

Spring Boot参考指南

作者

菲利普·韦伯，戴夫 Syer，约什·长，斯特凡·尼科尔，罗布·绞车，安迪·威尔金森，马塞尔·Overdijk，基督教·杜普伊斯，塞巴斯蒂安·德勒兹，迈克尔·西蒙斯，韦德兰·Pavić，周杰伦·科比，Madhura·巴维

2.0.1.BUILD-快照

版权所有©2012-2018

本文档的副本可以为您自己使用并分发给其他人，前提是您不收取这些副本的任何费用，并进一步规定每份副本均包含此版权声明，无论是以印刷版还是电子版分发。

目录

I. Spring Boot文档

1.关于文档

2.获得帮助

3.第一步

4.使用Spring Boot

5.了解Spring Boot特性

6.转向生产

7.高级主题

II. 入门

8.介绍Spring Boot

9.系统要求

9.1。Servlet容器

10.安装Spring Boot

10.1。Java开发人员的安装说明

10.1.1。Maven安装

10.1.2。Gradle安装

10.2。安装Spring Boot CLI

10.2.1。手动安装

10.2.2。使用SDKMAN进行安装！

10.2.3。OSX Homebrew安装

10.2.4。MacPorts安装

10.2.5。命令行完成

10.2.6。快速启动Spring CLI示例

10.3。从较早版本的Spring Boot升级

11.开发你的第一个Spring Boot应用程序

11.1。创建POM

11.2。添加类路径依赖关系

11.3。编写代码

11.3.1。@RestController和@RequestMapping注解

11.3.2。@EnableAutoConfiguration注释

11.3.3. “主要”方法

11.4. 运行示例

11.5. 创建一个可执行的Jar

12. 下一步阅读什么

III. 使用Spring Boot

13. 建立系统

13.1. 依赖管理

13.2. Maven的

13.2.1. 继承初始父项

13.2.2. 使用没有父POM的Spring Boot

13.2.3. 使用Spring Boot Maven插件

13.3. 摆籃

13.4. 蚂蚁

13.5. 首发

14. 构建你的代码

14.1. 使用“默认”包

14.2. 查找主要应用程序类

15. 配置类

15.1. 导入其他配置类

15.2. 导入XML配置

16. 自动配置

16.1. 逐渐替换自动配置

16.2. 禁用特定的自动配置类

17. 春豆和依赖注入

18. 使用@SpringBootApplication注释

19. 运行你的应用程序

19.1. 从IDE运行

19.2. 作为打包应用程序运行

19.3. 使用Maven插件

19.4. 使用Gradle插件

19.5. 热交换

20. 开发人员工具

20.1. 属性默认值

20.2. 自动重启

20.2.1. 记录条件评估中的更改

20.2.2. 排除资源

20.2.3. 看额外的路径

20.2.4. 禁用重新启动

20.2.5. 使用触发文件

20.2.6. 自定义重启类加载器

20.2.7. 已知限制

20.3. LiveReload

20.4. 全局设置

20.5. 远程应用

20.5.1. 运行远程客户端应用程序

20.5.2. 远程更新

21. 包装您的生产申请

22. 下一步阅读什么

IV。Spring Boot功能

23. SpringApplication

- 23.1. 启动失败
- 23.2. 自定义横幅
- 23.3. 自定义SpringApplication
- 23.4. Fluent Builder API
- 23.5. 应用程序事件和监听器
- 23.6. Web环境
- 23.7. 访问应用程序参数
- 23.8. 使用ApplicationRunner或CommandLineRunner
- 23.9. 申请退出
- 23.10. 管理功能

24. 外部化配置

- 24.1. 配置随机值
- 24.2. 访问命令行属性
- 24.3. 应用程序属性文件
- 24.4. 配置文件特定的属性
- 24.5. 属性中的占位符
- 24.6. 使用YAML而不是属性
 - 24.6.1. 正在加载YAML
 - 24.6.2. 在Spring环境中将YAML作为属性公开
 - 24.6.3. 多配置文件YAML文件
 - 24.6.4. YAML的优点
 - 24.6.5. 合并YAML列表
- 24.7. 类型安全的配置属性
 - 24.7.1. 第三方配置
 - 24.7.2. 轻松的绑定
 - 24.7.3. 属性转换
 - 转换持续时间
 - 24.7.4. @ConfigurationProperties验证
 - 24.7.5. @ConfigurationProperties与@Value

25. 简介

- 25.1. 添加活动配置文件
- 25.2. 编程设置配置文件
- 25.3. 配置文件特定的配置文件

26. 记录

- 26.1. 日志格式
- 26.2. 控制台输出
 - 26.2.1. 彩色编码输出
- 26.3. 文件输出
- 26.4. 日志级别
- 26.5. 自定义日志配置
- 26.6. Logback扩展
 - 26.6.1. 配置文件特定的配置
 - 26.6.2. 环境属性

27. 开发Web应用程序

27.1. “Spring Web MVC框架”

- 27.1.1. Spring MVC自动配置
- 27.1.2. HttpMessageConverters
- 27.1.3. 自定义JSON序列化器和反序列化器
- 27.1.4. MessageCodesResolver的信息
- 27.1.5. 静态内容
- 27.1.6. 欢迎页面
- 27.1.7. 自定义Favicon
- 27.1.8. 路径匹配和内容协商
- 27.1.9. ConfigurableWebBindingInitializer
- 10年1月27日。模板引擎
- 11年1月27日。错误处理
 - 自定义错误页面
 - 在Spring MVC之外映射错误页面
- 12年1月27日。春天的HATEOAS
- 13年1月27日。CORS支持

27.2. “Spring WebFlux框架”

- 27.2.1. Spring WebFlux自动配置
- 27.2.2. 使用HttpMessageReaders和HttpMessageWriters的HTTP编解码器
- 27.2.3. 静态内容
- 27.2.4. 模板引擎
- 27.2.5. 错误处理
- 自定义错误页面
- 27.2.6. 网页过滤器

27.3. JAX-RS和泽西岛

27.4. 嵌入式Servlet容器支持

- 27.4.1. Servlet，过滤器和监听器
 - 将Spring Servlet，过滤器和监听器注册为Spring Bean
- 27.4.2. Servlet上下文初始化
 - 扫描Servlet，筛选器和侦听器
- 27.4.3. ServletWebServerApplicationContext
- 27.4.4. 定制嵌入式Servlet容器
 - 程序化定制
 - 直接自定义ConfigurableServletWebServerFactory
- 27.4.5. JSP限制

28.安全

28.1. MVC安全

28.2. WebFlux安全

28.3. 的OAuth2

- 28.3.1. 客户

28.4. 执行器安全

- 28.4.1. 跨站请求伪造保护

29.使用SQL数据库

29.1. 配置一个数据源

- 29.1.1. 嵌入数据库支持
- 29.1.2. 连接到生产数据库
- 29.1.3. 连接到JNDI数据源

29.2. 使用JdbcTemplate

29.3. JPA和“Spring Data”

- 29.3.1. 实体类
- 29.3.2. Spring Data JPA存储库
- 29.3.3. 创建和删除JPA数据库
- 29.3.4. 在View中打开EntityManager

29.4. 使用H2的Web控制台

- 29.4.1. 更改H2 Console的路径

29.5. 使用jOOQ

- 29.5.1. 代码生成
- 29.5.2. 使用DSLContext
- 29.5.3. jOOQ SQL方言
- 29.5.4. 定制jOOQ

30.与NoSQL Technologies合作

30.1. Redis的

- 30.1.1. 连接到Redis

30.2. MongoDB的

- 30.2.1. 连接到MongoDB数据库
- 30.2.2. MongoTemplate
- 30.2.3. Spring Data MongoDB存储库
- 30.2.4. 嵌入式Mongo

30.3. Neo4j的

- 30.3.1. 连接到Neo4j数据库
- 30.3.2. 使用嵌入式模式
- 30.3.3. Neo4jSession
- 30.3.4. Spring Data Neo4j存储库
- 30.3.5. 存储库示例

30.4. 的GemFire

30.5. Solr的

- 30.5.1. 连接到Solr
- 30.5.2. Spring Data Solr存储库

30.6. Elasticsearch

- 30.6.1. 使用Jest连接到Elasticsearch
- 30.6.2. 通过使用Spring数据连接到Elasticsearch
- 30.6.3. Spring Data Elasticsearch存储库

30.7. 卡桑德拉

- 30.7.1. 连接到Cassandra
- 30.7.2. Spring Data Cassandra存储库

30.8. Couchbase

- 30.8.1. 连接到Couchbase
- 30.8.2. Spring Data Couchbase存储库

30.9. LDAP

- 30.9.1. 连接到LDAP服务器
- 30.9.2. Spring数据LDAP存储库
- 30.9.3. 嵌入式内存LDAP服务器

30.10. InfluxDB

- 30.10.1. 连接到InfluxDB

31.缓存

31.1. 支持的缓存提供程序

- 31.1.1. 通用
- 31.1.2. JCache (JSR-107)
- 31.1.3. EhCache 2.x
- 31.1.4. Hazelcast
- 31.1.5. Infinispan的
- 31.1.6. Couchbase
- 31.1.7. Redis的
- 31.1.8. 咖啡因
- 31.1.9. 简单
- 10年1月31日。没有

信息

32.1. JMS

- 32.1.1. ActiveMQ支持
- 32.1.2. Artemis支持
- 32.1.3. 使用JNDI ConnectionFactory
- 32.1.4. 发送消息
- 32.1.5. 接收消息

32.2. AMQP

- 32.2.1. RabbitMQ支持
- 32.2.2. 发送消息
- 32.2.3. 接收消息

32.3. Apache Kafka支持

- 32.3.1. 发送消息
- 32.3.2. 接收消息
- 32.3.3. 额外的卡夫卡属性

33.用REST调用REST服务 RestTemplate

33.1. RestTemplate自定义

34.用REST调用REST服务 WebClient

34.1. WebClient自定义

35.验证

36.发送电子邮件

37.与JTA的分布式事务

- 37.1. 使用Atomikos事务管理器
- 37.2. 使用Bitronix事务管理器
- 37.3. 使用Narayana事务管理器
- 37.4. 使用Java EE托管事务管理器
- 37.5. 混合XA和非XA JMS连接
- 37.6. 支持替代嵌入式事务管理器

38. Hazelcast

39.石英调度器

40.春季融合

41.春季会议

42.通过JMX进行监视和管理

43.测试

- 43.1. 测试范围依赖关系
- 43.2. 测试Spring应用程序
- 43.3. 测试Spring Boot应用程序
 - 43.3.1. 检测Web应用程序类型
 - 43.3.2. 检测测试配置
 - 43.3.3. 不包括测试配置
 - 43.3.4. 使用运行的服务器进行测试
 - 43.3.5. 嘲笑和侦察豆
 - 43.3.6. 自动配置的测试
 - 43.3.7. 自动配置的JSON测试
 - 43.3.8. 自动配置的Spring MVC测试
 - 43.3.9. 自动配置的Spring WebFlux测试
 - 43.3.10. 自动配置的数据JPA测试
 - 43.3.11. 自动配置的JDBC测试
 - 43.3.12. 自动配置的jOOQ测试
 - 43.3.13. 自动配置的数据MongoDB测试
 - 43.3.14. 自动配置的数据Neo4j测试
 - 43.3.15. 自动配置的数据Redis测试
 - 43.3.16. 自动配置的数据LDAP测试
 - 43.3.17. 自动配置的REST客户端
 - 43.3.18. 自动配置的Spring REST Docs测试
 - 自动配置的Spring REST Docs使用Mock MVC进行测试
 - 自动配置的Spring REST Docs使用REST Assured进行测试
 - 43.3.19. 用户配置和切片
 - 43.3.20. 使用Spock测试Spring Boot应用程序

43.4. 测试实用程序

- 43.4.1. ConfigFileApplicationContextInitializer
- 43.4.2. EnvironmentTestUtils
- 43.4.3. OutputCapture
- 43.4.4. TestRestTemplate

44. WebSockets

45.网络服务

46.创建您自己的自动配置

- 46.1. 了解自动配置的Bean
- 46.2. 查找自动配置候选人
- 46.3. 条件注释
 - 46.3.1. 班级条件
 - 46.3.2. 豆条件

- 46.3.3. 财产状况
- 46.3.4. 资源条件
- 46.3.5. Web应用程序条件
- 46.3.6. SpEL表达条件

46.4. 测试你的自动配置

- 46.4.1. 模拟Web上下文
- 46.4.2. 覆盖类路径

46.5. 创建你自己的启动器

- 46.5.1. 命名
- 46.5.2. `autoconfigure` 模块
- 46.5.3. 入门模块

47. Kotlin的支持

47.1. 要求

47.2. 空安全

47.3. Kotlin API

- 47.3.1. `runApplication`
- 47.3.2. 扩展

47.4. 依赖管理

47.5. `@ConfigurationProperties`

47.6. 测试

47.7. 资源

- 47.7.1. 进一步阅读
- 47.7.2. 例子

48.下一步阅读什么

V. Spring Boot执行器：生产就绪功能

49.启用生产就绪功能

50.终点

50.1. 启用端点

50.2. 暴露端点

50.3. 保护HTTP端点

50.4. 配置端点

50.5. 执行器Web终端的超媒体

50.6. 执行器Web端点路径

50.7. CORS支持

50.8. 实现自定义端点

- 50.8.1. 接收输入
输入类型转换
- 50.8.2. 自定义Web端点
Web端点请求谓词
路径
HTTP方法
消费
产生
Web端点响应状态
Web端点范围请求
Web端点安全
- 50.8.3. Servlet端点
- 50.8.4. 控制器端点

50.9. 健康信息

- 50.9.1. 自动配置的HealthIndicators
- 50.9.2. 编写自定义HealthIndicators
- 50.9.3. 反应性健康指标
- 50.9.4. 自动配置的ReactiveHealthIndicators

50.10. 应用信息

- 50.10.1. 自动配置InfoContributors
- 50.10.2. 自定义应用信息
- 50.10.3. Git提交信息

- 50.10.4. 构建信息
- 50.10.5. 编写自定义InfoContributors

51.通过HTTP进行监控和管理

- 51.1. 自定义管理端点路径
- 51.2. 自定义管理服务器端口
- 51.3. 配置管理特定的SSL
- 51.4. 自定义管理服务器地址
- 51.5. 禁用HTTP端点

52.通过JMX进行监控和管理

- 52.1. 定制MBean名称
- 52.2. 禁用JMX终结点
- 52.3. 通过HTTP使用Jolokia进行JMX
 - 52.3.1. 定制Jolokia
 - 52.3.2. 禁用Jolokia

伐木者

- 53.1. 配置记录器

54.度量

- 54.1. 入门
- 54.2. 支持的监测系统
 - 54.2.1. 奥图
 - 54.2.2. Datadog
 - 54.2.3. 神经节
 - 54.2.4. 石墨
 - 54.2.5. 辐辏
 - 54.2.6. JMX
 - 54.2.7. 新的遗物
 - 54.2.8. 普罗米修斯
 - 54.2.9. SignalFx
 - 54.2.10. 简单
 - 54.2.11. StatsD
 - 54.2.12. 波前
- 54.3. 支持的度量标准
 - 54.3.1. Spring MVC度量标准
 - 54.3.2. Spring WebFlux指标
 - 54.3.3. RestTemplate指标
 - 54.3.4. 高速缓存指标
 - 54.3.5. 数据源指标
 - 54.3.6. RabbitMQ指标
- 54.4. 注册自定义指标
- 54.5. 自定义各个指标
 - 54.5.1. 每米性能
- 54.6. 指标终点

55.审计

56. HTTP跟踪

- 56.1. 自定义HTTP跟踪

57.过程监测

- 57.1. 扩展配置
- 57.2. 编程

58. Cloud Foundry支持

- 58.1. 禁用扩展Cloud Foundry执行器支持
- 58.2. Cloud Foundry自签名证书
- 58.3. 自定义上下文路径

59.接下来要读什么

VI。部署Spring Boot应用程序

60.部署到云

60.1。Cloud Foundry

60.1.1. 绑定到服务

60.2。Heroku的

60.3。OpenShift

60.4。亚马逊网络服务 (AWS)

60.4.1. AWS Elastic Beanstalk

使用Tomcat平台

使用Java SE平台

60.4.2. 概要

60.5。Boxfuse和亚马逊网络服务

60.6。Google Cloud

61.安装Spring Boot应用程序

61.1。支持的操作系统

61.2。Unix / Linux服务

61.2.1. 安装即 init.d 服务 (System V)

确保 init.d 服务

61.2.2. 安装即 systemd 服务

61.2.3. 自定义启动脚本

在写入时自定义启动脚本

在运行时自定义脚本

61.3。Microsoft Windows服务

62.接下来要读什么

七。Spring Boot CLI

63.安装CLI

64.使用CLI

64.1。使用CLI运行应用程序

64.1.1. 推导出“抢”依赖

64.1.2. 推导出“抢”坐标

64.1.3. 默认导入语句

64.1.4. 自动主要方法

64.1.5. 定制依赖管理

64.2。有多个源文件的应用程序

64.3。打包你的应用程序

64.4。初始化新项目

64.5。使用嵌入式外壳

64.6。向CLI添加扩展

65.使用Groovy Beans DSL开发应用程序

66.使用CLI配置CLI `settings.xml`

67.接下来要读什么

八。构建工具插件

68. Spring Boot Maven插件

68.1。包括插件

68.2。打包可执行的jar和war文件

69. Spring Boot Gradle插件

70. Spring Boot AntLib模块

70.1. Spring Boot Ant任务

- 70.1.1. `spring-boot:execjar`
- 70.1.2. 例子

70.2. `spring-boot:findmainclass`

- 70.2.1. 例子

71.支持其他构建系统

71.1. 重新包装档案

71.2. 嵌套库

71.3. 找到一个主要类

71.4. 示例重新打包实施

72.接下来要读什么

IX. '指导'指南

73. Spring Boot应用程序

- 73.1. 创建你自己的FailureAnalyzer
- 73.2. 解决自动配置问题
- 73.3. 在开始之前自定义环境或ApplicationContext
- 73.4. 构建ApplicationContext层次结构（添加父级或根级上下文）
- 73.5. 创建一个非Web应用程序

74.属性和配置

74.1. 在构建时自动扩展属性

- 74.1.1. 使用Maven自动扩展属性
- 74.1.2. 使用Gradle的自动属性扩展

74.2. 外部化配置`SpringApplication`

74.3. 更改应用程序的外部属性的位置

74.4. 使用'短'命令行参数

74.5. 使用YAML作为外部属性

74.6. 设置活动的弹簧配置文件

74.7. 根据环境更改配置

74.8. 发现外部属性的内置选项

75.嵌入式Web服务器

75.1. 使用另一个Web服务器

75.2. 配置码头

75.3. 将Servlet，Filter或Listener添加到应用程序

- 75.3.1. 使用Spring Bean添加Servlet，Filter或Listener
禁用Servlet或Filter的注册
- 75.3.2. 通过使用类路径扫描添加Servlet，筛选器和监听器

75.4. 更改HTTP端口

75.5. 使用随机未分配的HTTP端口

75.6. 在运行时发现HTTP端口

75.7. 配置SSL

75.8. 配置HTTP / 2

- 75.8.1. HTTP / 2与Undertow
- 75.8.2. HTTP / 2与Jetty
- 75.8.3. HTTP / 2与Tomcat

75.9. 配置访问日志记录

75.10. 运行在前端代理服务器后面

75.10.1. 自定义Tomcat的代理配置

75.11. 配置Tomcat

75.12. 使用Tomcat启用多个连接器

75.13. 使用Tomcat的LegacyCookieProcessor

75.14. 配置Undertow

75.15. 使用Undertow启用多个监听器

75.16. 使用@ServerEndpoint创建WebSocket端点

75.17. 启用HTTP响应压缩

76. Spring MVC

76.1. 编写一个JSON REST服务

76.2. 编写一个XML REST服务

76.3. 自定义Jackson ObjectMapper

76.4. 自定义@ResponseBody呈现

76.5. 处理多部分文件上传

76.6. 关闭Spring MVC DispatcherServlet

76.7. 关闭默认的MVC配置

76.8. 自定义ViewResolvers

77. HTTP客户端

77.1. 配置RestTemplate以使用代理

记录

78.1. 配置Logback进行日志记录

78.1.1. 为纯文件输出配置Logback

78.2. 配置Log4j进行日志记录

78.2.1. 使用YAML或JSON配置Log4j 2

79. 数据访问

79.1. 配置一个自定义数据源

79.2. 配置两个数据源

79.3. 使用Spring数据存储库

79.4. Spring配置分离@实体定义

79.5. 配置JPA属性

79.6. 配置Hibernate命名策略

79.7. 使用自定义EntityManagerFactory

79.8. 使用两个EntityManagers

79.9. 使用传统 persistence.xml 文件

79.10. 使用Spring Data JPA和Mongo仓库

79.11. 将Spring数据存储库公开为REST端点

79.12. 配置由JPA使用的组件

79.13. 用两个数据源配置jOOQ

80. 数据库初始化

80.1. 使用JPA初始化数据库

80.2. 使用Hibernate初始化数据库

80.3. 初始化数据库

80.4. 初始化一个Spring批处理数据库

80.5. 使用更高级别的数据库迁移工具

80.5.1. 启动时执行Flyway数据库迁移

80.5.2。在启动时执行Liquibase数据库迁移

信息

81.1。禁用事务处理JMS会话

82.批量应用程序

82.1。在启动时执行Spring批处理作业

83.执行器

83.1。更改执行器端点的HTTP端口或地址

83.2。自定义'whitelabel'错误页面

84.安全

84.1。关闭Spring Boot安全配置

84.2。更改UserDetailsService和添加用户帐户

84.3。在代理服务器后运行时启用HTTPS

85.热插拔

85.1。重新加载静态内容

85.2。重新加载模板而不重新启动容器

85.2.1。Thymeleaf模板

85.2.2。FreeMarker模板

85.2.3。Groovy模板

85.3。快速应用程序重启

85.4。重新加载Java类而不重新启动容器

86.建设

86.1。生成构建信息

86.2。生成Git信息

86.3。自定义依赖版本

86.4。用Maven创建一个可执行的JAR

86.5。使用Spring Boot应用程序作为依赖项

86.6。当可执行jar运行时提取特定的库

86.7。用排除项创建一个不可执行的JAR

86.8。远程调试Maven启动的Spring Boot应用程序

86.9。在不使用的情况下从Ant构建可执行文件 `spring-boot-antlib`

87.传统部署

87.1。创建一个可部署的战争文件

87.2。为较老的Servlet容器创建一个可部署的战争文件

87.3。将现有的应用程序转换为Spring Boot

87.4。将WAR部署到WebLogic

87.5。在旧的（Servlet 2.5）容器中部署WAR

87.6。使用Jedis代替生菜

十，附录

A.通用应用程序属性

B.配置元数据

B.1。元数据格式

B.1.1。组属性

B.1.2。属性属性

B.1.3。提示属性

B.1.4。重复的元数据项目

B.2. 提供手册提示

B.2.1。价值提示

B.2.2。价值提供者

任何

类参考

处理为

记录器名称

Spring Bean参考

Spring配置文件名称

B.3. 使用注释处理器生成您自己的元数据

B.3.1。嵌套属性

B.3.2。添加额外的元数据

C. 自动配置类

C.1. 从“spring-boot-autoconfigure”模块

C.2. 从“spring-boot-actuator-autoconfigure”模块

D. 测试自动配置注释

E. 可执行的Jar格式

E.1. 嵌套JAR

E.1.1。可执行jar文件结构

E.1.2。可执行的战争文件结构

E.2. Spring Boot的“JarFile”类

E.2.1。与标准Java“JarFile”兼容

E.3. 启动可执行的罐子

E.3.1。启动器清单

E.3.2。爆炸档案

E.4. PropertiesLauncher 特征

E.5. 可执行的jar限制

E.6. 替代性单罐解决方案

F. 依赖版本

第一部分Spring Boot文档

本节简要介绍Spring Boot参考文档。它作为文档其余部分的地图。

1. 关于文档

Spring Boot参考指南可用

- [HTML](#)
- [PDF](#)
- [EPUB](#)

最新副本位于docs.spring.io/spring-boot/docs/current/reference/。

本文档的副本可以为您自己使用并分发给其他人，前提是您不收取这些副本的任何费用，并进一步规定每份副本均包含此版权声明，无论是以印刷版还是电子版分发。

2. 获得帮助

如果您在Spring Boot遇到问题，我们希望能提供帮助。

- 试用[How-to文档](#)。他们为最常见的问题提供解决方案。
- 学习Spring基础知识。Spring Boot构建在许多其他Spring项目上。查阅spring.io网站获取大量参考文档。如果您刚开始使用Spring，请尝试其中一个指南。

- 问一个问题。我们用标记的问题监视[stackoverflow.com/spring-boot](https://stackoverflow.com/questions/tagged/spring-boot)。
- 使用Spring Boot在github.com/spring-projects/spring-boot/issues报告错误。



所有的Spring Boot都是开源的，包括文档。如果您发现文档有问题，或者您想改进它们，请参与其中。

3.第一步

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

- [从头开始](#)：概览 | 要求 | 安装
- [教程](#)：第1部分 | 第2部分
- [运行您的示例](#)：第1部分 | 第2部分

4.使用Spring Boot

准备好真正开始使用Spring Boot？我们有你涵盖：

- [构建系统](#)：Maven | Gradle | Ant | 首发
- [最佳实践](#)：代码结构 | @Configuration | @EnableAutoConfiguration | 豆类和依赖注入
- [运行您的代码](#) IDE | 打包 | Maven | 摆籃
- [打包您的应用程序](#)：生产罐子
- [Spring Boot CLI](#)：使用CLI

5.了解Spring Boot特性

需要更多关于Spring Boot核心功能的细节？[以下内容适合您](#)：

- [核心特性](#)：SpringApplication | 外部配置 | 配置文件 | 记录
- [Web应用程序](#)：MVC | 嵌入式容器
- [使用数据](#)：SQL | NO-SQL
- [消息](#)：概述 | JMS
- [测试](#)：概览 | 启动应用程序 | utils的
- [扩展](#)：自动配置 | @条件

6.转向生产

当您准备将Spring Boot应用程序推向生产时，我们有一些您可能会喜欢的技巧：

- [管理端点](#)：概览 | 定制
- [连接选项](#)：HTTP | JMX
- [监测](#)：指标 | 审计 | 追踪 | 处理

7.高级主题

最后，我们有更多高级用户的几个主题：

- [Spring Boot应用程序部署](#)：云部署 | OS服务
- [构建工具插件](#)：Maven | 摆籃
- [附录](#)：应用程序属性 | 自动配置类 | 可执行的罐子

第二部分。入门

如果您正在开始使用Spring Boot，或者通常使用“Spring”，请先阅读本节。它回答了基本的“什么？”，“如何？”和“为什么？”的问题。它包括Spring Boot的介绍以及安装说明。然后，我们将引导您构建您的第一个Spring Boot应用程序，并讨论一些核心原则。

8.介绍Spring Boot

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

您可以使用Spring Boot来创建可以使用`java -jar`或更传统的战争部署来启动的Java应用程序。我们还提供了一个运行“春天脚本”的命令行工具。

我们的主要目标是：

- 为所有Spring开发提供一个更快，更广泛的入门体验。
- 立即开始斟酌，但是随着需求开始偏离默认值，快速避开。
- 提供大型项目（如嵌入式服务器，安全性，指标，运行状况检查和外部配置）通用的一系列非功能性功能。
- 绝对不会生成代码，并且不需要XML配置。

9.系统要求

Spring Boot 2.0.1.BUILD-SNAPSHOT需要Java 8或9以及 Spring Framework 5.0.5.BUILD-SNAPSHOT或更高版本。为Maven 3.2+和Gradle 4提供了明确的构建支持。

9.1 Servlet容器

Spring Boot支持以下嵌入式servlet容器：

名称	Servlet版本
Tomcat 8.5	3.1
码头9.4	3.1
Undertow 1.4	3.1

您也可以将Spring Boot应用程序部署到任何Servlet 3.0+兼容容器。

10.安装Spring Boot

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

```
$ java -version
```

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

10.1 Java开发人员的安装说明

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

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

10.1.1 Maven安装

Spring Boot与Apache Maven 3.2或更高版本兼容。如果您尚未安装Maven，则可以按照maven.apache.org上的说明进行操作。



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

Spring Boot依赖关系使用`org.springframework.boot groupId`。通常，您的Maven POM文件从`spring-boot-starter-parent`项目中继承并向一个或多个“Starter”声明依赖关系。Spring Boot还提供了一个可选的 [Maven插件](#)来创建可执行的jar。

以下清单显示了一个典型的 `pom.xml` 文件：

```

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

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

    <! - 从Spring Boot继承默认值 - >
    <parent>
        <groupId> org.springframework.boot </ groupId>
        <artifactId> spring-boot-starter-parent </ artifactId>
        <version> 2.0.1.BUILD-SNAPSHOT </ version>
    </ parent>

    <! - 为Web应用程序添加典型的依赖关系 - >
    <dependencies>
        <dependency>
            <groupId> org.springframework.boot </ groupId>
            <artifactId> spring-boot-starter-web </ artifactId>
        </ dependency>
    </ dependencies>

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

    <! - 添加Spring存储库 - >
    <! - (如果您使用的是.RELEASE版本，则不需要此操作) - >
    <repositories>
        <repository>
            <id> spring-snapshots </ id>
            <url> https://repo.spring.io/snapshot </ url>
            <snapshot> true </ snapshot>
        </ repository>
    </ repositories>
    <pluginRepositories>
        <pluginRepository>
            <id> spring-snapshots </ id>
            <url> https://repo.spring.io/snapshot </ url>
        </ pluginRepository>
        <pluginRepository>
            <id> spring-milestones </ id>
            <url> https://repo.spring.io/milestone </ url>
        </ pluginRepository>
    </ pluginRepositories>
</ project>
```



这 `spring-boot-starter-parent` 是使用 Spring Boot 的好方法，但它可能并不适合所有的时间。有时您可能需要从不同的父 POM 继承，或者您可能不喜欢我们的默认设置。在这些情况下，请参见 第13.2.2节“使用没有父 POM 的 Spring Boot”，以获得使用 `import` 范围的替代解决方案。

10.1.2 Gradle 安装

Spring Boot 与 Gradle 4 兼容。如果您还没有安装 Gradle，可以按照 [gradle.org 上的说明进行操作](#)。

Spring引导依赖可以通过使用 `org.springframework.boot` group。通常，您的项目将依赖项声明为一个或多个“Starter”。Spring Boot提供了一个有用的Gradle插件，可以用来简化依赖声明和创建可执行的jar。

Gradle包装

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

以下示例显示了一个典型的 `build.gradle` 文件：

```
buildscript {
    存储库{
        jcenter()
        maven {url 'https://repo.spring.io/snapshot' }
        maven {url 'https://repo.spring.io/milestone' }
    }
    依赖关系{
        classpath 'org.springframework.boot: spring-boot-gradle-plugin: 2.0.1.BUILD-SNAPSHOT'
    }
}

apply plugin: 'java'
plugin: 'org.springframework.boot' apply
plugin: 'io.spring.dependency-management'

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

存储库{
    jcenter()
    maven {url "https://repo.spring.io/snapshot" }
    maven {url "https://repo.spring.io/milestone" }
}

依赖关系{
    编译 ("org.springframework.boot: spring-boot-starter-web")
    testCompile ("org.springframework.boot: spring-boot-starter-test")
}
```

10.2 安装Spring Boot CLI

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

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

10.2.1 手动安装

您可以从Spring软件存储库下载Spring CLI分发版：

- [spring-boot-cli-2.0.1.BUILD-SNAPSHOT-bin.zip](#)
- [弹簧引导CLI-2.0.1.BUILD-快照bin.tar.gz](#)

最先进的快照分布也可用。

下载完成后，请按照解压缩归档中的 `INSTALL.txt` 说明进行操作。总之，文件中的目录中有一个 `spring` 脚本（`spring.bat` 用于Windows）。或者，您可以使用该文件（该脚本可帮助您确保类路径设置正确）。`bin/` `.zip` `java -jar .jar`

10.2.2 使用SDKMAN安装！

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

```
$ sdk安装springboot
$ spring --version
Spring Boot v2.0.1.BUILD-SNAPSHOT
```

如果您为CLI开发功能并希望轻松访问您构建的版本，请使用以下命令：

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-2.0.1.BUILD-SNAPSHOT-bin/spring-2
$ sdk默认springboot dev
$ spring --version
Spring CLI v2.0.1.BUILD-SNAPSHOT
```

前面的说明安装一个 `spring` 称为 `dev` 实例的本地实例。它指向您的目标构建位置，因此每次重建Spring Boot时 `spring` 都是最新的。

您可以通过运行以下命令来查看它：

```
$ sdk ls springboot

=====
可用的Springboot版本
=====
> + dev
* 2.0.1.BUILD-SNAPSHOT

=====
+ - 本地版本
* - 已安装
> - 目前正在使用
=====
```

10.2.3 OSX Homebrew安装

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

```
$ brew tap pivotal / tap
$ brew安装springboot
```

Homebrew安装 `spring` 到 `/usr/local/bin`。



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

10.2.4 MacPorts安装

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

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

10.2.5命令行完成

Spring Boot CLI包含为BASH和zsh shell 提供命令完成的脚本。您可以在任何shell `source` 中将脚本（也称为命名 `spring`）放入您的个人或系统范围的bash完成初始化中。在Debian系统上，系统范围的脚本处于启动状态，`/shell-completion/bash` 并且该目录中的所有脚本都会在新shell启动时执行。例如，如果您使用SDKMAN！安装了手动运行脚本，请使用以下命令：

```
$. ~/ .sdkman /候选人/ springboot /电流/壳完成/庆典/弹簧
$ spring <HIT TAB HERE>
抓取帮助jar运行测试版本
```



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

10.2.6快速启动Spring CLI示例

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

```
@RestController
类 ThisWillActuallyRun {

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

```
    "你好，世界！"
}

}
```

然后从一个shell运行它，如下所示：

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



由于下载依赖项，应用程序的第一次运行速度很慢。后续运行速度更快。

`localhost:8080` 在你喜欢的网页浏览器中打开。您应该看到以下输出：

```
你好，世界！
```

10.3从较早版本的Spring Boot升级

如果您是从较早版本的Spring Boot进行升级，请查看 [项目wiki上的“迁移指南”](#)，其中提供了详细的升级说明。还要检查“发行说明”，以获取每个发行版的“新功能”和“值得注意”功能列表。

要升级现有的CLI安装，请使用相应的软件包管理器命令（例如`brew upgrade`），或者如果您手动安装了CLI，请按照[标准说明进行操作](#)，并记住更新`PATH`环境变量以删除所有旧的引用。

11.开发你的第一个Spring Boot应用程序

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



该spring.io网站包含了许多“入门”指南使用Spring的引导。如果您需要解决特定问题，请先在那里查看。

您可以通过转到[start.spring.io](#)并从依赖关系搜索器中选择“Web”起始器来[缩短](#)以下步骤。这样做会产生一个新的项目结构，以便您可以立即开始编码。查看[Spring Initializr文档](#)以获取更多详细信息。

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

```
$ java -version
java版本“1.8.0_102”
Java (TM) SE运行时环境 (build 1.8.0_102-b14)
Java HotSpot (TM) 64位服务器虚拟机 (构建25.102-b14, 混合模式)
```

```
$ mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T16: 41: 47 + 00: 00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java版本: 1.8.0_102, 供应商: 甲骨文公司
```



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

11.1创建POM

我们需要先创建一个Maven `pom.xml` 文件。本 `pom.xml` 是用来构建项目的配方。打开您最喜欢的文本编辑器并添加以下内容：

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

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

    <parent>
        <groupId> org.springframework.boot </ groupId>
        <artifactId> spring-boot-starter-parent </ artifactId>
```

```

<version> 2.0.1.BUILD-SNAPSHOT </ version>
</ parent>

<! - 在此处添加的其他行... - >

<! - (如果您使用.RELEASE 版本, 则不需要此操作) - >
<repositories>
    <repository>
        <id> spring-snapshots </ id>
        <url> https://repo.spring.io/snapshot </ url>
        <snapshots> <enabled> true </ enabled> </ snapshots>
    </ repository>
    <repository>
        <id> spring-milestones </ id>
        <url> https://repo.spring.io/milestone </ url>
    </ repository>
</ repositories>
<pluginRepositories>
    <pluginRepository>
        <id> spring-snapshots </ id>
        <url> https://repo.spring.io/snapshot </ url>
    </ pluginRepository>
    <pluginRepository>
        <id> spring-milestones </ id>
        <url> https://repo.spring.io/milestone </ url>
    </ pluginRepository>
</ pluginRepositories>
</ project>

```

上面的列表应该给你一个工作版本。您可以通过运行来测试它 `mvn package` (现在, 您可以忽略“jar将为空 - 没有内容被标记为包含!”警告)。



此时, 您可以将项目导入IDE (大多数现代Java IDE包含对Maven的内置支持)。为了简单起见, 我们在本例中继续使用纯文本编辑器。

11.2添加类路径依赖关系

Spring Boot提供了许多“入门”, 可让您将jar添加到类路径中。我们的示例应用程序已经用于POM `spring-boot-starter-parent` 的 `parent` 部分。这 `spring-boot-starter-parent` 是一个提供有用的Maven默认值的特殊启动器。它还提供了一个 `dependency-management` 部分, 以便您可以省略 `version` “祝福”依赖项的标签。

其他“Starter”提供了在开发特定类型的应用程序时可能需要的依赖关系。由于我们正在开发一个Web应用程序, 我们添加了一个 `spring-boot-starter-web` 依赖项。在此之前, 我们可以通过运行以下命令来查看我们目前的功能:

```
$ mvn 依赖: 树
[INFO] com.example: myproject: jar: 0.0.1-SNAPSHOT
```

该 `mvn dependency:tree` 命令打印您的项目依赖关系的树形表示。你可以看到它 `spring-boot-starter-parent` 本身不提供依赖关系。要添加必要的依赖关系, 请编辑 `pom.xml` 并 `spring-boot-starter-web` 在该 `parent` 部分正下方添加 依赖项:

```

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

```

如果你 `mvn dependency:tree` 再次运行, 你会发现现在有很多附加的依赖关系, 包括Tomcat Web服务器和Spring Boot本身。

11.3编写代码

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

```

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

```

```
@RestController  
@EnableAutoConfiguration  
public class Example {  
  
    @RequestMapping(“/”)  
    String home() {  
        返回 “Hello World! ” ;  
    }  
  
    公共 静态 无效的主要 (字符串 []参数) 抛出异常 {  
        SpringApplication.run (实施例类, 参数);  
    }  
}
```

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

11.3.1 @RestController和@RequestMapping注解

我们 Example 班的第一个注释是 `@RestController`。这被称为 **刻板印记**。它为阅读代码的人提供了线索，对于 Spring 来说，这个类扮演着特定的角色。在这种情况下，我们的类是一个 Web `@Controller`，所以 Spring 在处理传入的 Web 请求时会考虑它。

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



在 `@RestController` 与 `@RequestMapping` 注解是 Spring MVC 的注解。（它们并不特定于 Spring Boot。）有关更多详细信息，请参阅 Spring 参考手册中的 [MVC 部分](#)。

11.3.2 @EnableAutoConfiguration注解

第二个级别注释是 `@EnableAutoConfiguration`。这个注解告诉Spring Boot根据你添加的jar依赖来“猜测”你想要如何配置Spring。自从 `spring-boot-starter-web` 添加了Tomcat和Spring MVC之后，自动配置假定您正在开发一个Web应用程序并相应地设置Spring。

启动器和自动配置

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

11.3.3“主要”方法

我们应用程序的最后一部分是该 `main` 方法。这只是一个遵循Java约定的应用程序入口点的标准方法。我们的主要方法 `SpringApplication` 通过调用委托给Spring Boot的类 `run`。`SpringApplication` 引导我们的应用程序，从Spring开始，然后启动自动配置的Tomcat Web服务器。我们需要 `Example.class` 将该 `run` 方法的参数作为参数传递，以确定 `SpringApplication` 哪些是主要的Spring组件。该 `args` 数组也通过以显示任何命令行参数。

11.4运行示例

在这一点上，你的应用程序应该工作。由于您使用了 `spring-boot-starter-parent` POM，`run` 因此您可以使用一个有用的目标来启动应用程序：`mvn spring-boot:run` 从根项目目录中键入以启动应用程序。您应该看到类似于以下内容的输出：

```
$ mvn spring-boot:运行
. __ - - -
/ \ \ / ____ - - - ( ) - - - - \ \ \ \
( ) \ \ | _ | _ | | _ \ / _ | \ \ \ \
\ \ / _ ) | | _ ) | | | | | | ( _ | ) ) )
' | _ | . _ | _ | _ | _ | | | \ _ , | / / /
===== | _ | ====== | _ / = / _ / _ / _ /
:: Spring Boot :: (v2.0.1.BUILD-SNAPSHOT)
..... .
..... .
..... (在这里输出日志)
..... .
..... 在2.222秒内启动示例 (运行6.514的JVM)
```

如果您打开一个Web浏览器 `localhost:8080`，您应该看到以下输出：

你好，世界！

要正常退出应用程序，请按 `ctrl+c`。

11.5 创建可执行jar

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

可执行的jar和Java

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

为了解决这个问题，许多开发者使用“超级”罐子。一个超级jar将所有应用程序依赖关系中的所有类打包到一个单独的存档中。这种方法的问题是很难看到你的应用程序中有哪些库。如果在多个罐子中使用相同的文件名（但是具有不同的内容），则它也可能是有问题的。

Spring Boot采用了不同的方法，可以让您直接嵌入罐子。

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

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



所述 `spring-boot-starter-parent` POM 包括 `<executions>` 配置以结合 `repackage` 目标。如果您不使用父POM，则需要自行声明此配置。有关详细信息，请参阅插件文档。

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

```
$ mvn包
[INFO]扫描项目...
[信息]
[INFO] -----
[INFO]构建myproject 0.0.1-SNAPSHOT
[INFO] -----
[信息] ....
[INFO] --- maven-jar-plugin: 2.4: jar (默认jar) @ myproject ---
[INFO]构建jar: /Users/developer/example/spring-boot-example/target/myproject-0.0.1-SNAPSHOT.jar
[信息]
[INFO] --- spring-boot-maven-plugin: 2.0.1.BUILD-SNAPSHOT: 重新包装 (默认) @ myproject ---
[INFO] -----
[信息]建立成功
[INFO] -----
```

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

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

你也应该看到一个名为小得多文件 `myproject-0.0.1-SNAPSHOT.jar.original` 的 `target` 目录。这是Maven在被Spring Boot重新包装之前创建的原始jar文件。

要运行该应用程序，请使用以下 `java -jar` 命令：

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

```
   _ _ _ _ _ 
  / \ / \ / \ / \ 
  ( ) ( ) ( ) ( ) 
  \ / \ / \ / \ / \ 
  | | | | | | | | 
  ' ' ' ' ' ' ' '
```

```
===== | _ | ===== | __ / = / _ / _ / _ / 
:: Spring Boot :: (v2.0.1.BUILD-SNAPSHOT)
..... .
..... . (在这里输出日志)
..... .
..... 在2.536秒内启动示例 (运行2.864的JVM)
```

和以前一样，按退出该应用程序 **ctrl-c**。

12.下一步阅读什么

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

在[春季启动库](#)也有一堆样品可以运行。样本与代码的其余部分无关（也就是说，您不需要构建其余代码以运行或使用样本）。

否则，下一个逻辑步骤是阅读[第三部分“使用Spring Boot”](#)。如果你真的不耐烦，你也可以跳到前面阅读[Spring Boot的功能](#)。

第三部分。使用Spring Boot

本节将详细介绍如何使用Spring Boot。它涵盖了构建系统，自动配置以及如何运行应用程序等主题。我们还介绍了一些Spring Boot的最佳实践。尽管Spring Boot没有特别的特殊之处（它只是您可以使用的另一个库），但有一些建议，如果遵循这些建议，您的开发过程会更容易一些。

如果您刚开始使用Spring Boot，那么在深入本节之前，您应该阅读[入门指南](#)。

13.建立系统

强烈建议您选择支持[依赖关系管理](#)的构建系统，并且可以使用发布到“Maven Central”存储库的构件。我们建议您选择Maven或Gradle。Spring Boot可以与其他构建系统（例如Ant）一起工作，但它们并没有得到特别好的支持。

13.1 依赖管理

Spring Boot的每个发行版都提供了它支持的依赖关系的策略列表。实际上，您不需要为构建配置中的任何这些依赖项提供版本，因为Spring Boot为您管理这些版本。当您升级Spring Boot本身时，这些依赖关系也会以一致的方式升级。



如果需要的话，你仍然可以指定一个版本并覆盖Spring Boot的建议。

策划列表包含您可以在Spring Boot中使用的所有Spring模块以及第三方库的精炼列表。该列表可用作标准的物料清单([spring-boot-dependencies](#))，可用于Maven和Gradle。



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

13.2 Maven

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

- Java 1.8作为默认的编译器级别。
- UTF-8源码编码。
- 一个[依赖管理部分](#)，从春天启动依赖性继承POM，管理公共依赖的版本。这种依赖关系管理可以让您在自己的pom中使用这些依赖项时忽略<version>标记。
- 明智的[资源过滤](#)。
- 明智的插件配置（[exec插件](#)，[Git提交ID](#)和[阴影](#)）。
- 明智的资源过滤[application.properties](#)和[application.yml](#)包括配置文件特定的文件（例如[application-dev.properties](#)和[application-dev.yml](#)）

请注意，由于[application.properties](#)和[application.yml](#)文件接受Spring样式占位符（ `${...}` ），Maven过滤被更改为使用`@..@`占位符。（你可以通过设置一个叫做Maven的属性来覆盖它[resource.delimiter](#)。）

13.2.1 继承初始父项

要配置您的项目从中继承 `spring-boot-starter-parent`，请设置 `parent` 如下：

```
<! - 从Spring Boot继承默认值 - >
<parent>
    <groupId> org.springframework.boot </ groupId>
    <artifactId> spring-boot-starter-parent </ artifactId>
    <version> 2.0.1.BUILD-SNAPSHOT </ version>
</ parent>
```



您应该只需在此依赖项上指定 Spring Boot 版本号。如果您导入额外的启动器，则可以安全地省略版本号。

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

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



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

13.2.2 在没有Parent POM的情况下使用Spring Boot

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

如果你不想使用它 `spring-boot-starter-parent`，你仍然可以通过依赖来保持依赖管理的好处（但不是插件管理）`scope=import`，如下所示：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <! - 从Spring Boot 导入依赖项管理 - >
            <groupId> org.springframework.boot </ groupId>
            <artifactId> spring-boot-dependencies </ artifactId>
            <version> 2.0.1.BUILD-SNAPSHOT </ version>
            <type> pom </ type>
            <scope> import </ scope>
        </ dependency>
    </ dependencies>
</ dependencyManagement>
```

如上所述，上述示例设置不会让您使用属性重写个别依赖关系。要达到相同的结果，您需要在输入之前在 `dependencyManagement` 项目中添加一个条目。例如，要升级到另一个 Spring Data 发行版，您可以将以下元素添加到您的：`spring-boot-dependencies pom.xml`

```
<dependencyManagement>
    <dependencies>
        <! - 覆盖Spring Boot 提供的数据发布训练 - >
        <dependency>
            <groupId> org.springframework.data </ groupId>
            <artifactId> spring-data-releasetrain </ artifactId>
            <version> Fowler-SR2 </ version>
            <type> pom </ type>
            <scope> import </ scope>
        </ dependency>
        <dependency>
            <groupId> org.springframework.boot </ groupId>
            <artifactId> spring-boot-dependencies </ artifactId>
            <version> 2.0.1.BUILD-SNAPSHOT </ version>
            <type> pom </ type>
            <scope> import </ scope>
        </ dependency>
    </ dependencies>
</ dependencyManagement>
```



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

13.2.3 使用Spring Boot Maven插件

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

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



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

13.3 Gradle

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

- 参考 (HTML 和 PDF)
- API

13.4 蚂蚁

可以使用Apache Ant + Ivy构建Spring Boot项目。该“spring-boot-antlib”的antlib模块还可以帮助蚂蚁创建可执行的JAR文件。

为了声明依赖关系，典型的ivy.xml文件看起来像下面的例子：

```
<ivy-module version = "2.0" >
    <info organization = "org.springframework.boot" module = "spring-boot-sample-ant" />
    <configurations>
        <conf name = "compile" description = "everything needed to编译这个模块" />
        <conf name = "runtime" extends = "compile" description = "运行此模块所需的一切" />
    </ configurations>
    <dependencies>
        <依赖 org = "org.springframework.boot" name = "spring-boot-starter"
              rev = "${spring-boot.version}" conf = "compile" />
    </ dependencies>
</ ivy-module>
```

典型的build.xml外观如下例所示：

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

    <property name = "spring-boot.version" value = "2.0.1.BUILD-SNAPSHOT" />

    <目标 名称 = "解析" 描述 = " - >检索与常春藤依赖性" >
        <常春藤: 检索 模式 = "LIB / [CONF] / [工件] - [式] - [修改] [EXT]" />
    </目标>

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

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

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

```

</ target>

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

```



如果你不希望使用 `spring-boot-antlib` 模块，请参阅 第86.9，“创建一个从蚂蚁的可执行文件存档，但不使用 `spring-boot-antlib`”“如何做”。

13.5 起动器

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

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

什么是名字

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

正如“[创建自己的启动器](#)”部分所述，第三方启动器不应该从第一启动器开始 `spring-boot`，因为它是为官方Spring Boot工件保留的。相反，第三方初学者通常以项目名称开头。例如，所调用的第三方初始项目 `thirdpartyproject` 通常会被命名为 `thirdpartyproject-spring-boot-starter`。

以下应用程序启动程序由Spring Boot在 `org.springframework.boot` 组中提供：

表13.1。Spring Boot应用程序启动器

名称	描述	
<code>spring-boot-starter</code>	核心入门者，包括自动配置支持，日志记录和YAML	双响炮
<code>spring-boot-starter-activemq</code>	使用Apache ActiveMQ启动JMS消息传递	双响炮
<code>spring-boot-starter-amqp</code>	使用Spring AMQP和Rabbit MQ的入门者	双响炮
<code>spring-boot-starter-aop</code>	使用Spring AOP和AspectJ进行面向方面编程的入门者	双响炮
<code>spring-boot-starter-artemis</code>	使用Apache Artemis开始JMS消息传递	双响炮
<code>spring-boot-starter-batch</code>	使用Spring Batch的入门者	双响炮
<code>spring-boot-starter-cache</code>	Starter使用Spring Framework的缓存支持	双响炮

名称	描述	双响炮
<code>spring-boot-starter-cloud-connectors</code>	Starter使用Spring Cloud Connectors，可简化Cloud Foundry和Heroku等云平台中的服务连接	双响炮
<code>spring-boot-starter-data-cassandra</code>	入门使用Cassandra分布式数据库和Spring Data Cassandra	双响炮
<code>spring-boot-starter-data-cassandra-reactive</code>	使用Cassandra分布式数据库和Spring Data Cassandra Reactive的初学者	双响炮
<code>spring-boot-starter-data-couchbase</code>	使用Couchbase面向文档的数据库和Spring Data Couchbase的初学者	双响炮
<code>spring-boot-starter-data-couchbase-reactive</code>	初级用于使用Couchbase面向文档的数据库和Spring Data Couchbase Reactive	双响炮
<code>spring-boot-starter-data-elasticsearch</code>	使用Elasticsearch搜索和分析引擎和Spring Data Elasticsearch的入门者	双响炮
<code>spring-boot-starter-data-jpa</code>	使用Spring数据JPA和Hibernate的入门者	双响炮
<code>spring-boot-starter-data-ldap</code>	使用Spring Data LDAP的入门者	双响炮
<code>spring-boot-starter-data-mongodb</code>	入门使用MongoDB面向文档的数据库和Spring Data MongoDB	双响炮
<code>spring-boot-starter-data-mongodb-reactive</code>	入门使用MongoDB面向文档的数据库和Spring Data MongoDB Reactive	双响炮
<code>spring-boot-starter-data-neo4j</code>	初学者使用Neo4j图形数据库和Spring Data Neo4j	双响炮
<code>spring-boot-starter-data-redis</code>	使用Spring Data Redis和Lettuce客户端使用Redis键值数据存储的入门者	双响炮
<code>spring-boot-starter-data-redis-reactive</code>	初学者使用Redis键值数据存储以及Spring Data Redis反应器和Lettuce客户端	双响炮
<code>spring-boot-starter-data-rest</code>	Starter使用Spring Data REST通过REST公开Spring Data存储库	双响炮
<code>spring-boot-starter-data-solr</code>	启动Spring Data Solr使用Apache Solr搜索平台	双响炮
<code>spring-boot-starter-freemarker</code>	使用FreeMarker视图构建MVC Web应用程序的入门者	双响炮
<code>spring-boot-starter-groovy-templates</code>	使用Groovy模板视图构建MVC Web应用程序的入门者	双响炮

名称	描述	双响炮
<code>spring-boot-starter-hateoas</code>	使用Spring MVC和Spring HATEOAS构建基于超媒体的RESTful Web应用程序的入门者	双响炮
<code>spring-boot-starter-integration</code>	使用Spring Integration的入门者	双响炮
<code>spring-boot-starter-jdbc</code>	使用JDBC和HikariCP连接池的入门者	双响炮
<code>spring-boot-starter-jersey</code>	使用JAX-RS和Jersey构建RESTful Web应用程序的入门者。替代方案 spring-boot-starter-web	双响炮
<code>spring-boot-starter-jooq</code>	使用jOOQ访问SQL数据库的入门者。替代 spring-boot-starter-data-jpa 或 spring-boot-starter-jdbc	双响炮
<code>spring-boot-starter-json</code>	用于阅读和编写json的初学者	双响炮
<code>spring-boot-starter-jta-atomikos</code>	使用Atomikos启动JTA交易	双响炮
<code>spring-boot-starter-jta-bitronix</code>	使用Bitronix启动JTA交易	双响炮
<code>spring-boot-starter-jta-narayana</code>	使用Narayana进行JTA交易的首发	双响炮
<code>spring-boot-starter-mail</code>	Starter使用Java Mail和Spring Framework的电子邮件发送支持	双响炮
<code>spring-boot-starter-mustache</code>	使用Mustache视图构建Web应用程序的入门者	双响炮
<code>spring-boot-starter-quartz</code>	使用Quartz调度器的入门者	双响炮
<code>spring-boot-starter-security</code>	Starter使用Spring Security	双响炮
<code>spring-boot-starter-test</code>	Starter用于测试包含JUnit, Hamcrest和Mockito等库的Spring Boot应用程序	双响炮
<code>spring-boot-starter-thymeleaf</code>	使用Thymeleaf视图构建MVC Web应用程序的入门者	双响炮
<code>spring-boot-starter-validation</code>	通过Hibernate Validator使用Java Bean验证的入门者	双响炮
<code>spring-boot-starter-web</code>	使用Spring MVC构建Web的初学者，包括RESTful应用程序。使用Tomcat作为默认的嵌入容器	双响炮

名称	描述	双响炮
<code>spring-boot-starter-web-services</code>	使用Spring Web Services的入门者	双响炮
<code>spring-boot-starter-webflux</code>	使用Spring Framework的Reactive Web支持构建WebFlux应用程序的入门者	双响炮
<code>spring-boot-starter-websocket</code>	使用Spring Framework的WebSocket支持构建WebSocket应用程序的入门者	双响炮

除应用程序启动器外，还可以使用以下启动器来添加 [生产准备功能](#)：

表13.2。 Spring Boot生产启动器

名称	描述	双响炮
<code>spring-boot-starter-actuator</code>	Starter使用Spring Boot的执行器提供生产就绪功能，可帮助您监控和管理您的应用程序	双响炮

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

表13.3。 Spring Boot技术首发

名称	描述	双响炮
<code>spring-boot-starter-jetty</code>	将Jetty用作嵌入式servlet容器的入门者。替代方案 spring-boot-starter-tomcat	双响炮
<code>spring-boot-starter-log4j2</code>	使用Log4j2进行日志记录的入门者。替代方案 spring-boot-starter-logging	双响炮
<code>spring-boot-starter-logging</code>	启动器使用Logback进行日志记录。默认日志启动器	双响炮
<code>spring-boot-starter-reactor-netty</code>	入门使用Reactor Netty作为嵌入式反应式HTTP服务器。	双响炮
<code>spring-boot-starter-tomcat</code>	使用Tomcat作为嵌入式servlet容器的入门。默认使用的servlet容器启动器 spring-boot-starter-web	双响炮
<code>spring-boot-starter-undertow</code>	使用Undertow作为嵌入式servlet容器的入门者。替代方案 spring-boot-starter-tomcat	双响炮



有关其他社区的列表贡献首先，看 README文件中 `spring-boot-starters` GitHub上的模块。

14.构建你的代码

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

14.1 使用“默认”软件包

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



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

14.2查找主要应用程序类

我们通常建议您将主应用程序类定位在其他类上方的根包中。该 `@EnableAutoConfiguration` 注解往往放在你的主类，它隐含地定义为某些项目一基地“搜索包”。例如，如果您正在编写JPA应用程序，`@EnableAutoConfiguration` 则使用带注释的类的包 搜索 `@Entity` 项目。

使用根包也可以使用 `@ComponentScan` 注释而无需指定 `basePackage` 属性。`@SpringBootApplication` 如果您的主类位于根包中，您还可以使用注释。

下面的清单显示了一个典型的布局：

```
COM
+ - 例子
  + - 我的应用程序
    + - Application.java
    |
    + - 客户
      | + - Customer.java
      | + - CustomerController.java
      | + - CustomerService.java
      | + - CustomerRepository.java
      |
      + - 订单
        + - Order.java
        + - OrderController.java
        + - OrderService.java
        + - OrderRepository.java
```

该 `Application.java` 文件将 `main` 与基本一起声明该方法，`@Configuration` 如下所示：

```
package com.example.myapplication;

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 (应用类, 参数);
    }
}
```

15.配置类

Spring Boot支持基于Java的配置。虽然可以使用 `SpringApplication` XML来源，但我们通常建议您的主要来源是单个 `@Configuration` 类别。通常，定义该 `main` 方法的类是主要的候选人 `@Configuration`。



许多使用XML配置的互联网上发布了Spring配置示例。如果可能，请始终尝试使用等效的基于Java的配置。搜索 `Enable*` 注释可能是一个很好的起点。

15.1导入其他配置类

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

15.2 导入XML配置

如果你绝对必须使用基于XML的配置，我们建议你仍然从一个`@Configuration`类开始。然后您可以使用`@ImportResource`注释来加载XML配置文件。

16.自动配置

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

您需要选择加入`@EnableAutoConfiguration`或`@SpringBootApplication`注释到您的某个`@Configuration`类来自动配置。



你应该只添加一个`@EnableAutoConfiguration`注释。我们通常建议您将其添加到您的主`@Configuration`课程中。

16.1 逐渐更换自动配置

自动配置是非侵入式的。在任何时候，您都可以开始定义自己的配置以替换自动配置的特定部分。例如，如果您添加自己的`DataSource`Bean，则默认的嵌入式数据库支持会被取消。

如果您需要了解当前正在应用的自动配置以及为什么使用`--debug`交换机启动应用程序。这样做可以为选定的核心记录器启用调试日志，并将条件报告记录到控制台。

16.2 禁用特定的自动配置类

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

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

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

如果该类不在类路径上，则可以使用`excludeName`注释的属性并改为指定完全限定名。最后，您还可以通过使用该`spring.autoconfigure.exclude`属性来控制要排除的自动配置类的列表。



您可以在注释级别和通过使用属性来定义排除。

17.春豆和依赖注入

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

如果按上面的建议构建代码（在根包中查找应用程序类），则可以添加`@ComponentScan`任何参数。您的所有应用程序组件（的`@Component`，`@Service`，`@Repository`，`@Controller`等）自动注册为春豆。

以下示例显示了`@Service`使用构造函数注入来获取所需`RiskAssessor`bean的Bean：

```
包 com.example.service;

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

@Service
公共 类 DatabaseAccountService实现 AccountService {

    私人 最终 RiskAssessssor风险评估师;

    @Autowired
    public DatabaseAccountService (RiskAssessor riskAssessor) {
```

```

        this.riskAssessor = riskAssessor;
    }

    // ...

}

```

如果一个bean有一个构造函数，则可以省略 `@Autowired`，如以下示例所示：

```

@Service
公共类 DatabaseAccountService实现 AccountService {

    私人 最终 RiskAssessor风险评估师;

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

    // ...
}

```



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

18. 使用`@SpringBootApplication`注释

许多春季引导开发者总是有其主类注解为 `@Configuration`，`@EnableAutoConfiguration` 和 `@ComponentScan`。由于这些注释经常一起使用（特别是如果您遵循上述 [最佳实践](#)），Spring Boot提供了一种便捷的 `@SpringBootApplication` 选择。

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

```

package com.example.myapplication;

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

@SpringBootApplication //与@Configuration相同@EnableAutoConfiguration @ComponentScan
public class Application {

    public static void main (String [] args) {
        SpringApplication.run (应用类, 参数);
    }

}

```



`@SpringBootApplication` 还提供了别名定制的属性 `@EnableAutoConfiguration` 和 `@ComponentScan`。

19. 运行你的应用程序

将应用程序打包为jar并使用嵌入式HTTP服务器的最大优点之一是，您可以像运行其他应用程序一样运行应用程序。调试Spring Boot应用程序也很容易。您不需要任何特殊的IDE插件或扩展。



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

19.1 从IDE运行

您可以从IDE运行Spring Boot应用程序作为简单的Java应用程序。但是，您首先需要导入您的项目。导入步骤取决于您的IDE和构建系统。大多数IDE可以直接导入Maven项目。例如，Eclipse用户可以从菜单中选择 `Import... → Existing Maven Projects File`

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



如果您意外运行了两次Web应用程序，则会看到“端口已被使用”错误。STS用户可以使用 Relaunch 按钮而不是 Run 按钮来确保任何现有的实例关闭。

19.2作为打包应用程序运行

如果您使用Spring Boot Maven或Gradle插件创建可执行jar，则可以使用 `java -jar` 以下示例所示运行应用程序：

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

也可以在启用远程调试支持的情况下运行打包的应用程序。这样做可以让您将调试器附加到打包的应用程序，如以下示例所示：

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

19.3使用Maven插件

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

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

您可能还想使用 `MAVEN_OPTS` 操作系统环境变量，如以下示例中所示：

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

19.4使用Gradle插件

Spring Boot Gradle插件还包含一个 `bootRun` 可用于以分解形式运行应用程序的任务。的 `bootRun`，只要你申请的任务添加 `org.springframework.boot` 和 `java` 插件，并在下面的示例所示：

```
$ gradle bootRun
```

您可能还想使用 `JAVA_OPTS` 操作系统环境变量，如以下示例中所示：

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

19.5热插拔

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

该 `spring-boot-devtools` 模块还包含对快速重新启动应用程序的支持。有关详细信息，请参阅本章后面的第20章开发人员工具部分以及 热插拔“操作方法”。

20.开发人员工具

Spring Boot包含一组额外的工具，可以使应用程序开发体验更愉快。该 `spring-boot-devtools` 模块可以包含在任何项目中以提供额外的开发时间功能。要包含devtools支持，请将模块依赖项添加到您的构建中，如Maven和Gradle的以下列表所示：

Maven的。

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

摇篮。

```
依赖关系{
    编译（“org.springframework.boot: spring-boot-devtools”）
```

}



运行完整打包的应用程序时，开发人员工具会自动禁用。如果你的应用程序是 `java -jar` 从一个特殊的类加载器启动的，或者它是一个特殊的类加载器启动的，那么它就被认为是“生产应用程序”。将依赖标记为可选项是一种最佳实践，可防止 `devtools` 被传递应用到其他使用您项目的模块。`Gradle` 不支持 `optional` 开箱即用的依赖关系，因此您可能需要查看一下 `propdeps-plugin`。



重新打包的档案在默认情况下不包含 `devtools`。如果你想使用某个远程 `devtools` 功能，你需要禁用 `excludeDevtools` `build` 属性来包含它。该属性支持 `Maven` 和 `Gradle` 插件。

20.1 属性默认值

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

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

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



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

20.2 自动重启

`spring-boot-devtools` 当类路径上的文件发生更改时，使用的应用程序会自动重新启动。在 IDE 中工作时，这可能是一个有用的功能，因为它为代码更改提供了非常快速的反馈循环。默认情况下，监视指向文件夹的类路径中的任何条目以进行更改。请注意，某些资源（如静态资产和视图模板）**不需要重新启动应用程序**。

触发重启

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



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



与 `LiveReload` 一起使用时，自动重启的效果非常好。有关详细信息，请参阅 `LiveReload` 部分。如果您使用 `JRebel`，则会禁用自动重新启动，以支持动态类重新加载。其他 `devtools` 功能（例如 `LiveReload` 和 属性覆盖）仍然可以使用。



`DevTools` 依靠应用程序上下文的关闭挂钩在重新启动期间关闭它。如果您禁用了关闭挂钩（ `SpringApplication.setRegisterShutdownHook(false)`），它将无法正常工作。



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



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

重新启动 vs 重新加载

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

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

20.2.1记录条件评估中的变化

默认情况下，每次应用程序重新启动时，都会记录显示条件评估增量的报告。该报告显示了在您进行更改（如添加或删除Bean以及设置配置属性）时对应用程序自动配置的更改。

要禁用报告的日志记录，请设置以下属性：

```
spring.devtools.restart.log 条件评价-Δ=假
```

20.2.2排除资源

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

```
spring.devtools.restart.exclude =静态/ **, 公共/ **
```



如果您想保留这些默认设置并添加其他排除项，请改用该 `spring.devtools.restart.additional-exclude` 属性。

20.2.3观察其他路径

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

20.2.4禁用重启

如果您不想使用重新启动功能，则可以使用该 `spring.devtools.restart.enabled` 属性将其禁用。在大多数情况下，你可以在你的这个属性中设置 `application.properties`（这样做仍然可以初始化重启类加载器，但它不会监视文件的变化）。

如果您需要完全禁用重新启动支持（例如，因为它不适用于特定的库），则需要在调用之前将 `spring.devtools.restart.enabled` `System` 属性设置为，如下例所示：`false` `SpringApplication.run(...)`

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

20.2.5使用触发文件

如果您使用持续编译更改文件的IDE，则可能只希望在特定时间触发重新启动。为此，您可以使用“触发文件”，这是一个特殊的文件，当您想要实际触发重新启动检查时必须对其进行修改。只更改文件会触发检查，只有在Devtools检测到必须执行某些操作时才会重新启动。触发文件可以手动更新或使用IDE插件更新。

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



您可能希望将其设置 `spring.devtools.restart.trigger-file` 为 全局设置，以便所有项目的行为方式都相同。

20.2.6自定义重启类加载器

如前面的“[重新启动vs重新加载](#)”部分所述，重新启动功能通过使用两个类加载器来实现。对于大多数应用程序，这种方法运作良好 但是，它有时会导致类加载问题。

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

该 `spring-devtools.properties` 文件可以包含前缀属性 `restart.exclude` 和 `restart.include`。这些 `include` 元素是应该被拉入“重新启动”类加载器 `exclude` 中的项目，元素是应该下推到“基本”类加载器中的项目。该属性的值是应用于类路径的正则表达式模式，如以下示例所示：

```
restart.exclude.companycommonlibs = / MyCorp的共用- [\瓦特- ]。+ \罐子
restart.include.projectcommon = / MyCorp的-的Myproj - [\瓦特- ]。+ \罐
```



所有属性键必须是唯一的。只要财产始于 `restart.include.` 或被 `restart.exclude.` 考虑。



所有 `META-INF/spring-devtools.properties` 来自 classpath 的都被加载。您可以将文件打包到您的项目中，也可以打包到项目使用的库中。

20.2.7 已知限制

对于使用标准进行反序列化的对象，重新启动功能无法正常工作 `ObjectInputStream`。如果你需要反序列化数据，你可能需要 `ConfigurableObjectInputStream` 结合 Spring 使用 `Thread.currentThread().getContextClassLoader()`。

不幸的是，有些第三方库反序列化而没有考虑上下文类加载器。如果您发现这样的问题，您需要向原作者请求修复。

20.3 LiveReload

该 `spring-boot-devtools` 模块包含一个嵌入式 LiveReload 服务器，可用于在资源发生更改时触发浏览器刷新。LiveReload 浏览器扩展可免费用于 livereload.com 的 Chrome，Firefox 和 Safari。

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



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

20.4 全局设置

您可以通过添加一个文件名为配置全局 devtools 设置 `.spring-boot-devtools.properties` 你的 `$HOME` 文件夹（注意：文件名开头“`.`”）。添加到此文件的任何属性都适用于使用 devtools 的计算机上的所有 Spring Boot 应用程序。例如，要将重新启动配置为始终使用 触发器文件，您可以添加以下属性：

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

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

20.5 远程应用程序

Spring Boot 开发人员工具不限于本地开发。远程运行应用程序时，您还可以使用多个功能。远程支持是选择加入。要启用它，您需要确保 `devtools` 包含在重新打包的存档中，如下所示：

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

然后你需要设置一个 `spring.devtools.remote.secret` 属性，如下例所示：

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



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

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

20.5.1 运行远程客户端应用程序

远程客户端应用程序旨在从您的IDE中运行。您需要使用与您连接的远程项目相同的类路径运行。`org.springframework.boot.devtools.RemoteSpringApplication` 应用程序的单个必需参数是它所连接的远程URL。

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

- 选择 `Run Configurations...` 从 `Run` 菜单。
 - 创建一个新的 `Java Application` “启动配置”。
 - 浏览 `my-app` 项目。
 - 使用 `org.springframework.boot.devtools.RemoteSpringApplication` 作为主类。
 - 添加 <https://myapp.cfapps.io> 到 `Program arguments` (或者任何你的远程URL)。

正在运行的远程客户端可能类似于以下列表：

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

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

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

20.5.2 远程更新

远程客户端以与本地重启相同的方式监视应用程序类路径的更改。任何更新的资源都会被推送到远程应用程序，并且（如果需要）会触发重新启动。如果您对使用本地没有的云服务的功能进行迭代，这会很有帮助。通常，远程更新和重新启动比完整的重建和部署周期快得多。

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

21. 包装您的生产申请

可执行的罐子可用于生产部署，由于它们是独立的，它们也非常适合基于云的部署。

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

22. 下一步阅读什么

您现在应该了解如何使用Spring Boot以及您应遵循的一些最佳实践。您现在可以继续深入了解特定的 [Spring Boot功能](#)，或者可以跳过并阅读 Spring Boot 的“[生产准备](#)”部分。

第四部分。Spring Boot功能

本节将介绍Spring Boot的细节。在这里，您可以了解您可能想要使用和定制的关键功能。如果您还没有这样做，您可能需要阅读“第II部分”，入门指南“”和“第III部分”，使用Spring Boot“”部分，以便您具备良好的基础知识。

23. SpringApplication

本 `SpringApplication` 类提供了一个方便的方式来引导该从开始Spring应用程序 `main()` 的方法。在许多情况下，您可以委派给静态 `SpringApplication.run` 方法，如以下示例所示：

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

当您的应用程序启动时，您应该看到类似于以下输出的内容：

```
.
.
.
:: Spring Boot :: v2.0.1.BUILD-SNAPSHOT
```

```
2013-07-31 00: 08: 16.117 INFO 56603 --- [main] osbsapp.SampleApplication: 在我的电脑上使用PID 56603启动SampleApplication v0
2013-07-31 00:08:16.166 INFO 56603 --- [           main] actionConfigServletWebServerApplicationContext : Refreshing org.s
2014-03-04 13:09:54.912 INFO 41370 --- [           main] .t.TomcatServletWebServerFactory : Server initialized with port:
2014-03-04 13:09:56.501 INFO 41370 --- [           main] o.s.b.s.app.SampleApplication         : Started SampleApplica
```

默认情况下，`INFO` 会显示日志消息，其中包括一些相关的启动详细信息，例如启动应用程序的用户。如果您需要日志级别以外的其他日志级别 `INFO`，则可以按照第26.4节“日志级别”中所述进行设置。

23.1启动失败

如果您的应用程序无法启动，注册 `FailureAnalyzers` 有机会提供专门的错误消息和具体操作来解决问题。例如，如果您在端口上启动Web应用程序 `8080` 并且该端口已被使用，则应该看到与以下消息类似的内容：

```
*****
应用程序无法启动
*****
```

描述：

嵌入式servlet容器无法启动。端口8080已被使用。

行动：

识别并停止在端口8080上侦听的进程或将此应用程序配置为侦听另一个端口。



Spring Boot提供了许多 `FailureAnalyzer` 实现，您可以 [添加自己的](#)。

如果没有故障分析仪能够处理异常情况，您仍然可以显示完整的情况报告以更好地了解问题所在。为此，您需要 [启用该 `debug` 属性或启用 `DEBUG` 日志记录功能 `org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener`](#)。

例如，如果您使用的是运行应用程序 `java -jar`，则可以 `debug` 按如下方式启用该属性：

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

23.2自定义横幅

启动时打印的横幅可以通过将 `banner.txt` 文件添加到类路径中或通过将该 `spring.banner.location` 属性设置为此类文件的位置来更改。如果文件的编码不是UTF-8，可以设置 `spring.banner.charset`。除了一个文本文件，你还可以添加一个 `banner.gif`，`banner.jpg`

或 `banner.png` 图像文件到类路径或设置 `spring.banner.image.location` 属性。图像被转换成ASCII艺术表现形式并打印在任何文字横幅上方。

在 `banner.txt` 文件内部，您可以使用以下任何占位符：

表23.1。横幅变量

变量	描述
<code> \${application.version} </code>	您的应用程序的版本号，请参阅 <code>MANIFEST.MF</code> 。例如， <code>Implementation-Version: 1.0</code> 打印为 <code>1.0</code> 。
<code> \${application.formatted-version} </code>	您的应用程序的版本号，在 <code>MANIFEST.MF</code> 显示器中声明并格式化（括在括号中并以前缀 <code>v</code> ）。例如 <code>(v1.0)</code> 。
<code> \${spring-boot.version} </code>	您正在使用的Spring Boot版本。例如 <code>2.0.1.BUILD-SNAPSHOT</code> 。
<code> \${spring-boot.formatted-version} </code>	您正在使用的Spring Boot版本，已格式化显示（用括号括起来并加上前缀 <code>v</code> ）。例如 <code>(v2.0.1.BUILD-SNAPSHOT)</code> 。
<code> \${Ansi.NAME} </code> (或 <code> \${AnsiColor.NAME} </code> , <code> \${AnsiBackground.NAME} </code> , <code> \${AnsiStyle.NAME} </code>)	<code>NAME</code> ANSI转义代码的名称在哪里？ 详情请参阅 <code>AnsiPropertySource</code> 。
<code> \${application.title} </code>	您的申请的标题，如申报 <code>MANIFEST.MF</code> 。例如 <code>Implementation-Title: MyApp</code> 打印为 <code>MyApp</code> 。



`SpringApplication.setBanner(...)` 如果您想以编程方式生成横幅，则可以使用该方法。使用该 `org.springframework.boot.Banner` 接口并实现您自己的 `printBanner()` 方法。

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

打印的横幅注册为下以下名称的单例的bean：`springBootBanner`。



YAML映射 `off` 到 `false`，因此如果要禁用应用程序中的横幅，请务必添加引号，如以下示例所示：

春季：

主要：

横幅模式：“关闭”

23.3自定义SpringApplication

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

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



传递给构造函数的参数 `SpringApplication` 是Spring bean的配置源。在大多数情况下，这些都是对 `@Configuration` 类的引用，但它们也可能是指向XML配置的引用或应该扫描的包。

也可以 `SpringApplication` 使用 `application.properties` 文件进行配置。有关详细信息，请参阅第24章，外部化配置。

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

23.4 Fluent Builder API

如果您需要构建 `ApplicationContext` 层次结构（具有父子关系的多个上下文），或者如果您更愿意使用“流利”构建器API，则可以使用 `SpringApplicationBuilder`。

在 `SpringApplicationBuilder` 让要链接的多个方法调用，并且包括 `parent` 和 `child` 方法，让你创建层次结构，以显示在下面的例子：

```
新的 SpringApplicationBuilder ()  
    .sources (父类)  
    .child (应用程序类)  
    .bannerMode (Banner.Mode.OFF)  
    .RUN (参数) ;
```



创建 `ApplicationContext` 层次结构时有一些限制。例如，Web组件必须包含在子上下文中，并且 `Environment` 父组件和子上下文都使用相同的组件。查看 `SpringApplicationBuilder` Javadoc 获取完整的细节。

23.5 应用程序事件和监听器

除了通常的Spring框架的事件，比如 `ContextRefreshedEvent`，一个 `SpringApplication` 发送一些附加的应用程序事件。



有些事件实际上 `ApplicationContext` 是在创建之前触发的，所以你不能在这些事件上注册一个监听器 `@Bean`。你可以使用 `SpringApplication.addListeners(...)` 方法或 `SpringApplicationBuilder.listeners(...)` 方法注册它们。

如果您希望自动注册这些侦听器，而不管创建应用的方式如何，则可以将 `META-INF/spring.factories` 文件添加到项目并使用该 `org.springframework.context.ApplicationListener` 键引用侦听器，如下例所示：

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

随着您的应用程序运行，应用程序事件按以下顺序发送：

1. An `ApplicationStartingEvent` 在运行开始时但在任何处理之前发送，除了注册监听器和初始化器之外。
2. `ApplicationEnvironmentPreparedEvent` 当 `Environment` 在上下文中使用时，但在创建上下文之前发送An。
3. `ApplicationPreparedEvent` 在刷新开始之前但在bean定义加载之后发送一个。
4. `ApplicationStartedEvent` 在上下文刷新之后但在任何应用程序和命令行参数被调用之前发送An。
5. `ApplicationReadyEvent` 在任何应用程序和命令行运行程序被调用后发送An。它表示应用程序已准备好为请求提供服务。
6. `ApplicationFailedEvent` 如果启动时发生异常，则发送An。



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

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

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

23.6 Web环境

A会 `SpringApplication` 尝试以 `ApplicationContext` 您的名义创建正确的类型。用于确定a的算法 `WebEnvironmentType` 非常简单：

- 如果存在Spring MVC，`AnnotationConfigServletWebServerApplicationContext` 则使用一个
- 如果Spring MVC不存在并且Spring WebFlux存在，`AnnotationConfigReactiveWebApplicationContext` 则使用一个
- 否则，`AnnotationConfigApplicationContext` 使用

这意味着如果您 `WebClient` 在同一应用程序中使用 Spring MVC 和 Spring WebFlux 中的新功能，则默认情况下会使用 Spring MVC。您可以通过调用轻松地覆盖它 `setWebApplicationType(WebApplicationType)`。

也可以完全控制 `ApplicationContext` 呼叫使用的类型 `setApplicationContextClass(...)`。



在JUnit测试中 `setWebApplicationType(WebApplicationType.NONE)` 使用时经常需要调用 `SpringApplication`。

23.7 访问应用程序参数

如果您需要访问传递给的应用程序参数，则 `SpringApplication.run(...)` 可以注入一个 `org.springframework.boot.ApplicationArguments` bean。该 `ApplicationArguments` 接口提供对原始 `String[]` 参数以及解析 `option` 和 `non-option` 参数的访问，如以下示例所示：

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

@Component
public class MyBean {

    @Autowired
    public MyBean (ApplicationArguments args) {
        boolean debug = args.containsOption ( "debug" );
        List <String> files = args.getNonOptionArgs () ;
        //如果用“--debug logfile.txt”debug = true, files = ["logfile.txt"]运行
    }

}
```



Spring Boot还 `CommandLinePropertySource` 向Spring 注册了一个 `Environment`。这使您可以通过使用 `@Value` 注释来注入单个应用程序参数。

23.8 使用 ApplicationRunner 或 CommandLineRunner

如果您需要在启动后运行某些特定的代码 `SpringApplication`，则可以实现 `ApplicationRunner` 或 `CommandLineRunner` 接口。两个接口都以相同的方式工作，并提供一种 `run` 方法，即在 `SpringApplication.run(...)` 完成之前调用。

这些 `CommandLineRunner` 接口作为一个简单的字符串数组提供对应用程序参数的访问，而 `ApplicationRunner` 使用 `ApplicationArguments` 前面讨论的接口。下面的例子显示了 `CommandLineRunner` 一个 `run` 方法：

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

@Component
公共类 MyBean实现了 CommandLineRunner {

    公共 无效运行 (字符串...参数) {
        //做些什么...
    }

}
```

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

23.9 申请退出

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

另外，`org.springframework.boot.ExitCodeGenerator` 如果bean 在 `SpringApplication.exit()` 调用时希望返回特定的退出代码，它们可以实现该接口。然后可以将此退出代码传递给 `System.exit()` 状态代码，如以下示例所示：

```
@SpringBootApplication
public class ExitCodeApplication {

    @Bean
    public ExitCodeGenerator exitCodeGenerator () {
        return () -> 42 ;
    }

}
```

```

public static void main (String [] args) {
    System.exit (SpringApplication
        .exit (SpringApplication.run (ExitCodeApplication 类, 参数)) );
}
}

```

而且，`ExitCodeGenerator` 界面可以通过例外来实现。遇到这样的异常时，Spring Boot将返回由实现的`getExitCode()`方法提供的退出代码。

23.10管理功能

通过指定`spring.application.admin.enabled`属性可以为应用程序启用与管理相关的功能。这暴露`SpringApplicationAdminMXBean`了平台上`MBeanServer`。您可以使用此功能远程管理您的Spring Boot应用程序。此功能对于任何服务包装器实现也可能有用。



如果您想知道应用程序在哪个HTTP端口上运行，请使用键来获取属性`local.server.port`。



警告

启用此功能时要小心，因为MBean公开了关闭应用程序的方法。

24.外部化配置

Spring Boot允许您将配置外部化，以便您可以在不同环境中使用相同的应用程序代码。您可以使用属性文件，YAML文件，环境变量和命令行参数来外部化配置。属性值可以通过直接注射到你的bean `@Value`注释，通过Spring的访问`Environment`抽象，或者被绑定到结构化对象通过`@ConfigurationProperties`。

