缩小任何自定义的范围，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

都反映在随后使用它创建的所有客户端中。 如果您想使用相同的构建器创建多个客户端，则还可以考虑使用`WebClient.Builder other = builder.clone();`克隆构建器。

要为所有`WebClient.Builder`实例进行应用程序范围的附加定制，可以声明`WebClientCustomizer` bean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`。

35. Validation译：35.验证

只要JSR-303实现（例如Hibernate验证器）位于类路径中，Bean Validation 1.1支持的方法验证功能就会自动启用。 这使bean方法可以用`javax.validation`的参数约束和/或返回值进行注释。 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，以便为其内联约束注释搜索其方法。

例如，以下服务触发第一个参数的验证，确保它的大小在8到10之间：

```
@Service
@Validated
public class MyBean {

 public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code,
   Author author) {
  ...
 }

}
```

36. Sending Email译：36.发送电子邮件

Spring Framework通过使用`JavaMailSender`接口为发送电子邮件提供了一个简单的抽象，而Spring Boot为它提供了自动配置以及一个入门模块。

见 reference documentation为你如何使用的详细解释`JavaMailSender`。

如果`spring.mail.host`和相关库（如`spring-boot-starter-mail`定义）可用，则创建默认`JavaMailSender`（如果不存在）。 发件人可以通过`spring.mail`命名空间中的配置项进一步进行自定义。 详情请参阅`MailProperties`。

特别是，某些默认超时值是无限的，并且您可能需要更改该值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
```

也可以使用来自JNDI的现有`Session`配置`JavaMailSender`：

```
spring.mail.jndi-name=mail/Session
```

当设置`jndi-name`，它优先于所有其他与会话相关的设置。

37. Distributed Transactions with JTA译：37.与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。 在部署到合适的Java EE应用程序服务器时，也支持JTA事务。

当检测到JTA环境时，将使用Spring的`JtaTransactionManager`来管理事务。 自动配置的JMS，DataSource和JPA bean已升级以支持XA事务。 您可以使用标准的Spring成语，如`@Transactional`参与分布式事务。 如果您处于JTA环境中并仍希望使用本地事务，则可以将`spring.jta.enabled`属性设置为`false`以禁用JTA自动配置。

37.1 Using an Atomikos Transaction Manager译：37.1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Spring Boot应用程序中。 您可以使用`spring-boot-starter-jta-atomikos` Starter来提取适当的Atomikos库。 Spring Boot会自动配置Atomikos并确保适当的`depends-on`设置应用于Spring bean，以便正确启动和关闭订单。

默认情况下，Atomikos事务日志被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定制Atomikos `UserTransactionServiceImp`。 有关完整的详细信息，请参阅`AtomikosProperties` Javadoc。

为了确保多个事务管理器可以安全地协调相同的资源管理器，每个Atomikos实例必须配置一个唯一的ID。 默认情况下，此ID是运行Atomikos的机器的IP地址。 为确保生产中的唯一性，您应为每个应用程序实例配置`spring.jta.transaction-manager-id`属性的不同值。

37.2 Using a Bitronix Transaction Manager译：37.2使用Bitronix事务管理器

Bitronix是一个流行的开源JTA事务管理器实现。 您可以使用`spring-boot-starter-jta-bitronix`初学者将适当的Bitronix依赖项添加到您的项目中。 与Atomikos一样，Spring Boot自动配置Bitronix并对bean进行后处理，以确保启动和关闭顺序是正确的。

默认情况下，Bitronix事务日志文件（`part1.btm`和`part2.btm`）被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 documentation。

为了确保多个事务管理器可以安全地协调相同的资源管理器，每个Bitronix实例必须配置一个唯一的ID。 默认情况下，此ID是运行Bitronix的计算机的IP地址。 为确保生产中的唯一性，您应该为每个应用程序实例配置`spring.jta.transaction-manager-id`属性的不同值。

37.3 Using a Narayana Transaction Manager译：37.3使用Narayana事务管理器

Narayana是JBoss支持的流行开源JTA事务管理器实现。 您可以使用`spring-boot-starter-jta-narayana`初学者将适当的Narayana依赖项添加到您的项目中。 与Atomikos和Bitronix一样，Spring Boot自动配置Narayana并对bean进行后处理，以确保启动和关闭顺序是正确的。

默认情况下，Narayana事务日志被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.narayana.properties`开头的属性也可用于自定义Narayana配置。 有关完整的详细信息，请参阅`NarayanaProperties` Javadoc 。

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Narayana实例配置一个唯一的ID。 默认情况下，此ID设置为`1` 。 为确保生产中的唯一性，您应为每个应用程序实例配置`spring.jta.transaction-manager-id`属性的不同值。

37.4 Using a Java EE Managed Transaction Manager 译: 37.4使用Java EE托管事务管理器

如果您将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。 春天引导试图通过寻找共同的JNDI位置（自动配置一个事务管理器`java:comp/UserTransaction`， `java:comp/TransactionManager` ，等等）。 如果您使用应用程序服务器提供的事务服务，则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` ）处查找`ConnectionFactory`来自动配置JMS，并且可以使用`spring.datasource.jndi-name` property配置`DataSource` 。

37.5 Mixing XA and Non-XA JMS Connections 译: 37.5混合XA和非XA JMS连接

在使用JTA时，主JMS `ConnectionFactory` bean具有XA感知能力并参与分布式事务。 在某些情况下，您可能需要使用非XA `ConnectionFactory`来处理某些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。

如果您想使用非XA `ConnectionFactory` ，则可以注入`nonXaJmsConnectionFactory` bean而不是`@Primary` `jmsConnectionFactory` bean。 为了保持一致性，`jmsConnectionFactory` bean还通过使用bean别名`xaJmsConnectionFactory` 。

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;
```

37.6 Supporting an Alternative Embedded Transaction Manager 译: 37.6支持替代嵌入式事务管理器

`XAConnectionFactoryWrapper`和`XADataSourceWrapper`接口可用于支持替代嵌入式事务管理器。 这些接口负责封装`XAConnectionFactory`和`XADataSource` bean，并将它们公开为常规的`ConnectionFactory`和`DataSource` bean，它们透明地注册到分布式事务中。 DataSource和JMS自动配置使用JTA变体，前提是您的`JtaTransactionManager` bean和适当的XA包装Bean注册在您的`ApplicationContext` 。

BitronixXAConnectionFactoryWrapper和 BitronixXADataSourceWrapper提供了如何编写XA包装的好例子。

38. Hazelcast 译: 38. Hazelcast

如果类路径中 包含Hazelcast并且找到合适的配置，Spring Boot会自动配置一个 `HazelcastInstance` ，您可以在应用程序中注入该配置。

如果你定义了一个`com.hazelcast.config.Config` bean，Spring Boot会使用它。 如果您的配置定义了实例名称，Spring Boot将尝试查找现有实例而不是创建新实例。

您还可以指定通过配置使用的 `hazelcast.xml`配置文件，如下例所示：

```
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

否则，Spring Boot将尝试从默认位置： `hazelcast.xml`找到工作目录或类路径根目录中的Hazelcast配置。 我们还检查`hazelcast.config`系统属性是否已设置。 有关更多详细信息，请参阅Hazelcast documentation 。

如果类路径中存在 `hazelcast-client` ，Spring Boot会首先尝试通过检查以下配置选项来创建客户端：

- The presence of a `com.hazelcast.client.config.ClientConfig` bean.
- A configuration file defined by the `spring.hazelcast.config` property.
- The presence of the `hazelcast.client.config` system property.
- A `hazelcast-client.xml` in the working directory or at the root of the classpath.

Spring Boot也有explicit caching support for Hazelcast 。 如果启用了缓存，则`HazelcastInstance`将自动包装在`CacheManager`实现中。

39. Quartz Scheduler 译: 39.石英调度程序

Spring Boot为Quartz scheduler提供了一些便利，包括`spring-boot-starter-quartz` Starter”。 如果石英是可用的，一个`Scheduler`是自动配置（通过`SchedulerFactoryBean`抽象）。

以下类型的豆类会自动拾取并与 `Scheduler`关联：

- `JobDetail`: defines a particular Job. `JobDetail` instances can be built with the `JobBuilder` API.
- `Calendar`.
- `Trigger`: defines when a particular job is triggered.

默认情况下，使用内存`JobStore`。 但是，如果`DataSource` bean在您的应用程序中可用并且`spring.quartz.job-store-type`属性已相应配置，则可以配置基于JDBC的存储，如以下示例中所示：

```
spring.quartz.job-store-type=jdbc
```

使用JDBC存储时，可以在启动时初始化模式，如以下示例所示：

```
spring.quartz.jdbc.initialize-schema=always
```

默认情况下，使用Quartz库提供的标准脚本检测和初始化数据库。 也可以通过设置`spring.quartz.jdbc.schema`属性来提供自定义脚本。

可以使用Quartz配置属性（） `spring.quartz.properties.*`和 `SchedulerFactoryBeanCustomizer` bean来定制Quartz Scheduler配置，这些bean允许编程`SchedulerFactoryBean`自定义。

特别是， `Executor` bean与调度程序没有关联，因为Quartz提供了一种通过`spring.quartz.properties`配置调度程序的`spring.quartz.properties`。 如果您需要定制任务执行程序，请考虑实施`SchedulerFactoryBeanCustomizer`。

作业可以定义设置器以注入数据映射属性。 常规bean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 {
  ...
 }

}
```

40. Spring Integration译:40.春季整合

Spring Boot为Spring Integration提供了一些便利，包括`spring-boot-starter-integration` Starter"。 Spring Integration提供对消息传递和其他传输（如HTTP，TCP和其他传输）的抽象。 如果Spring集成在类路径中可用，则它将通过`@EnableIntegration`注释进行初始化。

Spring Boot还配置了一些由额外的Spring Integration模块触发的功能。 如果`spring-integration-jmx`也位于类路径中，则会通过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
```

有关更多详细信息，请参阅 `IntegrationAutoConfiguration`和 `IntegrationProperties`类。

默认情况下，如果存在千分尺`meterRegistry` bean，Spring集成度量标准将由千分尺管理。 如果您希望使用传统的Spring集成度量标准，请将`DefaultMetricsFactory` Bean添加到应用程序上下文中。

41. Spring Session译:41.春季会议

Spring Boot为各种数据存储提供Spring Session自动配置。 在构建Servlet Web应用程序时，可以自动配置以下商店：

- JDBC
- Redis
- Hazelcast
- MongoDB

构建响应式Web应用程序时，可以自动配置以下商店：

- Redis
- MongoDB

如果类路径中存在单个Spring Session模块，Spring Boot会自动使用该存储实现。 如果您有多个实现，则必须选择您希望用来存储会话的`StoreType`。 例如，要将JDBC用作后端存储，可以按如下所示配置应用程序：

```
spring.session.store-type=jdbc
```

您可以通过设置禁用春季会议 `store-type`至 `none`。

每家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
```

要设置会话的超时时间，您可以使用`spring.session.timeout`属性。 如果该属性未设置，则自动配置将回到`server.servlet.session.timeout`的值。

42. Monitoring and Management over JMX译：42.监控和管理过JMX

Java管理扩展（JMX）提供了一个标准机制来监视和管理应用程序。 默认情况下，春季启动创建`MBeanServer`豆用的ID `mbeanServer`，并公开你的任何豆被标注了春天JMX注释（的`@ManagedResource`， `@ManagedAttribute`，或`@ManagedOperation`）。

有关更多详细信息，请参阅`JmxAutoConfiguration`类。

43. Testing译：43.测试

Spring Boot提供了许多实用程序和注释以帮助您测试应用程序。 测试支持由两个模块提供： `spring-boot-test`包含核心项目，`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”，它可以导入Spring Boot测试模块以及JUnit，AssertJ，Hamcrest和其他一些有用的库。

43.1 Test Scope Dependencies译：43.1测试范围依赖关系

`spring-boot-starter-test`启动器”（位于`test` `scope`）包含以下提供的库：

- JUnit: The de-facto standard for unit testing Java applications.
- Spring Test & Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- AssertJ: A fluent assertion library.
- Hamcrest: A library of matcher objects (also known as constraints or predicates).
- Mockito: A Java mocking framework.
- JSONassert: An assertion library for JSON.
- JsonPath: XPath for JSON.

我们通常在编写测试时发现这些通用库是有用的。 如果这些库不适合您的需求，您可以添加您自己的附加测试依赖项。

43.2 Testing Spring Applications译：43.2测试Spring应用程序

依赖注入的一个主要优点是它可以让你的代码更容易进行单元测试。 您甚至可以使用`new`运算符来实例化对象，而不涉及Spring。 您也可以使用*模拟对象*而不是真正的依赖关系。

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 Framework包含一个用于这种集成测试的专用测试模块。 您可以直接声明一个依赖项到`org.springframework:spring-test`或使用`spring-boot-starter-test`启动器”来传递它。

如果您之前没有使用`spring-test`模块，则应该先阅读Spring Framework参考文档的 relevant section 。

43.3 Testing Spring Boot Applications译：43.3测试Spring Boot应用程序

Spring Boot应用程序是一个Spring `ApplicationContext`，所以没有什么特别的事情要做，以便测试它超出了通常使用vanilla Spring上下文所做的事情。

Spring Boot的外部属性，日志记录和其他功能默认情况下仅在使用`SpringApplication`创建时才安装在上下文中。

Spring Boot提供了一个`@SpringBootTest`注释，当您需要Spring Boot功能时，它可以用作标准`spring-test` `@ContextConfiguration`注释的替代品。 注释的工作原理是创建`ApplicationContext`通过在测试中使用`SpringApplication`。 除了`@SpringBootTest`之外，还为应用程序的testing more specific slices提供了一些其他注释。

不要忘记在测试中添加`@RunWith(SpringRunner.class)`，否则注释将被忽略。

您可以使用`webEnvironment`的属性`@SpringBootTest`，以便进一步优化测试的运行：

- `MOCK`: Loads a `WebApplicationContext` and provides a mock servlet environment. Embedded servlet containers are not started when using this annotation. If servlet APIs are not on your classpath, this mode transparently falls back to creating a regular non-web `ApplicationContext`. It can be used in conjunction with `@AutoConfigureMockMvc` for `MockMvc`-based testing of your application.
- `RANDOM_PORT`: Loads an `ServletWebServerApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listen on a random port.
- `DEFINED_PORT`: Loads a `ServletWebServerApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listen on a defined port (from your `application.properties` or on the default port of `8080`).
- `NONE`: Loads an `ApplicationContext` by using `SpringApplication` but does not provide *any* servlet environment (mock or otherwise).

如果您的测试是`@Transactional`，则默认情况下它会在每种测试方法结束时回退事务。 然而，由于使用这种安排或者`RANDOM_PORT`或者`DEFINED_PORT`隐式地提供真正的servlet环境，所以HTTP客户机和服务器在单独的线程中运行，并且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。

43.3.1 Detecting Web Application Type译：43.3.1检测Web应用程序类型

如果Spring MVC可用，则配置常规的基于MVC的应用程序上下文。 如果你只有Spring WebFlux，我们会检测它并配置基于WebFlux的应用程序上下文。

如果两者都存在，则Spring MVC优先。 如果您想在此场景中测试反应型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 { ... }
```

43.3.2 Detecting Test Configuration 译: 43.3.2检测测试配置

如果您熟悉Spring Test Framework，则可能习惯使用`@ContextConfiguration(classes=… )`来指定要加载哪个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。 Spring Boot的`@*Test`注释会在您没有明确定义注释时自动搜索您的主要配置。

搜索算法从包含测试的软件包开始工作，直到找到用`@SpringBootApplication`或`@SpringBootConfiguration`注解的类。 只要你明智的方式你structured your code ，你的主要配置通常被发现。

如果使用 test annotation to test a more specific slice of your application ，则应避免添加特定于 main method's application class上特定区域的配置设置。 `@SpringBootApplication`的基础组件扫描配置定义了用于确保分片按预期工作的排除过滤器。 如果您在`@SpringBootApplication` -annoted类上使用明确的`@ComponentScan`指令，请注意，这些过滤器将被禁用。 如果你正在使用切片，你应该再次定义它们。

如果要定制主配置，则可以使用嵌套的`@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， `@TestConfiguration`使用嵌套的`@TestConfiguration`类。

Spring的测试框架在测试之间缓存应用上下文。 因此，只要您的测试共享相同的配置（不管它如何被发现），加载上下文的潜在耗时过程只会发生一次。

43.3.3 Excluding Test Configuration 译: 43.3.3不包括测试配置

如果您的应用程序使用组件扫描（例如，如果您使用的是`@SpringBootApplication`或`@ComponentScan` ），那么您可能会发现仅为特定测试创建的顶级配置类意外地在任何地方都可以找到。

正如我们have seen earlier， `@TestConfiguration`可以在一个内部类的测试的用于定制的主配置。 当放置在顶层类， `@TestConfiguration`表明，在类`src/test/java`不应通过扫描被拾起。 然后，您可以在需要的地方明确导入该类，如以下示例所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest
@Import(MyTestsConfiguration.class)
public class MyTests {

 @Test
 public void exampleTest() {
  ...
 }

}
```

如果您直接使用`@ComponentScan` （即不通过`@SpringBootApplication` ），则需要使用它注册`TypeExcludeFilter` 。 有关详细信息，请参阅the Javadoc 。

43.3.4 Testing with a running server 译: 43.3.4使用正在运行的服务器进行测试

如果您需要启动完整的运行服务器，我们建议您使用随机端口。 如果您使用`@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` ，则每次运行测试时都会随机选取一个可用端口。

`@LocalServerPort`注释可用于inject the actual port used进入您的测试。 为了方便起见，需要对已启动的服务器进行REST调用的测试还可以使用`@Autowire`和`WebTestClient` ，它解析了与正在运行的服务器的相关链接，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");
 }

}
```

Spring Boot还提供了一个`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");
 }

}
```

43.3.5 Using JMX译：43.3.5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以防止相同组件在同一个域上注册。 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() {
  // ...
 }

}
```

43.3.6 Mocking and Spying Beans译：43.3.6模拟和侦察豆类

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，您可能对开发期间不可用的某些远程服务有一定的看法。 当你想模拟在真实环境中很难触发的故障时，模拟也很有用。

春季启动包括`@MockBean`注释，可以用来定义一个内部模拟的Mockito一个bean `ApplicationContext`。 您可以使用注释来添加新的bean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`类和字段上。 当在字段上使用时，创建的模拟实例也被注入。 模拟豆类在每种测试方法后自动重置。

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。 要以不同的安排使用此功能，必须明确添加侦听器，如以下示例所示：

```
@TestExecutionListeners(MockitoTestExecutionListener.class)
```

以下示例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，还可以使用`@SpyBean`用的Mockito包装任何现有的bean `spy`。 有关完整的详细信息，请参阅Javadoc。

尽管Spring的测试框架在测试之间缓存应用上下文，并为共享相同配置的测试重用上下文，但使用 `@MockBean` 或 `@SpyBean` 影响缓存密钥，这很可能会增加上下文的数量。

43.3.7 Auto-configured Tests 译：43.3.7自动配置的测试

Spring Boot的自动配置系统适用于应用程序，但有时可能对测试有点过分。 通常只会加载测试应用程序的“片段”所需的配置部分。 例如，您可能想要测试Spring MVC控制器是否正确映射了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，并且在这些Web层没有兴趣时测试运行。

`spring-boot-test-autoconfigure` 模块包含许多可用于自动配置这些“片段”的注释。 它们中的每一个都以类似的方式工作，提供 `@… Test` 注释，该注释可加载 `ApplicationContext` 以及一个或多个可用于自定义自动配置设置的 `@AutoConfigure…` 注释。

每个片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` 。

也可以将 `@AutoConfigure…` 注释与标准 `@SpringBootTest` 注释一起使用。 如果您对“应用程序”不感兴趣，但想要一些自动配置的测试bean，则可以使用此组合。

43.3.8 Auto-configured JSON Tests 译：43.3.8自动配置的JSON测试

要测试该对象，JSON序列化和反序列化按预期工作，可以使用 `@JsonTest` 注释。 `@JsonTest` 自动配置可用的受支持的JSON映射器，该映射器可以是以下某个库：

- Jackson `ObjectMapper`, any `@JsonComponent` beans and any Jackson `Module`s
- `Gson`
- `Jsonb`

如果您需要配置自动配置的元素，则可以使用 `@AutoConfigureJsonTesters` 注释。

Spring Boot包含基于AssertJ的助手，它们与JSONassert和JsonPath库一起工作，以检查JSON是否按预期显示。 的 `JacksonTester` ， `GsonTester` ， `JsonbTester` 和 `BasicJsonTester` 类可用于杰克逊，GSON，Jsonb，和字符串分别。 在测试类的任何帮助字段可以是 `@Autowired` 使用时 `@JsonTest` 。 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");
 }

}
```

JSON帮助程序类也可以直接在标准单元测试中使用。 为此，如果不使用 `@JsonTest` ，请在 `initFields` 方法中调用助手的 `@Before` 方法。

由 `@JsonTest` 启用的自动配置列表可以是 found in the appendix 。

43.3.9 Auto-configured Spring MVC Tests 译：43.3.9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` 批注。 `@WebMvcTest` 自动配置Spring MVC的基础设施和限制扫描豆 `@Controller` ， `@ControllerAdvice` ， `@JsonComponent` ， `Converter` ， `GenericConverter` ， `Filter` ， `WebMvcConfigurer` ，并 `HandlerMethodArgumentResolver` 。使用此批注时，不会扫描常规 `@Component` 豆类。

如果您需要注册额外的组件（例如Jackson `Module` ，则可以通过在测试中使用 `@Import` 来导入其他配置类。

通常， `@WebMvcTest` 仅限于单个控制器，并与 `@MockBean` 结合使用，为所需的合作者提供模拟实现。

`@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。

您还可以自动配置`MockMvc`在非`@WebMvcTest`（如`@SpringBootTest`），通过注释它`@AutoConfigureMockMvc`。 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"));
 }

}
```

如果需要配置自动配置的元素（例如，应应用servlet过滤器），则可以使用`@AutoConfigureMockMvc`注释中的属性。

如果您使用HtmlUnit或Selenium，则自动配置还会提供一个HTMLUnit `WebClient` bean和/或一个`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");
 }

}
```

默认情况下，Spring Boot将`WebDriver` bean放入特殊的“scope”中，以确保驱动程序在每次测试后退出并注入新实例。 如果您不想要这种行为，则可以将`@Scope("singleton")`添加到`WebDriver` `@Bean`定义中。

由`@WebMvcTest`启用的自动配置设置的`@WebMvcTest`可以是 found in the appendix 。

有时编写Spring MVC测试是不够的; Spring Boot可以帮助你运行full end-to-end tests with an actual server 。

43.3.10 Auto-configured Spring WebFlux Tests 译：43.3.10自动配置的Spring WebFlux测试

要测试Spring WebFlux控制器是否按预期工作，可以使用`@WebFluxTest`注释。 `@WebFluxTest`自动配置春季WebFlux基础设施和限制扫描豆`@Controller`，`@ControllerAdvice`， `@JsonComponent`， `Converter`， `GenericConverter`，并`WebFluxConfigurer`。 当使用`@WebFluxTest`注释时，不扫描常规`@Component`豆。

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， `@WebFluxTest` 仅限于单个控制器，并与 `@MockBean` 注释结合使用，为所需的合作者提供模拟实现。

`@WebFluxTest` 也自动配置 `WebTestClient` ，它提供了一种快速测试WebFlux控制器而无需启动完整HTTP服务器的强大方法。

您还可以自动配置 `WebTestClient` 在非 `@WebFluxTest` （如 `@SpringBootTest` ），通过注释它 `@AutoConfigureWebTestClient` 。 以下示例显示了同时使用 `@WebFluxTest` 和 `WebTestClient` ：

```
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");
 }

}
```

此设置仅受WebFlux应用程序支持，因为在 `WebTestClient` Web应用程序中使用 `WebTestClient` 仅适用于WebFlux。

由 `@WebFluxTest` 启用的自动配置列表可以是 found in the appendix 。

`@WebFluxTest` 无法检测通过功能性Web框架注册的路由。 为了测试 `RouterFunction` 的背景下，豆类，考虑导入您 `RouterFunction` 自己通过 `@Import` 或使用 `@SpringBootTest` 。

有时编写Spring WebFlux测试是不够的; Spring Boot可以帮助您运行full end-to-end tests with an actual server 。

43.3.11 Auto-configured Data JPA Tests 译：43.3.11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。 `@Component` 常规bean未加载到 `ApplicationContext` 。

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 Framework参考手册中的relevant section 。 如果这不是您想要的，您可以按照以下步骤禁用测试或整个课程的事务管理：

```
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测试也可以注入一个 `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");
 }

}
```

内存中的嵌入式数据库通常可以很好地用于测试，因为它们速度快，不需要任何安装。 但是，如果您希望针对实际数据库运行测试，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 {

 // ...

}
```

由`@DataJpaTest`启用的自动配置设置的`@DataJpaTest`可以是 found in the appendix 。

43.3.12 Auto-configured JDBC Tests 译：43.3.12自动配置的JDBC测试

`@JdbcTest`类似于`@DataJpaTest`但用于纯JDBC相关测试。 默认情况下，它还配置内存嵌入式数据库和`JdbcTemplate` 。 普通`@Component`豆没有加载到`ApplicationContext` 。

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 Framework参考手册中的relevant section 。 如果这不是您想要的，那么您可以为测试或整个班级禁用事务管理，如下所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.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 {

}
```

如果你喜欢你的测试对一个真正的数据库上运行，你可以使用`@AutoConfigureTestDatabase`以同样的方式作为注释`DataJpaTest` 。（请参阅“ Section 43.3.11, “Auto-configured Data JPA Tests” ”。）

由`@JdbcTest`启用的自动配置列表可以是 found in the appendix 。

43.3.13 Auto-configured jOOQ Tests 译：43.3.13自动配置的jOOQ测试

您可以使用`@JooqTest`以类似的方式为`@JdbcTest`但jOOQ相关的测试。 由于jOOQ严重依赖与数据库模式相对应的基于Java的模式，因此使用现有的`DataSource` 。 如果您想将其替换为内存数据库，则可以使用`@AutoConfigureTestDatabase`来覆盖这些设置。（关于在Spring Boot中使用jOOQ的更多信息，请参阅本章前面的“ Section 29.5, “Using jOOQ” ”。）

`@JooqTest`配置`DSLContext` 。 常规`@Component`豆没有加载到`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，默认情况下在每次测试结束时回滚。 如果这不是您想要的，您可以禁用针对测试或整个测试类的事务管理shown in the JDBC example 。

由`@JooqTest`启用的自动配置列表可以是 found in the appendix 。

43.3.14 Auto-configured Data MongoDB Tests 译：43.3.14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。 普通`@Component`豆没有加载到`ApplicationContext`。 （有关在Spring Boot中使用MongoDB的更多信息，请参阅本章前面的“Section 30.2, “MongoDB” ”。）

以下类显示使用中的`@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;

 //
}
```

内存中的嵌入式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，如果您希望针对真正的MongoDB服务器运行测试，则应排除嵌入式MongoDB自动配置，如下例所示：

```
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 {

}
```

由`@DataMongoTest`启用的自动配置设置的`@DataMongoTest`可以是 found in the appendix 。

43.3.15 Auto-configured Data Neo4j Tests 译：43.3.15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，它使用内存中嵌入的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。 普通`@Component`豆没有加载到`ApplicationContext`。 （有关在Spring Boot中使用Neo4J的更多信息，请参阅本章前面的“ Section 30.3, “Neo4j” ”。）

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relevant section 。 如果这不是您想要的，那么您可以为测试或整个班级禁用事务管理，如下所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.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 {

}
```

由`@DataNeo4jTest`启用的自动配置设置的`@DataNeo4jTest`可以是 found in the appendix 。

43.3.16 Auto-configured Data Redis Tests 译：43.3.16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，它扫描`@RedisHash`类并配置Spring Data Redis存储库。 普通`@Component`豆没有加载到`ApplicationContext`。 （有关在Spring Boot中使用Redis的更多信息，请参阅本章前面的“ Section 30.1, “Redis” ”。）

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;

 //
}
```

由 `@DataRedisTest` 启用的自动配置设置的 `@DataRedisTest` 可以是 found in the appendix 。

43.3.17 Auto-configured Data LDAP Tests 译：43.3.17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，它配置内存中的嵌入式LDAP（如果可用），配置 `LdapTemplate` ，扫描 `@Entry` 类，并配置Spring Data LDAP存储库。 常规 `@Component` 豆没有加载到 `ApplicationContext` 。 （有关在Spring Boot中使用LDAP的更多信息，请参阅本章前面的“ Section 30.9, “LDAP” ”。）

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;

 //
}
```

内存中的嵌入式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的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 {

}
```

由 `@DataLdapTest` 启用的自动配置设置的 `@DataLdapTest` 可以是 found in the appendix 。

43.3.18 Auto-configured REST Clients 译：43.3.18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` 支持。 应该使用 `value` 或 `components` 属性 `@RestClientTest` 来指定要测试的特定Bean，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");
 }

}
```

由 `@RestClientTest` 启用的自动配置设置的 `@RestClientTest` 可以是 found in the appendix 。

43.3.19 Auto-configured Spring REST Docs Tests 译：43.3.19自动配置的Spring REST Docs测试

您可以使用 `@AutoConfigureRestDocs` 注释在使用Mock MVC或REST Assured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` 均可使用（覆盖默认的输出目录 `target/generated-snippets` 如果你正在使用Maven或 `build/generated-snippets` 如果您使用的摇篮）。 它也可以用来配置出现在任何记录的URI中的主机，方案和端口。

Auto-configured Spring REST Docs Tests with Mock MVC 译:自动配置的Spring REST Docs使用Mock MVC进行测试

`@AutoConfigureRestDocs`定制了`MockMvc` bean以使用Spring REST文档。 您可以使用`@Autowired`注入它，并像使用Mock MVC和Spring REST Docs时一样在测试中使用它，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.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"));
 }

}
```

如果您需要更多地控制Spring REST Docs配置，而不是`@AutoConfigureRestDocs`的属性， `@AutoConfigureRestDocs`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());
 }

}
```

如果您想使用Spring REST Docs支持参数化输出目录，则可以创建一个`RestDocumentationResultHandler` bean。 自动配置调用`alwaysDo`与此结果处理程序，从而导致每个`MockMvc`调用自动生成默认片段。 以下示例显示了定义的`RestDocumentationResultHandler`：

```
@TestConfiguration
static class ResultHandlerConfiguration {

 @Bean
 public RestDocumentationResultHandler restDocumentation() {
  return MockMvcRestDocumentation.document("{method-name}");
 }

}
```

Auto-configured Spring REST Docs Tests with REST Assured 译:使用REST保证的自动配置的Spring REST Docs测试

`@AutoConfigureRestDocs`会生成一个`RequestSpecification` bean，该bean已预先配置为使用Spring REST Docs，可用于您的测试。 您可以使用`@Autowired`注入它，并像使用REST Assured和Spring REST Docs时一样在测试中使用它，如以下示例所示：

```
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 `@AutoConfigureRestDocs` 属性提供的Spring REST Docs配置进行更多控制，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());
 }

}
```

43.3.20 User Configuration and Slicing 译：43.3.20用户配置和切片

如果您 structure your code在一个合理的方式，你 `@SpringBootApplication` 类是 used by default作为测试的配置。

然后重要的是不要让应用程序的主类使用特定于其功能特定区域的配置设置。

假定您正在使用Spring Batch，并且您依赖于它的自动配置。 你可以定义你的 `@SpringBootApplication` 如下：

```
@SpringBootApplication
@EnableBatchProcessing
public class SampleApplication { ... }
```

因为这个类是测试的源代码配置，所以任何切片测试都会尝试启动Spring Batch，这绝对不是您想要做的。 推荐的方法是将该特定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 { ... }
```

根据应用程序的复杂程度，您可以为您的自定义设置一个 `@Configuration` 类，也可以为每个域的区域设置一个类。 后一种方法允许您在其中一个测试中启用它，如有必要，请使用 `@Import` 注释。

类路径扫描是造成混淆的另一个原因。 假设，虽然您以合理的方式构建代码，但您需要扫描一个额外的软件包。 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，而无论您选择哪个片，都会扫描这两个软件包的副作用。 例如， `@DataJpaTest` 似乎突然扫描应用程序的组件和用户配置。 再次，将自定义指令移至单独的类是解决此问题的好方法。

如果这不是您的选择，您可以在测试层次结构中的某个位置创建一个 `@SpringBootConfiguration` ，以便使用它。 或者，您可以指定测试源，以禁用找到默认行为的行为。

43.3.21 Using Spock to Test Spring Boot Applications 译：43.3.21使用Spock测试Spring Boot应用程序

如果您希望使用Spock测试Spring Boot应用程序，则应该在Spock的 `spock-spring` 模块中添加对应用程序构建的依赖关系。 `spock-spring` 将Spring的测试框架集成到Spock中。 建议您使用Spock 1.1或更高版本来从Spock的Spring Framework和Spring Boot集成的许多改进中受益。 详情请参阅the documentation for Spock's Spring module 。

43.4 Test Utilities 译：43.4测试实用程序

测试应用程序时通常使用的一些测试实用程序类将作为 `spring-boot` 一部分进行 `spring-boot` 。

43.4.1 ConfigFileApplicationContextInitializer 译：43.4.1 ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer` 是一个 `ApplicationContextInitializer` ，您可以将其应用于您的测试以加载Spring Boot `application.properties` 文件。 如果您不需要 `@SpringBootTest` 提供的全部功能， `@SpringBootTest` 可以使用它，如以下示例所示：

```
@ContextConfiguration(classes = Config.class,
 initializers = ConfigFileApplicationContextInitializer.class)
```

单独使用 `ConfigFileApplicationContextInitializer` 不支持 `@Value("${… }")` 注射。 它唯一的工作是确保将 `application.properties` 文件加载到Spring的 `Environment` 。 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。

43.4.2 TestPropertyValues 译：43.4.2 TestPropertyValues

`TestPropertyValues` 可让您快速将属性添加到 `ConfigurableEnvironment` 或 `ConfigurableApplicationContext` 。 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);
```

43.4.3 OutputCapture 译：43.4.3 OutputCapture

`OutputCapture` 是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"));
 }

}
```

43.4.4 TestRestTemplate 译：43.4.4 TestRestTemplate

Spring框架5.0提供了一个新 `WebTestClient` ，对工作WebFlux integration tests两者WebFlux and MVC end-to-end testing 。 它为断言提供流畅的API，与 `TestRestTemplate` 不同。

`TestRestTemplate` 是一个方便的替代Springâ€™的 `RestTemplate` 是在集成测试非常有用。 您可以获得一个香草模板或一个发送基本HTTP认证（使用用户名和密码）的模板。 在任何一种情况下，模板的行为都是通过不会在服务器端错误上抛出异常的方式进行测试。 建议使用Apache HTTP Client（版本4.3.2或更高版本），但不是强制性的。 如果你的类路径中有这个， `TestRestTemplate` 通过适当地配置客户端来做出响应。 如果您确实使用Apache的HTTP客户端，则会启用一些其他易于使用的测试功能：

- Redirects are not followed (so you can assert the response location).
- Cookies are ignored (so the template is stateless).

`TestRestTemplate` 可以在集成测试中直接实例化，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。 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，如下例所示：

```
@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(1000).setReadTimeout(1000);
  }

 }

}
```

44. WebSockets 译：44. WebSockets

Spring Boot为嵌入式Tomcat 8.5，Jetty 9和Undertow提供WebSockets自动配置。 如果将war文件部署到独立容器中，Spring Boot假定该容器负责配置其WebSocket支持。

Spring Framework提供了 rich WebSocket support ，可以通过 `spring-boot-starter-websocket` 模块轻松访问。

45. Web Services 译：45.网络服务

Spring Boot提供Web服务自动配置，因此您只需定义您的 `Endpoints` 。

该 Spring Web Services features可以用轻松访问 `spring-boot-starter-webservices` 模块。

可以分别为WSDL和XSD自动创建 `SimpleWsdl11Definition` 和 `SimpleXsdSchema` bean。 为此，请配置它们的位置，如下例所示：

```
spring.webservices.wsdl-locations=classpath:/wsdl
```

46. Creating Your Own Auto-configuration 译：46.创建自己的自动配置

如果您在开发共享库的公司工作，或者如果您在开源或商业库上工作，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瓶中，并且仍然可以通过Spring Boot获取。

自动配置可以关联到一个“启动器”，它提供了自动配置代码以及您将使用的典型库。 我们首先介绍您需要了解的内容以构建自己的自动配置，然后转到typical steps required to create a custom starter 。

demo project可用于展示如何逐步创建启动器。

46.1 Understanding Auto-configured Beans 译：46.1了解自动配置的豆类

在引擎盖下，自动配置通过标准的 `@Configuration` 类来实现。 额外的 `@Conditional` 批注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。 这可确保只有在找到相关类别并且尚未声明自己的 `@Configuration` 时，才会应用自动配置。

您可以浏览 `spring-boot-autoconfigure` 的源代码以查看Spring提供的 `@Configuration` 类（请参阅 `META-INF/spring.factories` 文件）。

46.2 Locating Auto-configuration Candidates 译：46.2查找自动配置候选

Spring Boot检查发布的jar中是否存在 `META-INF/spring.factories` 文件。 该文件应在 `EnableAutoConfiguration` 密钥下列出您的配置类，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` 之后应用您的课程。

如果您想要订购某些不应彼此直接了解的自动配置，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的顺序。

自动配置*只能*以这种方式加载。 确保它们是在特定的包装空间中定义的，特别是它们永远不是组件扫描的目标。

46.3 Condition Annotations 译：46.3条件注释

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` 注释是一个常用示例，用于允许开发人员在对默认设置不满意时覆盖自动配置。

Spring Boot包含多个 `@Conditional` 注释，您可以在自己的代码中重复使用注释 `@Configuration` 类或单个 `@Bean` 方法。 这些注释包括：

- Section 46.3.1, “Class Conditions”
- Section 46.3.2, “Bean Conditions”
- Section 46.3.3, “Property Conditions”
- Section 46.3.4, “Resource Conditions”
- Section 46.3.5, “Web Application Conditions”
- Section 46.3.6, “SpEL Expression Conditions”

46.3.1 Class Conditions 译: 46.3.1类级条件

`@ConditionalOnClass`和`@ConditionalOnMissingClass`批注可以根据是否存在特定类来包含配置。 由于使用ASM解析了注记元数据，因此即使该类可能实际上并未出现在正在运行的应用程序类路径中，也可以使用`value`属性来引用真实类。 如果您希望使用`String`值指定类名称，也可以使用`name`属性。

如果使用`@ConditionalOnClass`或`@ConditionalOnMissingClass`作为元注释的一部分来组成自己的注释，则必须使用`name`作为引用该类的类，在这种情况下不处理。

46.3.2 Bean Conditions 译: 46.3.2豆条件

`@ConditionalOnBean`和`@ConditionalOnMissingBean`注释允许根据是否存在特定的bean来包含bean。 您可以使用`value`属性通过类型指定bean，或使用`name`通过名称指定bean。 使用`search`属性可以限制在搜索bean时应考虑的`ApplicationContext`层次结构。

放置在`@Bean`方法中时，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() { ... }

}
```

