

Spring Boot 中文文档

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

Table of Contents

1. Legal	2
2. Spring Boot 文档	3
2.1. 关于文档	3
2.2. 获取帮助	3
2.3. 从早期版本升级	4
2.4. 第一步	4
2.5. 使用 Spring Boot	4
2.6. 了解 Spring Boot 新特性	4
2.7. 生产环境	5
2.8. 高级内容	5
3. 入门	6
3.1. Spring Boot 简介	6
3.2. 系统要求	6
3.2.1. Servlet 容器	7
3.3. 安装 Spring Boot	7
3.3.1. 针对 Java 开发人员的安装说明	7
使用 Maven 安装	7
使用 Gradle 安装	8
3.3.2. 安装 Spring Boot CLI	8
手动安装	9
使用 SDKMAN! 安装	9
使用 OSX Homebrew 安装	10
使用 MacPorts 安装	10
命令行完成	10
Windows Scoop 安装	11
快速入门 Spring CLI 示例	11
3.3.3. 升级旧版 Spring Boot	12
3.4. 开发第一个 Spring Boot 应用	13
3.4.1. 创建 POM	13
3.4.2. 添加 Classpath 依赖	15
3.4.3. 编码	16
@RestController 与 @RequestMapping 注解	16
@EnableAutoConfiguration 注解	17
“main” 方法	17
3.4.4. 运行示例	17

3.4.5. 创建可执行 jar	18
3.5. 下一步	20
4. 使用 Spring Boot	21
4.1. 构建系统	21
4.1.1. 依赖管理	21
4.1.2. Maven	21
4.1.3. Gradle	22
4.1.4. Ant	22
4.1.5. Starters	23
4.2. 组织代码	29
4.2.1. 使用 “default” 包	29
4.2.2. 定位主应用类	29
4.3. 配置类	30
4.3.1. 导入额外的配置类	31
4.3.2. 导入 XML 配置	31
4.4. 自动配置	31
4.4.1. 平滑替换自动配置	31
4.4.2. 禁用指定的自动配置类	32
4.5. Spring Bean 与依赖注入	32
4.6. 使用 @SpringBootApplication 注解	33
4.7. 运行您的应用	35
4.7.1. 使用 IDE 运行	35
4.7.2. 作为打包应用运行	36
4.7.3. 使用 Maven 插件	36
4.7.4. 使用 Gradle 插件	37
4.7.5. 热交换	37
4.8. 开发者工具	37
4.8.1. Property 默认值	38
4.8.2. 自动重启	39
条件评估变更日志	41
排除资源	41
监视附加路径	42
禁用重启	42
使用触发文件	42
自定义重启类加载器	43
已知限制	44
4.8.3. LiveReload	44

4.8.4. 全局设置	45
配置文件系统监视器	46
4.8.5. 远程应用	46
运行远程客户端应用	47
远程更新	49
4.9. 打包生产应用	49
4.10. 下一步	49
5. Spring Boot 特性	50
5.1. SpringApplication	50
5.1.1. 启动失败	51
5.1.2. 延迟初始化	52
5.1.3. 自定义 banner	52
5.1.4. 自定义 SpringApplication	54
5.1.5. Fluent Builder API	54
5.1.6. 应用程序的可用性	55
Liveness State	55
Readiness State	56
管理应用程序可用性状态	56
5.1.7. 应用程序事件与监听器	58
5.1.8. Web 环境	59
5.1.9. 访问应用程序参数	60
5.1.10. 使用 ApplicationRunner 或 CommandLineRunner	61
5.1.11. 应用程序退出	61
5.1.12. 管理功能	62
5.1.13. Application Startup tracking	62
5.2. 外部化配置	63
5.2.1. 访问命令行属性	65
5.2.2. JSON Application Properties	66
5.2.3. 外部应用程序属性	66
Optional Locations	69
Wildcard Locations(通配符位置)	69
特定 Profile 的属性文件	70
导入其他数据	71
Importing Extensionless Files	72
使用配置树	72
属性中的占位符	74
处理多文档文件	75