Spring Boot使用非常特定的`PropertySource`顺序，旨在允许明智的重写值。属性按以下顺序考虑：

1. 在主目录上开发Devtools全局设置属性 (`~/.spring-boot-devtools.properties`当devtools处于活动状态时)。
2. `@TestPropertySource` 您的测试中的注释。
3. `@SpringBootTest#properties` 您的测试中的注释属性。
4. 命令行参数。
5. 从属性`SPRING_APPLICATION_JSON` (嵌入在环境变量或系统属性直列JSON)。
6. `ServletConfig` 初始化参数。
7. `ServletContext` 初始化参数。
8. 来自JNDI的属性`java:comp/env`。
9. Java系统属性 (`System.getProperties()`)。
10. OS环境变量。
11. 一个`RandomValuePropertySource`只有属性的`random.*`。
12. 打包jar (`application-{profile}.properties`和YAML变体)之外的特定于配置文件的应用程序属性。
13. 打包在您的jar (`application-{profile}.properties`和YAML变体)中的特定于配置文件的应用程序属性。
14. 应用程序属性在打包jar (`application.properties`和YAML变体)之外。
15. 打包在jar中的应用程序属性 (`application.properties`以及YAML变体)。
16. `@PropertySource` 您的`@Configuration`课程注释。
17. 默认属性 (由设置指定`SpringApplication.setDefaultProperties()`)。

为了提供一个具体的例子，假设你开发一个`@Component`使用`name`属性的属性，如下例所示：

```

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

@Component
public class MyBean {

    @Value ("${name}")
    私人字符串名称;

    // ...
}

```

在您的应用程序类路径中（例如，在您的jar `application.properties`文件中），您可以有一个文件提供一个合理的默认属性值`name`。在新环境中运行时，`application.properties`可以在您的jar外部提供一个文件来覆盖该文件`name`。对于一次性测试，您可以使用特定的命令行开

关(例如，`java -jar app.jar --name="Spring"`)启动。



这些`SPRING_APPLICATION_JSON`属性可以通过环境变量在命令行中提供。例如，您可以在UN*X shell中使用以下行：

```
$ SPRING_APPLICATION_JSON='{"acme": {"name": "test"}}'java -jar myapp.jar
```

在前面的例子中，你最终`acme.name=test`在Spring中`Environment`。您还可以像`spring.application.json`在System属性中一样提供JSON，如以下示例中所示：

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

您还可以使用命令行参数提供JSON，如以下示例所示：

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

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

24.1配置随机值

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

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

该`random.int*`语法是`OPEN value (,max) CLOSE`其中的`OPEN,CLOSE`任何字符和`value,max`是整数。如果`max`提供，则`value`是最小值并且`max`是最大值(不包括)。

24.2访问命令行属性

默认情况下，`SpringApplication`将任何命令行选项参数(即，从参数`--`，如`--server.port=9000`)的`property`，并将它们添加到章节`Environment`。如前所述，命令行属性始终优先于其他属性源。

如果您不希望命令行属性被添加到`Environment`，您可以通过使用禁用它们`SpringApplication.setAddCommandLineProperties(false)`。

24.3应用程序属性文件

`SpringApplication`从`application.properties`以下位置的文件加载属性并将它们添加到Spring中`Environment`：

1. 一个`/config`当前目录的子目录
2. 当前目录
3. 一个类路径`/config`包
4. 类路径根

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



您也可以使用YAML(`'.yml'`)文件替代`'properties'`。

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

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

以下示例显示如何指定两个位置：

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



`spring.config.name`并且`spring.config.location`很早就用于确定哪些文件必须加载，因此它们必须定义为环境属性(通常是在OS环境变量，系统属性或命令行参数)。

如果 `spring.config.location` 包含目录（而不是文件），它们应该结束 /（并在运行时加入从 `spring.config.name` 加载之前生成的名称，包括配置文件特定的文件名）。指定的文件 `spring.config.location` 按原样使用，不支持特定于配置文件的变体，并且被特定于配置文件的特性覆盖。

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

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

通过使用配置自定义配置位置时 `spring.config.location`，它们会替换默认位置。例如，如果 `spring.config.location` 使用该值配置 `classpath:/custom-config/,file:./custom-config/`，则搜索顺序变为以下内容：

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

或者，使用自定义配置位置进行配置时 `spring.config.additional-location`，除了默认位置以外，还会使用它们。在默认位置之前搜索其他位置。例如，如果 `classpath:/custom-config/,file:./custom-config/` 配置了其他位置，则搜索顺序如下所示：

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

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



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



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

24.4 配置文件特定的属性

除 `application.properties` 文件外，还可以使用以下命名约定来定义特定于配置文件的属性：`application-{profile}.properties`。在 `Environment` 具有一组默认的配置文件（默认 `[default]`）如果没有活动的简档设置中使用。换句话说，如果没有显式激活配置文件，`application-default.properties` 则会加载属性。

特定于配置文件的属性从标准的相同位置加载 `application.properties`，特定于配置文件的文件总是覆盖非特定的文件，而不管配置文件特定的文件是否位于打包的jar内部或外部。

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



如果您已指定了任何文件 `spring.config.location`，则不考虑这些文件的特定文件变体。`spring.config.location` 如果您还想使用特定于配置文件的属性，请使用目录。

属性中有24.5个占位符

这些值在使用时会 `application.properties` 通过现有值进行过滤 `Environment`，因此您可以返回以前定义的值（例如，从系统属性中）。

```
app.name = MyApp
app.description = ${app.name}是一个Spring Boot应用程序
```



您也可以使用这种技术来创建现有Spring Boot属性的“简短”变体。有关详细信息，请参见第74.4节“使用简短命令行参数”。

24.6 使用YAML而不是属性

YAML是JSON的超集，因此是用于指定分层配置数据的便利格式。该 `SpringApplication` 级自动支持YAML来替代，只要你有属性 `SnakeYAML` 在classpath库。



如果你使用“Starters”，`SnakeYAML`会自动提供 `spring-boot-starter`。

24.6.1 加载YAML

Spring框架提供了两个方便的类，可以用来加载YAML文档。该 `YamlPropertiesFactoryBean` 负载YAML作为 `Properties` 和 `YamlMapFactoryBean` 负载YAML作为 `Map`。

例如，请考虑以下YAML文档：

```
环境:
  dev:
    url: http://dev.example.com
    名称: Developer Setup
  prod:
    url: http://another.example.com
    name: My Cool App
```

前面的示例将转换为以下属性：

```
environments.dev.url = http://dev.example.com
environments.dev.name = 开发人员设置
environments.prod.url = http://another.example.com
environments.prod.name = 我的酷应用程序
```

YAML列表被表示为带 `[index]` 解引用器的属性键。例如，请考虑以下YAML：

```
我:
  服务器:
    - dev.example.com
    - another.example.com
```

前面的例子将被转换成这些属性：

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

要通过使用Spring `DataBinder` 实用程序来绑定属性（这就是做什么的 `@ConfigurationProperties`），您需要在类型为 `java.util.List`（或 `Set`）的目标bean中拥有一个属性，并且您需要提供setter或使用可变值初始化它。例如，以下示例绑定到以前显示的属性：

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

  私人列表<String> servers = new ArrayList <String>();

  公共列表<字符串> getServers () {
    返回 这个 .servers;
  }
}
```



当列表配置在多个地方时，通过替换整个列表来覆盖作品。在前面的示例中，当 `my.servers` 在几个地方定义时，`PropertySource` 具有较高优先级的整个列表将覆盖该列表的任何其他配置。逗号分隔列表和YAML列表都可用于完全覆盖列表的内容。

24.6.2 在Spring环境中将YAML作为属性公开

本 `YamlPropertySourceLoader` 类可用于暴露YAML作为 `PropertySource` 在特定 `Environment`。这样做可让您使用 `@Value` 带占位符语法的注释来访问YAML属性。

24.6.3多配置文件YAML文件

您可以在一个文件中指定多个特定于配置文件的YAML文档，方法是使用一个 `spring.profiles` 键来指示该文档何时适用，如以下示例所示：

```
服务器:
  地址: 192.168.1.100
---
spring:
  profiles: 开发
服务器:
  地址: 127.0.0.1
---
弹簧:
  配置文件: 生产
服务器:
  地址: 192.168.1.120
```

在前面的示例中，如果 `development` 配置文件处于活动状态，则 `server.address` 属性为 `127.0.0.1`。同样，如果 `production` 配置文件处于活动状态，则 `server.address` 属性为 `192.168.1.120`。如果 `development` 和 `production` 配置文件未启用，则该属性的值为 `192.168.1.100`。

如果应用程序上下文启动时没有显式激活，则激活默认配置文件。因此，在下面的YAML中，我们为其设置了一个值，仅在“默认”配置文件中 `spring.security.user.password` 可用：

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

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

```
服务器:
  端口: 8000
弹簧:
  安全:
    用户:
      密码: 弱
```

通过使用该 `spring.profiles` 元素指定的弹簧配置文件可以可选地通过使用该 `!` 字符来取消。如果为单个文档指定了否定配置文件和非否定配置文件，则至少有一个非否定配置文件必须匹配，且不存在否定配置文件可能匹配。

24.6.4 YAML的缺点

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

24.6.5合并YAML列表

正如我们前面所展示的，任何YAML内容最终都会转换为属性。当通过配置文件覆盖“列表”属性时，该过程可能不直观。

例如，假定默认情况下 `MyPojo` 具有 `name` 和 `description` 属性的对象 `null`。以下示例显示 `MyPojo` 来自以下对象的列表 `AcmeProperties`：

```
@ConfigurationProperties ("acme")
公共类 AcmeProperties {
    private final List <MyPojo> list = new ArrayList <> () ;
    public List <MyPojo> getList () {
        return this.list;
    }
}
```

考虑以下配置：

```
acme:
  list:
```

```

- name: my name
  description: my description
---

spring:
  profiles: dev
acme:
  list:
    - name: my another name

```

如果 `dev` 配置文件未激活，则 `AcmeProperties.list` 包含一个 `MyPojo` 条目，如前所述。`dev` 但是，如果配置文件已启用，则 `list` 仍只包含一个条目（名称 `my another name` 和描述 `null`）。此配置不会将第二个 `MyPojo` 实例添加到列表中，也不会合并这些项目。

当在多个配置文件中指定一个集合时，将使用具有最高优先级（仅限那个）的集合。考虑下面的例子：

```

acme:
  list:
    - name: my name
      description: my description
    - name: another name
      description: another description
---

spring:
  profiles: dev
acme:
  list:
    - name: my another name

```

在前面的示例中，如果 `dev` 配置文件处于活动状态，则 `AcmeProperties.list` 包含一个 `MyPojo` 条目（名称 `my another name` 和描述 `null`）。

24.7类型安全的配置属性

使用 `@Value("${property}")` 注释来注入配置属性有时会很麻烦，特别是如果您正在处理多个属性或者您的数据本质上是分层的。Spring Boot 提供了另一种使用属性的方法，可以让强类型bean管理和验证应用程序的配置，如以下示例所示：

```

package com.example;

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

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

@ConfigurationProperties ("acme")
公共类 AcmeProperties {

    私有 布尔启用;

    私人 InetAddress remoteAddress;

    私人 最终安全安全= 新安全 () ;

    public boolean isEnabled () {...}

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

    public InetAddress getRemoteAddress () {...}

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

    public Security getSecurity () {...}

    公共 静态 类安全{

        私人字符串用户名

        私人字符串密码;

        私人列表<String> roles = new ArrayList <> (Collections.singleton ("USER")) ;
    }
}

```

```

public String getUsername () {...}

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

public String getPassword () {...}

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

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

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

}

}

```

前面的POJO定义了以下属性：

- `acme.enabled`，`false`默认值为。
- `acme.remote-address`，可以强制的类型`String`。
- `acme.security.username`，一个嵌套的“安全”对象，其名称由属性的名称决定。特别是，返回类型根本没有被使用了`SecurityProperties`。
- `acme.security.password`。
- `acme.security.roles`，收集了一些`String`。

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

- 只要它们被初始化，Maps就需要一个getter，但不一定是setter，因为它们可以通过绑定器进行变异。
- 集合和数组可以通过索引（通常使用YAML）或使用单个逗号分隔值（属性）来访问。在后一种情况下，制定者是强制性的。我们建议始终为这些类型添加setter。如果初始化一个集合，确保它不是不可变的（如上例）。
- 如果嵌套的POJO属性被初始化（如前例中的`Security`字段），则不需要setter。如果您希望活页夹通过使用其默认构造函数即时创建实例，则需要一个setter。

有些人使用Project Lombok来自动添加getter和setter。确保Lombok不会为这种类型生成任何特定的构造函数，因为它被容器自动使用来实例化对象。



另请参阅 `@Value` 和 `@ConfigurationProperties` 的区别。

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

```

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

```



当以`@ConfigurationProperties`这种方式注册bean时，该bean具有常规名称：`<prefix>-<fqn>`，其中`<prefix>`是在`@ConfigurationProperties`注释中指定的环境键前缀，并且`<fqn>`是bean的完全限定名称。如果注释没有提供任何前缀，则只使用bean的完全限定名称。

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

即使前面的配置创建了一个常规bean`AcmeProperties`，我们也建议`@ConfigurationProperties`只处理环境，特别是不要从上下文中注入其他bean。话虽如此，`@EnableConfigurationProperties`注释也会自动应用到您的项目中，以便通过配置注释的任何现有bean都已`@ConfigurationProperties`配置`Environment`。你可以`MyConfiguration`通过确认`AcmeProperties`已经是一个bean的捷径，如下例所示：

```

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

    // ... 参见前面的例子

}

```

这种配置方式对于`SpringApplication`外部YAML配置特别有效，如下例所示：

```
#application.yml
```

```

acme:
  remote-address: 192.168.1.1
  安全性:
    用户名: admin
    角色:
      - 用户
      - 管理员

# 根据需要额外配置

```

要使用 `@ConfigurationProperties` bean，可以像使用其他bean一样注入它们，如下例所示：

```

@Service
公共类 MyService {
  私人 最终的 AcmeProperties属性;

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

  // ...

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

```



使用 `@ConfigurationProperties` 还可以让您生成元数据文件，IDE可以使用这些元数据文件为自己的密钥提供自动完成功能。有关详细信息，请参阅 [附录B“配置元数据”附录](#)。

24.7.1第三方配置

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

要从 `Environment` 属性配置一个bean，添加 `@ConfigurationProperties` 到它的bean注册中，如下例所示：

```

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

```

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

24.7.2轻松绑定

Spring Boot使用一些宽松的规则来绑定bean的 `Environment` 属性 `@ConfigurationProperties`，所以不需要在 `Environment` 属性名称和bean属性名称之间完全匹配。常见的例子，这是有用的包括破折号分隔的环境属性（例如，`context-path` 绑定到 `contextPath`）和大写的环境属性（例如，`PORT` 绑定到 `port`）。

例如，请考虑以下 `@ConfigurationProperties` 课程：

```

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

  私人字符串firstName;

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

  public void setFirstName (String firstName) {

```

```

        this.firstName = firstName;
    }

}

```

在前面的示例中，可以使用以下属性名称：

表24.1。松绑定

属性	注意
acme.my-project.person.firstName	标准骆驼大小写语法。
acme.my-project.person.first-name	烤肉串的情况下，建议用于 <code>.properties</code> 和 <code>.yml</code> 文件。
acme.my-project.person.first_name	下划线表示法，这是用于 <code>.properties</code> 和 <code>.yml</code> 文件的替代格式。
ACME_MYPROJECT_PERSON_FIRSTNAME	大写格式，使用系统环境变量时建议使用。

 的`prefix`用于注释值必须是在串的情况下（小写和通过分离`-`，例如`acme.my-project.person`）。

表24.2。放宽每个财产来源的绑定规则

财产来源	简单	名单
属性文件	骆驼案，烤肉串案或下划线标记	使用 <code>[]</code> 或逗号分隔值的标准列表语法
YAML文件	骆驼案，烤肉串案或下划线标记	标准YAML列表语法或逗号分隔值
环境变量	大写格式，下划线为分隔符。 <code>_</code> 不应该在属性名称中使用	由下划线包围的数字值，例如 <code>MY_ACME_1_OTHER = my.acme[1].other</code>
系统属性	骆驼案，烤肉串案或下划线标记	使用 <code>[]</code> 或逗号分隔值的标准列表语法

 我们建议，如有可能，属性以小写的烤肉串格式存储，例如`my.property-name=acme`。

24.7.3属性转换

Spring绑定到`@ConfigurationProperties`bean时，会尝试将外部应用程序属性强制转换为正确的类型。如果您需要自定义类型转换，您可以提供一个`ConversionService`bean（带有一个名为bean`conversionService`）或定制属性编辑器（通过一个`CustomEditorConfigurer`bean）或定制`Converters`（带有注释为的bean定义`@ConfigurationPropertiesBinding`）。

 由于此bean在应用程序生命周期中很早被请求，因此请确保限制您`ConversionService`正在使用的依赖关系。通常，您需要的任何依赖项可能在创建时未完全初始化。`ConversionService`如果不配置密钥强制转换，并且只依赖合格的自定义转换器，则可能需要重命名自定义`@ConfigurationPropertiesBinding`。

转换持续时间

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

- 常规`long`表示（除非`@DefaultUnit`指定了a，则使用毫秒作为默认单位）
- 标准ISO-8601格式使用`java.util.Duration`
- 数值和单位耦合的更可读格式（例如，`10s`意味着10秒）

考虑下面的例子：

```

@ConfigurationProperties ("app.system")
公共类 AppSystemProperties {

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

    private Duration readTimeout = Duration.ofMillis ( 1000 ) ;

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

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

    公共持续时间getReadTimeout () {
        return this .readTimeout;
    }

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

}

```

要指定30秒的会话超时时间 30 , PT30S 和 30s 都是等效的。的500ms的读超时可以以任何形式如下指定 : 500 , PT0.5S 和 500ms 。

您也可以使用任何支持的单位。这些是 :

- ns 几纳秒
- ms 为毫秒
- s 几秒钟
- m 几分钟
- h 用了几个小时
- d 好几天

默认单位是毫秒，可以按照 `@DefaultUnit` 上面的示例所示进行覆盖。



如果您正在使用仅 `Long` 用于表示持续时间的先前版本进行升级，请确保在使用单位 (`@DefaultUnit` 如果不是毫秒) 的情况下定义单位 (使用) `Duration`。这样做提供了一个透明的升级途径，同时支持更丰富的格式。

24.7.4 `@ConfigurationProperties` 验证

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

```

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

    @NotNull
    私人 InetAddress remoteAddress;

    // ... getters 和 setter

}

```



您也可以通过注释 `@Bean` 创建配置属性的方法来触发验证 `@Validated`。

虽然嵌套属性也会在绑定时进行验证，但最好也将相关字段注释为 `@Valid`。这确保即使未找到嵌套属性也会触发验证。以下示例构建在上述 `AcmeProperties` 示例上：

```

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

```

```

@NotNull
私人 InetAddress remoteAddress;

@Valid
private final security = new Security();

// ... getters 和 setter

公共 静态 类安全{

    @NotEmpty
    公共字符串用户名;

    // ... getters 和 setter

}

}

```

您还可以 `Validator` 通过创建一个名为的bean定义 来添加自定义Spring `configurationPropertiesValidator`。该 `@Bean` 方法应该声明 `static`。配置属性验证器在应用程序生命周期的早期就被创建，并且声明 `@Bean` 方法为静态，这样就可以创建bean而无需实例化 `@Configuration` 类。这样做可以避免早期实例化可能导致的任何问题。有一个 属性验证示例 显示了如何设置。



该 `spring-boot-actuator` 模块包含一个公开所有 `@ConfigurationProperties` bean 的端点。将您的Web浏览器指向 `/actuator/configprops` 或使用等效的JMX端点。详细信息请参见“ 生产准备就绪功能 ”部分。

24.7.5 @ConfigurationProperties与@Value

的 `@Value` 注释是核心容器的功能，和它不提供相同的功能，类型安全配置属性。下表总结了 `@ConfigurationProperties` 和所支持的功能 `@Value`：

特征	<code>@ConfigurationProperties</code>	<code>@Value</code>
轻松的绑定	是	没有
元数据支持	是	没有
<code>SpEL</code> 评估	没有	是

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

最后，尽管您可以在其中编写 `SpEL` 表达式 `@Value`，但这些表达式不会从应用程序属性文件处理。

25.简介

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

```

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

    // ...
}

```

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

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

您也可以使用下面的开关在命令行中指定它：`--spring.profiles.active=dev,hsqldb`。

25.1添加活动配置文件

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

有时候，将配置文件特定的属性添加到活动配置文件而不是替换它们会很有用。该 `spring.profiles.include` 属性可用于无条件添加活动配置文件。该 `SpringApplication` 入口点还设置附加配置文件的 Java API（即那些由活化的顶级 `spring.profiles.active` 属性）。请参阅 `SpringApplication` 中的 `setAdditionalProfiles()` 方法。

例如，当使用交换机运行具有以下属性的应用程序时 `--spring.profiles.active=prod`，还会激活 `proddb` 和 `prodmq` 配置文件：

```
---
my.property: fromyamlfile
---

spring.profiles: PROD
spring.profiles.include:
- proddb
- prodmq
```



请记住，`spring.profiles` 可以在 YAML 文档中定义该属性，以确定何时将此特定文档包含在配置中。有关更多详细信息，请参见 第 74.7 节“根据环境更改配置”。

25.2以编程方式设置配置文件

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

25.3配置文件特定的配置文件

通过引用的 `application.properties`（或 `application.yml`）文件的特定于配置文件的变体 `@ConfigurationProperties` 被视为文件并加载。有关详细信息，请参见“第 24.4 节”特定于配置文件的属性“。

26.记录

Spring Boot 使用 Commons Logging 进行所有内部日志记录，但将底层日志实现保持打开状态。为 Java Util Logging，Log4J2 和 Logback 提供了默认配置。在每种情况下，记录器都预先配置为使用控制台输出，并提供可选的文件输出。

默认情况下，如果您使用“Starters”，则使用 Logback 进行日志记录。还包括适当的 Logback 路由，以确保使用 Java Util 日志记录，Commons Logging，Log4J 或 SLF4J 的相关库全部正常工作。



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

26.1日志格式

Spring Boot 的默认日志输出类似于以下示例：

```
2014-03-05 10:57:51.112 信息 45469 --- [main] org.apache.catalina.core.StandardEngine: 启动Servlet引擎: Apache Tomcat / 7.0.
2014-03-05 10:57:51.253 INFO 45469 --- [ost-startStop-1] oaccC [Tomcat]. [localhost]. [/]: 初始化Spring嵌入WebApplication
2014-03-05 10:57:51.253 信息 45469 --- [ost-startStop-1] osweb.context.ContextLoader: 根WebApplicationContext: 初始化在1358
2014-03-05 10:57:51.698 信息 45469 --- [ost-startStop-1] osbceServletRegistrationBean: 将servlet: 'dispatcherServlet' 映射到|
2014-03-05 10:57:51.702 INFO 45469 --- [ost-startStop-1] osbcembedded.FilterRegistrationBean: 映射过滤器: 'hiddenHttpMetho
```

以下项目被输出：

- 日期和时间：毫秒精度，可轻松排序。
- 日志级别：`ERROR`，`WARN`，`INFO`，`DEBUG`，或 `TRACE`。
- 进程ID。
- 一个 `--` 分离器来区分实际日志消息的开始。
- 线程名称：括在方括号中（可能会截断控制台输出）。
- 记录器名称：这通常是源类名称（通常缩写）。

- 日志消息。



Logback没有**FATAL**关卡。它被映射到**ERROR**。

26.2控制台输出

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

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



你也可以**debug=true**在你的**application.properties**。

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

或者，您可以启用“跟踪”模式，方法是使用**--trace**标志（或**trace=true**您的**application.properties**）启动应用程序。这样做可以为选择的核心记录器（嵌入式容器，Hibernate模式生成和整个Spring产品组合）启用跟踪记录。

26.2.1颜色编码的输出

如果您的终端支持ANSI，则会使用彩色输出来提高可读性。您可以设置**spring.output.ansi.enabled**为支持的值来覆盖自动检测。

颜色编码通过使用**%clr**转换字来配置。最简单的形式是，转换器根据日志级别为输出着色，如以下示例所示：

```
%CLR (%5P)
```

下表描述了日志级别到颜色的映射：

水平	颜色
FATAL	红
ERROR	红
WARN	黄色
INFO	绿色
DEBUG	绿色
TRACE	绿色

或者，您可以通过将其作为选项提供给转换来指定应使用的颜色或样式。例如，要使文本变为黄色，请使用以下设置：

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

支持以下颜色和样式：

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

26.3文件输出

默认情况下，Spring Boot仅记录到控制台，不写入日志文件。如果除了控制台输出之外还想写日志文件，则需要设置一个 `logging.file` 或一个 `logging.path` 属性（例如，在您的 `application.properties`）。

下表显示了这些 `logging.*` 属性如何一起使用：

表26.1。记录属性

<code>logging.file</code>	<code>logging.path</code>	例	描述
(没有)	(没有)		仅限控制台日志。
具体文件	(没有)	<code>my.log</code>	写入指定的日志文件。名称可以是确切的位置或相对于当前目录。
(没有)	具体目录	<code>/var/log</code>	写入 <code>spring.log</code> 指定的目录。名称可以是确切的位置或相对于当前目录。

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



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



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

26.4 日志级别

所有支持的日志记录系统都可以通过使用 `TRACE`，`DEBUG`，`INFO`，`WARN`，`ERROR`，`FATAL` 或 `OFF` 之一的其中一个来在 Spring 中设置记录器级别 `Environment`（例如，在 `application.properties`）。该记录器可以通过使用被配置。`logging.level.<logger-name>=<level>` `level` `root` `logging.level.root`

以下示例显示了可能的日志记录设置 `application.properties`：

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

26.5 自定义日志配置

可以通过在类路径中包含适当的库来激活各种日志记录系统，并且可以通过在类路径的根目录或由以下 Spring `Environment` 属性指定的位置提供合适的配置文件来进一步进行自定义：`logging.config`。

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



由于记录被初始化之前的 `ApplicationContext` 创建，这是不可能控制来自伐木 `@PropertySources` 春季 `@Configuration` 文件。更改日志记录系统或完全禁用它的唯一方法是通过系统属性。

根据您的日志记录系统，加载以下文件：

记录系统	定制
的 logback	<code>logback-spring.xml</code> ， <code>logback-spring.groovy</code> ， <code>logback.xml</code> ，或者 <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> 要么 <code>log4j2.xml</code>
JDK (Java Util 日志记录)	<code>logging.properties</code>



如果可能，我们建议您使用 `-spring` 变体进行日志配置（例如，`logback-spring.xml` 而不是 `logback.xml`）。如果您使用标准配置位置，则 Spring 无法完全控制日志初始化。



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

为了帮助定制，一些其他属性从Spring传递 `Environment` 到系统属性，如下表所述：

春天的环境	系统属性	注释
<code>logging.exception-conversion-word</code>	<code>LOG_EXCEPTION_CONVERSION_WORD</code>	记录异常时使用的转换字。
<code>logging.file</code>	<code>LOG_FILE</code>	如果已定义，则用于默认的日志配置。
<code>logging.file.max-size</code>	<code>LOG_FILE_MAX_SIZE</code>	最大日志文件大小（如果启用 <code>LOG_FILE</code> ）。（仅支持默认的Logback设置。）
<code>logging.file.max-history</code>	<code>LOG_FILE_MAX_HISTORY</code>	保留的归档日志文件的最大数量（如果启用 <code>LOG_FILE</code> ）。（仅支持默认的Logback设置。）
<code>logging.path</code>	<code>LOG_PATH</code>	如果已定义，则用于默认的日志配置。
<code>logging.pattern.console</code>	<code>CONSOLE_LOG_PATTERN</code>	在控制台上使用的日志模式（ <code>stdout</code> ）。（仅支持默认的Logback设置。）
<code>logging.pattern.dateformat</code>	<code>LOG_DATEFORMAT_PATTERN</code>	日志日期格式的Appender模式。（仅支持默认的Logback设置。）
<code>logging.pattern.file</code>	<code>FILE_LOG_PATTERN</code>	在文件中使用的日志模式（如果 <code>LOG_FILE</code> 已启用）。（仅支持默认的Logback设置。）
<code>logging.pattern.level</code>	<code>LOG_LEVEL_PATTERN</code>	呈现日志级别时使用的格式（默认 <code>%5p</code> ）。（仅支持默认的Logback设置。）
<code>PID</code>	<code>PID</code>	当前进程ID（如果可能，还没有定义为OS环境变量时发现）。

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

- 的logback
- Log4j 2
- Java Util日志记录



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



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

```
2015-09-30 12: 30: 04.031 user: someone信息22174 --- [nio-8080-exec-0] demo.Controller  
处理认证请求
```

26.6 Logback扩展

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



由于标准 `logback.xml` 配置文件加载得太早，因此无法在其中使用扩展名。您需要使用 `logback-spring.xml` 或定义一个 `logging.config` 属性。



这些扩展名不能用于Logback的 配置扫描。如果您尝试这样做，则更改配置文件会导致类似于以下记录之一的错误：

```
ch.qos.logback.core.joran.spi.Interpreter@4 : 71中的错误- [springProperty]不适用，当前ElementPath为[[configuration] [spring
错误- 对[springProfile]没有适用的操作，当前的ElementPath是[[configuration] [springProfile]]]
```

26.6.1配置文件特定的配置

该 `<springProfile>` 标签可让您根据活动的Spring配置文件选择性地包含或排除配置的各个部分。`<configuration>` 元素中的任何位置都支持配置文件部分。使用该 `name` 属性来指定哪个配置文件接受配置。多个配置文件可以用逗号分隔列表指定。以下清单显示了三个样本配置文件：

```
<springProfile name = "staging" >
    <! - 当“暂存”配置文件处于活动状态时要启用的配置 - >
</ springProfile>

<springProfile name = "dev, staging" >
    <! - 当“dev”或“staging”配置文件处于活动状态时要启用的配置 - >
</ springProfile>

<springProfile name = "! production" >
    <! - “production”配置文件未处于活动状态时要启用的配置 - >
</ springProfile>
```

26.6.2环境属性

该 `<springProperty>` 标签可让您公开Spring中的属性 `Environment` 以便在Logback中使用。如果要在 Logback 配置中访问文件中的值，这样做会很有用。该标签的工作方式与Logback的标准 `<property>` 标签类似。然而，而不是指定一个直接的 `value`，你指定 `source` 的属性（从的 `Environment`）。如果您需要将属性存储在 `local` 范围以外的其他位置，则可以使用该 `scope` 属性。如果您需要回退值（如果该属性未在其中设置 `Environment`），则可以使用该 `defaultValue` 属性。以下示例显示如何公开要在Logback中使用的属性：

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



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

27.开发Web应用程序

Spring Boot非常适合Web应用程序开发。您可以使用嵌入式Tomcat，Jetty，Undertow或Netty创建自包含的HTTP服务器。大多数Web应用程序都使用该 `spring-boot-starter-web` 模块快速启动并运行。您也可以选择通过使用 `spring-boot-starter-webflux` 模块构建反应性Web应用程序。

如果您尚未开发Spring Boot Web应用程序，则可以按照“Hello World！”进行操作。示例在 [入门](#)部分。

27.1“Spring Web MVC框架”

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

以下代码显示了 `@RestController` 提供JSON数据的典型代码：

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

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

```

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

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

}

```

Spring MVC是核心Spring框架的一部分，详细信息参见[参考文档](#)。也有涵盖Spring MVC中提供一些指南spring.io/guides。

27.1.1 Spring MVC自动配置

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

自动配置会在Spring的默认设置之上添加以下功能：

- 包括`ContentNegotiatingViewResolver`和`BeanNameViewResolver`豆类。
- 支持提供静态资源，包括对WebJars的支持（稍后在本文档中介绍）。
- 自动注册`Converter`，`GenericConverter`和`Formatter`豆类。
- 支持`HttpMessageConverters`（稍后在本文档中介绍）。
- 自动注册`MessageCodesResolver`（本文稍后介绍）。
- 静态`index.html`支持。
- 自定义`Favicon`支持（本文稍后会介绍）。
- 自动使用`ConfigurableWebBindingInitializer`bean（本文稍后会介绍）。

如果您想保留Spring Boot MVC功能，并且想要添加额外的MVC配置（拦截器，格式化器，视图控制器和其他功能），则可以添加自己`@Configuration`的类型类别，`WebMvcConfigurer`但不添加`@EnableWebMvc`。如果您希望提供的定制情况`RequestMappingHandlerMapping`，`RequestMappingHandlerAdapter`或者`ExceptionHandlerExceptionResolver`，你可以声明一个`WebMvcRegistrationsAdapter`实例来提供这样的组件。

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

27.1.2 HttpMessageConverters

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

如果您需要添加或自定义转换器，则可以使用Spring Boot的`HttpMessageConverters`类，如下所示：

```

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

@Configuration
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters () {
        HttpMessageConverter <? > additional = ...
        HttpMessageConverter <? >另一个= ...
        返回 新的 HttpMessageConverters (另外, 另一个);
    }

}

```

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

27.1.3 自定义JSON序列化器和反序列化器

如果您使用Jackson来序列化和反序列化JSON数据，您可能需要编写自己的`JsonSerializer`和`JsonDeserializer`类。自定义序列化器通常通过模块向杰克逊注册，但Spring Boot提供了一种备选`@JsonComponent`注释，可以更容易地直接注册Spring Bean。

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

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

@JsonComponent
public class Example {

    公共 静态 类 Serializer 扩展 JsonSerializer <SomeObject> {
        // ...
    }

    公共 静态 类 Deserializer 扩展 JsonDeserializer <SomeObject> {
        // ...
    }

}
```

这些 `@JsonComponent` 中的所有bean都会 `ApplicationContext` 自动向Jackson注册。由于使用了 `@JsonComponent` 元注释 `@Component`，因此适用通常的组件扫描规则。

春季启动也提供 `JsonObjectSerializer` 和 `JsonObjectDeserializer` 基础类，序列化对象时提供标准版本的杰克逊有用的替代。见 `JsonObjectSerializer` 和 `JsonObjectDeserializer` 在Javadoc了解详情。

27.1.4 MessageCodesResolver

Spring MVC有一个策略来生成错误代码，用于从绑定错误中呈现错误消息：`MessageCodesResolver`。如果你设置 `spring.mvc.message-codes-resolver.format` 属性，`PREFIX_ERROR_CODE` 或者 `POSTFIX_ERROR_CODE` Spring Boot为你创建一个属性（参见枚举 `DefaultMessageCodesResolver.Format`）。

27.1.5 静态内容

默认情况下，Spring Boot将从类路径或根目录中的 `/static`（或 `/public` 或 `/resources` 或 `/META-INF/resources`）目录中提供静态内容 `ServletContext`。它使用 `ResourceHttpRequestHandler` 来自Spring MVC，以便您可以通过添加自己 `WebMvcConfigurer` 的 `addResourceHandlers` 方法并重写该方法来修改该行为。

在独立的web应用程序中，容器中的默认servlet也被启用，并充当回退，`ServletContext` 如果Spring决定不处理它，则从其根目录提供内容。大多数情况下，这种情况不会发生（除非你修改了默认的MVC配置），因为Spring总是可以处理请求 `DispatcherServlet`。

默认情况下，映射资源 `/**`，但您可以使用该 `spring.mvc.static-path-pattern` 属性调整 资源。例如，重新部署所有资源 `/resources/**` 可以实现如下：

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

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

除了前面提到的“标准”静态资源位置之外，还为Webjars内容制作了一个特例。如果包含路径的任何资源以 `/webjars/**` Webjars格式打包，则会从jar文件提供。



`src/main/webapp` 如果您的应用程序打包为jar，请不要使用该目录。虽然这个目录是一个通用的标准，它的工作原理只是战争的包装，它是默默大多数构建工具忽略，如果你生成一个罐子。

Spring Boot还支持Spring MVC提供的高级资源处理功能，允许使用例如缓存清除静态资源或使用Webjars的版本不可知URL。

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



如果您使用JBoss，则需要声明 `webjars-locator-jboss-vfs` 依赖项而不是 `webjars-locator-core`。否则，所有Webjars都会解析为 `404`。

要使用缓存清除，以下配置可为所有静态资源配置缓存清除解决方案，从而有效地添加内容哈希，例如
`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>` 在URL中：

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



由于 `ResourceUrlEncodingFilter` 为 Thymeleaf 和 FreeMarker 自动配置了资源链接，因此资源链接将在运行时在模板中重写。使用 JSP 时，应该手动声明此过滤器。其他模板引擎目前不会自动支持，但可以使用自定义模板宏/助手和使用 `ResourceUrlProvider`。

例如，在使用 JavaScript 模块加载程序动态加载资源时，重命名文件不是一个选项。这就是为什么其他战略也得到支持并可以合并的原因。“固定”策略在 URL 中添加静态版本字符串而不更改文件名，如以下示例所示：

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

通过这种配置，位于 `"/js/lib/"` 使用固定版本控制策略 (`"/v12/js/lib/mymodule.js"`) 的 JavaScript 模块，而其他资源仍然使用内容 one (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`)。

查看 [ResourceProperties](#) 更多支持的选项。



这个特性已经在专门的 [博客文章](#) 和 Spring 框架的 [参考文档](#) 中进行了详细描述。

27.1.6 欢迎页面

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

27.1.7 自定义 Favicon

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

27.1.8 路径匹配和内容协商

Spring MVC 可以通过查看请求路径并将它匹配到应用程序中定义的映射（例如 `@GetMapping` Controller 方法上的注释），将传入的 HTTP 请求映射到处理程序。

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

还有其他方法可以处理不一致地发送适当的“接受”请求标头的 HTTP 客户端。我们可以使用查询参数来确保类似的请求 `"GET /projects/spring-boot?format=json"` 将映射到 `@GetMapping("/projects/spring-boot")` 以下内容，而不是使用后缀匹配：

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

# 我们可以更改参数名称，默认为“格式”：
#spring.mvc.contentnegotiation.parameter-name = myparam

# 我们还可以通过以下方式注册其他文件扩展名/媒体类型：
spring.mvc.contentnegotiation.media-types.markdown = text / markdown
```

如果您了解注意事项并仍然希望应用程序使用后缀模式匹配，则需要进行以下配置：

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

# 您也可以将该功能限制为已知扩展
#spring.mvc.pathmatch.use-registered-suffix-pattern = true
```

```
#我们还可以通过以下方式注册其他文件扩展名/媒体类型:  
#spring.mvc.contentnegotiation.media-types.adoc = text / asciidoc
```

27.1.9 ConfigurableWebBindingInitializer

Spring MVC使用a `WebBindingInitializer`来初始化 `WebDataBinder`一个特定的请求。如果你自己创建 `ConfigurableWebBindingInitializer @Bean`，Spring Boot会自动配置Spring MVC来使用它。

27.1.10 模板引擎

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

Spring Boot包含以下模板引擎的自动配置支持：

- FreeMarker的
- Groovy的
- Thymeleaf
- 胡子



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

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



根据您运行应用程序的方式，IntelliJ IDEA以不同的方式排序类路径。使用主方法在IDE中运行应用程序会导致与使用Maven或Gradle或从其打包的jar运行应用程序时不同的顺序。这可能会导致Spring Boot无法在类路径中找到模板。如果您遇到此问题，可以在IDE中重新排序类路径，以便首先放置模块的类和资源。或者，您可以配置模板前缀以搜索 `templates` 类路径中的每个目录，如下所示：`classpath*:templates/`。

27.1.11 错误处理

默认情况下，Spring Boot提供了一种 `/error` 以合理的方式处理所有错误的映射，并且它被注册为servlet容器中的“全局”错误页面。对于机器客户端，它会生成一个JSON响应，其中包含错误，HTTP状态和异常消息的详细信息。对于浏览器客户端，有一个“白色标签”错误视图，它以HTML格式呈现相同的数据（对其进行自定义，并添加一个 `View` 解析为 `error`）。要完全替换默认行为，可以实现 `ErrorController` 并注册该类型的bean定义，或者添加一个类型的bean `ErrorAttributes` 以使用现有机制，但替换内容。



在 `BasicErrorController` 可以用作自定义基类 `ErrorController`。如果要为新的内容类型添加处理程序（默认情况下是 `text/html` 专门处理并为其他所有内容提供回退），此功能特别有用。为此，请扩展 `BasicErrorController`，添加一个 `@RequestMapping` 具有 `produces` 属性的公共方法，并创建一个新类型的bean。

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

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

    @ExceptionHandler (YourException.class)
    @ResponseBody

    ResponseEntity <? > handleControllerException (HttpServletRequest request, Throwable ex) {
        HttpStatus status = getStatus (request);
        返回 新的 ResponseEntity <> (新的 CustomErrorResponse (status.value (), ex.getMessage (), status));
    }

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

在前面的示例中，如果 `YourException` 由在同一个包中定义的控制器抛出，则使用POJO `AcmeController` 的JSON表示 `CustomErrorType` 而不是 `ErrorAttributes` 表示形式。

自定义错误页面

如果您想为给定的状态代码显示自定义的HTML错误页面，则可以将文件添加到文件 `/error` 夹。错误页面可以是静态HTML（即，添加到任何静态资源文件夹下），也可以使用模板构建。文件的名称应该是确切的状态码或系列掩码。

例如，要映射 `404` 到静态HTML文件，您的文件夹结构如下所示：

```
SRC /
+ - 主/
  + - java /
    | + <源代码>
  + - 资源/
    + - 公共/
      + - 错误/
        | + - 404.html
        + - <其他公共资源>
```

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

```
SRC /
+ - 主/
  + - java /
    | + <源代码>
  + - 资源/
    + - 模板/
      + - 错误/
        | + - 5xx.ftl
        + - <其他模板>
```

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

```
公共 类 MyErrorViewResolver 实现 ErrorViewResolver {
  @覆盖
  公共的 ModelAndView resolveErrorView (HttpServletRequest 的请求,
    HttpStatus 状态, Map <String, Object> 模型) {
    // 使用请求或状态来选择返回 ModelAndView
    返回值 ...
  }
}
```

您还可以使用常规的Spring MVC功能，例如 `@ExceptionHandler` 方法和 `@ControllerAdvice`。在 `ErrorController` 随后拿起任何未处理的异常。

在Spring MVC之外映射错误页面

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

```
@Bean
public ErrorResponseRegistrar errorPageRegistrar () {
  return new MyErrorResponseRegistrar ();
}

// ...

私有 静态 类 MyErrorResponseRegistrar 实现 ErrorResponseRegistrar {

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

}
```



如果您注册的 `ErrorPage` 路径最终由 `Filter` 某个非Spring Web框架（如Jersey和Wicket）处理，那么 `Filter` 必须将其明确注册为 `ERROR` 调度程序，如以下示例所示：

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

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

小心：在部署到servlet容器时，Spring Boot使用其错误页面过滤器将具有错误状态的请求转发到适当的错误页面。如果响应尚未提交，则只能将请求转发到正确的错误页面。默认情况下，WebSphere Application Server 8.0和更高版本在成功完成servlet的服务方法后提交响应。您应该通过设置 `com.ibm.ws.webcontainer.invokeFlushAfterService` 为禁用此行为 `false`。

27.1.12 Spring HATEOAS

如果您开发了一个使用超媒体的RESTful API，Spring Boot为Spring HATEOAS提供了自动配置，可与大多数应用程序配合使用。自动配置取代了使用 `@EnableHypermediaSupport` 和注册大量bean的需求，以便构建基于超媒体的应用程序，其中包括 `LinkDiscoverers`（用于客户端支持）和 `ObjectMapper` 配置为将响应正确地编组到所需表示中。的 `ObjectMapper` 是通过设置各种定制的 `spring.jackson.*` 属性，或者，如果存在的话，通过一个 `Jackson2ObjectMapperBuilder` 豆。

你可以通过使用来控制Spring HATEOAS的配置 `@EnableHypermediaSupport`。请注意，这样做会禁用 `ObjectMapper` 前面所述的自定义。

27.1.13 CORS支持

跨源资源共享（CORS）是大多数浏览器实现的W3C规范，它允许您以灵活的方式指定授权哪种跨域请求，而不是使用诸如IFRAME或JSONP等安全性较低且功能较弱的方法。

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

```
@Configuration
public class MyConfiguration {

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

27.2“Spring WebFlux框架”

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

Spring WebFlux有两种风格：基于功能和基于注释的。基于注释的注释非常接近Spring MVC模型，如以下示例所示：

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

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

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

```

    // ...
}

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

}

```

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

```

@Configuration
public class RoutingConfiguration {

    @Bean
    public RouterFunction <ServerResponse> monoRouterFunction (UserHandler userHandler) {
        return route (GET (“/ {user}”).and (accept (APPLICATION_JSON)) , userHandler :: getUser)
            .andRoute (GET (“/ {user} / customers”).and (accept (APPLICATION_JSON)) , userHandler :: getUserCustomers)
            .andRoute (DELETE (“/ {user}”).and (accept (APPLICATION_JSON)) , userHandler :: deleteUser)
    }

    @Component
    公共 类 UserHandler {

        公共 Mono <ServerResponse> getUser (ServerRequest 请求) {
            // ...
        }

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

        公共 Mono <ServerResponse> deleteUser (ServerRequest 请求) {
            // ...
        }
    }
}

```

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



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

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



在您的应用程序中添加两个模块`spring-boot-starter-web` 和`spring-boot-starter-webflux` 模块会导致Spring Boot自动配置Spring MVC，而不是WebFlux。之所以选择这种行为，是因为许多Spring开发人员添加`spring-boot-starter-webflux` 到他们的Spring MVC应用程序中以使用被动方法`WebClient`。您仍然可以通过设置所选的应用程序类型来执行您的选择`SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`。

27.2.1 Spring WebFlux自动配置

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

自动配置会在Spring的默认设置之上添加以下功能：

- 配置编解码器`HttpMessageReader` 和`HttpMessageWriter` 实例（稍后在本文档中介绍）。
- 支持提供静态资源，包括对WebJars的支持（稍后在本文档中介绍）。

如果你想保持Spring Boot WebFlux的功能，并且你想添加额外的 WebFlux配置，你可以添加你自己`@Configuration`的类型`WebFluxConfigurer`但没有`@EnableWebFlux`。

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

27.2.2 使用HttpMessageReaders和HttpMessageWriters的HTTP编解码器

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

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

如果您需要添加或自定义编解码器，则可以创建一个自定义 `CodecCustomizer` 组件，如下例所示：

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

@Configuration
public class MyConfiguration {

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

}
```

您还可以利用Boot的自定义JSON序列化器和反序列化器。

27.2.3静态内容

默认情况下，Spring Boot从类路径中的 `/static` (`/public` 或 `/resources` or `/META-INF/resources`) 目录中提供静态内容。它使用 `ResourceWebHandler` 来自Spring WebFlux的内容，以便您可以通过添加自己 `WebFluxConfigurer` 的 `addResourceHandlers` 方法并重写该方法来修改该行为。

默认情况下，映射资源 `/**`，但您可以通过设置 `spring.webflux.static-path-pattern` 属性来调整 资源。例如，重新部署所有资源 `/resources/**` 可以实现如下：

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

您还可以使用自定义静态资源位置 `spring.resources.static-locations`。这样做会用目录位置列表替换默认值。如果这样做，默认的欢迎页面检测会切换到您的自定义位置。因此，如果 `index.html` 您的任何位置在启动时存在，则它是应用程序的主页。

除了前面列出的“标准”静态资源位置之外，还为Webjars内容制作了一个特例。如果包含路径的任何资源以 `/webjars/**` Webjars格式打包，则会从jar文件提供。



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

27.2.4模板引擎

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

Spring Boot包含以下模板引擎的自动配置支持：

- FreeMarker的
- Thymeleaf
- 胡子

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

27.2.5错误处理

Spring Boot提供了 `WebExceptionHandler` 一个以合理的方式处理所有错误的方法。它在处理顺序中的位置就在WebFlux提供的处理程序之前，这被认为是最后一个处理程序。对于机器客户端，它会生成一个JSON响应，其中包含错误，HTTP状态和异常消息的详细信息。对于浏览器客户端，有一个“whitelabel”错误处理程序，它以HTML格式呈现相同的数据。您也可以提供自己的HTML模板来显示错误（请参阅 [下一节](#)）。

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

要更改错误处理行为，可以实现 `ErrorWebExceptionHandler` 并注册该类型的bean定义。由于a `WebExceptionHandler` 的级别较低，因此 Spring Boot还提供了一种方便的方式 `AbstractErrorWebExceptionHandler`，让您以WebFlux功能的方式处理错误，如以下示例所示：

```
公共 类 CustomErrorHandler 扩展 AbstractErrorWebExceptionHandler {
    //在这里定义构造函数

    @覆盖
    保护 RouterFunction <ServerResponse> getRoutingFunction (ErrorAttributes errorAttributes) {

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

}
```

要获得更完整的图片，您还可以 [DefaultErrorHandler](#) 直接子类化并覆盖特定的方法。

自定义错误页面

如果您想为给定的状态代码显示自定义的HTML错误页面，则可以将文件添加到文件 [/error](#) 夹。错误页面可以是静态HTML（即，添加到任何静态资源文件夹下）或使用模板构建。文件的名称应该是确切的状态码或系列掩码。

例如，要映射 [404](#) 到静态HTML文件，您的文件夹结构如下所示：

```
SRC /
+ - 主/
  + - java /
  | + <源代码>
  + - 资源/
    + - 公共/
      + - 错误/
        | + - 404.html
        + - <其他公共资源>
```

要 [5xx](#) 使用Mustache模板映射所有错误，您的文件夹结构如下所示：

```
SRC /
+ - 主/
  + - java /
  | + <源代码>
  + - 资源/
    + - 模板/
      + - 错误/
        | + - 5xx.mustache
        + - <其他模板>
```

27.2.6网页过滤器

Spring WebFlux提供了一个 [WebFilter](#) 可以被实现来过滤HTTP请求 - 响应交换的接口。[WebFilter](#) 应用程序上下文中找到的bean将自动用于过滤每个交换。

如果过滤器的顺序很重要，可以实施 [Ordered](#) 或注释 [@Order](#)。Spring Boot自动配置可以为您配置网页过滤器。当它这样做时，将使用下表中显示的订单：

网页过滤器	订购
MetricsWebFilter	Ordered.HIGHEST_PRECEDENCE + 1
WebFilterChainProxy (Spring Security)	-100
HttpTraceWebFilter	Ordered.LOWEST_PRECEDENCE - 10

27.3 JAX-RS和泽西岛

如果您更喜欢REST端点的JAX-RS编程模型，则可以使用其中一个可用实现，而不是Spring MVC。如果您在自己的应用程序环境中注册或作为[a Bean](#)，[Jersey 1.x](#)和[Apache CXF](#)可以很好地工作。Jersey 2.x具有一些原生Spring支持，因此我们还在Spring Boot中为它提供了自动配置支持，以及一个启动器。[Servlet Filter @Bean](#)

要开始使用Jersey 2.x，请将其 `spring-boot-starter-jersey` 作为依赖项包含在内，然后您需要注册所有端点 `@Bean` 的类型 `ResourceConfig` 之一，如以下示例所示：

```
@Component
公共 类 JerseyConfig 扩展了 ResourceConfig {

    公共 JerseyConfig () {
        寄存器 (端点.类) ;
    }

}
```

 泽西对扫描可执行档案的支持相当有限。例如，它无法扫描 `WEB-INF/classes` 运行可执行文件时发现的包中的端点。为了避免这种限制，`packages` 不应该使用该方法，并且应该使用该 `register` 方法单独注册端点，如上例所示。

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

所有注册的端点应该 `@Components` 使用HTTP资源注释（`@GET`和其他），如以下示例所示：

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

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

}
```

由于 `Endpoint` Spring是Spring `@Component`，它的生命周期由Spring管理，您可以使用 `@Autowired` 注释来注入依赖关系并使用 `@Value` 注释来注入外部配置。默认情况下，Jersey servlet已注册并映射到 `/*`。您可以通过添加 `@ApplicationPath` 到您的 更改映射 `ResourceConfig`。

默认情况下，Jersey被设置为一个名为 `@Bean` 的类型的Servlet。默认情况下，该servlet是懒惰初始化的，但您可以通过设置来自定义该行为。您可以通过使用相同名称创建自己的一个来禁用或覆盖该bean。您也可以通过设置（在这种情况下，替换或覆盖）来使用过滤器而不是servlet。过滤器有一个，你可以设置。servlet和过滤器注册都可以通过使用指定的属性映射给出init参数。`ServletRegistrationBean jerseyServletRegistration spring.jersey.servlet.load-on-startup spring.jersey.type=filter`
`@Bean jerseyFilterRegistration @Order spring.jersey.filter.order spring.jersey.init.*`

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

27.4 嵌入式Servlet容器支持

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

 如果您选择在CentOS上使用Tomcat，请注意，默认情况下，临时目录用于存储已编译的JSP，文件上载等。`tmpwatch` 当您的应用程序运行时，该目录可能会被删除，从而导致失败。为了避免这种情况，您可能需要自定义 `tmpwatch` 配置，以便 `tomcat.*` 不删除或配置目录，以 `server.tomcat.basedir` 使嵌入式Tomcat使用不同的位置。

27.4.1 Servlets，过滤器和监听器

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

将Spring Servlet，过滤器和监听器注册为Spring Bean

任何 `Servlet`，`Filter` 或servlet `*Listener` 实例，它是一个Spring bean与嵌入的容器中注册。如果您想 `application.properties` 在配置期间引用您的值，这可能特别方便。

默认情况下，如果上下文仅包含一个Servlet，则将其映射到 `/`。在多个servlet bean的情况下，bean名称被用作路径前缀。过滤映射到 `/*`。

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

Spring Boot附带了许多可以定义Filter beans的自动配置。以下是过滤器及其各自顺序的几个示例（低位值意味着更高的优先级）：

Servlet过滤器	订购
<code>OrderedCharacterEncodingFilter</code>	<code>Ordered.HIGHEST_PRECEDENCE</code>
<code>WebMvcMetricsFilter</code>	<code>Ordered.HIGHEST_PRECEDENCE + 1</code>
<code>ErrorPageFilter</code>	<code>Ordered.HIGHEST_PRECEDENCE + 1</code>
<code>HttpTraceFilter</code>	<code>Ordered.LOWEST_PRECEDENCE - 10</code>

离开Filter beans无序通常是安全的。

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

27.4.2 Servlet上下文初始化

嵌入式servlet容器不直接执行Servlet 3.0+ `javax.servlet.ServletContainerInitializer` 接口或Spring的 `org.springframework.web.WebApplicationInitializer` 接口。这是一个有意的设计决策，旨在降低设计在战争中运行的第三方库可能破坏Spring Boot应用程序的风险。

如果您需要在Spring Boot应用程序中执行servlet上下文初始化，则应该注册一个实现该 `org.springframework.boot.web.servlet.ServletContextInitializer` 接口的bean。单一 `onStartup` 方法提供了访问权限，`ServletContext` 并且如果有必要，可以很容易地用作现有的适配器 `WebApplicationInitializer`。

扫描Servlet，筛选器和侦听器

当使用嵌入式容器中，类自动登记注释有 `@WebServlet`，`@WebFilter` 和 `@WebListener` 可以通过使用被使能 `@ServletComponentScan`。



`@ServletComponentScan` 在使用容器的内置发现机制的独立容器中没有效果。

27.4.3 ServletWebServerApplicationContext

在底层，Spring Boot使用不同类型的 `ApplicationContext` 嵌入式servlet容器支持。这 `ServletWebServerApplicationContext` 是一种特殊的类型 `WebApplicationContext`，它通过搜索单个 `ServletWebServerFactory` bean 来引导自身。通常一个 `TomcatServletWebServerFactory`，`JettyServletWebServerFactory` 或者 `UndertowServletWebServerFactory` 已经被自动配置。



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

27.4.4 定制嵌入式Servlet容器

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

通用服务器设置包括：

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

Spring Boot尽可能地尝试暴露常见设置，但这并非总是可行。对于这些情况，专用名称空间提供了特定于服务器的定制（请参阅 `server.tomcat` 和 `server.undertow`）。例如，可以使用嵌入式servlet容器的特定功能来配置访问日志。



查看 [ServerProperties](#) 课程以获取完整列表。

程序化定制

如果您需要以编程方式配置嵌入式servlet容器，则可以注册一个实现该 `WebServerFactoryCustomizer` 接口的Spring bean。 `WebServerFactoryCustomizer` 提供对其的访问 `ConfigurableServletWebServerFactory`，其中包括许多定制设置器方法。Tomcat，Jetty和Undertow都有专门的变体。以下示例以编程方式显示设置端口：

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

@Component
public class CustomizationBean实现了 WebServerFactoryCustomizer <ConfigurableServletWebServerFactory> {

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

}
```

直接自定义ConfigurableServletWebServerFactory

如果前面的定制技术太有限，你可以注册 `TomcatServletWebServerFactory`，`JettyServletWebServerFactory` 或 `UndertowServletWebServerFactory` 豆你自己。

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

为许多配置选项提供安装程序。如果你需要做一些更奇特的事情，还提供了一些受保护的方法“挂钩”。有关详细信息，请参阅 [源代码文档](#)。

27.4.5 JSP限制

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

- 有了Tomcat，如果你使用war包装，它应该可以工作。也就是说，一个可执行的战争工作，也可以部署到一个标准的容器（不限于，但包括Tomcat）。由于Tomcat中的硬编码文件模式，可执行jar无法使用。
- 使用Jetty，如果您使用战争包装，它应该可以工作。也就是说，一个可执行的战争有效，并且也可以部署到任何标准容器。
- Undertow不支持JSP。
- 创建自定义 `error.jsp` 页面不会覆盖错误处理的默认视图。应该使用自定义错误页面。

有一个 [JSP示例](#)，以便您可以看到如何设置。

28.安全

如果 `Spring Security` 位于类路径中，则默认情况下将保护Web应用程序。Spring Boot依靠 `Spring Security` 的内容协商策略来决定是使用 `httpBasic` 还是使用 `formLogin`。要向Web应用程序添加方法级安全性，您还可以添加 `@EnableGlobalMethodSecurity` 所需的设置。更多信息可以在 [Spring Security参考指南](#) 中找到。

默认情况下 `UserDetailsService` 有一个用户。用户名是 `user`，密码是随机的，并在应用程序启动时以INFO级别打印，如下例所示：

使用生成的安全密码：78fa095d-3f4c-48b1-ad50-e24c31d5cf35



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

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

您在Web应用程序中默认获得的基本功能是：

- A `UserDetailsService` (或 `ReactiveUserDetailsService` WebFlux应用程序的情况下) 带有内存存储的Bean和带有生成密码的单个用户 (请参阅 `SecurityProperties.User` 用户的属性) 。
- 基于表单的登录或HTTP基本安全性 (取决于内容类型) 用于整个应用程序 (包括执行机构在类路径上时的执行机构端点) 。
- A `DefaultAuthenticationEventPublisher` 用于发布身份验证事件。

你可以 `AuthenticationEventPublisher` 通过为它添加一个bean 来提供一个不同的东西。

28.1 MVC安全

默认的安全配置是在 `SecurityAutoConfiguration` 和中 实现 的 `UserDetailsServiceAutoConfiguration`。 `SecurityAutoConfiguration` 导入 `SpringBootWebSecurityConfiguration` Web安全性 并 `UserDetailsServiceAutoConfiguration` 配置身份验证，这在非Web应用程序中也是相关的。要完全关闭默认的Web应用程序安全配置，您可以添加一个类型的bean `WebSecurityConfigurerAdapter` (这样做不会禁用 `UserDetailsService` 配置或Actuator的安全性) 。

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

访问规则可以通过添加自定义来覆盖 `WebSecurityConfigurerAdapter`。 Spring Boot提供了可用于覆盖执行器端点和静态资源的访问规则的便捷方法。`EndpointRequest`可以用来创建一个 `RequestMatcher` 基于 `management.endpoints.web.base-path` 属性的。`PathRequest`可用于为 `RequestMatcher` 常用位置中的资源创建一个资源。

28.2 WebFlux安全

与Spring MVC应用程序类似，您可以通过添加 `spring-boot-starter-security` 依赖关系来保护您的WebFlux应用程序。默认的安全配置是在 `ReactiveSecurityAutoConfiguration` 和中 实现 的 `UserDetailsServiceAutoConfiguration`。 `ReactiveSecurityAutoConfiguration` 导入 `WebFluxSecurityConfiguration` Web安全性 并 `UserDetailsServiceAutoConfiguration` 配置身份验证，这在非Web应用程序中也是相关的。要完全关闭默认的Web应用程序安全配置，您可以添加一个类型的bean `WebFilterChainProxy` (这样做不会禁用 `UserDetailsService` 配置或Actuator的安全性) 。

要关闭 `UserDetailsService` 配置，可以添加一个类型为 `ReactiveUserDetailsService` or 的bean `ReactiveAuthenticationManager`。

访问规则可以通过添加自定义进行配置 `SecurityWebFilterChain`。 Spring Boot提供了可用于覆盖执行器端点和静态资源的访问规则的便捷方法。`EndpointRequest`可以用来创建一个 `ServerWebExchangeMatcher` 基于 `management.endpoints.web.base-path` 属性的。

`PathRequest` 可用于为 `ServerWebExchangeMatcher` 常用位置中的资源创建一个资源。

例如，您可以通过添加如下内容来自定义您的安全配置：

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

28.3 OAuth2

OAuth2是Spring支持的广泛使用的授权框架。

28.3.1客户端

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

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

```
spring.security.oauth2.client.registration.my-client-1.client-id = abcd
spring.security.oauth2.client.registration.my-client-1.client-secret = password
spring.security.oauth2.client. registration.my-client-1.client-name = 用户范围
的客户端
spring.security.oauth2.client.registration.my-client-1.provider = my-oauth-provider
spring.security.oauth2.client.registration.my-client-1.scope = user
```

```

spring.security.oauth2.client.registration.my-client-1.redirect-uri-template = http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-1.client-authentication-method = 基本
spring.security.oauth2.client.registration.my-client-1.authorization-grant-type = authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id = abcd
spring.security.oauth2.client.registration.my-client-2.client-secret = password
spring.security.oauth2.client.registration.my-client-2.client-name = 电子邮件范围
的客户端spring.security.oauth2.client.registration.my-client-2.provider = my-oauth-provider
spring.security.oauth2.client.registration.my-client-2.scope = email
spring.security.oauth2.client.registration.my-client-2.redirect-uri-template = http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-2.client-authentication-method = 基本的
spring.security.oauth2.client.registration.my-client-2.authorization-grant-type = authorization_code

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

```

默认情况下，Spring Security `OAuth2LoginAuthenticationFilter` 只处理匹配的URL `/login/oauth2/code/*`。如果您想自定义 `redirect-uri-template` 以使用不同的模式，则需要提供配置以处理该自定义模式。例如，您可以添加自己的 `WebSecurityConfigurerAdapter` 类似于以下内容的内容：

```

公共类 OAuth2LoginSecurityConfig 扩展了 WebSecurityConfigurerAdapter {

    @Override
    保护 无效配置 (HttpSecurity http) 抛出异常 {
        HTTP
            .authorizeRequests ()
                .anyRequest () .认证 ()
                    和 ()
            .oauth2Login ()
                . redirectionEndpoint ()
                    .baseUri ("/custom-callback");
    }
}

```

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

如果您不需要定制这些提供程序，则可以将该 `provider` 属性设置为需要推断默认值的属性。另外，如果您的客户端的ID与默认支持的提供者相匹配，那么Spring Boot也会推断这一点。

换句话说，以下示例中的两种配置使用Google提供程序：

```

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

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

```

28.4 执行器安全

为了安全起见，比其他所有的驱动器 `/health` 和 `/info` 默认情况下禁用。该 `management.endpoints.web.exposure.include` 属性可用于启用执行器。

如果Spring Security位于类路径中，并且没有其他WebSecurityConfigurerAdapter存在，则执行器通过Spring Boot auto-config进行保护。如果您定义了一个定制 `WebSecurityConfigurerAdapter`，Spring Boot auto-config将会退出，您将完全控制执行器访问规则。



在设置之前 `management.endpoints.web.exposure.include`，请确保暴露的执行器不包含敏感信息和/或通过将它们放置在防火墙或Spring Security之类的设备上进行安全保护。

28.4.1 跨站请求伪造保护

由于Spring Boot依赖于Spring Security的默认设置，默认情况下，CSRF保护功能处于打开状态。这意味着执行器端点需要一个 `POST`（关闭和记录器端点），`PUT` 或 `DELETE` 将在默认安全配置使用时获得403禁止错误。



我们建议只有在创建非浏览器客户端使用的服务时才能完全禁用CSRF保护。

有关CSRF保护的更多信息，请参见“[Spring Security参考指南](#)”。

29.使用SQL数据库

在Spring框架提供了广泛的支持使用使用SQL数据库，直接JDBC访问[JdbcTemplate](#)来完成“对象关系映射”技术，比如Hibernate。Spring Data提供了额外的功能级别：[Repository](#)直接从接口创建实现，并使用约定从方法名称中生成查询。

29.1配置数据源

Java的[javax.sql.DataSource](#)接口提供了使用数据库连接的标准方法。传统上，“数据源”使用[URL](#)一些凭证来建立数据库连接。



有关更高级的示例，请参阅“[操作方法](#)”部分，通常可以完全控制DataSource的配置。

29.1.1嵌入式数据库支持

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



“[操作方法](#)”部分包含有关如何初始化数据库的一节。

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



如果您在测试中使用此功能，则可能会注意到，无论您使用的是多少个应用程序上下文，整个测试套件都会重复使用相同的数据。如果你想确保每个上下文都有一个单独的嵌入式数据库，你应该设置[spring.datasource.generate-unique-name](#)为[true](#)。

例如，典型的POM依赖关系如下所示：

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



您需要依赖于[spring-jdbc](#)嵌入式数据库才能自动配置。在这个例子中，它被传递通过[spring-boot-starter-data-jpa](#)。



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

29.1.2连接到生产数据库

生产数据库连接也可以使用池进行自动配置[DataSource](#)。Spring Boot使用以下算法来选择特定的实现：

1. 我们更喜欢HikariCP的性能和并发性。如果HikariCP可用，我们总是选择它。
2. 否则，如果Tomcat池[DataSource](#)可用，我们使用它。
3. 如果HikariCP和Tomcat池数据源都不可用，并且[Commons DBCP2](#)可用，那么我们使用它。

如果你使用[spring-boot-starter-jdbc](#)或[spring-boot-starter-data-jpa](#)“starters”，你会自动获得依赖[HikariCP](#)。



您可以完全绕过该算法，并通过设置`spring.datasource.type`属性来指定要使用的连接池。如果您在Tomcat容器中运行应用程序，这`tomcat-jdbc`是默认情况下提供的，这一点尤其重要。



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

数据源配置由外部配置属性控制`spring.datasource.*`。例如，您可以在以下部分声明以下部分`application.properties`：

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



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



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



为了`DataSource`创建池，我们需要能够验证一个有效的`Driver`类是否可用，所以我们在做任何事之前检查它。换句话说，如果你设置了`spring.datasource.driver-class-name=com.mysql.jdbc.Driver`，那么这个类必须是可加载的。

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

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

```
#如果没有连接可用，则在抛出异常之前等待的毫秒数。
spring.datasource.tomcat.max-wait = 10000

#可以同时从该池中分配的最大活动连接数。
spring.datasource.tomcat.max-active = 50

#在从池中借用它之前验证连接。
spring.datasource.tomcat.test-on-borrow = true
```

29.1.3连接到JNDI数据源

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

该`spring.datasource.jndi-name`属性可以被用作一个替代`spring.datasource.url`，`spring.datasource.username`和`spring.datasource.password`属性来访问`DataSource`从一个特定的JNDI位置。例如，以下部分`application.properties`向您展示了如何访问定义的JBoss AS`DataSource`：

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

29.2使用JdbcTemplate

Spring的类`JdbcTemplate`和`NamedParameterJdbcTemplate`类都是自动配置的，你可以`@Autowired`直接将它们放到你自己的bean中，如下例所示：

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

@Component
public class MyBean {

    私人 最终的 JdbcTemplate jdbcTemplate;

    @Autowired
```

```

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

// ...

}

```

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

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



在 `NamedParameterJdbcTemplate` 重复使用相同的 `JdbcTemplate` 幕后情况。如果 `JdbcTemplate` 定义了多个并且不存在主要候选者，`NamedParameterJdbcTemplate` 则不会自动配置。

29.3 JPA和“Spring数据”

Java持久性API是一种标准技术，可让您将对象映射到关系数据库。该 `spring-boot-starter-data-jpa` POM提供了上手的快捷方式。它提供了以下关键依赖关系：

- Hibernate：最流行的JPA实现之一。
- Spring Data JPA：可以轻松实现基于JPA的存储库。
- Spring ORMs：Spring框架的核心ORM支持。



我们在这里没有涉及JPA或Spring Data的太多细节。您可以按照“访问数据与JPA”从指导spring.io并宣读了春天的数据JPA和Hibernate的参考文档。

29.3.1实体类

传统上，JPA“实体”类是在 `persistence.xml` 文件中指定的。使用Spring Boot时，此文件不是必需的，而是使用“实体扫描”。默认情况下，搜索主要配置类（使用 `@EnableAutoConfiguration` 或注释的那个 `@SpringBootApplication`）下的所有包。

任何类别标注了 `@Entity`，`@Embeddable` 或者 `@MappedSuperclass` 被认为是。典型的实体类与以下示例类似：

```

包 com.example.myapp.domain;

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

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    私人长ID;

    @Column(nullable = false)
    私人字符串名称;

    @Column(nullable = false)
    私有字符串状态;

    // ... 其他成员，通常包含@OneToMany 映射

    protected City () {
        // JPA spec 要求的无参数构造函数
        // 这个函数是受保护的，因为它不应该直接使用
    }

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

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

```

```

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

// ... 等

}

```



您可以使用 `@EntityScan` 注释来自定义实体扫描位置。请参阅第79.4节“从Spring配置中分离@实体定义”，操作方法。

29.3.2 Spring数据JPA存储库

Spring Data JPA存储库是您可以定义以访问数据的接口。JPA查询是从您的方法名称自动创建的。例如，一个 `CityRepository` 界面可能会声明一种 `findAllByState(String state)` 方法来查找给定状态下的所有城市。

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

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

以下示例显示了一个典型的Spring数据存储库接口定义：

```

包 com.example.myapp.domain;

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

公共 接口 CityRepository 扩展了 Repository <City, Long> {

    Page <City> findAll (Pageable pageable);

    城市findByNameAndCountryAllIgnoringCase (String name, String country);
}

```



我们几乎没有触及Spring Data JPA的表面。有关完整的细节，请参阅Spring Data JPA参考文档。

29.3.3 创建和删除JPA数据库

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

```
spring.jpa.hibernate.ddl-AUTO =创造降
```



Hibernate自己的内部属性名称（如果你碰巧记得它更好）是 `hibernate.hbm2ddl.auto`。您可以通过使用 `spring.jpa.properties.*`（在将前缀添加到实体管理器之前将其剥离）将其与其他Hibernate本机属性一起设置。以下一行显示了为Hibernate设置JPA属性的示例：

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

上例中的行 `true` 将该 `hibernate.globally_quoted_identifiers` 属性的值传递给Hibernate实体管理器。

默认情况下，DDL执行（或验证）被推迟到 `ApplicationContext` 已经启动。还有一个 `spring.jpa.generate-ddl` 标志，但是如果Hibernate自动配置处于活动状态，则不会使用它，因为这些 `ddl-auto` 设置更加精细。

29.3.4 在View中打开EntityManager

如果您运行的是Web应用程序，则默认情况下，Spring Boot将注册 `OpenEntityManagerInViewInterceptor` 应用“在视图中打开 EntityManager”模式，以允许在Web视图中进行延迟加载。如果你不希望这种行为，你应该设置 `spring.jpa.open-in-view` 到 `false` 你 `application.properties`。

29.4 使用H2的Web控制台

该H2数据库提供了一个基于浏览器的控制台，是春天开机即可自动为您配置。满足以下条件时，控制台会自动配置：

- 您正在开发一个Web应用程序。
- `com.h2database:h2` 在类路径上。
- 您正在使用Spring Boot的开发人员工具。



如果你没有使用Spring Boot的开发工具，但仍想使用H2的控制台，你可以`spring.h2.console.enabled`使用值来配置属性`true`。



H2控制台仅供在开发过程中使用，因此您应注意确保`spring.h2.console.enabled`不会`true`在生产中使用。

29.4.1 更改H2 Console的路径

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

29.5 使用jOOQ

Java面向对象查询（jOOQ）是Data Geekery的一款流行产品，它可以从数据库中生成Java代码，并允许您通过其流畅的API构建类型安全的SQL查询。商业和开源版本都可以与Spring Boot一起使用。

29.5.1 代码生成

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

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

29.5.2 使用DSLContext

jOOQ提供的流畅的API是通过`org.jooq.DSLContext`接口启动的。Spring Boot将自动配置`DSLContext`为Spring Bean并将其连接到您的应用程序`DataSource`。要使用它`DSLContext`，你可以`@Autowired`，如下例所示：

```
@Component
公共 类 JooqExample实现了 CommandLineRunner {

    私人 最终 DSLContext创建;

    @Autowired
    public JooqExample (DSLContext dslContext) {
```

```

    this.create = dslContext;
}

}

```



jOOQ手册倾向于使用一个变量`create`来保存`DSLContext`。

然后，您可以使用它`DSLContext`来构建查询，如以下示例所示：

```

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

```

29.5.3 jOOQ SQL方言

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



Spring Boot只能自动配置jOOQ的开源版本支持的方言。

29.5.4 自定义jOOQ

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

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

`org.jooq.Configuration @Bean`如果你想完全控制JOOQ配置，你也可以创建你自己的。

30.与NoSQL Technologies合作

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

30.1 Redis

Redis是一个缓存，消息代理和功能丰富的键值存储。Spring Boot为Lettuce和Jedis客户端库以及Spring Data Redis提供的抽象类提供基本的自动配置。

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



我们还提供了一个`spring-boot-starter-data-redis-reactive`“入门级”，以便与被动支持的其他商店保持一致。

30.1.1 连接到Redis

您可以像注入其他Spring Bean一样注入自动配置的`RedisConnectionFactory`，`StringRedisTemplate`或者vanilla `RedisTemplate`实例。默认情况下，该实例尝试连接到Redis服务器`localhost:6379`。下面的列表显示了这样一个bean的例子：

```

@Component
public class MyBean {

    @Autowired
    private StringRedisTemplate template;
}

```

```

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

// ...
}

```



您还可以注册任意数量的实现 `LettuceClientConfigurationBuilderCustomizer` 更高级自定义的bean。如果您使用 `Jedis`，`JedisClientConfigurationBuilderCustomizer` 也可以使用。

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

30.2 MongoDB

MongoDB是一个开源的NoSQL文档数据库，它使用类似JSON的模式而不是传统的基于表格的关系数据。春季启动提供了一些便利与MongoDB的工作，包括 `spring-boot-starter-data-mongodb` 和 `spring-boot-starter-data-mongodb-reactive` “启动器”。

30.2.1 连接到MongoDB数据库

要访问Mongo数据库，您可以注入一个自动配置的 `org.springframework.data.mongodb.MongoDbFactory`。默认情况下，实例尝试连接到MongoDB服务器 `mongodb://localhost/test`。以下示例显示如何连接到MongoDB数据库：

```

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

@Component
public class MyBean {

    私人 最终的 MongoDbFactory mongo;

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

    // ...