在前面的例子中，`myService`豆会如果没有类型的豆，以创建`MyService`已经包含在`ApplicationContext`。

您需要非常小心添加bean定义的顺序，因为这些条件是基于迄今为止已处理的内容进行评估的。 出于这个原因，我们建议在自动配置类上仅使用`@ConditionalOnBean`和`@ConditionalOnMissingBean`注释（因为在添加任何用户定义的bean定义后，这些注释将被保证加载）。

`@ConditionalOnBean`和`@ConditionalOnMissingBean`不会阻止创建`@Configuration`类。 在课堂级别使用这些条件等同于使用注释标记每个包含`@Bean`方法。

46.3.3 Property Conditions 译: 46.3.3财产状况

`@ConditionalOnProperty`注释允许基于Spring Environment属性包含配置。 使用`prefix`和`name`属性来指定应该检查的属性。 默认情况下，存在且不等于`false`任何属性`false`被匹配。 您还可以使用`havingValue`和`matchIfMissing`属性创建更高级的检查。

46.3.4 Resource Conditions 译: 46.3.4资源条件

`@ConditionalOnResource`注释仅允许在特定资源存在时包含配置。 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`。

46.3.5 Web Application Conditions 译: 46.3.5 Web应用程序条件

`@ConditionalOnWebApplication`和`@ConditionalOnNotWebApplication`注释根据应用程序是否为“网络应用程序”进行配置。 Web应用程序是使用Spring `WebApplicationContext`，定义`session`范围或具有`StandardServletEnvironment`任何应用程序。

46.3.6 SpEL Expression Conditions 译: 46.3.6 SpEL表达条件

所述`@ConditionalOnExpression`注解让配置基于一个的结果被包括 SpEL expression。

46.4 Testing your Auto-configuration 译: 46.4测试您的自动配置

自动配置可能受多种因素影响：用户配置（`@Bean`定义和`Environment`定制），条件评估（存在特定库）等。 具体来说，每个测试都应该创建一个明确定义的`ApplicationContext`，它代表这些定制的组合。 `ApplicationContextRunner`提供了一个很好的方法来实现这一点。

通常将`ApplicationContextRunner`定义为测试类的字段以收集基础，常见配置。 以下示例确保总是调用`UserServiceAutoConfiguration`：

```
private final ApplicationContextRunner contextRunner = new ApplicationContextRunner()
  .withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class));
```

如果必须定义多个自动配置，则不需要对它们的声明进行排序，因为它们的调用顺序与运行应用程序时的顺序完全相同。

每个测试都可以使用跑步者来表示特定的用例。 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出。 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");
 }

}
```

也可以轻松定制 `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");
 });
}
```

46.4.1 Simulating a Web Context 译: 46.4.1模拟Web上下文

如果您需要测试仅在Servlet或Reactive Web应用程序上下文中运行的自动配置， `ReactiveWebApplicationContextRunner` 分别使用 `WebApplicationContextRunner` 或 `ReactiveWebApplicationContextRunner` 。

46.4.2 Overriding the Classpath 译: 46.4.2覆盖类路径

也可以测试在运行时不存在特定类和/或包时会发生什么情况。 Spring Boot附带 `FilteredClassLoader` ，可以很容易地被跑步者使用。 在以下示例中，我们断言如果 `UserService` 不存在，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"));
}
```

46.5 Creating Your Own Starter 译: 46.5创建您自己的启动器

一个库的完整Spring Boot启动程序可能包含以下组件：

- The `autoconfigure` module that contains the auto-configuration code.
- The `starter` module that provides a dependency to the `autoconfigure` module as well as the library and any additional dependencies that are typically useful. In a nutshell, adding the starter should provide everything needed to start using that library.

如果你不需要分开这两个问题，你可以将自动配置代码和依赖管理结合在一个模块中。

46.5.1 Naming 译: 46.5.1命名

您应该确保为您的初学者提供适当的名称空间。 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东西提供官方支持。

作为一个经验法则，你应该在起动器之后命名一个组合模块。 例如，假设您正在创建一个“acme”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` 。 如果您只有一个模块组合了这两个模块，请将其命名为 `acme-spring-boot-starter` 。

另外，如果您的初学者提供配置密钥，请为它们使用唯一的名称空间。 特别是，不包括你在春天开机使用的命名空间键（如 `server` ， `management` ， `spring` ，等等）。 如果您使用相同的命名空间，我们可能会以破坏模块的方式修改这些命名空间。

请确保至trigger meta-data generation，以便您的密钥也可以使用IDE帮助。 您可能需要查看生成的元数据（ `META-INF/spring-configuration-metadata.json` ）以确保您的密钥已正确记录。

46.5.2 `autoconfigure` Module 译: 46.5.2 `autoconfigure` 模块

`autoconfigure` 模块包含开始使用库所需的一切。 它还可能包含配置密钥定义（例如 `@ConfigurationProperties` ）和任何可用于进一步自定义组件初始化方式的回调接口。

您应该将库的依赖关系标记为可选，以便您可以更轻松地将 `autoconfigure` 模块包含在项目中。 如果你这样做，库不会提供，并且默认情况下，Spring Boot会退出。

Spring Boot使用注释处理器来收集元数据文件中自动配置的条件（ `META-INF/spring-autoconfigure-metadata.properties` ）。 如果该文件存在，它将用于过滤不匹配的自动配置，从而缩短启动时间。 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，应该在`compileOnly`配置中声明依赖关系，如以下示例所示：

```
dependencies {
 compileOnly "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

在Gradle 4.6及更高版本中，应该在`annotationProcessor`配置中声明依赖关系，如以下示例所示：

```
dependencies {
 annotationProcessor "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

46.5.3 Starter Module译：46.5.3入门模块

起动器真的是一个空罐子。 它唯一的目的是提供必要的依赖关系来与库一起工作。 您可以将其视为对开始所需要的自己的看法。

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，请提及它们。 如果可选依赖关系的数量很高，那么提供一组适当的默认依赖关系可能会很困难，因为您应该避免包含对库的典型用法不必要的依赖关系。 换句话说，你不应该包含可选的依赖关系。

无论哪种方式，您的初学者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另一个启动器，则无需添加它）。 如果一个项目只使用您的自定义启动器创建，那么Spring Boot的核心功能将因核心启动器的存在而受到尊重。

47. Kotlin support译：47. Kotlin支持

Kotlin是一种针对JVM（和其他平台）的静态类型语言，它允许编写简洁优雅的代码，同时提供 interoperability与用Java编写的现有库。

Spring Boot通过利用其他Spring项目（如Spring Framework，Spring Data和Reactor）中的支持来提供Kotlin支持。 有关更多信息，请参阅Spring Framework Kotlin support documentation 。

从Spring Boot和Kotlin开始的最简单的方法是遵循this comprehensive tutorial 。 您可以通过start.spring.io创建新的Kotlin项目。 随时加入的#spring通道Kotlin Slack或提问与`spring`个`kotlin`在标签上Stack Overflow ，如果你需要的支持。

47.1 Requirements译：47.1要求

Spring Boot支持Kotlin 1.2.x. 要使用Kotlin， `org.jetbrains.kotlin:kotlin-stdlib`和`org.jetbrains.kotlin:kotlin-reflect`必须存在于类路径中。 也可以使用`kotlin-stdlib`变体`kotlin-stdlib-jdk7`和`kotlin-stdlib-jdk8` 。

自 Kotlin classes are final by default以来，您可能需要配置 kotlin-spring插件以自动打开Spring注释的类，以便它们可以被代理。

在Kotlin中序列化/反序列化JSON数据需要Jackson's Kotlin module 。 它在类路径中找到时会自动注册。 如果Jackson和Kotlin存在，但Jackson Kotlin模块不存在，则会记录一条警告消息。

如果您在 start.spring.io上引导Kotlin项目， 则会默认提供这些依赖项和插件。

47.2 Null-safety译：47.2零安全

Kotlin的主要特点之一是null-safety 。 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，而无需支付像`Optional`这样的包装器的成本。 Kotlin还允许使用comprehensive guide to null-safety in Kotlin中描述的具有可为空值的功能性结构。

虽然Java不允许在类型系统中表示空安全性，但Spring Framework，Spring Data和Reactor现在通过易于使用工具的注释提供了API的无安全性。 默认情况下，Kotlin中使用的Java API的类型被识别为platform types ，其中放宽了空检查。 Kotlin's support for JSR 305 annotations结合可空性注释为Kotlin中的相关Spring API提供null-safety。

可以通过添加`-Xjsr305`编译器标志来配置JSR 305检查，其中包含以下选项： `-Xjsr305={strict|warn|ignore}` 。 默认行为与`-Xjsr305=warn`相同。 从Spring API推断出的Kotlin类型需要`strict`值具有空值安全性，但应该在知道Spring API可空性声明甚至可能在次要版本之间演变并且将来可能会添加更多检查的情况下使用）。

警告：通用类型参数，可变参数和数组元素可空性尚不支持。 有关最新信息，请参阅SPR-15942 。 另外请注意，Spring Boot自己的API是not yet annotated 。

47.3 Kotlin API译：47.3 Kotlin API

47.3.1 runApplication译：47.3.1 runApplication

Spring Boot提供了一种通过`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)`替代`SpringApplication.run(MyApplication::class.java, *args)` 。 它还允许定制应用程序，如以下示例所示：

```
runApplication<MyApplication>(*args) {
 setBannerMode(OFF)
}
```

47.3.2 Extensions译：47.3.2扩展

Kotlin extensions提供了使用附加功能扩展现有类的能力。 Spring Boot Kotlin API利用这些扩展来为现有API添加新的Kotlin特定便利。

`TestRestTemplate`扩展，类似于Spring框架为Spring Framework提供的`RestOperations`提供的扩展。 除此之外，这些扩展可以充分利用Kotlin的通用类型参数。

47.4 Dependency management译：47.4依赖管理

为了避免在类路径中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`

通过Maven，Kotlin版本可以通过`kotlin.version`属性进行定制，插件管理提供给`kotlin-maven-plugin` 。 借助Gradle，Spring Boot插件会自动将`kotlin.version`与Kotlin插件的版本对齐。

47.5 `@ConfigurationProperties`译：47.5`@ConfigurationProperties`

`@ConfigurationProperties`目前仅适用于 `lateinit`或可为空的 `var`属性（前者是推荐的），因为由构造函数初始化的不可变类为 not yet supported 。

```
@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

 }

}
```

为了产生 your own metadata使用注释处理器， `kapt` should be configured与 `spring-boot-configuration-processor`依赖性。

47.6 Testing译：47.6测试

虽然可以使用JUnit 4（默认由`spring-boot-starter-test`提供）来测试Kotlin代码，但推荐使用JUnit 5。 JUnit 5使测试类能够被实例化一次，并且可以重用于所有类的测试。 这使得在非静态方法上使用`@BeforeAll`和`@AfterAll`注释成为可能，这非常适合Kotlin。

要使用JUnit 5， `junit:junit`从`spring-boot-starter-test`排除`junit:junit`依赖`spring-boot-starter-test` ，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 documentation 。 您还需要switch test instance lifecycle to "per-class" 。

47.7 Resources译：47.7资源

47.7.1 Further reading译：47.7.1进一步阅读

- Kotlin language reference
- Kotlin Slack (with a dedicated #spring channel)
- Stackoverflow with `spring` and `kotlin` tags
- Try Kotlin in your browser
- Kotlin blog
- Awesome Kotlin
- Tutorial: building web applications with Spring Boot and Kotlin
- Developing Spring Boot applications with Kotlin
- A Geospatial Messenger with Kotlin, Spring Boot and PostgreSQL
- Introducing Kotlin support in Spring Framework 5.0
- Spring Framework 5 Kotlin APIs, the functional way

47.7.2 Examples译：47.7.2示例

- spring-boot-kotlin-demo: regular Spring Boot + Spring Data JPA project
- mixit: Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB
- spring-kotlin-fullstack: WebFlux Kotlin fullstack example with Kotlin2js for frontend instead of JavaScript or TypeScript
- spring-petclinic-kotlin: Kotlin version of the Spring PetClinic Sample Application

- spring-kotlin-deepdive: a step by step migration for Boot 1.0 + Java to Boot 2.0 + Kotlin

48. What to Read Next译：48.接下来要阅读的内容

如果您想详细了解本节中讨论的任何课程，可以查看Spring Boot API documentation或浏览source code directly 。 如果您有具体问题，请查看how-to部分。

如果您对Spring Boot的核心功能感到满意，您可以继续阅读并阅读 production-ready features 。

Part V. Spring Boot Actuator: Production-ready features译：第五部分 Spring Boot Actuator：生产就绪功能

Spring Boot包含许多附加功能，可帮助您在将应用程序投入生产时监视和管理应用程序。 您可以选择使用HTTP端点或JMX来管理和监控您的应用程序。 审计，健康和指标收集也可以自动应用于您的应用程序。

49. Enabling Production-ready Features译：49.启用生产就绪功能

`spring-boot-actuator`模块提供了Spring Boot的所有生产就绪功能。 启用的功能最简单的方法是依赖添加到`spring-boot-starter-actuator` â€~Starterâ€™。

> **执行器的定义**
>
> 致动器是制造术语，是指用于移动或控制某物的机械装置。 执行器可以从一个小的变化中产生大量的运动。

要将执行器添加到基于Maven的项目中，请添加以下“启动器”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")
}
```

50. Endpoints译：50.终点

执行器端点允许您监控应用程序并与其进行交互。 Spring Boot包含许多内置端点，并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。

每个单独的端点可以是enabled or disabled 。 这控制着端点是否被创建，并且它的bean是否存在于应用程序上下文中。 要远程访问端点，还必须是exposed via JMX or HTTP 。 大多数应用程序选择HTTP，其中端点的ID和`/actuator`的前缀映射到URL。 例如，默认情况下， `health`端点映射到`/actuator/health` 。

以下与技术无关的端点可用：

ID	描述	Enabled by default
`auditevents`	公开当前应用程序的审计事件信息。	是
`beans`	显示应用程序中所有Spring bean的完整列表。	是
`conditions`	显示在配置和自动配置类上评估的条件以及他们做了或不匹配的原因。	是
`configprops`	显示所有`@ConfigurationProperties`的整理列表。	是
`env`	公开来自Spring的`ConfigurableEnvironment` 。	是
`flyway`	显示已应用的所有Flyway数据库迁移。	是
`health`	显示应用健康信息。	是
`httptrace`	显示HTTP跟踪信息（默认情况下为最后100个HTTP请求 - 响应交换）。	是
`info`	显示任意的应用信息。	是
`loggers`	显示和修改应用程序中记录器的配置。	是
`liquibase`	显示已应用的任何Liquibase数据库迁移。	是
`metrics`	显示当前应用程序的“度量”信息。	是
`mappings`	显示所有`@RequestMapping`路径的整理列表。	是
`scheduledtasks`	显示应用程序中的计划任务。	是
`sessions`	允许从Spring会话支持的会话存储中检索和删除用户会话。 使用Spring Session对反应式Web应用程序的支持时不可用。	是

ID	描述	Enabled by default
`shutdown`	让应用程序正常关机。	没有
`threaddump`	执行线程转储。	是

如果您的应用程序是一个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	Enabled by default
`heapdump`	返回一个GZip压缩的 `hprof` 堆转储文件。	是
`jolokia`	通过HTTP公开JMX bean（当Jolokia在类路径上时，不可用于WebFlux）。	是
`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` 标题来检索部分日志文件的内容。	是
`prometheus`	以可以被Prometheus服务器抓取的格式显示指标。	是

要了解有关执行器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 ）。

50.1 Enabling Endpoints译：50.1启用端点

默认情况下，除 `shutdown` 之外的所有端点均已启用。 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
```

如果你喜欢端点启用要选择加入，而不是选择退出，设定 `management.endpoints.enabled-by-default` 属性为 `false` ，并使用单独的端点 `enabled` 性能可以选择退出，下面的示例启用。 `info` 端点并禁止其他所有端点：

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

禁用的端点将从应用程序上下文中完全删除。 如果您只想更改端点所暴露的技术， 请改为使用 `include` and `exclude` properties 。

50.2 Exposing Endpoints译：50.2暴露端点

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	Web
`auditevents`	是	没有
`beans`	是	没有
`conditions`	是	没有
`configprops`	是	没有
`env`	是	没有
`flyway`	是	没有
`health`	是	是
`heapdump`	N / A	没有
`httptrace`	是	没有
`info`	是	是
`jolokia`	N / A	没有
`logfile`	N / A	没有
`loggers`	是	没有
`liquibase`	是	没有
`metrics`	是	没有
`mappings`	是	没有
`prometheus`	N / A	没有

ID	JMX	Web
`scheduledtasks`	是	没有
`sessions`	是	没有
`shutdown`	是	没有
`threaddump`	是	没有

要更改公开哪些端点，请使用以下特定`include`技术的`include`和`exclude`属性：

Property	Default
`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`属性列出不应该公开的端点的ID。 `exclude`属性优先于`include`属性。 `include`和`exclude`属性都可以使用端点ID列表进行配置。

例如，要停止通过JMX公开所有端点并仅公开`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
```

`*`在YAML中有特殊含义，因此如果要包含（或排除）所有端点，请务必添加引号，如以下示例中所示：

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

如果您的应用程序公开曝光，我们强烈建议您也是 secure your endpoints 。

如果你想实现你自己的策略，当端点被暴露时，你可以注册一个`EndpointFilter` bean。

50.3 Securing HTTP Endpoints译：50.3保护HTTP端点

您应该注意保护HTTP端点的方式与使用其他任何敏感网址的方式相同。 如果存在Spring Security，则使用Spring Security的内容协商策略默认保护端点。 例如，如果您希望为HTTP端点配置自定义安全性，则只允许具有特定角色的用户访问它们，但Spring Boot提供了一些方便的`RequestMatcher`对象，可以与Spring Security结合使用。

一个典型的Spring Security配置可能看起来像下面的例子：

```
@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();
 }

}
```

上例使用`EndpointRequest.toAnyEndpoint()`将请求与任何端点进行匹配，然后确保全部具有`ENDPOINT_ADMIN`角色。 其他几种匹配方法也可以在`EndpointRequest`上`EndpointRequest`。 有关详细信息，请参阅API文档（ HTML或PDF ）。

如果您在防火墙后面部署应用程序，您可能更喜欢所有的执行器端点都可以在无需验证的情况下进行访问。 您可以通过更改`management.endpoints.web.exposure.include`属性来完成此`management.endpoints.web.exposure.include`，如下所示：

application.properties。

```
management.endpoints.web.exposure.include=*
```

此外，如果存在Spring Security，则需要添加自定义安全配置，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();
 }

}
```

50.4 Configuring Endpoints译：50.4配置端点

端点自动缓存响应以读取不带任何参数的操作。 要配置端点缓存响应的时间量，请使用其属性`cache.time-to-live`。 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
```

前缀`management.endpoint.<name>`用于唯一标识正在配置的端点。

在进行经过验证的HTTP请求时，`Principal`被视为端点的输入，因此不会缓存响应。

50.5 Hypermedia for Actuator Web Endpoints译：50.5超媒体的执行器网络端点

一个“发现页面”添加了指向所有端点的链接。 默认情况下，“查找页面”在`/actuator`上可用。

当配置自定义管理上下文路径时，“发现页面”自动从`/actuator`移至管理上下文的根目录。 例如，如果管理上下文路径为`/management`，则发现页面可从`/management`。 当管理上下文路径设置为`/`，将禁用发现页面以防止与其他映射发生冲突的可能性。

50.6 Actuator Web Endpoint Paths译：50.6执行器Web端点路径

默认情况下，端点通过使用端点的ID在`/actuator`路径下通过HTTP公开。 例如，`beans`端点暴露在`/actuator/beans`。 如果要将端点映射到其他路径，则可以使用`management.endpoints.web.path-mapping`属性。 另外，如果您想更改基本路径，则可以使用`management.endpoints.web.base-path`。

下面的例子重新映射`/actuator/health`至`/healthcheck`：

application.properties。

```
management.endpoints.web.base-path=/
management.endpoints.web.path-mapping.health=healthcheck
```

50.7 CORS Support译：50.7 CORS支持

Cross-origin resource sharing （CORS）是一个W3C specification，它允许您以灵活的方式指定授权哪种跨域请求。 如果您使用Spring MVC或Spring WebFlux，则可以配置Actuator的Web端点来支持这些场景。

CORS支持在默认情况下处于禁用状态，只有在`management.endpoints.web.cors.allowed-origins`属性设置后才能启用。 以下配置允许来自`example.com`域的`GET`和`POST`调用：

```
management.endpoints.web.cors.allowed-origins=http://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

有关选项的完整列表，请参阅 CorsEndpointProperties 。

50.8 Implementing Custom Endpoints译：50.8实现自定义端点

如果添加`@Bean`带注释`@Endpoint`，任何方法标注有`@ReadOperation`，`@WriteOperation`，或`@DeleteOperation`会自动显示在JMX，并在Web应用程序，通过HTTP为好。 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`编写技术特定的端点。 这些端点仅限于各自的技术。 例如，`@WebEndpoint`仅通过HTTP公开，而不通过JMX公开。

您可以使用`@EndpointWebExtension`和`@EndpointJmxExtension`编写技术特定的扩展。 这些注释可让您提供技术特定的操作，以增强现有端点。

最后，如果你需要访问网络的框架，具体的功能，可以实现Servlet或者春天`@Controller`和`@RestController`他们不被过度使用JMX或者使用不同的网络架构时的成本终点。

50.8.1 Receiving Input译：50.8.1接收输入

端点上的操作通过参数接收输入。 当通过网络公开时，这些参数的值取自URL的查询参数和JSON请求主体。 当通过JMX公开时，这些参数将映射到MBean操作的参数。 参数是默认需要的。 可以通过`@org.springframework.lang.Nullable`它们进行注释来选择它们。

为了允许输入映射到操作方法的参数，实现端点的Java代码应该编译为`-parameters`，实现端点的Kotlin代码应该编译为`-java-parameters`。 如果您使用的是Spring Boot的Gradle插件，或者您正在使用Maven和`spring-boot-starter-parent`，`spring-boot-starter-parent`。

Input type conversion译:输入类型转换

传递给端点操作方法的参数在必要时会自动转换为所需的类型。 在调用操作方法之前，使用`ApplicationConversionService`的实例将通过JMX或HTTP请求接收到的输入转换为所需的类型。

50.8.2 Custom Web Endpoints 译: 50.8.2自定义Web端点

在操作的`@Endpoint`， `@WebEndpoint`，或`@WebEndpointExtension`使用的球衣，Spring MVC的，或Spring WebFlux自动曝光通过HTTP。

Web Endpoint Request Predicates 译: Web端点请求谓词

一个请求谓词会自动为网络暴露端点上的每个操作生成。

Path 译:路径

谓词的路径由端点的ID和Web暴露端点的基本路径决定。 默认的基本路径是`/actuator`。 例如，ID为`sessions`的端点将在谓词中使用`/actuator/sessions`作为其路径。

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。 当端点操作被调用时，变量的值被传递到操作方法。

HTTP method 译: HTTP方法

谓词的HTTP方法由操作类型决定，如下表所示：

Operation	HTTP method
`@ReadOperation`	`GET`
`@WriteOperation`	`POST`
`@DeleteOperation`	`DELETE`

Consumes 译:消费

对于使用请求主体的`@WriteOperation`（HTTP `POST`），谓词的`application/vnd.spring-boot.actuator.v2+json, application/json`子句是`application/vnd.spring-boot.actuator.v2+json, application/json`。 对于所有其他操作，消费条款是空的。

Produces 译:生产

的产生谓词子句可以通过确定`produces`所述的属性`@DeleteOperation`， `@ReadOperation`，和`@WriteOperation`注释。 该属性是可选的。 如果未使用，则自动确定产生子句。

如果操作方法返回`void`或`Void`，则`Void`子句为空。 如果操作方法返回`org.springframework.core.io.Resource`，则生成子句为`application/octet-stream`。 对于所有其他操作，产生子句是`application/vnd.spring-boot.actuator.v2+json, application/json`。

Web Endpoint Response Status 译: 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（如果有的话）。

一个`@ReadOperation`返回一个值，响应状态将是200（OK）。 如果它没有返回值，则响应状态将为404（未找到）。

如果`@WriteOperation`或`@DeleteOperation`返回一个值，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）。

如果调用没有必需参数的操作，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，响应状态将为400（错误请求）。

Web Endpoint Range Requests 译: Web端点范围请求

HTTP范围请求可用于请求部分HTTP资源。 当使用Spring MVC或Spring Web Flux时，返回`org.springframework.core.io.Resource`操作自动支持范围请求。

使用Jersey时不支持范围请求。

Web Endpoint Security 译: Web端点安全

在Web端点或Web特定端点扩展上的操作可以接收当前的`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，以便为经过身份验证的用户和未经身份验证的用户提供不同的行为。 后者通常用于使用其`isUserInRole(String)`方法执行授权检查。

50.8.3 Servlet endpoints 译: 50.8.3 Servlet端点

一个`Servlet`可以通过实现一个`@ServletEndpoint`注解的类来公开，该类也实现了`Supplier<EndpointServlet>`。 Servlet端点提供了与Servlet容器的更深入集成，但代价是可移植性。 它们旨在用于公开现有的`Servlet`作为端点。 对于新的端点，只要有可能，应该首选`@Endpoint`和`@WebEndpoint`注释。

50.8.4 Controller endpoints 译: 50.8.4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`，并将端点的ID用作路径的前缀。 控制器端点提供了与Spring的Web框架的更深层次的集成，但代价是可移植性。 只要有可能，应该首选`@Endpoint`和`@WebEndpoint`注释。

50.9 Health Information 译: 50.9健康信息

您可以使用健康信息来检查正在运行的应用程序的状态。 当生产系统停机时，它经常被监控软件用来提醒某人。 `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某个值进行配置：

Name	描述
`never`	细节永远不会显示。
`when-authorized`	详细信息仅向授权用户显示。 授权角色可以使用`management.endpoint.health.roles`进行配置。
`always`	详细信息显示给所有用户。

默认值是`never` 。 当用户处于一个或多个端点角色时，它被认为是被授权的。 如果端点没有配置角色（默认），则认为所有经过身份验证的用户均被授权。 角色可以使用`management.endpoint.health.roles`属性进行配置。

如果您已保护您的应用程序并希望使用`always` ，则您的安全配置必须允许经过身份验证的用户和未经身份验证的用户访问运行状况端点。

健康信息从您的`ApplicationContext`定义的所有`HealthIndicator`豆中`ApplicationContext` 。 Spring Boot包含一些自动配置的`HealthIndicators` ，您也可以编写自己的。 默认情况下，最终系统状态由`HealthAggregator` ，该状态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`返回一个已知状态`HealthAggregator` ，一个`UNKNOWN`使用状态。

50.9.1 Auto-configured HealthIndicators 译：50.9.1自动配置的HealthIndicators

适当时，以下`HealthIndicators`由Spring Boot自动配置：

Name	描述
`CassandraHealthIndicator`	检查Cassandra数据库是否启动。
`DiskSpaceHealthIndicator`	检查磁盘空间不足。
`DataSourceHealthIndicator`	检查是否可以获得到`DataSource`的连接。
`ElasticsearchHealthIndicator`	检查Elasticsearch集群是否启动。
`InfluxDbHealthIndicator`	检查InfluxDB服务器是否启动。
`JmsHealthIndicator`	检查JMS代理是否启动。
`MailHealthIndicator`	检查邮件服务器是否启动。
`MongoHealthIndicator`	检查Mongo数据库是否启动。
`Neo4jHealthIndicator`	检查Neo4j服务器是否启动。
`RabbitHealthIndicator`	检查Rabbit服务器是否启动。
`RedisHealthIndicator`	检查Redis服务器是否启动。
`SolrHealthIndicator`	检查Solr服务器是否已启动。

您可以通过设置`management.health.defaults.enabled`属性来禁用它们。

50.9.2 Writing Custom HealthIndicators 译：50.9.2编写自定义HealthIndicators

要提供自定义健康信息，您可以注册实现`HealthIndicator`界面的Spring bean。 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，并可以选择包含要显示的其他详细信息。 以下代码显示了`HealthIndicator`实现的示例：

```
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();
 }

}
```

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`my` 。

除了春Bootâ€™的预定义`Status`种类型，还可以用于`Health`返回一个自定义`Status`代表一个新的系统状态。 在这种情况下，还需要提供`HealthAggregator`接口的自定义实现，或者必须使用`management.health.status.order`配置属性来配置默认实现。

例如，假设在您的一个`HealthIndicator`实现中使用了代码为`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中：

```
management.health.status.order=FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

响应中的HTTP状态代码反映整体健康状况（例如， `UP`映射到200，而`OUT_OF_SERVICE`和`DOWN`映射到503）。 如果您通过HTTP访问健康端点，则可能还需要注册自定义状态映射。 例如，以下属性映射`FATAL`到503（服务不可用）：

```
management.health.status.http-mapping.FATAL=503
```

如果你需要更多的控制，你可以定义你自己的`HealthStatusHttpMapper` bean。

下表显示了内置状态的默认状态映射：

Status	Mapping
下	SERVICE_UNAVAILABLE（503）
暂停服务	SERVICE_UNAVAILABLE（503）
向上	默认情况下没有映射，所以http状态是200
未知	默认情况下没有映射，所以http状态是200

50.9.3 Reactive Health Indicators译：50.9.3反应性健康指标

对于无功的应用，比如使用Spring WebFlux那些`ReactiveHealthIndicator`提供了获取应用程序运行状况非阻塞的合同。 与传统的`HealthIndicator`类似，从您的`ApplicationContext`定义的所有`ReactiveHealthIndicator`豆中收集健康信息。 常规`HealthIndicator`不针对反应式API进行检查的bean将包含在弹性调度程序中并执行。

要从反应式API提供自定义健康信息，您可以注册实现`ReactiveHealthIndicator`接口的Spring bean。 以下代码显示了`ReactiveHealthIndicator`实现的示例：

```
@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()));
 }

}
```

要自动处理错误，请考虑从`AbstractReactiveHealthIndicator`延伸。

50.9.4 Auto-configured ReactiveHealthIndicators译：50.9.4自动配置的ReactiveHealthIndicators

适当时，以下`ReactiveHealthIndicators`由Spring Boot自动配置：

Name	描述
`MongoReactiveHealthIndicator`	检查Mongo数据库是否启动。
`RedisReactiveHealthIndicator`	检查Redis服务器是否启动。

必要时，反应性指标取代了常规指标。 此外，没有明确处理的任何`HealthIndicator`都会自动`HealthIndicator`。

50.10 Application Information译：1750650889275390 50.10应用信息

应用程序信息显示从您的`ApplicationContext`定义的所有`InfoContributor`豆收集的各种信息。 Spring Boot包含大量自动配置的`InfoContributor`豆，您可以编写自己的豆。

50.10.1 Auto-configured InfoContributors译：50.10.1自动配置InfoContributors

适当时，以下`InfoContributor` bean将由Spring Boot自动配置：

Name	描述
`EnvironmentInfoContributor`	在`info`密钥下公开`Environment`的`info`密钥。
`GitInfoContributor`	如果`git.properties`文件可用，则显示git信息。
`BuildInfoContributor`	如果`META-INF/build-info.properties`文件可用，则显示构建信息。

可以通过设置`management.info.defaults.enabled`属性来禁用它们。

50.10.2 Custom Application Information译：50.10.2自定义应用程序信息

您可以通过设置`info.*`弹簧属性来自定义`info`端点公开的数据。 `info`密钥下的所有`Environment`属性`info`自动公开。 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
```

而不是硬编码这些值，你也可以 expand info properties at build time 。
假设你使用Maven，你可以重写前面的例子，如下所示：

```
info.app.encoding=@project.build.sourceEncoding@
info.app.java.source=@java.version@
info.app.java.target=@java.version@
```

50.10.3 Git Commit Information 译：50.10.3 Git提交信息

在另一个有用的功能`info`端点是其公布有关的状态信息的能力`git`源代码库项目建成时。 如果`GitProperties` bean可用， `git.branch` `git.commit.id`和`git.commit.time`属性。

如果`git.properties`文件在类路径的根目录中可用，则`GitProperties` bean将自动配置。 有关更多详细信息，请参阅“ Generate git information ”。

如果要显示完整的git信息（即全部内容 `git.properties` ），请使用 `management.info.git.mode`属性，如下所示：

```
management.info.git.mode=full
```

50.10.4 Build Information 译：50.10.4构建信息

如果`BuildProperties` bean可用，则`info`端点还可以发布关于您的构建的信息。 如果`META-INF/build-info.properties`文件在类路径中可用，`META-INF/build-info.properties`发生这种情况。

Maven和Gradle插件都可以生成该文件。 有关更多详细信息，请参阅“ Generate build information ”。

50.10.5 Writing Custom InfoContributors 译：50.10.5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 `InfoContributor`接口的Spring bean。

以下示例使用一个值贡献 `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"));
 }

}
```

如果您到达 `info`端点，则应该看到包含以下附加条目的响应：

```
{
 "example": {
  "key" : "value"
 }
}
```

51. Monitoring and Management over HTTP 译：51.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进行公开。 默认约定是使用端点`id` ，其前缀为`/actuator`作为URL路径。 例如， `health`为`/actuator/health` 。

执行器本身支持Spring MVC，Spring WebFlux和Jersey。

51.1 Customizing the Management Endpoint Paths 译：51.1自定义管理端点路径

有时候，自定义管理端点的前缀非常有用。 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` ）。

除非管理端口已配置为expose endpoints by using a different HTTP port， `management.endpoints.web.base-path`与`server.servlet.context-path`。 如果`management.server.port`配置， `management.endpoints.web.base-path`是相对于`management.server.servlet.context-path`。

51.2 Customizing the Management Server Port译：51.2自定义管理服务器端口

通过使用默认的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，则可能希望使用不同的HTTP端口来公开端点。

您可以设置`management.server.port`属性来更改HTTP端口，如以下示例所示：

```
management.server.port=8081
```

51.3 Configuring Management-specific SSL译：51.3配置特定于管理的SSL

配置为使用自定义端口时，管理服务器也可以使用各种`management.server.ssl.*`属性配置自己的SSL。 例如，通过这样做，管理服务器可通过HTTP使用，而主应用程序使用HTTPS，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，但使用不同的密钥存储区，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
```

51.4 Customizing the Management Server Address译：51.4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。 如果您只想在内部网络或面向操作的网络上收听，或只收听来自`localhost`连接，则这样做会很有用。

只有当端口与主服务器端口不同时，您才可以监听其他地址。

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
```

51.5 Disabling HTTP Endpoints译：51.5禁用HTTP端点

如果您不想通过HTTP公开端点，则可以将管理端口设置为`-1`，如以下示例所示：

```
management.server.port=-1
```

52. Monitoring and Management over JMX译：52.通过JMX监控和管理

Java管理扩展（JMX）提供了一个标准机制来监视和管理应用程序。 默认情况下，Spring Boot将管理端点公开为`org.springframework.boot`域下的JMX MBean。

52.1 Customizing MBean Names译：52.1定制MBean名称

MBean的名称通常由端点的`id`生成。 例如， `health`端点显示为`org.springframework.boot:type=Endpoint,name=Health`。

如果您的应用程序包含多个Spring `ApplicationContext`，则可能会发现名称发生冲突。 为了解决这个问题，你可以设定`management.endpoints.jmx.unique-names`属性为`true`，这样的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`application.properties`一个示例：

```
management.endpoints.jmx.domain=com.example.myapp
management.endpoints.jmx.unique-names=true
```

52.2 Disabling JMX Endpoints译：52.2禁用JMX端点

如果您不想通过JMX公开端点，则可以将`management.endpoints.jmx.exposure.exclude`属性设置为`*`，如以下示例所示：

```
management.endpoints.jmx.exposure.exclude=*
```

52.3 Using Jolokia for JMX over HTTP译：52.3使用Jolokia进行JMX over HTTP

Jolokia是一个JMX-HTTP桥，它提供了访问JMX bean的另一种方法。 要使用Jolokia， `org.jolokia:jolokia-core`依赖项添加到`org.jolokia:jolokia-core`。 例

如，使用Maven，您可以添加以下依赖项：

```
<dependency>
 <groupId>org.jolokia</groupId>
 <artifactId>jolokia-core</artifactId>
 </dependency>
```

Jolokia端点可通过向`management.endpoints.web.exposure.include`属性添加`jolokia`或`*`来暴露。 然后，您可以通过在管理HTTP服务器上使用`/actuator/jolokia`来访问它。

52.3.1 Customizing Jolokia译：52.3.1定制 Jolokia

Jolokia有许多您通常会通过设置servlet参数进行配置的设置。 使用Spring Boot，您可以使用`application.properties`文件。 为此，请将参数前缀为`management.endpoint.jolokia.config.`，如以下示例所示：

```
management.endpoint.jolokia.config.debug=true
```

52.3.2 Disabling Jolokia译：52.3.2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
```

53. Loggers译：53.记录仪

Spring Boot Actuator包含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，该配置由明确配置的记录级别以及记录框架赋予的有效记录级别组成。 这些级别可以是以下之一：

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `OFF`
- `null`

`null`表示没有显式配置。

53.1 Configure a Logger译：53.1配置记录器

要配置给定的记录器，`POST`部分实体配置为资源的URI，如以下示例所示：

```
{
 "configuredLevel": "DEBUG"
}
```

要“记录”记录器的特定级别（并使用默认配置），可以将值`null`作为`configuredLevel`。

54. Metrics译：54.指标

Spring Boot Actuator为 Micrometer提供依赖管理和自动配置， 这是一个支持多种监控系统的应用指标外观，其中包括：

- Atlas
- Datadog
- Ganglia
- Graphite
- Influx
- JMX
- New Relic
- Prometheus
- SignalFx
- Simple (in-memory)
- StatsD
- Wavefront

要了解更多关于千分表的功能，请参阅其 reference documentation，特别是 concepts section。

54.1 Getting started译：54.1入门

Spring Boot会自动配置复合`MeterRegistry`，并为注册表添加到它在类路径中找到的每个受支持实现的组合。 在运行时类路径中依赖于`micrometer-registry-{system}`对于Spring Boot来配置注册表来说足够了。

大多数注册表具有共同的特征。 例如，即使Micrometer注册表实现位于类路径中，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`类的全局静态组合注册表中，除非您明确告诉它不要：

