geekidentity

Spring Boot中文文档（官方文档翻译 基于1.5.2.RELEASE）

2017-07-28 | spring

作者：Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons

1.5.2.RELEASE

翻译：geekidentity

持续更新： http://blog.geekidentity.com/spring/spring_boot_translation/

欢迎加群交流：490788638

Part I. Spring Boot 文档

本节简要介绍了Spring Boot文档，是整个文档的参考指南。 您可以完整阅读本参考指南，或者如果您不感兴趣的话可以跳过该部分。

1. 关于文档

Spring Boot参考指南可以以 html，pdf 和 epub 文档的形式获取。 最新版本的文档可在 http://docs.spring.io/spring-boot/docs/current/reference 中找到。

2. 获得帮助

如果使用 Spring Boot 时遇到问题，可以在下面获取帮助：

- 尝试 How-to’ s - 这里为最常见的问题提供解决方案。
- 了解Spring的基础知识 - Spring Boot建立在许多其他Spring项目上，请查看 spring.io 网站以获取其他项目的参考文档。 如果您刚刚开始使用Spring，请阅读这个指南。
- 在stackoverflow上提问题 - 我们会一起关注 stackoverflow.com 上有spring-boot标签的问题。
- 在Github上报告bug。

> Spring Boot 所有的东西都是开源的，包括文档！ 如果您发现文档有问题; 或者如果你想改进他们，欢迎参与。

3. 第一步

如果你刚刚开始使用 Spring Boot，或刚刚开始使用“Spring” ，请从这里开始！

- 从头开始：概述 | 要求 | 安装
- 教程：Part 1 | Part 2
- 运行你的例子：Part 1 | Part 2

4. 使用 Spring Boot

- 构建系统: Maven | Gradle | Ant | Starters
- 最佳做法：代码结构 | @Configuration | @EnableAutoConfiguration | Beans 和依赖注入
- 运行代码：IDE | Packaged | Maven | Gradle
- 打包应用程序： Production jars
- Spring Boot CLI：使用CLI

5. 了解Spring Boot功能

需要有关Spring Boots核心功能的更多细节？请看这里

- 核心功能: SpringApplication | External Configuration | Profiles | Logging
- Web 应用: MVC | Embedded Containers
- 数据处理: SQL | NO-SQL
- 消息处理: Overview | JMS
- 测试: Overview | Boot Applications | Utils
- 扩展: Auto-configuration | @Conditions

6. 转移到生产环境

当您准备好将Spring Boot 应用程序放到生产环境时，我们有一些您可能会喜欢的技巧！

- 部署 Spring Boot 应用程序: Cloud Deployment | OS Service
  构建工具插件：Maven | Gradle
  附录: Application Properties | 自动配置类 | 可执行 Jars

Part II. 入门指南

如果你刚刚开始使用Spring Boot，这是你的一部分内容！ 在这里我们将会回答一些基本的“what?”，“how?” 和 “why?”的问题。 在这里你会找到一个详细的Spring Boot介绍和安装说明。 然后，我们将构建我们的第一个Spring Boot应用程序，并讨论一些核心原则。

8. Spring Boot 介绍

Spring Boot可以基于Spring轻松创建可以“运行”的、独立的、生产级的应用程序。 对Spring平台和第三方类库我们有自己看法和意见（约定大于配置），所以你最开始的时候不要感到奇怪。大多数Spring Boot应用程序需要很少的Spring配置。

您可以使用Spring Boot创建可以使用 `java -jar` 或传统 war 包部署启动的Java应用程序。 我们还提供一个运行“spring scripts”的命令行工具。

我们的主要目标是：

- 为所有的Spring开发者提供一个更快，更广泛接受的入门体验。
- 开始使用开箱即用的配置（极少配置甚至不用配置），但随着需求开始配置自己所需要的值（即所有配置都有默认值，同时也可以根据自己的需求进行配置）。
- 提供大量项目中常见的一系列非功能特征（例如嵌入式服务器，安全性，指标，运行状况检查，外部化配置）。
- 绝对没有代码生成，也不需要XML配置。

9. 系统要求

默认情况下，Spring Boot 1.5.2.RELEASE需要Java 7和Spring Framework 4.3.7.RELEASE或更高版本。 您可以进行一些其他配置在Java 6上使用Spring Boot。 有关详细信息，请参见第84.11节“如何使用Java 6”。 为Maven（3.2+）、Gradle 2（2.9或更高版本）和3提供了显式构建支持。

> 虽然您可以在Java 6或7上使用 Spring Boot，但我们通常推荐Java 8。

9.1 Servlet容器

以下嵌入式servlet容器可以直接使用：

名称	Servlet 版本	Java 版本
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9.3	3.1	Java 8+
Jetty 9.2	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.3	3.1	Java 7+

您还可以将Spring Boot应用程序部署到任何兼容Servlet 3.0+ 的容器中。

10. 安装 Spring Boot

Spring Boot可以与“经典(classic)”Java开发工具一起使用或作为命令行工具安装。 无论如何，您将需要Java SDK v1.6或更高版本。 在开始之前检查当前的Java安装：

```
1   $ java -version
```

如果您是Java开发的新手，或者您只想尝试一下 Spring Boot，您可能需要首先尝试使用 Spring Boot CLI，如果想正式使用Spring Boot，请阅读“经典(classic)”安装说明。

虽然Spring Boot 与Java 1.6兼容，但我们建议使用最新版本的Java。

10.1 针对Java开发程序员安装说明

Spring Boot的使用方式与标准Java库的使用相同，只需在类路径中包含适当的 `spring-boot-*.jar` 文件。Spring Boot不需要任何特殊的集成工具，所以可以使用任何IDE或文本编辑器进行开发；并且Spring Boot 应用程序没有什么特殊的地方，因此您可以像其他Java程序一样运行和调试。虽然您可以直接复制Spring Boot的jar包，但我们通常建议您使用依赖关系管理的构建工具（如Maven或Gradle）。

10.1.1 Maven安装

Spring Boot 兼容 Apache Maven 3.2。 如果您还没有安装Maven，可以按照 https://maven.apache.org/ 上的说明进行安装。

在许多操作系统上，Maven可以通过软件包管理器进行安装。 如果您是OSX Homebrew用户，请尝试使用命令：brew install maven。 Ubuntu用户可以运行命令：sudo apt-get install maven。

Spring Boot 依赖 org.springframework.boot groupId。通常，您的Maven POM文件将从 spring-boot-starter-parent 项目继承，并声明一个或多个“启动器(启动器)”的依赖关系。Spring Boot还提供了一个可选的Maven插件来创建可执行的jar包。

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/XMLS
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/mav
    <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>1.5.2.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的一个很好的方式，但它并不是所有的时候都适合。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。 请参见第13.2.2节“使用不带父

> POM的Spring Boot”作为使用导入作用域(import scope)的替代解决方案。

10.1.2 Gradle 安装

Spring Boot 兼容 Gradle 2（2.9或更高版本）和Gradle 3。如果您尚未安装Gradle，您可以按照 http://www.gradle.org/ 上的说明进行操作。

可以使用org.springframework.boot 组(group)声明Spring Boot 的依赖项。 通常，您的项目将声明一个或多个“启动器(Starters)”的依赖。Spring Boot提供了一个有用的Gradle插件，可用于简化依赖关系声明和创建可执行 jar包。

Gradle Wrapper

当您需要构建项目时，Gradle Wrapper提供了一种“获取(obtaining)”Gradle的更好的方式。 它是一个小脚本和库，它与代码一起引导构建过程。 有关详细信息，请参阅 https://docs.gradle.org/2.14.1/userguide/gradle_wrapper.html 。

典型的 build.gradle 文件：

```
plugins {
    id 'org.springframework.boot' version '1.5.2.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 安装Spring Boot CLI

Spring Boot CLI是一个命令行工具，如果要使用Spring快速原型(quickly prototype)，可以使用它。 它允许您运行Groovy脚本，这意味着会有您熟悉的类似Java的语法，没有太多的样板代码(boilerplate code)。

您也不必要通过CLI来使用Spring Boot，但它绝对是开始Spring应用程序最快方法。

10.2.1 手动安装

您可以从Spring软件版本库下载Spring CLI发行版：

- spring-boot-cli-1.5.2.RELEASE-bin.zip
- spring-boot-cli-1.5.2.RELEASE-bin.tar.gz

各发布版本的快照。

下载完成后，请按照解压缩后文件中的INSTALL.txt的说明进行操作。 总而言之：在.zip文件的bin/目录中有一个spring脚本（Windows的spring.bat），或者你可以使用java -jar（脚本可以帮助您确保类路径设置正确）。

10.2.2 使用SDKMAN!安装

SDKMAN!（软件开发套件管理器）可用于管理各种二进制SDK的多个版本，包括Groovy和Spring Boot CLI。从http://sdkman.io/ 获取SDKMAN！并安装Spring Boot。

```
$ sdk install springboot
$ spring --version
Spring Boot v1.5.2.RELEASE
```

如果您正在开发CLI的功能，并希望轻松访问刚创建的版本，请遵循以下额外说明。

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-1
$ sdk default springboot dev
$ spring --version
Spring CLI v1.5.2.RELEASE
```

这将安装一个称为dev的spring的本地实例(instance)。 它指向您构建位置的target，所以每次重建(rebuild)Spring Boot时，Spring 将是最新的。

你可以看到：

```
$ sdk ls springboot

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

================================================================================
```

```
10  + - local version
11  * - installed
12  > - currently in use
13  ================================================================================
```

10.2.3 OSX Homebrew 安装

如果您在Mac上使用 Homebrew，安装Spring Boot CLI 只需要下面命令：

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

Homebrew会将Spring 安装到 /usr/local/bin。

> 如果您没有看到公式(formula)，您的安装可能会过期。 只需执行brew更新，然后重试。

10.2.4 MacPorts安装

如果您在Mac上使用 MacPorts，安装Spring Boot CLI 只需要下面命令：

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

10.2.5 命令行提示

Spring Boot CLI为BASH和zsh shell提供命令提示的功能。 您可以在任何shell中引用脚本（也称为spring），或将其放在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 中，当新的shell启动时，该目录中的所有脚本将被执行。 手动运行脚本，例如 如果您使用SDKMAN安装了！

```
1  $ . ~/.sdkman/candidates/springboot/current/shell-completion/bash/spring
2  $ spring <HIT TAB HERE>
3    grab  help  jar  run  test  version
```

> 如果使用Homebrew或MacPorts安装Spring Boot CLI，则命令行补全脚本将自动注册到您的shell。

10.2.6 快速启动Spring CLI示例

这是一个非常简单的Web应用程序，可用于测试您的安装是否正确。 创建一个名为app.groovy的文件：

```
1  @RestController
2  class ThisWillActuallyRun {
3
4      @RequestMapping("/")
```

```
5       String home() {
6           "Hello World!"
7       }
8
9   }
```

然后从shell运行它：

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

因为下载依赖的库，首次运行应用程序需要一些时间，。 后续运行将会更快。

在浏览器中打开 http://localhost:8080 ，您应该会看到以下输出：

```
1   Hello World!
```

10.3 从早期版本的Spring Boot升级

如果您从早期版本的 Spring Boot 升级，请检查项目wiki上托管的“发行说明”。 您将找到升级说明以及每个版本的“新的和值得注意的”功能的列表。

要升级现有的CLI安装，请使用包管理工具相应的package manager命令（例如brew upgrade），如果您手动安装了CLI，请按照标准说明记住更新PATH环境变量以删除任何旧的引用。

11. 开发您的第一个Spring Boot应用程序

让我们在Java中开发一个简单的“Hello World！”Web应用程序，突显Spring Boot一些主要的功能。 我们将使用Maven构建该项目，因为大多数IDE支持它。

> https://spring.io/ 包含许多使用Spring Boot的“入门指南”。 如果您正在寻求解决一些具体问题; 可以先看一下那里。
>
> 您可以在 https://start.spring.io/ 的依赖关系搜索器中选择Web启动器来快速完成以下步骤。
> 这会自动生成一个新的项目结构，方便您立即开始编码。 查看文档了解更多详细信息。

在开始之前，打开终端来检查您是否安装了有效的Java和Maven版本。

```
1   $ java -version
2   java version "1.7.0_51"
3   Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
4   Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
5
```

```
$ mvn -v
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T13:58:10-07:00)
Maven home: /Users/user/tools/apache-maven-3.1.1
Java version: 1.7.0_51, vendor: Oracle Corporation
```

> 这个示例需要在其自己的文件夹中创建。 后面我们假设您在当前目录已经创建了一个正确的文件夹。

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置文件。打开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/XMLS
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/mav
    <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>1.5.2.RELEASE</version>
    </parent>

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

</project>
```

这应该给你一个工作构建(working build)，你可以通过运行 mvn package 进行测试（你可以暂时忽略警告：“jar will be empty - no content was marked for inclusion!”）。

> 现在，您可以将项目导入到IDE中（最新的Java IDE内置对Maven的支持）。 为了简单起见，这个示例我们继续使用纯文本编辑器。

11.2 添加类路径依赖关系

Spring Boot提供了一些“启动器(Starters)”，可以方便地将jar添加到类路径中。我们的示例应用程序已经在POM的父部分使用了spring-boot-starter-parent。spring-boot-starter-parent是一个特殊启动器，提供一些Maven的默认值。它还提供依赖管理 dependency-management 标签，以便您可以省略子模块依赖关系的版本标签。

其他“启动器(Starters)”只是提供您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Web应用程序，所以我们将添加一个spring-boot-starter-web依赖关系，但在此之前，我们来看看我们目前的依赖。

```
$ mvn dependency:tree

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

mvn dependency:tree：打印项目依赖关系的树形表示。 您可以看到spring-boot-starter-parent本身不在依赖关系中。 编辑pom.xml并在 parent 下添加spring-boot-starter-web依赖关系：

```
<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 ，您将看到现在有许多附加依赖关系，包括Tomcat Web服务器和Spring Boot本身。

11.3 编写代码

要完成我们的应用程序，我们需要创建一个的Java文件。 默认情况下，Maven将从src/main/java编译源代码，因此您需要创建该文件夹结构，然后添加一个名为src/main/java/Example.java的文件：

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
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，但是有一些重要的部分。

11.3.1 @RestController和@RequestMapping 注解

我们的Example类的第一个注解是@RestController。 这被称为 stereotype annotation。它为人们阅读代码提供了一些提示，对于Spring来说，这个类具有特定的作用。在这里，我们的类是一个web @Controller，所以Spring在处理传入的Web请求时会考虑这个类。

@RequestMapping注解提供“路由”信息。 告诉Spring，任何具有路径“/”的HTTP请求都应映射到home方法。 @RestController注解告诉Spring将生成的字符串直接返回给调用者。

> @RestController和@RequestMapping注解是Spring MVC 的注解（它们不是Spring Boot特有的）。 有关更多详细信息，请参阅Spring参考文档中的MVC部分。

11.3.2 @EnableAutoConfiguration注解

第二个类级别的注释是@EnableAutoConfiguration。 这个注解告诉 Spring Boot 根据您添加的jar依赖关系来“猜(guess)”你将如何配置Spring。由于spring-boot-starter-web添加了Tomcat和Spring MVC，自动配置将假定您正在开发Web应用程序并相应地配置Spring。

启动器和自动配置

自动配置旨在与“起动器”配合使用，但两个概念并不直接相关。 您可以自由选择启动器之外的jar依赖项，Spring Boot仍然会自动配置您的应用程序。

11.3.3 “main”方法

我们的应用程序的最后一部分是main()方法。 这只是一个遵循Java惯例的应用程序入口点的标准方法。 我们的main()方法通过调用run()委托(delegates)给Spring Boot的SpringApplication类。 SpringApplication将引导我们的应用程序，启动Spring，然后启动自动配置的Tomcat Web服务器。 我们需要将Example.class作为一个参数传递给run方法来告诉SpringApplication，它是主要的Spring组件。 还传递了args数组以传递命令行参数。

11.4 运行示例

由于我们使用了spring-boot-starter-parent POM，所以我们有一个可用的运行目标，我们可以使用它来启动应用程序。 键入mvn spring-boot：从根目录运行以启动应用程序：

```
$ mvn spring-boot:run

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
```

```
  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
    '  |____| .__|_| |_|_| |_\__, | / / / /
   =========|_|==============|___/=/_/_/_/
   :: Spring Boot ::  (v1.5.2.RELEASE)
  ....... . . .
  ....... . . . (log output here)
  ....... . . .
  ........ Started Example in 2.222 seconds (JVM running for 6.514)
```

如果你用浏览器打开 http://localhost:8080 你应该看到以下输出：