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

```

您可以设置 `spring.data.mongodb.uri` 属性来更改URL并配置其他设置（如副本集），如以下示例中所示：

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

另外，只要你使用Mongo 2.x，你可以指定一个 `host` / `port`。例如，你可以在你的下面声明以下设置 `application.properties`：

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



如果您使用Mongo 3.0 Java驱动程序，`spring.data.mongodb.host` 并且 `spring.data.mongodb.port` 不支持。在这种情况下，`spring.data.mongodb.uri` 应该用来提供所有的配置。



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



如果你不使用Spring Data Mongo，你可以注入 `com.mongodb.MongoClient` bean 而不是使用 `MongoDbFactory`。如果你想完全控

制建立MongoDB连接，你也可以声明你自己的 `MongoDbFactory` 或 `MongoClient` bean。



如果您正在使用反应驱动程序，则需要Netty才能使用SSL。如果Netty可用并且要使用的工厂尚未自定义，则自动配置会自动配置此工厂。

30.2.2 MongoTemplate

Spring Data MongoDB提供了一个 `MongoTemplate` 类似于Spring的设计的类 `JdbcTemplate`。和 `JdbcTemplate` Spring Boot一样，Spring Boot会自动配置一个bean来注入模板，如下所示：

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

@Component
public class MyBean {

    私人 最终的 MongoTemplate mongoTemplate;

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

    // ...
}
```

有关完整的详细信息，请参阅 [MongoOperations Javadoc](#)。

30.2.3 Spring Data MongoDB存储库

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

实际上，Spring Data JPA和Spring Data MongoDB共享相同的通用基础结构。你可以从之前的JPA例子中，假设它 `City` 现在是一个Mongo数据类而不是JPA `@Entity`，它的工作方式是相同的，如下例所示：

```
包 com.example.myapp.domain;

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

公共 接口 CityRepository 扩展了 Repository <City, Long> {

    Page <City> findAll (Pageable pageable);

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



您可以使用 `@EntityScan` 注释来自定义文档扫描位置。



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

30.2.4 嵌入式Mongo

Spring Boot为嵌入式Mongo提供自动配置。要在Spring Boot应用程序中使用它，请添加依赖项 `de.flapdoodle.embed:de.flapdoodle.embed.mongo`。

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



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

如果在类路径中有SLF4J，则由Mongo生成的输出将自动路由到名为的记录

器`org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo`。

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

30.3 Neo4j

Neo4j是一个开放源代码的NoSQL图形数据库，它使用由一级关系相关的丰富的节点数据模型，它比传统的dbms方法更适合于连接大数据。

Spring Boot为使用Neo4j提供了一些便利，包括`spring-boot-starter-data-neo4j`“Starter”。

30.3.1 连接到Neo4j数据库

你可以注入的自动配置`Neo4jSession`，`Session`或者`Neo4jOperations`，就像任何其他的Spring Bean实例。默认情况下，该实例尝试连接到一个Neo4j服务器`localhost:7474`。以下示例显示如何注入Neo4j bean：

```
@Component
public class MyBean {

    私人 最终 Neo4jTemplate neo4jTemplate;

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

    // ...
}
```

您可以通过添加`org.neo4j.ogm.config.Configuration` `@Bean`自己的配置来完全控制配置。另外，添加一个`@Bean`类型`Neo4jOperations`将禁用自动配置。

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

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

30.3.2 使用嵌入模式

如果添加`org.neo4j:neo4j-ogm-embedded-driver`到应用程序的依赖项中，Spring Boot会自动配置Neo4j的进程内嵌入式实例，该应用程序在应用程序关闭时不会保留任何数据。您可以通过设置明确禁用该模式`spring.data.neo4j.embedded.enabled=false`。您还可以通过提供数据库文件的路径来为嵌入模式启用持久性，如以下示例所示：

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



Neo4j OGM嵌入式驱动程序不提供Neo4j内核。用户需要手动提供这种依赖关系。有关更多详情，请参阅 [文档](#)。

30.3.3 Neo4jSession

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

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

30.3.4 Spring数据Neo4j存储库

Spring Data包含Neo4j的存储库支持。

事实上，Spring Data JPA和Spring Data Neo4j共享相同的通用基础架构。你可以从之前的JPA例子中，假设`City`现在是Neo4j OGM `@NodeEntity`而不是JPA `@Entity`，它的工作方式是相同的。



您可以使用`@EntityScan`注释来自定义实体扫描位置。

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

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

30.3.5 存储库示例

以下示例显示了Neo4j存储库的接口定义：

```
包 com.example.myapp.domain;

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

公共 接口 CityRepository 扩展 GraphRepository <City> {

    Page <City> findAll (Pageable pageable);

    城市findByNameAndCountry (String name, String country);

}
```



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

30.4 Gemfire

Spring Data Gemfire为访问Pivotal Gemfire数据管理平台提供了方便Spring的工具。有一个`spring-boot-starter-data-gemfire`“Starter”以便捷的方式收集依赖关系。目前尚没有任何的GemFire自动配置的支持，但你可以让Spring数据存储库与一个单一的注释：`@EnableGemfireRepositories`。

30.5 Solr

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

30.5.1 连接到Solr

您可以`SolrClient`像其他任何Spring bean一样注入一个自动配置的实例。默认情况下，该实例尝试连接到服务器`localhost:8983/solr`。以下示例显示了如何注入Solr bean：

```
@Component
public class MyBean {

    私人 SolrClient solr;

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

    // ...
}
```

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

30.5.2 Spring Data Solr存储库

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

实际上，Spring Data JPA和Spring Data Solr都共享相同的通用基础结构。你可以从之前的JPA例子中，假设`City`现在是一个`@SolrDocument`类而不是JPA`@Entity`，它的工作方式是相同的。



有关Spring Data Solr的完整详细信息，请参阅[参考文档](#)。

30.6 Elasticsearch

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

30.6.1 使用Jest连接到Elasticsearch

如果你有`Jest`类路径，你可以注入一个`JestClient`默认目标的自动配置`localhost:9200`。您可以进一步调整客户端的配置方式，如以下示例所示：

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

您还可以注册任意数量的实现`HttpClientConfigBuilderCustomizer`更高级自定义的bean。以下示例调整其他HTTP设置：

```
静态类 HttpClientSettingsCustomizer 实现 HttpClientConfigBuilderCustomizer {
    @Override
    public void customize(HttpClientConfig.Builder builder) {
        builder.maxTotalConnection(100).defaultMaxTotalConnectionPerRoute(5);
    }
}
```

要完全控制注册，请定义一个`JestClient`bean。

30.6.2 通过使用Spring数据连接到Elasticsearch

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

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

```
@Component
public class MyBean {

    私人 最终的 ElasticsearchTemplate模板;

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

    // ...
}
```

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

30.6.3 Spring Data Elasticsearch存储库

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

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



有关Spring Data Elasticsearch的完整详细信息，请参阅 [参考文档](#)。

30.7 卡桑德拉

Cassandra是一个开源的分布式数据库管理系统，旨在处理大量商品服务器上的大量数据。Spring Boot为Cassandra提供了自动配置，Spring Data Cassandra提供了它的顶层抽象。有一个[spring-boot-starter-data-cassandra](#)“Starter”以便捷的方式收集依赖关系。

30.7.1 连接到Cassandra

您可以像使用其他Spring Bean一样注入自动配置[CassandraTemplate](#)或Cassandra[Session](#)实例。这些[spring.data.cassandra.*](#)属性可用于定制连接。一般来说，您提供[keyspace-name](#)和[contact-points](#)属性，如以下示例中所示：

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

以下代码清单显示了如何注入Cassandra bean：

```
@Component
public class MyBean {

    私人 CassandraTemplate模板;

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

    // ...
}
```

如果您添加自己[@Bean](#)的类型[CassandraTemplate](#)，它将替换默认值。

30.7.2 Spring Data Cassandra存储库

Spring Data包含对Cassandra的基本存储库支持。目前，这比前面讨论的JPA存储库更有限，需要使用注释查找器方法[@Query](#)。



有关Spring Data Cassandra的完整详细信息，请参阅 [参考文档](#)。

30.8 Couchbase

Couchbase是一款开源，分布式，多模型NoSQL面向文档的数据库，针对交互式应用进行了优化。Spring Boot提供了Couchbase的自动配置和Spring Data Couchbase提供的抽象。有[spring-boot-starter-data-couchbase](#)和[spring-boot-starter-data-couchbase-reactive](#)“Starter”以便捷的方式收集依赖关系。

30.8.1 连接到Couchbase

你可以得到一个[Bucket](#)和[Cluster](#)通过添加Couchbase SDK和一些配置。这些[spring.couchbase.*](#)属性可用于定制连接。通常，您提供引导程序主机，存储区名称和密码，如以下示例中所示：

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



您至少需要提供引导主机，在这种情况下，存储桶名称[default](#)和密码都是空字符串。或者，您可以定义自己的权限[org.springframework.data.couchbase.config.CouchbaseConfigurer](#) [@Bean](#)来控制整个配置。

也可以自定义一些[CouchbaseEnvironment](#)设置。例如，以下配置更改用于打开新的[Bucket](#)启用SSL支持的超时时间：

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

```
spring.couchbase.env.ssl.key-store-password = secret
```

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

30.8.2 Spring数据Couchbase存储库

Spring Data包含Couchbase的存储库支持。有关Spring Data Couchbase的完整详细信息，请参阅 [参考文档](#)。

您可以 `CouchbaseTemplate` 像使用其他Spring Bean一样注入自动配置的实例，前提是提供了默认值 `CouchbaseConfigurer`（当您启用 Couchbase 支持时会发生，如前所述）。

以下示例显示如何注入Couchbase bean：

```
@Component
public class MyBean {

    私人 最终的 CouchbaseTemplate模板;

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

    // ...
}
```

您可以在自己的配置中定义几个bean，以覆盖由自动配置提供的那些bean：

- A `CouchbaseTemplate` `@Bean` 的名字是 `couchbaseTemplate`。
- 一个 `IndexManager` `@Bean` 用的名称 `couchbaseIndexManager`。
- A `CustomConversions` `@Bean` 的名字是 `couchbaseCustomConversions`。

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

```
@Configuration
public class SomeConfiguration {

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

    // ...
}
```



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

30.9 LDAP

LDAP（轻量级目录访问协议）是一种开放的，厂商中立的行业标准应用协议，用于通过IP网络访问和维护分布式目录信息服务。Spring Boot为任何兼容的LDAP服务器提供自动配置，并支持来自UnboundID的嵌入式内存LDAP服务器。

Spring Data LDAP提供LDAP抽象。有一个 `spring-boot-starter-data-ldap` “Starter”以便捷的方式收集依赖关系。

30.9.1 连接到LDAP服务器

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

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

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

30.9.2 Spring数据LDAP存储库

Spring Data包含对LDAP的存储库支持。有关Spring Data LDAP的完整详细信息，请参阅 [参考文档](#)。

您也可以 `LdapTemplate` 像使用其他Spring Bean一样注入自动配置的实例，如以下示例所示：

```
@Component
public class MyBean {

    私人 最终 LdapTemplate模板;

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

    // ...
}
```

30.9.3 嵌入式内存中的LDAP服务器

出于测试目的，Spring Boot支持从UnboundID自动配置内存中的LDAP服务器。要配置服务器，请添加一个依赖项 `com.unboundid:unboundid-ldapsdk` 并声明一个 `base-dn` 属性，如下所示：

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



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

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

在属性文件中，您必须包含索引作为属性名称的一部分：

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

默认情况下，服务器在随机端口上启动并触发常规LDAP支持。没有必要指定一个 `spring.ldap.urls` 属性。

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

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

30.10 InfluxDB

InfluxDB是一款开源时间序列数据库，针对操作监控，应用程序指标，物联网传感器数据和实时分析等领域中的时间序列数据的快速，高可用性存储和检索进行了优化。

30.10.1 连接InfluxDB

InfluxDB如果 `influxdb-java` 客户端位于类路径上并设置了数据库的URL，则Spring Boot会自动配置一个实例，如下例所示：

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

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

InfluxDB依赖于OkHttp。如果你需要调整http客户端 `InfluxDB` 在幕后使用，你可以注册一个 `OkHttpClient.Builder` bean。

31.缓存

Spring框架提供了对应用程序透明地添加缓存的支持。其核心是抽象将缓存应用于方法，从而根据缓存中可用的信息减少执行次数。缓存逻辑是透明应用的，不会对调用者产生任何干扰。只要通过`@EnableCaching`注释启用缓存支持，Spring Boot就会自动配置缓存基础架构。



查看Spring Framework参考的[相关部分](#)以获取更多详细信息。

简而言之，将缓存添加到服务的操作中非常简单，只需将相关注释添加到其方法中即可，如以下示例所示：

```
import org.springframework.cache.annotation.Cacheable
import org.springframework.stereotype.Component;

@Component
public class MathService {

    @Cacheable ("piDecimals")
    public int computePiDecimal ( int i) {
        // ...
    }

}
```

这个例子演示了如何在一个潜在的昂贵操作上使用缓存。调用之前`computePiDecimal`，抽象查找`piDecimals`缓存中与该`i`参数匹配的条目。如果找到一个条目，则缓存中的内容立即返回给调用者，并且该方法不会被调用。否则，将调用该方法，并在返回值之前更新缓存。



警告

您也可以`@CacheResult`透明地使用标准的JSR-107 (JCache) 注释（例如）。但是，我们强烈建议您不要混合使用Spring Cache和JCache注释。

如果您不添加任何特定的缓存库，Spring Boot会自动配置一个在内存中使用并发映射的[简单提供程序](#)。当需要缓存时（例如`piDecimals`在前面的例子中），这个提供者会为你创建它。简单的提供者并不是真正推荐用于生产用途，但对于入门并确保您了解这些功能非常有用。当您决定使用缓存提供程序时，请确保阅读其文档以了解如何配置应用程序使用的缓存。几乎所有提供程序都要求您显式配置您在应用程序中使用的每个缓存。有些提供了一种自定义`spring.cache.cache-names`属性定义的默认缓存的方法。



透明地[更新或驱逐缓存](#)中的数据也是可能的。

31.1 支持的缓存提供程序

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

如果您尚未定义类型`CacheManager`或`CacheResolver`名称的bean`cacheResolver`（请参阅参考资料[CachingConfigurer](#)），则Spring Boot会尝试检测以下提供程序（按指定顺序）：

1. 通用
2. JCache (JSR-107) (EhCache 3 , Hazelcast , Infinispan等)
3. EhCache 2.x
4. Hazelcast
5. Infinispan的
6. Couchbase
7. Redis的
8. 咖啡因
9. 简单



也可以通过设置属性来强制某个缓存提供者`spring.cache.type`。如果您需要在特定环境（如测试）中完全禁用缓存，请使用此属性。



使用`spring-boot-starter-cache`“Starter”快速添加基本的缓存依赖关系。首发引入`spring-context-support`。如果手动添加依赖项，则必须包含`spring-context-support`以使用JCache，EhCache 2.x或Guava支持。

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

```

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

```



在前面的例子中，`ConcurrentMapCacheManager` 预计会自动配置。如果不是这种情况（您提供了自己的配置或者自动配置了不同的缓存提供程序），则根本不调用定制程序。您可以拥有任意数量的自定义器，也可以使用 `@Order` 或命令它们 `Ordered`。

31.1.1通用

如果上下文定义了至少一个 `org.springframework.cache.Cache` bean，则使用通用缓存。一个 `CacheManager` 包装创建该类型的所有豆类。

31.1.2 JCache (JSR-107)

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

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

```

#只有当有多个提供者时才需要
spring.cache.jcache.provider = com.acme.MyCachingProvider
spring.cache.jcache.config = classpath: acme.xml

```



当缓存库提供本机实现和JSR-107支持时，Spring Boot更倾向于JSR-107支持，因此，如果切换到其他JSR-107实现，则可以使用相同的功能。



Spring Boot 对Hazelcast有广泛的支持。如果单个 `HazelcastInstance` 可用，它也会自动重用 `CacheManager`，除非该 `spring.cache.jcache.config` 属性被指定。

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

- 通过设置 `spring.cache.cache-names` 属性可以在启动时创建缓存。如果定义了一个自定义 `javax.cache.configuration.Configuration` bean，它将用于自定义它们。
- `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` bean被引用 `CacheManager` 完全自定义引用。



如果 `javax.cache.CacheManager` 定义了标准bean，它将自动包装在 `org.springframework.cache.CacheManager` 抽象所期望的实现中。没有进一步的定制应用于它。

31.1.3 EhCache 2.x

如果 `ehcache.xml` 可以在类路径的根目录找到一个名为的文件，则使用EhCache 2.x。如果找到EhCache 2.x，则使用“Starter” `EhCacheCacheManager` 提供的 `spring-boot-starter-cache` 引导程序来引导缓存管理器。还可以提供备用配置文件，如以下示例所示：

```

spring.cache.ehcache.config = classpath: config / another-config.xml

```

31.1.4 Hazelcast

Spring Boot 对Hazelcast有广泛的支持。如果a `HazelcastInstance` 已被自动配置，它会自动包装在a中 `CacheManager`。

31.1.5 Infinispan

Infinispan没有默认配置文件位置，因此必须明确指定。否则，使用默认的引导程序。

```
spring.cache.infinispan.config = infinispan.xml
```

通过设置`spring.cache.cache-names`属性可以在启动时创建缓存。如果定义了一个自定义`ConfigurationBuilder`bean，它将用于自定义缓存。



Infinispan在Spring Boot中的支持仅限于嵌入式模式，并且非常基础。如果你想要更多的选择，你应该使用官方的Infinispan Spring Boot启动器。有关更多详细信息，请参阅 [Infinispan的文档](#)。

31.1.6 Couchbase

如果Couchbase Java客户端和`couchbase-spring-cache`实现可用并且Couchbase已配置，`CouchbaseCacheManager`则自动配置。通过设置`spring.cache.cache-names`属性，还可以在启动时创建其他缓存。这些缓存在`Bucket`自动配置的情况下运行。您可以还创建另一个附加缓存中`Bucket`，通过使用定制。假设你需要两个缓存（`cache1`和`cache2`）在“主”`Bucket`和一个（`cache3`）缓存上，自定义时间在“另一个”上生存2秒`Bucket`。您可以通过配置创建前两个缓存，如下所示：

```
spring.cache.cache-names = cache1, cache2
```

然后你可以定义一个`@Configuration`类来配置额外`Bucket`和`cache3`缓存，如下所示：

```
@Configuration
public class CouchbaseCacheConfiguration {

    私人 最终群集群;

    公共 CouchbaseCacheConfiguration(群集群集) {
        this.cluster = cluster;
    }

    @Bean
    public Bucket anotherBucket() {
        return this.cluster.openBucket("another", "secret");
    }

    @Bean
    public CacheManagerCustomizer<CouchbaseCacheManager> cacheManagerCustomizer() {
        return c -> {
            c.prepareCache("cache3", CacheBuilder.newInstance(anotherBucket())
                .withExpiration(2));
        };
    }
}
```

此示例配置重用`Cluster`通过自动配置创建的配置。

31.1.7 Redis

如果Redis可用且已配置，`RedisCacheManager`则会自动配置。通过设置`spring.cache.cache-names`属性可以在启动时创建额外的缓存，并且可以使用`spring.cache.redis.*`属性配置缓存默认值。例如，以下配置创建`cache1`并`cache2`缓存10分钟的生存时间：

```
spring.cache.cache-names = cache1, cache2
spring.cache.redis.time-to-live = 600000
```



默认情况下，会添加一个键前缀，这样，如果两个单独的缓存使用相同的键，则Redis不会有重叠的键并且无法返回无效值。我们强烈建议您在创建自己的设置时保持启用此设置`RedisCacheManager`。



您可以通过添加`RedisCacheConfiguration``@Bean`自己的配置来完全控制配置。如果您正在寻找自定义序列化策略，这会很有用。

31.1.8 咖啡因

`Caffeine`是对Guava缓存的Java 8重写，取代了对Guava的支持。如果存在咖啡因，则会自动配置`CaffeineCacheManager`（由`spring-boot-starter-cache`“启动器”提供）。通过设置`spring.cache.cache-names`属性可以在启动时创

建议缓存，并可以通过以下任一项（按指定顺序）进行自定义：

1. 缓存规范定义 `spring.cache.caffeine.spec`
2. 一个 `com.github.benmanes.caffeine.cache.CaffeineSpec` bean 被定义
3. 一个 `com.github.benmanes.caffeine.cache.Caffeine` bean 被定义

例如，以下配置会创建 `cache1` 并 `cache2` 缓存最大大小为 500 和 10 分钟的生存时间

```
spring.cache.cache-names = cache1, cache2
spring.cache.caffeine.spec = maximumSize = 500, expireAfterAccess = 600s
```

如果 `com.github.benmanes.caffeine.cache.CacheLoader` 定义了一个 bean，它会自动关联到 `CaffeineCacheManager`。由于将 `CacheLoader` 与缓存管理器管理的所有缓存关联，因此必须将其定义为 `CacheLoader<Object, Object>`。自动配置忽略任何其他泛型类型。

31.1.9 简单

如果找不到任何其他提供者，`ConcurrentHashMap` 则配置使用作为缓存存储的简单实现。如果您的应用程序中没有缓存库，则这是默认值。默认情况下，根据需要创建缓存，但可以通过设置 `cache-names` 属性来限制可用缓存的列表。例如，如果您只需要 `cache1` 和 `cache2` 缓存，请 `cache-names` 按如下所示设置属性：

```
spring.cache.cache-names = cache1, cache2
```

如果这样做并且您的应用程序使用未列出的缓存，那么它在运行时需要缓存时会失败，但不会在启动时缓存。这与“真实”缓存提供程序在使用未声明的缓存时的行为方式类似。

31.1.10 没有

何时 `@EnableCaching` 出现在您的配置中，预计也会有合适的缓存配置。如果您需要在某些环境中完全禁用缓存，则强制缓存类型 `none` 使用 `noop` 实现，如下例所示：

```
spring.cache.type = none
```

信息

Spring 框架为集成消息传递系统提供了广泛的支持，从简化的 JMS API 使用 `JmsTemplate` 到完整的基础架构以异步接收消息。Spring AMQP 为高级消息队列协议提供了类似的功能集。Spring Boot 还为 `RabbitTemplate` RabbitMQ 提供了自动配置选项。Spring WebSocket 本身就包含对 STOMP 消息传递的支持，Spring Boot 通过启动器和少量的自动配置提供支持。Spring Boot 也支持 Apache Kafka。

32.1 JMS

该 `javax.jms.ConnectionFactory` 接口提供了创建 `javax.jms.Connection` 用于与 JMS 代理进行交互的标准方法。虽然 Spring 需要 `ConnectionFactory` 与 JMS 一起工作，但您通常不需要直接使用它，而是可以依赖更高级别的消息抽象。（有关详细信息，请参阅 Spring Framework 参考文档的 [相关部分](#)。）Spring Boot 还自动配置必要的基础结构以发送和接收消息。

32.1.1 ActiveMQ 支持

当 ActiveMQ 在类路径中可用时，Spring Boot 也可以配置一个 `ConnectionFactory`。如果代理存在，则会自动启动并配置嵌入式代理（如果未通过配置指定代理 URL）。



如果您使用 `spring-boot-starter-activemq`，提供必要的依赖关系来连接或嵌入 ActiveMQ 实例，就像 Spring 基础设施要与 JMS 集成一样。

ActiveMQ 配置由外部配置属性控制 `spring.activemq.*`。例如，您可以在以下部分声明以下部分 `application.properties`：

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

您还可以通过添加依赖项 `org.apache.activemq:activemq-pool` 并相应地配置 JMS 资源 `PooledConnectionFactory`，如以下示例所示：

```
spring.activemq.pool.enabled = true
spring.activemq.pool.max-connections = 50
```



查看 [ActiveMQProperties](#) 更多支持的选项。您还可以注册任意数量的实现 [ActiveMQConnectionFactoryCustomizer](#) 更高级自定义的bean。

默认情况下，如果ActiveMQ尚不存在，ActiveMQ将创建一个目标，以便按照其提供的名称解析目标。

32.1.2 Artemis支持

当Spring Boot [ConnectionFactory](#) 检测到Artemis在类路径中可用时，它可以自动配置它。如果代理存在，将自动启动并配置嵌入式代理（除非已明确设置mode属性）。支持的模式 [embedded](#)（明确指出需要嵌入式代理，并且如果代理在类路径中不可用，则会发生错误）以及 [native](#)（使用 [netty](#) 传输协议连接到代理）。当后者配置时，Spring Boot会 [ConnectionFactory](#) 使用默认设置配置一个连接到本地计算机上运行的代理。



如果使用 [spring-boot-starter-artemis](#)，则提供必要的依赖关系以连接到现有的Artemis实例，并提供Spring基础结构以与JMS集成。添加 [org.apache.activemq:artemis-jms-server](#) 到您的应用程序可让您使用嵌入模式。

Artemis配置由外部配置属性控制 [spring.artemis.*](#)。例如，您可以在以下部分声明以下部分 [application.properties](#)：

```
spring.artemis.mode = native
spring.artemis.host = 192.168.1.210
spring.artemis.port = 9876
spring.artemis.user = admin
spring.artemis.password = secret
```

嵌入代理时，可以选择是否启用持久性，并列出应该提供的目标。可以将它们指定为逗号分隔列表，以使用默认选项创建它们，或者可以分别定义类型的bean [org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration](#) 或 [org.apache.activemq.artemis.jms.server.config.TopicConfiguration](#)（对于高级队列和主题配置）。

查看 [ArtemisProperties](#) 更多支持的选项。

不涉及JNDI查找，并且使用 [name](#) Artemis配置中的属性或通过配置提供的名称，针对其名称解析目标。

32.1.3使用JNDI ConnectionFactory

如果您在应用程序服务器中运行应用程序，Spring Boot将尝试 [ConnectionFactory](#) 使用JNDI 查找JMS。默认情况下，检查 [java:/JmsXA](#) 和 [java:/XAConnectionFactory](#) 位置。[spring.jms.jndi-name](#) 如果您需要指定替代位置，则可以使用该属性，如以下示例中所示：

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

32.1.4发送消息

Spring的 [JmsTemplate](#) 是自动配置的，你可以直接将它自动装入自己的bean中，如下例所示：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    私人 最终 JmsTemplate jmsTemplate;

    @Autowired
    public MyBean (JmsTemplate jmsTemplate) {
        this .jmsTemplate = jmsTemplate;
    }

    // ...
}
```



`JmsMessagingTemplate`可以以类似的方式注入。如果定义了一个`DestinationResolver`或一个`MessageConverter`bean，它将自动关联到自动配置`JmsTemplate`。

32.1.5接收消息

当存在JMS基础结构时，可以使用注释`@JmsListener`创建侦听器端点。如果no `JmsListenerContainerFactory`已定义，则会自动配置默认值。如果定义了一个`DestinationResolver`或一个`MessageConverter`bean，它将自动关联到默认工厂。

默认情况下，默认工厂是事务性的。如果您在`JtaTransactionManager`存在a的基础结构中运行，则默认情况下它将与侦听器容器关联。如果不是，则`sessionTransacted`标志被启用。在后一种情况下，您可以通过添加`@Transactional`侦听器方法（或其委托）来将本地数据存储事务与传入消息的处理关联起来。这确保了一旦本地事务完成，传入的消息就会被确认。这还包括发送已在相同的JMS会话上执行的响应消息。

以下组件在`someQueue`目标上创建侦听器端点：

```
@Component
public class MyBean {

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



请参阅Javadoc`@EnableJms`了解更多详情。

如果您需要创建更多`JmsListenerContainerFactory`实例，或者您想要覆盖默认值，则Spring Boot会提供一个`DefaultJmsListenerContainerFactoryConfigurer`可用于使用`DefaultJmsListenerContainerFactory`与自动配置相同的设置来初始化a的实例。

例如，以下示例公开了使用特定的另一个工厂`MessageConverter`：

```
@Configuration
静态类 JmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory myFactory (
        DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory =
            新的 DefaultJmsListenerContainerFactory ();
        configurer.configure (factory, connectionFactory ());
        factory.setMessageConverter (myMessageConverter ());
        退货工厂;
    }
}
```

然后，您可以`@JmsListener`按照以下方式在任何注释方法中使用工厂：

```
@Component
public class MyBean {

    @JmsListener(destination = "someQueue", containerFactory ="myFactory")
    public void processMessage (String content) {
        // ...
    }
}
```

32.2 AMQP

高级消息队列协议（AMQP）是面向消息中间件的平台中立的有线协议。Spring AMQP项目将核心Spring概念应用于基于AMQP的消息传递解决方案的开发。Spring Boot为通过RabbitMQ与AMQP一起工作提供了一些便利，包括`spring-boot-starter-amqp`“Starter”。

32.2.1 RabbitMQ支持

RabbitMQ是基于AMQP协议的轻量级，可靠，可扩展，可移植的消息代理。Spring使用 RabbitMQ 通过AMQP协议进行通信。

RabbitMQ配置由外部配置属性控制 `spring.rabbitmq.*`。例如，您可以在以下部分声明以下部分 `application.properties`：

```
spring.rabbitmq.host = localhost
spring.rabbitmq.port = 5672
spring.rabbitmq.username = admin
spring.rabbitmq.password = secret
```

如果一个 `ConnectionNameStrategy` bean 存在于上下文中，它将自动用于命名由自动配置创建的连接 `ConnectionFactory`。查看 `RabbitProperties` 更多支持的选项。



有关更多详细信息，请参阅 [了解AMQP，RabbitMQ使用的协议](#)。

32.2.2发送消息

Spring的 `AmqpTemplate` 并且 `AmqpAdmin` 是自动配置的，并且你可以直接将它们自动装入到你自己的bean中，如下例所示：

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    私人 最终 AmqpAdmin amqpAdmin;
    私人 最终 AmqpTemplate amqpTemplate;

    @Autowired
    public MyBean (AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
        this.amqpAdmin = amqpAdmin;
        这个.amqpTemplate = amqpTemplate;
    }

    // ...
}
```



`RabbitMessagingTemplate` 可以以类似的方式注入。如果 `MessageConverter` 定义了一个bean，它会自动关联到自动配置的 `AmqpTemplate`。

如有必要，任何 `org.springframework.amqp.core.Queue` 被定义为bean的都会自动用于在RabbitMQ实例上声明相应的队列。

要重试操作，您可以启用重试 `AmqpTemplate`（例如，在代理连接丢失的情况下）。重试是默认禁用的。

32.2.3接收消息

当存在Rabbit基础结构时，可以使用注释 `@RabbitListener` 来创建监听端点。如果 no `RabbitListenerContainerFactory` 已定义，`SimpleRabbitListenerContainerFactory` 则会自动配置默认值，您可以使用该 `spring.rabbitmq.listener.type` 属性切换到直接容器。如果 定义了一个 `MessageConverter` 或一个 `MessageRecoverer` bean，它将自动关联到默认工厂。

以下示例组件在 `someQueue` 队列上创建一个侦听器端点：

```
@Component
public class MyBean {

    @RabbitListener (queues ="someQueue")
    public void processMessage (String content) {
        // ...
    }
}
```



请参阅 [Javadoc @EnableRabbit](#) 了解更多详情。

如果你需要创建更多的 `RabbitListenerContainerFactory` 实例，或者你想重写默认值，Spring Boot提供了一个 `SimpleRabbitListenerContainerFactoryConfigurer` 和 `DirectRabbitListenerContainerFactoryConfigurer` 你可以用来初始化一个 `SimpleRabbitListenerContainerFactory` 和一个 `DirectRabbitListenerContainerFactory` 与自动配置使用的工厂相同的设置。



您选择哪种容器类型并不重要。这两个bean通过自动配置暴露出来。

例如，以下配置类公开了使用特定的另一个工厂 `MessageConverter`：

```
@Configuration
静态 类 RabbitConfiguration {

    @Bean
    public SimpleRabbitListenerContainerFactory myFactory (
        SimpleRabbitListenerContainerFactoryConfigurer 配置器) {
        SimpleRabbitListenerContainerFactory factory =
            新的 SimpleRabbitListenerContainerFactory ();
        configurer.configure (factory, connectionFactory);
        factory.setMessageConverter (myMessageConverter ());
        退货工厂;
    }

}
```

然后，您可以使用任何 `@RabbitListener`-annotated方法的工厂，如下所示：

```
@Component
public class MyBean {

    @RabbitListener (queues = "someQueue", containerFactory ="myFactory")
    public void processMessage (String content) {
        // ...
    }
}
```

您可以启用重试来处理侦听器引发异常的情况。默认情况下，`RejectAndDontRequeueRecoverer` 使用，但你可以定义一个 `MessageRecoverer` 你自己的。当重试耗尽时，如果代理被配置为这样，则该消息被拒绝并丢弃或路由到死信交换。默认情况下，重试被禁用。



重要

默认情况下，如果重试被禁用并且侦听器引发异常，则传递将无限期地重试。你可以通过两种方式修改此行为：

将 `defaultRequeueRejected` 属性设置为 `false` 使尝试零次重新递送或抛出一个 `AmqpRejectAndDontRequeueException` 信号以表示应该拒绝该消息。后者是启用重试并达到最大传送尝试次数时使用的机制。

32.3 Apache Kafka支持

通过提供 `spring-kafka` 项目的自动配置来支持Apache Kafka。

卡夫卡配置由外部配置属性控制 `spring.kafka.*`。例如，您可以在以下部分声明以下部分 `application.properties`：

```
spring.kafka.bootstrap-servers = localhost: 9092
spring.kafka.consumer.group-id = myGroup
```



要在启动时创建主题，请添加一个类型的bean `NewTopic`。如果该主题已经存在，则该bean将被忽略。

查看 `KafkaProperties` 更多支持的选项。

32.3.1发送消息

Spring的 `KafkaTemplate` 是自动配置的，您可以直接在自己的bean中自动装配它，如下例所示：

```
@Component
public class MyBean {
```

```

    私人 最终 KafkaTemplate kafkaTemplate;

    @Autowired
    public MyBean (KafkaTemplate kafkaTemplate) {
        this .kafkaTemplate = kafkaTemplate;
    }

    // ...
}

```



如果 `RecordMessageConverter` 定义了一个bean，它会自动关联到自动配置的 `KafkaTemplate`。

32.3.2接收消息

当存在Apache Kafka基础架构时，可以使用注释 `@KafkaListener` 来创建侦听器端点。如果no `KafkaListenerContainerFactory` 已定义，则会使用中定义的键自动配置默认值 `spring.kafka.listener.*`。另外，如果 `RecordMessageConverter` 定义了一个bean，它会自动关联到默认工厂。

以下组件为该 `someTopic` 主题创建一个侦听器端点：

```

@Component
public class MyBean {

    @KafkaListener (topics =“someTopic”)
    public void processMessage (String content) {
        // ...
    }
}

```

32.3.3其他卡夫卡属性

附录A“[常用应用程序属性](#)”中显示了自动配置支持的 [属性](#)。请注意，大多数情况下，这些属性（连字符或camelCase）直接映射到Apache Kafka虚线属性。有关详细信息，请参阅Apache Kafka文档。

前几个属性适用于生产者和消费者，但如果您希望为每个生产者和消费者使用不同的值，可以在生产者或消费者级别指定。Apache Kafka指定具有HIGH，MEDIUM或LOW重要性的属性。Spring Boot自动配置支持所有HIGH重要属性，一些选定的MEDIUM和LOW属性以及任何没有默认值的属性。

只有Kafka支持的属性的子集可以通过 `KafkaProperties` 该类获得。如果您希望使用不直接支持的其他属性来配置生产者或消费者，请使用以下属性：

```

spring.kafka.properties.prop.one =第一个
spring.kafka.admin.properties.prop.two =第二个
spring.kafka.consumer.properties.prop.three =第三个
spring.kafka.producer.properties.prop.four =第四个

```

这将常见的 `prop.one` 卡夫卡属性设置为 `first`（适用于生产者，消费者和管理员），`prop.two` 管理属性 `second`，`prop.three` 消费者属性 `third` 和 `prop.four` 生产者属性 `fourth`。

你也可以 `JsonDeserializer` 像下面这样配置Spring Kafka：

```

spring.kafka.consumer.value-deserializer = org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.value.default.type = org.foo.Invoice
spring.kafka.consumer.properties.spring.json.trusted.packages = org.foo, org.bar

```

同样，您可以禁用 `JsonSerializer` 在标头中发送类型信息的默认行为：

```

spring.kafka.producer.value-serializer = org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.add.type.headers = false

```



重要

以这种方式设置的属性会覆盖Spring Boot明确支持的任何配置项目。

33.用REST调用REST服务 RestTemplate

如果您需要从应用程序调用远程REST服务，则可以使用Spring Framework的 `RestTemplate` 类。由于 `RestTemplate` 实例经常需要在使用之前进行定制，因此Spring Boot不提供任何单个自动配置的 `RestTemplate` bean。但是，它会自动配置一个 `RestTemplateBuilder`，可用
于 `RestTemplate` 在需要时创建实例。自动配置 `RestTemplateBuilder` 确保合理 `HttpMessageConverters` 应用于 `RestTemplate` 实例。

以下代码显示了一个典型示例：

```
@Service
公共类 MyService {

    私人 挑战 RestTemplate restTemplate;

    public MyBean (RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate = restTemplateBuilder.build();
    }

    公众详细someRestCall (字符串名称) {
        返回此 restTemplate.getForObject ("/{名} /详细信息", 详细信息。类, 名);
    }

}
```



`RestTemplateBuilder` 包括一些可用于快速配置的有用方法 `RestTemplate`。例如，要添加BASIC认证支持，您可以使用
`builder.basicAuthorization("user", "password").build()`。

33.1 RestTemplate自定义

`RestTemplate` 定制有三种主要方法，具体取决于您希望自定义应用的范围。

为了尽可能缩小任何自定义的范围，请注入自动配置 `RestTemplateBuilder`，然后根据需要调用其方法。每个方法调用都会返回一个新 `RestTemplateBuilder` 实例，所以自定义仅影响构建器的这种使用。

要进行应用程序范围的添加剂自定义，请使用 `RestTemplateCustomizer` bean。所有这些bean都会自动注册到自动配置中 `RestTemplateBuilder`，并应用于随其构建的任何模板。

以下示例显示了一个定制程序，该定制程序为所有主机配置了代理的使用，除了 `192.168.0.5`：

```
静态类 ProxyCustomizer 实现了 RestTemplateCustomizer {

    @Override
    public void customize (RestTemplate restTemplate) {
        HttpHost代理= 新的 HttpHost ("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create ()
            .setRoutePlanner (新的 DefaultProxyRoutePlanner (代理) {

                @覆盖
                公共 HttpHost determineProxy (HttpHost目标,
                    HttpRequest请求, HttpContext上下文)
                    抛出 HttpException {
                        if (target.getHostName () .equals ("192.168.0.5")) {
                            return null;
                        }
                        返回 超级 .determineProxy (目标, 请求, 上下文);
                    }

            }) .建立 ();
        restTemplate.setRequestFactory (
            新的 HttpComponentsClientHttpRequestFactory (httpClient));
    }
}
```

最后，最极端的（也是很少使用的）选项是创建你自己的 `RestTemplateBuilder` bean。这样做会关闭a的自动配置 `RestTemplateBuilder` 并防止使用任何 `RestTemplateCustomizer` bean。

34.用REST调用REST服务 WebClient

如果你的类路径上有Spring WebFlux，你也可以选择使用[WebClient](#)来调用远程REST服务。相比之下[RestTemplate](#)，这个客户有更多的功能，而且完全被动。您可以使用构建器创建您自己的客户端实例，[WebClient.create\(\)](#)。请参阅[WebClient](#)上的相关部分。

Spring Boot为您创建并预配置这样的构建器。例如，客户端HTTP编解码器的配置方式与服务器的相同（请参阅[WebFlux HTTP编解码器自动配置](#)）。

以下代码显示了一个典型示例：

```
@Service
公共类 MyService {

    私人 最终 WebClient webClient;

    public MyBean (WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder.baseUrl ("http://example.org") .build ();
    }

    市民单<详细> someRestCall (字符串名称) {
        返回 此 .webClient.get () 。网址 (" / {名} / 详细信息" , 名字)
            .retrieve () bodyToMono (详情.类);
    }

}
```

34.1 WebClient自定义

[WebClient](#)定制有三种主要方法，具体取决于您希望自定义应用的范围。

为了尽可能缩小任何自定义的范围，请注入自动配置[WebClient.Builder](#)，然后根据需要调用其方法。[WebClient.Builder](#)实例是有状态的：构建器上的任何更改都会反映在随后用它创建的所有客户端中。如果您想使用相同的构建器创建多个客户端，则还可以考虑使用克隆构建器[WebClient.Builder other = builder.clone\(\);](#)。

要为所有[WebClient.Builder](#)实例进行应用程序范围的附加定制，您可以声明[WebClientCustomizer](#)bean并[WebClient.Builder](#)在注入点本地进行更改。

最后，你可以回到原来的API并使用[WebClient.create\(\)](#)。在这种情况下，没有自动配置或[WebClientCustomizer](#)应用。

35.验证

只要JSR-303实现（例如Hibernate验证器）位于类路径中，Bean Validation 1.1支持的方法验证功能就会自动启用。这让bean方法可以用[javax.validation](#)参数和/或返回值的约束来注释。具有这种带批注方法的目标类需要使用[@Validated](#)类型级别的批注注释其内联约束批注的搜索方法。

例如，以下服务触发第一个参数的验证，确保它的大小在8到10之间：

```
@Service
@Validated
public class MyBean {

    public Archive findByCodeAndAuthor (@Size (min = 8, max = 10) String code,
        作者作者) {
        ...
    }
}
```

36.发送电子邮件

Spring Framework通过使用[JavaMailSender](#)接口为发送电子邮件提供了一个简单的抽象，Spring Boot为它提供了自动配置以及一个入门模块。



有关如何使用的详细说明，请参阅[参考文档 JavaMailSender](#)。

如果[spring.mail.host](#)和相关库（如定义的[spring-boot-starter-mail](#)）可用，[JavaMailSender](#)则如果不存在，则创建默认值。发件人可以通过[spring.mail](#)命名空间中的配置项进一步进行自定义。查看[MailProperties](#)更多细节。

特别是，某些默认超时值是无限的，并且您可能需要更改该值以避免线程被无响应的邮件服务器阻塞，如以下示例所示：

```
spring.mail.properties.mail.smtp.connectiontimeout = 5000
spring.mail.properties.mail.smtp.timeout = 3000
spring.mail.properties.mail.smtp.writetimeout = 5000
```

37.与JTA的分布式事务

Spring Boot通过使用Atomikos或Bitronix嵌入式事务管理器支持跨多个XA资源的分布式JTA事务。在部署到合适的Java EE应用程序服务器时，也支持JTA事务。

当检测到JTA环境时，Spring's `JtaTransactionManager` 将用于管理事务。自动配置的JMS，DataSource和JPA bean已升级以支持XA事务。您可以使用标准的Spring成语，例如`@Transactional`，参与分布式事务。如果您处于JTA环境中并仍希望使用本地事务，则可以将该`spring.jta.enabled`属性设置`false`为禁用JTA自动配置。

37.1 使用Atomikos事务管理器

Atomikos是一个流行的开源事务管理器，可以嵌入到Spring Boot应用程序中。您可以使用`spring-boot-starter-jta-atomikos` Starter来拉入适当的Atomikos库。Spring Boot自动配置Atomikos并确保适当的`depends-on`设置适用于Spring bean，以便正确启动和关闭订购。

默认情况下，Atomikos事务日志被写入`transaction-logs` 应用程序主目录（应用程序jar文件所在的目录）中的目录。您可以通过`spring.jta.log-dir`在`application.properties`文件中设置属性来自定义此目录的位置。以开始的属性`spring.jta.atomikos.properties`也可以用来定制Atomikos `UserTransactionServiceImp`。有关完整的详细信息，请参阅`AtomikosProperties` Javadoc。



为了确保多个事务管理器可以安全地协调相同的资源管理器，每个Atomikos实例必须配置一个唯一的ID。默认情况下，此ID是运行Atomikos的机器的IP地址。为确保生产中的唯一性，您应该`spring.jta.transaction-manager-id` 为应用程序的每个实例配置不同的值。

37.2 使用Bitronix事务管理器

Bitronix是一个流行的开源JTA事务管理器实现。您可以使用`spring-boot-starter-jta-bitronix` 启动器将适当的Bitronix依赖项添加到您的项目中。与Atomikos一样，Spring Boot会自动配置Bitronix并对bean进行后处理，以确保启动和关闭顺序是正确的。

默认情况下，Bitronix事务日志文件（`part1.btm` 和 `part2.btm`）被写入`transaction-logs` 应用程序主目录中的目录。您可以通过设置`spring.jta.log-dir`属性来自定义此目录的位置。开头的属性`spring.jta.bitronix.properties` 也绑定到`bitronix.tm.Configuration` bean，允许完全自定义。有关详细信息，请参阅 Bitronix文档。



为了确保多个事务管理器可以安全地协调相同的资源管理器，每个Bitronix实例必须配置一个唯一的ID。默认情况下，此ID是运行Bitronix的计算机的IP地址。为确保生产中的唯一性，您应该`spring.jta.transaction-manager-id` 为应用程序的每个实例配置不同的值。

37.3 使用Narayana事务管理器

Narayana是JBoss支持的流行开源JTA事务管理器实现。您可以使用`spring-boot-starter-jta-narayana` 初学者将适当的Narayana依赖项添加到您的项目中。与Atomikos和Bitronix一样，Spring Boot自动配置Narayana并对bean进行后处理，以确保启动和关闭顺序是正确的。

默认情况下，Narayana事务日志被写入`transaction-logs` 应用程序主目录（应用程序jar文件所在的目录）中的目录。您可以通过`spring.jta.log-dir`在`application.properties`文件中设置属性来自定义此目录的位置。以开始的属性`spring.jta.narayana.properties` 也可用于自定义Narayana配置。有关完整的详细信息，请参阅`NarayanaProperties` Javadoc。



为确保多个事务管理器可以安全地协调相同的资源管理器，必须为每个Narayana实例配置一个唯一的ID。默认情况下，此ID设置为`1`。为确保生产中的唯一性，您应该`spring.jta.transaction-manager-id` 为应用程序的每个实例配置不同的值。

37.4 使用Java EE托管事务管理器

如果您将Spring Boot应用程序打包为`.war` 或 `.ear` 文件并将其部署到Java EE应用程序服务器，则可以使用应用程序服务器的内置事务管理器。Spring Boot试图通过查看常见的JNDI位置（`java:comp/UserTransaction`，`java:comp/TransactionManager` 等等）来自动配置事务管理

器。如果您使用应用程序服务器提供的事务服务，则通常还需要确保所有资源都由服务器管理并通过JNDI公开。Spring Boot尝试通过`ConnectionFactory`在JNDI路径 (`java:/JmsXA`或`java:/XAConnectionFactory`) 上查找来自动配置JMS，并且可以使用该`spring.datasource.jndi-name`属性来配置您的`DataSource`。

37.5混合XA和非XA JMS连接

使用JTA时，主要的JMS `ConnectionFactory` bean具有XA感知能力并参与分布式事务。在某些情况下，您可能希望通过使用非XA来处理某些JMS消息`ConnectionFactory`。例如，您的JMS处理逻辑可能需要比XA超时更长的时间。

如果你想使用非XA `ConnectionFactory`，你可以注入`nonXaJmsConnectionFactory` bean而不是`@Primary jmsConnectionFactory` bean。为了一致性，`jmsConnectionFactory`还通过使用bean别名来提供bean `xaJmsConnectionFactory`。

以下示例显示如何注入`ConnectionFactory`实例：

```
//注入主（支持XA）ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;

//注入XA感知的ConnectionFactory（使用别名并注入相同）
@Autowired
@Qualifier ("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;

//注入非XA感知的ConnectionFactory
@Autowired
@Qualifier ("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

37.6支持备用嵌入式事务管理器

该`XAConnectionFactoryWrapper` 和`XADatasourceWrapper` 接口可用于支持替代嵌入式事务经理。这些接口负责封装`XAConnectionFactory` 和`XADatasource` bean，并将它们公开为常规`ConnectionFactory` 和`DataSource` bean，这些都透明地注册到分布式事务中。DataSource和JMS自动配置使用JTA变体，前提是您有一个`JtaTransactionManager` bean和适当的XA包装Bean注册在您的`ApplicationContext`。

该BitronixXAConnectionFactoryWrapper 和BitronixXADatasourceWrapper 提供了如何编写XA包装很好的例子。

38. Hazelcast

如果Hazelcast在类路径中，并且找到合适的配置，Spring Boot会自动配置一个`HazelcastInstance`可以注入到应用程序中的配置。

如果你定义一个`com.hazelcast.config.Config` bean，Spring Boot会使用它。如果您的配置定义了实例名称，Spring Boot将尝试查找现有实例而不是创建新实例。

您还`hazelcast.xml`可以通过配置指定要使用的配置文件，如以下示例所示：

```
spring.hazelcast.config = classpath: config / my-hazelcast.xml
```

否则，Spring Boot将尝试从默认位置：`hazelcast.xml`工作目录或类路径的根目录中查找Hazelcast配置。我们还检查`hazelcast.config`系统属性是否设置。有关更多详细信息，请参阅 Hazelcast文档。

如果`hazelcast-client`存在于类路径中，Spring Boot首先尝试通过检查以下配置选项来创建客户端：

- 一个`com.hazelcast.client.config.ClientConfig` bean 的存在。
- 由`spring.hazelcast.config`属性定义的配置文件。
- `hazelcast.client.config`系统属性的存在。
- A `hazelcast-client.xml`在工作目录或类路径的根目录中。



Spring Boot 对Hazelcast也有 明确的缓存支持。如果启用了缓存，`HazelcastInstance`则会自动将其包含在`CacheManager` 实现中。

39.石英调度器

Spring Boot为使用Quartz调度程序提供了一些便利，包括`spring-boot-starter-quartz`“Starter”。如果Quartz可用，`Scheduler`则会自动配置（通过`SchedulerFactoryBean`抽象）。

以下类型的豆类会自动拾取并与以下类型相关联`Scheduler`：

- `JobDetail`：定义一个特定的Job。`JobDetail`可以使用`JobBuilder`API构建实例。
- `Calendar`。
- `Trigger`：定义特定作业何时被触发。

默认情况下，使用内存`JobStore`。但是，如果`DataSource`应用程序中有可用的bean，并且该`spring.quartz.job-store-type`属性已相应配置，则可以配置基于JDBC的存储，如以下示例所示：

```
spring.quartz.job-store-type = jdbc
```

使用JDBC存储时，可以在启动时初始化模式，如以下示例所示：

```
spring.quartz.jdbc.initialize-schema = always
```



默认情况下，使用Quartz库提供的标准脚本检测和初始化数据库。也可以通过设置`spring.quartz.jdbc.schema`属性来提供自定义脚本。

可以使用Quartz配置属性（`spring.quartz.properties.*`）和`SchedulerFactoryBeanCustomizer`Bean来定制Quartz Scheduler配置，这允许程序`SchedulerFactoryBean`化定制。

作业可以定义设置器以注入数据映射属性。常规bean也可以以类似的方式注入，如以下示例所示：

```
公共 类 SampleJob 扩展 QuartzJobBean {
    私人 MyService myService;
    私人字符串名称;
    //注入“MyService”bean
    公共 void setMyService (MyService myService) {...}
    //注入“name”作业数据属性
    公共 void setName (String name) {...}
    @Override
    保护 void executeInternal (JobExecutionContext context)
        引发 JobExecutionException {
    ...
}
}
```

40.春季融合

Spring Boot为Spring Integration提供了一些便利，包括`spring-boot-starter-integration`“Starter”。Spring Integration提供对消息传递和其他传输（如HTTP，TCP和其他传输）的抽象。如果Spring集成在类路径中可用，则它将通过`@EnableIntegration`注释进行初始化。

Spring Boot还配置了一些由额外的Spring Integration模块触发的功能。如果`spring-integration-jmx`还在类路径上，则消息处理统计信息将在JMX上发布。如果`spring-integration-jdbc`可用，则可以在启动时创建默认数据库模式，如下面的行所示：

```
spring.integration.jdbc.initialize-schema = always
```

有关更多详细信息，请参阅[IntegrationAutoConfiguration](#)和[IntegrationProperties](#)类。

默认情况下，如果存在Micrometer`meterRegistry`bean，则Spring集成度量标准将由千分尺管理。如果您希望使用传统的Spring集成度量标准，请将`DefaultMetricsFactory`bean添加到应用程序上下文中。

41.春季会议

Spring Boot为各种数据存储提供Spring Session自动配置。在构建Servlet Web应用程序时，可以自动配置以下商店：

- JDBC
- Redis的

- Hazelcast
- MongoDB的

构建响应式Web应用程序时，可以自动配置以下商店：

- Redis的
- MongoDB的

如果Spring Session可用，则必须选择 `StoreType` 您希望用来存储会话的那个。例如，要将JDBC用作后端存储，可以按如下方式配置应用程序：

```
spring.session.store-type = jdbc
```



您可以通过设置禁用春季会议 `store-type` 到 `none`。

每家商店都有特定的附加设置。例如，可以为JDBC存储定制表的名称，如以下示例所示：

```
spring.session.jdbc.table-name = SESSIONS
```

42.通过JMX进行监视和管理

Java管理扩展（JMX）提供了一个标准机制来监视和管理应用程序。默认情况下，春季启动创建 `MBeanServer` 具有的ID豆 `mbeanServer` 和任何暴露你的bean是被注释与Spring JMX注释（`@ManagedResource`，`@ManagedAttribute`，或 `@ManagedOperation`）。

查看 [JmxAutoConfiguration](#) 班级了解更多详情。

43.测试

Spring Boot提供了许多实用程序和注释以帮助您测试应用程序。测试支持由两个模块提供：`spring-boot-test` 包含核心项目，并 `spring-boot-test-autoconfigure` 支持测试的自动配置。

大多数开发人员使用 `spring-boot-starter-test` “Starter”，它可以导入Spring Boot测试模块以及JUnit，AssertJ，Hamcrest和其他一些有用的库。

43.1 测试范围依赖关系

在 `spring-boot-starter-test` “入门”（中 `test` `scope`）包含以下提供的库：

- JUnit：单元测试Java应用程序的事实标准。
- Spring测试和Spring Boot测试：Spring Boot应用程序的实用程序和集成测试支持。
- AssertJ：流畅的断言库。
- Hamcrest：匹配器对象库（也称为约束或谓词）。
- Mockito：Java嘲笑框架。
- JSONAssert：JSON的断言库。
- JsonPath：JSON的XPath。

我们通常在编写测试时发现这些通用库是有用的。如果这些库不适合您的需求，您可以添加您自己的附加测试依赖项。

43.2 测试Spring应用程序

依赖注入的一个主要优点是它可以让您的代码更容易进行单元测试。您 `new` 甚至可以使用运算符来实例化对象，而不涉及Spring。您也可以使用模拟对象而不是真正的依赖关系。

通常，您需要超越单元测试并开始集成测试（使用Spring `ApplicationContext`）。能够在不需要部署应用程序或需要连接到其他基础架构的情况下执行集成测试非常有用。

Spring Framework包含一个用于这种集成测试的专用测试模块。您可以直接声明依赖项 `org.springframework:spring-test` 或使用 `spring-boot-starter-test` “Starter”以传递方式进行依赖。

如果您之前没有使用该 `spring-test` 模块，您应该先阅读Spring Framework参考文档的 [相关部分](#)。

43.3 测试Spring Boot应用程序

一个Spring Boot应用程序是一个Spring `ApplicationContext`，所以没有什么特别的必须做，以超越你通常用vanilla Spring上下文做的测试。



Spring Boot的外部属性，日志记录和其他功能默认情况下仅在`SpringApplication`用于创建时才安装在上下文中。

Spring Boot提供了一个`@SpringBootTest`注释，`spring-test` `@ContextConfiguration` 当您需要Spring Boot功能时，它可以用作标准注释的替代方法。注释通过`ApplicationContext`在测试中创建使用过程来工作`SpringApplication`。

您可以使用该`webEnvironment`属性`@SpringBootTest`来进一步优化测试的运行方式：

- `MOCK`：加载`WebApplicationContext`并提供一个模拟servlet环境。使用此注释时，嵌入式servlet容器不会启动。如果servlet API不在你的类路径上，这个模式透明地退到创建一个普通的非web页面`ApplicationContext`。它可以结合使用，`@AutoConfigureMockMvc` 为`MockMvc`您的应用程序的基于测试。
- `RANDOM_PORT`：加载`ServletWebServerApplicationContext`并提供一个真正的servlet环境。嵌入式servlet容器启动并在随机端口上侦听。
- `DEFINED_PORT`：加载`ServletWebServerApplicationContext`并提供一个真正的servlet环境。嵌入式servlet容器被启动并在定义的端口上侦听（从您的`application.properties`或默认端口`8080`）。
- `NONE`：`ApplicationContext`通过使用加载，`SpringApplication`但不提供任何servlet环境（模拟或其他）。



如果您的测试是`@Transactional`，则默认情况下它将在每种测试方法结束时回滚事务。但是，由于使用这种安排`RANDOM_PORT` 或者`DEFINED_PORT` 隐式地提供真正的servlet环境，HTTP客户端和服务器在单独的线程中运行，因此在单独的事务中运行。在这种情况下，在服务器上启动的任何事务都不会回滚。



除此之外`@SpringBootTest`，还提供了许多其他注释来测试应用程序的更具体的切片。你可以在本章中找到更多细节。



不要忘记添加`@RunWith(SpringRunner.class)`到您的测试。否则，注释将被忽略。

43.3.1检测Web应用程序类型

如果Spring MVC可用，则配置常规的基于MVC的应用程序上下文。如果你只有Spring WebFlux，那么我们会检测它并配置一个基于WebFlux的应用程序上下文。

如果两者都存在，则Spring MVC优先。如果您想在此场景中测试反应型Web应用程序，则必须设置`spring.main.web-application-type` 属性：

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.main.web-application-type = reactive")
public class MyWebFluxTests {...}
```

43.3.2检测测试配置

如果您熟悉Spring Test Framework，那么您可能习惯于使用`@ContextConfiguration(classes=...)`以指定`@Configuration`要加载哪个Spring。或者，您可能经常`@Configuration`在测试中使用嵌套类。

在测试Spring Boot应用程序时，通常不需要这样做。Spring Boot的`@*Test`注释会在您没有明确定义主要配置时自动搜索您的主要配置。

搜索算法从包含测试的包直到找到用`@SpringBootApplication` or 注释的类`@SpringBootConfiguration`。只要您以合理的方式构建代码，通常会找到主要配置。



如果您使用测试注释来测试应用程序的更具体的切片，则应避免添加特定于主方法应用程序类上的特定区域的配置设置。

如果你想定制主配置，你可以使用嵌套`@TestConfiguration`类。与嵌套`@Configuration`类不同，它将被用来代替应用程序的主要配置，`@TestConfiguration`除了应用程序的主要配置之外，还会使用嵌套类。



Spring的测试框架在测试之间缓存应用程序上下文。因此，只要您的测试共享相同的配置（不管它如何被发现），加载上下文的潜在耗时过程只会发生一次。

43.3.3排除测试配置

如果您的应用程序使用组件扫描（例如，如果您使用 `@SpringBootApplication` 或 `@ComponentScan`），则可能会发现您为特定测试创建的顶级配置类意外地在任何地方都被拾取。

正如我们前面看到的，`@TestConfiguration` 可以用于测试的内部类来定制主要配置。当放置在顶层课程时，`@TestConfiguration` 表示 `src/test/java` 不应通过扫描拾取课程。然后，您可以在需要的地方明确导入该类，如以下示例所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest @Import
(MyTestsConfiguration.class)
public class MyTests {

    @Test
    public void exampleTest () {
        ...
    }

}
```



如果你直接使用 `@ComponentScan`（即不通过 `@SpringBootApplication`），你需要注册 `TypeExcludeFilter` 它。有关详细信息，请参阅 Javadoc。

43.3.4 使用正在运行的服务器进行测试

如果您需要启动完整的运行服务器，我们建议您使用随机端口。如果您使

用 `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`，每次运行测试时都会随机选取一个可用端口。

该 `@LocalServerPort` 注释可用于 注射使用的实际端口到您的测试。为了方便起见，需要对已启动的服务器进行REST调用的测试还可以使用 `@Autowired` a `WebTestClient` 来解析与正在运行的服务器的相关链接，并附带用于验证响应的专用API，如以下示例所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortWebTestClientExampleTests {

    @Autowired
    私人 WebTestClient webClient;

    @Test
    public void exampleTest () {
        this.webClient.get().uri(“/”).exchange().expectStatus().isOk()
            .expectBody(串类).isEqualTo(的“Hello World”);
    }

}
```

Spring Boot还提供了一个 `TestRestTemplate` 设施：

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

导入 静态 org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortTestRestTemplateExampleTests {
```

```

@.Autowired
私人 TestRestTemplate restTemplate;

@Test
public void exampleTest () {
    串体= 此 .restTemplate.getForObject (“/” , 串.类) ;
    assertThat (body) .isEqualTo (“Hello World”) ;
}

}

```

43.3.5嘲笑和侦察豆子

运行测试时，有时需要在应用程序上下文中模拟某些组件。例如，您可能对开发期间不可用的某些远程服务有一个正面看法。当你想模拟在真实环境中很难触发的故障时，模拟也很有用。

Spring Boot包含一个`@MockBean`注释，可用于为您的内部Bean定义Mockito模拟`ApplicationContext`。您可以使用注释来添加新的bean或替换单个现有的bean定义。注释可以直接用于测试类，测试中的字段或者`@Configuration`类和字段。当在字段上使用时，创建的模拟实例也被注入。模拟豆类在每种测试方法后自动重置。



如果您的测试使用Spring Boot的测试注释之一（如`@SpringBootTest`），则会自动启用此功能。要以不同的安排使用此功能，必须明确添加监听器，如以下示例所示：

```
@TestExecutionListeners (MockitoTestExecutionListener.类)
```

以下示例用`RemoteService`模拟实现替换现有的bean：

```

import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.test.context.junit4.*;

导入 静态 org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith (SpringRunner.class)
@SpringBootTest
public class MyTests {

    @MockBean
    私人 RemoteService remoteService ;

    @Autowired
    私人 翻转器反向器；

    @Test
    public void exampleTest () {
        // RemoteService已被注入反向bean
        ( this .remoteService.someCall () ) .willReturn ( “mock” ) ;
        String reverse = reverser.reverseSomeCall () ;
        assertThat (reverse) .isEqualTo (“kcom”) ;
    }
}

```

此外，您可以使用`@SpyBean` Mockito来包装任何现有的bean `spy`。查看Javadoc获取完整的细节。



虽然Spring的测试框架在测试之间缓存应用上下文，并为共享相同配置的测试重新使用上下文，但是使用`@MockBean`或`@SpyBean`影响缓存密钥，这很可能会增加上下文的数量。

43.3.6自动配置的测试

Spring Boot的自动配置系统适用于应用程序，但有时可能对测试有点过分。通常只会加载测试应用程序“切片”所需的配置部分。例如，您可能想要测试Spring MVC控制器是否正确映射了URL，并且您不希望在这些测试中涉及数据库调用，或者您可能想要测试JPA实体，并且在这些Web层没有兴趣时测试运行。

该 `spring-boot-test-autoconfigure` 模块包含许多可用于自动配置这些“切片”的注释。它们中的每一个都以类似的方式工作，提供 `@...Test` 加载该注释的注释以及可用于定制自动配置设置的 `ApplicationContext` 一个或多个 `@AutoConfigure...` 注释。



每个片加载一组非常有限的自动配置类。如果您需要排除其中的一个，则大多数 `@...Test` 注释都会提供一个 `excludeAutoConfiguration` 属性。或者，您可以使用 `@ImportAutoConfiguration#exclude`。



也可以将 `@AutoConfigure...` 注释与标准 `@SpringBootTest` 注释一起使用。如果您对“切片”应用程序不感兴趣，但想要一些自动配置的测试Bean，则可以使用此组合。

43.3.7 自动配置的JSON测试

要测试该对象，JSON序列化和反序列化按预期工作，可以使用 `@JsonTest` 注释。`@JsonTest` 自动配置可用的受支持的JSON映射器，该映射器可以是以下某个库：

- 杰克逊 `ObjectMapper`，任 `@JsonComponent` 豆和任何杰克逊 `Module` 小号
- `Gson`
- `Jsonb`

如果您需要配置自动配置的元素，则可以使用 `@AutoConfigureJsonTesters` 注释。

Spring Boot包含基于AssertJ的助手，它们与JSONassert和JsonPath库一起工作，以检查JSON是否按预期显示。

的 `JacksonTester`，`GsonTester`，`JsonbTester`，和 `BasicJsonTester` 类可以分别用于杰克逊，GSON，Jsonb，和字符串。测试类中的任何助手字段可以 `@Autowired` 在使用时使用 `@JsonTest`。以下示例显示了Jackson的测试类：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import org.springframework.test.context.junit4.*;
```

导入 静态 `org.assertj.core.api.Assertions.*`;

```
@RunWith(SpringRunner.class)
@JsonTest
公共类 MyJsonTests {

    @Autowired
    私人 JacksonTester<VehicleDetails> json;

    @Test
    公共 无效 testSerialize() 抛出异常{
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        //断言与测试断言相同的包中的`json`文件
        (this.json.write(details)).isEqualToJson("expected.json");
        //或者使用基于JSON路径的断言
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
        assertThat(this.json.write(details)).extractedJsonPathStringValue("@.make")
            .isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\", \"model\":\"Focus\"}";
        assertThat(this.json.parse(content))
            .isEqualTo(new VehicleDetails("Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }
}
```



JSON帮助程序类也可以直接在标准单元测试中使用。为此，请 `initFields` 在您的 `@Before` 方法中调用助手的方法，如果不使用 `@JsonTest`。

在附录中 `@JsonTest` 可以 找到启用的自动配置列表。

43.3.8 自动配置的Spring MVC测试

要测试Spring MVC控制器是否按预期工作，请使用 `@WebMvcTest` 注释。`@WebMvcTest` 自动配置Spring MVC的基础设施和限制扫描豆 `@Controller` , `@ControllerAdvice` , `@JsonComponent` , `Converter` , `GenericConverter` , `Filter` , `WebMvcConfigurer` , 和 `HandlerMethodArgumentResolver`。常规 `@Component` bean在使用此注释时不会被扫描。



如果您需要注册额外的组件（如Jackson）`Module`，则可以通过 `@Import` 在测试中使用导入其他配置类。

通常 `@WebMvcTest` 仅限于单个控制器，并与其结合使用 `@MockBean` 来为所需的合作者提供模拟实现。

`@WebMvcTest` 也可以自动配置 `MockMvc`。Mock MVC提供了一种快速测试MVC控制器的强大方法，无需启动完整的HTTP服务器。



您也可以 `MockMvc` 在非 `@WebMvcTest`（例如 `@SpringBootTest`）中通过注释来自动配置 `@AutoConfigureMockMvc`。以下示例使用 `MockMvc`：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

导入 静态 org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
导入 静态 org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
导入 静态 org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    私人 UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        给(这个).userVehicleService.getVehicleDetails("sboot")
            .willReturn(new VehicleDetails("Honda", "Civic"));
        这个.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk()).andExpect(content().string("Honda Civic"));
    }
}
```



如果您需要配置自动配置的元素（例如，应该应用servlet过滤器），则可以在 `@AutoConfigureMockMvc` 注释中使用属性。

如果您使用HtmlUnit或Selenium，则自动配置还会提供HTMLUnit `WebClient` Bean和/或 `WebDriver` Bean。以下示例使用HtmlUnit：

```
import com.gargoylesoftware.htmlunit.*;
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

导入 静态 org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

    @Autowired
    私有 WebClient webClient;

    @MockBean
}
```

```

    私人 UserVehicleService userVehicleService;

    @Test
    public void testExample () throws Exception {
        给(这个).userVehicleService.getVehicleDetails("sboot")
            .willReturn(new VehicleDetails("Honda", "Civic"));
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
    }
}

```



默认情况下，Spring Boot将 WebDriver bean放入一个特殊的“范围”中，以确保驱动程序在每次测试之后退出并注入一个新实例。如果你不想要这种行为，你可以添加 @Scope("singleton") 到你的 WebDriver @Bean 定义中。

@WebMvcTest 可以 在附录中找到启用的自动配置设置列表。



有时编写Spring MVC测试是不够的；Spring Boot可以帮助您使用实际的服务器运行完整的端到端测试。

43.3.9 自动配置的Spring WebFlux测试

为了测试Spring WebFlux控制器是否按预期工作，可以使用 @WebFluxTest 注释。@WebFluxTest 自动配置春季WebFlux基础设施和限制扫描豆 @Controller，@ControllerAdvice，@JsonComponent，Converter，GenericConverter，和 WebFluxConfigurer。常规 @Component bean在使用 @WebFluxTest 注释时不会被扫描。



如果您需要注册额外的组件，例如Jackson Module，则可以使用 @Import 您的测试导入其他配置类。

通常 @WebFluxTest 仅限于单个控制器，并与 @MockBean 注释组合使用，为所需的合作者提供模拟实现。

@WebFluxTest 也可以自动配置 WebTestClient，这为快速测试WebFlux控制器提供了一种强大的方法，无需启动完整的HTTP服务器。



您也可以 WebTestClient 在非 @WebFluxTest (例如 @SpringBootTest) 中通过注释来自动配置 @AutoConfigureWebTestClient。以下示例显示了一个使用两者 @WebFluxTest 和 a 的类 WebTestClient：

```

import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@WebFluxTest(UserVehicleController.class)

public class MyControllerTests {

    @Autowired
    私人 WebTestClient webClient;

    @MockBean
    私人 UserVehicleService userVehicleService;

    @Test
    public void testExample () throws Exception {
        给(这个).userVehicleService.getVehicleDetails("sboot")
            .willReturn(new VehicleDetails("Honda", "Civic"));
        这个.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN)
            .exchange()
            .expectStatus().isOk()
            .expectBody(字符串类).isEqualTo("本田思域");
    }
}

```

在附录中[@WebFluxTest](#)可以找到启用的自动配置列表。



有时编写Spring MVC测试是不够的; Spring Boot可以帮助您使用实际的服务器运行完整的端到端测试。

43.3.10 自动配置的数据JPA测试

您可以使用[@DataJpaTest](#)注释来测试JPA应用程序。默认情况下，它配置内存中的嵌入式数据库，扫描[@Entity](#)类并配置Spring Data JPA存储库。普通的[@Component](#) bean没有加载到[ApplicationContext](#)。

默认情况下，数据JPA测试是事务性的，并在每次测试结束时回滚。有关更多详细信息，请参阅Spring Framework参考手册中的相关章节。如果不是您想要的，您可以按照以下步骤禁用测试或整个课程的事务管理：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.junit4.SpringRunner;
导入 org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

数据JPA测试也可以注入一个[TestEntityManager](#) bean，它提供了[EntityManager](#)专门为测试而设计的标准JPA的替代方案。如果你想[TestEntityManager](#)在[@DataJpaTest](#)实例外使用，你也可以使用[@AutoConfigureTestEntityManager](#)注释。[JdbcTemplate](#)如果您需要，也可以使用A。以下示例显示[@DataJpaTest](#)正在使用的注释：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

导入 静态 org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    私人 TestEntityManager entityManager;

    @Autowired
    私人 UserRepository存储库;

    @Test
    public void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getVin()).isEqualTo("1234");
    }
}
```

内存中的嵌入式数据库通常可以很好地用于测试，因为它们速度快，不需要任何安装。但是，如果您希望针对真实数据库运行测试，则可以使用[@AutoConfigureTestDatabase](#)注释，如以下示例中所示：

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
public class ExampleRepositoryTests {

    // ...

}
```

`@DataJpaTest`可以在附录中找到启用的自动配置设置列表。

43.3.11自动配置的JDBC测试

`@JdbcTest`与`@DataJpaTest`纯JDBC相关的测试类似，但是相似。默认情况下，它还配置内存嵌入式数据库和a `JdbcTemplate`。普通的`@Component` bean没有加载到`ApplicationContext`。

默认情况下，JDBC测试是事务性的，并在每次测试结束时回滚。有关更多详细信息，请参阅Spring Framework参考手册中的[相关章节](#)。如果这不是您想要的，那么您可以为测试或整个班级禁用事务管理，如下所示：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.test.context.junit4.SpringRunner;
导入 org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@JdbcTest
@Transactional (propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

如果您希望您的测试针对真实数据库运行，则可以`@AutoConfigureTestDatabase`按照与for相同的方式使用注释`DataJpaTest`。（请参见“第43.3.10节”自动配置的数据JPA测试”）。

在附录中`@JdbcTest`可以找到启用的自动配置列表。

43.3.12自动配置的jOOQ测试

您可以`@JooqTest`以类似的方式使用，`@JdbcTest`但与jOOQ相关的测试相同。由于jOOQ严重依赖与数据库模式相对应的基于Java的模式，因此使用现有的模式`DataSource`。如果您想将其替换为内存数据库，则可以使用它`@AutoConfigureTestDatabase`来覆盖这些设置。（有关在Spring Boot中使用jOOQ的更多信息，请参阅本章前面的[第29.5节“使用jOOQ”](#)）。

`@JooqTest`配置一个`DSLContext`。普通的`@Component` bean没有加载到`ApplicationContext`。以下示例显示`@JooqTest`正在使用的注释：

```
import org.jooq.DSLContext;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@JooqTest
public class ExampleJooqTests {

    @Autowired
    私人 DSLContext dslContext;
}
```

jOOQ测试是事务性的，默认情况下在每次测试结束时回滚。如果这不是您想要的，那么可以禁用针对测试或整个测试类的事务管理，如[JDBC示例中所示](#)。

在附录中`@JooqTest`可以找到启用的自动配置列表。

43.3.13自动配置的数据MongoDB测试

您可以`@DataMongoTest`用来测试MongoDB应用程序。默认情况下，它配置内存中嵌入的MongoDB（如果可用），配置a `MongoTemplate`，扫描`@Document`类并配置Spring Data MongoDB存储库。普通的`@Component` bean没有加载到`ApplicationContext`。（有关在Spring Boot中使用MongoDB的更多信息，请参阅本章前面的[第30.2节“MongoDB”](#)）。

以下类显示`@DataMongoTest`使用中的注释：

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;
```

```

import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest
public class ExampleDataMongoTests {

    @Autowired
    私人 MongoTemplate mongoTemplate;

    //
}

```

内存中的嵌入式MongoDB通常适用于测试，因为它速度快，不需要任何开发人员安装。但是，如果您希望针对真正的MongoDB服务器运行测试，则应排除嵌入式MongoDB自动配置，如以下示例所示：

```

import org.junit.runner.RunWith;
import org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)
public class ExampleDataMongoNonEmbeddedTests {

}

```

`@DataMongoTest`可以在附录中找到启用的自动配置设置列表。

43.3.14 自动配置的数据Neo4j测试

您可以`@DataNeo4jTest`来测试Neo4j应用程序。默认情况下，它使用内存中嵌入的Neo4j（如果嵌入式驱动程序可用），扫描`@NodeEntity`类并配置Spring Data Neo4j存储库。普通的`@Component` bean没有加载到`ApplicationContext`。（有关在Spring Boot中使用Neo4J的更多信息，请参阅本章前面的第30.3节“Neo4j”）。

以下示例显示了在Spring Boot中使用Neo4J测试的典型设置：

```

import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@ DataNeo4jTest
public class ExampleDataNeo4jTests {

    @Autowired
    私有 YourRepository存储库;

    //
}

```

默认情况下，Data Neo4j测试是事务性的，并在每次测试结束时回滚。有关更多详细信息，请参阅Spring Framework参考手册中的相关章节。如果这不是您想要的，那么您可以为测试或整个班级禁用事务管理，如下所示：

```

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;
导入 org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@ DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}

```

`@DataNeo4jTest`可以在附录中找到启用的自动配置设置列表。

43.3.15自动配置的数据Redis测试

您可以使用`@DataRedisTest`测试Redis应用程序。默认情况下，它扫描`@RedisHash`类并配置Spring Data Redis存储库。普通的`@Component`bean没有加载到`ApplicationContext`。（有关在Spring Boot中使用Redis的更多信息，请参阅本章前面的第30.1节“Redis”）。

以下示例显示`@DataRedisTest`正在使用的注释：

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataRedisTest
public class ExampleDataRedisTests {

    @Autowired
    私有 YourRepository存储库;

    //
}
```

`@DataRedisTest`可以在附录中找到启用的自动配置设置列表。

43.3.16自动配置的数据LDAP测试

您可以使用`@DataLdapTest`来测试LDAP应用程序。默认情况下，它配置内存中的嵌入式LDAP（如果可用），配置`LdapTemplate`，扫描`@Entry`类，并配置Spring Data LDAP存储库。普通的`@Component`bean没有加载到`ApplicationContext`。（有关在Spring Boot中使用LDAP的更多信息，请参阅本章前面的第30.9节“LDAP”）。

以下示例显示`@DataLdapTest`正在使用的注释：

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest
public class ExampleDataLdapTests {

    @Autowired
    专用 LdapTemplate ldapTemplate;

    //
}
```

内存中的嵌入式LDAP通常适用于测试，因为它速度快，不需要任何开发人员安装。但是，如果您希望针对真实的LDAP服务器运行测试，则应排除嵌入式LDAP自动配置，如以下示例所示：

```
import org.junit.runner.RunWith;
import org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
public class ExampleDataLdapNonEmbeddedTests {

}
```

`@DataLdapTest`可以在附录中找到启用的自动配置设置列表。

43.3.17自动配置的REST客户端

您可以使用`@RestClientTest`注释来测试REST客户端。默认情况下，它会自动配置Jackson，GSON和Jsonb支持，配置一个`RestTemplateBuilder`并添加支持`MockRestServiceServer`。您要测试的特定bean应该使用`value`或的`components`属性来指定`@RestClientTest`，如以下示例所示：

```

@RunWith(SpringRunner.class)
@RestClientTest(RemoteVehicleDetailsService.class)
public class ExampleRestClientTest {

    @Autowired
    私人 RemoteVehicleDetailsService服务;

    @Autowired
    私有 MockRestServiceServer服务器;

    @Test
    public void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails () throws Exception {
        this.server.expect(requestTo("/greet/details"))
            .andRespond(withSuccess("hello", MediaType.TEXT_PLAIN));
        String greeting = this.service.callRestService();
        assertThat(greeting).isEqualTo("hello");
    }
}

```

`@RestClientTest`可以在附录中找到启用的自动配置设置列表。

43.3.18 自动配置的Spring REST Docs测试

您可以使用`@AutoConfigureRestDocs`注释在Mock MVC或REST Assured的测试中使用Spring REST Docs。它消除了对Spring REST Docs中JUnit规则的需求。

`@AutoConfigureRestDocs`可以用来覆盖默认输出目录 (`target/generated-snippets` 如果您使用的是Maven或者`build/generated-snippets` 您正在使用Gradle)。它也可以用来配置出现在任何记录的URI中的主机，方案和端口。

自动配置的Spring REST Docs使用Mock MVC进行测试

`@AutoConfigureRestDocs`定制该`MockMvc`bean以使用Spring REST Docs。您可以`@Autowired`像使用Mock MVC和Spring REST Docs时一样在测试中使用它并将其用于注入，如以下示例所示：

```

import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
导入 静态 org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
导入 静态 org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
public class UserDocumentationTests {

    @Autowired
    私有 MockMvc mvc;

    @测试
    公共 无效 listUsers () 抛出异常{
        这个 .mvc.perform(获取("/用户")).接受(MediaType.TEXT_PLAIN)
            .andExpect(状态().ISOK())
            .andDo(document("list-users"));
    }
}

```

如果您需要对Spring REST Docs配置进行更多的控制`@AutoConfigureRestDocs`，则可以使用`RestDocsMockMvcConfigurationCustomizer`bean，如以下示例所示：

```

@TestConfiguration
静态 类 CustomizationConfiguration
    实现了 RestDocsMockMvcConfigurationCustomizer {

```

```

@Override
public void customize(MockMvcRestDocumentationConfigurer configurer) {
    configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
}

}

```

如果您想使用Spring REST Docs支持参数化输出目录，您可以创建一个`RestDocumentationResultHandler` bean。自动配置调用`alwaysDo`此结果处理器，从而导致每个`MockMvc`调用自动生成默认片段。以下示例显示`RestDocumentationResultHandler`正在定义：

```

@Configuration
static class ResultHandlerConfiguration {

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}");
    }

}

```

自动配置的Spring REST Docs使用REST Assured进行测试

`@AutoConfigureRestDocs`使`RequestSpecification`预先配置为使用Spring REST Docs的bean可用于您的测试。您可以`@Autowired`像使用REST Assured和Spring REST Docs时一样在测试中使用它并在测试中使用它来注入它，如以下示例所示：

```

导入 io.restassured.specification.RequestSpecification;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.junit4.SpringRunner;

import static io.restassured.RestAssured.given;
导入 静态 org.hamcrest.CoreMatchers.is;
import static org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.document;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
public class UserDocumentationTests {

    @LocalServerPort
    私有 int 端口;

    @Autowired
    私人 RequestSpecification 文档规格;

    @Test
    public void listUsers() {
        给出(这个.documentationSpec).filter(document("list-users")).when()
            .port(this.port).get("/").then().assertThat().statusCode(is(200));
    }

}

```

如果您需要对Spring REST Docs配置进行更多的控制`@AutoConfigureRestDocs`，那么`RestDocsRestAssuredConfigurationCustomizer`可以使用bean，如以下示例所示：

```

@Configuration
public static class CustomizationConfiguration
    实现了 RestDocsRestAssuredConfigurationCustomizer {

    @Override
    public void customize(RestAssuredRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}

```

43.3.19 用户配置和切片

如果您以合理的方式构建代码，`@SpringBootApplication` 则默认使用您的类作为测试的配置。

然后重要的是不要使用特定于其功能的特定区域的配置设置来丢弃应用程序的主类。

假定您正在使用 Spring Batch，并且您依赖于它的自动配置。你可以定义你的`@SpringBootApplication`如下：

```
@SpringBootApplication
@EnableBatchProcessing
public class SampleApplication {...}
```

因为这个类是测试的源代码配置，所以任何切片测试都会尝试启动 Spring Batch，这绝对不是您想要做的。推荐的方法是将该特定于区域的配置移动到与`@Configuration`应用程序相同级别的单独类，如以下示例所示：

```
@Configuration
@EnableBatchProcessing
public class BatchConfiguration {...}
```



根据应用程序的复杂程度，您可以`@Configuration`为自定义设置一个类，也可以为每个域的区域设置一个类。后一种方法允许您在其中一个测试中启用它，如有必要，还可以使用`@Import`注释。

类路径扫描是造成混淆的另一个原因。假设，虽然您以合理的方式构建代码，但您需要扫描一个额外的软件包。您的应用程序可能类似于以下代码：

```
@SpringBootApplication
@ComponentScan ({“com.example.app”, “org.acme.another”})
public class SampleApplication {...}
```

这样做会有效地覆盖默认的组件扫描指令，而无论您选择哪个片，都会扫描这两个软件包的副作用。例如，a`@DataJpaTest`似乎突然扫描应用程序的组件和用户配置。再次，将自定义指令移至单独的类是解决此问题的好方法。



如果这不是您的选择，您可以`@SpringBootConfiguration`在测试的层次结构中创建一个地方，以便使用它。或者，您可以指定测试源，以禁用找到默认行为的行为。

43.3.20 使用 Spock 测试 Spring Boot 应用程序

如果您希望使用 Spock 来测试 Spring Boot 应用程序，则应该在 Spock 的`spock-spring`模块中添加对应用程序构建的依赖关系。`spock-spring`将 Spring 的测试框架集成到 Spock 中。建议您使用 Spock 1.1 或更高版本以从 Spock 的 Spring Framework 和 Spring Boot 集成的许多改进中受益。有关更多详细信息，请参阅 Spock 的 Spring 模块文档。

43.4 测试实用程序

测试应用程序时通常会使用的一些测试实用程序类将作为其一部分进行打包`spring-boot`。

43.4.1 ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer`是一个`ApplicationContextInitializer`你可以应用到你的测试来加载 Spring Boot `application.properties`文件。您可以在不需要所提供的全部功能时使用它`@SpringBootTest`，如以下示例所示：

```
@ContextConfiguration (classes =配置类,
    初始化器= ConfigFileApplicationContextInitializer.类)
```



`ConfigFileApplicationContextInitializer`单独使用不提供`@Value("${...}")`注入支持。它唯一的工作是确保`application.properties`文件加载到 Spring 中`Environment`。为了`@Value`支持，您需要另外配置一个`PropertySourcesPlaceholderConfigurer`或使用`@SpringBootTest`，它会自动为您配置一个。

43.4.2 EnvironmentTestUtils

`EnvironmentTestUtils` 可让您快速将属性添加到 `ConfigurableEnvironment` 或 `ConfigurableApplicationContext`。你可以用 `key=value` 字符串来调用它，如下所示：

```
EnvironmentTestUtils.addEnvironment (env, "org = Spring", "name = Boot") ;
```

43.4.3 OutputCapture

`OutputCapture` 是一个 `Rule` 可用于捕获 `System.out` 和 `System.err` 输出的 JUnit。您可以将捕获声明为 a `@Rule`，然后 `toString()` 用于声明，如下所示：

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.rule.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

公共类 MyTest {

    @Rule
    public OutputCapture capture = new OutputCapture () ;

    @Test
    公共 无效 testName () 抛出异常{
        System.out.println ("Hello World!") ;
        assertThat (capture.toString () , containsString ("World")) ;
    }

}
```

43.4.4 TestRestTemplate



Spring Framework 5.0提供了一个新 `WebTestClient` 的 `WebFlux` 集成测试以及 `WebFlux` 和 `MVC` 端到端测试。它提供了一个流畅的断言 API，而不像 `TestRestTemplate`。

`TestRestTemplate` 是 Spring 的一个便捷替代方案，`RestTemplate` 在集成测试中很有用。您可以获得一个香草模板或一个发送基本 HTTP 认证（使用用户名和密码）的模板。在任何一种情况下，模板的行为都是通过不会在服务器端错误上抛出异常的方式进行测试。建议使用 Apache HTTP Client（版本 4.3.2 或更高版本），但不是强制性的。如果你的类路径中有这个，`TestRestTemplate` 通过适当地配置客户端来做出响应。如果您确实使用 Apache 的 HTTP 客户端，则会启用一些其他易于使用的测试功能：

- 不遵循重定向（所以你可以断言响应位置）。
- Cookie 被忽略（所以模板是无状态的）。

`TestRestTemplate` 可以直接在集成测试中实例化，如以下示例所示：

```
公共类 MyTest {

    私人 TestRestTemplate 模板= 新 TestRestTemplate () ;

    @Test
    公共 无效 testRequest () 抛出异常{
        HttpHeaders 头= 此 .template.getForEntity (
            "http://myhost.example.com/example" , 字符串类) .getHeaders () ;
        assertThat (headers.getLocation () ) .hasHost ("other.example.com") ;
    }

}
```

或者，如果您在 `or` 中使用 `@SpringBootTest` 注释，则可以注入完全配置并开始使用它。如有必要，可以通过 bean 应用其他自定义。任何未指定主机和端口的 URL 都会自动连接到嵌入式服务器，如下例所示：

示：`WebEnvironment.RANDOM_PORT` `WebEnvironment.DEFINED_PORT` `TestRestTemplate` `RestTemplateBuilder`

```
@RunWith (SpringRunner.class)
@SpringBootTest (webEnvironment = WebEnvironment.RANDOM_PORT)
public class SampleWebClientTests {

    @Autowired
    .....
```

私人 TestRestTemplate模板；

```

@Test
public void testRequest () {
    HttpHeaders头= 此 .template.getForEntity (“/示例”, 字符串。类)
        .getHeaders () ;
    assertThat (headers.getLocation () ) .hasHost (“other.example.com”);
}

@TestConfiguration
静态 类 Config {

    @Bean
    public RestTemplateBuilder restTemplateBuilder () {
        return new RestTemplateBuilder () .setConnectTimeout ( 1000 ) .setReadTimeout ( 1000 );
    }

