Table of Contents

Spring-Boot-Reference-Guide

Spring Boot Reference Guide中文翻译 -《Spring Boot参考指南》

说明：本文档针对的是最新版本：1.4.x.BUILD-SNAPSHOT。

如感兴趣，可以star或fork该仓库！

Github：https://github.com/qibaoguang/

GitBook : Spring Boot参考指南

Email：qibaoguang@gmail.com

从这里开始

更新开始啦，召集小伙伴了，有兴趣的进Spring Boot QQ交流群：445015546

注 目前新版本正在翻译中，所以可能导航之类的有错位现象，望大家谅解，如果影响阅读可以查看1.3.x版本（release版）。

Spring Boot文档

本节对Spring Boot参考文档做了一个简单概述。你可以参考本节，从头到尾依次阅读该文档，也可以跳过不感兴趣的章节。

1. 关于本文档

Spring Boot参考指南有html，pdf和epub等形式的文档，你可以从docs.spring.io/spring-boot/docs/current/reference获取到最新版本。

2. 获取帮助

使用Spring Boot遇到麻烦，我们很乐意帮忙！

- 尝试How-to's－它们为多数常见问题提供解决方案。
- 学习Spring基础知识－Spring Boot是在很多其他Spring项目上构建的，查看spring.io站点可以获取丰富的参考文档。如果你刚开始使用Spring，可以尝试这些指导中的一个。
- 提问题－我们时刻监控着stackoverflow.com上标记为spring-boot的问题。
- 在github.com/spring-projects/spring-boot/issues上报告Spring Boot的bug。

注：Spring Boot的一切都是开源的，包括文档！如果你发现文档有问题，或只是想提高它们的质量，请参与进来！

3. 第一步

如果你想对Spring Boot或Spring有个整体认识，可以从这里开始！

- 从零开始：概述 | 要求 | 安装
- 教程：第一部分 | 第二部分
- 运行示例：第一部分 | 第二部分

4. 使用Spring Boot

准备好使用Spring Boot了？我们已经为你铺好道路.

- 构建系统：Maven | Gradle | Ant | Starter POMs
- 最佳实践：代码结构 | @Configuration | @EnableAutoConfiguration | Beans和依赖注入
- 运行代码：IDE | Packaged | Maven | Gradle
- 应用打包：产品级jars
- Spring Boot命令行：使用CLI

5. 了解Spring Boot特性

想要了解更多Spring Boot核心特性的详情？这就是为你准备的！

- 核心特性：SpringApplication | 外部化配置 | Profiles | 日志
- Web应用：MVC | 内嵌容器
- 使用数据：SQL | NO-SQL
- 消息：概述 | JMS
- 测试：概述 | Boot应用 | 工具
- 扩展：Auto-configuration | @Conditions

6. 迁移到生产环境

当你准备将Spring Boot应用发布到生产环境时，我们提供了一些你可能喜欢的技巧！

- 管理端点：概述 | 自定义
- 连接可选项：HTTP | JMX | SSH
- 监控：指标 | 审计 | 跟踪 | 进程

7. 高级主题

最后，我们为高级用户准备了一些主题。

- 部署Spring Boot应用：云部署 | 操作系统服务
- 构建工具插件：Maven | Gradle
- 附录：应用属性 | Auto-configuration类 | 可执行Jars

入门指南

如果你想从大体上了解Spring Boot或Spring，本章节正是你所需要的！本节中，我们会回答基本的"what?"，"how?"和"why?"等问题，并通过一些安装指南简单介绍下Spring Boot。然后我们会构建第一个Spring Boot应用，并讨论一些需要遵循的核心原则。

8. Spring Boot介绍

Spring Boot简化了基于Spring的应用开发，你只需要"run"就能创建一个独立的，产品级别的Spring应用。 我们为Spring平台及第三方库提供开箱即用的设置，这样你就可以有条不紊地开始。多数Spring Boot应用只需要很少的Spring配置。

你可以使用Spring Boot创建Java应用，并使用 `java -jar` 启动它或采用传统的war部署方式。我们也提供了一个运行"spring脚本"的命令行工具。

我们主要的目标是：

- 为所有Spring开发提供一个从根本上更快，且随处可得的入门体验。
- 开箱即用，但通过不采用默认设置可以快速摆脱这种方式。
- 提供一系列大型项目常用的非功能性特征，比如：内嵌服务器，安全，指标，健康检测，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 1.4.0.BUILD-SNAPSHOT 需要Java7环境，Spring框架4.3.2.BUILD-SNAPSHOT或以上版本。你可以在Java6下使用Spring Boot，不过需要添加额外配置。具体参考Section 82.11, “How to use Java 6”。明确提供构建支持的有Maven（3.2+）和Gradle（1.12+）。

注：尽管你可以在Java6或Java7环境下使用Spring Boot，通常建议尽可能使用Java8。

9.1. Servlet容器

下列内嵌容器支持开箱即用（out of the box）：

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+

你也可以将Spring Boot应用部署到任何兼容Servlet 3.0+的容器。

10. Spring Boot安装

Spring Boot可以跟经典的Java开发工具（Eclipse，IntelliJ等）一起使用或安装成一个命令行工具。不管怎样，你都需要安装Java SDK v1.6 或更高版本。在开始之前，你需要检查下当前安装的Java版本：

```
$ java -version
```

如果你是一个Java新手，或只是想体验一下Spring Boot，你可能想先尝试Spring Boot CLI，否则继续阅读“经典”地安装指南。

注：尽管Spring Boot兼容Java 1.6，如果可能的话，你应该考虑使用Java最新版本。

10.1. 为Java开发者准备的安装指南

对于java开发者来说，使用Spring Boot就跟使用其他Java库一样，只需要在你的classpath下引入适当的 `spring-boot-*.jar` 文件。Spring Boot不需要集成任何特殊的工具，所以你可以使用任何IDE或文本编辑器；同时，Spring Boot应用也没有什么特殊之处，你可以像对待其他Java程序那样运行，调试它。

尽管可以拷贝Spring Boot jars，但我们还是建议你使用支持依赖管理的构建工具，比如Maven或Gradle。

10.1.1. Maven安装

Spring Boot兼容Apache Maven 3.2或更高版本。如果本地没有安装Maven，你可以参考maven.apache.org上的指南。

注：在很多操作系统上，可以通过包管理器来安装Maven。OSX Homebrew用户可以尝试 `brew install maven` ，Ubuntu用户可以运行 `sudo apt-get install maven` 。

Spring Boot依赖使用的groupId为 `org.springframework.boot` 。通常，你的Maven POM文件会继承 `spring-boot-starter-parent` 工程，并声明一个或多个“Starter POMs”依赖。此外，Spring Boot提供了一个可选的Maven插件，用于创建可执行jars。

下面是一个典型的pom.xml文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/X
MLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <!-- Inherit defaults from Spring Boot -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.0.BUILD-SNAPSHOT</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>
```

```
    <!-- Add Spring repositories -->
    <!-- (you don't need this if you are using a .RELEASE version) -->
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </pluginRepository>
    </pluginRepositories>
</project>
```

注：`spring-boot-starter-parent`是使用Spring Boot的一种不错的方式，但它并不总是最合适的。有时你可能需要继承一个不同的父 POM，或只是不喜欢我们的默认配置，那你可以使用import作用域这种替代方案，具体查看Section 13.2.2, “Using Spring Boot without the parent POM”。

10.1.2. Gradle安装

Spring Boot兼容Gradle 1.12或更高版本。如果本地没有安装Gradle，你可以参考www.gradle.org上的指南。

Spring Boot的依赖可通过groupId `org.springframework.boot` 来声明。通常，你的项目将声明一个或多个“Starter POMs”依赖。Spring Boot提供了一个很有用的Gradle插件，可以用来简化依赖声明，创建可执行jars。

注：当你需要构建项目时，Gradle Wrapper提供一种给力的获取Gradle的方式。它是一小段脚本和库，跟你的代码一块提交，用于启动构建进程，具体参考Gradle Wrapper。

下面是一个典型的 `build.gradle` 文件：

```
buildscript {
    repositories {
        jcenter()
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.4.0.BUILD-SNAP
SHOT")
    }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

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

repositories {
    jcenter()
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

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

10.2. Spring Boot CLI安装

Spring Boot CLI是一个命令行工具，可用于快速搭建基于Spring的原型。它支持运行Groovy脚本，这也就意味着你可以使用类似Java的语法，但不用写很多的模板代码。

Spring Boot不一定非要配合CLI使用，但它绝对是Spring应用取得进展的最快方式（你咋不飞上天呢？）。

10.2.1. 手动安装

Spring CLI分发包可以从Spring软件仓库下载：

1. spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin.zip
2. spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin.tar.gz

不稳定的snapshot分发包也可以获取到。

下载完成后，解压分发包，根据存档里的INSTALL.txt操作指南进行安装。总的来说，在 `.zip` 文件的 `bin/` 目录下会有一个spring脚本（Windows下是 `spring.bat` ），或使用 `java -jar` 运行 `lib/` 目录下的 `.jar` 文件（该脚本会帮你确保classpath被正确设置）。

10.2.2. 使用SDKMAN安装

SDKMAN（软件开发包管理器）可以对各种各样的二进制SDK包进行版本管理，包括Groovy和Spring Boot CLI。可以从sdkman.io下载SDKMAN，并使用以下命令安装Spring Boot：

```
$ sdk install springboot
$ spring --version
Spring Boot v1.4.0.BUILD-SNAPSHOT
```

如果你正在为CLI开发新的特性，并想轻松获取刚构建的版本，可以使用以下命令：

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-c
li-1.4.0.BUILD-SNAPSHOT-bin/spring-1.4.0.BUILD-SNAPSHOT/
$ sdk default springboot dev
$ spring --version
Spring CLI v1.4.0.BUILD-SNAPSHOT
```

这将会安装一个名叫dev的本地spring实例，它指向你的目标构建位置，所以每次你重新构建Spring Boot，spring都会更新为最新的。

你可以通过以下命令来验证：

```
$ sdk ls springboot

================================================================================
Available Springboot Versions
================================================================================
> + dev
* 1.4.0.BUILD-SNAPSHOT

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

10.2.3. 使用OSX Homebrew进行安装

如果你的环境是Mac，并使用Homebrew，想要安装Spring Boot CLI只需以下操作：

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

Homebrew将把spring安装到 `/usr/local/bin` 下。

注：如果该方案不可用，可能是因为你的brew版本太老了。你只需执行 `brew update` 并重试即可。

10.2.4. 使用MacPorts进行安装

如果你的环境是Mac，并使用MacPorts，想要安装Spring Boot CLI只需以下操作：

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

10.2.5. 命令行实现

Spring Boot CLI启动脚本为BASH和zsh shells提供完整的命令行实现。你可以在任何shell中source脚本（名称也是spring），或将它放到用户或系统范围内的bash初始化脚本里。在Debian系统中，系统级的脚本位于 `/shell-completion/bash` 下，当新的shell启动时该目录下的所有脚本都会被执行。如果想要手动运行脚本，假如你已经安装了SDKMAN，可以使用以下命令：

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

注：如果你使用Homebrew或MacPorts安装Spring Boot CLI，命令行实现脚本会自动注册到你的shell。

10.2.6. Spring CLI示例快速入门

下面是一个相当简单的web应用，你可以用它测试Spring CLI安装是否成功。创建一个名叫 `app.groovy` 的文件：

```
@RestController
class ThisWillActuallyRun {

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

}
```

然后只需在shell中运行以下命令：

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

注：首次运行该应用将会花费一些时间，因为需要下载依赖，后续运行将会快很多。

使用你最喜欢的浏览器打开localhost:8080，然后就可以看到如下输出：

```
Hello World!
```

10.3. 版本升级

如果你正在升级Spring Boot的早期发布版本，那最好查看下project wiki上的"release notes"，你会发现每次发布对应的升级指南和一个"new and noteworthy"特性列表。

想要升级一个已安装的CLI，你需要使用合适的包管理命令，例如 `brew upgrade` ；如果是手动安装CLI，按照standard instructions操作并记得更新你的PATH环境变量以移除任何老的引用。

11. 开发你的第一个Spring Boot应用

我们将使用Java开发一个简单的"Hello World" web应用，以此强调下Spring Boot的一些关键特性。项目采用Maven进行构建，因为大多数IDEs都支持它。

注：spring.io网站包含很多Spring Boot"入门"指南，如果你正在找特定问题的解决方案，可以先去那瞅瞅。你也可以简化下面的步骤，直接从start.spring.io的依赖搜索器选中 `web` starter，这会自动生成一个新的项目结构，然后你就可以happy的敲代码了。具体详情参考文档。

在开始前，你需要打开终端检查下安装的Java和Maven版本是否可用：

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

```
$ mvn -v
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T13:58:10-07:0
0)
Maven home: /Users/user/tools/apache-maven-3.1.1
Java version: 1.7.0_51, vendor: Oracle Corporation
```

注：该示例需要创建单独的文件夹，后续的操作建立在你已创建一个合适的文件夹，并且它是你的“当前目录”。

11.1. 创建POM

让我们以创建一个Maven `pom.xml` 文件作为开始吧，因为 `pom.xml` 是构建项目的处方！打开你最喜欢的文本编辑器，并添加以下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/X
MLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.BUILD-SNAPSHOT</version>
    </parent>

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

    <!-- (you don't need this if you are using a .RELEASE version) -->
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </pluginRepository>
    </pluginRepositories>
</project>
```

这样一个可工作的构建就完成了，你可以通过运行 `mvn package` 测试它（暂时忽略"jar将是空的-没有包含任何内容！"的警告）。

注：此刻，你可以将该项目导入到IDE中（大多数现代的Java IDE都包含对Maven的内建支持）。简单起见，我们将继续使用普通的文本编辑器完成该示例。

11.2. 添加classpath依赖

Spring Boot提供很多"Starter POMs"，用来简化添加jars到classpath的操作。示例程序中已经在POM的partent节点使用了 `spring-boot-starter-parent` ，它是一个特殊的starter，提供了有用的Maven默认设置。同时，它也提供一个 `dependency-management` 节点，这样对于期望（"blessed"）的依赖就可以省略version标记了。

其他"Starter POMs"只简单提供开发特定类型应用所需的依赖。由于正在开发web应用，我们将添加 `spring-boot-starter-web` 依赖-但在此之前，让我们先看下目前的依赖：

```
$ mvn dependency:tree
[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT
```

`mvn dependency:tree` 命令可以将项目依赖以树形方式展现出来，你可以看到 `spring-boot-starter-parent` 本身并没有提供依赖。编辑 `pom.xml` ，并在parent节点下添加 `spring-boot-starter-web` 依赖：

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

如果再次运行 `mvn dependency:tree` ，你将看到现在多了一些其他依赖，包括Tomcat web服务器和Spring Boot自身。

11.3. 编写代码

为了完成应用程序，我们需要创建一个单独的Java文件。Maven默认会编译 `src/main/java` 下的源码，所以你需要创建那样的文件结构，并添加一个名为 `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);
    }

}
```

尽管代码不多，但已经发生了很多事情，让我们分步探讨重要的部分吧！

11.3.1. @RestController和@RequestMapping注解

Example类上使用的第一个注解是 `@RestController` ，这被称为构造型（stereotype）注解。它为阅读代码的人提供暗示（这是一个支持REST的控制器），对于Spring，该类扮演了一个特殊角色。在本示例中，我们的类是一个web `@Controller` ，所以当web请求进来时，Spring会考虑是否使用它来处理。

`@RequestMapping` 注解提供路由信息，它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到 `home` 方法。 `@RestController` 注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。

注： `@RestController` 和 `@RequestMapping` 是Spring MVC中的注解（它们不是Spring Boot的特定部分），具体参考Spring文档的MVC章节。

11.3.2. @EnableAutoConfiguration注解

第二个类级别的注解是`@EnableAutoConfiguration`，这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于`spring-boot-starter-web`添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用，并对Spring进行相应地设置。

Starters和**Auto-Configuration**：Auto-configuration设计成可以跟"Starters"一起很好的使用，但这两个概念没有直接的联系。你可以自由地挑选starters以外的jar依赖，Spring Boot仍会尽最大努力去自动配置你的应用。

11.3.3. main方法

应用程序的最后部分是main方法，这是一个标准的方法，它遵循Java对于一个应用程序入口点的约定。我们的main方法通过调用 `run` ，将业务委托给了Spring Boot的SpringApplication类。SpringApplication将引导我们的应用，启动Spring，相应地启动被自动配置的Tomcat web服务器。我们需要将 `Example.class` 作为参数传递给 `run` 方法，以此告诉SpringApplication谁是主要的Spring组件，并传递args数组以暴露所有的命令行参数。

11.4. 运行示例

到此，示例应用可以工作了。由于使用了 `spring-boot-starter-parent` POM，这样我们就有了一个非常有用的run目标来启动程序。在项目根目录下输入 `mvn spring-boot:run` 启动应用：

```
$ mvn spring-boot:run

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.4.1.BUILD-SNAPSHOT)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

如果使用浏览器打开localhost:8080，你应该可以看到如下输出：

```
Hello World!
```

点击 `ctrl-c` 温雅地关闭应用程序。

11.5. 创建可执行jar

让我们通过创建一个完全自包含，并可以在生产环境运行的可执行jar来结束示例吧！可执行jars（有时被称为胖jars "fat jars"）是包含编译后的类及代码运行所需依赖jar的存档。

可执行**jars**和**Java**：Java没有提供任何标准方式，用于加载内嵌jar文件（即jar文件中还包含jar文件），这对分发自包含应用来说是个问题。为了解决该问题，很多开发者采用"共享的"jars。共享的jar只是简单地将所有jars的类打包进一个单独的存档，这种方式存在的问题是，很难区分应用程序中使用了哪些库。在多个jars中如果存在相同的文件名（但内容不一样）也会是一个问题。Spring Boot采取一个不同的方式，允许你真正的直接内嵌jars。

为了创建可执行的jar，我们需要将 `spring-boot-maven-plugin` 添加到 `pom.xml` 中，在dependencies节点后面插入以下内容：

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

注： `spring-boot-starter-parent` POM包含绑定到repackage目标的 `<executions>` 配置。如果不使用parent POM，你需要自己声明该配置，具体参考插件文档。

保存 `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-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.4.1.BUILD-SNAPSHOT:repackage (default) @ myproje
ct ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

如果查看target目录，你应该可以看到 `myproject-0.0.1-SNAPSHOT.jar` ，该文件大概有10Mb。想查看内部结构，可以运行 `jar tvf` ：

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

在该目录下，你应该还能看到一个很小的名为 `myproject-0.0.1-SNAPSHOT.jar.original` 的文件，这是在Spring Boot重新打包前，Maven创建的原始jar文件。

可以使用 `java -jar` 命令运行该应用程序：

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

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.3.0.BUILD-SNAPSHOT)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864)
```

如上所述，点击 `ctrl-c` 可以温雅地退出应用。

12. 接下来阅读什么

希望本章节已为你提供一些Spring Boot的基础部分，并帮你找到开发自己应用的方式。如果你是任务驱动型的开发者，那可以直接跳到spring.io，check out一些入门指南，以解决特定的"使用Spring如何做"的问题；我们也有Spring Boot相关的How-to参考文档。

Spring Boot仓库有大量可以运行的示例，这些示例代码是彼此独立的(运行或使用示例的时候不需要构建其他示例)。

否则，下一步就是阅读 III、使用Spring Boot，如果没耐心，可以跳过该章节，直接阅读 IV、Spring Boot特性。

使用Spring Boot

本章节将详细介绍如何使用Spring Boot，不仅覆盖构建系统，自动配置，如何运行应用等主题，还包括一些Spring Boot的最佳实践。尽管Spring Boot本身没有什么特别的（跟其他一样，它只是另一个你可以使用的库），但仍有一些建议，如果遵循的话将会事半功倍。

如果你刚接触Spring Boot，那最好先阅读上一章节的Getting Started指南。

13. 构建系统

强烈建议你选择一个支持依赖管理，能消费发布到"Maven中央仓库"的artifacts的构建系统，比如Maven或Gradle。使用其他构建系统也是可以的，比如Ant，但它们可能得不到很好的支持。

14. 组织你的代码

Spring Boot不要求使用任何特殊的代码结构，不过，遵循以下的一些最佳实践还是挺有帮助的。

14.1. 使用"default"包

当类没有声明 `package` 时，它被认为处于 `default package` 下。通常不推荐使用 `default package` ，因为对于使用 `@ComponentScan` ， `@EntityScan` 或 `@SpringBootApplication` 注解的Spring Boot应用来说，它会扫描每个jar中的类，这会造成一定的问题。

注 我们建议你遵循Java推荐的包命名规范，使用一个反转的域名（例如 `com.example.project` ）。

14.2. 放置应用的main类

通常建议将应用的main类放到其他类所在包的顶层(root package)，并将 `@EnableAutoConfiguration` 注解到你的main类上，这样就隐式地定义了一个基础的包搜索路径（search package），以搜索某些特定的注解实体（比如@Service，@Component等）。例如，如果你正在编写一个JPA应用，Spring将搜索 `@EnableAutoConfiguration` 注解的类所在包下的 `@Entity` 实体。

采用root package方式，你就可以使用 `@ComponentScan` 注解而不需要指定 `basePackage` 属性，也可以使用 `@SpringBootApplication` 注解，只要将main类放到root package中。

下面是一个典型的结构：

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

`Application.java` 将声明 `main` 方法，还有基本的 `@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提倡基于Java的配置。尽管你可以使用XML源调用 `SpringApplication.run()` ，不过还是建议你使用 `@Configuration` 类作为主要配置源。通常定义了 `main` 方法的类也是使用 `@Configuration` 注解的一个很好的替补。

注：虽然网络上有很多使用XML配置的Spring示例，但你应该尽可能的使用基于Java的配置，搜索查看 `enable*` 注解就是一个好的开端。

15.1. 导入其他配置类

你不需要将所有的 `@Configuration` 放进一个单独的类， `@Import` 注解可以用来导入其他配置类。另外，你也可以使用 `@ComponentScan` 注解自动收集所有Spring组件，包括 `@Configuration` 类。

15.2. 导入XML配置

如果必须使用XML配置，建议你仍旧从一个 `@Configuration` 类开始，然后使用 `@ImportResource` 注解加载XML配置文件。

16. 自动配置

Spring Boot自动配置（auto-configuration）尝试根据添加的jar依赖自动配置你的Spring应用。例如，如果classpath下存在 `HSQLDB` ，并且你没有手动配置任何数据库连接的beans，那么Spring Boot将自动配置一个内存型（in-memory）数据库。

实现自动配置有两种可选方式，分别是
将 `@EnableAutoConfiguration` 或 `@SpringBootApplication` 注解到 `@Configuration` 类上。

注：你应该只添加一个 `@EnableAutoConfiguration` 注解，通常建议将它添加到主配置类（primary `@Configuration` ）上。

16.1. 逐步替换自动配置

自动配置（Auto-configuration）是非侵入性的，任何时候你都可以定义自己的配置类来替换自动配置的特定部分。例如，如果你添加自己的 `DataSource` bean，默认的内嵌数据库支持将不被考虑。

如果需要查看当前应用启动了哪些自动配置项，你可以在运行应用时打开 `--debug` 开关，这将为核心日志开启debug日志级别，并将自动配置相关的日志输出到控制台。

16.2. 禁用特定的自动配置项

如果发现启用了不想要的自动配置项，你可以使用 `@EnableAutoConfiguration` 注解的exclude属性禁用它们：

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

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

如果该类不在classpath中，你可以使用该注解的excludeName属性，并指定全限定名来达到相同效果。最后，你可以通过 `spring.autoconfigure.exclude` 属性exclude多个自动配置项（一个自动配置项集合）。

注 通过注解级别或exclude属性都可以定义排除项。

17. Spring Beans和依赖注入

你可以自由地使用任何标准的Spring框架技术去定义beans和它们注入的依赖。简单起见，我们经常使用 `@ComponentScan` 注解搜索beans，并结合 `@Autowired` 构造器注入。

如果遵循以上的建议组织代码结构（将应用的main类放到包的最上层，即root package），那么你就可以添加 `@ComponentScan` 注解而不需要任何参数，所有应用组件（ `@Component` , `@Service` , `@Repository` , `@Controller` 等）都会自动注册成Spring Beans。

下面是一个 `@Service` Bean的示例，它使用构建器注入获取一个需要的 `RiskAssessor` bean。

```
package com.example.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...
}
```

注 注意使用构建器注入允许 `riskAssessor` 字段被标记为 `final` ，这意味着 `riskAssessor` 后续是不能改变的。

18. 使用@SpringBootApplication注解

很多Spring Boot开发者经常使用 `@Configuration` ， `@EnableAutoConfiguration` ， `@ComponentScan` 注解他们的main类，由于这些注解如此频繁地一块使用（特别是遵循以上最佳实践的时候），Spring Boot就提供了一个方便的 `@SpringBootApplication` 注解作为代替。

`@SpringBootApplication` 注解等价于以默认属性使用 `@Configuration` ， `@EnableAutoConfiguration` 和 `@ComponentScan` ：

```
package com.example.myproject;

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

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentSc
an
public class Application {

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

}
```

注 `@SpringBootApplication` 注解也提供了用于自定义 `@EnableAutoConfiguration` 和 `@ComponentScan` 属性的别名（aliases）。

19. 运行应用程序

将应用打包成jar，并使用内嵌HTTP服务器的一个最大好处是，你可以像其他方式那样运行你的应用程序。调试Spring Boot应用也很简单，你都不需要任何特殊IDE插件或扩展！

注：本章节只覆盖基于jar的打包，如果选择将应用打包成war文件，你最好参考相关的服务器和IDE文档。

19.1. 从IDE中运行

你可以从IDE中运行Spring Boot应用，就像一个简单的Java应用，但首先需要导入项目。导入步骤取决于你的IDE和构建系统，大多数IDEs能够直接导入Maven项目，例如Eclipse用户可以选择 `File` 菜单的 `Import…` --> `Existing Maven Projects` 。

如果不能直接将项目导入IDE，你可以使用构建系统生成IDE的元数据。Maven有针对Eclipse和IDEA的插件；Gradle为各种IDEs提供插件。

注 如果意外地多次运行一个web应用，你将看到一个"端口已被占用"的错误。STS用户可以使用 `Relaunch` 而不是 `Run` 按钮，以确保任何存在的实例是关闭的。

19.2. 作为一个打包后的应用运行

如果使用Spring Boot Maven或Gradle插件创建一个可执行jar，你可以使用 `java -jar` 运行应用。例如：

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

Spring Boot支持以远程调试模式运行一个打包的应用，下面的命令可以为应用关联一个调试器：

```
$ 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插件运行

Spring Boot Maven插件包含一个 `run` 目标，可用来快速编译和运行应用程序，并且跟在IDE运行一样支持热加载。

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

你可以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插件运行

Spring Boot Gradle插件也包含一个 `bootRun` 任务，可用来运行你的应用程序。无论你何时 import `spring-boot-gradle-plugin` ， `bootRun` 任务总会被添加进去。

```
$ gradle bootRun
```

你可能想使用一些有用的操作系统环境变量：

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

19.5. 热交换

由于Spring Boot应用只是普通的Java应用，所以JVM热交换（hot-swapping）也能开箱即用。不过JVM热交换能替换的字节码有限制，想要更彻底的解决方案可以使用Spring Loaded项目或JRebel。`spring-boot-devtools`模块也支持应用快速重启(restart)。

详情参考下面的Chapter 20, Developer tools和“How-to”章节。

Spring Boot特性

22. SpringApplication

SpringApplication类提供了一种从main()方法启动Spring应用的便捷方式。在很多情况下，你只需委托给SpringApplication.run这个静态方法：

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

当应用启动时，你应该会看到类似下面的东西（这是何方神兽？？）：

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v1.2.2.BUILD-SNAPSHOT

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplicatio
n            : Starting SampleApplication v0.1.0 on mycomputer with PID 56603 (/apps/m
yapp.jar started by pwebb)
2013-07-31 00:08:16.166  INFO 56603 --- [           main] ationConfigEmbeddedWebApplic
ationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEm
beddedWebApplicationContext@6e5a8246: startup date [Wed Jul 31 00:08:16 PDT 2013]; roo
t of context hierarchy
2014-03-04 13:09:54.912  INFO 41370 --- [           main] .t.TomcatEmbeddedServletCont
ainerFactory : Server initialized with port: 8080
2014-03-04 13:09:56.501  INFO 41370 --- [           main] o.s.b.s.app.SampleApplicatio
n            : Started SampleApplication in 2.992 seconds (JVM running for 3.658)
```

默认情况下会显示INFO级别的日志信息，包括一些相关的启动详情，比如启动应用的用户等。

22.1. 自定义Banner

通过在classpath下添加一个banner.txt或设置banner.location来指定相应的文件可以改变启动过程中打印的banner。如果这个文件有特殊的编码，你可以使用banner.encoding设置它（默认为UTF-8）。

在banner.txt中可以使用如下的变量：

变量	描述
${application.version}	MANIFEST.MF中声明的应用版本号，例如1.0
${application.formatted-version}	MANIFEST.MF中声明的被格式化后的应用版本号（被括号包裹且以v作为前缀），用于显示，例如(v1.0)
${spring-boot.version}	正在使用的Spring Boot版本号，例如1.2.2.BUILD-SNAPSHOT
${spring-boot.formatted-version}	正在使用的Spring Boot被格式化后的版本号（被括号包裹且以v作为前缀）, 用于显示，例如(v1.2.2.BUILD-SNAPSHOT)

注：如果想以编程的方式产生一个banner，可以使用SpringBootApplication.setBanner(…)方法。使用org.springframework.boot.Banner接口，实现你自己的printBanner()方法。

22.2. 自定义SpringApplication

如果默认的SpringApplication不符合你的口味，你可以创建一个本地的实例并自定义它。例如，关闭banner你可以这样写：

```
public static void main(String[] args){
    SpringApplication app = new SpringApplication(MySpringConfiguration.class);
    app.setShowBanner(false);
    app.run(args);
}
```

注：传递给SpringApplication的构造器参数是spring beans的配置源。在大多数情况下，这些将是@Configuration类的引用，但它们也可能是XML配置或要扫描包的引用。

你也可以使用application.properties文件来配置SpringApplication。具体参考Externalized 配置。查看配置选项的完整列表，可参考SpringApplication Javadoc.