```
management.metrics.use-global-registry=false
```

您可以注册任意数量的`MeterRegistryCustomizer` bean，以便在向注册表注册任何仪表之前进一步配置注册表，例如应用通用标记：

```
@Bean
MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
 return registry -> registry.config().commonTags("region", "us-east-1");
}
```

您可以通过更具体地了解泛型类型来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);
}
```

通过该设置，您可以将`MeterRegistry`注入到组件中并注册指标：

```
@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
 }

}
```

还可以通过配置或专用注释标记来控制Spring Boot configures built-in instrumentation （即`MeterBinder`实现）。

54.2 Supported monitoring systems译：54.2支持的监测系统

54.2.1 Atlas译：54.2.1地图集

默认情况下，指标将导出到本地机器上运行的Atlas 。 Atlas server的使用位置可以通过以下方式提供：

```
management.metrics.export.atlas.uri=http://atlas.example.com:7101/api/v1/publish
```

54.2.2 Datadog译：54.2.2 Datadog

Datadog注册中心定期推送指标到datadoghq 。 要将度量标准导出到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
```

54.2.3 Ganglia译：54.2.3神经节

默认情况下，指标会导出到本地机器上运行的Ganglia 。 使用的Ganglia server主机和端口可以使用：

```
management.metrics.export.ganglia.host=ganglia.example.com
management.metrics.export.ganglia.port=9649
```

54.2.4 Graphite译：54.2.4石墨

默认情况下，指标会导出到本地机器上运行的Graphite 。 使用的Graphite server主机和端口可以使用：

```
management.metrics.export.graphite.host=graphite.example.com
management.metrics.export.graphite.port=9004
```

千分尺提供一个默认的`HierarchicalNameMapper` ，它管理一个尺寸计量器id是 mapped to flat hierarchical names 。

要控制此行为，请定义您的`GraphiteMeterRegistry`并提供您自己的`HierarchicalNameMapper` 。 除非您定义自己的配置， `Clock`将提供自动配置的`GraphiteConfig`和`Clock`豆类：

```
@Bean
public GraphiteMeterRegistry graphiteMeterRegistry(GraphiteConfig config, Clock clock) {
 return new GraphiteMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

54.2.5 Influx译：54.2.5流入

默认情况下，指标将导出到本地计算机上运行的Influx 。 使用的Influx server的位置可以使用：

```
management.metrics.export.influx.uri=http://influx.example.com:8086
```

54.2.6 JMX译：54.2.6 JMX

千分尺提供到JMX的分层映射，主要作为便捷便携的方式查看指标在本地。默认情况下，指标导出到`metrics` JMX域。 要使用的域可以通过以下方式提供：

```
management.metrics.export.jmx.domain=com.example.app.metrics
```

千分尺提供一个默认的 `HierarchicalNameMapper` ，它管理一个尺寸计量器id是 mapped to flat hierarchical names 。

为了控制这种行为，请定义您的`JmxMeterRegistry`并提供您自己的`HierarchicalNameMapper` 。 除非您自定义自己的配置， `Clock`将提供自动配置的`JmxConfig`和`Clock`豆类：

```
@Bean
public JmxMeterRegistry jmxMeterRegistry(JmxConfig config, Clock clock) {
 return new JmxMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

54.2.7 New Relic译：54.2.7新遗物

New Relic注册中心定期推送指标到New Relic 。 要将度量标准导出到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
```

您还可以更改度量标准发送到New Relic的时间间隔：

```
management.metrics.export.newrelic.step=30s
```

54.2.8 Prometheus译：54.2.8普罗米修斯

Prometheus预计会针对度量标准刮取或轮询单个应用实例。 Spring Boot提供了一个可在`/actuator/prometheus`处提供的执行器端点， `/actuator/prometheus`以适当的格式显示Prometheus scrape 。

该端点默认不可用，必须公开，请参阅 exposing endpoints了解更多详细信息。

以下是 `scrape_config`添加到 `prometheus.yml` ：

```
scrape_configs:
  - job_name: 'spring'
 metrics_path: '/actuator/prometheus'
 static_configs:
   - targets: ['HOST:PORT']
```

54.2.9 SignalFx译：54.2.9 SignalFx

SignalFx注册表定期推送指标到SignalFx 。 要将度量标准导出到SignalFx ，必须提供您的访问令牌：

```
management.metrics.export.signalfx.access-token=YOUR_ACCESS_TOKEN
```

您还可以更改度量标准发送到SignalFx的时间间隔：

```
management.metrics.export.signalfx.step=30s
```

54.2.10 Simple译：54.2.10简单

千分尺附带一个简单的内存后端，如果没有配置其他注册表，将自动用作后备。 这使您可以查看metrics endpoint中收集的指标 。

只要您使用任何其他可用的后端，内存后端就会自行禁用。 您也可以显式禁用它：

```
management.metrics.export.simple.enabled=false
```

54.2.11 StatsD译：54.2.11 StatsD

StatsD注册表通过UDP将度量标准推向StatsD代理。 默认情况下，指标将导出到本地计算机上运行的StatsD代理。 需要使用的StatsD代理主机和端口可以使用：

```
management.metrics.export.statsd.host=statsd.example.com
management.metrics.export.statsd.port=9125
```

您还可以更改StatsD行协议以使用（默认为Datadog）：

```
management.metrics.export.statsd.flavor=etsy
```

54.2.12 Wavefront译：54.2.12波前

Wavefront注册表定期推送指标到Wavefront 。 如果您直接将度量标准导出到Wavefront ，则必须提供您的API令牌：

```
management.metrics.export.wavefront.api-token=YOUR_API_TOKEN
```

或者，您可以使用在您的环境中设置的Wavefront边车或内部代理，将度量数据转发给Wavefront API主机：

```
management.metrics.export.wavefront.uri=proxy://localhost:2878
```

如果将度量标准发布到Wavefront代理（如 the documentation中所述 ），则主机必须采用 `proxy://HOST:PORT` 格式。

您还可以更改度量标准发送到Wavefront的时间间隔：

```
management.metrics.export.wavefront.step=30s
```

54.3 Supported Metrics 译：54.3支持的度量标准

Spring适用时会注册以下核心指标：

- JVM指标，报告使用情况：
  - Various memory and buffer pools
  - Statistics related to garbage collection
  - Threads utilization
  - Number of classes loaded/unloaded
- CPU metrics
- File descriptor metrics
- Logback metrics: record the number of events logged to Logback at each level
- Uptime metrics: report a gauge for uptime and a fixed gauge representing the application's absolute start time
- Tomcat metrics
- Spring Integration metrics

54.3.1 Spring MVC Metrics 译：54.3.1 Spring MVC度量标准

自动配置使得能够检测由Spring MVC处理的请求。 `management.metrics.web.server.auto-time-requests` 为 `true` ，此检测会针对所有请求进行。 或者，如果设置为 `false` ，则可以通过将 `@Timed` 添加到请求处理方法来启用检测：

```
@RestController
@Timed ❶
public class MyController {

 @GetMapping("/api/people")
 @Timed(extraTags = { "region", "us-east-1" }) ❷
 @Timed(value = "all.people", longTask = true) ❸
 public List<Person> listPeople() { ... }

}
```

❶ 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。
❷ 一种启用单个端点的方法。 如果您在课堂上拥有此功能，则这不是必需的，但可用于进一步自定义此特定终端的计时器。
❸ 使用 `longTask = true` 方法启用长任务计时器的方法。 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。 该名称可以通过设置 `management.metrics.web.server.requests-metric-name` 属性进行自定义。

默认情况下，与Spring MVC相关的度量标记包含以下信息：

- `method` , the request's method (for example, `GET` or `POST`).
- `uri` , the request's URI template prior to variable substitution, if possible (for example, `/api/person/{id}`).
- `status` , the response's HTTP status code (for example, `200` or `500`).
- `exception` , the simple class name of any exception that was thrown while handling the request.

要自定义标签，提供 `@Bean` 实现 `WebMvcTagsProvider` 。

54.3.2 Spring WebFlux Metrics 译：54.3.2 Spring WebFlux度量

自动配置可以检测由WebFlux控制器和功能处理程序处理的所有请求。

默认情况下，将使用名称 `http.server.requests` 生成度量标准。 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相关的度量标记包含以下信息：

- `method` , the request's method (for example, `GET` or `POST`).
- `uri` , the request's URI template prior to variable substitution, if possible (for example, `/api/person/{id}`).
- `status` , the response's HTTP status code (for example, `200` or `500`).
- `exception` , the simple class name of any exception that was thrown while handling the request.

要自定义标签，提供 `@Bean` 实现 `WebFluxTagsProvider` 。

54.3.3 RestTemplate Metrics 译：54.3.3 RestTemplate指标

任何使用自动配置的 `RestTemplateBuilder` 创建的 `RestTemplate` 的工具 `RestTemplateBuilder` 启用。 也可以手动应用 `MetricsRestTemplateCustomizer` 。

默认情况下，将使用名称 `http.client.requests` 生成度量标准。 该名称可以通过设置 `management.metrics.web.client.requests-metric-name` 属性进行自定义。

默认情况下，由仪表 `RestTemplate` 生成的度量标记包含以下信息：

- `method` , the request's method (for example, `GET` or `POST`).
- `uri` , the request's URI template prior to variable substitution, if possible (for example, `/api/person/{id}`).
- `status` , the response's HTTP status code (for example, `200` or `500`).
- `clientName` , the host portion of the URI.

要自定义标签，提供 `@Bean` 实现 `RestTemplateExchangeTagsProvider` 。 `RestTemplateExchangeTags` 有便利的静态函数。

54.3.4 Cache Metrics 译：54.3.4高速缓存指标

自动配置使所有可用的仪器`Cache` S于启动与前缀指标`cache` 。 高速缓存检测是针对一组基本指标进行标准化的。 另外，缓存特定的指标也可用。

以下缓存库受支持：

- Caffeine
- EhCache 2
- Hazelcast
- Any compliant JCache (JSR-107) implementation

度量标准由缓存的名称和从bean名称派生的 `CacheManager`的名称进行 `CacheManager` 。

只有启动时可用的缓存绑定到注册表。 对于在启动阶段之后即时创建或以编程方式创建的缓存，需要进行显式注册。 一个`CacheMetricsRegistrar`豆可用于使该过程更容易。

54.3.5 DataSource Metrics 译：54.3.5数据源度量标准

通过自动配置，可以使用名为`jdbc`的指标检测所有可用的`DataSource`对象。 数据源检测会生成代表池中当前活动，最大允许和最小允许连接的量表。 每个量表都有一个名称，前缀为`jdbc` 。

度量标准还通过基于bean名称计算出的 `DataSource`的名称进行标记。

默认情况下，Spring Boot为所有支持的数据源提供元数据; 如果您最喜爱的数据源不支持开箱即用，您可以添加额外的`DataSourcePoolMetadataProvider`豆。 例子见`DataSourcePoolMetadataProvidersConfiguration` 。

另外，Hikari特定的度量标准`hikaricp`前缀。 每个指标都由池的名称标记（可以用`spring.datasource.name`来控制）。

54.3.6 RabbitMQ Metrics 译：54.3.6 RabbitMQ度量标准

自动配置将启用所有可用的RabbitMQ连接工厂的检测，命名为 `rabbitmq` 。

54.4 Registering custom metrics 译：54.4注册自定义指标

要注册自定义指标， `MeterRegistry`注入到组件中，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);
 }

 // …

}
```

如果您发现您不断在组件或应用程序中测量一套度量标准，则可以将此套件封装在`MeterBinder`实现中。 默认情况下，所有`MeterBinder` bean的指标都将自动绑定到Spring管理的`MeterRegistry` 。

54.5 Customizing individual metrics 译：54.5自定义各个指标

如果您需要将自定义应用于特定的`Meter`实例，则可以使用`io.micrometer.core.instrument.config.MeterFilter`界面。 默认情况下，所有`MeterFilter`豆将被自动应用到千分尺`MeterRegistry.Config` 。

例如，如果你想重命名 `mytag.region`标签 `mytag.area`为开头的每一米的ID `com.example` ，你可以做到以下几点：

```
@Bean
public MeterFilter renameRegionTagMeterFilter() {
 return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");
}
```

54.5.1 Per-meter properties 译：54.5.1每米性能

除了`MeterFilter`豆以外，还可以使用属性以每米为单位应用有限的一组定制。 每米定制适用于以给定名称开头的所有仪表ID。 例如，以下内容将禁用所有以`example.remote`开头的ID

```
management.metrics.enable.example.remote=false
```

以下属性允许每米自定义：

表54.1。 每米自定义

Property	描述
`management.metrics.enable`	是否拒绝发布任何指标。
`management.metrics.distribution.percentiles-histogram`	是否发布适合计算可聚合（跨维）百分比近似值的直方图。
`management.metrics.distribution.percentiles`	发布您的应用程序中计算的百分比值
`management.metrics.distribution.sla`	用SLA定义的桶发布累积直方图。

有关概念背后的更多详细信息 `percentiles-histogram` ， `percentiles` 和 `sla` 指 "Histograms and percentiles" section微米文档。

54.6 Metrics endpoint译: 54.6度量标准终点

Spring Boot提供了一个 `metrics` 端点，可以用于诊断以检查应用程序收集的指标。 该端点默认不可用，必须公开，请参阅exposing endpoints了解更多详细信息。

导航到 `/actuator/metrics` 会显示可用仪表名称的列表。 您可以通过提供其名称作为选择器来深入查看关于特定仪表的信息，例如 `/actuator/metrics/jvm.memory.max` 。

> 这里使用的名称应该与代码中使用的名称相匹配，而不是命名后的名称 - 约定为其运输到的监视系统。 换句话说，如果 `jvm.memory.max` 由于其蛇形命名约定而在Prometheus中显示为 `jvm_memory_max` ，则在检查 `metrics` 端点中的仪表时，仍然应该使用 `jvm.memory.max` 作为选择器。

您还可以将任意数量的 `tag=KEY:VALUE` 查询参数添加到URL的末尾，以在计量器上进行维度向下钻取，例如 `/actuator/metrics/jvm.memory.max?tag=area:nonheap` 。

> 报告的测量结果是与仪表名称匹配的所有仪表的统计数据和已经应用的所有标记的*总和* 。 因此，在上面的示例中，返回的“Value”统计量是堆的“代码缓存”，“压缩类空间”和“元空间”区域的最大内存空间总和。 如果您只想查看“Metaspace”的最大大小，则可以添加额外的 `tag=id:Metaspace` ，即 `/actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace` 。

55. Auditing译: 55.审计

一旦Spring Security发挥作用，Spring Boot Actuator就会有一个灵活的审计框架来发布事件（默认情况下，“认证成功”，“失败”和“拒绝访问”例外）。 此功能对于报告和实施基于认证失败的锁定策略非常有用。 要定制已发布的安全事件，您可以提供自己的实现 `AbstractAuthenticationAuditListener` 和 `AbstractAuthorizationAuditListener` 。

您也可以将审计服务用于您自己的业务事件。 要做到这一点，无论是注入现有 `AuditEventRepository` 成自己的组件，并直接使用或发布一个 `AuditApplicationEvent` 同春 `ApplicationEventPublisher` （通过实施 `ApplicationEventPublisherAware` ）。

56. HTTP Tracing译: 56. HTTP跟踪

所有HTTP请求都会自动启用跟踪。 您可以查看 `httptrace` 端点并获取有关最后100个请求 - 响应交换的基本信息。

56.1 Custom HTTP tracing译: 56.1自定义HTTP跟踪

要自定义每条跟踪中包含的项目，请使用 `management.trace.http.include` 配置属性。

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，则可以定义自己的 `InMemoryHttpTraceRepository` bean实例。 您也可以创建自己的替代 `HttpTraceRepository` 实现。

57. Process Monitoring译: 57.过程监控

在 `spring-boot` 模块中，可以找到两个类来创建通常用于进程监视的文件：

- `ApplicationPidFileWriter` creates a file containing the application PID (by default, in the application directory with a file name of `application.pid`).
- `WebServerPortFileWriter` creates a file (or files) containing the ports of the running web server (by default, in the application directory with a file name of `application.port`).

默认情况下，这些编写器不会被激活，但您可以启用：

- By Extending Configuration
- Section 57.2, “Programmatically”

57.1 Extending Configuration译: 57.1扩展配置

在 `META-INF/spring.factories` 文件中，可以激活写入PID文件的侦听器，如下例所示：

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.context.ApplicationPidFileWriter,\
org.springframework.boot.web.context.WebServerPortFileWriter
```

57.2 Programmatically译: 57.2以编程方式

您还可以通过调用 `SpringApplication.addListeners(…)` 方法并传递适当的 `Writer` 对象来激活侦听器。 此方法还允许您自定义 `Writer` 构造函数中的文件名和路径。

58. Cloud Foundry Support译: 58. Cloud Foundry支持

Spring Boot的执行器模块包含在部署到兼容的Cloud Foundry实例时激活的其他支持。 `/cloudfoundryapplication` 路径为所有 `@Endpoint` 豆提供了替代安全路线。

通过扩展支持，Cloud Foundry管理用户界面（例如，可用于查看已部署应用程序的Web应用程序）可通过Spring Boot执行程序信息进行扩展。 例如，应用程序状态页面可能包含完整的健康信息，而不是典型的“正在运行”或“已停用”状态。

> 常规用户无法直接访问 `/cloudfoundryapplication` 路径。 为了使用端点，必须将有效的UAA令牌与请求一起传递。

58.1 Disabling Extended Cloud Foundry Actuator Support译：58.1禁用扩展Cloud Foundry执行器支持

如果要完全禁用`/cloudfoundryapplication`端点，则可以将以下设置添加到`application.properties`文件中：

application.properties。

```
management.cloudfoundry.enabled=false
```

58.2 Cloud Foundry Self-signed Certificates译：58.2 Cloud Foundry自签名证书

默认情况下，`/cloudfoundryapplication`端点的安全验证`/cloudfoundryapplication`各种Cloud Foundry服务进行SSL调用。 如果您的Cloud Foundry UAA或云控制器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
```

58.3 Custom context path译：58.3自定义上下文路径

如果服务器的上下文路径已配置为`/`以外的其他任何内容，则Cloud Foundry端点将无法在应用程序的根目录中使用。 例如，如果`server.servlet.context-path=/app`，Cloud Foundry端点将在`/app/cloudfoundryapplication/*`处`/app/cloudfoundryapplication/*`。

如果您期望Cloud Foundry端点始终在`/cloudfoundryapplication/*`处`/cloudfoundryapplication/*`，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该端点。 根据使用的网络服务器，配置会有所不同。 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("/*");
 };
}
```

59. What to Read Next译：59.接下来要读什么

如果你想探索本章讨论的一些概念，你可以看看执行器sample applications 。 你也可能想阅读关于图形工具，如Graphite 。

否则，你就可以继续，阅读有关 'deployment options'或直接跳到了一些深入的关于SpringBootâ€™的信息 *build tool plugins* 。

Part VI. Deploying Spring Boot Applications译：第六部分．部署Spring Boot应用程序

Spring Boot的灵活打包选项在部署应用程序时提供了大量选择。 您可以将Spring Boot应用程序部署到各种云平台，容器映像（如Docker）或虚拟/真实机器。

本节介绍一些更常见的部署方案。

60. Deploying to the Cloud译：60.部署到云

Spring Boot的可执行文件夹可供大多数常用云PaaS（平台即服务）提供商使用。 这些提供者倾向于要求你“创造你自己的容器”。 他们管理应用程序进程（特别是Java应用程序），因此他们需要一个中间层，使*您的*应用程序适应*云*对正在运行的进程*的*看法。

两家流行的云提供商Heroku和Cloud Foundry采用了“buildpack”方法。 buildpack将您部署的代码封装在*启动*应用程序所需的任何代码中。 它可能是JDK和`java`的调用，嵌入式Web服务器或完整的应用程序服务器。 buildpack是可插入的，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，您的应用程序就像Spring Boot可执行程序jar一样，具有打包的所有内容。

在本节中，我们将看看如何使“ 入门”部分中的simple application that we developed在云中运行。

60.1 Cloud Foundry 译：60.1 Cloud Foundry

如果没有指定其他buildpack，Cloud Foundry会提供默认构建包。 Cloud Foundry Java buildpack对Spring应用程序（包括Spring Boot）提供了出色的支持。 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`）并拥有installed the `cf` command line tool后，请使用`cf push`命令部署应用程序，将路径替换为已编译的`.jar`。 推送应用程序之前一定要有logged in with your `cf` command line client。 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
```

在前面的例子中，我们用`acloudyspringtime`替代`cf`作为应用程序的名称。

请参阅`cf push` documentation了解更多选项。 如果在同一目录中存在Cloud Foundry `manifest.yml`文件，则会考虑该文件。

此时，`cf`开始上传您的应用程序，产生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-
       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/cor
       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-
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！ 该应用程序现在已经生效！

一旦您的应用程序处于运行状态，您就可以使用`cf apps`命令来验证已部署的应用程序的状态，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完毕，您应该能够在给定的URI处找到该应用程序。 在前面的例子中，你可以在`http://acloudyspringtime.cfapps.io/`找到它。

60.1.1 Binding to Services 译：60.1.1绑定到服务

默认情况下，关于正在运行的应用程序以及服务连接信息的元数据作为环境变量向应用程序公开（例如：`$VCAP_SERVICES`）。 此架构决定归功于Cloud Foundry的多语言支持（任何语言和平台均可作为构建包支持）。 进程范围的环境变量是语言不可知的。

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平滑到可通过Spring的`Environment`抽象访问的属性中，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属性的前缀为`vcap`。 您可以使用`vcap`属性来访问应用程序信息（例如应用程序的公用URL）和服务信息（如数据库凭证）。 有关完整的详细信息，请参阅'CloudFoundryVcapEnvironmentPostProcessor' Javadoc。

Spring Cloud Connectors项目更适合配置数据源等任务。 Spring Boot包含自动配置支持和`spring-boot-starter-cloud-connectors`启动器。

60.2 Heroku 译：60.2 Heroku

Heroku是另一个流行的PaaS平台。 要自定义Heroku版本，您提供了一个`Procfile`，它提供了部署应用程序所需的咒语。 Heroku分配一个`port`供Java应用程序使用，然后确保到外部URI的路由工作。

您必须配置您的应用程序以侦听正确的端口。 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`参数成为可从Spring `Environment`实例访问的属性。 配置属性`server.port`被馈送到嵌入式Tomcat，Jetty或Undertow实例，然后在启动时使用该端口。 Heroku PaaS为我们分配了`$PORT`环境变量。

这应该是你需要的一切。 Heroku部署的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上运行。

60.3 OpenShift 译：60.3 OpenShift

OpenShift是Kubernetes容器编排平台的红帽公共（和企业）扩展。 与Kubernetes类似，OpenShift有许多用于安装基于Spring Boot的应用程序的选项。

OpenShift有许多描述如何部署Spring Boot应用程序的资源，其中包括：

- Using the S2I builder
- Architecture guide
- Running as a traditional web application on Wildfly
- OpenShift Commons Briefing

60.4 Amazon Web Services (AWS) 译：60.4亚马逊网络服务（AWS）

亚马逊网络服务提供了多种方式来安装基于Spring Boot的应用程序，既可以作为传统的Web应用程序（war），也可以作为带有嵌入式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 Container Registry

每个都有不同的功能和定价模式。 在本文中，我们只描述最简单的选项：AWS Elastic Beanstalk。

60.4.1 AWS Elastic Beanstalk 译：60.4.1 AWS Elastic Beanstalk

如官方Elastic Beanstalk Java guide所述，部署Java应用程序有两个主要选项。 你可以使用“Tomcat平台”或“Java平台”。

Using the Tomcat Platform 译：使用Tomcat平台

此选项适用于生成war文件的Spring Boot项目。 不需要特殊配置。 你只需要遵循官方指南。

Using the Java SE Platform 译：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
```

默认情况下，Elastic Beanstalk会上传源并在AWS中编译它们。 但是，最好是上传二进制文件。 为此，请在`.elasticbeanstalk/config.yml`文件中添加类似于以下内容的`.elasticbeanstalk/config.yml`：

```
deploy:
 artifact: target/demo-0.0.1-SNAPSHOT.jar
```

默认情况下，Elastic Beanstalk环境负载均衡。 负载平衡器有很大的成本。 为避免这种成本，请将环境类型设置为“单一实例”，如the Amazon documentation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
```

60.4.2 Summary 译：60.4.2总结

这是获得AWS的最简单方法之一，但是还有更多内容需要解决，比如如何将Elastic Beanstalk集成到任何CI / CD工具中，使用Elastic Beanstalk Maven插件而不是CLI等等。 有一个blog post更详细地涵盖了这些话题。

60.5 Boxfuse and Amazon Web Services 译：60.5 Boxfuse和亚马逊网络服务

Boxfuse通过将您的Spring Boot可执行程序jar或war转换为可以在VirtualBox或AWS上以未改变的方式部署的最小VM镜像来工作。 Boxfuse为Spring Boot提供了深度集成，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将这些信息用于它生成的图像以及它提供的所有资源（实例，安全组，弹性负载均衡器等）。

创建 Boxfuse account后 ，将其连接到您的AWS账户，安装最新版本的Boxfuse Client，并确保该应用程序由Maven或Gradle构建（例如使用 `mvn clean package` ），则可以部署Spring使用与以下类似的命令将应用程序引导至AWS：

```
$ boxfuse run myapp-1.0.jar -env=prod
```

请参阅`boxfuse run` documentation了解更多选项。 如果当前目录中存在`boxfuse.conf`文件，则会考虑该文件。

默认情况下， `boxfuse`在启动时激活名为`boxfuse`的Spring配置文件。 如果您的可执行jar或war包含`application-boxfuse.properties`文件，Boxfuse将其配置基于其包含的属性。

此时， `boxfuse`会为您的应用程序创建一个映像，并上传它，并在AWS上配置并启动必要的资源，从而得到类似于以下示例的输出结果：

```
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上运行。

请参阅 deploying Spring Boot apps on EC2上的博 文以及 documentation for the Boxfuse Spring Boot integration开始使用Maven构建来运行该应用程序。

60.6 Google Cloud 译：60.6 Google Cloud

Google Cloud有几个可用于启动Spring Boot应用程序的选项。 最容易入门的可能是App Engine，但您也可以找到在容器中使用Container Engine或在具有Compute Engine的虚拟机上运行Spring Boot的方法。

要在App Engine中运行，您可以先在用户界面中创建一个项目，为您设置唯一的标识符并设置HTTP路由。 将Java应用程序添加到项目中，并将其保留为空，然后使用Google Cloud SDK将Spring Boot应用程序从命令行或CI构建推入该插槽。

App Engine标准要求您使用WAR包装。 遵循these steps将App Engine标准应用程序部署到Google Cloud。

另外，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`，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>
```

然后用 `mvn appengine:deploy` 部署（如果您需要首先进行身份验证，则生成失败）。

61. Installing Spring Boot Applications 译：61.安装Spring Boot应用程序

除了使用 `java -jar` 运行Spring Boot应用程序 `java -jar` ，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，也可以是registered with `init.d` or `systemd` 。 这使得在普通生产环境中安装和管理Spring Boot应用程序变得非常简单。

Caution

通过在文件的前面嵌入一个额外的脚本，完全可执行的jar工作。 目前，有些工具不接受这种格式，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war。 建议您仅在您打算直接执行jar或war时才使其可执行，而不是将其与 `java -jar` 一起 `java -jar` 或将其部署到servlet容器。

要用Maven创建一个“可执行的”jar，使用下面的插件配置：

```
<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，您可以键入 `./my-application.jar` （其中 `my-application` 是您的工件的名称）来运行应用程序。 包含jar的目录被用作应用程序的工作目录。

61.1 Supported Operating Systems 译：61.1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` 。

61.2 Unix/Linux Services 译：61.2 Unix / Linux服务

Spring Boot应用程序可以通过使用 `init.d` 或 `systemd` 作为Unix / Linux服务轻松启动。

61.2.1 Installation as an `init.d` Service (System V) 译：61.2.1作为 `init.d` 服务安装（系统V）

如果您将Spring Boot的Maven或Gradle插件配置为生成fully executable jar ，并且您不使用自定义 `embeddedLaunchScript` ，则可以将您的应用程序用作 `init.d` 服务。 要做到这一点，符号链接罐子 `init.d` 支持标准 `start` ， `stop` ， `restart` ，并 `status` 命令。

该脚本支持以下功能：

- Starts the services as the user that owns the jar file
- Tracks the application’s PID by using `/var/run/<appname>/<appname>.pid`
- Writes console logs to `/var/log/<appname>.log`

假设您在 `/var/myapp` 中安装了Spring Boot应用程序， `init.d` Spring Boot应用程序安装为 `init.d` 服务，请创建符号链接，如下所示：

```
$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp
```

安装后，您可以按照通常的方式启动和停止服务。 例如，在基于Debian的系统上，可以使用以下命令启动它：

```
$ service myapp start
```

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`/var/log/<appname>.log`存在错误。

您还可以使用标准操作系统工具将应用程序标记为自动启动。 例如，在Debian上，您可以使用以下命令：

```
$ update-rc.d myapp defaults <priority>
```

Securing an `init.d` Service 译:确保`init.d`服务的安全

以下是关于如何保护作为init.d服务运行的Spring Boot应用程序的一组准则。 它并不打算成为一个应用程序及其运行环境的完整列表。

以root身份执行时，与使用root启动init.d服务的情况相同，缺省可执行脚本以拥有该jar文件的用户身份运行应用程序。 你永远不应该运行Spring Boot应用程序`root`，所以你的应用程序的jar文件不应该被root所有。 相反，创建一个特定用户来运行应用程序，并使用`chown`使其成为jar文件的所有者，如下例所示：

```
$ chown bootapp:bootapp your-app.jar
```

在这种情况下，默认的可执行脚本以`bootapp`用户的身份运行应用程序。

为了减少应用程序的用户帐户被攻破的可能性，您应该考虑防止它使用登录shell。 例如，您可以将帐户的外壳设置为`/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，以便它不能被写入，只能由其所有者读取或执行，如下例所示：

```
$ chmod 500 your-app.jar
```

其次，如果您的应用程序或运行该应用程序的帐户遭到入侵，您还应采取措施限制损害。 如果攻击者获得访问权限，他们可以将jar文件写入并更改其内容。 防止这种情况的一种方法是通过使用`chattr`使其不可变，如以下示例所示：

```
$ sudo chattr +i your-app.jar
```

这将阻止包括root在内的任何用户修改jar。

如果使用root来控制应用程序的服务，并且use a `.conf` file自定义其启动，则`.conf`文件将被root用户读取和评估。 它应该得到相应的保证。 使用`chmod`以便该文件只能由所有者读取，并使用`chown`来使所有者为root，如以下示例所示：

```
$ chmod 400 your-app.conf
$ sudo chown root:root your-app.conf
```

61.2.2 Installation as a `systemd` Service 译:61.2.2作为`systemd`服务安装

`systemd`是System V init系统的继承者，现在正被许多现代Linux发行版使用。 虽然可以继续使用`init.d`脚本与`systemd`，还可以通过使用来启动春季启动应用`systemd` â€~serviceâ€™脚本。

假设您在`/var/myapp`中安装了Spring Boot应用程序， `systemd` Spring Boot应用程序安装为`systemd`服务，请创建一个名为`myapp.service`的脚本并将`myapp.service`放置在`/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
```

Important

请记住，改变`描述`， `User`，并`ExecStart`领域为您的应用。

`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`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“服务”脚本中的相应字段进行配置。 有关更多详细信息，请参阅service unit configuration man page 。

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
```

有关更多详细信息，请参阅`man systemctl`。

61.2.3 Customizing the Startup Script 译:61.2.3自定义启动脚本

由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，使用默认脚本以及一些自定义功能通常就足够了。 如果您发现无法自定义您需要的内容，请使用`embeddedLaunchScript`选项完全编写自己的文件。

Customizing the Start Script when It Is Written 译:在写入时自定义启动脚本

在写入jar文件时，定制启动脚本的元素通常很有意义。 例如，init.d脚本可以提供一个“描述”。 由于您知道前面的描述（并且不需要更改），因此您可以在生成jar时提供它。

要定制书写元素，请使用Spring Boot Maven或Gradle插件的 `embeddedLaunchScriptProperties` 选项。

默认脚本支持以下属性替换：

Name	描述
`mode`	脚本模式。 默认为 `auto` 。
`initInfoProvides`	“ `Provides` ”的 `Provides` 部分。 默认为 `spring-boot-application` 的摇篮，并 `${project.artifactId}` 对Maven。
`initInfoRequiredStart`	“ `Required-Start` ”的 `Required-Start` 部分。 默认为 `$remote_fs $syslog $network` 。
`initInfoRequiredStop`	“ `Required-Stop` ”的 `Required-Stop` 部分。 默认为 `$remote_fs $syslog $network` 。
`initInfoDefaultStart`	“ `Default-Start` ”的 `Default-Start` 部分。 默认为 `2 3 4 5` 。
`initInfoDefaultStop`	“ `Default-Stop` ”的 `Default-Stop` 部分。 默认为 `0 1 6` 。
`initInfoShortDescription`	“ `Short-Description` ”的 `Short-Description` 部分。 默认为 `Spring Boot Application` 的摇篮，并 `${project.name}` 对Maven。
`initInfoDescription`	“ `描述` ”的 `描述` 部分。 默认为 `Spring Boot Application` 的摇篮和 `${project.description}` （回落至 `${project.name}` ）对Maven。
`initInfoChkconfig`	“ `chkconfig` ”的 `chkconfig` 部分。 默认为 `2345 99 01` 。
`confFolder`	默认值为 `CONF_FOLDER` 。 默认为包含jar的文件夹。
`inlinedConfScript`	引用应在默认启动脚本中内联的文件脚本。 这可用于在加载任何外部配置文件之前设置环境变量（如 `JAVA_OPTS` 。
`logFolder`	默认值为 `LOG_FOLDER` 。 仅对 `init.d` 服务有效。
`logFilename`	默认值为 `LOG_FILENAME` 。 仅对 `init.d` 服务有效。
`pidFolder`	默认值为 `PID_FOLDER` 。 仅对 `init.d` 服务有效。
`pidFilename`	`PID_FOLDER` 中的PID文件名称的默认值。 仅对 `init.d` 服务有效。
`useStartStopDaemon`	`start-stop-daemon` 命令是否可用，应该用于控制过程。 默认为 `true` 。
`stopWaitTime`	默认值为 `STOP_WAIT_TIME` 。 仅对 `init.d` 服务有效。 默认为60秒。

Customizing a Script When It Runs 译:运行时自定义脚本

对于在写入jar *之后*需要定制的脚本项目，可以使用环境变量或 config file 。

默认脚本支持以下环境属性：

Variable	描述
`MODE`	操作的“模式”。 缺省值取决于jar的构建方式，但通常为 `auto` （意思是通过检查它是否为 `init.d` 目录中的符号链接来尝试猜测它是否为init脚本）。 如果要在前台运行脚本，可以明确将其设置为 `service` 以便 `stop|start|status|restart` 命令可以工作，或者设置为 `run` 。
`USE_START_STOP_DAEMON`	`start-stop-daemon` 命令是否可用，应该用来控制过程。 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-file相同的文件夹）。
`LOG_FILENAME`	在日志文件的名称 `LOG_FOLDER` （ `<appname>.log` 默认情况下）。
`APP_NAME`	应用的名称。 如果jar是从符号链接运行的，则脚本会猜测应用程序名称。 如果它不是符号链接或者您想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`	`java` 可执行文件的位置默认情况下是使用 `PATH` 发现的，但如果在 `$JAVA_HOME/bin/java` 处有可执行文件，则可以明确地设置它。
`JAVA_OPTS`	启动时传递给JVM的选项。
`JARFILE`	jar文件的显式位置，以防脚本被用来启动一个实际上没有嵌入的jar。
`DEBUG`	如果不为空，则在shell进程上设置 `-x` 标志，以便查看脚本中的逻辑。

Variable	描述
`STOP_WAIT_TIME`	在强制关闭之前停止应用程序时等待的时间（以秒为单位）（默认为 `60` ）。

该`PID_FOLDER`， `LOG_FOLDER`，并`LOG_FILENAME`变量仅适用于一个`init.d`服务。 对于`systemd`，通过使用“服务”脚本进行等效的自定义。 有关更多详细信息，请参阅service unit configuration man page 。

除`JARFILE`和`APP_NAME`，上一节中列出的设置可以使用`.conf`文件进行配置。 该文件预计将在jar文件旁边，并具有相同的名称，但后缀`.conf`而不是`.jar` 。 例如，一个名为罐子`/var/myapp/myapp.jar`使用名为配置文件`/var/myapp/myapp.conf`，如图以下示例：

myapp.conf。

```
JAVA_OPTS=-Xmx1024M
LOG_FOLDER=/custom/log/folder
```

如果你不喜欢在jar文件旁边有配置文件，你可以设置一个`CONF_FOLDER`环境变量来自定义配置文件的位置。

要了解有关适当保护此文件的信息，请参阅 the guidelines for securing an init.d service 。

61.3 Microsoft Windows Services译：61.3 Microsoft Windows服务

Spring Boot应用程序可以通过使用 `winsw`作为Windows服务 启动 。

A（ separately maintained sample ）逐步介绍如何为Spring Boot应用程序创建Windows服务。

62. What to Read Next译：62接下来要读什么

退房Cloud Foundry， Heroku， OpenShift，并Boxfuse关于该种功能，PaaS的可提供更多信息的网站。 这些只是四个最流行的Java PaaS提供商。 由于Spring Boot非常适合基于云的部署，因此您可以自由考虑其他提供商。

下一部分继续讨论 *Spring Boot CLI*，或者您可以跳到 *build tool plugins* 。

Part VII. Spring Boot CLI译：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，您可以使用它。 它可以让你运行Groovy脚本，这意味着你有一个熟悉的类Java语法，没有太多的样板代码。 您也可以引导一个新项目或编写自己的命令。

63. Installing the CLI译：63.安装CLI

Spring Boot CLI（命令行界面）可以使用SDKMAN手动安装！ （SDK管理器），或者如果您是OSX用户，则使用Homebrew或MacPorts。 有关全面的安装说明，请参见“*入门*”部分中的Section 10.2, “Installing the Spring Boot CLI”。

64. Using the CLI译：64.使用CLI

一旦你安装了CLI，你可以通过输入`spring`并在命令行按Enter键来运行它。 如果不带任何参数运行`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
```

您可以输入 `spring help`以获取有关任何受支持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 v2.0.3.RELEASE
```