Activation Properties	76
5.2.4. 加密属性	77
5.2.5. 使用 YAML	77
使用 YAML 代替属性文件	77
Directly Loading YAML	79
5.2.6. 配置随机值	79
5.2.7. 类型安全的配置属性	79
JavaBean 属性绑定	79
构造函数绑定	82
启用 <code>@ConfigurationProperties</code> 注解的类型	85
使用 <code>@ConfigurationProperties</code> 注解类型	86
第三方配置	87
宽松绑定	88
合并复杂类型	91
属性转换	93
<code>@ConfigurationProperties</code> 验证	99
<code>@ConfigurationProperties</code> 与 <code>@Value</code> 对比	101
5.3. Profiles	101
5.3.1. 添加激活 Profile	102
5.3.2. Profile 组	103
5.3.3. 以编程方式设置 Profile	103
5.3.4. 特定 Profile 的配置文件	103
5.4. 日志记录	103
5.4.1. 日志格式	104
5.4.2. 控制台输出	105
着色输出	105
5.4.3. 文件输出	106
5.4.4. File Rotation	107
5.4.5. 日志等级	108
5.4.6. 日志组	109
5.4.7. 使用日志 Shutdown 钩子	110
5.4.8. 自定义日志配置	110
5.4.9. Logback 扩展	113
特定 Profile 配置	114
环境属性	114
5.5. 国际化	115
5.6. JSON	116

5.6.1. Jackson	116
5.6.2. Gson	116
5.6.3. JSON-B	116
5.7. 开发 Web 应用程序	116
5.7.1. Spring Web MVC 框架	117
Spring MVC 自动配置	117
HttpMessageConverters	118
自定义 JSON Serializer 和 Deserializer	119
MessageCodesResolver	120
静态内容	120
欢迎页面	123
路径匹配与内容协商	123
ConfigurableWebBindingInitializer	125
模板引擎	125
错误处理	126
Spring HATEOAS	130
CORS 支持	130
5.7.2. Spring WebFlux 框架	131
Spring WebFlux 自动配置	133
使用 HttpMessageReader 和 HttpMessageWriter 作为 HTTP 编解码器	133
静态内容	134
欢迎页面	135
模板引擎	135
Error Handling	135
Web 过滤器	137
5.7.3. JAX-RS 与 Jersey	137
5.7.4. 内嵌 Servlet 容器支持	139
Servlets, Filters, 与 listeners	139
Servlet 上下文初始化	140
ServletWebServerApplicationContext	140
自定义内嵌 Servlet 容器	141
JSP 局限	143
5.7.5. 内嵌响应式服务器支持	143
5.7.6. 响应式服务器资源配置	143
5.8. Graceful shutdown	144
5.9. RSocket	144
5.9.1. RSocket策略自动配置	145

5.9.2. RSocket 服务器自动配置	145
5.9.3. Spring Messaging RSocket 支持	146
5.9.4. 使用 RSocketRequester 调用 RSocket 服务	146
5.10. 安全	147
5.10.1. MVC 安全	148
5.10.2. WebFlux 安全	148
5.10.3. OAuth2	149
客户端	149
资源服务器	152
授权服务器	153
5.10.4. SAML 2.0	153
依赖方	154
5.10.5. Actuator 安全	155
跨站请求伪造保护	155
5.11. 使用 SQL 数据库	155
5.11.1. 配置数据源	155
内嵌数据库支持	156
连接生产数据库	157
DataSource 配置	157
连接池支持	158
连接 JNDI 数据源	159
5.11.2. 使用 JdbcTemplate	159
5.11.3. JPA 与 Spring Data JPA	160
实体类	161
Spring Data JPA 资源库	163
创建和删除 JPA 数据库	164
在视图中打开 EntityManager	164
5.11.4. Spring Data JDBC	164
5.11.5. 使用 H2 的 Web 控制台	165
更改 H2 控制台的路径	165
5.11.6. 使用 jOOQ	165
代码生成	166
使用 DSLContext	166
jOOQ SQL 方言	167
自定义 jOOQ	167
5.11.7. 使用 R2DBC	168
嵌入式数据库支持	169

使用 DatabaseClient	170
Spring Data R2DBC Repositories	170
5.12. 使用 NoSQL 技术	171
5.12.1. Redis	172
连接 Redis	172
5.12.2. MongoDB	173
连接 MongoDB 数据库	173
MongoTemplate	175
Spring Data MongoDB Repositories	176
内嵌 Mongo	177
5.12.3. Neo4j	177
连接 Neo4j 数据库	177
Spring Data Neo4j 资源库	178
5.12.4. Solr	179
连接 Solr	180
Spring Data Solr 资源库	180
5.12.5. Elasticsearch	181
使用 REST 客户端连接 Elasticsearch	181
使用 Reactive REST 客户端连接	182
使用 Spring Data 连接 Elasticsearch	182
Spring Data Elasticsearch 资源库	183
5.12.6. Cassandra	184
连接 Cassandra	184
Spring Data Cassandra 资源库	185
5.12.7. Couchbase	186
连接 Couchbase	186
Spring Data Couchbase 资源库	187
5.12.8. LDAP	188
连接 LDAP 服务器	188
Spring Data LDAP 资源库	189
内嵌内存式 LDAP 服务器	189
5.12.9. InfluxDB	190
连接 InfluxDB	190
5.13. 缓存	191
5.13.1. 支持的缓存提供者	192
Generic	193
JCache (JSR-107)	193