```
Hello World!
```

ctrl-c 正常(gracefully)退出应用程序。

11.5 创建可执行的jar

让我们完成我们的例子，创建一个完全自包含的可执行jar文件，我们可以在生产环境中运行。 可执行的jar（有时称为“fat jars”）是包含编译的类以及代码运行所需要的所有jar包依赖的归档(archives)。

可执行jar和Java

Java不提供任何标准的方法来加载嵌套的jar文件（即本身包含在jar中的jar文件）。 如果您正在寻找可以发布自包含的应用程序，这可能是有问题的。
为了解决这个问题，许多开发人员使用“uber” jars。 一个uber jar简单地将所有类、jar包进行档案。 这种方法的问题是，很难看到您在应用程序中实际使用哪些库。 如果在多个jar中使用相同的文件名（但具有不同的内容），也可能会出现问题。
Spring Boot采用一个不同的方法这样可以直接对jar进行嵌套。

要创建可执行的jar，我们需要将spring-boot-maven-plugin添加到我们的pom.xml中。 在 dependencies 标签 下方插入以下行：

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

> spring-boot-starter-parent POM 包括重新打包目标的 executions标签 配置。 如果您不使用该父POM，您将需要自己声明此配置。 有关详细信息，请参阅插件文档。

保存您的pom.xml并从命令行运行 mvn package：

```
$ 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-
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.2.RELEASE:repackage (default) @ myproject ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

如果你看看target目录，你应该看到myproject-0.0.1-SNAPSHOT.jar。 该文件的大小约为10 MB。 如果你想查看里面，可以使用jar tvf：

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

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

使用java -jar命令运行该应用程序：

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

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.5.2.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864)
```

像之前一样，ctrl+c正常退出应用程序。

12. 接下来应该读什么

希望本节能为您提供一些Spring Boot基础知识，并让您准备编写自己的应用程序。 如果你是一个面向具体任务的开发人员，你可能想跳过 https://spring.io/ ，看看一些解决具体的“如何用Spring”问题的入门指南; 我们还有Spring Boot-specific How-to参考文档。

Spring Boot库还有一大堆可以运行的示例。 示例与代码的其余部分是独立的（这样您不需要构建多余的代码来运行或使用示例）。

下一个是第三部分“使用 Spring Boot”。 如果你真的没有这个耐心，也可以跳过去阅读Spring Boot功能。

Part III. 使用 Spring Boot

本部分将详细介绍如何使用Spring Boot。 这部分涵盖诸如构建系统，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(best practices)。 虽然Spring Boot没有什么特殊之处（它只是一个可以使用的库），但是有一些建议可以让您的开发过程更容易一些。

如果您刚刚开始使用Spring Boot，那么在深入本部分之前，您应该阅读入门指南。

13. 构建系统

强烈建议您选择一个支持依赖管理并可以使用“Maven Central”存储库的构建系统。 我们建议您选择Maven或Gradle。 Spring Boot 可以与其他构建系统（例如 Ant ）配合使用，但是它们不会得到很好的支持。

13.1 依赖管理

每个版本的Spring Boot提供了一个它所支持的依赖关系列表。 实际上，您不需要为构建配置文件中的这些依赖关系提供版本，因为Spring Boot会为您进行管理这些依赖的版本。 当您升级Spring Boot本身时，这些依赖关系也将以一致的进行升级。

> 如果您觉得有必要，您仍然可以指定一个版本并覆盖Spring Boot建议的版本。

管理的列表中包含可以使用Spring Boot的所有Spring模块以及第三方库的精简列表。 该列表可作为标准的物料(Materials)清单（spring-boot-dependencies）使用，并且还提供了对 Maven 和 Gradle 的额外支持。

> Spring Boot的每个版本与Spring Framework的基本版本相关联，因此我们强烈建议您不要自己指定其版本。

13.2 Maven

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

- Java 1.6作为默认编译器级别。
- 源代码UTF-8编码。
- 依赖关系管理，允许您省略常见依赖的标签，其默认版本继承自spring-boot-dependencies POM。
- 更合理的资源过滤。
- 更合理的插件配置（exec plugin，surefire，Git commit ID，shade）。
- 针对application.properties和application.yml的更合理的资源过滤，包括特定的文件（例如application-foo.properties和application-foo.yml）
- 最后一点：由于默认的配置文件接受Spring样式占位符（${...}），Maven过滤更改为使用 @..@ 占位符（您可以使用Maven属性resource.delimiter覆盖它）。

13.2.1 继承启动器parent

要将项目配置为继承spring-boot-starter-parent，只需设置标签如下：

```
<!-- Inherit defaults from Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.2.RELEASE</version>
</parent>
```

> 您只需要在此依赖项上指定Spring Boot版本号。 如果您导入其他起始器，则可以放心地省略他们的版本号。

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

```
<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 使用没有父POM的 Spring Boot

不是每个人都喜欢从spring-boot-starter-parent POM继承。 您公司可能有自己标准的父母，或者您可能只希望明确声明所有的Maven配置。

如果您不想使用spring-boot-starter-parent，则仍然可以通过使用scope=import依赖来保持依赖管理（但不能进行插件管理）的好处：

```
<dependencyManagement>
     <dependencies>
        <dependency>
            <!-- Import dependency management from Spring Boot -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.5.2.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

该设置不允许您使用如13.2.1 所述的属性来覆盖单个依赖关系。 要实现相同的结果，您需要在spring-boot-dependencies条目之前在项目的dependencyManagement中添加一个条目。 例如，要升级到另一个Spring Data发行版本，您需要将以下内容添加到您的pom.xml中。

```
<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>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.5.2.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

> 在上面的例子中，我们指定了一个BOM，但是任何依赖关系类型都可以被这样覆盖。

13.2.3 更改Java版本

spring-boot-starter-parent选择相当保守的Java兼容性版本。 如果要遵循我们的建议并使用更高版本的Java版本，可以添加java.version属性：

```
<properties>
    <java.version>1.8</java.version>
</properties>
```

13.2.4 使用Spring Boot Maven插件

Spring Boot包括一个Maven插件，可以将项目打包成可执行jar。 如果要使用它，请将插件添加到部分：

```
<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 parent pom，则只需要添加这个插件，除非您要更改parent中定义的设置，否则不需要进行配置。

13.3 Gradle

Gradle用户可以直接在其依赖关系部分导入“启动器”。 不像Maven，没有“超级父”导入来共享一些配置。

```
repositories {
    jcenter()
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.5.2.RELEASE")
}
```

spring-boot-gradle-plugin也是可用的，它提供了从源代码创建可执行jar并运行项目的任务。 它还提供依赖关系管理，除其他功能外，还允许您省略由Spring Boot管理的任何依赖关系的版本号：

```
 plugins {
     id 'org.springframework.boot' version '1.5.2.RELEASE'
     id 'java'
 }


 repositories {
     jcenter()
 }
```

```
10
11  dependencies {
12      compile("org.springframework.boot:spring-boot-starter-web")
13      testCompile("org.springframework.boot:spring-boot-starter-test")
14  }
```

13.4 Ant

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。

要声明依赖关系，典型的ivy.xml文件将如下所示：

```
1  <ivy-module version="2.0">
2      <info organisation="org.springframework.boot" module="spring-boot-sample-ant" />
3      <configurations>
4          <conf name="compile" description="everything needed to compile this module" />
5          <conf name="runtime" extends="compile" description="everything needed to run this
6      </configurations>
7      <dependencies>
8          <dependency org="org.springframework.boot" name="spring-boot-starter"
9              rev="${spring-boot.version}" conf="compile" />
10     </dependencies>
11 </ivy-module>
```

典型的build.xml将如下所示：

```
1  <project
2      xmlns:ivy="antlib:org.apache.ivy.ant"
3      xmlns:spring-boot="antlib:org.springframework.boot.ant"
4      name="myapp" default="build">
5
6      <property name="spring-boot.version" value="1.3.0.BUILD-SNAPSHOT" />
7
8      <target name="resolve" description="--> retrieve dependencies with ivy">
9          <ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[revision].[ext]" />
10     </target>
11
12     <target name="classpaths" depends="resolve">
13         <path id="compile.classpath">
14             <fileset dir="lib/compile" includes="*.jar" />
15         </path>
16     </target>
17
18     <target name="init" depends="classpaths">
19         <mkdir dir="build/classes" />
20     </target>
```

```
21
22      <target name="compile" depends="init" description="compile">
23          <javac srcdir="src/main/java" destdir="build/classes" classpathref="compile.class
24      </target>
25
26      <target name="build" depends="compile">
27          <spring-boot:exejar destfile="build/myapp.jar" classes="build/classes">
28              <spring-boot:lib>
29                  <fileset dir="lib/runtime" />
30              </spring-boot:lib>
31          </spring-boot:exejar>
32      </target>
33  </project>
```

> 请参见第84.10节“从Ant构建可执行存档，而不使用spring-boot-antlib”如果不想使用spring-boot-antlib模块，请参阅“操作方法”。

13.5 启动器

启动器是一组方便的依赖关系描述符，可以包含在应用程序中。 您可以获得所需的所有Spring和相关技术的一站式服务，无需通过示例代码搜索和复制粘贴依赖配置。 例如，如果要开始使用Spring和JPA进行数据库访问，那么只需在项目中包含spring-boot-starter-data-jpa依赖关系即可。

启动器包含许多依赖关系，包括您需要使项目快速启动并运行，并具有一致的受支持的依赖传递关系。

What’s in a name

所有正式起动器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。 这个命名结构旨在帮助你快速找到一个启动器。 许多IDE中的Maven插件允许您按名称搜索依赖项。 例如，安装Eclipse或STS的Maven插件后，您可以简单地在POM编辑器中点击 Dependency Hierarchy，并在filter输入“spring-boot-starter”来获取完整的列表。
如创建自己的启动器部分所述，第三方启动程序不应该从Spring-boot开始，因为它是为正式的Spring Boot artifacts 保留的。 acme 的 第三方启动器通常被命名为acme-spring-boot-starter。

Spring Boot在org.springframework.boot组下提供了以下应用程序启动器：

表13.1. Spring Boot应用程序启动器

名称	描述	Pom
spring-boot-starter-thymeleaf	使用Thymeleaf视图构建MVC Web应用程序的启动器	Pom

名称	描述	Pom
spring-boot-starter-data-couchbase	使用Couchbase面向文档的数据库和Spring Data Couchbase的启动器	Pom
spring-boot-starter-artemis	使用Apache Artemis的JMS启动器	Pom
spring-boot-starter-web-services	Spring Web Services 启动器	Pom
spring-boot-starter-mail	Java Mail和Spring Framework的电子邮件发送支持的启动器	Pom
spring-boot-starter-data-redis	Redis key-value 数据存储与Spring Data Redis和Jedis客户端启动器	Pom
spring-boot-starter-web	使用Spring MVC构建Web，包括RESTful应用程序。使用Tomcat作为默认的嵌入式容器的启动器	Pom
spring-boot-starter-data-gemfire	使用GemFire分布式数据存储和Spring Data GemFire的启动器	Pom
spring-boot-starter-activemq	使用Apache ActiveMQ的JMS启动器	Pom
spring-boot-starter-data-elasticsearch	使用Elasticsearch搜索和分析引擎和Spring Data Elasticsearch的启动器	Pom
spring-boot-starter-integration	Spring Integration 启动器	Pom
spring-boot-starter-test	使用JUnit，Hamcrest和Mockito的库测试Spring Boot应用程序的启动器	Pom

名称	描述	Pom
spring-boot-starter-jdbc	使用JDBC与Tomcat JDBC连接池的启动器	Pom
spring-boot-starter-mobile	使用Spring Mobile构建Web应用程序的启动器	Pom
spring-boot-starter-validation	使用Java Bean Validation 与Hibernate Validator的启动器	Pom
spring-boot-starter-hateoas	使用Spring MVC和Spring HATEOAS构建基于超媒体的RESTful Web应用程序的启动器	Pom
spring-boot-starter-jersey	使用JAX-RS和Jersey构建RESTful Web应用程序的启动器。spring-boot-starter-web的替代方案	Pom
spring-boot-starter-data-neo4j	使用Neo4j图数据库和Spring Data Neo4j的启动器	Pom
spring-boot-starter-data-ldap	使用Spring Data LDAP的启动器	Pom
spring-boot-starter-websocket	使用Spring Framework的WebSocket支持构建WebSocket应用程序的启动器	Pom
spring-boot-starter-aop	使用Spring AOP和AspectJ进行面向切面编程的启动器	Pom
spring-boot-starter-amqp	使用Spring AMQP和Rabbit MQ的启动器	Pom
spring-boot-starter-data-cassandra	使用Cassandra分布式数据库和Spring Data Cassandra的启动器	Pom
spring-boot-starter-social-facebook	使用Spring Social Facebook 的启动器	Pom

名称	描述	Pom
spring-boot-starter-jta-atomikos	使用Atomikos的JTA事务的启动器	Pom
spring-boot-starter-security	使用Spring Security的启动器	Pom
spring-boot-starter-mustache	使用Mustache视图构建MVC Web应用程序的启动器	Pom
spring-boot-starter-data-jpa	使用Spring数据JPA与Hibernate的启动器	Pom
spring-boot-starter	核心启动器，包括自动配置支持，日志记录和YAML	Pom
spring-boot-starter-groovy-templates	使用Groovy模板视图构建MVC Web应用程序的启动器	Pom
spring-boot-starter-freemarker	使用FreeMarker视图构建MVC Web应用程序的启动器	Pom
spring-boot-starter-batch	使用Spring Batch的启动器	Pom
spring-boot-starter-social-linkedin	使用Spring Social LinkedIn的启动器	Pom
spring-boot-starter-cache	使用Spring Framework缓存支持的启动器	Pom
spring-boot-starter-data-solr	使用Apache Solr搜索平台与Spring Data Solr的启动器	Pom
spring-boot-starter-data-mongodb	使用MongoDB面向文档的数据库和Spring Data MongoDB的启动器	Pom
spring-boot-starter-jooq	使用jOOQ访问SQL数据库的启动器。 spring-boot-starter-data-jpa或spring-boot-starter-jdbc的替代方案	Pom

名称	描述	Pom
spring-boot-starter-jta-narayana	Spring Boot Narayana JTA 启动器	Pom
spring-boot-starter-cloud-connectors	使用Spring Cloud连接器，简化了与Cloud Foundry和Heroku等云平台中的服务连接的启动器	Pom
spring-boot-starter-jta-bitronix	使用Bitronix进行JTA 事务的启动器	Pom
spring-boot-starter-social-twitter	使用Spring Social Twitter的启动器	Pom
spring-boot-starter-data-rest	通过使用Spring Data REST在REST上暴露Spring数据库的启动器	Pom

除了应用程序启动器，以下启动器可用于添加生产准备(production ready)功能：

表13.2 Spring Boot生产环境启动器

名称	描述	Pom
spring-boot-starter-actuator	使用Spring Boot Actuator提供生产准备功能，可帮助您监控和管理应用程序的启动器	Pom
spring-boot-starter-remote-shell	使用CRaSH远程shell通过SSH监视和管理您的应用程序的启动器。 自1.5以来已弃用	Pom

最后，Spring Boot还包括一些启动器，如果要排除或替换特定的技术，可以使用它们：

名称	描述	Pom

名称	描述	Pom
spring-boot-starter-undertow	使用Undertow作为嵌入式servlet容器的启动器。 spring-boot-starter-tomcat的替代方案	Pom
spring-boot-starter-jetty	使用Jetty作为嵌入式servlet容器的启动器。 spring-boot-starter-tomcat的替代方案	Pom
spring-boot-starter-logging	使用Logback进行日志记录的启动器。 默认的日志启动器	Pom
spring-boot-starter-tomcat	使用Tomcat作为嵌入式servlet容器的启动器。 spring-boot-starter-web的默认servlet容器启动器	Pom
spring-boot-starter-log4j2	使用Log4j2进行日志记录的启动器。 spring-boot-start-logging的替代方法	Pom

> 有关社区贡献的更多启动器的列表，请参阅GitHub上的spring-boot-startters模块中的README文件。

14. 构建代码

Spring Boot不需要任何特定的代码组织结构，但是有一些最佳实践可以帮助您。

14.1 不要使用“default”包

当类不包括包声明时，它被认为是在“默认包”中。 通常不鼓励使用“默认包”，并应该避免使用。 对于使用@ComponentScan，@EntityScan或@SpringBootApplication注解的Spring Boot应用程序，可能会有一些特殊的问题，因为每个jar的每个类都将被读取。

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

14.2 查找主应用程序类

我们通常建议您将应用程序主类放到其他类之上的根包(root package)中。 @EnableAutoConfiguration注解通常放置在您的主类上，它隐式定义了某些项目的基本“搜索包”。 例如，如果您正在编写JPA应用程序，

则@EnableAutoConfiguration注解类的包将用于搜索@Entity项。

使用根包(root package)还可以使用@ComponentScan注释，而不需要指定basePackage属性。 如果您的主类在根包中，也可以使用@SpringBootApplication注释。

这是一个典型的布局：