22.3. 流畅的构建API

如果你需要创建一个分层的ApplicationContext（多个具有父子关系的上下文），或你只是喜欢使用流畅的构建API，你可以使用SpringApplicationBuilder。SpringApplicationBuilder允许你以链式方式调用多个方法，包括可以创建层次结构的parent和child方法。

```
new SpringApplicationBuilder()
    .showBanner(false)
    .sources(Parent.class)
    .child(Application.class)
    .run(args);
```

注：创建ApplicationContext层次时有些限制，比如，Web组件(components)必须包含在子上下文(child context)中，且相同的Environment即用于父上下文也用于子上下文中。具体参考SpringApplicationBuilder javadoc

22.4. Application事件和监听器

除了常见的Spring框架事件，比如ContextRefreshedEvent，一个SpringApplication也发送一些额外的应用事件。一些事件实际上是在ApplicationContext被创建前触发的。

你可以使用多种方式注册事件监听器，最普通的是使用SpringApplication.addListeners(…)方法。在你的应用运行时，应用事件会以下面的次序发送：

1. 在运行开始，但除了监听器注册和初始化以外的任何处理之前，会发送一个ApplicationStartedEvent。
2. 在Environment将被用于已知的上下文，但在上下文被创建前，会发送一个ApplicationEnvironmentPreparedEvent。
3. 在refresh开始前，但在bean定义已被加载后，会发送一个ApplicationPreparedEvent。
4. 启动过程中如果出现异常，会发送一个ApplicationFailedEvent。

注：你通常不需要使用应用程序事件，但知道它们的存在会很方便（在某些场合可能会使用到）。在Spring内部，Spring Boot使用事件处理各种各样的任务。

22.5. Web环境

一个SpringApplication将尝试为你创建正确类型的ApplicationContext。在默认情况下，使用AnnotationConfigApplicationContext或AnnotationConfigEmbeddedWebApplicationContext取决于你正在开发的是否是web应用。

用于确定一个web环境的算法相当简单（基于是否存在某些类）。如果需要覆盖默认行为，你可以使用setWebEnvironment(boolean webEnvironment)。通过调用setApplicationContextClass(…)，你可以完全控制ApplicationContext的类型。

注：当JUnit测试里使用SpringApplication时，调用setWebEnvironment(false)是可取的。

22.6. 命令行启动器

如果你想获取原始的命令行参数，或一旦SpringApplication启动，你需要运行一些特定的代码，你可以实现CommandLineRunner接口。在所有实现该接口的Spring beans上将调用run(String... args)方法。

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

@Component
public class MyBean implements CommandLineRunner {
    public void run(String... args) {
        // Do something...
    }
}
```

如果一些CommandLineRunner beans被定义必须以特定的次序调用，你可以额外实现org.springframework.core.Ordered接口或使用org.springframework.core.annotation.Order注解。

22.7. Application退出

每个SpringApplication在退出时为了确保ApplicationContext被优雅的关闭，将会注册一个JVM的shutdown钩子。所有标准的Spring生命周期回调（比如，DisposableBean接口或@PreDestroy注解）都能使用。

此外，如果beans想在应用结束时返回一个特定的退出码（exit code），可以实现org.springframework.boot.ExitCodeGenerator接口。

23.外化配置

Spring Boot允许外化（externalize）你的配置，这样你能够在不同的环境下使用相同的代码。你可以使用properties文件，YAML文件，环境变量和命令行参数来外化配置。使用@Value注解，可以直接将属性值注入到你的beans中，并通过Spring的Environment抽象或绑定到结构化对象来访问。

Spring Boot使用一个非常特别的PropertySource次序来允许对值进行合理的覆盖，需要以下面的次序考虑属性：

1. 命令行参数
2. 来自于java:comp/env的JNDI属性
3. Java系统属性（System.getProperties()）
4. 操作系统环境变量
5. 只有在random.*里包含的属性会产生一个RandomValuePropertySource
6. 在打包的jar外的应用程序配置文件（application.properties，包含YAML和profile变量）
7. 在打包的jar内的应用程序配置文件（application.properties，包含YAML和profile变量）
8. 在@Configuration类上的@PropertySource注解
9. 默认属性（使用SpringApplication.setDefaultProperties指定）

下面是一个具体的示例（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;
    // ...
}
```

你可以将一个application.properties文件捆绑到jar内，用来提供一个合理的默认name属性值。当运行在生产环境时，可以在jar外提供一个application.properties文件来覆盖name属性。对于一次性的测试，你可以使用特定的命令行开关启动（比如，java -jar app.jar --name="Spring"）。

23.1. 配置随机值

RandomValuePropertySource在注入随机值（比如，密钥或测试用例）时很有用。它能产生整数，longs或字符串，比如：

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

random.int*语法是OPEN value (,max) CLOSE，此处OPEN，CLOSE可以是任何字符，并且value，max是整数。如果提供max，那么value是最小的值，max是最大的值（不包含在内）。

23.2. 访问命令行属性

默认情况下，SpringApplication将任何可选的命令行参数（以'--'开头，比如，--server.port=9000）转化为property，并将其添加到Spring Environment中。如上所述，命令行属性总是优先于其他属性源。

如果你不想将命令行属性添加到Environment里，你可以使用SpringApplication.setAddCommandLineProperties(false)来禁止它们。

23.3. Application属性文件

SpringApplication将从以下位置加载application.properties文件，并把它们添加到Spring Environment中：

1. 当前目录下的一个/config子目录
2. 当前目录
3. 一个classpath下的/config包
4. classpath根路径（root）

这个列表是按优先级排序的（列表中位置高的将覆盖位置低的）。

注：你可以使用YAML（'.yml'）文件替代'.properties'。

如果不喜欢将application.properties作为配置文件名，你可以通过指定spring.config.name环境属性来切换其他的名称。你也可以使用spring.config.location环境属性来引用一个明确的路径（目录位置或文件路径列表以逗号分割）。

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

如果spring.config.location包含目录（相对于文件），那它们应该以/结尾（在加载前，spring.config.name产生的名称将被追加到后面）。不管spring.config.location是什么值，默认的搜索路径classpath:,classpath:/config,file:,file:config/总会被使用。以这种方式，你可以在application.properties中为应用设置默认值，然后在运行的时候使用不同的文件覆盖它，同时保留默认配置。

注：如果你使用环境变量而不是系统配置，大多数操作系统不允许以句号分割（period-separated）的key名称，但你可以使用下划线（underscores）代替（比如，使用SPRING_CONFIG_NAME代替spring.config.name）。如果你的应用运行在一个容器中，那么JNDI属性（java:comp/env）或servlet上下文初始化参数可以用来取代环境变量或系统属性，当然也可以使用环境变量或系统属性。

23.4. 特定的Profile属性

除了application.properties文件，特定配置属性也能通过命令惯例application-{profile}.properties来定义。特定Profile属性从跟标准application.properties相同的路径加载，并且特定profile文件会覆盖默认的配置。

23.5. 属性占位符

当application.properties里的值被使用时，它们会被存在的Environment过滤，所以你能够引用先前定义的值（比如，系统属性）。

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

注：你也能使用相应的技巧为存在的Spring Boot属性创建'短'变量，具体参考Section 63.3, “Use ‘short’ command line arguments”。

23.6. 使用YAML代替Properties

YAML是JSON的一个超集，也是一种方便的定义层次配置数据的格式。无论你何时将SnakeYAML 库放到classpath下，SpringApplication类都会自动支持YAML作为properties的替换。

注：如果你使用'starter POMs'，spring-boot-starter会自动提供SnakeYAML。

23.6.1. 加载YAML

Spring框架提供两个便利的类用于加载YAML文档，YamlPropertiesFactoryBean会将YAML作为Properties来加载，YamlMapFactoryBean会将YAML作为Map来加载。

示例：

```
environments:
    dev:
        url: http://dev.bar.com
        name: Developer Setup
    prod:
        url: http://foo.bar.com
        name: My Cool App
```

上面的YAML文档会被转化到下面的属性中：

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

YAML列表被表示成使用[index]间接引用作为属性keys的形式，例如下面的YAML：

```
my:
   servers:
       - dev.bar.com
       - foo.bar.com
```

将会转化到下面的属性中:

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

使用Spring DataBinder工具绑定那样的属性（这是@ConfigurationProperties做的事），你需要确定目标bean中有个java.util.List或Set类型的属性，并且需要提供一个setter或使用可变的值初始化它，比如，下面的代码将绑定上面的属性：

```
@ConfigurationProperties(prefix="my")
public class Config {
    private List<String> servers = new ArrayList<String>();
    public List<String> getServers() {
        return this.servers;
    }
}
```

23.6.2. 在Spring环境中使用YAML暴露属性

YamlPropertySourceLoader类能够用于将YAML作为一个PropertySource导出到Sprig Environment。这允许你使用熟悉的@Value注解和占位符语法访问YAML属性。

23.6.3. Multi-profile YAML文档

你可以在单个文件中定义多个特定配置（profile-specific）的YAML文档，并通过一个spring.profiles key标示应用的文档。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
```

在上面的例子中，如果development配置被激活，那server.address属性将是127.0.0.1。如果development和production配置（profiles）没有启用，则该属性的值将是192.168.1.100。

23.6.4. YAML缺点

YAML文件不能通过@PropertySource注解加载。所以，在这种情况下，如果需要使用@PropertySource注解的方式加载值，那就要使用properties文件。

23.7. 类型安全的配置属性

使用@Value("${property}")注解注入配置属性有时可能比较笨重，特别是需要使用多个properties或你的数据本身有层次结构。为了控制和校验你的应用配置，Spring Boot提供一个允许强类型beans的替代方法来使用properties。

示例：

```
@Component
@ConfigurationProperties(prefix="connection")
public class ConnectionSettings {
    private String username;
    private InetAddress remoteAddress;
    // ... getters and setters
}
```

当@EnableConfigurationProperties注解应用到你的@Configuration时，任何被@ConfigurationProperties注解的beans将自动被Environment属性配置。这种风格的配置特别适合与SpringApplication的外部YAML配置进行配合使用。

```
# application.yml
connection:
    username: admin
    remoteAddress: 192.168.1.1
# additional configuration as required
```

为了使用@ConfigurationProperties beans，你可以使用与其他任何bean相同的方式注入它们。

```
@Service
public class MyService {
    @Autowired
    private ConnectionSettings connection;
     //...
    @PostConstruct
    public void openConnection() {
        Server server = new Server();
        this.connection.configure(server);
    }
}
```

你可以通过在@EnableConfigurationProperties注解中直接简单的列出属性类来快捷的注册@ConfigurationProperties bean的定义。

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

注：使用@ConfigurationProperties能够产生可被IDEs使用的元数据文件。具体参考Appendix B, Configuration meta-data。

23.7.1. 第三方配置

正如使用@ConfigurationProperties注解一个类，你也可以在@Bean方法上使用它。当你需要绑定属性到不受你控制的第三方组件时，这种方式非常有用。

为了从Environment属性配置一个bean，将@ConfigurationProperties添加到它的bean注册过程：

```
@ConfigurationProperties(prefix = "foo")
@Bean
public FooComponent fooComponent() {
    ...
}
```

和上面ConnectionSettings的示例方式相同，任何以foo为前缀的属性定义都会被映射到FooComponent上。

23.7.2. 松散的绑定（Relaxed binding）

Spring Boot使用一些宽松的规则用于绑定Environment属性到@ConfigurationProperties beans，所以Environment属性名和bean属性名不需要精确匹配。常见的示例中有用的包括虚线分割（比如，context--path绑定到contextPath）和将环境属性转为大写字母（比如，PORT绑定port）。

示例：

```
@Component
@ConfigurationProperties(prefix="person")
public class ConnectionSettings {
    private String firstName;
}
```

下面的属性名都能用于上面的@ConfigurationProperties类：

属性	说明
person.firstName	标准驼峰规则
person.first-name	虚线表示，推荐用于.properties和.yml文件中
PERSON_FIRST_NAME	大写形式，使用系统环境变量时推荐

Spring会尝试强制外部的应用属性在绑定到@ConfigurationProperties beans时类型是正确的。如果需要自定义类型转换，你可以提供一个ConversionService bean（bean id为conversionService）或自定义属性编辑器（通过一个CustomEditorConfigurer bean）。

23.7.3. @ConfigurationProperties校验

Spring Boot将尝试校验外部的配置，默认使用JSR-303（如果在classpath路径中）。你可以轻松的为你的@ConfigurationProperties类添加JSR-303 javax.validation约束注解：

```
@Component
@ConfigurationProperties(prefix="connection")
public class ConnectionSettings {
    @NotNull
    private InetAddress remoteAddress;
    // ... getters and setters
}
```

你也可以通过创建一个叫做configurationPropertiesValidator的bean来添加自定义的Spring Validator。

注：spring-boot-actuator模块包含一个暴露所有@ConfigurationProperties beans的端点。简单地将你的web浏览器指向/configprops或使用等效的JMX端点。具体参考Production ready features。

24. Profiles

Spring Profiles提供了一种隔离应用程序配置的方式，并让这些配置只能在特定的环境下生效。任何@Component或@Configuration都能被@Profile标记，从而限制加载它的时机。

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

以正常的Spring方式，你可以使用一个spring.profiles.active的Environment属性来指定哪个配置生效。你可以使用平常的任何方式来指定该属性，例如，可以将它包含到你的application.properties中：

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

或使用命令行开关：

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

24.1. 添加激活的配置(profiles)

spring.profiles.active属性和其他属性一样都遵循相同的排列规则，最高的PropertySource获胜。也就是说，你可以在application.properties中指定生效的配置，然后使用命令行开关替换它们。

有时，将特定的配置属性添加到生效的配置中而不是替换它们是有用的。spring.profiles.include属性可以用来无条件的添加生效的配置。SpringApplication的入口点也提供了一个用于设置额外配置的Java API（比如，在那些通过spring.profiles.active属性生效的配置之上）：参考setAdditionalProfiles()方法。

示例：当一个应用使用下面的属性，并用 `--spring.profiles.active=prod` 开关运行，那proddb和prodmq配置也会生效：

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

注：spring.profiles属性可以定义到一个YAML文档中，用于决定什么时候该文档被包含进配置中。具体参考Section 63.6, “Change configuration depending on the environment”

24.2.以编程方式设置profiles

在应用运行前，你可以通过调用SpringApplication.setAdditionalProfiles(…)方法，以编程的方式设置生效的配置。使用Spring的ConfigurableEnvironment接口激动配置也是可行的。

24.3. Profile特定配置文件

application.properties（或application.yml）和通过@ConfigurationProperties引用的文件这两种配置特定变种都被当作文件来加载的，具体参考Section 23.3, “Profile specific properties”。

25. 日志

Spring Boot内部日志系统使用的是Commons Logging，但开放底层的日志实现。默认为会Java Util Logging, Log4J, Log4J2和Logback提供配置。每种情况下都会预先配置使用控制台输出，也可以使用可选的文件输出。

默认情况下，如果你使用'Starter POMs'，那么就会使用Logback记录日志。为了确保那些使用Java Util Logging, Commons Logging, Log4J或SLF4J的依赖库能够正常工作，正确的Logback路由也被包含进来。

注：如果上面的列表看起来令人困惑，不要担心，Java有很多可用的日志框架。通常，你不需要改变日志依赖，Spring Boot默认的就能很好的工作。

25.1. 日志格式

Spring Boot默认的日志输出格式如下：

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main] org.apache.catalina.core.Sta
ndardEngine  : Starting Servlet Engine: Apache Tomcat/7.0.52
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhos
t].[/]       : Initializing Spring embedded WebApplicationContext
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoade
r            : Root WebApplicationContext: initialization completed in 1358 ms
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistratio
nBean        : Mapping servlet: 'dispatcherServlet' to [/]
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegis
trationBean  : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
```

输出的节点（items）如下：

1. 日期和时间 - 精确到毫秒，且易于排序。
2. 日志级别 - ERROR, WARN, INFO, DEBUG 或 TRACE。
3. Process ID。
4. 一个用于区分实际日志信息开头的---分隔符。
5. 线程名 - 包括在方括号中（控制台输出可能会被截断）。
6. 日志名 - 通常是源class的类名（缩写）。
7. 日志信息。

25.2. 控制台输出

默认的日志配置会在写日志消息时将它们回显到控制台。默认，ERROR, WARN和INFO级别的消息会被记录。可以在启动应用时，通过 `--debug` 标识开启控制台的DEBUG级别日志记录。

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

如果你的终端支持ANSI，为了增加可读性将会使用彩色的日志输出。你可以设置 `spring.output.ansi.enabled` 为一个支持的值来覆盖自动检测。

25.3. 文件输出

默认情况下，Spring Boot只会将日志记录到控制台而不会写进日志文件。如果除了输出到控制台你还想写入到日志文件，那你需要设置 `logging.file` 或 `logging.path` 属性（例如在你的application.properties中）。

下表显示如何组合使用 `logging.*` ：

logging.file	**logging.path**	示例	描述
(none)	(none)		只记录到控制台
Specific file	(none)	my.log	写到特定的日志文件里，名称可以是一个精确的位置或相对于当前目录
(none)	Specific folder	/var/log	写到特定文件夹下的spring.log里，名称可以是一个精确的位置或相对于当前目录

日志文件每达到10M就会被轮换（分割），和控制台一样，默认记录ERROR, WARN和INFO级别的信息。

25.4. 日志级别

所有支持的日志系统在Spring的Environment（例如在application.properties里）都有通过'logging.level.*=LEVEL'（'LEVEL'是TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF中的一个）设置的日志级别。

示例：application.properties

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

25.5. 自定义日志配置

通过将适当的库添加到classpath，可以激活各种日志系统。然后在classpath的根目录(root)或通过Spring Environment的 `logging.config` 属性指定的位置提供一个合适的配置文件来达到进一步的定制（注意由于日志是在ApplicationContext被创建之前初始化的，所以不可能在Spring的@Configuration文件中，通过@PropertySources控制日志。系统属性和平常的Spring Boot外部配置文件能正常工作）。

根据你的日志系统，下面的文件会被加载：

日志系统	定制
Logback	logback.xml
Log4j	log4j.properties或log4j.xml
Log4j2	log4j2.xml
JDK (Java Util Logging)	logging.properties

为了帮助定制一些其他的属性，从Spring的Envrionment转换到系统属性：

Spring Environment	System Property	评价
logging.file	LOG_FILE	如果定义，在默认的日志配置中使用
logging.path	LOG_PATH	如果定义，在默认的日志配置中使用
PID	PID	当前的处理进程(process)ID（如果能够被发现且还没有作为操作系统环境变量被定义）

所有支持的日志系统在解析它们的配置文件时都能查询系统属性。具体可以参考spring-boot.jar中的默认配置。

注：在运行可执行的jar时，Java Util Logging有类加载问题，我们建议你尽可能避免使用它。

26. 开发Web应用

Spring Boot非常适合开发web应用程序。你可以使用内嵌的Tomcat，Jetty或Undertow轻轻松松地创建一个HTTP服务器。大多数的web应用都使用spring-boot-starter-web模块进行快速搭建和运行。

26.1. Spring Web MVC框架

Spring Web MVC框架（通常简称为"Spring MVC"）是一个富"模型，视图，控制器"的web框架。 Spring MVC允许你创建特定的@Controller或@RestController beans来处理传入的HTTP请求。 使用@RequestMapping注解可以将控制器中的方法映射到相应的HTTP请求。

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) {
        // ...
    }
}
```

26.1.1. Spring MVC自动配置

Spring Boot为Spring MVC提供适用于多数应用的自动配置功能。在Spring默认基础上，自动配置添加了以下特性：

1. 引入ContentNegotiatingViewResolver和BeanNameViewResolver beans。
2. 对静态资源的支持，包括对WebJars的支持。
3. 自动注册Converter，GenericConverter，Formatter beans。
4. 对HttpMessageConverters的支持。
5. 自动注册MessageCodeResolver。
6. 对静态index.html的支持。
7. 对自定义Favicon的支持。

如果想全面控制Spring MVC，你可以添加自己的@Configuration，并使用@EnableWebMvc对其注解。如果想保留Spring Boot MVC的特性，并只是添加其他的MVC配置(拦截器，formatters，视图控制器等)，你可以添加自己的WebMvcConfigurerAdapter类型的@Bean（不使用@EnableWebMvc注解）。

26.1.2. HttpMessageConverters

Spring MVC使用HttpMessageConverter接口转换HTTP请求和响应。合理的缺省值被包含的恰到好处（out of the box），例如对象可以自动转换为JSON（使用Jackson库）或XML（如果Jackson XML扩展可用则使用它，否则使用JAXB）。字符串默认使用UTF-8编码。

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

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

@Configuration
public class MyConfiguration {

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

任何在上下文中出现的HttpMessageConverter bean将会添加到converters列表，你可以通过这种方式覆盖默认的转换器（converters）。

26.1.3. MessageCodesResolver

Spring MVC有一个策略，用于从绑定的errors产生用来渲染错误信息的错误码：MessageCodesResolver。如果设置 `spring.mvc.message-codes-resolver.format` 属性为 `PREFIX_ERROR_CODE` 或 `POSTFIX_ERROR_CODE` （具体查看 `DefaultMessageCodesResolver.Format` 枚举值），Spring Boot会为你创建一个MessageCodesResolver。

26.1.4. 静态内容

默认情况下，Spring Boot从classpath下一个叫/static（/public，/resources或/META-INF/resources）的文件夹或从ServletContext根目录提供静态内容。这使用了Spring MVC的ResourceHttpRequestHandler，所以你可以通过添加自己的WebMvcConfigurerAdapter并覆写addResourceHandlers方法来改变这个行为（加载静态文件）。

在一个单独的web应用中，容器默认的servlet是开启的，如果Spring决定不处理某些请求，默认的servlet作为一个回退（降级）将从ServletContext根目录加载内容。大多数时候，这不会发生（除非你修改默认的MVC配置），因为Spring总能够通过DispatcherServlet处理请求。

此外，上述标准的静态资源位置有个例外情况是Webjars内容。任何在/webjars/**路径下的资源都将从jar文件中提供，只要它们以Webjars的格式打包。

注：如果你的应用将被打包成jar，那就不要使用src/main/webapp文件夹。尽管该文件夹是一个共同的标准，但它仅在打包成war的情况下起作用，并且如果产生一个jar，多数构建工具都会静悄悄的忽略它。

26.1.5. 模板引擎

正如REST web服务，你也可以使用Spring MVC提供动态HTML内容。Spring MVC支持各种各样的模板技术，包括Velocity, FreeMarker和JSPs。很多其他的模板引擎也提供它们自己的Spring MVC集成。

Spring Boot为以下的模板引擎提供自动配置支持：

1. FreeMarker
2. Groovy
3. Thymeleaf
4. Velocity

注：如果可能的话，应该忽略JSPs，因为在内嵌的servlet容器使用它们时存在一些已知的限制。

当你使用这些引擎的任何一种，并采用默认的配置，你的模板将会从src/main/resources/templates目录下自动加载。

注：IntelliJ IDEA根据你运行应用的方式会对classpath进行不同的整理。在IDE里通过main方法运行你的应用跟从Maven或Gradle或打包好的jar中运行相比会导致不同的顺序。这可能导致Spring Boot不能从classpath下成功地找到模板。如果遇到这个问题，你可以在IDE里重新对classpath进行排序，将模块的类和资源放到第一位。或者，你可以配置模块的前缀为classpath*:/templates/，这样会查找classpath下的所有模板目录。

26.1.6. 错误处理

Spring Boot默认提供一个/error映射用来以合适的方式处理所有的错误，并且它在servlet容器中注册了一个全局的 错误页面。对于机器客户端（相对于浏览器而言，浏览器偏重于人的行为），它会产生一个具有详细错误，HTTP状态，异常信息的JSON响应。对于浏览器客户端，它会产生一个白色标签样式（whitelabel）的错误视图，该视图将以HTML格式显示同样的数据（可以添加一个解析为erro的View来自定义它）。为了完全替换默认的行为，你可以实现ErrorController，并注册一个该类型的bean定义，或简单地添加一个ErrorAttributes类型的bean以使用现存的机制，只是替换显示的内容。

如果在某些条件下需要比较多的错误页面，内嵌的servlet容器提供了一个统一的Java DSL（领域特定语言）来自定义错误处理。 示例：

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer(){
    return new MyCustomizer();
}

// ...
private static class MyCustomizer implements EmbeddedServletContainerCustomizer {
    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }
}
```

你也可以使用常规的Spring MVC特性来处理错误，比如@ExceptionHandler方法和@ControllerAdvice。ErrorController将会捡起任何没有处理的异常。

N.B. 如果你为一个路径注册一个ErrorPage，最终被一个过滤器（Filter）处理（对于一些非Spring web框架，像Jersey和Wicket这很常见），然后过滤器需要显式注册为一个ERROR分发器（dispatcher）。

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

注：默认的FilterRegistrationBean没有包含ERROR分发器类型。

26.1.7. Spring HATEOAS

如果你正在开发一个使用超媒体的RESTful API，Spring Boot将为Spring HATEOAS提供自动配置，这在多数应用中都工作良好。自动配置替换了对使用@EnableHypermediaSupport的需求，并注册一定数量的beans来简化构建基于超媒体的应用，这些beans包括一个LinkDiscoverer和配置好的用于将响应正确编排为想要的表示的ObjectMapper。ObjectMapper可以根据spring.jackson.*属性或一个存在的Jackson2ObjectMapperBuilder bean进行自定义。

通过使用@EnableHypermediaSupport，你可以控制Spring HATEOAS的配置。注意这会禁用上述的对ObjectMapper的自定义。

26.2. JAX-RS和Jersey

如果喜欢JAX-RS为REST端点提供的编程模型，你可以使用可用的实现替代Spring MVC。如果在你的应用上下文中将Jersey 1.x和Apache Celtix的Servlet或Filter注册为一个@Bean，那它们工作的相当好。Jersey 2.x有一些原生的Spring支持，所以我们会在Spring Boot为它提供自动配置支持，连同一个启动器（starter）。

想要开始使用Jersey 2.x只需要加入spring-boot-starter-jersey依赖，然后你需要一个ResourceConfig类型的@Bean，用于注册所有的端点（endpoints）。

```
@Component
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        register(Endpoint.class);
    }
}
```

所有注册的端点都应该被@Components和HTTP资源annotations（比如@GET）注解。

```
@Component
@Path("/hello")
public class Endpoint {
    @GET
    public String message() {
        return "Hello";
    }
}
```

由于Endpoint是一个Spring组件（@Component），所以它的生命周期受Spring管理，并且你可以使用@Autowired添加依赖及使用@Value注入外部配置。Jersey servlet将被注册，并默认映射到/*。你可以将@ApplicationPath添加到ResourceConfig来改变该映射。

默认情况下，Jersey将在一个ServletRegistrationBean类型的@Bean中被设置成名称为jerseyServletRegistration的Servlet。通过创建自己的相同名称的bean，你可以禁止或覆盖这个bean。你也可以通过设置 `spring.jersey.type=filter` 来使用一个Filter代替Servlet（在这种情况下，被覆盖或替换的@Bean是jerseyFilterRegistration）。该servlet有@Order属性，你可以通过 `spring.jersey.filter.order` 进行设置。不管是Servlet还是Filter注册都可以使用spring.jersey.init.*定义一个属性集合作为初始化参数传递过去。

这里有一个Jersey示例，你可以查看如何设置相关事项。

26.3. 内嵌servlet容器支持

Spring Boot支持内嵌的Tomcat, Jetty和Undertow服务器。多数开发者只需要使用合适的'Starter POM'来获取一个完全配置好的实例即可。默认情况下，内嵌的服务器会在8080端口监听HTTP请求。

26.3.1. Servlets和Filters

当使用内嵌的servlet容器时，你可以直接将servlet和filter注册为Spring的beans。在配置期间，如果你想引用来自application.properties的值，这是非常方便的。默认情况下，如果上下文只包含单一的Servlet，那它将被映射到根路径（/）。在多Servlet beans的情况下，bean的名称将被用作路径的前缀。过滤器会被映射到/*。

如果基于约定（convention-based）的映射不够灵活，你可以使用ServletRegistrationBean和FilterRegistrationBean类实现完全的控制。如果你的bean实现了ServletContextInitializer接口，也可以直接注册它们。

26.3.2. EmbeddedWebApplicationContext

Spring Boot底层使用了一个新的ApplicationContext类型，用于对内嵌servlet容器的支持。EmbeddedWebApplicationContext是一个特殊类型的WebApplicationContext，它通过搜索一个单一的EmbeddedServletContainerFactory bean来启动自己。通常，TomcatEmbeddedServletContainerFactory，JettyEmbeddedServletContainerFactory或UndertowEmbeddedServletContainerFactory将被自动配置。

注：你通常不需要知道这些实现类。大多数应用将被自动配置，并根据你的行为创建合适的ApplicationContext和EmbeddedServletContainerFactory。

26.3.3. 自定义内嵌servlet容器

常见的Servlet容器设置可以通过Spring Environment属性进行配置。通常，你会把这些属性定义到application.properties文件中。 常见的服务器设置包括：

1. server.port - 进来的HTTP请求的监听端口号
2. server.address - 绑定的接口地址
3. server.sessionTimeout - session超时时间

具体参考ServerProperties。

- 编程方式的自定义

如果需要以编程的方式配置内嵌的servlet容器，你可以注册一个实现EmbeddedServletContainerCustomizer接口的Spring bean。EmbeddedServletContainerCustomizer提供对ConfigurableEmbeddedServletContainer的访问，ConfigurableEmbeddedServletContainer包含很多自定义的setter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

如果上面的自定义手法过于受限，你可以自己注册TomcatEmbeddedServletContainerFactory，JettyEmbeddedServletContainerFactory或UndertowEmbeddedServletContainerFactory。

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

很多可选的配置都提供了setter方法，也提供了一些受保护的钩子方法以满足你的某些特殊需求。具体参考相关文档。

26.3.4. JSP的限制

在内嵌的servlet容器中运行一个Spring Boot应用时（并打包成一个可执行的存档archive），容器对JSP的支持有一些限制。

1. tomcat只支持war的打包方式，不支持可执行的jar。
2. 内嵌的Jetty目前不支持JSPs。
3. Undertow不支持JSPs。

这里有个JSP示例，你可以查看如何设置相关事项。

27. 安全

如果Spring Security在classpath下，那么web应用默认对所有的HTTP路径（也称为终点，端点，表示API的具体网址）使用'basic'认证。为了给web应用添加方法级别的保护，你可以添加@EnableGlobalMethodSecurity并使用想要的设置。其他信息参考Spring Security Reference。

默认的AuthenticationManager有一个单一的user（'user'的用户名和随机密码会在应用启动时以INFO日志级别打印出来）。如下：

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

注：如果你对日志配置进行微调，确保`org.springframework.boot.autoconfigure.security`类别能记录INFO信息，否则默认的密码不会被打印。

你可以通过提供`security.user.password`改变默认的密码。这些和其他有用的属性通过SecurityProperties（以security为前缀的属性）被外部化了。

默认的安全配置（security configuration）是在SecurityAutoConfiguration和导入的类中实现的（SpringBootWebSecurityConfiguration用于web安全，AuthenticationManagerConfiguration用于与非web应用也相关的认证配置）。你可以添加一个@EnableWebSecurity bean来彻底关掉Spring Boot的默认配置。为了对它进行自定义，你需要使用外部的属性配置和WebSecurityConfigurerAdapter类型的beans（比如，添加基于表单的登陆）。在Spring Boot示例里有一些安全相关的应用可以带你体验常见的用例。

在一个web应用中你能得到的基本特性如下：

1. 一个使用内存存储的AuthenticationManager bean和唯一的user（查看SecurityProperties.User获取user的属性）。
2. 忽略（不保护）常见的静态资源路径（`/css/**, /js/**, /images/**`和`**/favicon.ico`）。
3. 对其他的路径实施HTTP Basic安全保护。
4. 安全相关的事件会发布到Spring的ApplicationEventPublisher（成功和失败的认证，拒绝访问）。
5. Spring Security提供的常见底层特性（HSTS, XSS, CSRF, 缓存）默认都被开启。

上述所有特性都能打开和关闭，或使用外部的配置进行修改（security.*）。为了覆盖访问规则（access rules）而不改变其他自动配置的特性，你可以添加一个使用@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)注解的WebSecurityConfigurerAdapter类型的@Bean。

如果Actuator也在使用，你会发现：

1. 即使应用路径不受保护，被管理的路径也会受到保护。
2. 安全相关的事件被转换为AuditEvents（审计事件），并发布给AuditService。
3. 默认的用户有ADMIN和USER的角色。

使用外部属性能够修改Actuator（执行器）的安全特性（management.security.*）。为了覆盖应用程序的访问规则，你可以添加一个WebSecurityConfigurerAdapter类型的@Bean。同时，如果不想覆盖执行器的访问规则，你可以使用@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)注解该bean，否则使用@Order(ManagementServerProperties.ACCESS_OVERRIDE_ORDER)注解该bean。

28. 使用SQL数据库

Spring框架为使用SQL数据库提供了广泛的支持。从使用JdbcTemplate直接访问JDBC到完全的对象关系映射技术，比如Hibernate。Spring Data提供一个额外的功能，直接从接口创建Repository实现，并使用约定从你的方法名生成查询。

28.1. 配置DataSource

Java的javax.sql.DataSource接口提供了一个标准的使用数据库连接的方法。传统做法是，一个DataSource使用一个URL连同相应的证书去初始化一个数据库连接。

28.1.1. 对内嵌数据库的支持

开发应用时使用内存数据库是很实用的。显而易见地，内存数据库不需要提供持久化存储。你不需要在应用启动时填充数据库，也不需要在应用结束时丢弃数据。

Spring Boot可以自动配置的内嵌数据库包括H2, HSQL和Derby。你不需要提供任何连接URLs，只需要简单的添加你想使用的内嵌数据库依赖。

示例：典型的POM依赖如下：

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

注：对于自动配置的内嵌数据库，你需要依赖spring-jdbc。在示例中，它通过 `spring-boot-starter-data-jpa` 被传递地拉过来了。

28.1.2. 连接到一个生产环境数据库

在生产环境中，数据库连接可以使用DataSource池进行自动配置。下面是选取一个特定实现的算法：

- 由于Tomcat数据源连接池的性能和并发，在tomcat可用时，我们总是优先使用它。
- 如果HikariCP可用，我们将使用它。
- 如果Commons DBCP可用，我们将使用它，但在生产环境不推荐使用它。
- 最后，如果Commons DBCP2可用，我们将使用它。

如果你使用spring-boot-starter-jdbc或spring-boot-starter-data-jpa 'starter POMs'，你将会自动获取对tomcat-jdbc的依赖。

注：其他的连接池可以手动配置。如果你定义自己的DataSource bean，自动配置不会发生。

DataSource配置通过外部配置文件的spring.datasource.*属性控制。示例中，你可能会在application.properties中声明下面的片段：

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

其他可选的配置可以查看DataSourceProperties。同时注意你可以通过spring.datasource.*配置任何DataSource实现相关的特定属性：具体参考你使用的连接池实现的文档。

注：既然Spring Boot能够从大多数数据库的url上推断出driver-class-name，那么你就不需要再指定它了。对于一个将要创建的DataSource连接池，我们需要能够验证Driver是否可用，所以我们会在做任何事情之前检查它。比如，如果你设置spring.datasource.driverClassName=com.mysql.jdbc.Driver，然后这个类就会被加载。

28.1.3. 连接到一个JNDI数据库

如果正在将Spring Boot应用部署到一个应用服务器，你可能想要用应用服务器内建的特性来配置和管理你的DataSource，并使用JNDI访问它。

spring.datasource.jndi-name属性可以用来替代spring.datasource.url，spring.datasource.username和spring.datasource.password去从一个特定的JNDI路径访问DataSource。比如，下面application.properties中的片段展示了如何获取JBoss定义的DataSource：

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

28.2. 使用JdbcTemplate

Spring的JdbcTemplate和NamedParameterJdbcTemplate类是被自动配置的，你可以在自己的beans中通过@Autowire直接注入它们。

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

@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

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

28.3. JPA和Spring Data

Java持久化API是一个允许你将对象映射为关系数据库的标准技术。spring-boot-starter-data-jpa POM提供了一种快速上手的方式。它提供下列关键的依赖：

- Hibernate - 一个非常流行的JPA实现。
- Spring Data JPA - 让实现基于JPA的repositories更容易。
- Spring ORMs - Spring框架的核心ORM支持。

注：我们不想在这涉及太多关于JPA或Spring Data的细节。你可以参考来自spring.io的指南使用JPA获取数据，并阅读Spring Data JPA和Hibernate的参考文档。

28.3.1. 实体类

传统上，JPA实体类被定义到一个persistence.xml文件中。在Spring Boot中，这个文件不是必需的，并被'实体扫描'替代。默认情况下，在你主（main）配置类（被@EnableAutoConfiguration或@SpringBootApplication注解的类）下的所有包都将被查找。