}
}

```

44. WebSockets

Spring Boot为嵌入式Tomcat 8.5，Jetty 9和Undertow提供WebSockets自动配置。如果将war文件部署到独立容器中，Spring Boot假定该容器负责配置其WebSocket支持。

Spring Framework提供了丰富的WebSocket支持，可以通过该 `spring-boot-starter-websocket` 模块轻松访问。

45. 网络服务

Spring Boot提供了Web服务自动配置功能，因此您只需定义自己的配置即可 `Endpoints`。

在春天的Web服务功能可以与轻松访问 `spring-boot-starter-webservices` 模块。

`SimpleWsdl11Definition` 并且 `SimpleXsdSchema` 可以分别为您的WSDL和XSD自动创建bean。为此，请配置它们的位置，如下例所示：

```
spring.webservices.wsdl-locations = classpath: / wsdl
```

46. 创建您自己的自动配置

如果您在开发共享库的公司工作，或者如果您在开源或商业库上工作，则可能需要开发自己的自动配置。自动配置类可以捆绑在外部瓶中，并且仍然可以通过Spring Boot获取。

自动配置可以与提供自动配置代码的“启动器”以及您将使用的典型库相关联。我们首先介绍您需要了解的内容以构建自己的自动配置，然后我们继续完成创建自定义启动器所需的典型步骤。



一个示范项目提供展示如何创建一个启动器一步一步。

46.1 了解自动配置的豆类

在引擎盖下，自动配置通过标准 `@Configuration` 类来实现。其他 `@Conditional` 注释用于约束何时应用自动配置。通常，自动配置类使用 `@ConditionalOnClass` 和 `@ConditionalOnMissingBean` 注释。这确保了只有在找到相关的类和未声明自己的类时才会应用自动配置 `@Configuration`。

您可以浏览Spring 的源代码 `spring-boot-autoconfigure` 以查看 `@Configuration` 它提供的类（请参阅 `META-INF/spring.factories` 文件）。

46.2 定位自动配置候选

Spring Boot将检查 `META-INF/spring.factories` 发布的jar中是否存在文件。该文件应该在 `EnableAutoConfiguration` 密钥下列出您的配置类，如以下示例所示：

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

```
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

如果需要按特定顺序应用您的配置，则可以使用该注释 `@AutoConfigureAfter` 或 `@AutoConfigureBefore` 注释。例如，如果您提供特定于网络的配置，则可能需要在之后应用您的课程 `WebMvcAutoConfiguration`。

如果你想订购某些不应该彼此直接了解的自动配置，你也可以使用 `@AutoConfigureOrder`。该注释具有与常规 `@Order` 注释相同的语义，但为自动配置类提供了专用的顺序。



自动配置只能以这种方式加载。确保它们是在特定的包装空间中定义的，特别是它们永远不是组件扫描的目标。

46.3 条件注释

您几乎总是希望 `@Conditional` 在自动配置类中包含一个或多个注释。该 `@ConditionalOnMissingBean` 注释是用来让开发者重写自动配置，如果他们不满意自己的缺省值一个常见的例子。

Spring Boot包含许多 `@Conditional` 注释，您可以在自己的代码中重复使用批注 `@Configuration` 类或单个 `@Bean` 方法。这些注释包括：

- 第46.3.1节“班级条件”
- 第46.3.2节“豆条件”
- 第46.3.3节“财产状况”
- 第46.3.4节“资源条件”
- 第46.3.5节“Web应用程序条件”
- 第46.3.6节“SpEL表达条件”

46.3.1 班级条件

的 `@ConditionalOnClass` 和 `@ConditionalOnMissingClass` 注解让配置基于特定类的存在或不存在被包括在内。由于使用ASM解析了注释元数据 `value`，因此即使该类可能实际上并未出现在正在运行的应用程序类路径中，也可以使用该属性来引用真实类。`name` 如果您希望通过使用 `String` 值指定类名称，也可以使用该属性。



如果您使用 `@ConditionalOnClass` 或 `@ConditionalOnMissingClass` 作为元注释的一部分来编写您自己构成的注释，则必须 `name` 在此类情况下使用引用该类的方式处理。

46.3.2 豆条件

根据特定bean的存在或不存在 `@ConditionalOnBean`，`@ConditionalOnMissingBean` 注释可以包含一个bean。您可以使用该 `value` 属性来按类型 `name` 指定bean 或按名称指定bean。该 `search` 属性使您可以限制 `ApplicationContext` 搜索bean时应考虑的层次结构。

放置在 `@Bean` 方法上时，目标类型默认为方法的返回类型，如下例所示：

```
@Configuration
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public MyService myService () {...}

}
```

在前面的例子中，`myService` 如果bean中没有 `MyService` 包含类型的bean，将会创建bean `ApplicationContext`。



您需要非常小心添加bean定义的顺序，因为这些条件是基于迄今为止已处理的内容进行评估的。出于这个原因，我们建议在自动配置类中仅使用 `@ConditionalOnBean` 和 `@ConditionalOnMissingBean` 注释（因为在添加任何用户定义的bean定义后，这些类会保证加载）。



`@ConditionalOnBean` 并且 `@ConditionalOnMissingBean` 不要阻止 `@Configuration` 创建类。在课堂上使用这些条件等同于 `@Bean` 用注释标记每个包含的方法。

46.3.3 财产状况

该 `@ConditionalOnProperty` 注释允许基于 Spring Environment 属性包含配置。使用 `prefix` 和 `name` 属性来指定应该检查的属性。默认情况下，任何存在且不相等的属性 `false` 都是匹配的。您还可以使用 `havingValue` 和 `matchIfMissing` 属性创建更高级的检查。

46.3.4 资源条件

该 `@ConditionalOnResource` 注解让配置被包括仅当特定资源是否存在。资源可以使用通常的 Spring 约定来指定，如以下示例所示：`file:/home/user/test.dat`。

46.3.5 Web 应用程序条件

在 `@ConditionalOnWebApplication` 和 `@ConditionalOnNotWebApplication` 注释，让配置取决于应用程序是否是一个“Web 应用程序”被包括在内。Web 应用程序是使用 Spring 的任何应用程序 `WebApplicationContext`，定义 `session` 范围或具有 `StandardServletEnvironment`。

46.3.6 SpEL 表达条件

该 `@ConditionalOnExpression` 注释可以根据 SpEL 表达式的结果包含配置。

46.4 测试您的自动配置

自动配置可能受多种因素影响：用户配置（`@Bean` 定义和 `Environment` 定制），条件评估（存在特定库）等。具体而言，每个测试都应创建一个定义良好的 `ApplicationContext` 代表这些自定义组合的组合。`ApplicationContextRunner` 提供了一个很好的方法来实现。

`ApplicationContextRunner` 通常定义为测试类的字段以收集基础，通用配置。以下示例确保 `UserServiceAutoConfiguration` 始终调用它：

```
private final ApplicationContextRunner contextRunner = new ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class));
```



如果必须定义多个自动配置，则不需要对它们的声明进行排序，因为它们的调用顺序与运行应用程序时的顺序完全相同。

每个测试都可以使用跑步者来表示特定的用例。例如，下面的示例调用用户配置（`UserConfiguration`）并检查自动配置是否正确退出。调用 `run` 提供了一个可以使用的回调上下文 `Assert4J`。

```
@测试
公共 无效 defaultServiceBacksOff () {
    这个 .contextRunner.withUserConfiguration(UserConfiguration.class)
        .run ((context) -> {
            assertThat(上下文).hasSingleBean(UserService.class);
            assertThat(context.getBean(UserService.class)).isSameAs(
                context.getBean(UserConfiguration.class).myUserService());
        });
}

@Configuration
静态 类 UserConfiguration {

    @Bean
    public UserService myUserService () {
        return new UserService ("mine");
    }
}
```

也可以轻松地自定义 `Environment`，如以下示例所示：

```
@Test
public void serviceNameCanBeConfigured () {
    this.contextRunner.setPropertyValues("user.name = test123").run ((context) -> {
        assertThat(上下文).hasSingleBean(UserService.class);
        assertThat(context.getBean(UserService.class).getName()).isEqualTo("test123");
    });
}
```

46.4.1模拟Web上下文

如果您需要测试仅在Servlet或Reactive Web应用程序上下文中运行的自动配置，请分别使用 `WebApplicationContextRunner` 或 `ReactiveWebApplicationContextRunner`。

46.4.2覆盖类路径

也可以测试在运行时不存在特定类和/或包时会发生什么情况。Spring Boot附带一个 `FilteredClassLoader` 可以被跑步者轻松使用的装备。在以下示例中，我们断言如果 `UserService` 不存在，则会正确禁用自动配置：

```
@测试
公共 无效 serviceIsIgnoredIfLibraryIsNotPresent () {
    这个 .contextRunner.withClassLoader (新 FilteredClassLoader (UserService. 类) )
        .run ( (context) -> assertThat (context) .doesNotHaveBean ("userService") );
}
```

46.5创建您自己的启动器

一个库的完整Spring Boot启动程序可能包含以下组件：

- `autoconfigure` 包含自动配置代码的模块。
- 为 `starter` 模块提供依赖关系的 `autoconfigure` 模块以及库和任何通常有用的附加依赖项。简而言之，添加启动器应该提供开始使用该库所需的一切。



如果你不需要分开这两个问题，你可以将自动配置代码和依赖管理结合在一个模块中。

46.5.1命名

您应该确保为您的初学者提供适当的名称空间。`spring-boot` 即使你使用不同的Maven，也不要使用你的模块名称 `groupId`。我们可能会为您将来自动配置的东西提供官方支持。

作为一个经验法则，你应该在起动器之后命名一个组合模块。例如，假设您正在为“acme”创建一个启动器，并且指定了自动配置模块 `acme-spring-boot-autoconfigure` 和启动器 `acme-spring-boot-starter`。如果您只有一个模块组合了这两个模块，请命名为 `acme-spring-boot-starter`。

另外，如果您的初学者提供配置密钥，请为它们使用唯一的名称空间。特别是，不包括你在春天开机使用的命名空间键（如 `server`，`management`，`spring`，等）。如果您使用相同的命名空间，我们可能会以破坏模块的方式修改这些命名空间。

确保 触发元数据生成，以便IDE的帮助也可用于您的密钥。您可能需要查看生成的元数据 (`META-INF/spring-configuration-metadata.json`) 以确保您的密钥已正确记录。

46.5.2 `autoconfigure` 模块

该 `autoconfigure` 模块包含开始使用库所需的一切。它还可能包含配置密钥定义（例如 `@ConfigurationProperties`）和任何可用于进一步自定义组件初始化方式的回调接口。



您应该将库的依赖关系标记为可选，以便您可以 `autoconfigure` 更轻松地将模块包含在项目中。如果你这样做，库不会提供，并且默认情况下，Spring Boot会退出。

46.5.3入门模块

起动器真的是一个空罐子。它唯一的目的是提供必要的依赖关系来与库一起工作。您可以将其视为对开始所需要的自己的看法。

不要对添加启动器的项目做出假设。如果您自动配置的库通常需要其他启动器，请提及它们。如果可选依赖关系的数量很高，那么提供一组适当的默认依赖关系可能会很困难，因为您应该避免包含对库的典型用法不必要的依赖关系。换句话说，你不应该包含可选的依赖关系。



无论哪种方式，您的初学者必须 `spring-boot-starter` 直接或间接引用核心Spring Boot starter（即，如果您的初学者依赖另一个初学者，则无需添加它）。如果一个项目只使用您的自定义启动器创建，则Spring Boot的核心功能将因核心启动器的存在而受到尊重。

47. Kotlin的支持

Kotlin是一种针对JVM（和其他平台）的静态类型语言，它允许编写简洁优雅的代码，同时提供与用Java编写的现有库的互操作性。

Spring Boot通过利用其他Spring项目（如Spring Framework，Spring Data和Reactor）中的支持来提供Kotlin支持。有关更多信息，请参阅Spring Framework Kotlin支持文档。

从Spring Boot和Kotlin开始的最简单的方法是通过start.spring.io创建一个项目。如果您需要支持，请随意加入Kotlin Slack的#spring频道，或者在Stack Overflow上提问[spring](#)并[kotlin](#)标记。

47.1要求

Spring Boot支持Kotlin 1.2.x。要使用Kotlin，`org.jetbrains.kotlin:kotlin-stdlib`并且`org.jetbrains.kotlin:kotlin-reflect`必须存在于类路径中。该`kotlin-stdlib`变种`kotlin-stdlib-jdk7`和`kotlin-stdlib-jdk8`也可以使用。

由于Kotlin类默认是最终的，因此您可能希望配置[kotlin-spring](#)插件以自动打开Spring注释的类，以便它们可以被代理。

Jackson的Kotlin模块对于在Kotlin中序列化/反序列化JSON数据是必需的。它在类路径中找到时会自动注册。如果Jackson和Kotlin存在，但Jackson Kotlin模块不存在，则会记录一条警告消息。



如果在start.spring.io上引导Kotlin项目，则会默认提供这些依赖项和插件。

47.2无安全

Kotlin的主要特点之一是无效安全。它[null](#)在编译时处理值，而不是将问题推迟到运行时并遇到一个[NullPointerException](#)。这有助于消除常见的错误来源，而无需支付类似包装的费用[Optional](#)。Kotlin还允许使用此综合指南中描述的具有可为空值的功能性结构以在Kotlin中实现无效安全。

虽然Java不允许在类型系统中表示空安全性，但Spring Framework，Spring Data和Reactor现在通过易于使用工具的注释提供了API的无安全性。默认情况下，Kotlin中使用的Java API的类型被识别为放宽null检查的平台类型。Kotlin对JSR 305注释和可空性注释的支持为Kotlin中的相关Spring API提供了无效安全性。

可以通过添加`-Xjsr305`带有以下选项的编译器标志来配置JSR 305检查：`-Xjsr305={strict|warn|ignore}`。默认行为与`-Xjsr305=warn`相同。该`strict`值必须在考虑采取从春季API推断科特林类型的空安全，但应与知识使用的春天API可空声明甚至次要版本和更多的检查可能会在将来添加之间的进化。

警告：通用类型参数，可变参数和数组元素可空性尚不支持。有关最新信息，请参阅[SPR-15942](#)。另外请注意，Spring Boot自己的API尚未注释。

47.3 Kotlin API

47.3.1 runApplication

Spring Boot提供了一种惯用的方式来运行应用程序，`runApplication<FooApplication>(*args)`如以下示例所示：

```
导入org.springframework.boot.autoconfigure.SpringBootApplication
导入org.springframework.boot.runApplication

@SpringBootApplication
FooApplication类

fun main(args: Array <String>) {
    runApplication <FooApplication> (*参数)
}
```

这是一个直接替换`SpringApplication.run(FooApplication::class.java, *args)`。它还允许定制应用程序，如以下示例所示：

```
runApplication <FooApplication> (* args) {
    setBannerMode(OFF)
}
```

47.3.2扩展

Kotlin 扩展提供了使用附加功能扩展现有类的功能。Spring Boot Kotlin API利用这些扩展来为现有API添加新的Kotlin特定便利。

`TestRestTemplate` 提供了类似于Spring框架`RestOperations`在Spring框架中提供的扩展。除此之外，这些扩展可以充分利用Kotlin的通用类型参数。

47.4 依赖管理

为了避免在类路径中混合使用不同版本的Kotlin依赖项，提供了以下Kotlin依赖项的依赖项管理：

- `kotlin-reflect`
- `kotlin-runtime`
- `kotlin-stdlib`
- `kotlin-stdlib-jdk7`
- `kotlin-stdlib-jdk8`
- `kotlin-stdlib-jre7`
- `kotlin-stdlib-jre8`

通过Maven，Kotlin版本可以通过`kotlin.version`属性进行定制，并提供插件管理`kotlin-maven-plugin`。使用Gradle时，Spring Boot插件会自动将`kotlin.version` Kotlin插件的版本对齐。

47.5 `@ConfigurationProperties`

`@ConfigurationProperties` 目前只适用于`lateinit`可空`var` 属性（前者是推荐的），因为由构造函数初始化的不可变类尚不受支持。

```
@ConfigurationProperties ("example.kotlin")
class KotlinExampleProperties {

    lateinit var foo1: String

    lateinit var foo2: String

    lateinit val bar = Bar ()

    班级酒吧{

        lateinit var bar1: String

        lateinit var bar2: String

    }

}
```

47.6 测试

尽管可以使用JUnit 4（缺省提供的`spring-boot-starter-test`）来测试Kotlin代码，但推荐使用JUnit 5。JUnit 5使测试类可以实例化一次，并可以重用于所有类的测试。这使得在非静态方法上使用`@BeforeAll`和`@AfterAll`注释成为可能，这非常适合Kotlin。

要使用JUnit 5，排除`junit:junit`依赖关系`spring-boot-starter-test`，添加JUnit 5依赖关系，并相应地配置Maven或Gradle插件。有关更多详细信息，请参阅 JUnit 5 文档。您还需要将 测试实例生命周期切换为“每类”。

47.7 资源

47.7.1 进一步阅读

- [Kotlin语言参考](#)
- [Kotlin Slack](#) (带有专门的#spring频道)
- [带有spring 和 kotlin 标签的Stackoverflow](#)
- [在您的浏览器中尝试Kotlin](#)
- [Kotlin博客](#)
- [真棒Kotlin](#)
- [用Kotlin开发Spring Boot应用程序](#)
- [一个带Kotlin，Spring Boot和PostgreSQL的地理空间信使](#)

- 在Spring Framework 5.0中引入Kotlin支持
- Spring框架5 Kotlin API的功能性方式

47.7.2例子

- `spring-boot-kotlin-demo`：定期的Spring Boot + Spring Data JPA项目
- `mixit`：Spring Boot 2 + WebFlux + 反应式Spring数据MongoDB
- `spring-kotlin-fullstack`：WebFlux Kotlin fullstack示例，用于前端的Kotlin2js而不是JavaScript或TypeScript
- `spring-petclinic-kotlin`：Spring PetClinic示例应用程序的Kotlin版本
- `spring-kotlin-deepdive`：Boot 1.0 + Java一步一步迁移到Boot 2.0 + Kotlin

48.下一步阅读什么

如果您想了解本节中讨论的任何类的更多信息，可以查看Spring Boot API文档，也可以直接浏览 源代码。如果您有具体问题，请参阅 操作指南 部分。

如果您对Spring Boot的核心功能感到满意，您可以继续阅读并了解有关生产就绪功能的信息。

第五部分Spring Boot Actuator：生产就绪功能

Spring Boot包含许多附加功能，可帮助您在将应用程序投入生产时监视和管理应用程序。您可以选择使用HTTP端点或JMX来管理和监控您的应用程序。审计，健康和指标收集也可以自动应用于您的应用程序。

49.启用生产就绪功能

该 `spring-boot-actuator` 模块提供了Spring Boot的所有生产就绪功能。启用这些功能的最简单方法是向 `spring-boot-starter-actuator` “Starter” 添加依赖项。

执行器的定义

致动器是制造术语，是指用于移动或控制某物的机械装置。执行器可以从一个小的变化中产生大量的运动。

要将执行器添加到基于Maven的项目中，请添加以下‘Starter’依赖项：

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

对于Gradle，请使用以下声明：

```
依赖关系{
    编译 ("org.springframework.boot: spring-boot-starter-actuator")
}
```

50.终点

执行器端点允许您监控应用程序并与之进行交互。Spring Boot包含许多内置端点，并允许您添加自己的端点。例如，`health` 端点提供基本的应用程序健康信息。

每个端点都可以启用或禁用。这控制着端点是否被创建，并且它的bean是否存在于应用程序上下文中。要远程访问端点，还必须通过JMX或HTTP进行公开。大多数应用程序选择HTTP，其中端点的ID与前缀一起 `/actuator` 映射到URL。例如，默认情况下，`health` 端点被映射到 `/actuator/health`。

以下与技术无关的端点可用：

ID	描述	默认启用
<code>auditevents</code>	公开当前应用程序的审计事件信息。	是
<code>beans</code>	显示应用程序中所有Spring bean的完整列表。	是
<code>conditions</code>	显示在配置和自动配置类上评估的条件以及他们做了或不匹配的原因。	是
<code>configprops</code>	显示所有的整理列表 <code>@ConfigurationProperties</code> 。	是
<code>env</code>	公开来自Spring的属性 <code>ConfigurableEnvironment</code> 。	是
<code>flyway</code>	显示已应用的所有Flyway数据库迁移。	是
<code>health</code>	显示应用健康信息。	是
<code>httptrace</code>	显示HTTP跟踪信息（默认情况下为最后100个HTTP请求 - 响应交换）。	是
<code>info</code>	显示任意的应用信息。	是
<code>loggers</code>	显示和修改应用程序中记录器的配置。	是
<code>liquibase</code>	显示已应用的任何Liquibase数据库迁移。	是
<code>metrics</code>	显示当前应用程序的“指标”信息。	是
<code>mappings</code>	显示所有 <code>@RequestMapping</code> 路径的整理列表。	是
<code>scheduledtasks</code>	显示应用程序中的计划任务。	是
<code>sessions</code>	允许从Spring会话支持的会话存储中检索和删除用户会话。使用Spring Session对反应性Web应用程序的支持时不可用。	是
<code>shutdown</code>	让应用程序正常关机。	没有
<code>threaddump</code>	执行线程转储。	是

如果您的应用程序是一个Web应用程序（Spring MVC，Spring WebFlux或Jersey），则可以使用以下附加端点：

ID	描述	默认启用
<code>heapdump</code>	返回一个GZip压缩 <code>hprof</code> 堆转储文件。	是
<code>jolokia</code>	通过HTTP公开JMX bean（当Jolokia在类路径上时，不可用于WebFlux）。	是
<code>logfile</code>	返回日志文件的内容（如果 <code>logging.file</code> 或 <code>logging.path</code> 属性已设置）。支持使用HTTP <code>Range</code> 头来检索部分日志文件的内容。	是
<code>prometheus</code>	以可以被Prometheus服务器抓取的格式显示指标。	是

要详细了解执行器的端点及其请求和响应格式，请参阅单独的API文档（HTML或PDF）。

50.1启用端点

默认情况下，除了以外的所有端点`shutdown`都已启用。要配置端点的启用，请使用其`management.endpoint.<id>.enabled`属性。以下示例启用`shutdown`端点：

```
management.endpoint.shutdown.enabled = true
```

如果您希望端点启用是选择加入而不是选择退出，请将 `management.endpoints.enabled-by-default` 属性设置为 `false` 并使用各个端点 `enabled` 属性重新加入。以下示例启用 `info` 端点并禁用所有其他端点：

```
management.endpoints.enabled-by-default = false
management.endpoint.info.enabled = true
```



禁用的端点将从应用程序上下文中完全删除。如果您只想更改端点所暴露的技术，请改用 `include` 和 `exclude` 属性。

50.2 暴露端点

由于端点可能包含敏感信息，因此应仔细考虑何时公开它们。下表显示了内置端点的默认曝光：

ID	JMX	卷筒纸
<code>auditevents</code>	是	没有
<code>beans</code>	是	没有
<code>conditions</code>	是	没有
<code>configprops</code>	是	没有
<code>env</code>	是	没有
<code>flyway</code>	是	没有
<code>health</code>	是	是
<code>heapdump</code>	N / A	没有
<code>httptrace</code>	是	没有
<code>info</code>	是	是
<code>jolokia</code>	N / A	没有
<code>logfile</code>	N / A	没有
<code>loggers</code>	是	没有
<code>liquibase</code>	是	没有
<code>metrics</code>	是	没有
<code>mappings</code>	是	没有
<code>prometheus</code>	N / A	没有
<code>scheduledtasks</code>	是	没有
<code>sessions</code>	是	没有
<code>shutdown</code>	是	没有
<code>threaddump</code>	是	没有

要更改公开哪些端点，请使用以下技术特性 `include` 和 `exclude` 属性：

属性	默认

属性**默认**`management.endpoints.jmx.exposure.exclude``management.endpoints.jmx.exposure.include`

*

`management.endpoints.web.exposure.exclude``management.endpoints.web.exposure.include`

info, health

该 `include` 属性列出了公开的端点的 ID。该 `exclude` 属性列出了不应该公开的端点的 ID。该 `exclude` 物业优先于 `include` 物业。两者 `include` 和 `exclude` 属性都可以使用端点 ID 列表进行配置。

例如，要停止暴露在 JMX 所有端点，仅暴露 `health` 和 `info` 终点，请使用以下属性：

`management.endpoints.jmx.exposure.include = 健康, 信息`

* 可以用来选择所有端点。例如，揭露一切通过 HTTP 除了 `env` 和 `beans` 端点，请使用以下属性：

```
management.endpoints.web.exposure.include = *
management.endpoints.web.exposure.exclude = env, beans
```



* 在 YAML 中有特殊的含义，所以如果你想包含（或排除）所有的端点，一定要加引号，如下例所示：

管理:

端点:

网页:

曝光:

包括: “**”



如果您的应用程序公开公开，我们强烈建议您也 [保护您的端点](#)。



如果您想要在暴露端点时实施您自己的策略，您可以注册一个 `EndpointFilter` bean。

50.3 保护HTTP端点

您应该注意保护 HTTP 端点的方式与使用其他任何敏感网址的方式相同。如果存在 Spring Security，则使用 Spring Security 的内容协商策略默认保护端点。例如，如果您希望为 HTTP 端点配置自定义安全性，则只允许具有特定角色的用户访问它们，但 Spring Boot 提供了一些方便的 `RequestMatcher` 对象，可以与 Spring Security 结合使用。

一个典型的 Spring Security 配置可能看起来像下面的例子：

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    保护 无效配置 (HttpSecurity http) 抛出异常{
        http.requestMatcher (EndpointRequest.toAnyEndpoint () ) .authorizeRequests ()
            .anyRequest () .hasRole ("ENDPOINT_ADMIN")
            。和 ()
        .httpBasic ();
    }

}
```

前面的例子使用 `EndpointRequest.toAnyEndpoint()` 将请求与任何端点进行匹配，然后确保所有端点都具有 `ENDPOINT_ADMIN` 角色。还有其他几种匹配器方法 `EndpointRequest`。有关详细信息，请参阅 API 文档 ([HTML](#) 或 [PDF](#))。

如果您在防火墙后面部署应用程序，您可能更喜欢所有的执行器端点都可以在无需验证的情况下进行访问。您可以通过更改 `management.endpoints.web.exposure.include` 属性来完成此操作，如下所示：

`application.properties`。

`management.endpoints.web.exposure.include = *`

此外，如果存在Spring Security，则需要添加自定义安全配置，以允许对端点进行未经身份验证的访问，如以下示例所示：

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    保护 无效配置 (HttpSecurity http) 抛出异常{
        http.requestMatcher (EndpointRequest.toAnyEndpoint () ) . authorizeRequests ()
            .anyRequest () . permitAll ()
    }

}
```

50.4 配置端点

端点自动缓存响应以读取不带任何参数的操作。要配置端点缓存响应的时间量，请使用其`cache.time-to-live`属性。以下示例将`beans`端点缓存的生存时间设置为10秒：

`application.properties`。

```
management.endpoint.beans.cache.time-to-live = 10s
```



前缀`management.endpoint.<name>`用于唯一标识正在配置的端点。



在进行经过验证的HTTP请求时，将`Principal`被视为端点的输入，因此不会缓存响应。

50.5 用于执行器Web端点的超媒体

“发现页面”添加了指向所有端点的链接。“发现页面”`/actuator`默认可用。

在配置自定义管理上下文路径时，“发现页面”会自动移至`/actuator`管理上下文的根目录。例如，如果管理上下文路径是`/management`，则可以从找到发现页面`/management`。当管理上下文路径设置为时`/`，禁用发现页面以防止与其他映射发生冲突的可能性。

50.6 执行器Web端点路径

默认情况下，端点`/actuator`通过使用端点的ID在路径下通过HTTP进行公开。例如，`beans`端点被暴露`/actuator/beans`。如果要将端点映射到其他路径，则可以使用该`management.endpoints.web.path-mapping`属性。另外，如果你想改变基本路径，你可以使用`management.endpoints.web.base-path`。

以下示例重新映射`/actuator/health`到`/healthcheck`：

`application.properties`。

```
management.endpoints.web.base-path = /
management.endpoints.web.path-mapping.health = healthcheck
```

50.7 CORS支持

跨源资源共享（CORS）是W3C规范，允许您以灵活的方式指定授权哪种跨域请求。如果您使用Spring MVC或Spring WebFlux，则可以配置Actuator的Web端点来支持这些场景。

CORS支持默认是禁用的，只有在`management.endpoints.web.cors.allowed-origins`属性设置后才能启用。以下配置允许`GET`和`POST`来自`example.com`域的呼叫：

```
management.endpoints.web.cors.allowed-origins = http://example.com
management.endpoints.web.cors.allowed-methods = GET, POST
```



请参阅`CorsEndpointProperties`以获取完整的选项列表。

50.8 实现自定义端点

如果添加 `@Bean` 带注释 `@Endpoint`，带注释的任何方法 `@ReadOperation`，`@WriteOperation` 或 `@DeleteOperation` 自动曝光过JMX，并在 Web应用程序，通过HTTP为好。可以使用Jersey，Spring MVC或Spring WebFlux通过HTTP公开端点。

您也可以使用 `@JmxEndpoint` 或 编写技术特定的端点 `@WebEndpoint`。这些端点仅限于各自的技术。例如，`@WebEndpoint` 只通过HTTP而不通过JMX公开。

您可以使用 `@EndpointWebExtension` 和 编写技术特定的扩展 `@EndpointJmxExtension`。这些注释可让您提供技术特定的操作，以增强现有端点。

最后，如果您需要访问特定于Web框架的功能，则可以实现Servlet或Spring `@Controller` 和 `@RestController` 端点，但代价是它们不能通过 JMX或使用其他Web框架提供。

50.8.1 接收输入

端点上的操作通过参数接收输入。当通过网络公开时，这些参数的值取自URL的查询参数和JSON请求主体。通过JMX公开时，参数将映射到 MBean操作的参数。参数是默认需要的。可以通过对它们进行注释来使它们成为可选项 `@org.springframework.lang.Nullable`。



为了使输入映射到操作方法的参数，实现端点的代码应该被编译 `-parameters`。如果您使用的是Spring Boot的Gradle插件，或者如果您使用的是Maven，则会自动发生 `spring-boot-starter-parent`。

输入类型转换

传递给端点操作方法的参数在必要时会自动转换为所需的类型。在调用操作方法之前，通过JMX或HTTP请求接收的输入将使用一个实例转换为所需的类型 `ApplicationConversionService`。

50.8.2 自定义Web端点

在操作 `@Endpoint`，`@WebEndpoint` 或 `@WebEndpointExtension` 使用新泽西州，Spring MVC的，或Spring WebFlux自动曝光通过HTTP。

Web端点请求谓词

一个请求谓词会自动为网络暴露端点上的每个操作生成。

路径

谓词的路径由端点的ID和Web暴露端点的基本路径决定。默认的基本路径是 `/actuator`。例如，具有ID的端点 `sessions` 将 `/actuator/sessions` 用作谓词中的路径。

可以通过使用注释操作方法的一个或多个参数来进一步定制路径 `@Selector`。这样的参数作为路径变量添加到路径谓词中。当调用端点操作时，该变量的值被传递给操作方法。

HTTP方法

谓词的HTTP方法由操作类型决定，如下表所示：

手术	HTTP方法
<code>@ReadOperation</code>	<code>GET</code>
<code>@WriteOperation</code>	<code>POST</code>
<code>@DeleteOperation</code>	<code>DELETE</code>

消费

对于使用请求主体的 `@WriteOperation` (HTTP `POST`)，谓词的 `consumes` 子句 是 `application/vnd.spring-boot.actuator.v2+json, application/json`。对于所有其他操作，消费条款是空的。

产生

的产生谓词子句可以由被确定 `produces` 的属性 `@DeleteOperation`，`@ReadOperation` 和 `@WriteOperation` 注解。该属性是可选的。如果未使用，则自动确定产生子句。

如果操作方法返回 `void` 或者 `Void` produce子句为空。如果操作方法返回 a `org.springframework.core.io.Resource` , 则生成子句为 `application/octet-stream`。对于所有其他操作，生产条款是 `application/vnd.spring-boot.actuator.v2+json, application/json`。

Web端点响应状态

端点操作的默认响应状态取决于操作类型（读取，写入或删除）以及操作返回的内容（如果有的话）。

A `@ReadOperation` 返回一个值，响应状态将为200（OK）。如果它没有返回值，则响应状态将为404（未找到）。

如果a `@WriteOperation` 或 `@DeleteOperation` 返回值，则响应状态将为200（OK）。如果它没有返回值，则响应状态将为204（无内容）。

如果调用没有必需参数的操作，或者使用无法转换为所需类型的参数，则不会调用操作方法，响应状态将为400（错误请求）。

Web端点范围请求

HTTP范围请求可用于请求部分HTTP资源。当使用Spring MVC或Spring Web Flux时，返回 `org.springframework.core.io.Resource` 自动支持范围请求的操作。



使用Jersey时不支持范围请求。

Web端点安全

对Web端点或Web特定的端点扩展的操作可以接收当前 `java.security.Principal` 或 `org.springframework.boot.actuate.endpoint.SecurityContext` 作为方法参数。前者通常 `@Nullable` 与认证用户和未认证用户一起使用以提供不同的行为。后者通常用于使用其 `isUserInRole(String)` 方法执行授权检查。

50.8.3 Servlet端点

阿 `Servlet` 可以公开为通过实施与注释的一个类的端点 `@ServletEndpoint` 也实现 `Supplier<EndpointServlet>`。Servlet端点提供了与Servlet容器的更深层次的集成，但是具有可移植性。它们旨在用于揭示现有 `Servlet` 的端点。对于新的端点，在 `@Endpoint` 和 `@WebEndpoint` 注释应该是首选只要有可能。

50.8.4 控制器端点

`@ControllerEndpoint` 并且 `@RestControllerEndpoint` 可以用于实现仅由Spring MVC或Spring WebFlux公开的端点。使用标准注释Spring MVC和Spring WebFlux注释（例如 `@RequestMapping` 和）来映射方法 `@GetMapping`，并将端点ID用作路径的前缀。控制器端点提供了与Spring的Web框架的更深层次的集成，但代价是可移植性的。的 `@Endpoint` 和 `@WebEndpoint` 注解应当优选只要有可能。

50.9 健康信息

您可以使用健康信息来检查正在运行的应用程序的状态。当生产系统停机时，它经常被监控软件用来提醒某人。`health` 端点公开的信息取决于 `management.endpoint.health.show-details` 可以使用以下值之一配置的属性：

名称	描述
<code>never</code>	细节永远不会显示。
<code>when-authorized</code>	详细信息仅向授权用户显示。授权角色可以使用配置 <code>management.endpoint.health.roles</code> 。
<code>always</code>	详细信息显示给所有用户。

默认值是 `never`。当用户处于一个或多个端点角色时，它被认为是被授权的。如果端点没有配置角色（默认），则认为所有经过身份验证的用户均被授权。角色可以使用 `management.endpoint.health.roles` 属性进行配置。



如果您保护了您的应用程序并希望使用 `always`，则您的安全配置必须允许经过身份验证的用户和未经身份验证的用户访问健康端点。

健康信息从您的所有 `HealthIndicator` bean中收集 `ApplicationContext`。Spring Boot包含一些自动配置的 `HealthIndicators`，你也可以自己编写。默认情况下，最终的系统状态是由 `HealthAggregator`，根据状态 `HealthIndicator` 的有序列表对每个状态进行排序。排序列表中的第一个状态用作整体健康状态。如果没有 `HealthIndicator` 返回一个已知状态 `HealthAggregator`，一种 `UNKNOWN` 是使用状态。

50.9.1 自动配置的HealthIndicators

以下 `HealthIndicator` 情况在适当时由 Spring Boot 自动配置：

名称	描述
<code>CassandraHealthIndicator</code>	检查 Cassandra 数据库是否启动。
<code>DiskSpaceHealthIndicator</code>	检查磁盘空间不足。
<code>DataSourceHealthIndicator</code>	检查是否可以获得连接 <code>DataSource</code> 。
<code>ElasticsearchHealthIndicator</code>	检查 Elasticsearch 集群是否启动。
<code>InfluxDbHealthIndicator</code>	检查 InfluxDB 服务器是否启动。
<code>JmsHealthIndicator</code>	检查 JMS 代理是否启动。
<code>MailHealthIndicator</code>	检查邮件服务器是否启动。
<code>MongoHealthIndicator</code>	检查 Mongo 数据库是否启动。
<code>Neo4jHealthIndicator</code>	检查 Neo4j 服务器是否启动。
<code>RabbitHealthIndicator</code>	检查 Rabbit 服务器是否启动。
<code>RedisHealthIndicator</code>	检查 Redis 服务器是否启动。
<code>SolrHealthIndicator</code>	检查 Solr 服务器是否已启动。



您可以通过设置 `management.health.defaults.enabled` 属性来禁用它们。

50.9.2 编写自定义健康指示器

要提供自定义健康信息，您可以注册实现该 `HealthIndicator` 接口的 Spring bean。您需要提供 `health()` 方法的实现并返回 `Health` 响应。的 `Health` 响应应该包括一个状态，并且可以任选地包括另外的细节被显示。以下代码显示了一个示例 `HealthIndicator` 实现：

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

@Component
public class MyHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // 执行一些特定的健康检查
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }
}
```



给定的标识符 `HealthIndicator` 是没有 `HealthIndicator` 后缀的 bean 的名称（如果存在）。在前面的示例中，健康信息在名为的条目中可用 `my`。

除了 Spring Boot 的预定义 `Status` 类型之外，还可以 `Health` 返回 `Status` 代表新系统状态的自定义。在这种情况下，`HealthAggregator` 还需要提供接口的自定义实现，或者必须使用 `management.health.status.order` 配置属性来配置默认实现。

例如，假设您的某个实现中正在使用新 `Status` 的代码。要配置严重性顺序，请将以下属性添加到应用程序属性中：`FATAL` `HealthIndicator`

```
management.health.status.order =致命, 关闭, OUT_OF_SERVICE, UNKNOWN, UP
```

在响应中的HTTP状态代码反映总体健康状况（例如，`UP` 映射到200，而`OUT_OF_SERVICE`并`DOWN`映射到503）。如果您通过HTTP访问健康端点，则可能还需要注册自定义状态映射。例如，以下属性映射`FATAL`到503（服务不可用）：

```
management.health.status.http-mapping.FATAL = 503
```



如果你需要更多的控制，你可以定义你自己的`HealthStatusHttpMapper`bean。

下表显示了内置状态的默认状态映射：

状态	制图
下	SERVICE_UNAVAILABLE (503)
暂停服务	SERVICE_UNAVAILABLE (503)
向上	默认情况下没有映射，所以http状态是200
未知	默认情况下没有映射，所以http状态是200

50.9.3 反应性健康指标

对于被动应用程序，例如那些使用Spring WebFlux的应用程序，`ReactiveHealthIndicator`为获得应用程序运行状况提供了非阻塞协议。与传统的类似`HealthIndicator`，健康信息从您的所有`ReactiveHealthIndicator`bean中收集`ApplicationContext`。`HealthIndicator`不包含反应式API的常规bean将包含在弹性调度程序中并执行。

为了从反应式API提供自定义健康信息，您可以注册实现该`ReactiveHealthIndicator`接口的Spring bean。以下代码显示了一个示例`ReactiveHealthIndicator`实现：

```
@Component
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {

    @Override
    public Mono<Health> health() {
        return doHealthCheck() //执行一些特定的健康检查，返回一个Mono<Health>
            .onErrorResume(ex -> Mono.just(new Health.Builder().down(ex).建立()));
    }
}
```



要自动处理错误，请考虑从中延伸`AbstractReactiveHealthIndicator`。

50.9.4 自动配置的ReactiveHealthIndicators

以下`ReactiveHealthIndicators`情况在适当时候由Spring Boot自动配置：

名称	描述
<code>MongoReactiveHealthIndicator</code>	检查Mongo数据库是否启动。
<code>RedisReactiveHealthIndicator</code>	检查Redis服务器是否启动。



必要时，反应性指标取代了常规指标。此外，任何`HealthIndicator`未明确处理的都会自动换行。

50.10 应用信息

应用程序信息公开从您的所有`InfoContributor`bean中收集的各种信息`ApplicationContext`。Spring Boot包含许多自动配置的`InfoContributor`bean，您可以编写自己的。

50.10.1自动配置InfoContributors

`InfoContributor`在适当的情况下，以下Bean由Spring Boot自动配置：

名称	描述
<code>EnvironmentInfoContributor</code>	公开钥匙 <code>Environment</code> 下的任何 <code>info</code> 钥匙。
<code>GitInfoContributor</code>	如果 <code>git.properties</code> 文件可用，则显示git信息。
<code>BuildInfoContributor</code>	如果 <code>META-INF/build-info.properties</code> 文件可用，则显示构建信息。



可以通过设置`management.info.defaults.enabled`属性来禁用它们。

50.10.2自定义应用程序信息

您可以`info`通过设置`info.*`Spring属性来自定义端点公开的数据。密钥`Environment`下的所有属性都会`info`自动公开。例如，您可以将以下设置添加到您的`application.properties`文件中：

```
info.app.encoding = UTF-8
info.app.java.source = 1.8
info.app.java.target = 1.8
```



与其对这些值进行硬编码，您还可以在构建时展开信息属性。

假设你使用Maven，你可以重写前面的例子，如下所示：

```
info.app.encoding=@project.build.sourceEncoding @
info.app.java.source=@java.version @
info.app.java.target=@java.version @
```

50.10.3 Git提交信息

`info`端点的另一个有用特性是它能够`git`在构建项目时发布有关源代码库状态的信息。如果`GitProperties`豆可用，`git.branch`，`git.commit.id`，和`git.commit.time`属性暴露出来。



`GitProperties`如果`git.properties`文件在类路径的根目录中可用，则会自动配置一个bean。有关更多详细信息，请参阅“生成git信息”。

如果您想要显示完整的git信息（即完整的内容`git.properties`），请使用该`management.info.git.mode`属性，如下所示：

```
management.info.git.mode = full
```

50.10.4构建信息

如果一个`BuildProperties`bean可用，`info`端点还可以发布关于您的构建的信息。如果`META-INF/build-info.properties`文件在类路径中可用，则会发生这种情况。



Maven和Gradle插件都可以生成该文件。有关更多详细信息，请参阅“生成构建信息”。

50.10.5编写自定义InfoContributors

为了提供定制的应用程序信息，您可以注册实现该`InfoContributor`接口的Spring bean。

以下示例为`example`单个值创建条目：

```
import java.util.Collections;
import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
```

```

import org.springframework.stereotype.Component;

@零件
公共 类 ExampleInfoContributor 实现 InfoContributor {

    @Override
    public void contribution (Info.Builder builder) {
        builder.withDetail ("example",
            Collections.singletonMap ("key", "value") );
    }

}

```

如果你达到了 `info` 端点，则应该看到包含以下附加条目的响应：

```
{
    "example": {
        "key": "value"
    }
}
```

51.通过HTTP进行监控和管理

如果您正在开发Web应用程序，Spring Boot Actuator会自动配置所有已启用的端点以通过HTTP进行公开。默认约定是使用 `id` 端点的前缀 `/actuator` 为URL路径。例如，`health` 被暴露为 `/actuator/health`。



执行器本身支持Spring MVC，Spring WebFlux和Jersey。

51.1自定义管理端点路径

有时候，自定义管理端点的前缀非常有用。例如，您的应用程序可能已经 `/actuator` 用于其他目的。您可以使用该 `management.endpoints.web.base-path` 属性更改管理端点的前缀，如以下示例中所示：

```
management.endpoints.web.base-path = / manage
```

前述 `application.properties` 示例更改从端点 `/actuator/{id}` 到 `/manage/{id}`（例如，`/manage/info`）。



除非管理端口已经被配置为 通过使用不同的HTTP端口暴露端点，`management.endpoints.web.base-path` 相对于 `server.servlet.context-path`。如果 `management.server.port` 已配置，`management.endpoints.web.base-path` 则相对于 `management.server.servlet.context-path`。

51.2自定义管理服务器端口

通过使用默认的HTTP端口公开管理端点是基于云的部署的明智选择。但是，如果您的应用程序在您自己的数据中心内运行，则可能希望使用不同的HTTP端口来公开端点。

您可以设置该 `management.server.port` 属性来更改HTTP端口，如以下示例所示：

```
management.server.port = 8081
```

51.3配置特定于管理的SSL

配置为使用自定义端口时，管理服务器也可以使用各种 `management.server.ssl.*` 属性配置自己的SSL。例如，通过这样做，管理服务器可通过HTTP使用，而主应用程序使用HTTPS，如以下属性设置所示：

```

server.port = 8443
server.ssl.enabled = true
server.ssl.key-store = classpath: store.jks
server.ssl.key-password = secret
management.server.port = 8080
management.server.ssl.enabled = false

```

或者，主服务器和管理服务器都可以使用SSL，但使用不同的密钥存储区，如下所示：

```
server.port = 8443
server.ssl.enabled = true
server.ssl.key-store = classpath: main.jks
server.ssl.key-password = secret
management.server.port = 8080
management.server.ssl.enabled = true
management.server.ssl.key-store = classpath: management.jks
management.server.ssl.key-password = secret
```

51.4自定义管理服务器地址

您可以通过设置`management.server.address`属性来自定义管理端点可用的地址。如果您只想在内部网络或面向操作的网络上收听，或只收听来自网络的连接，那么这样做可能很有用`localhost`。



只有当端口与主服务器端口不同时，您才可以监听其他地址。

以下示例`application.properties`不允许远程管理连接：

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

51.5禁用HTTP端点

如果您不想通过HTTP公开端点，则可以将管理端口设置为`-1`，如以下示例所示：

```
management.server.port = -1
```

52.通过JMX进行监控和管理

Java管理扩展（JMX）提供了一个标准机制来监视和管理应用程序。默认情况下，Spring Boot将管理端点公开为`org.springframework.boot`域下的JMX MBean。

52.1自定义MBean名称

MBean的名称通常是从`id`端点生成的。例如，`health`端点被暴露为`org.springframework.boot:type=Endpoint, name=Health`。

如果您的应用程序包含多个Spring`ApplicationContext`，您可能会发现名称发生冲突。要解决此问题，可以将该`management.endpoints.jmx.unique-names`属性设置为`true`使MBean名称始终唯一。

您还可以自定义公开端点的JMX域。以下设置显示了这样做的一个示例`application.properties`：

```
management.endpoints.jmx.domain = com.example.myapp
management.endpoints.jmx.unique-names = true
```

52.2禁用JMX端点

如果您不想通过JMX公开端点，则可以将该`management.endpoints.jmx.exposure.exclude`属性设置为`*`，如以下示例中所示：

```
management.endpoints.jmx.exposure.exclude = *
```

52.3使用Jolokia进行JMX over HTTP

Jolokia是一个JMX-HTTP桥，它提供了访问JMX bean的另一种方法。要使用Jolokia，请包含依赖项`org.jolokia:jolokia-core`。例如，使用Maven，您可以添加以下依赖项：

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

的椒端点可以然后通过添加被暴露 `jolokia` 或 `*` 所述 `management.endpoints.web.exposure.include` 属性。然后您可以以 `/actuator/jolokia` 在管理HTTP服务器上使用它。

52.3.1定制Jolokia

Jolokia有许多您通常会通过设置servlet参数进行配置的设置。使用Spring Boot，您可以使用您的 `application.properties` 文件。为此，请使用前缀参数 `management.endpoint.jolokia.config.`，如以下示例中所示：

```
management.endpoint.jolokia.config.debug = true
```

52.3.2禁用Jolokia

如果您使用Jolokia但不希望Spring Boot配置它，请将 `management.endpoint.jolokia.enabled` 属性设置为 `false`，如下所示：

```
management.endpoint.jolokia.enabled = false
```

伐木者

Spring Boot Actuator includes the ability to view and configure the log levels of your application at runtime. You can view either the entire list or an individual logger's configuration, which is made up of both the explicitly configured logging level as well as the effective logging level given to it by the logging framework. These levels can be one of:

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `OFF`
- `null`

`null` indicates that there is no explicit configuration.

53.1 Configure a Logger

To configure a given logger, `POST` a partial entity to the resource's URI, as shown in the following example:

```
{
    "configuredLevel": "DEBUG"
}
```



To “reset” the specific level of the logger (and use the default configuration instead), you can pass a value of `null` as the `configuredLevel`.

54. Metrics

Spring Boot Actuator为Micrometer提供依赖管理和自动配置， Micrometer是一个支持众多监控系统的应用指标外观，其中包括：

- 奥图
- Datadog
- 神经节
- 石墨
- 辐辏
- JMX
- 新的遗物
- 普罗米修斯
- SignalFx
- 简单（内存中）
- StatsD
- 波前



要了解更多关于千分表功能的信息，请参阅其[参考文档](#)，特别是[概念部分](#)。

54.1入门

Spring Boot会自动配置一个组合[MeterRegistry](#)，并将注册表添加到它在类路径中找到的每个受支持实现的组合中。[micrometer-registry-{system}](#)在您的运行时类路径中依赖于Spring Boot来配置注册表就足够了。

大多数注册表具有共同的特征。例如，即使Micrometer注册表实现位于类路径中，您也可以禁用特定的注册表。例如，要禁用Datadog：

```
management.metrics.export.datadog.enabled = false
```

[Metrics](#)除非您明确告诉它不要：Spring Boot还会将任何自动配置的注册表添加到该类的全局静态组合注册表中。

```
management.metrics.use-global-registry = false
```

[MeterRegistryCustomizer](#)在注册表中注册任何仪表之前，您可以注册任意数量的bean以进一步配置注册表，例如应用通用标记：

```
@豆
MeterRegistryCustomizer <MeterRegistry> metricsCommonTags () {
    返回注册表 -> registry.config () .commonTags ("region", "us-east-1");
}
```

您可以通过更具体地了解泛型类型来将自定义应用于特定的注册表实现：

```
@豆
MeterRegistryCustomizer <GraphiteMeterRegistry> graphiteMetricsNamingConvention () {
    返回注册表 -> registry.config () .namingConvention (MY_CUSTOM_CONVENTION);
}
```

使用该设置，您可以注入[MeterRegistry](#)组件并注册指标：

```
@Component
公共 类 SampleBean {

    私人 决赛柜台;

    公共 SampleBean (MeterRegistry注册表) {
        this .counter = registry.counter ("received.messages");
    }

    public void handleMessage (String message) {
        this .counter.increment ();
        // 处理消息实现
    }
}
```

Spring Boot还配置内置的仪器（即[MeterBinder](#)实现），您可以通过配置或专用注释标记进行控制。

54.2支持的监测系统

54.2.1地图集

默认情况下，指标会导出到本地机器上运行的Atlas。要使用的Atlas服务器的位置可以使用：

```
management.metrics.export.atlas.uri = http://atlas.example.com:7101/api/v1/publish
```

54.2.2 Datadog

Datadog注册中心会定期将度量指标推送到datadoghq。要将度量标准导出到Datadog，必须提供您的API密钥：

```
management.metrics.export.datadog.api-key = YOUR_KEY
```

您还可以更改度量标准发送到Datadog的时间间隔：

```
management.metrics.export.datadog.step = 30s
```

54.2.3 神经节

默认情况下，度量标准将导出到本地计算机上运行的Ganglia。要使用的Ganglia服务器主机和端口可以使用：

```
management.metrics.export.ganglia.host = ganglia.example.com
management.metrics.export.ganglia.port = 9649
```

54.2.4 石墨

默认情况下，指标会导出到本地机器上运行的Graphite。要使用的Graphite服务器主机和端口可以使用：

```
management.metrics.export.graphite.host = graphite.example.com
management.metrics.export.graphite.port = 9004
```

54.2.5 流入

默认情况下，指标会导出到本地机器上运行的Influx。使用Influx服务器的位置可以使用：

```
management.metrics.export.influx.uri = http://influx.example.com: 8086
```

54.2.6 JMX

Micrometer为JMX提供了一种分层映射，主要作为便于查看本地度量标准的便捷方式。Spring Boot提供了一个默认设置HierarchicalNameMapper，用于管理维度计ID如何映射到平面分层名称。



为了控制这种行为，定义你自己的HierarchicalNameMapper bean。

54.2.7 新遗物

New Relic注册表会定期向New Relic推送度量标准。要将度量标准导出到New Relic，必须提供您的API密钥和帐户ID：

```
management.metrics.export.newrelic.api-key = YOUR_KEY
management.metrics.export.newrelic.account-id = YOUR_ACCOUNT_ID
```

您还可以更改度量标准发送到New Relic的时间间隔：

```
management.metrics.export.newrelic.step = 30s
```

54.2.8 普罗米修斯

普罗米修斯希望能够为个别应用程序实例刮取或轮询指标。Spring Boot提供了一个可用的执行机构端点，/actuator/prometheus以适当的格式显示Prometheus刮板。



该端点默认情况下不可用，并且必须公开，请参阅公开终端以了解更多详细信息。

这是一个scrape_config要添加到的示例prometheus.yml：

```
scrape_configs:
- JOB_NAME: '弹簧'
  metrics_path: '/致动器/普罗米修斯'
  static_configs:
    - 目标: [ 'HOST: PORT' ]
```

54.2.9 SignalFx

SignalFx registry pushes metrics to SignalFx periodically. To export metrics to SignalFx, your access token must be provided:

```
management.metrics.export.signalfx.access-token=YOUR_ACCESS_TOKEN
```

You can also change the interval at which metrics are sent to SignalFx:

```
management.metrics.export.signalfx.step=30s
```

54.2.10 Simple

Micrometer ships with a simple, in-memory backend that is automatically used as a fallback if no other registry is configured. This allows you to see what metrics are collected in the metrics endpoint.

The in-memory backend disables itself as soon as you're using any of the other available backend. You can also disable it explicitly:

```
management.metrics.export.simple.enabled=false
```

54.2.11 StatsD

StatsD注册表通过UDP将度量标准推向StatsD代理。默认情况下，度量标准被导出到本地机器上运行的StatsD代理程序。需要使用的StatsD代理主机和端口可以使用：

```
management.metrics.export.statsd.host = statsd.example.com
management.metrics.export.statsd.port = 9125
```

您还可以更改StatsD行协议以使用（默认为Datadog）：

```
management.metrics.export.statsd.flavor = etsy
```

54.2.12 波前

Wavefront注册表会定期向Wavefront推送指标。如果您直接将度量标准导出到Wavefront，则必须提供您的API令牌：

```
management.metrics.export.wavefront.api-token = YOUR_API_TOKEN
```

或者，您可以使用Wavefront边车或在您的环境中设置的内部代理，将度量数据转发给Wavefront API主机：

```
management.metrics.export.uri = proxy://localhost: 7828
```



如果将度量标准发布到Wavefront代理（如文档中所述），则主机必须采用该proxy://HOST:PORT格式。

您还可以更改度量标准发送到Wavefront的时间间隔：

```
management.metrics.export.wavefront.step = 30s
```

54.3 支持的度量

Spring适用时会注册以下核心指标：

- JVM指标，报告使用情况：
 - 各种内存和缓冲池
 - 有关垃圾收集的统计
 - 线程利用率
 - 加载/卸载的类数
- CPU指标
- 文件描述符度量
- Logback指标：记录每个级别记录到Logback的事件数量
- 正常运行时间度量标准：报告正常运行时间的量表和代表应用程序绝对开始时间的固定量具
- Tomcat指标
- Spring Integration指标

54.3.1 Spring MVC度量

自动配置使得能够检测由Spring MVC处理的请求。什么时候 `management.metrics.web.server.auto-time-requests` 是 `true`，这个仪器适用于所有请求。或者，设置为时，`false`可以通过添加 `@Timed` 请求处理方法来启用检测：

```
@RestController
@Timed ①
公共类 MyController {

    @GetMapping (“/api/people”)
    @Timed (extraTags = {“region”, “us-east-1”}) ②
    @Timed (value =“all.people”, LongTask = true) ③
    public List <Person> listPeople () {...}

}
```

- ① 一个控制器类，用于在控制器中的每个请求处理程序上启用计时。
- ② 一种启用单个端点的方法。如果您在课堂上拥有此功能，则这不是必需的，但可用于进一步自定义此特定终端的计时器。
- ③ 一种 `longTask = true` 为该方法启用长任务计时器的方法。长任务计时器需要单独的度量标准名称，并且可以使用短任务计时器进行堆叠。

默认情况下，使用名称生成度量标准 `http.server.requests`。该名称可以通过设置 `management.metrics.web.server.requests-metric-name` 属性进行自定义。

默认情况下，与Spring MVC相关的度量标记包含以下信息：

- `method`，请求的方法（例如，`GET` 或 `POST`）。
- `uri`，如果可能的话（例如 `/api/person/{id}`），在变量替换之前请求的URI模板。
- `status`，响应的HTTP状态码（例如 `200` 或 `500`）。
- `exception`，处理请求时抛出的任何异常的简单类名。

要定制标签，请提供一个 `@Bean` 实现 `WebMvcTagsProvider`。

54.3.2 Spring WebFlux度量

自动配置能够检测由WebFlux控制器处理的所有请求。您也可以使用助手类 `RouterFunctionMetrics` 来测试使用WebFlux功能性编程模型的应用程序。

默认情况下，指标是使用名称生成的 `http.server.requests`。您可以通过设置 `management.metrics.web.server.requests-metric-name` 属性来自定义名称。

默认情况下，基于注释的编程模型的WebFlux相关度量标记了以下信息：

- `method`，请求的方法（例如，`GET` 或 `POST`）。
- `uri`，如果可能的话（例如 `/api/person/{id}`），在变量替换之前请求的URI模板。
- `status`，响应的HTTP状态码（例如 `200` 或 `500`）。
- `exception`，处理请求时抛出的任何异常的简单类名。

要定制标签，请提供一个 `@Bean` 实现 `WebFluxTagsProvider`。

默认情况下，功能性编程模型的度量标记包含以下信息：

- `method`，请求的方法（例如，`GET` 或 `POST`）。
- `uri`，如果可能的话（例如 `/api/person/{id}`），在变量替换之前请求的URI模板。
- `status`，响应的HTTP状态码（例如 `200` 或 `500`）。

要自定义标签，请 `defaultTags` 在您的 `RouterFunctionMetrics` 实例上使用该方法。

54.3.3 RestTemplate指标

启用 `RestTemplate` 使用自动配置创建的任何工具 `RestTemplateBuilder`。也可以 `MetricsRestTemplateCustomizer` 手动应用。

默认情况下，使用名称生成度量标准 `http.client.requests`。该名称可以通过设置 `management.metrics.web.client.requests-metric-name` 属性进行自定义。

默认情况下，由检测生成的度量 `RestTemplate` 标记包含以下信息：

- `method`，请求的方法（例如，`GET` 或 `POST`）。
- `uri`，如果可能的话（例如 `/api/person/{id}`），在变量替换之前请求的URI模板。
- `status`，响应的HTTP状态码（例如 `200` 或 `500`）。
- `clientName`，这是URI的主机部分。

要定制标签，请提供一个 `@Bean` 实现 `RestTemplateExchangeTagsProvider`。其中有便利的静态功能 `RestTemplateExchangeTags`。

54.3.4 高速缓存指标

自动配置使 `Cache` 启动时所有可用的仪器具有带有前缀的度量标准 `cache`。高速缓存检测是针对一组基本指标进行标准化的。另外，缓存特定的指标也可用。

以下缓存库受支持：

- 咖啡因
- EhCache 2
- Hazelcast
- 任何符合要求的JCache (JSR-107) 实现

度量标准由缓存的名称和 `CacheManager` 从bean名称派生的名称进行标记。



只有启动时可用的缓存绑定到注册表。对于在启动阶段之后即时创建或以编程方式创建的缓存，需要进行显式注册。一个 `CacheMetricsRegistrar` bean可以让这个过程更容易。

54.3.5 数据源度量

通过自动配置，可以检测所有可用DataSource对象的名称为度量标准 `jdbc`。数据源检测会生成代表池中当前活动，最大允许和最小允许连接的量表。这些仪表中的每一个都有一个前缀为的名称 `jdbc`。

度量标准还通过 `DataSource` 基于bean名称的计算名称进行标记。

另外，Hikari特定的度量标准还会显示 `hikaricp` 前缀。每个度量都由池的名称标记（可以用其控制 `spring.datasource.name`）。

54.3.6 RabbitMQ度量

自动配置将启用所有可用RabbitMQ连接工厂的检测，其命名为 `rabbitmq`。

54.4 注册自定义指标

要注册自定义指标，请注入 `MeterRegistry` 您的组件，如以下示例所示：

```
class Dictionary {
    private final List<String> words = new CopyOnWriteArrayList<>();
    Dictionary(MeterRegistry注册表) {
        registry.gaugeCollectionSize("dictionary.size", Tags.empty(), this.words);
    }
    // ...
}
```

如果您发现您不断在组件或应用程序中使用一套度量标准，则可以将此套件封装在一个 `MeterBinder` 实现中。默认情况下，所有 `MeterBinder` bean的度量标准将自动绑定到Spring管理的 `MeterRegistry`。

54.5 自定义各个指标

如果您需要将自定义应用于特定 `Meter` 实例，则可以使用该 `io.micrometer.core.instrument.config.MeterFilter` 界面。默认情况下，所有的 `MeterFilter` 豆将被自动应用到千分尺 `MeterRegistry.Config`。

例如，如果您想要将 `mytag.region` 标记重命名 `mytag.area` 为所有以开头的仪表ID `com.example`，则可以执行以下操作：

```
@Bean
public MeterFilter renameRegionTagMeterFilter() {
    return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");
}
```

54.5.1每米性能

除了 `MeterFilter` 豆类之外，还可以使用属性以每米为单位应用有限的一组定制。每米定制适用于以给定名称开头的所有仪表ID。例如，以下内容将禁用所有以ID开头的仪表 `example.remote`

```
management.metrics.enable.example.remote = false
```

以下属性允许每米自定义：

表54.1。每米自定义

属性	描述
<code>management.metrics.enable</code>	是否拒绝发布任何指标。
<code>management.metrics.distribution.percentiles-histogram</code>	是否发布适合计算可聚合（跨维）百分比近似值的直方图。
<code>management.metrics.distribution.percentiles</code>	发布您的应用程序中计算的百分比值
<code>management.metrics.distribution.sla</code>	用SLA定义的桶发布累积直方图。

有关概念的更多细节的背景 `percentiles-histogram`，`percentiles` 并 `sla` 请参阅“直方图和百分”部分微米文档。

54.6度量终点

Spring Boot提供了一个 `metrics` 可用于诊断的端点，用于检查应用程序收集的度量标准。该端点默认情况下不可用，并且必须公开，请参阅[公开终端](#)以了解更多详细信息。

浏览以 `/actuator/metrics` 显示可用仪表名称的列表。您可以通过提供其名称作为选择器来深入查看有关特定仪表的信息，例如 `/actuator/metrics/jvm.memory.max`。



这里使用的名称应该与代码中使用的名称相匹配，而不是命名后的名称 - 约定为其运输到的监视系统。换句话说，如果 `jvm.memory.max` 出现 `jvm_memory_max` 在普罗米修斯因为它的蛇的情况下命名约定的，还是应该 `jvm.memory.max` 作为选择检验在仪表时 `metrics` 端点。

您还可以将任意数量的 `tag=KEY:VALUE` 查询参数添加到URL的末尾，以便在度量工具上进行深度细分，例如 `/actuator/metrics/jvm.memory.max?tag=area:nonheap`。



报告的测量结果是与仪表名称匹配的所有仪表的统计数据和已经应用的所有标记的总和。因此，在上面的示例中，返回的“Value”统计量是堆的“代码缓存”，“压缩类空间”和“元空间”区域的最大内存空间总和。如果你只是想看到“Metaspace”的最大尺寸，你可以添加一个额外的 `tag=id:Metaspace`，即 `/actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace`。

55.审计

一旦Spring Security发挥作用，Spring Boot Actuator就会有一个灵活的审计框架来发布事件（默认情况下为“认证成功”，“失败”和“拒绝访问”异常）。此功能对于报告和实施基于认证失败的锁定策略非常有用。要定制已发布的安全事件，您可以提供自己的 `AbstractAuthenticationAuditListener` 和的实现 `AbstractAuthorizationAuditListener`。

您也可以将审计服务用于您自己的业务事件。为此，可以将现有 `AuditEventRepository` 注入到自己的组件中，直接使用它或 `AuditApplicationEvent` 使用Spring 发布 `ApplicationEventPublisher`（通过实现 `ApplicationEventPublisherAware`）。

56. HTTP跟踪

所有HTTP请求都会自动启用跟踪。您可以查看 `httptrace` 端点并获取有关最近100次请求 - 响应交换的基本信息。

56.1自定义HTTP跟踪

要自定义每个跟踪中包含的项目，请使用 `management.trace.http.include` 配置属性。

默认情况下，使用 `InMemoryHttpTraceRepository` 存储最近100次请求 - 响应交换的跟踪。如果你需要扩展容量，你可以定义你自己的 `InMemoryHttpTraceRepository` bean 实例。您也可以创建自己的替代 `HttpTraceRepository` 实现。

57.过程监测

在 `spring-boot` 模块中，您可以找到两个类来创建通常用于进程监视的文件：

- `ApplicationPidFileWriter` creates a file containing the application PID (by default, in the application directory with a file name of `application.pid`).
- `WebServerPortFileWriter` creates a file (or files) containing the ports of the running web server (by default, in the application directory with a file name of `application.port`).

By default, these writers are not activated, but you can enable:

- By Extending Configuration
- Section 57.2, “Programmatically”

57.1 Extending Configuration

In the `META-INF/spring.factories` file, you can activate the listener(s) that writes a PID file, as shown in the following example:

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

57.2 Programmatically

You can also activate a listener by invoking the `SpringApplication.addListeners(...)` method and passing the appropriate `Writer` object. This method also lets you customize the file name and path in the `Writer` constructor.

58. Cloud Foundry Support

Spring Boot's actuator module includes additional support that is activated when you deploy to a compatible Cloud Foundry instance. The `/cloudfoundryapplication` path provides an alternative secured route to all `@Endpoint` beans.

The extended support lets Cloud Foundry management UIs (such as the web application that you can use to view deployed applications) be augmented with Spring Boot actuator information. For example, an application status page may include full health information instead of the typical “running” or “stopped” status.



The `/cloudfoundryapplication` path is not directly accessible to regular users. In order to use the endpoint, a valid UAA token must be passed with the request.

58.1 Disabling Extended Cloud Foundry Actuator Support

If you want to fully disable the `/cloudfoundryapplication` endpoints, you can add the following setting to your `application.properties` file:

`application.properties.`

```
management.cloudfoundry.enabled=false
```

58.2 Cloud Foundry Self-signed Certificates

By default, the security verification for `/cloudfoundryapplication` endpoints makes SSL calls to various Cloud Foundry services. If your Cloud Foundry UAA or Cloud Controller services use self-signed certificates, you need to set the following property:

`application.properties.`

```
management.cloudfoundry.skip-ssl-validation=true
```

58.3 Custom context path

如果服务器的上下文路径已配置为其他任何内容 /，那么Cloud Foundry端点将无法在应用程序的根目录中使用。例如，如果 `server.servlet.context-path=/foo` Cloud Foundry端点将可用于 `/foo/cloudfoundryapplication/*`。

如果您希望 `/cloudfoundryapplication/*` 无论服务器的上下文路径如何，Cloud Foundry端点都始终处于可用状态，则需要在应用程序中明确配置该端点。根据使用的网络服务器，配置会有所不同。对于Tomcat，可以添加以下配置：

```

@Bean
public TomcatServletWebServerFactory servletWebServerFactory () {
    return new TomcatServletWebServerFactory () {

        @Override
        保护 无效 prepareContext (主机主机,
            ServletContextInitializer [] 初始化程序) {
            super .prepareContext (host, initializers) ;
            StandardContext child = new StandardContext () ;
            child.addLifecycleListener (new Tomcat.FixContextListener () ) ;
            child.setPath ("/" cloudfoundryapplication") ;
            ServletContainerInitializer initializer = getServletContextInitializer (
                getContextPath () ) ;
            child.addServletContainerInitializer (initializer, Collections.emptySet () ) ;
            child.setCrossContext (真) ;
            host.addChild (孩子) ;
        }

    };

}

private ServletContainerInitializer getServletContextInitializer (String contextPath) {
    return (c, context) -> {
        Servlet servlet = new GenericServlet () {

            @Override
            public void service (ServletRequest req, ServletResponse res)
                throws ServletException, IOException {
                ServletContext context = req.getServletContext ()
                    .getContext (contextPath中) ;
                context.getRequestDispatcher ("/" cloudfoundryapplication") .forward (req,
                    res) ;
            }

        };
        context.addServlet ("cloudfoundry", servlet) .addMapping ("/*") ;
    };
}
}

```

59.接下来要读什么

如果你想探索本章讨论的一些概念，你可以看一下执行器的示例应用。您也可能想阅读Graphite等图形工具。

否则，您可以继续阅读“部署选项”，或者继续阅读关于Spring Boot [构建工具插件](#)的深入信息。

第六部分。部署Spring Boot应用程序

Spring Boot的灵活打包选项在部署应用程序时提供了大量选择。您可以将Spring Boot应用程序部署到各种云平台，容器映像（如Docker）或虚拟/真实机器。

本节介绍一些更常见的部署方案。

60.部署到云

Spring Boot's executable jars are ready-made for most popular cloud PaaS (Platform-as-a-Service) providers. These providers tend to require that you “bring your own container”. They manage application processes (not Java applications specifically), so they need an intermediary layer that adapts *your* application to the *cloud's* notion of a running process.

Two popular cloud providers, Heroku and Cloud Foundry, employ a “buildpack” approach. The buildpack wraps your deployed code in whatever is needed to *start* your application. It might be a JDK and a call to `java`, an embedded web server, or a full-fledged application server. A buildpack is pluggable, but ideally you should be able to get by with as few customizations to it as possible. This reduces the footprint of functionality that is not under your control. It minimizes divergence between development and production environments.

Ideally, your application, like a Spring Boot executable jar, has everything that it needs to run packaged within it.

在本节中，我们将着眼于如何让我们在“入门”一节中开发并在云中运行的 [简单应用程序](#)。

60.1 Cloud Foundry

如果没有指定其他buildpack，Cloud Foundry会提供默认构建包。Cloud Foundry Java buildpack对Spring应用程序（包括Spring Boot）提供了出色的 support。您可以部署独立的可执行jar应用程序以及传统的 `.war` 打包应用程序。

一旦构建了应用程序（例如，通过使用 `mvn clean package`）并安装了 `cf` 命令行工具，则可以使用该 `cf push` 命令部署应用程序，将路径替换为编译的应用程序 `.jar`。在推送应用程序之前，请务必使用 `cf` 命令行客户端登录。以下行显示使用该 `cf push` 命令部署应用程序：

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



在前面的例子中，我们用 `acloudyspringtime` 您提供的任何值 `cf` 作为您的应用程序的名称。

有关更多选项，请参阅 `cf push` 文档。如果 `manifest.yml` 在同一目录中存在 Cloud Foundry 文件，则会考虑该文件。

此时，`cf` 开始上传您的应用程序，产生类似于以下示例的输出：

```
上传acloudyspringtime ... 确定
准备开始acloudyspringtime ... 确定
----> 下载的应用程序包 (8.9M)
----> Java Buildpack版本: v3.12 (离线) | https://github.com/cloudfoundry/java-buildpack.git#6f25b7e
----> 从https://java-buildpack.cloudfoundry.org/openjdk/trusty/x86_64/openjdk-1.8.0_121.tar.gz下载打开的Jdk JRE 1.8.0_121 (将Open Jdk JRE扩展为.java-buildpack / open_jdk_jre (1.6s)
----> 从https://java-buildpack.cloudfoundry.org/memory-calculator/trusty/x86_64/memory-calculator-2.0.2_RELEASE.tar.gz下载
    内存设置: -Xss349K -Xmx681574K -XX: MaxMetaspaceSize = 104857K -Xms681574K -XX: MetaspaceSize = 104857K
----> 从https://java-buildpack.cloudfoundry.org/container-certificate-trust-store/container-certificate-trust-store-1.0.0_0将证书添加到.java-buildpack / container_certificate_trust_store / truststore.jks (0.6s)
----> 从https://java-buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-1.10.0_RELEASE.jar下载Spring自动检查应用“acloudyspringtime”的状态...
运行1个实例中的0个 (1个启动)
...
运行1个实例中的0个 (1个启动)
...
运行1个实例中的0个 (1个启动)
...
1个正在运行的实例 (1个正在运行)

应用已启动
```

恭喜！该应用程序现在已经生效！

一旦您的应用程序处于运行状态，您可以使用该 `cf apps` 命令验证已部署的应用程序的状态，如以下示例所示：

```
$ cf apps
获取应用程序...
好

名称请求状态实例内存磁盘URL
...
acloudyspringtime 开始1/1 512M 1G acloudyspringtime.cfapps.io
...
```

一旦Cloud Foundry确认您的应用程序已部署完毕，您应该能够根据给定的URI找到该应用程序。在前面的例子中，你可以找到它 <http://acloudyspringtime.cfapps.io/>。

60.1.1 绑定到服务

默认情况下，有关正在运行的应用程序的元数据以及服务连接信息将作为环境变量（例如：`$VCAP_SERVICES`）。此架构决定归功于Cloud Foundry的多语言（任何语言和平台都可以作为buildpack支持）。进程范围的环境变量是语言不可知的。

环境变量并不总是适用于最简单的API，因此Spring Boot会自动提取它们并将数据平整为可通过Spring [Environment](#) 抽象访问的属性，如以下示例所示：

```
@Component
类 MyBean实现了 EnvironmentAware {

    private String instanceId;

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

    // ...
}
```

All Cloud Foundry properties are prefixed with `vcap`. You can use `vcap` properties to access application information (such as the public URL of the application) and service information (such as database credentials). See the '`CloudFoundryVcapEnvironmentPostProcessor`' Javadoc for complete details.



The Spring Cloud Connectors project is a better fit for tasks such as configuring a DataSource. Spring Boot includes auto-configuration support and a `spring-boot-starter-cloud-connectors` starter.

60.2 Heroku

Heroku is another popular PaaS platform. To customize Heroku builds, you provide a `Procfile`, which provides the incantation required to deploy an application. Heroku assigns a `port` for the Java application to use and then ensures that routing to the external URI works.

You must configure your application to listen on the correct port. The following example shows the `Procfile` for our starter REST application:

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

Spring Boot makes `-D` arguments available as properties accessible from a Spring `Environment` instance. The `server.port` configuration property is fed to the embedded Tomcat, Jetty, or Undertow instance, which then uses the port when it starts up. The `$PORT` environment variable is assigned to us by the Heroku PaaS.

这应该是你需要的一切。Heroku部署中最常见的部署工作流程是 `git push` 生产代码，如以下示例所示：

```
$ git push heroku master

初始化仓库, 完成。
计数对象: 95, 完成。
增量压缩使用多达8个线程。
压缩对象: 100% (78/78), 完成。
写作对象: 100% (95/95), 8.66 MiB | 606.00 KiB / s, 完成。
总计95 (增量31), 重用0 (增量0)

----->检测到Java应用程序
----->安装OpenJDK 1.8 ... 完成
----->安装Maven 3.3.1 ... 完成
----->安装settings.xml ... 完成
----->执行: mvn -B -DskipTests = true干净安装

[INFO]扫描项目...
正在下载: https://repo.spring.io / ...
已下载: https://repo.spring.io / ... (8 KB, 每秒1.8 KB)
...
已下载: http://s3pository.herokuapp.com/jvm/ ... (152 KB, 每秒595.3 KB)
[INFO]正在安装/ tmp / build_0c35a5d2-a067-4abc-a232-14b1fb7a8229 / target / ...
[INFO]正在安装/tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ...
[INFO] -----
[信息]建立成功
[INFO] -----
[信息]总时间: 59.358s
[INFO]完成时间: 星期五三月07 07:28:25 UTC 2014
[INFO]最终内存: 20M / 493M
[INFO] -----
```

```
---->发现过程类型
      Procfile声明类型 - > web

---->压缩完成, 70.4MB
---->启动完成, v6
      http://agile-sierra-1405.herokuapp.com/ 部署到Heroku

要git @ heroku.com: agile-sierra-1405.git
* [新分支]主人 - >主人
```

您的应用程序现在应该在Heroku上运行。

60.3 OpenShift

OpenShift是Kubernetes容器编排平台的红帽公共（和企业）扩展。与Kubernetes类似，OpenShift有许多用于安装基于Spring Boot的应用程序的选项。

OpenShift有许多资源来描述如何部署Spring Boot应用程序，其中包括：

- 使用S2I生成器
- 建筑指南
- 作为Wildfly上的传统Web应用程序运行
- OpenShift Commons Briefing

60.4 亚马逊网络服务 (AWS)

亚马逊网络服务提供了多种方式来安装基于Spring Boot的应用程序，既可以作为传统的Web应用程序（war），也可以作为嵌入式web服务器的可执行jar文件。选项包括：

- AWS Elastic Beanstalk
- AWS代码部署
- AWS OPS Works
- AWS云阵形
- AWS容器注册表

每个都有不同的功能和定价模式。在本文中，我们只描述最简单的选项：AWS Elastic Beanstalk。

60.4.1 AWS Elastic Beanstalk

正如 [Elastic Beanstalk Java指南中所述](#)，部署Java应用程序有两个主要选项。您可以使用“Tomcat平台”或“Java SE平台”。

使用Tomcat平台

此选项适用于生成war文件的Spring Boot项目。不需要特殊配置。你只需要遵循官方指南。

使用Java SE平台

此选项适用于生成jar文件并运行嵌入式Web容器的Spring Boot项目。Elastic Beanstalk环境在端口80上运行nginx实例，以代理在端口5000上运行的实际应用程序。要配置它，请将以下行添加到 `application.properties` 文件中：

```
server.port = 5000
```



默认情况下，Elastic Beanstalk会上传源并在AWS中编译它们。但是，最好是上传二进制文件。为此，请在 `.elasticbeanstalk/config.yml` 文件中添加类似于以下内容的行：

部署：

```
工作: target / demo-0.0.1-SNAPSHOT.jar
```



默认情况下，Elastic Beanstalk环境负载均衡。负载平衡器有很大的成本。为避免此费用，请将环境类型设置为“单实例”，如 [Amazon文档中所述](#)。您还可以使用CLI和以下命令创建单实例环境：

```
eb create -s
```

60.4.2总结

这是获得AWS最简单的方法之一，但还有更多内容需要解决，比如如何将Elastic Beanstalk集成到任何CI / CD工具中，使用Elastic Beanstalk Maven插件而不是CLI等等。有一篇[博客文章](#)详细介绍了这些主题。

60.5 Boxfuse和亚马逊网络服务

Boxfuse通过将Spring Boot可执行jar或war转换为可以在VirtualBox或AWS上无变化部署的最小VM映像来实现。Boxfuse为Spring Boot提供了深度集成，并使用Spring Boot配置文件中的信息自动配置端口和运行状况检查URL。Boxfuse将这些信息用于它生成的图像以及它提供的所有资源（实例，安全组，弹性负载均衡器等）。

创建Boxfuse账户后，将其连接到您的AWS账户，安装最新版本的Boxfuse Client，并确保该应用程序由Maven或Gradle（例如使用）构建 `mvn clean package`，您可以部署Spring使用与以下类似的命令将应用程序引导至AWS：

```
$ boxfuse运行myapp-1.0.jar -env = prod
```

有关更多选项，请参阅 `boxfuse run` 文档。如果 `boxfuse.conf` 当前目录中存在文件，则会考虑该文件。



默认情况下，Boxfuse激活一个 `boxfuse` 在启动时命名的Spring配置文件。如果您的可执行jar或war包含一个 `application-boxfuse.properties` 文件，Boxfuse将其配置基于它所包含的属性。

此时，`boxfuse` 为您的应用程序创建一个映像，上传它，并在AWS上配置和启动必要的资源，从而得到类似于以下示例的输出：

```
为myapp-1.0.jar融合图像...
图像融合在00: 06.838s (53937 K) -> axelfontaine / myapp: 1.0中
创建axelfontaine / myapp ...
推axelfontaine / myapp: 1.0 ...
正在验证axelfontaine / myapp: 1.0 ...
创建弹性IP ...
将myapp-axelfontaine.boxfuse.io映射到52.28.233.167 ...
等待AWS为eu-central-1中的axelfontaine / myapp: 1.0创建AMI（这可能需要长达50秒）...
AMI在00: 23.557s -> ami-d23f38cf中创建
创建安全组boxfuse-sg_axelfontaine / myapp: 1.0 ...
在eu-central-1中启动axelfontaine / myapp: 1.0 (ami-d23f38cf) 的t2.micro实例...
实例在00: 30.306s -> i-92ef9f53启动
等待AWS启动实例i-92ef9f53和有效负载从http://52.28.235.61/开始...
有效负载在00: 29.266s -> http://52.28.235.61/
将弹性IP 52.28.233.167重新映射到i-92ef9f53 ...
等待15秒完成Elastic IP Zero Downtime迁移...
部署已成功完成。axelfontaine / myapp: 1.0已启动并运行于http://myapp-axelfontaine.boxfuse.io/
```

您的应用程序现在应该在AWS上运行。

有关在EC2上部署Spring Boot应用程序的博文以及Boxfuse Spring Boot集成的 文档，请参阅 Maven构建以运行该应用程序。

60.6 Google Cloud

Google Cloud有几个可用于启动Spring Boot应用程序的选项。最容易入门的可能是App Engine，但您也可以找到在容器中使用Container Engine或在具有Compute Engine的虚拟机上运行Spring Boot的方法。

要在App Engine中运行，您可以先在UI中创建一个项目，为您设置唯一的标识符并设置HTTP路由。在项目中添加一个Java应用程序并将其保留为空，然后使用Google Cloud SDK将Spring Boot应用程序从命令行或CI构建推入该插槽。

App Engine需要您创建一个 `app.yaml` 文件来描述您的应用所需的资源。通常情况下，你把这个文件放进去 `src/main/appengine`，它应该类似于下面的文件：