EhCache 2.x	195
Hazelcast	195
Infinispan	195
Couchbase	195
Redis	196
Caffeine	197
Simple	198
None	198
5.14. 消息传递	199
5.14.1. JMS	199
ActiveMQ 支持	199
ActiveMQ Artemis 支持	200
使用 JNDI ConnectionFactory	202
发送消息	202
接收消息	203
5.14.2. AMQP	204
RabbitMQ 支持	204
发送消息	205
接收消息	207
5.14.3. Apache Kafka 支持	209
发送消息	209
接收消息	210
Kafka Streams	210
其他 Kafka 属性	211
使用嵌入式 Kafka 进行测试	213
5.15. 使用 RestTemplate 调用 REST 服务	214
5.15.1. 自定义 RestTemplate	215
5.16. 使用 WebClient 调用 REST 服务	216
5.16.1. WebClient 运行时	217
5.16.2. 自定义 WebClient	218
5.17. 验证	218
5.18. 发送邮件	219
5.19. JTA 分布式事务	219
5.19.1. 使用 Atomikos 事务管理器	220
5.19.2. 使用 Bitronix 事务管理器	220
5.19.3. 使用 Java EE 管理的事务管理器	221
5.19.4. 混合使用 XA 与非 XA JMS 连接	221

5.19.5. 支持嵌入式事务管理器	222
5.20. Hazelcast	222
5.21. Quartz 调度器	223
5.22. 任务执行与调度	225
5.23. Spring Integration	226
5.24. Spring Session	228
5.25. 通过 JMX 监控和管理	229
5.26. 测试	229
5.26.1. 依赖范围测试	230
5.26.2. 测试 Spring 应用程序	231
5.26.3. 测试 Spring Boot 应用程序	231
检测 Web 应用程序类型	232
检测测试配置	233
排除测试配置	233
使用应用程序参数	234
在模拟环境中进行测试	234
使用正在运行的服务器进行测试	236
自定义 WebTestClient	238
使用 JMX	238
Using Metrics	238
模拟和检测 Bean	238
自动配置测试	240
自动配置的 JSON 测试	241
自动配置的 Spring MVC 测试	243
自动配置的 Spring WebFlux 测试	246
自动配置的 Data JPA 测试	248
自动配置的 JDBC 测试	249
自动配置的 Data JDBC 测试	250
自动配置的 jOOQ Tests	251
自动配置的 Data MongoDB 测试	251
自动配置的 Data Neo4j 测试	252
自动配置的 Data Redis 测试	253
自动配置的 Data LDAP 测试	254
自动配置的 REST Clients	255
自动配置的 Spring REST Docs 测试	256
自动配置的 Spring Web Services 测试	261
其他的自动配置和切片	262

用户配置和切片	263
使用 Spock 测试 Spring Boot 应用程序	264
5.26.4. 测试实用工具	265
ConfigFileApplicationContextInitializer	265
TestPropertyValues	265
OutputCapture	265
TestRestTemplate	266
5.27. WebSockets	268
5.28. Web Services	269
5.28.1. 使用 WebServiceTemplate 调用 Web Service	269
5.29. 创建自己的自动配置	270
5.29.1. 理解 自动配置的 Beans	270
5.29.2. 找到候选的自动配置	270
5.29.3. 条件注解	271
类条件	272
Bean 条件	273
属性条件	274
资源条件	274
Web 应用程序条件	274
SpEL 表达式条件	274
5.29.4. 测试自动配置	274
模拟一个 Web 上下文	276
覆盖 Classpath	276
5.29.5. 创建自己的 Starter	277
命名	277
配置 keys	278
autoconfigure 模块	279
Starter 模块	280
5.30. Kotlin 支持	281
5.30.1. 要求	281
5.30.2. Null 安全	282
5.30.3. Kotlin API	282
runApplication	282
扩展	283
5.30.4. 依赖管理	283
5.30.5. <code>@ConfigurationProperties</code>	283
5.30.6. 测试	284
5.30.7. 资源	285