任何被@Entity，@Embeddable或@MappedSuperclass注解的类都将被考虑。一个普通的实体类看起来像下面这样：

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

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

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;

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

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

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

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

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

注：你可以使用@EntityScan注解自定义实体扫描路径。具体参考Section 67.4, “Separate @Entity definitions from Spring configuration”。

28.3.2. Spring Data JPA仓库

Spring Data JPA仓库（repositories）是用来定义访问数据的接口。根据你的方法名，JPA查询会被自动创建。比如，一个CityRepository接口可能声明一个findAllByState(String state)方法，用来查找给定状态的所有城市。

对于比较复杂的查询，你可以使用Spring Data的Query来注解你的方法。

Spring Data仓库通常继承自Repository或CrudRepository接口。如果你使用自动配置，包括在你的主配置类（被@EnableAutoConfiguration或@SpringBootApplication注解的类）的包下的仓库将会被搜索。

下面是一个传统的Spring Data仓库：

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

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

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String country);
}
```

注：我们仅仅触及了Spring Data JPA的表面。具体查看它的参考指南。

28.3.3. 创建和删除JPA数据库

默认情况下，只有在你使用内嵌数据库（H2, HSQL或Derby）时，JPA数据库才会被自动创建。你可以使用spring.jpa.*属性显示的设置JPA。比如，为了创建和删除表你可以将下面的配置添加到application.properties中：

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

注：Hibernate自己内部对创建，删除表支持（如果你恰好记得这回事更好）的属性是hibernate.hbm2ddl.auto。使用spring.jpa.properties.*（前缀在被添加到实体管理器之前会被剥离掉），你可以设置Hibernate本身的属性，比如hibernate.hbm2ddl.auto。示例：`spring.jpa.properties.hibernate.globally_quoted_identifiers=true`将传递hibernate.globally_quoted_identifiers到Hibernate实体管理器。

默认情况下，DDL执行（或验证）被延迟到ApplicationContext启动。这也有一个spring.jpa.generate-ddl标识，如果Hibernate自动配置被激活，那该标识就不会被使用，因为ddl-auto设置粒度更细。

29. 使用NoSQL技术

Spring Data提供其他项目，用来帮你使用各种各样的NoSQL技术，包括MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Couchbase和Cassandra。Spring Boot为Redis, MongoDB, Elasticsearch, Solr和Gemfire提供自动配置。你可以充分利用其他项目，但你需要自己配置它们。具体查看projects.spring.io/spring-data中合适的参考文档。

29.1. Redis

Redis是一个缓存，消息中间件及具有丰富特性的键值存储系统。Spring Boot为Jedis客户端库和由Spring Data Redis提供的基于Jedis客户端的抽象提供自动配置。 `spring-boot-starter-redis` 'Starter POM'为收集依赖提供一种便利的方式。

29.1.1. 连接Redis

你可以注入一个自动配置的RedisConnectionFactory，StringRedisTemplate或普通的跟其他Spring Bean相同的RedisTemplate实例。默认情况下，这个实例将尝试使用localhost:6379连接Redis服务器。

```
@Component
public class MyBean {

    private StringRedisTemplate template;

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

如果你添加一个你自己的任何自动配置类型的@Bean，它将替换默认的（除了RedisTemplate的情况，它是根据bean的名称'redisTemplate'而不是它的类型进行排除的）。如果在classpath路径下存在commons-pool2，默认你会获得一个连接池工厂。

29.2. MongoDB

MongoDB是一个开源的NoSQL文档数据库，它使用一个JSON格式的模式（schema）替换了传统的基于表的关系数据。Spring Boot为使用MongoDB提供了很多便利，包括 `spring-boot-starter-data-mongodb` 'Starter POM'。

29.2.1. 连接MongoDB数据库

你可以注入一个自动配置的 `org.springframework.data.mongodb.MongoDbFactory` 来访问Mongo数据库。默认情况下，该实例将尝试使用URL： `mongodb://localhost/test` 连接一个MongoDB服务器。

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

@Component
public class MyBean {

    private final MongoDbFactory mongo;

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

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

你可以通过设置 `spring.data.mongodb.uri` 来改变该url，或指定一个host/port。比如，你可能会在你的application.properties中设置如下的属性：

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

注：如果没有指定 `spring.data.mongodb.port` ，那将使用默认的端口27017。你可以简单的从上面的示例中删除这一行。如果不使用Spring Data Mongo，你可以注入com.mongodb.Mongo beans而不是使用MongoDbFactory。

如果想全面控制MongoDB连接的建立，你也可以声明自己的MongoDbFactory或Mongo，@Beans。

29.2.2. MongoDBTemplate

Spring Data Mongo提供了一个MongoTemplate类，它的设计和Spring的JdbcTemplate很相似。正如JdbcTemplate一样，Spring Boot会为你自动配置一个bean，你只需简单的注入它即可：

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

@Component
public class MyBean {

    private final MongoTemplate mongoTemplate;

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

具体参考MongoOperations Javadoc。

29.2.3. Spring Data MongoDB仓库

Spring Data的仓库包括对MongoDB的支持。正如上面讨论的JPA仓库，基本的原则是查询会自动基于你的方法名创建。

实际上，不管是Spring Data JPA还是Spring Data MongoDB都共享相同的基础设施。所以你可以使用上面的JPA示例，并假设那个City现在是一个Mongo数据类而不是JPA　@Entity，它将以同样的方式工作。

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

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

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

29.3. Gemfire

Spring Data Gemfire为使用Pivotal Gemfire数据管理平台提供了方便的，Spring友好的工具。Spring Boot提供了一个用于聚集依赖的 `spring-boot-starter-data-gemfire` 'Starter POM'。目前不支持Gemfire的自动配置，但你可以使用一个单一的注解使Spring Data仓库支持它。

29.4. Solr

Apache Solr是一个搜索引擎。Spring Boot为solr客户端库及Spring Data Solr提供的基于solr客户端库的抽象提供了基本的配置。Spring Boot提供了一个用于聚集依赖的 `spring-boot-starter-data-solr` 'Starter POM'。

29.4.1. 连接Solr

你可以像其他Spring beans一样注入一个自动配置的SolrServer实例。默认情况下，该实例将尝试使用 `localhost:8983/solr` 连接一个服务器。

```
@Component
public class MyBean {

    private SolrServer solr;

    @Autowired
    public MyBean(SolrServer solr) {
        this.solr = solr;
    }
    // ...
}
```

如果你添加一个自己的SolrServer类型的@Bean，它将会替换默认的。

29.4.2. Spring Data Solr仓库

Spring Data的仓库包括了对Apache Solr的支持。正如上面讨论的JPA仓库，基本的原则是查询会自动基于你的方法名创建。

实际上，不管是Spring Data JPA还是Spring Data Solr都共享相同的基础设施。所以你可以使用上面的JPA示例，并假设那个City现在是一个@SolrDocument类而不是JPA　@Entity，它将以同样的方式工作。

注：具体参考Spring Data Solr文档。

29.5. Elasticsearch

Elastic Search是一个开源的，分布式，实时搜索和分析引擎。Spring Boot为Elasticsearch及Spring Data Elasticsearch提供的基于它的抽象提供了基本的配置。Spring Boot提供了一个用于聚集依赖的 `spring-boot-starter-data-elasticsearch` 'Starter POM'。

29.5.1. 连接Elasticsearch

你可以像其他Spring beans那样注入一个自动配置的ElasticsearchTemplate或Elasticsearch客户端实例。默认情况下，该实例将尝试连接到一个本地内存服务器（在Elasticsearch项目中的一个NodeClient），但你可以通过设置 `spring.data.elasticsearch.clusterNodes` 为一个以逗号分割的host:port列表来将其切换到一个远程服务器（比如，TransportClient）。

```
@Component
public class MyBean {

    private ElasticsearchTemplate template;

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

如果你添加一个你自己的ElasticsearchTemplate类型的@Bean，它将替换默认的。

29.5.2. Spring Data Elasticseach仓库

Spring Data的仓库包括了对Elasticsearch的支持。正如上面讨论的JPA仓库，基本的原则是查询会自动基于你的方法名创建。

实际上，不管是Spring Data JPA还是Spring Data Elasticsearch都共享相同的基础设施。所以你可以使用上面的JPA示例，并假设那个City现在是一个Elasticsearch @Document类而不是JPA @Entity，它将以同样的方式工作。

注：具体参考Spring Data Elasticsearch文档。

30. 消息

Spring Framework框架为集成消息系统提供了扩展（extensive）支持：从使用JmsTemplate简化JMS　API，到实现一个完整异步消息接收的底层设施。

Spring AMQP提供一个相似的用于'高级消息队列协议'的特征集，并且Spring Boot也为RabbitTemplate和RabbitMQ提供了自动配置选项。

Spring Websocket提供原生的STOMP消息支持，并且Spring Boot通过starters和一些自动配置也提供了对它的支持。

30.1. JMS

javax.jms.ConnectionFactory接口提供了一个标准的用于创建一个javax.jms.Connection的方法，javax.jms.Connection用于和JMS代理（broker）交互。

尽管为了使用JMS，Spring需要一个ConnectionFactory，但通常你不需要直接使用它，而是依赖于上层消息抽象（具体参考Spring框架的相关章节）。Spring Boot也会自动配置发送和接收消息需要的设施（infrastructure）。

30.1.1. HornetQ支持

如果在classpath下发现HornetQ，Spring Boot会自动配置ConnectionFactory。如果需要代理，将会开启一个内嵌的，已经自动配置好的代理（除非显式设置mode属性）。支持的modes有：embedded（显式声明使用一个内嵌的代理，如果该代理在classpath下不可用将导致一个错误），native（使用netty传输协议连接代理）。当后者被配置，Spring Boot配置一个连接到一个代理的ConnectionFactory，该代理运行在使用默认配置的本地机器上。

注：如果使用spring-boot-starter-hornetq，连接到一个已存在的HornetQ实例所需的依赖都会被提供，同时还有用于集成JMS的Spring基础设施。将org.hornetq:hornetq-jms-server添加到你的应用中，你就可以使用embedded模式。

HornetQ配置被spring.hornetq.*中的外部配置属性所控制。例如，你可能在application.properties声明以下片段：

```
spring.hornetq.mode=native
spring.hornetq.host=192.168.1.210
spring.hornetq.port=9876
```

当内嵌代理时，你可以选择是否启用持久化，并且列表中的目标都应该是可用的。这些可以通过一个以逗号分割的列表来指定一些默认的配置项，或定义org.hornetq.jms.server.config.JMSQueueConfiguration或org.hornetq.jms.server.config.TopicConfiguration类型的bean(s)来配置更高级的队列和主题。具体参考HornetQProperties。

没有涉及JNDI查找，目标是通过名字解析的，名字即可以使用HornetQ配置中的name属性，也可以是配置中提供的names。

30.1.2. ActiveQ支持

如果发现ActiveMQ在classpath下可用，Spring Boot会配置一个ConnectionFactory。如果需要代理，将会开启一个内嵌的，已经自动配置好的代理（只要配置中没有指定代理URL）。

ActiveMQ配置是通过spring.activemq.*中的外部配置来控制的。例如，你可能在application.properties中声明下面的片段：

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

具体参考ActiveMQProperties。

默认情况下，如果目标还不存在，ActiveMQ将创建一个，所以目标是通过它们提供的名称解析出来的。

30.1.3. 使用JNDI ConnectionFactory

如果你在一个应用服务器中运行你的应用，Spring Boot将尝试使用JNDI定位一个JMS ConnectionFactory。默认情况会检查java:/JmsXA和java:/ XAConnectionFactory。如果需要的话，你可以使用spring.jms.jndi-name属性来指定一个替代位置。

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

30.1.4. 发送消息

Spring的JmsTemplate会被自动配置，你可以将它直接注入到你自己的beans中：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
@Component
public class MyBean {
private final JmsTemplate jmsTemplate;
@Autowired
public MyBean(JmsTemplate jmsTemplate) {
this.jmsTemplate = jmsTemplate;
}
// ...
}
```

注：JmsMessagingTemplate(Spring4.1新增的)也可以使用相同的方式注入

30.1.5. 接收消息

当JMS基础设施能够使用时，任何bean都能够被@JmsListener注解，以创建一个监听者端点。如果没有定义JmsListenerContainerFactory，一个默认的将会被自动配置。下面的组件在someQueue目标上创建一个监听者端点。

```
@Component
public class MyBean {
@JmsListener(destination = "someQueue")
public void processMessage(String content) {
// ...
}
}
```

具体查看@EnableJms javadoc。

31. 发送邮件

Spring框架使用JavaMailSender接口为发送邮件提供了一个简单的抽象，并且Spring Boot也为它提供了自动配置和一个starter模块。 具体查看JavaMailSender参考文档。

如果spring.mail.host和相关的库（通过spring-boot-starter-mail定义）都存在，一个默认的JavaMailSender将被创建。该sender可以通过spring.mail命名空间下的配置项进一步自定义，具体参考MailProperties。

32. 使用JTA处理分布式事务

Spring Boot使用一个Atomkos或Bitronix的内嵌事务管理器来支持跨多个XA资源的分布式JTA事务。当部署到一个恰当的J2EE应用服务器时也会支持JTA事务。

当发现一个JTA环境时，Spring Boot将使用Spring的JtaTransactionManager来管理事务。自动配置的JMS，DataSource和JPA beans将被升级以支持XA事务。你可以使用标准的Spring idioms，比如@Transactional，来参与到一个分布式事务中。如果你处于JTA环境里，但仍旧想使用本地事务，你可以将spring.jta.enabled属性设置为false来禁用JTA自动配置功能。

32.1. 使用一个Atomikos事务管理器

Atomikos是一个非常流行的开源事务管理器，它可以嵌入到你的Spring Boot应用中。你可以使用 `spring-boot-starter-jta-atomikos` Starter POM去获取正确的Atomikos库。Spring Boot会自动配置Atomikos，并将合适的depends-on应用到你的Spring Beans上，确保它们以正确的顺序启动和关闭。

默认情况下，Atomikos事务日志将被记录在应用home目录（你的应用jar文件放置的目录）下的transaction-logs文件夹中。你可以在application.properties文件中通过设置spring.jta.log-dir属性来自定义该目录。以spring.jta.开头的属性能用来自定义Atomikos的UserTransactionServiceImpl实现。具体参考AtomikosProperties javadoc。

注：为了确保多个事务管理器能够安全地和相应的资源管理器配合，每个Atomikos实例必须设置一个唯一的ID。默认情况下，该ID是Atomikos实例运行的机器上的IP地址。为了确保生产环境中该ID的唯一性，你需要为应用的每个实例设置不同的spring.jta.transaction-manager-id属性值。

32.2. 使用一个Bitronix事务管理器

Bitronix是另一个流行的开源JTA事务管理器实现。你可以使用 `spring-boot-starter-jta-bitronix` starter POM为项目添加合适的Birtronix依赖。和Atomikos类似，Spring Boot将自动配置Bitronix，并对beans进行后处理（post-process）以确保它们以正确的顺序启动和关闭。

默认情况下，Bitronix事务日志将被记录到应用home目录下的transaction-logs文件夹中。通过设置spring.jta.log-dir属性，你可以自定义该目录。以spring.jta.开头的属性将被绑定到bitronix.tm.Configuration bean，你可以通过这完成进一步的自定义。具体参考Bitronix文档。

注：为了确保多个事务管理器能够安全地和相应的资源管理器配合，每个Bitronix实例必须设置一个唯一的ID。默认情况下，该ID是Bitronix实例运行的机器上的IP地址。为了确保生产环境中该ID的唯一性，你需要为应用的每个实例设置不同的spring.jta.transaction-manager-id属性值。

32.3. 使用一个J2EE管理的事务管理器

如果你将Spring Boot应用打包为一个war或ear文件，并将它部署到一个J2EE的应用服务器中，那你就能使用应用服务器内建的事务管理器。Spring Boot将尝试通过查找常见的JNDI路径（java:comp/UserTransaction, java:comp/TransactionManager等）来自动配置一个事务管理器。如果使用应用服务器提供的事务服务，你通常需要确保所有的资源都被应用服务器管理，并通过JNDI暴露出去。Spring Boot通过查找JNDI路径java:/JmsXA或java:/XAConnectionFactory获取一个ConnectionFactory来自动配置JMS，并且你可以使用spring.datasource.jndi-name属性配置你的DataSource。

32.4. 混合XA和non-XA的JMS连接

当使用JTA时，主要的JMS ConnectionFactory bean将是XA aware，并参与到分布式事务中。有些情况下，你可能需要使用non-XA的ConnectionFactory去处理一些JMS消息。例如，你的JMS处理逻辑可能比XA超时时间长。

如果想使用一个non-XA的ConnectionFactory，你可以注入nonXaJmsConnectionFactory bean而不是@Primary jmsConnectionFactory bean。为了保持一致，jmsConnectionFactory bean将以别名xaJmsConnectionFactor来被使用。

示例如下：

```
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;
// Inject the XA aware ConnectionFactory (uses the alias and injects the same as above)

@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;
// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

33. Spring集成

Spring集成提供基于消息和其他协议的，比如HTTP，TCP等的抽象。如果Spring集成在classpath下可用，它将会通过@EnableIntegration注解被初始化。如果classpath下'spring-integration-jmx'可用，则消息处理统计分析将被通过JMX发布出去。具体参考IntegrationAutoConfiguration类。

34. 基于JMX的监控和管理

Java管理扩展（JMX）提供了一个标准的用于监控和管理应用的机制。默认情况下，Spring Boot将创建一个id为‘mbeanServer’的MBeanServer，并导出任何被Spring JMX注解（@ManagedResource,@ManagedAttribute,@ManagedOperation）的beans。具体参考JmxAutoConfiguration类。

35. 测试

Spring Boot提供很多有用的测试应用的工具。spring-boot-starter-test POM提供Spring Test，JUnit，Hamcrest和Mockito的依赖。在spring-boot核心模块org.springframework.boot.test包下也有很多有用的测试工具。

35.1. 测试作用域依赖

如果使用spring-boot-starter-test ‘Starter POM’（在test作用域内），你将发现下列被提供的库：

- Spring Test - 对Spring应用的集成测试支持
- JUnit - 事实上的(de-facto)标准，用于Java应用的单元测试。
- Hamcrest - 一个匹配对象的库（也称为约束或前置条件），它允许assertThat等JUnit类型的断言。
- Mockito - 一个Java模拟框架。

这也有一些我们写测试用例时经常用到的库。如果它们不能满足你的要求，你可以随意添加其他的测试用的依赖库。

35.2. 测试Spring应用

依赖注入最大的优点就是它能够让你的代码更容易进行单元测试。你只需简单的通过new操作符实例化对象，而不需要涉及Spring。你也可以使用模拟对象替换真正的依赖。

你常常需要在进行单元测试后，开始集成测试（在这个过程中只需要涉及到Spring的ApplicationContext）。在执行集成测试时，不需要部署应用或连接到其他基础设施是非常有用的。

Spring框架包含一个dedicated测试模块，用于这样的集成测试。你可以直接声明对org.springframework:spring-test的依赖，或使用spring-boot-starter-test ‘Starter POM’以透明的方式拉取它。

如果你以前没有使用过spring-test模块，可以查看Spring框架参考文档中的相关章节。

35.3. 测试Spring Boot应用

一个Spring Boot应用只是一个Spring ApplicationContext，所以在测试它时除了正常情况下处理一个vanilla Spring　context外不需要做其他特别事情。唯一需要注意的是，如果你使用SpringApplication创建上下文，外部配置，日志和Spring Boot的其他特性只会在默认的上下文中起作用。

Spring Boot提供一个@SpringApplicationConfiguration注解用来替换标准的spring-test @ContextConfiguration注解。如果使用@SpringApplicationConfiguration来设置你的测试中使用的ApplicationContext，它最终将通过SpringApplication创建，并且你将获取到Spring Boot的其他特性。

示例如下：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
public class CityRepositoryIntegrationTests {
@Autowired
CityRepository repository;
// ...
}
```

提示：上下文加载器会通过查找@WebIntegrationTest或@WebAppConfiguration注解来猜测你想测试的是否是web应用（例如，是否使用MockMVC，MockMVC和@WebAppConfiguration是spring-test的一部分）。

如果想让一个web应用启动，并监听它的正常的端口，你可以使用HTTP来测试它（比如，使用RestTemplate），并使用@WebIntegrationTest注解你的测试类（或它的一个父类）。这很有用，因为它意味着你可以对你的应用进行全栈测试，但在一次HTTP交互后，你需要在你的测试类中注入相应的组件并使用它们断言应用的内部状态。

示例：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
@WebIntegrationTest
public class CityRepositoryIntegrationTests {
@Autowired
CityRepository repository;
RestTemplate restTemplate = new TestRestTemplate();
// ... interact with the running server
}
```

注：Spring测试框架在每次测试时会缓存应用上下文。因此，只要你的测试共享相同的配置，不管你实际运行多少测试，开启和停止服务器只会发生一次。

你可以为@WebIntegrationTest添加环境变量属性来改变应用服务器端口号，比如@WebIntegrationTest("server.port:9000")。此外，你可以将server.port和management.port属性设置为 0 来让你的集成测试使用随机的端口号，例如：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MyApplication.class)
@WebIntegrationTest({"server.port=0", "management.port=0"})
public class SomeIntegrationTests {
// ...
}
```

可以查看Section 64.4, “Discover the HTTP port at runtime”，它描述了如何在测试期间发现分配的实际端口。

35.3.1. 使用Spock测试Spring Boot应用

如果期望使用Spock测试一个Spring Boot应用，你应该将Spock的spock-spring模块依赖添加到应用的构建中。spock-spring将Spring的测试框架集成到了Spock里。

注意你不能使用上述提到的@SpringApplicationConfiguration注解，因为Spock找不到@ContextConfiguration元注解。为了绕过该限制，你应该直接使用@ContextConfiguration注解，并使用Spring Boot特定的上下文加载器来配置它。

```
@ContextConfiguration(loader = SpringApplicationContextLoader.class)
class ExampleSpec extends Specification {
// ...
}
```

注：上面描述的注解在Spock中可以使用，比如，你可以使用@WebIntegrationTest注解你的Specification以满足测试需要。

35.4. 测试工具

打包进spring-boot的一些有用的测试工具类。

35.4.1. ConfigFileApplicationContextInitializer

ConfigFileApplicationContextInitializer是一个ApplicationContextInitializer，可以用来测试加载Spring Boot的application.properties文件。当不需要使用@SpringApplicationConfiguration提供的全部特性时，你可以使用它。

```
@ContextConfiguration(classes = Config.class,initializers = ConfigFileApplicationConte
xtInitializer.class)
```

35.4.2. EnvironmentTestUtils

EnvironmentTestUtils允许你快速添加属性到一个ConfigurableEnvironment或ConfigurableApplicationContext。只需简单的使用key=value字符串调用它：```java
EnvironmentTestUtils.addEnvironment(env, "org=Spring", "name=Boot");

35.4.3. OutputCapture

OutputCapture是一个JUnit Rule，用于捕获System.out和System.err输出。只需简单的将捕获声明为一个@Rule，并使用toString()断言：

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.OutputCapture;
import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {
@Rule
public OutputCapture capture = new OutputCapture();
@Test
public void testName() throws Exception {
System.out.println("Hello World!");
assertThat(capture.toString(), containsString("World"));
}
}
```

35.4.4. TestRestTemplate

TestRestTemplate是一个方便进行集成测试的Spring RestTemplate子类。你会获取到一个普通的模板或一个发送基本HTTP认证（使用用户名和密码）的模板。在任何情况下，这些模板都表现出对测试友好：不允许重定向（这样你可以对响应地址进行断言），忽略cookies（这样模板就是无状态的），对于服务端错误不会抛出异常。推荐使用Apache HTTP Client(4.3.2或更好的版本)，但不强制这样做。如果在classpath下存在Apache HTTP Client，TestRestTemplate将以正确配置的client进行响应。

```
public class MyTest {
RestTemplate template = new TestRestTemplate();
@Test
public void testRequest() throws Exception {
HttpHeaders headers = template.getForEntity("http://myhost.com", String.class).getHead
ers();
assertThat(headers.getLocation().toString(), containsString("myotherhost"));
}
}
```

36. 开发自动配置和使用条件

如果你在一个开发者共享库的公司工作，或你在从事一个开源或商业型的库，你可能想要开发自己的auto-configuration。Auto-configuration类能够在外部的jars中绑定，并仍能被Spring Boot发现。

36.1. 理解auto-configured beans

从底层来讲，auto-configured是使用标准的@Configuration实现的类，另外的@Conditional注解用来约束在什么情况下使用auto-configuration。通常auto-configuration类使用@ConditionalOnClass和@ConditionalOnMissingBean注解。这是为了确保只有在相关的类被发现，和你没有声明自己的@Configuration时才应用auto-configuration。

你可以浏览spring-boot-autoconfigure的源码，查看我们提供的@Configuration类（查看META-INF/spring.factories文件）。

36.2. 定位auto-configuration候选者

Spring Boot会检查你发布的jar中是否存在META-INF/spring.factories文件。该文件应该列出以EnableAutoConfiguration为key的配置类：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

如果配置需要应用特定的顺序，你可以使用@AutoConfigureAfter或@AutoConfigureBefore注解。例如，你想提供web-specific配置，你的类就需要应用在WebMvcAutoConfiguration后面。

36.3. Condition注解

你几乎总是需要在你的auto-configuration类里添加一个或更多的@Condition注解。@ConditionalOnMissingBean注解是一个常见的示例，它经常用于允许开发者覆盖auto-configuration，如果他们不喜欢你提供的默认行为。

Spring Boot包含很多@Conditional注解，你可以在自己的代码中通过注解@Configuration类或单独的@Bean方法来重用它们。

36.3.1. Class条件

@ConditionalOnClass和@ConditionalOnMissingClass注解允许根据特定类是否出现来跳过配置。由于注解元数据是使用ASM来解析的，你实际上可以使用value属性来引用真正的类，即使该类可能实际上并没有出现在运行应用的classpath下。如果你倾向于使用一个String值来指定类名，你也可以使用name属性。

36.3.2. Bean条件

@ConditionalOnBean和@ConditionalOnMissingBean注解允许根据特定beans是否出现来跳过配置。你可以使用value属性来指定beans（by type），也可以使用name来指定beans（by name）。search属性允许你限制搜索beans时需要考虑的ApplicationContext的层次。

注：当@Configuration类被解析时@Conditional注解会被处理。Auto-configure @Configuration总是最后被解析（在所有用户定义beans后面），然而，如果你将那些注解用到常规的@Configuration类，需要注意不能引用那些还没有创建好的bean定义。

36.3.3. Property条件

@ConditionalOnProperty注解允许根据一个Spring Environment属性来决定是否包含配置。可以使用prefix和name属性指定要检查的配置属性。默认情况下，任何存在的只要不是false的属性都会匹配。你也可以使用havingValue和matchIfMissing属性创建更高级的检测。

36.3.4. Resource条件

@ConditionalOnResource注解允许只有在特定资源出现时配置才会被包含。资源可以使用常见的Spring约定命名，例如file:/home/user/test.dat。

36.3.5. Web Application条件

@ConditionalOnWebApplication和@ConditionalOnNotWebApplication注解允许根据应用是否为一个'web应用'来决定是否包含配置。一个web应用是任何使用Spring WebApplicationContext，定义一个session作用域或有一个StandardServletEnvironment的应用。

36.3.6. SpEL表达式条件

@ConditionalOnExpression注解允许根据SpEL表达式结果来跳过配置。

37. WebSockets

Spring Boot为内嵌的Tomcat(8和7)，Jetty 9和Undertow提供WebSockets自动配置。如果你正在将一个war包部署到一个单独的容器，Spring Boot会假设该容器会对它的WebSocket支持相关的配置负责。

Spring框架提供丰富的WebSocket支持，通过spring-boot-starter-websocket模块可以轻易获取到。

38. 接下来阅读什么

Spring Boot执行器：Production-ready特性

Spring Boot包含很多其他的特性，它们可以帮你监控和管理发布到生产环境的应用。你可以选择使用HTTP端点，JMX或远程shell（SSH或Telnet）来管理和监控应用。审计（Auditing），健康（health）和数据采集（metrics gathering）会自动应用到你的应用。

39. 开启production-ready特性

spring-boot-actuator模块提供了Spring Boot所有的production-ready特性。启用该特性的最简单方式就是添加对spring-boot-starter-actuator ‘Starter POM’的依赖。

执行器（**Actuator**）的定义：执行器是一个制造业术语，指的是用于移动或控制东西的一个机械装置。一个很小的改变就能让执行器产生大量的运动。

基于Maven的项目想要添加执行器只需添加下面的'starter'依赖：

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

对于Gradle，使用下面的声明：

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

40. 端点

执行器端点允许你监控应用及与应用进行交互。Spring Boot包含很多内置的端点，你也可以添加自己的。例如，health端点提供了应用的基本健康信息。

端点暴露的方式取决于你采用的技术类型。大部分应用选择HTTP监控，端点的ID映射到一个URL。例如，默认情况下，health端点将被映射到/health。

下面的端点都是可用的：

ID	描述	敏感（**Sensitive**）
autoconfig	显示一个auto-configuration的报告，该报告展示所有auto-configuration候选者及它们被应用或未被应用的原因	true
beans	显示一个应用中所有Spring Beans的完整列表	true
configprops	显示一个所有@ConfigurationProperties的整理列表	true
dump	执行一个线程转储	true
env	暴露来自Spring　ConfigurableEnvironment的属性	true
health	展示应用的健康信息（当使用一个未认证连接访问时显示一个简单的'status'，使用认证连接访问则显示全部信息详情）	false
info	显示任意的应用信息	false
metrics	展示当前应用的'指标'信息	true
mappings	显示一个所有@RequestMapping路径的整理列表	true
shutdown	允许应用以优雅的方式关闭（默认情况下不启用）	true
trace	显示trace信息（默认为最新的一些HTTP请求）	true

注：根据一个端点暴露的方式，sensitive参数可能会被用做一个安全提示。例如，在使用HTTP访问sensitive端点时需要提供用户名/密码（如果没有启用web安全，可能会简化为禁止访问该端点）。

40.1. 自定义端点

使用Spring属性可以自定义端点。你可以设置端点是否开启（enabled），是否敏感（sensitive），甚至它的id。例如，下面的application.properties改变了敏感性和beans端点的id，也启用了shutdown。

```
endpoints.beans.id=springbeans
endpoints.beans.sensitive=false
endpoints.shutdown.enabled=true
```

注：前缀 `endpoints + . + name` 被用来唯一的标识被配置的端点。

默认情况下，除了shutdown外的所有端点都是启用的。如果希望指定选择端点的启用，你可以使用endpoints.enabled属性。例如，下面的配置禁用了除info外的所有端点：

```
endpoints.enabled=false
endpoints.info.enabled=true
```

40.2. 健康信息

健康信息可以用来检查应用的运行状态。它经常被监控软件用来提醒人们生产系统是否停止。health端点暴露的默认信息取决于端点是如何被访问的。对于一个非安全，未认证的连接只返回一个简单的'status'信息。对于一个安全或认证过的连接其他详细信息也会展示（具体参考Section 41.6, “HTTP Health endpoint access restrictions”）。

健康信息是从你的ApplicationContext中定义的所有HealthIndicator beans收集过来的。Spring Boot包含很多auto-configured的HealthIndicators，你也可以写自己的。

40.3. 安全与HealthIndicators

HealthIndicators返回的信息常常性质上有点敏感。例如，你可能不想将数据库服务器的详情发布到外面。因此，在使用一个未认证的HTTP连接时，默认只会暴露健康状态（health status）。如果想将所有的健康信息暴露出去，你可以把endpoints.health.sensitive设置为false。

为防止'拒绝服务'攻击，Health响应会被缓存。你可以使用 `endpoints.health.time-to-live` 属性改变默认的缓存时间（1000毫秒）。

40.3.1. 自动配置的HealthIndicators

下面的HealthIndicators会被Spring Boot自动配置（在合适的时候）：

名称	描述
DiskSpaceHealthIndicator	低磁盘空间检测
DataSourceHealthIndicator	检查是否能从DataSource获取连接
MongoHealthIndicator	检查一个Mongo数据库是否可用（up）
RabbitHealthIndicator	检查一个Rabbit服务器是否可用（up）
RedisHealthIndicator	检查一个Redis服务器是否可用（up）
SolrHealthIndicator	检查一个Solr服务器是否可用（up）

40.3.2. 编写自定义HealthIndicators

想提供自定义健康信息，你可以注册实现了HealthIndicator接口的Spring beans。你需要提供一个health()方法的实现，并返回一个Health响应。Health响应需要包含一个status和可选的用于展示的详情。

```
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealth implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

}
```

除了Spring Boot预定义的Status类型，Health也可以返回一个代表新的系统状态的自定义Status。在这种情况下，需要提供一个HealthAggregator接口的自定义实现，或使用management.health.status.order属性配置默认的实现。

例如，假设一个新的，代码为FATAL的Status被用于你的一个HealthIndicator实现中。为了配置严重程度，你需要将下面的配置添加到application属性文件中：

```
management.health.status.order: DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

如果使用HTTP访问health端点，你可能想要注册自定义的status，并使用HealthMvcEndpoint进行映射。例如，你可以将FATAL映射为HttpStatus.SERVICE_UNAVAILABLE。

40.4. 自定义应用info信息

通过设置Spring属性info.*，你可以定义info端点暴露的数据。所有在info关键字下的Environment属性都将被自动暴露。例如，你可以将下面的配置添加到application.properties：

```
info.app.name=MyService
info.app.description=My awesome service
info.app.version=1.0.0
```