```
服务: 默认

运行时: java
env: flex

runtime_config:
  jdk: openjdk8

处理程序:
- url: /.*
  script: 该字段是必需的，但被忽略
```

```

manual_scaling:
  实例: 1

health_check:
  enable_health_check: False

env_variables:
  ENCRYPT_KEY: your_encryption_key_here

```

您可以通过将项目ID添加到构建配置来部署应用程序（例如，使用Maven插件），如以下示例所示：

```

<plugin>
  <groupId> com.google.cloud.tools </ groupId>
  <artifactId> appengine-maven-plugin </ artifactId>
  <version> 1.3.0 </ version>
  <configuration>
    <project> myproject </ project>
  </ configuration>
</ plugin>

```

然后部署 `mvn appengine:deploy`（如果您需要首先进行身份验证，则构建失败）。



Google App Engine Classic绑定到Servlet 2.5 API，所以你不能在没有修改的情况下部署Spring应用程序。请参阅本指南的 [Servlet 2.5部分](#)。

61.安装Spring Boot应用程序

除了使用Spring Boot应用程序外 `java -jar`，还可以为Unix系统创建完全可执行的应用程序。完全可执行的jar可以像任何其他可执行的二进制文件一样执行，也可以使用 `init.d` 或注册 `systemd`。这使得在普通生产环境中安装和管理Spring Boot应用程序变得非常简单。



警告

Fully executable jars work by embedding an extra script at the front of the file. Currently, some tools do not accept this format, so you may not always be able to use this technique. For example, `jar -xf` may silently fail to extract a jar or war that has been made fully executable. It is recommended that you make your jar or war fully executable only if you intend to execute it directly, rather than running it with `java -jar` or deploying it to a servlet container.

To create a ‘fully executable’ jar with Maven, use the following plugin configuration:

```

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

```

以下示例显示了等效的Gradle配置：

```

bootJar {
    launchScript()
}

```

然后，您可以通过键入 `./my-application.jar`（其中 `my-application` 是工件的名称）来运行应用程序。包含jar的目录被用作应用程序的工作目录。

61.1支持的操作系统

默认脚本支持大多数Linux发行版，并在CentOS和Ubuntu上进行测试。其他平台，如OS X和FreeBSD，需要使用自定义 `embeddedLaunchScript`。

61.2 Unix / Linux服务

春季启动应用程序可以通过使用很容易开始作为Unix / Linux操作系统服务 `init.d` 或 `systemd`。

61.2.1作为 init.d 服务安装 (系统V)

如果您将Spring Boot的Maven或Gradle插件配置为生成完全可执行的jar，并且您不使用自定义 `embeddedLaunchScript`，则可以将您的应用程序用作 `init.d` 服务。要做到这一点，符号链接罐子 `init.d`，支持标准 `start`, `stop`, `restart`，和 `status` 命令。

该脚本支持以下功能：

- 以拥有该jar文件的用户身份启动服务
- 通过使用追踪应用程序的PID `/var/run/<appname>/<appname>.pid`
- 将控制台日志写入 `/var/log/<appname>.log`

假设您安装了Spring Boot应用程序 `/var/myapp`，要将Spring Boot应用程序安装为 `init.d` 服务，请创建一个符号链接，如下所示：

```
$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp
```

安装后，您可以按照通常的方式启动和停止服务。例如，在基于Debian的系统上，可以使用以下命令启动它：

```
$ service myapp start
```

 如果您的应用程序无法启动，请检查写入 `/var/log/<appname>.log` 的错误日志文件。

您还可以使用标准操作系统工具将应用程序标记为自动启动。例如，在Debian上，您可以使用以下命令：

```
$ update-rc.d myapp默认值<priority>
```

确保 init.d 服务



以下是关于如何保护作为init.d服务运行的Spring Boot应用程序的一组准则。它并不打算成为一个应用程序及其运行环境的完整列表。

以root身份执行时，与使用root启动init.d服务的情况相同，缺省可执行脚本以拥有该jar文件的用户身份运行应用程序。你永远不应该运行Spring Boot应用程序 `root`，所以你的应用程序的jar文件不应该被root所拥有。相反，创建一个特定的用户来运行应用程序，并使用 `chown` 它来使其成为jar文件的所有者，如下例所示：

```
$ chown bootapp: bootapp your-app.jar
```

在这种情况下，默认的可执行脚本以 `bootapp` 用户身份运行应用程序。



为了减少应用程序用户帐户受到攻击的可能性，您应该考虑防止它使用登录shell。例如，您可以将帐户的外壳设置为 `/usr/sbin/nologin`。

您还应该采取措施来防止修改应用程序的jar文件。首先，配置其权限，以便它不能被写入，只能由其所有者读取或执行，如下例所示：

```
$ chmod 500 your-app.jar
```

其次，如果您的应用程序或运行该应用程序的帐户受到威胁，您还应该采取措施来限制损害。如果攻击者获得访问权限，他们可以将jar文件写入并更改其内容。防止这种情况的一种方法是通过使用它来使其不可变 `chattr`，如以下示例所示：

```
$ sudo chattr + i your-app.jar
```

这将阻止包括root在内的任何用户修改jar。

如果root用于控制应用程序的服务，并使用 `.conf` 文件来定制其启动，则该 `.conf` 文件将由root用户读取并进行评估。它应该得到相应的保证。使用 `chmod` 以使文件只能由所有者读取并用于 `chown` 使所有者成为root，如以下示例所示：

```
$ chmod 400 your-app.conf
$ sudo chown root: root your-app.conf
```

61.2.2安装即 systemd 服务

`systemd` 是System V init系统的继任者，现在被许多现代Linux发行版所使用。虽然您可以继续使用 `init.d` 脚本 `systemd`，但也可以使用 `systemd` “服务”脚本启动Spring Boot应用程序。

假设您安装了Spring Boot应用程序 /var/myapp，要将Spring Boot应用程序安装为 `systemd` 服务，请创建一个名为脚本的脚本 `myapp.service` 并将其放在 /etc/systemd/system 目录中。以下脚本提供了一个示例：

```
[单元]
描述的myapp =
之后= syslog.target

[服务]
用户的myapp =
ExecStart =的/ var / MyApp的/ myapp.jar
SuccessExitStatus = 143

[安装]
WantedBy = multi-user.target
```

重要

请记住，改变 `Description`，`User` 和 `ExecStart` 域为您的应用。

 该 `ExecStart` 字段不声明脚本动作命令，这意味着 `run` 默认使用该命令。

请注意，与作为 `init.d` 服务运行不同，运行应用程序的用户，PID文件和控制台日志文件是由其 `systemd` 自身管理的，因此必须使用“服务”脚本中的相应字段进行配置。有关更多详细信息，请参阅 [服务单位配置手册页](#)。

要将应用程序标记为在系统引导时自动启动，请使用以下命令：

```
$ systemctl启用myapp.service
```

参考 [man systemctl](#) 更多细节。

61.2.3自定义启动脚本

由Maven或Gradle插件编写的默认嵌入式启动脚本可以通过多种方式进行自定义。对于大多数人来说，使用默认脚本以及一些自定义功能通常就足够了。如果您发现无法自定义您需要的内容，请使用该 `embeddedLaunchScript` 选项完全编写自己的文件。

在写入时自定义启动脚本

在写入jar文件时，定制启动脚本的元素通常很有意义。例如，init.d脚本可以提供“描述”。由于您知道前面的描述（并且不需要更改），因此您可以在生成jar时提供它。

要定制书写元素，请使用 `embeddedLaunchScriptProperties` Spring Boot Maven或Gradle插件的选项。

默认脚本支持以下属性替换：

名称	描述
<code>mode</code>	脚本模式。默认为 <code>auto</code> 。
<code>initInfoProvides</code>	在 <code>Provides</code> “INIT INFO”的部分。默认 <code>spring-boot-application</code> 为Gradle 和 <code> \${project.artifactId}</code> Maven。
<code>initInfoRequiredStart</code>	在 <code>Required-Start</code> “INIT INFO”的部分。默认为 <code>\$remote_fs \$syslog \$network</code> 。
<code>initInfoRequiredStop</code>	在 <code>Required-Stop</code> “INIT INFO”的部分。默认为 <code>\$remote_fs \$syslog \$network</code> 。
<code>initInfoDefaultStart</code>	在 <code>Default-Start</code> “INIT INFO”的部分。默认为 <code>2 3 4 5</code> 。
<code>initInfoDefaultStop</code>	在 <code>Default-Stop</code> “INIT INFO”的部分。默认为 <code>0 1 6</code> 。
<code>initInfoShortDescription</code>	在 <code>Short-Description</code> “INIT INFO”的部分。默认 <code>Spring Boot Application</code> 为Gradle 和 <code> \${project.name}</code> Maven。

名称	描述
<code>initInfoDescription</code>	在 <code>Description</code> “INIT INFO”的部分。默认 <code>Spring Boot Application</code> 为Gradle和Maven的 <code> \${project.description} </code> (回退 <code> \${project.name} </code>)。
<code>initInfoChkconfig</code>	在 <code>chkconfig</code> “INIT INFO”的部分。默认为 <code>2345 99 01</code> 。
<code>confFolder</code>	默认值 <code>CONF_FOLDER</code> 。默认为包含jar的文件夹。
<code>inlinedConfScript</code>	引用应在默认启动脚本中内联的文件脚本。这可以用来设置环境变量，例如 <code>JAVA_OPTS</code> 在加载任何外部配置文件之前。
<code>logFolder</code>	默认值 <code>LOG_FOLDER</code> 。仅适用于 <code>init.d</code> 服务。
<code>logFilename</code>	默认值 <code>LOG_FILENAME</code> 。仅适用于 <code>init.d</code> 服务。
<code>pidFolder</code>	默认值 <code>PID_FOLDER</code> 。仅适用于 <code>init.d</code> 服务。
<code>pidFilename</code>	中的PID文件名称的默认值 <code>PID_FOLDER</code> 。仅适用于 <code>init.d</code> 服务。
<code>useStartStopDaemon</code>	<code>start-stop-daemon</code> 命令是否可用，是否应该用来控制过程。默认为 <code>true</code> 。
<code>stopWaitTime</code>	默认值 <code>STOP_WAIT_TIME</code> 。仅适用于 <code>init.d</code> 服务。默认为60秒。

在运行时自定义脚本

对于在写入jar 之后需要定制的脚本项目，您可以使用环境变量或配置文件。

默认脚本支持以下环境属性：

变量	描述
<code>MODE</code>	操作的“模式”。默认取决于jar的构建方式，但通常 <code>auto</code> 是这样的（这意味着它通过检查它是否是所调用目录中的符号链接来尝试猜测它是否为init脚本 <code>init.d</code> ）。您可以明确地将其设置为 <code>service</code> 使 <code>stop start status restart</code> 命令正常工作，或者 <code>run</code> 如果要在前台运行该脚本。
<code>USE_START_STOP_DAEMON</code>	<code>start-stop-daemon</code> 命令是否可用，是否应该用来控制过程。默认为 <code>true</code> 。
<code>PID_FOLDER</code>	pid文件夹的根名称（ <code>/var/run</code> 默认情况下）。
<code>LOG_FOLDER</code>	放置日志文件的文件夹的名称（ <code>/var/log</code> 默认情况下）。
<code>CONF_FOLDER</code>	从中读取.conf文件的文件夹的名称（默认情况下，与jar-file相同的文件夹）。
<code>LOG_FILENAME</code>	<code>LOG_FOLDER</code> (<code><appname>.log</code> 默认情况下) 日志文件的名称。
<code>APP_NAME</code>	应用的名称。如果jar是从符号链接运行的，则脚本会猜测应用程序名称。如果它不是符号链接，或者您想显式设置应用程序名称，这可能很有用。
<code>RUN_ARGS</code>	传递给程序的参数（Spring Boot应用程序）。
<code>JAVA_HOME</code>	<code>java</code> 可执行文件的位置是 <code>PATH</code> 通过默认使用默认情况下发现的，但如果有关于可执行文件，则可以明确地设置它 <code>\$JAVA_HOME/bin/java</code> 。
<code>JAVA_OPTS</code>	启动时传递给JVM的选项。
<code>JARFILE</code>	jar文件的显式位置，以防脚本被用来启动一个实际上没有嵌入的jar。
<code>DEBUG</code>	如果不为空，则 <code>-x</code> 在shell进程上设置标志，以便于查看脚本中的逻辑。

变量**描述****STOP_WAIT_TIME**在强制关闭之前停止应用程序时等待的时间 (**60** 默认值)。

该 **PID_FOLDER** , **LOG_FOLDER** 和 **LOG_FILENAME** 变量仅适用于 **init.d** 服务。因为 **systemd** , 通过使用“服务”脚本进行等效的自定义。有关更多详细信息 , 请参阅 [服务单位配置手册页](#)。

除了 **JARFILE** 和以外 **APP_NAME** , 可以使用 **.conf** 文件来配置上一节中列出的设置。该文件预计将在jar文件旁边 , 并且具有相同的名称 , 但是后缀 **.conf** 而不是 **.jar** 。例如 , 一个名为jar的程序 **/var/myapp/myapp.jar** 使用名为的配置文件 **/var/myapp/myapp.conf** , 如以下示例所示 :

myapp.conf.

```
JAVA_OPTS = -Xmx1024M
LOG_FOLDER = /自定义/日志/文件夹
```



如果你不喜欢在jar文件旁边有配置文件 , 你可以设置一个 **CONF_FOLDER** 环境变量来定制配置文件的位置。

要了解有关适当保护此文件的信息 , 请参阅 [保护init.d服务的指导原则](#)。

61.3 Microsoft Windows服务

Spring Boot应用程序可以通过使用Windows服务启动 [winsw](#) 。

一个 ([单独维护的示例](#)) 逐步介绍如何为Spring Boot应用程序创建Windows服务。

62.接下来要读什么

查看Cloud Foundry , Heroku , OpenShift和 Boxfuse网站 , 了解有关PaaS可提供的各种功能的更多信息。这些只是四个最流行的Java PaaS提供商。由于Spring Boot非常适合基于云的部署 , 因此您可以自由考虑其他提供商。

下一节继续介绍 [Spring Boot CLI](#) , 或者您可以跳到前面阅读 [构建工具插件](#)。

第七部分。 Spring Boot CLI

Spring Boot CLI是一个命令行工具 , 如果您想快速开发Spring应用程序 , 您可以使用它。它可以让你运行Groovy脚本 , 这意味着你有一个熟悉的类Java语法 , 没有太多的样板代码。您也可以引导一个新项目或编写自己的命令。

63.安装CLI

Spring Boot CLI (命令行界面) 可以使用SDKMAN手动安装 ! (SDK管理器) , 或者如果您是OSX用户 , 则使用Homebrew或MacPorts。有关全面的安装说明 , 请参见 “入门”一节中的[第10.2节“安装Spring Boot CLI”](#)。

64.使用CLI

一旦你安装了CLI , 你可以通过 **spring** 在命令行键入并按下Enter 来运行它。如果您 **spring** 没有任何参数运行 , 则会显示一个简单的帮助屏幕 , 如下所示 :

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

可用的命令是:

```
运行[options] <files> [ - ] [args]
运行一个春天groovy脚本
```

... 此处显示更多命令帮助

您可以输入 `spring help` 以获取有关任何受支持命令的更多详细信息，如以下示例所示：

```
$ spring help run
春季运行 - 运行一个春天groovy脚本

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

选项说明
-----
--autoconfigure [布尔]添加自动配置编译器
    转换 (默认: true)
--classpath, -cp附加的类路径条目
-e, --edit使用默认系统打开文件
    编辑
--no-guess-dependencies不要试图猜测依赖关系
--no-guess-imports不要尝试猜测导入
-q, --quiet安静的日志记录
-v, --verbose详细记录依赖关系
    解析度
--watch观看指定文件的更改
```

该 `version` 命令提供了一种快速检查您正在使用的Spring Boot版本的方法，如下所示：

```
$ 春天版本
Spring CLI v2.0.1.BUILD-SNAPSHOT
```

64.1 使用CLI运行应用程序

您可以使用该 `run` 命令编译和运行Groovy源代码。Spring Boot CLI完全独立，因此您不需要任何外部Groovy安装。

以下示例显示了用Groovy编写的“hello world”Web应用程序：

`hello.groovy`。

```
@RestController
类 WebApplication {
    @RequestMapping (“/")
    String home () {
        “你好，世界！”
    }
}
```

要编译并运行该应用程序，请键入以下命令：

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

要将命令行参数传递给应用程序，请使用 `--` 以将命令与“spring”命令参数分开，如以下示例所示：

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

要设置JVM命令行参数，可以使用 `JAVA_OPTS` 环境变量，如下例所示：

```
$ JAVA_OPTS = -Xmx1024m 春季运行hello.groovy
```



`JAVA_OPTS` 在Microsoft Windows上进行设置时，请确保引用整个指令，例如 `set "JAVA_OPTS=-Xms256m -Xmx2048m"`。这样做可确保将值正确传递给流程。

64.1.1 推导出“抢”依赖

标准Groovy包含一个 `@Grab` 注释，它允许您声明对第三方库的依赖关系。这个有用的技术可以让Groovy像Maven或Gradle一样下载jar，但不需要使用构建工具。

Spring Boot进一步扩展了这种技术，并试图根据您的代码推断出哪些库需要“抓取”。例如，由于 `WebApplication` 之前显示的代码使用 `@RestController` 注释，因此 Spring Boot 抓住了“Tomcat”和“Spring MVC”。

以下项目用作“抓取提示”：

项目	抓斗
<code>JdbcTemplate</code> , <code>NamedParameterJdbcTemplate</code> , <code>DataSource</code>	JDBC应用程序。
<code>@EnableJms</code>	JMS应用程序。
<code>@EnableCaching</code>	缓存抽象。
<code>@Test</code>	JUnit的。
<code>@EnableRabbit</code>	RabbitMQ的。
<code>@EnableReactor</code>	项目反应堆。
扩展 <code>Specification</code>	Spock测试。
<code>@EnableBatchProcessing</code>	春天的批次。
<code>@MessageEndpoint</code> <code>@EnableIntegration</code>	Spring集成。
<code>@Controller</code> <code>@RestController</code> <code>@EnableWebMvc</code>	Spring MVC +嵌入式Tomcat。
<code>@EnableWebSecurity</code>	Spring Security。
<code>@EnableTransactionManagement</code>	春季交易管理。



请参阅 `CompilerAutoConfiguration` Spring Boot CLI源代码中的子类 以准确了解自定义如何应用。

64.1.2 推导出“抓取”坐标

Spring Boot `@Grab` 通过让你指定一个没有组或版本的依赖来扩展Groovy的标准支持（例如 `@Grab('freemarker')`）。这样做可以咨询Spring Boot的默认依赖关系元数据来推断工件的组和版本。



默认元数据与您使用的CLI版本相关联。它只会在您转移到CLI的新版本时才会更改，从而使您可以控制何时可能更改您的依赖项的版本。显示默认元数据中所包含的依赖关系及其版本的表可以在[附录中](#)找到。

64.1.3 默认导入语句

To help reduce the size of your Groovy code, several `import` statements are automatically included. Notice how the preceding example refers to `@Component`, `@RestController`, and `@RequestMapping` without needing to use fully-qualified names or `import` statements.



Many Spring annotations work without using `import` statements. Try running your application to see what fails before adding imports.

64.1.4 Automatic Main Method

Unlike the equivalent Java application, you do not need to include a `public static void main(String[] args)` method with your `Groovy` scripts. A `SpringApplication` is automatically created, with your compiled code acting as the `source`.

64.1.5 Custom Dependency Management

默认情况下，CLI使用`spring-boot-dependencies`解析`@Grab`依赖项时声明的依赖项管理。可以使用`@DependencyManagementBom`注释来配置其他依赖项管理，它可以覆盖默认的依赖项管理。注解的值应该指定`groupId:artifactId:version`一个或多个Maven BOM 的坐标()。

例如，请考虑以下声明：

```
@DependencyManagementBom ("com.example.custom-bom: 1.0.0")
```

前面的声明`custom-bom-1.0.0.pom`在下面的Maven仓库中找到`com/example/custom-versions/1.0.0/`。

当您指定多个物料清单时，将按照您声明它们的顺序应用它们，如以下示例所示：

```
@DependencyManagementBom ([ "com.example.custom-bom: 1.0.0",
    "com.example.another-bom: 1.0.0" ])
```

前面的例子表明依赖管理`another-bom`覆盖了依赖管理`custom-bom`。

您可以`@DependencyManagementBom`在任何可以使用的地方使用`@Grab`。但是，为了确保依赖性管理的顺序一致，您最多可以`@DependencyManagementBom`在应用程序中使用一次。一个有用的依赖管理源（它是Spring Boot的依赖管理的超集）是 Spring IO平台，您可以在其中包含以下行：

```
@DependencyManagementBom ('io.spring.platform: platform-bom: 1.1.2.RELEASE')
```

64.2具有多个源文件的应用程序

您可以在接受文件输入的所有命令中使用“shell globbing”。这样做可让您使用单个目录中的多个文件，如以下示例所示：

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

64.3打包您的应用程序

您可以使用该`jar`命令将应用程序打包为独立的可执行jar文件，如以下示例所示：

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

生成的jar包含通过编译应用程序和所有应用程序的依赖关系生成的类，以便可以使用它运行`java -jar`。该jar文件还包含来自应用程序类路径的条目。使用`--include`和可以添加和删除显式路径`--exclude`。两者都以逗号分隔，并且都以“+”和“-”的形式接受前缀，以表示它们应从默认值中删除。默认包括如下：

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

默认排除如下：

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

键入`spring help jar`更多信息，在命令行上。

64.4初始化新项目

该`init`命令允许您在不离开shell的情况下使用start.spring.io创建新项目，如以下示例所示：

```
$ spring init --dependencies = web, data-jpa my-project
在https://start.spring.io使用服务
项目解压到'/ Users / developer / example / my-project'
```

前面的示例`my-project`使用`spring-boot-starter-web`和创建一个具有基于Maven的项目的目录`spring-boot-starter-data-jpa`。您可以使用该`--list`标志列出服务的功能，如以下示例所示：

```
$ spring init --list
=====
https://start.spring.io的功能
=====

可用的依赖关系:
-----
执行器 - 执行器: 生产就绪功能可帮助您监控和管理您的应用程序
```

```
...
Web - Web: 支持全栈Web开发，包括Tomcat和spring-webmvc
websocket - WebSocket: 支持WebSocket开发
WS - WS: 支持Spring Web服务
```

可用的项目类型：

```
gradle-build - Gradle配置[格式: build, build: gradle]
gradle-project - Gradle Project [格式: project, build: gradle]
maven-build - Maven POM [格式: build, build: maven]
maven-project - Maven Project [格式: 项目, 构建: maven] (默认)
...
```

该 `init` 命令支持许多选项。查看 `help` 输出了解更多详情。例如，以下命令创建一个使用Java 8和 `war` 打包的Gradle项目：

```
$ spring init --build = gradle --java-version = 1.8 --dependencies = websocket --packaging = war sample-app.zip
在https://start.spring.io使用服务
内容保存到'sample-app.zip'
```

64.5 使用嵌入式外壳

Spring Boot包含BASH和zsh shell的命令行完成脚本。如果您不使用这两个shell中的任何一个（可能您是Windows用户），则可以使用该 `shell` 命令来启动集成shell，如以下示例所示：

```
$ spring shell
Spring Boot (v2.0.1.BUILD-SNAPSHOT)
点击TAB即可完成。输入'help'，然后按回车寻求帮助，'exit'退出。
```

从嵌入式shell中，您可以直接运行其他命令：

```
$版本
Spring CLI v2.0.1.BUILD-SNAPSHOT
```

嵌入式外壳支持ANSI颜色输出以及 `tab` 完成。如果你需要运行一个本地命令，你可以使用 `!` 前缀。要退出嵌入式外壳，请按 `ctrl-c`。

64.6 将扩展添加到CLI

您可以使用该 `install` 命令向CLI添加扩展。该命令以该格式获取一组或多组工件坐标 `group:artifact:version`，如以下示例所示：

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

除了安装由您提供的坐标标识的工件之外，还安装了所有工件的依赖关系。

要卸载依赖项，请使用该 `uninstall` 命令。与 `install` 命令一样，它以格式为一组或多组工件坐标 `group:artifact:version`，如以下示例所示：

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

它会卸载由您提供的坐标和它们的依赖关系标识的工件。

要卸载所有其他依赖项，可以使用该 `--all` 选项，如以下示例所示：

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

65. 使用Groovy Beans DSL开发应用程序

Spring Framework 4.0具有对 `beans{}` “DSL”（从Grails借用）的本地支持，您可以使用相同的格式将Groovy应用程序脚本中的bean定义嵌入到其中。这有时是包含中间件声明等外部功能的好方法，如下例所示：

```
@Configuration
类应用程序实现了 CommandLineRunner {

    @Autowired
    SharedService 服务

    @Override
```

```

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

}

导入 my.company.SharedService

豆 {
    服务 (SharedService) {
        message = "Hello World"
    }
}

```

beans{} 只要他们停留在顶层，您可以将类声明与同一个文件混合使用，或者，如果您愿意，可以将beans DSL放在单独的文件中。

66. 使用CLI配置CLI `settings.xml`

Spring Boot CLI使用Maven的依赖关系解析引擎Aether来解决依赖关系。CLI使用找到的Maven配置`~/.m2/settings.xml` 来配置Aether。CLI遵循以下配置设置：

- 离线
- 镜子
- 服务器
- 代理
- 简介
 - 激活
 - 库
- 活动档案

有关更多信息，请参阅[Maven的设置文档](#)。

67. 接下来要读什么

There are some [sample groovy scripts](#) available from the GitHub repository that you can use to try out the Spring Boot CLI. There is also extensive Javadoc throughout the source code.

If you find that you reach the limit of the CLI tool, you probably want to look at converting your application to a full Gradle or Maven built “Groovy project”. The next section covers Spring Boot’s “Build tool plugins”, which you can use with Gradle or Maven.

Part VIII. Build tool plugins

Spring Boot provides build tool plugins for Maven and Gradle. The plugins offer a variety of features, including the packaging of executable jars. This section provides more details on both plugins as well as some help should you need to extend an unsupported build system. If you are just getting started, you might want to read “[Chapter 13, Build Systems](#)” from the “[Part III, “Using Spring Boot”](#)” section first.

68. Spring Boot Maven Plugin

The [Spring Boot Maven Plugin](#) provides Spring Boot support in Maven, letting you package executable jar or war archives and run an application “in-place”. To use it, you must use Maven 3.2 (or later).



See the [Spring Boot Maven Plugin Site](#) for complete plugin documentation.

68.1 Including the Plugin

To use the Spring Boot Maven Plugin, include the appropriate XML in the `plugins` section of your `pom.xml`, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <!-- ... -->
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>2.0.1.BUILD-SNAPSHOT</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

The preceding configuration repackages a jar or war that is built during the `package` phase of the Maven lifecycle. The following example shows both the repackaged jar as well as the original jar in the `target` directory:

```

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

```

If you do not include the `<execution/>` configuration, as shown in the prior example, you can run the plugin on its own (but only if the package goal is used as well), as shown in the following example:

```

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

```

If you use a milestone or snapshot release, you also need to add the appropriate `pluginRepository` elements, as shown in the following listing:

```

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

```

68.2 Packaging Executable Jar and War Files

Once `spring-boot-maven-plugin` has been included in your `pom.xml`, it automatically tries to rewrite archives to make them executable by using the `spring-boot:repackage` goal. You should configure your project to build a jar or war (as appropriate) by using the usual `packaging` element, as shown in the following example:

```

<? 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>

```

您的现有存档在此 `package` 阶段由 Spring Boot 增强。您想要启动的主类可以通过使用配置选项或通过 `Main-Class` 以常用方式向清单中添加属性来指定。如果您不指定主类，则该插件将使用 `public static void main(String[] args)` 方法搜索类。

要构建和运行项目工件，可以键入以下内容：

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

要构建可执行并可部署到外部容器中的war文件，需要将嵌入容器依赖项标记为“provided”，如以下示例所示：

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



有关如何创建可部署战争文件的更多详细信息，请参阅“[第87.1节”创建可部署战争文件“部分。](#)

高级配置选项和示例可在[插件信息页面中找到。](#)

69. Spring Boot Gradle插件

Spring Boot Gradle插件在Gradle中提供Spring Boot支持，让您可以打包可执行jar或war档案，运行Spring Boot应用程序，并使用提供的依赖关系管理[spring-boot-dependencies](#)。它需要Gradle 4.0或更高版本。请参阅插件的文档以了解更多信息：

- 参考 ([HTML](#)和 [PDF](#))
- [API](#)

70. Spring Boot AntLib模块

Spring Boot AntLib模块为Apache Ant提供了基本的Spring Boot支持。您可以使用该模块创建可执行文件夹。要使用该模块，您需要[spring-boot](#)在您的声明一个额外的名称空间[build.xml](#)，如下面的示例所示：

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

您需要记住使用该 [-lib](#) 选项启动Ant，如以下示例所示：

```
$ ant -lib <包含spring-boot-antlib-2.0.1.BUILD-SNAPSHOT.jar的文件夹>
```



“使用Spring Boot”部分包含更完整的[使用Apache Ant的spring-boot-antlib示例](#)。

70.1 Spring Boot Ant任务

一旦[spring-boot-antlib](#)声明了名称空间，就可以使用以下附加任务：

- 第70.1.1节“，[spring-boot:execjar](#)”
- 第70.2节“，[spring-boot:findmainclass](#)”

70.1.1 [spring-boot:execjar](#)

您可以使用该 `exejar` 任务来创建 Spring Boot 可执行程序 jar。该任务支持以下属性：

属性	描述	需要
<code>destfile</code>	要创建的目标 jar 文件	是
<code>classes</code>	Java 类文件的根目录	是
<code>start-class</code>	运行的主要应用程序类	No (<i>the default is the first class found that declares a <code>main</code> method</i>)

The following nested elements can be used with the task:

Element	Description
<code>resources</code>	One or more Resource Collections describing a set of Resources that should be added to the content of the created jar file.
<code>lib</code>	One or more Resource Collections that should be added to the set of jar libraries that make up the runtime dependency classpath of the application.

70.1.2 Examples

This section shows two examples of Ant tasks.

Specify start-class.

```
<spring-boot:exejar destfile="target/my-application.jar"
                     classes="target/classes" start-class="com.example.MyApplication">
    <resources>
        <fileset dir="src/main/resources" />
    </resources>
    <lib>
        <fileset dir="lib" />
    </lib>
</spring-boot:exejar>
```

Detect start-class.

```
<exejar destfile = "target / my-application.jar" classes = "target / classes" >
    <lib>
        <fileset dir = "lib" />
    </ lib>
</ exejar>
```

70.2 `spring-boot:findmainclass`

这个 `findmainclass` 任务在内部被 `exejar` 用来定位一个声明一个类的类 `main`。如有必要，您还可以直接在您的构建中使用此任务。支持以下属性：

属性	描述	需要
<code>classesroot</code>	Java 类文件的根目录	是 (除非 <code>mainclass</code> 指定)
<code>mainclass</code>	可用于短路 <code>main</code> 班级搜索	没有
<code>property</code>	应该与结果一起设置的 Ant 属性	否 (如果未指定, 将会记录结果)

70.2.1例子

本节包含三个使用示例 `findmainclass`。

查找并记录。

```
<findmainclass classesroot = "target / classes" />
```

查找并设置。

```
<findmainclass classesroot = "target / classes" property = "main-class" />
```

覆盖并设置。

```
<findmainclass mainclass = "com.example.MainClass" property = "main-class" />
```

71.支持其他构建系统

如果您想使用Maven，Gradle或Ant以外的构建工具，您可能需要开发自己的插件。可执行jar需要遵循特定的格式，并且某些条目需要以未压缩的形式写入（有关详细信息，请参阅附录中的“可执行jar格式”部分）。

Spring Boot Maven和Gradle插件都使用它[spring-boot-loader-tools](#)来实际生成罐子。如果你需要，你可以直接使用这个库。

71.1重新包装档案

要重新打包现有存档以使其成为自包含的可执行存档，请使用[org.springframework.boot.loader.tools.Repackager](#)。该[Repackager](#)班采取的是指现有的罐子或战争归档单个构造函数的参数。使用两种可用[repackage\(\)](#)方法之一来替换原始文件或写入新的目标。重新打包程序运行之前，还可以配置各种设置。

71.2嵌套库

重新打包存档时，可以使用该[org.springframework.boot.loader.tools.Libraries](#)界面包含对依赖项文件的引用。我们没有提供[Libraries](#)这里的具体实现，因为它们通常是构建系统特定的。

如果您的存档已包含库，则可以使用[Libraries.NONE](#)。

71.3找到主要类

If you do not use [Repackager.setMainClass\(\)](#) to specify a main class, the repackager uses ASM to read class files and tries to find a suitable class with a [public static void main\(String\[\] args\)](#) method. An exception is thrown if more than one candidate is found.

71.4 Example Repackage Implementation

The following example shows a typical repackage implementation:

```
Repackager repackager = new Repackager(sourceJarFile);
repackager.setBackupSource(false);
repackager.repackage(新库 () {
    @覆盖
    公共 无效 doWithLibraries (LibraryCallback回调) 抛出 IOException 异常{
        // 构建系统具体实施，回调每个依存性
        // callback.library (新库 (nestedFile, LibraryScope.COMPILE));
    }
});
```

72.接下来要读什么

如果您对构建工具插件的工作方式感兴趣，可以查看[spring-boot-tools](#) GitHub 上的模块。附录中介绍了有关可执行jar格式的更多技术细节。

如果您有特定的构建相关的问题，你可以检查出“[如何做](#)”的指南。

第九部分。'指导'指南

本节为使用Spring Boot时经常出现的一些常见的“我该怎么做……”问题提供了答案。其覆盖范围并不详尽，但确实涵盖了很多。

如果你有一个我们在这里没有涉及的具体问题，你可能想查看 [stackoverlow.com](https://stackoverflow.com)，看看是否有人已经提供了答案。这也是提出新问题的好地方（请使用 `spring-boot` 标签）。

我们也非常乐意扩大这一部分。如果你想添加一个'如何做'，给我们一个拉请求。

73. Spring Boot应用程序

本节包含与Spring Boot应用程序直接相关的主题。

73.1 创建您自己的FailureAnalyzer

`FailureAnalyzer` 是在启动时拦截异常并将其转换为可读的消息的一种很好的方式，并以 `FailureAnalysis`。Spring Boot为应用程序上下文相关异常，JSR-303验证等提供了这样的分析器。你也可以创建你自己的。

`AbstractFailureAnalyzer` 是一个方便的扩展，`FailureAnalyzer` 用于检查异常中是否存在指定的异常类型。您可以从中进行扩展，以便您的实现只有在实际存在时才有机会处理异常。如果出于某种原因无法处理异常，请返回 `null` 给另一个实现处理异常的机会。

`FailureAnalyzer` 实现必须注册 `META-INF/spring.factories`。以下示例注册 `ProjectConstraintViolationFailureAnalyzer`：

```
org.springframework.boot.diagnostics.FailureAnalyzer = \
com.example.ProjectConstraintViolationFailureAnalyzer
```



如果你需要访问 `BeanFactory` 或者 `Environment`，你 `FailureAnalyzer` 可以简单地实现 `BeanFactoryAware` 或 `EnvironmentAware` 分别。

73.2 排除自动配置故障

Spring Boot自动配置尽最大努力“做正确的事情”，但有时候事情会失败，并且很难说明原因。

`ConditionEvaluationReport` 在任何Spring Boot中都有一个非常有用的功能 `ApplicationContext`。你可以看到它，如果你启用 `DEBUG` 日志输出。如果您使用 `spring-boot-actuator`（参见执行器章节），还有一个 `conditions` 端点以JSON呈现报表。使用该端点调试应用程序，并查看Spring Boot在运行时添加了哪些功能（以及哪些尚未添加）。

通过查看源代码和Javadoc可以回答更多的问题。阅读代码时，请记住以下经验法则：

- 寻找所需的课程 `*AutoConfiguration` 并阅读他们的来源。要特别注意 `@Conditional*` 注释以找出它们启用的功能和时间。添加 `--debug` 到命令行或System属性中 `-Ddebug`，可以在控制台上获取应用程序中所做的所有自动配置决策的日志。在正在运行的Actuator应用程序中，查看 `conditions` 端点（`/actuator/conditions` 或JMX等效）以获取相同的信息。
- 寻找类 `@ConfigurationProperties`（例如 `ServerProperties`）并从那里读取可用的外部配置选项。该 `@ConfigurationProperties` 注释具有 `name` 充当外部属性的前缀的属性。因此，`ServerProperties` 拥有 `prefix="server"` 和它的配置性能 `server.port`，`server.address` 以及其他。在运行的执行器应用程序中，查看 `configprops` 端点。
- 查找 `bind` 方法的使用，`Binder` 以便 `Environment` 以放松的方式明确地将配置值拉出。它通常与一个前缀一起使用。
- 寻找 `@Value` 直接绑定到的注释 `Environment`。
- 查找 `@ConditionalOnExpression` 注释，以便根据SpEL表达式开启和关闭功能，通常使用从中解析出的占位符进行评估 `Environment`。

73.3 在开始之前自定义环境或ApplicationContext

A `SpringApplication` has `ApplicationListeners` 和 `ApplicationContextInitializers` that用于将自定义应用于上下文或环境。Spring Boot加载了许多这样的自定义内部使用 `META-INF/spring.factories`。有多种方法可以注册其他自定义设置：

- 以编程方式，根据应用程序，在运行之前调用 `addListeners` 和 `addInitializers` 方法 `SpringApplication`。
- 声明，每个应用程序，通过设置 `context.initializer.classes` 或 `context.listener.classes` 属性。
- 对于所有应用程序，声明性地，通过添加 `META-INF/spring.factories` 和打包应用程序都用作库的jar文件。

该 `SpringApplication` 发送一些特殊 `ApplicationEvents` 的听众（某些情况下被创建甚至更早），然后注册了在公布的事件监听器 `ApplicationContext` 为好。请参阅“Spring Boot功能”一节中的第23.5节“应用程序事件和监听器”以获取完整列表。

也可以 `Environment` 在应用程序上下文通过使用刷新之前自定义 `EnvironmentPostProcessor`。`META-INF/spring.factories` 如下面的例子所示，每个实现应该被注册到：

```
org.springframework.boot.env.EnvironmentPostProcessor = com.example.YourEnvironmentPostProcessor
```

该实现可以加载任意文件并将其添加到 `Environment`。例如，以下示例从类路径加载YAML配置文件：

```

公共 类 EnvironmentPostProcessorExample 实现 EnvironmentPostProcessor {

    private final YamlPropertySourceLoader loader = new YamlPropertySourceLoader();

    @Override
    public void postProcessEnvironment (ConfigurableEnvironment environment,
                                         SpringApplication 应用程序) {
        资源路径= 新的 ClassPathResource ("com / example / myapp / config.yml");
        PropertySource <? > propertySource = loadYaml (path);
        .environment.getPropertySources () addlast 仅 (propertySource);
    }

    private PropertySource <? > loadYaml (Resource path) {
        if (! path.exists ()) {
            throw new IllegalArgumentException ("Resource" + path + "does not exist");
        }
        尝试 {
            返回 这个 .loader.load ("自定义资源", 路径).get (0);
        }
        catch (IOException 异常) {
            抛出 新的 IllegalStateException (
                "未能加载yaml配置从" + 路径, 前);
        }
    }
}

```



在 `Environment` 已经准备与所有常见的财产来源春天引导加载默认。因此可以从环境中获取文件的位置。前面的示例 `custom-resource` 在列表末尾添加属性源，以便在任何常用其他位置定义的键优先。自定义实现可能会定义另一个订单。



警告

虽然 `@PropertySource` 在你的使用上 `@SpringBootApplication` 似乎是一个方便而简单的方法来加载自定义资源，但 `Environment` 我们不推荐它，因为 Spring Boot `Environment` 在 `ApplicationContext` 刷新之前 准备好了。定义的任何键都 `@PropertySource` 被加载太迟而不能对自动配置产生任何影响。

73.4 构建 ApplicationContext 层次结构（添加父级或根级上下文）

您可以使用 `ApplicationBuilder` 该类来创建父/子 `ApplicationContext` 层次结构。有关更多信息，请参阅“Spring Boot功能”一节中的第 23.4 节“Fluent Builder API”。

73.5 创建一个非Web应用程序

并非所有的 Spring 应用程序都必须是 Web 应用程序（或 Web 服务）。如果你想在一个 `main` 方法中执行一些代码，并且引导一个 Spring 应用程序来设置要使用的基础结构，你可以使用 `SpringApplication` Spring Boot 的功能。A `SpringApplication` 会 `ApplicationContext` 根据是否认为需要 Web 应用程序来更改其类。你可以做的第一件事就是将服务器相关的依赖关系（例如 servlet API）关闭到类路径中。如果你不能这样做（例如，你从相同的代码库运行两个应用程序），那么你可以明确地调用 `setWebApplicationType(WebApplicationType.NONE)` 你的 `SpringApplication` 实例，或者设置 `applicationContextClass` 属性（通过 Java API 或外部属性）。您想要作为业务逻辑运行的应用程序代码可以实现为 `CommandLineRunner` 并作为 `@Bean` 定义放入上下文中。

74. 属性和配置

本节包含有关设置和读取属性和配置设置及其与 Spring Boot 应用程序交互的主题。

74.1 在构建时自动扩展属性

您可以使用现有的生成配置自动扩展它们，而不是硬编码在项目的生成配置中指定的一些属性。这在 Maven 和 Gradle 中都是可能的。

74.1.1 使用 Maven 自动扩展属性

您可以使用资源过滤自动扩展 Maven 项目中的属性。如果您使用该方法 `spring-boot-starter-parent`，则可以使用 `@...@` 占位符引用 Maven 的“项目属性”，如以下示例所示：

```
app.encoding=@project.build.sourceEncoding @
app.java.version=@java.version @
```



只有生产配置以这种方式过滤（换句话说，不应用过滤`src/test/resources`）。



如果启用`addResources`标志，`spring-boot:run`目标可以`src/main/resources`直接添加到类路径（用于热重载）。这样做会绕过资源过滤和此功能。相反，您可以使用该`exec:java`目标或自定义插件的配置。请参阅[插件使用页面](#)了解更多详细信息。

如果不使用起动机家长，你需要包括中引入下列元素`<build/>`的元素`pom.xml`：

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

您还需要在内部包含以下元素`<plugins/>`：

```
<plugin>
    <groupId> org.apache.maven.plugins </ groupId>
    <artifactId> maven-resources-plugin </ artifactId>
    <version> 2.7 </ version>
    <configuration>
        <delimiter>
            <delimiter> @ </ delimiter >
        </ delimiters>
        <useDefaultDelimiters> false </ useDefaultDelimiters>
    </ configuration>
</ plugin>
```



`useDefaultDelimiters` 如果您 `${placeholder}` 在配置中使用标准的Spring占位符（例如），则该属性非常重要。如果该属性没有设置`false`，这些可能会被构建扩展。

74.1.2 使用Gradle自动扩展属性

您可以通过配置Java插件的`processResources`任务来自动扩展Gradle项目中的属性，如下例所示：

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

然后，您可以使用占位符引用您的Gradle项目的属性，如以下示例所示：

```
app.name = $ {name}
app.description = $ {description}
```



Gradle的`expand`方法使用了Groovy`SimpleTemplateEngine`，它转换了 `${..}`令牌。这种 `${..}`风格与Spring自己的属性占位机制有冲突。要将Spring属性占位符与自动扩展一起使用，请按如下方式转义Spring属性占位符：`\${..}`。

74.2 外部化配置 SpringApplication

A `SpringApplication` has bean properties (mainly setters), so you can use its Java API as you create the application to modify its behavior. Alternatively, you can externalize the configuration by setting properties in `spring.main.*`. For example, in `application.properties`, you might have the following settings:

```
spring.main.web-environment=false
spring.main.banner-mode=off
```

Then the Spring Boot banner is not printed on startup, and the application is not a web application.



The preceding example also demonstrates how flexible binding allows the use of underscores (`_`) as well as dashes (`-`) in property names.

外部配置中定义的属性会覆盖使用Java API指定的值，但用于创建该属性的源显然是例外 `ApplicationContext`。考虑以下应用程序：

```
新的 SpringApplicationBuilder()
    .bannerMode(Banner.Mode.OFF)
    .sources(demo.MyApp.类)
    .RUN(参数);
```

现在考虑以下配置：

```
spring.main.sources = com.acme.Config, com.acme.ExtraConfig
spring.main.banner-mode = console
```

实际应用中，现在示出的旗帜（如通过配置覆盖），并使用了三个源 `ApplicationContext`（按以下顺序）：，`demo.MyApp`，`com.acme.Config` 和 `com.acme.ExtraConfig`。

74.3更改应用程序的外部属性的位置

默认情况下，来自不同源的属性会 `Environment` 按照定义的顺序添加到 Spring 中（请参阅“Spring Boot功能”一节中的第24章“外部化配置”）。

增加和修改此顺序的一个好方法是向 `@PropertySource` 应用程序源添加注释。传递给 `SpringApplication` 静态便利方法的类和那些添加使用的类将 `setSources()` 被检查以查看它们是否具有 `@PropertySources`。如果他们这样做，这些属性将被添加到 `Environment` 足够早的时间以便在 `ApplicationContext` 生命周期的所有阶段使用。以这种方式添加的属性的优先级低于使用默认位置（如 `application.properties`），系统属性，环境变量或命令行添加的属性的优先级。

您还可以提供以下系统属性（或环境变量）来更改行为：

- `spring.config.name` (`SPRING_CONFIG_NAME`)：默认 `application` 为文件名的根。
- `spring.config.location` (`SPRING_CONFIG_LOCATION`)：要加载的文件（例如类路径资源或URL）。`Environment` 为此文档设置了一个单独的属性源，它可以被系统属性，环境变量或命令行覆盖。

不管你在环境中设置了什么，Spring Boot总是 `application.properties` 按照上面所述加载。默认情况下，如果使用YAML，那么带有'.yml'扩展名的文件也会添加到列表中。

Spring Boot会记录在该 `DEBUG` 级别加载的配置文件以及在 `TRACE` 级别上未找到的候选项。

查看 `ConfigFileApplicationListener` 更多细节。

74.4使用'Short'命令行参数

有些人喜欢使用（例如）`--port=9000` 而不是 `--server.port=9000` 在命令行上设置配置属性。您可以通过使用占位符来启用此行为 `application.properties`，如以下示例所示：

```
server.port = ${port: 8080}
```



如果您从 `spring-boot-starter-parent` POM 继承，则默认的过滤器标记 `maven-resources-plugins` 已从其更改 `{}*` 为 `@`（即，`@maven.token@` 而不是 `{}{maven.token}`）以防止与Spring样式的占位符冲突。如果您已为 `application.properties` 直接启用Maven过滤，则可能还需要更改默认过滤令牌以使用 其他分隔符。



在这种特定情况下，端口绑定可在PaaS环境（如Heroku或Cloud Foundry）中运行。在这两个平台中，`PORT` 环境变量是自动设置的，Spring可以绑定到大写的 `Environment` 属性同义词。

74.5使用YAML作为外部属性

YAML是JSON的超集，因此，它是用于以分层格式存储外部属性的便捷语法，如以下示例所示：

```
spring:
  application:
    name:
  cruncher:
    datasource:
```

```

driverClassName: com.mysql.jdbc.Driver
url: jdbc: mysql:// localhost / test
server:
port: 9000

```

Create a file called `application.yml` and put it in the root of your classpath. Then add `snakeyaml` to your dependencies (Maven coordinates `org.yaml:snakeyaml`), already included if you use the `spring-boot-starter`). A YAML file is parsed to a Java `Map<String, Object>` (like a JSON object), and Spring Boot flattens the map so that it is one level deep and has period-separated keys, as many people are used to with `Properties` files in Java.

The preceding example YAML corresponds to the following `application.properties` file:

```

spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000

```

See “Section 24.6, “Using YAML Instead of Properties”” in the ‘Spring Boot features’ section for more information about YAML.

74.6 Set the Active Spring Profiles

The Spring `Environment` has an API for this, but you would normally set a System property (`spring.profiles.active`) or an OS environment variable (`SPRING_PROFILES_ACTIVE`). Also, you can launch your application with a `-D` argument (remember to put it before the main class or jar archive), as follows:

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

In Spring Boot, you can also set the active profile in `application.properties`, as shown in the following example:

```
spring.profiles.active=production
```

这种方式设置的值被系统属性或环境变量设置取代，但不是由 `SpringApplicationBuilder.profiles()` 方法取代。因此，后一个Java API可以用来扩充配置文件而不更改默认值。

有关更多信息，请参阅“Spring Boot功能”一节中的“[第25章，配置文件](#)”。

74.7根据环境更改配置

YAML文件实际上是一系列按 `---` 行分隔的文档，并且每个文档都被分别解析为拼合的地图。

如果一个YAML文档包含一个 `spring.profiles` 键值，那么配置文件值（一个用逗号分隔的配置文件列表）被输入到 Spring `Environment.acceptsProfiles()` 方法中。如果这些配置文件中的任何一个处于活动状态，则该文档将包含在最终合并中（否则不是），如下示例所示：

```

服务器:
    端口: 9000
---

spring:
    profiles: 开发
服务器:
    port: 9001
---

spring:
    配置文件: 生产
服务器:
    端口: 0

```

在上例中，缺省端口是9000。但是，如果名为'development'的Spring配置文件处于活动状态，则端口为9001。如果'production'处于活动状态，则端口为0。



YAML文档按其遇到的顺序合并。后来的值覆盖较早的值。

要对属性文件执行相同的操作，可以使用 `application-${profile}.properties` 指定特定于配置文件的值。

74.8发现外部属性的内置选项

Spring Boot 在运行时将 `application.properties` (或 `.yml` 文件和其他位置) 的外部属性绑定到应用程序中。没有 (也没有技术上不可能) 在单个位置中提供所有受支持属性的详尽列表 , 因为贡献可能来自类路径上的其他jar文件。

具有执行器功能的正在运行的应用程序具有一个 `configprops` 端点 , 可显示所有可用的绑定和可绑定属性 `@ConfigurationProperties`。

附录包含一个 `application.properties` 带有 Spring Boot 支持的最常见属性列表的示例。最终列表来自搜索源代码 `@ConfigurationProperties` 和 `@Value` 注释以及偶尔使用 `Binder`。有关加载属性的确切顺序的更多信息 , 请参见“ 第24章 , 外部化配置 ”。

75.嵌入式Web服务器

每个 Spring Boot Web 应用程序都包含一个嵌入式 Web 服务器。此功能会导致一些操作方面的问题 , 包括如何更改嵌入式服务器以及如何配置嵌入式服务器。本节回答了这些问题。

75.1 使用另一个Web服务器

许多 Spring Boot 启动器都包含默认的嵌入式容器。`spring-boot-starter-web` 包括 Tomcat `spring-boot-starter-tomcat` , 但您可以使用 `spring-boot-starter-jetty` 或 `spring-boot-starter-undertow` 替代。`spring-boot-starter-webflux` 包括电抗器的 Netty 通过包括 `spring-boot-starter-reactor-netty` , 但你可以使用 `spring-boot-starter-tomcat` , `spring-boot-starter-jetty` 或 `spring-boot-starter-undertow` 替代。



许多初学者只支持 Spring MVC , 所以它们可以传递 `spring-boot-starter-web` 到你的应用程序类路径中。

如果您需要使用不同的 HTTP 服务器 , 则需要排除默认依赖关系并包含您需要的依赖关系。Spring Boot 为 HTTP 服务器提供单独的启动器 , 以帮助尽可能简化此过程。

以下 Maven 示例显示了如何排除 Tomcat 并将 Jetty 包含在 Spring MVC 中 :

```
<依赖>
    <groupId> org.springframework.boot </ groupId>
    <artifactId> spring-boot-starter-web </ artifactId>
    <排除>
        <! - 排除Tomcat 依赖关系 - >
        <排除>
            <groupId>组织.springframework.boot </ groupId>
            <artifactId> spring-boot-starter-tomcat </ artifactId>
        </ exclusion>
    </ exclusions>
</ dependency>
<! - 使用Jetty代替 - >
<dependency>
    <groupId> org.springframework.boot </ groupId>
    <artifactId> spring-boot-starter-jetty </ artifactId>
</ dependency>
```

以下 Gradle 示例显示了如何排除 Netty 并将 Undertow 包含在 Spring WebFlux 中 :

```
配置{
    //排除反应堆Netty
    compile.exclude 模块: 'spring-boot-starter-reactor-netty'
}

依赖关系{
    编译'org.springframework.boot: spring-boot-starter-webflux'
    //使用Undertow代替
    编译'org.springframework.boot: spring-boot-starter-undertow'
    // ...
}
```



`spring-boot-starter-reactor-netty` 需要使用 `WebClient` 该类 , 因此即使需要包含不同的 HTTP 服务器 , 您也可能需要保持对 Netty 的依赖。

75.2 配置码头

Generally, you can follow the advice from “Section 74.8, “Discover Built-in Options for External Properties”” about `@ConfigurationProperties` (`ServerProperties` is the main one here). However, you should also look at `WebServerFactoryCustomizer`. The Jetty APIs are quite rich, so, once you have access to the `JettyServletWebServerFactory`, you can modify it in a number of ways. Alternatively, if you need more control and customization, you can add your own `JettyServletWebServerFactory`.

75.3 Add a Servlet, Filter, or Listener to an Application

There are two ways to add `Servlet`, `Filter`, `ServletContextListener`, and the other listeners supported by the Servlet spec to your application:

- Section 75.3.1, “Add a Servlet, Filter, or Listener by Using a Spring Bean”
- Section 75.3.2, “Add Servlets, Filters, and Listeners by Using Classpath Scanning”

75.3.1 Add a Servlet, Filter, or Listener by Using a Spring Bean

To add a `Servlet`, `Filter`, or Servlet `*Listener` by using a Spring bean, you must provide a `@Bean` definition for it. Doing so can be very useful when you want to inject configuration or dependencies. However, you must be very careful that they do not cause eager initialization of too many other beans, because they have to be installed in the container very early in the application lifecycle. (For example, it is not a good idea to have them depend on your `DataSource` or JPA configuration.) You can work around such restrictions by initializing the beans lazily when first used instead of on initialization.

在的情况下 `Filters` 和 `Servlets`，还可以通过添加添加映射和初始化参数 `FilterRegistrationBean` 或 `ServletRegistrationBean` 代替或除了下面的部件。

 如果 `dispatcherType` 过滤器注册中没有指定，`REQUEST` 则使用。这与Servlet规范的默认分派器类型一致。

像任何其他Spring bean一样，您可以定义Servlet过滤器bean的顺序；请确保检查“注册Servlet，过滤器和监听器为Spring Beans”一节。

禁用Servlet或Filter的注册

正如前面所述，任何 `Servlet` 或 `Filter` 豆与servlet容器自动注册。要禁用特定 `Filter` 或 `Servlet` bean的注册，请为其创建注册bean并将其标记为禁用，如下示例所示：

```
@Bean
public FilterRegistrationBean<MyFilter>注册 (MyFilter过滤器) {
    FilterRegistrationBean<MyFilter> registration = new FilterRegistrationBean<MyFilter>(filter);
    registration.setEnabled(假);
    返回注册;
}
```

75.3.2 使用类路径扫描添加Servlet，筛选器和监听器

`@WebServlet`，`@WebFilter` 和 `@WebListener` 注释类可以通过注释一个 `@Configuration` 类 `@ServletComponentScan` 并指定包含要注册的组件的包来自动向嵌入式servlet容器注册。默认情况下，`@ServletComponentScan` 从注释类的包中扫描。

75.4 更改HTTP端口

在独立应用程序中，主HTTP端口默认为 `8080` 但可以设置为 `server.port`（例如，在 `application.properties` 或作为系统属性）。由于放宽了 `Environment` 值的绑定，您还可以使用 `SERVER_PORT`（例如，作为OS环境变量）。

要完全关闭HTTP端点，但仍要创建一个 `WebApplicationContext`，请使用 `server.port=-1`。（这样做有时对测试有用。）

有关更多详细信息，请参阅“Spring Boot功能”一节或 源代码中的第27.4.4节“自定义嵌入式Servlet容器”`ServerProperties`。

75.5 使用随机未分配的HTTP端口

扫描一个空闲端口（使用操作系统本机防止冲突）使用 `server.port=0`。

75.6 在运行时发现HTTP端口

您可以通过日志输出或 `ServletWebServerApplicationContext` 通过其访问服务器运行的端口 `WebServer`。最好的方法是确保它已被初始化，并添加一个 `@Bean` 类型，`ApplicationListener<ServletWebServerInitializedEvent>` 并在容器发布时将容器拉出。

测试使用 `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` 也可以使用 `@LocalServerPort` 注释将实际端口注入到字段中，如以下示例中所示：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

    @Autowired
    ServletWebServerApplicationContext 服务器;

    @LocalServerPort
    int port;

    // ...

}
```



`@LocalServerPort` 是一个元注释 `@Value("${local.server.port}")`。不要试图在普通应用程序中注入端口。正如我们刚刚看到的，只有在容器初始化后才设置该值。与测试相反，应用程序代码回调会提前处理（在值实际可用之前）。

75.7 配置SSL

可以通过设置各种 `server.ssl.*` 属性来声明性地配置SSL，通常在 `application.properties` 或中 `application.yml`。以下示例显示了在以下位置设置SSL属性 `application.properties`：

```
server.port = 8443
server.ssl.key-store = classpath: keystore.jks
server.ssl.key-store-password = secret
server.ssl.key-password = another-secret
```

查看 `Ssl` 所有支持的属性的详细信息。

使用前面示例中的配置意味着应用程序不再支持8080端口上的普通HTTP连接器。Spring Boot不支持同时配置HTTP连接器和HTTPS连接器 `application.properties`。如果你想拥有两者，你需要以编程方式配置其中之一。我们建议使用 `application.properties` 来配置 HTTPS，因为HTTP连接器更容易以编程方式进行配置。有关 `spring-boot-sample-tomcat-multi-connectors` 示例，请参阅示例项目。

75.8 配置HTTP / 2

您可以使用 `server.http2.enabled` 配置属性在Spring Boot应用程序中启用HTTP / 2支持。这种支持取决于选择的Web服务器和应用程序环境，因为该协议不受JDK8开箱即用的支持。



Spring Boot不支持 `h2c` HTTP / 2协议的明文版本。所以你必须先配置SSL。

75.8.1 HTTP / 2与Undertow

从Undertow 1.4.0+开始，支持HTTP / 2，而不需要JDK8。

75.8.2 使用Jetty的HTTP / 2

从Jetty 9.4.8开始，HTTP / 2也支持 `Conscrypt库`。要启用该支持，您的应用程序需要另外两个依赖项：

`org.eclipse.jetty:jetty-alpn-conscrypt-server` 和 `org.eclipse.jetty.http2:http2-server`。

75.8.3 使用Tomcat的HTTP / 2

Spring Boot默认使用Tomcat 8.5.x。在该版本中，只有 `libtcnative` 在主机操作系统上安装了库及其依赖关系时才支持HTTP / 2。

库文件夹必须可用（如果尚未提供）到JVM库路径。您可以使用JVM参数（如 `-Djava.library.path=/usr/local/opt/tomcat-native/lib`）。更多关于这个的 [官方Tomcat文档](#)。

启动没有本地支持的Tomcat 8.5.x会记录以下错误：

```
错误8787 --- [main] oacoyote.http11.Http11NioProtocol: [h2]的升级处理程序[org.apache.coyote.http2.Http2Protocol]仅支持通过ALI
```

这个错误不是致命的，应用程序仍然以HTTP / 1.1 SSL支持开始。

使用Tomcat 9.0.x和JDK9运行应用程序不需要安装任何本机库。要使用Tomcat 9，您可以使用`tomcat.version`您选择的版本来覆盖构建属性。

75.9配置访问日志记录

访问日志可以通过各自的命名空间为Tomcat，Undertow和Jetty进行配置。

例如，以下设置使用自定义模式在Tomcat上记录访问权限。

```
server.tomcat.basedir = my-tomcat
server.tomcat.accesslog.enabled = true
server.tomcat.accesslog.pattern =%t%a"%r">%s (%D ms)
```

 日志的默认位置是`logs`相对于Tomcat基本目录的目录。默认情况下，该`logs`目录是一个临时目录，因此您可能需要修复Tomcat的基本目录或使用日志的绝对路径。在前面的示例中，日志`my-tomcat/logs`相对于应用程序的工作目录是可用的。

Undertow的访问日志记录可以以类似的方式进行配置，如以下示例所示：

```
server.undertow.accesslog.enabled = true
server.undertow.accesslog.pattern =%t%a"%r">%s (%D ms)
```

日志存储在`logs`相对于应用程序工作目录的目录中。您可以通过设置`server.undertow.accesslog.directory`属性来自定义此位置。

最后，Jetty的访问日志也可以配置如下：

```
server.jetty.accesslog.enabled = true
server.jetty.accesslog.filename = / var / log / jetty-access.log
```

默认情况下，日志被重定向到`System.err`。有关更多详细信息，请参阅 Jetty 文档。

75.10运行在前端代理服务器后面

您的应用程序可能需要发送 302 重定向或使用绝对链接呈现内容。当在代理后面运行时，调用者需要链接到代理而不是托管应用的机器的物理地址。通常情况下，这种情况是通过与代理签订合同来处理的，代理添加了头文件来告诉后端如何构建到自身的链接。

如果代理添加常规`X-Forwarded-For` 和`X-Forwarded-Proto` 标头（大多数代理服务器都这样做），绝对链接应该正确呈现，只要 在您的代码中`server.use-forward-headers` 设置。`true` `application.properties`

 如果您的应用程序在Cloud Foundry或Heroku中运行，则`server.use-forward-headers` 属性默认为`true`。在所有其他情况下，它默认为`false`。

75.10.1自定义Tomcat的代理配置

如果您使用Tomcat，则可以另外配置用于携带“转发”信息的标头名称，如以下示例所示：

```
server.tomcat.remote-IP头= X-您的远程-IP头
server.tomcat.protocol头= X-您的协议报头
```

Tomcat还配置了一个默认正则表达式，该正则表达式与要受信任的内部代理相匹配。默认情况下，IP地址中`10/8`，`192.168/16`，`169.254/16` 和`127/8`是值得信赖的。您可以通过添加一个条目来自定义阀门的配置`application.properties`，如以下示例所示：

```
server.tomcat.internal的代理= 192 \\. 168 \\. \\\ d {1,3} \\\ d {1,3}
```

 只有在使用属性文件进行配置时才需要双反斜杠。如果您使用YAML，则单个反斜杠就足够了，并且与上例中显示的值相同`192\168\.\d{1,3}\.\d{1,3}`。



你可以通过设置 `internal-proxies` 为空来信任所有代理（但不要在生产中这样做）。

您可以 `RemoteIpValve` 通过切换自动关闭（这样做，设置 `server.use-forward-headers=false`）并在 `TomcatServletWebServerFactory` bean 中添加新的阀门实例来完全控制 Tomcat 的配置。

7.5.11 配置 Tomcat

一般情况下，你可以遵循的建议“第 7.4.8，‘发现内置的外部属性选项’”大约 `@ConfigurationProperties`（`ServerProperties` 是这里的主要原因之一）。不过，你也应该看看你可以添加的 `WebServerFactoryCustomizer` 各种 Tomcat 特有的 `*Customizers`。Tomcat API 非常丰富。因此，一旦你有权访问 `TomcatServletWebServerFactory`，你可以通过多种方式修改它。或者，如果您需要更多的控制和定制，您可以添加自己的 `TomcatServletWebServerFactory`。

7.5.12 使用 Tomcat 启用多个连接器

您可以添加一个 `org.apache.catalina.connector.Connector` 到 `TomcatServletWebServerFactory`，它可以允许多个连接器，包括 HTTP 和 HTTPS 连接器，如以下示例所示：

```
@Bean
public ServletWebServerFactory servletContainer () {
    TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory () ;
    tomcat.addAdditionalTomcatConnectors (createSslConnector () ) ;
    return tomcat;
}

私人连接器createSslConnector () {
    连接器连接器= 新连接器 (“org.apache.coyote.http11.Http11NioProtocol”);
    Http11NioProtocol协议= (Http11NioProtocol) connector.getProtocolHandler () ;
    尝试 {
        File keystore = new ClassPathResource (“keystore”).getFile () ;
        File truststore = new ClassPathResource (“keystore”).getFile () ;
        connector.setScheme (“https”);
        connector.setSecure (真);
        connector.setPort (8443);
        protocol.setSSLEnabled (真);
        protocol.setKeystoreFile (keystore.getAbsolutePath ());
        protocol.setKeystorePass (“changeit”);
        protocol.setTruststoreFile (truststore.getAbsolutePath ());
        protocol.setTruststorePass (“changeit”);
        protocol.setKeyAlias (“apitester”);
        返回 连接器;
    }
    catch (IOException ex) {
        抛出 new IllegalStateException (“无法访问密钥库: [” + “keystore”
            + “]或信任库: [” + “keystore” + “]”, ex);
    }
}
```

7.5.13 使用 Tomcat 的 LegacyCookieProcessor

默认情况下，Spring Boot 使用的嵌入式 Tomcat 不支持 Cookie 格式的“版本 0”，因此您可能会看到以下错误：

```
java.lang.IllegalArgumentException: Cookie值中存在无效字符[32]
```

如果可能的话，你应该考虑更新你的代码，以便只存储符合以后 Cookie 规范的值。但是，如果您无法更改 cookie 的写入方式，则可以将 Tomcat 配置为使用 `LegacyCookieProcessor`。要切换到 `LegacyCookieProcessor`，使用 `WebServerFactoryCustomizer` 添加了一个的 bean `TomcatContextCustomizer`，如以下示例所示：

```
@Bean
public WebServerFactoryCustomizer <TomcatServletWebServerFactory> cookieProcessorCustomizer () {
    return (factory) -> factory.addContextCustomizers ()
        (context) -> context.setCookieProcessor (new LegacyCookieProcessor ());
}
```

7.5.14 配置 Undertow

一般来说，你可以遵循的意见“第74.8，‘发现内置的外部属性选项’”关于 `@ConfigurationProperties` (`ServerProperties` 和 `ServerProperties.Undertow` 这里是主要的)。但是，你也应该看看 `WebServerFactoryCustomizer`。一旦你有权访问 `UndertowServletWebServerFactory`，你可以使用 `UndertowBuilderCustomizer` 修改Undertow的配置来满足你的需求。或者，如果您需要更多的控制和定制，您可以添加自己的 `UndertowServletWebServerFactory`。

75.15 使用Undertow启用多个监听器

一个添加 `UndertowBuilderCustomizer` 到 `UndertowServletWebServerFactory` 与收听添加到 `Builder`，如图所示在下面的例子：

```
@Bean
public UndertowServletWebServerFactory servletWebServerFactory () {
    UndertowServletWebServerFactory factory = new UndertowServletWebServerFactory () ;
    factory.addBuilderCustomizers (new UndertowBuilderCustomizer () {

        @Override
        public void customize (Builder builder) {
            builder.addHttpListener (8080, "0.0.0.0") ;
        }
    }) ;
    退货工厂;
}
```

75.16 使用@ServerEndpoint创建WebSocket端点

如果您想在使用 `@ServerEndpoint` 嵌入式容器的Spring Boot应用程序中使用，则必须声明一个 `ServerEndpointExporter` `@Bean`，如以下示例中所示：

```
@Bean
public ServerEndpointExporter serverEndpointExporter () {
    return new ServerEndpointExporter () ;
}
```

上例中显示的 `@ServerEndpoint` bean 使用基础WebSocket容器注册任何带注释的bean。当部署到独立的servlet容器时，该角色由servlet容器初始化程序执行，并且 `ServerEndpointExporter` 不需要该 bean。

75.17 启用HTTP响应压缩

Jetty，Tomcat和Undertow支持HTTP响应压缩。它可以被启用 `application.properties`，如下所示：

```
server.compression.enabled = true
```

默认情况下，为了执行压缩，响应长度必须至少为2048个字节。您可以通过设置 `server.compression.min-response-size` 属性来配置此行为。

默认情况下，响应仅在其内容类型为以下内容之一时才被压缩：

- `text/html`
- `text/xml`
- `text/plain`
- `text/css`

您可以通过设置 `server.compression.mime-types` 属性来配置此行为。

76. Spring MVC

Spring Boot有许多包含Spring MVC的初学者。请注意，一些初学者包括对Spring MVC的依赖，而不是直接包含它。本节回答有关Spring MVC和Spring Boot的常见问题。

76.1 编写JSON REST服务

`@RestController` Spring Boot应用程序中的任何Spring 应该默认呈现JSON响应，只要Jackson2位于类路径中，如以下示例所示：

```

@RestController
public class MyController {

    @RequestMapping("/thing")
    public MyThing thing() {
        return new MyThing();
    }

}

```

只要`MyThing`可以由Jackson2序列化（对于普通的POJO或Groovy对象为true），`localhost:8080/thing`默认情况下将为其提供JSON表示。请注意，在浏览器中，您有时可能会看到XML响应，因为浏览器倾向于发送喜欢XML的接受标头。

76.2编写XML REST服务

如果您`jackson-dataformat-xml`在类路径上有Jackson XML扩展名（），则可以使用它来呈现XML响应。前面我们用于JSON的例子可以工作。要使用Jackson XML渲染器，请将以下依赖项添加到您的项目中：

```

<dependency>
    <groupId> com.fasterxml.jackson.dataformat </ groupId>
    <artifactId> jackson-dataformat-xml </ artifactId>
</ dependency>

```

您可能还想添加对Woodstox的依赖关系。它比JDK提供的默认StAX实现更快，并且还增加了漂亮的打印支持和改进的命名空间处理。以下清单显示了如何在Woodstox中包含依赖项：

```

<dependency>
    <groupId> org.codehaus.woodstox </ groupId>
    <artifactId> woodstox-core-asl </ artifactId>
</ dependency>

```

如果Jackson的XML扩展不可用，则使用JAXB（在JDK中默认提供），并附加`MyThing`注释为的要求`@XmlRootElement`，如以下示例所示：

```

@XmlRootElement
public class MyThing {
    private String name;
    // .. getters 和 setters
}

```

要让服务器呈现XML而不是JSON，您可能必须发送一个`Accept: text/xml`标头（或使用浏览器）。

76.3自定义Jackson ObjectMapper

Spring MVC（客户端和服务器端）用于`HttpMessageConverters`在HTTP交换中协商内容转换。如果Jackson在类路径中，则已经获得由其提供的默认转换器`Jackson2ObjectMapperBuilder`，其实例将为您自动配置。

该`ObjectMapper`（或`XmlMapper`为杰克逊XML转换器）实例（默认创建）具有以下定义的属性：

- `MapperFeature.DEFAULT_VIEW_INCLUSION` 被禁用
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 被禁用
- `SerializationFeature.WRITE_DATES_AS_TIMESTAMPS` 被禁用

Spring Boot还具有一些功能，可以更轻松地自定义此行为。

您可以使用环境配置`ObjectMapper`和`XmlMapper`实例。Jackson提供了一整套简单的开/关功能，可用于配置其处理的各个方面。这些特征在映射到环境中的属性的六个枚举（在Jackson中）中进行了描述：

杰克逊枚举

`com.fasterxml.jackson.databind.DeserializationFeature`

`com.fasterxml.jackson.core.JsonGenerator.Feature`

`com.fasterxml.jackson.databind.MapperFeature`

`com.fasterxml.jackson.core.JsonParser.Feature`

环境属性

`spring.jackson.deserialization.<feature_name>=true|false`

`spring.jackson.generator.<feature_name>=true|false`

`spring.jackson.mapper.<feature_name>=true|false`

`spring.jackson.parser.<feature_name>=true|false`

杰克逊枚举

`com.fasterxml.jackson.databind.SerializationFeature`

`com.fasterxml.jackson.annotation.JsonInclude.Include`

环境属性

`spring.jackson.serialization.<feature_name>=true|false`

`spring.jackson.default-property-inclusion=always|non_null|no`

例如，要启用漂亮的打印，请设置`spring.jackson.serialization.indent_output=true`。请注意，由于使用了宽松的绑定，所以`indent_output`不需要匹配相应的枚举常量的情况，也就是说`INDENT_OUTPUT`。

此基于环境的配置应用于自动配置的`Jackson2ObjectMapperBuilder` bean，并适用于使用构建器创建的任何映射器，包括自动配置的`ObjectMapper` bean。

上下文`Jackson2ObjectMapperBuilder`可以由一个或多个`Jackson2ObjectMapperBuilderCustomizer` bean 自定义。这些定制程序bean可以被订购（Boot自己的定制程序的顺序为0），允许在定制开机前和定制后应用额外的定制。

任何类型的bean都会`com.fasterxml.jackson.databind.Module`自动注册到自动配置中`Jackson2ObjectMapperBuilder`，并应用于`ObjectMapper`它创建的任何实例。这为向应用程序添加新功能时为自定义模块提供了全局机制。

如果要`ObjectMapper`完全替换默认值，请定义一个`@Bean`类型并将其标记为`@Primary`或，如果您更喜欢基于构建器的方法，请定义一个`Jackson2ObjectMapperBuilder @Bean`。请注意，无论如何，这样做都会禁用所有的自动配置`ObjectMapper`。

如果您提供任何`@Beans`类型`MappingJackson2HttpMessageConverter`，它们将替换MVC配置中的默认值。另外，`HttpMessageConverters`还提供了一个便捷的类型的bean（如果使用默认的MVC配置，它总是可用的）。它有一些有用的方法来访问默认和用户增强的消息转换器。

请参阅“[第76.4节”自定义@ResponseBody渲染“部分](#)”和[WebMvcAutoConfiguration](#) 源代码以获取更多详细信息。

76.4自定义@ResponseBody渲染

Spring `HttpMessageConverters`用来渲染`@ResponseBody`（或来自`@RestController`）。您可以通过在Spring Boot环境中添加适当类型的Bean来提供额外的转换器。如果您添加的bean是默认包含的类型（例如`MappingJackson2HttpMessageConverter`JSON转换），它将替换默认值。提供了一个便捷的类型的bean，`HttpMessageConverters`如果使用默认的MVC配置，它总是可用的。它有一些有用的方法来访问默认和用户增强的消息转换器（例如，如果您想手动将它们注入到自定义中，它会很有用`RestTemplate`）。

与正常的MVC用法一样，`WebMvcConfigurer`您提供的任何bean也可以通过覆盖该`configureMessageConverters`方法来贡献转换器。但是，与普通的MVC不同，您只能提供您需要的其他转换器（因为Spring Boot使用相同的机制来提供其默认值）。最后，如果您通过提供自己的`@EnableWebMvc`配置选择退出Spring Boot默认MVC配置，则可以完全控制并使用`getMessageConverters` from 完成所有操作`WebMvcConfigurationSupport`。

有关[WebMvcAutoConfiguration](#) 更多详细信息，请参阅 源代码。

76.5处理多部分文件上传

Spring Boot包含Servlet 3 `javax.servlet.http.Part` API以支持上传文件。默认情况下，Spring Boot在每个文件中配置Spring MVC的最大大小为1MB，单个请求中最大为10MB的文件数据。您可以覆盖这些值，中间数据存储到的位置（例如，`/tmp` 目录）以及使用`MultipartProperties`该类中公开的属性将数据刷新到磁盘的阈值。例如，如果您要指定文件不受限制，请将该`spring.servlet.multipart.max-file-size`属性设置为`-1`。

当您想要在Spring MVC控制器处理程序方法中将多部分编码文件数据作为`@RequestParam`类型的注释参数接收时，多部分支持很有用`MultipartFile`。

查看[MultipartAutoConfiguration](#) 来源获取更多细节。

76.6关闭Spring MVC DispatcherServlet

Spring Boot希望将应用程序根目录下的所有内容都放在`/`下面。如果您宁愿将自己的servlet映射到该URL，则可以执行此操作。但是，您可能会失去其他一些Boot MVC功能。要添加自己的servlet并将其映射到根资源，请声明一个`@Bean`类型`Servlet`并为其指定特殊的bean名称，`dispatcherServlet`。（如果你想关闭它而不是替换它，你也可以用这个名称创建一个不同类型的bean。）

76.7关闭默认的MVC配置

完全控制MVC配置的最简单方法是提供自己`@Configuration`的`@EnableWebMvc`注释。这样做会将所有MVC配置留在您的手中。

76.8自定义ViewResolvers

A `ViewResolver`是Spring MVC的核心组件，将视图名称转换`@Controller`为实际的`View`实现。请注意，`ViewResolvers`它们主要用于UI应用程序，而不是REST风格的服务（a `View`不用于渲染`@ResponseBody`）。有许多`ViewResolver`可供选择的实现，Spring本身并不认为你应该使用哪一个。另一方面，Spring Boot会为您安装一个或两个，具体取决于它在类路径和应用程序上下文中找到的内容。

在`DispatcherServlet`使用所有解析器发现在应用方面，试着用每一个转动，直到它得到的结果，所以，如果你添加你自己的，你必须要知道的秩序，在加入哪个位置您解析。

`WebMvcAutoConfiguration`将以下内容添加`ViewResolvers`到上下文中：

- 一个`InternalResourceViewResolver`名为“defaultViewResolver”。这个定位可以通过使用`DefaultServlet`（包括静态资源和JSP页面，如果使用这些页面）呈现的物理资源。它将前缀和后缀应用于视图名称，然后在servlet上下文中查找具有该路径的物理资源（缺省值均为空，但可通过`spring.mvc.view.prefix`和进行外部配置访问`spring.mvc.view.suffix`）。您可以通过提供相同类型的bean来覆盖它。
- 一个`BeanNameViewResolver`名为'beanNameViewResolver'的。这是视图解析器链中的一个有用的成员，并且可以获取与`View`被解析的名称相同的所有bean。不应该有必要重写或替换它。
- 一个`ContentNegotiatingViewResolver`名为“ViewResolver”的只如果有添加的 实际类型的豆类`View`存在。这是一个“主”解析器，委托给所有其他人，并试图找到与客户端发送的“Accept”HTTP头匹配。有一个有用的 博客 `ContentNegotiatingViewResolver`，您可能想了解更多信息，也可以查看源代码以获取详细信息。您可以`ContentNegotiatingViewResolver`通过定义一个名为'viewResolver'的bean 来关闭自动配置。
- 如果你使用Thymeleaf，你也有一个`ThymeleafViewResolver`名为'thymeleafViewResolver'。它通过用前缀和后缀包围视图名称来查找资源。前缀是`spring.thymeleaf.prefix`，后缀是`spring.thymeleaf.suffix`。前缀和后缀的值分别默认为'classpath : / templates /'和'.html'。您可以`ThymeleafViewResolver`通过提供相同名称的bean来进行覆盖。
- 如果你使用FreeMarker，你也有一个`FreeMarkerViewResolver`名为'freeMarkerViewResolver'。它`spring.freemarker.templateLoaderPath`通过用前缀和后缀包围视图名称来查找加载器路径中的资源（将其外部化并具有默认值'classpath : / templates /'）。前缀是外部化的`spring.freemarker.prefix`，而后缀是外部化的`spring.freemarker.suffix`。前缀和后缀的默认值分别为空和'.ftl'。您可以`FreeMarkerViewResolver`通过提供相同名称的bean来进行覆盖。
- 如果您使用Groovy模板（实际上，如果`groovy-templates`在您的类路径中），那么您还有一个`GroovyMarkupViewResolver`名为“groovyMarkupViewResolver”的名称。它通过用前缀和后缀（外化到`spring.groovy.template.prefix`和`spring.groovy.template.suffix`）包围视图名称来查找加载器路径中的资源。前缀和后缀分别具有'classpath : / templates /'和'.tpl'的默认值。您可以`GroovyMarkupViewResolver`通过提供相同名称的bean来进行覆盖。

有关更多详细信息，请参阅以下部分：

- `WebMvcAutoConfiguration`
- `ThymeleafAutoConfiguration`
- `FreeMarkerAutoConfiguration`
- `GroovyTemplateAutoConfiguration`

77. HTTP客户端

Spring Boot提供了许多与HTTP客户端一起工作的入门者。本节回答与使用它们有关的问题。

77.1配置RestTemplate以使用代理

如第33.1节“RestTemplate自定义”中所述，您可以使用`RestTemplateCustomizer` with `RestTemplateBuilder`来构建自定义`RestTemplate`。这是创建`RestTemplate`配置为使用代理的推荐方法。

代理配置的确切细节取决于正在使用的底层客户端请求工厂。以下示例使用为所有主机使用代理的配置`HttpComponentsClientRequestFactory`，`HttpClient`除了`192.168.0.5`：

```
静态 类 ProxyCustomizer 实现了 RestTemplateCustomizer {
    @Override
    public void customize(RestTemplate restTemplate) {
        HttpHost代理= 新的 HttpHost ("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create()
            .setRoutePlanner (新的 DefaultProxyRoutePlanner (代理) {
                @覆盖
                公共 HttpHost determineProxy (HttpHost目标,
                    HttpRequest请求, HttpContext上下文)
                    抛出 HttpException {
                        if (target.getHostName () .equals ("192.168.0.5")) {
                            return null;
                        }
                    }
            });
    }
}
```

```

        }
        返回 超级 .determineProxy(目标, 请求, 上下文) ;
    }

}。建立();
restTemplate.setRequestFactory(
    新的 HttpComponentsClientHttpRequestFactory (httpClient) );
}

}

```

记录

除了Commons Logging API (通常由Spring Framework `spring-jcl`模块提供)之外, Spring Boot没有强制日志依赖性。要使用Logback, 您需要将其包含`spring-jcl`在类路径中。最简单的方法是通过所有依赖的初学者`spring-boot-starter-logging`。对于Web应用程序, 只需要`spring-boot-starter-web`它, 因为它依赖于日志启动器。如果您使用Maven, 则以下依赖项会为您添加日志记录:

```

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