64.1 Running Applications with the CLI 译：64.1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完全独立，因此您不需要任何外部Groovy安装。

以下示例显示了用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`--`将命令从“spring”命令参数中分离出来，如下例所示：

```
$ 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，如下例所示：

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

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。

64.1.1 Deduced “grab” Dependencies 译：64.1.1推导出“grab”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关系。 这个有用的技术可以让Groovy像Maven或Gradle一样下载jar，但不需要使用构建工具。

Spring Boot进一步扩展了这一技术，并尝试根据您的代码推导出哪些库为“grab”。 例如，由于之前显示的`WebApplication`代码使用了`@RestController`注释，Spring Boot抓住了“Tomcat”和“Spring MVC”。

以下项目用作“抓取提示”：

Items	Grabs
`JdbcTemplate`， `NamedParameterJdbcTemplate`， `DataSource`	JDBC应用程序。
`@EnableJms`	JMS应用程序。
`@EnableCaching`	缓存抽象。
`@Test`	JUnit的。
`@EnableRabbit`	RabbitMQ的。
延伸至`Specification`	Spock测试。
`@EnableBatchProcessing`	春天的批次。
`@MessageEndpoint` `@EnableIntegration`	Spring集成。
`@Controller` `@RestController` `@EnableWebMvc`	Spring MVC +嵌入式Tomcat。
`@EnableWebSecurity`	Spring Security。
`@EnableTransactionManagement`	春季交易管理。

请参阅Spring Boot CLI源代码中的`CompilerAutoConfiguration`的子类，以准确了解自定义如何应用。

64.1.2 Deduced “grab” Coordinates 译：64.1.2推导出“grab”坐标

Spring Boot通过让您指定不带组或版本的依赖关系（例如，`@Grab('freemarker')`），扩展了Groovy的标准`@Grab`支持。 这样做可以咨询Spring Boot的默认依赖关系元数据来推断工件的组和版本。

默认元数据与您使用的CLI版本相关联。 它只会在您转移到CLI的新版本时才会更改，从而使您可以控制何时可能更改您的依赖项的版本。 显示默认元数据中所包含的依赖关系及其版本的表可以在appendix中找到。

64.1.3 Default Import Statements译：64.1.3默认导入语句

为了减少Groovy代码的大小，几个`import`语句会自动包含在内。 请注意前面的例子中如何引用`@Component`， `@RestController`，并`@RequestMapping`无需使用完全合格的名称或`import`声明。

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查看添加导入之前的失败。

64.1.4 Automatic Main Method译：64.1.4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。 自动创建一个`SpringApplication`，编译代码为`source`。

64.1.5 Custom Dependency Management译：64.1.5自定义相关性管理

默认情况下，CLI在解析`@Grab`依赖`spring-boot-dependencies`时使用`spring-boot-dependencies`声明的依赖项管理。 可以使用`@DependencyManagementBom`注释来配置其他依赖项管理，这些管理覆盖默认的依赖关系管理。 注释的值应该指定一个或多个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`custom-bom-1.0.0.pom`下的Maven仓库中`com/example/custom-versions/1.0.0/`。

当您指定多个物料清单时，将按照您声明它们的顺序应用它们，如以下示例所示：

```
@DependencyManagementBom(["com.example.custom-bom:1.0.0",
  "com.example.another-bom:1.0.0"])
```

在前面的例子表明，在依赖关系管理`another-bom`覆盖依赖管理在`custom-bom`。

您可以在任何可以使用`@DependencyManagementBom`地方使用`@Grab`。 但是，为确保依赖性管理的顺序一致，最多可以在应用程序中使用`@DependencyManagementBom`。 依赖管理的一个有用的来源（它是Spring Boot的依赖管理的超集）是Spring IO Platform，您可能会在下面一行中包含它：

```
@DependencyManagementBom('io.spring.platform:platform-bom:1.1.2.RELEASE')
```

64.2 Applications with Multiple Source Files译：64.2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”。 这样做可让您使用单个目录中的多个文件，如以下示例所示：

```
$ spring run *.groovy
```

64.3 Packaging Your Application译：64.3打包您的应用程序

您可以使用`jar`命令将应用程序打包为独立的可执行jar文件，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关系生成的类，以便可以通过使用`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`添加和删除指向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。

64.4 Initialize a New Project译：64.4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'
```

前面的示例使用`spring-boot-starter-web`和`spring-boot-starter-data-jpa`创建了一个基于Maven的项目的`my-project`目录。 您可以使用`--list`标志列出服务的功能，如下例所示：

```
$ 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。 有关更多详细信息，请参阅`help`输出。 例如，以下命令将创建一个使用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'
```

64.5 Using the Embedded Shell 译：64.5使用嵌入式Shell

Spring Boot包含BASH和zsh shell的命令行完成脚本。 如果您没有使用这两个shell（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 (v2.0.3.RELEASE)
Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.
```

从嵌入式shell中，您可以直接运行其他命令：

```
$ version
Spring CLI v2.0.3.RELEASE
```

嵌入式外壳支持ANSI颜色输出以及`tab`完成。 如果您需要运行本机命令，则可以使用`!`前缀。 要退出嵌入式shell，请按`ctrl-c`。

64.6 Adding Extensions to the CLI 译：64.6将扩展添加到CLI

您可以使用`install`命令向CLI添加扩展。 该命令采用一组或多组工件坐标，格式为`group:artifact:version`，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之外，还安装了所有工件的依赖关系。

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和它们的依赖关系标识的工件。

要卸载所有附加依赖项，可以使用`--all`选项，如以下示例所示：

```
$ spring uninstall --all
```

65. Developing Applications with the Groovy Beans DSL 译：65.使用Groovy Beans DSL开发应用程序

Spring Framework 4.0对`beans{}` DSL”（从Grails借用）有本地支持，并且可以使用相同的格式将Groovy应用程序脚本中的bean定义嵌入到其中。 这有时是包含中间件声明等外部功能的好方法，如下例所示：

```
@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"
 }
}
```

只要它们保持在顶层，您可以将类声明与`beans{}`混合在同一个文件中，或者，如果您愿意，可以将beans DSL放在单独的文件中。

66. Configuring the CLI with `settings.xml` 译:66.使用 `settings.xml` 配置CLI

Spring Boot CLI使用Maven的依赖关系解析引擎Aether来解决依赖关系。 CLI使用`~/.m2/settings.xml`中的Maven配置来配置Aether。 CLI遵循以下配置设置：

- Offline
- Mirrors
- Servers
- Proxies
- 简介
  - Activation
  - Repositories
- Active profiles

有关更多信息，请参阅 Maven's settings documentation 。

67. What to Read Next 译:67.接下来要读什么

GitHub存储库中有一些可用于试用Spring Boot CLI的sample groovy scripts 。 整个source code也有广泛的Javadoc。

如果你发现你达到了CLI工具的极限，那么你可能需要考虑将你的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" Build tool plugins "，您可以在Gradle或Maven中使用它。

Part VIII. Build tool plugins 译:第八部分. 构建工具插件

Spring Boot为Maven和Gradle提供了构建工具插件。 这些插件提供了多种功能，包括可执行文件包的打包。 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。 如果您刚开始使用，您可能首先要从" Part III, "Using Spring Boot""部分阅读"Chapter 13, *Build Systems*" 。

68. Spring Boot Maven Plugin 译: 68. Spring Boot Maven插件

Spring Boot Maven Plugin在Maven中提供了Spring Boot支持，让您可以打包可执行jar或war档案并运行应用程序"就地"。 要使用它，你必须使用Maven 3.2（或更高版本）。

有关完整的插件文档，请参阅 Spring Boot Maven Plugin Site 。

68.1 Including the Plugin 译: 68.1包括插件

使用Spring启动Maven插件，包括在适当的XML `plugins`你的第 `pom.xml` ，如下面的例子：

```
<?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>2.0.3.RELEASE</version>
    <executions>
     <execution>
      <goals>
       <goal>repackage</goal>
      </goals>
     </execution>
    </executions>
   </plugin>
  </plugins>
 </build>
</project>
```

上面的配置重新包装了在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
```

如果没有包含 `<execution/>`配置（如前面的示例所示），则可以自行运行该插件（但只有在使用软件包目标时也如此），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，则还需要添加适当的 `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>
```

68.2 Packaging Executable Jar and War Files译：68.2打包可执行jar和 war文件

一旦`spring-boot-maven-plugin`已包含在您的`pom.xml`，它会自动尝试通过使用`spring-boot:repackage`目标来重写归档以使其可执行。 您应该将项目配置为通过使用通常的`packaging`元素来构建jar或war（如适用），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增强了您的现有存档。 您想要启动的主类可以通过使用配置选项或通过以常用方式向清单添加`Main-Class`属性来指定。 如果您未指定主类，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
```

要构建可执行并可部署到外部容器中的war文件，需要将嵌入容器依赖项标记为“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>
```

有关如何创建可部署战争文件的更多详细信息，请参见“ Section 88.1, “Create a Deployable War File””部分。

高级配置选项和示例可在 plugin info page中找到 。

69. Spring Boot Gradle Plugin译：69. Spring Boot Gradle插件

Spring Boot Gradle插件在Gradle中提供Spring Boot支持，让您可以打包可执行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 4.0或更高版本。 请参阅插件的文档以了解更多信息：

- Reference (HTML and PDF)
- API

70. Spring Boot AntLib Module译：70. Spring Boot AntLib模块

Spring Boot AntLib模块为Apache Ant提供了基本的Spring Boot支持。 您可以使用该模块创建可执行的jar。 要使用该模块，您需要声明一个额外`spring-boot`在命名空间`build.xml`，如下面的例子：

```
<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-2.0.3.RELEASE.jar>
```

“使用Spring Boot”部分包含更完整的 using Apache Ant with `spring-boot-antlib`示例。

70.1 Spring Boot Ant Tasks 译：70.1 Spring Boot Ant任务

一旦声明了 `spring-boot-antlib` 名称空间，则可以使用以下附加任务：

- Section 70.1.1, “`spring-boot:exejar`”
- Section 70.2, “`spring-boot:findmainclass`”

70.1.1 `spring-boot:exejar` 译：70.1.1 `spring-boot:exejar`

您可以使用 `exejar` 任务来创建Spring Boot可执行文件jar。 该任务支持以下属性：

Attribute	描述	Required
`destfile`	要创建的目标jar文件	是
`classes`	Java类文件的根目录	是
`start-class`	运行的主要应用程序类	否 *（默认是第一个发现声明 `main` 方法的类）*

以下嵌套元素可与任务一起使用：

Element	描述
`resources`	一个或多个 Resource Collections描述应该添加到创建的jar文件的内容中的一组 Resources 。
`lib`	应该将一个或多个 Resource Collections添加到构成应用程序的运行时依赖关系类路径的一组jar库中。

70.1.2 Examples 译：70.1.2例子

本节展示了两个Ant任务的例子。

指定开始课程。

```
<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>
```

检测开始课程。

```
<exejar destfile="target/my-application.jar" classes="target/classes">
 <lib>
  <fileset dir="lib" />
 </lib>
</exejar>
```

70.2 `spring-boot:findmainclass` 译：70.2 `spring-boot:findmainclass`

`findmainclass` 任务由 `exejar` 在内部使用，以定位声明 `main` 的类。 如有必要，您还可以直接在您的构建中使用此任务。 支持以下属性：

Attribute	描述	Required
`classesroot`	Java类文件的根目录	是 *（除非指定 `mainclass` ）*
`mainclass`	可用于短路 `main` 类搜索	没有
`property`	应该与结果一起设置的Ant属性	否 *（如果未指定，将会记录结果）*

70.2.1 Examples 译：70.2.1例子

本部分包含使用 `findmainclass` 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" />
```

71. Supporting Other Build Systems 译：71.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，您可能需要开发自己的插件。 可执行文件需要遵循特定的格式，并且某些条目需要以非压缩格式写入（有关详细信

息，请参阅附录中的“ executable jar format”部分）。

Spring Boot Maven和Gradle插件都使用`spring-boot-loader-tools`来实际生成罐子。 如果你需要，你可以直接使用这个库。

71.1 Repackaging Archives译：71.1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`类采用引用现有jar或war归档的单个构造函数参数。 使用两种可用`repackage()`方法之一来替换原始文件或写入新的目标。 在重新打包程序运行之前，还可以对各种设置进行配置。

71.2 Nested Libraries译：71.2嵌套库

重新打包存档时，可以使用`org.springframework.boot.loader.tools.Libraries`界面包含对相关文件的引用。 这里我们没有提供`Libraries`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` 。

71.3 Finding a Main Class译：71.3找到主要类

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。 如果找到多个候选人，则会引发异常。

71.4 Example Repackage Implementation译：71.4示例重 打包实施

以下示例显示了典型的重新打包实施：

```
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));
   }
  });
```

72. What to Read Next译：72下一步阅读什么

如果您对构建工具插件的工作方式感兴趣，可以查看GitHub上的`spring-boot-tools`模块。 有关可执行jar格式的更多技术细节，请参阅the appendix 。

如果您有特定的构建相关问题，可以查看“ how-to”指南。

Part IX. ‘How-to’ guides译：第九部分。 “如何操作”指南

本节提供了一些常见的答案 - “我该怎么做.....”使用Spring Boot时经常出现的问题。 其覆盖范围并不详尽，但确实涵盖了很多。

如果你有一个我们在这里没有涉及的具体问题，你可能想查看stackoverflow.com ，看看是否有人已经提供了答案。 这也是提出新问题的好地方（请使用`spring-boot`标签）。

我们也非常乐意扩大这一部分。 如果你想添加一个“如何操作”，请给我们一个pull request 。

73. Spring Boot Application译：73.弹簧启动应用程序

本节包含与Spring Boot应用程序直接相关的主题。

73.1 Create Your Own FailureAnalyzer译：73.1创建您自己的FailureAnalyzer

`FailureAnalyzer`是在启动时拦截异常并将其转换为可读的消息的好方法，它包装在`FailureAnalysis`中 。 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。 你也可以创建你自己的。

`AbstractFailureAnalyzer`是一个方便的扩展`FailureAnalyzer`，它检查在该异常处理一个指定的异常类型的存在。 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。 如果由于某种原因无法处理异常，请返回`null`以给另一个实现机会来处理异常。

`FailureAnalyzer`实现必须在`META-INF/spring.factories`注册。 以下示例注册`ProjectConstraintViolationFailureAnalyzer`：

```
org.springframework.boot.diagnostics.FailureAnalyzer=\
com.example.ProjectConstraintViolationFailureAnalyzer
```

如果您需要访问`BeanFactory`或`Environment`，则您的`FailureAnalyzer`可以分别实施`BeanFactoryAware`或`EnvironmentAware` 。

73.2 Troubleshoot Auto-configuration译：73.2自动配置故障排除

Spring Boot自动配置尽最大努力去做“正确的事情”，但有时候事情会失败，并且很难说明原因。

有一个真正有用的`ConditionEvaluationReport`提供任何Spring引导`ApplicationContext` 。 如果启用`DEBUG`日志记录输出，则可以看到它。 如果您使用`spring-boot-actuator` （请参阅the Actuator chapter ），还有一个`conditions`端点以JSON呈现报表。 使用该端点调试应用程序，并查看Spring Boot在运行时添加了哪些功能（以及哪些尚未添加）。

通过查看源代码和Javadoc可以回答更多的问题。 阅读代码时，请记住以下经验法则：

- Look for classes called `*AutoConfiguration` and read their sources. Pay special attention to the `@Conditional*` annotations to find out what features they enable and when. Add `--debug` to the command line or a System property `-Ddebug` to get a log on the console of all the auto-configuration decisions that were made in your app. In a running Actuator app, look at the `conditions` endpoint (`/actuator/conditions` or the JMX equivalent) for the same information.
- Look for classes that are `@ConfigurationProperties` (such as `ServerProperties`) and read from there the available external configuration options. The `@ConfigurationProperties` annotation has a `name` attribute that acts as a prefix to external properties. Thus, `ServerProperties` has `prefix="server"` and its configuration properties are `server.port`, `server.address`, and others. In a running Actuator app, look at the `configprops` endpoint.
- Look for uses of the `bind` method on the `Binder` to pull configuration values explicitly out of the `Environment` in a relaxed manner. It is often used with a prefix.
- Look for `@Value` annotations that bind directly to the `Environment`.
- Look for `@ConditionalOnExpression` annotations that switch features on and off in response to SpEL expressions, normally evaluated with placeholders resolved from the `Environment`.

73.3 Customize the Environment or ApplicationContext Before It Starts 译：73.3在开始之前自定义环境或ApplicationContext

`SpringApplication`有`ApplicationListeners`和`ApplicationContextInitializers`，用于将定制应用于上下文或环境。 Spring Boot从`META-INF/spring.factories`加载了许多这样的自定义内部使用。 有多种方法可以注册其他自定义设置：

- Programmatically, per application, by calling the `addListeners` and `addInitializers` methods on `SpringApplication` before you run it.
- Declaratively, per application, by setting the `context.initializer.classes` or `context.listener.classes` properties.
- Declaratively, for all applications, by adding a `META-INF/spring.factories` and packaging a jar file that the applications all use as a library.

`SpringApplication`向侦听`SpringApplication`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。 看到€œ Section 23.5, “Application Events and Listeners”在甲A€€~Spring引导featuresâ€™一个完整的列表部分。

也可以自定义`Environment`之前的应用程序上下文是通过使用刷新`EnvironmentPostProcessor`。 每个实现应在`META-INF/spring.factories`中注册，如以下示例所示：

```
org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPostProcessor
```

该实现可以加载任意文件并将其添加到`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);
  }
 }

}
```

`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。 上例将`custom-resource`属性源添加到列表的末尾，以便在任何常用其他位置中定义的键优先。 自定义实现可能会定义另一个订单。

Caution

在使用`@PropertySource`您`@SpringBootApplication`似乎是加载在一个自定义资源的便利和容易的方式`Environment`，我们不建议这样做，因为春天开机准备`Environment`前`ApplicationContext`被刷新。 使用`@PropertySource`定义的任何密钥加载太晚都不会对自动配置产生任何影响。

73.4 Build an ApplicationContext Hierarchy (Adding a Parent or Root Context) 译：73.4构建ApplicationContext层次结构（添加父级或根级上下文）

您可以使用`ApplicationBuilder`类创建父级/子级`ApplicationContext`层次结构。 看到€œ Section 23.4, “Fluent Builder API” â€在A€~Spring引导featuresâ€™节以获取更多信息。

73.5 Create a Non-web Application 译：73.5创建非Web应用程序

并非所有的Spring应用程序都必须是Web应用程序（或Web服务）。 如果您想在`main`方法中执行一些代码，但也可以引导Spring应用程序以设置要使用的基础结构，则可以使用Spring Boot的`SpringApplication`功能。 A `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服务器相关的依赖关系（例如servlet API）关闭到类路径中。 如果你不能这样做（例如，你从同一个代码库运行两个应用程序），那么你可以在你的`SpringApplication`实例上显式地调用`setWebApplicationType(WebApplicationType.NONE)`或者设置`applicationContextClass`属性（通过Java API或者外部属性）。 您想要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。

74. Properties and Configuration 译：74.属性和配置

本节包含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。

74.1 Automatically Expand Properties at Build Time译：74.1在构建时自动扩展属性

而不是硬编码在您的项目的构建配置中指定的一些属性，而不是使用现有的构建配置来自动扩展它们。 这在Maven和Gradle中都是可能的。

74.1.1 Automatic Property Expansion Using Maven译：74.1.1使用Maven自动扩展属性

您可以使用资源过滤自动扩展Maven项目中的属性。 如果您使用`spring-boot-starter-parent`，则可以使用`@..@`占位符引用Maven的“项目属性”，如下例所示：

```
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

只有生产配置以这种方式进行过滤（换句话说，不对`src/test/resources`应用过滤）。

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（用于热重载）。 这样做会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。 有关更多详细信息，请参阅plugin usage page 。

如果不使用起动机家长，你需要包括中引入下列元素`<build/>`你的元素`pom.xml`：

```
<resources>
 <resource>
  <directory>src/main/resources</directory>
  <filtering>true</filtering>
 </resource>
</resources>
```

您还需要在`<plugins/>`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>
```

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`，则可以通过构建来扩展这些属性。

74.1.2 Automatic Property Expansion Using Gradle译：74.1.2使用Gradle自动扩展属性

您可以通过配置Java插件的`processResources`任务来自动从Gradle项目扩展属性，如以下示例所示：

```
processResources {
 expand(project.properties)
}
```

然后，您可以使用占位符引用您的Gradle项目的属性，如以下示例所示：

```
app.name=${name}
app.description=${description}
```

Gradle的`expand`方法使用Groovy的`SimpleTemplateEngine`，它转换`${..}`令牌。 `${..}`风格与Spring自己的属性占位符机制冲突。 要将Spring属性占位符与自动展开一起使用，请按照以下方式转义Spring属性占位符：`\${..}`。

74.2 Externalize the Configuration of `SpringApplication`译：74.2外部配置`SpringApplication`

`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，您可以通过设置`spring.main.*`属性来使配置`spring.main.*`。 例如，在`application.properties`，您可能有以下设置：

```
spring.main.web-application-type=none
spring.main.banner-mode=off
```

然后，Spring Boot横幅在启动时不会打印，应用程序不会启动嵌入式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
```

实际应用中 *现在*显示横幅（如通过配置覆盖），并使用了三个源 `ApplicationContext` （按以下顺序）： `demo.MyApp` ， `com.acme.Config` ，和 `com.acme.ExtraConfig` 。

74.3 Change the Location of External Properties of an Application译：74.3更改应用程序的外部属性的位置

默认情况下，来自不同来源的属性添加到春节 `Environment` 以定义的顺序（见A€œ Chapter 24, *Externalized Configuration* â€在A€~Spring引导featuresâ€™的确切顺序部分）。

增加和修改此顺序的一个好方法是将 `@PropertySource` 注释添加到您的应用程序源中。 传递给 `SpringApplication` 静态便利方法的类以及使用 `setSources()` 添加的 `setSources()` 将检查是否有 `@PropertySources` 。 如果他们这样做，那些属性将尽早添加到 `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（如 `application.properties` ），系统属性，环境变量或命令行添加的属性的优先级。

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`): Defaults to `application` as the root of the file name.
- `spring.config.location` (`SPRING_CONFIG_LOCATION`): The file to load (such as a classpath resource or a URL). A separate `Environment` property source is set up for this document and it can be overridden by system properties, environment variables, or the command line.

无论您在环境中设置了什么，Spring Boot始终会按上述方式加载 `application.properties` 。 默认情况下，如果使用YAML，那么带有".yml"扩展名的文件也会添加到列表中。

Spring Boot会记录在 `DEBUG` 级别加载的配置文件以及在 `TRACE` 级别未找到的候选。

有关更多详细信息，请参阅 `ConfigFileApplicationListener` 。

74.4 Use 'Short' Command Line Arguments译：74.4使用'Short'Command Line Arguments

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。 您可以通过使用 `application.properties` 占位符启用此行为，如以下示例所示：

```
server.port=${port:8080}
```

如果从继承 `spring-boot-starter-parent` POM，则默认过滤器令牌 `maven-resources-plugins` 已经从改变 `${*}` 至 `@` （即 `@maven.token@` 代替 `${maven.token}` ），以防止弹簧式的占位符冲突。 如果您已直接为 `application.properties` 启用Maven筛选，则可能还需要将默认筛选标记更改为使用other delimiters 。

在这种特定情况下，端口绑定可在PaaS环境（如Heroku或Cloud Foundry）中运行。 在这两个平台中， `PORT` 环境变量会自动设置，并且Spring可以绑定到 `Environment` 属性的大写同义词。

74.5 Use YAML for External Properties译：74.5将 YAML用于外部属性

YAML是JSON的超集，因此，它是用于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，并将其放入类路径的根目录中。 然后加入 `snakeyaml` 到你的依赖（Maven的坐标 `org.yaml:snakeyaml` ，已列入如果使用 `spring-boot-starter` ）。 YAML文件被解析为Java `Map<String,Object>` （如JSON对象），并且Spring Boot将地图展平，使其具有一级深度，并且具有按句点分隔的键，因为许多人习惯使用Java中的 `Properties` 文件。

前面的示例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
```

看到€œ Section 24.6, "Using YAML Instead of Properties" â€在A€~Spring引导featuresâ€™部分，了解YAML更多信息。

74.6 Set the Active Spring Profiles译：74.6设置活动弹簧配置文件

Spring `Environment` 提供了一个API，但通常会设置一个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` 设置活动配置文件，如以下示例所示：

```
spring.profiles.active=production
```

以此方式设置的值将由系统属性或环境变量设置取代，但不会由 `SpringApplicationBuilder.profiles()` 方法 `SpringApplicationBuilder.profiles()` 。 因此，后一个Java API可以用来扩充配置文件而不更改默认值。

看到€œ Chapter 25, *Profiles* â€在A€œSpring引导featuresâ更多信息€部分。

74.7 Change Configuration Depending on the Environment译：74.7根据环境更改配置

YAML文件实际上是由`---`行分隔的一系列文档，并且每个文档都被分别解析为拼合的地图。

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）将被输入到Spring `Environment.acceptsProfiles()`方法中。 如果这些配置文件中的任何一个处于活动状态，则该文档将包含在最终合并中（否则不是），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，缺省端口是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。

YAML文档按其遇到的顺序合并。 后来的值覆盖较早的值。

要使用属性文件执行同样的操作，可以使用`application-${profile}.properties`来指定特定于配置文件的值。

74.8 Discover Built-in Options for External Properties译：74.8发现外部属性的内置选项

Spring Boot将`application.properties`（或`.yml`文件和其他位置）的外部属性绑定到运行时的应用程序中。 没有（也没有技术上不可能）在单个位置中提供所有受支持属性的详尽列表，因为贡献可能来自类路径上的其他jar文件。

具有执行器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。

附录包括一个`application.properties`示例，其中列出了Spring Boot支持的最常见属性。 最终列表来自搜索源代码`@ConfigurationProperties`和`@Value`注释以及偶尔使用`Binder`。 有关加载属性的确切顺序的更多信息，请参阅“ Chapter 24, *Externalized Configuration* ”。

75. Embedded Web Servers译：75.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。 此功能会导致一些操作问题，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。

75.1 Use Another Web Server译：75.1使用另一台Web服务器

许多Spring Boot启动器都包含默认的嵌入式容器。

- For servlet stack applications, the `spring-boot-starter-web` includes Tomcat by including `spring-boot-starter-tomcat`, but you can use `spring-boot-starter-jetty` or `spring-boot-starter-undertow` instead.
- For reactive stack applications, the `spring-boot-starter-webflux` includes Reactor Netty by including `spring-boot-starter-reactor-netty`, but you can use `spring-boot-starter-tomcat`, `spring-boot-starter-jetty`, or `spring-boot-starter-undertow` instead.

当切换到不同的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器，以帮助尽可能简化此过程。

以下Maven示例显示了如何排除Tomcat并将Jetty包含在Spring MVC中：

```
<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>
```

以下Gradle示例显示了如何排除Netty并将Undertow包含在Spring WebFlux中：

```
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'
 // ...
}
```

需要`spring-boot-starter-reactor-netty`才能使用`WebClient`类，因此即使需要包含不同的HTTP服务器，您也可能需要保留Netty的依赖关系。

75.2 Disabling the Web Server译：75.2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`WebApplicationType`中配置`application.properties`，如下例所示：

```
spring.main.web-application-type=none
```

75.3 Change the HTTP Port译：75.3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`（例如`application.properties`或作为System属性）进行设置。 由于`Environment`值的放宽绑定，您还可以使用`SERVER_PORT`（例如，作为OS环境变量）。

要完全关闭HTTP端点但仍创建`WebApplicationContext`，请使用`server.port=-1`。（这样做有时对测试有用。）

有关详细信息，请参阅€œ Section 27.4.4, “Customizing Embedded Servlet Containers” â€在A€~Spring引导featuresâ€™部分，或`ServerProperties`源代码。

75.4 Use a Random Unassigned HTTP Port译：75.4使用随机未分配的HTTP端口

要扫描空闲端口（使用操作系统本机防止冲突），请使用`server.port=0`。

75.5 Discover the HTTP Port at Runtime译：75.5在运行时发现HTTP端口

您可以从日志输出或从`ServletWebServerApplicationContext`通过其`WebServer`访问运行服务器的端口。 最好的方法是确保它已被初始化，并添加`@Bean`类型`ApplicationListener<ServletWebServerInitializedEvent>`并在容器发布时将容器拉出。

使用`@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`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;

 // ...

}
```

`@LocalServerPort`是元注解为`@Value("${local.server.port}")`。 不要试图在普通应用程序中注入端口。 正如我们刚刚看到的，只有在容器初始化后才设置该值。 与测试相反，应用程序代码回调会提前处理（在值实际可用之前）。

75.6 Enable HTTP Response Compression译：75.6启用HTTP响应压缩

Jetty，Tomcat和Undertow支持HTTP响应压缩。 它可以在`application.properties`启用，如下所示：

```
server.compression.enabled=true
```

默认情况下，为了执行压缩，响应长度必须至少为2048个字节。 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，响应仅在其内容类型为以下内容之一时才被压缩：

- `text/html`
- `text/xml`
- `text/plain`
- `text/css`

您可以通过设置`server.compression.mime-types`属性来配置此行为。

75.7 Configure SSL译：75.7配置SSL

可以通过设置各种`server.ssl.*`属性（通常在`application.properties`或`application.yml`以声明方式配置SSL。 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
```

有关所有受支持的属性的详细信息，请参阅`Ssl`。

使用上述示例中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`application.properties` Boot不支持通过`application.properties`配置HTTP连接器和HTTPS连接器。 如果你想拥有两者，你需要以编程方式配置其中之一。 我们建议使用`application.properties`来配置HTTPS，因为HTTP连接器更容易以编程方式进行配置。 示例请参阅`spring-boot-sample-tomcat-multi-connectors`示例项目。

75.8 Configure HTTP/2译：75.8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。 这种支持取决于选择的Web服务器和应用程序环境，因为该协议不受JDK8开箱即用的支持。

Spring Boot不支持`h2c`，这是HTTP / 2协议的明文版本。 所以你必须configure SSL first 。

75.8.1 HTTP/2 with Undertow译：75.8.1 HTTP / 2与Undertow

从Undertow 1.4.0+开始，支持HTTP / 2，而不需要JDK8。

75.8.2 HTTP/2 with Jetty译：75.8.2使用Jetty的HTTP / 2

从Jetty 9.4.8开始，HTTP / 2也支持Conscrypt library 。 要启用该支持，您的应用程序需要有两个额外的依赖关系：`org.eclipse.jetty:jetty-alpn-conscrypt-server`和`org.eclipse.jetty.http2:http2-server` 。

75.8.3 HTTP/2 with Tomcat译：75.8.3使用Tomcat的HTTP / 2

Spring Boot默认使用Tomcat 8.5.x。 使用该版本时，只有在主机操作系统上安装了`libtcnative`库及其依赖关系时才支持HTTP / 2。

库文件夹必须可用（如果尚未提供）到JVM库路径。 您可以使用JVM参数（如`-Djava.library.path=/usr/local/opt/tomcat-native/lib`来执行此`-Djava.library.path=/usr/local/opt/tomcat-native/lib` 。 更多关于这个在official Tomcat documentation 。

启动没有本地支持的Tomcat 8.5.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
```

这个错误不是致命的，应用程序仍然以HTTP / 1.1 SSL支持开始。

使用Tomcat 9.0.x和JDK9运行应用程序不需要安装任何本机库。 要使用Tomcat 9，您可以用您选择的版本覆盖`tomcat.version`构建属性。

75.9 Configure the Web Server译：75.9配置Web服务器

通常，您应该首先考虑使用许多可用的配置密钥之一，并通过在您的`application.properties` （或`application.yml`或环境等中添加新条目（参见“ Section 74.8, “Discover Built-in Options for External Properties”” ））来自定义您的Web服务器。 该`server.*`命名空间是非常有用的在这里，它包括像命名空间`server.tomcat.*` ，`server.jetty.*`等人，针对特定服务器的功能。 请参阅Appendix A, *Common application properties*的列表。

前面几节介绍了很多常见的用例，如压缩，SSL或HTTP / 2。 但是，如果您的用例中没有配置密钥，则应该查看`WebServerFactoryCustomizer` 。 您可以声明这样的组件并访问与您的选择相关的服务器工厂：应该为所选服务器（Tomcat，Jetty，Reactor Netty，Undertow）和所选Web栈（Servlet或Reactive）选择变体。

下面的示例适用于`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
 }
}
```

另外Spring Boot提供：

Server	Servlet stack	Reactive stack
Tomcat的	`TomcatServletWebServerFactory`	`TomcatReactiveWebServerFactory`
码头	`JettyServletWebServerFactory`	`JettyReactiveWebServerFactory`
暗潮	`UndertowServletWebServerFactory`	`UndertowReactiveWebServerFactory`
反应堆	N / A	`NettyReactiveWebServerFactory`

一旦你`WebServerFactory`访问`WebServerFactory` ，你通常可以添加定制器来配置特定的部分，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，你也可以声明自己的`WebServerFactory`组件，它将覆盖Spring Boot提供的组件。 在这种情况下，您不能依赖`server`命名空间中的配置属性。

75.10 Add a Servlet, Filter, or Listener to a Application译：75.10将Servlet，Filter或Listener添加到应用程序

在一个servlet栈的应用，即与`spring-boot-starter-web` ，有两种方法可以添加`Servlet` ，`Filter` ，`ServletContextListener` ，以及其他聆听由Servlet API到您的应用程序支持：

- Section 75.10.1, “Add a Servlet, Filter, or Listener by Using a Spring Bean”
- Section 75.10.2, “Add Servlets, Filters, and Listeners by Using Classpath Scanning”

75.10.1 Add a Servlet, Filter, or Listener by Using a Spring Bean译：75.10.1使用Spring Bean添加Servlet，Filter或Listener

要添加`Servlet` ，`Filter` ，或者Servlet `*Listener`通过使用一个Spring bean，你必须提供一个`@Bean`它的定义。 当你想要注入配置或依赖时，这样做可能非常有用。 但是，您必须非常小心，它们不会导致太多其他bean的急切初始化，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。）您可以通过在第一次使用而非初始化时懒惰地初始化bean来解决这些限制。

在 `Filters` 和 `Servlets` 的情况下，您还可以添加映射和初始化参数，方法是添加 `FilterRegistrationBean` 或 `ServletRegistrationBean` 而不是或添加到基础组件。

如果没有 `dispatcherType` 在过滤器上登记被指定时， `REQUEST` 被使用。 这与Servlet规范的默认调度程序类型一致。

像任何其他Spring bean一样，您可以定义Servlet过滤器bean的顺序; 请确保检查“ the section called “Registering Servlets, Filters, and Listeners as Spring Beans””部分。

Disable Registration of a Servlet or Filter 译:禁用Servlet或过滤器的注册

作为described earlier ，任何 `Servlet` 或 `Filter` bean都会自动注册到servlet容器。 要禁用特定的 `Filter` 或 `Servlet` bean的注册，请为其创建一个注册Bean并将其标记为禁用，如以下示例所示：

```
@Bean
public FilterRegistrationBean registration(MyFilter filter) {
 FilterRegistrationBean registration = new FilterRegistrationBean(filter);
 registration.setEnabled(false);
 return registration;
}
```

75.10.2 Add Servlets, Filters, and Listeners by Using Classpath Scanning 译: 75.10.2使用类路径扫描添加Servlet，过滤器和监听器

`@WebServlet` ， `@WebFilter` 和 `@WebListener` 注解的类可以具有嵌入的servlet容器通过注释一个自动注册 `@Configuration` 类 `@ServletComponentScan` 和指定包含要注册的部件的封装（多个）。 默认情况下，从注释类的包中扫描 `@ServletComponentScan` 。

75.11 Configure Access Logging 译: 75.11配置访问日志记录

访问日志可以通过各自的命名空间为Tomcat，Undertow和Jetty进行配置。

例如，以下设置将使用 custom pattern在Tomcat上登录访问。

```
server.tomcat.basedir=my-tomcat
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.pattern=%t %a "%r" %s (%D ms)
```

日志的默认位置是相对于Tomcat基本目录的 `logs` 目录。 默认情况下， `logs` 目录是一个临时目录，因此您可能需要修复Tomcat的基本目录或使用日志的绝对路径。 在前面的例子中，相对于应用程序的工作目录，日志在 `my-tomcat/logs` 中可用。

Undertow的访问日志记录可以以类似的方式进行配置，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，日志被重定向到 `System.err` 。 有关更多详细信息，请参阅the Jetty documentation 。

75.12 Running Behind a Front-end Proxy Server 译: 75.12运行在前端代理服务器后面

您的应用程序可能需要发送 `302` 重定向或使用绝对链接呈现内容。 当在代理后面运行时，调用者需要链接到代理而不是托管应用的机器的物理地址。 通常情况下，这种情况是通过与代理签订合同来处理的，代理添加了头文件来告诉后端如何构建到自身的链接。

如果代理增加了传统 `X-Forwarded-For` 个 `X-Forwarded-Proto` 头（大多数代理服务器这样做），绝对链接应正确地呈现，提供 `server.use-forward-headers` 设置为 `true` 在 `application.properties` 。

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，它默认为 `false` 。

75.12.1 Customize Tomcat’s Proxy Configuration 译: 75.12.1自定义Tomcat的代理配置

如果您使用Tomcat，则可以另外配置用于携带“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，该正则表达式与要受信任的内部代理相匹配。 默认情况下，IP地址在 `10/8` ， `192.168/16` ， `169.254/16` 和 `127/8` 是值得信赖的。 您可以通过将条目添加到 `application.properties` 来自定义阀的配置，如以下示例所示：

```
server.tomcat.internal-proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}
```

只有在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同的值为 `192\.168\.\d{1,3}\.\d{1,3}` 。

您可以通过将 `internal-proxies` 设置为空来信任所有代理（但不要在生产中这样做）。

你可以采取的Tomcatâ€™的配置完全控制 `RemoteIpValve` 通过切换自动一关（这样做，设置 `server.use-forward-headers=false` ），并在添加一个新瓣膜实例 `TomcatServletWebServerFactory` 豆。

75.13 Enable Multiple Connectors with Tomcat译：75.13使用Tomcat启用多个连接器

您可以将 `org.apache.catalina.connector.Connector` 添加到 `TomcatServletWebServerFactory` ，它可以允许多个连接器，包括HTTP和HTTPS连接器，如以下示例中所示：