```
com
 +- example
     +- myproject
         +- Application.java
         |
         +- domain
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
         |
         +- web
             +- CustomerController.java
```

Application.java文件将声明main方法以及基本的@Configuration。

```
package com.example.myproject;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {

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

}
```

15. 配置类

Spring Boot支持基于Java的配置。虽然可以使用XML配置用SpringApplication.run()，但我们通常建议您的主source是@Configuration类。 通常，定义main方法的类也是作为主要的@Configuration一个很好的选

择。

> 许多使用XML配置的Spring示例已经在网上发布。 如果可能的话我们建议始终尝试使用等效的基于Java的配置。 搜索 enable* 注解可以是一个很好的起点。

15.1 导入其他配置类

您不需要将所有的@Configuration放在一个类中。 @Import注解可用于导入其他配置类。 或者，您可以使用@ComponentScan自动扫描所有Spring组件，包括@Configuration类。

15.2 导入XML配置

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

16. 自动配置

Spring Boot 会根据您添加的jar依赖关系自动配置您的Spring应用程序。例如，如果HSQLDB在您的类路径上，并且您没有手动配置任何数据库连接bean，那么我们将自动配置内存数据库。

您需要通过将@EnableAutoConfiguration或@SpringBootApplication注解添加到您的一个@Configuration类中来选择自动配置。

> 您应该只添加一个@EnableAutoConfiguration注解。 我们通常建议您将其添加到主@Configuration类中。

16.1逐渐取代自动配置

自动配置是非侵入式的，您可以随时定义自己的配置来替换自动配置。 例如，如果您添加自己的 DataSource bean，则默认的嵌入式数据库支持将会退回。

如果您需要了解当前有哪些自动配置，以及为什么，请使用–debug开关启动应用程序。 这将启用debug日志，并将自动配置日志记录到控制台。

16.2 禁用指定的自动配置

如果您发现正在使用一些不需要的自动配置类，可以使用@EnableAutoConfiguration的exclude属性来禁用它们。

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

```
4
5  @Configuration
6  @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
7  public class MyConfiguration {
8  }
```

如果类不在classpath路径上，则可以使用注释的excludeName属性，并指定全限定名(fully qualified name)。 最后，您还可以通过spring.autoconfigure.exclude属性控制要排除的自动配置类列表。

> 注解和使用属性(property)定义都可以指定要排除的自动配置类。

17. Spring Beans 和 依赖注入

您可以自由使用任何标准的Spring Framework技术来定义您的bean及其依赖注入关系。 为了简单起见，我们发现使用@ComponentScan搜索bean，结合@Autowired构造函数(constructor)注入效果很好。

如果您按照上述建议（将应用程序类放在根包(root package)中）构建代码，则可以使用@ComponentScan而不使用任何参数。 所有应用程序组件（@Component，@Service，@Repository，@Controller等）将自动注册为Spring Bean。

以下是一个@Service Bean的例子，我们可以使用构造函数注入获取RiskAssessor bean。

```
1  package com.example.service;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.stereotype.Service;
5
6  @Service
7  public class DatabaseAccountService implements AccountService {
8
9      private final RiskAssessor riskAssessor;
10
11     @Autowired
12     public DatabaseAccountService(RiskAssessor riskAssessor) {
13         this.riskAssessor = riskAssessor;
14     }
15
16     // ...
17
18 }
```

如果一个bean 只有一个构造函数，则可以省略@Autowired。

```
1  @Service
2  public class DatabaseAccountService implements AccountService {
```

```
3
4       private final RiskAssessor riskAssessor;
5
6       public DatabaseAccountService(RiskAssessor riskAssessor) {
7           this.riskAssessor = riskAssessor;
8       }
9
10      // ...
11
12  }
```

> 注意，如何使用构造函数注入允许将RiskAssessor字段标记为final，表示不能更改。

18. 使用@SpringBootApplication注解

许多Spring Boot开发人员总是使用@Configuration，@EnableAutoConfiguration和@ComponentScan来标注它们的主类。 由于这些注解经常一起使用（特别是如果您遵循之前说的最佳实践），Spring Boot提供了一个方便的@SpringBootApplication注解作为这三个的替代方法。

@SpringBootApplication注解相当于使用@Configuration，@EnableAutoConfiguration和@ComponentScan和他们的默认属性：

```
1   package com.example.myproject;
2
3   import org.springframework.boot.SpringApplication;
4   import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6   @SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
7   public class Application {
8
9       public static void main(String[] args) {
10          SpringApplication.run(Application.class, args);
11      }
12
13  }
```

> @SpringBootApplication还提供了别名来定制@EnableAutoConfiguration和@ComponentScan的属性。

19. 运行你的应用程序

将应用程序打包成jar并使用嵌入式HTTP服务器的最大优点之一就是可以按照你想用其他任何方式运行应用程序。调试Spring Boot应用程序也很容易; 您不需要任何专门的IDE插件或扩展。

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

19.1 从IDE运行

您可以从IDE中运行 Spring Boot 应用程序作为一个简单的Java应用程序，但是首先需要导入项目。 导入步骤将根据您的IDE和构建系统而有所不同。 大多数IDE可以直接导入Maven项目，例如Eclipse用户可以从File菜单中选择import...→Existing Maven Projects。

如果您无法将项目直接导入到IDE中，则可以使用构建插件生成IDE元数据。 Maven包括Eclipse和IDEA的插件; Gradle为各种IDE提供插件。

> 如果您不小心运行了两次Web应用程序，您将看到“Port already in use”中的错误。 使用STS用户可以使用重新启动按钮而不是运行以确保任何现有实例已关闭。

19.2 作为已打包应用程序运行

如果您使用Spring Boot 的 Maven或Gradle插件创建可执行jar，则可以使用java -jar运行应用程序。 例如：

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

也可以启用远程调试支持运行打包的应用程序。 这允许您将调试器添加到打包的应用程序中：

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

19.3 使用 Maven 插件

Spring Boot Maven 插件包含一个运行目标(goal)，可用于快速编译和运行应用程序。 应用程序以exploded的形式运行，就像在IDE中一样。

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

您可能还需要使用一些有用的操作系统环境变量：

```
$ export MAVEN_OPTS=-Xmx1024m -XX:MaxPermSize=128M
```

19.4 使用Gradle插件

Spring Boot Gradlet插件还包括一个bootRun任务，可用于以exploded 形式运行应用程序。 每当导入spring-boot-gradle-plugin时，都会添加bootRun任务：

```
$ gradle bootRun
```

您可能还想使用这个有用的操作系统环境变量：

```
$ export JAVA_OPTS=-Xmx1024m -XX:MaxPermSize=128M
```

19.5 热插拔

由于Spring Boot应用程序只是纯Java应用程序，所以JVM热插拔应该是开箱即用的。 JVM热插拔在一定程度上受到可替代的字节码的限制，更完整的解决方案，可以使用 JRebel 或者 Spring Loaded 项目。 spring-boot-devtools模块还支持快速重新启动应用程序。

有关详细信息，请参阅第20章“开发人员工具”部分和热插拔“操作方法”。

20. 开发工具

Spring Boot包括一组额外的工具，可以使应用程序开发体验更加愉快。 spring-boot-devtools模块可以包含在任何项目中，以提供额外的开发时功能。 要包含devtools支持，只需将模块依赖关系添加到您的构建中：

Maven：

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

Gradle：

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

> 当运行完全打包的应用程序时，开发人员工具将自动禁用。 如果您的应用程序是使用java -jar启动的，或者是使用特殊的类加载器启动，那么它将会被认为是“生产环境的应用程序”。 将开发工具依赖关系标记为可选(optional)是一种最佳做法，可以防止使用项目将devtools传递性地应用于其他模块。 Gradle不支持开箱即用的可选依赖项，因此您可能希望在此期间查看propdeps-plugin。
>
> 重新打包的jar包默认情况下不包含devtools。 如果要使用某些远程devtools功能，您需要禁用excludeDevtools 构建下的属性以包含devtools。 该属性支持Maven和Gradle插件。

20.1 属性默认值

Spring Boots支持的几个库使用缓存来提高性能。 例如，模板引擎将缓存编译的模板，以避免重复解析模板文件。 此外，Spring MVC可以在返回静态资源时向响应中添加HTTP缓存头。

虽然缓存在生产中非常有益，但它在开发过程中可能会产生反效果，从而阻止您看到刚刚在应用程序中进行的更改。 因此，spring-boot-devtools将默认禁用这些缓存选项。

缓存选项通常由您的application.properties文件中的设置配置。 例如，Thymeleaf提供了spring.thymeleaf.cache属性。 spring-boot-devtools模块不需要手动设置这些属性，而是自动应用更加合理的开发时(development-time)配置。

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

20.2 自动重启

使用spring-boot-devtools的应用程序将在类路径上的文件发生更改时自动重新启动。 这在IDE中开发时可能是一个有用的功能，因为它为代码更改提供了非常快的反馈循环。 默认情况下，将监视指向文件夹的类路径上的任何条目。 请注意，某些资源（如静态资源和视图模板）不需要重新启动应用程序。

触发重启

当DevTools监视类路径资源时，触发重新启动的唯一方法是更新类路径中的文件时。 导致类路径更新的方式取决于您正在使用的IDE。 在Eclipse中，保存修改的文件将导致类路径被更新并触发重新启动。 在IntelliJ IDEA中，构建项目（Build→Make Project）将具有相同的效果。

> 只要forking被启用，您也可以通过支持的构建插件（即Maven和Gradle）启动应用程序，因为DevTools需要一个单独的应用程序类加载器才能正常运行。Gradle和Maven默认情况下在类路径上检DevTools。
>
> 自动重启当与LiveReload一起使用时工作非常好。 详见下文。 如果您使用JRebel，自动重启将被禁用，有利于动态类重新加载。 其他devtools功能仍然可以使用（如LiveReload和属性覆盖）。
>
> DevTools依赖于应用程序上下文的关闭钩子，以在重新启动期间关闭它。 如果禁用了关闭挂钩（SpringApplication.setRegisterShutdownHook（false）），DevTools将无法正常工作。
>
> 当判断类路径中的项目是否会在更改时触发重新启动时，DevTools会自动忽略名为spring-boot，spring-boot-devtools，spring-boot-autoconfigure，spring-boot-actuator和spring-boot-start的项目。

重新启动(Restart) vs 重新加载(Reload)

Spring Boot提供的重新启动技术使用两个类加载器。 不会改的类（例如，来自第三方的jar）被加载到基类加载器中。 您正在开发的类被加载到重新启动(restart)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，并创建一个新的类加载器。 这种方法意味着应用程序重新启动通常比“冷启动”快得多，因为基类加载器已经可以使用。

如果发现重新启动对应用程序不够快，或遇到类加载问题，您可以考虑来自ZeroTurnaround的JRebel等重新加载技术。 这些工作通过在加载类时重写(rewriting)类，使其更适合重新加载。 Spring Loaded提供了另一个选项，但是它在很多框架上不支持，并且不支持商用。

20.2.1 排除资源

在类路径下，某些资源在更改时不一定需要触发重新启动。 例如，Thymeleaf模板可以直接编辑不需重启。默认情况下，有一些排除项，更改 /META-INF/maven，/META-INF/resources，/resources，/static，/public或/templates中的资源不会触发重新启动，但会触发实时重新加载。
如果要自定义这些排除项，可以使用spring.devtools.restart.exclude属性。 例如，要仅排除 /static和/public，您可以设置：

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

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

20.2.2 监视额外的路径

有时当您对不在类路径中的文件进行更改时，需要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。 您可以使用上述的spring.devtools.restart.exclude属性来控制附加路径下的更改是否会触发完全重新启动或只是实时重新加载。

20.2.3 禁用重启

如果不想使用重新启动功能，可以使用spring.devtools.restart.enabled属性来禁用它。 在大多数情况下，您可以在application.properties中设置此项（这仍将初始化重新启动类加载器，但不会监视文件更改）。

例如，如果您需要完全禁用重新启动支持，因为它在一些特定库中不能正常运行，则需要在调用SpringApplication.run（...）之前设置System属性。 例如：

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

20.2.4 使用触发文件

如果您使用IDE工具编写代码，更改文件，则您可能希望仅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，当您要实际触发重新启动检查时，必须修改它。 更改文件只会触发检查，只有在Devtools检测到它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，也可以通过IDE插件更新。

要使用触发器文件，请使用spring.devtools.restart.trigger-file属性。

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

20.2.5 自定义重新启动类加载器

如上面的 Restart vs Reload 部分所述，重新启动功能是通过使用两个类加载器实现的。 对于大多数应用程序，此方法运行良好，但有时可能会导致类加载问题。

默认情况下，IDE中的任何打开的项目将使用“重新启动”类加载器加载，任何常规.jar文件将使用“base”类加载器加载。 如果您在多模块项目上工作，而不是每个模块都导入到IDE中，则可能需要自定义事件。 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restart.exclude 和 restart.include.prefixed属性。 include元素是应该被拉入“重新启动(restart)”类加载器的项目，排除元素是应该向下推入“基本(base)”类加载器的项目。 属性的值是将应用于类路径的正则表达式模式。

例如：

```
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。 只要一个属性从restart.include. 或restart.exclude. 开始，将被考虑。
>
> 将加载类路径中的所有META-INF/spring-devtools.properties。 您可以在项目中打包文件，或者在项目所使用的库中打包文件。

20.2.6 已知的限制

重新启动功能对于使用标准ObjectInputStream反序列化的对象无效。 如果需要反序列化数据，可能需要使用Spring的ConfigurableObjectInputStream与Thread.currentThread()。getContextClassLoader()组合使用。

不幸的是，几个第三方库在不考虑上下文类加载器的情况下反序列化。 如果您发现这样的问题，您需要向原始作者请求修复。

20.3 LiveReload

spring-boot-devtools模块包括一个嵌入式LiveReload服务器，可以在资源更改时用于触发浏览器刷新。LiveReload浏览器扩展程序可以从 http://livereload.com 免费获取Chrome，Firefox和Safari的插件。

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。 开始应用程序之前，请确保没有其他LiveReload服务器正在运行。 如果从IDE启动多个应用程序，则只有第一个应用程序将支持LiveReload。

20.4 全局设置

您可以通过向 $HOME 文件夹添加名为.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将适用于您的计算机上使用devtools的所有Spring Boot应用程序。 例如，要配置重新启动以始终使用触发器文件，您可以添加以下内容：

~/.spring-boot-devtools.properties.