40.4.1. 在构建时期自动扩展info属性

你可以使用已经存在的构建配置自动扩展info属性，而不是对在项目构建配置中存在的属性进行硬编码。这在Maven和Gradle都是可能的。

使用**Maven**自动扩展属性

对于Maven项目，你可以使用资源过滤来自动扩展info属性。如果使用spring-boot-starter-parent，你可以通过 `@..@` 占位符引用Maven的'project properties'。

```
project.artifactId=myproject
project.name=Demo
project.version=X.X.X.X
project.description=Demo project for info endpoint
info.build.artifact=@project.artifactId@
info.build.name=@project.name@
info.build.description=@project.description@
info.build.version=@project.version@
```

注：在上面的示例中，我们使用project.*来设置一些值以防止由于某些原因Maven的资源过滤没有开启。Maven目标 `spring-boot:run` 直接将 `src/main/resources` 添加到classpath下（出于热加载的目的）。这就绕过了资源过滤和自动扩展属性的特性。你可以使用 `exec:java` 替换该目标或自定义插件的配置，具体参考plugin usage page。

如果你不使用starter parent，在你的pom.xml你需要添加（处于元素内）：

```
<resources>
    <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
    </resource>
</resources>
```

和（处于内）：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <delimiters>
            <delimiter>@</delimiter>
        </delimiters>
    </configuration>
</plugin>
```

使用**Gradle**自动扩展属性

通过配置Java插件的processResources任务，你也可以自动使用来自Gradle项目的属性扩展info属性。

```
processResources {
    expand(project.properties)
}
```

然后你可以通过占位符引用Gradle项目的属性：

```
info.build.name=${name}
info.build.description=${description}
info.build.version=${version}
```

40.4.2. Git提交信息

info端点的另一个有用特性是，当项目构建完成后，它可以发布关于你的git源码仓库状态的信息。如果在你的jar中包含一个git.properties文件，git.branch和git.commit属性将被加载。

对于Maven用户，`spring-boot-starter-parent` POM包含一个能够产生git.properties文件的预配置插件。只需要简单的将下面的声明添加到你的POM中：

```
<build>
    <plugins>
        <plugin>
            <groupId>pl.project13.maven</groupId>
            <artifactId>git-commit-id-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

对于Gradle用户可以使用一个相似的插件gradle-git，尽管为了产生属性文件可能需要稍微多点工作。

41. 基于HTTP的监控和管理

如果你正在开发一个Spring MVC应用，Spring Boot执行器自动将所有启用的端点通过HTTP暴露出去。默认约定使用端点的id作为URL路径，例如，health暴露为/health。

41.1. 保护敏感端点

如果你的项目中添加的有Spring Security，所有通过HTTP暴露的敏感端点都会受到保护。默认情况下会使用基本认证（basic authentication，用户名为user，密码为应用启动时在控制台打印的密码）。

你可以使用Spring属性改变用户名，密码和访问端点需要的安全角色。例如，你可能会在application.properties中添加下列配置：

```
security.user.name=admin
security.user.password=secret
management.security.role=SUPERUSER
```

注：如果你不使用Spring Security，那你的HTTP端点就被公开暴露，你应该慎重考虑启用哪些端点。具体参考Section 40.1, “Customizing endpoints”。

41.2. 自定义管理服务器的上下文路径

有时候将所有的管理端口划分到一个路径下是有用的。例如，你的应用可能已经将 `/info` 作为他用。你可以用 `management.contextPath` 属性为管理端口设置一个前缀：

```
management.context-path=/manage
```

上面的application.properties示例将把端口从 `/{id}` 改为 `/manage/{id}` （比如，/manage/info）。

41.3. 自定义管理服务器的端口

对于基于云的部署，使用默认的HTTP端口暴露管理端点（endpoints）是明智的选择。然而，如果你的应用是在自己的数据中心运行，那你可能倾向于使用一个不同的HTTP端口来暴露端点。

`management.port` 属性可以用来改变HTTP端口：

```
management.port=8081
```

由于你的管理端口经常被防火墙保护，不对外暴露也就不需要保护管理端点，即使你的主要应用是安全的。在这种情况下，classpath下会存在Spring Security库，你可以设置下面的属性来禁用安全管理策略（management security）：

```
management.security.enabled=false
```

（如果classpath下不存在Spring Security，那也就不需要显示的以这种方式来禁用安全管理策略，它甚至可能会破坏应用程序。）

41.4. 自定义管理服务器的地址

你可以通过设置`management.address`属性来定义管理端点可以使用的地址。这在你只想监听内部或面向生产环境的网络，或只监听来自localhost的连接时非常有用。

下面的application.properties示例不允许远程管理连接：

```
management.port=8081
management.address=127.0.0.1
```

41.5. 禁用HTTP端点

如果不想通过HTTP暴露端点，你可以将管理端口设置为-1： `management.port=-1`

41.6. HTTP Health端点访问限制

通过health端点暴露的信息根据是否为匿名访问而不同。默认情况下，当匿名访问时，任何有关服务器的健康详情都被隐藏了，该端点只简单的指示服务器是运行（up）还是停止（down）。此外，当匿名访问时，响应会被缓存一个可配置的时间段以防止端点被用于'拒绝服务'攻击。`endpoints.health.time-to-live`属性被用来配置缓存时间（单位为毫秒），默认为1000毫秒，也就是1秒。

上述的限制可以被禁止，从而允许匿名用户完全访问health端点。想达到这个效果，可以将`endpoints.health.sensitive`设为`false`。

42. 基于JMX的监控和管理

Java管理扩展（JMX）提供了一种标准的监控和管理应用的机制。默认情况下，Spring Boot在 `org.springframework.boot` 域下将管理端点暴露为JMX MBeans。

42.1. 自定义MBean名称

MBean的名称通常产生于端点的id。例如，health端点被暴露为 `org.springframework.boot/Endpoint/HealthEndpoint` 。

如果你的应用包含多个Spring ApplicationContext，你会发现存在名称冲突。为了解决这个问题，你可以将 `endpoints.jmx.uniqueNames` 设置为true，这样MBean的名称总是唯一的。

你也可以自定义JMX域，所有的端点都在该域下暴露。这里有个application.properties示例：
```java endpoints.jmx.domain=myapp endpoints.jmx.uniqueNames=true
```

42.2. 禁用JMX端点

如果不想通过JMX暴露端点，你可以将 `spring.jmx.enabled` 属性设置为false：

```
spring.jmx.enabled=false
```

42.3. 使用Jolokia通过HTTP实现JMX远程管理

Jolokia是一个JMX-HTTP桥，它提供了一种访问JMX beans的替代方法。想要使用Jolokia，只需添加 `org.jolokia:jolokia-core` 的依赖。例如，使用Maven需要添加下面的配置：

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
 </dependency>
```

在你的管理HTTP服务器上可以通过 `/jolokia` 访问Jolokia。

42.3.1. 自定义Jolokia

Jolokia有很多配置，传统上一般使用servlet参数进行设置。使用Spring Boot，你可以在application.properties中通过把参数加上 `jolokia.config.` 前缀来设置：

```
jolokia.config.debug=true
```

42.3.2. 禁用Jolokia

如果你正在使用Jolokia，但不想让Spring Boot配置它，只需要简单的将 `endpoints.jolokia.enabled` 属性设置为false：

```
endpoints.jolokia.enabled=false
```

43. 使用远程shell来进行监控和管理

Spring Boot支持集成一个称为'CRaSH'的Java shell。你可以在CRaSH中使用ssh或telnet命令连接到运行的应用。为了启用远程shell支持，你只需添加 `spring-boot-starter-remote-shell` 的依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-remote-shell</artifactId>
 </dependency>
```

注：如果想使用telnet访问，你还需添加对 `org.crsh:crsh.shell.telnet` 的依赖。

43.1. 连接远程shell

默认情况下，远程shell监听端口2000以等待连接。默认用户名为 `user` ，密码为随机生成的，并且在输出日志中会显示。如果应用使用Spring Security，该shell默认使用相同的配置。如果不是，将使用一个简单的认证策略，你可能会看到类似这样的信息：

```
Using default password for shell access: ec03e16c-4cf4-49ee-b745-7c8255c1dd7e
```

Linux和OSX用户可以使用 `ssh` 连接远程shell，Windows用户可以下载并安装PuTTY。

```
$ ssh -p 2000 user@localhost

user@localhost's password:
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.3.0.BUILD-SNAPSHOT) on myhost
```

输入help可以获取一系列命令的帮助。Spring boot提供 `metrics` ， `beans` ， `autoconfig` 和 `endpoint` 命令。

43.1.1. 远程shell证书

你可以使用`shell.auth.simple.user.name`和`shell.auth.simple.user.password`属性配置自定义的连接证书。也可以使用Spring Security的AuthenticationManager处理登录职责。具体参考JavadocCrshAutoConfiguration和ShellProperties。

43.2. 扩展远程shell

有很多有趣的方式可以用来扩展远程shell。

43.2.1. 远程shell命令

你可以使用Groovy或Java编写其他的shell命令（具体参考CRaSH文档）。默认情况下，Spring Boot会搜索以下路径的命令：

- `classpath*:/commands/**`
- `classpath*:/crash/commands/**`

注：可以通过`shell.commandPathPatterns`属性改变搜索路径。

下面是一个从`src/main/resources/commands/hello.groovy`加载的'hello world'命令：

```
package commands

import org.crsh.cli.Usage
import org.crsh.cli.Command

class hello {

    @Usage("Say Hello")
    @Command
    def main(InvocationContext context) {
        return "Hello"
    }

}
```

Spring Boot将一些额外属性添加到了InvocationContext，你可以在命令中访问它们：

属性名称	描述
spring.boot.version	Spring Boot的版本
spring.version	Spring框架的核心版本
spring.beanfactory	获取Spring的BeanFactory
spring.environment	获取Spring的Environment

43.2.2. 远程shell插件

除了创建新命令，也可以扩展CRaSH shell的其他特性。所有继承 `org.crsh.plugin.CRaSHPlugin` 的Spring Beans将自动注册到shell。

具体查看CRaSH参考文档。

44. 度量指标（Metrics）

Spring Boot执行器包括一个支持'gauge'和'counter'级别的度量指标服务。'gauge'记录一个单一值；'counter'记录一个增量（增加或减少）。同时，Spring Boot提供一个PublicMetrics接口，你可以实现它，从而暴露以上两种机制不能记录的指标。具体参考SystemPublicMetrics。

所有HTTP请求的指标都被自动记录，所以如果点击`metrics`端点，你可能会看到类似以下的响应：

```
{
    "counter.status.200.root": 20,
    "counter.status.200.metrics": 3,
    "counter.status.200.star-star": 5,
    "counter.status.401.root": 4,
    "gauge.response.star-star": 6,
    "gauge.response.root": 2,
    "gauge.response.metrics": 3,
    "classes": 5808,
    "classes.loaded": 5808,
    "classes.unloaded": 0,
    "heap": 3728384,
    "heap.committed": 986624,
    "heap.init": 262144,
    "heap.used": 52765,
    "mem": 986624,
    "mem.free": 933858,
    "processors": 8,
    "threads": 15,
    "threads.daemon": 11,
    "threads.peak": 15,
    "uptime": 494836,
    "instance.uptime": 489782,
    "datasource.primary.active": 5,
    "datasource.primary.usage": 0.25
}
```

此处我们可以看到基本的`memory`，`heap`，`class loading`，`processor`和`thread pool`信息，连同一些HTTP指标。在该实例中，`root`('/')，`/metrics` URLs分别返回20次，3次`HTTP 200`响应。同时可以看到`root` URL返回了4次`HTTP 401`（unauthorized）响应。双asterix（star-star）来自于被Spring MVC `/**`匹配到的一个请求（通常为一个静态资源）。

`gauge`级别展示了一个请求的最后响应时间。所以，`root`的最后请求被响应耗时2毫秒，`/metrics`耗时3毫秒。

44.1. 系统指标

Spring Boot暴露以下系统指标：

- 系统内存总量（mem），单位:Kb
- 空闲内存数量（mem.free），单位:Kb
- 处理器数量（processors）
- 系统正常运行时间（uptime），单位:毫秒
- 应用上下文（就是一个应用实例）正常运行时间（instance.uptime），单位:毫秒
- 系统平均负载（systemload.average）
- 堆信息（heap，heap.committed，heap.init，heap.used），单位:Kb
- 线程信息（threads，thread.peak，thead.daemon）
- 类加载信息（classes，classes.loaded，classes.unloaded）
- 垃圾收集信息（gc.xxx.count, gc.xxx.time）

44.2. 数据源指标

Spring Boot会为你应用中定义的支持的DataSource暴露以下指标：

- 最大连接数（datasource.xxx.max）
- 最小连接数（datasource.xxx.min）
- 活动连接数（datasource.xxx.active）
- 连接池的使用情况（datasource.xxx.usage）

所有的数据源指标共用 `datasoure.` 前缀。该前缀对每个数据源都非常合适：

- 如果是主数据源（唯一可用的数据源或存在的数据源中被@Primary标记的）前缀为datasource.primary
- 如果数据源bean名称以dataSource结尾，那前缀就是bean的名称去掉dataSource的部分（例如，batchDataSource的前缀是datasource.batch）
- 其他情况使用bean的名称作为前缀

通过注册一个自定义版本的DataSourcePublicMetrics bean，你可以覆盖部分或全部的默认行为。默认情况下，Spring Boot提供支持所有数据源的元数据；如果你喜欢的数据源恰好不被支持，你可以添加另外的DataSourcePoolMetadataProvider beans。具体参考DataSourcePoolMetadataProvidersConfiguration。

44.3. Tomcat session指标

如果你使用Tomcat作为内嵌的servlet容器，session指标将被自动暴露出去。

`httpsessions.active` 和 `httpsessions.max` 提供了活动的和最大的session数量。

44.4. 记录自己的指标

想要记录你自己的指标，只需将CounterService或GaugeService注入到你的bean中。CounterService暴露increment，decrement和reset方法；GaugeService提供一个submit方法。

下面是一个简单的示例，它记录了方法调用的次数：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final CounterService counterService;

    @Autowired
    public MyService(CounterService counterService) {
        this.counterService = counterService;
    }

    public void exampleMethod() {
        this.counterService.increment("services.system.myservice.invoked");
    }

}
```

注：你可以将任何的字符串用作指标的名称，但最好遵循所选存储或图技术的指南。Matt Aimonetti's Blog中有一些好的关于图（Graphite）的指南。

44.5. 添加你自己的公共指标

想要添加额外的，每次指标端点被调用时都会重新计算的度量指标，只需简单的注册其他的PublicMetrics实现bean(s)。默认情况下，端点会聚合所有这样的beans，通过定义自己的MetricsEndpoint可以轻易改变这种情况。

44.6. 指标仓库

通过绑定一个MetricRepository来实现指标服务。`MetricRepository`负责存储和追溯指标信息。Spring Boot提供一个`InMemoryMetricRepository`和一个`RedisMetricRepository`（默认使用in-memory仓库），不过你可以编写自己的`MetricRepository`。`MetricRepository`接口实际是`MetricReader`接口和`MetricWriter`接口的上层组合。具体参考Javadoc

没有什么能阻止你直接将`MetricRepository`的数据导入应用中的后端存储，但我们建议你使用默认的`InMemoryMetricRepository`（如果担心堆使用情况，你可以使用自定义的Map实例），然后通过一个scheduled export job填充后端仓库（意思是先将数据保存到内存中，然后通过异步job将数据持久化到数据库，可以提高系统性能）。通过这种方式，你可以将指标数据缓存到内存中，然后通过低频率或批量导出来减少网络拥堵。Spring Boot提供一个`Exporter`接口及一些帮你开始的基本实现。

44.7. Dropwizard指标

Dropwizard ‘Metrics’库的用户会发现Spring Boot指标被发布到了 `com.codahale.metrics.MetricRegistry` 。当你声明对 `io.dropwizard.metrics:metrics-core` 库的依赖时会创建一个默认的 `com.codahale.metrics.MetricRegistry` Spring bean；如果需要自定义，你可以注册自己的@Bean实例。来自于 `MetricRegistry` 的指标也是自动通过 `/metrics` 端点暴露的。

用户可以通过使用合适类型的指标名称作为前缀来创建Dropwizard指标（比如， `histogram.*` , `meter.*` ）。

44.8. 消息渠道集成

如果你的classpath下存在'Spring Messaging' jar，一个名为 `metricsChannel` 的 `MessageChannel` 将被自动创建（除非已经存在一个）。此外，所有的指标更新事件作为'messages'发布到该渠道上。订阅该渠道的客户端可以进行额外的分析或行动。

45. 审计

Spring Boot执行器具有一个灵活的审计框架，一旦Spring Security处于活动状态（默认抛出'authentication success'，'failure'和'access denied'异常），它就会发布事件。这对于报告非常有用，同时可以基于认证失败实现一个锁定策略。

你也可以使用审计服务处理自己的业务事件。为此，你可以将存在的 `AuditEventRepository` 注入到自己的组件，并直接使用它，或者只是简单地通过Spring `ApplicationEventPublisher` 发布 `AuditApplicationEvent` （使用 `ApplicationEventPublisherAware` ）。

46. 追踪（Tracing）

对于所有的HTTP请求Spring Boot自动启用追踪。你可以查看`trace`端点，并获取最近一些请求的基本信息：

```
[{
    "timestamp": 1394343677415,
    "info": {
        "method": "GET",
        "path": "/trace",
        "headers": {
            "request": {
                "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
=0.8",
                "Connection": "keep-alive",
                "Accept-Encoding": "gzip, deflate",
                "User-Agent": "Mozilla/5.0 Gecko/Firefox",
                "Accept-Language": "en-US,en;q=0.5",
                "Cookie": "_ga=GA1.1.827067509.1390890128; ..."
                "Authorization": "Basic ...",
                "Host": "localhost:8080"
            },
            "response": {
                "Strict-Transport-Security": "max-age=31536000 ; includeSubDomains",
                "X-Application-Context": "application:8080",
                "Content-Type": "application/json;charset=UTF-8",
                "status": "200"
            }
        }
    }
},{
    "timestamp": 1394343684465,
    ...
}]
```

46.1. 自定义追踪

如果需要追踪其他的事件，你可以将一个TraceRepository注入到你的Spring Beans中。`add`方法接收一个将被转化为JSON的`Map`结构，该数据将被记录下来。

默认情况下，使用的`InMemoryTraceRepository`将存储最新的100个事件。如果需要扩展该容量，你可以定义自己的`InMemoryTraceRepository`实例。如果需要，你可以创建自己的替代`TraceRepository`实现。

47. 进程监控

在Spring Boot执行器中，你可以找到几个创建有利于进程监控的文件的类：

- `ApplicationPidFileWriter` 创建一个包含应用PID的文件（默认位于应用目录，文件名为application.pid）
- `EmbeddedServerPortFileWriter` 创建一个或多个包含内嵌服务器端口的文件（默认位于应用目录，文件名为application.port）

默认情况下，这些writers没有被激活，但你可以使用下面描述的任何方式来启用它们。

47.1. 扩展属性

你需要激活 `META-INF/spring.factories` 文件里的listener(s)：

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.actuate.system.ApplicationPidFileWriter,
org.springframework.boot.actuate.system.EmbeddedServerPortFileWriter
```

47.2. 以编程方式

你也可以通过调用 `SpringApplication.addListeners(…)` 方法来激活一个监听器，并传递相应的 `Writer` 对象。该方法允许你通过 `Writer` 构造器自定义文件名和路径。

部署到云端

对于大多数流行云PaaS（平台即服务）提供商，Spring Boot的可执行jars就是为它们准备的。这些提供商往往要求你带上自己的容器；它们管理应用的进程（不特别针对Java应用程序），所以它们需要一些中间层来将你的应用适配到云概念中的一个运行进程。

两个流行的云提供商，Heroku和Cloud Foundry，采取一个打包（'buildpack'）方法。为了启动你的应用程序，不管需要什么，buildpack都会将它们打包到你的部署代码：它可能是一个JDK和一个java调用，也可能是一个内嵌的webserver，或者是一个成熟的应用服务器。buildpack是可插拔的，但你最好尽可能少的对它进行自定义设置。这可以减少不受你控制的功能范围，最小化部署和生产环境的发散。

理想情况下，你的应用就像一个Spring Boot可执行jar，所有运行需要的东西都打包到它内部。

49. Cloud Foundry

如果不指定其他打包方式，Cloud Foundry会启用它提供的默认打包方式。Cloud Foundry的Java buildpack对Spring应用有出色的支持，包括Spring Boot。你可以部署独立的可执行jar应用，也可以部署传统的.war形式的应用。

一旦你构建了应用（比如，使用`mvn clean package`）并安装了cf命令行工具，你可以使用下面的`cf push`命令（将路径指向你编译后的.jar）来部署应用。在发布一个应用前，确保你已登陆cf命令行客户端。

```
$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

查看`cf push`文档获取更多可选项。如果相同目录下存在manifest.yml，Cloud Foundry会使用它。

就此，cf开始上传你的应用：

```
Uploading acloudyspringtime... OK
Preparing to start acloudyspringtime... OK
-----> Downloaded app package (8.9M)
-----> Java Buildpack source: system
-----> Downloading Open JDK 1.7.0_51 from .../x86_64/openjdk-1.7.0_51.tar.gz (1.8s)
       Expanding Open JDK to .java-buildpack/open_jdk (1.2s)
-----> Downloading Spring Auto Reconfiguration from  0.8.7 .../auto-reconfiguration-0.
8.7.jar (0.1s)
-----> Uploading droplet (44M)
Checking status of app 'acloudyspringtime'...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 down)
  ...
  0 of 1 instances running (1 starting)
  ...
  1 of 1 instances running (1 running)

App started
```

恭喜！应用现在处于运行状态！

检验部署应用的状态是很简单的：

```
$ cf apps
Getting applications in ...
OK

name                requested state   instances   memory   disk   urls
...
acloudyspringtime   started           1/1         512M     1G     acloudyspringtime.c
fapps.io
...
```

一旦Cloud Foundry意识到你的应用已经部署，你就可以点击给定的应用URI，此处是acloudyspringtime.cfapps.io/。

49.1. 绑定服务

默认情况下，运行应用的元数据和服务连接信息被暴露为应用的环境变量（比如，$VCAP_SERVICES）。采用这种架构的原因是因为Cloud Foundry多语言特性（任何语言和平台都支持作为buildpack）。进程级别的环境变量是语言无关（language agnostic）的。

环境变量并不总是有利于设计最简单的API，所以Spring Boot自动提取它们，然后将这些数据导入能够通过Spring `Environment` 抽象访问的属性里：

```
@Component
class MyBean implements EnvironmentAware {

    private String instanceId;

    @Override
    public void setEnvironment(Environment environment) {
        this.instanceId = environment.getProperty("vcap.application.instance_id");
    }

    // ...

}
```

所有的Cloud Foundry属性都以vcap作为前缀。你可以使用vcap属性获取应用信息（比如应用的公共URL）和服务信息（比如数据库证书）。具体参考VcapApplicationListener Javadoc。

注：Spring Cloud Connectors项目很适合比如配置数据源的任务。Spring Boot提供自动配置支持和一个 `spring-boot-starter-cloud-connectors` starter POM。

50. Heroku

Heroku是另外一个流行的Paas平台。想要自定义Heroku的构建过程，你可以提供一个 `Procfile` ，它提供部署一个应用所需的指令。Heroku为Java应用分配一个端口，确保能够路由到外部URI。

你必须配置你的应用监听正确的端口。下面是用于我们的starter REST应用的Procfile：

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

Spring Boot将 `-D` 参数作为属性，通过一个Spring的Environment实例访问。 `server.port` 配置属性适合于内嵌的Tomcat，Jetty或Undertow实例启用时使用。 `$PORT` 环境变量被分配给Heroku Paas使用。

Heroku默认使用Java 1.6。只要你的Maven或Gradle构建时使用相同的版本就没问题（Maven用户可以设置 `java.version` 属性）。如果你想使用JDK 1.7，在你的pom.xml和Procfile临近处创建一个system.properties文件。在该文件中添加以下设置：

```
java.runtime.version=1.7
```

这就是你需要做的一切。对于Heroku部署来说，经常做的工作就是使用 `git push` 将代码推送到生产环境。

```
$ git push heroku master

Initializing repository, done.
Counting objects: 95, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (78/78), done.
Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, done.
Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected
-----> Installing OpenJDK 1.7... done
-----> Installing Maven 3.2.3... done
-----> Installing settings.xml... done
-----> executing /app/tmp/cache/.maven/bin/mvn -B
       -Duser.home=/tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229
       -Dmaven.repo.local=/app/tmp/cache/.m2/repository
       -s /app/tmp/cache/.m2/settings.xml -DskipTests=true clean install

       [INFO] Scanning for projects...
       Downloading: http://repo.spring.io/...
       Downloaded: http://repo.spring.io/... (818 B at 1.8 KB/sec)
        ....
       Downloaded: http://s3pository.heroku.com/jvm/... (152 KB at 595.3 KB/sec)
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/target/...
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ...
       [INFO] ------------------------------------------------------------------------
       [INFO] BUILD SUCCESS
       [INFO] ------------------------------------------------------------------------
       [INFO] Total time: 59.358s
       [INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014
       [INFO] Final Memory: 20M/493M
       [INFO] ------------------------------------------------------------------------

-----> Discovering process types
       Procfile declares types -> web

-----> Compressing... done, 70.4MB
-----> Launching... done, v6
       http://agile-sierra-1405.herokuapp.com/ deployed to Heroku

To git@heroku.com:agile-sierra-1405.git
 * [new branch]      master -> master
```

现在你的应用已经启动并运行在Heroku。

51. Openshift

Openshift是RedHat公共（和企业）PaaS解决方案。和Heroku相似，它也是通过运行被git提交触发的脚本来工作的，所以你可以使用任何你喜欢的方式编写Spring Boot应用启动脚本，只要Java运行时环境可用（这是在Openshift上可以要求的一个标准特性）。为了实现这样的效果，你可以使用DIY Cartridge，并在`.openshift/action_scripts`下hooks你的仓库：

基本模式如下：

1.确保Java和构建工具已被远程安装，比如使用一个`pre_build` hook（默认会安装Java和Maven，不会安装Gradle）。

2.使用一个`build` hook去构建你的jar（使用Maven或Gradle），比如

```
#!/bin/bash
cd $OPENSHIFT_REPO_DIR
mvn package -s .openshift/settings.xml -DskipTests=true
```

3.添加一个调用`java -jar …`的`start` hook

```
#!/bin/bash
cd $OPENSHIFT_REPO_DIR
nohup java -jar target/*.jar --server.port=${OPENSHIFT_DIY_PORT} --server.address=${OP
ENSHIFT_DIY_IP} &
```

4.使用一个`stop` hook

```
#!/bin/bash
source $OPENSHIFT_CARTRIDGE_SDK_BASH
PID=$(ps -ef | grep java.*\.jar | grep -v grep | awk '{ print $2 }')
if [ -z "$PID" ]
then
    client_result "Application is already stopped"
else
    kill $PID
fi
```

5.将内嵌的服务绑定到平台提供的在application.properties定义的环境变量，比如

```
spring.datasource.url: jdbc:mysql://${OPENSHIFT_MYSQL_DB_HOST}:${OPENSHIFT_MYSQL_DB_PO
RT}/${OPENSHIFT_APP_NAME}
spring.datasource.username: ${OPENSHIFT_MYSQL_DB_USERNAME}
spring.datasource.password: ${OPENSHIFT_MYSQL_DB_PASSWORD}
```

在Openshift的网站上有一篇running Gradle in Openshift博客，如果想使用gradle构建运行的应用可以参考它。由于一个Gradle bug，你不能使用高于1.6版本的Gradle。

52. Google App Engine

Google App Engine跟Servlet 2.5 API是有联系的，所以在不修改的情况系你是不能部署一个Spring应用的。具体查看本指南的Servlet 2.5章节 Container.md)。

53. 接下来阅读什么

Spring Boot CLI

Spring Boot CLI是一个命令行工具，如果想使用Spring进行快速开发可以使用它。它允许你运行Groovy脚本，这意味着你可以使用熟悉的类Java语法，并且没有那么多的模板代码。你也可以启动一个新的项目或为Spring Boot CLI编写自己的命令。

54. 安装CLI

你可以手动安装Spring Boot CLI，也可以使用GVM（Groovy环境管理工具）或Homebrew，MacPorts（如果你是一个OSX用户）。参考"Getting started"的Section 10.2, “Installing the Spring Boot CLI” 可以看到全面的安装指令。

55. 使用CLI

一旦安装好CLI，你可以输入 `spring` 来运行它。如果你不使用任何参数运行 `spring` ，将会展现一个简单的帮助界面：

```
$ spring
usage: spring [--help] [--version]
       <command> [<args>]

Available commands are:

  run [options] <files> [--] [args]
    Run a spring groovy script

  ... more command help is shown here
```

你可以使用 `help` 获取任何支持命令的详细信息。例如：

```
$ spring help run
spring run - Run a spring groovy script

usage: spring run [options] <files> [--] [args]

Option                     Description
------                     -----------
--autoconfigure [Boolean]  Add autoconfigure compiler
                             transformations (default: true)
--classpath, -cp           Additional classpath entries
-e, --edit                 Open the file with the default system
                             editor
--no-guess-dependencies    Do not attempt to guess dependencies
--no-guess-imports         Do not attempt to guess imports
-q, --quiet                Quiet logging
-v, --verbose              Verbose logging of dependency
                             resolution
--watch                    Watch the specified file for changes
```

`version` 命令提供一个检查你正在使用的Spring Boot版本的快速方式：

```
$ spring version
Spring CLI v1.3.0.BUILD-SNAPSHOT
```

55.1. 使用CLI运行应用

你可以使用 `run` 命令编译和运行Groovy源代码。Spring Boot CLI完全自包含，以致于你不需要安装任何外部的Groovy。

下面是一个使用Groovy编写的"hello world" web应用： hello.grooy

```
@RestController
class WebApplication {

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

}
```

想要编译和运行应用，输入：

```
$ spring run hello.groovy
```

想要给应用传递命令行参数，你需要使用一个 `--` 来将它们和"spring"命令参数区分开来。例如：

```
$ spring run hello.groovy -- --server.port=9000
```

想要设置JVM命令行参数，你可以使用 `JAVA_OPTS` 环境变量，例如：

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

55.1.1. 推断"grab"依赖

标准的Groovy包含一个 `@Grab` 注解，它允许你声明对第三方库的依赖。这项有用的技术允许Groovy以和Maven或Gradle相同的方式下载jars，但不需要使用构建工具。

Spring Boot进一步延伸了该技术，它会基于你的代码尝试推导你"grab"哪个库。例如，由于WebApplication代码上使用了 `@RestController` 注解，"Tomcat"和"Spring MVC"将被获取（grabbed）。

下面items被用作"grab hints"：

items	Grabs
JdbcTemplate,NamedParameterJdbcTemplate,DataSource	JDBC应用
@EnableJms	JMS应用
@EnableCaching	Caching abstraction
@Test	JUnit
@EnableRabbit	RabbitMQ
@EnableReactor	Project Reactor
继承Specification	Spock test
@EnableBatchProcessing	Spring Batch
@MessageEndpoint,@EnableIntegrationPatterns	Spring Integration
@EnableDeviceResolver	Spring Mobile
@Controller,@RestController,@EnableWebMvc	Spring MVC + Embedded Tomcat
@EnableWebSecurity	Spring Security
@EnableTransactionManagement	Spring Transaction Management

注：想要理解自定义是如何生效，可以查看Spring Boot CLI源码中的CompilerAutoConfiguration子类。

55.1.2. 推断"grab"坐标

Spring Boot扩展Groovy标准"@Grab"注解使其能够允许你指定一个没有group或version的依赖，例如`@Grab('freemarker')`。 artifact's的组和版本是通过查看Spring Boot的依赖元数据推断出来的。注意默认的元数据是和你使用的CLI版本绑定的－只有在你迁移到一个CLI新版本时它才会改变，这样当你的依赖改变时你就可以控制了。在附录的表格中可以查看默认元数据包含的依赖和它们的版本。

55.1.3. 默认import语句

为了帮助你减少Groovy代码量，一些 `import` 语句被自动包含进来了。注意上面的示例中引用 `@Component` ， `@RestController` 和 `@RequestMapping` 而没有使用全限定名或 `import` 语句。

注：很多Spring注解在不使用 `import` 语句的情况下可以正常工作。尝试运行你的应用，看一下在添加imports之前哪些会失败。

55.1.4. 自动创建main方法

跟等效的Java应用不同，你不需要在Groovy脚本中添加一个 `public static void main(String[] args)` 方法。Spring Boot 会使用你编译后的代码自动创建一个SpringApplication。

55.1.5. 自定义"grab"元数据

Spring Boot提供一个新的 `@GrabMetadata` 注解，你可以使用它提供自定义的依赖元数据，以覆盖Spring Boot的默认配置。该元数据通过使用提供一个或多个配置文件坐标的注解来指定（使用一个属性标识符"type"部署到Maven仓库）.。配置文件中的每个实体必须遵循 `group:module=version` 的格式。

例如，下面的声明：

```
`@GrabMetadata("com.example.custom-versions:1.0.0")`
```

将会加载Maven仓库处于 `com/example/custom-versions/1.0.0/` 下的 `custom-versions-1.0.0.properties` 文件。

可以通过注解指定多个属性文件，它们会以声明的顺序被使用。例如：

```
`@GrabMetadata(["com.example.custom-versions:1.0.0",
       "com.example.more-versions:1.0.0"])`
```

意味着位于 `more-versions` 的属性将覆盖位于 `custom-versions` 的属性。

你可以在任何能够使用 `@Grab` 的地方使用 `@GrabMetadata` ，然而，为了确保元数据的顺序一致，你在应用程序中最多只能使用一次 `@GrabMetadata` 。Spring IO Platform是一个非常有用的依赖元数据源(Spring Boot的超集)，例如：