进阶阅读	285
示例	285
5.31. 容器镜像	285
5.31.1. 分层 Docker 镜像	286
5.31.2. 构建容器镜像	287
Dockerfiles	287
Cloud Native Buildpacks	288
5.32. 下一步	289
6. Spring Boot Actuator: 生产就绪功能	290
6.1. 启用生产就绪功能	290
6.2. 端点	291
6.2.1. 启用端点	293
6.2.2. 暴露端点	293
6.2.3. 保护 HTTP 端点	295
6.2.4. 配置端点	297
6.2.5. Actuator Web 端点超媒体	297
6.2.6. 跨域支持	297
6.2.7. 实现自定义端点	298
接收输入	299
自定义 Web 端点	300
Servlet 端点	301
Controller 端点	302
6.2.8. 健康信息	302
自动配置的 HealthIndicators	303
编写自定义 HealthIndicators	304
响应式健康指示器	306
自动配置的 ReactiveHealthIndicators	307
Health 组	308
6.2.9. Kubernetes Probes	309
使用 Kubernetes 探针检查外部状态	310
应用程序生命周期和探针状态	312
6.2.10. 应用程序信息	313
自动配置的 InfoContributors	313
自定义应用程序信息	314
Git 提交信息	314
构建信息	315
编写自定义 InfoContributors	315
6.3. 通过 HTTP 监控和管理	316

6.3.1. 自定义 Management 端点路径	317
6.3.2. 自定义 Management 服务器端口	317
6.3.3. 配置 Management 的 SSL	318
6.3.4. 配置 Management 服务器地址	319
6.3.5. 禁用 HTTP 端点	319
6.4. 通过 JMX 监控和管理	320
6.4.1. 自定义 MBean 名称	320
6.4.2. 禁用 JMX 端点	320
6.4.3. 通过 HTTP 使用 Jolokia 访问 JMX	321
自定义 Jolokia	321
禁用 Jolokia	322
6.5. 日志记录器	322
6.5.1. 配置一个日志记录器	322
6.6. 指标	323
6.6.1. 入门	324
6.6.2. 支持的监控系统	325
AppOptics	325
Atlas	325
Datadog	326
Dynatrace	326
Elastic	327
Ganglia	327
Graphite	327
Humio	328
Influx	328
JMX	329
KairosDB	329
New Relic	330
Prometheus	331
SignalFx	331
Simple	332
Stackdriver	332
StatsD	333
Wavefront	333
6.6.3. 支持的指标	334
Spring MVC 指标	335
Spring WebFlux 指标	336

Jersey Server 指标	337
HTTP Client 指标	338
Cache 指标	339
DataSource 指标	339
Hibernate 指标	340
RabbitMQ 指标	340
Kafka Metrics	340
6.6.4. 注册自定义指标	340
6.6.5. 自定义单个指标	341
常用标签	342
Per-meter 属性	342
6.6.6. 指标端点	343
6.7. 审计	344
6.7.1. Custom Auditing	344
6.8. HTTP 追踪	344
6.8.1. 自定义 HTTP 追踪	345
6.9. 进程监控	345
6.9.1. 扩展配置	345
6.9.2. 编程方式	345
6.10. Cloud Foundry 支持	346
6.10.1. 禁用 Cloud Foundry Actuator 扩展支持	346
6.10.2. Cloud Foundry 自签名证书	346
6.10.3. 自定义上下文路径	347
6.11. 下一步	348
7. 部署 Spring Boot 应用程序	350
7.1. 打包成容器	350
7.2. 部署到云端	350
7.2.1. Cloud Foundry	351
绑定到服务	353
7.2.2. Kubernetes	354
Kubernetes 容器生命周期	354
7.2.3. Heroku	355
7.2.4. OpenShift	357
7.2.5. Amazon Web Services (AWS)	357
AWS Elastic Beanstalk	357
简介	358
7.2.6. Boxfuse 和 Amazon Web Services	359