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

20.5远程应用

Spring Boot开发工具不仅限于本地开发。 远程运行应用程序时也可以使用多种功能。 远程支持是可选择的，要使其能够确保重新打包的存档中包含devtools：

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

那么你需要设置一个spring.devtools.remote.secret属性，例如：

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

> 在远程应用程序上启用spring-boot-devtools是一种安全隐患。 您不应该在生产部署中启用该支持。

远程devtools支持分为两部分： 有一个接受连接的服务器端和您在IDE中运行的客户端应用程序。 当spring.devtools.remote.secret属性设置时，服务器组件将自动启用。 客户端组件必须手动启动。

20.5.1 运行远程客户端应用程序

远程客户端应用程序旨在从IDE中运行。 您需要使用与要连接的远程项目相同的类路径运行org.springframework.boot.devtools.RemoteSpringApplication。 传递给应用程序的必选参数应该是您要连接到的远程URL。

例如，如果您使用Eclipse或STS，并且有一个名为my-app的项目已部署到Cloud Foundry，则可以执行以下操作：

- 从Run 菜单中选择Run Configurations...。
- 创建一个新的Java Application “launch configuration”。
- 浏览my-app项目。
- 使用org.springframework.boot.devtools.RemoteSpringApplication作为主类。
- 将https://myapp.cfapps.io添加到程序参数（或任何远程URL）中。

运行的远程客户端将如下所示：

```
  .   ____          _                                              __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___               _      \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_ ___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/ -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\___/\__\___|/ / / /
 =========|_|==============|___/=================================/_/_/_/
 :: Spring Boot Remote :: 1.5.2.RELEASE

2015-06-10 18:25:06.632  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringAppl
2015-06-10 18:25:06.671  INFO 14938 --- [           main] s.c.a.AnnotationConfigApplicati
2015-06-10 18:25:07.043  WARN 14938 --- [           main] o.s.b.d.r.c.RemoteClientConfigu
2015-06-10 18:25:07.074  INFO 14938 --- [           main] o.s.b.d.a.OptionalLiveReloadSer
2015-06-10 18:25:07.130  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringAppl
```

> 由于远程客户端正在使用与实际应用程序相同的类路径，因此可以直接读取应用程序属性。 这是spring.devtools.remote.secret属性如何读取并传递到服务器进行身份验证。
>
> 建议使用https//作为连接协议，以便流量被加密，防止密码被拦截。
>
> 如果需要使用代理访问远程应用程序，请配置spring.devtools.remote.proxy.host和spring.devtools.remote.proxy.port属性。

20.5.2 远程更新

远程客户端将以与本地相同的方式监视应用程序类路径的更改。 任何更新的资源将被推送到远程应用程序，并且（如果需要的话）触发重新启动。 如果您正在迭代使用您当地没有的云服务的功能，这可能会非常有用。 通常，远程更新和重新启动比完全重建和部署周期要快得多。

> 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。

20.5.3 远程调试隧道

在远程应用程序诊断问题时，Java远程调试非常有用。 不幸的是，当您的应用程序部署在数据中心之外时，并不总是能够进行远程调试。 如果您正在使用基于容器的技术（如Docker），远程调试也可能难以设置。

为了帮助解决这些限制，devtools支持基于HTTP隧道的传输远程调试传输。 远程客户端在端口8000上提供本地服务器，您可以连接远程调试器。 建立连接后，通过HTTP将调试数据发送到远程应用程序。 如果要使用其他端口，可以使用spring.devtools.remote.debug.local-port属性更改。

您需要确保远程应用程序启用远程调试启用。 通常可以通过配置JAVA_OPTS来实现。 例如，使用Cloud Foundry，您可以将以下内容添加到manifest.yml中：

```
---
    env:
        JAVA_OPTS: "-Xdebug -Xrunjdwp:server=y,transport=dt_socket,suspend=n"
```

> 请注意，您不需要将address=NNNN选项传递给-Xrunjdwp。 如果省略Java将随机选择一个的空闲端口。
>
> 通过网络调试远程服务可能很慢，您可能需要在IDE中增加超时时间。 例如，在Eclipse中，您可以从Preferences...中选择Java→Debug，并将Debugger timeout (ms)更改为更合适的值（大多数情况下，60000可以正常工作）。
>
> 当使用IntelliJ IDEA的远程调试隧道时，必须将所有调试断点配置为挂起线程而不是挂起VM。 默认情况下，IntelliJ IDEA中的断点会挂起整个VM，而不是仅挂起触发断点的线程。 这会导致挂起管理远程调试通道的线程等不必要的副作用，导致调试会话冻结。 当使用IntelliJ IDEA的远程调试隧道时，应将所有断点配置为挂起线程而不是VM。 有关详细信息，请参阅IDEA-165769。

21. 包装您的应用程序到生产环境

可执行的jar可用于生产部署。 由于它们是相互独立的，它们也非常适合基于云的部署。

对于其他“生产环境准备”功能，如健康，审计和metric REST或JMX端点; 考虑添加spring-boot-actuator。 有关详细信息，请参见第V部分“Spring Boot Actuator：生产环境准备功能”。

22. 接下来应该读什么

您现在应该很好地了解如何使用Spring Boot以及您应该遵循的一些最佳做法。 您现在可以深入了解特定的Spring Boot功能，或者您可以跳过这部分，真的阅读Spring Boot的“生产环境准备”方面。

Part IV. Spring Boot 功能

本节将会介绍Spring Boot的一些细节。 在这里，您可以了解您将要使用和自定义的主要功能。 如果还没有准备好，您可能需要阅读第二部分“入门指南”和第三部分“使用 Spring Boot”部分，以使您有基础的良好基础。

23. SpringApplication

SpringApplication类提供了一种方便的方法来引导将从main()方法启动的Spring应用程序。 在许多情况下，您只需委派静态SpringApplication.run()方法：

```
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内容：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v1.5.2.RELEASE

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplication
2013-07-31 00:08:16.166  INFO 56603 --- [           main] ationConfigEmbeddedWebApplicati
2014-03-04 13:09:54.912  INFO 41370 --- [           main] .t.TomcatEmbeddedServletContair
2014-03-04 13:09:56.501  INFO 41370 --- [           main] o.s.b.s.app.SampleApplication
```

默认情况下，将显示INFO 级别log消息，包括用户启动应用程序一些相关的启动细节。

23.1 启动失败

如果您的应用程序无法启动，则注册的FailureAnalyzers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，并且该端口已在使用中，则应该会看到类似于以下内容的内容：

```
***************************
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 applicatior
```

> Spring Boot提供了众多的FailureAnalyzer实现，您可以非常容易地添加自己的实现。

如果没有故障分析器(analyzers)能够处理异常，您仍然可以显示完整的自动配置报告，以更好地了解出现的问题。 为此，您需要启用debug属性或启用org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer的DEBUG日志。

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

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

23.2 自定义Banner

可以通过在您的类路径中添加一个 banner.txt 文件，或者将banner.location设置到banner文件的位置来更改启动时打印的banner。 如果文件有一些不常用的编码，你可以设置banner.charset（默认为UTF-8）。除了文本文件，您还可以将banner.gif，banner.jpg或banner.png图像文件添加到您的类路径中，或者设置一个banner.image.location属性。 图像将被转换成ASCII艺术表现，并打印在任何文字banner上方。

您可以在banner.txt文件中使用以下占位符：

表23.1. banner变量

变量名	描述
${application.version}	在MANIFEST.MF中声明的应用程序的版本号。例如， Implementation-Version: 1.0 被打印为 1.0.
${application.formatted-version}	在MANIFEST.MF中声明的应用程序版本号的格式化显示（用括号括起来，以v为前缀）。 例如 (v1.0)。

变量名	描述
${spring-boot.version}	您正在使用的Spring Boot版本。 例如1.5.2.RELEASE。
${spring-boot.formatted-version}	您正在使用格式化显示的Spring Boot版本（用括号括起来，以v为前缀）。 例如（v1.5.2.RELEASE）。
[Math Processing Error]{AnsiColor.NAME}, *[Math Processing Error]*{AnsiStyle.NAME})	其中NAME是ANSI转义码的名称。 有关详细信息，请参阅 AnsiPropertySource。
${application.title}	您的应用程序的标题在MANIFEST.MF中声明。 例如Implementation-Title：MyApp打印为MyApp。

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

您还可以使用spring.main.banner-mode属性来决定是否必须在System.out（控制台）上打印banner，使用配置的logger（log）或不打印（off）。

23.3 定制SpringApplication

如果SpringApplication默认值不符合您的想法，您可以创建本地实例并进行自定义。 例如，关闭banner：

```
1   public static void main(String[] args) {
2       SpringApplication app = new SpringApplication(MySpringConfiguration.class);
3       app.setBannerMode(Banner.Mode.OFF);
4       app.run(args);
5   }
```

> 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，这些将引用@Configuration类，但它们也可以引用XML配置或应扫描的包。

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

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

23.4 流式构建 API

如果您需要构建一个ApplicationContext层次结构（具有父/子关系的多个上下文），或者如果您只想使用“流式（fluent）”构建器API，则可以使用SpringApplicationBuilder。

SpringApplicationBuilder允许您链式调用多个方法，并包括允许您创建层次结构的父和子方法。

例如：

```
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，并且相同的环境将用于父和子上下文。 有关详细信息，请参阅SpringApplicationBuilder Javadoc。

23.5 Application events and listeners

除了常见的Spring Framework事件（如 ContextRefreshedEvent）之外，SpringApplication还会发送一些其他应用程序事件。

> 在创建ApplicationContext之前，实际上触发了一些事件，因此您不能在@Bean上注册一个监听器。 您可以通过SpringApplication.addListeners(...) 或SpringApplicationBuilder.listeners(...)方法注册它们。
>
> 如果您希望自动注册这些侦听器，无论创建应用程序的方式如何，都可以将META-INF / spring.factories文件添加到项目中，并使用org.springframework.context.ApplicationListener引用您的侦听器。
> org.springframework.context.ApplicationListener=com.example.project.MyListener

当您的应用程序运行时，事件按照以下顺序发送：

1. ApplicationStartingEvent在运行开始时发送，但在注册侦听器和注册初始化器之后。
2. 当已经知道要使用的上下文(context)环境，并在context创建之前，将发送ApplicationEnvironmentPreparedEvent。
3. ApplicationPreparedEvent在启动刷新(refresh)之前发送，但在加载了bean定义之后。
4. ApplicationReadyEvent在刷新之后被发送，并且处理了任何相关的回调以指示应用程序准备好服务请求。
5. 如果启动时发生异常，则发送ApplicationFailedEvent。

> 一般您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。

23.6 Web 环境

SpringApplication将尝试代表您创建正确类型的ApplicationContext。 默认情况下，将使用AnnotationConfigApplicationContext或AnnotationConfigEmbeddedWebApplicationContext，具体取决于您是否正在开发Web应用程序。

用于确定“Web环境”的算法是相当简单的（基于几个类的存在）。 如果需要覆盖默认值，可以使用setWebEnvironment（boolean webEnvironment）。

也可以通过调用setApplicationContextClass() 对ApplicationContext完全控制。

> 在JUnit测试中使用SpringApplication时，通常需要调用setWebEnvironment()

23.7 访问应用程序参数

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

```
1  import org.springframework.boot.*
2  import org.springframework.beans.factory.annotation.*
3  import org.springframework.stereotype.*
4
5  @Component
6  public class MyBean {
7
8      @Autowired
9      public MyBean(ApplicationArguments args) {
10         boolean debug = args.containsOption("debug");
11         List<String> files = args.getNonOptionArgs();
12         // if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]
13     }
14
15 }
```

> Spring Boot还将向Spring Environment 注册一个CommandLinePropertySource。 这允许您也使用@Value注解注入应用程序参数。

23.8 使用ApplicationRunner或CommandLineRunner

SpringApplication启动时如果您需要运行一些特定的代码，就可以实现ApplicationRunner或CommandLineRunner接口。 两个接口都以相同的方式工作，并提供一个单独的运行方法，这将在SpringApplication.run（...）完成之前调用。

CommandLineRunner接口提供对应用程序参数的访问（简单的字符串数组），而ApplicationRunner使用上述的ApplicationArguments接口。

```
@Component
public class MyBean implements CommandLineRunner {

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

}
```

如果定义了若干CommandLineRunner或ApplicationRunner bean，这些bean必须按特定顺序调用，您可以实现org.springframework.core.Ordered接口，也可以使用org.springframework.core.annotation.Order注解。

23.9 Application exit

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

另外，如果希望在应用程序结束时返回特定的退出代码，那么bean可以实现org.springframework.boot.ExitCodeGenerator接口。

23.10 管理功能

可以通过指定spring.application.admin.enabled属性来为应用程序启用与管理相关的功能。 这会在平台MBeanServer上暴露SpringApplicationAdminMXBean。 您可以使用此功能来远程管理您的Spring Boot应用程序。 这对于任何服务包装器(service wrapper)实现也是有用的。

> 如果您想知道应用程序在哪个HTTP端口上运行，请使用local.server.port键获取该属性。
>
> 启用此功能时请小心，因为MBean公开了关闭应用程序的方法。

24. 外部配置

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

Spring Boot使用非常特别的PropertySource命令，旨在允许合理地覆盖值。属性按以下顺序选择：

1. 在您的HOME目录设置的Devtools全局属性（~/.spring-boot-devtools.properties）。
2. 单元测试中的 @TestPropertySource 注解。
3. 单元测试中的 @SpringBootTest#properties 注解属性
4. 命令行参数。
5. SPRING_APPLICATION_JSON 中的属性值（内嵌JSON嵌入到环境变量或系统属性中）。
6. ServletConfig 初始化参数。
7. ServletContext 初始化参数。
8. 来自 java:comp/env 的JNDI属性。
9. Java系统属性（System.getProperties()）。
10. 操作系统环境变量。
11. RandomValuePropertySource，只有随机的属性 random.* 中。
12. jar包外面的 Profile-specific application properties （application- {profile} .properties和YAML变体）
13. jar包内的 Profile-specific application properties （application-{profile}.properties和YAML变体）
14. jar包外的应用属性文件（application.properties和YAML变体）。
15. jar包内的应用属性文件（application.properties和YAML变体）。
16. 在@Configuration上的@PropertySource注解。
17. 默认属性（使用SpringApplication.setDefaultProperties设置）。

一个具体的例子，假设你开发一个使用name属性的@Component：

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

@Component
public class MyBean {

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

    // ...

}
```

在应用程序类路径（例如，您的jar中）中，您可以拥有一个application.properties，它为 name 属性提供了默认属性值。 在新环境中运行时，可以在您的jar外部提供一个application.properties来覆盖 name 属性; 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”）。

SPRING_APPLICATION_JSON属性可以在命令行中提供一个环境变量。 例如在UN*X shell中：

```
$ SPRING_APPLICATION_JSON='{"foo":{"bar":"spam"}}' java -jar myapp.jar
```

在本例中，您将在Spring环境中使用foo.bar = spam。 您也可以在系统变量中将JSON作为spring.application.json提供：

```
$ java -Dspring.application.json='{"foo":"bar"}' -jar myapp.jar
```

或命令行参数：

```
$ java -jar myapp.jar --spring.application.json='{"foo":"bar"}'
```

或作为JNDI变量 java:comp/env/spring.application.json 。

24.1 配置随机值

RandomValuePropertySource可用于注入随机值（例如，进入秘密或测试用例）。 它可以产生整数，长整数，uuid或字符串，例如

```
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是任何字符和值，max是整数。 如果提供max，则值为最小值，max为最大值（独占）。

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（以’－ ‘开头，例如–server.port=9000）转换为属性，并将其添加到Spring环境中。 如上所述，命令行属性始终优先于其他属性来源。

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

24.3 应用程序属性文件

SpringApplication将从以下位置的application.properties文件中加载属性，并将它们添加到Spring Environment中：

1. 当前目录的/config子目录
2. 当前目录

3. classpath中/config包
4. classpath root路径

该列表按优先级从高到低排序。

> 也可以使用YAML（’.yml’）文件替代“.properties”。

如果您不喜欢application.properties作为配置文件名，可以通过指定一个spring.config.name Spring environment属性来切换到另一个。 您还可以使用spring.config.location环境属性（用逗号分隔的目录位置列表或文件路径）显式引用位置。

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

或

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

> spring.config.name和spring.config.location一开始就被用于确定哪些文件必须被加载，因此必须将它们定义为环境属性（通常是OS env，system属性或命令行参数）。

如果spring.config.location包含的如果是目录而非文件，那么它们应该以/结尾（并将在加载之前附加从spring.config.name生成的名称，包括profile-specific的文件名）。 在spring.config.location中指定的文件按原样使用，不支持特定于配置文件的变体，并且将被任何特定于配置文件的属性覆盖。

默认的搜索路径 classpath:,classpath:/config,file:,file:config/ 始终会被搜索，不管spring.config.location的值如何。 该搜索路径从优先级排序从低到高（file:config/最高）。 如果您指定自己的位置，则它们优先于所有默认位置，并使用相同的从最低到最高优先级排序。 这样，您可以在application.properties（或使用spring.config.name选择的任何其他基础名称）中为应用程序设置默认值，并在运行时使用不同的文件覆盖它，并保留默认值。

> 如果您使用环境(environment)变量而不是系统属性，大多数操作系统不允许使用句点分隔(period-separated)的键名称，但可以使用下划线（例如，SPRING_CONFIG_NAME，而不是spring.config.name）
>
> 如果您运行在容器中，则可以使用JNDI属性（在 java:comp/env 中）或servlet上下文初始化参数，而不是环境变量或系统属性。

24.4 指定配置(Profile-specific)的属性

除了application.properties文件外，还可以使用命名约定application- {profile}.properties定义的指定配置文件。 环境具有一组默认配置文件，如果没有设置活动配置文件（即，如果没有显式激活配置文件，则加载了来自application-default.properties的属性）。

指定配置文件（Profile-specific）的属性从与标准application.properties相同的位置加载，指定配置(profile-specific)文件始终覆盖非指定文件，而不管指定配置文件是否在打包的jar内部或外部。

如果有几个指定配置文件，则应用最后一个配置。 例如，由spring.profiles.active属性指定的配置文件在通过SpringApplication API配置的配置之后添加，因此优先级高。

> 如果您在spring.config.location中指定了任何文件，则不会考虑这些特定配置(profile-specific)文件的变体。 如果您还想使用指定配置(profile-specific)文件的属性，请使用 `spring.config.location` 中的目录。

24.5 properties 文件中的占位符

application.properties中的值在使用时通过已有的环境进行过滤，以便您可以引用之前定义的值（例如，从系统属性）。

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

> 您也可以使用此技术创建现有Spring Boot属性的“简写“。 有关详细信息，请参见第72.4节“使用”短命令行参数“how-to”。

24.6 使用YAML替代 Properties

YAML是JSON的超集，因此这是分层配置数据一种非常方便的格式，。 每当您的类路径中都有SnakeYAML库时，SpringApplication类将自动支持YAML作为 properties 的替代方法。

> 如果您使用“Starters”， SnakeYAML将通过spring-boot-starter自动提供。

24.6.1 加载 YAML

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

例如，下面YAML文档：