```
@GrabMetadata('io.spring.platform:platform-versions:1.0.4.RELEASE')
```

55.2. 测试你的代码

`test` 命令允许你编译和运行应用程序的测试用例。常规使用方式如下：

```
$ spring test app.groovy tests.groovy
Total: 1, Success: 1, : Failures: 0
Passed? true
```

在这个示例中，`test.groovy` 包含JUnit `@Test` 方法或Spock `Specification` 类。所有的普通框架注解和静态方法在不使用import导入的情况下，仍旧可以使用。

下面是我们使用的 `test.groovy` 文件（含有一个JUnit测试）：

```
class ApplicationTests {

    @Test
    void homeSaysHello() {
        assertEquals("Hello World!", new WebApplication().home())
    }

}
```

注：如果有多个测试源文件，你可以倾向于使用一个test目录来组织它们。

55.3. 多源文件应用

你可以在所有接收文件输入的命令中使用shell通配符。这允许你轻松处理来自一个目录下的多个文件，例如：

```
$ spring run *.groovy
```

如果你想将'test'或'spec'代码从主应用代码中分离，这项技术就十分有用了：

```
$ spring test app/*.groovy test/*.groovy
```

55.4. 应用打包

你可以使用 `jar` 命令打包应用程序为一个可执行的jar文件。例如：

```
$ spring jar my-app.jar *.groovy
```

最终的jar包括编译应用产生的类和所有依赖，这样你就可以使用 `java -jar` 来执行它了。该jar文件也包括来自应用classpath的实体。你可以使用 `--include` 和 `--exclude` 添加明确的路径（两者都是用逗号分割，同样都接收值为'+'和'-'的前缀，'-'意味着它们将从默认设置中移除）。默认包含（includes）：

```
public/**, resources/**, static/**, templates/**, META-INF/**, *
```

默认排除(excludes)：

```
.*, repository/**, build/**, target/**, **/*.jar, **/*.groovy
```

查看 `spring help jar` 可以获得更多信息。

55.5. 初始化新工程

`init` 命令允许你使用start.spring.io在不离开shell的情况下创建一个新的项目。例如：

```
$ spring init --dependencies=web,data-jpa my-project
Using service at https://start.spring.io
Project extracted to '/Users/developer/example/my-project'
```

这创建了一个 `my-project` 目录，它是一个基本Maven且依赖 `spring-boot-starter-web` 和 `spring-boot-starter-data-jpa` 的项目。你可以使用 `--list` 参数列出该服务的能力。

```
$ spring init --list
=======================================
Capabilities of https://start.spring.io
=======================================

Available dependencies:
-----------------------
actuator - Actuator: Production ready features to help you monitor and manage your app
lication
...
web - Web: Support for full-stack web development, including Tomcat and spring-webmvc
websocket - Websocket: Support for WebSocket development
ws - WS: Support for Spring Web Services

Available project types:
------------------------
gradle-build -  Gradle Config [format:build, build:gradle]
gradle-project -  Gradle Project [format:project, build:gradle]
maven-build -  Maven POM [format:build, build:maven]
maven-project -  Maven Project [format:project, build:maven] (default)

...
```

`init` 命令支持很多选项，查看 `help` 输出可以获得更多详情。例如，下面的命令创建一个使用Java8和war打包的gradle项目：

```
$ spring init --build=gradle --java-version=1.8 --dependencies=websocket --packaging=w
ar sample-app.zip
Using service at https://start.spring.io
Content saved to 'sample-app.zip'
```

55.6. 使用内嵌shell

Spring Boot包括完整的BASH和zsh shells的命令行脚本。如果你不使用它们中的任何一个（可能你是一个Window用户），那你可以使用 `shell` 命令启用一个集成shell。

```
$ spring shell
Spring Boot (v1.3.0.BUILD-SNAPSHOT)
Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.
```

从内嵌shell中可以直接运行其他命令：

```
$ version
Spring CLI v1.3.0.BUILD-SNAPSHOT
```

内嵌shell支持ANSI颜色输出和tab补全。如果需要运行一个原生命令，你可以使用 `$` 前缀。点击ctrl-c将退出内嵌shell。

55.7. 为CLI添加扩展

使用 `install` 命令可以为CLI添加扩展。该命令接收一个或多个格式为 `group:artifact:version` 的artifact坐标集。例如：

```
$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

除了安装你提供坐标的artifacts标识外，所有依赖也会被安装。使用 `uninstall` 可以卸载一个依赖。和 `install` 命令一样，它接收一个或多个格式为 `group:artifact:version` 的artifact坐标集。例如：

```
$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

它会通过你提供的坐标卸载相应的artifacts标识和它们的依赖。

为了卸载所有附加依赖，你可以使用 `--all` 选项。例如：

```
$ spring uninstall --all
```

56. 使用Groovy beans DSL开发应用

Spring框架4.0版本对beans{} DSL（借鉴自Grails）提供原生支持，你可以使用相同的格式在你的Groovy应用程序脚本中嵌入bean定义。有时候这是一个包括外部特性的很好的方式，比如中间件声明。例如：

```
@Configuration
class Application implements CommandLineRunner {

    @Autowired
    SharedService service

    @Override
    void run(String... args) {
        println service.message
    }

}

import my.company.SharedService

beans {
    service(SharedService) {
        message = "Hello World"
    }
}
```

你可以使用beans{}混合位于相同文件的类声明，只要它们都处于顶级，或如果你喜欢的话，可以将beans DSL放到一个单独的文件中。

57. 接下来阅读什么

构建工具插件

Spring Boot为Maven和Gradle提供构建工具插件。该插件提供各种各样的特性，包括打包可执行jars。本节提供关于插件的更多详情及用于扩展一个不支持的构建系统所需的帮助信息。如果你是刚刚开始，那可能需要先阅读Part III, “Using Spring Boot”章节的“Chapter 13, Build systems”。

58. Spring Boot Maven插件

Spring Boot Maven插件为Maven提供Spring Boot支持，它允许你打包可执行jar或war存档，然后就地运行应用。为了使用它，你需要使用Maven 3.2（或更高版本）。

注：参考Spring Boot Maven Plugin Site可以获取全部的插件文档。

58.1. 包含该插件

想要使用Spring Boot Maven插件只需简单地在你的pom.xml的 `plugins` 部分包含相应的XML：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/X
MLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!-- ... -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>1.3.0.BUILD-SNAPSHOT</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

该配置会在Maven生命周期的 `package` 阶段重新打包一个jar或war。下面的示例显示在 `target` 目录下既有重新打包后的jar，也有原始的jar：

```
$ mvn package
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

如果不包含像上面那样的 `<execution/>` ，你可以自己运行该插件（但只有在package目标也被使用的情况）。例如：

```
$ mvn package spring-boot:repackage
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

如果使用一个里程碑或快照版本，你还需要添加正确的pluginRepository元素：

```
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <url>http://repo.spring.io/snapshot</url>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <url>http://repo.spring.io/milestone</url>
    </pluginRepository>
</pluginRepositories>
```

58.2. 打包可执行jar和war文件

一旦 `spring-boot-maven-plugin` 被包含到你的pom.xml中，它就会自动尝试使用 `spring-boot:repackage` 目标重写存档以使它们能够执行。为了构建一个jar或war，你应该使用常规的packaging元素配置你的项目：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <!-- ... -->
    <packaging>jar</packaging>
    <!-- ... -->
</project>
```

生成的存档在 `package` 阶段会被Spring Boot增强。你想启动的main类即可以通过指定一个配置选项，也可以通过为manifest添加一个 `Main-Class` 属性这种常规的方式实现。如果你没有指定一个main类，该插件会搜索带有 `public static void main(String[] args)` 方法的类。

为了构建和运行一个项目的artifact，你可以输入以下命令：

```
$ mvn package
$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar
```

为了构建一个即是可执行的，又能部署到一个外部容器的war文件，你需要标记内嵌容器依赖为"provided"，例如：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/X
MLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <!-- ... -->
    <packaging>war</packaging>
    <!-- ... -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <!-- ... -->
    </dependencies>
</project>
```

注：具体参考“Section 74.1, “Create a deployable war file”” 章节。

在插件信息页面有高级的配置选项和示例。

59. Spring Boot Gradle插件

Spring Boot Gradle插件为Gradle提供Spring Boot支持，它允许你打包可执行jar或war存档，运行Spring Boot应用，对于"神圣的"依赖可以在你的build.gradle文件中省略版本信息。

59.1. 包含该插件

想要使用Spring Boot Gradle插件，你只需简单的包含一个 `buildscript` 依赖，并应用 `spring-boot` 插件：

```
buildscript {
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.0.BUILD-SNAP
SHOT")
    }
}
apply plugin: 'spring-boot'
```

如果想使用一个里程碑或快照版本，你可以添加相应的repositories引用：

```
buildscript {
    repositories {
        maven.url "http://repo.spring.io/snapshot"
        maven.url "http://repo.spring.io/milestone"
    }
    // ...
}
```

59.2. 声明不带版本的依赖

`spring-boot` 插件会为你的构建注册一个自定义的Gradle `ResolutionStrategy` ，它允许你在声明对"神圣"的artifacts的依赖时获取版本号。为了充分使用该功能，只需要想通常那样声明依赖，但将版本号设置为空：

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    compile("org.thymeleaf:thymeleaf-spring4")
    compile("nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect")
}
```

注：你声明的 `spring-boot` Gradle插件的版本决定了"blessed"依赖的实际版本（确保可以重复构建）。你最好总是将 `spring-boot` gradle插件版本设置为你想用的Spring Boot实际版本。提供的版本详细信息可以在附录中找到。

`spring-boot` 插件对于没有指定版本的依赖只会提供一个版本。如果不想使用插件提供的版本，你可以像平常那样在声明依赖的时候指定版本。例如：

```
dependencies {
    compile("org.thymeleaf:thymeleaf-spring4:2.1.1.RELEASE")
}
```

59.2.1. 自定义版本管理

如果你需要不同于Spring Boot的"blessed"依赖，有可能的话可以自定义 `ResolutionStrategy` 使用的版本。替代的版本元数据使用 `versionManagement` 配置。例如：

```
dependencies {
    versionManagement("com.mycorp:mycorp-versions:1.0.0.RELEASE@properties")
    compile("org.springframework.data:spring-data-hadoop")
}
```

版本信息需要作为一个 `.properties` 文件发布到一个仓库中。对于上面的示例，`mycorp-versions.properties` 文件可能包含以下内容：

```
org.springframework.data\:spring-data-hadoop=2.0.0.RELEASE
```

属性文件优先于Spring Boot默认设置，如果有必要的话可以覆盖版本号。

59.3. 默认排除规则

Gradle处理"exclude rules"的方式和Maven稍微有些不同，在使用starter POMs时这可能会引起无法预料的结果。特别地，当一个依赖可以通过不同的路径访问时，对该依赖声明的exclusions将不会生效。例如，如果一个starter POM声明以下内容：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.0.5.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.0.5.RELEASE</version>
    </dependency>
</dependencies>
```

`commons-logging` jar不会被Gradle排除，因为通过没有 `exclusion` 元素的 `spring-context` 可以传递性的拉取到它（spring-context → spring-core → commons-logging）。

为了确保正确的排除被实际应用，Spring Boot Gradle插件将自动添加排除规则。所有排除被定义在 `spring-boot-dependencies` POM，并且针对"starter" POMs的隐式规则也会被添加。

如果不想自动应用排除规则，你可以使用以下配置：

```
springBoot {
    applyExcludeRules=false
}
```

59.4. 打包可执行jar和war文件

一旦 `spring-boot` 插件被应用到你的项目，它将使用 `bootRepackage` 任务自动尝试重写存档以使它们能够执行。为了构建一个jar或war，你需要按通常的方式配置项目。

你想启动的main类既可以通过一个配置选项指定，也可以通过向manifest添加一个 `Main-Class` 属性。如果你没有指定main类，该插件会搜索带有 `public static void main(String[] args)` 方法的类。

为了构建和运行一个项目artifact，你可以输入以下内容：

```
$ gradle build
$ java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar
```

为了构建一个即能执行也可以部署到外部容器的war包，你需要将内嵌容器依赖标记为"providedRuntime"，比如：

```
...
apply plugin: 'war'

war {
    baseName = 'myapp'
    version =  '0.5.0'
}

repositories {
    jcenter()
    maven { url "http://repo.spring.io/libs-snapshot" }
}

configurations {
    providedRuntime
}

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

注：具体参考“Section 74.1, “Create a deployable war file””。

59.5. 就地（in-place）运行项目

为了在不先构建jar的情况下运行项目，你可以使用"bootRun"任务：

```
$ gradle bootRun
```

默认情况下，以这种方式运行项目可以让你的静态classpath资源（比如，默认位于 `src/main/resources` 下）在应用运行期间被重新加载。使静态资源可以重新加载意味着 `bootRun` 任务不会使用 `processResources` 任务的输出，比如，当调用 `bootRun` 时，你的应用将以资源未处理的形式来使用它们。

你可以禁止直接使用静态classpath资源。这意味着资源不再是可重新加载的，但 `processResources` 任务的输出将会被使用。想要这样做，只需将 `bootRun` 任务的 `addResources` 设为false：

```
bootRun {
    addResources = false
}
```

59.6. Spring Boot插件配置

Gradle插件自动扩展你的构建脚本DSL，它为脚本添加一个 `springBoot` 元素以此作为Boot插件的全局配置。你可以像配置其他Gradle扩展那样为 `springBoot` 设置相应的属性（下面有配置选项列表）。

```
springBoot {
    backupSource = false
}
```

59.7. Repackage配置

该插件添加了一个bootRepackage任务，你可以直接配置它，比如：

```
bootRepackage {
    mainClass = 'demo.Application'
}
```

下面是可用的配置选项：

名称	描述
enabled	布尔值，用于控制repackager的开关（如果你只想要Boot的其他特性而不是这个，那它就派上用场了）
mainClass	要运行的main类。如果没有指定，则使用project属性mainClassName。如果没有定义mainClassName id，则搜索存档以寻找一个合适的类。"合适"意味着一个唯一的，具有良好格式的 `main()` 方法的类（如果找到多个则构建会失败）。你也可以通过"run"任务（main属性）指定main类的名称，和/或将"startScripts"（mainClassName属性）作为"springBoot"配置的替代。
classifier	添加到存档的一个文件名字段（在扩展之前），这样最初保存的存档仍旧存放在最初的位置。在存档被重新打包（repackage）的情况下，该属性默认为null。默认值适用于多数情况，但如果你想在另一个项目中使用原jar作为依赖，最好使用一个扩展来定义该可执行jar
withJarTask	Jar任务的名称或值，用于定位要被repackage的存档
customConfiguration	自定义配置的名称，用于填充内嵌的lib目录（不指定该属性，你将获取所有编译和运行时依赖）

59.8. 使用Gradle自定义配置进行Repackage

有时候不打包解析自compile，runtime和provided作用域的默认依赖可能更合适些。如果创建的可执行jar被原样运行，你需要将所有的依赖内嵌进该jar中；然而，如果目的是explode一个jar文件，并手动运行main类，你可能在CLASSPATH下已经有一些可用的库了。在这种情况下，你可以使用不同的依赖集重新打包（repackage）你的jar。

使用自定义的配置将自动禁用来自compile，runtime和provided作用域的依赖解析。自定义配置即可以定义为全局的（处于springBoot部分内），也可以定义为任务级的。

```
task clientJar(type: Jar) {
    appendix = 'client'
    from sourceSets.main.output
    exclude('**/*Something*')
}

task clientBoot(type: BootRepackage, dependsOn: clientJar) {
    withJarTask = clientJar
    customConfiguration = "mycustomconfiguration"
}
```

在以上示例中，我们创建了一个新的clientJar Jar任务从你编译后的源中打包一个自定义文件集。然后我们创建一个新的clientBoot BootRepackage任务，并让它使用clientJar任务和mycustomconfiguration。

```
configurations {
    mycustomconfiguration.exclude group: 'log4j'
}

dependencies {
    mycustomconfiguration configurations.runtime
}
```

在BootRepackage中引用的配置是一个正常的Gradle配置。在上面的示例中，我们创建了一个新的名叫mycustomconfiguration的配置，指示它来自一个runtime，并排除对log4j的依赖。如果clientBoot任务被执行，重新打包的jar将含有所有来自runtime作用域的依赖，除了log4j jars。

59.8.1. 配置选项

可用的配置选项如下：

名称	描述
mainClass	可执行jar运行的main类
providedConfiguration	provided配置的名称（默认为providedRuntime）
backupSource	在重新打包之前，原先的存档是否备份（默认为true）
customConfiguration	自定义配置的名称
layout	存档类型，对应于内部依赖是如何制定的（默认基于存档类型进行推测）
requiresUnpack	一个依赖列表（格式为"groupId:artifactId"，为了运行，它们需要从fat jars中解压出来。）所有节点被打包进胖jar，但运行的时候它们将被自动解压

59.9. 理解Gradle插件是如何工作的

当 `spring-boot` 被应用到你的Gradle项目，一个默认的名叫 `bootRepackage` 的任务被自动创建。 `bootRepackage` 任务依赖于Gradle `assemble` 任务，当执行时，它会尝试找到所有限定符为空的jar artifacts（也就是说，tests和sources jars被自动跳过）。

由于 `bootRepackage` 查找'所有'创建jar artifacts的事实，Gradle任务执行的顺序就非常重要了。多数项目只创建一个单一的jar文件，所以通常这不是一个问题。然而，如果你正打算创建一个更复杂的，使用自定义 `jar` 和 `BootRepackage` 任务的项目setup，有几个方面需要考虑。

如果'仅仅'从项目创建自定义jar文件，你可以简单地禁用默认的 `jar` 和 `bootRepackage` 任务：

```
jar.enabled = false
bootRepackage.enabled = false
```

另一个选项是指示默认的 `bootRepackage` 任务只能使用一个默认的 `jar` 任务：

```
bootRepackage.withJarTask = jar
```

如果你有一个默认的项目setup，在该项目中，主（main）jar文件被创建和重新打包。并且，你仍旧想创建额外的自定义jars，你可以将自定义的repackage任务结合起来，然后使用 `dependsOn` ，这样 `bootJars` 任务就会在默认的 `bootRepackage` 任务执行以后运行：

```
task bootJars
bootJars.dependsOn = [clientBoot1,clientBoot2,clientBoot3]
build.dependsOn(bootJars)
```

上面所有方面经常用于避免一个已经创建的boot jar又被重新打包的情况。重新打包一个存在的boot jar不是什么大问题，但你可能会发现它包含不必要的依赖。

60. 对其他构建系统的支持

如果想使用除了Maven和Gradle之外的构建工具，你可能需要开发自己的插件。可执行jars需要遵循一个特定格式，并且一些实体需要以不压缩的方式写入（详情查看附录中的可执行jar格式章节）。

Spring Boot Maven和Gradle插件都利用 `spring-boot-loader-tools` 来实际地产生jars。如果需要，你也可以自由地直接使用该库。

60.1. 重新打包存档

使用 `org.springframework.boot.loader.tools.Repackager` 可以将一个存在的存档重新打包，这样它就变成一个自包含的可执行存档。 `Repackager` 类需要提供单一的构造器参数，它引用一个存在的jar或war包。使用两个可用的 `repackage()` 方法中的一个来替换原始的文件或写入一个新的目标。在repackager运行前还可以设置各种配置。

60.2. 内嵌的库

当重新打包一个存档时，你可以使用 `org.springframework.boot.loader.tools.Libraries` 接口来包含对依赖文件的引用。在这里我们不提供任何该Libraries接口的具体实现，因为它们通常跟具体的构建系统相关。

如果你的存档已经包含libraries，你可以使用 `Libraries.NONE` 。

60.3. 查找main类

如果你没有使用 `Repackager.setMainClass()` 指定一个main类，该repackager将使用ASM去读取class文件，然后尝试查找一个合适的，具有 `public static void main(String[] args)` 方法的类。如果发现多个候选者，将会抛出异常。

60.4. repackage实现示例

这里是一个传统的repackage示例：

```
Repackager repackager = new Repackager(sourceJarFile);
repackager.setBackupSource(false);
repackager.repackage(new Libraries() {
            @Override
            public void doWithLibraries(LibraryCallback callback) throws IOException {
                // Build system specific implementation, callback for each dependency
                // callback.library(new Library(nestedFile, LibraryScope.COMPILE));
            }
        });
```

61. 接下来阅读什么

How-to指南

本章节将回答一些常见的"我该怎么做"类型的问题，这些问题在我们使用Spring Boot时经常遇到。这绝不是一个详尽的列表，但它覆盖了很多方面。

如果遇到一个特殊的我们没有覆盖的问题，你可能想去查看stackoverflow.com，看是否有人已经给出了答案;这也是一个很好的提新问题的地方（请使用 `spring-boot` 标签）。

我们也乐意扩展本章节;如果想添加一个'how-to'，你可以给我们发一个pull请求。

62. Spring Boot应用

62.1. 解决自动配置问题

Spring Boot自动配置总是尝试尽最大努力去做正确的事，但有时候会失败并且很难说出失败原因。

在每个Spring Boot ApplicationContext中都存在一个相当有用的ConditionEvaluationReport。如果开启 `DEBUG` 日志输出，你将会看到它。如果你使用 `spring-boot-actuator` ，则会有一个autoconfig的端点，它将以JSON形式渲染该报告。可以使用它调试应用程序，并能查看Spring Boot运行时都添加了哪些特性（及哪些没添加）。

通过查看源码和javadoc可以获取更多问题的答案。以下是一些经验：

- 查找名为 `*AutoConfiguration` 的类并阅读源码，特别是 `@Conditional*` 注解，这可以帮你找出它们启用哪些特性及何时启用。 将 `--debug` 添加到命令行或添加系统属性 `-Ddebug` 可以在控制台查看日志，该日志会记录你的应用中所有自动配置的决策。在一个运行的Actuator app中，通过查看autoconfig端点（ `/autoconfig` 或等效的JMX）可以获取相同信息。
- 查找是 `@ConfigurationProperties` 的类（比如ServerProperties）并看下有哪些可用的外部配置选项。 `@ConfigurationProperties` 类有一个用于充当外部配置前缀的name属性，因此 `ServerProperties` 的值为 `prefix="server"` ，它的配置属性有 `server.port` ， `server.address` 等。在运行的Actuator应用中可以查看configprops端点。
- 查看使用RelaxedEnvironment明确地将配置从Environment暴露出去。它经常会使用一个前缀。
- 查看 `@Value` 注解，它直接绑定到Environment。相比RelaxedEnvironment，这种方式稍微缺乏灵活性，但它也允许松散的绑定，特别是OS环境变量（所以 `CAPITALS_AND_UNDERSCORES` 是 `period.separated` 的同义词）。
- 查看 `@ConditionalOnExpression` 注解，它根据SpEL表达式的结果来开启或关闭特性，通常使用解析自Environment的占位符进行计算。

62.2. 启动前自定义Environment或ApplicationContext

每个SpringApplication都有ApplicationListeners和ApplicationContextInitializers，用于自定义上下文（context）或环境(environment)。Spring Boot从 `META-INF/spring.factories` 下加载很多这样的内部使用的自定义。有很多方法可以注册其他的自定义：

- 以编程方式为每个应用注册自定义，通过在SpringApplication运行前调用它的 `addListeners` 和 `addInitializers` 方法来实现。
- 以声明方式为每个应用注册自定义，通过设置 `context.initializer.classes` 或 `context.listener.classes` 来实现。
- 以声明方式为所有应用注册自定义，通过添加一个 `META-INF/spring.factories` 并打包成一个jar文件（该应用将它作为一个库）来实现。

SpringApplication会给监听器（即使是在上下文被创建之前就存在的）发送一些特定的ApplicationEvents，然后也会注册监听ApplicationContext发布的事件的监听器。查看Spring Boot特性章节中的Section 22.4, “Application events and listeners” 可以获取一个完整列表。

62.4. 创建一个非web（non-web）应用

不是所有的Spring应用都必须是web应用（或web服务）。如果你想在main方法中执行一些代码，但需要启动一个Spring应用去设置需要的底层设施，那使用Spring Boot的`SpringApplication`特性可以很容易实现。`SpringApplication`会根据它是否需要一个web应用来改变它的`ApplicationContext`类。首先你需要做的是去掉servlet API依赖，如果不能这样做（比如，基于相同的代码运行两个应用），那你可以明确地调用`SpringApplication.setWebEnvironment(false)`或设置`applicationContextClass`属性（通过Java API或使用外部配置）。你想运行的，作为业务逻辑的应用代码可以实现为一个`CommandLineRunner`，并将上下文降级为一个`@Bean`定义。

63. 属性&配置

63.1. 外部化SpringApplication配置

SpringApplication已经被属性化（主要是setters），所以你可以在创建应用时使用它的Java API修改它的行为。或者你可以使用properties文件中的 `spring.main.*` 来外部化（在应用代码外配置）这些配置。比如，在 `application.properties` 中可能会有以下内容：

```
spring.main.web_environment=false
spring.main.show_banner=false
```

然后Spring Boot在启动时将不会显示banner，并且该应用也不是一个web应用。

63.2. 改变应用程序外部配置文件的位置

默认情况下，来自不同源的属性以一个定义好的顺序添加到Spring的 `Environment` 中（查看'Sprin Boot特性'章节的Chapter 23, Externalized Configuration获取精确的顺序）。

为应用程序源添加 `@PropertySource` 注解是一种很好的添加和修改源顺序的方法。传递给 `SpringApplication` 静态便利设施（convenience）方法的类和使用 `setSources()` 添加的类都会被检查，以查看它们是否有 `@PropertySources` ，如果有，这些属性会被尽可能早的添加到 `Environment` 里，以确保 `ApplicationContext` 生命周期的所有阶段都能使用。以这种方式添加的属性优先于任何使用默认位置添加的属性，但低于系统属性，环境变量或命令行参数。

你也可以提供系统属性（或环境变量）来改变该行为：

- `spring.config.name` （ `SPRING_CONFIG_NAME` ）是根文件名，默认为 `application` 。
- `spring.config.location` （ `SPRING_CONFIG_LOCATION` ）是要加载的文件（例如，一个classpath资源或一个URL）。Spring Boot为该文档设置一个单独的 `Environment` 属性，它可以被系统属性，环境变量或命令行参数覆盖。

不管你在environment设置什么，Spring Boot都将加载上面讨论过的 `application.properties` 。如果使用YAML，那具有'.yml'扩展的文件默认也会被添加到该列表。

详情参考ConfigFileApplicationListener

63.3. 使用'short'命令行参数

有些人喜欢使用（例如） `--port=9000` 代替 `--server.port=9000` 来设置命令行配置属性。你可以通过在application.properties中使用占位符来启用该功能，比如：

```
server.port=${port:8080}
```

注：如果你继承自 `spring-boot-starter-parent` POM，为了防止和Spring-style的占位符产生冲突， `maven-resources-plugins` 默认的过滤令牌（filter token）已经从 `${*}` 变为 `@` （即 `@maven.token@` 代替了 `${maven.token}` ）。如果已经直接启用maven对application.properties的过滤，你可能也想使用其他的分隔符替换默认的过滤令牌。

注：在这种特殊的情况下，端口绑定能够在一个PaaS环境下工作，比如Heroku和Cloud Foundry，因为在这两个平台中 `PORT` 环境变量是自动设置的，并且Spring能够绑定 `Environment` 属性的大写同义词。

63.4. 使用YAML配置外部属性

YAML是JSON的一个超集，可以非常方便的将外部配置以层次结构形式存储起来。比如：

```
spring:
    application:
        name: cruncher
    datasource:
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost/test
server:
    port: 9000
```

创建一个application.yml文件，将它放到classpath的根目录下，并添加snakeyaml依赖（Maven坐标为 `org.yaml:snakeyaml` ，如果你使用 `spring-boot-starter` 那就已经被包含了）。一个YAML文件会被解析为一个Java `Map<String,Object>` （和一个JSON对象类似），Spring Boot会平伸该map，这样它就只有1级深度，并且有period-separated的keys，跟人们在Java中经常使用的Properties文件非常类似。 上面的YAML示例对应于下面的application.properties文件：

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

查看'Spring Boot特性'章节的Section 23.6, “Using YAML instead of Properties”可以获取更多关于YAML的信息。

63.5. 设置生效的Spring profiles

Spring `Environment` 有一个API可以设置生效的profiles，但通常你会设置一个系统profile（ `spring.profiles.active` ）或一个OS环境变量（ `SPRING_PROFILES_ACTIVE` ）。比如，使用一个 `-D` 参数启动应用程序（记着把它放到main类或jar文件之前）：

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

在Spring Boot中，你也可以在application.properties里设置生效的profile，例如：

```
spring.profiles.active=production
```

通过这种方式设置的值会被系统属性或环境变量替换，但不会被 `SpringApplicationBuilder.profiles()` 方法替换。因此，后面的Java API可用来在不改变默认设置的情况下增加profiles。

想要获取更多信息可查看'Spring Boot特性'章节的Chapter 24, Profiles。

63.6. 根据环境改变配置

一个YAML文件实际上是一系列以 `---` 线分割的文档，每个文档都被单独解析为一个平坦的（flattened）map。

如果一个YAML文档包含一个 `spring.profiles` 关键字，那profiles的值（以逗号分割的profiles列表）将被传入Spring的 `Environment.acceptsProfiles()` 方法，并且如果这些profiles的任何一个被激活，对应的文档被包含到最终的合并中（否则不会）。

示例：

```
server:
    port: 9000
---

spring:
    profiles: development
server:
    port: 9001

---

spring:
    profiles: production
server:
    port: 0
```

在这个示例中，默认的端口是9000，但如果Spring profile 'development'生效则该端口是9001，如果'production'生效则它是0。

YAML文档以它们遇到的顺序合并（所以后面的值会覆盖前面的值）。

想要使用profiles文件完成同样的操作，你可以使用 `application-${profile}.properties` 指定特殊的，profile相关的值。

63.7. 发现外部属性的内置选项

Spring Boot在运行时将来自application.properties（或.yml）的外部属性绑定进一个应用中。在一个地方不可能存在详尽的所有支持属性的列表（技术上也是不可能的），因为你的classpath下的其他jar文件也能够贡献。

每个运行中且有Actuator特性的应用都会有一个 `configprops` 端点，它能够展示所有边界和可通过 `@ConfigurationProperties` 绑定的属性。

附录中包含一个application.properties示例，它列举了Spring Boot支持的大多数常用属性。获取权威列表可搜索 `@ConfigurationProperties` 和 `@Value` 的源码，还有不经常使用的 `RelaxedEnvironment` 。

64. 内嵌的servlet容器

64.1. 为应用添加Servlet，Filter或ServletContextListener

Servlet规范支持的Servlet，Filter，ServletContextListener和其他监听器可以作为 `@Bean` 定义添加到你的应用中。需要格外小心的是，它们不会引起太多的其他beans的热初始化，因为在应用生命周期的早期它们已经被安装到容器里了（比如，让它们依赖你的DataSource或JPA配置就不是一个好主意）。你可以通过延迟初始化它们到第一次使用而不是初始化时来突破该限制。

在Filters和Servlets的情况下，你也可以通过添加一个 `FilterRegistrationBean` 或 `ServletRegistrationBean` 代替或以及底层的组件来添加映射（mappings）和初始化参数。

64.2. 改变HTTP端口

在一个单独的应用中，主HTTP端口默认为8080，但可以使用 `server.port` 设置（比如，在application.properties中或作为一个系统属性）。由于 `Environment` 值的宽松绑定，你也可以使用 `SERVER_PORT` （比如，作为一个OS环境变）。

为了完全关闭HTTP端点，但仍创建一个WebApplicationContext，你可以设置 `server.port=-1` （测试时可能有用）。

想获取更多详情可查看'Spring Boot特性'章节的Section 26.3.3, “Customizing embedded servlet containers”，或ServerProperties源码。

64.3. 使用随机未分配的HTTP端口

想扫描一个未使用的端口（为了防止冲突使用OS本地端口）可以使用 `server.port=0` 。

64.4. 发现运行时的HTTP端口

你可以通过日志输出或它的EmbeddedServletContainer的EmbeddedWebApplicationContext获取服务器正在运行的端口。获取和确认服务器已经初始化的最好方式是添加一个 `ApplicationListener<EmbeddedServletContainerInitializedEvent>` 类型的 `@Bean` ，然后当事件发布时将容器pull出来。

使用 `@WebIntegrationTests` 的一个有用实践是设置 `server.port=0` ，然后使用 `@Value` 注入实际的（'local'）端口。例如：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
@WebIntegrationTest("server.port:0")
public class CityRepositoryIntegrationTests {

    @Autowired
    EmbeddedWebApplicationContext server;

    @Value("${local.server.port}")
    int port;

    // ...