```
@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);
 }
}
```

75.14 Use Tomcat’s LegacyCookieProcessor译：75.14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“版本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，你应该考虑更新你的代码，以便只存储符合以后Cookie规范的值。 但是，如果您无法更改写入cookie的方式，则可以将Tomcat配置为使用 `LegacyCookieProcessor` 。 要切换到 `LegacyCookieProcessor` ，请使用添加了 `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()));
}
```

75.15 Enable Multiple Listeners with Undertow译：75.15使用Undertow启用多个监听器

将一个 `UndertowBuilderCustomizer` 添加到 `UndertowServletWebServerFactory` ，并将一个侦听器添加到 `Builder` ，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;
}
```

75.16 Create WebSocket Endpoints Using @ServerEndpoint译：75.16使用@ServerEndpoint创建WebSocket端点

如果你想使用 `@ServerEndpoint` ，所使用的嵌入式容器春季启动应用程序，您必须声明一个 `ServerEndpointExporter` `@Bean` ，如下面的例子：

```
@Bean
public ServerEndpointExporter serverEndpointExporter() {
 return new ServerEndpointExporter();
}
```

上例中显示的bean使用底层WebSocket容器注册任何 `@ServerEndpoint` 注释的bean。 当部署到独立的servlet容器时，该角色由servlet容器初始化程序执行，并且 `ServerEndpointExporter` bean。

76. Spring MVC译：76. Spring MVC

Spring Boot有许多包含Spring MVC的初学者。 请注意，一些初学者包括对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。

76.1 Write a JSON REST Service 译：76.1编写 JSON REST服务

在Spring Boot应用程序中，任何Spring `@RestController`应默认呈现JSON响应，只要Jackson2位于类路径中，如以下示例所示：

```
@RestController
public class MyController {

 @RequestMapping("/thing")
 public MyThing thing() {
   return new MyThing();
 }

}
```

只要`MyThing`可以由Jackson2序列化（正常POJO或Groovy对象为true），那么默认情况下`localhost:8080/thing`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喜欢XML的接受标头。

76.2 Write an XML REST Service 译：76.2编写 XML REST服务

如果您在类路径上具有Jackson XML扩展（`jackson-dataformat-xml`），则可以使用它来呈现XML响应。 前面我们用于JSON的例子可以工作。 要使用Jackson XML渲染器，请将以下依赖项添加到您的项目中：

```
<dependency>
 <groupId>com.fasterxml.jackson.dataformat</groupId>
 <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

您可能还想添加对Woodstox的依赖关系。 它比JDK提供的默认StAX实现更快，并且还增加了漂亮的打印支持和改进的命名空间处理。 以下清单显示了如何在Woodstox上包含依赖项：

```
<dependency>
 <groupId>org.codehaus.woodstox</groupId>
 <artifactId>woodstox-core-asl</artifactId>
</dependency>
```

如果杰克逊的XML扩展不可用，则使用JAXB（在JDK中默认提供），并附加要求`MyThing`注释为`@XmlRootElement`，如以下示例所示：

```
@XmlRootElement
public class MyThing {
 private String name;
 // .. getters and setters
}
```

要让服务器呈现XML而不是JSON，您可能必须发送`Accept: text/xml`标头（或使用浏览器）。

76.3 Customize the Jackson ObjectMapper 译：76.3自定义 Jackson ObjectMapper

Spring MVC（客户端和服务器端）使用`HttpMessageConverters`来协商HTTP交换中的内容转换。 如果Jackson在类路径中，则已经获得由`Jackson2ObjectMapperBuilder`提供的默认转换器，该转换器的一个实例将为您自动配置。

`ObjectMapper`（或Jackson XML转换器的`XmlMapper`）实例（默认情况下创建）具有以下自定义属性：

- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled
- `SerializationFeature.WRITE_DATES_AS_TIMESTAMPS` is disabled

Spring Boot还具有一些功能，可以更轻松地自定义此行为。

您可以使用该环境配置`ObjectMapper`和`XmlMapper`实例。 Jackson提供了一整套简单的开/关功能，可用于配置其处理的各个方面。 这些功能在映射到环境中的属性的六个枚举（在Jackson中）中进行了描述：

Jackson enum	Environment property
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jackson.deserialization.<feature_name>=true|false`
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jackson.generator.<feature_name>=true|false`
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jackson.mapper.<feature_name>=true|false`
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jackson.parser.<feature_name>=true|false`
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jackson.serialization.<feature_name>=true|false`
`com.fasterxml.jackson.annotation.JsonInclude.Include`	`spring.jackson.default-property-inclusion=always|non_null|non_absent|non_default|non_`

例如，要启用漂亮打印，请设置`spring.jackson.serialization.indent_output=true`。 需要注意的是，由于使用的relaxed binding的情况下`indent_output`不具有相应的枚举常量，这是大小写相匹配`INDENT_OUTPUT`。

此基于环境的配置应用于自动配置的`Jackson2ObjectMapperBuilder` bean，并适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`可以由一个或多个`Jackson2ObjectMapperBuilderCustomizer`豆进行定制。 这些定制程序bean可以定制（Boot自己的定制程序的顺序为0），可以在Boot定制之前和之后应用额外的定制。

`com.fasterxml.jackson.databind.Module`类型的任何bean `com.fasterxml.jackson.databind.Module`自动注册到自动配置

的`Jackson2ObjectMapperBuilder`，并应用于它创建的任何`ObjectMapper`实例。 这为您在向应用程序添加新功能时提供了一种全球机制，可以为自定义模块提供支持

如果要完全替换默认的`ObjectMapper`，则可以定义该类型的`@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，无论如何，这样做都会禁用`ObjectMapper`所有自动配置。

如果您提供任何`@Beans`类型`MappingJackson2HttpMessageConverter`，它们将替换MVC配置中的默认值。 此外，还提供了一个类型为`HttpMessageConverters`的便捷bean（如果您使用默认的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。

有关更多详细信息，请参阅“ Section 76.4, “Customize the @ResponseBody Rendering””部分和 `WebMvcAutoConfiguration`源代码。

76.4 Customize the @ResponseBody Rendering 译：76.4自定义@ResponseBody渲染

Spring使用`HttpMessageConverters`来呈现`@ResponseBody` （或来自`@RestController`响应）。 您可以通过在Spring Boot环境中添加适当类型的Bean来提供额外的转换器。 如果您添加的bean是默认包含的类型（例如用于JSON转换的`MappingJackson2HttpMessageConverter` ），它将替换默认值。 提供了一个类型为`HttpMessageConverters`便捷bean，如果您使用默认的MVC配置，该便利bean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，如果您想手动将它们注入到自定义`RestTemplate`，它可能很有用）。

与正常的MVC用法一样，您提供的任何`WebMvcConfigurer` bean也可以通过覆盖`configureMessageConverters`方法来贡献转换器。 然而，与普通MVC不同的是，您只能提供您需要的其他转换器（因为Spring Boot使用相同的机制来提供其默认值）。 最后，如果你提供你自己的选择了春季启动默认的MVC配置`@EnableWebMvc`配置，你完全可以采取控制和使用手工做的一切`getMessageConverters`从`WebMvcConfigurationSupport` 。

有关更多详细信息，请参阅 `WebMvcAutoConfiguration`源代码。

76.5 Handling Multipart File Uploads 译：76.5处理多部分文件上传

Spring Boot支持Servlet 3 `javax.servlet.http.Part` API来支持上传文件。 默认情况下，Spring Boot在每个文件中配置Spring MVC的最大大小为1MB，单个请求中最大为10MB的文件数据。 您可以通过使用`MultipartProperties`类中公开的属性来覆盖这些值，中间数据存储的位置（例如， `/tmp`目录）以及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` 。

当您想要在Spring MVC控制器处理程序方法中以 `@RequestParam`类型 `MultipartFile`的 `@RequestParam`接收多部分编码文件数据时，多部分支持很有用。

有关更多详细信息，请参阅 `MultipartAutoConfiguration`源代码。

76.6 Switch Off the Spring MVC DispatcherServlet 译：76.6关闭Spring MVC DispatcherServlet

Spring Boot希望向下提供应用程序根目录下的所有内容（ `/`）。 如果您宁愿将您自己的servlet映射到该URL，则可以执行此操作。 但是，您可能会失去其他一些Boot MVC功能。 要添加自己的servlet并将其映射到根资源，请声明类型为`@Bean`的`Servlet`并为其指定特殊的bean名称`dispatcherServlet` 。 （如果你想关闭它而不是替换它，你也可以用这个名称创建一个不同类型的bean。）

76.7 Switch off the Default MVC Configuration 译：76.7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。

76.8 Customize ViewResolvers 译：76.8自定义ViewResolvers

`ViewResolver`是Spring MVC的核心组件，将`@Controller`视图名称翻译为实际的`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`未用于呈现`@ResponseBody` ）。 `ViewResolver`有很多实现可供选择，而Spring本身并不认为你应该使用哪一个。 另一方面，Spring Boot会为您安装一两个，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获得结果，因此，如果添加自己的解析器，则必须知道订单以及解析器添加到的位置。

`WebMvcAutoConfiguration`增加了以下 `ViewResolvers`对上下文：

- An `InternalResourceViewResolver` named ‘defaultViewResolver’. This one locates physical resources that can be rendered by using the `DefaultServlet` (including static resources and JSP pages, if you use those). It applies a prefix and a suffix to the view name and then looks for a physical resource with that path in the servlet context (the defaults are both empty but are accessible for external configuration through `spring.mvc.view.prefix` and `spring.mvc.view.suffix`). You can override it by providing a bean of the same type.
- A `BeanNameViewResolver` named ‘beanNameViewResolver’. This is a useful member of the view resolver chain and picks up any beans with the same name as the `View` being resolved. It should not be necessary to override or replace it.
- A `ContentNegotiatingViewResolver` named ‘viewResolver’ is added only if there **are** actually beans of type `View` present. This is a ‘master’ resolver, delegating to all the others and attempting to find a match to the ‘Accept’ HTTP header sent by the client. There is a useful blog about `ContentNegotiatingViewResolver` that you might like to study to learn more, and you might also look at the source code for detail. You can switch off the auto-configured `ContentNegotiatingViewResolver` by defining a bean named ‘viewResolver’.
- If you use Thymeleaf, you also have a `ThymeleafViewResolver` named ‘thymeleafViewResolver’. It looks for resources by surrounding the view name with a prefix and suffix. The prefix is `spring.thymeleaf.prefix`, and the suffix is `spring.thymeleaf.suffix`. The values of the prefix and suffix default to ‘classpath:/templates/’ and ‘.html’, respectively. You can override `ThymeleafViewResolver` by providing a bean of the same name.
- If you use FreeMarker, you also have a `FreeMarkerViewResolver` named ‘freeMarkerViewResolver’. It looks for resources in a loader path (which is externalized to `spring.freemarker.templateLoaderPath` and has a default value of ‘classpath:/templates/’) by surrounding the view name with a prefix and a suffix. The prefix is externalized to `spring.freemarker.prefix`, and the suffix is externalized to `spring.freemarker.suffix`. The default values of the prefix and suffix are empty and ‘.ftl’, respectively. You can override `FreeMarkerViewResolver` by providing a bean of the same name.
- If you use Groovy templates (actually, if `groovy-templates` is on your classpath), you also have a `GroovyMarkupViewResolver` named ‘groovyMarkupViewResolver’. It looks for resources in a loader path by surrounding the view name with a prefix and suffix (externalized to `spring.groovy.template.prefix` and `spring.groovy.template.suffix`). The prefix and suffix have default values of ‘classpath:/templates/’ and ‘.tpl’, respectively. You can override `GroovyMarkupViewResolver` by providing a bean of the same name.

有关更多详细信息，请参阅以下部分：

- `WebMvcAutoConfiguration`
- `ThymeleafAutoConfiguration`
- `FreeMarkerAutoConfiguration`
- `GroovyTemplateAutoConfiguration`

77. Jersey译：77.泽西岛

77.1 Secure Jersey endpoints with Spring Security译：77.1使用Spring Security安全泽西端点

Spring Security可以用来保护一个基于Jersey的Web应用程序，就像它可以用来保护基于Spring MVC的Web应用程序一样。 但是，如果您希望在Jersey中使用Spring Security的方法级安全性，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 Security有机会向客户端报告身份验证或授权失败之前提交响应。

该`jersey.config.server.response.setStatusOverSendError`属性必须设置为`true`上对这个应用程序€™的`ResourceConfig`豆，如下面的例子：

```
@Component
public class JerseyConfig extends ResourceConfig {

 public JerseyConfig() {
  register(Endpoint.class);
  setProperties(Collections.singletonMap(
    "jersey.config.server.response.setStatusOverSendError", true));
 }

}
```

78. HTTP Clients译：78.HTTP客户端

Spring Boot提供了许多与HTTP客户端一起工作的入门者。 本节回答与使用它们有关的问题。

78.1 Configure RestTemplate to Use a Proxy译：78.1配置RestTemplate以使用代理

如Section 33.1, “RestTemplate Customization”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。

代理配置的确切细节取决于正在使用的底层客户端请求工厂。 以下示例配置`HttpComponentsClientRequestFactory`与`HttpClient`使用所有主机代理除了`192.168.0.5`：

```
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));
 }

}
```

79. Logging译：79.伐木

Spring Boot没有强制日志依赖性，Commons Logging API除外，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起始者，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有一个`LoggingSystem`抽象，它尝试根据类路径的内容配置日志记录。 如果Logback可用，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日志记录器的级别，则可以使用“logging.level”前缀在`application.properties`执行此操作，如以下示例中所示：

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

您还可以通过使用“logging.file”来设置要写入日志的文件的位置（除了控制台之外）。

要配置日志记录系统的更精细的设置，您需要使用`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`classpath:logback.xml`的classpath:logback.xml）中选取本地配置，但可以使用“logging.config”属性设置配置文件的位置。

79.1 Configure Logback for Logging译：79.1配置Logback进行日志记录

如果将`logback.xml`放在类路径的根目录中，则会从此处拾取（或从`logback-spring.xml`，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，如果您想设置级别，可以包含该配置，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>
```

如果您在spring-boot jar中查看 `base.xml` ，可以看到它使用了 `LoggingSystem` 为您创建的一些有用的系统属性：

- `${PID}`: The current process ID.
- `${LOG_FILE}`: Whether `logging.file` was set in Boot's external configuration.
- `${LOG_PATH}`: Whether `logging.path` (representing a directory for log files to live in) was set in Boot's external configuration.
- `${LOG_EXCEPTION_CONVERSION_WORD}`: Whether `logging.exception-conversion-word` was set in Boot's external configuration.

Spring Boot还通过使用自定义Logback转换器在控制台（但不是日志文件）上提供了一些漂亮的ANSI彩色终端输出。 有关详细信息，请参阅默认的 `base.xml` 配置。

如果Groovy位于类路径中，那么您应该也可以使用 `logback.groovy` 配置Logback。 如果存在，则优先选择此设置。

79.1.1 Configure Logback for File-only Output 译：79.1.1为仅文件输出配置Logback

如果要禁用控制台记录和写仅输出到一个文件，你需要自定义 `logback-spring.xml` 是进口 `file-appender.xml` 但不 `console-appender.xml` ，如下面的例子：

```
<?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
```

79.2 Configure Log4j for Logging 译：79.2配置Log4j进行日志记录

Spring Boot支持Log4j 2用于记录配置，如果它在类路径上。 如果您使用starters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，则除了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，即使它需要一些排除的问题。 以下示例显示如何在Maven中设置初学者：

```
<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>
```

Log4j初学者将通用日志记录要求的依赖关系集中在一起（例如让Tomcat使用 `java.util.logging` 但使用Log4j 2配置输出）。 有关更多详细信息，请参阅Actuator Log4j 2示例，并查看它的实际使用情况。

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 `java.util.logging.manager` 来配置其 `org.apache.logging.log4j.jul.LogManager` 。

79.2.1 Use YAML or JSON to Configure Log4j 2 译：79.2.1使用YAML或JSON配置Log4j 2

除了默认的XML配置格式之外，Log4j 2还支持YAML和JSON配置文件。 要配置Log4j 2以使用备用配置文件格式，请将相应的依赖关系添加到类路径中，并将您的配置文件命名为与您选择的文件格式相匹配，如下例所示：

Format	Dependencies	File names
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`

80. Data Access 译：80.数据访问

Spring Boot包含许多用于处理数据源的入门者。 本节回答与此相关的问题。

80.1 Configure a Custom DataSource译：80.1配置自定义数据源

要配置您自己的`DataSource`，请在您的配置中定义该类型的`@Bean`。 Spring Boot会在`DataSource`地方重新使用您的`DataSource`，包括数据库初始化。 如果您需要外部化某些设置，则可以将`DataSource`绑定到环境（请参阅“ Section 24.7.1, “Third-party Configuration”” ）。

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`对于URL，用户名和池大小具有常规的JavaBean属性，则在`DataSource`可供其他组件使用之前，会自动绑定这些设置。 常规database initialization也发生（这样的相关子集`spring.datasource.*`仍然可以与您的自定义配置中使用）。

如果您配置自定义JNDI`DataSource`，则可以应用相同的原则，如下例所示：

```
@Bean(destroyMethod="")
@ConfigurationProperties(prefix="app.datasource")
public DataSource dataSource() throws Exception {
 JndiDataSourceLookup dataSourceLookup = new JndiDataSourceLookup();
 return dataSourceLookup.getDataSource("java:comp/env/jdbc/YourDS");
}
```

Spring Boot还提供了一个实用程序构建器类，名为`DataSourceBuilder`，可用于创建其中一个标准数据源（如果它位于类路径上）。 构建器可以根据类路径中可用的内容来检测要使用的那个。 它还根据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();
}
```

要运行`DataSource`的应用程序，您只需要连接信息。 也可以提供池特定的设置。 查看将在运行时使用的实现以获取更多详细信息。

以下示例显示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
```

但是，有一个问题。 由于连接池的实际类型未公开，因此您的自定义`DataSource`的元数据中不会生成任何密钥，并且您的IDE中没有完成（因为`DataSource`接口未公开任何属性）。 另外，如果你碰巧在类路径中有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，您必须按照以下方式重写您的配置：

```
app.datasource.jdbc-url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.maximum-pool-size=30
```

您可以通过强制使用连接池并返回专用实现而不是`DataSource`。 您不能在运行时更改实现，但选项列表将是明确的。

下面的例子演示如何创建一个`HikariDataSource`与`DataSourceBuilder`：

```
@Bean
@ConfigurationProperties("app.datasource")
public HikariDataSource dataSource() {
 return DataSourceBuilder.create().type(HikariDataSource.class).build();
}
```

您甚至可以利用`DataSourceProperties`为您做的`DataSourceProperties`工作，也就是说，如果没有提供URL，通过提供默认的嵌入式数据库，提供明智的用户名和密码。 您可以轻松地从任何`DataSourceProperties`对象的状态初始化`DataSourceBuilder`，这样您还可以注入Spring Boot自动创建的DataSource。 然而，这将拆分配置成两个命名空间：`url`，`username`，`password`，`type`，并`driver`上`spring.datasource`并在您的自定义命名空间（休息`app.datasource`）。 为避免这种情况，您可以在自定义名称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")
public HikariDataSource dataSource(DataSourceProperties properties) {
 return properties.initializeDataSourceBuilder().type(HikariDataSource.class)
   .build();
}
```

除了选择了一个专用连接池（在代码中）并且它的设置暴露在相同的命名空间中之外，这个设置使您*可以*与默认情况下的Spring Boot为您做*同步*。 由于`DataSourceProperties`走的是照顾`url`/`jdbcUrl`翻译，你可以按如下步骤进行设置：

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.maximum-pool-size=30
```

因为您的自定义配置选择与Hikari一起使用， `app.datasource.type`不起作用。 在实践中，构建器会初始化为您可能设置的任何值，然后通过调用`.type()`来`.type()`。

看到€œ Section 29.1, “Configure a DataSource” â€在A€œSpring引导featuresâ€段和 `DataSourceAutoConfiguration`类的更多细节。

80.2 Configure Two DataSources译：80.2配置两个数据源

如果您需要配置多个数据源，则可以应用上一节中介绍的相同技巧。 但是，您必须将其中一个`DataSource`实例标记为`@Primary`，因为各种自动配置都期望能够逐个获取。

如果您创建自己的`DataSource`， `DataSource`自动配置。 在以下示例中，我们提供了与自动配置在主数据源上提供的*完全相同的*功能集：

```
@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
 return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSource firstDataSource() {
 return firstDataSourceProperties().initializeDataSourceBuilder().build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public BasicDataSource secondDataSource() {
 return DataSourceBuilder.create().type(BasicDataSource.class).build();
}
```

必须将`firstDataSourceProperties`标记为`@Primary`以便数据库初始值设定项功能使用您的副本（如果使用初始值设定项）。

这两个数据源也都是用于高级自定义的。 例如，你可以如下配置它们：

```
app.datasource.first.type=com.zaxxer.hikari.HikariDataSource
app.datasource.first.maximum-pool-size=30

app.datasource.second.url=jdbc:mysql://localhost/test
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")
public DataSource firstDataSource() {
 return firstDataSourceProperties().initializeDataSourceBuilder().build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public DataSourceProperties secondDataSourceProperties() {
 return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public DataSource secondDataSource() {
 return secondDataSourceProperties().initializeDataSourceBuilder().build();
}
```

前面的示例在自定义命名空间上配置两个数据源，其使用与Spring Boot在自动配置中使用的逻辑相同的逻辑。

80.3 Use Spring Data Repositories译：80.3使用Spring数据存储库

Spring Data可以创建各种风格的`@Repository`接口的实现。 只要这些`@Repositories`包含在`@EnableAutoConfiguration`类的相同包（或子包）中，Spring Boot就可以为您处理所有这些`@Repositories`。

对于许多应用程序，你需要的是把正确的数据春季依赖于你的classpath（有一个`spring-boot-starter-data-jpa`的JPA和`spring-boot-starter-data-mongodb`MongoDB的），并创建一些库接口来处理你`@Entity`对象。 例子是JPA sample和Mongodb sample。

根据Spring发现的`@EnableAutoConfiguration`，Spring Boot尝试猜测`@Repository`定义的位置。 要获得更多控制权，请使用`@EnableJpaRepositories`注释（来自Spring Data JPA）。

有关Spring Data的更多信息，请参阅 Spring Data project page 。

80.4 Separate @Entity Definitions from Spring Configuration 译：80.4来自Spring配置的单独的@实体定义

根据它发现的`@EnableAutoConfiguration`，Spring Boot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 {

 //...

}
```

80.5 Configure JPA Properties 译：80.5配置JPA属性

Spring Data JPA已经提供了一些与供应商无关的配置选项（比如那些用于SQL日志记录的配置选项），并且Spring Boot公开了这些选项以及一些Hibernate作为外部配置属性。 其中一些是根据上下文自动检测的，所以您不必设置它们。

`spring.jpa.hibernate.ddl-auto`是一个特殊情况，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并且没有模式管理器（例如Liquibase或Flyway）处理`DataSource`，则默认为`create-drop`。 在所有其他情况下，它默认为`none`。

使用的方言也会根据当前的`DataSource`自动检测，但如果您想明确并在启动时绕过该检查，则可以`spring.jpa.database`设置`spring.jpa.database`。

指定`database`将导致定义明确的Hibernate方言的配置。 几个数据库有多个`Dialect`，这可能不适合您的需要。 在这种情况下，您可以设置`spring.jpa.database`到`default`以让Hibernate弄清楚事情或通过设置`spring.jpa.database-platform`属性来设置方言。

以下示例显示了最常见的设置选项：

```
spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhysicalNamingStrategy
spring.jpa.show-sql=true
```

另外，当创建本地`EntityManagerFactory`时，`spring.jpa.properties.*`中的所有属性都将作为正常的JPA属性（删除了前缀）`EntityManagerFactory`。

如果您需要将高级定制应用于Hibernate属性，请考虑注册将在创建`EntityManagerFactory`之前调用的`HibernatePropertiesCustomizer` bean。 这优先于由自动配置应用的任何内容。

80.6 Configure Hibernate Naming Strategy 译：80.6配置Hibernate命名策略

Hibernate使用two different naming strategies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`属性来配置物理和隐式策略实现的完全限定类名称。 或者，如果`ImplicitNamingStrategy`或`PhysicalNamingStrategy` bean在应用程序上下文中可用，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。 这个实现提供了和Hibernate 4一样的表结构：所有的点都被下划线替代，骆驼套也被下划线替代。 默认情况下，所有表名均以小写形式生成，但如果您的模式需要它，则可以覆盖该标志。

例如，一个`TelephoneNumber`实体被映射到`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，您可以配置下列bean：

```
@Bean
public PhysicalNamingStrategy physicalNamingStrategy() {
 return new PhysicalNamingStrategyStandardImpl();
}
```

有关更多详细信息，请参阅`HibernateJpaAutoConfiguration`和`JpaBaseConfiguration`。

80.7 Use a Custom EntityManagerFactory 译：80.7使用自定义EntityManagerFactory

要采取的配置完全控制`EntityManagerFactory`，你需要添加一个`@Bean`命名为€~entityManagerFactoryâ€™。 Spring Boot自动配置在存在该类型bean的情况下关闭其实体管理器。

80.8 Use Two EntityManagers 译：80.8使用两个EntityManagers

即使默认`EntityManagerFactory`正常工作，您需要定义一个新的。 否则，该类型的第二个bean的存在会关闭默认值。 为了`EntityManagerBuilder`操作，您可以使用Spring Boot提供的方便的`EntityManagerBuilder`。 或者，您可以直接从Spring ORM `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自行完成。 要完成此图片，您还需要为两个`EntityManagers`配置`TransactionManagers`。 如果您将其中的一个标记为`@Primary`，则可以使用Spring Boot中的默认`JpaTransactionManager`。 另一个必须明确地注入一个新的实例。 或者，您可以使用跨越两者的JTA事务管理器。

如果您使用Spring Data，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 {
 ...
}
```

80.9 Use a Traditional `persistence.xml` File译：80.9使用传统的`persistence.xml`文件

Spring Boot默认不会搜索或使用`META-INF/persistence.xml`。 如果你喜欢使用传统`persistence.xml`，你需要定义自己的`@Bean`型`LocalEntityManagerFactoryBean`（带有€~entityManagerFactoryâ€™的ID），并设置持久性单元的名称出现。

有关默认设置，请参阅`JpaBaseConfiguration`。

80.10 Use Spring Data JPA and Mongo Repositories译：80.10使用 Spring Data JPA和 Mongo Repositories

Spring Data JPA和Spring Data Mongo都可以为您自动创建`Repository`实现。 如果它们都出现在类路径中，则可能需要做一些额外的配置来告诉Spring Boot要创建的存储库。 最明确的方法是使用标准的Spring Data `@EnableJpaRepositories`和`@EnableMongoRepositories`注释并提供`Repository`接口的位置。

还有一些标志（`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存储库并仍使用自动配置的`MongoTemplate`。

其他自动配置的Spring Data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。

80.11 Expose Spring Data Repositories as REST Endpoint译：801将 Spring数据存储库作为 REST端点公开

Spring Data REST可以为您提供`Repository`实现作为REST端点，前提是为应用程序启用了Spring MVC。

Spring Boot公开了一组自定义`RepositoryRestConfiguration`的有用属性（来自`spring.data.rest`名称空间）。 如果您需要提供其他定制，则应使用`RepositoryRestConfigurer` bean。

如果您没有在自定义`RepositoryRestConfigurer`上指定任何顺序，它将在一个Spring Boot在内部使用后运行。 如果您需要指定订单，请确保它大于0。

80.12 Configure a Component that is Used by JPA译：80.12配置 JPA使用的组件

如果你想配置一个JPA使用的组件，那么你需要确保组件在JPA之前被初始化。 当组件自动配置时，Spring Boot会为您提供帮助。 例如，当Flyway被自动配置时，Hibernate被配置为依赖于Flyway，因此在Hibernate尝试使用它之前，Flyway有机会初始化数据库。

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式。 例如，如果将Hibernate Search与Elasticsearch一起用作其索引管理器，`EntityManagerFactory`必须将任何`EntityManagerFactory` bean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");
 }

}
```

80.13 Configure jOOQ with Two DataSources 译：80.13用两个数据源配置jOOQ

如果您需要将jOOQ与多个数据源一起使用，则应为每个数据源创建您自己的`DSLContext`。 有关更多详细信息，请参阅JooqAutoConfiguration 。

特别是， `JooqExceptionTranslator`和 `SpringTransactionProvider`可以重复使用，以提供与单个 `DataSource`自动配置所做的功能类似的功能。

81. Database Initialization 译：81.数据库初始化

SQL数据库可以用不同的方式初始化，具体取决于你的堆栈是什么。 当然，如果数据库是一个独立的进程，您也可以手动完成。

81.1 Initialize a Database Using JPA 译：81.1使用JPA初始化数据库

JPA具有用于生成DDL的功能，可以将这些功能设置为在启动时针对数据库运行。 这是通过两个外部属性来控制的：

- `spring.jpa.generate-ddl` (boolean) switches the feature on and off and is vendor independent.
- `spring.jpa.hibernate.ddl-auto` (enum) is a Hibernate feature that controls the behavior in a more fine-grained way. This feature is described in more detail later in this guide.

81.2 Initialize a Database Using Hibernate 译：81.2使用Hibernate初始化数据库

您可以设置`spring.jpa.hibernate.ddl-auto`明确和标准的Hibernate属性值是`none`， `validate`， `update`， `create`，并`create-drop`。 Spring Boot根据是否认为数据库已嵌入为您选择默认值。 如果没有检测到模式管理器，则默认为`create-drop`，或者在所有其他情况下为`none`。 通过查看`Connection`类型来检测嵌入式数据库。 `hsqldb`， `h2`，并`derby`嵌入，而有些则没有。 从内存切换到“真实”数据库时，请小心谨慎，因为您不需要对新平台中表和数据的存在情况进行假设。 您必须明确设置`ddl-auto`或使用其他机制之一来初始化数据库。

您可以通过启用`org.hibernate.SQL`记录器来输出架构创建。 如果启用debug mode，则会自动为您完成。

另外，如果Hibernate从头开始创建模式（即`ddl-auto`属性设置为`create`或`create-drop`），则在启动时会在启动时执行类路径根目录中名为`import.sql`的文件。 如果您非常小心，这对于演示和测试可能很有用，但可能不是您想要在生产中的类路径上进行的操作。 它是一个Hibernate特性（并且与Spring无关）。

81.3 Initialize a Database 译：81.3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）。 它从标准的根类路径位置分别加载SQL： `schema.sql`和`data.sql`。 此外，弹簧引导处理`schema-${platform}.sql`个`data-${platform}.sql`文件（如果存在），其中`platform`是的值`spring.datasource.platform`。 这使您可以根据需要切换到数据库特定的脚本。 例如，您可以选择将其设置为数据库中的供应商名称（ `hsqldb`， `h2`， `oracle`， `mysql`， `postgresql`，等等）。

Spring Boot会自动创建嵌入`DataSource`的模式。 此行为可以通过使用`spring.datasource.initialization-mode`属性进行自定义。 例如，如果您想始终初始化`DataSource`而不考虑其类型：

```
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。

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`，但不能同时执行这两个操作。 确保禁用`spring.jpa.hibernate.ddl-auto`如果使用`schema.sql`。

81.4 Initialize a Spring Batch Database 译：81.4初始化Spring批处理数据库

如果您使用Spring Batch，则它将为大多数常用数据库平台预先打包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。 如果您使用嵌入式数据库，则默认情况下发生这种情况 您还可以为任何数据库类型启用它，如以下示例中所示：

```
spring.batch.initialize-schema=always
```

您也可以通过设置 `spring.batch.initialize-schema=never`明确地关闭初始化。

81.5 Use a Higher-level Database Migration Tool 译：81.5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： Flyway和 Liquibase 。

81.5.1 Execute Flyway Database Migrations on Startup 译：81.5.1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`格式的`V<VERSION>__<NAME>.sql`（`<VERSION>`为下划线分隔的版本，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，但可以通过设置`spring.flyway.locations`来修改该位置。您还可以添加特殊的`{vendor}`占位符以使用供应商特定的脚本。假设如下：

```
spring.flyway.locations=db/migration/{vendor}
```

上述配置不是使用`db/migration`，而是根据数据库的类型（如MySQL的`db/migration/mysql`）设置要使用的文件夹。受支持的数据库列表可在`DatabaseDriver`中找到。

有关可用设置（如架构和其他）的详细信息，请参阅flyway-core的Flyway类。另外，Spring Boot提供了一组属性（在`FlywayProperties`中），可用于禁用迁移或关闭位置检查。Spring Boot调用`Flyway.migrate()`来执行数据库迁移。如果您想了解更多的控制，提供了一个`@Bean`实现`FlywayMigrationStrategy`。

Flyway支持SQL和Java callbacks。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`FlywayCallback`或最好是扩展`BaseFlywayCallback`。任何这样的豆类都会自动注册到`Flyway`。可以使用`@Order`或通过执行`Ordered`来订购它们。

默认情况下，`DataSource`在您的上下文中自动`@Primary`（`@Primary`）`DataSource`，并将其用于迁移。如果您想使用不同的`DataSource`，则可以创建一个并将其`@Bean`标记为`@FlywayDataSource`。如果你这样做，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`DataSource`使用其自己的`DataSource`。如果没有设置这三个属性中的任何一个，则将使用其等效`spring.datasource`属性的值。

有一个 Flyway sample，这样你就可以看到如何设置。

您还可以使用Flyway为特定场景提供数据。例如，您可以在`src/test/resources`放置特定于测试的迁移，并且仅在您的应用程序启动进行测试时才运行它们。另外，您可以使用配置文件特定配置来自定义`spring.flyway.locations`以便某些迁移仅在特定配置文件处于活动状态时才运行。例如，在`application-dev.properties`，您可以指定以下设置：

```
spring.flyway.locations=classpath:/db/migration,classpath:/dev/db/migration
```

通过该设置，仅当`dev`配置文件处于活动状态时`dev/db/migration`运行`dev/db/migration`迁移。

81.5.2 Execute Liquibase Database Migrations on Startup译：81.5.2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`添加到您的类路径中。

默认情况下，从`db/changelog/db.changelog-master.yaml`读取主更改日志，但可以通过设置`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在您的上下文中自动装载（`@Primary`）`DataSource`并将其用于迁移。如果您需要使用不同的`DataSource`，则可以创建一个并将其`@Bean`标记为`@LiquibaseDataSource`。如果你这样做，并且你想要两个数据源，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的原生`DataSource`。设置`spring.liquibase.url`或`spring.liquibase.user`足以使Liquibase使用自己的`DataSource`。如果没有设置这三个属性中的任何一个，则将使用其等效的`spring.datasource`属性的值。

有关可用设置的详细信息，请参阅`LiquibaseProperties`，例如上下文，默认模式等。

有一个 Liquibase sample，这样你就可以看到如何设置。

82. Messaging译：82.消息

Spring Boot提供了许多包含消息的初学者。本节回答了使用Spring Boot进行消息传递时出现的问题。

82.1 Disable Transacted JMS Session译：82.1禁用事务处理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处理。如果您想使用`DefaultJmsListenerContainerFactoryConfigurer`来重用Spring Boot的默认设置，则可以禁用事务会话，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;
}
```

上面的例子覆盖了默认的工厂，它应该被应用到你的应用程序定义的任何其他工厂（如果有的话）。

83. Batch Applications译：83.批量应用程序

本节回答在Spring Boot中使用Spring Batch时出现的问题。

默认情况下，批处理应用程序需要`DataSource`来存储作业详细信息。如果你想偏离这一点，你需要实现`BatchConfigurer`。详情请参阅The Javadoc of `@EnableBatchProcessing`。

有关Spring Batch的更多信息，请参阅 Spring Batch project page。

83.1 Execute Spring Batch Jobs on Startup译：83.1在启动时执行Spring批处理作业

Spring Batch自动配置通过在您的上下文中添加`@EnableBatchProcessing`（来自Spring Batch）来启用。

默认情况下，它在启动时执行应用程序上下文中的**所有**`Jobs`（有关详细信息，请参见JobLauncherCommandLineRunner）。您可以通过指

定`spring.batch.job.names`（它采用逗号分隔的作业名称模式列表）来缩小特定作业或作业范围。

如果应用程序上下文包含`JobRegistry`，则在`spring.batch.job.names`中查找`spring.batch.job.names`中的作业，而不是从上下文自动装入。 对于更复杂的系统，这是一种常见模式，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 。

84. Actuator译：84.执行器

Spring Boot包含Spring Boot Actuator。 本部分回答了使用中经常出现的问题。

84.1 Change the HTTP Port or Address of the Actuator Endpoints译：84.1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。 要使应用程序在不同的端口上侦听，请设置外部属性：`management.server.port`。 要监听一个完全不同的网络地址（例如，当您有一个内部网络管理和一个外部网络管理用户应用程序时），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“ Section 51.2, “Customizing the Management Server Port”” 。

84.2 Customize the ‘whitelabel’ Error Page译：84.2自定义“白标签”错误页面

如果您遇到服务器错误（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），Spring Boot会安装您在浏览器客户端中看到的“白色标签”错误页面。

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仍会尝试解决错误视图，因此您应该添加自己的错误页面，而不是完全禁用它。

用你自己覆盖错误页面取决于你使用的模板技术。 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。 在一般情况下，你需要一个`View`，随着一个名称解析`error`或者`@Controller`处理该`/error`路径。 除非你更换了一些默认配置，你应该找一个`BeanNameViewResolver`在`ApplicationContext`，所以`@Bean`命名为`error`是这样做的一个简单的方法。 有关更多选项，请参阅`ErrorMvcAutoConfiguration` 。

有关如何在servlet容器中注册处理程序的详细信息，另请参阅“ Error Handling ”一节。

84.3 Sanitize sensible values译：84.3消除敏感的价值

由`env`和`configprops`端点返回的信息可能有些敏感，因此默认情况下会对与特定模式匹配的键进行消毒（即它们的值由`(******)`替代）。

Spring Boot对这些键使用合理的默认设置：例如，任何以“password”，“secret”，“key”或“token”结尾的键都将被清理。 也可以使用正则表达式，例如`credentials.`来`credentials.`任何包含单词`credentials`的密钥作为密钥的一部分。

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`management.endpoint.env.keys-to-sanitize` 。

85. Security译：85.安全

本节讨论有关使用Spring Boot时的安全性问题，包括使用Spring Security和Spring Boot引起的问题。

有关Spring Security的更多信息，请参阅 Spring Security project page 。

85.1 Switch off the Spring Boot Security Configuration译：85.1关闭Spring Boot安全配置

如果您在应用程序中使用`WebSecurityConfigurerAdapter`定义了`@Configuration`，它将关闭Spring Boot中的默认`WebSecurityConfigurerAdapter`应用程序安全设置。