7.2.7. Google Cloud	360
7.3. 安装 Spring Boot 应用程序	361
7.3.1. 支持的操作系统	362
7.3.2. Unix/Linux 服务	363
作为 init.d 服务安装 (系统V)	363
作为 systemd 服务安装	365
自定义启动脚本	366
7.3.3. Microsoft Windows 服务	371
7.4. 下一步	371
8. Spring Boot CLI	372
8.1. 安装 CLI	372
8.2. 使用 CLI	372
8.2.1. 使用 CLI 运行应用程序	373
推测 “grab” 依赖	374
推测 “grab” 坐标	375
默认导入语句	375
自动创建 Main 方法	376
自定义依赖管理	376
8.2.2. 具有多个源文件的应用程序	376
8.2.3. 打包你的应用程序	377
8.2.4. 初始化新项目	377
8.2.5. 使用嵌入式 shell	378
8.2.6. 将扩展添加到CLI	379
8.3. 使用Groovy Beans DSL开发应用程序	379
8.4. 使用 settings.xml 配置CLI	380
8.5. 下一步	381
9. 构建工具插件	382
9.1. Spring Boot Maven 插件	382
9.2. Spring Boot Gradle 插件	382
9.3. Spring Boot AntLib 模块	382
9.3.1. Spring Boot Ant 任务	383
使用 “exejar” Task	383
例子	384
9.3.2. 使用 “findmainclass” Task	384
例子	385
9.4. 支持其他构建系统	385
9.4.1. 重新打包 Archives	385

9.4.2. 嵌套库	385
9.4.3. 查找 Main Class	386
9.4.4. 重新打包示例实现	386
9.5. 接下来阅读什么	386
10. “使用方法” 指南	387
10.1. Spring Boot 应用程序	387
10.1.1. 创建自己的FailureAnalyzer	387
10.1.2. 自动配置故障排除	388
10.1.3. 启动之前自定义环境或ApplicationContext	388
10.1.4. 建立 ApplicationContext 层次结构（添加父上下文或根上下文）	391
10.1.5. 创建一个非 Web 应用程序	391
10.2. 属性和配置	391
10.2.1. 在构建时自动扩展属性	391
使用 Maven 自动扩展属性	391
使用Gradle自动扩展属性	393
10.2.2. 外部化配置 SpringApplication	393
10.2.3. 更改应用程序外部属性的位置	394
10.2.4. 使用 ‘Short’ 命令行参数	395
10.2.5. 对外部属性使用 YAML	395
10.2.6. 设置 Active Spring Profiles	396
10.2.7. 根据环境更改配置	397
10.2.8. 发现外部属性的内置选项	398
10.3. 嵌入式 Web 服务器	398
10.3.1. 使用其他 Web 服务器	398
10.3.2. 禁用 Web 服务器	400
10.3.3. 更改 HTTP 端口	400
10.3.4. 使用随机未分配的 HTTP 端口	401
10.3.5. 在运行时发现 HTTP 端口	401
10.3.6. 启用 HTTP 响应压缩	401
10.3.7. 配置 SSL	402
10.3.8. 配置 HTTP/2	403
Tomcat HTTP/2	403
Jetty HTTP/2	404
Reactor Netty HTTP/2	404
HTTP/2 with Undertow	404
HTTP/2 Cleartext with supported servers	404
10.3.9. 配置 Web 服务器	406

10.3.10. 将 <code>Servlet</code> , <code>Filter</code> , <code>Listener</code> 添加到应用程序	407
使用 <code>Spring Bean</code> 添加 <code>Servlet</code> , 过滤器或监听器	407
使用类路径扫描添加 <code>Servlet</code> , 过滤器和监听器	408
10.3.11. 配置访问日志	408
10.3.12. 在前端代理服务器后面运行	409
自定义 <code>Tomcat</code> 的代理配置	410
10.3.13. 使用 <code>Tomcat</code> 启用多个连接器	411
10.3.14. 使用 <code>Tomcat</code> 的 <code>LegacyCookieProcessor</code>	412
10.3.15. 启用 <code>Tomcat</code> 的 <code>MBean</code> 注册表	413
10.3.16. 使用 <code>Undertow</code> 启用多个监听器	413
10.3.17. 使用 <code>@ServerEndpoint</code> 创建 <code>WebSocket</code> 端点	414
10.4. Spring MVC	414
10.4.1. 编写 JSON REST 服务	414
10.4.2. 编写 XML REST 服务	415
10.4.3. 自定义 <code>Jackson ObjectMapper</code>	415
10.4.4. 自定义 <code>@ResponseBody</code> 渲染	417
10.4.5. 处理分段文件上传	418
10.4.6. 关闭 <code>Spring MVC DispatcherServlet</code>	418
10.4.7. 关闭默认的 MVC 配置	419
10.4.8. 自定义 <code>ViewResolvers</code>	419
10.5. 使用 <code>Spring Security</code> 进行测试	421
10.6. Jersey	421
10.6.1. 使用 <code>Spring Security</code> 保护 Jersey 端点	421
10.6.2. 与另一个 Web 框架一起使用 Jersey	422
10.7. HTTP Clients	422
10.7.1. 配置 <code>RestTemplate</code> 使用代理	422
10.7.2. 配置基于 <code>Reactor Netty</code> 的 <code>WebClient</code> 使用的 <code>TcpClient</code> . .	423
10.8. Logging	424
10.8.1. 配置 <code>Logback Logging</code>	425
配置仅文件输出的 <code>Logback</code>	426
10.8.2. 配置 <code>Log4j</code> 日志	427
使用 <code>YAML</code> 或 <code>JSON</code> 配置 <code>Log4j 2</code>	429
10.9. 数据访问	429
10.9.1. 配置自定义数据源	429
10.9.2. 配置两个数据源	433
10.9.3. 使用 <code>Spring Data Repositories</code>	435
10.9.4. 将 <code>@Entity</code> 定义与 <code>Spring</code> 配置分开	436