```

Spring Boot有一个`LoggingSystem`抽象, 试图根据类路径的内容配置日志记录。如果Logback可用, 它是第一选择。

如果您需要对日志记录进行的唯一更改是设置各种日志记录器的级别, 则可以`application.properties`使用“`logging.level`”前缀来完成此操作, 如以下示例中所示:

```

logging.level.org.springframework.web = DEBUG
logging.level.org.hibernate = 错误

```

您还可以通过使用“`logging.file`”来设置要写入日志的文件的位置(除了控制台之外)。

要配置日志记录系统的更精细的设置, 您需要使用有问题的原生配置格式`LoggingSystem`。默认情况下, Spring Boot从系统的默认位置(例如`classpath:logback.xml`Logback)获取本地配置, 但可以使用“`logging.config`”属性设置配置文件的位置。

78.1配置Logback进行日志记录

如果将一个`logback.xml`放入类路径的根目录中, 则会从那里(或从中`logback-spring.xml`获取)利用Boot提供的模板功能。Spring Boot提供了一个默认的基本配置, 如果您想设置级别, 可以包含该配置, 如以下示例所示:

```

<? xml version =“1.0”encoding =“UTF-8”? >
<configuration>
    <include resource = “org / springframework / boot / logging / logback / base.xml” />
    <logger name = “org.springframework. web” level = “DEBUG” />
</ configuration>

```

如果你看一下`base.xml`spring-boot jar, 你可以看到它使用了一些有用的系统属性, 它们`LoggingSystem`为你创建:

- `${PID}`: 当前进程ID。
- `${LOG_FILE}`: 是否`logging.file`在Boot的外部配置中设置。
- `${LOG_PATH}`: 是否`logging.path`在Boot的外部配置中设置(代表日志文件的目录)。
- `${LOG_EXCEPTION_CONVERSION_WORD}`: 是否`logging.exception-conversion-word`在Boot的外部配置中设置。

Spring Boot还通过使用自定义Logback转换器在控制台(但不是日志文件)上提供了一些漂亮的ANSI彩色终端输出。详情请参阅默认`base.xml`配置。

如果Groovy位于类路径中, 那么您应该也可以配置Logback `logback.groovy`。如果存在, 则优先选择此设置。

78.1.1为仅文件输出配置Logback

如果要禁用控制台日志并仅将输出写入文件, 则需要`logback-spring.xml`导入`file-appender.xml`但不导入的定制`console-appender.xml`, 如以下示例所示:

```

<? xml version =“1.0”encoding =“UTF-8”? >
<configuration>
    <include resource = “org / springframework / boot / logging / logback / defaults.xml” />
    <property name = “LOG_FILE” value = “${LOG_FILE: - ${LOG_PATH: - ${LOG_TEMP: - ${java.io.tmpdir: - /}}}}” />
    <include resource = “org / springframework / boot / logging / logback/file-appender.xml” />

```

```

<root level = "INFO" >
    <appender-ref ref = "FILE" />
</root>
</configuration>

```

您还需要添加 `logging.file` 到您的 `application.properties`，如以下示例所示：

```
logging.file = myapplication.log
```

78.2配置Log4j进行日志记录

Spring Boot支持Log4j 2用于记录配置，如果它在类路径上。如果您使用starters来组装依赖项，则必须排除Logback，然后包含log4j 2。如果您不使用初学者，则 `spring-jcl` 除了Log4j 2之外，您还需要提供（至少）。

最简单的路径可能是通过初学者，即使它需要一些排除跳跃。以下示例显示如何在Maven中设置初学者：

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>

```



Log4j初学者将通用日志记录需求的依赖关系（例如使用Tomcat，`java.util.logging`但使用Log4j 2配置输出）收集在一起。有关更多详细信息，请参阅 [执行器Log4j 2样本](#)，并查看它的实际使用情况。



要确保使用的调试日志记录 `java.util.logging` 路由到Log4j 2，请通过将系统属性 `java.util.logging.manager` 配置为 `org.apache.logging.log4j.jul.LogManager`

78.2.1使用YAML或JSON配置Log4j 2

除了默认的XML配置格式之外，Log4j 2还支持YAML和JSON配置文件。要配置Log4j 2以使用备用配置文件格式，请将相应的依赖关系添加到类路径中，并将您的配置文件命名为与您选择的文件格式相匹配，如下例所示：

格式	依赖	文件名称
YAML	<code>com.fasterxml.jackson.core:jackson-databind</code> <code>com.fasterxml.jackson.dataformat:jackson-dataformat-yaml</code>	<code>log4j2.yaml</code> <code>log4j2.yml</code>
JSON	<code>com.fasterxml.jackson.core:jackson-databind</code>	<code>log4j2.json</code> <code>log4j2.jsn</code>

79.数据访问

Spring Boot包含许多用于处理数据源的入门者。本节回答与此相关的问题。

79.1配置自定义数据源

要配置您自己的配置，请在您的配置中 `DataSource` 定义 `@Bean` 该类型。Spring Boot会重用您 `DataSource` 需要的任何地方，包括数据库初始化。如果您需要外部化某些设置，则可以将其绑定 `DataSource` 到环境（请参阅“[第24.7.1节”第三方配置](#)”）。

以下示例显示如何在bean中定义数据源：

```
@Bean
@ConfigurationProperties (prefix =“app.datasource”)
public DataSource dataSource () {
    return new FancyDataSource () ;
}
```

以下示例显示如何通过设置属性来定义数据源：

```
app.datasource.url = jdbc: h2: mem: mydb
app.datasource.username = sa
app.datasource.pool-size = 30
```

假设您 `FancyDataSource` 的URL，用户名和池大小具有常规的JavaBean属性，则这些设置会在 `DataSource` 其他组件可用之前自动绑定。常规的 数据库初始化 也会发生（所以相关的子集 `spring.datasource.*` 仍然可以与您的自定义配置一起使用）。

如果您配置自定义JNDI `DataSource`，则可以应用相同的原则，如下例所示：

```
@Bean (destroyMethod =““)
@ConfigurationProperties (prefix =“app.datasource”)
public DataSource dataSource () throws Exception {
    JndiDataSourceLookup dataSourceLookup = new JndiDataSourceLookup () ;
    返回 dataSourceLookup.getDataSource (“java: comp / env / jdbc / YourDS”);
}
```

Spring Boot还提供了一个实用程序构建器类，`DataSourceBuilder` 可以用来创建其中一个标准数据源（如果它位于类路径中）。构建器可以根据类路径上可用的内容来检测要使用的那个。它还根据JDBC URL自动检测驱动程序。

以下示例显示如何使用以下命令创建数据源 `DataSourceBuilder`：

```
@Bean
@ConfigurationProperties (“app.datasource”)
public DataSource dataSource () {
    return DataSourceBuilder.create () .build ();
}
```

要运行一个应用程序，只 `DataSource` 需要连接信息。也可以提供池特定的设置。查看将在运行时使用的实现以获取更多详细信息。

以下示例显示如何通过设置属性来定义JDBC数据源：

```
app.datasource.url = JDBC: MySQL的: //本地主机/测试
app.datasource.username = DBUSER
app.datasource.password = DBPASS
app.datasource.pool尺寸 = 30
```

However, there is a catch. Because the actual type of the connection pool is not exposed, no keys are generated in the metadata for your custom `DataSource` and no completion is available in your IDE (because the `DataSource` interface exposes no properties). Also, if you happen to have Hikari on the classpath, this basic setup does not work, because Hikari has no `url` property (but does have a `jdbcUrl` property). In that case, you must rewrite your configuration as follows:

```
app.datasource.jdbc-url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.maximum-pool-size=30
```

你可以通过强制使用连接池并返回一个专用的实现来解决这个问题 `DataSource`。您不能在运行时更改实现，但选项列表将是显式的。

以下示例显示了如何创建一个 `HikariDataSource` with `DataSourceBuilder`：

```
@Bean
@ConfigurationProperties (“app.datasource”)
公开 HikariDataSource数据源 () {
    返回 DataSourceBuilder.create () 类型 (HikariDataSource.类) .build ();
}
```

`DataSourceProperties` 如果没有提供URL，您甚至可以利用为您提供的内容进一步深入研究- 即通过提供默认的嵌入式数据库，提供明智的用户名和密码。你可以很容易地 `DataSourceBuilder` 从任何 `DataSourceProperties` 对象的状态 初始化一个，所以你也可以注入Spring Boot自

动创建的DataSource。然而，这将配置分为两个命名空间：`url`，`username`，`password`，`type`，和`driver`上`spring.datasource`和您的自定义命名空间中的其余部分（`app.datasource`）。为了避免这种情况，您可以`DataSourceProperties`在自定义名称空间上重新定义自定义，如以下示例所示：

```
@Bean
@Primary
@ConfigurationProperties ("app.datasource")
public DataSourceProperties dataSourceProperties () {
    return new DataSourceProperties ();
}

@Bean
@ConfigurationProperties ("app.datasource")
public HikariDataSource dataSource (DataSourceProperties properties) {
    return properties.initializeDataSourceBuilder ().type (HikariDataSource.class)
        .build ();
}
```

除了选择了一个专用连接池（在代码中）并且它的设置暴露在相同的命名空间中之外，这个设置使您可以与默认情况下的Spring Boot为您做同步。由于`DataSourceProperties`您正在为您翻译`url`/ `jdbcUrl`翻译，因此您可以按如下方式进行配置：

```
app.datasource.url = jdbc: mysql: // localhost / test
app.datasource.username = dbuser
app.datasource.password =
dbpass app.datasource.maximum-pool-size = 30
```



因为您的自定义配置选择与Hikari一起使用，`app.datasource.type`所以不起作用。在实践中，构建器会以您可能设置的任何值进行初始化，然后由呼叫覆盖`.type()`。

有关更多详细信息，请参见“Spring Boot功能”部分和`DataSourceAutoConfiguration`类中的“第29.1节”配置数据源“”。

79.2 配置两个数据源

如果您需要配置多个数据源，则可以应用上一节中介绍的相同技巧。但是，您必须标记其中一个`DataSource`实例`@Primary`，因为各种自动配置都期望能够逐个获取。

如果您创建了自己`DataSource`的配置，则会自动配置。在以下示例中，我们提供了与自动配置在主数据源上提供的完全相同的功能集：

```
@Bean
@Primary
@ConfigurationProperties ("app.datasource.first")
public DataSourceProperties firstDataSourceProperties () {
    return new DataSourceProperties ();
}

@Bean
@Primary
@ConfigurationProperties ("app.datasource.first")
public BasicDataSource firstDataSource () {
    return firstDataSourceProperties ().initializeDataSourceBuilder ().build ();
}

@Bean
@ConfigurationProperties ("app.datasource.second")
public BasicDataSource secondDataSource () {
    return DataSourceBuilder.create ().type (BasicDataSource.class).build ();
}
```



`firstDataSourceProperties`必须标记为`@Primary`使数据库初始值设定项功能使用您的副本（如果您使用初始值设定项）。

这两个数据源也都是用于高级自定义的。例如，你可以如下配置它们：

```
app.datasource.first.type = com.zaxxer.hikari.HikariDataSource
app.datasource.first.maximum-pool-size = 30

app.datasource.second.url = jdbc: mysql: // localhost / test
app.datasource.second.username = dbuser
```

```
app.datasource.second.password =
dbpass app.datasource.second.max-total = 30
```

您也可以将相同的概念应用于辅助`DataSource`，如以下示例所示：

```
@Bean
@Primary
@ConfigurationProperties ("app.datasource.first")
public DataSourceProperties firstDataSourceProperties () {
    return new DataSourceProperties () ;
}

@Bean
@Primary
@ConfigurationProperties ("app.datasource.first")
的公共数据源firstDataSource () {
    回报 firstDataSourceProperties () initializeDataSourceBuilder () 建 () 。
}

@Bean
@ConfigurationProperties ("app.datasource.second")
public DataSourceProperties secondDataSourceProperties () {
    return new DataSourceProperties () ;
}

@Bean
@ConfigurationProperties ("app.datasource.second")
的公共数据源secondDataSource () {
    回报 secondDataSourceProperties () initializeDataSourceBuilder () 建 () 。
}
```

前面的示例在自定义命名空间上配置两个数据源，其使用与Spring Boot在自动配置中使用的逻辑相同的逻辑。

79.3 使用Spring数据仓库

Spring Data可以创建`@Repository`各种风格的接口的实现。Spring Boot可以为你处理所有这些事情，只要这些`@Repositories`包含在你`@EnableAutoConfiguration`班级的同一个包（或一个子包）中。

对于许多应用程序，您只需将正确的Spring Data依赖项放在您的类路径中（`spring-boot-starter-data-jpa`对于JPA和`spring-boot-starter-data-mongodb`MongoDB有一个），并创建一些存储库接口来处理您的`@Entity`对象。例子在 JPA示例 和 MongoDB示例中。

Spring Boot尝试根据找到的内容猜测`@Repository`定义的位置`@EnableAutoConfiguration`。为了获得更多控制权，请使用`@EnableJpaRepositories`注释（来自Spring Data JPA）。

有关Spring Data的更多信息，请参阅Spring Data项目页面。

79.4 来自Spring配置的单独实体定义

Spring Boot尝试根据找到的内容猜测`@Entity`定义的位置`@EnableAutoConfiguration`。为了获得更多控制，可以使用`@EntityScan`注释，如以下示例所示：

```
@Configuration
@EnableAutoConfiguration
@EntityScan (basePackageClasses = City.class)
public class Application {

    // ...

}
```

79.5 配置JPA属性

Spring Data JPA已经提供了一些与供应商无关的配置选项（比如那些用于SQL日志记录的配置选项），并且Spring Boot公开了这些选项以及一些Hibernate作为外部配置属性。其中一些是根据上下文自动检测的，所以您不必设置它们。

这`spring.jpa.hibernate.ddl-auto`是一个特殊情况，因为根据运行时条件，它有不同的默认值。如果使用嵌入式数据库并且没有模式管理器（如Liquibase或Flyway）正在处理该数据库`DataSource`，则默认为`create-drop`。在所有其他情况下，它默认为`none`。

使用的方言也会根据当前情况自动检测 `DataSource`，但 `spring.jpa.database` 如果您想明确并在启动时绕过该检查，则可以设置自己的方言。



指定一个 `database` 导致定义明确的Hibernate方言的配置。几个数据库有不止一个 `Dialect`，这可能不适合您的需要。在这种情况下，您可以设置 `spring.jpa.database` 为 `default` 让Hibernate通过设置 `spring.jpa.database-platform` 属性来设置方言或设置方言。

以下示例显示了最常见的设置选项：

```
spring.jpa.hibernate.naming.physical-strategy= com.example.MyPhysicalNamingStrategy
spring.jpa.show-SQL =真
```

另外，创建 `spring.jpa.properties.*` 本地时，所有属性都作为普通的JPA属性传递（剥离前缀）`EntityManagerFactory`。



如果您需要将高级定制应用于Hibernate属性，请考虑注册一个 `HibernatePropertiesCustomizer` 将在创建之前调用的bean `EntityManagerFactory`。这优先于由自动配置应用的任何内容。

79.6 配置Hibernate命名策略

Hibernate使用两种不同的命名策略将名称从对象模型映射到相应的数据库名称。物理和隐式策略实现的全限定类名称可分别通过设置 `spring.jpa.hibernate.naming.physical-strategy` 和 `spring.jpa.hibernate.naming.implicit-strategy` 属性进行配置。替代地，如果 `ImplicitNamingStrategy` 或 `PhysicalNamingStrategy` 豆在应用程序上下文是可用的，Hibernate会被自动配置为使用它们。

默认情况下，Spring Boot使用配置物理命名策略 `SpringPhysicalNamingStrategy`。这个实现提供了和Hibernate 4一样的表结构：所有的点都被下划线替代，骆驼套也被下划线替代。默认情况下，所有表名均以小写形式生成，但如果您的模式需要它，则可以覆盖该标志。

例如，一个 `TelephoneNumber` 实体被映射到该 `telephone_number` 表。

如果您更喜欢使用Hibernate 5的默认设置，请设置以下属性：

```
spring.jpa.hibernate.naming.physical-strategy= org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

或者，您可以配置下列bean：

```
@Bean
public PhysicalNamingStrategy physicalNamingStrategy() {
    return new PhysicalNamingStrategyStandardImpl();
}
```

请参阅 `HibernateJpaAutoConfiguration` 并 `JpaBaseConfiguration` 了解更多详情。

79.7 使用自定义 EntityManagerFactory

要完全控制配置 `EntityManagerFactory`，你需要添加一个 `@Bean` 名为'entityManagerFactory'。Spring Boot自动配置在存在该类型bean的情况下关闭其实体管理器。

79.8 使用两个EntityManagers

Even if the default `EntityManagerFactory` works fine, you need to define a new one. Otherwise, the presence of the second bean of that type switches off the default. To make it easy to do, you can use the convenient `EntityManagerBuilder` provided by Spring Boot.

Alternatively, you can just the `LocalContainerEntityManagerFactoryBean` directly from Spring ORM, as shown in the following example:

```
// 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();
}

```

The configuration above almost works on its own. To complete the picture, you need to configure `TransactionManagers` for the two `EntityManagers` as well. If you mark one of them as `@Primary`, it could be picked up by the default `JpaTransactionManager` in Spring Boot. The other would have to be explicitly injected into a new instance. Alternatively, you might be able to use a JTA transaction manager that spans both.

如果您使用Spring Data，则需要进行相应的配置`@EnableJpaRepositories`，如以下示例所示：

```

@Configuration
@EnableJpaRepositories (basePackageClasses = Customer.class,
    entityManagerFactoryRef ="customerEntityManagerFactory")
public class CustomerConfiguration {
    ...
}

@Configuration
@EnableJpaRepositories (basePackageClasses = Order.class,
    entityManagerFactoryRef ="orderEntityManagerFactory")
public class OrderConfiguration {
    ...
}

```

79.9 使用传统 `persistence.xml` 文件

Spring不需要使用XML来配置JPA提供程序，Spring Boot假定您想要利用该功能。如果你喜欢使用`persistence.xml`，你需要定义你自己`@Bean`的类型`LocalEntityManagerFactoryBean`（ID为'entityManagerFactory'）并在那里设置持久化单元名称。

请参阅`JpaBaseConfiguration`默认设置。

79.10 使用Spring Data JPA和Mongo仓库

Spring Data JPA和Spring Data Mongo都可`Repository`以为你自动创建实现。如果它们都出现在类路径中，则可能需要做一些额外的配置来告诉Spring Boot要创建的存储库。最明确的做法是使用标准的Spring Data `@EnableJpaRepositories` 和 `@EnableMongoRepositories` 注解，并提供`Repository`接口的位置。

还有一些标志（`spring.data.*.repositories.enabled` 和 `spring.data.*.repositories.type`）可用于在外部配置中打开和关闭自动配置的存储库。这样做很有用，例如，如果您想关闭Mongo存储库并仍然使用自动配置`MongoTemplate`。

其他自动配置的Spring Data存储库类型（Elasticsearch，Solr等）存在相同的障碍和相同的功能。要使用它们，请相应地更改注释和标志的名称。

79.11 将Spring数据存储库公开为REST端点

Spring Data REST可以`Repository`为您提供作为REST端点的实现，前提是为应用程序启用了Spring MVC。

Spring Boot公开了一组`spring.data.rest`自定义属性的有用属性（来自命名空间）`RepositoryRestConfiguration`。如果你需要提供额外的定制，你应该使用一个`RepositoryRestConfigurer` bean。



如果您没有在自定义中指定任何顺序`RepositoryRestConfigurer`，它将在Spring Boot在内部使用后运行。如果您需要指定订单，请确保它大于0。

79.12 配置JPA使用的组件

如果你想配置一个JPA使用的组件，那么你需要确保组件在JPA之前被初始化。当组件自动配置时，Spring Boot会为您提供帮助。例如，当Flyway被自动配置时，Hibernate被配置为依赖于Flyway，因此在Hibernate尝试使用它之前，Flyway有机会初始化数据库。

如果您自己配置组件，则可以使用 `EntityManagerFactoryDependsOnPostProcessor` 子类作为设置必要依赖项的便捷方式。例如，如果将 Hibernate Search 与 Elasticsearch 一起用作其索引管理器，则 `EntityManagerFactory` 必须将任何 bean 配置为依赖于该 `elasticsearchClient` bean，如以下示例所示：

```

/**
 * {@link EntityManagerFactoryDependsOnPostProcessor} 确保
 * {@link EntityManagerFactory} bean 依赖于 elasticsearchClient bean。
 */
@Configuration
static class ElasticsearchJpaDependencyConfiguration
    extends EntityManagerFactoryDependsOnPostProcessor {

    ElasticsearchJpaDependencyConfiguration() {
        super("elasticsearchClient");
    }

}

```

79.13 Configure jOOQ with Two DataSources

If you need to use jOOQ with multiple data sources, you should create your own `DSLContext` for each one. Refer to `JooqAutoConfiguration` for more details.



In particular, `JooqExceptionTranslator` and `SpringTransactionProvider` can be reused to provide similar features to what the auto-configuration does with a single `DataSource`.

80. Database Initialization

An SQL database can be initialized in different ways depending on what your stack is. Of course, you can also do it manually, provided the database is a separate process.

80.1 Initialize a Database Using JPA

JPA has features for DDL generation, and these can be set up to run on startup against the database. This is controlled through two external properties:

- `spring.jpa.generate-ddl` (boolean) switches the feature on and off and is vendor independent.
- `spring.jpa.hibernate.ddl-auto` (enum) is a Hibernate feature that controls the behavior in a more fine-grained way. This feature is described in more detail later in this guide.

80.2 Initialize a Database Using Hibernate

You can set `spring.jpa.hibernate.ddl-auto` explicitly and the standard Hibernate property values are `none`, `validate`, `update`, `create`, and `create-drop`. Spring Boot chooses a default value for you based on whether it thinks your database is embedded. It defaults to `create-drop` if no schema manager has been detected or `none` in all other cases. An embedded database is detected by looking at the `Connection` type. `hsqldb`, `h2`, and `derby` are embedded, and others are not. Be careful when switching from in-memory to a 'real' database that you do not make assumptions about the existence of the tables and data in the new platform. You either have to set `ddl-auto` explicitly or use one of the other mechanisms to initialize the database.



You can output the schema creation by enabling the `org.hibernate.SQL` logger. This is done for you automatically if you enable the `debug` mode.

In addition, a file named `import.sql` in the root of the classpath is executed on startup if Hibernate creates the schema from scratch (that is, if the `ddl-auto` property is set to `create` or `create-drop`). This can be useful for demos and for testing if you are careful but is probably not something you want to be on the classpath in production. It is a Hibernate feature (and has nothing to do with Spring).

80.3 Initialize a Database

Spring Boot 可以自动创建您的模式（DDL脚本）`DataSource` 并初始化它（DML脚本）。：它加载SQL从标准的根类路径的位置 `schema.sql` 和 `data.sql` 分别。另外，Spring Boot 处理 `schema-${platform}.sql` 和 `data-${platform}.sql` 文件（如果存在），其中 `platform` 的值

是 `spring.datasource.platform`。这使您可以根据需要切换到数据库特定的脚本。例如，您可以选择将其设置为数据库（的供应商名称 `hsqldb`, `h2`, `oracle`, `mysql`, `postgresql`, 等）。

Spring Boot自动创建嵌入式模式`DataSource`。这种行为可以通过使用自定义`spring.datasource.initialization-mode`属性（也可以是`always`或`never`）。

默认情况下，Spring Boot启用Spring JDBC初始化程序的快速失败功能。这意味着，如果脚本导致异常，则应用程序无法启动。您可以通过设置来调整该行为`spring.datasource.continue-on-error`。



在基于JPA的应用程序中，您可以选择让Hibernate创建模式或使用`schema.sql`，但不能同时执行这两个操作。`spring.jpa.hibernate.ddl-auto`如果您使用，请确保禁用`schema.sql`。

80.4 初始化一个Spring批处理数据库

如果您使用Spring Batch，则它将为大多数常用数据库平台预先打包SQL初始化脚本。Spring Boot可以检测您的数据库类型并在启动时执行这些脚本。如果您使用嵌入式数据库，则默认情况下发生这种情况 您还可以为任何数据库类型启用它，如以下示例所示：

```
spring.batch.initialize-mode=始终
```

您也可以通过设置明确关闭初始化`spring.batch.initialize-schema=never`。

80.5 使用更高级别的数据库迁移工具

春天引导支持两种更高级别的迁移工具：[迁移](#) 和[Liquibase](#)。

80.5.1 在启动时执行Flyway数据库迁移

要在启动时自动运行Flyway数据库迁移，请将其添加`org.flywaydb:flyway-core`到类路径中。

迁移是表单中的脚本`V<VERSION>_<NAME>.sql`（带有`<VERSION>`下划线分隔的版本，如'1'或'2_1'）。默认情况下，它们位于名为的文件夹中`classpath:db/migration`，但您可以通过设置修改该位置`spring.flyway.locations`。您还可以添加一个特殊的`{vendor}`占位符来使用供应商特定的脚本。假设如下：

```
spring.flyway.locations = db / migration / {vendor}
```

`db/migration`上面的配置不是使用，而是根据数据库的类型（如`db/migration/mysql` MySQL）设置要使用的文件夹。支持的数据库列表可在`DatabaseDriver`。

有关可用设置（如架构和其他）的详细信息，请参阅flyway-core的Flyway类。另外，Spring Boot提供了一组属性（在`FlywayProperties`），可用于禁用迁移或关闭位置检查。Spring Boot调用`Flyway.migrate()`来执行数据库迁移。如果你想要更多的控制，提供一个`@Bean`实现`FlywayMigrationStrategy`。

Flyway支持SQL和Java回调。要使用基于SQL的回调，请将回调脚本放在`classpath:db/migration`文件夹中。要使用基于Java的回调，创建一个或多个实现`FlywayCallback`或最好扩展的bean`BaseFlywayCallback`。任何这样的bean都会自动注册`Flyway`。他们可以通过使用`@Order`或通过实施来订购`Ordered`。

默认情况下，Flyway 在您的上下文中自动装载（`@Primary`）`DataSource`，并将其用于迁移。如果你喜欢使用不同的`DataSource`，你可以创建一个并将其标记`@Bean`为`@FlywayDataSource`。如果你这样做，并想要两个数据源，请记住创建另一个数据源并将其标记为`@Primary`。或者，您可以`DataSource`通过设置`spring.flyway.[url,user,password]`外部属性来使用Flyway的本机。设置`spring.flyway.url`或`spring.flyway.user`足以让Flyway使用它自己的`DataSource`。如果没有设置这三个属性中的任何一个，`spring.datasource`则将使用其等效属性的值。

有一个Flyway示例，以便您可以看到如何设置。

您还可以使用Flyway为特定场景提供数据。例如，您可以将特定于测试的迁移置于其中，`src/test/resources`并且仅在您的应用程序启动测试时才运行。另外，您可以使用特定于配置文件的配置进行自定义，`spring.flyway.locations`以便某些特定配置文件处于活动状态时才会运行某些迁移。例如，在`application-dev.properties`，您可以指定以下设置：

```
spring.flyway.locations = classpath: / db / migration, classpath: / dev / db / migration
```

使用该设置，`dev/db/migration`只有在`dev`配置文件处于活动状态时才能运行迁移。

80.5.2 在启动时执行Liquibase数据库迁移

要在启动时自动运行Liquibase数据库迁移，请将其添加 `org.liquibase:liquibase-core` 到类路径中。

默认情况下，主更改日志是从中读取的 `db/changelog/db.changelog-master.yaml`，但您可以通过设置更改位置 `spring.liquibase.change-log`。除了YAML，Liquibase还支持JSON，XML和SQL更改日志格式。

默认情况下，Liquibase在您的上下文中自动装载（`@Primary`）`DataSource`，并将其用于迁移。如果你需要使用不同的`DataSource`，你可以创建一个并将其标记`@Bean`为`@LiquibaseDataSource`。如果你这样做，你想要两个数据源，记得创建另一个数据源并将其标记为`@Primary`。或者，您可以`DataSource`通过设置`spring.liquibase.[url,user,password]`外部属性来使用Liquibase的本机。设定`spring.liquibase.url`或`spring.liquibase.user`足以使Liquibase使用它自己的`DataSource`。如果没有设置这三个属性中的任何一个，`spring.datasource`则将使用其等效属性的值。

查看 [LiquibaseProperties](#) 有关可用设置的详细信息，如上下文，默认架构等。

有一个Liquibase样本，以便您可以看到如何设置。

信息

Spring Boot提供了许多包含消息的初学者。本节回答了使用Spring Boot进行消息传递时出现的问题。

81.1禁用事务性JMS会话

如果您的JMS代理不支持事务会话，则必须完全禁用事务支持。如果您自己创建`JmsListenerContainerFactory`，则无需执行任何操作，因为默认情况下无法进行事务处理。如果您想使用`DefaultJmsListenerContainerFactoryConfigurer` Spring Boot的默认值，可以禁用事务会话，如下所示：

```
@Bean
public DefaultJmsListenerContainerFactory jmsListenerContainerFactory (
    ConnectionFactory connectionFactory,
    DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory listenerFactory =
        new DefaultJmsListenerContainerFactory();
    configurer.configure(listenerFactory, connectionFactory);
    listenerFactory.setTransactionManager(null);
    listenerFactory.setSessionTransacted(false);
    return listenerFactory;
}
```

上面的例子覆盖了默认的工厂，它应该被应用到你的应用程序定义的任何其他工厂（如果有的话）。

82.批量应用程序

本节回答在Spring Boot中使用Spring Batch时出现的问题。



默认情况下，批处理应用程序需要`DataSource`存储作业详细信息。如果你想偏离这一点，你需要实施`BatchConfigurer`。请参阅 Javadoc的`@EnableBatchProcessing` 更多详细信息。

有关Spring Batch的更多信息，请参阅Spring Batch项目页面。

82.1在启动时执行Spring批处理作业

Spring Batch自动配置通过`@EnableBatchProcessing` 在上下文中的某个地方添加（从Spring Batch）启用。

默认情况下，它在启动时在应用程序上下文中执行所有操作`Jobs`（有关详细信息，请参阅`JobLauncherCommandLineRunner`）。您可以通过指定`spring.batch.job.names`（以逗号分隔的作业名称模式列表）来缩小特定作业或作业的范围。

如果应用程序上下文包含`JobRegistry`，则`spring.batch.job.names`在注册表中查找作业，而不是从上下文自动装入。对于更复杂的系统，这是一种常见模式，其中多个作业在子上下文中定义并集中注册。

有关更多详细信息，请参阅`BatchAutoConfiguration` 和`@EnableBatchProcessing`。

83.执行器

Spring Boot包含Spring Boot Actuator。本部分回答了使用中经常出现的问题。

83.1更改执行器端点的HTTP端口或地址

In a standalone application, the Actuator HTTP port defaults to the same as the main HTTP port. To make the application listen on a different port, set the external property: `management.server.port`. To listen on a completely different network address (such as when you have an internal network for management and an external one for user applications), you can also set `management.server.address` to a valid IP address to which the server is able to bind.

For more detail, see the [ManagementServerProperties](#) source code and “Section 51.2, “Customizing the Management Server Port”” in the “Production-ready features” section.

83.2 Customize the ‘whitelabel’ Error Page

如果您遇到服务器错误（使用JSON和其他媒体类型的计算机客户端应该看到具有正确错误代码的明智响应），Spring Boot会安装您在浏览器客户端中看到的‘whitelabel’错误页面。



设置`server.error.whitelabel.enabled=false`为关闭默认错误页面。这样做会恢复您正在使用的servlet容器的默认值。请注意，Spring Boot仍会尝试解决错误视图，因此您应该添加自己的错误页面，而不是完全禁用它。

用你自己覆盖错误页面取决于你使用的模板技术。例如，如果您使用Thymeleaf，则可以添加一个`error.html`模板。如果您使用FreeMarker，则可以添加一个`error.ftl`模板。一般来说，您需要[View](#)使用名称`error`或者`@Controller`处理`/error`路径的名称来解析。除非你替换了某些默认配置，否则你应该[BeanNameViewResolver](#)在你的中找到一个`ApplicationContext`，所以一个`@Bean` named `error`将是一个简单的方法。查看[ErrorMvcAutoConfiguration](#)更多选项。

有关如何在servlet容器中注册处理程序的详细信息，另请参阅“[错误处理](#)”一节。

84.安全

本节讨论有关使用Spring Boot时的安全性问题，包括使用Spring Security和Spring Boot引起的问题。

有关Spring Security的更多信息，请参阅[Spring Security项目页面](#)。

84.1关闭Spring Boot安全配置

如果在应用程序中[@Configuration](#)使用a 定义a `WebSecurityConfigurerAdapter`，它将关闭Spring Boot中的默认webapp安全设置。

84.2更改UserDetailsService和添加用户帐户

如果你提供了一个`@Bean`类型的`AuthenticationManager`，`AuthenticationProvider`或者`UserDetailsService`，默认`@Bean`为`InMemoryUserDetailsManager`不创建，让你有完整的功能集的Spring Security的可用的（如[各种身份验证选项](#)）。

添加用户帐户的最简单方法是提供您自己的`UserDetailsService`bean。

84.3在代理服务器后运行时启用HTTPS

确保所有主要端点都只能通过HTTPS访问，这对于任何应用程序来说都是一件非常重要的事情。如果您使用Tomcat作为servlet容器，那么Spring Boot会`RemoteIpValve`在检测到某些环境设置时自动添加Tomcat，并且您应该能够依靠`HttpServletRequest`它来报告它是否安全（即使是在处理代理服务器的下游真正的SSL终止）。标准行为取决于是否存在某些请求头（`x-forwarded-for`和`x-forwarded-proto`），其名称是常规的，所以它应该可以与大多数前端代理一起使用。您可以通过添加一些条目来打开阀门`application.properties`，如以下示例所示：

```
server.tomcat.remote-ip-header = x-forwarded-for
server.tomcat.protocol-header = x-forwarded-proto
```

（这些属性中的任何一个属性都会切换到阀门上，或者可以`RemoteIpValve`通过添加一个`TomcatServletWebServerFactory`bean来添加。）

要将Spring Security配置为需要所有（或某些）请求的安全通道，请考虑添加您自己的`WebSecurityConfigurerAdapter`以添加以下`HttpSecurity`配置：

```
@Configuration
public class SslWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        ...
    }
}
```

```
//自定义应用程序的安全性
    .http.requiresChannel().anyRequest().requiresSecure();
}

}
```

85.热插拔

Spring Boot支持热插拔。本节回答有关它如何工作的问题。

85.1重新加载静态内容

There are several options for hot reloading. The recommended approach is to use `spring-boot-devtools`, as it provides additional development-time features, such as support for fast application restarts and LiveReload as well as sensible development-time configuration (such as template caching). Devtools works by monitoring the classpath for changes. This means that static resource changes must be "built" for the change to take affect. By default, this happens automatically in Eclipse when you save your changes. In IntelliJ IDEA, the Make Project command triggers the necessary build. Due to the `default restart exclusions`, changes to static resources do not trigger a restart of your application. They do, however, trigger a live reload.

Alternatively, running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also allow hot-swapping of Java class changes).

Finally, the [Maven and Gradle plugins](#) can be configured (see the `addResources` property) to support running from the command line with reloading of static files directly from source. You can use that with an external css/js compiler process if you are writing that code with higher-level tools.

85.2 Reload Templates without Restarting the Container

Spring Boot支持的大多数模板技术都包含一个禁用缓存的配置选项（稍后在本文档中介绍）。如果您使用该 `spring-boot-devtools` 模块，则在开发时会自动为您配置这些属性。

85.2.1百里香叶模板

如果您使用Thymeleaf，请设置`spring.thymeleaf.cache`为`false`。请参阅 [ThymeleafAutoConfiguration](#) 其他Thymeleaf自定义选项。

85.2.2 FreeMarker模板

如果您使用FreeMarker，请设置`spring.freemarker.cache`为`false`。查看 [FreeMarkerAutoConfiguration](#) 其他FreeMarker自定义选项。

85.2.3 Groovy模板

如果您使用Groovy模板，请设置`spring.groovy.template.cache`为`false`。请参阅 [GroovyTemplateAutoConfiguration](#) 其他Groovy自定义选项。

85.3快速应用程序重新启动

该 `spring-boot-devtools` 模块包含对自动应用程序重新启动的支持。虽然速度不如JRebel这样的技术，但它通常比“冷启动”快得多。在调查本文后面讨论的一些更复杂的重新加载选项之前，您应该先尝试一下。

有关更多详细信息，请参阅第20章[开发人员工具](#)部分。

85.4重新加载Java类而不重新启动容器

许多现代IDE（Eclipse，IDEA和其他）支持热插拔字节码。因此，如果您进行的更改不会影响类或方法签名，则应该重新加载干净且无副作用。

86.建设

Spring Boot包含Maven和Gradle的构建插件。本节回答关于这些插件的常见问题。

86.1生成构建信息

Maven插件和Gradle插件都允许生成包含项目坐标，名称和版本的构建信息。插件也可以配置为通过配置添加其他属性。当这样的文件存在时，Spring Boot自动配置一个`BuildProperties` bean。

要使用Maven生成构建信息，请为`build-info`目标添加一个执行，如以下示例所示：

```
<build>
    <plugins>
        <plugin>
            <groupId> org.springframework.boot </ groupId>
            <artifactId> spring-boot-maven-plugin </ artifactId>
            <version> 2.0.1.BUILD-SNAPSHOT </ version>
            <executions>
                <execution>
                    <goals>
                        <goal> build-info </ goal>
                    </ goals>
                </ execution>
            </ executions>
        </ plugin>
    </ plugins>
</ build>
```



有关更多详细信息，请参阅[Spring Boot Maven Plugin](#)文档。

下面的例子和Gradle一样：

```
springBoot {
    buildInfo()
}
```



有关更多详细信息，请参阅[Spring Boot Gradle插件](#)文档。

86.2生成Git信息

Maven和Gradle都允许在构建项目时生成一个`git.properties`包含有关`git`源代码库状态信息的文件。

对于Maven用户，`spring-boot-starter-parent` POM包含一个预先配置的插件来生成一个`git.properties`文件。要使用它，请将以下声明添加到您的POM中：

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

Gradle用户可以使用该`gradle-git-properties` 插件获得相同的结果，如下例所示：

```
plugins{
    id "com.gorylenko.gradle-git-properties" version "1.4.17"
}
```



提交时间`git.properties`预计与以下格式匹配：`yyyy-MM-dd'T'HH:mm:ssZ`。这是上面列出的两个插件的默认格式。使用这种格式时，可以将时间解析为一种`Date`格式，并将其格式化为JSON，并由Jackson的日期序列化配置设置进行控制。

86.3自定义依赖关系版本

如果您使用直接或间接从`spring-boot-dependencies`（例如`spring-boot-starter-parent`）继承的Maven构建，但您想覆盖特定的第三方依赖项，则可以添加适当的`<properties>`元素。浏览`spring-boot-dependencies` POM以获取完整的属性列表。例如，要选择不同

的 `slf4j` 版本，您可以添加以下属性：

```
<properties>
    <slf4j.version> 1.7.5 </slf4j.version>
</ properties>
```



这样做只在你的Maven项目从（直接或间接）继承时起作用 `spring-boot-dependencies`。如果你 `spring-boot-dependencies` 在自己的 `dependencyManagement` 部分添加了 `<scope>import</scope>`，你必须自己重新定义神器，而不是重写属性。



每个Spring Boot版本都是针对这组特定的第三方依赖项进行设计和测试的。覆盖版本可能会导致兼容性问题。

86.4用Maven创建一个可执行的JAR

该 `spring-boot-maven-plugin` 可用于创建可执行的“肥肉” JAR。如果您使用 `spring-boot-starter-parent` POM，您可以声明插件并将您的罐子重新包装，如下所示：

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

如果你不使用父POM，你仍然可以使用插件。但是，您必须另外添加一个 `<executions>` 部分，如下所示：

```
<build>
    <plugins>
        <plugin>
            <groupId> org.springframework.boot </ groupId>
            <artifactId> spring-boot-maven-plugin </ artifactId>
            <version> 2.0.1.BUILD-SNAPSHOT </ version>
            <executions>
                <execution>
                    <goals>
                        <goal>重新包装</goal>
                    </goals>
                </execution>
            </executions>
        </ plugin>
    </ plugins>
</ build>
```

有关完整的使用详细信息，请参阅[插件文档](#)。

86.5使用Spring Boot应用程序作为依赖项

像战争文件一样，Spring Boot应用程序不打算用作依赖项。如果您的应用程序包含要与其他项目共享的类，则推荐的方法是将该代码移到单独的模块中。这个单独的模块可以被你的应用程序和其他项目所依赖。

如果您不能像上面推荐的那样重新安排您的代码，Spring Boot的Maven和Gradle插件必须配置为生成适合用作依赖关系的单独工件。可执行文件不能用作依赖项，因为[可执行的jar格式将应用程序类包装进来](#) `BOOT-INF/classes`。这意味着当可执行jar用作依赖项时，它们不能被找到。

为了产生两个工件，一个可以用作依赖关系和一个可执行工件，必须指定一个分类器。此分类器应用于可执行档案的名称，保留默认归档以用作依赖项。

要 `exec` 在 Maven 中配置分类器，您可以使用以下配置：

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

```

        <classifier> exec </ classifier>
    </ configuration>
</ plugin>
</ plugins>
</ build>

```

86.6当可执行jar运行时提取特定库

可执行jar中的大多数嵌套库不需要解压缩以便运行。但是，某些图书馆可能会遇到问题。例如，JRuby包含它自己的嵌套jar支持，它假设它 `jruby-complete.jar` 始终直接作为文件独立存在。

要处理任何有问题的库，可以标记特定的嵌套jar应该在第一次运行时自动解压缩到“temp文件夹”。

例如，为了表明应该通过使用Maven插件标记JRuby进行解包，您可以添加以下配置：

```

<build>
    <plugins>
        <plugin>
            <groupId> org.springframework.boot </ groupId>
            <artifactId> spring-boot-maven-plugin </ artifactId>
            <configuration>
                <requiresUnpack>
                    <dependency>
                        <groupId> org.jruby </ groupId>
                        <artifactId> jruby-complete </ artifactId>
                    </ dependency>
                </ requiresUnpack>
            </ configuration>
        </ plugin>
    </ plugins>
</ build>

```

86.7用排除项创建一个不可执行的JAR

通常，如果您将可执行文件和不可执行的jar作为两个独立的构建产品，那么可执行版本具有库jar中不需要的其他配置文件。例如，`application.yml` 配置文件可能会从不可执行的JAR中排除。

在Maven中，可执行jar必须是主要的工件，你可以为库添加一个分类的jar，如下所示：

```

<建立>
    <插件>
        <插件>
            <的groupId> org.springframework.boot </的groupId>
            <artifactId的>弹簧引导行家-插件</ artifactId的>
        </插件>
        <插件>
            <artifactId的>行家-JAR-插件</ artifactId的>
            <处决>
                <执行>
                    <ID> LIB </ ID>
                    <相位>包</相位>
                    <目标>
                        <目标>罐</目标>
                    </目标>
                    <结构>
                        <分类> LIB </分类器>
                        <excludes>
                            <排除> application.yml </ exclude>
                        </ excludes>
                    </ configuration>
                </ execution>
            </ executions>
        </ plugin>
    </ plugins>
</ build>

```

86.8远程调试Maven启动的Spring Boot应用程序

要将远程调试器附加到使用Maven启动的Spring Boot应用程序，可以使用maven插件的jvmArguments属性。

看到这个例子的更多细节。

86.9在不使用的情况下从Ant构建可执行文件 spring-boot-antlib

要使用Ant进行构建，您需要获取依赖关系，编译并创建jar或war归档文件。要使其可执行，您可以使用该spring-boot-antlib模块，也可以按照以下说明操作：

1. 如果您正在构建jar，请将应用程序的类和资源打包到嵌套BOOT-INF/classes目录中。如果您正在构建战争，请WEB-INF/classes像往常一样将应用程序的类打包到嵌套目录中。
2. 将运行时依赖关系添加到BOOT-INF/lib jar或WEB-INF/lib战争的嵌套目录中。切记不要压缩存档中的条目。
3. 将provided（嵌入容器）依赖项添加到BOOT-INF/lib jar或WEB-INF/lib-provided战争的嵌套目录中。切记不要压缩存档中的条目。
4. spring-boot-loader在档案的根目录添加这些类（以便Main-Class可用）。
5. 在清单中使用适当的启动程序（例如JarLauncher用于jar文件）作为Main-Class属性，并指定清单条目所需的其他属性 - 主要是通过设置Start-Class属性。

以下示例显示如何使用Ant构建可执行档案：

```
<target name = "build" depends = "compile" >
    <jar destfile = "target / ${ant.project.name} - ${spring-boot.version}.jar" compress = "false" >
        <mappedresources>
            <fileset dir = "target / classes" />
            <globmapper from = "*" to = "BOOT-INF / classes / *" />
        </mappedresources>
        <mappedresources>
            <fileset dir = "src / main / resources" erroronmissingdir = "false" />
            <globmapper from = "*" to = "BOOT-INF / classes / *" />
        </mappedresources>
        <mappedresources>
            <fileset dir = "${lib.dir} / runtime" />
            <globmapper from = "*" to = "BOOT-INF / lib / *" />
        </mappedresources>
        <zipfileset src = "${lib.dir} / loader / spring-boot-loader-jar - ${spring-boot.version}.jar" />
        <manifest>
            <attribute name = "Main-Class" value = "org.springframework.boot.loader.JarLauncher" />
            <attribute name = "Start-Class" value = "${start-class}" />
        </manifest>
    </jar>
</target>
```

该蚂蚁样品有build.xml一个文件manual，如果你用下面的命令运行它应该工作任务：

```
$ ant -lib <包含ivy-2.2.jar的文件夹>清洁手册
```

然后，您可以使用以下命令运行该应用程序：

```
$ java -jar target / *.jar
```

87.传统部署

Spring Boot支持传统部署以及更现代的部署形式。本节回答有关传统部署的常见问题。

87.1创建一个可部署的战争文件

生成可展开war文件的第一步是提供一个SpringBootServletInitializer子类并覆盖它的configure方法。这样做可以利用Spring Framework的Servlet 3.0支持，并允许您在应用程序由servlet容器启动时进行配置。通常，您应该更新应用程序的主类以扩展SpringBootServletInitializer，如以下示例所示：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected void configure(SpringApplicationBuilder application) {
        application.sources(Application.class);
    }
}
```

```

    公共 静态 无效的主要（字符串[]参数）抛出异常{
        SpringApplication.run (应用类, 参数);
    }
}

```

下一步是更新您的构建配置，以便您的项目生成war文件而不是jar文件。如果你使用Maven和[spring-boot-starter-parent](#)（为你配置Maven的战争插件），你所需要做的就是修改[pom.xml](#)以将包装更改为war，如下所示：

```
<包装>战争</ packaging>
```

如果您使用Gradle，则需要修改[build.gradle](#)以将战争插件应用于该项目，如下所示：

```
应用插件: '战争'
```

该过程的最后一步是确保嵌入式servlet容器不会干扰部署war文件的servlet容器。为此，您需要将嵌入式servlet容器依赖项标记为正在提供。

如果您使用Maven，则下面的示例将servlet容器（本例中为Tomcat）标记为提供：

```

<依赖关系>
<! - ... - >
<dependency>
    <groupId> org.springframework.boot </ groupId>
    <artifactId> spring-boot-starter-tomcat </ artifactId>
    <scope> provided </ scope>
</依赖关系>
<! - ... - >
</ dependencies>

```

如果您使用Gradle，则以下示例将servlet容器（本例中为Tomcat）标记为正在提供：

```

依赖关系{
    // ...
    providedRuntime'org.springframework.boot : spring-boot-starter-tomcat'
    // ...
}

```



[providedRuntime](#) 优于Gradle的[compileOnly](#)配置。在其他限制中，[compileOnly](#) 依赖关系不在测试类路径上，因此任何基于Web的集成测试都会失败。

如果您使用Spring Boot构建工具，则根据提供标记嵌入式Servlet容器依赖关系会生成一个可执行的war文件，并在[lib-provided](#)目录中打包提供的依赖项。这意味着，除了可部署到servlet容器之外，您还可以通过[java -jar](#)在命令行上使用来运行应用程序。



看一下Spring Boot的示例应用程序，以获取前面介绍的配置的基于Maven的示例。

87.2为较老的Servlet容器创建一个Deployable War File

较老的Servlet容器不支持[ServletContextInitializer](#) Servlet 3.0中使用的引导过程。你仍然可以在这些容器中使用Spring和Spring Boot，但是你将需要添加一个[web.xml](#)到你的应用程序并配置它来[ApplicationContext](#)通过a 加载[DispatcherServlet](#)。

87.3将现有的应用程序转换为Spring Boot

对于非Web应用程序，应该很容易将现有的Spring应用程序转换为Spring Boot应用程序。为此，请丢弃创建您的代码[ApplicationContext](#)并将其替换为对[SpringApplication](#)或的调用[SpringApplicationBuilder](#)。Spring MVC Web应用程序通常可以首先创建可部署的战争应用程序，然后将其稍后迁移到可执行的战争或jar。请参阅[将jar转换为war的入门指南](#)。

要通过扩展[SpringBootServletInitializer](#)（例如，在一个名为的类中[Application](#)）并添加Spring Boot [@SpringBootApplication](#)注释来创建可部署的战争，请使用类似于以下示例中所示的代码：

```

@SpringBootApplication
公共 类应用程序扩展 SpringBootServletInitializer {

    @覆盖
    保护 SpringApplicationBuilder配置 (SpringApplicationBuilder应用) {
        //自定义应用程序或致电application.sources (...) 来添加源
        //由于我们的例子本身就是一个@Configuration类 (通过@SpringBootApplication)
    }
}

```

```
//实际上我们不要“t”需要重写此方法。
退货申请;
}

}
```

请记住，无论你放在哪里，`sources`只是一个春天`ApplicationContext`。通常，任何已经有效的东西都应该在这里工作。可能稍后会删除一些bean，并让Spring Boot为它们提供自己的默认值，但在您需要之前应该可以做些工作。

可以将静态资源移动到类路径根中`/public` (`/static`或`/resources`或`/META-INF/resources`)。这同样适用于`messages.properties`(Spring Boot自动检测类路径的根目录)。

Spring `DispatcherServlet`和Spring Security的香草应用不需要进一步修改。如果您的应用程序中有其他功能(例如，使用其他servlet或过滤器)，则可能需要为您的`Application`上下文添加一些配置，方法是从中替换这些元素`web.xml`，如下所示：

- 用`@Bean`类型的`Servlet`或`ServletRegistrationBean`将安装豆在容器中，好像它是一个`<servlet/>`与`<servlet-mapping/>`在`web.xml`。
- A `@Bean`类型`Filter`或`FilterRegistrationBean`行为相似(如a`<filter/>`和`<filter-mapping/>`)。
- 一个`ApplicationContext`XML文件可以通过添加`@ImportResource`在你的`Application`。或者，已经大量使用注释配置的简单情况可以作为`@Bean`定义在几行中重新创建。

一旦战争文件正在工作，您可以通过向`main`您的方法添加一个方法来使其可执行`Application`，如以下示例所示：

```
public static void main (String [] args) {
    SpringApplication.run (应用类, 参数);
}
```



如果您打算将应用程序作为战争或作为可执行应用程序启动，则需要以可用于`SpringBootServletInitializer`回调的`main`方法和类中的方法共享构建器的自定义设置：

```
@SpringBootApplication
公共 类应用程序扩展 SpringBootServletInitializer {

    @覆盖
    保护 SpringApplicationBuilder配置 (SpringApplicationBuilder建设者) {
        返回 configureApplication (制造商);
    }

    public static void main (String [] args) {
        configureApplication (new SpringApplicationBuilder () ) .run (args);
    }

    私人 静态 SpringApplicationBuilder configureApplication (SpringApplicationBuilder建设者) {
        回报 builder.sources (申请.类) .bannerMode (Banner.Mode.OFF);
    }

}
```

应用程序可以分为多个类别：

- Servlet 3.0+应用程序没有`web.xml`。
- 应用程序使用`web.xml`。
- 具有上下文层次的应用程序
- 没有上下文层次的应用程序

所有这些都应该适合翻译，但每个可能需要稍微不同的技术。

如果Servlet 3.0+应用程序已经使用Spring Servlet 3.0+初始化器支持类，它们可能会非常容易转换。通常情况下，现有的所有代码`WebApplicationInitializer`都可以移入`SpringBootServletInitializer`。如果您现有的应用程序有多个`ApplicationContext`(例如，如果它使用`AbstractDispatcherServletInitializer`)，那么您可能能够将所有上下文源合并为一个`SpringApplication`。您可能遇到的主要难题是如果组合不起作用并且您需要维护上下文层次结构。请参阅构建示例层次结构的条目。包含Web特定功能的现有父上下文通常需要分解，以便所有`ServletContextAware`组件都位于子上下文中。

不是Spring应用程序的应用程序可能会转换为Spring Boot应用程序，前面提到的指导可能会有所帮助。但是，您可能会遇到问题。在这种情况下，我们建议用Stack的标签提问Stack Overflow`spring-boot`。

87.4将WAR部署到WebLogic

要将Spring Boot应用程序部署到WebLogic，必须确保您的Servlet初始化程序直接实现 `WebApplicationInitializer`（即使您从已经实现它的基类中进行扩展）。

WebLogic的典型初始化程序应该类似于以下示例：

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements WebApplicationInitializer {
}
```

如果使用Logback，则还需要告知WebLogic更喜欢打包的版本，而不是与服务器预安装的版本。您可以通过添加 `WEB-INF/weblogic.xml` 具有以下内容的文件来完成此操作：

```
<? xml version =“1.0”encoding =“UTF-8”? >
<wls: weblogic-web-app
    xmlns: wls = “http://xmlns.oracle.com/weblogic/weblogic-web-app”
    xmlns: xsi = “http://www.w3.org/2001/XMLSchema-instance”
    xsi: schemaLocation = “http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
        http://xmlns.oracle.com/weblogic/weblogic-web-app
        http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd” >
<wls: container-descriptor>
    <wls: prefer-application-packages>
        <wls: package- name> org.slf4j </ wls: package-name>
    </ wls: prefer-application-packages>
</ wls: container-descriptor>
</ wls: weblogic-web-app>
```

87.5在旧的（Servlet 2.5）容器中部署WAR

Spring Boot使用Servlet 3.0 API来初始化 `ServletContext`（注册 `Servlets` 等），因此您不能在Servlet 2.5容器中使用同一个应用程序。但是，可以运行一些特殊工具的旧容器上的春天启动应用程序。如果包括 `org.springframework.boot:spring-boot-legacy` 作为一个依赖（分别维持到春季启动的核心，目前可在1.1.0.RELEASE），所有你需要做的就是创建一个 `web.xml`，并宣布上下文监听器创建应用程序上下文和您的过滤器和servlet。上下文监听器是Spring Boot的特殊用途，但其余部分对于Servlet 2.5中的Spring应用程序来说是正常的。以下Maven示例显示了如何设置Spring Boot项目以在Servlet 2.5容器中运行：

```
<? XML版本=“1.0”编码=“UTF-8”? >
<web应用 版本 = “2.5” 的xmlns = “http://java.sun.com/xml/ns/javaee”
    的xmlns: 的xsi = “ http://www.w3.org/2001/XMLSchema-instance ”
    的xsi: 的schemaLocation = “ http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/ javaee / web-app_2_5.xsd”

    <context-param>
        <param-name> contextConfigLocation </ param-name>
        <param-value> demo.Application </ param-value>
    </ context-param>

    <listener>
        <listener-class> org.springframework.boot.legacy.context.web.SpringBootContextLoaderListener </ listener-class>
    </ listener>

    <filter>
        <filter-name> metricsFilter </ filter-name>
        <filter-class> org.springframework.web.filter.DelegatingFilterProxy </ filter-class>
    </ filter>

    <filter-mapping>
        <filter-name> metricsFilter </ filter-name>
        <url-pattern> /* </ url-pattern>
    </ filter-mapping>

    <servlet>
        <servlet-name> appServlet </ servlet-name>
        <servlet-class> org.springframework.web.servlet.DispatcherServlet </ servlet-class>
        <init-param>
            <param-name> contextAttribute </
            param- name> <param-value> org.springframework.web.context.WebApplicationContext.ROOT </ param-value>
        </ init-param>
```

```

<load-on-startup> 1 </ load-on-startup>
</ servlet>

<servlet-mapping>
    <servlet-name> appServlet </ servlet-name>
    <url-pattern> / </ url-pattern>
</ servlet-mapping>

</ web的应用>

```

在前面的示例中，我们使用单个应用程序上下文（由上下文侦听器创建的上下文），并 `DispatcherServlet` 使用 `init` 参数将其附加到该上下文。这在 Spring Boot 应用程序中很正常（通常只有一个应用程序上下文）。

87.6 使用Jedis代替生菜

默认情况下，Spring Boot starter（`spring-boot-starter-data-redis`）使用 `Lettuce`。您需要排除该依赖关系，并包含 `Jedis`。Spring Boot 管理这些依赖关系，以便尽可能简化此过程。

以下示例显示了如何在 Maven 中执行此操作：

```

<dependency>
    <groupId> org.springframework.boot </ groupId>
    <artifactId> spring-boot-starter-data-redis </ artifactId>
    <排除>
        <排除>
            <groupId> io.lettuce </ groupId>
            <artifactId> 莴苣芯 </ artifactId>
        </排除>
    </排除>
</依赖性>
<依赖性>
    <groupId> redis.clients </ groupId>
    <artifactId> jedis </ artifactId>
</依赖性>

```

以下示例显示了如何在 Gradle 中执行此操作：

```

配置{
    compile.exclude 模块: "生菜"
}

依赖关系{
    编译 ("redis.clients: jedis")
    // ...
}

```

第十部分附录

附录A.通用应用程序属性

可以在 `application.properties` 文件内部 `application.yml`，文件内部或命令行开关中指定各种属性。本附录提供了常用 Spring Boot 属性的列表以及对使用它们的基础类的引用。



属性贡献可能来自类路径上的其他 jar 文件，因此您不应将其视为详尽的列表。此外，你可以定义你自己的属性。



此示例文件仅作为指导。千万不能复制和粘贴的全部内容到应用程序中。相反，只挑选您需要的属性。

```

#=====
#COMMON SPRING BOOT PROPERTIES
# #
此示例文件作为指导提供。请勿将其

```

全部内容

复制到您自己的应用程序中。^^^ #===== ===== ===== ===== ===== ===== =====

```

#-----
#核心属性
#-----
debug = false #启用调试日志。
trace = false #启用跟踪日志。

#LOGGING
logging.config = #日志配置文件的位置。例如，Logback的classpath: Logback.xml`。
logging.exception-conversion-word =%wEx #记录异常时使用的转换字。
logging.file = #日志文件名（例如myapp.log`）。名称可以是确切的位置或相对于当前目录。
logging.file.max-history = 0 #要保留的归档日志文件的最大数量。仅支持默认的登录设置。
logging.file.max-size = 10MB #最大日志文件大小。仅支持默认的登录设置。
logging.level.* = #日志级别严重性映射。例如 Logging.Level.org.springframework = DEBUG`。
logging.path = #日志文件的位置。例如，`/var/log`。
logging.pattern.console = #输出到控制台的Appender模式。仅使用默认的Logback设置支持。
logging.pattern.dateformat = yyyy-MM-dd HH:mm:ss.SSS #日志格式的Appender模式。仅使用默认的Logback设置支持。
logging.pattern.file = #输出到文件的Appender模式。仅使用默认的Logback设置支持。
logging.pattern.level =%5p #日志级别的Appender模式。仅使用默认的Logback设置支持。
logging.register-shutdown-hook = false #为日志记录系统初始化时注册一个关闭钩子。

#AOP
spring.aop.auto =真 #添加@EnableAspectJAutoProxy。
spring.aop.proxy-target-class = true #是否创建基于子类的（CGLIB）代理（true），而不是基于标准Java接口的代理（false）。

# IDENTITY （ContextIdApplicationContextInitializer）
spring.application.name = #应用程序名称。

#ADMIN （SpringApplicationAdminJmxAutoConfiguration）
spring.application.admin.enabled = false #是否为应用程序启用管理功能。
spring.application.admin.jmx-name = org.springframework.boot:type = Admin, name = SpringApplication #JMX应用程序的名称admin

#AUTO-CONFIGURATION
spring.autoconfigure.exclude = #要排除的自动配置类。

#BANNER
spring.banner.charset = UTF-8 #横幅文件编码。
spring.banner.location = classpath: banner.txt #横幅文本资源位置。
spring.banner.image.location = classpath: banner.gif #横幅图像文件位置（也可以使用jpg或png）。
spring.banner.image.width = 76 #字符图片的宽度。
spring.banner.image.height = #以字符形式显示横幅图像的高度（默认基于图像高度）。
spring.banner.image.margin = 2 #在字符中留下左手边缘图像。
spring.banner.image.invert = false #图像是否应该颠倒黑暗的终端主题。

#SPRING CORE spring.beaninfo.ignore = true #是否跳过对BeanInfo类的搜索。

#SPRING CACHE （CacheProperties）
spring.cache.cache-names = #如果基础高速缓存管理器支持，将创建缓存名称的逗号分隔列表。
spring.cache.caffeine.spec = #用于创建缓存的规范。有关规格格式的更多详细信息，请参阅CaffeineSpec。
spring.cache.couchbase.expiration = 0ms #进入到期。默认情况下，这些条目永不过期。请注意，该值最终转换为秒。
spring.cache.ehcache.config = #用于初始化EhCache的配置文件的位置。
spring.cache.infinispan.config = #用于初始化Infinispan的配置文件的位置。
spring.cache.jcache.config = #用于初始化缓存管理器的配置文件的位置。
spring.cache.jcache.provider = #用于检索符合JSR-107的缓存管理器的CachingProvider实现的完全限定名称。只有在类路径中有多个JSR-107实现时才有效。
spring.cache.redis.cache-null-values = true #允许缓存空值。
spring.cache.redis.key-prefix = #键字前缀。
spring.cache.redis.time-to-live = 0ms #进入到期。默认情况下，这些条目永不过期。
spring.cache.redis.use-key-prefix = true #写入Redis时是否使用密钥前缀。
spring.cache.type = #缓存类型。默认情况下，根据环境自动检测。

#SPRING CONFIG - 仅使用环境属性（ConfigFileApplicationListener）
spring.config.additional-location = #除默认值之外使用的配置文件位置。
spring.config.location = #配置替换默认值的文件位置。
spring.config.name = application #配置文件名。

#HAZELCAST （HazelcastProperties）
spring.hazelcast.config = #用于初始化Hazelcast的配置文件的位置。

#项目信息（ProjectInfoProperties）
spring.info.build.location = classpath: META-INF / build-info.properties #生成的build-info.properties文件的位置。

```

```

spring.info.git.location =类路径: git.properties 生成的git.properties文件#所在。

#JMX
spring.jmx.default域 = #JMX域名。
spring.jmx.enabled = true #将管理bean展示给JMX域。
spring.jmx.server = mbeanServer #MBeanServer bean名称。

#电子邮件 (MailProperties)
spring.mail.default-encoding = UTF-8 #默认MimeType编码。
spring.mail.host = #SMTP服务器主机。例如, `smtp.example.com`。
spring.mail.jndi-name = #会话JNDI名称。设置时, 优先于其他邮件设置。
spring.mail.password = #登录SMTP服务器的密码。
spring.mail.port = #SMTP服务器端口。
spring.mail.properties.* = #其他JavaMail会话属性。
spring.mail.protocol = smtp #SMTP服务器使用的协议。
spring.mail.test-connection = false #是否测试邮件服务器在启动时是否可用。
spring.mail.username = #登录SMTP服务器的用户。

#应用程序设置 (SpringApplication)
spring.main.banner-mode = console #用于在应用程序运行时显示
标题的模式。spring.main.sources = #包含在ApplicationContext中的源 (类名, 包名或XML资源位置)。
spring.main.web-application-type = #显式请求特定类型的Web应用程序的标志。如果未设置, 则根据类路径自动检测。

#FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding = #应用程序必须使用的期望字符编码。

# INTERNATIONALIZATION (MessageSourceProperties)
spring.messages.always-use-message-format = false #是否始终应用MessageFormat规则, 甚至可以解析不带参数的消息。
spring.messages.basename = messages #以逗号分隔的基本名称列表 (本质上是一个完全限定的类路径位置), 每个都遵循ResourceBundle约定。
spring.messages.cache-duration = #加载的资源包文件缓存持续时间。未设置时, 捆绑包将永久缓存。如果未指定持续时间后缀, 则将使用秒。
spring.messages.encoding = UTF-8 #消息包编码。
spring.messages.fallback-to-system-locale = true #是否回退到系统区域设置, 如果没有找到特定语言环境的文件。
spring.messages.use-code-as-default-message = false #是否使用消息代码作为默认消息, 而不是抛出"NoSuchMessageException"。仅在开

#OUTPUT
spring.output.ansi.enabled =检测 #配置的ANSI输出。

#PID FILE (ApplicationPidFileWriter)
spring.pid.fail-on-write-error = #如果使用ApplicationPidFileWriter, 将失败, 但不能写入PID文件。
spring.pid.file = #要写入的PID文件的位置 (如果使用ApplicationPidFileWriter)。

#PROFILES
spring.profiles.active = #逗号分隔的有源配置文件列表。可以被命令行开关覆盖。
spring.profiles.include = #无条件激活指定的以逗号分隔的配置文件列表 (或使用YAML配置文件列表)。

#Quartz调度 (QuartzProperties)
spring.quartz.jdbc.initialize-架构 =嵌入 #数据库模式初始化模式。
spring.quartz.jdbc.schema = classpath中: 组织/石英/ IMPL / jdbcjobstore / tables_ @ @ 平台@ @ .SQL #的路径SQL文件, 以用于初
spring.quartz.job-store-type = 内存 #石英作业存储类型。
spring.quartz.properties.* = #额外的Quartz Scheduler属性。

#REACTOR (ReactorCoreProperties)
spring.reactor.stacktrace-mode.enabled = false #Reactor是否应该在运行时收集堆栈跟踪信息。

#SENDGRID (SendGridAutoConfiguration)
spring.sendgrid.api-key = #SendGrid API密钥。
spring.sendgrid.proxy.host = #SendGrid代理主机。
spring.sendgrid.proxy.port = #SendGrid代理端口。

#-----#
#WEB PROPERTIES
#-----#

#嵌入式服务器配置 (ServerProperties)
server.address = #服务器应绑定到的网络地址。
server.compression.enabled = false #是否启用响应压缩。
server.compression.excluded-user-agents = #要从压缩中排除的用户代理列表。
server.compression.mime-types = text / html, text / xml, text / plain, text / css, text / javascript, application / javascr
server.compression.min-response-size = 2048 #压缩执行所需的最小"Content-Length"值。
server.connection超时= #连接器在关闭连接之前等待另一个HTTP请求的时间。未设置时, 使用连接器的容器特定默认值。使用值-1来表示否 (即无
server.error.include-exception = false #包含"exception"属性。
server.error.include-stacktrace = never #何时包含"stacktrace"属性。
server.error.path = / error #错误控制器的路径。

```

```

server.error.whitelabel.enabled = true #是否在服务器出错时启用浏览器中显示的默认错误页面。
server.http2.enabled = false #如果当前环境支持，是否启用HTTP / 2支持。
server.jetty.acceptors = #要使用的接受者线程的数量。
server.jetty.accesslog.append = false #附加到日志。
server.jetty.accesslog.date-format = dd / MMM / yyyy: HH: mm: ss Z #请求日志的时间戳格式。
server.jetty.accesslog.enabled = false #启用访问日志。
server.jetty.accesslog.extended-format = false #启用扩展的NCSA格式。
server.jetty.accesslog.file-date-format = #放置在日志文件名中的日期格式。
server.jetty.accesslog.filename = #日志文件名。如果未指定，日志重定向到“System.err”。
server.jetty.accesslog.locale = #请求日志的语言环境。
server.jetty.accesslog.log-cookies = false #启用记录请求cookie。
server.jetty.accesslog.log-latency = false #启用记录请求处理时间。
server.jetty.accesslog.log-server = false #启用对请求主机名的记录。
server.jetty.accesslog.retention-period = 31 #删除旋转的日志文件之前的天数。
server.jetty.accesslog.time-zone = GMT #请求日志的时区。
server.jetty.max-http-post-size = 0 #HTTP帖子或放置内容的最大大小（以字节为单位）。
server.jetty.selectors = #要使用的选择器线程数。
server.max-http-header-size = 0 #HTTP消息头的最大大小（以字节为单位）。
server.port = 8080 #服务器HTTP端口。
server.server-header = #用于服务器响应头的值（如果为空，则不会发送头）。
server.use-forward-headers = #是否应将X-Forwarded- *标头应用于HttpRequest。
server.servlet.context-parameters.* = #Servlet上下文初始化参数。
server.servlet.context-path = #应用程序的上下文路径。
server.servlet.application-display-name = application #显示
应用程序的名称。server.servlet.jsp.class-name = org.apache.jasper.servlet.JspServlet #JSP servlet的类名称。
server.servlet.jsp.init-parameters.* = #用于配置JSP servlet的初始参数。
server.servlet.jsp.registered = true #JSP servlet是否已注册。
server.servlet.path = / #主调度程序servlet的路径。
server.servlet.session.cookie.comment = #评论会话cookie。
server.servlet.session.cookie.domain = #会话cookie的域名。
server.servlet.session.cookie.http-only = #会话cookie的“HttpOnly”标志。
server.servlet.session.cookie.max-age = #会话cookie的最大年龄。如果未指定持续时间后缀，则将使用秒。
server.servlet.session.cookie.name = #会话cookie名称。
server.servlet.session.cookie.path = #会话cookie的路径。
server.servlet.session.cookie.secure = #会话cookie的“安全”标志。
server.servlet.session.persistent = false #是否在重新启动之间保留会话数据。
server.servlet.session.store-dir = #用于存储会话数据的目录。
server.servlet.session.timeout = #会话超时。如果未指定持续时间后缀，则将使用秒。
server.servlet.session.tracking-modes = #会话跟踪模式（以下一项或多项：“cookie”，“url”，“ssl”）。
server.ssl.ciphers = #支持的SSL密码。
server.ssl.client-auth = #是否需要客户端身份验证（“需要”）或需要（“需要”）。需要信任商店。
server.ssl.enabled = #启用SSL支持。
server.ssl.enabled-protocols = #启用SSL协议。
server.ssl.key-alias = #标识密钥库中密钥的别名。
server.ssl.key-password = #用于访问密钥存储区中密钥的密码。
server.ssl.key-store = #保存SSL证书的密钥存储区的路径（通常是一个jks文件）。
server.ssl.key-store-password = #用于访问密钥存储区的密码。
server.ssl.key-store-provider = #密钥存储的提供者。
server.ssl.key-store-type = #密钥存储的类型。
server.ssl.protocol = 要使用的
    TLS #SSL协议。server.ssl.trust-store = #持有SSL证书的信任库。
server.ssl.trust-store-password = #用于访问信任存储的密码。
server.ssl.trust-store-provider = #信任存储的提供程序。
server.ssl.trust-store-type = #信任存储的类型。
server.tomcat.accept-count = 0 #所有可能的请求处理线程正在使用时传入连接请求的最大队列长度。
server.tomcat.accesslog.buffered = true #是否缓冲输出，使其仅定期刷新。
server.tomcat.accesslog.directory = logs #创建日志文件的目录。可以是绝对的或相对于Tomcat的基本目录。
server.tomcat.accesslog.enabled = false #启用访问日志。
server.tomcat.accesslog.file最新格式= .yyyy-MM-dd #放置在日志文件名中的日期格式。
server.tomcat.accesslog.pattern = common #访问日志的格式模式。
server.tomcat.accesslog.prefix = access_log #记录文件名前缀。
server.tomcat.accesslog.rename-on-rotate = false #是否推迟在文件名中包含日期标记，直到旋转时间。
server.tomcat.accesslog.request-attributes-enabled = false #为请求使用的IP地址，主机名，协议和端口设置请求属性。
server.tomcat.accesslog.rotate = true #是否启用访问日志循环。
server.tomcat.accesslog.suffix = .log #日志文件名后缀。
server.tomcat.additional-tld-skip-patterns = #与TLD扫描相匹配的要匹配的瓶子的逗号分隔列表。
server.tomcat.background-processor-delay = 30s #调用backgroundProcess方法之间的延迟。如果未指定持续时间后缀，则将使用秒。
server.tomcat.basedir = #Tomcat基本目录。如果未指定，则使用临时目录。
server.tomcat.internal-proxies = 10 \\. \\ d {1,3} \\ . \\ d {1,3} \\ . \\ d {1,3} | \\
    . 192 \\ 168 \\ d {1,3} \\ d {1,3} | \\
    . 169 \\ 254 \\ d {1,3} \\ d {1,3} | \\
    . 127 \\ d {1,3} \\ d {1,3} \\ d {1,3} | \\
    172 \\ 1 [6-9] {1} \\ d {1,3} \\ d {1,3} | . . \\
    172 \\ 2 [0-9] {1} \\ d {1,3} \\ d {1,3} | . .