```
environments:
    dev:
        url: http://dev.bar.com
```

```
4           name: Developer Setup
5       prod:
6           url: http://foo.bar.com
7           name: My Cool App
```

将转化为属性：

```
1   environments.dev.url=http://dev.bar.com
2   environments.dev.name=Developer Setup
3   environments.prod.url=http://foo.bar.com
4   environments.prod.name=My Cool App
```

YAML列表表示为具有[index] dereferencers的属性键，例如YAML：

```
1   my:
2      servers:
3          - dev.bar.com
4          - foo.bar.com
```

将转化为属性：

```
1   my.servers[0]=dev.bar.com
2   my.servers[1]=foo.bar.com
```

要使用Spring DataBinder工具（@ConfigurationProperties做的）绑定到这样的属性，您需要有一个属性类型为java.util.List（或Set）的目标bean，并且您需要提供一个setter，或者 用可变值初始化它，例如 这将绑定到上面的属性

```
1   @ConfigurationProperties(prefix="my")
2   public class Config {
3
4       private List<String> servers = new ArrayList<String>();
5
6       public List<String> getServers() {
7           return this.servers;
8       }
9   }
```

24.6.2 将YAML作为Spring环境中的属性文件

可以使用YamlPropertySourceLoader类在Spring环境中将YAML作为PropertySource暴露出来。 这允许您使用熟悉的@Value注解和占位符语法来访问YAML属性。

24.6.3 多个YAML文件

您可以使用 spring.profiles 键指定单个文件中的多个特定配置文件YAML文档，以指示文档何时应用。 例如：

```
server:
    address: 192.168.1.100
---
spring:
    profiles: development
server:
    address: 127.0.0.1
---
spring:
    profiles: production
server:
    address: 192.168.1.120
```

在上面的示例中，如果开发配置文件处于活动状态，则server.address属性将为127.0.0.1。 如果开发和生产配置文件未启用，则该属性的值将为192.168.1.100。

如果应用程序上下文启动时没有显式激活，默认配置文件将被激活。 所以在这个YAML中，我们为security.user.password设置一个仅在“默认”配置文件中可用的值：

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

使用“spring.profiles”元素指定的Spring profiles 以选择使用！ 字符。 如果为单个文档指定了否定和非否定的配置文件，则至少有一个非否定配置文件必须匹配，没有否定配置文件可能匹配。

24.6.4 YAML的缺点

YAML文件无法通过@PropertySource注解加载。 因此，在需要以这种方式加载值的情况下，需要使用properties文件。

24.6.5 合并YAML列表

如上所述，任何YAML内容最终都会转换为属性。 当通过配置文件覆盖“列表”属性时，该过程可能比较直观。

例如，假设名称和描述属性默认为空的MyPojo对象。 让我们从FooProperties中公开MyPojo的列表：

```
@ConfigurationProperties("foo")
public class FooProperties {

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

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

}
```

类比以下配置：

```
foo:
  list:
    - name: my name
      description: my description
---
spring:
  profiles: dev
foo:
  list:
    - name: my another name
```

如果dev配置没有激活，FooProperties.list将包含一个如上定义的MyPojo条目。 如果启用了配置文件，列表仍将包含一个条目（名称为“my another name”，description=null）。 此配置不会将第二个MyPojo实例添加到列表中，并且不会将项目合并。

当在多个配置文件中指定集合时，使用具有最高优先级的集合（并且仅使用该配置文件）：

```
foo:
  list:
    - name: my name
      description: my description
    - name: another name
      description: another description
---
spring:
  profiles: dev
foo:
  list:
     - name: my another name
```

在上面的示例中，考虑到dev配置文件处于激活状态，FooProperties.list将包含一个MyPojo条目（名称为“my another name”和description=null）。

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可能很麻烦，特别是如果您正在使用多个层次结构的属性或数据时。 Spring Boot提供了一种处理属性的替代方法，允许强类型Bean管理并验证应用程序的配置。

```
1  package com.example;
2
3  import java.net.InetAddress;
4  import java.util.ArrayList;
5  import java.util.Collections;
6  import java.util.List;
7
8  import org.springframework.boot.context.properties.ConfigurationProperties;
9
10 @ConfigurationProperties("foo")
11 public class FooProperties {
12
13     private boolean enabled;
14
15     private InetAddress remoteAddress;
16
17     private final Security security = new Security();
18
19     public boolean isEnabled() { ... }
20
21     public void setEnabled(boolean enabled) { ... }
22
23     public InetAddress getRemoteAddress() { ... }
24
25     public void setRemoteAddress(InetAddress remoteAddress) { ... }
26
27     public Security getSecurity() { ... }
28
29     public static class Security {
30
31         private String username;
32
33         private String password;
34
35         private List<String> roles = new ArrayList<>(Collections.singleton("USER"));
36
37         public String getUsername() { ... }
38
39         public void setUsername(String username) { ... }
40
41         public String getPassword() { ... }
```

```
42
43          public void setPassword(String password) { ... }
44
45          public List<String> getRoles() { ... }
46
47          public void setRoles(List<String> roles) { ... }
48
49      }
50  }
```

上述POJO定义了以下属性：

- foo.enabled，默认为false
- foo.remote-address，具有可以从String强转的类型
- foo.security.username，具有内置的“安全性(security)”，其名称由属性名称决定。 特别是返回类型并没有被使用，可能是SecurityProperties
- foo.security.password
- foo.security.roles，一个String集合

Getters和setter方法通常是必须要有的，因为绑定是通过标准的Java Beans属性描述符，就像在Spring MVC中一样。 在某些情况下可能会省略setter方法：

- Map 只要它们被初始化，需要一个getter，但不一定是一个setter，因为它们可以被binder修改。
- 集合和数组可以通过索引（通常使用YAML）或使用单个逗号分隔值（Properties中）来访问。 在后一种情况下，setter方法是强制性的。 我们建议总是为这样的类型添加一个设置器。 如果您初始化集合，请确保它不是不可变的（如上例所示）
- 如果已初始化嵌套POJO属性（如上例中的Security字段），则不需要setter方法。如果您希望binder使用其默认构造函数即时创建实例，则需要一个setter。

有些人使用Project Lombok自动添加getter和setter。 确保Lombok不会为这种类型生成任何特定的构造函数，因为构造函将被容器自动用于实例化对象。

> 另请参阅@Value和@ConfigurationProperties之间的不同。

您还需要列出在 `@EnableConfigurationProperties` 注解中注册的属性类：

```
1  @Configuration
2  @EnableConfigurationProperties(FooProperties.class)
3  public class MyConfiguration {
4  }
```

> 当 `@ConfigurationProperties` bean以这种方式注册时，该bean将具有常规名称： `<prefix>` - `<fqn>`，其中是@ConfigurationProperties注解中指定的环境密钥前缀，是bean的全名(fully qualified

> name)。 如果注解不提供任何前缀，则仅使用该bean的全名。上面示例中的bean名称将是foo-com.example.FooProperties。

即使上述配置将为FooProperties创建一个常规bean，我们建议@ConfigurationProperties仅处理环境，特别是不从上下文中注入其他bean。 话虽如此，@EnableConfigurationProperties注释也会自动应用于您的项目，以便使用@ConfigurationProperties注释的任何现有的bean都将从环境配置。 您可以通过确保FooProperties已经是一个bean来快速上面的MyConfiguration

```
@Component
@ConfigurationProperties(prefix="foo")
public class FooProperties {

    // ... see above

}
```

这种配置方式与SpringApplication外部的YAML配置相当：

```
# application.yml

foo:
    remote-address: 192.168.1.1
    security:
        username: foo
        roles:
          - USER
          - ADMIN

# additional configuration as required
```

要使用@ConfigurationProperties bean，您可以像其他任何bean一样注入它们。

```
@Service
public class MyService {

    private final FooProperties properties;

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

     //...

    @PostConstruct
    public void openConnection() {
```

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

}
```

> 使用@ConfigurationProperties还可以生成IDE可以为自己的密钥提供自动完成的元数据文件，有关详细信息，请参见附录B，配置元数据附录。

24.7.1第三方配置

除了使用@ConfigurationProperties来注解类，还可以在public @Bean方法中使用它。 当您希望将属性绑定到不受控制的第三方组件时，这可能特别有用。

```
@ConfigurationProperties(prefix = "bar")
@Bean
public BarComponent barComponent() {
    ...
}
```

使用 bar 前缀定义的任何属性将以与上述FooProperties示例类似的方式映射到该BarComponent bean。

24.7.2 宽松的绑定

Spring Boot使用一些宽松的规则将环境属性绑定到@ConfigurationProperties bean，因此不需要在Environment属性名称和bean属性名称之间进行完全匹配。 常用的例子是这样有用的：虚分离（例如上下文路径绑定到contextPath）和大写（例如PORT绑定到端口）环境属性。

例如，给定以下@ConfigurationProperties类：

```
@ConfigurationProperties(prefix="person")
public class OwnerProperties {

    private String firstName;

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

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

}
```

可以使用以下属性名称：

表格 24.1. relaxed binding

Property	Note
person.firstName	标准骆峰命名法。
person.first-name	虚线符号，推荐用于.properties和.yml文件。
person.first_name	下划线符号，用于.properties和.yml文件的替代格式。
PERSON_FIRST_NAME	大写格式 推荐使用系统环境变量时。

24.7.3属性转换

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

> 由于在应用程序生命周期期间非常早请求此Bean，请确保限制ConversionService正在使用的依赖关系。通常，您需要的任何依赖关系可能无法在创建时完全初始化。 如果配置密钥强制不需要，只需依赖使用@ConfigurationPropertiesBinding限定的自定义转换器，就可以重命名自定义ConversionService。

24.7.4 @ConfigurationProperties验证

当Spring的@Validated注释解时，Spring Boot将尝试验证@ConfigurationProperties类。 您可以直接在配置类上使用JSR-303 javax.validation约束注释。 只需确保您的类路径中符合JSR-303实现，然后在您的字段中添加约束注释：

```
1  @ConfigurationProperties(prefix="foo")
2  @Validated
3  public class FooProperties {
4
5      @NotNull
6      private InetAddress remoteAddress;
7
8      // ... getters and setters
9
10 }
```

为了验证嵌套属性的值，您必须将关联字段注释为@Valid以触发其验证。 例如，基于上述FooProperties示例：

```
@ConfigurationProperties(prefix="connection")
@Validated
public class FooProperties {

    @NotNull
    private InetAddress remoteAddress;

    @Valid
    private final Security security = new Security();

    // ... getters and setters

    public static class Security {

        @NotEmpty
        public String username;

        // ... getters and setters

    }

}
```

您还可以通过创建名为configurationPropertiesValidator的bean定义来添加自定义的Spring Validator。 @Bean方法应声明为static。 配置属性验证器在应用程序的生命周期早期创建，并声明@Bean方法，因为static允许创建bean，而无需实例化@Configuration类。 这避免了早期实例化可能引起的任何问题。 这里有一个属性验证的例子，所以你可以看到如何设置。

> spring-boot-actuator 模块包括一个暴露所有@ConfigurationProperties bean的端点。 只需将您的Web浏览器指向/configprops 或使用等效的JMX端点。 请参阅生产就绪功能 细节。

24.7.5 @ConfigurationProperties 对比 @Value

@Value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：

功能	@ConfigurationProperties	@Value
Relaxed binding	Yes	No
Meta-data support	Yes	No

功能	@ConfigurationProperties	@Value
SpEL evaluation	No	Yes

如果您为自己的组件定义了一组配置密钥，我们建议您将其分组到使用@ConfigurationProperties注释的POJO中。 还请注意，由于@Value不支持宽松的绑定，如果您需要使用环境变量提供值，那么它不是一个很好的选择。

最后，当您可以在@Value中编写一个Spel表达式时，这些表达式不会从应用程序属性文件中处理。

25. 配置文件(Profiles)

Spring 配置文件提供了将应用程序配置隔离的方法，使其仅在某些环境中可用。 任何@Component或@Configuration都可以使用@Profile进行标记，以限制其在什么时候加载：

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

    // ...

}
```

一般，您可以使用spring.profiles.active Environment属性来指定哪些配置文件处于激活状态。 您可以以任何方式指定属性，例如，您可以将其包含在您的application.properties中：

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

或者使用命令行 `--spring.profiles.active=dev,hsqldb` 在命令行中指定。

25.1添加激活配置文件

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

有时，将特定于配置文件的属性添加到激活的配置文件而不是替换它们是有用的。 spring.profiles.include属性可用于无条件添加激活配置文件。 SpringApplication入口点还具有用于设置其他配置文件的Java API（即，在由spring.profiles.active属性激活的那些配置文件之上）：请参阅setAdditionalProfiles()方法。

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

```
---
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。 有关详细信息，请参见第72.7节“根据环境更改配置”。

25.2 以编程方式设置配置文件

您可以通过在应用程序运行之前调用SpringApplication.setAdditionalProfiles(...)以编程方式设置激活配置文件。 也可以使用Spring的ConfigurableEnvironment接口激活配置文件。

25.3 配置文件指定的配置文件

通过@ConfigurationProperties引用的application.properties（或application.yml）和文件的配置文件特定变体都被视为加载文件。 有关详细信息，请参见第24.4节指定配置(Profile-specific)的属性”。

26. 日志

Spring Boot使用Commons Logging进行所有内部日志记录，但使基础日志实现开放。 默认配置提供了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和可选文件输出都可用。

默认情况下，如果使用’Starters’，将会使用Logback。 还包括适当的Logback路由，以确保使用Java Util Logging，Commons Logging，Log4J或SLF4J的依赖库都能正常工作。