10.9.5. 配置 JPA 属性	436
10.9.6. 配置 Hibernate 命名策略	437
10.9.7. 配置 Hibernate 二级缓存	438
10.9.8. 在 Hibernate 组件中使用依赖注入	439
10.9.9. 使用自定义 EntityManagerFactory	439
10.9.10. 使用两个 EntityManager	439
10.9.11. 使用传统的 persistence.xml 文件	441
10.9.12. 使用 Spring Data JPA 和 Mongo 存储库	441
10.9.13. 定制 Spring Data 的 Web 支持	442
10.9.14. 将 Spring Data Repositories 暴露为 REST 端点	442
10.9.15. 配置 JPA 使用的组件	442
10.9.16. 使用两个数据源配置 jOOQ	443
10.10. 初始化 Database	443
10.10.1. 使用 JPA 初始化数据库	443
10.10.2. 使用 Hibernate 初始化数据库	444
10.10.3. 使用 SQL 脚本初始化数据库	444
10.10.4. 使用 R2DBC 初始化数据库	445
10.10.5. 初始化一个 Spring Batch 数据库	446
10.10.6. 使用高级数据库迁移工具	446
在启动时执行 Flyway 数据库迁移	447
在启动时执行 Liquibase 数据库迁移	448
10.11. 消息	449
10.11.1. 禁用事务 JMS 会话	449
10.12. Batch Applications	449
10.12.1. 指定批处理数据源	450
10.12.2. 在启动时执行 Spring Batch 作业	450
10.12.3. 在命令行上运行	450
10.12.4. 存储作业库	451
10.13. Actuator	451
10.13.1. 更改 Actuator 端点的HTTP端口或地址	451
10.13.2. 自定义 ‘whitelabel’ 错误页面	451
10.13.3. Sanitize Sensitive Values	452
10.13.4. 映射健康状态	452
10.14. 安全	453
10.14.1. 关闭 Spring Boot 安全性配置	454
10.14.2. 更改UserDetailsService并添加用户帐户	454
10.14.3. 在代理服务器后运行时启用HTTPS	454

10.15. 热交换	455
10.15.1. 重新加载静态内容	455
10.15.2. 重新加载模板, 而无需重新启动容器	455
Thymeleaf 模板	455
FreeMarker 模板	456
Groovy 模板	456
10.15.3. 快速重启应用程序	456
10.15.4. 重新加载Java类而无需重新启动容器	456
10.16. 构建	456
10.16.1. 生成构建信息	456
10.16.2. 生成 Git 信息	457
10.16.3. 自定义依赖版本	458
10.16.4. 使用 Maven 创建可执行 JAR	459
10.16.5. 使用Spring Boot应用程序作为依赖	459
10.16.6. 运行可执行 jar 时提取特定的库	460
10.16.7. 创建带有排除项的不可执行的 JAR	461
10.16.8. 远程调试以 Maven 开头的 Spring Boot 应用程序	462
10.16.9. 在不使用 spring-boot-antlib 的情况下从Ant构建可执行归档文件	462
10.17. 传统部署	463
10.17.1. 创建可部署的 War 文件	464
10.17.2. 将现有应用程序转换为 Spring Boot	465
10.17.3. 将 WAR 部署到 WebLogic	468
10.17.4. 使用 Jedis 代替 Lettuce	469
10.17.5. 在集成测试中使用 Testcontainers	470
11. Appendices	472
Appendix A: 公共应用程序属性	472
11.A.1. Core Properties	472
11.A.2. Cache Properties	490
11.A.3. Mail Properties	493
11.A.4. JSON Properties	494
11.A.5. Data Properties	498
11.A.6. Transaction Properties	538
11.A.7. Data migration Properties	552
11.A.8. Integration Properties	565
11.A.9. Web Properties	602
11.A.10. Templating Properties	617
11.A.11. Server Properties	630