85.2 Change the UserDetailsService and Add User Accounts译：85.2更改UserDetailsService和添加用户帐户

如果你提供了一个`@Bean`型`AuthenticationManager`，`AuthenticationProvider`，或`UserDetailsService`，默认`@Bean`为`InMemoryUserDetailsManager`未创建，让你有完整的功能集的Spring Security的可用（如 various authentication options ）。

添加用户帐户的最简单方法是提供您自己的`UserDetailsService` bean。

85.3 Enable HTTPS When Running behind a Proxy Server译：85.3在代理服务器后运行时启用HTTPS

确保所有主要端点都只能通过HTTPS访问，这对于任何应用程序来说都是一件非常重要的事情。 如果您使用Tomcat作为servlet容器，那么Spring Boot会自动添加Tomcat自己的`RemoteIpValve`如果它检测到某些环境设置，并且您应该能够依靠`HttpServletRequest`来报告它是否安全（即使是在代理服务器处理真正的SSL终止）。 标准行为取决于是否存在某些请求头（`x-forwarded-for`和`x-forwarded-proto`），其名称是常规的，所以它应该可以与大多数前端代理一起使用。 您可以通过向`application.properties`添加一些条目来`application.properties`，如以下示例所示：

```
server.tomcat.remote-ip-header=x-forwarded-for
server.tomcat.protocol-header=x-forwarded-proto
```

（任一这些特性的存在开关上的阀。另外，也可以添加`RemoteIpValve`通过添加`TomcatServletWebServerFactory`豆）。

要将Spring Security配置为需要所有（或某些）请求的安全通道，请考虑添加您自己的`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();
 }

}
```

86. Hot Swapping译: 86.热插拔

Spring Boot支持热插拔。 本节回答有关它如何工作的问题。

86.1 Reload Static Content译: 86.1重新加载静态内容

有几种热重载的选项。 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，例如对快速应用程序重新启动和LiveReload的支持以及合理的开发时配置（如模板缓存）。 Devtools通过监视类路径的变化来工作。 这意味着静态资源更改必须“建立”，以使更改生效。 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命令触发必要的构建。 由于default restart exclusions，对静态资源的更改不会触发应用程序的重新启动。 但是，他们确实会触发实时重新加载。

或者，在IDE中运行（特别是在调试时）是开发的好方法（所有现代IDE允许重新加载静态资源，并且通常还允许热切换Java类更改）。

最后，可以配置Maven and Gradle plugins （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。 如果使用更高级别的工具编写该代码，则可以将其用于外部css / js编译器进程。

86.2 Reload Templates without Restarting the Container译: 86.2在不重新启动容器的情况下重新加载模板

Spring Boot支持的大多数模板技术都包含一个禁用缓存的配置选项（稍后在本文档中介绍）。 如果您使用`spring-boot-devtools`模块，则在开发时为您提供这些属性为automatically configured 。

86.2.1 Thymeleaf Templates译: 86.2.1 Thymeleaf模板

如果您使用Thymeleaf，请将`spring.thymeleaf.cache`设置为`false` 。 有关其他Thymeleaf自定义选项，请参阅`ThymeleafAutoConfiguration` 。

86.2.2 FreeMarker Templates译: 86.2.2 FreeMarker模板

如果您使用FreeMarker，请将`spring.freemarker.cache`设置为`false` 。 有关其他FreeMarker自定义选项，请参阅`FreeMarkerAutoConfiguration` 。

86.2.3 Groovy Templates译: 86.2.3 Groovy模板

如果您使用Groovy模板，请将`spring.groovy.template.cache`设置为`false` 。 有关其他Groovy自定义选项，请参阅`GroovyTemplateAutoConfiguration` 。

86.3 Fast Application Restarts译: 86.3快速应用程序重新启动

`spring-boot-devtools`模块包含对自动应用程序重新启动的支持。 虽然速度不如JRebel等技术，但通常比“冷启动”快得多。 在调查本文稍后讨论的一些更复杂的重新加载选项之前，您应该尝试一下。

有关更多详细信息，请参阅 Chapter 20, *Developer Tools*部分。

86.4 Reload Java Classes without Restarting the Container译: 86.4重新加载Java类而不重新启动容器

许多现代IDE（Eclipse，IDEA和其他）支持热插拔字节码。 因此，如果您进行的更改不会影响类或方法签名，则应该重新加载干净且无副作用。

87. Build译: 87.建立

Spring Boot包含Maven和Gradle的构建插件。 本节回答关于这些插件的常见问题。

87.1 Generate Build Information译: 87.1生成生成信息

Maven插件和Gradle插件都允许生成包含项目坐标，名称和版本的构建信息。 插件也可以配置为通过配置添加其他属性。 当这样的文件存在时，Spring Boot会自动配置一个`BuildProperties` bean。

要使用Maven生成构建信息，请为`build-info`目标添加一个执行，如以下示例所示：

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <version>2.0.3.RELEASE</version>
   <executions>
    <execution>
     <goals>
      <goal>build-info</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

有关更多详细信息，请参阅 Spring Boot Maven Plugin documentation 。

下面的例子和Gradle一样：

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

有关更多详细信息，请参阅 Spring Boot Gradle Plugin documentation 。

87.2 Generate Git Information 译：87.2生成Git信息

Maven和Gradle都允许生成一个 `git.properties` 文件，该文件包含有关项目生成时 `git` 源代码库状态的信息。

对于Maven用户， `spring-boot-starter-parent` POM包含一个预配置的插件以生成 `git.properties` 文件。 要使用它，请将以下声明添加到您的POM中：

```
<build>
 <plugins>
  <plugin>
   <groupId>pl.project13.maven</groupId>
   <artifactId>git-commit-id-plugin</artifactId>
  </plugin>
 </plugins>
</build>
```

Gradle用户可以使用 `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 "1.4.21"
}
```

预计在 `git.properties` 的提交时间将与以下格式匹配： `yyyy-MM-dd'T'HH:mm:ssZ` 。 这是上面列出的两个插件的默认格式。 使用这种格式可以将时间解析为 `Date` 及其格式，并将其序列化为JSON，并由Jackson的日期序列化配置设置进行控制。

87.3 Customize Dependency Versions 译：87.3自定义相关性版本

如果您使用直接或间接从 `spring-boot-dependencies` （例如 `spring-boot-starter-parent` ）继承的Maven构建，但想要覆盖特定的第三方依赖关系，则可以添加适当的 `<properties>` 元素。 浏览 `spring-boot-dependencies` POM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>
```

这样做只在您的Maven项目从 `spring-boot-dependencies` 继承（直接或间接）的情况下 `spring-boot-dependencies` 。 如果您在 `spring-boot-dependencies` 自己的 `dependencyManagement` 部分中添加了 `<scope>import</scope>` ，则必须自己重新定义该神器，而不是重写该属性。

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。

要覆盖依赖版本的摇篮，看到 this section的摇篮pluginâ€™的文档。

87.4 Create an Executable JAR with Maven 译：87.4使用Maven创建可执行JAR

`spring-boot-maven-plugin` 可以用来创建一个可执行的“fat”JAR。 如果您使用 `spring-boot-starter-parent` POM，则可以声明该插件，并将您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>
```

如果你不使用父POM，你仍然可以使用插件。 但是，您必须另外添加一个`<executions>`部分，如下所示：

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <version>2.0.3.RELEASE</version>
   <executions>
    <execution>
     <goals>
      <goal>repackage</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

有关完整的使用细节，请参阅 plugin documentation 。

87.5 Use a Spring Boot Application as a Dependency译：87.5使用 Spring Boot应用程序作为依赖项

像战争文件一样，Spring Boot应用程序不打算用作依赖项。 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入单独的模块。 这个单独的模块可以被你的应用程序和其他项目所依赖。

如果您不能像上面推荐的那样重新排列代码，那么Spring Boot的Maven和Gradle插件必须配置为生成适合用作依赖项的单独工件。 可执行文件不能用作依赖项，因为executable jar format将应用程序包分类为`BOOT-INF/classes` 。 这意味着当可执行jar用作依赖项时，它们不能被找到。

为了产生两个工件，一个可以用作依赖关系和一个可执行工件，必须指定一个分类器。 此分类器应用于可执行档案的名称，保留默认归档以用作依赖项。

要在Maven中配置 `exec`的分类器，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>
```

87.6 Extract Specific Libraries When an Executable Jar Runs译：87.6当可执行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以便运行。 但是，某些图书馆可能会遇到问题。 例如，JRuby包含自己的嵌套jar支持，它假定`jruby-complete.jar`总是直接作为文件独立存在。

要处理任何有问题的库，你可以标记特定的嵌套jar应该在可执行jar第一次运行时自动解压到“temp文件夹”。

例如，为了表明应该通过使用Maven插件标记JRuby进行解包，您可以添加以下配置：

```
<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>
```

87.7 Create a Non-executable JAR with Exclusions译：87.7创建带有排除项的非可执行 JAR

通常，如果您将可执行文件和不可执行的jar作为两个独立的构建产品，那么可执行版本具有库jar中不需要的其他配置文件。 例如， `application.yml`配置文件可能被排除在非可执行JAR之外。

在Maven中，可执行jar必须是主要的工件，你可以为库添加一个分类的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>
```

87.8 Remote Debug a Spring Boot Application Started with Maven 译：87.8远程调试Spring引导应用程序从Maven开始

要安装远程调试到开始使用Maven一个春天启动的应用程序，你可以使用 `jvmArguments` 的财产 maven plugin 。

有关更多详细信息，请参阅 this example 。

87.9 Build an Executable Archive from Ant without Using `spring-boot-antlib` 译：87.9在不使用 `spring-boot-antlib` 情况下从Ant生成可执行文件

要使用Ant进行构建，您需要获取依赖关系，编译并创建jar或war归档文件。 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. If you are building a jar, package the application's classes and resources in a nested `BOOT-INF/classes` directory. If you are building a war, package the application's classes in a nested `WEB-INF/classes` directory as usual.
2. Add the runtime dependencies in a nested `BOOT-INF/lib` directory for a jar or `WEB-INF/lib` for a war. Remember **not** to compress the entries in the archive.
3. Add the `provided` (embedded container) dependencies in a nested `BOOT-INF/lib` directory for a jar or `WEB-INF/lib-provided` for a war. Remember **not** to compress the entries in the archive.
4. Add the `spring-boot-loader` classes at the root of the archive (so that the `Main-Class` is available).
5. Use the appropriate launcher (such as `JarLauncher` for a jar file) as a `Main-Class` attribute in the manifest and specify the other properties it needs as manifest entries — principally, by setting a `Start-Class` property.

以下示例显示如何使用Ant构建可执行档案：

```
<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>
```

Ant Sample有一个 `build.xml` 文件，其中包含 `manual` 任务，如果使用以下命令运行它，该任务应该可以工作：

```
$ 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
```

88. Traditional Deployment 译：88.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。

88.1 Create a Deployable War File 译：88.1创建可部署的战争文件

由于Spring WebFlux不严格依赖于Servlet API，并且应用程序默认部署在嵌入式Reactor Netty服务器上，因此WebFlux应用程序不支持War部署。

生成可部署战争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，并允许您在应用程序由servlet容器启动时进行配置。 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);
 }

}
```

下一步是更新您的构建配置，以便您的项目生成war文件而不是jar文件。 如果您使用Maven和`spring-boot-starter-parent`（它为您配置Maven的战争插件），则您只需修改`pom.xml`即可将包装更改为war，如下所示：

```
<packaging>war</packaging>
```

如果您使用Gradle，则需要修改`build.gradle`以将战争插件应用于项目，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容器依赖项标记为提供。

如果您使用Maven，则下面的示例将servlet容器（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>
```

如果您使用Gradle，则以下示例将servlet容器（本例中为Tomcat）标记为正在提供：

```
dependencies {
 // …
 providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
 // …
}
```

`providedRuntime`优于Gradle的`compileOnly`配置。 除其他限制外， `compileOnly`依赖项不在测试类路径中，因此任何基于Web的集成测试都会失败。

如果使用Spring Boot build tools，则标记所提供的嵌入式Servlet容器依赖关系会生成一个可执行的war文件，并提供打包在`lib-provided`目录中的依赖关系。 这意味着，除了可部署到servlet容器外，您还可以通过在命令行上使用`java -jar`来运行应用程序。

查看Spring Boot的示例应用程序， 了解上述配置的 Maven-based example 。

88.2 Convert an Existing Application to Spring Boot 译：88.2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，请丢弃创建您的`ApplicationContext`的代码，并将其替换为`SpringApplication`或`SpringApplicationBuilder`调用。 Spring MVC Web应用程序通常可以首先创建可部署的战争应用程序，然后将其稍后迁移到可执行的战争或jar。 请参阅Getting Started Guide on Converting a jar to a war 。

要通过扩展`SpringBootServletInitializer`（例如，在名为`Application`的类中）并添加Spring Boot `@SpringBootApplication`注释来创建可展开的战争，请使用类似于以下示例中显示的代码：

```
@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，无论你放在`sources`中，只是一个春天`ApplicationContext`。 通常，任何已经有效的东西都应该在这里工作。 可能稍后会删除一些bean，并让Spring Boot为它们提供自己的默认值，但在您需要之前应该可以做些工作。

可以将静态资源移动到类路径根中的`/public`（或`/static`或`/resources`或`/META-INF/resources`）。 这同样适用于`messages.properties`（Spring Boot自动检测类路径的根目录）。

Spring `DispatcherServlet`和Spring Security的香草应用不需要进一步修改。 如果您的应用程序中有其他功能（例如，使用其他servlet或过滤器），则可能需要将一些配

置添加到`Application`上下文中，方法是从`web.xml`替换这些元素，如下所示：

- A `@Bean` of type `Servlet` or `ServletRegistrationBean` installs that bean in the container as if it were a `<servlet/>` and `<servlet-mapping/>` in `web.xml`.
- A `@Bean` of type `Filter` or `FilterRegistrationBean` behaves similarly (as a `<filter/>` and `<filter-mapping/>`).
- An `ApplicationContext` in an XML file can be added through an `@ImportResource` in your `Application`. Alternatively, simple cases where annotation configuration is heavily used already can be recreated in a few lines as `@Bean` definitions.

一旦战争文件的工作，你可以把它可执行通过增加`main`方法你`Application`，如下面的例子：

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

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`SpringBootServletInitializer`回调可用的方法和`main`方法中类似以下类的方法来共享构建器的自定义设置：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
  return configureApplication(builder);
 }

 public static void main(String[] args) {
  configureApplication(new SpringApplicationBuilder()).run(args);
 }

 private static SpringApplicationBuilder configureApplication(SpringApplicationBuilder builder) {
  return builder.sources(Application.class).bannerMode(Banner.Mode.OFF);
 }

}
```

应用程序可以分为多个类别：

- Servlet 3.0+ applications with no `web.xml`.
- Applications with a `web.xml`.
- Applications with a context hierarchy.
- Applications without a context hierarchy.

所有这些都应该适合翻译，但每个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+初始化器支持类，它们可能会很容易转换。 通常，来自现有`WebApplicationInitializer`所有代码都可以移入`SpringBootServletInitializer`。 如果您现有的应用程序有多个`ApplicationContext`（例如，如果它使用`AbstractDispatcherServletInitializer`），那么您可能可以将所有上下文源合并为一个`SpringApplication`。 您可能遇到的主要难题是如果组合不起作用并且您需要维护上下文层次结构。 有关示例，请参见entry on building a hierarchy。 包含Web特定功能的现有父上下文通常需要分解，以便所有`ServletContextAware`组件都位于子上下文中。

不是Spring应用程序的应用程序可能会转换为Spring Boot应用程序，前面提到的指导可能会有所帮助。 但是，您可能会遇到问题。 在这种情况下，我们建议asking questions on Stack Overflow with a tag of `spring-boot`。

88.3 Deploying a WAR to WebLogic译：88.3将WAR部署到WebLogic

要将Spring Boot应用程序部署到WebLogic，必须确保您的servlet初始化程序 **直接** 实现`WebApplicationInitializer`（即使您已从实现它的基类中进行扩展）。

WebLogic的典型初始化程序应该类似于以下示例：

```
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更喜欢打包的版本，而不是与服务器预安装的版本。 您可以通过添加具有以下内容的`WEB-INF/weblogic.xml`文件来完成此操作：

```
<?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>
```

88.4 Use Jedis Instead of Lettuce译：88.4使用Jedis代替生菜

默认情况下，Spring Boot启动器（`spring-boot-starter-data-redis`）使用Lettuce。 您需要排除该依赖关系，并包含Jedis。 Spring Boot管理这些依赖关系，以便尽可能简化此过程。

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")
 // ...
}
```

Part X. Appendices译:第十部分附录

Appendix A. Common application properties译:附录A通用应用程序属性

各种属性可以在内部被指定`application.properties`文件，您的内部`application.yml`文件，或作为命令行开关。 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。

属性贡献可能来自类路径上的其他jar文件，因此您不应将其视为详尽的列表。 此外，你可以定义你自己的属性。

此示例文件仅作为指导。 **不要**复制和粘贴的全部内容到应用程序中。 相反，只挑选您需要的属性。

```
# ===================================================================
# 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.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 pro

# 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=0ms # Entry expiration. By default the entries never expire. Note that this value is ultimately converted to second
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. Ne
spring.cache.redis.cache-null-values=true # Allow caching null values.
spring.cache.redis.key-prefix= # Key prefix.
spring.cache.redis.time-to-live=0ms # 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.location=classpath:META-INF/build-info.properties # Location of the generated build-info.properties file.
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.

# 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.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 Resource
spring.messages.cache-duration= # Loaded resource bundle files cache duration. When not set, bundles are cached forever. If a duration suffix is not
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 "NoSuchMessageExcept

# 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.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 sch
spring.quartz.job-store-type=memory # Quartz job store type.
spring.quartz.properties.*= # Additional Quartz Scheduler properties.

# 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.


# ----------------------------------------
# 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= # List of user-agents to exclude from compression.
server.compression.mime-types=text/html,text/xml,text/plain,text/css,text/javascript,application/javascript # Comma-separated list of MIME types that
server.compression.min-response-size=2048 # 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 containe
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= # Number of acceptor threads to use.
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=0 # Maximum size, in bytes, of the HTTP post or put content.
server.jetty.selectors= # Number of selector threads to use.
server.max-http-header-size=0 # Maximum size, in bytes, 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 # The class name of the JSP servlet.
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.path=/ # Path of the main dispatcher servlet.
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= # "HttpOnly" flag for the session cookie.
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= # "Secure" flag for the session cookie.
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= # Session timeout. If a duration suffix is not specified, seconds will be used.
server.servlet.session.tracking-modes= # Session tracking modes (one or more of the following: "cookie", "url", "ssl").
server.ssl.ciphers= # Supported SSL ciphers.
server.ssl.client-auth= # Whether client authentication is wanted ("want") or needed ("need"). Requires a trust store.
server.ssl.enabled= # 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=0 # 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 reque
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=30s # Delay between the invocation of backgroundProcess methods. If a duration suffix is not specified, secc
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} # Regular expression matching trusted IP addresses.
server.tomcat.max-connections=0 # Maximum number of connections that the server accepts and processes at any given time.
server.tomcat.max-http-header-size=0 # Maximum size, in bytes, of the HTTP message header.
server.tomcat.max-http-post-size=0 # Maximum size, in bytes, of the HTTP post content.
server.tomcat.max-threads=0 # Maximum number of worker threads.
server.tomcat.min-spare-threads=0 # Minimum number 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= # 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.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
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, in bytes.
server.undertow.direct-buffers= # Whether to allocate buffers outside the Java heap.
server.undertow.io-threads= # Number of I/O threads to create for the worker.
server.undertow.eager-filter-init=true # Whether servlet filters should be initialized on startup.
server.undertow.max-http-post-size=0 # Maximum size, in bytes, of the HTTP post content.
server.undertow.worker-threads= # Number of worker threads.

# FREEMARKER (FreeMarkerProperties)
spring.freemarker.allow-request-override=false # Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attr
spring.freemarker.allow-session-override=false # Whether HttpSession attributes are allowed to override (hide) controller generated model attributes
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 templat
spring.freemarker.expose-spring-macro-helpers=true # Whether to expose a RequestContext for use by Spring's macro library, under the name "springMacr
spring.freemarker.prefer-file-system-access=true # Whether to prefer file system access for template loading. File system access enables hot detectio
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
spring.groovy.template.allow-session-override=false # Whether HttpSession attributes are allowed to override (hide) controller generated model attrib
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 templa
spring.groovy.template.expose-session-attributes=false # Whether all HttpSession attributes should be added to the model prior to merging with the te
spring.groovy.template.expose-spring-macro-helpers=true # Whether to expose a RequestContext for use by Spring's macro library, under the name "sprin
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/js

# HTTP message conversion
spring.http.converters.preferred-json-mapper= # Preferred JSON mapper to use for HTTP message conversion. By default, auto-detected according to the

# HTTP encoding (HttpEncodingProperties)
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
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.

# MULTIPART (MultipartProperties)
spring.servlet.multipart.enabled=true # Whether to enable support of multipart uploads.
spring.servlet.multipart.file-size-threshold=0 # Threshold after which files are written to disk. Values can use the suffixes "MB" or "KB" to indicat
spring.servlet.multipart.location= # Intermediate location of uploaded files.
spring.servlet.multipart.max-file-size=1MB # Max file size. Values can use the suffixes "MB" or "KB" to indicate megabytes or kilobytes, respectively
spring.servlet.multipart.max-request-size=10MB # Max request size. Values can use the suffixes "MB" or "KB" to indicate megabytes or kilobytes, respe
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'
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
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 Pr
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".

# 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
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 attrib
spring.mustache.allow-session-override=false # Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of
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 "springMacroR
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 mec
spring.mvc.contentnegotiation.favor-path-extension=false # Whether the path extension in the URL path should be used to determine the requested media
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.putfilter.enabled=true # Whether to enable Spring's HttpPutFormContentFilter.
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".
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 "sprir
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.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 requ
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 sha
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-validatir
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 cac
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
spring.resources.cache.cachecontrol.stale-if-error= # Maximum time the response may be used when errors are encountered, in seconds if no duration su
spring.resources.cache.cachecontrol.stale-while-revalidate= # Maximum time the response can be served after it becomes stale, in seconds if no durati
spring.resources.cache.period= # Cache period for the resources served by the resource handler. If a duration suffix is not specified, seconds will b
spring.resources.chain.cache=true # Whether to enable caching in the Resource chain.
spring.resources.chain.enabled= # Whether to enable the Spring Resource Handling chain. By default, disabled unless at least one strategy has been er
spring.resources.chain.gzipped=false # Whether to enable resolution of already gzipped resources.
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 re

# 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 databas
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 C
spring.thymeleaf.reactive.full-mode-view-names= # Comma-separated list of view names (patterns allowed) that should be executed in FULL mode even if
spring.thymeleaf.reactive.max-chunk-size=0 # Maximum size of data buffers used for writing to the response, in bytes.
spring.thymeleaf.reactive.media-types= # Media types supported by the view technology.
spring.thymeleaf.servlet.content-type=text/html # Content-Type value written to HTTP responses.
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.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.

# ----------------------------------------
# DATA PROPERTIES
# ----------------------------------------

# FLYWAY (FlywayProperties)
spring.flyway.baseline-description= #
spring.flyway.baseline-on-migrate= #
spring.flyway.baseline-version=1 # Version to start migration
spring.flyway.check-location=true # Whether to check that migration scripts location exists.
spring.flyway.clean-disabled= #
spring.flyway.clean-on-validation-error= #
spring.flyway.dry-run-output= #
spring.flyway.enabled=true # Whether to enable flyway.
spring.flyway.encoding= #
spring.flyway.error-handlers= #
spring.flyway.group= #
spring.flyway.ignore-future-migrations= #
spring.flyway.ignore-missing-migrations= #
spring.flyway.init-sqls= # SQL statements to execute to initialize a connection immediately after obtaining it.
spring.flyway.installed-by= #
spring.flyway.locations=classpath:db/migration # The locations of migrations scripts.
spring.flyway.mixed= #
spring.flyway.out-of-order= #
spring.flyway.password= # JDBC password to use if you want Flyway to create its own DataSource.
spring.flyway.placeholder-prefix= #
spring.flyway.placeholder-replacement= #
spring.flyway.placeholder-suffix= #
spring.flyway.placeholders.*= #
spring.flyway.repeatable-sql-migration-prefix= #
spring.flyway.schemas= # schemas to update
spring.flyway.skip-default-callbacks= #
spring.flyway.skip-default-resolvers= #
```

```
spring.flyway.skip-default-resolvers= #
spring.flyway.sql-migration-prefix=V #
spring.flyway.sql-migration-separator= #
spring.flyway.sql-migration-suffix=.sql #
spring.flyway.sql-migration-suffixes= #
spring.flyway.table= #
spring.flyway.target= #
spring.flyway.undo-sql-migration-prefix= #
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= #

# 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.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.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.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.keyspace-name= # Keyspace name to use.
spring.data.cassandra.load-balancing-policy= # Class name of the load balancing policy.
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 al
spring.data.cassandra.pool.idle-timeout=120s # Idle timeout before an idle connection is removed. If a duration suffix is not specified, seconds will
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.reconnection-policy= # Reconnection policy class.
spring.data.cassandra.repositories.type=auto # Type of Cassandra repositories to enable.
spring.data.cassandra.retry-policy= # Class name of the retry policy.
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 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 rec
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.

# 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 specifiec
```

```
# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
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" er
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 usir
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
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 nc
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 transactic
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') lo
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 partic
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 invalic
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 er
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 suppc
spring.jta.bitronix.connectionfactory.disabled= # Whether this resource is disabled, meaning it's temporarily forbidden to acquire a connection from
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 cor
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
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 connec
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 tra
spring.jta.bitronix.datasource.disabled= # Whether this resource is disabled, meaning it's temporarily forbidden to acquire a connection from its poo
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
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 Inte
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 wi
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 th
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 enlist

# NARAYANA (NarayanaProperties)
spring.jta.narayana.default-timeout=60s # Transaction timeout. If a duration suffix is not specified, seconds will be used.
spring.jta.narayana.expiry-scanners=com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner # Comma-separated list of expiry s
spring.jta.narayana.log-dir= # Transaction object store directory.
spring.jta.narayana.one-phase-commit=true # Whether to enable one phase commit optimization.
spring.jta.narayana.periodic-recovery-period=120s # Interval in which periodic recovery scans are performed. If a duration suffix is not specified, s
spring.jta.narayana.recovery-backoff-period=10s # Back off period between first and second phases of the recovery scan. If a duration suffix is not s
spring.jta.narayana.recovery-db-pass= # Database password to be used by the recovery manager.
spring.jta.narayana.recovery-db-user= # Database username to be used by the recovery manager.
spring.jta.narayana.recovery-jms-pass= # JMS password to be used by the recovery manager.
spring.jta.narayana.recovery-jms-user= # JMS username to be used by the recovery manager.
spring.jta.narayana.recovery-modules= # Comma-separated list of recovery modules.
spring.jta.narayana.transaction-manager-id=1 # Unique transaction manager id.
spring.jta.narayana.xa-resource-orphan-filters= # Comma-separated list of orphan filters.

# 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, in megabytes.
spring.mongodb.embedded.storage.repl-set-name= # Name of the replica set.
spring.mongodb.embedded.version=3.2.2 # 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 l
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 c
spring.redis.jedis.pool.max-wait=-1ms # Maximum amount of time a connection allocation should block before throwing an exception when the pool is exh
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
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
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
spring.redis.lettuce.pool.max-wait=-1ms # Maximum amount of time a connection allocation should block before throwing an exception when the pool is e
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
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 i
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 "JMSExcept
spring.activemq.pool.block-if-full-timeout=-1ms # Blocking period before throwing an exception if the pool is still full.
spring.activemq.pool.create-connection-on-startup=true # Whether to create a connection on startup. Can be used to warm up the pool on startup.
spring.activemq.pool.enabled=false # Whether a PooledConnectionFactory should be created, instead of a regular ConnectionFactory.
spring.activemq.pool.expiry-timeout=0ms # Connection expiration timeout.
spring.activemq.pool.idle-timeout=30s # Connection idle timeout.
spring.activemq.pool.max-connections=1 # Maximum number of pooled connections.
spring.activemq.pool.maximum-active-session-per-connection=500 # Maximum number of active sessions per connection.
spring.activemq.pool.reconnect-on-exception=true # Reset the connection when a "JMSException" occurs.
spring.activemq.pool.time-between-expiration-check=-1ms # Time to sleep between runs of the idle connection eviction thread. When negative, no idle c
spring.activemq.pool.use-anonymous-producers=true # Whether to use only one anonymous "MessageProducer" instance. Set it to false to create one "Mess

# 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.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 cont
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

# JMS (JmsProperties)
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.keystore-location= # Location of the key store file.
spring.kafka.admin.ssl.keystore-password= # Store password for the key store file.
spring.kafka.admin.ssl.keystore-type= # Type of the key store.
spring.kafka.admin.ssl.protocol= # SSL protocol to use.
spring.kafka.admin.ssl.truststore-location= # Location of the trust store file.
spring.kafka.admin.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.admin.ssl.truststore-type= # Type of the trust store.
spring.kafka.bootstrap-servers= # Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluster.
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
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 serve
spring.kafka.consumer.bootstrap-servers= # Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluste
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
spring.kafka.consumer.fetch-min-size= # Minimum amount of data, in bytes, 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.keystore-location= # Location of the key store file.
spring.kafka.consumer.ssl.keystore-password= # Store password for the key store file.
spring.kafka.consumer.ssl.keystore-type= # Type of the key store.
spring.kafka.consumer.ssl.protocol= # SSL protocol to use.
spring.kafka.consumer.ssl.truststore-location= # Location of the trust store file.
spring.kafka.consumer.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.consumer.ssl.truststore-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 us
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 in bytes.
spring.kafka.producer.bootstrap-servers= # Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluste
spring.kafka.producer.buffer-memory= # Total bytes of memory 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.keystore-location= # Location of the key store file.
spring.kafka.producer.ssl.keystore-password= # Store password for the key store file.
spring.kafka.producer.ssl.keystore-type= # Type of the key store.
spring.kafka.producer.ssl.protocol= # SSL protocol to use.
spring.kafka.producer.ssl.truststore-location= # Location of the trust store file.
spring.kafka.producer.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.producer.ssl.truststore-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.keystore-location= # Location of the key store file.
spring.kafka.ssl.keystore-password= # Store password for the key store file.
spring.kafka.ssl.keystore-type= # Type of the key store.
spring.kafka.ssl.protocol= # SSL protocol to use.
spring.kafka.ssl.truststore-location= # Location of the trust store file.
spring.kafka.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.ssl.truststore-type= # Type of the trust 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.prefetch= # Number of messages to be handled in a single request. It should be greater than or equal to the transacti
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.prefetch= # Number of messages to be handled in a single request. It should be greater than or equal to the transacti
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 in a transaction. That is, the number of messages between acks
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.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.algorithm= # SSL algorithm to use. By default, configured by the Rabbit client library.
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 managem
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. Requires a custom management.port.
management.server.ssl.client-auth= # Whether client authentication is wanted ("want") or needed ("need"). Requires a trust store. Requires a custom m
management.server.ssl.enabled= # Whether to enable SSL support. Requires a custom management.server.port.
management.server.ssl.enabled-protocols= # Enabled SSL protocols. Requires a custom management.server.port.
management.server.ssl.key-alias= # Alias that identifies the key in the key store. Requires a custom management.server.port.
management.server.ssl.key-password= # Password used to access the key in the key store. Requires a custom management.server.port.
management.server.ssl.key-store= # Path to the key store that holds the SSL certificate (typically a jks file). Requires a custom management.server.p
management.server.ssl.key-store-password= # Password used to access the key store. Requires a custom management.server.port.
management.server.ssl.key-store-provider= # Provider for the key store. Requires a custom management.server.port.
management.server.ssl.key-store-type= # Type of the key store. Requires a custom management.server.port.
management.server.ssl.protocol=TLS # SSL protocol to use. Requires a custom management.server.port.
management.server.ssl.trust-store= # Trust store that holds SSL certificates. Requires a custom management.server.port.
management.server.ssl.trust-store-password= # Password used to access the trust store. Requires a custom management.server.port.
management.server.ssl.trust-store-provider= # Provider for the trust store. Requires a custom management.server.port.
management.server.ssl.trust-store-type= # Type of the trust store. Requires a custom management.server.port.

# 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.
management.endpoints.jmx.static-names= # Additional static properties to append to all ObjectNames of MBeans representing Endpoints.
management.endpoints.jmx.unique-names=false # Whether to ensure that ObjectNames are modified in case of conflict.

# 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.
management.endpoints.web.base-path=/actuator # Base path for Web endpoints. Relative to server.servlet.context-path or management.server.servlet.cont
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 disa
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 s

# 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.

# 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 sanit

# 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. Ke

# 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 us
management.endpoint.health.show-details=never # When to show full health details.

# HEAP DUMP ENDPOINT (HeapDumpWebEndpoint)
```

```
# 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.

# 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

# 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=0 # Minimum disk space, in bytes, 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 defa
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.binders.files.enabled=true # Whether to enable files metrics.
management.metrics.binders.integration.enabled=true # Whether to enable Spring Integration metrics.
management.metrics.binders.jvm.enabled=true # Whether to enable JVM metrics.
management.metrics.binders.logback.enabled=true # Whether to enable Logback metrics.
management.metrics.binders.processor.enabled=true # Whether to enable processor metrics.
management.metrics.binders.uptime.enabled=true # Whether to enable uptime metrics.
```

```
management.metrics.distribution.percentiles-histogram.*= # Whether meter IDs starting-with the specified name should be publish percentile histograms
management.metrics.distribution.percentiles.*= # Specific computed non-aggregable percentiles to ship to the backend for meter IDs starting-with the
management.metrics.distribution.sla.*= # Specific SLA boundaries for meter IDs starting-with the specified name. The longest match wins, the key `all
management.metrics.enable.*= # Whether meter IDs starting-with the specified name should be enabled. The longest match wins, the key `all` can also b
management.metrics.export.atlas.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then m
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 currer
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 subscr
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 c
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 m
management.metrics.export.datadog.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, ther
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 me
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-rout
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 nu
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.influx.auto-create-db=true # Whether to create the Influx database if it does not exist before attempting to publish metric
management.metrics.export.influx.batch-size=10000 # Number of measurements per request to use for this backend. If more measurements are found, then
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-policy= # Retention policy to use (Influx writes to the DEFAULT retention policy if one is not specified).
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.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, the
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
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.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, the
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
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 enable
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 th
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, th
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 tho
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. Defa
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
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, metr
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 or WebFlux should be automatically timed.
management.metrics.web.server.requests-metric-name=http.server.requests # Name of the metric for received requests.


# ----------------------------------------
# DEVTOOLS PROPERTIES
# ----------------------------------------

# DEVTOOLS (DevToolsProperties)
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,**/*Tests.class,
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

# 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.
```

Appendix B. Configuration Metadata译:附录B配置元数据

Spring Boot jar包含元数据文件，提供所有支持的配置属性的详细信息。 这些文件旨在让IDE开发人员提供上下文帮助和“代码完成”，因为用户正在使用`application.properties`或`application.yml`文件。

大部分元数据文件在编译时自动生成，处理所有注释为`@ConfigurationProperties`项目。 但是，对于角落案件或更高级的使用案例， write part of the metadata manually是可能的。

B.1 Metadata Format译: 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文件内部。它们使用简单的JSON格式，其中按“组”或“属性”分类的项目以及分类在“提示”下的附加值提示，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$Hibernate",
  "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
  "sourceMethod": "getHibernate()"
 }
 ...
],"properties": [
 {
  "name": "server.port",
  "type": "java.lang.Integer",
  "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
 },
 {
  "name": "server.servlet.path",
  "type": "java.lang.String",
  "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties",
  "defaultValue": "/"
 },
 {
    "name": "spring.jpa.hibernate.ddl-auto",
    "type": "java.lang.String",
    "description": "DDL mode. This is actually a shortcut for the \"hibernate.hbm2ddl.auto\" property.",
    "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$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”是用户用给定值指定的配置项目。 例如， `server.port`和`server.servlet.path`可能在`application.properties`指定，如下所示：

```
server.port=9090
server.servlet.path=/home
```

“组”是较高级别的项目，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.servlet.path`属性是`server`组的一部分。

并不要求每个“特性”都有一个“组”。 一些属性可能存在于他们自己的权利中。

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`财产，工具可以使用提示提供了一些自动完成帮助`none`， `validate`， `update`， `create`，并`create-drop`值。

B.1.1 Group Attributes译：B.1.1组属性

包含在 `groups`数组中的JSON对象可以包含下表中显示的属性：

Name	Type	Purpose
`name`	串	组的全名。 该属性是强制性的。
`type`	串	组的数据类型的类名称。 例如，如果该组基于以`@ConfigurationProperties`注解的类，则该属性将包含该类的完全限定名称。 如果它基于`@Bean`方法，那将是该方法的返回类型。 如果该类型未知，则可以省略该属性。
`description`	串	可向用户显示的组的简短说明。 如果没有描述可用，则可以省略。 建议描述为简短段落，第一行提供简明摘要。 描述中的最后一行应以句点（`.`）`.`。
`sourceType`	串	贡献此组的源的类名称。 例如，如果该组基于`@Bean`方法注释了`@ConfigurationProperties`，则此属性将包含包含该方法的`@Configuration`类的完全限定名称。 如果源类型未知，则可以省略该属性。

Name	Type	Purpose
`sourceMethod`	串	贡献该组的方法的全名（包括括号和参数类型）（例如， `@ConfigurationProperties` 注解 `@Bean` 方法的名称）。 如果源方法未知，则可能会被忽略。

B.1.2 Property Attributes 译：B.1.2属性属性

包含在 `properties` 数组中的JSON对象可以包含下表中描述的属性：

Name	Type	Purpose
`name`	串	财产的全名。 名称以小写字母分隔的格式显示（例如， `server.servlet.path` ）。 该属性是强制性的。
`type`	串	该属性的数据类型（例如 `java.lang.String` ）的完整签名，也是完整的泛型类型（如 `java.util.Map<java.util.String,acme.MyEnum>` ）。 您可以使用此属性来指导用户可以输入的值的类型。 为了一致性，通过使用其包装对象来指定基元的类型（例如， `boolean` 变为 `java.lang.Boolean` ）。 请注意，该类可能是一个复杂的类型，它会在值绑定时从 `String` 转换而来。 如果类型未知，可能会被忽略。
`description`	串	可向用户显示的组的简短说明。 如果没有可用的说明，可能会被忽略。 建议描述为简短段落，第一行提供简明摘要。 说明中的最后一行应以句点（ `.` ） `.` 。
`sourceType`	串	贡献此属性的源的类名称。 例如，如果该属性来自使用 `@ConfigurationProperties` 注释的类，则此属性将包含该类的完全限定名称。 如果源类型未知，则可能会被忽略。
`defaultValue`	目的	如果未指定属性，则使用默认值。 如果属性的类型是一个数组，它可以是一个值的数组。 如果默认值是未知的，则可以省略。
`deprecation`	弃用	指定该属性是否被弃用。 如果该字段未被弃用或者该信息未知，则可以省略。 下表提供了有关 `deprecation` 属性的更多详细信息。

每个 `properties` 元素的 `deprecation` 属性中包含的JSON对象可以包含以下属性：

Name	Type	Purpose
`level`	串	弃用级别可以是 `warning` （默认值）或 `error` 。 当某个属性的弃用级别为 `warning` ，它应该仍然在环境中绑定。 但是，如果它具有 `error` 弃用级别，则该属性不再受管理且 `error` 。
`reason`	串	对房产被弃用的原因的简短描述。 如果没有理由可用，它可能被省略。 建议描述为简短段落，第一行提供简明摘要。 说明中的最后一行应以句点（ `.` ） `.` 。
`replacement`	串	*替换*此已弃用属性的属性的全名。 如果这个属性没有替换，它可能被省略。

在Spring Boot 1.3之前，可以使用一个 `deprecated` 布尔属性来代替 `deprecation` 元素。 这仍以不推荐的方式支持，不应再使用。 如果没有理由和替换可用，则应设置一个空的 `deprecation` 对象。

弃用也可以在代码中以声明方式指定，方法是将 `@DeprecatedConfigurationProperty` 注释添加到暴露不赞成使用的属性的getter中。 例如，假设 `app.acme.target` 属性很混乱，并且被重命名为 `app.acme.name` 。 以下示例显示如何处理这种情况：

```
@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);
 }
}
```

没有办法设置 `level` 。 总是假定为 `warning` ，因为代码仍处理该属性。

前面的代码确保已弃用的属性仍然有效（在幕后委派到 `name` 属性）。 一旦 `getTarget` 和 `setTarget` 方法可以从您的公共API中移除，元数据中的自动弃用提示也会消失。 如果您想保留提示，那么添加具有 `error` 弃用级别的手动元数据可确保用户仍被通知该属性。 这样做在提供 `replacement` 时特别有用。

B.1.3 Hint Attributes 译:4 B.1.3提示属性

包含在 `hints` 数组中的JSON对象可以包含下表中显示的属性：

Name	Type	Purpose

Name	Type	Purpose
`name`	串	此提示引用的财产的全名。 名称以小写字母分隔的形式（例如`server.servlet.path`）。 如果属性是指地图（例如`system.contexts`），提示无论是适用于图（的*键* `system.context.keys`）或*值*（`system.context.values`的地图）。 该属性是强制性的。
`values`	ValueHint []	由`ValueHint`对象定义的有效值列表（如下表所述）。 每个条目定义该值并可能有说明。
`providers`	ValueProvider []	由`ValueProvider`对象定义的提供者列表（在本文档稍后描述）。 每个条目定义提供者的名称及其参数（如果有的话）。

包含在每个`hint`元素的`values`属性中的JSON对象可以包含下表中描述的属性：

Name	Type	Purpose
`value`	目的	提示引用的元素的有效值。 如果属性的类型是一个数组，它也可以是一个值的数组。 该属性是强制性的。
`description`	串	可以向用户显示的值的简短说明。 如果没有可用的说明，可能会被忽略。 建议描述为简短段落，第一行提供简明摘要。 说明中的最后一行应以句点（`.`）`.`。

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：

Name	Type	Purpose
`name`	串	提供程序的名称，用于为提示所引用的元素提供其他内容帮助。
`parameters`	JSON对象	提供者支持的任何其他参数（查看提供者的文档以获取更多详细信息）。

B.1.4 Repeated Metadata Items译：B.1.4重复的元数据项目

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相同的前缀，每个类都有可能重叠的属性名称。 尽管多次出现在元数据中的相同名称不应该很常见，但元数据的使用者应该注意确保它们支持它。

B.2 Providing Manual Hints译：B.2提供了手册的提示

为了改善用户体验并进一步帮助用户配置给定属性，可以提供以下附加元数据：

- Describes the list of potential values for a property.
- Associates a provider, to attach a well defined semantic to a property, so that a tool can discover the list of potential values based on the project's context.