```

```

172 \\. 3 [0-1] {1} \\. \\d {1,3} \\. \\d {1,3} #匹配可信IP地址的正则表达式。
server.tomcat.max-connections = 0 #服务器在任何给定时间接受和处理的最大连接数。
server.tomcat.max-http-header-size = 0 #HTTP消息头的最大大小（以字节为单位）。
server.tomcat.max-http-post-size = 0 #HTTP邮件内容的最大大小（以字节为单位）。
server.tomcat.max-threads = 0 #工作线程的最大数量。
server.tomcat.min-spare-threads = 0 #工作线程的最小数量。
server.tomcat.port-header = X-Forwarded-Port #用于覆盖原始端口值的HTTP标头的名称。
server.tomcat.protocol-header = #保存传入协议的头部，通常名为“X-Forwarded-Proto”。
server.tomcat.protocol-header-https-value = https #协议头的值，指示传入请求是否使用SSL。
server.tomcat.redirect-context-root = #是否应通过将/附加到路径来重定向对上下文根的请求。
server.tomcat.remote-ip-header = #从中提取远程IP的HTTP头的名称。例如，“X-FORWARDED-FOR”。
server.tomcat.resource.cache-ttl = #静态资源缓存的生存时间。
server.tomcat.uri-encoding = UTF-8 #用于解码URI的字符编码。
server.tomcat.use-relative-redirects = #对于sendRedirect调用生成的HTTP 1.1和更高版本位置标头是否使用相对或绝对重定向。
server.undertow.accesslog.dir = #
取消访问日志目录。server.undertow.accesslog.enabled = false #是否启用访问日志。
server.undertow.accesslog.pattern = common #访问日志的格式模式。
server.undertow.accesslog.prefix = access_log。#日志文件名称前缀。
server.undertow.accesslog.rotate = true #是否启用访问日志轮换。
server.undertow.accesslog.suffix = log #日志文件名后缀。
server.undertow.buffer-size = #每个缓冲区的大小，以字节为单位。
server.undertow.direct-buffers = #是否在Java堆外分配缓冲区。
server.undertow.io-threads = #为worker创建的I / O线程数量。
server.undertow.eager-filter-init = true #是否应该在启动时初始化servlet过滤器。
server.undertow.max-http-post-size = 0 #HTTP邮件内容的最大大小（以字节为单位）。
server.undertow.worker-threads = #工作线程数。

#FREEMARKER (FreeMarkerProperties)
spring.freemarker.allow-request-override = false #是否允许HttpServletRequest属性覆盖（隐藏）具有相同名称的控制器生成的模型属性。
spring.freemarker.allow-session-override = false #是否允许HttpSession属性覆盖（隐藏）具有相同名称的控制器生成的模型属性。
spring.freemarker.cache = false #是否启用模板缓存。
spring.freemarker.charset = UTF-8 #模板编码。
spring.freemarker.check-template-location = true #是否检查模板位置是否存在。
spring.freemarker.content-type = text / html #Content-Type值。
spring.freemarker.enabled = true #是否为此技术启用MVC视图分辨率。
spring.freemarker.expose-request-attributes = false #在与模板合并之前是否应将所有请求属性添加到模型中。
spring.freemarker.expose-session-attributes = false #是否应该在与模板合并之前将所有HttpSession属性添加到模型中。
spring.freemarker.expose-spring-macro-helpers = true #是否公开名为“springMacroRequestContext”的Spring的宏库使用的RequestCon
spring.freemarker.prefer-file-system-access = true #是否喜欢文件系统访问模板加载。文件系统访问使模板更改的热检测成为可能。
spring.freemarker.prefix = #构建URL时预先查看名称的前缀。
spring.freemarker.request-context-attribute = #所有视图的
RequestContext属性的名称。spring.freemarker.settings.* = #众所周知的传递给FreeMarker配置的FreeMarker密钥。
spring.freemarker.suffix = .ftl #在构建URL时附加到查看名称的后缀。
spring.freemarker.template-loader-path = classpath: / templates / #逗号分隔的模板路径列表。
spring.freemarker.view-names = #可以解析的视图名称的白名单。

#GROOVY TEMPLATES (GroovyTemplateProperties)
spring.groovy.template.allow-request-override = false #是否允许HttpServletRequest属性覆盖（隐藏）具有相同名称的控制器生成的模
spring.groovy.template.allow-session-override = false #是否允许HttpSession属性覆盖（隐藏）具有相同名称的控制器生成的模型属性。
spring.groovy.template.cache = false #是否启用模板缓存。
spring.groovy.template.charset = UTF-8 #模板编码。
spring.groovy.template.check-template-location = true #是否检查模板位置是否存在。
spring.groovy.template.configuration.* = #请参阅GroovyMarkupConfigurer
spring.groovy.template.content-type = text / html #Content-Type值。
spring.groovy.template.enabled = true #是否为此技术启用MVC视图分辨率。
spring.groovy.template.expose-request-attributes = false #在与模板合并之前是否应将所有请求属性添加到模型中。
spring.groovy.template.expose-session-attributes = false #是否应该在与模板合并之前将所有HttpSession属性添加到模型中。
spring.groovy.template.expose-spring-macro-helpers = true #是否公开名为“springMacroRequestContext”的Spring的宏库使用的Reque
spring.groovy.template.prefix = #构建URL时预先查看名称的前缀。
spring.groovy.template.request-context-attribute = #所有视图的
RequestContext属性的名称。spring.groovy.template.resource-loader-path = classpath: / templates / #模板路径。
spring.groovy.template.suffix = .tpl #在构建URL时被附加到视图名称后缀。
spring.groovy.template.view-names = #可以解析的视图名称的白名单。

#SPRING HATEOAS (HateoasProperties)
spring.hateoas.use-hal-as-default-json-media-type = true #应用程序/ hal + json响应是否应发送到接受application / json的请求。

#HTTP 消息转换spring.http.converters.preferred-json-mapper = #用于HTTP消息转换的首选JSON映射器。默认情况下，根据环境自动检测。

#HTTP 编码 (HttpEncodingProperties)
spring.http.encoding.charset = UTF-8 #HTTP请求和响应的字符集。如果未明确设置，则添加到“Content-Type”标题中。
spring.http.encoding.enabled = true #是否启用http编码支持。
spring.http.encoding.force = #是否强制编码到HTTP请求和响应的配置字符集。

```

```

spring.http.encoding.force-request = #是否强制编码到HTTP请求上配置的字符集。未指定“强制”时默认为true。
spring.http.encoding.force-response = #是否强制编码到HTTP响应上配置的字符集。
spring.http.encoding.mapping = #映射的编码区域。

#MULTIPART (MultipartProperties)
spring.servlet.multipart.enabled = true #是否启用对分段上传的支持。
spring.servlet.multipart.file-size-threshold = 0 #文件写入磁盘后的阈值。值可以使用后缀“MB”或“KB”分别表示兆字节或千字节。
spring.servlet.multipart.location = #上传文件的中间位置。
spring.servlet.multipart.max-file-size = 1MB #最大文件大小。值可以使用后缀“MB”或“KB”分别表示兆字节或千字节。
spring.servlet.multipart.max-request-size = 10MB #最大请求大小。值可以使用后缀“MB”或“KB”分别表示兆字节或千字节。
spring.servlet.multipart.resolve-lazily = false #是否在文件或参数访问时懒惰地解析多部分请求。

# JACKSON (JacksonProperties)
spring.jackson.date-format = #日期格式字符串或完全合格的日期格式类名称。例如，`yyyy-MM-dd HH:mm:ss`。
spring.jackson.default-property-inclusion = #在序列化过程中控制属性的包含。使用Jackson的JsonInclude.Include枚举中的一个值进行配置。spring.jackson.deserialization.* = #杰克逊开/关功能，影响spring.jackson.generator.* = #生成器的Jackson开/关功能。
spring.jackson.joda-date-time-format = #乔达日期时间格式字符串。如果未配置，如果使用格式字符串配置“date-format”作为后备。
spring.jackson.locale = #用于格式化的区域设置。
spring.jackson.mapper.* = #杰克逊通用开/关功能。
spring.jackson.parser.* = #解析器的Jackson开/关功能。
spring.jackson.property-naming-strategy = #Jackson的PropertyNamingStrategy上的常量之一。也可以是PropertyNamingStrategy子类的spring.jackson.serialization.* = #杰克逊开/关功能，影响Java对象序列化的方式。
spring.jackson.time-zone = #格式化日期时使用的时区。例如“America / Los_Angeles”或“GMT + 10”。

#GSON (GsonProperties)
spring.gson.date-format = #在序列化Date对象时使用的格式。
spring.gson.disable-html-escaping = #是否禁用HTML字符的转义，例如'<', '>'等。
spring.gson.disable-inner-class-serialization = #是否在期间排除内部类序列化。
spring.gson.enable-complex-map-key-serialization = #是否启用复杂映射键的序列化（而非基元化）。
spring.gson.exclude-fields-without-expose-annotation = #是否排除没有“Expose”注释的序列化或反序列化考虑的所有字段。
spring.gson.field-naming-policy = #在序列化和反序列化过程中应该应用于对象字段的命名策略。
spring.gson.generate-non-executable-json = #是否通过在输出前添加一些特殊文本来生成不可执行的JSON。
spring.gson.lenient = #是否对分析不符合RFC 4627的JSON宽容。
spring.gson.long-serialization-policy = #长和长类型的序列化策略。
spring.gson.pretty-printing = #是否输出适合漂亮打印的页面的序列化JSON。
spring.gson.serialize-nulls = #是否序列化空字段。

# JERSEY (JerseyProperties)
spring.jersey.application-path = #作为应用程序的基本URI的路径。如果指定，则覆盖“@ApplicationPath”的值。
spring.jersey.filter.order = 0 #Jersey过滤器链顺序。
spring.jersey.init.* = #通过servlet或过滤器传递给Jersey的初始化参数。
spring.jersey.servlet.load-on-startup = -1 #加载泽西servlet的启动优先级。
spring.jersey.type = servlet #Jersey集成类型。

#SPRING LDAP (LdapProperties)
spring.ldap.anonymous-read-only = false #只读操作是否应使用匿名环境。
spring.ldap.base = #所有操作应从其发起的基本后缀。
spring.ldap.base-environment.* = #LDAP规范设置。
spring.ldap.password = #登录服务器的密码。
spring.ldap.urls = #服务器的LDAP URL。
spring.ldap.username = #登录服务器的用户名。

#EMBEDDED LDAP (EmbeddedLdapProperties)
spring.ldap.embedded.base-dn = #基本DN的列表。
spring.ldap.embedded.credential.username = #嵌入式LDAP用户名。
spring.ldap.embedded.credential.password = #嵌入式LDAP密码。
spring.ldap.embedded.ldif = classpath: schema.ldif #Schema (LDIF) 脚本资源引用。
spring.ldap.embedded.port = 0 #嵌入式LDAP端口。
spring.ldap.embedded.validation.enabled = true #是否启用LDAP模式验证。
spring.ldap.embedded.validation.schema = #自定义模式的路径。

#MUSTACHE TEMPLATES (MustacheAutoConfiguration)
spring.mustache.allow-request-override = false #是否允许HttpServletRequest属性覆盖（隐藏）同名控制器生成的模型属性。
spring.mustache.allow-session-override = false #是否允许HttpSession属性覆盖（隐藏）控制器生成的具有相同名称的模型属性。
spring.mustache.cache = false #是否启用模板缓存。
spring.mustache.charset = UTF-8 #模板编码。
spring.mustache.check-template-location = true #是否检查模板位置是否存在。
spring.mustache.content-type = text / html #Content-Type值。
spring.mustache.enabled = true #是否为此技术启用MVC视图分辨率。
spring.mustache.expose-request-attributes = false #是否所有请求属性都应该在与模板合并之前添加到模型中。
spring.mustache.expose-session-attributes = false #是否应该在与模板合并之前将所有HttpSession属性添加到模型中。
spring.mustache.expose-spring-macro-helpers = true #是否公开名为“springMacroRequestContext”的Spring的宏库使用的RequestConte。
spring.mustache.prefix= classpath: / templates / #应用于模板名称的前缀。

```



```

spring.session.redis.namespace = spring: session #用于存储会话的密钥的命名空间。

#THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.cache = true #是否启用模板缓存。
spring.thymeleaf.check-template = true #是否在渲染之前检查模板是否存在。
spring.thymeleaf.check-template-location = true #是否检查模板位置是否存在。
spring.thymeleaf.enabled = true #是否为Web框架启用Thymeleaf视图分辨率。
spring.thymeleaf.enable-spring-el-compiler = false #在SpringEL表达式中启用SpringEL编译器。
spring.thymeleaf.encoding = UTF-8 #模板文件编码。
spring.thymeleaf.excluded-view-names = #应该从解析中排除的逗号分隔的视图名称列表（允许的模式）。
spring.thymeleaf.mode = HTML #应用于模板的模板模式。另请参阅Thymeleaf的TemplateMode枚举。
spring.thymeleaf.prefix = classpath: / templates / #构建URL时预先查看名称的前缀。
spring.thymeleaf.reactive.chunked-mode-view-names = #逗号分隔的视图名称列表（允许的模式），当设置最大块大小时，应该是CHUNKED模式。
spring.thymeleaf.reactive.full-mode-view-names = #即使设置了最大块大小，也应该在FULL模式下执行逗号分隔的视图名称列表（允许的模式）。
spring.thymeleaf.reactive.max-chunk-size = 0 #用于写入响应的数据缓冲区的最大大小（以字节为单位）。
spring.thymeleaf.reactive.media-types = #视图技术支持的媒体类型。
spring.thymeleaf.servlet.content-type = text / html #写入HTTP响应的Content-Type值。
spring.thymeleaf.suffix = .html #在构建URL时附加到视图名称的后缀。
spring.thymeleaf.template-resolver-order = #链中模板解析器的顺序。
spring.thymeleaf.view-name = #可以解析的逗号分隔的视图名称列表（允许的模式）。

#SPRING WEBFLUX (WebFluxProperties)
spring.webflux.date-format = #要使用的日期格式。例如，`dd / MM / yyyy`。
spring.webflux.static-path-pattern = / ** #用于静态资源的路径模式。

#SPRING WEB SERVICES (WebServicesProperties)
spring.webservices.path = / services #作为服务基础URI的路径。
spring.webservices.servlet.init = #传递给Spring Web Services的Servlet init参数。
spring.webservices.servlet.load-on-startup = -1 #加载Spring Web Services servlet的启动优先级。
spring.webservices.wsdl-locations = #以逗号分隔的WSDL位置以及随附的XSD将作为bean公开的位置列表。

#-----
#SECURITY PROPERTIES
#-----
#SECURITY (SecurityProperties)
spring.security.filter.order = -100 #安全过滤器链顺序。
spring.security.filter.dispatcher-types = async, error, request #安全性筛选器链调度程序类型。
spring.security.user.name = user #默认用户名。
spring.security.user.password = #默认用户名的密码。
spring.security.user.roles = #授予默认用户名的角色。

#SECURITY OAUTH2 客户端 (OAuth2ClientProperties)
spring.security.oauth2.client.provider.* = #OAuth提供程序详细信息。
spring.security.oauth2.client.registration.* = #OAuth客户端注册。

#-----
#DATA PROPERTIES
#-----

#FLYWAY (FlywayProperties)
spring.flyway.baseline-description = #
spring.flyway.baseline-on-migrate = #
spring.flyway.baseline-version = 1 #开始迁移的版本。
spring.flyway.check-location = true #是否检查是否存在迁移脚本位置。
spring.flyway.clean-disabled = #
spring.flyway.clean-on-validation-error = #
spring.flyway.dry-run-output = #
spring.flyway.enabled = true #是否启用Flyway。
spring.flyway.encoding = #
spring.flyway.error-handlers = #
spring.flyway.group = #
spring.flyway.ignore-future-migrations = #
spring.flyway.ignore-missing-migrations = #
spring.flyway.init-sqls = #SQL语句在获得它之后立即执行初始化连接。
spring.flyway.installed-by = #
spring.flyway.locations = classpath: db / migration #迁移脚本的位置。
spring.flyway.mixed = #
spring.flyway.out-of-order = #
spring.flyway.password = #要使用Flyway创建自己的DataSource的JDBC密码。
spring.flyway.placeholder-prefix = #
spring.flyway.placeholder-replacement = #
spring.flyway.placeholder-suffix = #

```

```

spring.flyway.placeholders.* = #
spring.flyway.repeatable-sql-migration-prefix = #
spring.flyway.schemas = #要更新的模式
spring.flyway.skip-default-callbacks = #
spring.flyway.skip-default-resolvers = #
spring.flyway.sql-migration-prefix = V #
spring.flyway.sql-migration-separator = #
spring.flyway.sql-migration-suffix = .sql #
spring.flyway.sql-migration-suffixes = #
spring.flyway.table = #
spring.flyway.target = #
spring.flyway.undo-sql-migration-prefix = #
spring.flyway.url = #要迁移的数据库的JDBC URL。如果未设置，则使用主要配置的数据源。
spring.flyway.user = #登录要迁移的数据库的用户。
spring.flyway.validate-on-migrate = #

#LIQUIBASE (LiquibaseProperties)
spring.liquibase.change-log = classpath: /db/changelog/db.changelog-master.yaml #更改日志配置路径。
spring.liquibase.check-change-log-location = true #是否检查更改日志位置是否存在。
spring.liquibase.contexts = #使用的运行时上下文的逗号分隔列表。
spring.liquibase.default-schema = #默认数据库模式。
spring.liquibase.drop-first = false #是否首先删除数据库模式。
spring.liquibase.enabled = true #是否启用Liquibase支持。
spring.liquibase.labels = #使用的运行时标签的逗号分隔列表。
spring.liquibase.parameters.* = #更改日志参数。
spring.liquibase.password = #登录要迁移的数据库的密码。
spring.liquibase.rollback-file = #执行更新时写回滚SQL的文件。
spring.liquibase.url = #要迁移的数据库的JDBC URL。如果未设置，则使用主要配置的数据源。
spring.liquibase.user = #登录要迁移的数据库的用户。

#COUCHBASE (CouchbaseProperties)
spring.couchbase.bootstrap-hosts = #从中引导的Couchbase节点（主机或IP地址）。
spring.couchbase.bucket.name = default #要连接的存储桶的名称。
spring.couchbase.bucket.password = #桶的密码。
spring.couchbase.env.endpoints.key-value = 1 #针对键/值服务的每个节点的套接字数量。
spring.couchbase.env.endpoints.query = 1 #针对查询（N1QL）服务的每个节点的套接字数量。
spring.couchbase.env.endpoints.view = 1 #针对视图服务的每个节点的套接字数量。
spring.couchbase.env.ssl.enabled = #是否启用SSL支持。除非另有规定，否则如果提供“keyStore”，则自动启用。
spring.couchbase.env.ssl.key-store = #持有证书的JVM密钥存储的路径。
spring.couchbase.env.ssl.key-store-password = #用于访问密钥存储区的密码。
spring.couchbase.env.timeouts.connect = 5000ms #桶连接超时。
spring.couchbase.env.timeouts.key-value = 2500ms #在特定的按键超时上执行阻塞操作。
spring.couchbase.env.timeouts.query = 7500ms #N1QL查询操作超时。
spring.couchbase.env.timeouts.socket-connect = 1000ms #套接字连接超时。
spring.couchbase.env.timeouts.view = 7500ms #定期和地理空间视图操作超时。

#DAO (PersistenceExceptionTranslationAutoConfiguration)
spring.dao.exceptiontranslation.enabled = true #是否启用PersistenceExceptionTranslationPostProcessor。

#CASSANDRA (CassandraProperties)
spring.data.cassandra.cluster-name = #Cassandra集群的名称。
spring.data.cassandra.compression = none #Cassandra二进制协议支持的压缩。
spring.data.cassandra.connect-timeout = #套接字选项：连接超时。
spring.data.cassandra.consistency-level = #查询一致性级别。
spring.data.cassandra.contact-points = localhost #集群节点地址。
spring.data.cassandra.fetch-size = #查询默认获取大小。
spring.data.cassandra.keyspace-name = #使用的Keyspace名称。
spring.data.cassandra.load-balancing-policy = #负载均衡策略的类名称。
spring.data.cassandra.port = #Cassandra服务器的端口。
spring.data.cassandra.password = #登录服务器的密码。
spring.data.cassandra.pool.heartbeat-interval = 30s #心跳间隔后，在空闲连接上发送消息以确保其仍处于活动状态。如果未指定持续时间，将使用秒。
spring.data.cassandra.pool.idle-timeout = 120s #空闲连接被移除前的空闲超时。如果未指定持续时间后缀，则将使用秒。
spring.data.cassandra.pool.max-queue-size = 256 #如果没有连接可用，请求排队的最大请求数。
spring.data.cassandra.pool.pool-timeout = 5000ms #尝试从主机池获取连接时的池超时。
spring.data.cassandra.read-timeout = #套接字选项：读取超时。
spring.data.cassandra.reconnection-policy = #重新连接策略类。
spring.data.cassandra.repositories.type = auto #启用Cassandra存储库的类型。
spring.data.cassandra.retry-policy = #重试策略的类名称。
spring.data.cassandra.serial-consistency-level = #查询串行一致性级别。
spring.data.cassandra.schema-action = none #在启动时采取的模式操作。
spring.data.cassandra.ssl = false #启用SSL支持。
spring.data.cassandra.username = #服务器的登录用户。

#DATA COUCHBASE (CouchbaseDataProperties)

```

```

spring.data.couchbase.auto-index = false #自动创建视图和索引。
spring.data.couchbase.consistency = read-your-own-writes #在生成的查询中默认应用的一致性。
spring.data.couchbase.repositories.type = auto #启用的Couchbase存储库的类型。

#ELASTICSEARCH (ElasticsearchProperties)
spring.data.elasticsearch.cluster-name = elasticsearch #Elasticsearch集群名称。
spring.data.elasticsearch.cluster-nodes = #以逗号分隔的集群节点地址列表。
spring.data.elasticsearch.properties.* = #用于配置客户端的其他属性。
spring.data.elasticsearch.repositories.enabled = true #是否启用Elasticsearch存储库。

#DATA LDAP spring.data.ldap.repositories.enabled = true #是否启用LDAP存储库。

#MONGODB (MongoProperties)
spring.data.mongodb.authentication-database = #认证数据库名称。
spring.data.mongodb.database = #数据库名称。
spring.data.mongodb.field-naming-strategy = #要使用的FieldNamingStrategy的完全限定名称。
spring.data.mongodb.grid-fs-database = #GridFS数据库名称。
spring.data.mongodb.host = #Mongo服务器主机。不能使用URI进行设置。
spring.data.mongodb.password = #登录mongo服务器的密码。不能使用URI进行设置。
spring.data.mongodb.port = #Mongo服务器端口。不能使用URI进行设置。
spring.data.mongodb.repositories.type = auto #启用Mongo存储库的类型。
spring.data.mongodb.uri = mongodb://localhost/test #Mongo数据库URI。无法使用主机，端口和凭证进行设置。
spring.data.mongodb.username = #mongo服务器的登录用户。不能使用URI进行设置。

#DATA REDIS
spring.data.redis.repositories.enabled = true #是否启用Redis存储库。

#NEO4J (Neo4jProperties)
spring.data.neo4j.auto-index = none #自动索引模式。
spring.data.neo4j.embedded.enabled = true #是否在嵌入式驱动程序可用时启用嵌入式模式。
spring.data.neo4j.open-in-view = true #注册OpenSessionInViewInterceptor。将Neo4j会话绑定到线程，以完成请求的整个处理。
spring.data.neo4j.password = #登录服务器的密码。
spring.data.neo4j.repositories.enabled = true #是否启用Neo4j存储库。
spring.data.neo4j.uri = 驱动程序使用的#URI。自动检测默认。
spring.data.neo4j.username = #服务器的登录用户。

#DATA REST (RepositoryRestProperties)
spring.data.rest.base-path = #Spring Data REST用于公开资源库资源的基础路径。
spring.data.rest.default-media-type = #当没有指定任何内容时，默认使用的内容类型。
spring.data.rest.default-page-size = #
页面的默认大小。spring.data.rest.detection-strategy = default #用于确定哪些存储库暴露的策略。
spring.data.rest.enable-enum-translation = #是否通过Spring Data REST默认资源包启用枚举值转换。
spring.data.rest.limit-param-name = #URL查询字符串参数的名称，指示一次返回多少个结果。
spring.data.rest.max-page-size = #
页面的最大尺寸。spring.data.rest.page-param-name = #指示要返回哪个页面的URL查询字符串参数的名称。
spring.data.rest.return-body-on-create = #是否在创建实体后返回响应主体。
spring.data.rest.return-body-on-update = #是否在更新实体后返回响应主体。
spring.data.rest.sort-param-name = #URL查询字符串参数的名称，指示对结果进行排序的方向。

#SOLR (SolrProperties)
spring.data.solr.host = http://127.0.0.1:8983/solr #Solr主机。如果设置了“zk-host”，则忽略。
spring.data.solr.repositories.enabled = true #是否启用Solr存储库。
spring.data.solr.zk-host = #HOST:PORT形式的#ZooKeeper主机地址。

#DATA WEB (SpringDataWebProperties)
spring.data.web.pageable.default-page-size = 20 #缺省页大小。
spring.data.web.pageable.max-page-size = 2000 #要接受的最大页面大小。
spring.data.web.pageable.one-indexed-parameters = false #是否公开并假设基于1的页码索引。
spring.data.web.pageable.page-parameter = page #页面索引参数名称。
spring.data.web.pageable.prefix = #页面编号和页面大小参数前面的一般前缀。
spring.data.web.pageable.qualifier-delimiter = _ #限定符与实际页码和大小属性之间使用的分隔符。
spring.data.web.pageable.size-parameter = size #页面大小参数名称。
spring.data.web.sort.sort-parameter = sort #排序参数名称。

#DATASOURCE (DataSourceAutoConfiguration&DataSourceProperties)
spring.datasource.continue-on-error = false #是否在初始化数据库时发生错误时停止。
spring.datasource.data = #数据(DML)脚本资源引用。
spring.datasource.data-username = #执行DML脚本的数据库用户名（如果不同）。
spring.datasource.data-password = #执行DML脚本的数据库的密码（如果不同）。
spring.datasource.dbcp2.* = #Commons DBCP2特定设置
spring.datasource.driver-class-name = #JDBC驱动程序的完全限定名称。默认情况下基于URL自动检测。
spring.datasource.generate-unique-name = false #是否生成随机数据源名称。
spring.datasource.hikari.* = #Hikari特定设置

```

```

spring.datasource.initialization-mode = embedded # 使用可用的DDL和DML脚本初始化数据源。
spring.datasource.jmx-enabled = false # 是否启用JMX支持（如果由底层池提供）。
spring.datasource.jndi-name = # 数据源的JNDI位置。设置时会忽略类，网址，用户名和密码。
spring.datasource.name = # 数据源的名称。使用嵌入式数据库时，默认为“testdb”。
spring.datasource.password = # 登录数据库的密码。
spring.datasource.platform = all # 在DDL或DML脚本中使用的平台（例如schema - $ {platform} .sql 或 data - $ {platform} .sql）。
spring.datasource.schema = # 架构（DDL）脚本资源引用。
spring.datasource.schema-username = # 执行DDL脚本的数据库的用户名（如果不同）。
spring.datasource.schema-password = # 执行DDL脚本的数据库的密码（如果不同）。
spring.datasource.separator =; # SQL 初始化脚本中的语句分隔符。
spring.datasource.sql-script-encoding = # SQL 脚本编码。
spring.datasource.tomcat.* = # Tomcat 数据源特定设置。
spring.datasource.type = # 要使用的连接池实现的完全限定名称。默认情况下，它是从类路径中自动检测的。
spring.datasource.url = # 数据库的 JDBC URL。
spring.datasource.username = # 登录数据库的用户名。
spring.datasource.xa.data-source-class-name = # XA 数据源完全限定名称。
spring.datasource.xa.properties = # 传递给XA 数据源的属性。

# JEST (Elasticsearch HTTP 客户端) (JestProperties)
spring.elasticsearch.jest.connection-timeout = 3s # 连接超时。
spring.elasticsearch.jest.multi-threaded = true # 是否启用来自多个执行线程的连接请求。
spring.elasticsearch.jest.password = # 登录密码。
spring.elasticsearch.jest.proxy.host = # HTTP 客户端应该使用的代理主机。
spring.elasticsearch.jest.proxy.port = # HTTP 客户端应该使用的代理端口。
spring.elasticsearch.jest.read-timeout = 3s # 读取超时。
spring.elasticsearch.jest.uris = http://localhost:9200 # 要使用的Elasticsearch 实例的逗号分隔列表。
spring.elasticsearch.jest.username = # 登录用户名。

# H2 Web 控制台 (H2ConsoleProperties)
spring.h2.console.enabled = false # 是否启用控制台。
spring.h2.console.path = / h2-console # 控制台可用的路径。
spring.h2.console.settings.trace = false # 是否启用跟踪输出。
spring.h2.console.settings.web-allow-others = false # 是否启用远程访问。

# InfluxDB (InfluxDbProperties)
spring.influx.password = # 登录密码。
spring.influx.url = # 要连接的InfluxDB 实例的URL。
spring.influx.user = # 登录用户。

# JOOQ (JooqProperties)
spring.jooq.sql-dialect = # 使用SQL 方言。自动检测默认。

# JDBC (JdbcProperties)
spring.jdbc.template.fetch-size = -1 # 当需要更多行时，应从数据库中获取的行数。
spring.jdbc.template.max-rows = -1 # 最大行数。
spring.jdbc.template.query-timeout = # 查询超时。默认是使用JDBC 驱动程序的默认配置。如果未指定持续时间后缀，则将使用秒。

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.data.jpa.repositories.enabled = true # 是否启用JPA 存储库。
spring.jpa.database = # 目标数据库进行操作，默认为自动检测。可以使用“databasePlatform”属性进行替代设置。
spring.jpa.database-platform = # 要运行的目标数据库的名称，默认为自动检测。也可以使用“数据库”枚举进行设置。
spring.jpa.generate-ddl = false # 是否在启动时初始化模式。
spring.jpa.hibernate.ddl-auto = # DDL 模式。这实际上是“hibernate.hbm2ddl.auto”属性的快捷方式。当使用嵌入式数据库并且没有检测到模式时，将使用此值。
spring.jpa.hibernate.naming.implicit-strategy = # 隐式命名策略的完全限定名称。
spring.jpa.hibernate.naming.physical-strategy = # 物理命名策略的完全限定名称。
spring.jpa.hibernate.use-new-id-generator-mappings = # 是否将Hibernate 的新的IdentifierGenerator 用于AUTO, TABLE 和SEQUENCE。
spring.jpa.mapping-resources = # 映射资源（相当于persistence.xml 中的“映射文件”条目）。
弹簧。= true # 注册OpenEntityManagerInViewInterceptor。将JPA EntityManager 绑定到线程，以完成请求的整个处理。
spring.jpa.properties.* = # 在JPA 提供程序上设置的其他本机属性。
spring.jpa.show-sql = false # 是否启用SQL 语句的日志记录。

# JTA (JtaAutoConfiguration)
spring.jta.enabled = true # 是否启用JTA 支持。
spring.jta.log-dir = # 事务日志目录。
spring.jta.transaction-manager-id = # 事务管理器唯一标识符。

# ATOMIKOS (AtomikosProperties)
spring.jta.atomikos.connectionfactory.borrow-connection-timeout = 30 # 从池中借用连接超时，以秒为单位。
spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag = true # 创建会话时是否忽略事务处理标志。
spring.jta.atomikos.connectionfactory.local-transaction-mode = false # 是否需要本地事务。
spring.jta.atomikos.connectionfactory.maintenance-interval = 60 # 池的维护线程运行之间的时间，以秒为单位。
spring.jta.atomikos.connectionfactory.max-idle-time = 60 # 从池中清除连接之后的时间，以秒为单位。
spring.jta.atomikos.connectionfactory.max-lifetime = 0 # 以秒为单位的连接可以在被销毁前汇集的时间。0 表示没有限制。
spring.jta.atomikos.connectionfactory.max-pool-size = 1 # 池的最大尺寸。

```

```

spring.jta.atomikos.connectionfactory.min-pool-size = 1 #池的最小大小。
spring.jta.atomikos.connectionfactory.reap-timeout = 0 #借用连接的收获超时（以秒为单位）。0表示没有限制。
spring.jta.atomikos.connectionfactory.unique-resource-name = jmsConnectionFactory #恢复期间用于识别资源的唯一名称。
spring.jta.atomikos.connectionfactory.xa-connection-factory-class-name = #供应商特定的XAConnectionFactory实现。
spring.jta.atomikos.connectionfactory.xa-properties = #供应商特定的XA属性。
spring.jta.atomikos.datasource.borrow-connection-timeout = 30 #用于从池中借用连接的超时，以秒为单位。
spring.jta.atomikos.datasource.concurrent-connection-validation = #是否使用并发连接验证。
spring.jta.atomikos.datasource.default-isolation-level = #池提供的连接的默认隔离级别。
spring.jta.atomikos.datasource.login-timeout = #建立数据库连接
的超时时间，以秒为单位。spring.jta.atomikos.datasource.maintenance-interval = 60 #池维护线程运行之间的时间（以秒为单位）。
spring.jta.atomikos.datasource.max-idle-time = 60 #从池中清除连接之后的时间，以秒为单位。
spring.jta.atomikos.datasource.max-lifetime = 0 #以秒为单位的连接可以在被销毁前汇集的时间。0表示没有限制。
spring.jta.atomikos.datasource.max-pool-size = 1 #池的最大尺寸。
spring.jta.atomikos.datasource.min-pool-size = 1 #池的最小尺寸。
spring.jta.atomikos.datasource.reap-timeout = 0 #借用连接的收获超时（以秒为单位）。0表示没有限制。
spring.jta.atomikos.datasource.test-query = #返回之前用于验证连接的SQL查询或语句。
spring.jta.atomikos.datasource.unique-resource-name = dataSource #在恢复期间用于标识资源的唯一名称。
spring.jta.atomikos.datasource.xa-data-source-class-name = #供应商特定的XAConnectionFactory实现。
spring.jta.atomikos.datasource.xa-properties = #供应商特定的XA属性。
spring.jta.atomikos.properties.allow-sub-transactions = true #指定是否允许子交易。
spring.jta.atomikos.properties.checkpoint-interval = 500 #检查点之间的时间间隔，表示为两个检查点之间的日志写入次数。
spring.jta.atomikos.properties.default-jta-timeout = 10000ms #JTA事务的默认超时。
spring.jta.atomikos.properties.default-max-wait-time-on-shutdown = 9223372036854775807 #正常关机（无强制）等待事务完成多长时间。
spring.jta.atomikos.properties.enable-logging = true #是否启用磁盘日志记录。
spring.jta.atomikos.properties.force-shutdown-on-vm-exit = false #VM关闭是否应触发事务核心的强制关闭。
spring.jta.atomikos.properties.log-base-dir = #应该存储日志文件的目录。
spring.jta.atomikos.properties.log -base -name = tmlog #事务日志文件的基本名称。
spring.jta.atomikos.properties.max-actives = 50 #活动事务的最大数量。
spring.jta.atomikos.properties.max-timeout = 300000ms #交易允许的最大超时时间。
spring.jta.atomikos.properties.recovery.delay = 10000ms #两次恢复扫描之间的延迟。
spring.jta.atomikos.properties.recovery.forget-orphaned-log-entries-delay = 86400000ms #延迟后恢复可以清除挂起（'孤立'）日志条目。
spring.jta.atomikos.properties.recovery.max-retries = 5 #抛出异常之前尝试提交事务的重试次数。
spring.jta.atomikos.properties.recovery.retry-interval = 10000ms #重试尝试之间的延迟。
spring.jta.atomikos.properties.serial-jta-transactions = true #是否应该在可能的情况下连接子事务。
spring.jta.atomikos.properties.service = #应该启动的事务管理器实现。
spring.jta.atomikos.properties.threaded-two-phase-commit = false #是否在参与资源上使用不同（并发）的线程进行两阶段提交。
spring.jta.atomikos.properties.transaction-manager-unique-name = #事务管理器的唯一名称。

```

```

#BITRONIX
spring.jta.bitronix.connectionfactory.acquire-increment = 1 #增长池时创建的连接数。
spring.jta.bitronix.connectionfactory.acquisition-interval = 1 #在获取无效连接后尝试重新获取连接之前，需要等待的时间（以秒为单位）。
spring.jta.bitronix.connectionfactory.acquisition-timeout = 30 #以秒为单位的超时时间，用于从池中获取连接。
spring.jta.bitronix.connectionfactory.allow-local-transactions = true #事务管理器是否应允许混合XA和非XA事务。
spring.jta.bitronix.connectionfactory.apply-transaction-timeout = false #在注册时是否应该在XAResource上设置事务超时。
spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled = true #资源是否应该自动注册和退出。
spring.jta.bitronix.connectionfactory.cache-producer-consumers = true #生产者和消费者是否应该被缓存。
spring.jta.bitronix.connectionfactory.class-name = #XA资源的基础实现类名称。
spring.jta.bitronix.connectionfactory.defer-connection-release = true #提供者是否可以在同一连接上运行多个事务并支持事务交叉。
spring.jta.bitronix.connectionfactory.disabled = #该资源是否被禁用，意味着暂时禁止从其池中获取连接。
spring.jta.bitronix.connectionfactory.driver-properties = #应该在底层实现上设置的属性。
spring.jta.bitronix.connectionfactory.failed = #标记此资源生产者失败。
spring.jta.bitronix.connectionfactory.ignore-recovery-failures = false #是否应该忽略恢复失败。
spring.jta.bitronix.connectionfactory.max-idle-time = 60 #连接从池中清理之后的时间，以秒为单位。
spring.jta.bitronix.connectionfactory.max-pool-size = 10 #池的最大尺寸。0表示没有限制。
spring.jta.bitronix.connectionfactory.min-pool-size = 0 #池的最小大小。
spring.jta.bitronix.connectionfactory.password = #用于连接到JMS提供程序的密码。
spring.jta.bitronix.connectionfactory.share-transaction-connections = false #ACCESSIBLE状态下的连接是否可以在事务上下文中共享。
spring.jta.bitronix.connectionfactory.test-connections = true #连接是否需要从池中获取时进行测试。
spring.jta.bitronix.connectionfactory.two-pc-ordering-position = 1 #这个资源在两阶段提交期间应该采用的位置（总是首先是Integer类型）。
spring.jta.bitronix.connectionfactory.unique-name = jmsConnectionFactory #在恢复期间用于标识资源的唯一名称。
spring.jta.bitronix.connectionfactory.use-tm-join = true #启动XAResources时是否应使用TMJOIN。
spring.jta.bitronix.connectionfactory.user = #用于连接到JMS提供者的用户。
spring.jta.bitronix.datasource.acquire-increment = 1 #增长池时创建的连接数。
spring.jta.bitronix.datasource.acquisition-interval = 1 #在获取无效连接后，尝试再次获取连接之前，需要等待几秒钟。
spring.jta.bitronix.datasource.acquisition-timeout = 30 #以秒为单位超时获取池中的连接。
spring.jta.bitronix.datasource.allow-local-transactions = true #事务管理器是否应允许混合XA和非XA事务。
spring.jta.bitronix.datasource.apply-transaction-timeout = false #在注册时是否应该在XAResource上设置事务超时。
spring.jta.bitronix.datasource.automatic-enlisting-enabled = true #资源是否应该自动登录和退出。
spring.jta.bitronix.datasource.class-name = #XA资源的基础实现类名称。
spring.jta.bitronix.datasource.cursor-holdability = #连接的默认光标可保存性。
spring.jta.bitronix.datasource.defer-connection-release = true #数据库是否可以在同一连接上运行多个事务并支持事务交叉。
spring.jta.bitronix.datasource.disabled = #该资源是否被禁用，意味着暂时禁止从其池中获取连接。
spring.jta.bitronix.datasource.driver-properties = #应该在底层实现上设置的属性。
spring.jta.bitronix.datasource.enable-jdbc4-connection-test = #是否在从池中获取连接时调用Connection.isValid()。

```

```

spring.jta.bitronix.datasource.failed = #将此资源生产者标记为失败。
spring.jta.bitronix.datasource.ignore-recovery-failures = false #是否应该忽略恢复失败。
spring.jta.bitronix.datasource.isolation-level = #连接的默认隔离级别。
spring.jta.bitronix.datasource.local-auto-commit = #本地事务的默认自动提交模式。
spring.jta.bitronix.datasource.login-timeout = #建立数据库连接的超时时间，以秒为单位。
spring.jta.bitronix.datasource.max-idle-time = 60 #从池中清除连接之后的时间，以秒为单位。
spring.jta.bitronix.datasource.max-pool-size = 10 #池的最大尺寸。0表示没有限制。
spring.jta.bitronix.datasource.min-pool-size = 0 #池的最小尺寸。
spring.jta.bitronix.datasource.prepared-statement-cache-size = 0 #准备好的语句缓存的目标大小。0禁用缓存。
spring.jta.bitronix.datasource.share-transaction-connections = false #在ACCESSIBLE状态下的连接是否可以在事务上下文中共享。
spring.jta.bitronix.datasource.test-query = #返回之前用于验证连接的SQL查询或语句。
spring.jta.bitronix.datasource.two-pc-ordering-position = 1 #这个资源在两阶段提交期间应该采取的位置（总是首先是Integer.MIN_VALUE）。
spring.jta.bitronix.datasource.unique-name = dataSource #在恢复期间用于标识资源的唯一名称。
spring.jta.bitronix.datasource.use-tm-join = true #启动XAResources时是否应使用TMJOIN。
spring.jta.bitronix.properties.allow-multiple-lrc = false #是否允许多个LRC资源被列入同一事务。
spring.jta.bitronix.properties.asynchronous2-pc = false #是否启用两阶段
落实的异步执行。spring.jta.bitronix.properties.background-recovery-interval-seconds = 60 #在后台运行恢复进程的间隔秒数。
spring.jta.bitronix.properties.current-node-only-recovery = true #是否仅恢复当前节点。
spring.jta.bitronix.properties.debug-zero-resource-transaction = false #是否记录创建并提交未执行单个登记资源的事务的调用堆栈。
spring.jta.bitronix.properties.default-transaction-timeout = 60 #默认事务超时，以秒为单位。
spring.jta.bitronix.properties.disable-jmx = false #是否启用JMX支持。
spring.jta.bitronix.properties.exception-analyzer = #设置要使用的异常分析器实现的完全限定名称。
spring.jta.bitronix.properties.filter-log-status = false #是否启用对日志的过滤，以便仅写入强制日志。
spring.jta.bitronix.properties.force-batching-enabled = true #磁盘力量是否成批。
spring.jta.bitronix.properties.forced-write-enabled = true #日志是否被强制为磁盘。
spring.jta.bitronix.properties.graceful-shutdown-interval = 60 #TM在等待事务在关闭时中止之前完成的最大时间量。
spring.jta.bitronix.properties.jndi-transaction-synchronization-registry-name = #TransactionSynchronizationRegistry的JNDI名称。
spring.jta.bitronix.properties.jndi-user-transaction-name = #UserTransaction的JNDI名称。
spring.jta.bitronix.properties.journal = disk #日志的名称。可以是'磁盘'，'空'或类名。
spring.jta.bitronix.properties.log -part1 -filename = btm1.tlog #日志的第一个片段的名称。
spring.jta.bitronix.properties.log-part2-filename = btm2.tlog #日志的第二个片段的名称。
spring.jta.bitronix.properties.max-log-size-in-mb = 2 #日志片段的最大大小（以兆字节为单位）。
spring.jta.bitronix.properties.resource-configuration-filename = #ResourceLoader配置文件名。
spring.jta.bitronix.properties.server-id = #必须唯一标识此TM实例的ASCII ID。默认为机器的IP地址。
spring.jta.bitronix.properties.skip-corrupted-logs = false #跳过损坏的事务日志条目。
spring.jta.bitronix.properties.warn-about-zero-resource-transaction = true #是否为未执行单个登记资源而执行的事务记录警告。

#NARAYANA (NarayanaProperties)
spring.jta.narayana.default-timeout = 60s #交易超时。如果未指定持续时间后缀，则将使用秒。
spring.jta.narayana.expiry-scanners = com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner #过期扫描器。
spring.jta.narayana.log-dir = #交易对象存储目录。
spring.jta.narayana.one-phase-commit = true #是否启用一个阶段提交优化。
spring.jta.narayana.periodic-recovery-period = 120s #执行周期性恢复扫描的时间间隔。如果未指定持续时间后缀，则将使用秒。
spring.jta.narayana.recovery-backoff-period = 10s #恢复扫描的第一阶段和第二阶段之间
的退避阶段。如果未指定持续时间后缀，则将使用秒。spring.jta.narayana.recovery-db-pass = #恢复管理器要使用的数据库密码。
spring.jta.narayana.recovery-db-user = #恢复管理器使用的数据库用户名。
spring.jta.narayana.recovery-jms-pass = #恢复管理器要使用的JMS密码。
spring.jta.narayana.recovery-jms-user = #恢复管理器使用的JMS用户名。
spring.jta.narayana.recovery-modules = #以逗号分隔的恢复模块列表。
spring.jta.narayana.transaction-manager-id = 1 #唯一的事务管理器ID。
spring.jta.narayana.xa-resource-orphan-filters = #孤立过滤器的逗号分隔列表。

#EMBEDDED MONGODB (EmbeddedMongoProperties)
spring.mongodb.embedded.features = sync_delay #要启用的功能的逗号分隔列表。
spring.mongodb.embedded.storage.database-dir = #用于数据存储的目录。
spring.mongodb.embedded.storage.oplog-size = #oplog的最大大小，以兆字节为单位。
spring.mongodb.embedded.storage.repl-set-name = #副本集的名称。
spring.mongodb.embedded.version = 3.2.2 #使用Mongo版本。

#REDIS (RedisProperties)
spring.redis.cluster.max -redirects = #在群集中执行命令时遵循的最大重定向数。
spring.redis.cluster.nodes = #以逗号分隔的“主机：端口”对列表进行引导。
spring.redis.database = 0 #连接工厂使用的数据库索引。
spring.redis.url = #连接网址。覆盖主机，端口和密码。用户被忽略。示例：redis://user:password@example.com:6379
spring.redis.host = localhost #Redis服务器主机。
spring.redis.jedis.pool.max-active = 8 #在给定时间池可以分配的最大连接数。使用负值无限制。
spring.redis.jedis.pool.max-idle = 8 #池中“空闲”连接的最大数量。使用负值表示无限数量的空闲连接。
spring.redis.jedis.pool.max -wait = -1ms #当池被耗尽时抛出异常之前连接分配应该阻塞的最大时间量。使用负值可以无限期地阻止。
spring.redis.jedis.pool.min-idle = 0 #目标为保持在池中的最小空闲连接数。如果该设置是肯定的，则该设置仅起作用。
spring.redis.lettuce.pool.max-active = 8 #在给定时间池可以分配的最大连接数。使用负值无限制。
spring.redis.lettuce.pool.max-idle = 8 #池中“空闲”连接的最大数量。使用负值表示无限数量的空闲连接。
spring.redis.lettuce.pool.max -wait = -1ms #连接分配在池耗尽时抛出异常之前应阻塞的最长时间量。使用负值可以无限期地阻止。
spring.redis.lettuce.pool.min-idle = 0 #目标为要在池中维护的最小空闲连接数。如果该设置是肯定的，则该设置仅起作用。
spring.redis.lettuce.shutdown-timeout = 100ms #关机超时。

```

```

spring.redis.password = #登录redis服务器的密码。
spring.redis.port = 6379 #Redis服务器端口。
spring.redis.sentinel.master = #Redis服务器的名称。
spring.redis.sentinel.nodes = #“主机: 端口”对的逗号分隔列表。
spring.redis.ssl = false #是否启用SSL支持。
spring.redis.timeout = #连接超时。

# TRANSACTION (TransactionProperties)
spring.transaction.default-timeout = #默认事务超时。如果未指定持续时间后缀，则将使用秒。
spring.transaction.rollback-on-commit-failure = #是否回滚提交失败。

#-----
#INTEGRATION PROPERTIES
#-----

#ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url = #ActiveMQ代理的URL。自动生成默认。
spring.activemq.close-timeout = 15s #在考虑完成之前等待的时间。
spring.activemq.in-memory = true #默认代理URL是否应该在内存中。如果指定了明确的代理，则忽略。
spring.activemq.non-blocking-redelivery = false #是否在重新传递回退事务中的消息之前停止消息传递。这意味着启用此功能时不会保留消息。
spring.activemq.password = #登录经纪人的密码。
spring.activemq.send-timeout = 0ms #等待响应消息发送的时间。将其设置为0以永久等待。
spring.activemq.user = #代理的登录用户。
spring.activemq.packages.trust-all = #是否信任所有包。
spring.activemq.packages.trusted = #以逗号分隔的特定软件包列表（不信任所有软件包）。
spring.activemq.pool.block-if-full = true #是否阻止请求连接并且池已满。将其设置为false以代替引发“JMSEException”。
spring.activemq.pool.block-if-full-timeout = -1ms #如果池仍然已满，则在抛出异常之前阻塞期。
spring.activemq.pool.create-connection-on-startup = true #是否在启动时创建连接。可用于在启动时预热池。
spring.activemq.pool.enabled = false #是否应该创建PooledConnectionFactory，而不是常规的ConnectionFactory。
spring.activemq.pool.expiry-timeout = 0ms #连接到期超时。
spring.activemq.pool.idle-timeout = 30s #连接空闲超时。
spring.activemq.pool.max-connections = 1 #
共享连接的最大数量。spring.activemq.pool.maximum-active-session-per-connection = 500 #每个连接的最大活动会话数。
spring.activemq.pool.reconnect-on-exception = true #发生“JMSEException”时重置连接。
spring.activemq.pool.time-between-expiration-check = -1ms #空闲连接驱逐线程运行之间的休眠时间。否定时，不会有空闲连接逐出线程运行。
spring.activemq.pool.use-anonymous-producers = true #是否只使用一个匿名的“MessageProducer”实例。将其设置为false以便在每次需要时都生成一个新实例。
#ARTEMIS (ArtemisProperties)
spring.artemis.embedded.cluster-password = #集群密码。默认情况下在启动时随机生成。
spring.artemis.embedded.data-directory = #日记文件目录。如果关闭持久性，则不需要。
spring.artemis.embedded.enabled = true #是否在Artemis服务器API可用时启用嵌入模式。
spring.artemis.embedded.persistent = false #是否启用持久存储。
spring.artemis.embedded.queues = #在启动时创建的逗号分隔列表。
spring.artemis.embedded.server-id = #服务器ID。默认情况下，使用自动递增的计数器。
spring.artemis.embedded.topics = #启动时要创建的主题的逗号分隔列表。
spring.artemis.host = localhost #阿蒂米斯经纪人主机。
spring.artemis.mode = #Artemis部署模式，默认为自动检测。
spring.artemis.password = #代理的登录密码。
spring.artemis.port = 61616 #阿蒂米斯经纪人港口。
spring.artemis.user = #代理的登录用户。

#SPRING BATCH (BatchProperties)
spring.batch.initialize-schema = embedded #数据库模式初始化模式。
spring.batch.job.enabled = true #启动时执行上下文中的所有Spring批处理作业。
spring.batch.job.names = #逗号分隔的启动时要执行的作业名称列表（例如`job1, job2`）。默认情况下，执行在上下文中找到的所有作业。
spring.batch.schema = classpath: org / springframework / batch / core / schema- @ @ platform @@ .sql #用于初始化数据库模式的SQL脚本。
spring.batch.table-prefix = #所有批量元数据表的表格前缀。

#SPRING INTEGRATION (IntegrationProperties)
spring.integration.jdbc.initialize-schema = embedded #数据库模式初始化模式。
spring.integration.jdbc.schema = classpath中: 组织/ springframework的/集成/ JDBC / schema- @ @ 平台@ @ .SQL #的路径SQL文件，

# JMS (JmsProperties)
spring.jms.jndi-name = #连接工厂的JNDI名称。设置时，优先于其他连接工厂自动配置。
spring.jms.listener.acknowledge-mode = #容器的确认模式。默认情况下，侦听器通过自动确认进行事务处理。
spring.jms.listener.auto-startup = true #启动时自动启动容器。
spring.jms.listener.concurrency = #最小并发消费者数量。
spring.jms.listener.max-concurrency = #最大并发消费者数量。
spring.jms.pub-sub-domain = false #默认目标类型是否为主题。
spring.jms.template.default-destination =spring.jms.template.qos-enabled = #发送消息时是否启用显式QoS（服务质量）。spring.jms.template.delivery-delay = #发送延迟用于发送呼叫。
spring.jms.template.delivery-mode = #传送模式。设置时启用QoS（服务质量）。

```

```

spring.jms.template.priority = #发送时的消息优先级。设置时启用QoS（服务质量）。
#超时使用接收电话。spring.jms.template.time生存

= #发送消息时的生存时间。设置时启用QoS（服务质量）。

#APACHE KAFKA (KafkaProperties)
spring.kafka.admin.client-id = #发送请求时传递给服务器的ID。用于服务器端日志记录。
spring.kafka.admin.fail-fast = false #如果代理在启动时不可用，是否快速失败。
spring.kafka.admin.properties.* = #用于配置客户端的其他特定于管理员的属性。
spring.kafka.admin.ssl.key-password = #密钥存储文件中的私钥密码。
spring.kafka.admin.ssl.keystore-location = #密钥存储文件的位置。
spring.kafka.admin.ssl.keystore-password = #存储密钥存储文件的密码。
spring.kafka.admin.ssl.truststore-location = #信任存储文件的位置。
spring.kafka.admin.ssl.truststore-password = #存储信任存储文件的密码。
spring.kafka.bootstrap-servers = #主机：端口对的逗号分隔列表，用于建立到Kafka集群的初始连接。
spring.kafka.client-id = #发送请求时传递给服务器的ID。用于服务器端日志记录。
spring.kafka.consumer.auto-commit-interval = #如果'enable.auto.commit'设置为true，则消费者偏移自动提交给Kafka的频率。
spring.kafka.consumer.auto-offset-reset = #在Kafka中没有初始偏移量或当前偏移量不再存在于服务器上时该做什么。
spring.kafka.consumer.bootstrap-servers = #主机：端口对的逗号分隔列表，用于建立与Kafka集群的初始连接。
spring.kafka.consumer.client-id = #在发出请求时传递给服务器的ID。用于服务器端日志记录。
spring.kafka.consumer.enable-auto-commit = #用户的偏移是否在后台定期提交。
spring.kafka.consumer.fetch-max-wait = #如果没有足够的数据立即满足“fetch.min.bytes”给出的要求，服务器在应答提取请求之前阻塞的最大时间。
spring.kafka.consumer.fetch-min-size = #服务器为获取请求返回的最小数据量（以字节为单位）。
spring.kafka.consumer.group-id = #标识此用户所属的用户组的唯一字符串。
spring.kafka.consumer.heartbeat-interval = #心跳到消费者协调员之间的预期时间。
spring.kafka.consumer.key-deserializer = #键的反序列化程序类。spring.kafka.consumer.max-poll-records = #在一次调用poll()中返回的最大记录数。
spring.kafka.consumer.properties.* = #用于配置客户端的其他消费者特定属性。
spring.kafka.consumer.ssl.key-password = #密钥存储文件中的私钥密码。
spring.kafka.consumer.ssl.keystore-location = #密钥存储文件的位置。
spring.kafka.consumer.ssl.keystore-password = #存储密钥存储文件的密码。
spring.kafka.consumer.ssl.truststore-location = #信任存储文件的位置。
spring.kafka.consumer.ssl.truststore-password = #存储信任存储文件的密码。
spring.kafka.consumer.value-deserializer = #
解码器类的值。spring.kafka.jaas.control-flag = required #登录配置的控制标志。
spring.kafka.jaas.enabled = false #是否启用JAAS配置。
spring.kafka.jaas.login-module = com.sun.security.auth.module.Krb5LoginModule #登录模块。
spring.kafka.jaas.options = #其他JAAS选项。
spring.kafka.listener.ack-count = #当ackMode为“COUNT”或“COUNT_TIME”时，偏移量之间的记录数。
spring.kafka.listener.ack-mode = #Listener AckMode。请参阅spring-kafka文档。
spring.kafka.listener.ack-time = #当ackMode为“TIME”或“COUNT_TIME”时，偏移提交之间的时间。
spring.kafka.listener.client-id = #监听器的消费者client.id属性的前缀。
spring.kafka.listener.concurrency = #在侦听器容器中运行的线程数。
spring.kafka.listener.idle-event-interval = #发布空闲消费者事件（未收到数据）之间的时间。
spring.kafka.listener.log-container-config = #是否在初始化期间记录容器配置（INFO级别）。
spring.kafka.listener.monitor-interval = #检查无响应客户的时间。如果未指定持续时间后缀，则将使用秒。
spring.kafka.listener.no-poll-threshold = #乘数应用于“pollTimeout”以确定消费者是否无响应。
spring.kafka.listener.poll-timeout = #轮询消费者时使用的超时。
spring.kafka.listener.type = single #监听器类型。
spring.kafka.producer.acks = #生产者在考虑请求完成之前要求领导者收到的确认数量。
spring.kafka.producer.batch-size = #发送前批量记录的数量。
spring.kafka.producer.bootstrap的服务器 = #主机：端口对的逗号分隔列表，用于建立到Kafka集群的初始连接。
spring.kafka.producer.buffer-memory = #生产者可用于缓冲等待发送到服务器的记录的总内存字节数。
spring.kafka.producer.client-id = #在发出请求时传递给服务器的ID。用于服务器端日志记录。
spring.kafka.producer.compression-type = #生产者生成的所有数据的压缩类型。
spring.kafka.producer.key-serializer = #键的序列化类。
spring.kafka.producer.properties.* = #用于配置客户端的其他特定于生产者的属性。
spring.kafka.producer.retries = #当大于零时，允许重试失败的发送。
spring.kafka.producer.ssl.key-password = #密钥存储文件中的私钥密码。
spring.kafka.producer.ssl.keystore-location = #密钥存储文件的位置。
spring.kafka.producer.ssl.keystore-password = #存储密钥存储文件的密码。
spring.kafka.producer.ssl.truststore-location = #信任存储文件的位置。
spring.kafka.producer.ssl.truststore-password = #存储信任存储文件的密码。
spring.kafka.producer.transaction-id-prefix = #非空时，为生产者启用事务支持。
spring.kafka.producer.value-serializer = #值的串行器类。
spring.kafka.properties.* = #用于配置客户端的其他属性，通常用于生产者和使用者。
spring.kafka.ssl.key-password = #密钥存储文件中的私钥密码。
spring.kafka.ssl.keystore-location = #密钥存储文件的位置。
spring.kafka.ssl.keystore-password = #存储密钥存储文件的密码。
spring.kafka.ssl.truststore-location = #信任存储文件的位置。
spring.kafka.ssl.truststore-password = #存储信任存储文件的密码。
spring.kafka.template.default-topic = #发送消息的默认主题。

#RABBIT (RabbitProperties)

```

```

spring.rabbitmq.addresses = #客户端应该连接的地址的逗号分隔列表。
spring.rabbitmq.cache.channel.checkout-timeout = #如果已达到缓存大小，则等待获取频道的持续时间。
spring.rabbitmq.cache.channel.size = #缓存中要保留的通道数。
spring.rabbitmq.cache.connection.mode = channel #连接工厂缓存模式。
spring.rabbitmq.cache.connection.size = #要缓存的连接数。
spring.rabbitmq.connection-timeout = #连接超时。将其设置为零以永久等待。
spring.rabbitmq.dynamic = true #是否创建一个AmqpAdmin bean。
spring.rabbitmq.host = localhost #RabbitMQ主机。
spring.rabbitmq.listener.direct.acknowledge-mode = #容器的确认模式。
spring.rabbitmq.listener.direct.auto-startup = true #是否在启动时自动启动容器。
spring.rabbitmq.listener.direct.consumers-per-queue = #每个队列的使用者数量。
spring.rabbitmq.listener.direct.default-requeue-rejected = #默认情况下拒绝的交付是否重新排队。
spring.rabbitmq.listener.direct.idle-event-interval = #空闲容器事件应该多久发布一次。
spring.rabbitmq.listener.direct.prefetch = #单个请求中要处理的消息数。它应该大于或等于事务大小（如果使用）。
spring.rabbitmq.listener.direct.retry.enabled = false #是否启用发布重试。
spring.rabbitmq.listener.direct.retry.initial-interval = 1000ms #第一次和第二次尝试传递消息之间的持续时间。
spring.rabbitmq.listener.direct.retry.max-attempts = 3 #传递消息的最大尝试次数。
spring.rabbitmq.listener.direct.retry.max -interval = 10000ms #尝试之间的最大持续时间。
spring.rabbitmq.listener.direct.retry.multiplier = 1 #乘数应用于之前的重试间隔。
spring.rabbitmq.listener.direct.retry.stateless = true #重试是无状态还是有状态。
spring.rabbitmq.listener.simple.acknowledge-mode = #容器的确认模式。
spring.rabbitmq.listener.simple.auto-startup = true #是否在启动时自动启动容器。
spring.rabbitmq.listener.simple.concurrency = #监听器调用者线程的最小数量。
spring.rabbitmq.listener.simple.default-requeue-rejected = #默认情况下是否拒绝交付重新排队。
spring.rabbitmq.listener.simple.idle-event-interval = #应该发布空闲容器事件的频率。
spring.rabbitmq.listener.simple.max-concurrency = #监听器调用者线程的最大数量。
spring.rabbitmq.listener.simple.prefetch = #在单个请求中要处理的消息数。它应该大于或等于事务大小（如果使用）。
spring.rabbitmq.listener.simple.retry.enabled = false #是否启用发布重试。
spring.rabbitmq.listener.simple.retry.initial-interval = 1000ms #第一次和第二次尝试传递消息之间的持续时间。
spring.rabbitmq.listener.simple.retry.max-尝试 = 3 #发送邮件的最大尝试次数。
spring.rabbitmq.listener.simple.retry.max -interval = 10000ms #尝试之间的最大持续时间。
spring.rabbitmq.listener.simple.retry.multiplier = 1 #乘数应用于之前的重试间隔。
spring.rabbitmq.listener.simple.retry.stateless = true #重试是无状态还是有状态。
spring.rabbitmq.listener.simple.transaction-size = #事务中要处理的消息数。也就是说，ack之间的消息数量。为了获得最佳结果，它应该
spring.rabbitmq.listener.type = simple #监听器容器类型。
spring.rabbitmq.password = guest #登录以对经纪人进行身份验证。
spring.rabbitmq.port = 5672 #RabbitMQ端口。
spring.rabbitmq.publisher-confirms = false #是否启用发布者确认。
spring.rabbitmq.publisher-returns = false #是否启用发布商退货。
spring.rabbitmq.requested-heartbeat = #请求的心跳超时；零为零。如果未指定持续时间后缀，则将使用秒。
spring.rabbitmq.ssl.enabled = false #是否启用SSL支持。
spring.rabbitmq.ssl.key-store = #保存SSL证书的密钥存储区的路径。
spring.rabbitmq.ssl.key-store-password = #用于访问密钥存储区的密码。
spring.rabbitmq.ssl.key-store-type = PKCS12 #密钥库类型。
spring.rabbitmq.ssl.trust-store = #持有SSL证书的信任库。
spring.rabbitmq.ssl.trust-store-password = #用于访问信任存储的密码。
spring.rabbitmq.ssl.trust-store-type = JKS #信任商店类型。
spring.rabbitmq.ssl.algorithm = #使用的SSL算法。默认情况下，由Rabbit客户端库配置。
spring.rabbitmq.template.exchange = #用于发送操作的默认交换的名称。
spring.rabbitmq.template.mandatory = #是否启用强制消息。
spring.rabbitmq.template.receive-timeout = #receive () 操作超时。
spring.rabbitmq.template.reply-timeout = #sendAndReceive () 操作超时。
spring.rabbitmq.template.retry.enabled = false #是否启用发布重试。
spring.rabbitmq.template.retry.initial-interval = 1000ms #第一次和第二次尝试传递消息之间的持续时间。
spring.rabbitmq.template.retry.max-attempts = 3 #传递消息的最大尝试次数。
spring.rabbitmq.template.retry.max -interval = 10000ms #尝试之间的最大持续时间。
spring.rabbitmq.template.retry.multiplier = 1 #乘数应用于之前的重试间隔。
spring.rabbitmq.template.routing-key = #用于发送操作的默认路由密钥的值。
spring.rabbitmq.username = guest #登录用户向代理进行身份验证。
spring.rabbitmq.virtual-host = #连接到代理时使用的虚拟主机。

#-----
#ACTUATOR PROPERTIES
#-----

#MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.server.add-application-context-header = false #在每个响应中添加“X-Application-Context”HTTP头。
management.server.address = #管理端点应该绑定的网络地址。需要一个自定义的management.server.port。
management.server.port = #管理端点HTTP端口（默认使用与应用程序相同的端口）。配置不同的端口以使用特定于管理的SSL。
management.server.servlet.context-path = #管理端点上下文路径（例如`/management`）。需要一个自定义的management.server.port。
management.server.ssl.ciphers = #支持的SSL密码。需要一个自定义的management.port。
management.server.ssl.client-auth = #是否需要客户端身份验证（“需要”）或需要（“需要”）。需要信任商店。需要一个自定义的management.
management.server.ssl.enabled = #是否启用SSL支持。需要一个自定义的management.server.port。

```

```

management.server.ssl.enabled-protocols = #启用SSL协议。需要一个自定义的management.server.port。
management.server.ssl.key-alias = #标识密钥库中密钥的别名。需要一个自定义的management.server.port。
management.server.ssl.key-password = #用于访问密钥存储区中密钥的密码。需要一个自定义的management.server.port。
management.server.ssl.key-store = #保存SSL证书的密钥存储区的路径（通常是一个jks文件）。需要一个自定义的management.server.port。
management.server.ssl.key-store-password = #用于访问密钥存储的密码。需要一个自定义的management.server.port。
management.server.ssl.key-store-provider = #密钥存储的提供者。需要一个自定义的management.server.port。
management.server.ssl.key-store-type = #密钥存储的类型。需要一个自定义的management.server.port。
management.server.ssl.protocol = TLS #使用SSL协议。需要一个自定义的management.server.port。
management.server.ssl.trust-store = #持有SSL证书的信任库。需要一个自定义的management.server.port。
management.server.ssl.trust-store-password = #用于访问信任存储的密码。需要一个自定义的management.server.port。
management.server.ssl.trust-store-provider = #信任存储的提供者。需要一个自定义的management.server.port。
management.server.ssl.trust-store-type = #信任存储的类型。需要一个自定义的management.server.port。

#CLOUDFOUNDRY
management.cloudfoundry.enabled = true #是否启用扩展Cloud Foundry执行器端点。
management.cloudfoundry.skip-ssl-validation = false #是否跳过针对Cloud Foundry执行器端点安全调用的SSL验证。

#
#终端常规配置management.endpoints.enabled-by-default = #是否默认启用或禁用所有终端。

# ENDPOINTS JMX 配置 (JmxEndpointProperties)
management.endpoints.jmx.domain = org.springframework.boot #终结点JMX域名。如果设置，则回退到'spring.jmx.default-domain'。
management.endpoints.jmx.exposure.include = * #应包含的端点ID或全部包含的**。
management.endpoints.jmx.exposure.exclude = #应排除的端点ID。
management.endpoints.jmx.static-names = #附加到表示端点的所有MBean的ObjectName的静态属性。
management.endpoints.jmx.unique-names = false #是否确保ObjectNames在发生冲突时被修改。

#终端WEB配置 (WebEndpointProperties)
management.endpoints.web.exposure.include = 健康, 信息 #应包含的终端ID或全部为'*'。
management.endpoints.web.exposure.exclude = #应该排除的端点ID。
management.endpoints.web.base-path = / actuators #Web端点的基本路径。相对于server.servlet.context-path或management.server.s
management.endpoints.web.path-mapping = #端点ID和应该暴露它们的路径之间的映射。

# ENDPOINTS CORS CONFIGURATION (CorsEndpointProperties)
management.endpoints.web.cors.allow-credentials = #是否支持凭证。未设置时，不支持凭证。
management.endpoints.web.cors.allowed-headers = #在请求中允许使用逗号分隔的标题列表。'*'允许所有标题。
management.endpoints.web.cors.allowed-methods = #允许使用逗号分隔的方法列表。'*'允许所有方法。未设置时，默认为GET。
management.endpoints.web.cors.allowed-origins = #逗号分隔的起源列表允许。'*'允许所有的来源。未设置时，CORS支持被禁用。
management.endpoints.web.cors.exposed-headers = #包含在响应中的逗号分隔的标题列表。
management.endpoints.web.cors.max-age = 1800s #客户端可以缓存飞行前请求的响应时间。如果未指定持续时间后缀，则将使用秒。

#审计事件ENDPOINT (AuditEventsEndpoint)
management.endpoint.auditevents.cache.time-to-live = 0ms #响应可以被缓存的最长时间。
management.endpoint.auditevents.enabled = true #是否启用auditevents端点。

#BEANS ENDPOINT (BeansEndpoint)
management.endpoint.beans.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.beans.enabled = true #是否启用bean端点。

#条件REPORT ENDPOINT (ConditionsReportEndpoint)
management.endpoint.conditions.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.conditions.enabled = true #是否启用条件端点。

#配置属性REPORT ENDPOINT (ConfigurationPropertiesReportEndpoint, ConfigurationPropertiesReportEndpointProperties)
management.endpoint.configprops.cache.time 生存 = 0毫秒 #，一个响应可以被缓存的最大时间。
management.endpoint.configprops.enabled = true #是否启用configprops端点。
management.endpoint.configprops.keys到了sanitize = 密码, 秘密密钥, 令牌, *凭证。*, VCAP_SERVICES 应消毒#键。键可以是属性以正则表

#ENVIRONMENT ENDPOINT (EnvironmentEndpoint, EnvironmentEndpointProperties)
management.endpoint.env.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.env.enabled = true #是否启用env端点。
management.endpoint.env.keys-to-sanitize = 密码, 秘密, 密钥, 令牌, *凭证。*, vcap_services #应该清理的密钥。键可以是属性以正则表

#FLYWAY ENDPOINT (FlywayEndpoint)
management.endpoint.flyway.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.flyway.enabled = true #是否启用flyway端点。

#HEALTH ENDPOINT (HealthEndpoint, HealthEndpointProperties)
management.endpoint.health.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.health.enabled = true #是否启用运行状况端点。
management.endpoint.health.roles = #用于确定用户是否有权显示详细信息的角色。当为空时，所有认证的用户都被授权。
management.endpoint.health.show-details = never #何时显示完整健康详情。

#HEAP DUMP ENDPOINT (HeapDumpWebEndpoint)

```

```

management.endpoint.heapdump.cache.time-to-live = 0ms #响应可以被缓存的最长时间。
management.endpoint.heapdump.enabled = true #是否启用heapdump端点。

#HTTP TRACE ENDPOINT (HttpTraceEndpoint)
management.endpoint.httptrace.cache.time-to-live = 0ms #响应可以被缓存的最大时间。
management.endpoint.httptrace.enabled = true #是否启用httptrace端点。

#INFO ENDPOINT (InfoEndpoint)
info = #添加到信息端点的任意属性。
management.endpoint.info.cache.time-to-live = 0ms #响应可以被缓存的最大时间。
management.endpoint.info.enabled = true #是否启用信息端点。

#JOLOKIA ENDPOINT (JolokiaProperties)
management.endpoint.jolokia.config.* = #Jolokia设置。有关更多详细信息，请参阅Jolokia的文档。
management.endpoint.jolokia.enabled = true #是否启用jolokia端点。

#LIQUIBASE ENDPOINT (LiquibaseEndpoint)
management.endpoint.liquibase.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.liquibase.enabled = true #是否启用liquibase端点。

#LOG FILE ENDPOINT (LogFileWebEndpoint, LogFileWebEndpointProperties)
management.endpoint.logfile.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.logfile.enabled = true #是否启用日志文件端点。
management.endpoint.logfile.external-file = #要访问的外部日志文件。如果日志文件是由输出重定向写入的，而不是日志记录系统本身，则可

#LOGGERS ENDPOINT (LoggersEndpoint)
management.endpoint.loggers.cache.time-to-live = 0ms #响应可以被缓存的最大时间。
management.endpoint.loggers.enabled = true #是否启用记录器端点。

#请求MAPPING ENDPOINT (MappingsEndpoint)
management.endpoint.mappings.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.mappings.enabled = true #是否启用映射端点。

#METRICS ENDPOINT (MetricsEndpoint)
management.endpoint.metrics.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.metrics.enabled = true #是否启用度量标准端点。

#PROMETHEUS ENDPOINT (PrometheusScrapeEndpoint)
management.endpoint.prometheus.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.prometheus.enabled = true #是否启用普罗米修斯端点。

#SCHEDULED TASKS ENDPOINT (ScheduledTasksEndpoint)
management.endpoint.scheduledtasks.cache.time-to-live = 0ms #响应可以被缓存的最大时间。
management.endpoint.scheduledtasks.enabled = true #是否启用scheduledtasks端点。

#SESSIONS ENDPOINT (SessionsEndpoint)
management.endpoint.sessions.enabled = true #是否启用会话端点。

#SHUTDOWN ENDPOINT (ShutdownEndpoint)
management.endpoint.shutdown.enabled = false #是否启用关闭端点。

#THREAD DUMP ENDPOINT (ThreadDumpEndpoint)
management.endpoint.threaddump.cache.time-to-live = 0ms #可以缓存响应的最长时间。
management.endpoint.threaddump.enabled = true #是否启用线程转储端点。

# 健康指标
management.health.db.enabled = true #是否启用数据库健康检查。
management.health.cassandra.enabled = true #是否启用Cassandra健康检查。
management.health.couchbase.enabled = true #是否启用Couchbase运行状况检查。
management.health.defaults.enabled = true #是否启用默认运行状况指示器。
management.health.diskspace.enabled = true #是否启用磁盘空间运行状况检查。
management.health.diskspace.path = #用于计算可用磁盘空间的路径。
management.health.diskspace.threshold = 0 #应该可用的最小磁盘空间（以字节为单位）。
management.health.elasticsearch.enabled = true #是否启用Elasticsearch运行状况检查。
management.health.elasticsearch.indices = #逗号分隔的索引名称。
management.health.elasticsearch.response-timeout = 100ms #等待集群响应的时间。
management.health.influxdb.enabled = true #是否启用InfluxDB运行状况检查。
management.health.jms.enabled = true #是否启用JMS运行状况检查。
management.health.ldap.enabled = true #是否启用LDAP运行状况检查。
management.health.mail.enabled = true #是否启用邮件运行状况检查。
management.health.mongo.enabled = true #是否启用MongoDB运行状况检查。
management.health.neo4j.enabled = true #是否启用Neo4j运行状况检查。
management.health.rabbit.enabled = true #是否启用RabbitMQ运行状况检查。
management.health.redis.enabled = true #是否启用Redis运行状况检查。

```

```

management.health.solr.enabled = true #是否启用Solr运行状况检查。
management.health.status.http-mapping = #健康状态到HTTP状态代码的映射。默认情况下，注册的健康状态映射到合理的默认值（例如，UP映射为DOWN，OUT_OF_SERVICE，UP，UNKNOWN）#以严重性顺序的逗号分隔的健康状态列表。

#HTTP TRACING (HttpTraceProperties)
management.trace.http.enabled = true #是否启用HTTP请求 - 响应跟踪。
management.trace.http.include = request-headers, response-headers, cookies, errors #要包含在跟踪中的项目。

#信息贡献者 (InfoContributorProperties)
management.info.build.enabled = true #是否启用编译信息。
management.info.defaults.enabled = true #是否启用默认信息撰稿人。
management.info.env.enabled = true #是否启用环境信息。
management.info.git.enabled = true #是否启用git信息。
management.info.git.mode = simple #用于公开git信息的模式。

#METRICS
management.metrics.binders.files.enabled = true #是否启用文件度量标准。
management.metrics.binders.integration.enabled = true #是否启用Spring Integration指标。
management.metrics.binders.jvm.enabled = true #是否启用JVM度量标准。
management.metrics.binders.logback.enabled = true #是否启用Logback指标。
management.metrics.binders.processor.enabled = true #是否启用处理器指标。
management.metrics.binders.uptime.enabled = true #是否启用正常运行时间指标。
management.metrics.distribution.percentiles-histogram.* = #以指定名称开头的电表ID是否应该公布百分比直方图。
management.metrics.distribution.percentiles.* = #从指定名称开始计算出的计量器ID从后端运送到特定计算的不可汇总百分点。
management.metrics.distribution.sla.* = #以特定名称开始的电表ID的特定SLA边界。最长的匹配胜出，关键'全部'也可用于配置所有仪表。
management.metrics.enable.* = #是否启用以指定名称开头的电表ID。最长的匹配胜出，关键'全部'也可用于配置所有仪表。
management.metrics.export.atlas.batch-size = 10000 #每个请求用于此后端的度量数。如果找到更多的测量结果，则会发出多个请求。
management.metrics.export.atlas.config-refresh-frequency = 10s #刷新LWC服务配置设置的频率。
management.metrics.export.atlas.config-time-to-live = 150s #为LWC服务的订阅生存的时间。
management.metrics.export.atlas.config-uri = http://localhost: 7101 / lwc / api / v1 / expressions / local-dev #Atlas LWC端点评估URL。
management.metrics.export.atlas.connect-timeout = 1s #对后端请求的连接超时。
management.metrics.export.atlas.enabled = true #是否启用指标到后端的导出。
management.metrics.export.atlas.eval-uri = http://localhost: 7101 / lwc / api / v1 / evaluate #URI用于Atlas LWC端点评估。
management.metrics.export.atlas.lwc-enabled = false #是否启用流式传输到Atlas LWC。
management.metrics.export.atlas.meter-time-to-live = 15m #生活在没有任何活动的米的时间。经过这段时间后，电表将被视为过期，并不再发送度量。
management.metrics.export.atlas.num-threads = 2 #用于度量标准发布计划程序的线程数。
management.metrics.export.atlas.read超时 = 10s #读取该后端请求的超时时间。
management.metrics.export.atlas.step = 1m #使用步长（即报告频率）。
management.metrics.export.atlas.uri = http://localhost: 7101 / api / v1 / publish #Atlas服务器的URI。
management.metrics.export.datadog.api-key = #Datadog API密钥。
management.metrics.export.datadog.application-key = #Datadog应用程序密钥。不是严格要求，而是通过向Datadog发送电表描述，类型和度量。
management.metrics.export.datadog.batch-size = 10000 #每个请求用于此后端的度量数。如果找到更多的测量结果，则会发出多个请求。
management.metrics.export.datadog.connect-timeout = 1s #对后端请求的连接超时。
management.metrics.export.datadog.descriptions = true #是否将描述元数据发布到Datadog。关闭此功能可以最大限度地减少发送的元数据。
management.metrics.export.datadog.enabled = true #是否启用指标到后端的导出。
management.metrics.export.datadog.host-tag = instance #将标准传送到Datadog时将被映射到“主机”的标签。
management.metrics.export.datadog.num线程 = 2 #用于度量标准发布计划程序的线程数。
management.metrics.export.datadog.read-timeout = 10s #读取该后端请求的超时时间。
management.metrics.export.datadog.step = 1m #使用步长（即报告频率）。
management.metrics.export.datadog.uri = https://app.datadoghq.com #将指标发送到的URI。如果您需要将指标发布到通往Datadog的内部。
management.metrics.export.ganglia.addressing-mode = multicast #UDP寻址模式，可以是单播或多播。
management.metrics.export.ganglia.duration-units = 毫秒 #用于报告持续时间的基本时间单位。
management.metrics.export.ganglia.enabled = true #是否启用向Ganglia导出度量标准。
management.metrics.export.ganglia.host = localhost #Ganglia服务器的主机接收导出的度量标准。
management.metrics.export.ganglia.port = 8649 #Ganglia服务器的端口，用于接收导出的度量标准。
management.metrics.export.ganglia.protocol-version = 3.1 #Ganglia协议版本。必须是3.1或3.0。
management.metrics.export.ganglia.rate-units = seconds #用于报告费率的基本时间单位。
management.metrics.export.ganglia.step = 1m #步长（即报告频率）使用。
management.metrics.export.ganglia.time-to-live = 1 #生活在Ganglia指标上的时间。将多播时间生存时间设置为比主机之间的跳数（路由器）。
management.metrics.export.graphite.duration-units = milliseconds #用于报告持续时间的基本时间单位。
management.metrics.export.graphite.enabled = true #是否启用指标到Graphite的导出。
management.metrics.export.graphite.host = localhost #接收导出指标的Graphite服务器的主机。
management.metrics.export.graphite.port = 2004 #接收导出指标的Graphite服务器的端口。
management.metrics.export.graphite.protocol = pickled #将数据传输到Graphite时使用的协议。
management.metrics.export.graphite.rate-units = seconds #用于报告费率的基本时间单位。
management.metrics.export.graphite.step = 1m #使用步长（即报告频率）。
management.metrics.export.graphite.tags-as-prefix = #对于默认的命名约定，将指定的标记键转换为度量标准前缀的一部分。
management.metrics.export.influx.auto-create-db = true #是否在尝试向其发布指标之前创建InfluxDB数据库（如果它不存在）。
management.metrics.export.influx.batch-size = 10000 #每个请求用于此后端的度量数。如果找到更多的测量结果，则会发出多个请求。
management.metrics.export.influx.compressed = true #是否启用发布到InfluxDB的指标批次的GZIP压缩。
management.metrics.export.influx.connect-timeout = 1s #对后端请求的连接超时。
management.metrics.export.influx.consistency = 1 #为每个点编写一致性。
management.metrics.export.influx.db = mydb #将标准传送到InfluxDB时将映射到“主机”的标签。
management.metrics.export.influx.enabled = true #是否启用指标到后端的导出。

```

```

management.metrics.export.influx.num-threads = 2 #用于指标发布计划程序的线程数。
management.metrics.export.influx.password = #Influx服务器的登录密码。
management.metrics.export.influx.read-timeout = 10s #读取该后端请求的超时时间。
management.metrics.export.influx.retention-policy = #使用的保留策略（如果未指定DEFAULT保留策略，Influx将写入该保留策略）。
management.metrics.export.influx.step = 1m #使用步长（即报告频率）。
management.metrics.export.influx.uri = http://localhost:8086 #Influx服务器的URI。
management.metrics.export.influx.user-name = #Influx服务器的登录用户。
management.metrics.export.jmx.enabled = true #是否启用指标到JMX的导出。
management.metrics.export.jmx.step = 1m #使用步长（即报告频率）。
management.metrics.export.newrelic.account-id = #新的新Relic账户ID。
management.metrics.export.newrelic.api-key = #新的Relic API密钥。
management.metrics.export.newrelic.batch-size = 10000 #每个请求用于此后端的度量数。如果找到更多的测量结果，则会发出多个请求。
management.metrics.export.newrelic.connect-timeout = 1s #对此后端请求的连接超时。
management.metrics.export.newrelic.enabled = true #是否启用指标到此后的导出。
management.metrics.export.newrelic.num-threads = 2 #用于指标发布调度程序的线程数。
management.metrics.export.newrelic.read-timeout = 10s #读取该后端请求的超时时间。
management.metrics.export.newrelic.step = 1m #使用步长（即报告频率）。
management.metrics.export.newrelic.uri = https://insights-collector.newrelic.com# 将指标发送到的URI。
management.metrics.export.prometheus.descriptions = true #是否启用发布说明，作为Prometheus的有效载荷的一部分。关闭此功能可以最
management.metrics.export.prometheus.enabled = true #是否启用指标到Prometheus的导出。
management.metrics.export.prometheus.step = 1m #使用步长（即报告频率）。
management.metrics.export.signalfx.access-token = #SignalFx访问令牌。
management.metrics.export.signalfx.batch-size = 10000 #每个请求用于此后端的度量数。如果找到更多的测量结果，则会发出多个请求。
management.metrics.export.signalfx.connect-timeout = 1s #对此后端请求的连接超时。
management.metrics.export.signalfx.enabled = true #是否启用度量标准导出到此后的端。
management.metrics.export.signalfx.num-threads = 2 #用于指标发布计划程序的线程数。
management.metrics.export.signalfx.read-timeout = 10s #读取该后端请求的超时时间。
management.metrics.export.signalfx.source = #唯一标识将度量标准发布到SignalFx的应用程序实例。默认为本地主机名称。
management.metrics.export.signalfx.step = 10s #使用步长（即报告频率）。
management.metrics.export.signalfx.uri = https://ingest.signalfx.com# 将指标发送到的URI。
management.metrics.export.simple.enabled = true #是否在没有任何其他导出器的情况下启用度量标准导出到内存后端。
management.metrics.export.simple.mode = 累计 #计数模式。
management.metrics.export.simple.step = 1m #使用步长（即报告频率）。
management.metrics.export.statsd.enabled = true #是否启用指标到StatsD的导出。
management.metrics.export.statsd.flavor = datadog #StatsD要使用的协议。
management.metrics.export.statsd.host = localhost #StatsD服务器的主机接收导出的度量标准。
management.metrics.export.statsd.max-packet-length = 1400 #单个有效负载的总长度应保留在网络的MTU内。
management.metrics.export.statsd.polling-frequency = 10s #调查仪表
的频率。当轮询仪表时，其值将被重新计算，如果值已更改（或者publishUnchangedMeters为true），则会将其发送到StatsD服务器。management.r
management.metrics.export.statsd.publish-unchanged-meters = true #是否向StatsD服务器发送未更改的计量表。
management.metrics.export.statsd.queue-size = 2147483647 #等待发送到StatsD服务器的项目队列的最大大小。
management.metrics.export.wavefront.api-token = #将标准直接发布到Wavefront API主机时使用的API标记。
management.metrics.export.wavefront.batch-size = 10000 #每个请求用于此后端的度量数。如果找到更多的测量结果，则会发出多个请求。
management.metrics.export.wavefront.connect-timeout = 1s #对后端请求的连接超时。
management.metrics.export.wavefront.enabled = true #是否启用度量标准导出到此后的端。
management.metrics.export.wavefront.global-prefix = #在Wavefront用户界面中查看时，全局前缀用于将源自此应用的白色盒子工具的指标与
management.metrics.export.wavefront.num-threads = 2 #用于指标发布计划程序的线程数。
management.metrics.export.wavefront.read-timeout = 10s #读取该后端请求的超时时间。
management.metrics.export.wavefront.source = #作为发布到Wavefront的指标来源的应用实例的唯一标识符。默认为本地主机名称。
management.metrics.export.wavefront.step = 10s #步长（即报告频率）使用。
management.metrics.export.wavefront.uri = https://longboard.wavefront.com# 将指标发送到的URI。
management.metrics.use-global-registry = true #自动配置的MeterRegistry实现是否应绑定到Metrics上的全局静态注册表。
management.metrics.web.client.max-uri-tags = 100 #允许的最大唯一URI标记值数量。在达到标签值的最大数量后，带有附加标签值的度量标准
management.metrics.web.client.requests-metric-name = http.client.requests #发送请求的度量标准名称。
management.metrics.web.server.auto-time-requests = true #Spring MVC或WebFlux处理的请求是否应该自动定时。
management.metrics.web.server.requests-metric-name = http.server.requests #接收请求的度量标准名称。

-----
#DEVTOOLS PROPERTIES
-----

#DEVTOOLS (DevToolsProperties)
spring.devtools.livereload.enabled = true #是否启用Livereload.com兼容服务器。
spring.devtools.livereload.port = 35729 #服务器端口。
spring.devtools.restart.additional-exclude = #应该从触发完全重新启动时排除的其他模式。
spring.devtools.restart.additional-paths = #观察更改的其他路径。
spring.devtools.restart.enabled = true #是否启用自动重启。
spring.devtools.restart.exclude = META-INF /行家/ **, META-INF /资源/ **, 资源/ **, 静态/ **, 公共/ **, 模板/ **, ** / * 的Test
spring.devtools.restart.log-condition-evaluation-delta = true #是否在重新启动时记录条件评估增量。
spring.devtools.restart.poll-interval = 1s #轮询类路径更改之间等待的时间。
spring.devtools.restart.quiet-period = 400ms #触发重新启动之前所需的静默时间，不需要任何类路径更改。
spring.devtools.restart.trigger文件 = #特定文件的名称，如果更改，则会触发重新启动检查。如果未指定，则任何类路径文件更改都会触发重新

```

```
#REMOTE DEVTOOLS (RemoteDevToolsProperties)
spring.devtools.remote.context-path = /。~ spring-boot! ~ #用于处理远程连接的上下文路径。
spring.devtools.remote.proxy.host = #用于连接远程应用程序的代理主机。
spring.devtools.remote.proxy.port = #用于连接远程应用程序的代理端口。
spring.devtools.remote.restart.enabled = true #是否启用远程重启。
spring.devtools.remote.secret = #建立连接所需的共享密钥（启用远程支持所必需的）。
spring.devtools.remote.secret头名= 用于传输共享密钥的 X-AUTH-TOKEN #HTTP标头。

#-----
#测试属性
#-----

spring.test.database.replace = any #要替换的现有DataSource的类型。
spring.test.mockmvc.print =默认 #MVC 打印选项。
```

附录B.配置元数据

Spring Boot jar包含元数据文件，提供所有支持的配置属性的详细信息。这些文件旨在让IDE开发人员在用户正在使用`application.properties`或`application.yml`文件时提供上下文帮助和“代码补全”。

大部分元数据文件是在编译时自动生成的，处理所有注释的项目`@ConfigurationProperties`。但是，可以手动编写部分元数据 用于角落案例或更高级的用例。

B.1元数据格式

配置元数据文件位于jar下面`META-INF/spring-configuration-metadata.json`它们使用简单的JSON格式，其中项目按“组”或“属性”分类，附加值提示按“提示”分类，如以下示例所示：

```
{
  "groups": [
    {
      "name": "server" ,
      "type": "org.springframework.boot.autoconfigure.web.ServerProperties" ,
      "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
      "name": "spring.jpa.hibernate" ,
      "type": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties $ Hibernate" ,
      "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties" ,
      "sourceMethod": "getHibernate ()"
    }
  ...
  ],
  "properties": [
    {
      "name": "server.port" ,
      "type": "java.lang.Integer" ,
      "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
      "name": "server.servlet.path" ,
      "type": "java.lang.String" ,
      "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties" ,
      "defaultValue": "/"
    } ,
    {
      "name": "spring.jpa.hibernate.ddl-auto" ,
      "type": "java.lang.String" ,
      "description": "DDL模式，这实际上是\\\"hibernate.hbm2ddl.auto \\“属性的快捷方式。”" ,
      "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties $ Hibernate"
    }
  ...
  ],
  "提示": [
    {
      "name": "spring.jpa.hibernate.ddl-auto" ,
      "values": [
        {
          "value": "none" ,
          "description": "禁用DDL处理。"
        },
        ...
      ]
    }
  ]
}
```

```

    "value": "validate" ,
    "description": "验证模式，不更改数据库。"
},
{
    "value": "update" ,
    "description": ""

    "create" ,
    "description": "创建模式并销毁先前的数据。"
},
{
    "value": "create-drop" ,
    "description": "在会话结束时创建并销毁模式。"
}
]
}
]
}