> 有很多可用于Java的日志记录框架。 如果上面的列表看起来很混乱，别担心。 一般来说，您不需要更改日志依赖关系，并且Spring Boot默认值将正常工作。

26.1 日志格式

Spring Boot的默认日志输出如下所示：

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main] org.apache.catalina.core.Standar
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBea
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrat
```

输出以下项目：

- 日期和时间 - 毫秒精度并且容易排序。
- 日志级别 - ERROR, WARN, INFO, DEBUG, TRACE.
- 进程ID。
- —分隔符来区分实际日志消息的开始。
- 线程名称 - 括在方括号中（可能会截断控制台输出）。
- 记录器名称 - 这通常是源类名（通常缩写）。
- 日志消息。

> Logback没有FATAL级别（对映ERROR）

26.2 控制台输出

默认的日志配置会在控制台显示消息。 默认情况下会记录ERROR，WARN和INFO级别的消息。 您还可以通过–debug启动您的应用程序来启用“debug”模式。

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

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

当启用debug模式时，配置核心记录器（嵌入式容器，Hibernate和Spring Boot）的选择可以输出更多信息。 启用debug 模式不会将应用程序配置为使用DEBUG级别记录所有消息。

或者，您可以使用–trace启动应用程序（或在您的application.properties中为trace=true）启用“trace”模式。 这将为核心记录器（嵌入式容器，Hibernate模式生成和整个Spring组合）启用trace日志。

26.2.1 日志颜色输出

如果您的终端支持ANSI，颜色输出可以增加可读性。 您可以将spring.output.ansi.enabled设置为支持的值来覆盖自动检测。

使用%clr关键字配置颜色编码。 在最简单的形式下，转换器将根据日志级别对输出进行着色，例如：

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

日志级别映射到颜色如下：

- blue
- cyan

- faint
- green
- magenta
- red
- yellow

26.3 文件输出

默认情况下，Spring Boot将仅将日志输出到控制台，不会写到文件。 如果要将控制台上的日志输出到日志文件，则需要设置logging.file或logging.path属性（例如在application.properties中）。

下表显示了如何一起使用logging.*属性：

表26.1 Logging属性

logging.file	logging.path	Example	Description
(none)	(none)		仅控制台输出
Specific file	(none)	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。
(none)	Specific directory	/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。

日志文件将在10 MB时滚动输出到文件，默认情况下会记录控制台输出，ERROR，WARN和INFO级别的消息。

> 日志记录系统在应用程序生命周期早期初始化，并且在通过@PropertySource注解加载的属性文件中将不会找到log属性。
>
> 日志属性独立于实际的日志记录基础结构。 因此，特定配置key（如Logback的logback.configurationFile）不受Spring Boot管理。

26.4 日志级别

所有支持的日志记录系统都可以在Spring Environment 中设置log级别（例如在application.properties中），使用‘logging.level.*=LEVEL’，其中‘LEVEL’是TRACE，DEBUG，INFO，WARN，ERROR，FATAL, OFF之一。 可以使用logging.level.root配置根记录器。 示例application.properties：

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

> 默认情况下，Spring Boot会重新启动Thymeleaf INFO消息，以便它们以DEBUG级别进行记录。 这有助于降低标准日志输出中的噪音。 有关如何在自己的配置中应用重映射的详细信息，请参阅LevelRemappingAppender。

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系统，并通过在类路径的根目录中提供合适的配置文件，或在Spring Environment属性logging.config指定的位置进一步配置。

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实现的全名。 您还可以使用 none 值完全禁用Spring Boot的日志记录配置。

> 由于在创建ApplicationContext之前初始化日志，因此无法在Spring @Configuration文件中控制@PropertySources的日志记录。 系统属性和常规的Spring Boot外部配置文件工作正常。

根据您的日志记录系统，将会加载以下文件：

Logging System	Customization
Logback	logback-spring.xml, logback-spring.groovy, logback.xml or logback.groovy
Log4j2	log4j2-spring.xml or log4j2.xml
JDK (Java Util Logging)	logging.properties

> 如果可能，我们建议您使用-spring变体进行日志记录配置（例如使用logback-spring.xml而不是logback.xml）。 如果使用标准配置位置，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，从“可执行jar”运行时会导致问题。 我们建议您尽可能避免。

帮助定制一些其他属性从Spring环境转移到系统属性：

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	如果定义了，则用于默认日志配置。
logging.path	LOG_PATH	如果定义了，则用于默认日志配置。
logging.pattern.console	CONSOLE_LOG_PATTERN	在控制台上使用的日志模式（stdout）。（仅支持默认logback设置。）
logging.pattern.file	FILE_LOG_PATTERN	在文件中使用的日志模式（如果LOG_FILE已启用）。（仅支持默认logback设置。）
logging.pattern.level	LOG_LEVEL_PATTERN	用于呈现日志级别的格式（默认%5p）。（仅支持默认logback设置。）
PID	PID	当前进程ID（如果可能的话，当未被定义为OS环境变量时被发现）。

支持的所有日志记录系统在分析其配置文件时可以查看系统属性。 有关示例，请参阅spring-boot.jar中的默认配置。

> 如果要在logging属性中使用占位符，则应使用Spring Boot的语法，而不是底层框架的语法。 值得注意的是，如果您使用Logback，您应该使用：作为属性名称与其默认值之间的分隔符，而不是 :- 。
>
> 您可以通过覆盖LOG_LEVEL_PATTERN（或Logback的log.pattern.level）来添加MDC和其他ad-hoc内容到日志行。 例如，如果使用logging.pattern.level = user：%X {user}%5p，则默认日志格式将包含“user”的MDC条目（如果存在）。

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

26.6 Logback扩展

Spring Boot包括大量的Logback扩展，可以帮助您进行高级配置。 您可以在logback-spring.xml配置文件中使用这些扩展。

> 您不能在标准logback.xml配置文件中使用扩展名，因为其加载时间太早。 您需要使用logback-spring.xml或定义logging.config属性。
>
> 扩展名不能与Logback的配置扫描一起使用。 如果您尝试这样做，对配置文件进行更改将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 [spring
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [spring
```

26.6.1 指定配置文件配置

`<springProfile>` 标签允许您根据活动的Spring配置文件可选地包含或排除配置部分。 配置文件部分支持 `<configuration>` 元素在任何位置。 使用name属性指定哪个配置文件接受配置。 可以使用逗号分隔列表指定多个配置文件。

```
<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 环境属性

标签允许您从Spring环境中显示属性，以便在Logback中使用。 如果您在logback中访问application.properties文件中的值，这将非常有用。 标签的工作方式与Logback标准的标签类似，但不是指定直接值，而是指定属性的来源（来自Environment）。 如果需要将属性存储在本地范围以外的位置，则可以使用scope属性。 如果在环境中未设置属性的情况下需要备用值，则可以使用defaultValue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>
```

RelaxedPropertyResolver用于访问Environment属性。 如果以虚线表示法（my-property-name）指定源，则将尝试所有宽松的变体（myPropertyName，MY_PROPERTY_NAME等）。

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或Undertow轻松创建自包含的HTTP服务器。 大多数Web应用程序将使用spring-boot-starter-web模块快速启动和运行。

如果您尚未开发Spring Boot Web应用程序，则可以按照“Hello World！”示例进行操作。 在“入门”部分中的示例。

27.1 “Spring Web MVC框架”

Spring Web MVC框架（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将使用@RequestMapping注释映射到HTTP。

以下是@RestController用于提供JSON数据的典型示例：

```
@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Spring Framework的一部分，详细信息可在参考文档中找到。 Spring.io/guide中还有几个指南可供Spring MVC使用。

27.1.1 Spring MVC自动配置

Spring Boot提供了适用于大多数应用程序的Spring MVC的自动配置。

自动配置在Spring的默认值之上添加以下功能：

- 包含ContentNegotiatingViewResolver和BeanNameViewResolver bean。
- 支持提供静态资源，包括对WebJars的支持（见下文）。
- Converter，GenericConverter，Formatter beans的自动注册。
- 支持HttpMessageConverters（见下文）。
- 自动注册MessageCodesResolver（见下文）。
- 静态index.html支持。
- 自定义Favicon支持（见下文）。
- 自动使用ConfigurableWebBindingInitializer bean（见下文）。

如果要保留Spring Boot MVC功能，并且您只需要添加其他MVC配置（interceptors, formatters, view, controllers等），你可以添加自己的WebConfigurerAdapter类型的@Configuration类，但不能使用@EnableWebMvc。 如果要提供自定义的RequestMappingHandlerMapping，RequestMappingHandlerAdapter或ExceptionHandlerExceptionResolver实例，您可以声明一个提供此类组件的WebMvcRegistrationsAdapter实例。

如果要完全控制Spring MVC，可以使用@EnableWebMvc添加您自己的@Configuration注释。

27.1.2 HttpMessageConverters

Spring MVC使用HttpMessageConverter接口转换HTTP请求和响应。 包括一些开箱即用的合理配置，例如对象可以自动转换为JSON（使用Jackson库）或XML（使用Jackson XML扩展，如果可用，否则使用JAXB）。 字符串默认使用UTF-8进行编码。

如果需要添加或自定义转换器，可以使用Spring Boot HttpMessageConverter类：

```
1  import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
2  import org.springframework.context.annotation.*;
3  import org.springframework.http.converter.*;
4
5  @Configuration
6  public class MyConfiguration {
7
8      @Bean
9      public HttpMessageConverters customConverters() {
```

```
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new HttpMessageConverters(additional, another);
    }

}
```

上下文中存在的任何HttpMessageConverter bean将被添加到转换器列表中。 您也可以以这种方式覆盖默认转换器。

27.1.3 自定义JSON序列化器和反序列化器

如果您使用Jackson序列化和反序列化JSON数据，则可能需要编写自己的JsonSerializer和JsonDeserializer类。 自定义序列化程序通常通过一个模块注册到Jackson，但是Spring Boot提供了一个备用的@JsonComponent注释，可以更容易地直接注册Spring Bean。

您可以直接在JsonSerializer或JsonDeserializer实现中使用@JsonComponent。 您也可以将它用于包含序列化器/解串器的类作为内部类。 例如：

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

@JsonComponent
public class Example {

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

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

}
```

ApplicationContext中的所有@JsonComponent bean将自动注册到Jackson，并且由于@JsonComponent是使用@Component进行元注解的，所以常规的组件扫描规则适用。

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

27.1.4 MessageCodesResolver

Spring MVC有一个生成错误代码的策略，用于从绑定错误中提取错误消息：MessageCodesResolver。Spring Boot将为您创建一个错误代码，如果您设置spring.mvc.message-codes-resolver.format属性PREFIX_ERROR_CODE或POSTFIX_ERROR_CODE（请参阅DefaultMessageCodesResolver.Format中的枚举）。

27.1.5 静态内容

默认情况下，Spring Boot将从类路径或ServletContext的根目录中的名为/static（或/ public或/resources或/META-INF/resources）的目录提供静态内容。 它使用Spring MVC中的ResourceHttpRequestHandler，因此您可以通过添加自己的WebMvcConfigurerAdapter并覆盖addResourceHandlers方法来修改该行为。

在独立的Web应用程序中，来自容器的默认servlet也被启用，并且作为后备，如果Spring决定不处理它，则从ServletContext的根目录提供内容。 大多数情况下，这不会发生（除非您修改默认的MVC配置），因为Spring将始终能够通过DispatcherServlet处理请求。

默认情况下，资源映射到/ **，但可以通过spring.mvc.static-path-pattern调整。 例如，将所有资源重定位到/resources/**可以配置如下：

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

您还可以使用spring.resources.static-locations（使用目录位置列表替换默认值）来自定义静态资源位置。如果这样做，默认欢迎页面检测将切换到您的自定义位置，因此，如果在启动时任何位置都有一个index.html，它将是应用程序的主页。

除了上述“标准”静态资源位置之外，还提供了一个特殊情况，用于Webjars内容。 任何具有/ webjars / **中路径的资源都将从jar文件中提供，如果它们以Webjars格式打包。

如果您的应用程序将被打包为jar，请不要使用 src/main/webapp 目录。 虽然这个目录是一个通用的标准，但它只适用于war包，如果生成一个jar，它将被大多数构建工具忽略。

Spring Boot还支持Spring MVC提供的高级资源处理功能，允许使用例如缓存静态资源或使用Webjars的版本无关的URL。

要为Webjars使用版本无关的URL，只需添加webjars-locator依赖关系即可。然后声明您的Webjar，以jQuery为例，如“/webjars/jquery/dist/jquery.min.js”，这将产生“/webjars/jquery/xyz/dist/jquery.min.js”，其中xyz是Webjar版本 。

> 如果您使用JBoss，则需要声明webjars-locator-jboss-vfs依赖关系而不是webjars-locator; 否则所有Webjars都将解析为404。

要使用缓存清除功能，以下配置将为所有静态资源配置缓存清除解决方案，从而有效地在URL中添加内容哈希值，例如：

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

> 链接资源在运行时在模板中被重写，这归功于自动配置为Thymeleaf和FreeMarker的ResourceUrlEncodingFilter。 使用JSP时，应手动声明此过滤器。 其他模板引擎现在不会自动支持，但可以使用自定义模板宏/帮助程序和使用ResourceUrlProvider。

当使用例如JavaScript模块加载器动态加载资源时，重命名文件不是一个选项。这就是为什么其他策略也得到支持并可以合并的原因。 “固定(fixed)”策略将在URL中添加静态版本字符串，而不更改文件名：

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

使用此配置，位于“/js/lib/”下的JavaScript模块将使用固定版本策略“/v12/js/lib/mymodule.js”，而其他资源仍将使用内容。

有关更多支持的选项，请参阅ResourceProperties。

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

27.1.6 自定义图标

Spring Boot在配置的静态内容位置和类路径的根目录（按顺序）中查找favicon.ico。 如果文件存在，它将被自动用作应用程序的图标。

27.1.7 ConfigurableWebBindingInitializer

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

27.1.8 模板引擎

除了REST Web服务，您还可以使用Spring MVC来提供动态HTML内容。 Spring MVC支持各种模板技术，包括Thymeleaf，FreeMarker和JSP。 许多其他模板引擎也运行自己的Spring MVC集成。

Spring Boot包括对以下模板引擎的自动配置支持：

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

> 如果可能，应避免使用JSP，当使用嵌入式servlet容器时，JSP有几个已知的限制。

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 自动获取。

> IntelliJ IDEA根据运行应用程序的方式对类路径进行不同的排序。通过main方法在IDE中运行应用程序将导致使用Maven或Gradle打包的jar运行应用程序时的不同顺序。这可能会导致Spring Boot找不到类路径上的模板。如果您受此问题的影响，您可以重新排序IDE中的类路径，以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模板目录：`classpath*:/templates/`。

27.1.9 错误处理

默认情况下，Spring Boot提供 /error 映射，以合理的方式处理所有错误，并在servlet容器中注册为“global”错误页面。对于机器客户端，它将产生JSON响应，其中包含错误，HTTP状态和异常消息的详细信息。对于浏览器客户端，有一个‘whitelabel’错误视图，以HTML格式呈现相同的数据（定制它只需添加一个解析“error”的视图）。要完全替换默认行为，您可以实现ErrorController并注册该类型的bean定义，或者简单地添加一个类型为ErrorAttributes的bean来使用现有机制，但只是替换内容。

> BasicErrorController可以用作自定义ErrorController的基类。如果要添加新内容类型的处理程序（默认情况下是专门处理text/html并为其他内容提供备选），这一点尤其有用。要做到这一点，只需扩展BasicErrorController并添加一个带有@RequestMapping的公共方法，并创建一个新类型的bean。

您还可以定义一个@ControllerAdvice来自定义为特定控制器 and/or 异常类型返回的JSON文档。

```
@ControllerAdvice(basePackageClasses = FooController.class)
public class FooControllerAdvice 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()),
    }

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

```
17      }
18
19  }
```

在上面的示例中，如果由FooController在同一个包中定义的控件抛出了YourException，则将使用CustomerErrorType POJO的json表示法而不是ErrorAttributes表示形式。

自定义错误页面

如果要显示给定状态代码的自定义HTML错误页面，请将文件添加到/error文件夹。 错误页面可以是静态HTML（即添加在任何静态资源文件夹下）或使用模板构建。 该文件的名称应该是确切的状态代码或一个序列掩码。

例如，要将404映射到静态HTML文件，您的文件夹结构将如下所示：

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

要使用FreeMarker模板映射所有5xx错误，使用如下结构：

```
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。

```
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，如@ExceptionHandler方法和@ControllerAdvice。 然后，ErrorController将接收任何未处理的异常。

映射Spring MVC之外的错误页面

对于不使用Spring MVC的应用程序，可以使用ErrorPageRegistrar接口来直接注册ErrorPages。这个抽象直接与底层的嵌入式servlet容器一起工作，即使没有Spring MVC DispatcherServlet也可以工作。

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

// ...

private static class MyErrorPageRegistrar implements ErrorPageRegistrar {

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

}
```

N.B. 如果您注册一个最终由Filter过滤的路径的ErrorPage（例如，像一些非Spring Web框架，例如Jersey和Wicket一样），则必须将Filter显式注册为ERROR dispatcher，例如。

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

（默认的FilterRegistrationBean不包括ERROR dispatcher 类型）。

WebSphere Application Server上的错误处理

当部署到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相应的错误页面。 如果响应尚未提交，则该请求只能转发到正确的错误页面。 默认情况下，WebSphere Application Server 8.0及更高版本在成功完成servlet的服务方法后提交响应。 您应该通过将com.ibm.ws.webcontainer.invokeFlushAfterService设置为false来禁用此行为