B.2.1 Value Hint译：B.2.1值提示

每个提示的`name`属性指的是财产的`name`。 在initial example shown earlier，我们提供了五个值`spring.jpa.hibernate.ddl-auto`属性：`none`，`validate`，`update`，`create`，并`create-drop`。 每个值也可以有一个描述。

如果您的财产属于`Map`类型，则可以为键和值提供提示（但不适用于地图本身）。 特殊的`.keys`和`.values`后缀必须分别指代键和值。

假定`sample.contexts`将magic `String`值映射为整数，如以下示例中所示：

```
@ConfigurationProperties("sample")
public class SampleProperties {

 private Map<String,Integer> contexts;
 // getters and setters
}
```

神奇值是（在这个例子中）是`sample1`和`sample2`。 为了为密钥提供其他内容帮助，可以将以下JSON添加到the manual metadata of the module：

```
{"hints": [
 {
  "name": "sample.contexts.keys",
  "values": [
   {
    "value": "sample1"
   },
   {
    "value": "sample2"
   }
  ]
 }
]}
```

我们建议您为这两个值使用`Enum`。 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 Value Providers译：4 B.2.2价值提供者

提供者是将语义附加到属性的有效方式。 在本节中，我们定义您可以用于自己提示的官方提供商。 但是，您最喜欢的IDE可能会实现其中的一部分或者其中任何一个。 此外，它最终可以提供自己的。

由于这是一项新功能，IDE供应商必须赶上它的工作原理。 采用时间自然有所不同。

下表总结了支持的提供商列表：

Name	描述
`any`	允许提供任何额外的价值。
`class-reference`	自动完成项目中可用的类。 通常受`target`参数指定的基类限制。
`handle-as`	处理该属性，就好像它是由强制`target`参数定义的类型所定义的`target`。
`logger-name`	自动完成有效的记录器名称。 通常，当前项目中可用的包名和类名可以自动完成。
`spring-bean-reference`	自动完成当前项目中的可用bean名称。 通常受`target`参数指定的基类限制。
`spring-profile-name`	自动完成项目中可用的Spring配置文件名称。

只有一个提供程序可以对某个给定的属性处于活动状态，但是您可以指定多个提供程序，只要它们能够*以某种方式*管理该属性。 确保首先放置功能最强大的提供者，因为IDE必须使用它可以处理的JSON部分中的第一个提供者。 如果没有支持给定财产的提供者，则也不提供特别的内容帮助。

Any 译:任何

特殊的**任何**提供者值允许提供**任何**附加值。 如果支持，则应应用基于属性类型的常规值验证。

如果您有一个值列表，并且任何额外的值仍应被视为有效，通常会使用此提供程序。

下面的例子提供`on`和`off`为自动完成值`system.state`：

```
{"hints": [
 {
  "name": "system.state",
  "values": [
   {
    "value": "on"
   },
   {
    "value": "off"
   }
  ],
  "providers": [
   {
    "name": "any"
   }
  ]
 }
]}
```

请注意，在前面的例子中，任何其他值也是允许的。

Class Reference 译:类参考

类参考提供程序自动完成项目中可用的类。 该提供程序支持以下参数：

Parameter	Type	Default value	描述
`target`	`String`（`Class`）	*没有*	应该可分配给所选值的类的全限定名称。 通常用于过滤非候选类。 请注意，这些信息可以由类型本身通过暴露具有适当上限的类来提供。
`concrete`	`boolean`	真正	指定是否只有具体的课程才被视为有效的候选人。

以下元数据片段对应于定义要使用的`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"
    }
   }
  ]
 }
]}
```

Handle As 译:处理为

handle-as提供程序可让您将属性的类型替换为更高级别的类型。 这通常发生在属性具有`java.lang.String`类型时，因为您不希望您的配置类依赖可能不在类路径中的类。 该提供程序支持以下参数：

Parameter	Type	Default value	描述
`target`	`String`（`Class`）	*没有*	要为属性考虑的类型的完全限定名称。 该参数是强制性的。

可以使用以下类型：

- Any `java.lang.Enum`: Lists the possible values for the property. (We recommend defining the property with the `Enum` type, as no further hint should be required for the IDE to auto-complete the values.)
- `java.nio.charset.Charset`: Supports auto-completion of charset/encoding values (such as `UTF-8`)
- `java.util.Locale`: auto-completion of locales (such as `en_US`)
- `org.springframework.util.MimeType`: Supports auto-completion of content type values (such as `text/plain`)
- `org.springframework.core.io.Resource`: Supports auto-completion of Spring's Resource abstraction to refer to a file on the filesystem or on the classpath. (such as `classpath:/sample.properties`)

如果可以提供多个值，请使用 `Collection` 或 *数组* 类型向IDE讲授它。

以下元数据片段对应于定义要使用的更改日志路径的标准`spring.liquibase.change-log`属性。 它实际上在内部用作`org.springframework.core.io.Resource`但不能像这样公开，因为我们需要保留原始字符串值以将其传递给Liquibase API。

```
{"hints": [
 {
  "name": "spring.liquibase.change-log",
  "providers": [
   {
    "name": "handle-as",
    "parameters": {
     "target": "org.springframework.core.io.Resource"
    }
   }
  ]
 }
]}
```

Logger Name 译:记录器名称

记录器名称提供程序会自动完成有效的记录器名称。 通常，当前项目中可用的包名和类名可以自动完成。 特定的框架可能还有额外的魔术记录器名称，这些名称也可以被支持。

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称。

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。

```
{"hints": [
 {
  "name": "logging.level.keys",
  "values": [
   {
    "value": "root",
    "description": "Root logger used to assign the default logging level."
   }
  ],
  "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 Reference 译: Spring Bean参考

spring-bean-reference提供程序自动完成在当前项目的配置中定义的bean。 该提供程序支持以下参数：

Parameter	Type	Default value	描述

Parameter	Type	Default value	描述
`target`	`String` （ `Class` ）	*没有*	应该可分配给候选人的bean类的完全限定名称。 通常用于过滤掉非候选bean。

以下元数据片段对应于定义要使用的 `MBeanServer` bean名称的标准 `spring.jmx.server` 属性：

```
{"hints": [
 {
  "name": "spring.jmx.server",
  "providers": [
   {
    "name": "spring-bean-reference",
    "parameters": {
     "target": "javax.management.MBeanServer"
    }
   }
  ]
 }
]}
```

活页夹不知道元数据。 如果你提供了这个提示，你仍然需要将bean的名字转换成 `ApplicationContext` 使用的实际Bean引用。

Spring Profile Name 译:春季档案名称

spring-profile-name提供程序自动完成在当前项目的配置中定义的Spring配置文件。

以下元数据片段对应于定义要启用的Spring配置文件名称的标准 `spring.profiles.active` 属性：

```
{"hints": [
 {
  "name": "spring.profiles.active",
  "providers": [
   {
    "name": "spring-profile-name"
   }
  ]
 }
]}
```

B.3 Generating Your Own Metadata by Using the Annotation Processor 译: B.3通过使用注解处理器生成自己的元数据

您可以轻松地生成带注释的项目自己的配置元数据文件 `@ConfigurationProperties` 使用 `spring-boot-configuration-processor` 罐子。 该jar包含一个Java注释处理器，在编译项目时调用它。 要使用处理器，请包含对 `spring-boot-configuration-processor` 的依赖关系。

使用Maven时，应将依赖项声明为可选项，如以下示例所示：

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-configuration-processor</artifactId>
 <optional>true</optional>
</dependency>
```

在Gradle 4.5及更早版本中，依赖关系应该在 `compileOnly` 配置中声明，如以下示例所示：

```
dependencies {
 compileOnly "org.springframework.boot:spring-boot-configuration-processor"
}
```

在Gradle 4.6和更高版本中，应该在 `annotationProcessor` 配置中声明依赖关系，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 `compileJava` 任务应配置为取决于 `processResources` 任务，如以下示例所示：

```
compileJava.dependsOn(processResources)
```

这种依赖性确保了在编译过程中注释处理器运行时附加元数据可用。

处理器选取了 `@ConfigurationProperties` 注解的类和方法。 配置类中字段值的Javadoc用于填充 `description` 属性。

您应该只使用 `@ConfigurationProperties` 字段Javadoc中的简单文本，因为它们在添加到JSON之前未经过处理。

属性是通过标准getter和setter的存在发现的，对集合类型有特殊处理（即使只有getter也可以检测到）。 注解处理器还支持使用的 `@Data` ， `@Getter` 和 `@Setter` Lombok的注释。

> 如果您在项目中使用AspectJ，则需要确保注释处理器只运行一次。 有几种方法可以做到这一点。 借助Maven，您可以明确地配置`maven-apt-plugin`并且只在那里将依赖项添加到注释处理器。 您还可以让AspectJ插件运行所有处理并禁用`maven-compiler-plugin`配置中的注释处理，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 Nested Properties 译:⁋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

 }

}
```

前面的例子中产生用于元数据信息`server.name`， `server.host.ip`，和`server.host.port`性质。 您可以在字段上使用`@NestedConfigurationProperty`批注来指示应将常规（非内部）类视为嵌套。

> 这对集合和地图没有影响，因为这些类型是自动识别的，并且为它们中的每一个生成单个元数据属性。

B.3.2 Adding Additional Metadata 译:§B.3.2添加附加元数据

Spring Boot的配置文件处理非常灵活，并且通常情况下可能存在不绑定到`@ConfigurationProperties` bean的`@ConfigurationProperties` 。 您可能还需要调整现有密钥的某些属性。 为了支持这种情况并让您提供自定义“提示”，注释处理器会自动将来自`META-INF/additional-spring-configuration-metadata.json`项目合并到主元数据文件中。

如果您引用自动检测到的属性，则说明，默认值和弃用信息将被覆盖（如果指定）。 如果手动属性声明在当前模块中未标识，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` 。 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。

Appendix C. Auto-configuration classes 译:附录C.自动配置类

这里是Spring Boot提供的所有自动配置类的列表，包含文档和源代码的链接。 请记住在应用程序中查看条件报告，了解哪些功能处于打开状态。 （为此，请使用`--debug`或`-Ddebug`启动应用程序，或者在执行器应用程序中使用`conditions`端点）。

C.1 From the “spring-boot-autoconfigure” module 译: C.1从“spring-boot-autoconfigure”模块

以下自动配置类来自`spring-boot-autoconfigure`模块：

Configuration Class	Links
`ActiveMQAutoConfiguration`	javadoc
`AopAutoConfiguration`	javadoc
`ArtemisAutoConfiguration`	javadoc
`BatchAutoConfiguration`	javadoc
`CacheAutoConfiguration`	javadoc
`CassandraAutoConfiguration`	javadoc
`CassandraDataAutoConfiguration`	javadoc
`CassandraReactiveDataAutoConfiguration`	javadoc
`CassandraReactiveRepositoriesAutoConfiguration`	javadoc
`CassandraRepositoriesAutoConfiguration`	javadoc

Configuration Class	Links
`CloudAutoConfiguration`	javadoc
`CodecsAutoConfiguration`	javadoc
`ConfigurationPropertiesAutoConfiguration`	javadoc
`CouchbaseAutoConfiguration`	javadoc
`CouchbaseDataAutoConfiguration`	javadoc
`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
`HypermediaAutoConfiguration`	javadoc
`InfluxDbAutoConfiguration`	javadoc
`IntegrationAutoConfiguration`	javadoc
`JacksonAutoConfiguration`	javadoc
`JdbcTemplateAutoConfiguration`	javadoc
`JerseyAutoConfiguration`	javadoc
`JestAutoConfiguration`	javadoc
`JmsAutoConfiguration`	javadoc

Configuration Class	Links
`JmxAutoConfiguration`	javadoc
`JndiConnectionFactoryAutoConfiguration`	javadoc
`JndiDataSourceAutoConfiguration`	javadoc
`JooqAutoConfiguration`	javadoc
`JpaRepositoriesAutoConfiguration`	javadoc
`JsonbAutoConfiguration`	javadoc
`JtaAutoConfiguration`	javadoc
`KafkaAutoConfiguration`	javadoc
`LdapAutoConfiguration`	javadoc
`LdapDataAutoConfiguration`	javadoc
`LdapRepositoriesAutoConfiguration`	javadoc
`LiquibaseAutoConfiguration`	javadoc
`MailSenderAutoConfiguration`	javadoc
`MailSenderValidatorAutoConfiguration`	javadoc
`MessageSourceAutoConfiguration`	javadoc
`MongoAutoConfiguration`	javadoc
`MongoDataAutoConfiguration`	javadoc
`MongoReactiveAutoConfiguration`	javadoc
`MongoReactiveDataAutoConfiguration`	javadoc
`MongoReactiveRepositoriesAutoConfiguration`	javadoc
`MongoRepositoriesAutoConfiguration`	javadoc
`MultipartAutoConfiguration`	javadoc
`MustacheAutoConfiguration`	javadoc
`Neo4jDataAutoConfiguration`	javadoc
`Neo4jRepositoriesAutoConfiguration`	javadoc
`OAuth2ClientAutoConfiguration`	javadoc
`PersistenceExceptionTranslationAutoConfiguration`	javadoc
`ProjectInfoAutoConfiguration`	javadoc
`PropertyPlaceholderAutoConfiguration`	javadoc
`QuartzAutoConfiguration`	javadoc
`RabbitAutoConfiguration`	javadoc
`ReactiveSecurityAutoConfiguration`	javadoc
`ReactiveUserDetailsServiceAutoConfiguration`	javadoc
`ReactiveWebServerFactoryAutoConfiguration`	javadoc
`ReactorCoreAutoConfiguration`	javadoc
`RedisAutoConfiguration`	javadoc
`RedisReactiveAutoConfiguration`	javadoc
`RedisRepositoriesAutoConfiguration`	javadoc

Configuration Class	Links
`RepositoryRestMvcAutoConfiguration`	javadoc
`RestTemplateAutoConfiguration`	javadoc
`SecurityAutoConfiguration`	javadoc
`SecurityFilterAutoConfiguration`	javadoc
`SendGridAutoConfiguration`	javadoc
`ServletWebServerFactoryAutoConfiguration`	javadoc
`SessionAutoConfiguration`	javadoc
`SolrAutoConfiguration`	javadoc
`SolrRepositoriesAutoConfiguration`	javadoc
`SpringApplicationAdminJmxAutoConfiguration`	javadoc
`SpringDataWebAutoConfiguration`	javadoc
`ThymeleafAutoConfiguration`	javadoc
`TransactionAutoConfiguration`	javadoc
`UserDetailsServiceAutoConfiguration`	javadoc
`ValidationAutoConfiguration`	javadoc
`WebClientAutoConfiguration`	javadoc
`WebFluxAutoConfiguration`	javadoc
`WebMvcAutoConfiguration`	javadoc
`WebServicesAutoConfiguration`	javadoc
`WebSocketMessagingAutoConfiguration`	javadoc
`WebSocketReactiveAutoConfiguration`	javadoc
`WebSocketServletAutoConfiguration`	javadoc
`XADataSourceAutoConfiguration`	javadoc

C.2 From the “spring-boot-actuator-autoconfigure” module译：C.2从 A€œspring启动制动器-autoconfigureâ€模块

以下自动配置类来自 `spring-boot-actuator-autoconfigure` 模块：

Configuration Class	Links
`AtlasMetricsExportAutoConfiguration`	javadoc
`AuditAutoConfiguration`	javadoc
`AuditEventsEndpointAutoConfiguration`	javadoc
`BeansEndpointAutoConfiguration`	javadoc
`CacheMetricsAutoConfiguration`	javadoc
`CassandraHealthIndicatorAutoConfiguration`	javadoc
`CloudFoundryActuatorAutoConfiguration`	javadoc
`CompositeMeterRegistryAutoConfiguration`	javadoc
`ConditionsReportEndpointAutoConfiguration`	javadoc
`ConfigurationPropertiesReportEndpointAutoConfiguration`	javadoc
`CouchbaseHealthIndicatorAutoConfiguration`	javadoc

Configuration Class	Links
`DataSourceHealthIndicatorAutoConfiguration`	javadoc
`DataSourcePoolMetricsAutoConfiguration`	javadoc
`DatadogMetricsExportAutoConfiguration`	javadoc
`DiskSpaceHealthIndicatorAutoConfiguration`	javadoc
`ElasticsearchHealthIndicatorAutoConfiguration`	javadoc
`EndpointAutoConfiguration`	javadoc
`EnvironmentEndpointAutoConfiguration`	javadoc
`FlywayEndpointAutoConfiguration`	javadoc
`GangliaMetricsExportAutoConfiguration`	javadoc
`GraphiteMetricsExportAutoConfiguration`	javadoc
`HealthEndpointAutoConfiguration`	javadoc
`HealthIndicatorAutoConfiguration`	javadoc
`HeapDumpWebEndpointAutoConfiguration`	javadoc
`HttpTraceAutoConfiguration`	javadoc
`HttpTraceEndpointAutoConfiguration`	javadoc
`InfluxDbHealthIndicatorAutoConfiguration`	javadoc
`InfluxMetricsExportAutoConfiguration`	javadoc
`InfoContributorAutoConfiguration`	javadoc
`InfoEndpointAutoConfiguration`	javadoc
`JmsHealthIndicatorAutoConfiguration`	javadoc
`JmxEndpointAutoConfiguration`	javadoc
`JmxMetricsExportAutoConfiguration`	javadoc
`JolokiaEndpointAutoConfiguration`	javadoc
`LdapHealthIndicatorAutoConfiguration`	javadoc
`LiquibaseEndpointAutoConfiguration`	javadoc
`LogFileWebEndpointAutoConfiguration`	javadoc
`LoggersEndpointAutoConfiguration`	javadoc
`MailHealthIndicatorAutoConfiguration`	javadoc
`ManagementContextAutoConfiguration`	javadoc
`MappingsEndpointAutoConfiguration`	javadoc
`MetricsAutoConfiguration`	javadoc
`MetricsEndpointAutoConfiguration`	javadoc
`MongoHealthIndicatorAutoConfiguration`	javadoc
`Neo4jHealthIndicatorAutoConfiguration`	javadoc
`NewRelicMetricsExportAutoConfiguration`	javadoc
`PrometheusMetricsExportAutoConfiguration`	javadoc
`RabbitHealthIndicatorAutoConfiguration`	javadoc
`RabbitMetricsAutoConfiguration`	javadoc

Configuration Class	Links
`ReactiveCloudFoundryActuatorAutoConfiguration`	javadoc
`ReactiveManagementContextAutoConfiguration`	javadoc
`RedisHealthIndicatorAutoConfiguration`	javadoc
`RestTemplateMetricsAutoConfiguration`	javadoc
`ScheduledTasksEndpointAutoConfiguration`	javadoc
`ServletManagementContextAutoConfiguration`	javadoc
`SessionsEndpointAutoConfiguration`	javadoc
`ShutdownEndpointAutoConfiguration`	javadoc
`SignalFxMetricsExportAutoConfiguration`	javadoc
`SimpleMetricsExportAutoConfiguration`	javadoc
`SolrHealthIndicatorAutoConfiguration`	javadoc
`StatsdMetricsExportAutoConfiguration`	javadoc
`ThreadDumpEndpointAutoConfiguration`	javadoc
`TomcatMetricsAutoConfiguration`	javadoc
`WavefrontMetricsExportAutoConfiguration`	javadoc
`WebEndpointAutoConfiguration`	javadoc
`WebFluxMetricsAutoConfiguration`	javadoc
`WebMvcMetricsAutoConfiguration`	javadoc

Appendix D. Test auto-configuration annotations译:附录D.测试自动配置注释

下表列出了各种 `@…Test` 注释， `@…Test` 注释可用于测试应用程序的切片以及默认导入的自动配置：

Test slice	Imported auto-configuration

`@DataJpaTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManagerAutoConfiguration`

`@DataLdapTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.ldap.LdapDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration`
	`org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration`

`@DataMongoTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration`
	`org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration`
	`org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration`

`@DataNeo4jTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`

Test slice	Imported auto-configuration

`@DataRedisTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration`

`@JdbcTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration`

`@JooqTest`	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`

`@JsonTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.json.JsonTestersAutoConfiguration`

`@RestClientTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.client.MockRestServiceServerAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.client.WebClientRestTemplateAutoConfiguration`

`@WebFluxTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.reactive.WebTestClientAutoConfiguration`

`@WebMvcTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration`
	`org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration`
	`org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.servlet.MockMvcAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.servlet.MockMvcSecurityAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.servlet.MockMvcWebClientAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.servlet.MockMvcWebDriverAutoConfiguration`

Appendix E. The Executable Jar Format译:附录E可执行Jar格式

`spring-boot-loader`模块让Spring Boot支持可执行jar和war文件。 如果您使用Maven插件或Gradle插件，可自动生成可执行文件，并且您通常不需要知道它们的工作方式。

如果您需要从不同的构建系统创建可执行文件，或者您只是对底层技术感兴趣，本节提供一些背景知识。

E.1 Nested JARs译:E.1嵌套JAR

Java没有提供任何标准的方法来加载嵌套的jar文件（也就是本身包含在jar中的jar文件）。 如果您需要分发可以从命令行运行而不打开的自包含应用程序，则这可能会有问题。

为了解决这个问题，许多开发人员使用“阴影”罐子。 阴影的jar将所有类的所有类从所有jar包装到一个单独的“jar”中。 带阴影的瓶子的问题是，很难看到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（但是具有不同的内容），则它也可能是有问题的。 Spring Boot采用了不同的方法，可以让您直接嵌入罐子。

E.1.1 The Executable Jar File Structure译：E.1.1可执行jar文件结构

Spring Boot Loader兼容的jar文件应该按以下方式构建：

```
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
```

应用程序类应放置在嵌套的`BOOT-INF/classes`目录中。 依赖关系应放置在嵌套的`BOOT-INF/lib`目录中。

E.1.2 The Executable War File Structure译：E.1.2可执行WAR文件结构

Spring Boot Loader兼容的war文件应该按以下方式构建：

```
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
```

依赖关系应放在嵌套的`WEB-INF/lib`目录中。 运行嵌入式时所需的任何依赖关系，但在部署到传统Web容器时不需要，应放置在`WEB-INF/lib-provided`。

E.2 Spring Boot's "JarFile" Class译：E.2你 Boot â€™的 â€œJarFileâ€ ç±»

用于支持加载嵌套罐的核心类是`org.springframework.boot.loader.jar.JarFile`。 它允许您从标准jar文件或嵌套的子jar数据中加载jar内容。 第一次加载时，每个`JarEntry`的位置都映射到外部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
```

前面的例子示出了如何`A.class`可参见`/BOOT-INF/classes`在`myapp.jar`在位置`0063`。 来自嵌套罐的`B.class`实际上可以在`myapp.jar`的位置`3452`处`3452`，并且`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。 我们不需要解压缩存档，我们也不需要将所有条目数据读入内存。

E.2.1 Compatibility with the Standard Java "JarFile"译：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`从扩展`java.util.jar.JarFile`，应该作为一个简易替换工作。 `getURL()`方法返回一个`URL`，打开一个与`java.net.JarURLConnection`兼容的连接，并且可以与Java的`URLClassLoader`。

E.3 Launching Executable Jars译：E.3启动可执行罐子

`org.springframework.boot.loader.Launcher`类是一个特殊的引导类，用作可执行jar的主入口点。 它是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。

有三个发射子类（`JarLauncher`， `WarLauncher`，并`PropertiesLauncher`）。 它们的目的是从嵌套的jar文件或目录中的war文件（与显式地位于类路径中的那些文件相对）加载资源（`.class`文件等）。 在`JarLauncher`和`WarLauncher`的情况下，嵌套路径是固定的。
`JarLauncher`的`BOOT-INF/lib/`和`WarLauncher`在`WEB-INF/lib/`和`WEB-INF/lib-provided/` `WarLauncher`查找。 如果您想要更多，可以在这些位置添加额外的罐子。 `PropertiesLauncher`默认在应用程序存档中查找`BOOT-INF/lib/`，但可以通过在`loader.properties`（这是存档中的目录，存档或目录的逗号分隔列表）中设置名为`LOADER_PATH`或`loader.path`的环境变量来添加其他位置。

E.3.1 Launcher Manifest译：E3.1启动清单

您需要指定一个合适的`Launcher`为`Main-Class`的属性`META-INF/MANIFEST.MF`。 应该在`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
```

对于战争档案，它将如下所示：

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

您不需要在清单文件中指定`Class-Path`条目。 类路径是从嵌套的jar中推导出来的。

E.3.2 Exploded Archives译：E3.2爆炸档案

某些PaaS实现可能会选择在运行之前解压缩归档文件。 例如，Cloud Foundry以这种方式运作。 您可以通过启动适当的启动程序运行解压缩的归档文件，如下所示：

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```

E.4 `PropertiesLauncher` Features译：E4 `PropertiesLauncher`功能

`PropertiesLauncher`有一些可以使用外部属性（系统属性，环境变量，清单条目或`loader.properties`）启用的特殊功能。 下表描述了这些属性：

Key	Purpose
`loader.path`	逗号分隔的Classpath，例如`lib,${HOME}/app/lib`。 早期条目优先，如`javac`命令行上的常规`-classpath`。
`loader.home`	用于解析`loader.path`相对路径。 例如，给定`loader.path=lib`，则`${loader.home}/lib`是一个类路径位置（以及该目录中的所有jar文件）。 该属性也用于查找`loader.properties`文件，如以下示例`/opt/app`它默认为`${user.dir}`。
`loader.args`	主方法的默认参数（空格分隔）。
`loader.main`	要启动的主要类的名称（例如，`com.app.Application`）。
`loader.config.name`	属性文件的名称（例如`launcher`）默认为`loader`。
`loader.config.location`	属性文件的路径（例如，`classpath:loader.properties`）。 它默认为`loader.properties`。
`loader.system`	用于指示所有属性应添加到系统属性的布尔标志它默认为`false`。

当指定为环境变量或清单条目时，应使用以下名称：

Key	Manifest entry	Environment variable
`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`

构建胖罐时，构建插件会自动将`Main-Class`属性移动到`Start-Class`。 如果使用该属性，请使用`Main-Class`属性指定要启动的类的名称，并`Start-Class`。

以下规则适用于使用`PropertiesLauncher`：

- `loader.properties` is searched for in `loader.home`, then in the root of the classpath, and then in `classpath:/BOOT-INF/classes`. The first location where a file with that name exists is used.
- `loader.home` is the directory location of an additional properties file (overriding the default) only when `loader.config.location` is not specified.
- `loader.path` can contain directories (which are scanned recursively for jar and zip files), archive paths, a directory within an archive that is scanned for jar files (for example, `dependencies.jar!/lib`), or wildcard patterns (for the default JVM behavior). Archive paths can be relative to `loader.home` or anywhere in the file system with a `jar:file:` prefix.
- `loader.path` (if empty) defaults to `BOOT-INF/lib` (meaning a local directory or a nested one if running from an archive). Because of this, `PropertiesLauncher` behaves the same as `JarLauncher` when no additional configuration is provided.
- `loader.path` can not be used to configure the location of `loader.properties` (the classpath used to search for the latter is the JVM classpath when `PropertiesLauncher` is launched).
- Placeholder replacement is done from System and environment variables plus the properties file itself on all values before use.
- The search order for properties (where it makes sense to look in more than one place) is environment variables, system properties, `loader.properties`, the exploded archive manifest, and the archive manifest.

E.5 Executable Jar Restrictions译:E5可执行瓶子限制

在使用Spring Boot Loader打包的应用程序时，您需要考虑以下限制：

- Zip entry compression: The `ZipEntry` for a nested jar must be saved by using the `ZipEntry.STORED` method. This is required so that we can seek directly to individual content within the nested jar. The content of the nested jar file itself can still be compressed, as can any other entries in the outer jar.
- System classLoader: Launched applications should use `Thread.getContextClassLoader()` when loading classes (most libraries and frameworks do so by default). Trying to load nested jar classes with `ClassLoader.getSystemClassLoader()` fails. `java.util.Logging` always uses the system classloader. For this reason, you should consider a different logging implementation.

E.6 Alternative Single Jar Solutions译:9 E6另一种单罐解决方案

如果上述限制意味着您不能使用Spring Boot Loader，请考虑以下选择：

- Maven Shade Plugin
- JarClassLoader
- OneJar

Appendix F. Dependency versions译:附录F.相关版本

下表提供了Spring Boot在其CLI（命令行界面），Maven依赖项管理和Gradle插件中提供的所有依赖项版本的详细信息。 当您声明对这些工件之一的依赖关系而未声明版本时，将使用表中列出的版本。

Group ID	Artifact ID	Version
`antlr`	`antlr`	2.7.7
`ch.qos.logback`	`logback-access`	1.2.3
`ch.qos.logback`	`logback-classic`	1.2.3
`ch.qos.logback`	`logback-core`	1.2.3
`com.atomikos`	`transactions-jdbc`	4.0.6
`com.atomikos`	`transactions-jms`	4.0.6
`com.atomikos`	`transactions-jta`	4.0.6
`com.couchbase.client`	`couchbase-spring-cache`	2.1.0
`com.couchbase.client`	`java-client`	2.5.9
`com.datastax.cassandra`	`cassandra-driver-core`	3.4.0
`com.datastax.cassandra`	`cassandra-driver-mapping`	3.4.0
`com.fasterxml`	`classmate`	1.3.4
`com.fasterxml.jackson.core`	`jackson-annotations`	2.9.0
`com.fasterxml.jackson.core`	`jackson-core`	2.9.6
`com.fasterxml.jackson.core`	`jackson-databind`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-avro`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-cbor`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-csv`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-ion`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-properties`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-protobuf`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-smile`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-xml`	2.9.6
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-yaml`	2.9.6
`com.fasterxml.jackson.datatype`	`jackson-datatype-guava`	2.9.6
`com.fasterxml.jackson.datatype`	`jackson-datatype-hibernate3`	2.9.6

Group ID	Artifact ID	Version
com.fasterxml.jackson.datatype	jackson-datatype-hibernate4	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-hibernate5	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-hppc	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-jaxrs	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-jdk8	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-joda	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-json-org	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-jsr310	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-jsr353	2.9.6
com.fasterxml.jackson.datatype	jackson-datatype-pcollections	2.9.6
com.fasterxml.jackson.jaxrs	jackson-jaxrs-base	2.9.6
com.fasterxml.jackson.jaxrs	jackson-jaxrs-cbor-provider	2.9.6
com.fasterxml.jackson.jaxrs	jackson-jaxrs-json-provider	2.9.6
com.fasterxml.jackson.jaxrs	jackson-jaxrs-smile-provider	2.9.6
com.fasterxml.jackson.jaxrs	jackson-jaxrs-xml-provider	2.9.6
com.fasterxml.jackson.jaxrs	jackson-jaxrs-yaml-provider	2.9.6
com.fasterxml.jackson.jr	jackson-jr-all	2.9.6
com.fasterxml.jackson.jr	jackson-jr-objects	2.9.6
com.fasterxml.jackson.jr	jackson-jr-retrofit2	2.9.6
com.fasterxml.jackson.jr	jackson-jr-stree	2.9.6
com.fasterxml.jackson.module	jackson-module-afterburner	2.9.6
com.fasterxml.jackson.module	jackson-module-guice	2.9.6
com.fasterxml.jackson.module	jackson-module-jaxb-annotations	2.9.6
com.fasterxml.jackson.module	jackson-module-jsonSchema	2.9.6
com.fasterxml.jackson.module	jackson-module-kotlin	2.9.6
com.fasterxml.jackson.module	jackson-module-mrbean	2.9.6
com.fasterxml.jackson.module	jackson-module-osgi	2.9.6
com.fasterxml.jackson.module	jackson-module-parameter-names	2.9.6
com.fasterxml.jackson.module	jackson-module-paranamer	2.9.6
com.fasterxml.jackson.module	jackson-module-scala_2.10	2.9.6
com.fasterxml.jackson.module	jackson-module-scala_2.11	2.9.6
com.fasterxml.jackson.module	jackson-module-scala_2.12	2.9.6
com.github.ben-manes.caffeine	caffeine	2.6.2
com.github.mxab.thymeleaf.extras	thymeleaf-extras-data-attribute	2.0.1
com.google.appengine	appengine-api-1.0-sdk	64年9月1日
com.google.code.gson	gson	2.8.5
com.googlecode.json-simple	json-simple	1.1.1
com.h2database	h2	1.4.197

Group ID	Artifact ID	Version
com.hazelcast	hazelcast	3.9.4
com.hazelcast	hazelcast-client	3.9.4
com.hazelcast	hazelcast-hibernate52	1.2.3
com.hazelcast	hazelcast-spring	3.9.4
com.jayway.jsonpath	json-path	2.4.0
com.jayway.jsonpath	json-path-assert	2.4.0
com.microsoft.sqlserver	mssql-jdbc	6.2.2.jre8
com.querydsl	querydsl-apt	4.1.4
com.querydsl	querydsl-collections	4.1.4
com.querydsl	querydsl-core	4.1.4
com.querydsl	querydsl-jpa	4.1.4
com.querydsl	querydsl-mongodb	4.1.4
com.rabbitmq	amqp-client	5.1.2
com.samskivert	jmustache	1.14
com.sendgrid	sendgrid-java	4.1.2
com.sun.mail	javax.mail	1.6.1
com.timgroup	java-statsd-client	3.1.0
com.unboundid	unboundid-ldapsdk	4.0.6
com.zaxxer	HikariCP	2.7.9
commons-codec	commons-codec	1.11
commons-pool	commons-pool	1.6
de.flapdoodle.embed	de.flapdoodle.embed.mongo	2.0.3
dom4j	dom4j	1.6.1
io.dropwizard.metrics	metrics-annotation	3.2.6
io.dropwizard.metrics	metrics-core	3.2.6
io.dropwizard.metrics	metrics-ehcache	3.2.6
io.dropwizard.metrics	metrics-ganglia	3.2.6
io.dropwizard.metrics	metrics-graphite	3.2.6
io.dropwizard.metrics	metrics-healthchecks	3.2.6
io.dropwizard.metrics	metrics-httpasyncclient	3.2.6
io.dropwizard.metrics	metrics-jdbi	3.2.6
io.dropwizard.metrics	metrics-jersey	3.2.6
io.dropwizard.metrics	metrics-jersey2	3.2.6
io.dropwizard.metrics	metrics-jetty8	3.2.6
io.dropwizard.metrics	metrics-jetty9	3.2.6
io.dropwizard.metrics	metrics-jetty9-legacy	3.2.6
io.dropwizard.metrics	metrics-json	3.2.6