}
```

64.5. 配置SSL

SSL能够以声明方式进行配置，一般通过在application.properties或application.yml设置各种各样的 `server.ssl.*` 属性。例如：

```
server.port = 8443
server.ssl.key-store = classpath:keystore.jks
server.ssl.key-store-password = secret
server.ssl.key-password = another-secret
```

获取所有支持的配置详情可查看Ssl。

注：Tomcat要求key存储（如果你正在使用一个可信存储）能够直接在文件系统上访问，即它不能从一个jar文件内读取。Jetty和Undertow没有该限制。

使用类似于以上示例的配置意味着该应用将不在支持端口为8080的普通HTTP连接。Spring Boot不支持通过application.properties同时配置HTTP连接器和HTTPS连接器。如果你两个都想要，那就需要以编程的方式配置它们中的一个。推荐使用application.properties配置HTTPS，因为HTTP连接器是两个中最容易以编程方式进行配置的。获取示例可查看spring-boot-sample-tomcat-multi-connectors示例项目。

64.6. 配置Tomcat

通常你可以遵循Section 63.7, “Discover built-in options for external properties”关于 `@ConfigurationProperties` （这里主要的是 `ServerProperties` ）的建议，但也看下 `EmbeddedServletContainerCustomizer` 和各种你可以添加的Tomcat-specific的 `*Customizers` 。

Tomcat APIs相当丰富，一旦获取到 `TomcatEmbeddedServletContainerFactory` ，你就能够以多种方式修改它。或核心选择是添加你自己的 `TomcatEmbeddedServletContainerFactory` 。

64.7. 启用Tomcat的多连接器（Multiple Connectors）

你可以将一个 `org.apache.catalina.connector.Connector` 添加到 `TomcatEmbeddedServletContainerFactory` ，这就能够允许多连接器，比如HTTP和HTTPS连接器：

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmbeddedServletContainerF
actory();
    tomcat.addAdditionalTomcatConnectors(createSslConnector());
    return tomcat;
}

private Connector createSslConnector() {
    Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
    Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolHandler();
    try {
        File keystore = new ClassPathResource("keystore").getFile();
        File truststore = new ClassPathResource("keystore").getFile();
        connector.setScheme("https");
        connector.setSecure(true);
        connector.setPort(8443);
        protocol.setSSLEnabled(true);
        protocol.setKeystoreFile(keystore.getAbsolutePath());
        protocol.setKeystorePass("changeit");
        protocol.setTruststoreFile(truststore.getAbsolutePath());
        protocol.setTruststorePass("changeit");
        protocol.setKeyAlias("apitester");
        return connector;
    }
    catch (IOException ex) {
        throw new IllegalStateException("can't access keystore: [" + "keystore"
                + "] or truststore: [" + "keystore" + "]", ex);
    }
}
```

64.8. 在前端代理服务器后使用Tomcat

Spring Boot将自动配置Tomcat的 `RemoteIpValve` ，如果你启用它的话。这允许你透明地使用标准的 `x-forwarded-for` 和 `x-forwarded-proto` 头，很多前端代理服务器都会添加这些头信息（headers）。通过将这些属性中的一个或全部设置为非空的内容来开启该功能（它们是大多数代理约定的值，如果你只设置其中的一个，则另一个也会被自动设置）。

```
server.tomcat.remote_ip_header=x-forwarded-for
server.tomcat.protocol_header=x-forwarded-proto
```

如果你的代理使用不同的头部（headers），你可以通过向application.properties添加一些条目来自定义该值的配置，比如：

```
server.tomcat.remote_ip_header=x-your-remote-ip-header
server.tomcat.protocol_header=x-your-protocol-header
```

该值也可以配置为一个默认的，能够匹配信任的内部代理的正则表达式。默认情况下，受信任的IP包括 10/8, 192.168/16, 169.254/16 和 127/8。可以通过向application.properties添加一个条目来自定义该值的配置，比如：

```
server.tomcat.internal_proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}
```

注：只有在你使用一个properties文件作为配置的时候才需要双反斜杠。如果你使用YAML，单个反斜杠就足够了， `192\.168\.\d{1,3}\.\d{1,3}` 和上面的等价。

另外，通过在一个 `TomcatEmbeddedServletContainerFactory` bean中配置和添加 `RemoteIpValve` ，你就可以完全控制它的设置了。

64.9. 使用Jetty替代Tomcat

Spring Boot starters（特别是spring-boot-starter-web）默认都是使用Tomcat作为内嵌容器的。你需要排除那些Tomcat的依赖并包含Jetty的依赖。为了让这种处理尽可能简单，Spring Boot将Tomcat和Jetty的依赖捆绑在一起，然后提供单独的starters。

Maven示例：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Gradle示例：

```
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.3.0.BUILD-SNAPSHOT")
    compile("org.springframework.boot:spring-boot-starter-jetty:1.3.0.BUILD-SNAPSHOT")
    // ...
}
```

64.10. 配置Jetty

通常你可以遵循Section 63.7, “Discover built-in options for external properties”关于 `@ConfigurationProperties` （此处主要是ServerProperties）的建议，但也要看下 `EmbeddedServletContainerCustomizer` 。Jetty API相当丰富，一旦获取到 `JettyEmbeddedServletContainerFactory` ，你就可以使用很多方式修改它。或更彻底地就是添加你自己的 `JettyEmbeddedServletContainerFactory` 。

64.11. 使用Undertow替代Tomcat

使用Undertow替代Tomcat和使用Jetty替代Tomcat非常类似。你需要排除Tomat依赖，并包含Undertow starter。

Maven示例：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

Gradle示例：

```
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web:1.3.0.BUILD-SNAPSHOT")
    compile 'org.springframework.boot:spring-boot-starter-undertow:1.3.0.BUILD-SNAPSHO
T")
    // ...
}
```

64.12. 配置Undertow

通常你可以遵循Section 63.7, “Discover built-in options for external properties”关于 `@ConfigurationProperties` （此处主要是ServerProperties和ServerProperties.Undertow），但也要看下 `EmbeddedServletContainerCustomizer` 。一旦获取到 `UndertowEmbeddedServletContainerFactory` ，你就可以使用一个 `UndertowBuilderCustomizer` 修改Undertow的配置以满足你的需求。或更彻底地就是添加你自己的 `UndertowEmbeddedServletContainerFactory` 。

64.13. 启用Undertow的多监听器（Multiple Listeners）

往 `UndertowEmbeddedServletContainerFactory` 添加一个 `UndertowBuilderCustomizer` ，然后添加一个监听者到 `Builder` ：

```
@Bean
public UndertowEmbeddedServletContainerFactory embeddedServletContainerFactory() {
    UndertowEmbeddedServletContainerFactory factory = new UndertowEmbeddedServletConta
inerFactory();
    factory.addBuilderCustomizers(new UndertowBuilderCustomizer() {

        @Override
        public void customize(Builder builder) {
            builder.addHttpListener(8080, "0.0.0.0");
        }

    });
    return factory;
}
```

64.14. 使用Tomcat7

Tomcat7可用于Spring Boot，但默认使用的是Tomcat8。如果不能使用Tomcat8（例如，你使用的是Java1.6），你需要改变classpath去引用Tomcat7。

64.14.1. 通过Maven使用Tomcat7

如果正在使用starter pom和parent，你只需要改变Tomcat的version属性，比如，对于一个简单的webapp或service：

```
<properties>
    <tomcat.version>7.0.59</tomcat.version>
</properties>
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    ...
</dependencies>
```

64.14.2. 通过Gradle使用Tomcat7

你可以通过设置 `tomcat.version` 属性改变Tomcat的版本：

```
ext['tomcat.version'] = '7.0.59'
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
}
```

64.15. 使用Jetty8

Jetty8可用于Spring Boot，但默认使用的是Jetty9。如果不能使用Jetty9（例如，因为你使用的是Java1.6），你只需改变classpath去引用Jetty8。你也需要排除Jetty的WebSocket相关的依赖。

64.15.1. 通过Maven使用Jetty8

如果正在使用starter pom和parent，你只需添加Jetty starter，去掉WebSocket依赖，并改变version属性，比如，对于一个简单的webapp或service：

```
<properties>
    <jetty.version>8.1.15.v20140411</jetty.version>
    <jetty-jsp.version>2.2.0.v201112011158</jetty-jsp.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.eclipse.jetty.websocket</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

64.15.2. 通过Gradle使用Jetty8

你可以设置 `jetty.version` 属性并排除相关的WebSocket依赖，比如对于一个简单的webapp或service：

```
ext['jetty.version'] = '8.1.15.v20140411'
dependencies {
    compile ('org.springframework.boot:spring-boot-starter-web') {
        exclude group: 'org.springframework.boot', module: 'spring-boot-starter-tomcat'

    }
    compile ('org.springframework.boot:spring-boot-starter-jetty') {
        exclude group: 'org.eclipse.jetty.websocket'
    }
}
```

64.16. 使用@ServerEndpoint创建WebSocket端点

如果想在一个使用内嵌容器的Spring Boot应用中使用@ServerEndpoint，你需要声明一个单独的ServerEndpointExporter @Bean：

```
@Bean
public ServerEndpointExporter serverEndpointExporter() {
    return new ServerEndpointExporter();
}
```

该bean将用底层的WebSocket容器注册任何的被 `@ServerEndpoint` 注解的beans。当部署到一个单独的servlet容器时，该角色将被一个servlet容器初始化方法履行，ServerEndpointExporter bean也就不是必需的了。

64.17. 启用HTTP响应压缩

Spring Boot提供两种启用HTTP压缩的机制;一种是Tomcat特有的，另一种是使用一个filter，可以配合Jetty，Tomcat和Undertow。

64.17.1. 启用Tomcat的HTTP响应压缩

Tomcat对HTTP响应压缩提供内建支持。默认是禁用的，但可以通过application.properties轻松的启用：

```
server.tomcat.compression: on
```

当设置为 `on` 时，Tomcat将压缩响应的长度至少为2048字节。你可以配置一个整型值来设置该限制而不只是 `on` ，比如：

```
server.tomcat.compression: 4096
```

默认情况下，Tomcat只压缩某些MIME类型的响应（text/html，text/xml和text/plain）。你可以使用 `server.tomcat.compressableMimeTypes` 属性进行自定义，比如：

```
server.tomcat.compressableMimeTypes=application/json,application/xml
```

64.17.2. 使用GzipFilter开启HTTP响应压缩

如果你正在使用Jetty或Undertow，或想要更精确的控制HTTP响应压缩，Spring Boot为Jetty的GzipFilter提供自动配置。虽然该过滤器是Jetty的一部分，但它也兼容Tomcat和Undertow。想要启用该过滤器，只需简单的为你的应用添加 `org.eclipse.jetty:jetty-servlets` 依赖。

GzipFilter可以使用 `spring.http.gzip.*` 属性进行配置。具体参考GzipFilterProperties。

65. Spring MVC

65.1. 编写一个JSON REST服务

在Spring Boot应用中，任何Spring `@RestController` 默认应该渲染为JSON响应，只要classpath下存在Jackson2。例如：

```
@RestController
public class MyController {

    @RequestMapping("/thing")
    public MyThing thing() {
            return new MyThing();
    }

}
```

只要MyThing能够通过Jackson2序列化（比如，一个标准的POJO或Groovy对象），localhost:8080/thing默认响应一个JSON表示。有时在一个浏览器中你可能看到XML响应因为浏览器倾向于发送XML 响应头。

65.2. 编写一个XML REST服务

如果classpath下存在Jackson XML扩展（jackson-dataformat-xml），它会被用来渲染XML响应，示例和JSON的非常相似。想要使用它，只需为你的项目添加以下的依赖：

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

你可能也想添加对Woodstox的依赖。它比JDK提供的默认Stax实现快很多，并且支持良好的格式化输出，提高了namespace处理能力：

```
<dependency>
    <groupId>org.codehaus.woodstox</groupId>
    <artifactId>woodstox-core-asl</artifactId>
</dependency>
```

如果Jackson的XML扩展不可用，Spring Boot将使用JAXB（JDK默认提供），不过你需要为MyThing添加额外的注解`@XmlRootElement`：

```
@XmlRootElement
public class MyThing {
    private String name;
    // .. getters and setters
}
```

想要服务器渲染XML而不是JSON，你可能需要发送一个`Accept: text/xml`头部（或使用浏览器）。

65.3. 自定义Jackson ObjectMapper

在一个HTTP交互中，Spring MVC（客户端和服务端）使用HttpMessageConverters协商内容转换。如果classpath下存在Jackson，你就已经获取到Jackson2ObjectMapperBuilder提供的默认转换器。

创建的ObjectMapper（或用于Jackson XML转换的XmlMapper）实例默认有以下自定义属性：

- `MapperFeature.DEFAULT_VIEW_INCLUSION` 禁用
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 禁用

Spring Boot也有一些简化自定义该行为的特性。

你可以使用当前的environment配置ObjectMapper和XmlMapper实例。Jackson提供一个扩展套件，可以用来简单的关闭或开启一些特性，你可以用它们配置Jackson处理的不同方面。这些特性在Jackson中使用5个枚举进行描述的，并被映射到environment的属性上：

Jackson枚举	Environment属性
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jackson.deserialization.
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jackson.generator.=true
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jackson.mapper.=true
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jackson.parser.=true
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jackson.serialization.=tr

例如，设置 `spring.jackson.serialization.indent_output=true` 可以开启漂亮打印。注意，由于松绑定的使用， `indent_output` 不必匹配对应的枚举常量 `INDENT_OUTPUT` 。

如果想彻底替换默认的ObjectMapper，你需要定义一个该类型的 `@Bean` 并将它标记为 `@Primary` 。

定义一个Jackson2ObjectMapperBuilder类型的 `@Bean` 将允许你自定义默认的ObjectMapper和XmlMapper（分别用于MappingJackson2HttpMessageConverter和MappingJackson2XmlHttpMessageConverter）。

另一种自定义Jackson的方法是向你的上下文添加 `com.fasterxml.jackson.databind.Module` 类型的beans。它们会被注册入每个ObjectMapper类型的bean，当为你的应用添加新特性时，这就提供了一种全局机制来贡献自定义模块。

最后，如果你提供任何MappingJackson2HttpMessageConverter类型的 `@Beans` ，那它们将替换MVC配置中的默认值。同时，也提供一个HttpMessageConverters类型的bean，它有一些有用的方法可以获取默认的和用户增强的message转换器。

想要获取更多细节可查看Section 65.4, “Customize the @ResponseBody rendering”和WebMvcAutoConfiguration源码。

65.4. 自定义@ResponseBody渲染

Spring使用HttpMessageConverters渲染`@ResponseBody`（或来自`@RestController`的响应）。你可以通过在Spring Boot上下文中添加该类型的beans来贡献其他的转换器。如果你添加的bean类型默认已经包含了（像用于JSON转换的MappingJackson2HttpMessageConverter），那它将替换默认的。Spring Boot提供一个方便的HttpMessageConverters类型的bean，它有一些有用的方法可以访问默认的和用户增强的message转换器（有用，比如你想要手动将它们注入到一个自定义的`RestTemplate`）。

在通常的MVC用例中，任何你提供的WebMvcConfigurerAdapter beans通过覆盖configureMessageConverters方法也能贡献转换器，但不同于通常的MVC，你可以只提供你需要的转换器（因为Spring Boot使用相同的机制来贡献它默认的转换器）。最终，如果你通过提供自己的`@EnableWebMvc`注解覆盖Spring Boot默认的MVC配置，那你就可以完全控制，并使用来自WebMvcConfigurationSupport的getMessageConverters手动做任何事。

具体参考WebMvcAutoConfiguration源码。

65.5. 处理Multipart文件上传

Spring Boot采用Servlet 3 `javax.servlet.http.Part` API来支持文件上传。默认情况下，Spring Boot配置Spring MVC在单个请求中每个文件最大1Mb，最多10Mb的文件数据。你可以覆盖那些值，也可以设置临时文件存储的位置（比如，存储到 `/tmp` 文件夹下）及传递数据刷新到磁盘的阀值（通过使用MultipartProperties类暴露的属性）。如果你需要设置文件不受限制，例如，可以设置 `multipart.maxFileSize` 属性值为 `-1` 。

当你想要接收部分（multipart）编码文件数据作为Spring MVC控制器（controller）处理方法中被 `@RequestParam` 注解的MultipartFile类型的参数时，multipart支持就非常有用了。

具体参考MultipartAutoConfiguration源码。

65.6. 关闭Spring MVC DispatcherServlet

Spring Boot想要服务来自应用程序root `/` 下的所有内容。如果你想将自己的servlet映射到该目录下也是可以的，但当然你可能失去一些Boot MVC特性。为了添加你自己的servlet，并将它映射到root资源，你只需声明一个Servlet类型的 `@Bean` ，并给它特定的bean名称 `dispatcherServlet` （如果只想关闭但不替换它，你可以使用该名称创建不同类型的bean）。

65.7. 关闭默认的MVC配置

完全控制MVC配置的最简单方式是提供你自己的被 `@EnableWebMvc` 注解的 `@Configuration` 。这样所有的MVC配置都逃不出你的掌心。

65.8. 自定义ViewResolvers

ViewResolver是Spring MVC的核心组件，它负责转换 `@Controller` 中的视图名称到实际的View实现。注意ViewResolvers主要用在UI应用中，而不是REST风格的服务（View不是用来渲染 `@ResponseBody` 的）。Spring有很多你可以选择的ViewResolver实现，并且Spring自己对如何选择相应实现也没发表意见。另一方面，Spring Boot会根据classpath上的依赖和应用上下文为你安装一或两个ViewResolver实现。DispatcherServlet使用所有在应用上下文中找到的解析器（resolvers），并依次尝试每一个直到它获取到结果，所以如果你正在添加自己的解析器，那就要小心顺序和你的解析器添加的位置。

WebMvcAutoConfiguration将会为你的上下文添加以下ViewResolvers：

- bean id为 `defaultViewResolver` 的InternalResourceViewResolver。这个会定位可以使用DefaultServlet渲染的物理资源（比如，静态资源和JSP页面）。它在视图（view name）上应用了一个前缀和后缀（默认都为空，但你可以通过 `spring.view.prefix` 和 `spring.view.suffix` 外部配置设置），然后查找在servlet上下文中具有该路径的物理资源。可以通过提供相同类型的bean覆盖它。
- id为 `beanNameViewResolver` 的BeanNameViewResolver。这是视图解析器链的一个非常有用的成员，它可以在View被解析时收集任何具有相同名称的beans。
- id为 `viewResolver` 的ContentNegotiatingViewResolver只会在实际View类型的beans出现时添加。这是一个'主'解析器，它的职责会代理给其他解析器，它会尝试找到客户端发送的一个匹配'Accept'的HTTP头部。这有一篇有用的，关于你需要更多了解的ContentNegotiatingViewResolver的博客，也要具体查看下源码。通过定义一个名叫'viewResolver'的bean，你可以关闭自动配置的ContentNegotiatingViewResolver。
- 如果使用Thymeleaf，你将有一个id为 `thymeleafViewResolver` 的ThymeleafViewResolver。它会通过加前缀和后缀的视图名来查找资源（外部配置为 `spring.thymeleaf.prefix` 和 `spring.thymeleaf.suffix` ，对应的默认为'classpath:/templates/'和'.html'）。你可以通过提供相同名称的bean来覆盖它。
- 如果使用FreeMarker，你将有一个id为 `freeMarkerViewResolver` 的FreeMarkerViewResolver。它会使用加前缀和后缀（外部配置为 `spring.freemarker.prefix` 和 `spring.freemarker.suffix` ，对应的默认值为空和'.ftl'）的视图名从加载路径（外部配置为 `spring.freemarker.templateLoaderPath` ，默认为'classpath:/templates/'）下查找资源。你可以通过提供一个相同名称的bean来覆盖它。
- 如果使用Groovy模板（实际上只要你把groovy-templates添加到classpath下），你将有一个id为 `groovyTemplateViewResolver` 的Groovy TemplateViewResolver。它会使用加前缀和后缀（外部属性为 `spring.groovy.template.prefix` 和 `spring.groovy.template.suffix` ，对应的默认值为'classpath:/templates/'和'.tpl'）的视图名从加载路径下查找资源。你可以通过提供一个相同名称的bean来覆盖它。

- 如果使用Velocity，你将有一个id为 `velocityViewResolver` 的VelocityViewResolver。它会使用加前缀和后缀（外部属性为 `spring.velocity.prefix` 和 `spring.velocity.suffix` ，对应的默认值为空和'.vm'）的视图名从加载路径（外部属性为 `spring.velocity.resourceLoaderPath` ，默认为'classpath:/templates/'）下查找资源。你可以通过提供一个相同名称的bean来覆盖它。

具体参考：
WebMvcAutoConfiguration，ThymeleafAutoConfiguration，FreeMarkerAutoConfiguration，GroovyTemplateAutoConfiguration，VelocityAutoConfiguration。

66. 日志

Spring Boot除了commons-logging API外没有其他强制性的日志依赖，你有很多可选的日志实现。想要使用Logback，你需要包含它，及一些对classpath下commons-logging的绑定。最简单的方式是通过依赖 `spring-boot-starter-logging` 的starter pom。对于一个web应用程序，你只需添加 `spring-boot-starter-web` 依赖，因为它依赖于logging starter。例如，使用Maven：

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

Spring Boot有一个LoggingSystem抽象，用于尝试通过classpath上下文配置日志系统。如果Logback可用，则首选它。如果你唯一需要做的就是设置不同日志的级别，那可以通过在application.properties中使用 `logging.level` 前缀实现，比如：

```
logging.level.org.springframework.web: DEBUG
logging.level.org.hibernate: ERROR
```

你也可以使用 `logging.file` 设置日志文件的位置（除控制台之外，默认会输出到控制台）。

想要对日志系统进行更细粒度的配置，你需要使用正在说的LoggingSystem支持的原生配置格式。默认情况下，Spring Boot从系统的默认位置加载原生配置（比如对于Logback为 `classpath:logback.xml` ），但你可以使用 `logging.config` 属性设置配置文件的位置。

66.1. 配置Logback

如果你将一个logback.xml放到classpath根目录下，那它将会被从这加载。Spring Boot提供一个默认的基本配置，如果你只是设置日志级别，那你可以包含它，比如：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

如果查看spring-boot jar包中的默认logback.xml，你将会看到LoggingSystem为你创建的很多有用的系统属性，比如：

- ${PID}，当前进程id
- ${LOG_FILE}，如果在Boot外部配置中设置了 `logging.file`
- ${LOG_PATH}，如果设置了 `logging.path` （表示日志文件产生的目录）

Spring Boot也提供使用自定义的Logback转换器在控制台上输出一些漂亮的彩色ANSI日志信息（不是日志文件）。具体参考默认的 `base.xml` 配置。

如果Groovy在classpath下，你也可以使用logback.groovy配置Logback。

66.2. 配置Log4j

Spring Boot也支持Log4j或Log4j 2作为日志配置，但只有在它们中的某个在classpath下存在的情况。如果你正在使用starter poms进行依赖装配，这意味着你需要排除Logback，然后包含你选择的Log4j版本。如果你不使用starter poms，那除了你选择的Log4j版本外还要提供commons-logging（至少）。

最简单的方式可能就是通过starter poms，尽管它需要排除一些依赖，比如，在Maven中：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j</artifactId>
</dependency>
```

想要使用Log4j 2，只需要依赖 `spring-boot-starter-log4j2` 而不是 `spring-boot-starter-log4j` 。

注：使用Log4j各版本的starters都会收集好依赖以满足common logging的要求（比如，Tomcat中使用 `java.util.logging` ，但使用Log4j或 Log4j 2作为输出）。具体查看Actuator Log4j或Log4j 2的示例，了解如何将它用于实战。

66.2.1. 使用YAML或JSON配置Log4j2

除了它的默认XML配置格式，Log4j 2也支持YAML和JSON配置文件。想要使用其他配置文件格式来配置Log4j 2，你需要添加合适的依赖到classpath。为了使用YAML，你需要添加 `com.fasterxml.jackson.dataformat:jackson-dataformat-yaml` 依赖，Log4j 2将查找名称为 `log4j2.yaml` 或 `log4j2.yml` 的配置文件。为了使用JSON，你需要添加 `com.fasterxml.jackson.core:jackson-databind` 依赖，Log4j 2将查找名称为 `log4j2.json` 或 `log4j2.jsn` 的配置文件

67. 数据访问

67.1. 配置一个数据源

想要覆盖默认的设置只需要定义一个你自己的DataSource类型的 `@Bean` 。Spring Boot提供一个工具构建类DataSourceBuilder，可用来创建一个标准的DataSource（如果它处于classpath下），或者仅创建你自己的DataSource，然后将它和在Section 23.7.1, “Third-party configuration”解释的一系列Environment属性绑定。

比如：

```
@Bean
@ConfigurationProperties(prefix="datasource.mine")
public DataSource dataSource() {
    return new FancyDataSource();
}
```

```
datasource.mine.jdbcUrl=jdbc:h2:mem:mydb
datasource.mine.user=sa
datasource.mine.poolSize=30
```

具体参考'Spring Boot特性'章节中的Section 28.1, “Configure a DataSource”和DataSourceAutoConfiguration类源码。

67.2. 配置两个数据源

创建多个数据源和创建第一个工作都是一样的。如果使用针对JDBC或JPA的默认自动配置，你可能想要将其中一个设置为 `@Primary` （然后它就能被任何 `@Autowired` 注入获取）。

```
@Bean
@Primary
@ConfigurationProperties(prefix="datasource.primary")
public DataSource primaryDataSource() {
    return DataSourceBuilder.create().build();
}

@Bean
@ConfigurationProperties(prefix="datasource.secondary")
public DataSource secondaryDataSource() {
    return DataSourceBuilder.create().build();
}
```

67.3. 使用Spring Data仓库

Spring Data可以为你的 `@Repository` 接口创建各种风格的实现。Spring Boot会为你处理所有事情，只要那些 `@Repositories` 接口跟你的 `@EnableAutoConfiguration` 类处于相同的包（或子包）。

对于很多应用来说，你需要做的就是将正确的Spring Data依赖添加到classpath下（对于JPA有一个 `spring-boot-starter-data-jpa` ，对于Mongodb有一个 `spring-boot-starter-data-mongodb` ），创建一些repository接口来处理 `@Entity` 对象。具体参考JPA sample或Mongodb sample。

Spring Boot会基于它找到的 `@EnableAutoConfiguration` 来尝试猜测你的 `@Repository` 定义的位置。想要获取更多控制，可以使用 `@EnableJpaRepositories` 注解（来自Spring Data JPA）。

67.4. 从Spring配置分离 `@Entity` 定义

Spring Boot会基于它找到的 `@EnableAutoConfiguration` 来尝试猜测你的 `@Entity` 定义的位置。想要获取更多控制，你可以使用 `@EntityScan` 注解，比如：

```
@Configuration
@EnableAutoConfiguration
@EntityScan(basePackageClasses=City.class)
public class Application {

    //...

}
```

67.5. 配置JPA属性

Spring Data JPA已经提供了一些独立的配置选项（比如，针对SQL日志），并且Spring Boot会暴露它们，针对hibernate的外部配置属性也更多些。最常见的选项如下：

```
spring.jpa.hibernate.ddl-auto: create-drop
spring.jpa.hibernate.naming_strategy: org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.database: H2
spring.jpa.show-sql: true
```

（由于宽松的数据绑定策略，连字符或下划线作为属性keys作用应该是等效的）`ddl-auto`配置是个特殊情况，它有不同的默认设置，这取决于你是否使用一个内嵌数据库（create-drop）。当本地EntityManagerFactory被创建时，所有`spring.jpa.properties.*`属性都被作为正常的JPA属性（去掉前缀）传递进去了。

具体参考HibernateJpaAutoConfiguration和JpaBaseConfiguration。

67.6. 使用自定义的EntityManagerFactory

为了完全控制EntityManagerFactory的配置，你需要添加一个名为 `entityManagerFactory` 的 `@Bean` 。Spring Boot自动配置会根据是否存在该类型的bean来关闭它的实体管理器（entity manager）。

67.7. 使用两个EntityManagers

即使默认的EntityManagerFactory工作的很好，你也需要定义一个新的EntityManagerFactory，因为一旦出现第二个该类型的bean，默认的将会被关闭。为了轻松的实现该操作，你可以使用Spring Boot提供的EntityManagerBuilder，或者如果你喜欢的话可以直接使用来自Spring ORM的LocalContainerEntityManagerFactoryBean。

示例：

```
// add two data sources configured as above

@Bean
public LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(
        EntityManagerFactoryBuilder builder) {
    return builder
            .dataSource(customerDataSource())
            .packages(Customer.class)
            .persistenceUnit("customers")
            .build();
}

@Bean
public LocalContainerEntityManagerFactoryBean orderEntityManagerFactory(
        EntityManagerFactoryBuilder builder) {
    return builder
            .dataSource(orderDataSource())
            .packages(Order.class)
            .persistenceUnit("orders")
            .build();
}
```

上面的配置靠自己基本可以运行。想要完成作品你也需要为两个EntityManagers配置TransactionManagers。其中的一个会被Spring Boot默认的JpaTransactionManager获取，如果你将它标记为 `@Primary` 。另一个需要显式注入到一个新实例。或你可以使用一个JTA事物管理器生成它两个。

67.8. 使用普通的persistence.xml

Spring不要求使用XML配置JPA提供者（provider），并且Spring Boot假定你想要充分利用该特性。如果你倾向于使用 `persistence.xml` ，那你需要定义你自己的id为'entityManagerFactory'的LocalEntityManagerFactoryBean类型的 `@Bean` ，并在那设置持久化单元的名称。

默认设置可查看JpaBaseConfiguration

67.9. 使用Spring Data JPA和Mongo仓库

Spring Data JPA和Spring Data Mongo都能自动为你创建Repository实现。如果它们同时出现在classpath下，你可能需要添加额外的配置来告诉Spring Boot你想要哪个（或两个）为你创建仓库。最明确地方式是使用标准的Spring Data `@Enable*Repositories`，然后告诉它你的Repository接口的位置（此处*即可以是Jpa，也可以是Mongo，或者两者都是）。

这里也有 `spring.data.*.repositories.enabled` 标志，可用来在外部配置中开启或关闭仓库的自动配置。这在你想关闭Mongo仓库，但仍旧使用自动配置的MongoTemplate时非常有用。

相同的障碍和特性也存在于其他自动配置的Spring Data仓库类型（Elasticsearch, Solr）。只需要改变对应注解的名称和标志。

67.10. 将Spring Data仓库暴露为REST端点

Spring Data REST能够将Repository的实现暴露为REST端点，只要该应用启用Spring MVC。

Spring Boot暴露一系列来自 `spring.data.rest` 命名空间的有用属性来定制化RepositoryRestConfiguration。如果需要提供其他定制，你可以创建一个继承自SpringBootRepositoryRestMvcConfiguration的 `@Configuration` 类。该类功能和RepositoryRestMvcConfiguration相同，但允许你继续使用 `spring.data.rest.*` 属性。

68. 数据库初始化

一个数据库可以使用不同的方式进行初始化，这取决于你的技术栈。或者你可以手动完成该任务，只要数据库是单独的过程。

68.1. 使用JPA初始化数据库

JPA有个生成DDL的特性，这些可以设置为在数据库启动时运行。这可以通过两个外部属性进行控制：

- `spring.jpa.generate-ddl`（boolean）控制该特性的关闭和开启，跟实现者没关系
- `spring.jpa.hibernate.ddl-auto`（enum）是一个Hibernate特性，用于更细力度的控制该行为。更多详情参考以下内容。

68.2. 使用Hibernate初始化数据库

你可以显式设置 `spring.jpa.hibernate.ddl-auto`，标准的Hibernate属性值有 `none`，`validate`，`update`，`create`，`create-drop`。Spring Boot根据你的数据库是否为内嵌数据库来选择相应的默认值，如果是内嵌型的则默认值为 `create-drop`，否则为 `none`。通过查看Connection类型可以检查是否为内嵌型数据库，hsqldb，h2和derby是内嵌的，其他都不是。当从内存数据库迁移到一个真正的数据库时，你需要当心，在新的平台中不能对数据库表和数据是否存在进行臆断。你也需要显式设置 `ddl-auto`，或使用其他机制初始化数据库。

此外，启动时处于classpath根目录下的import.sql文件会被执行。这在demos或测试时很有用，但在生产环境中你可能不期望这样。这是Hibernate的特性，和Spring没有一点关系。

68.3. 使用Spring JDBC初始化数据库

Spring JDBC有一个DataSource初始化特性。Spring Boot默认启用了该特性，并从标准的位置schema.sql和data.sql（位于classpath根目录）加载SQL。此外，Spring Boot将加载 `schema-${platform}.sql` 和 `data-${platform}.sql` 文件（如果存在），在这里platform是 `spring.datasource.platform` 的值，比如，你可以将它设置为数据库的供应商名称（hsqldb, h2, oracle, mysql, postgresql等）。Spring Boot默认启用Spring JDBC初始化快速失败特性，所以如果脚本导致异常产生，那应用程序将启动失败。脚本的位置可以通过设置 `spring.datasource.schema` 和 `spring.datasource.data` 来改变，如果设置 `spring.datasource.initialize=false` 则哪个位置都不会被处理。

你可以设置 `spring.datasource.continueOnError=true` 禁用快速失败特性。一旦应用程序成熟并被部署了很多次，那该设置就很有用，因为脚本可以充当"可怜人的迁移"-例如，插入失败时意味着数据已经存在，也就没必要阻止应用继续运行。

如果你想要在一个JPA应用中使用schema.sql，那如果Hibernate试图创建相同的表， `ddl-auto=create-drop` 将导致错误产生。为了避免那些错误，可以将 `ddl-auto` 设置为“”（推荐）或“none”。不管是否使用 `ddl-auto=create-drop` ，你总可以使用data.sql初始化新数据。

68.4. 初始化Spring Batch数据库

如果你正在使用Spring Batch，那么它会为大多数的流行数据库平台预装SQL初始化脚本。Spring Boot会检测你的数据库类型，并默认执行那些脚本，在这种情况下将关闭快速失败特性（错误被记录但不会阻止应用启动）。这是因为那些脚本是可信任的，通常不会包含bugs，所以错误会被忽略掉，并且对错误的忽略可以让脚本具有幂等性。你可以使用 `spring.batch.initializer.enabled=false` 显式关闭初始化功能。