11.A.12. Security Properties	655
11.A.13. RSocket Properties	657
11.A.14. Actuator Properties	658
11.A.15. Devtools Properties	708
11.A.16. Testing Properties	711
Appendix B: 配置元数据	712
11.B.1. Metadata 格式	712
Group 属性	714
Property 属性	715
Hint 属性	717
重复的元数据项	718
11.B.2. 提供手动提示	718
Value Hint	719
Value Providers	720
11.B.3. 使用注解处理器生成您自己的元数据	726
配置注解处理器	726
自动生成元数据	727
嵌套属性	730
添加其他元数据	730
Appendix C: 自动配置类	731
11.C.1. spring-boot-autoconfigure	731
11.C.2. spring-boot-actuator-autoconfigure	736
Appendix D: 测试自动配置注解	740
11.D.1. 测试切片	740
Appendix E: 可执行Jar格式	753
11.E.1. 嵌套 JARs	753
可执行的 Jar 文件结构	754
可执行 War 文件结构	754
文件索引	755
Classpath Index	755
Layer Index(分层索引)	756
11.E.2. Spring Boot “JarFile” 类	756
与标准Java “JarFile” 的兼容性	757
11.E.3. 运行可执行 Jars	757
运行 Manifest	758
11.E.4. PropertiesLauncher 特性	758
11.E.5. 可执行 Jar 限制	760

11.E.6. 替代 单一 Jar 方案	761
Appendix F: 依赖版本	761
11.F.1. 管理依赖坐标	761
11.F.2. Version Properties	815

2.4.5

Chapter 1. Legal

Copyright © 2012-2021

本文档的副本可以供您自己使用，也可以分发给其他人，但前提是您不对此类副本收取任何费用，并且还应确保每份副本均包含本版权声明（无论是印刷版本还是电子版本）。

Chapter 2. Spring Boot 文档

本节简述了 Spring Boot 参考文档的内容，并且可作为文档的导航地图。

2.1. 关于文档

Spring Boot 参考指南可通过以下方式获得：

- [Multi-page HTML](#)
- [Single page HTML](#)
- [PDF](#)

最新的文档位于 docs.spring.io/spring-boot/docs/current/reference。

本文档的副本可以供您自己使用，也可以分发给其他人，但前提是您不对此类副本收取任何费用，并且还应确保每份副本均包含本版权声明（无论是印刷版本还是电子版本）。

2.2. 获取帮助

如果您在使用 Spring Boot 时遇到了麻烦，可参考以下指南。

- 尝试 阅读[How-to 文档](#) — 最常见问题的解决方案都在这里。.
- 学习 Spring 基础 — Spring Boot 是建立在多个 Spring 项目之上，请查看 spring.io 网站以获取更多参考文档。如果您是刚刚开始使用 Spring，请尝试其中一个 [guides](#)。
- 提提问题 — 我们时刻关注着 stackoverflow.com 上有关 [spring-boot](#) 标签相关的问题。
- 在 github.com/spring-projects/spring-boot/issues 报告 Spring Boot 的 bug。



Spring Boot 是全部开源的，包括文档！如果您发现文档中存在错误了，或者您想改进它们，请 [参与我们](#)。

2.3. 从早期版本升级

项目 [wiki](#) 提供了有关如何从早期版本的 Spring Boot 升级的说明。按照 [release notes](#) 说明部分中的链接查找要升级到的版本。

升级说明始终是发行版本说明中的第一项。如果您落后多个发行版本，请确保您还查看了所跳版本的发行版本说明。

您应该始终确保运行的是 [受支持的 Spring Boot 版本](#)。

2.4. 第一步

如果您是刚开始使用 Spring Boot，或者想对 Spring 有个大体 印象，您可以从 [这里开始学习！](#)

- [从零开始](#): [概述](#) | [要求](#) | [安装](#)
- [教程](#): [第 1 部分](#) | [第 2 部分](#)
- [运行您的例子](#): [第 1 部分](#) | [第 2 部分](#)

2.5. 使用 Spring Boot

准备开始使用 Spring Boot 了？[立即上手](#)