Group ID	Artifact ID	Version
io.dropwizard.metrics	metrics-jvm	3.2.6
io.dropwizard.metrics	metrics-log4j	3.2.6
io.dropwizard.metrics	metrics-log4j2	3.2.6
io.dropwizard.metrics	metrics-logback	3.2.6
io.dropwizard.metrics	metrics-servlet	3.2.6
io.dropwizard.metrics	metrics-servlets	3.2.6
io.lettuce	lettuce-core	5.0.4.RELEASE
io.micrometer	micrometer-core	1.0.5
io.micrometer	micrometer-registry-atlas	1.0.5
io.micrometer	micrometer-registry-datadog	1.0.5
io.micrometer	micrometer-registry-ganglia	1.0.5
io.micrometer	micrometer-registry-graphite	1.0.5
io.micrometer	micrometer-registry-influx	1.0.5
io.micrometer	micrometer-registry-jmx	1.0.5
io.micrometer	micrometer-registry-new-relic	1.0.5
io.micrometer	micrometer-registry-prometheus	1.0.5
io.micrometer	micrometer-registry-signalfx	1.0.5
io.micrometer	micrometer-registry-statsd	1.0.5
io.micrometer	micrometer-registry-wavefront	1.0.5
io.netty	netty-all	4.1.25.Final
io.netty	netty-buffer	4.1.25.Final
io.netty	netty-codec	4.1.25.Final
io.netty	netty-codec-dns	4.1.25.Final
io.netty	netty-codec-haproxy	4.1.25.Final
io.netty	netty-codec-http	4.1.25.Final
io.netty	netty-codec-http2	4.1.25.Final
io.netty	netty-codec-memcache	4.1.25.Final
io.netty	netty-codec-mqtt	4.1.25.Final
io.netty	netty-codec-redis	4.1.25.Final
io.netty	netty-codec-smtp	4.1.25.Final
io.netty	netty-codec-socks	4.1.25.Final
io.netty	netty-codec-stomp	4.1.25.Final
io.netty	netty-codec-xml	4.1.25.Final
io.netty	netty-common	4.1.25.Final
io.netty	netty-dev-tools	4.1.25.Final
io.netty	netty-example	4.1.25.Final
io.netty	netty-handler	4.1.25.Final
io.netty	netty-handler-proxy	4.1.25.Final

Group ID	Artifact ID	Version
io.netty	netty-resolver	4.1.25.Final
io.netty	netty-resolver-dns	4.1.25.Final
io.netty	netty-transport	4.1.25.Final
io.netty	netty-transport-native-epoll	4.1.25.Final
io.netty	netty-transport-native-kqueue	4.1.25.Final
io.netty	netty-transport-native-unix-common	4.1.25.Final
io.netty	netty-transport-rxtx	4.1.25.Final
io.netty	netty-transport-sctp	4.1.25.Final
io.netty	netty-transport-udt	4.1.25.Final
io.projectreactor	reactor-core	3.1.8.RELEASE
io.projectreactor	reactor-test	3.1.8.RELEASE
io.projectreactor.addons	reactor-adapter	3.1.6.RELEASE
io.projectreactor.addons	reactor-extra	3.1.6.RELEASE
io.projectreactor.addons	reactor-logback	3.1.6.RELEASE
io.projectreactor.ipc	reactor-netty	0.7.8.RELEASE
io.projectreactor.kafka	reactor-kafka	1.0.0.RELEASE
io.reactivex	rxjava	1.3.8
io.reactivex	rxjava-reactive-streams	1.2.1
io.reactivex.rxjava2	rxjava	2.1.14
io.rest-assured	json-path	3.0.7
io.rest-assured	json-schema-validator	3.0.7
io.rest-assured	rest-assured	3.0.7
io.rest-assured	scala-support	3.0.7
io.rest-assured	spring-mock-mvc	3.0.7
io.rest-assured	xml-path	3.0.7
io.searchbox	jest	5.3.3
io.undertow	undertow-core	1.4.25.Final
io.undertow	undertow-servlet	1.4.25.Final
io.undertow	undertow-websockets-jsr	1.4.25.Final
javax.annotation	javax.annotation-api	1.3.2
javax.cache	cache-api	1.1.0
javax.jms	javax.jms-api	2.0.1
javax.json	javax.json-api	1.1.2
javax.json.bind	javax.json.bind-api	1.0
javax.mail	javax.mail-api	1.6.1
javax.money	money-api	1.0.3
javax.servlet	javax.servlet-api	3.1.0
javax.servlet	jstl	1.2

Group ID	Artifact ID	Version
javax.transaction	javax.transaction-api	1.2
javax.validation	validation-api	2.0.1.Final
javax.xml.bind	jaxb-api	2.3.0
jaxen	jaxen	1.1.6
joda-time	joda-time	2.9.9
junit	junit	4.12
mysql	mysql-connector-java	46年5月1日
net.bytebuddy	byte-buddy	1.7.11
net.bytebuddy	byte-buddy-agent	1.7.11
net.java.dev.jna	jna	4.5.1
net.java.dev.jna	jna-platform	4.5.1
net.sf.ehcache	ehcache	2.10.5
net.sourceforge.htmlunit	htmlunit	2.29
net.sourceforge.jtds	jtds	1.3.1
net.sourceforge.nekohtml	nekohtml	22年9月1日
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect	2.3.0
org.apache.activemq	activemq-amqp	5.15.4
org.apache.activemq	activemq-blueprint	5.15.4
org.apache.activemq	activemq-broker	5.15.4
org.apache.activemq	activemq-camel	5.15.4
org.apache.activemq	activemq-client	5.15.4
org.apache.activemq	activemq-console	5.15.4
org.apache.activemq	activemq-http	5.15.4
org.apache.activemq	activemq-jaas	5.15.4
org.apache.activemq	activemq-jdbc-store	5.15.4
org.apache.activemq	activemq-jms-pool	5.15.4
org.apache.activemq	activemq-kahadb-store	5.15.4
org.apache.activemq	activemq-karaf	5.15.4
org.apache.activemq	activemq-leveldb-store	5.15.4
org.apache.activemq	activemq-log4j-appender	5.15.4
org.apache.activemq	activemq-mqtt	5.15.4
org.apache.activemq	activemq-openwire-generator	5.15.4
org.apache.activemq	activemq-openwire-legacy	5.15.4
org.apache.activemq	activemq-osgi	5.15.4
org.apache.activemq	activemq-partition	5.15.4
org.apache.activemq	activemq-pool	5.15.4
org.apache.activemq	activemq-ra	5.15.4
org.apache.activemq	activemq-run	5.15.4

Group ID	Artifact ID	Version
org.apache.activemq	activemq-runtime-config	5.15.4
org.apache.activemq	activemq-shiro	5.15.4
org.apache.activemq	activemq-spring	5.15.4
org.apache.activemq	activemq-stomp	5.15.4
org.apache.activemq	activemq-web	5.15.4
org.apache.activemq	artemis-amqp-protocol	2.4.0
org.apache.activemq	artemis-commons	2.4.0
org.apache.activemq	artemis-core-client	2.4.0
org.apache.activemq	artemis-jms-client	2.4.0
org.apache.activemq	artemis-jms-server	2.4.0
org.apache.activemq	artemis-journal	2.4.0
org.apache.activemq	artemis-native	2.4.0
org.apache.activemq	artemis-selector	2.4.0
org.apache.activemq	artemis-server	2.4.0
org.apache.activemq	artemis-service-extensions	2.4.0
org.apache.commons	commons-dbcp2	2.2.0
org.apache.commons	commons-lang3	3.7
org.apache.commons	commons-pool2	2.5.0
org.apache.derby	derby	10.14.1.0
org.apache.httpcomponents	fluent-hc	4.5.5
org.apache.httpcomponents	httpasyncclient	4.1.3
org.apache.httpcomponents	httpclient	4.5.5
org.apache.httpcomponents	httpclient-cache	4.5.5
org.apache.httpcomponents	httpclient-osgi	4.5.5
org.apache.httpcomponents	httpclient-win	4.5.5
org.apache.httpcomponents	httpcore	4.4.9
org.apache.httpcomponents	httpcore-nio	4.4.9
org.apache.httpcomponents	httpmime	4.5.5
org.apache.johnzon	johnzon-jsonb	1.1.7
org.apache.kafka	connect-api	1.0.1
org.apache.kafka	connect-file	1.0.1
org.apache.kafka	connect-json	1.0.1
org.apache.kafka	connect-runtime	1.0.1
org.apache.kafka	connect-transforms	1.0.1
org.apache.kafka	kafka_2.11	1.0.1
org.apache.kafka	kafka_2.12	1.0.1
org.apache.kafka	kafka-clients	1.0.1
org.apache.kafka	kafka-log4j-appender	1.0.1

Group ID	Artifact ID	Version
org.apache.kafka	kafka-streams	1.0.1
org.apache.kafka	kafka-tools	1.0.1
org.apache.logging.log4j	log4j-1.2-api	2.10.0
org.apache.logging.log4j	log4j-api	2.10.0
org.apache.logging.log4j	log4j-cassandra	2.10.0
org.apache.logging.log4j	log4j-core	2.10.0
org.apache.logging.log4j	log4j-couchdb	2.10.0
org.apache.logging.log4j	log4j-flume-ng	2.10.0
org.apache.logging.log4j	log4j-iostreams	2.10.0
org.apache.logging.log4j	log4j-jcl	2.10.0
org.apache.logging.log4j	log4j-jmx-gui	2.10.0
org.apache.logging.log4j	log4j-jul	2.10.0
org.apache.logging.log4j	log4j-liquibase	2.10.0
org.apache.logging.log4j	log4j-mongodb	2.10.0
org.apache.logging.log4j	log4j-slf4j-impl	2.10.0
org.apache.logging.log4j	log4j-taglib	2.10.0
org.apache.logging.log4j	log4j-to-slf4j	2.10.0
org.apache.logging.log4j	log4j-web	2.10.0
org.apache.solr	solr-analysis-extras	6.6.4
org.apache.solr	solr-analytics	6.6.4
org.apache.solr	solr-cell	6.6.4
org.apache.solr	solr-clustering	6.6.4
org.apache.solr	solr-core	6.6.4
org.apache.solr	solr-dataimporthandler	6.6.4
org.apache.solr	solr-dataimporthandler-extras	6.6.4
org.apache.solr	solr-langid	6.6.4
org.apache.solr	solr-solrj	6.6.4
org.apache.solr	solr-test-framework	6.6.4
org.apache.solr	solr-uima	6.6.4
org.apache.solr	solr-velocity	6.6.4
org.apache.tomcat	tomcat-annotations-api	31年8月5日
org.apache.tomcat	tomcat-catalina-jmx-remote	31年8月5日
org.apache.tomcat	tomcat-jdbc	31年8月5日
org.apache.tomcat	tomcat-jsp-api	31年8月5日
org.apache.tomcat.embed	tomcat-embed-core	31年8月5日
org.apache.tomcat.embed	tomcat-embed-el	31年8月5日
org.apache.tomcat.embed	tomcat-embed-jasper	31年8月5日
org.apache.tomcat.embed	tomcat-embed-websocket	31年8月5日

Group ID	Artifact ID	Version
org.aspectj	aspectjrt	1.8.13
org.aspectj	aspectjtools	1.8.13
org.aspectj	aspectjweaver	1.8.13
org.assertj	assertj-core	3.9.1
org.codehaus.btm	btm	2.1.4
org.codehaus.groovy	groovy	2.4.15
org.codehaus.groovy	groovy-all	2.4.15
org.codehaus.groovy	groovy-ant	2.4.15
org.codehaus.groovy	groovy-bsf	2.4.15
org.codehaus.groovy	groovy-console	2.4.15
org.codehaus.groovy	groovy-docgenerator	2.4.15
org.codehaus.groovy	groovy-groovydoc	2.4.15
org.codehaus.groovy	groovy-groovysh	2.4.15
org.codehaus.groovy	groovy-jmx	2.4.15
org.codehaus.groovy	groovy-json	2.4.15
org.codehaus.groovy	groovy-jsr223	2.4.15
org.codehaus.groovy	groovy-nio	2.4.15
org.codehaus.groovy	groovy-servlet	2.4.15
org.codehaus.groovy	groovy-sql	2.4.15
org.codehaus.groovy	groovy-swing	2.4.15
org.codehaus.groovy	groovy-templates	2.4.15
org.codehaus.groovy	groovy-test	2.4.15
org.codehaus.groovy	groovy-testng	2.4.15
org.codehaus.groovy	groovy-xml	2.4.15
org.codehaus.janino	janino	3.0.8
org.eclipse.jetty	apache-jsp	9.4.11.v20180605
org.eclipse.jetty	apache-jstl	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-client	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-conscrypt-client	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-conscrypt-server	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-java-client	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-java-server	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-openjdk8-client	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-openjdk8-server	9.4.11.v20180605
org.eclipse.jetty	jetty-alpn-server	9.4.11.v20180605
org.eclipse.jetty	jetty-annotations	9.4.11.v20180605
org.eclipse.jetty	jetty-ant	9.4.11.v20180605
org.eclipse.jetty	jetty-client	9.4.11.v20180605

Group ID	Artifact ID	Version
org.eclipse.jetty	jetty-continuation	9.4.11.v20180605
org.eclipse.jetty	jetty-deploy	9.4.11.v20180605
org.eclipse.jetty	jetty-distribution	9.4.11.v20180605
org.eclipse.jetty	jetty-hazelcast	9.4.11.v20180605
org.eclipse.jetty	jetty-home	9.4.11.v20180605
org.eclipse.jetty	jetty-http	9.4.11.v20180605
org.eclipse.jetty	jetty-http-spi	9.4.11.v20180605
org.eclipse.jetty	jetty-infinispan	9.4.11.v20180605
org.eclipse.jetty	jetty-io	9.4.11.v20180605
org.eclipse.jetty	jetty-jaas	9.4.11.v20180605
org.eclipse.jetty	jetty-jaspi	9.4.11.v20180605
org.eclipse.jetty	jetty-jmx	9.4.11.v20180605
org.eclipse.jetty	jetty-jndi	9.4.11.v20180605
org.eclipse.jetty	jetty-nosql	9.4.11.v20180605
org.eclipse.jetty	jetty-plus	9.4.11.v20180605
org.eclipse.jetty	jetty-proxy	9.4.11.v20180605
org.eclipse.jetty	jetty-quickstart	9.4.11.v20180605
org.eclipse.jetty	jetty-rewrite	9.4.11.v20180605
org.eclipse.jetty	jetty-security	9.4.11.v20180605
org.eclipse.jetty	jetty-server	9.4.11.v20180605
org.eclipse.jetty	jetty-servlet	9.4.11.v20180605
org.eclipse.jetty	jetty-servlets	9.4.11.v20180605
org.eclipse.jetty	jetty-spring	9.4.11.v20180605
org.eclipse.jetty	jetty-unixsocket	9.4.11.v20180605
org.eclipse.jetty	jetty-util	9.4.11.v20180605
org.eclipse.jetty	jetty-util-ajax	9.4.11.v20180605
org.eclipse.jetty	jetty-webapp	9.4.11.v20180605
org.eclipse.jetty	jetty-xml	9.4.11.v20180605
org.eclipse.jetty.cdi	cdi-core	9.4.11.v20180605
org.eclipse.jetty.cdi	cdi-servlet	9.4.11.v20180605
org.eclipse.jetty.fcgi	fcgi-client	9.4.11.v20180605
org.eclipse.jetty.fcgi	fcgi-server	9.4.11.v20180605
org.eclipse.jetty.gcloud	jetty-gcloud-session-manager	9.4.11.v20180605
org.eclipse.jetty.http2	http2-client	9.4.11.v20180605
org.eclipse.jetty.http2	http2-common	9.4.11.v20180605
org.eclipse.jetty.http2	http2-hpack	9.4.11.v20180605
org.eclipse.jetty.http2	http2-http-client-transport	9.4.11.v20180605

Group ID	Artifact ID	Version
org.eclipse.jetty.http2	http2-server	9.4.11.v20180605
org.eclipse.jetty.memcached	jetty-memcached-sessions	9.4.11.v20180605
org.eclipse.jetty.orbit	javax.servlet.jsp	2.2.0.v201112011158
org.eclipse.jetty.osgi	jetty-httpservice	9.4.11.v20180605
org.eclipse.jetty.osgi	jetty-osgi-boot	9.4.11.v20180605
org.eclipse.jetty.osgi	jetty-osgi-boot-jsp	9.4.11.v20180605
org.eclipse.jetty.osgi	jetty-osgi-boot-warurl	9.4.11.v20180605
org.eclipse.jetty.websocket	javax-websocket-client-impl	9.4.11.v20180605
org.eclipse.jetty.websocket	javax-websocket-server-impl	9.4.11.v20180605
org.eclipse.jetty.websocket	websocket-api	9.4.11.v20180605
org.eclipse.jetty.websocket	websocket-client	9.4.11.v20180605
org.eclipse.jetty.websocket	websocket-common	9.4.11.v20180605
org.eclipse.jetty.websocket	websocket-server	9.4.11.v20180605
org.eclipse.jetty.websocket	websocket-servlet	9.4.11.v20180605
org.ehcache	ehcache	3.5.2
org.ehcache	ehcache-clustered	3.5.2
org.ehcache	ehcache-transactions	3.5.2
org.elasticsearch	elasticsearch	5.6.10
org.elasticsearch.client	transport	5.6.10
org.elasticsearch.plugin	transport-netty4-client	5.6.10
org.firebirdsql.jdbc	jaybird-jdk17	3.0.4
org.firebirdsql.jdbc	jaybird-jdk18	3.0.4
org.flywaydb	flyway-core	5.0.7
org.freemarker	freemarker	2.3.28
org.glassfish	javax.el	3.0.0
org.glassfish.jersey.containers	jersey-container-servlet	2.26
org.glassfish.jersey.containers	jersey-container-servlet-core	2.26
org.glassfish.jersey.core	jersey-client	2.26
org.glassfish.jersey.core	jersey-common	2.26
org.glassfish.jersey.core	jersey-server	2.26
org.glassfish.jersey.ext	jersey-bean-validation	2.26
org.glassfish.jersey.ext	jersey-entity-filtering	2.26
org.glassfish.jersey.ext	jersey-spring4	2.26
org.glassfish.jersey.media	jersey-media-jaxb	2.26
org.glassfish.jersey.media	jersey-media-json-jackson	2.26
org.glassfish.jersey.media	jersey-media-multipart	2.26
org.hamcrest	hamcrest-core	1.3
org.hamcrest	hamcrest-library	1.3

Group ID	Artifact ID	Version
org.hibernate	hibernate-c3p0	5.2.17.Final
org.hibernate	hibernate-core	5.2.17.Final
org.hibernate	hibernate-ehcache	5.2.17.Final
org.hibernate	hibernate-entitymanager	5.2.17.Final
org.hibernate	hibernate-envers	5.2.17.Final
org.hibernate	hibernate-hikaricp	5.2.17.Final
org.hibernate	hibernate-infinispan	5.2.17.Final
org.hibernate	hibernate-java8	5.2.17.Final
org.hibernate	hibernate-jcache	5.2.17.Final
org.hibernate	hibernate-jpamodelgen	5.2.17.Final
org.hibernate	hibernate-proxool	5.2.17.Final
org.hibernate	hibernate-spatial	5.2.17.Final
org.hibernate	hibernate-testing	5.2.17.Final
org.hibernate.javax.persistence	hibernate-jpa-2.1-api	1.0.2.Final
org.hibernate.validator	hibernate-validator	6.0.10.Final
org.hibernate.validator	hibernate-validator-annotation-processor	6.0.10.Final
org.hsqldb	hsqldb	2.4.1
org.infinispan	infinispan-cachestore-jdbc	9.1.7.Final
org.infinispan	infinispan-cachestore-jpa	9.1.7.Final
org.infinispan	infinispan-cachestore-leveldb	9.1.7.Final
org.infinispan	infinispan-cachestore-remote	9.1.7.Final
org.infinispan	infinispan-cachestore-rest	9.1.7.Final
org.infinispan	infinispan-cachestore-rocksdb	9.1.7.Final
org.infinispan	infinispan-cdi-common	9.1.7.Final
org.infinispan	infinispan-cdi-embedded	9.1.7.Final
org.infinispan	infinispan-cdi-remote	9.1.7.Final
org.infinispan	infinispan-cli	9.1.7.Final
org.infinispan	infinispan-client-hotrod	9.1.7.Final
org.infinispan	infinispan-cloud	9.1.7.Final
org.infinispan	infinispan-clustered-counter	9.1.7.Final
org.infinispan	infinispan-commons	9.1.7.Final
org.infinispan	infinispan-core	9.1.7.Final
org.infinispan	infinispan-directory-provider	9.1.7.Final
org.infinispan	infinispan-embedded	9.1.7.Final
org.infinispan	infinispan-embedded-query	9.1.7.Final
org.infinispan	infinispan-hibernate-cache	9.1.7.Final
org.infinispan	infinispan-jcache	9.1.7.Final
org.infinispan	infinispan-jcache-commons	9.1.7.Final

Group ID	Artifact ID	Version
org.infinispan	infinispan-jcache-remote	9.1.7.Final
org.infinispan	infinispan-lucene-directory	9.1.7.Final
org.infinispan	infinispan-objectfilter	9.1.7.Final
org.infinispan	infinispan-osgi	9.1.7.Final
org.infinispan	infinispan-persistence-cli	9.1.7.Final
org.infinispan	infinispan-persistence-soft-index	9.1.7.Final
org.infinispan	infinispan-query	9.1.7.Final
org.infinispan	infinispan-query-dsl	9.1.7.Final
org.infinispan	infinispan-remote	9.1.7.Final
org.infinispan	infinispan-remote-query-client	9.1.7.Final
org.infinispan	infinispan-remote-query-server	9.1.7.Final
org.infinispan	infinispan-scripting	9.1.7.Final
org.infinispan	infinispan-server-core	9.1.7.Final
org.infinispan	infinispan-server-hotrod	9.1.7.Final
org.infinispan	infinispan-server-memcached	9.1.7.Final
org.infinispan	infinispan-server-router	9.1.7.Final
org.infinispan	infinispan-server-websocket	9.1.7.Final
org.infinispan	infinispan-spring4-common	9.1.7.Final
org.infinispan	infinispan-spring4-embedded	9.1.7.Final
org.infinispan	infinispan-spring4-remote	9.1.7.Final
org.infinispan	infinispan-tasks	9.1.7.Final
org.infinispan	infinispan-tasks-api	9.1.7.Final
org.infinispan	infinispan-tools	9.1.7.Final
org.infinispan	infinispan-tree	9.1.7.Final
org.influxdb	influxdb-java	2.9
org.jboss	jboss-transaction-spi	7.6.0.Final
org.jboss.logging	jboss-logging	3.3.2.Final
org.jboss.narayana.jta	jdbc	5.8.2.Final
org.jboss.narayana.jta	jms	5.8.2.Final
org.jboss.narayana.jta	jta	5.8.2.Final
org.jboss.narayana.jts	narayana-jts-integration	5.8.2.Final
org.jdom	jdom2	2.0.6
org.jetbrains.kotlin	kotlin-reflect	41年2月1日
org.jetbrains.kotlin	kotlin-runtime	41年2月1日
org.jetbrains.kotlin	kotlin-stdlib	41年2月1日
org.jetbrains.kotlin	kotlin-stdlib-jdk7	41年2月1日
org.jetbrains.kotlin	kotlin-stdlib-jdk8	41年2月1日
org.jetbrains.kotlin	kotlin-stdlib-jre7	41年2月1日

Group ID	Artifact ID	Version
org.jetbrains.kotlin	kotlin-stdlib-jre8	41年2月1日
org.jolokia	jolokia-core	1.5.0
org.jooq	jooq	3.10.7
org.jooq	jooq-codegen	3.10.7
org.jooq	jooq-meta	3.10.7
org.junit.jupiter	junit-jupiter-api	5.1.1
org.junit.jupiter	junit-jupiter-engine	5.1.1
org.junit.jupiter	junit-jupiter-params	5.1.1
org.junit.vintage	junit-vintage-engine	5.1.1
org.liquibase	liquibase-core	3.5.5
org.mariadb.jdbc	mariadb-java-client	2.2.5
org.mockito	mockito-core	2.15.0
org.mockito	mockito-inline	2.15.0
org.mongodb	bson	3.6.4
org.mongodb	mongodb-driver	3.6.4
org.mongodb	mongodb-driver-async	3.6.4
org.mongodb	mongodb-driver-core	3.6.4
org.mongodb	mongodb-driver-reactivestreams	1.7.1
org.mongodb	mongo-java-driver	3.6.4
org.mortbay.jasper	apache-el	8.5.24.2
org.neo4j	neo4j-ogm-api	3.1.0
org.neo4j	neo4j-ogm-bolt-driver	3.1.0
org.neo4j	neo4j-ogm-core	3.1.0
org.neo4j	neo4j-ogm-http-driver	3.1.0
org.postgresql	postgresql	42.2.2
org.projectlombok	lombok	22年1月16日
org.quartz-scheduler	quartz	2.3.0
org.quartz-scheduler	quartz-jobs	2.3.0
org.reactivestreams	reactive-streams	1.0.2
org.seleniumhq.selenium	htmlunit-driver	2.29.3
org.seleniumhq.selenium	selenium-api	3.9.1
org.seleniumhq.selenium	selenium-chrome-driver	3.9.1
org.seleniumhq.selenium	selenium-edge-driver	3.9.1
org.seleniumhq.selenium	selenium-firefox-driver	3.9.1
org.seleniumhq.selenium	selenium-ie-driver	3.9.1
org.seleniumhq.selenium	selenium-java	3.9.1
org.seleniumhq.selenium	selenium-opera-driver	3.9.1
org.seleniumhq.selenium	selenium-remote-driver	3.9.1

Group ID	Artifact ID	Version
org.seleniumhq.selenium	selenium-safari-driver	3.9.1
org.seleniumhq.selenium	selenium-support	3.9.1
org.skyscreamer	jsonassert	1.5.0
org.slf4j	jcl-over-slf4j	1.7.25
org.slf4j	jul-to-slf4j	1.7.25
org.slf4j	log4j-over-slf4j	1.7.25
org.slf4j	slf4j-api	1.7.25
org.slf4j	slf4j-ext	1.7.25
org.slf4j	slf4j-jcl	1.7.25
org.slf4j	slf4j-jdk14	1.7.25
org.slf4j	slf4j-log4j12	1.7.25
org.slf4j	slf4j-nop	1.7.25
org.slf4j	slf4j-simple	1.7.25
org.springframework	spring-aop	5.0.7.RELEASE
org.springframework	spring-aspects	5.0.7.RELEASE
org.springframework	spring-beans	5.0.7.RELEASE
org.springframework	spring-context	5.0.7.RELEASE
org.springframework	spring-context-indexer	5.0.7.RELEASE
org.springframework	spring-context-support	5.0.7.RELEASE
org.springframework	spring-core	5.0.7.RELEASE
org.springframework	spring-expression	5.0.7.RELEASE
org.springframework	spring-instrument	5.0.7.RELEASE
org.springframework	spring-jcl	5.0.7.RELEASE
org.springframework	spring-jdbc	5.0.7.RELEASE
org.springframework	spring-jms	5.0.7.RELEASE
org.springframework	spring-messaging	5.0.7.RELEASE
org.springframework	spring-orm	5.0.7.RELEASE
org.springframework	spring-oxm	5.0.7.RELEASE
org.springframework	spring-test	5.0.7.RELEASE
org.springframework	spring-tx	5.0.7.RELEASE
org.springframework	spring-web	5.0.7.RELEASE
org.springframework	spring-webflux	5.0.7.RELEASE
org.springframework	spring-webmvc	5.0.7.RELEASE
org.springframework	spring-websocket	5.0.7.RELEASE
org.springframework.amqp	spring-amqp	2.0.4.RELEASE
org.springframework.amqp	spring-rabbit	2.0.4.RELEASE
org.springframework.amqp	spring-rabbit-junit	2.0.4.RELEASE
org.springframework.amqp	spring-rabbit-test	2.0.4.RELEASE

Group ID	Artifact ID	Version
org.springframework.batch	spring-batch-core	4.0.1.RELEASE
org.springframework.batch	spring-batch-infrastructure	4.0.1.RELEASE
org.springframework.batch	spring-batch-integration	4.0.1.RELEASE
org.springframework.batch	spring-batch-test	4.0.1.RELEASE
org.springframework.boot	spring-boot	2.0.3.RELEASE
org.springframework.boot	spring-boot-actuator	2.0.3.RELEASE
org.springframework.boot	spring-boot-actuator-autoconfigure	2.0.3.RELEASE
org.springframework.boot	spring-boot-autoconfigure	2.0.3.RELEASE
org.springframework.boot	spring-boot-autoconfigure-processor	2.0.3.RELEASE
org.springframework.boot	spring-boot-configuration-metadata	2.0.3.RELEASE
org.springframework.boot	spring-boot-configuration-processor	2.0.3.RELEASE
org.springframework.boot	spring-boot-devtools	2.0.3.RELEASE
org.springframework.boot	spring-boot-loader	2.0.3.RELEASE
org.springframework.boot	spring-boot-loader-tools	2.0.3.RELEASE
org.springframework.boot	spring-boot-properties-migrator	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-activemq	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-actuator	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-amqp	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-aop	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-artemis	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-batch	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-cache	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-cloud-connectors	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-cassandra	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-cassandra-reactive	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-couchbase	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-couchbase-reactive	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-elasticsearch	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-jpa	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-ldap	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-mongodb	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-mongodb-reactive	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-neo4j	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-redis	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-redis-reactive	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-rest	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-data-solr	2.0.3.RELEASE

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-freemarker	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-groovy-templates	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-hateoas	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-integration	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-jdbc	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-jersey	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-jetty	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-jooq	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-json	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-jta-atomikos	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-jta-bitronix	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-jta-narayana	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-log4j2	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-logging	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-mail	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-mustache	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-quartz	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-reactor-netty	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-security	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-test	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-thymeleaf	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-tomcat	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-undertow	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-validation	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-web	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-webflux	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-web-services	2.0.3.RELEASE
org.springframework.boot	spring-boot-starter-websocket	2.0.3.RELEASE
org.springframework.boot	spring-boot-test	2.0.3.RELEASE
org.springframework.boot	spring-boot-test-autoconfigure	2.0.3.RELEASE
org.springframework.cloud	spring-cloud-cloudfoundry-connector	2.0.2.RELEASE
org.springframework.cloud	spring-cloud-connectors-core	2.0.2.RELEASE
org.springframework.cloud	spring-cloud-heroku-connector	2.0.2.RELEASE
org.springframework.cloud	spring-cloud-localconfig-connector	2.0.2.RELEASE
org.springframework.cloud	spring-cloud-spring-service-connector	2.0.2.RELEASE
org.springframework.data	spring-data-cassandra	2.0.8.RELEASE
org.springframework.data	spring-data-commons	2.0.8.RELEASE
org.springframework.data	spring-data-couchbase	3.0.8.RELEASE

Group ID	Artifact ID	Version
org.springframework.data	spring-data-elasticsearch	3.0.8.RELEASE
org.springframework.data	spring-data-envers	2.0.8.RELEASE
org.springframework.data	spring-data-gemfire	2.0.8.RELEASE
org.springframework.data	spring-data-geode	2.0.8.RELEASE
org.springframework.data	spring-data-jpa	2.0.8.RELEASE
org.springframework.data	spring-data-keyvalue	2.0.8.RELEASE
org.springframework.data	spring-data-ldap	2.0.8.RELEASE
org.springframework.data	spring-data-mongodb	2.0.8.RELEASE
org.springframework.data	spring-data-mongodb-cross-store	2.0.8.RELEASE
org.springframework.data	spring-data-neo4j	5.0.8.RELEASE
org.springframework.data	spring-data-redis	2.0.8.RELEASE
org.springframework.data	spring-data-rest-core	3.0.8.RELEASE
org.springframework.data	spring-data-rest-hal-browser	3.0.8.RELEASE
org.springframework.data	spring-data-rest-webmvc	3.0.8.RELEASE
org.springframework.data	spring-data-solr	3.0.8.RELEASE
org.springframework.hateoas	spring-hateoas	0.24.0.RELEASE
org.springframework.integration	spring-integration-amqp	5.0.6.RELEASE
org.springframework.integration	spring-integration-core	5.0.6.RELEASE
org.springframework.integration	spring-integration-event	5.0.6.RELEASE
org.springframework.integration	spring-integration-feed	5.0.6.RELEASE
org.springframework.integration	spring-integration-file	5.0.6.RELEASE
org.springframework.integration	spring-integration-ftp	5.0.6.RELEASE
org.springframework.integration	spring-integration-gemfire	5.0.6.RELEASE
org.springframework.integration	spring-integration-groovy	5.0.6.RELEASE
org.springframework.integration	spring-integration-http	5.0.6.RELEASE
org.springframework.integration	spring-integration-ip	5.0.6.RELEASE
org.springframework.integration	spring-integration-jdbc	5.0.6.RELEASE
org.springframework.integration	spring-integration-jms	5.0.6.RELEASE
org.springframework.integration	spring-integration-jmx	5.0.6.RELEASE
org.springframework.integration	spring-integration-jpa	5.0.6.RELEASE
org.springframework.integration	spring-integration-mail	5.0.6.RELEASE
org.springframework.integration	spring-integration-mongodb	5.0.6.RELEASE
org.springframework.integration	spring-integration-mqtt	5.0.6.RELEASE
org.springframework.integration	spring-integration-redis	5.0.6.RELEASE
org.springframework.integration	spring-integration-rmi	5.0.6.RELEASE
org.springframework.integration	spring-integration-scripting	5.0.6.RELEASE
org.springframework.integration	spring-integration-security	5.0.6.RELEASE
org.springframework.integration	spring-integration-sftp	5.0.6.RELEASE

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-stomp	5.0.6.RELEASE
org.springframework.integration	spring-integration-stream	5.0.6.RELEASE
org.springframework.integration	spring-integration-syslog	5.0.6.RELEASE
org.springframework.integration	spring-integration-test	5.0.6.RELEASE
org.springframework.integration	spring-integration-test-support	5.0.6.RELEASE
org.springframework.integration	spring-integration-twitter	5.0.6.RELEASE
org.springframework.integration	spring-integration-webflux	5.0.6.RELEASE
org.springframework.integration	spring-integration-websocket	5.0.6.RELEASE
org.springframework.integration	spring-integration-ws	5.0.6.RELEASE
org.springframework.integration	spring-integration-xml	5.0.6.RELEASE
org.springframework.integration	spring-integration-xmpp	5.0.6.RELEASE
org.springframework.integration	spring-integration-zookeeper	5.0.6.RELEASE
org.springframework.kafka	spring-kafka	2.1.7.RELEASE
org.springframework.kafka	spring-kafka-test	2.1.7.RELEASE
org.springframework.ldap	spring-ldap-core	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-core-tiger	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-ldif-batch	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-ldif-core	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-odm	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-test	2.3.2.RELEASE
org.springframework.plugin	spring-plugin-core	1.2.0.RELEASE
org.springframework.plugin	spring-plugin-metadata	1.2.0.RELEASE
org.springframework.restdocs	spring-restdocs-asciidoctor	2.0.1.RELEASE
org.springframework.restdocs	spring-restdocs-core	2.0.1.RELEASE
org.springframework.restdocs	spring-restdocs-mockmvc	2.0.1.RELEASE
org.springframework.restdocs	spring-restdocs-restassured	2.0.1.RELEASE
org.springframework.restdocs	spring-restdocs-webtestclient	2.0.1.RELEASE
org.springframework.retry	spring-retry	1.2.2.RELEASE
org.springframework.security	spring-security-acl	5.0.6.RELEASE
org.springframework.security	spring-security-aspects	5.0.6.RELEASE
org.springframework.security	spring-security-cas	5.0.6.RELEASE
org.springframework.security	spring-security-config	5.0.6.RELEASE
org.springframework.security	spring-security-core	5.0.6.RELEASE
org.springframework.security	spring-security-crypto	5.0.6.RELEASE
org.springframework.security	spring-security-data	5.0.6.RELEASE
org.springframework.security	spring-security-ldap	5.0.6.RELEASE
org.springframework.security	spring-security-messaging	5.0.6.RELEASE
org.springframework.security	spring-security-oauth2-client	5.0.6.RELEASE

Group ID	Artifact ID	Version
org.springframework.security	spring-security-oauth2-core	5.0.6.RELEASE
org.springframework.security	spring-security-oauth2-jose	5.0.6.RELEASE
org.springframework.security	spring-security-openid	5.0.6.RELEASE
org.springframework.security	spring-security-remoting	5.0.6.RELEASE
org.springframework.security	spring-security-taglibs	5.0.6.RELEASE
org.springframework.security	spring-security-test	5.0.6.RELEASE
org.springframework.security	spring-security-web	5.0.6.RELEASE
org.springframework.session	spring-session-core	2.0.4.RELEASE
org.springframework.session	spring-session-data-gemfire	2.0.2.RELEASE
org.springframework.session	spring-session-data-geode	2.0.2.RELEASE
org.springframework.session	spring-session-data-mongodb	2.0.2.RELEASE
org.springframework.session	spring-session-data-redis	2.0.4.RELEASE
org.springframework.session	spring-session-hazelcast	2.0.4.RELEASE
org.springframework.session	spring-session-jdbc	2.0.4.RELEASE
org.springframework.ws	spring-ws-core	3.0.1.RELEASE
org.springframework.ws	spring-ws-security	3.0.1.RELEASE
org.springframework.ws	spring-ws-support	3.0.1.RELEASE
org.springframework.ws	spring-ws-test	3.0.1.RELEASE
org.springframework.ws	spring-xml	3.0.1.RELEASE
org.synchronoss.cloud	nio-multipart-parser	1.1.0
org.thymeleaf	thymeleaf	3.0.9.RELEASE
org.thymeleaf	thymeleaf-spring5	3.0.9.RELEASE
org.thymeleaf.extras	thymeleaf-extras-java8time	3.0.1.RELEASE
org.thymeleaf.extras	thymeleaf-extras-springsecurity4	3.0.2.RELEASE
org.webjars	hal-browser	3325375
org.webjars	webjars-locator-core	0.35
org.xerial	sqlite-jdbc	3.21.0.1
org.xmlunit	xmlunit-core	2.5.1
org.xmlunit	xmlunit-legacy	2.5.1
org.xmlunit	xmlunit-matchers	2.5.1
org.yaml	snakeyaml	1.19
redis.clients	jedis	2.9.0
wsdl4j	wsdl4j	1.6.3
xml-apis	xml-apis	1.4.01