68.5. 使用一个高级别的数据迁移工具

Spring Boot跟高级别的数据迁移工具Flyway(基于SQL)和Liquibase(XML)工作的很好。通常我们倾向于Flyway，因为它一眼看去好像很容易，另外它通常不需要平台独立：一般一个或至多需要两个平台。

68.5.1. 启动时执行Flyway数据库迁移

想要在启动时自动运行Flyway数据库迁移，需要将 `org.flywaydb:flyway-core` 添加到你的classpath下。

迁移是一些 `V<VERSION>__<NAME>.sql` 格式的脚本（ `<VERSION>` 是一个下划线分割的版本号，比如'1'或'2_1'）。默认情况下，它们存放在一个 `classpath:db/migration` 的文件夹中，但你可以使用 `flyway.locations` （一个列表）来改变它。详情可参考flyway-core中的Flyway类，查看一些可用的配置，比如schemas。Spring Boot在FlywayProperties中提供了一个小的属性集，可用于禁止迁移，或关闭位置检测。

默认情况下，Flyway将自动注入（ `@Primary` ）DataSource到你的上下文，并用它进行数据迁移。如果你想使用一个不同的DataSource，你可以创建一个，并将它标记为 `@FlywayDataSource` 的 `@Bean` -如果你这样做了，且想要两个数据源，记得创建另一个并将它标记为 `@Primary` 。或者你可以通过在外部配置文件中设置 `flyway.[url,user,password]` 来使用Flyway的原生DataSource。

这是一个Flyway示例，你可以作为参考。

68.5.2. 启动时执行Liquibase数据库迁移

想要在启动时自动运行Liquibase数据库迁移，你需要将 `org.liquibase:liquibase-core` 添加到classpath下。

主改变日志（master change log）默认从 `db/changelog/db.changelog-master.yaml` 读取，但你可以使用 `liquibase.change-log` 进行设置。详情查看LiquibaseProperties以获取可用设置，比如上下文，默认的schema等。

这里有个Liquibase示例可作为参考。

69. 批处理应用

69.1. 在启动时执行Spring Batch作业

你可以在上下文的某个地方添加 `@EnableBatchProcessing` 来启用Spring Batch的自动配置功能。

默认情况下，在启动时它会执行应用的所有作业（Jobs），具体查看JobLauncherCommandLineRunner。你可以通过指定 `spring.batch.job.names` （多个作业名以逗号分割）来缩小到一个特定的作业或多个作业。

如果应用上下文包含一个JobRegistry，那么处于 `spring.batch.job.names` 中的作业将会从registry中查找，而不是从上下文中自动装配。这是复杂系统中常见的一个模式，在这些系统中多个作业被定义在子上下文和注册中心。

具体参考BatchAutoConfiguration和@EnableBatchProcessing。

70. 执行器（Actuator）

70.1. 改变HTTP端口或执行器端点的地址

在一个单独的应用中，执行器的HTTP端口默认和主HTTP端口相同。想要让应用监听不同的端口，你可以设置外部属性`management.port`。为了监听一个完全不同的网络地址（比如，你有一个用于管理的内部网络和一个用于用户应用程序的外部网络），你可以将`management.address`设置为一个可用的IP地址，然后将服务器绑定到该地址。

查看ManagementServerProperties源码和'Production-ready特性'章节中的Section 41.3, “Customizing the management server port”来获取更多详情。

70.2. 自定义'白标'（whitelabel，可以了解下相关理念）错误页面

Spring Boot安装了一个'whitelabel'错误页面，如果你遇到一个服务器错误（机器客户端消费的是JSON，其他媒体类型则会看到一个具有正确错误码的合乎情理的响应），那就能在客户端浏览器中看到该页面。你可以设置 `error.whitelabel.enabled=false` 来关闭该功能，但通常你想要添加自己的错误页面来取代whitelabel。确切地说，如何实现取决于你使用的模板技术。例如，你正在使用Thymeleaf，你将添加一个error.html模板。如果你正在使用FreeMarker，那你将添加一个error.ftl模板。通常，你需要的只是一个名称为error的View，和/或一个处理 `/error` 路径的 `@Controller` 。除非你替换了一些默认配置，否则你将在你的ApplicationContext中找到一个BeanNameViewResolver，所以一个id为error的 `@Bean` 可能是完成该操作的一个简单方式。详情参考ErrorMvcAutoConfiguration。

查看Error Handling章节，了解下如何将处理器（handlers）注册到servlet容器中。

71. 安全

71.1. 关闭Spring Boot安全配置

不管你在应用的什么地方定义了一个使用`@EnableWebSecurity`注解的`@Configuration`，它将会关闭Spring Boot中的默认webapp安全设置。想要调整默认值，你可以尝试设置`security.*`属性（具体查看SecurityProperties和常见应用属性的SECURITY章节）。

71.2. 改变AuthenticationManager并添加用户账号

如果你提供了一个AuthenticationManager类型的 `@Bean` ，那么默认的就不会被创建了，所以你可以获得Spring Security可用的全部特性（比如，不同的认证选项）。

Spring Security也提供了一个方便的AuthenticationManagerBuilder，可用于构建具有常见选项的AuthenticationManager。在一个webapp中，推荐将它注入到WebSecurityConfigurerAdapter的一个void方法中，比如：

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
            auth.inMemoryAuthentication()
                .withUser("barry").password("password").roles("USER"); // ... etc.
    }

    // ... other stuff for application security
}
```

如果把它放到一个内部类或一个单独的类中，你将得到最好的结果（也就是不跟很多其他 `@Beans` 混合在一起将允许你改变实例化的顺序）。secure web sample是一个有用的参考模板。

如果你遇到了实例化问题（比如，使用JDBC或JPA进行用户详细信息的存储），那将AuthenticationManagerBuilder回调提取到一个GlobalAuthenticationConfigurerAdapter（放到init()方法内以防其他地方也需要authentication manager）可能是个不错的选择，比如：

```
@Configuration
public class AuthenticationManagerConfiguration extends

    GlobalAuthenticationConfigurerAdapter {
    @Override
    public void init(AuthenticationManagerBuilder auth) {
        auth.inMemoryAuthentication() // ... etc.
    }

}
```

71.3. 当前端使用代理服务器时，启用HTTPS

对于任何应用来说，确保所有的主端点（URL）都只在HTTPS下可用是个重要的苦差事。如果你使用Tomcat作为servlet容器，那Spring Boot如果发现一些环境设置的话，它将自动添加Tomcat自己的RemoteIpValve，你也可以依赖于HttpServletRequest来报告是否请求是安全的（即使代理服务器的downstream处理真实的SSL终端）。这个标准行为取决于某些请求头是否出现（`x-forwarded-for`和`x-forwarded-proto`），这些请求头的名称都是约定好的，所以对于大多数前端和代理都是有效的。

你可以向application.properties添加以下设置里开启该功能，比如：

```
server.tomcat.remote_ip_header=x-forwarded-for
server.tomcat.protocol_header=x-forwarded-proto
```

（这些属性出现一个就会开启该功能，或者你可以通过添加一个TomcatEmbeddedServletContainerFactory bean自己添加RemoteIpValve）

Spring Security也可以配置成针对所以或某些请求需要一个安全渠道（channel）。想要在一个Spring Boot应用中开启它，你只需将application.properties中的`security.require_ssl`设置为`true`即可。

72. 热交换

72.1. 重新加载静态内容

Spring Boot有很多用于热加载的选项。使用IDE开发是一个不错的方式，特别是需要调试的时候（所有的现代IDEs都允许重新加载静态资源，通常也支持对变更的Java类进行热交换）。Maven和Gradle插件也支持命令行下的静态文件热加载。如果你使用其他高级工具编写css/js，并使用外部的css/js编译器，那你就可以充分利用该功能。

72.2. 在不重启容器的情况下重新加载Thymeleaf模板

如果你正在使用Thymeleaf，那就将 `spring.thymeleaf.cache` 设置为false。查看ThymeleafAutoConfiguration可以获取其他Thymeleaf自定义选项。

72.3. 在不重启容器的情况下重新加载FreeMarker模板

如果你正在使用FreeMarker，那就将 `spring.freemarker.cache` 设置为false。查看FreeMarkerAutoConfiguration 可以获取其他FreeMarker自定义选项。

72.4. 在不重启容器的情况下重新加载Groovy模板

如果你正在使用Groovy模板，那就将 `spring.groovy.template.cache` 设置为false。查看GroovyTemplateAutoConfiguration可以获取其他Groovy自定义选项。

72.5. 在不重启容器的情况下重新加载Velocity模板

如果你正在使用Velocity，那就将 `spring.velocity.cache` 设置为false。查看VelocityAutoConfiguration可以获取其他Velocity自定义选项。

72.6. 在不重启容器的情况下重新加载Java类

现代IDEs（Eclipse, IDEA等）都支持字节码的热交换，所以如果你做了一个没有影响类或方法签名的改变，它会利索地重新加载并没有任何影响。

Spring Loaded在这方面走的更远，它能够重新加载方法签名改变的类定义。如果对它进行一些自定义配置可以强制ApplicationContext刷新自己（但没有通用的机制来确保这对一个运行中的应用总是安全的，所以它可能只是一个开发时间的技巧）。

72.6.1. 使用Maven配置Spring Loaded

为了在Maven命令行下使用Spring Loaded，你只需将它作为一个依赖添加到Spring Boot插件声明中即可，比如：

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>springloaded</artifactId>
            <version>1.2.0.RELEASE</version>
        </dependency>
    </dependencies>
</plugin>
```

正常情况下，这在Eclipse和IntelliJ中工作的相当漂亮，只要它们有相应的，和Maven默认一致的构建配置（Eclipse m2e对此支持的更好，开箱即用）。

72.6.2. 使用Gradle和IntelliJ配置Spring Loaded

如果想将Spring Loaded和Gradle，IntelliJ结合起来，那你需要付出代价。默认情况下，IntelliJ将类编译到一个跟Gradle不同的位置，这会导致Spring Loaded监控失败。

为了正确配置IntelliJ，你可以使用 `idea` Gradle插件：

```
buildscript {
    repositories { jcenter() }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-plugin:1.3.0.BUILD-SNAP
SHOT"
        classpath 'org.springframework:springloaded:1.2.0.RELEASE'
    }
}

apply plugin: 'idea'

idea {
    module {
        inheritOutputDirs = false
        outputDir = file("$buildDir/classes/main/")
    }
}

// ...
```

注：IntelliJ必须配置跟命令行Gradle任务相同的Java版本，并且springloaded必须作为一个buildscript依赖被包含进去。

此外，你也可以启用Intellij内部的 `Make Project Automatically` ，这样不管什么时候只要文件被保存都会自动编译你的代码。

73. 构建

73.1. 使用Maven自定义依赖版本

如果你使用Maven进行一个直接或间接继承 `spring-boot-dependencies` （比如 `spring-boot-starter-parent` ）的构建，并想覆盖一个特定的第三方依赖，那你可以添加合适的 `<properties>` 元素。浏览spring-boot-dependencies POM可以获取一个全面的属性列表。例如，想要选择一个不同的slf4j版本，你可以添加以下内容：

```
<properties>
    <slf4j.version>1.7.5<slf4j.version>
</properties>
```

注：这只在你的Maven项目继承（直接或间接）自 `spring-boot-dependencies` 才有用。如果你使用 `<scope>import</scope>` ，将 `spring-boot-dependencies` 添加到自己的 `dependencyManagement` 片段，那你必须自己重新定义artifact而不是覆盖属性。

注：每个Spring Boot发布都是基于一些特定的第三方依赖集进行设计和测试的，覆盖版本可能导致兼容性问题。

73.2. 使用Maven创建可执行JAR

`spring-boot-maven-plugin` 能够用来创建可执行的'胖'JAR。如果你正在使用 `spring-boot-starter-parent` POM，你可以简单地声明该插件，然后你的jar将被重新打包：

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

如果没有使用parent POM，你仍旧可以使用该插件。不过，你需要另外添加一个 `<executions>` 片段：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>1.3.0.BUILD-SNAPSHOT</version>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

查看插件文档获取详细的用例。

73.3. 创建其他的可执行JAR

如果你想将自己的项目以library jar的形式被其他项目依赖，并且需要它是一个可执行版本（例如demo），你需要使用略微不同的方式来配置该构建。

对于Maven来说，正常的JAR插件和Spring Boot插件都有一个'classifier'，你可以添加它来创建另外的JAR。示例如下（使用Spring Boot Starter Parent管理插件版本，其他配置采用默认设置）：

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

上述配置会产生两个jars，默认的一个和使用带有classifier 'exec'的Boot插件构建的可执行的一个。

对于Gradle用户来说，步骤类似。示例如下：

```
bootRepackage  {
    classifier = 'exec'
}
```

73.4. 在可执行jar运行时提取特定的版本

在一个可执行jar中，为了运行，多数内嵌的库不需要拆包（unpacked），然而有一些库可能会遇到问题。例如，JRuby包含它自己的内嵌jar，它假定 `jruby-complete.jar` 本身总是能够直接作为文件访问的。

为了处理任何有问题的库，你可以标记那些特定的内嵌jars，让它们在可执行jar第一次运行时自动解压到一个临时文件夹中。例如，为了将JRuby标记为使用Maven插件拆包，你需要添加如下的配置：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <requiresUnpack>
                    <dependency>
                        <groupId>org.jruby</groupId>
                        <artifactId>jruby-complete</artifactId>
                    </dependency>
                </requiresUnpack>
            </configuration>
        </plugin>
    </plugins>
</build>
```

使用Gradle完全上述操作：

```
springBoot  {
    requiresUnpack = ['org.jruby:jruby-complete']
}
```

73.6. 远程调试一个使用Maven启动的Spring Boot项目

想要为使用Maven启动的Spring Boot应用添加一个远程调试器，你可以使用mave插件的jvmArguments属性。详情参考示例。

73.7. 远程调试一个使用Gradle启动的Spring Boot项目

想要为使用Gradle启动的Spring Boot应用添加一个远程调试器，你可以使用build.gradle的applicationDefaultJvmArgs属性或`--debug-jvm`命令行选项。

build.gradle：

```
applicationDefaultJvmArgs = [
    "-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005"
]
```

命令行：

```
$ gradle run --debug-jvm
```

详情查看Gradle应用插件。

73.8. 使用Ant构建可执行存档（archive）

想要使用Ant进行构建，你需要抓取依赖，编译，然后像通常那样创建一个jar或war存档。为了让它可以执行：

1. 使用合适的启动器配置 `Main-Class` ，比如对于jar文件使用JarLauncher，然后将其他需要的属性以manifest实体指定，主要是一个 `Start-Class` 。
2. 将运行时依赖添加到一个内嵌的'lib'目录（对于jar），`provided` （内嵌容器）依赖添加到一个内嵌的 `lib-provided` 目录。记住不要压缩存档中的实体。
3. 在存档的根目录添加 `spring-boot-loader` 类（这样 `Main-Class` 就可用了）。

示例：

```
<target name="build" depends="compile">
    <copy todir="target/classes/lib">
        <fileset dir="lib/runtime" />
    </copy>
    <jar destfile="target/spring-boot-sample-actuator-${spring-boot.version}.jar" comp
ress="false">
        <fileset dir="target/classes" />
        <fileset dir="src/main/resources" />
        <zipfileset src="lib/loader/spring-boot-loader-jar-${spring-boot.version}.jar"
 />
        <manifest>
            <attribute name="Main-Class" value="org.springframework.boot.loader.JarLau
ncher" />
            <attribute name="Start-Class" value="${start-class}" />
        </manifest>
    </jar>
</target>
```

该Actuator示例中有一个build.xml文件，可以使用以下命令来运行：

```
$ ant -lib <path_to>/ivy-2.2.jar
```

在上述操作之后，你可以使用以下命令运行该应用：

```
$ java -jar target/*.jar
```

73.9. 如何使用Java6

如果想在Java6环境中使用Spring Boot，你需要改变一些配置。具体的变化取决于你应用的功能。

73.9.1. 内嵌Servlet容器兼容性

如果你在使用Boot的内嵌Servlet容器，你需要使用一个兼容Java6的容器。Tomcat 7和Jetty 8都是Java 6兼容的。具体参考Section 63.15, “Use Tomcat 7”和Section 63.16, “Use Jetty 8”。

73.9.2. JTA API兼容性

Java事务API自身并不要求Java 7，而是官方的API jar包含的已构建类要求Java 7。如果你正在使用JTA，那么你需要使用能够在Java 6工作的构建版本替换官方的JTA 1.2 API jar。为了完成该操作，你需要排除任何对 `javax.transaction:javax.transaction-api` 的传递依赖，并使用 `org.jboss.spec.javax.transaction:jboss-transaction-api_1.2_spec:1.0.0.Final` 依赖替换它们。

74. 传统部署

74.1. 创建一个可部署的war文件

产生一个可部署war包的第一步是提供一个SpringBootServletInitializer子类，并覆盖它的configure方法。这充分利用了Spring框架对Servlet 3.0的支持，并允许你在应用通过servlet容器启动时配置它。通常，你只需把应用的主类改为继承SpringBootServletInitializer即可：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
{
        return application.sources(Application.class);
    }

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

}
```

下一步是更新你的构建配置，这样你的项目将产生一个war包而不是jar包。如果你使用Maven，并使用`spring-boot-starter-parent`（为了配置Maven的war插件），所有你需要做的就是更改pom.xml的packaging为war：

```
<packaging>war</packaging>
```

如果你使用Gradle，你需要修改build.gradle来将war插件应用到项目上：

```
apply plugin: 'war'
```

该过程最后的一步是确保内嵌的servlet容器不能干扰war包将部署的servlet容器。为了达到这个目的，你需要将内嵌容器的依赖标记为provided。

如果使用Maven：

```
<dependencies>
    <!-- … -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <!-- … -->
</dependencies>
```

如果使用Gradle：

```
dependencies {
    // …
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    // …
}
```

如果你使用Spring Boot构建工具，将内嵌容器依赖标记为provided将产生一个可执行war包，在`lib-provided`目录有该war包的provided依赖。这意味着，除了部署到servlet容器，你还可以通过使用命令行`java -jar`命令来运行应用。

注：查看Spring Boot基于以上配置的一个Maven示例应用。

74.2. 为老的servlet容器创建一个可部署的war文件

老的Servlet容器不支持在Servlet 3.0中使用的ServletContextInitializer启动处理。你仍旧可以在这些容器使用Spring和Spring Boot，但你需要为应用添加一个web.xml，并将它配置为通过一个DispatcherServlet加载一个ApplicationContext。

74.3. 将现有的应用转换为Spring Boot

对于一个非web项目，转换为Spring Boot应用很容易（抛弃创建ApplicationContext的代码，取而代之的是调用SpringApplication或SpringApplicationBuilder）。Spring MVC web应用通常先创建一个可部署的war应用，然后将它迁移为一个可执行的war或jar。建议阅读Getting Started Guide on Converting a jar to a war.。

通过继承SpringBootServletInitializer创建一个可执行war（比如，在一个名为Application的类中），然后添加Spring Boot的 `@EnableAutoConfiguration` 注解。示例：

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
{
        // Customize the application or call application.sources(...) to add sources
        // Since our example is itself a @Configuration class we actually don't
        // need to override this method.
        return application;
    }

}
```

记住不管你往sources放什么东西，它仅是一个Spring ApplicationContext，正常情况下，任何生效的在这里也会起作用。有一些beans你可以先移除，然后让Spring Boot提供它的默认实现，不过有可能需要先完成一些事情。

静态资源可以移到classpath根目录下的 `/public` （或 `/static` ， `/resources` ， `/META-INF/resources` ）。同样的方式也适合于 `messages.properties` （Spring Boot在classpath根目录下自动发现这些配置）。

美妙的（Vanilla usage of）Spring DispatcherServlet和Spring Security不需要改变。如果你的应用有其他特性，比如使用其他servlets或filters，那你可能需要添加一些配置到你的Application上下文中，按以下操作替换web.xml的那些元素：

- 在容器中安装一个Servlet或ServletRegistrationBean类型的 `@Bean` ，就好像web.xml中的 `<servlet/>` 和 `<servlet-mapping/>` 。
- 同样的添加一个Filter或FilterRegistrationBean类型的 `@Bean` （类似于 `<filter/>` 和 `<filter-mapping/>` ）。
- 在XML文件中的ApplicationContext可以通过 `@Import` 添加到你的Application中。简单的情况下，大量使用注解配置可以在几行内定义 `@Bean` 定义。

一旦war可以使用，我们就通过添加一个main方法到Application来让它可以执行，比如：

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

应用可以划分为多个类别：

- 没有web.xml的Servlet 3.0+应用
- 有web.xml的应用
- 有上下文层次的应用
- 没有上下文层次的应用

所有这些都可以进行适当的转化，但每个可能需要稍微不同的技巧。

Servlet 3.0+的应用转化的相当简单，如果它们已经使用Spring Servlet 3.0+初始化器辅助类。通常所有来自一个存在的WebApplicationInitializer的代码可以移到一个SpringBootServletInitializer中。如果一个存在的应用有多个ApplicationContext（比如，如果它使用AbstractDispatcherServletInitializer），那你可以将所有上下文源放进一个单一的SpringApplication。你遇到的主要难题可能是如果那样不能工作，那你就要维护上下文层次。参考示例entry on building a hierarchy。一个存在的包含web相关特性的父上下文通常需要分解，这样所有的ServletContextAware组件都处于子上下文中。

对于还不是Spring应用的应用来说，上面的指南有助于你把应用转换为一个Spring Boot应用，但你也可以选择其他方式。

74.4. 部署WAR到Weblogic

想要将Spring Boot应用部署到Weblogic，你需要确保你的servlet初始化器直接实现WebApplicationInitializer（即使你继承的基类已经实现了它）。

一个传统的Weblogic初始化器可能如下所示：

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.web.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements WebApplicat
ionInitializer {

}
```

如果使用logback，你需要告诉Weblogic你倾向使用的打包版本而不是服务器预装的版本。你可以通过添加一个具有如下内容的 `WEB-INF/weblogic.xml` 实现该操作：

```
<?xml version="1.0" encoding="UTF-8"?>
<wls:weblogic-web-app
    xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
        http://xmlns.oracle.com/weblogic/weblogic-web-app
        http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd">
    <wls:container-descriptor>
        <wls:prefer-application-packages>
            <wls:package-name>org.slf4j</wls:package-name>
        </wls:prefer-application-packages>
    </wls:container-descriptor>
</wls:weblogic-web-app>
```

X.附录

附录A. 常见应用属性

你可以在 `application.properties/application.yml` 文件内部或通过命令行开关来指定各种属性。本章节提供了一个常见Spring Boot属性的列表及使用这些属性的底层类的引用。