27.1.10 Spring HATEOAS

如果您正在开发一种利用超媒体的RESTful API，Spring Boot可以为Spring HATEOAS提供自动配置，适用于大多数应用程序。 自动配置取代了使用@EnableHypermediaSupport的需求，并注册了一些Bean，以便轻松构建基于超媒体的应用程序，包括LinkDiscoverers（用于客户端支持）和配置为将响应正确地组织到所需表示中的ObjectMapper。 ObjectMapper将根据spring.jackson。*属性或Jackson2ObjectMapperBuilder bean（如果存在）进行自定义。

您可以使用@EnableHypermediaSupport控制Spring HATEOAS配置。 请注意，这将禁用上述ObjectMapper定制。

27.1.11 CORS 支持

跨原始资源共享（CORS）是大多数浏览器实现的W3C规范，允许您以灵活的方式指定什么样的跨域请求被授权，而不是使用一些不太安全和不太强大的方法，如IFRAME或JSONP。

从版本4.2起，Spring MVC支持CORS开箱即用。 在Spring Boot应用程序中的controller方法使用@CrossOrigin注解的CORS配置不需要任何特定的配置。 可以通过使用自定义的addCorsMappings(CorsRegistry)方法注册WebMvcConfigurer bean来定义全局CORS配置：

```
@Configuration
public class MyConfiguration {

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

27.2 JAX-RS 和 Jersey

如果您喜欢JAX-RS编程模型的REST endpoints ，您可以使用一个可用的实现而不是Spring MVC。 如果您刚刚在应用程序上下文中注册了一个@Bean的Servlet或Filter，那么Jersey 1.x和Apache CXF的功能非常出色。

Jersey2.x有一些本地Spring支持，所以我们也提供自动配置支持它在Spring Boot与启动器。

要开始使用Jersey 2.x，只需将spring-boot-starter-jersey作为依赖项，然后您需要一个@Bean类型ResourceConfig，您可以在其中注册所有端点(endpoints)：

```
@Component
public class JerseyConfig extends ResourceConfig {

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

}
```

> Jersey对扫描可执行档案的包是相当有限的。 例如，当运行可执行的war文件时，它无法扫描在WEB-INF/classes中找到的包中的端点(endpoints)。 为了避免这种限制，不应使 packages 方法，并且应使用上述寄存器方法单独注册(register)端点。

您还可以注册任意数量的ResourceConfigCustomizer的实现bean，以实现更高级的自定义。

所有注册的端点都应为具有HTTP资源注解（@GET等）的@Components，例如。

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

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

}
```

由于Endpoint是一个Spring @Component，所以Spring的生命周期由Spring管理，您可以使用@Autowired依赖关系并使用@Value注入外部配置。 默认情况下，Jersey servlet将被注册并映射到/ *。 您可以通过将@ApplicationPath添加到ResourceConfig来更改映射。

默认情况下，Jersey将通过@Bean以名为jerseyServletRegistration的ServletRegistrationBean类型在Servlet进行设置。 默认情况下，servlet将被初始化，但是您可以使用spring.jersey.servlet.load-on-startup进行自定义。您可以通过创建一个自己的同名文件来禁用或覆盖该bean。 您也可以通过设置spring.jersey.type = filter（在这种情况下，@Bean来替换或替换为jerseyFilterRegistration），使用Filter而不是Servlet。 servlet有一个@Order，您可以使用spring.jersey.filter.order设置。 可以使用spring.jersey.init.* 给出Servlet和过滤器注册的init参数，以指定属性的映射。

有一个Jersey示例，所以你可以看到如何设置。 还有一个Jersey1.x示例。 请注意，在Jersey1.x示例中，spring-boot maven插件已经被配置为打开一些Jersey jar，以便它们可以被JAX-RS实现扫描（因为示例要求它们在Filter注册中进行扫描）。 如果您的任何JAX-RS资源作为嵌套的jar打包，您可能需要执行相同操作。

27.3 嵌入式servlet容器支持

Spring Boot包括对嵌入式Tomcat，Jetty和Undertow服务器的支持。 大多数开发人员将简单地使用适当的“Starter”来获取完全配置的实例。 默认情况下，嵌入式服务器将监听端口8080上的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，文件上传等。当您的应用程序正在运行导致故障时，该目录可能会被tmpwatch删除。 为了避免这种情况，您可能需要自定义tmpwatch配置，以便tomcat.*目录不被删除，或配置server.tomcat.basedir，以便嵌入式Tomcat使用不同的位置

27.3.1 Servlets, Filters 和 listeners

当使用嵌入式servlet容器时，可以使用Spring bean或通过扫描Servlet组件（例如HttpSessionListener）注册Servlet规范中的Servlet，过滤器和所有监听器。

将Servlets，过滤器和监听器注册为Spring bean

任何Servlet，Filter或Servlet Listener 实例都会作为Spring bean注册到嵌入式容器中。 可以非常方便地在配置过程中引用您的application.properties中的值。

默认情况下，如果容器中只包含一个Servlet，它将映射到/。 在多个Servlet bean的情况下，bean名称将作为路径前缀。 过滤器(Filters)将映射到/*，默认过滤所有请求。

如果基于惯例的映射不够灵活，可以使用ServletRegistrationBean，FilterRegistrationBean和ServletListenerRegistrationBean类来完成控制。

27.3.2 Servlet Context 初始化

嵌入式servlet容器不会直接执行Servlet 3.0+ javax.servlet.ServletContainerInitializer接口或Spring的org.springframework.web.WebApplicationInitializer接口。 这样设计的目的旨在降低在war中运行的第三方库破坏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 context 初始化，则应注册一个实现org.springframework.boot.context.embedded.ServletContextInitializer接口的bean。 单个onStartup方法提供对ServletContext的访问，并且如果需要，可以轻松地用作现有WebApplicationInitializer的适配器。

扫描Servlet，过滤器和监听器

使用嵌入式容器时，可以使用@ServletComponentScan启用@WebServlet，@WebFilter和@WebListener注解类的自动注册。

> @ServletComponentScan在独立容器中不起作用，在该容器中将使用容器的内置发现机制。

27.3.3 EmbeddedWebApplicationContext

在Spring Boot引导下，将会使用一种新类型的ApplicationContext来支持嵌入式的servlet容器。EmbeddedWebApplicationContext是一种特殊类型的WebApplicationContext，它通过搜索单个EmbeddedServletContainerFactory bean来引导自身。 通常，TomcatEmbeddedServletContainerFactory，JettyEmbeddedServletContainerFactory或UndertowEmbeddedServletContainerFactory将被自动配置。

> 您通常不需要知道这些实现类。 大多数应用程序将被自动配置，并将代表您创建适当的ApplicationContext和EmbeddedServletContainerFactory。

27.3.4 定制嵌入式servlet容器

可以使用Spring Environment属性配置常见的servlet容器设置。 通常您可以在application.properties文件中定义属性。

常用服务器设置包括：

- 网络设置：侦听端口的HTTP请求（server.port），接口地址绑定到server.address等。
- 会话设置：会话是否持久化（server.session.persistence），会话超时（server.session.timeout），会话数据的位置（server.session.store-dir）和session-cookie配置（server.session.cookie.*）。
- 错误管理：错误页面的位置（server.error.path）等
- SSL
- HTTP压缩

Spring Boot尽可能地尝试公开常见设置，但并不总是可能的。 对于这些情况，专用命名空间提供服务器特定的定制（请参阅server.tomcat和server.undertow）。 例如，可以使用嵌入式servlet容器的特定功能来配置访问日志。

> 有关完整列表，请参阅 ServerProperties 类。

用程序定制

如果需要以编程方式配置嵌入式servlet容器，您可以注册一个实现EmbeddedServletContainerCustomizer接口的Spring bean。 EmbeddedServletContainerCustomizer提供对ConfigurableEmbeddedServletContainer的访问，其中包含许多自定义设置方法。

```
import org.springframework.boot.context.embedded.*;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements EmbeddedServletContainerCustomizer {

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.setPort(9000);
    }

}
```

直接自定义ConfigurableEmbeddedServletContainer

如果上述定制技术有太多限制，您可以自己注册TomcatEmbeddedServletContainerFactory，JettyEmbeddedServletContainerFactory或UndertowEmbeddedServletContainerFactory bean。

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

setter方法提供了许多配置选项。 如果您需要做更多的自定义，还会提供几种保护方法“钩子”。 有关详细信息，请参阅源代码文档。

27.3.5 JSP限制

当运行使用嵌入式servlet容器（并打包为可执行文档）的Spring Boot应用程序时，对JSP支持有一些限制。

- 可以使用Tomcat和war包，即可执行的war将会起作用，并且也可以部署到标准容器（不限于但包括Tomcat）中。 由于Tomcat中的硬编码文件模式，可执行的jar将无法正常工作。
- 可以使用Jetty和war包，即可执行的war将会起作用,并且也可以部署到任何标准的容器，它应该可以工作。
- Undertow不支持JSP。
- 创建自定义的error.jsp页面将不会覆盖默认视图以进行错误处理，而应使用自定义错误页面。

28. Security

如果Spring Security位于类路径上，则默认情况下，Web应用程序将在所有HTTP端点上使用“basic”身份验证。 要向Web应用程序添加方法级安全性，您还可以使用所需的设置添加@EnableGlobalMethodSecurity。 有关更多信息，请参见“Spring Security Reference”。

默认的AuthenticationManager有一个用户（用户名’user’和随机密码，在应用程序启动时以INFO级别打印）

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

> 如果您调整日志记录配置，请确保将org.springframework.boot.autoconfigure.security类别设置为记录INFO消息，否则将不会打印默认密码。

您可以通过提供security.user.password来更改密码。 这个和其他有用的属性通过 SecurityProperties（属性前缀“security”）进行外部化。

默认的安全配置在SecurityAutoConfiguration和从那里导入的类中实现（用于Web安全的SpringBootWebSecurityConfiguration和用于认证配置的AuthenticationManagerConfiguration，这在非Web应用程序中也是相关的）。 要完全关闭默认的Web应用程序安全配置，您可以使用@EnableWebSecurity添加一个bean（这不会禁用身份验证管理器配置或Actuator的安全性）。 要定制它，您通常使用WebSecurityConfigurerAdapter类型的外部属性和bean（例如添加基于表单的登录）。 要关闭身份验证管理器配置，您可以添加AuthenticationManager类型的bean，或者通过将AuthenticationManagerBuilder自动连接到您的一个@Configuration类中的方法来配置全局AuthenticationManager。 Spring Boot示例中有几个安全应用程序可以让您开始使用常见的用例。

您在Web应用程序中获得的基本功能包括：

- 具有内存存储和单个用户的AuthenticationManager Bean（请参阅用于用户属性的SecurityProperties.User）。
- 对于常见的静态资源位置，忽略（不安全）路径(/css/**, /js/**, /images/**, /webjars/** and **/favicon.ico)。
- HTTP所有其他端点的baseic security 。
- 安全事件发布到Spring的ApplicationEventPublisher（成功、不成功的身份验证、拒绝访问）。
- 默认情况下，Spring Security提供的常见的底层功能（HSTS，XSS，CSRF，缓存）都是打开的。

所有上述可以使用外部属性（security.*）打开、关闭或修改。 要覆盖访问规则而不更改任何其他自动配置的功能，请添加一个带有@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)的WebSecurityConfigurerAdapter类型的Bean，并配置它以满足您的需要。

> 默认情况下，WebSecurityConfigurerAdapter将匹配任何路径。 如果您不想完全覆盖Spring Boot自动配置的访问规则，您的适配器必须显式配置您要覆盖的路径。

28.1 OAuth2

如果您的类路径中有spring-security-oauth2，您可以利用一些自动配置来轻松设置授权或资源服务器。 有关完整的详细信息，请参阅“Spring Security OAuth 2开发人员指南”。

28.1.1 授权服务器

要创建授权服务器并授予访问令牌，您需要使用@EnableAuthorizationServer并提供security.oauth2.client.client-id和security.oauth2.client.client-secret]属性。 客户端将为您注册在内存中。

```
1  $ curl client:secret@localhost:8080/oauth/token -d grant_type=password -d username=user -d
```

/token 端点的基本身份验证凭证是client-id和client-secret。 用户凭据是普通的Spring Security用户details（在Spring引导中默认为“user”和随机密码）。

要关闭自动配置并自行配置授权服务器功能，只需添加一个类型为AuthorizationServerConfigurer的@Bean。

28.1.2 资源服务器

要使用访问令牌(token)，您需要一个资源服务器（可以与授权服务器相同）。 创建资源服务器很简单，只需添加@EnableResourceServer并提供一些配置，以允许服务器解码访问令牌。 如果您的应用程序也是授权服务器，则它已经知道如何解码令牌，无需做其他事情。 如果你的应用程序是一个独立的服务，那么你需要给它一些更多的配置，以下选项之一：

- security.oauth2.resource.user-info-uri使用/me资源（例如PWS上的https://uaa.run.pivotal.io/userinfo）
- security.oauth2.resource.token-info-uri使用令牌解码端点（例如，PWS上的https://uaa.run.pivotal.io/check_token）。

如果您同时指定user-info-uri和token-info-uri，那么您可以设置一个标志，表示优先于另一个（prefer-token-inf=true是默认值）。

或者（不是user-info-uri或token-info-uri的情况）如果令牌是JWT，您可以配置security.oauth2.resource.jwt.key-value来本地解码（key是验证密钥verification key）。 验证密钥值是对称秘密或PEM编码的RSA公钥。 如果您没有密钥，并且它是公开的，您可以提供一个可以使用security.oauth2.resource.jwt.key-uri下载的URI（具有“value”字段的JSON对象）。 例如在PWS上：

```
$ curl https://uaa.run.pivotal.io/token_key
{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMIIBI...\n-----END PUBLIC KEY-
```

如果您使用security.oauth2.resource.jwt.key-uri，则应用程序启动时需要运行授权服务器。 如果找不到密钥，它将记录一个警告，并告诉您如何解决该问题。

> 如果您使用security.oauth2.resource.jwt.key-uri，则应用程序启动时需要运行授权服务器。 如果找不到密钥，它将会在日志记录一个警告，并告诉您如何解决该问题。

OAuth2资源由order security.oauth2.resource.filter-order的过滤器链保护，默认情况下保护执行器(actuator)端点的过滤器（所以执行器(actuator)端点将保留在HTTP Basic上，除非更改顺序）。

28.2 User Info中的令牌类型

Google和某些其他第三方身份认证提供商对在header中发送到用户信息端点的令牌类型名称更为严格。 默认值为“Bearer”，适合大多数提供程序并匹配规范，但如果需要更改，可以设置security.oauth2.resource.token-type。

28.3 自定义用户信息RestTemplate

如果您有user-info-uri，则资源服务器功能在内部使用OAuth2RestTemplate来获取用户身份验证信息。 这是以UserInfoRestTemplateFactory类型的@Bean提供的。 大多数提供程序的默认值应该是能满足正常使用，但有时您可能需要添加其他拦截器，或者更改请求验证器（例如：令牌如何附加到传出请求）。 进行自定义，只需创建一个类型为UserInfoRestTemplateCustomizer的bean - 它具有一个方法，在bean创建之后但在初始化之前将被调用。 这里定制的rest模板只能在内部进行验证。 或者，您可以定义自己的UserInfoRestTemplateFactory @Bean来完全控制。

要在YAML中设置RSA密钥值，请使用“pipe”继续标记将其分割成多行（ “|” ），并记住缩进键值（它是标准的 YAML 语言功能）。 例：

```
security:
    oauth2:
        resource:
            jwt:
                keyValue: |
                    -----BEGIN PUBLIC KEY-----
                    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC...
                    -----END PUBLIC KEY-----
```

28.3.1 Client

要使您的 web-app 进入OAuth2客户端，您可以简单地添加@ EnableOAuth2Client，Spring Boot将创建一个OAuth2ClientContext和OAuth2ProtectedResourceDetails，这些是创建OAuth2RestOperations所必需的。 Spring Boot不会自动创建这样的bean，但是您可以轻松创建自己的bean：

```
@Bean
public OAuth2RestTemplate oauth2RestTemplate(OAuth2ClientContext oauth2ClientContext,
        OAuth2ProtectedResourceDetails details) {
    return new OAuth2RestTemplate(details, oauth2ClientContext);
}
```