```

每个“属性”是用户用给定值指定的配置项目。例如，`server.port`并且`server.servlet.path`可以被指定在`application.properties`，具体如下：

```

server.port = 9090
server.servlet.path = / home

```

“组”是较高级别的项目，它们本身不指定值，而是为属性提供上下文分组。例如，`server.port`和`server.servlet.path`属性是`server`组的一部分。



并不要求每个“财产”都有一个“组”。一些属性可能存在于他们自己的权利中。

最后，“提示”是用于帮助用户配置给定属性的附加信息。例如，当开发人员配置`spring.jpa.hibernate.ddl-auto`属性，工具可以使用提示提供了一些自动完成的帮助`none`，`validate`，`update`，`create`，和`create-drop`值。

B.1.1组属性

包含在`groups`数组中的JSON对象可以包含下表中显示的属性：

名称	类型	目的
<code>name</code>	串	组的全名。该属性是强制性的。
<code>type</code>	串	组的数据类型的类名称。例如，如果该组基于注释了的类 <code>@ConfigurationProperties</code> ，则该属性将包含该类的完全限定名称。如果它基于一个 <code>@Bean</code> 方法，那将是该方法的返回类型。如果该类型未知，则可以省略该属性。
<code>description</code>	串	可向用户显示的组的简短说明。如果没有描述可用，则可以省略。建议描述为简短段落，第一行提供简明摘要。描述中的最后一行应以句点(。)结尾。
<code>sourceType</code>	串	贡献此组的源的类名称。例如，如果该组基于 <code>@Bean</code> 注解的方法 <code>@ConfigurationProperties</code> ，则此属性将包含 <code>@Configuration</code> 包含该方法的类的完全限定名称。如果源类型未知，则可以省略该属性。
<code>sourceMethod</code>	串	贡献该组的方法的全名（包括括号和参数类型）（例如， <code>@ConfigurationProperties</code> 注释 <code>@Bean</code> 方法的名称）。如果源方法未知，则可能会被忽略。

B.1.2属性属性

包含在`properties`数组中的JSON对象可以包含下表中描述的属性：

名称	类型	目的
<code>name</code>	串	财产的全名。名称以小写字母分隔的格式（例如， <code>server.servlet.path</code> ）。该属性是强制性的。

名称	类 型	目的
<code>type</code>	串	属性的数据类型（例如 <code>java.lang.String</code> ）的完整签名，也是完整的泛型类型（如 <code>java.util.Map<java.util.String,acme.MyEnum></code> ）。您可以使用此属性来指导用户可以输入的值的类型。为了保证一致性，通过使用它的包装对象（例如， <code>boolean</code> 变成 <code>java.lang.Boolean</code> ）来指定基元的类型。请注意，这个类可能是一个复杂的类型，它会 <code>String</code> 在值被绑定时从a转换而来。如果类型未知，可能会被忽略。
<code>description</code>	串	可向用户显示的组的简短说明。如果没有可用的说明，可能会被忽略。建议描述为简短段落，第一行提供简明摘要。描述中的最后一行应以句点（.）结尾。
<code>sourceType</code>	串	贡献此属性的源的类名称。例如，如果属性来自注释了的类 <code>@ConfigurationProperties</code> ，则此属性将包含该类的完全限定名称。如果源类型未知，则可能会被忽略。
<code>defaultValue</code>	目 的	如果未指定属性，则使用默认值。如果属性的类型是一个数组，它可以是一个值的数组。如果默认值是未知的，则可以省略。
<code>deprecation</code>	弃 用	指定该属性是否被弃用。如果该字段未被弃用或者该信息未知，则可以省略。下表提供了有关该 <code>deprecation</code> 属性的更多详细信息。

包含在`deprecation`每个`properties`元素的属性中的JSON对象可以包含以下属性：

名称	类 型	目的
<code>level</code>	串	弃用级别可以是 <code>warning</code> （默认）或 <code>error</code> 。当一个属性具有 <code>warning</code> 弃用级别时，它应该仍然在环境中绑定。但是，当它具有 <code>error</code> 弃用级别时，该属性将不再受管理且不受限制。
<code>reason</code>	串	对房产被弃用的原因的简短描述。如果没有理由可用，它可能被省略。建议描述为简短段落，第一行提供简明摘要。描述中的最后一行应以句点（.）结尾。
<code>replacement</code>	串	替换此已弃用属性的属性的全名。如果这个属性没有替换，它可能被省略。



在Spring Boot 1.3之前，`deprecated`可以使用单个布尔属性来代替`deprecation`元素。这仍以不推荐的方式支持，不应再使用。如果没有理由和替换可用，`deprecation`应该设置一个空对象。

弃用也可以在代码中声明性地指定，方法是将`@DeprecatedConfigurationProperty`注释添加到暴露不赞成使用的属性的getter中。例如，假设该`app.acme.target`属性很混乱，并被重命名为`app.acme.name`。以下示例显示如何处理这种情况：

```
@ConfigurationProperties ("app.acme")
公共类 AcmeProperties {

    私人字符串名称;

    public String getName () {...}

    public void setName (String name) {...}

    @DeprecatedConfigurationProperty (replacement ="app.acme.name")
    @Deprecated
    public String getTarget () {
        return getName ();
    }

    @Deprecated
    public void setTarget (String target) {
        的setName (目标);
    }
}
```



没有办法设置`level`。`warning`总是假定，因为代码仍在处理该属性。

前面的代码确保已弃用的属性仍然有效（委托给 `name` 后台的属性）。一旦 `getTarget` 和 `setTarget` 方法可以从公共 API 中删除，元数据中的自动弃用提示也会消失。如果您想保留提示，那么添加具有 `error` 弃用级别的手动元数据可确保用户仍被通知该属性。这样做在 `replacement` 提供时特别有用。

B.1.3 提示属性

包含在 `hints` 数组中的 JSON 对象可以包含下表中显示的属性：

名称	类型	目的
<code>name</code>	串	此提示引用的财产的全名。名称以小写字母分隔的形式（例如 <code>server.servlet.path</code> ）。如果该属性引用一个地图（如 <code>system.contexts</code> ），则该提示将应用于地图的键（ <code>system.context.keys</code> ）或地图的值（ <code>system.context.values</code> ）。该属性是强制性的。
<code>values</code>	<code>ValueHint[]</code>	由 <code>ValueHint</code> 对象定义的有效值列表（在下表中进行了介绍）。每个条目定义该值并可能有说明。
<code>providers</code>	<code>ValueProvider[]</code>	由 <code>ValueProvider</code> 对象定义的提供者列表（在本文档稍后描述）。每个条目定义提供者的名称及其参数（如果有的话）。

包含在 `values` 每个 `hint` 元素的属性中的 JSON 对象可以包含下表中描述的属性：

名称	类型	目的
<code>value</code>	目 的	提示引用的元素的有效值。如果属性的类型是一个数组，它也可以是一个值的数组。该属性是强制性的。
<code>description</code>	串	可以向用户显示的值的简短说明。如果没有可用的说明，可能会被忽略。建议描述为简短段落，第一行提供简明摘要。描述中的最后一行应以句点（.）结尾。

包含在 `providers` 每个 `hint` 元素的属性中的 JSON 对象可以包含下表中描述的属性：

名称	类型	目的
<code>name</code>	串	提供程序的名称，用于为提示所引用的元素提供其他内容帮助。
<code>parameters</code>	JSON 对象	提供者支持的任何其他参数（查看提供者的文档以获取更多详细信息）。

B.1.4 重复的元数据项目

具有相同“属性”和“组”名称的对象可以在元数据文件中多次出现。例如，您可以将两个单独的类绑定到相同的前缀，每个类都有可能重叠的属性名称。尽管多次出现在元数据中的相同名称不应该很常见，但元数据的使用者应该注意确保它们支持它。

B.2 提供手册提示

为了改善用户体验并进一步帮助用户配置给定属性，可以提供以下附加元数据：

- 描述一个属性的潜在值列表。
- 关联提供者，将明确定义的语义附加到属性，以便工具可以根据项目的上下文发现潜在值的列表。

B.2.1 价值提示

`name` 每个提示的属性指的 `name` 是一个属性。在前面所示的初始例子中，我们提供了五个值 `spring.jpa.hibernate.ddl-auto` 属性：`none`，`validate`，`update`，`create`，和 `create-drop`。每个值也可以有一个描述。

如果您的属性是键入的 `Map`，则可以为键和值提供提示（但不能为地图本身提供）。特殊 `.keys` 和 `.values` 后缀分别指的是键和值。

假设 `sample.contexts` 将魔术 `String` 值映射为整数，如以下示例所示：

```
@ConfigurationProperties ("sample")
public class SampleProperties {
    私人地图<字符串, 整数>上下文;
```

```
// getters and setters
}
```

神奇的值是（在这个例子中）是`sample1`和`sample2`。为了为密钥提供额外的内容帮助，您可以将以下JSON添加到模块的手动元数据中：

```
{
  "hints": [
    {
      "name": "sample.contexts.keys",
      "values": [
        {
          "value": "sample1"
        },
        {
          "value": "sample2"
        }
      ]
    }
]
```



我们建议您改用`Enum`这两个值来代替。如果您的IDE支持它，这是迄今为止最有效的自动完成方法。

B.2.2 价值提供者

提供者是将语义附加到属性的有效方式。在本节中，我们定义您可以用于自己提示的官方提供商。但是，您最喜欢的IDE可能会实现其中的一部分或者其中任何一个。此外，它最终可以提供自己的。



由于这是一项新功能，IDE供应商必须赶上它的工作原理。采用时间自然有所不同。

下表总结了支持的提供商列表：

名称	描述
<code>any</code>	允许提供任何额外的价值。
<code>class-reference</code>	自动完成项目中可用的类。通常受 <code>target</code> 参数指定的基类限制。
<code>handle-as</code>	处理属性，就好像它是由强制性 <code>target</code> 参数定义的类型一样。
<code>logger-name</code>	自动完成有效的记录器名称。通常，当前项目中可用的包名和类名可以自动完成。
<code>spring-bean-reference</code>	自动完成当前项目中的可用bean名称。通常受 <code>target</code> 参数指定的基类限制。
<code>spring-profile-name</code>	自动完成项目中可用的Spring配置文件名称。



只有一个提供程序可以对某个给定的属性处于活动状态，但是您可以指定多个提供程序，只要它们能够以某种方式管理该属性。确保首先放置功能最强大的提供者，因为IDE必须使用它可以处理的JSON部分中的第一个提供者。如果没有支持给定财产的提供者，则也不提供特别的内容帮助。

任何

特殊的**任何**提供者值允许提供**任何**附加值。如果支持，则应应用基于属性类型的常规值验证。

如果您有一个值列表，并且任何额外的值仍应被视为有效，通常会使用此提供程序。

下面的例子提供`on`，并`off`为自动完成值`system.state`：

```
{
  "hints": [
    {
      "name": "system.state",
      "values": [
        {
          "value": "on"
        },
        {
          "value": "off"
        }
      ]
    }
]
```

```

        {
            "value": "off"
        }
    ],
    "providers": [
        {
            "名称": "any"
        }
    ]
}
]
}

```

请注意，在前面的例子中，任何其他值也是允许的。

类参考

在类引用提供商自动完成项目中可用的类。该提供程序支持以下参数：

参数	类型	默认值	描述
<code>target</code>	<code>String</code> (<code>Class</code>)	没有	应该分配给所选值的类的全限定名称。通常用于过滤非候选类。请注意，这些信息可以由类型本身通过暴露具有适当上限的类来提供。
<code>concrete</code>	<code>boolean</code>	真正	指定是否只有具体的课程才被视为有效的候选人。

以下元数据片段对应于`server.servlet.jsp.class-name` 定义`JspServlet`要使用的类名称的标准属性：

```

{ "hints": [
    {
        "name": "server.servlet.jsp.class-name",
        "providers": [
            {
                "name": "class-reference",
                "parameters": {
                    "target": "javax.servlet.http.HttpServlet"
                }
            }
        ]
    }
]
}

```

处理为

该手柄的供应商，您可以替代属性的类型到一个更高层次的类型。这通常发生在属性具有`java.lang.String`类型时，因为您不希望配置类依赖可能不在类路径中的类。该提供程序支持以下参数：

参数	类型	默认值	描述
<code>target</code>	<code>String</code> (<code>Class</code>)	没有	要为属性考虑的类型的完全限定名称。该参数是强制性的。

可以使用以下类型：

- 任何`java.lang.Enum`：列出属性的可能值。（我们建议使用`Enum`类型定义属性，因为IDE不需要进一步提示来自动生成值。）
- `java.nio.charset.Charset`：支持自动完成字符集/编码值（如`UTF-8`）
- `java.util.Locale`：自动完成语言环境（如`en_US`）
- `org.springframework.util.MimeType`：支持内容类型值的自动完成（如`text/plain`）
- `org.springframework.core.io.Resource`：支持Spring的资源抽象的自动完成来引用文件系统或类路径上的文件。（如`classpath:/sample.properties`）



如果可以提供多个值，请使用`a Collection`或数组类型向IDE讲授它。

以下元数据片段对应于`spring.liquibase.change-log` 定义要使用的更改日志路径的标准属性。它实际上是作为一个内部使用的，`org.springframework.core.io.Resource`但不能像这样公开，因为我们需要保持原始的String值传递给Liquibase API。

```
{
  "hints": [
    {
      "name": "spring.liquibase.change-log",
      "providers": [
        {
          "name": "handle-as",
          "parameters": {
            "target": "org.springframework.core.io.Resource"
          }
        }
      ]
    }
  ]
}
```

记录器名称

该记录器名提供商自动完成有效的记录程序的名称。通常，当前项目中可用的包名和类名可以自动完成。特定的框架可能还有额外的魔术记录器名称，这些名称也可以被支持。

由于记录器名称可以是任意名称，因此此提供程序应允许任何值，但可以突出显示项目类路径中不可用的有效包名和类名。

以下元数据片段对应于标准 `logging.level` 属性。键是记录器名称，值对应于标准日志级别或任何自定义级别。

```
{
  "hints": [
    {
      "name": "logging.level.keys",
      "values": [
        {
          "value": "root",
          "description": "用于指定默认日志记录级别的根记录器。"
        }
      ],
      "providers": [
        {
          "name": "logger-name"
        }
      ]
    },
    {
      "name": "logging.level.values",
      "values": [
        {
          "值": "调试"
        },
        {
          "值": "信息"
        },
        {
          "值": "警告"
        },
        {
          "值": "错误"
        },
        {
          "值": "致命"
        },
        {
          "value": "off"
        }
      ],
      "providers": [
        {
          "name": "any"
        }
      ]
    }
  ]
}
```

Spring Bean参考

该**弹簧豆参考**提供商自动完成当前项目的配置中定义的豆类。该提供程序支持以下参数：

参数	类型	默认值	描述
<code>target</code>	<code>String</code> (<code>Class</code>)	没有	应该可分配给候选人的bean类的完全限定名称。通常用于过滤掉非候选bean。

以下元数据片段对应于`spring.jmx.server` 定义 `MBeanServer` 要使用的bean 的名称的标准属性：

```
{ "提示": [
    {
        "姓名": "spring.jmx.server" ,
        "提供者": [
            {
                "姓名": "弹簧豆参考" ,
                "参数": {
                    "目标": "javax.management .MBeanServer"
                }
            }
        ]
    }
]}
```



活页夹不知道元数据。如果你提供了这个提示，你仍然需要将bean的名字转换成一个实际的Bean引用 `ApplicationContext`。

Spring配置文件名称

该**弹簧轮廓名**提供商自动完成当前项目的配置中定义的春天配置文件。

以下元数据片段对应于`spring.profiles.active` 定义要启用的Spring配置文件名称的标准属性：

```
{ "hints": [
    {
        "name": "spring.profiles.active" ,
        "providers": [
            {
                "name": "spring-profile-name"
            }
        ]
    }
]}
```

B.3使用注释处理器生成自己的元数据

您可以 `@ConfigurationProperties` 通过使用 `spring-boot-configuration-processor` jar 注释的项目轻松生成您自己的配置元数据文件。该jar包含一个Java注释处理器，在编译项目时调用它。要使用处理器，请包含依赖项 `spring-boot-configuration-processor`。

使用Maven时，应将依赖项声明为可选项，如下例所示：

```
<dependency>
    <groupId> org.springframework.boot </ groupId>
    <artifactId> spring-boot-configuration-processor </ artifactId>
    <optional> true </ optional>
</ dependency>
```

在Gradle 4.5及更早版本中，应该在 `compileOnly` 配置中声明依赖关系，如以下示例所示：

```
依赖关系{
    compileOnly "org.springframework.boot: spring-boot-configuration-processor"
}
```

在Gradle 4.6及更高版本中，应该在 `annotationProcessor` 配置中声明依赖关系，如以下示例所示：

```
依赖关系{
    annotationProcessor "org.springframework.boot: spring-boot-configuration-processor"
}
```

如果您正在使用 `additional-spring-configuration-metadata.json` 文件，`compileJava` 则应将任务配置为依赖于该 `processResources` 任务，如以下示例所示：

```
compileJava.dependsOn(processResources)
```

这种依赖性确保了在编译过程中注释处理器运行时附加元数据可用。

处理器拾取注释了的类和方法 `@ConfigurationProperties`。配置类中字段值的Javadoc用于填充 `description` 属性。



您应该只使用带有 `@ConfigurationProperties` 字段Javadoc的简单文本，因为它们在添加到JSON之前不会被处理。

属性是通过标准getter和setter的存在发现的，对集合类型有特殊处理（即使只有getter也可以检测到）。注释处理器还支持使用的 `@Data`，`@Getter` 和 `@Setter` Lombok的注释。



如果您在项目中使用AspectJ，则需要确保注释处理器只运行一次。有几种方法可以做到这一点。通过Maven，您可以以 `maven-apt-plugin` 明确地配置并且仅在那里将依赖关系添加到注释处理器。您还可以让AspectJ插件运行所有处理并禁用 `maven-compiler-plugin` 配置中的注释处理，如下所示：

```
<plugin>
    <groupId> org.apache.maven.plugins </ groupId>
    <artifactId> maven-compiler-plugin </ artifactId>
    <configuration>
        <proc> none </ proc>
    </ configuration>
</ plugin>
```

B.3.1 嵌套属性

注释处理器自动将内部类视为嵌套属性。考虑以下课程：

```
@ConfigurationProperties (prefix =“server”)
public class ServerProperties {

    私人字符串名称;

    私人主机主机;

    // ... getter 和 setter

    私人 静态 类主机{

        私人字符串IP;

        private int port;

        // ... getter 和 setter

    }

}
```

前面的例子中产生用于元数据信息 `server.name`，`server.host.ip` 和 `server.host.port` 属性。您可以 `@NestedConfigurationProperty` 在字段上使用注释来指示应该像处理嵌套一样处理常规（非内部）类。



这对集合和地图没有影响，因为这些类型是自动识别的，并且为它们中的每一个生成单个元数据属性。

B.3.2 添加额外的元数据

Spring Boot的配置文件处理非常灵活，并且通常情况下可能存在不绑定到 `@ConfigurationProperties` bean的属性。您可能还需要调整现有密钥的某些属性。为了支持这种情况并让您提供自定义“提示”，注释处理器会自动将项目合，并 `META-INF/additional-spring-configuration-metadata.json` 到主元数据文件中。

如果您引用自动检测到的属性，则说明，默认值和弃用信息将被覆盖（如果指定）。如果手动属性声明在当前模块中未被识别，则将其添加为新属性。

该 `additional-spring-configuration-metadata.json` 文件的格式与普通文件完全相同 `spring-configuration-metadata.json`。附加属性文件是可选的。如果您没有任何其他属性，请不要添加该文件。

附录C.自动配置类

这里是Spring Boot提供的所有自动配置类的列表，包含文档和源代码的链接。请记住在应用程序中查看条件报告，了解哪些功能处于打开状态。（为此，请使用 `--debug` 或 `-Ddebug` 在Actuator应用程序中使用 `conditions` 端点来启动应用程序）。

C.1从“spring-boot-autoconfigure”模块

以下自动配置类来自 `spring-boot-autoconfigure` 模块：

配置类	链接
ActiveMQAutoConfiguration	的javadoc
AopAutoConfiguration	的javadoc
ArtemisAutoConfiguration	的javadoc
BatchAutoConfiguration	的javadoc
CacheAutoConfiguration	的javadoc
CassandraAutoConfiguration	的javadoc
CassandraDataAutoConfiguration	的javadoc
CassandraReactiveDataAutoConfiguration	的javadoc
CassandraReactiveRepositoriesAutoConfiguration	的javadoc
CassandraRepositoriesAutoConfiguration	的javadoc
CloudAutoConfiguration	的javadoc
CodecsAutoConfiguration	的javadoc
ConfigurationPropertiesAutoConfiguration	的javadoc
CouchbaseAutoConfiguration	的javadoc
CouchbaseDataAutoConfiguration	的javadoc
CouchbaseReactiveDataAutoConfiguration	的javadoc
CouchbaseReactiveRepositoriesAutoConfiguration	的javadoc
CouchbaseRepositoriesAutoConfiguration	的javadoc
DataSourceAutoConfiguration	的javadoc
DataSourceTransactionManagerAutoConfiguration	的javadoc
DispatcherServletAutoConfiguration	的javadoc
ElasticsearchAutoConfiguration	的javadoc
ElasticsearchDataAutoConfiguration	的javadoc

配置类

ElasticsearchRepositoriesAutoConfiguration	的javadoc
EmbeddedLdapAutoConfiguration	的javadoc
EmbeddedMongoAutoConfiguration	的javadoc
EmbeddedWebServerFactoryCustomizerAutoConfiguration	的javadoc
ErrorMvcAutoConfiguration	的javadoc
ErrorWebFluxAutoConfiguration	的javadoc
FlywayAutoConfiguration	的javadoc
FreeMarkerAutoConfiguration	的javadoc
GroovyTemplateAutoConfiguration	的javadoc
GsonAutoConfiguration	的javadoc
H2ConsoleAutoConfiguration	的javadoc
HazelcastAutoConfiguration	的javadoc
HazelcastJpaDependencyAutoConfiguration	的javadoc
HibernateJpaAutoConfiguration	的javadoc
HttpEncodingAutoConfiguration	的javadoc
HttpHandlerAutoConfiguration	的javadoc
HttpMessageConvertersAutoConfiguration	的javadoc
HypermediaAutoConfiguration	的javadoc
InfluxDbAutoConfiguration	的javadoc
IntegrationAutoConfiguration	的javadoc
JacksonAutoConfiguration	的javadoc
JdbcTemplateAutoConfiguration	的javadoc
JerseyAutoConfiguration	的javadoc
JestAutoConfiguration	的javadoc
JmsAutoConfiguration	的javadoc
JmxAutoConfiguration	的javadoc
JndiConnectionFactoryAutoConfiguration	的javadoc
JndiDataSourceAutoConfiguration	的javadoc
JooqAutoConfiguration	的javadoc

配置类

链接	
的javadoc	JpaRepositoriesAutoConfiguration
的javadoc	JsonbAutoConfiguration
的javadoc	JtaAutoConfiguration
的javadoc	KafkaAutoConfiguration
的javadoc	LdapAutoConfiguration
的javadoc	LdapDataAutoConfiguration
的javadoc	LdapRepositoriesAutoConfiguration
的javadoc	LiquibaseAutoConfiguration
的javadoc	MailSenderAutoConfiguration
的javadoc	MailSenderValidatorAutoConfiguration
的javadoc	MessageSourceAutoConfiguration
的javadoc	MongoAutoConfiguration
的javadoc	MongoDataAutoConfiguration
的javadoc	MongoReactiveAutoConfiguration
的javadoc	MongoReactiveDataAutoConfiguration
的javadoc	MongoReactiveRepositoriesAutoConfiguration
的javadoc	MongoRepositoriesAutoConfiguration
的javadoc	MultipartAutoConfiguration
的javadoc	MustacheAutoConfiguration
的javadoc	Neo4jDataAutoConfiguration
的javadoc	Neo4jRepositoriesAutoConfiguration
的javadoc	OAuth2ClientAutoConfiguration
的javadoc	PersistenceExceptionTranslationAutoConfiguration
的javadoc	ProjectInfoAutoConfiguration
的javadoc	PropertyPlaceholderAutoConfiguration
的javadoc	QuartzAutoConfiguration
的javadoc	RabbitAutoConfiguration
的javadoc	ReactiveSecurityAutoConfiguration
的javadoc	ReactiveUserDetailsServiceAutoConfiguration

配置类

ReactiveWebServerFactoryAutoConfiguration	的javadoc
ReactorCoreAutoConfiguration	的javadoc
RedisAutoConfiguration	的javadoc
RedisReactiveAutoConfiguration	的javadoc
RedisRepositoriesAutoConfiguration	的javadoc
RepositoryRestMvcAutoConfiguration	的javadoc
RestTemplateAutoConfiguration	的javadoc
SecurityAutoConfiguration	的javadoc
SecurityFilterAutoConfiguration	的javadoc
SendGridAutoConfiguration	的javadoc
ServletWebServerFactoryAutoConfiguration	的javadoc
SessionAutoConfiguration	的javadoc
SolrAutoConfiguration	的javadoc
SolrRepositoriesAutoConfiguration	的javadoc
SpringApplicationAdminJmxAutoConfiguration	的javadoc
SpringDataWebAutoConfiguration	的javadoc
ThymeleafAutoConfiguration	的javadoc
TransactionAutoConfiguration	的javadoc
UserDetailsServiceAutoConfiguration	的javadoc
ValidationAutoConfiguration	的javadoc
WebClientAutoConfiguration	的javadoc
WebFluxAutoConfiguration	的javadoc
WebMvcAutoConfiguration	的javadoc
WebServicesAutoConfiguration	的javadoc
WebSocketMessagingAutoConfiguration	的javadoc
WebSocketReactiveAutoConfiguration	的javadoc
WebSocketServletAutoConfiguration	的javadoc
XADataSourceAutoConfiguration	的javadoc

C.2从“spring-boot-actuator-autoconfigure”模块

以下自动配置类来自[spring-boot-actuator-autoconfigure](#)模块：

配置类	链接
AtlasMetricsExportAutoConfiguration	的javadoc
AuditAutoConfiguration	的javadoc
AuditEventsEndpointAutoConfiguration	的javadoc
BeansEndpointAutoConfiguration	的javadoc
CacheMetricsAutoConfiguration	的javadoc
CassandraHealthIndicatorAutoConfiguration	的javadoc
CloudFoundryActuatorAutoConfiguration	的javadoc
CompositeMeterRegistryAutoConfiguration	的javadoc
ConditionsReportEndpointAutoConfiguration	的javadoc
ConfigurationPropertiesReportEndpointAutoConfiguration	的javadoc
CouchbaseHealthIndicatorAutoConfiguration	的javadoc
DataSourceHealthIndicatorAutoConfiguration	的javadoc
DataSourcePoolMetricsAutoConfiguration	的javadoc
DatadogMetricsExportAutoConfiguration	的javadoc
DiskSpaceHealthIndicatorAutoConfiguration	的javadoc
ElasticsearchHealthIndicatorAutoConfiguration	的javadoc
EndpointAutoConfiguration	的javadoc
EnvironmentEndpointAutoConfiguration	的javadoc
FlywayEndpointAutoConfiguration	的javadoc
GangliaMetricsExportAutoConfiguration	的javadoc
GraphiteMetricsExportAutoConfiguration	的javadoc
HealthEndpointAutoConfiguration	的javadoc
HealthIndicatorAutoConfiguration	的javadoc
HeapDumpWebEndpointAutoConfiguration	的javadoc
HttpTraceAutoConfiguration	的javadoc
HttpTraceEndpointAutoConfiguration	的javadoc
InfluxDbHealthIndicatorAutoConfiguration	的javadoc

配置类

InfluxMetricsExportAutoConfiguration	的javadoc
InfoContributorAutoConfiguration	的javadoc
InfoEndpointAutoConfiguration	的javadoc
JmsHealthIndicatorAutoConfiguration	的javadoc
JmxEndpointAutoConfiguration	的javadoc
JmxMetricsExportAutoConfiguration	的javadoc
JolokiaEndpointAutoConfiguration	的javadoc
LdapHealthIndicatorAutoConfiguration	的javadoc
LiquibaseEndpointAutoConfiguration	的javadoc
LogFileWebEndpointAutoConfiguration	的javadoc
LoggersEndpointAutoConfiguration	的javadoc
MailHealthIndicatorAutoConfiguration	的javadoc
ManagementContextAutoConfiguration	的javadoc
MappingsEndpointAutoConfiguration	的javadoc
MetricsAutoConfiguration	的javadoc
MetricsEndpointAutoConfiguration	的javadoc
MongoHealthIndicatorAutoConfiguration	的javadoc
Neo4jHealthIndicatorAutoConfiguration	的javadoc
NewRelicMetricsExportAutoConfiguration	的javadoc
PrometheusMetricsExportAutoConfiguration	的javadoc
RabbitHealthIndicatorAutoConfiguration	的javadoc
RabbitMetricsAutoConfiguration	的javadoc
ReactiveCloudFoundryActuatorAutoConfiguration	的javadoc
ReactiveManagementContextAutoConfiguration	的javadoc
RedisHealthIndicatorAutoConfiguration	的javadoc
RestTemplateMetricsAutoConfiguration	的javadoc
ScheduledTasksEndpointAutoConfiguration	的javadoc
ServletManagementContextAutoConfiguration	的javadoc
SessionsEndpointAutoConfiguration	的javadoc

配置类	链接
<code>ShutdownEndpointAutoConfiguration</code>	的javadoc
<code>SignalFxMetricsExportAutoConfiguration</code>	的javadoc
<code>SimpleMetricsExportAutoConfiguration</code>	的javadoc
<code>SolrHealthIndicatorAutoConfiguration</code>	的javadoc
<code>StatsdMetricsExportAutoConfiguration</code>	的javadoc
<code>ThreadDumpEndpointAutoConfiguration</code>	的javadoc
<code>TomcatMetricsAutoConfiguration</code>	的javadoc
<code>WavefrontMetricsExportAutoConfiguration</code>	的javadoc
<code>WebEndpointAutoConfiguration</code>	的javadoc
<code>WebFluxMetricsAutoConfiguration</code>	的javadoc
<code>WebMvcMetricsAutoConfiguration</code>	的javadoc

附录D. 测试自动配置注释

下表列出了`@...Test`可用于测试应用程序切片以及默认导入的自动配置的各种注释：

测试片	导入自动配置
<code>@DataJpaTest</code>	<code>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration</code> <code>org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration</code> <code>org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManagerAutoConfiguration</code>
<code>@DataLdapTest</code>	<code>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.ldap.LdapDataAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration</code>
<code>@DataMongoTest</code>	<code>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration</code>

测试片**导入自动配置**

@DataNeo4jTest

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration
```

@DataRedisTest

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration
```

@JdbcTest

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration
org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration
```

@JooqTest

```
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration
```

@JsonTest

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration
org.springframework.boot.test.autoconfigure.json.JsonTestersAutoConfiguration
```

@RestClientTest

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration
org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration
org.springframework.boot.test.autoconfigure.web.client.MockRestServiceServerAutoConfiguration
org.springframework.boot.test.autoconfigure.web.client.WebClientRestTemplateAutoConfiguration
```

@WebFluxTest

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration
org.springframework.boot.test.autoconfigure.web.reactive.WebTestClientAutoConfiguration
```

测试片

导入自动配置

@WebMvcTest

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration
org.springframework.boot.test.autoconfigure.web.servlet.MockMvcMvcAutoConfiguration
org.springframework.boot.test.autoconfigure.web.servlet.MockMvcMvcSecurityAutoConfiguration
org.springframework.boot.test.autoconfigure.web.servlet.MockMvcWebClientAutoConfiguration
org.springframework.boot.test.autoconfigure.web.servlet.MockMvcWebDriverAutoConfiguration
```

附录E.可执行Jar格式

这些 `spring-boot-loader` 模块让 Spring Boot 支持可执行的 jar 和 war 文件。如果您使用 Maven 插件或 Gradle 插件，可自动生成可执行文件，并且您通常不需要知道它们的工作方式。

如果您需要从不同的构建系统创建可执行文件，或者您只是对底层技术感兴趣，本节提供一些背景知识。

E.1 嵌套JAR

Java 没有提供任何标准的方法来加载嵌套的 jar 文件（也就是本身包含在 jar 中的 jar 文件）。如果您需要分发可以从命令行运行而不打开的自包含应用程序，则这可能会有问题。

为了解决这个问题，许多开发人员使用“阴影”罐子。阴影罐将所有罐子中的所有类包装成一个“超级罐子”。带阴影的瓶子的问题是，很难看到哪些库实际上在您的应用程序中。如果在多个罐子中使用相同的文件名（但是具有不同的内容），则它也可能是有问题的。Spring Boot 采用了不同的方法，可以让您直接嵌入罐子。

E.1.1 可执行jar文件结构

Spring Boot Loader 兼容的 jar 文件应该按以下方式构建：

```
example.jar
|
+ -META-INF
| + -MANIFEST.MF
+ -org
| + -springframework
| + -boot
| + -loader
| + - <spring引导装载程序类>
+ -boot-INF
    + - 班
    | + -mycompany
    | + - 项目
    | + -YourClasses.class
+ -lib
    + -dependency1.jar
    + -dependency2.jar
```

应用程序类应放置在嵌套 `BOOT-INF/classes` 目录中。依赖关系应放置在嵌套的 `BOOT-INF/lib` 目录中。

E.1.2 可执行的战争文件结构

Spring Boot Loader 兼容的 war 文件应该按照以下方式构建：

```

example.war
|
+ -META-INF
| + -MANIFEST.MF
+ -org
| + -springframework
| + -boot
| + -loader
| + - <spring引导装载程序类>
+ -WEB-INF
  + - 班
  | + -com
  | + -mycompany
  | + - 项目
  | + -YourClasses.class
  + -lib
  | + -dependency1.jar
  | + -dependency2.jar
  + -lib提供的
    + -servlet-api.jar文件
    + -dependency3.jar

```

依赖关系应放置在嵌套的 `WEB-INF/lib` 目录中。运行嵌入式时需要的任何依赖关系，但部署到传统Web容器时不需要的依赖关系应放置在 `WEB-INF/lib-provided`。

E.2 Spring Boot的“JarFile”类

用于支持加载嵌套罐的核心类是 `org.springframework.boot.loader.jar.JarFile`。它允许您从标准jar文件或嵌套jar数据中加载jar内容。第一次加载时，它们的位置 `JarEntry` 映射到外部jar的物理文件偏移量，如以下示例所示：

```

myapp.jar
+ ----- + ----- +
| / BOOT-INF / classes | /BOOT-INF/lib/mylib.jar | | | | | |
| + ----- + || + ----- + ----- + |
|| A.class ||| B.class | C.class ||
| + ----- + || + ----- + ----- + |
+ ----- + ----- + |
^ ^ ^
0063 3452 3980

```

前面的例子示出了如何 `A.class` 可以发现 `/BOOT-INF/classes` 在 `myapp.jar` 在位置 `0063`。`B.class` 从嵌套的罐子里实际上可以找到的 `myapp.jar` 位置 `3452`，并 `C.class` 在位置 `3980`。

有了这些信息，我们可以通过寻找外部jar的适当部分来加载特定的嵌套条目。我们不需要解压缩存档，我们也不需要将所有条目数据读入内存。

E.2.1与标准Java“JarFile”的兼容性

Spring Boot Loader努力保持与现有代码和库的兼容性。`org.springframework.boot.loader.jar.JarFile` 延伸 `java.util.jar.JarFile` 并应该作为一个直接替代品。该 `getURL()` 方法返回一个 `URL` 打开 `java.net.JarURLConnection` 与Java 兼容的连接并且可以与Java一起使用的连接 `URLClassLoader`。

E.3启动可执行的罐子

该 `org.springframework.boot.loader.Launcher` 班是作为一个可执行的JAR的主入口点一个特殊的启动类。这是 `Main-Class` 您的jar文件中的实际内容，它用于设置合适的 `URLClassLoader` 并最终调用您的 `main()` 方法。

有三个发射子类 (`JarLauncher`, `WarLauncher`, 和 `PropertiesLauncher`)。它们的目的是 `.class` 从目录中的嵌套jar文件或war文件 (与显式地在类路径中的那些文件相对) 加载资源 (文件等)。在的情况下 `JarLauncher` 和 `WarLauncher`，嵌套路径是固定的。`JarLauncher` 看起来 `BOOT-INF/lib/`，`WarLauncher` 看起来 `WEB-INF/lib/` 和 `WEB-INF/lib-provided/`。如果您想要更多，可以在这些位置添加额外的罐子。将 `PropertiesLauncher` 在外观 `BOOT-INF/lib/` 默认情况下你的应用程序归档文件，但是你可以通过设置所谓的环境变量添加其他位置 `LOADER_PATH` 或 `loader.path` 在 `loader.properties` (这是目录，归档或目录的归档中的一个逗号分隔的列表)。

E.3.1启动器清单

你需要指定一个适当 `Launcher` 的 `Main-Class` 属性 `META-INF/MANIFEST.MF`。`main` 应该在 `Start-Class` 属性中指定要启动的实际类 (即包含方法的类)。

以下示例显示了 `MANIFEST.MF` 可执行jar文件的典型示例：

```
主要类: org.springframework.boot.loader.JarLauncher
开始 - 类: com.mycompany.project.MyApplication
```

对于战争档案，它将如下所示：

```
主要类: org.springframework.boot.loader.WarLauncher
开始 - 类: com.mycompany.project.MyApplication
```



您不需要 `Class-Path` 在清单文件中指定条目。类路径是从嵌套的jar中推导出来的。

E.3.2 分解档案

某些PaaS实现可能会选择在运行之前解压缩归档文件。例如，Cloud Foundry以这种方式运作。您可以通过启动适当的启动程序运行解压缩的归档文件，如下所示：

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```

E.4 PropertiesLauncher 特征

`PropertiesLauncher`有一些可以通过外部属性（系统属性，环境变量，清单条目或者 `loader.properties`）启用的特殊功能。下表描述了这些属性：

键	目的
<code>loader.path</code>	逗号分隔的Classpath，如 <code>lib,\${HOME}/app/lib</code> 。早期的条目优先，就像命令行 <code>-classpath</code> 上的常规条目 <code>javac</code> 。
<code>loader.home</code>	用于解析中的相对路径 <code>loader.path</code> 。例如，给定 <code>loader.path=lib</code> ，然后 <code> \${loader.home}/lib</code> 是一个类路径位置（以及该目录中的所有jar文件）。该属性也用于查找 <code>loader.properties</code> 文件，如下例所示： <code>/opt/app</code> 它默认为 <code> \${user.dir}</code> 。
<code>loader.args</code>	主方法的默认参数（空格分隔）。
<code>loader.main</code>	要启动的主要类的名称（例如， <code>com.app.Application</code> ）。
<code>loader.config.name</code>	属性文件的名称（例如， <code>launcher</code> ）默认为 <code>loader</code> 。
<code>loader.config.location</code>	属性文件的路径（例如， <code>classpath:loader.properties</code> ）。它默认为 <code>loader.properties</code> 。
<code>loader.system</code>	布尔标志，指示应将所有属性添加到系统属性它默认为 <code>false</code> 。

当指定为环境变量或清单条目时，应使用以下名称：

键	清单条目	环境变量
<code>loader.path</code>	<code>Loader-Path</code>	<code>LOADER_PATH</code>
<code>loader.home</code>	<code>Loader-Home</code>	<code>LOADER_HOME</code>
<code>loader.args</code>	<code>Loader-Args</code>	<code>LOADER_ARGS</code>
<code>loader.main</code>	<code>Start-Class</code>	<code>LOADER_MAIN</code>
<code>loader.config.location</code>	<code>Loader-Config-Location</code>	<code>LOADER_CONFIG_LOCATION</code>
<code>loader.system</code>	<code>Loader-System</code>	<code>LOADER_SYSTEM</code>



构建插件会在构建胖jar时自动移动该 `Main-Class` 属性 `Start-Class`。如果使用该 `Main-Class` 属性，请使用该属性指定要启动的类的名称并省略 `Start-Class`。

以下规则适用于处理 `PropertiesLauncher`：

- `loader.properties` `loader.home` 在类路径的根目录中搜索，然后在中搜索 `classpath:/BOOT-INF/classes`。使用具有该名称的文件的第一个位置。
- `loader.home` 是仅当 `loader.config.location` 未指定时，附加属性文件的目录位置（覆盖默认值）。
- `loader.path` 可以包含用于jar和zip文件递归扫描的目录，归档路径，扫描jar文件（例如 `dependencies.jar!/lib`）的归档中的目录或通配符模式（用于缺省JVM行为）。归档路径可以是相对于 `loader.home` 文件系统中的任何位置，也可以是带有 `jar:file:` 前缀的任
- `loader.path`（如果为空）默认为 `BOOT-INF/lib`（表示本地目录或从存档运行的嵌套目录）。因此，`PropertiesLauncher` 与 `JarLauncher` 没有提供附加配置时的行为相同。
- `loader.path` 不能用于配置位置 `loader.properties`（用于搜索后者的类路径 `PropertiesLauncher` 是启动时的JVM类路径）。
- 占位符替换是在使用前从系统和环境变量以及所有值的属性文件本身完成的。
- 属性的搜索顺序（在多个位置查找是有意义的）是环境变量，系统属性，`loader.properties` 分解的归档清单和归档清单。

E.5 可执行的瓶子限制

在使用Spring Boot Loader打包的应用程序时，您需要考虑以下限制：

- Zip条目压缩：`ZipEntry` 必须使用该 `ZipEntry.STORED` 方法保存嵌套的jar。这是必需的，以便我们可以直接寻找嵌套jar中的单个内容。嵌套jar文件本身的内容仍然可以压缩，外部jar中的任何其他条目也可以压缩。
- 系统 `Thread.getContextClassLoader()` 类加载器：启动的应用程序应该在加载类时使用（大多数库和框架默认是这样做的）。尝试加载 `ClassLoader.getSystemClassLoader()` 失败的嵌套jar类。`java.util.Logging` 始终使用系统类加载器。出于这个原因，你应该考虑一个不同的日志实现。

E.6 另一种单罐解决方案

如果上述限制意味着您不能使用Spring Boot Loader，请考虑以下选择：

- Maven Shade插件
- 的JarClassLoader
- OneJar

附录F.相关版本

下表提供了Spring Boot在其CLI（命令行界面），Maven依赖项管理和Gradle插件中提供的所有依赖项版本的详细信息。当您声明对这些工件之一的依赖关系而未声明版本时，将使用表中列出的版本。

组ID	工件ID	版
<code>antlr</code>	<code>antlr</code>	2.7.7
<code>ch.qos.logback</code>	<code>logback-access</code>	1.2.3
<code>ch.qos.logback</code>	<code>logback-classic</code>	1.2.3
<code>ch.qos.logback</code>	<code>logback-core</code>	1.2.3
<code>com.atomikos</code>	<code>transactions-jdbc</code>	4.0.6
<code>com.atomikos</code>	<code>transactions-jms</code>	4.0.6
<code>com.atomikos</code>	<code>transactions-jta</code>	4.0.6
<code>com.couchbase.client</code>	<code>couchbase-spring-cache</code>	2.1.0
<code>com.couchbase.client</code>	<code>java-client</code>	2.5.5
<code>com.datastax.cassandra</code>	<code>cassandra-driver-core</code>	3.4.0

组ID	工件ID	版
com.datastax.cassandra	cassandra-driver-mapping	3.4.0
com.fasterxml	classmate	1.3.4
com.fasterxml.jackson.core	jackson-annotations	2.9.0
com.fasterxml.jackson.core	jackson-core	2.9.4
com.fasterxml.jackson.core	jackson-databind	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-avro	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-cbor	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-csv	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-ion	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-properties	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-protobuf	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-smile	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-xml	2.9.4
com.fasterxml.jackson.dataformat	jackson-dataformat-yaml	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-guava	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-hibernate3	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-hibernate4	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-hibernate5	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-hppc	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-jaxrs	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-jdk8	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-joda	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-json-org	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-jsr310	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-jsr353	2.9.4
com.fasterxml.jackson.datatype	jackson-datatype-pcollections	2.9.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-base	2.9.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-cbor-provider	2.9.4

组ID	工件ID	版
com.fasterxml.jackson.jaxrs	jackson-jaxrs-json-provider	2.9.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-smile-provider	2.9.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-xml-provider	2.9.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-yaml-provider	2.9.4
com.fasterxml.jackson.jr	jackson-jr-all	2.9.4
com.fasterxml.jackson.jr	jackson-jr-objects	2.9.4
com.fasterxml.jackson.jr	jackson-jr-retrofit2	2.9.4
com.fasterxml.jackson.jr	jackson-jr-stree	2.9.4
com.fasterxml.jackson.module	jackson-module-afterburner	2.9.4
com.fasterxml.jackson.module	jackson-module-guice	2.9.4
com.fasterxml.jackson.module	jackson-module-jaxb-annotations	2.9.4
com.fasterxml.jackson.module	jackson-module-jsonSchema	2.9.4
com.fasterxml.jackson.module	jackson-module-kotlin	2.9.4
com.fasterxml.jackson.module	jackson-module-mrbean	2.9.4
com.fasterxml.jackson.module	jackson-module-osgi	2.9.4
com.fasterxml.jackson.module	jackson-module-parameter-names	2.9.4
com.fasterxml.jackson.module	jackson-module-paranamer	2.9.4
com.fasterxml.jackson.module	jackson-module-scala_2.10	2.9.4
com.fasterxml.jackson.module	jackson-module-scala_2.11	2.9.4
com.fasterxml.jackson.module	jackson-module-scala_2.12	2.9.4
com.github.ben-manes.caffeine	caffeine	2.6.2
com.github.mxab.thymeleaf.extras	thymeleaf-extras-data-attribute	2.0.1
com.google.appengine	appengine-api-1.0-sdk	63年9月1日
com.google.code.gson	gson	2.8.2
com.googlecode.json-simple	json-simple	1.1.1
com.h2database	h2	1.4.196
com.hazelcast	hazelcast	3.9.3
com.hazelcast	hazelcast-client	3.9.3
com.hazelcast	hazelcast-hibernate52	1.2.3

组ID	工件ID	版
com.hazelcast	hazelcast-spring	3.9.3
com.jayway.jsonpath	json-path	2.4.0
com.jayway.jsonpath	json-path-assert	2.4.0
com.microsoft.sqlserver	mssql-jdbc	6.2.2.jre8
com.querydsl	querydsl-apt	4.1.4
com.querydsl	querydsl-collections	4.1.4
com.querydsl	querydsl-core	4.1.4
com.querydsl	querydsl-jpa	4.1.4
com.querydsl	querydsl-mongodb	4.1.4
com.rabbitmq	amqp-client	5.1.2
com.samskivert	jmustache	1.14
com.sendgrid	sendgrid-java	4.1.2
com.sun.mail	javax.mail	1.6.1
com.timgroup	java-statsd-client	3.1.0
com.unboundid	unboundid-ldapsdk	4.0.4
com.zaxxer	HikariCP	2.7.8
commons-codec	commons-codec	1.11
commons-pool	commons-pool	1.6
de.flapdoodle.embed	de.flapdoodle.embed.mongo	2.0.3
dom4j	dom4j	1.6.1
io.dropwizard.metrics	metrics-annotation	3.2.6
io.dropwizard.metrics	metrics-core	3.2.6
io.dropwizard.metrics	metrics-ehcache	3.2.6
io.dropwizard.metrics	metrics-ganglia	3.2.6
io.dropwizard.metrics	metrics-graphite	3.2.6
io.dropwizard.metrics	metrics-healthchecks	3.2.6
io.dropwizard.metrics	metrics-httpasyncclient	3.2.6
io.dropwizard.metrics	metrics-jdbi	3.2.6
io.dropwizard.metrics	metrics-jersey	3.2.6

组ID	工件ID	版
io.dropwizard.metrics	metrics-jersey2	3.2.6
io.dropwizard.metrics	metrics-jetty8	3.2.6
io.dropwizard.metrics	metrics-jetty9	3.2.6
io.dropwizard.metrics	metrics-jetty9-legacy	3.2.6
io.dropwizard.metrics	metrics-json	3.2.6
io.dropwizard.metrics	metrics-jvm	3.2.6
io.dropwizard.metrics	metrics-log4j	3.2.6
io.dropwizard.metrics	metrics-log4j2	3.2.6
io.dropwizard.metrics	metrics-logback	3.2.6
io.dropwizard.metrics	metrics-servlet	3.2.6
io.dropwizard.metrics	metrics-servlets	3.2.6
io.lettuce	lettuce-core	5.0.2.RELEASE
io.micrometer	micrometer-core	1.0.1
io.micrometer	micrometer-registry-atlas	1.0.1
io.micrometer	micrometer-registry-datadog	1.0.1
io.micrometer	micrometer-registry-ganglia	1.0.1
io.micrometer	micrometer-registry-graphite	1.0.1
io.micrometer	micrometer-registry-influx	1.0.1
io.micrometer	micrometer-registry-jmx	1.0.1
io.micrometer	micrometer-registry-new-relic	1.0.1
io.micrometer	micrometer-registry-prometheus	1.0.1
io.micrometer	micrometer-registry-signalfx	1.0.1
io.micrometer	micrometer-registry-statsd	1.0.1
io.micrometer	micrometer-registry-wavefront	1.0.1
io.netty	netty-all	4.1.22.Final
io.netty	netty-buffer	4.1.22.Final
io.netty	netty-codec	4.1.22.Final
io.netty	netty-codec-dns	4.1.22.Final
io.netty	netty-codec-haproxy	4.1.22.Final

组ID	工件ID	版
io.netty	netty-codec-http	4.1.22.Final
io.netty	netty-codec-http2	4.1.22.Final
io.netty	netty-codec-memcache	4.1.22.Final
io.netty	netty-codec-mqtt	4.1.22.Final
io.netty	netty-codec-redis	4.1.22.Final
io.netty	netty-codec-smtp	4.1.22.Final
io.netty	netty-codec-socks	4.1.22.Final
io.netty	netty-codec-stomp	4.1.22.Final
io.netty	netty-codec-xml	4.1.22.Final
io.netty	netty-common	4.1.22.Final
io.netty	netty-dev-tools	4.1.22.Final
io.netty	netty-example	4.1.22.Final
io.netty	netty-handler	4.1.22.Final
io.netty	netty-handler-proxy	4.1.22.Final
io.netty	netty-resolver	4.1.22.Final
io.netty	netty-resolver-dns	4.1.22.Final
io.netty	netty-transport	4.1.22.Final
io.netty	netty-transport-native-epoll	4.1.22.Final
io.netty	netty-transport-native-kqueue	4.1.22.Final
io.netty	netty-transport-native-unix-common	4.1.22.Final
io.netty	netty-transport-rxtx	4.1.22.Final
io.netty	netty-transport-sctp	4.1.22.Final
io.netty	netty-transport-udt	4.1.22.Final
io.projectreactor	reactor-core	3.1.5.RELEASE
io.projectreactor	reactor-test	3.1.5.RELEASE
io.projectreactor.addons	reactor-adapter	3.1.6.RELEASE
io.projectreactor.addons	reactor-extra	3.1.6.RELEASE
io.projectreactor.addons	reactor-logback	3.1.6.RELEASE
io.projectreactor.ipc	reactor-netty	0.7.5.RELEASE

组ID	工件ID	版
io.projectreactor.kafka	reactor-kafka	1.0.0.RELEASE
io.reactivex	rxjava	1.3.6
io.reactivex	rxjava-reactive-streams	1.2.1
io.reactivex.rxjava2	rxjava	2.1.10
io.rest-assured	json-path	3.0.7
io.rest-assured	json-schema-validator	3.0.7
io.rest-assured	rest-assured	3.0.7
io.rest-assured	scala-support	3.0.7
io.rest-assured	spring-mock-mvc	3.0.7
io.rest-assured	xml-path	3.0.7
io.searchbox	jest	5.3.3
io.undertow	undertow-core	1.4.23.Final
io.undertow	undertow-servlet	1.4.23.Final
io.undertow	undertow-websockets-jsr	1.4.23.Final
javax.annotation	javax.annotation-api	1.3.2
javax.cache	cache-api	1.1.0
javax.jms	javax.jms-api	2.0.1
javax.json	javax.json-api	1.1.2
javax.json.bind	javax.json.bind-api	1.0
javax.mail	javax.mail-api	1.6.1
javax.money	money-api	1.0.1
javax.servlet	javax.servlet-api	3.1.0
javax.servlet	jstl	1.2
javax.transaction	javax.transaction-api	1.2
javax.validation	validation-api	2.0.1.Final
javax.xml.bind	jaxb-api	2.3.0
jaxen	jaxen	1.1.6
joda-time	joda-time	2.9.9
junit	junit	4.12

组ID	工件ID	版
mysql	mysql-connector-java	45年5月1日
net.bytebuddy	byte-buddy	1.7.10
net.bytebuddy	byte-buddy-agent	1.7.10
net.java.dev.jna	jna	4.5.1
net.java.dev.jna	jna-platform	4.5.1
net.sf.ehcache	ehcache	2.10.4
net.sourceforge.htmlunit	htmlunit	2.29
net.sourceforge.jtds	jtds	1.3.1
net.sourceforge.nekohtml	nekohtml	22年9月1日
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect	2.3.0
org.apache.activemq	activemq-amqp	5.15.3
org.apache.activemq	activemq-blueprint	5.15.3
org.apache.activemq	activemq-broker	5.15.3
org.apache.activemq	activemq-camel	5.15.3
org.apache.activemq	activemq-client	5.15.3
org.apache.activemq	activemq-console	5.15.3
org.apache.activemq	activemq-http	5.15.3
org.apache.activemq	activemq-jaas	5.15.3
org.apache.activemq	activemq-jdbc-store	5.15.3
org.apache.activemq	activemq-jms-pool	5.15.3
org.apache.activemq	activemq-kahadb-store	5.15.3
org.apache.activemq	activemq-karaf	5.15.3
org.apache.activemq	activemq-leveledb-store	5.15.3
org.apache.activemq	activemq-log4j-appender	5.15.3
org.apache.activemq	activemq-mqtt	5.15.3
org.apache.activemq	activemq-openwire-generator	5.15.3
org.apache.activemq	activemq-openwire-legacy	5.15.3
org.apache.activemq	activemq-osgi	5.15.3
org.apache.activemq	activemq-partition	5.15.3

组ID	工件ID	版
org.apache.activemq	activemq-pool	5.15.3
org.apache.activemq	activemq-ra	5.15.3
org.apache.activemq	activemq-run	5.15.3
org.apache.activemq	activemq-runtime-config	5.15.3
org.apache.activemq	activemq-shiro	5.15.3
org.apache.activemq	activemq-spring	5.15.3
org.apache.activemq	activemq-stomp	5.15.3
org.apache.activemq	activemq-web	5.15.3
org.apache.activemq	artemis-amqp-protocol	2.4.0
org.apache.activemq	artemis-commons	2.4.0
org.apache.activemq	artemis-core-client	2.4.0
org.apache.activemq	artemis-jms-client	2.4.0
org.apache.activemq	artemis-jms-server	2.4.0
org.apache.activemq	artemis-journal	2.4.0
org.apache.activemq	artemis-native	2.4.0
org.apache.activemq	artemis-selector	2.4.0
org.apache.activemq	artemis-server	2.4.0
org.apache.activemq	artemis-service-extensions	2.4.0
org.apache.commons	commons-dbcp2	2.2.0
org.apache.commons	commons-lang3	3.7
org.apache.commons	commons-pool2	2.5.0
org.apache.derby	derby	10.14.1.0
org.apache.httpcomponents	fluent-hc	4.5.5
org.apache.httpcomponents	httpasyncclient	4.1.3
org.apache.httpcomponents	httpclient	4.5.5
org.apache.httpcomponents	httpclient-cache	4.5.5
org.apache.httpcomponents	httpclient-osgi	4.5.5
org.apache.httpcomponents	httpclient-win	4.5.5
org.apache.httpcomponents	httpcore	4.4.9

组ID	工件ID	版
org.apache.httpcomponents	httpcore-nio	4.4.9
org.apache.httpcomponents	httpmime	4.5.5
org.apache.johnzon	johnzon-jsonb	1.1.6
org.apache.logging.log4j	log4j-1.2-api	2.10.0
org.apache.logging.log4j	log4j-api	2.10.0
org.apache.logging.log4j	log4j-cassandra	2.10.0
org.apache.logging.log4j	log4j-core	2.10.0
org.apache.logging.log4j	log4j-couchdb	2.10.0
org.apache.logging.log4j	log4j-flume-ng	2.10.0
org.apache.logging.log4j	log4j-iostreams	2.10.0
org.apache.logging.log4j	log4j-jcl	2.10.0
org.apache.logging.log4j	log4j-jmx-gui	2.10.0
org.apache.logging.log4j	log4j-jul	2.10.0
org.apache.logging.log4j	log4j-liquibase	2.10.0
org.apache.logging.log4j	log4j-mongodb	2.10.0
org.apache.logging.log4j	log4j-slf4j-impl	2.10.0
org.apache.logging.log4j	log4j-taglib	2.10.0
org.apache.logging.log4j	log4j-to-slf4j	2.10.0
org.apache.logging.log4j	log4j-web	2.10.0
org.apache.solr	solr-analysis-extras	6.6.2
org.apache.solr	solr-analytics	6.6.2
org.apache.solr	solr-cell	6.6.2
org.apache.solr	solr-clustering	6.6.2
org.apache.solr	solr-core	6.6.2
org.apache.solr	solr-dataimporthandler	6.6.2
org.apache.solr	solr-dataimporthandler-extras	6.6.2
org.apache.solr	solr-langid	6.6.2
org.apache.solr	solr-solrj	6.6.2
org.apache.solr	solr-test-framework	6.6.2

组ID	工件ID	版
org.apache.solr	solr-uima	6.6.2
org.apache.solr	solr-velocity	6.6.2
org.apache.tomcat	tomcat-annotations-api	8.5.28
org.apache.tomcat	tomcat-catalina-jmx-remote	8.5.28
org.apache.tomcat	tomcat-jdbc	8.5.28
org.apache.tomcat	tomcat-jsp-api	8.5.28
org.apache.tomcat.embed	tomcat-embed-core	8.5.28
org.apache.tomcat.embed	tomcat-embed-el	8.5.28
org.apache.tomcat.embed	tomcat-embed-jasper	8.5.28
org.apache.tomcat.embed	tomcat-embed-websocket	8.5.28
org.aspectj	aspectjrt	1.8.13
org.aspectj	aspectjtools	1.8.13
org.aspectj	aspectjweaver	1.8.13
org.assertj	assertj-core	3.9.1
org.codehaus.btm	btm	2.1.4
org.codehaus.groovy	groovy	2.4.14
org.codehaus.groovy	groovy-all	2.4.14
org.codehaus.groovy	groovy-ant	2.4.14
org.codehaus.groovy	groovy-bsf	2.4.14
org.codehaus.groovy	groovy-console	2.4.14
org.codehaus.groovy	groovy-docgenerator	2.4.14
org.codehaus.groovy	groovy-groovydoc	2.4.14
org.codehaus.groovy	groovy-groovysh	2.4.14
org.codehaus.groovy	groovy-jmx	2.4.14
org.codehaus.groovy	groovy-json	2.4.14
org.codehaus.groovy	groovy-jsr223	2.4.14
org.codehaus.groovy	groovy-nio	2.4.14
org.codehaus.groovy	groovy-servlet	2.4.14
org.codehaus.groovy	groovy-sql	2.4.14

组ID	工件ID	版
org.codehaus.groovy	groovy-swing	2.4.14
org.codehaus.groovy	groovy-templates	2.4.14
org.codehaus.groovy	groovy-test	2.4.14
org.codehaus.groovy	groovy-testng	2.4.14
org.codehaus.groovy	groovy-xml	2.4.14
org.codehaus.janino	janino	3.0.8
org.eclipse.jetty	apache-jsp	9.4.8.v20171121
org.eclipse.jetty	apache-jstl	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-client	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-conscrypt-client	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-conscrypt-server	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-java-client	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-java-server	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-openjdk8-client	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-openjdk8-server	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-server	9.4.8.v20171121
org.eclipse.jetty	jetty-annotations	9.4.8.v20171121
org.eclipse.jetty	jetty-ant	9.4.8.v20171121
org.eclipse.jetty	jetty-client	9.4.8.v20171121
org.eclipse.jetty	jetty-continuation	9.4.8.v20171121
org.eclipse.jetty	jetty-deploy	9.4.8.v20171121
org.eclipse.jetty	jetty-distribution	9.4.8.v20171121
org.eclipse.jetty	jetty-hazelcast	9.4.8.v20171121
org.eclipse.jetty	jetty-home	9.4.8.v20171121
org.eclipse.jetty	jetty-http	9.4.8.v20171121
org.eclipse.jetty	jetty-http-spi	9.4.8.v20171121
org.eclipse.jetty	jetty-infinispan	9.4.8.v20171121
org.eclipse.jetty	jetty-io	9.4.8.v20171121
org.eclipse.jetty	jetty-jaas	9.4.8.v20171121

组ID	工件ID	版
org.eclipse.jetty	jetty-jaspi	9.4.8.v20171121
org.eclipse.jetty	jetty-jmx	9.4.8.v20171121
org.eclipse.jetty	jetty-jndi	9.4.8.v20171121
org.eclipse.jetty	jetty-nosql	9.4.8.v20171121
org.eclipse.jetty	jetty-plus	9.4.8.v20171121
org.eclipse.jetty	jetty-proxy	9.4.8.v20171121
org.eclipse.jetty	jetty-quickstart	9.4.8.v20171121
org.eclipse.jetty	jetty-rewrite	9.4.8.v20171121
org.eclipse.jetty	jetty-security	9.4.8.v20171121
org.eclipse.jetty	jetty-server	9.4.8.v20171121
org.eclipse.jetty	jetty-servlet	9.4.8.v20171121
org.eclipse.jetty	jetty-servlets	9.4.8.v20171121
org.eclipse.jetty	jetty-spring	9.4.8.v20171121
org.eclipse.jetty	jetty-unixsocket	9.4.8.v20171121
org.eclipse.jetty	jetty-util	9.4.8.v20171121
org.eclipse.jetty	jetty-util-ajax	9.4.8.v20171121
org.eclipse.jetty	jetty-webapp	9.4.8.v20171121
org.eclipse.jetty	jetty-xml	9.4.8.v20171121
org.eclipse.jetty.cdi	cdi-core	9.4.8.v20171121
org.eclipse.jetty.cdi	cdi-full-servlet	9.4.8.v20171121
org.eclipse.jetty.cdi	cdi-servlet	9.4.8.v20171121
org.eclipse.jetty.fcg	fcgi-client	9.4.8.v20171121
org.eclipse.jetty.fcg	fcgi-server	9.4.8.v20171121
org.eclipse.jetty.gcloud	jetty-gcloud-session-manager	9.4.8.v20171121
org.eclipse.jetty.http2	http2-client	9.4.8.v20171121
org.eclipse.jetty.http2	http2-common	9.4.8.v20171121
org.eclipse.jetty.http2	http2-hpack	9.4.8.v20171121
org.eclipse.jetty.http2	http2-http-client-transport	9.4.8.v20171121
org.eclipse.jetty.http2	http2-server	9.4.8.v20171121

组ID	工件ID	版
org.eclipse.jetty.memcached	jetty-memcached-sessions	9.4.8.v20171121
org.eclipse.jetty.orbit	javax.servlet.jsp	2.2.0.v201112011158
org.eclipse.jetty.osgi	jetty-httpservice	9.4.8.v20171121
org.eclipse.jetty.osgi	jetty-osgi-boot	9.4.8.v20171121
org.eclipse.jetty.osgi	jetty-osgi-boot-jsp	9.4.8.v20171121
org.eclipse.jetty.osgi	jetty-osgi-boot-warurl	9.4.8.v20171121
org.eclipse.jetty.websocket	javax-websocket-client-impl	9.4.8.v20171121
org.eclipse.jetty.websocket	javax-websocket-server-impl	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-api	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-client	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-common	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-server	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-servlet	9.4.8.v20171121
org.ehcache	ehcache	3.5.0
org.ehcache	ehcache-clustered	3.5.0
org.ehcache	ehcache-transactions	3.5.0
org.elasticsearch	elasticsearch	5.6.8
org.elasticsearch.client	transport	5.6.8
org.elasticsearch.plugin	transport-netty4-client	5.6.8
org.firebirdsql.jdbc	jaybird-jdk17	3.0.3
org.firebirdsql.jdbc	jaybird-jdk18	3.0.3
org.flywaydb	flyway-core	5.0.7
org.freemarker	freemarker	2.3.27-孵化
org.glassfish	javax.el	3.0.0
org.glassfish.jersey.containers	jersey-container-servlet	2.26
org.glassfish.jersey.containers	jersey-container-servlet-core	2.26
org.glassfish.jersey.core	jersey-client	2.26
org.glassfish.jersey.core	jersey-common	2.26
org.glassfish.jersey.core	jersey-server	2.26

组ID	工件ID	版
org.glassfish.jersey.ext	jersey-bean-validation	2.26
org.glassfish.jersey.ext	jersey-entity-filtering	2.26
org.glassfish.jersey.ext	jersey-spring4	2.26
org.glassfish.jersey.media	jersey-media-jaxb	2.26
org.glassfish.jersey.media	jersey-media-json-jackson	2.26
org.glassfish.jersey.media	jersey-media-multipart	2.26
org.hamcrest	hamcrest-core	1.3
org.hamcrest	hamcrest-library	1.3
org.hibernate	hibernate-c3p0	5.2.14.Final
org.hibernate	hibernate-core	5.2.14.Final
org.hibernate	hibernate-ehcache	5.2.14.Final
org.hibernate	hibernate-entitymanager	5.2.14.Final
org.hibernate	hibernate-envers	5.2.14.Final
org.hibernate	hibernate-hikaricp	5.2.14.Final
org.hibernate	hibernate-infinispan	5.2.14.Final
org.hibernate	hibernate-java8	5.2.14.Final
org.hibernate	hibernate-jcache	5.2.14.Final
org.hibernate	hibernate-jpamodelgen	5.2.14.Final
org.hibernate	hibernate-proxool	5.2.14.Final
org.hibernate	hibernate-spatial	5.2.14.Final
org.hibernate	hibernate-testing	5.2.14.Final
org.hibernate.validator	hibernate-validator	6.0.7.Final
org.hibernate.validator	hibernate-validator-annotation-processor	6.0.7.Final
org.hsqldb	hsqldb	2.4.0
org.infinispan	infinispan-jcache	9.1.6.Final
org.infinispan	infinispan-spring4-common	9.1.6.Final
org.infinispan	infinispan-spring4-embedded	9.1.6.Final
org.influxdb	influxdb-java	2.9
org.jboss	jboss-transaction-spi	7.6.0.Final

组ID	工件ID	版
org.jboss.logging	jboss-logging	3.3.2.Final
org.jboss.narayana.jta	jdbc	5.8.0.Final
org.jboss.narayana.jta	jms	5.8.0.Final
org.jboss.narayana.jta	jta	5.8.0.Final
org.jboss.narayana.jts	narayana-jts-integration	5.8.0.Final
org.jdom	jdom2	2.0.6
org.jetbrains.kotlin	kotlin-reflect	1.2.30
org.jetbrains.kotlin	kotlin-runtime	1.2.30
org.jetbrains.kotlin	kotlin-stdlib	1.2.30
org.jetbrains.kotlin	kotlin-stdlib-jdk7	1.2.30
org.jetbrains.kotlin	kotlin-stdlib-jdk8	1.2.30
org.jetbrains.kotlin	kotlin-stdlib-jre7	1.2.30
org.jetbrains.kotlin	kotlin-stdlib-jre8	1.2.30
org.jolokia	jolokia-core	1.5.0
org.jooq	jooq	3.10.5
org.jooq	jooq-codegen	3.10.5
org.jooq	jooq-meta	3.10.5
org.junit.jupiter	junit-jupiter-api	5.1.0
org.junit.jupiter	junit-jupiter-engine	5.1.0
org.liquibase	liquibase-core	3.5.5
org.mariadb.jdbc	mariadb-java-client	2.2.2
org.mockito	mockito-core	2.15.0
org.mockito	mockito-inline	2.15.0
org.mongodb	bson	3.6.3
org.mongodb	mongodb-driver	3.6.3
org.mongodb	mongodb-driver-async	3.6.3
org.mongodb	mongodb-driver-core	3.6.3
org.mongodb	mongodb-driver-reactivestreams	1.7.1
org.mongodb	mongo-java-driver	3.6.3

组ID	工件ID	版
org.mortbay.jasper	apache-el	8.5.24.2
org.neo4j	neo4j-ogm-api	3.1.0
org.neo4j	neo4j-ogm-bolt-driver	3.1.0
org.neo4j	neo4j-ogm-core	3.1.0
org.neo4j	neo4j-ogm-http-driver	3.1.0
org.postgresql	postgresql	42.2.1
org.projectlombok	lombok	20年1月16日
org.quartz-scheduler	quartz	2.3.0
org.reactivestreams	reactive-streams	1.0.2
org.seleniumhq.selenium	htmlunit-driver	2.29.2
org.seleniumhq.selenium	selenium-api	3.9.1
org.seleniumhq.selenium	selenium-chrome-driver	3.9.1
org.seleniumhq.selenium	selenium-edge-driver	3.9.1
org.seleniumhq.selenium	selenium-firefox-driver	3.9.1
org.seleniumhq.selenium	selenium-ie-driver	3.9.1
org.seleniumhq.selenium	selenium-java	3.9.1
org.seleniumhq.selenium	selenium-opera-driver	3.9.1
org.seleniumhq.selenium	selenium-remote-driver	3.9.1
org.seleniumhq.selenium	selenium-safari-driver	3.9.1
org.seleniumhq.selenium	selenium-support	3.9.1
org.skyscreamer	jsonassert	1.5.0
org.slf4j	jcl-over-slf4j	1.7.25
org.slf4j	jul-to-slf4j	1.7.25
org.slf4j	log4j-over-slf4j	1.7.25
org.slf4j	slf4j-api	1.7.25
org.slf4j	slf4j-ext	1.7.25
org.slf4j	slf4j-jcl	1.7.25
org.slf4j	slf4j-jdk14	1.7.25
org.slf4j	slf4j-log4j12	1.7.25

组ID	工件ID	版
org.slf4j	slf4j-nop	1.7.25
org.slf4j	slf4j-simple	1.7.25
org.springframework	spring-aop	5.0.5.BUILD-快照
org.springframework	spring-aspects	5.0.5.BUILD-快照
org.springframework	spring-beans	5.0.5.BUILD-快照
org.springframework	spring-context	5.0.5.BUILD-快照
org.springframework	spring-context-indexer	5.0.5.BUILD-快照
org.springframework	spring-context-support	5.0.5.BUILD-快照
org.springframework	spring-core	5.0.5.BUILD-快照
org.springframework	spring-expression	5.0.5.BUILD-快照
org.springframework	spring-instrument	5.0.5.BUILD-快照
org.springframework	spring-jcl	5.0.5.BUILD-快照
org.springframework	spring-jdbc	5.0.5.BUILD-快照
org.springframework	spring-jms	5.0.5.BUILD-快照
org.springframework	spring-messaging	5.0.5.BUILD-快照
org.springframework	spring-orm	5.0.5.BUILD-快照
org.springframework	spring-oxm	5.0.5.BUILD-快照
org.springframework	spring-test	5.0.5.BUILD-快照
org.springframework	spring-tx	5.0.5.BUILD-快照
org.springframework	spring-web	5.0.5.BUILD-快照
org.springframework	spring-webflux	5.0.5.BUILD-快照
org.springframework	spring-webmvc	5.0.5.BUILD-快照
org.springframework	springwebsocket	5.0.5.BUILD-快照
org.springframework.amqp	spring-amqp	2.0.2.RELEASE
org.springframework.amqp	spring-rabbit	2.0.2.RELEASE
org.springframework.batch	spring-batch-core	4.0.1.RELEASE
org.springframework.batch	spring-batch-infrastructure	4.0.1.RELEASE
org.springframework.batch	spring-batch-integration	4.0.1.RELEASE
org.springframework.batch	spring-batch-test	4.0.1.RELEASE

组ID	工件ID	版
org.springframework.boot	spring-boot	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-actuator	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-actuator-autoconfigure	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-autoconfigure	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-autoconfigure-processor	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-configuration-metadata	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-configuration-processor	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-devtools	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-loader	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-loader-tools	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-properties-migrator	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-activemq	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-actuator	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-amqp	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-aop	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-artemis	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-batch	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-cache	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-cloud-connectors	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-cassandra	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-cassandra-reactive	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-couchbase	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-couchbase-reactive	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-elasticsearch	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-jpa	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-ldap	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-mongodb	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-mongodb-reactive	2.0.1.BUILD-快照

组ID	工件ID	版
org.springframework.boot	spring-boot-starter-data-neo4j	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-redis	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-redis-reactive	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-rest	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-data-solr	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-freemarker	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-groovy-templates	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-hateoas	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-integration	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-jdbc	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-jersey	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-jetty	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-jooq	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-json	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-jta-atomikos	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-jta-bitronix	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-jta-narayana	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-log4j2	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-logging	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-mail	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-mustache	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-quartz	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-reactor-netty	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-security	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-test	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-thymeleaf	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-tomcat	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-undertow	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-validation	2.0.1.BUILD-快照

组ID	工件ID	版
org.springframework.boot	spring-boot-starter-web	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-webflux	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-web-services	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-starter-websocket	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-test	2.0.1.BUILD-快照
org.springframework.boot	spring-boot-test-autoconfigure	2.0.1.BUILD-快照
org.springframework.cloud	spring-cloud-cloudfoundry-connector	2.0.1.RELEASE
org.springframework.cloud	spring-cloud-connectors-core	2.0.1.RELEASE
org.springframework.cloud	spring-cloud-heroku-connector	2.0.1.RELEASE
org.springframework.cloud	spring-cloud-localconfig-connector	2.0.1.RELEASE
org.springframework.cloud	spring-cloud-spring-service-connector	2.0.1.RELEASE
org.springframework.data	spring-data-cassandra	2.0.5.RELEASE
org.springframework.data	spring-data-commons	2.0.5.RELEASE
org.springframework.data	spring-data-couchbase	3.0.5.RELEASE
org.springframework.data	spring-data-elasticsearch	3.0.5.RELEASE
org.springframework.data	spring-data-envers	2.0.5.RELEASE
org.springframework.data	spring-data-gemfire	2.0.5.RELEASE
org.springframework.data	spring-data-geode	2.0.5.RELEASE
org.springframework.data	spring-data-jpa	2.0.5.RELEASE
org.springframework.data	spring-data-keyvalue	2.0.5.RELEASE
org.springframework.data	spring-data-ldap	2.0.5.RELEASE
org.springframework.data	spring-data-mongodb	2.0.5.RELEASE
org.springframework.data	spring-data-mongodb-cross-store	2.0.5.RELEASE
org.springframework.data	spring-data-neo4j	5.0.5.RELEASE
org.springframework.data	spring-data-redis	2.0.5.RELEASE
org.springframework.data	spring-data-rest-core	3.0.5.RELEASE
org.springframework.data	spring-data-rest-hal-browser	3.0.5.RELEASE
org.springframework.data	spring-data-rest-webmvc	3.0.5.RELEASE
org.springframework.data	spring-data-solr	3.0.5.RELEASE

组ID	工件ID	版
org.springframework.hateoas	spring-hateoas	0.24.0.RELEASE
org.springframework.integration	spring-integration-amqp	5.0.3.RELEASE
org.springframework.integration	spring-integration-core	5.0.3.RELEASE
org.springframework.integration	spring-integration-event	5.0.3.RELEASE
org.springframework.integration	spring-integration-feed	5.0.3.RELEASE
org.springframework.integration	spring-integration-file	5.0.3.RELEASE
org.springframework.integration	spring-integration-ftp	5.0.3.RELEASE
org.springframework.integration	spring-integration-gemfire	5.0.3.RELEASE
org.springframework.integration	spring-integration-groovy	5.0.3.RELEASE
org.springframework.integration	spring-integration-http	5.0.3.RELEASE
org.springframework.integration	spring-integration-ip	5.0.3.RELEASE
org.springframework.integration	spring-integration-jdbc	5.0.3.RELEASE
org.springframework.integration	spring-integration-jms	5.0.3.RELEASE
org.springframework.integration	spring-integration-jmx	5.0.3.RELEASE
org.springframework.integration	spring-integration-jpa	5.0.3.RELEASE
org.springframework.integration	spring-integration-mail	5.0.3.RELEASE
org.springframework.integration	spring-integration-mongodb	5.0.3.RELEASE
org.springframework.integration	spring-integration-mqtt	5.0.3.RELEASE
org.springframework.integration	spring-integration-redis	5.0.3.RELEASE
org.springframework.integration	spring-integration-rmi	5.0.3.RELEASE
org.springframework.integration	spring-integration-scripting	5.0.3.RELEASE
org.springframework.integration	spring-integration-security	5.0.3.RELEASE
org.springframework.integration	spring-integration-sftp	5.0.3.RELEASE
org.springframework.integration	spring-integration-stomp	5.0.3.RELEASE
org.springframework.integration	spring-integration-stream	5.0.3.RELEASE
org.springframework.integration	spring-integration-syslog	5.0.3.RELEASE
org.springframework.integration	spring-integration-test	5.0.3.RELEASE
org.springframework.integration	spring-integration-test-support	5.0.3.RELEASE
org.springframework.integration	spring-integration-twitter	5.0.3.RELEASE

组ID	工件ID	版
org.springframework.integration	spring-integration-webflux	5.0.3.RELEASE
org.springframework.integration	spring-integration-websocket	5.0.3.RELEASE
org.springframework.integration	spring-integration-ws	5.0.3.RELEASE
org.springframework.integration	spring-integration-xml	5.0.3.RELEASE
org.springframework.integration	spring-integration-xmpp	5.0.3.RELEASE
org.springframework.integration	spring-integration-zookeeper	5.0.3.RELEASE
org.springframework.kafka	spring-kafka	2.1.4.RELEASE
org.springframework.kafka	spring-kafka-test	2.1.4.RELEASE
org.springframework.ldap	spring-ldap-core	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-core-tiger	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-ldif-batch	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-ldif-core	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-odm	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-test	2.3.2.RELEASE
org.springframework.plugin	spring-plugin-core	1.2.0.RELEASE
org.springframework.plugin	spring-plugin-metadata	1.2.0.RELEASE
org.springframework.restdocs	spring-restdocs-asciidoc	2.0.0.RELEASE
org.springframework.restdocs	spring-restdocs-core	2.0.0.RELEASE
org.springframework.restdocs	spring-restdocs-mockMvc	2.0.0.RELEASE
org.springframework.restdocs	spring-restdocs-restassured	2.0.0.RELEASE
org.springframework.restdocs	spring-restdocs-webtestclient	2.0.0.RELEASE
org.springframework.retry	spring-retry	1.2.2.RELEASE
org.springframework.security	spring-security-acl	5.0.3.RELEASE
org.springframework.security	spring-security-aspects	5.0.3.RELEASE
org.springframework.security	spring-security-cas	5.0.3.RELEASE
org.springframework.security	spring-security-config	5.0.3.RELEASE
org.springframework.security	spring-security-core	5.0.3.RELEASE
org.springframework.security	spring-security-crypto	5.0.3.RELEASE
org.springframework.security	spring-security-data	5.0.3.RELEASE

组ID	工件ID	版
org.springframework.security	spring-security-ldap	5.0.3.RELEASE
org.springframework.security	spring-security-messaging	5.0.3.RELEASE
org.springframework.security	spring-security-oauth2-client	5.0.3.RELEASE
org.springframework.security	spring-security-oauth2-core	5.0.3.RELEASE
org.springframework.security	spring-security-oauth2-jose	5.0.3.RELEASE
org.springframework.security	spring-security-openid	5.0.3.RELEASE
org.springframework.security	spring-security-remoting	5.0.3.RELEASE
org.springframework.security	spring-security-taglibs	5.0.3.RELEASE
org.springframework.security	spring-security-test	5.0.3.RELEASE
org.springframework.security	spring-security-web	5.0.3.RELEASE
org.springframework.session	spring-session-core	2.0.2.RELEASE
org.springframework.session	spring-session-data-mongodb	2.0.2.RELEASE
org.springframework.session	spring-session-data-redis	2.0.2.RELEASE
org.springframework.session	spring-session-hazelcast	2.0.2.RELEASE
org.springframework.session	spring-session-jdbc	2.0.2.RELEASE
org.springframework.ws	spring-ws-core	3.0.0.RELEASE
org.springframework.ws	spring-ws-security	3.0.0.RELEASE
org.springframework.ws	spring-ws-support	3.0.0.RELEASE
org.springframework.ws	spring-ws-test	3.0.0.RELEASE
org.synchronoss.cloud	nio-multipart-parser	1.1.0
org.thymeleaf	thymeleaf	3.0.9.RELEASE
org.thymeleaf	thymeleaf-spring5	3.0.9.RELEASE
org.thymeleaf.extras	thymeleaf-extras-java8time	3.0.1.RELEASE
org.thymeleaf.extras	thymeleaf-extras-springsecurity4	3.0.2.RELEASE
org.webjars	hal-browser	3325375
org.webjars	webjars-locator-core	0.35
org.xerial	sqlite-jdbc	3.21.0.1
org.xmlunit	xmlunit-core	2.5.1
org.xmlunit	xmlunit-legacy	2.5.1

组ID	工件ID	版
org.xmlunit	xmlunit-matchers	2.5.1
org.yaml	snakeyaml	1.19
redis.clients	jedis	2.9.0
wsdl4j	wsdl4j	1.6.3
xml-apis	xml-apis	1.4.01