注：属性可以来自classpath下的其他jar文件中，所以你不应该把它当成详尽的列表。定义你自己的属性也是相当合法的。

注：示例文件只是一个指导。不要拷贝/粘贴整个内容到你的应用，而是只提取你需要的属性。

```
# ===================================================================
# COMMON SPRING BOOT PROPERTIES
#
# This sample file is provided as a guideline. Do NOT copy it in its
# entirety to your own application.               ^^^
# ===================================================================

# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------

# SPRING CONFIG (ConfigFileApplicationListener)
spring.config.name= # config file name (default to 'application')
spring.config.location= # location of config file

# PROFILES
spring.profiles.active= # comma list of active profiles
spring.profiles.include= # unconditionally activate the specified comma separated prof
iles

# APPLICATION SETTINGS (SpringApplication)
spring.main.sources=
spring.main.web-environment= # detect by default
spring.main.show-banner=true
spring.main....= # see class for all properties

# LOGGING
logging.path=/var/logs
logging.file=myapp.log
logging.config= # location of config file (default classpath:logback.xml for logback)
logging.level.*= # levels for loggers, e.g. "logging.level.org.springframework=DEBUG"
(TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF)

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.name=
spring.application.index=

# EMBEDDED SERVER CONFIGURATION (ServerProperties)
```

```
server.port=8080
server.address= # bind to a specific NIC
server.session-timeout= # session timeout in seconds
server.context-parameters.*= # Servlet context init parameters, e.g. server.context-pa
rameters.a=alpha
server.context-path= # the context path, defaults to '/'
server.servlet-path= # the servlet path, defaults to '/'
server.ssl.enabled=true # if SSL support is enabled
server.ssl.client-auth= # want or need
server.ssl.key-alias=
server.ssl.ciphers= # supported SSL ciphers
server.ssl.key-password=
server.ssl.key-store=
server.ssl.key-store-password=
server.ssl.key-store-provider=
server.ssl.key-store-type=
server.ssl.protocol=TLS
server.ssl.trust-store=
server.ssl.trust-store-password=
server.ssl.trust-store-provider=
server.ssl.trust-store-type=
server.tomcat.access-log-pattern= # log pattern of the access log
server.tomcat.access-log-enabled=false # is access logging enabled
server.tomcat.compression=off # is compression enabled (off, on, or an integer content
 length limit)
server.tomcat.compressable-mime-types=text/html,text/xml,text/plain # comma-separated
list of mime types that Tomcat will compress
server.tomcat.internal-proxies=10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\
        192\\.168\\.\\d{1,3}\\.\\d{1,3}|\\
        169\\.254\\.\\d{1,3}\\.\\d{1,3}|\\
        127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3} # regular expression matching trusted IP
addresses
server.tomcat.protocol-header=x-forwarded-proto # front end proxy forward header
server.tomcat.port-header= # front end proxy port header
server.tomcat.remote-ip-header=x-forwarded-for
server.tomcat.basedir=/tmp # base dir (usually not needed, defaults to tmp)
server.tomcat.background-processor-delay=30; # in seconds
server.tomcat.max-http-header-size= # maximum size in bytes of the HTTP message header
server.tomcat.max-threads = 0 # number of threads in protocol handler
server.tomcat.uri-encoding = UTF-8 # character encoding to use for URL decoding

# SPRING MVC (WebMvcProperties)
spring.mvc.locale= # set fixed locale, e.g. en_UK
spring.mvc.date-format= # set fixed date format, e.g. dd/MM/yyyy
spring.mvc.favicon.enabled=true
spring.mvc.message-codes-resolver-format= # PREFIX_ERROR_CODE / POSTFIX_ERROR_CODE
spring.mvc.ignore-default-model-on-redirect=true # If the the content of the "default"
 model should be ignored redirects
spring.view.prefix= # MVC view prefix
spring.view.suffix= # ... and suffix

# SPRING RESOURCES HANDLING (ResourceProperties)
spring.resources.cache-period= # cache timeouts in headers sent to browser
```

```
spring.resources.add-mappings=true # if default mappings should be added

# MULTIPART (MultipartProperties)
multipart.enabled=true
multipart.file-size-threshold=0 # Threshold after which files will be written to disk.
multipart.location= # Intermediate location of uploaded files.
multipart.max-file-size=1Mb # Max file size.
multipart.max-request-size=10Mb # Max request size.

# SPRING HATEOAS (HateoasProperties)
spring.hateoas.apply-to-primary-object-mapper=true # if the primary mapper should also
 be configured

# HTTP encoding (HttpEncodingProperties)
spring.http.encoding.charset=UTF-8 # the encoding of HTTP requests/responses
spring.http.encoding.enabled=true # enable http encoding support
spring.http.encoding.force=true # force the configured encoding

# HTTP message conversion
spring.http.converters.preferred-json-mapper= # the preferred JSON mapper to use for H
TTP message conversion. Set to "gson" to force the use of Gson when both it and Jackso
n are on the classpath.

# HTTP response compression (GzipFilterProperties)
spring.http.gzip.buffer-size= # size of the output buffer in bytes
spring.http.gzip.deflate-compression-level= # the level used for deflate compression (
0-9)
spring.http.gzip.deflate-no-wrap= # noWrap setting for deflate compression (true or fa
lse)
spring.http.gzip.enabled=true # enable gzip filter support
spring.http.gzip.excluded-agents= # comma-separated list of user agents to exclude fro
m compression
spring.http.gzip.excluded-agent-patterns= # comma-separated list of regular expression
 patterns to control user agents excluded from compression
spring.http.gzip.excluded-paths= # comma-separated list of paths to exclude from compr
ession
spring.http.gzip.excluded-path-patterns= # comma-separated list of regular expression
patterns to control the paths that are excluded from compression
spring.http.gzip.methods= # comma-separated list of HTTP methods for which compression
 is enabled
spring.http.gzip.mime-types= # comma-separated list of MIME types which should be comp
ressed
spring.http.gzip.min-gzip-size= # minimum content length required for compression to o
ccur
spring.http.gzip.vary= # Vary header to be sent on responses that may be compressed

# JACKSON (JacksonProperties)
spring.jackson.date-format= # Date format string (e.g. yyyy-MM-dd HH:mm:ss), or a full
y-qualified date format class name (e.g. com.fasterxml.jackson.databind.util.ISO8601Da
teFormat)
spring.jackson.property-naming-strategy= # One of the constants on Jackson's PropertyN
amingStrategy (e.g. CAMEL_CASE_TO_LOWER_CASE_WITH_UNDERSCORES) or the fully-qualified
class name of a PropertyNamingStrategy subclass
```

```
spring.jackson.deserialization.*= # see Jackson's DeserializationFeature
spring.jackson.generator.*= # see Jackson's JsonGenerator.Feature
spring.jackson.mapper.*= # see Jackson's MapperFeature
spring.jackson.parser.*= # see Jackson's JsonParser.Feature
spring.jackson.serialization.*= # see Jackson's SerializationFeature
spring.jackson.serialization-inclusion= # Controls the inclusion of properties during
serialization (see Jackson's JsonInclude.Include)

# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.check-template-location=true
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.excluded-view-names= # comma-separated list of view names that should
 be excluded from resolution
spring.thymeleaf.view-names= # comma-separated list of view names that can be resolved
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML5
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.content-type=text/html # ;charset=<encoding> is added
spring.thymeleaf.cache=true # set to false for hot refresh

# FREEMARKER (FreeMarkerAutoConfiguration)
spring.freemarker.allow-request-override=false
spring.freemarker.cache=true
spring.freemarker.check-template-location=true
spring.freemarker.charset=UTF-8
spring.freemarker.content-type=text/html
spring.freemarker.expose-request-attributes=false
spring.freemarker.expose-session-attributes=false
spring.freemarker.expose-spring-macro-helpers=false
spring.freemarker.prefix=
spring.freemarker.request-context-attribute=
spring.freemarker.settings.*=
spring.freemarker.suffix=.ftl
spring.freemarker.template-loader-path=classpath:/templates/ # comma-separated list
spring.freemarker.view-names= # whitelist of view names that can be resolved

# GROOVY TEMPLATES (GroovyTemplateAutoConfiguration)
spring.groovy.template.cache=true
spring.groovy.template.charset=UTF-8
spring.groovy.template.configuration.*= # See Groovy's TemplateConfiguration
spring.groovy.template.content-type=text/html
spring.groovy.template.prefix=classpath:/templates/
spring.groovy.template.suffix=.tpl
spring.groovy.template.view-names= # whitelist of view names that can be resolved

# VELOCITY TEMPLATES (VelocityAutoConfiguration)
spring.velocity.allow-request-override=false
spring.velocity.cache=true
spring.velocity.check-template-location=true
spring.velocity.charset=UTF-8
spring.velocity.content-type=text/html
spring.velocity.date-tool-attribute=
spring.velocity.expose-request-attributes=false
```

```
spring.velocity.expose-session-attributes=false
spring.velocity.expose-spring-macro-helpers=false
spring.velocity.number-tool-attribute=
spring.velocity.prefer-file-system-access=true # prefer file system access for templat
e loading
spring.velocity.prefix=
spring.velocity.properties.*=
spring.velocity.request-context-attribute=
spring.velocity.resource-loader-path=classpath:/templates/
spring.velocity.suffix=.vm
spring.velocity.toolbox-config-location= # velocity Toolbox config location, for examp
le "/WEB-INF/toolbox.xml"
spring.velocity.view-names= # whitelist of view names that can be resolved

# JERSEY (JerseyProperties)
spring.jersey.type=servlet # servlet or filter
spring.jersey.init= # init params
spring.jersey.filter.order=

# INTERNATIONALIZATION (MessageSourceAutoConfiguration)
spring.messages.basename=messages
spring.messages.cache-seconds=-1
spring.messages.encoding=UTF-8


# SECURITY (SecurityProperties)
security.user.name=user # login username
security.user.password= # login password
security.user.role=USER # role assigned to the user
security.require-ssl=false # advanced settings ...
security.enable-csrf=false
security.basic.enabled=true
security.basic.realm=Spring
security.basic.path= # /**
security.basic.authorize-mode= # ROLE, AUTHENTICATED, NONE
security.filter-order=0
security.headers.xss=false
security.headers.cache=false
security.headers.frame=false
security.headers.content-type=false
security.headers.hsts=all # none / domain / all
security.sessions=stateless # always / never / if_required / stateless
security.ignored= # Comma-separated list of paths to exclude from the default secured
paths

# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.name= # name of the data source
spring.datasource.initialize=true # populate using data.sql
spring.datasource.schema= # a schema (DDL) script resource reference
spring.datasource.data= # a data (DML) script resource reference
spring.datasource.sql-script-encoding= # a charset for reading SQL scripts
spring.datasource.platform= # the platform to use in the schema resource (schema-${pla
tform}.sql)
```

```
spring.datasource.continue-on-error=false # continue even if can't be initialized
spring.datasource.separator=; # statement separator in SQL initialization scripts
spring.datasource.driver-class-name= # JDBC Settings...
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
spring.datasource.jndi-name= # For JNDI lookup (class, url, username & password are ig
nored when set)
spring.datasource.max-active=100 # Advanced configuration...
spring.datasource.max-idle=8
spring.datasource.min-idle=8
spring.datasource.initial-size=10
spring.datasource.validation-query=
spring.datasource.test-on-borrow=false
spring.datasource.test-on-return=false
spring.datasource.test-while-idle=
spring.datasource.time-between-eviction-runs-millis=
spring.datasource.min-evictable-idle-time-millis=
spring.datasource.max-wait=
spring.datasource.jmx-enabled=false # Export JMX MBeans (if supported)

# DAO (PersistenceExceptionTranslationAutoConfiguration)
spring.dao.exceptiontranslation.enabled=true

# MONGODB (MongoProperties)
spring.data.mongodb.host= # the db host
spring.data.mongodb.port=27017 # the connection port (defaults to 27107)
spring.data.mongodb.uri=mongodb://localhost/test # connection URL
spring.data.mongodb.database=
spring.data.mongodb.authentication-database=
spring.data.mongodb.grid-fs-database=
spring.data.mongodb.username=
spring.data.mongodb.password=
spring.data.mongodb.repositories.enabled=true # if spring data repository support is e
nabled

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.jpa.properties.*= # properties to set on the JPA connection
spring.jpa.open-in-view=true
spring.jpa.show-sql=true
spring.jpa.database-platform=
spring.jpa.database=
spring.jpa.generate-ddl=false # ignored by Hibernate, might be useful for other vendor
s
spring.jpa.hibernate.naming-strategy= # naming classname
spring.jpa.hibernate.ddl-auto= # defaults to create-drop for embedded dbs
spring.data.jpa.repositories.enabled=true # if spring data repository support is enabl
ed

# JTA (JtaAutoConfiguration)
spring.jta.log-dir= # transaction log dir
spring.jta.*= # technology specific configuration
```

```
# ATOMIKOS
spring.jta.atomikos.connectionfactory.borrow-connection-timeout=30 # Timeout, in secon
ds, for borrowing connections from the pool
spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag=true # Whether or
 not to ignore the transacted flag when creating session
spring.jta.atomikos.connectionfactory.local-transaction-mode=false # Whether or not lo
cal transactions are desired
spring.jta.atomikos.connectionfactory.maintenance-interval=60 # The time, in seconds,
between runs of the pool's maintenance thread
spring.jta.atomikos.connectionfactory.max-idle-time=60 # The time, in seconds, after w
hich connections are cleaned up from the pool
spring.jta.atomikos.connectionfactory.max-lifetime=0 # The time, in seconds, that a co
nnection can be pooled for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.max-pool-size=1 # The maximum size of the pool
spring.jta.atomikos.connectionfactory.min-pool-size=1 # The minimum size of the pool
spring.jta.atomikos.connectionfactory.reap-timeout=0 # The reap timeout, in seconds, f
or borrowed connections. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.unique-resource-name=jmsConnectionFactory # The
unique name used to identify the resource during recovery
spring.jta.atomikos.datasource.borrow-connection-timeout=30 # Timeout, in seconds, for
 borrowing connections from the pool
spring.jta.atomikos.datasource.default-isolation-level= # Default isolation level of c
onnections provided by the pool
spring.jta.atomikos.datasource.login-timeout= # Timeout, in seconds, for establishing
a database connection
spring.jta.atomikos.datasource.maintenance-interval=60 # The time, in seconds, between
 runs of the pool's maintenance thread
spring.jta.atomikos.datasource.max-idle-time=60 # The time, in seconds, after which co
nnections are cleaned up from the pool
spring.jta.atomikos.datasource.max-lifetime=0 # The time, in seconds, that a connectio
n can be pooled for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.datasource.max-pool-size=1 # The maximum size of the pool
spring.jta.atomikos.datasource.min-pool-size=1 # The minimum size of the pool
spring.jta.atomikos.datasource.reap-timeout=0 # The reap timeout, in seconds, for borr
owed connections. 0 denotes no limit.
spring.jta.atomikos.datasource.test-query= # SQL query or statement used to validate a
 connection before returning it
spring.jta.atomikos.datasource.unique-resource-name=dataSource # The unique name used
to identify the resource during recovery

# BITRONIX
spring.jta.bitronix.connectionfactory.acquire-increment=1 # Number of connections to c
reate when growing the pool
spring.jta.bitronix.connectionfactory.acquisition-interval=1 # Time, in seconds, to wa
it before trying to acquire a connection again after an invalid connection was acquire
d
spring.jta.bitronix.connectionfactory.acquisition-timeout=30 # Timeout, in seconds, fo
r acquiring connections from the pool
spring.jta.bitronix.connectionfactory.allow-local-transactions=true # Whether or not t
he transaction manager should allow mixing XA and non-XA transactions
spring.jta.bitronix.connectionfactory.apply-transaction-timeout=false # Whether or not
 the transaction timeout should be set on the XAResource when it is enlisted
spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled=true # Whether or no
```

```
t resources should be enlisted and delisted automatically
spring.jta.bitronix.connectionfactory.cache-producers-consumers=true # Whether or not
produces and consumers should be cached
spring.jta.bitronix.connectionfactory.defer-connection-release=true # Whether or not t
he provider can run many transactions on the same connection and supports transaction
interleaving
spring.jta.bitronix.connectionfactory.ignore-recovery-failures=false # Whether or not
recovery failures should be ignored
spring.jta.bitronix.connectionfactory.max-idle-time=60 # The time, in seconds, after w
hich connections are cleaned up from the pool
spring.jta.bitronix.connectionfactory.max-pool-size=10 # The maximum size of the pool.
 0 denotes no limit
spring.jta.bitronix.connectionfactory.min-pool-size=0 # The minimum size of the pool
spring.jta.bitronix.connectionfactory.password= # The password to use to connect to th
e JMS provider
spring.jta.bitronix.connectionfactory.share-transaction-connections=false #  Whether o
r not connections in the ACCESSIBLE state can be shared within the context of a transa
ction
spring.jta.bitronix.connectionfactory.test-connections=true # Whether or not connectio
ns should be tested when acquired from the pool
spring.jta.bitronix.connectionfactory.two-pc-ordering-position=1 # The postion that th
is resource should take during two-phase commit (always first is Integer.MIN_VALUE, al
ways last is Integer.MAX_VALUE)
spring.jta.bitronix.connectionfactory.unique-name=jmsConnectionFactory # The unique na
me used to identify the resource during recovery
spring.jta.bitronix.connectionfactory.use-tm-join=true Whether or not TMJOIN should be
 used when starting XAResources
spring.jta.bitronix.connectionfactory.user= # The user to use to connect to the JMS pr
ovider
spring.jta.bitronix.datasource.acquire-increment=1 # Number of connections to create w
hen growing the pool
spring.jta.bitronix.datasource.acquisition-interval=1 # Time, in seconds, to wait befo
re trying to acquire a connection again after an invalid connection was acquired
spring.jta.bitronix.datasource.acquisition-timeout=30 # Timeout, in seconds, for acqui
ring connections from the pool
spring.jta.bitronix.datasource.allow-local-transactions=true # Whether or not the tran
saction manager should allow mixing XA and non-XA transactions
spring.jta.bitronix.datasource.apply-transaction-timeout=false # Whether or not the tr
ansaction timeout should be set on the XAResource when it is enlisted
spring.jta.bitronix.datasource.automatic-enlisting-enabled=true # Whether or not resou
rces should be enlisted and delisted automatically
spring.jta.bitronix.datasource.cursor-holdability= # The default cursor holdability fo
r connections
spring.jta.bitronix.datasource.defer-connection-release=true # Whether or not the data
base can run many transactions on the same connection and supports transaction interle
aving
spring.jta.bitronix.datasource.enable-jdbc4-connection-test # Whether or not Connectio
n.isValid() is called when acquiring a connection from the pool
spring.jta.bitronix.datasource.ignore-recovery-failures=false # Whether or not recover
y failures should be ignored
spring.jta.bitronix.datasource.isolation-level= # The default isolation level for conn
ections
spring.jta.bitronix.datasource.local-auto-commit # The default auto-commit mode for lo
```

```
cal transactions
spring.jta.bitronix.datasource.login-timeout= # Timeout, in seconds, for establishing
a database connection
spring.jta.bitronix.datasource.max-idle-time=60 # The time, in seconds, after which co
nnections are cleaned up from the pool
spring.jta.bitronix.datasource.max-pool-size=10 # The maximum size of the pool. 0 deno
tes no limit
spring.jta.bitronix.datasource.min-pool-size=0 # The minimum size of the pool
spring.jta.bitronix.datasource.prepared-statement-cache-size=0 # The target size of th
e prepared statement cache. 0 disables the cache
spring.jta.bitronix.datasource.share-transaction-connections=false #  Whether or not c
onnections in the ACCESSIBLE state can be shared within the context of a transaction
spring.jta.bitronix.datasource.test-query # SQL query or statement used to validate a
connection before returning it
spring.jta.bitronix.datasource.two-pc-ordering-position=1 # The position that this res
ource should take during two-phase commit (always first is Integer.MIN_VALUE, always l
ast is Integer.MAX_VALUE)
spring.jta.bitronix.datasource.unique-name=dataSource # The unique name used to identi
fy the resource during recovery
spring.jta.bitronix.datasource.use-tm-join=true Whether or not TMJOIN should be used w
hen starting XAResources

# SOLR (SolrProperties)
spring.data.solr.host=http://127.0.0.1:8983/solr
spring.data.solr.zk-host=
spring.data.solr.repositories.enabled=true # if spring data repository support is enab
led

# ELASTICSEARCH (ElasticsearchProperties)
spring.data.elasticsearch.cluster-name= # The cluster name (defaults to elasticsearch)
spring.data.elasticsearch.cluster-nodes= # The address(es) of the server node (comma-s
eparated; if not specified starts a client node)
spring.data.elasticsearch.properties.*= # Additional properties used to configure the
client
spring.data.elasticsearch.repositories.enabled=true # if spring data repository suppor
t is enabled

# DATA REST (RepositoryRestConfiguration)
spring.data.rest.base-uri= # base URI against which the exporter should calculate its
links

# FLYWAY (FlywayProperties)
flyway.*= # Any public property available on the auto-configured `Flyway` object
flyway.check-location=false # check that migration scripts location exists
flyway.locations=classpath:db/migration # locations of migrations scripts
flyway.schemas= # schemas to update
flyway.init-version= 1 # version to start migration
flyway.init-sqls= # SQL statements to execute to initialize a connection immediately a
fter obtaining it
flyway.sql-migration-prefix=V
flyway.sql-migration-suffix=.sql
flyway.enabled=true
flyway.url= # JDBC url if you want Flyway to create its own DataSource
```

```
flyway.user= # JDBC username if you want Flyway to create its own DataSource
flyway.password= # JDBC password if you want Flyway to create its own DataSource

# LIQUIBASE (LiquibaseProperties)
liquibase.change-log=classpath:/db/changelog/db.changelog-master.yaml
liquibase.check-change-log-location=true # check the change log location exists
liquibase.contexts= # runtime contexts to use
liquibase.default-schema= # default database schema to use
liquibase.drop-first=false
liquibase.enabled=true
liquibase.url= # specific JDBC url (if not set the default datasource is used)
liquibase.user= # user name for liquibase.url
liquibase.password= # password for liquibase.url

# JMX
spring.jmx.enabled=true # Expose MBeans from Spring

# RABBIT (RabbitProperties)
spring.rabbitmq.host= # connection host
spring.rabbitmq.port= # connection port
spring.rabbitmq.addresses= # connection addresses (e.g. myhost:9999,otherhost:1111)
spring.rabbitmq.username= # login user
spring.rabbitmq.password= # login password
spring.rabbitmq.virtual-host=
spring.rabbitmq.dynamic=

# REDIS (RedisProperties)
spring.redis.database= # database name
spring.redis.host=localhost # server host
spring.redis.password= # server password
spring.redis.port=6379 # connection port
spring.redis.pool.max-idle=8 # pool settings ...
spring.redis.pool.min-idle=0
spring.redis.pool.max-active=8
spring.redis.pool.max-wait=-1
spring.redis.sentinel.master= # name of Redis server
spring.redis.sentinel.nodes= # comma-separated list of host:port pairs

# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url=tcp://localhost:61616 # connection URL
spring.activemq.user=
spring.activemq.password=
spring.activemq.in-memory=true # broker kind to create if no broker-url is specified
spring.activemq.pooled=false

# HornetQ (HornetQProperties)
spring.hornetq.mode= # connection mode (native, embedded)
spring.hornetq.host=localhost # hornetQ host (native mode)
spring.hornetq.port=5445 # hornetQ port (native mode)
spring.hornetq.embedded.enabled=true # if the embedded server is enabled (needs hornet
q-jms-server.jar)
spring.hornetq.embedded.server-id= # auto-generated id of the embedded server (integer
)
```

```
spring.hornetq.embedded.persistent=false # message persistence
spring.hornetq.embedded.data-directory= # location of data content (when persistence i
s enabled)
spring.hornetq.embedded.queues= # comma-separated queues to create on startup
spring.hornetq.embedded.topics= # comma-separated topics to create on startup
spring.hornetq.embedded.cluster-password= # customer password (randomly generated by d
efault)

# JMS (JmsProperties)
spring.jms.jndi-name= # JNDI location of a JMS ConnectionFactory
spring.jms.pub-sub-domain= # false for queue (default), true for topic

# Email (MailProperties)
spring.mail.host=smtp.acme.org # mail server host
spring.mail.port= # mail server port
spring.mail.username=
spring.mail.password=
spring.mail.default-encoding=UTF-8 # encoding to use for MimeMessages
spring.mail.properties.*= # properties to set on the JavaMail session

# SPRING BATCH (BatchDatabaseInitializer)
spring.batch.job.names=job1,job2
spring.batch.job.enabled=true
spring.batch.initializer.enabled=true
spring.batch.schema= # batch schema to load

# SPRING CACHE (CacheProperties)
spring.cache.type= # generic, ehcache, hazelcast, jcache, redis, guava, simple, none
spring.cache.config= #
spring.cache.cache-names= # cache names to create on startup
spring.cache.jcache.provider= # fully qualified name of the CachingProvider implementa
tion to use
spring.cache.guava.spec= # guava specs

# AOP
spring.aop.auto=
spring.aop.proxy-target-class=

# FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding= # Expected character encoding the application must use

# SPRING SOCIAL (SocialWebAutoConfiguration)
spring.social.auto-connection-views=true # Set to true for default connection views or
 false if you provide your own

# SPRING SOCIAL FACEBOOK (FacebookAutoConfiguration)
spring.social.facebook.app-id= # your application's Facebook App ID
spring.social.facebook.app-secret= # your application's Facebook App Secret

# SPRING SOCIAL LINKEDIN (LinkedInAutoConfiguration)
spring.social.linkedin.app-id= # your application's LinkedIn App ID
spring.social.linkedin.app-secret= # your application's LinkedIn App Secret
```

```
# SPRING SOCIAL TWITTER (TwitterAutoConfiguration)
spring.social.twitter.app-id= # your application's Twitter App ID
spring.social.twitter.app-secret= # your application's Twitter App Secret

# SPRING MOBILE SITE PREFERENCE (SitePreferenceAutoConfiguration)
spring.mobile.sitepreference.enabled=true # enabled by default

# SPRING MOBILE DEVICE VIEWS (DeviceDelegatingViewResolverAutoConfiguration)
spring.mobile.devicedelegatingviewresolver.enabled=true # disabled by default
spring.mobile.devicedelegatingviewresolver.normal-prefix=
spring.mobile.devicedelegatingviewresolver.normal-suffix=
spring.mobile.devicedelegatingviewresolver.mobile-prefix=mobile/
spring.mobile.devicedelegatingviewresolver.mobile-suffix=
spring.mobile.devicedelegatingviewresolver.tablet-prefix=tablet/
spring.mobile.devicedelegatingviewresolver.tablet-suffix=

# ----------------------------------------
# ACTUATOR PROPERTIES
# ----------------------------------------

# MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.port= # defaults to 'server.port'
management.address= # bind to a specific NIC
management.context-path= # default to '/'
management.add-application-context-header= # default to true
management.security.enabled=true # enable security
management.security.role=ADMIN # role required to access the management endpoint
management.security.sessions=stateless # session creating policy to use (always, never
, if_required, stateless)

# PID FILE (ApplicationPidFileWriter)
spring.pidfile= # Location of the PID file to write

# ENDPOINTS (AbstractEndpoint subclasses)
endpoints.autoconfig.id=autoconfig
endpoints.autoconfig.sensitive=true
endpoints.autoconfig.enabled=true
endpoints.beans.id=beans
endpoints.beans.sensitive=true
endpoints.beans.enabled=true
endpoints.configprops.id=configprops
endpoints.configprops.sensitive=true
endpoints.configprops.enabled=true
endpoints.configprops.keys-to-sanitize=password,secret,key # suffix or regex
endpoints.dump.id=dump
endpoints.dump.sensitive=true
endpoints.dump.enabled=true
endpoints.env.id=env
endpoints.env.sensitive=true
endpoints.env.enabled=true
endpoints.env.keys-to-sanitize=password,secret,key # suffix or regex
endpoints.health.id=health
endpoints.health.sensitive=true
```

```
endpoints.health.enabled=true
endpoints.health.mapping.*= # mapping of health statuses to HttpStatus codes
endpoints.health.time-to-live=1000
endpoints.info.id=info
endpoints.info.sensitive=false
endpoints.info.enabled=true
endpoints.mappings.enabled=true
endpoints.mappings.id=mappings
endpoints.mappings.sensitive=true
endpoints.metrics.id=metrics
endpoints.metrics.sensitive=true
endpoints.metrics.enabled=true
endpoints.shutdown.id=shutdown
endpoints.shutdown.sensitive=true
endpoints.shutdown.enabled=false
endpoints.trace.id=trace
endpoints.trace.sensitive=true
endpoints.trace.enabled=true

# HEALTH INDICATORS (previously health.*)
management.health.db.enabled=true
management.health.elasticsearch.enabled=true
management.health.elasticsearch.response-timeout=100 # the time, in milliseconds, to w
ait for a response from the cluster
management.health.diskspace.enabled=true
management.health.diskspace.path=.
management.health.diskspace.threshold=10485760
management.health.mongo.enabled=true
management.health.rabbit.enabled=true
management.health.redis.enabled=true
management.health.solr.enabled=true
management.health.status.order=DOWN, OUT_OF_SERVICE, UNKNOWN, UP

# MVC ONLY ENDPOINTS
endpoints.jolokia.path=jolokia
endpoints.jolokia.sensitive=true
endpoints.jolokia.enabled=true # when using Jolokia

# JMX ENDPOINT (EndpointMBeanExportProperties)
endpoints.jmx.enabled=true
endpoints.jmx.domain= # the JMX domain, defaults to 'org.springboot'
endpoints.jmx.unique-names=false
endpoints.jmx.static-names=

# JOLOKIA (JolokiaProperties)
jolokia.config.*= # See Jolokia manual

# REMOTE SHELL
shell.auth=simple # jaas, key, simple, spring
shell.command-refresh-interval=-1
shell.command-path-patterns= # classpath*:/commands/**, classpath*:/crash/commands/**
shell.config-path-patterns= # classpath*:/crash/*
shell.disabled-commands=jpa*,jdbc*,jndi* # comma-separated list of commands to disable
```

```
shell.disabled-plugins=false # don't expose plugins
shell.ssh.enabled= # ssh settings ...
shell.ssh.key-path=
shell.ssh.port=
shell.telnet.enabled= # telnet settings ...
shell.telnet.port=
shell.auth.jaas.domain= # authentication settings ...
shell.auth.key.path=
shell.auth.simple.user.name=
shell.auth.simple.user.password=
shell.auth.spring.roles=

# SENDGRID (SendGridAutoConfiguration)
spring.sendgrid.username= # SendGrid account username
spring.sendgrid.password= # SendGrid account password
spring.sendgrid.proxy.host= # SendGrid proxy host
spring.sendgrid.proxy.port= # SendGrid proxy port

# GIT INFO
spring.git.properties= # resource ref to generated git info properties file
```

附录B. 配置元数据

Spring Boot jars包含元数据文件，它们提供了所有支持的配置属性详情。这些文件设计用于让IDE开发者能够为使用application.properties或application.yml文件的用户提供上下文帮助及代码完成功能。

主要的元数据文件是在编译器通过处理所有被 `@ConfigurationProperties` 注解的节点来自动生成的。

附录B.1. 元数据格式

配置元数据位于jars文件中的 `META-INF/spring-configuration-metadata.json` ，它们使用一个具有"groups"或"properties"分类节点的简单JSON格式：

```
{"groups": [
    {
        "name": "server",
        "type": "org.springframework.boot.autoconfigure.web.ServerProperties",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    }
    ...
],"properties": [
    {
        "name": "server.port",
        "type": "java.lang.Integer",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
        "name": "server.servlet-path",
        "type": "java.lang.String",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
        "defaultValue": "/"
    }
    ...
]}
```

每个"property"是一个配置节点，用户可以使用特定的值指定它。例如， `server.port` 和 `server.servlet-path` 可能在 `application.properties` 中如以下定义：

```
server.port=9090
server.servlet-path=/home
```

"groups"是高级别的节点，它们本身不指定一个值，但为properties提供一个有上下文关联的分组。例如， `server.port` 和 `server.servlet-path` 属性是 `server` 组的一部分。

注：不需要每个"property"都有一个"group"，一些属性可以以自己的形式存在。

附录B.1.1. Group属性

`groups` 数组包含的JSON对象可以由以下属性组成：

名称	类型	目的
name	String	group的全名，该属性是强制性的
type	String	group数据类型的类名。例如，如果group是基于一个被 `@ConfigurationProperties` 注解的类，该属性将包含该类的全限定名。如果基于一个 `@Bean` 方法，它将是该方法的返回类型。如果该类型未知，则该属性将被忽略
description	String	一个简短的group描述，用于展示给用户。如果没有可用描述，该属性将被忽略。推荐使用一个简短的段落描述，第一行提供一个简洁的总结，最后一行以句号结尾
sourceType	String	贡献该组的来源类名。例如，如果组基于一个被 `@ConfigurationProperties` 注解的 `@Bean` 方法，该属性将包含 `@Configuration` 类的全限定名，该类包含此方法。如果来源类型未知，则该属性将被忽略
sourceMethod	String	贡献该组的方法的全名（包含括号及参数类型）。例如，被 `@ConfigurationProperties` 注解的 `@Bean` 方法名。如果源方法未知，该属性将被忽略

附录B.1.2. Property属性

`properties` 数组中包含的JSON对象可由以下属性构成：

名称	类型	目的
name	String	property的全名，格式为小写虚线分割的形式（比如 `server.servlet-path` ）。该属性是强制性的
type	String	property数据类型的类名。例如 `java.lang.String` 。该属性可以用来指导用户他们可以输入值的类型。为了保持一致，原生类型使用它们的包装类代替，比如 `boolean` 变成了 `java.lang.Boolean` 。注意，这个类可能是个从一个字符串转换而来的复杂类型。如果类型未知则该属性会被忽略
description	String	一个简短的组的描述，用于展示给用户。如果没有描述可用则该属性会被忽略。推荐使用一个简短的段落描述，开头提供一个简洁的总结，最后一行以句号结束
sourceType	String	贡献property的来源类名。例如，如果property来自一个被 `@ConfigurationProperties` 注解的类，该属性将包括该类的全限定名。如果来源类型未知则该属性会被忽略
defaultValue	Object	当property没有定义时使用的默认值。如果property类型是个数组则该属性也可以是个数组。如果默认值未知则该属性会被忽略
deprecated	boolean	指定该property是否过期。如果该字段没有过期或该信息未知则该属性会被忽略

附录B.1.3. 可重复的元数据节点

在同一个元数据文件中出现多次相同名称的"property"和"group"对象是可以接受的。例如，Spring Boot将 `spring.datasource` 属性绑定到Hikari，Tomcat和DBCP类，并且每个都潜在的提供了重复的属性名。这些元数据的消费者需要确保他们支持这样的场景。

附录B.2. 使用注解处理器产生自己的元数据

通过使用 `spring-boot-configuration-processor` jar，你可以从被 `@ConfigurationProperties` 注解的节点轻松的产生自己的配置元数据文件。该jar包含一个在你的项目编译时会被调用的Java注解处理器。想要使用该处理器，你只需简单添加 `spring-boot-configuration-processor` 依赖，例如使用Maven你需要添加：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

使用Gradle时，你可以使用propdeps-plugin并指定：

```
dependencies {
        optional "org.springframework.boot:spring-boot-configuration-processor"
    }

    compileJava.dependsOn(processResources)
}
```

注：你需要将 `compileJava.dependsOn(processResources)` 添加到构建中，以确保资源在代码编译之前处理。如果没有该指令，任何 `additional-spring-configuration-metadata.json` 文件都不会被处理。

该处理器会处理被 `@ConfigurationProperties` 注解的类和方法，description属性用于产生配置类字段值的Javadoc说明。

注：你应该使用简单的文本来设置 `@ConfigurationProperties` 字段的Javadoc，因为在没有被添加到JSON之前它们是不被处理的。

属性是通过判断是否存在标准的getters和setters来发现的，对于集合类型有特殊处理（即使只出现一个getter）。该注解处理器也支持使用lombok的 `@Data`，`@Getter` 和 `@Setter` 注解。

附录 B.2.1. 内嵌属性

该注解处理器自动将内部类当做内嵌属性处理。例如，下面的类：

```
@ConfigurationProperties(prefix="server")
public class ServerProperties {

    private String name;

    private Host host;

    // ... getter and setters

    private static class Host {

        private String ip;

        private int port;

        // ... getter and setters

    }

}
```

附录 B.2.2. 添加其他的元数据

附录C. 自动配置类

这里有一个Spring Boot提供的所有自动配置类的文档链接和源码列表。也要记着看一下你的应用都开启了哪些自动配置（使用 `--debug` 或 `-Debug` 启动应用，或在一个Actuator应用中使用 `autoconfig` 端点）。

附录 C.1 来自 `spring-boot-autoconfigure` 模块

下面的自动配置类来自 `spring-boot-autoconfigure` 模块：

配置类	链接
ActiveMQAutoConfiguration	javadoc
AopAutoConfiguration	javadoc
BatchAutoConfiguration	javadoc
CacheAutoConfiguration	javadoc
CloudAutoConfiguration	javadoc
DataSourceAutoConfiguration	javadoc
DataSourceTransactionManagerAutoConfiguration	javadoc
DeviceDelegatingViewResolverAutoConfiguration	javadoc
DeviceResolverAutoConfiguration	javadoc
DispatcherServletAutoConfiguration	javadoc

附录C.2 来自 `spring-boot-actuator` 模块

下列的自动配置类来自于 `spring-boot-actuator` 模块：

附录D. 可执行jar格式

`spring-boot-loader` 模块允许Spring Boot对可执行jar和war文件的支持。如果你正在使用Maven或Gradle插件，可执行jar会被自动产生，通常你不需要了解它是如何工作的。

如果你需要从一个不同的构建系统创建可执行jars，或你只是对底层技术好奇，本章节将提供一些背景资料。

附录D.1. 内嵌JARs

Java没有提供任何标准的方式来加载内嵌的jar文件（也就是jar文件自身包含到一个jar中）。如果你正分发一个在不解压缩的情况下可以从命令行运行的自包含应用，那这将是个问题。

为了解决这个问题，很多开发者使用"影子" jars。一个影子jar只是简单的将所有jars的类打包进一个单独的"超级jar"。使用影子jars的问题是它很难分辨在你的应用中实际可以使用的库。在多个jars中存在相同的文件名（内容不同）也是一个问题。Spring Boot另劈蹊径，让你能够直接嵌套jars。

附录D.1.1 可执行jar文件结构

Spring Boot Loader兼容的jar文件应该遵循以下结构：

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-com
 |  +-mycompany
 |     + project
 |        +-YouClasses.class
 +-lib
    +-dependency1.jar
    +-dependency2.jar
```

依赖需要放到内部的lib目录下。

附录D.1.2. 可执行war文件结构

Spring Boot Loader兼容的war文件应该遵循以下结构：

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |     +-mycompany
    |        +-project
    |           +-YouClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
       +-servlet-api.jar
       +-dependency3.jar
```

依赖需要放到内嵌的 `WEB-INF/lib` 目录下。任何运行时需要但部署到普通web容器不需要的依赖应该放到 `WEB-INF/lib-provided` 目录下。

附录D.2. Spring Boot的"JarFile"类

Spring Boot用于支持加载内嵌jars的核心类是 `org.springframework.boot.loader.jar.JarFile` 。它允许你从一个标准的jar文件或内嵌的子jar数据中加载jar内容。当首次加载的时候，每个JarEntry的位置被映射到一个偏移于外部jar的物理文件：

```
myapp.jar
+---------+---------------------+
|         | /lib/mylib.jar      |
| A.class |+---------+---------+|
|         || B.class | B.class ||
|         |+---------+---------+|
+---------+---------------------+
^          ^          ^
0063       3452       3980
```

上面的示例展示了如何在myapp.jar的0063处找到A.class。来自于内嵌jar的B.class实际可以在myapp.jar的3452处找到，B.class可以在3980处找到（图有问题？）。

有了这些信息，我们就可以通过简单的寻找外部jar的合适部分来加载指定的内嵌实体。我们不需要解压存档，也不需要将所有实体读取到内存中。

附录D.2.1 对标准Java "JarFile"的兼容性

Spring Boot Loader努力保持对已有代码和库的兼容。`org.springframework.boot.loader.jar.JarFile`继承自`java.util.jar.JarFile`，可以作为降级替换。

附录D.3. 启动可执行jars

`org.springframework.boot.loader.Launcher` 类是个特殊的启动类，用于一个可执行jars的主要入口。它实际上就是你jar文件的 `Main-Class` ，并用来设置一个合适的 `URLClassLoader` ，最后调用你的 `main()` 方法。

这里有3个启动器子类，JarLauncher，WarLauncher和PropertiesLauncher。它们的作用是从嵌套的jar或war文件目录中（相对于显示的从classpath）加载资源（.class文件等）。在 `[Jar|War]Launcher` 情况下，嵌套路径是固定的（ `lib/*.jar` 和war的 `lib-provided/*.jar` ），所以如果你需要很多其他jars只需添加到那些位置即可。PropertiesLauncher默认查找你应用存档的 `lib/` 目录，但你可以通过设置环境变量 `LOADER_PATH` 或application.properties中的 `loader.path` 来添加其他的位置（逗号分割的目录或存档列表）。

附录D.3.1 Launcher manifest

你需要指定一个合适的Launcher作为 `META-INF/MANIFEST.MF` 的 `Main-Class` 属性。你实际想要启动的类（也就是你编写的包含main方法的类）需要在 `Start-Class` 属性中定义。

例如，这里有个典型的可执行jar文件的MANIFEST.MF：

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

对于一个war文件，它可能是这样的：

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

注：你不需要在manifest文件中指定 `Class-Path` 实体，classpath会从嵌套的jars中被推导出来。

附录D.3.2. 暴露的存档

一些PaaS实现可能选择在运行前先解压存档。例如，Cloud Foundry就是这样操作的。你可以运行一个解压的存档，只需简单的启动合适的启动器：

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLaunche
```

附录D.4. PropertiesLauncher特性

PropertiesLauncher有一些特殊的性质，它们可以通过外部属性来启用（系统属性，环境变量，manifest实体或application.properties）。

Key	作用
loader.path	逗号分割的classpath，比如 `lib:${HOME}/app/lib`
loader.home	其他属性文件的位置，比如/opt/app（默认为 `${user.dir}` ）
loader.args	main方法的默认参数（以空格分割）
loader.main	要启动的main类名称，比如 `com.app.Application`
loader.config.name	属性文件名，比如loader（默认为application）
loader.config.location	属性文件路径，比如 `classpath:loader.properties` （默认为application.properties）
loader.system	布尔标识，表明所有的属性都应该添加到系统属性中（默认为false）

Manifest实体keys通过大写单词首字母及将分隔符从"."改为"-"（比如 `Loader-Path` ）来进行格式化。 `loader.main` 是个特例，它是通过查找manifest的 `Start-Class` ，这样也兼容JarLauncher。

环境变量可以大写字母并且用下划线代替句号。

- `loader.home` 是其他属性文件（覆盖默认）的目录位置，只要没有指定 `loader.config.location` 。
- `loader.path` 可以包含目录（递归地扫描jar和zip文件），存档路径或通配符模式（针对默认的JVM行为）。
- 占位符在使用前会被系统和环境变量加上属性文件本身的值替换掉。

附录D.5. 可执行jar的限制

当使用Spring Boot Loader打包的应用时有一些你需要考虑的限制。

附录D.5.1 Zip实体压缩

对于一个嵌套jar的ZipEntry必须使用 `ZipEntry.STORED` 方法保存。这是需要的，这样我们可以直接查找嵌套jar中的个别内容。嵌套jar的内容本身可以仍旧被压缩，正如外部jar的其他任何实体。

附录D.5.2. 系统ClassLoader

启动的应用在加载类时应该使用 `Thread.getContextClassLoader()`（多数库和框架都默认这样做）。尝试通过 `ClassLoader.getSystemClassLoader()` 加载嵌套的类将失败。请注意 `java.util.Logging` 总是使用系统类加载器，由于这个原因你需要考虑一个不同的日志实现。

附录D.6. 可替代的单一jar解决方案

如果以上限制造成你不能使用Spring Boot Loader，那可以考虑以下的替代方案：

- Maven Shade Plugin
- JarClassLoader
- OneJar

附录E. 依赖版本

下面的表格提供了详尽的依赖版本信息，这些版本由Spring Boot自身的CLI，Maven的依赖管理（dependency management）和Gradle插件提供。当你声明一个对以下artifacts的依赖而没有声明版本时，将使用下面表格中列出的版本。