> 您可能需要添加限定符并查看您的配置，因为您的应用程序可能会定义多个RestTemplate。

此配置使用security.oauth2.client.*作为凭据（可能与授权服务器中使用的相同），但另外还需要知道授权服务器中的授权和令牌URI。 例如：

application.yml.

```
security:
    oauth2:
        client:
            clientId: bd1c0a783ccdd1c9b9e4
            clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
            accessTokenUri: https://github.com/login/oauth/access_token
            userAuthorizationUri: https://github.com/login/oauth/authorize
            clientAuthenticationScheme: form
```

当您尝试使用OAuth2RestTemplate时，具有此配置的应用程序将重定向到Github进行授权。 如果您已经登录Github，您甚至不会注意到它已经被认证。 如果您的应用程序在端口8080上运行（在Github或其他提供商注册自己的客户端应用程序以获得更大的灵活性），这些特定的凭据才会起作用。

要限制客户端在获取访问令牌时要求的范围，您可以设置security.oauth2.client.scope（逗号分隔或YAML中的数组）。 默认情况下，scope是空的，由授权服务器决定其默认值，通常取决于客户端注册中的设置。

> 还有一个security.oauth2.client.client-authentication-scheme的设置，默认为“header”（但是如果像Github那样，您可能需要将其设置为“form”，例如，您的OAuth2提供程序不喜欢header 认证）。 事实上，security.oauth2.client.*属性绑定到AuthorizationCodeResourceDetails的一个实例，因此可以指定其所有的属性。
>
> 在非Web应用程序中，您仍然可以创建一个OAuth2RestOperations，它仍然连接到security.oauth2.client.*配置中。 在这种情况下，它是一个“客户端凭据令牌授予”，您如果使用它就请求它（并且不需要使用@EnableOAuth2Client或@EnableOAuth2Sso）。为了防止定义基础设施，只需从配置中删除security.oauth2.client.client-id（或使其成为空字符串）。

28.3.2 单点登录

OAuth2客户端可用于从提供商获取用户详细信息（如果此类功能可用），然后将其转换为Spring Security的身份验证令牌。 以上资源服务器通过user-info-uri属性支持此功能这是基于OAuth2的单点登录（SSO）协议的基础，Spring Boot可以通过提供@ EnableOAuth2Sso注解来轻松加入。 上面的Github客户端可以通过添加该注释并声明在何处查找端点（除了上面列出的security.oauth2.client.*配置）外，还可以保护所有资源并使用Github/user/endpoint进行身份验证：

application.yml.

```
security:
    oauth2:
...
    resource:
        userInfoUri: https://api.github.com/user
        preferTokenInfo: false
```

由于默认情况下所有路径都是安全的，所以没有可以向未经身份验证的用户显示“家”页面，并邀请他们登录（通过访问/登录路径或由security.oauth2.sso.login-path指定的路径）。

由于默认情况下所有路径都是要求安全的，所以没有可以向未经身份验证的用户显示“home”页面，并邀请他们登录（通过访问/login 路径或由security.oauth2.sso.login-path指定的路径）。

要自定义保护的访问规则或路径，因此您可以添加“home”页面，例如，@EnableOAuth2Sso可以添加到WebSecurityConfigurerAdapter，并且注解将使其被修饰和增强，以使所需的/login路径可以工作。 例如，这里我们简单地允许未经身份验证的访问“/“下的主页面，并保留其他所有内容的默认值：

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    public void init(WebSecurity web) {
        web.ignore("/");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/**").authorizeRequests().anyRequest().authenticated();
    }

}
```

28.4 Actuator Security

如果Actuator也在使用中，您会发现：

- 即使应用程序端点不安全，管理端点也是安全的。
- Security 事件将转换为AuditEvent实例，并发布到AuditEventRepository。
- 默认用户将具有ACTUATOR角色以及USER角色。

Actuator的安全功能可以使用外部属性（management.security.*）进行修改。要覆盖应用程序访问规则，请添加一个类型为WebSecurityConfigurerAdapter的@Bean，如果您不想覆盖执行程序访问规则，则使用@Order（SecurityProperties.ACCESS_OVERRIDE_ORDER）或@Order（ManagementServerProperties.ACCESS_OVERRIDE_ORDER）覆盖执行器访问规则。

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访问到完成“对象关系映射”技术，如Hibernate。Spring Data提供了额外的功能，直接从接口创建Repository实现，并使用约定从方法名称生成查询。

29.1 配置DataSource

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，DataSource使用URL和一些凭据来建立数据库连接。

> 还可以查看更多高级示例的“操作方法”部分，通常可以完全控制DataSource的配置。

29.1.1 嵌入式数据库支持

使用内存中嵌入式数据库开发应用程序通常很方便。 显然，内存数据库不提供持久化存储; 您的应用程序启动时，您将需要初始化数据库，并在应用程序结束时丢弃数据。

> “How-to”部分包含如何初始化数据库

Spring Boot可以自动配置嵌入式 H2，HSQL 和 Derby 数据库。 您不需要提供任何连接URL，只需将要使用的嵌入式数据库的依赖关系包含进去即可。

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

例如，典型的POM依赖关系是：

```
1  <dependency>
2      <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>
```

> 对于要自动配置的嵌入式数据库，您需要依赖spring-jdbc。 在这个例子中，它是通过spring-boot-starter-data-jpa传递的。
>
> 如果由于某种原因配置嵌入式数据库的连接URL，则应注意确保数据库的自动关闭被禁用。 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允许Spring Boot控制数据库何时关闭，从而确保在不再需要访问数据库时发生这种情况。

29.1.2 连接到生产环境数据库

生产数据库连接也可以使用连接池数据源自动配置。 这是选择具体实现的算法：

- 我们更喜欢Tomcat连接池DataSource的性能和并发性，所以如果可用，我们总是选择它。
- 否则，如果HikariCP可用，我们将使用它。
- 如果Tomcat池数据源和HikariCP都不可用，并且如果Commons DBCP可用，我们将使用它，但是我们不建议在生产中使用它，并且不支持它。
- 最后，如果Commons DBCP2可用，我们将使用它。

如果您使用spring-boot-starter-jdbc或spring-boot-starter-data-jpa 的startters，您将自动获得对tomcat-jdbc的依赖。

> 您可以完全绕过该算法，并通过spring.datasource.type属性指定要使用的连接池。 如果您在Tomcat容器中运行应用程序，则默认情况下提供tomcat-jdbc，这一点尤为重要。
>
> 可以随时手动配置其他连接池。如果您定义自己的DataSource bean，则不会发生自动配置。

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

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

> 您应至少使用spring.datasource.url属性指定url，否则Spring Boot将尝试自动配置嵌入式数据库。
>
> 您通常不需要指定驱动程序类名称，因为Spring Boot可以从url为大多数数据库推断出驱动程序名称。
>
> 对于要创建的池数据源，我们需要能够验证有效的Driver类是否可用，所以我们在做任何事情之前检查它。 即如果您设置spring.datasource.driver-class-name=com.mysql.jdbc.Driver，那么该类必须可加载。

有关更多支持的选项，请参阅 DataSourceProperties。 这些是标准选项，无论实际执行情况如何。 还可以使用各自的前缀（spring.datasource.tomcat.*，spring.datasource.hikari.*和spring.datasource.dbcp2.*）微调实现特定的设置。 有关更多详细信息，请参阅您正在使用的连接池实现的文档。

例如，如果您正在使用Tomcat连接池，您可以自定义许多其他设置：

```
# 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 ti
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 连接到JNDI DataSource

如果要将Spring Boot应用程序部署到应用程序服务器，则可能需要使用应用程序服务器内置功能来配置和管理DataSource，并使用JNDI进行访问。

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

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

29.2 使用JdbcTemplate

Spring的JdbcTemplate和NamedParameterJdbcTemplate类是自动配置的，您可以将它们直接连接到您自己的bean中：

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

```
4
5  @Component
6  public class MyBean {
7
8      private final JdbcTemplate jdbcTemplate;
9
10     @Autowired
11     public MyBean(JdbcTemplate jdbcTemplate) {
12         this.jdbcTemplate = jdbcTemplate;
13     }
14
15     // ...
16
17 }
```

29.3 JPA 和 ‘Spring Data’

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

- Hibernate - 最受欢迎的JPA实现之一。
- Spring Data JPA - 可以轻松实现基于JPA的存储库。
- Spring ORMs - 来自Spring Framework的核心ORM支持。

> 我们不会在这里介绍太多的JPA或Spring Data的细节。 您可以从spring.io中查看“使用JPA访问数据”指南，并阅读Spring Data JPA和Hibernate参考文档。
>
> 默认情况下，Spring Boot使用Hibernate 5.0.x. 但是，如果您愿意，也可以使用4.3.x或5.2.x。 请参考 Hibernate 4 和 Hibernate 5.2 示例，看看如何做到这一点。

29.3.1 实体类

传统上，JPA’Entity’类在persistence.xml文件中指定。 使用Spring Boot此文件不是必需的，而是使用“实体扫描”。 默认情况下，将搜索主配置类下面的所有包（用@EnableAutoConfiguration或@SpringBootApplication注解的类）。

任何用@Entity，@Embeddable或@MappedSuperclass注解的类将被考虑。 典型的实体类将如下所示：

```
1  package com.example.myapp.domain;
2
3  import java.io.Serializable;
4  import javax.persistence.*;
5
6  @Entity
7  public class City implements Serializable {
```

```
 8
 9       @Id
10       @GeneratedValue
11       private Long id;
12
13       @Column(nullable = false)
14       private String name;
15
16       @Column(nullable = false)
17       private String state;
18
19       // ... additional members, often include @OneToMany mappings
20
21       protected City() {
22           // no-args constructor required by JPA spec
23           // this one is protected since it shouldn't be used directly
24       }
25
26       public City(String name, String state) {
27           this.name = name;
28           this.country = country;
29       }
30
31       public String getName() {
32           return this.name;
33       }
34
35       public String getState() {
36           return this.state;
37       }
38
39       // ... etc
40
41   }
```

> 您可以使用@EntityScan注解自定义实体扫描位置。 请参见第77.4节“从Spring配置中分离@Entity定义”操作方法。

29.3.2 Spring Data JPA Repositories

Spring Data JPA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，CityRepository接口可以声明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，您可以使用Spring数据查询注解来注解您的方法。

Spring数据存储库通常从Repository或CrudRepository接口扩展。 如果您正在使用自动配置，将从包含主配置类（通过@EnableAutoConfiguration或@SpringBootApplication注解的包）的包中搜索存储库(repositories)。

这是一个典型的Spring数据库：

```
1  package com.example.myapp.domain;
2
3  import org.springframework.data.domain.*;
4  import org.springframework.data.repository.*;
5
6  public interface CityRepository extends Repository<City, Long> {
7
8      Page<City> findAll(Pageable pageable);
9
10     City findByNameAndCountryAllIgnoringCase(String name, String country);
11
12 }
```

> 我们只是触及了Spring Data JPA的表面。 有关完整的详细信息，请查阅其参考文档。

29.3.3 创建和删除JPA数据库

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

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

Hibernate自己的内部属性名称（如果你记得更好）是hibernate.hbm2ddl.auto。您可以使用spring.jpa.properties *（将其添加到实体管理器时这个前缀会被删除）与其他Hibernate属性一起设置。例：

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

将 `hibernate.globally_quoted_identifiers` 传递给Hibernate实体管理器。

默认情况下，DDL执行（或验证）将延迟到ApplicationContext启动。 还有一个spring.jpa.generate-ddl标志，但是如果Hibernate 自动配置是激活的，那么它将不会被使用，因为ddl-auto配置更好。

29.3.4 在View中打开EntityManager

如果您正在运行Web应用程序，Spring Boot将默认注册OpenEntityManagerInViewInterceptor来应用“查看”中的“打开EntityManager”模式，即允许在Web视图中进行延迟加载。 如果你不想要这个行为，你应该在你的application.properties中将spring.jpa.open-in-view设置为false。

29.4 使用H2的Web控制台

H2数据库提供了一个基于浏览器的控制台，Spring Boot可以为您自动配置。 满足以下条件时，控制台将自动配置：

- 您正在开发一个Web应用程序
- com.h2database：h2在类路径上
- 您正在使用Spring Boot的开发者工具

> 如果您不使用Spring Boot的开发人员工具，但仍希望使用H2的控制台，那么可以通过配置一个值为true的spring.h2.console.enabled属性来实现。 H2控制台仅用于开发期间，因此应注意确保spring.h2.console.enabled在生产中未设置为true。

29.4.1 更改H2控制台的路径

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

29.4.2 保护H2控制台

当Spring Security位于类路径上且启用了基本身份验证时，H2控制台将自动使用基本身份验证进行保护。 以下属性可用于自定义安全配置：

- security.user.role
- security.basic.authorize-mode
- security.basic.enabled

29.5 使用jOOQ

Java面向对象查询（jOOQ）是Data Geekery的产品，它从数据库生成Java代码，并通过流畅的API构建类型安全的SQL查询。 商业版和开源版都可以与Spring Boot一起使用。

29.5.1 代码生成

为了使用jOOQ类型安全的查询，您需要从数据库模式生成Java类。 您可以按照jOOQ用户手册中的说明进行操作。 如果您正在使用 `jooq-codegen-maven` 插件（并且还使用 `spring-boot-starter-parent` “父POM”），您可以安全地省略插件的标签。 您还可以使用Spring Boot定义的版本变量（例如h2.version）来声明插件的数据库依赖关系。 以下是一个例子：

```
1  <plugin>
2      <groupId>org.jooq</groupId>
3      <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 使用 DSLContext

jOOQ提供的流畅的API是通过org.jooq.DSLContext接口启动的。 Spring Boot将自动配置DSLContext作为Spring Bean并将其连接到应用程序DataSource。 要使用DSLContext，您只需@Autowire它：

```
@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，我们在此示例中也是这样。

然后，您可以使用DSLContext构建查询：

```
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 定制jOOQ

您可以通过在application.properties中设置spring.jooq.sql-dialect来自定义jOOQ使用的SQL方言。 例如，要指定Postgres，您可以添加：

```
spring.jooq.sql-dialect=Postgres
```

通过定义自己的@Bean定义可以实现更高级的定制，这些定义将在创建jOOQ配置时使用。您可以为以下jOOQ类型定义bean：

- ConnectionProvider
- TransactionProvider
- RecordMapperProvider
- RecordListenerProvider
- ExecuteListenerProvider
- VisitListenerProvider

如果要完全控制jOOQ配置，您还可以创建自己的 `org.jooq.Configuration` @Bean。

30. 使用NoSQL技术

Spring Data提供了额外的项目，可帮助您访问各种NoSQL技术，包括MongoDB，Neo4J，Elasticsearch，Solr，Redis，Gemfire，Cassandra，Couchbase和LDAP。 Spring Boot为Redis，MongoDB，Neo4j，Elasticsearch，Solr Cassandra，Couchbase和LDAP提供了自动配置; 您也可以使用其他项目，但您需要自行配置它们。 请参阅projects.spring.io/spring-data中相应的参考文档。

30.1 Redis

Redis是一个缓存，消息代理并有功能丰富的键值存储数据库。Spring Boot提供了Jedis客户端库的基本自动配置和Spring Data Redis提供的抽象。 有一个 `spring-boot-starter-data-redis` “Starter”用于以方便的方式收集依赖关系。

30.1.1 连接到Redis

您可以像任何其他Spring Bean一样注入自动配置的RedisConnectionFactory，StringRedisTemplate或vanilla RedisTemplate实例。 默认情况下，实例将尝试使用localhost:6379连接到Redis服务器：

```
@Component
public class MyBean {

    private StringRedisTemplate template;

```

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

    // ...

}
```

如果您添加了您自己的任何自动配置类型的@Bean，它将替换默认值（除了在RedisTemplate的情况下，排除是基于bean名称“redisTemplate”而不是其类型）。 如果commons-pool2在类路径上，则默认情况下将获得一个pooled连接工厂。

30.2 MongoDB

MongoDB是一个开源的NoSQL文档数据库，它使用类似JSON的架构，而不是传统的基于表的关系数据。Spring Boot为MongoDB提供了几种便利，包括spring-boot-starter-data-mongodb’Starter’。

既然看到最后了，一起来搞基吧：264133057

Spring # Java # Spring Boot

‹ Linux命令 -- source（点命令）

HTTP状态码 ›