- [构建系统](#): [Maven](#) | [Gradle](#) | [Ant](#) | [Starters](#)
- [最佳实践](#): [代码结构](#) | [@Configuration](#) | [@EnableAutoConfiguration](#) | [Bean 与依赖注入](#)
- [运行代码](#): [IDE](#) | [Packaged](#) | [Maven](#) | [Gradle](#)
- [打包应用](#): [生产环境下的 jar](#)
- [Spring Boot CLI](#): [使用 CLI](#)

2.6. 了解 Spring Boot 新特性

需要更多关于 Spring Boot 核心特性？[Spring 特性](#)：

- 核心特性: [SpringApplication](#) | [外部配置](#) | [Profiles](#) | [日志](#)
- Web 应用程序: [MVC](#) | [嵌入式容器](#)
- 使用数据: [SQL](#) | [NO-SQL](#)
- 消息传递: [概述](#) | [JMS](#)
- 测试: [概述](#) | [Boot Applications](#) | [实用工具](#)
- 扩展: [自动配置](#) | [@Conditions](#)

2.7. 生产环境

您如果准备将 Spring Boot 应用推送至生产环境,或许会对以下内容[感兴趣](#).

- 管理端点: [概述](#)
- 连接方式: [HTTP](#) | [JMX](#)
- 监控: [指标](#) | [审计](#) | [HTTP 追踪](#) | [流程](#)

2.8. 高级内容

最后,我们为高级用户提供了几个主题.

- 部署 Spring Boot 应用: [云部署](#) | [OS 服务](#)
- 构建工具插件: [Maven](#) | [Gradle](#)
- 附录: [应用程序属性](#) | [配置元数据](#) | [自动配置类](#) | [测试自动配置注解](#) | [可执行 Jars](#) | [版本依赖](#)

Chapter 3. 入门

如果您是刚开始使用 Spring Boot, 或者对 Spring 有点印象, 那么这部分内容是为您准备的! 在这里我们将给出基本的 "是什么? "、"怎么做? "、"为什么? " 这类问题的答案. 这是一份友好的 Spring Boot 简介和安装说明. 当我们在讨论一些核心原理之后, 我们将构建第一个 Spring Boot 应用.

3.1. Spring Boot 简介

使用 Spring Boot 可以很容易地创建出能直接运行的独立的、生产级别的 Spring 的应用. 我们对 Spring 平台和第三方类库都有自己的考虑, 因此您可以从最基本的开始. 大多数 Spring Boot 应用只需要很少的 Spring 配置.

您可以使用 Spring Boot 来创建一个可以使用 `java -jar` 命令来运行或者基于传统的 war 包部署的应用程序. 我们还提供了一个用于运行 `spring scripts` 的命令行工具.

我们的主要目标是:

- 为所有 Spring Boot 开发提供一个更快、更全面的入门体验.
- 坚持自我, 但当需求出现偏离, 您需要能迅速摆脱出来.
- 提供大量非功能性特性相关项目 (例如: 内嵌服务器、安全、指标、健康检查、外部配置).
- 绝对没有代码生成, 也不要求 XML 配置.

3.2. 系统要求

Spring Boot 2.4.5 需要 Java 8 并且与 Java 16 (包括) 兼容. 还需要 Spring Framework 5.3.6 或更高版本.

为以下构建工具提供了明确的构建支持:

Build Tool	Version
Maven	3.3+
Gradle	6 (6.3 or later) (支持 5.6.x, 但已弃用)

3.2.1. Servlet 容器

支持以下嵌入式容器：

Name	Servlet Version
Tomcat 9.0	4.0
Jetty 9.4	3.1
Undertow 2.0	4.0

您可以将 Spring Boot 应用部署到任何一个与 Servlet 3.0+ 兼容的容器中。

3.3. 安装 Spring Boot

Spring Boot 可以与经典的 Java 开发工具一起使用或者作为命令行工具安装。无论如何，您都需要 [Java SDK v1.8](#) 或者更高版本。在开始之前您应该检查 Java 的安装情况：

```
$ java -version
```

如果您是 Java 开发新手，或者您只想尝试使用 Spring Boot，您可能需要首先尝试使用 [Spring Boot CLI](#)，否则请阅读经典的安装说明。

3.3.1. 针对 Java 开发人员的安装说明

您可以跟使用任何标准 Java 库的方式一样使用 Spring Boot。只需要在 classpath 下包含相应的 `spring-boot-*.jar` 文件即可。Spring Boot 不需要任何专用的工具来集成，因此您可以使用任何 IDE 或者文本编辑器，并且 Spring Boot 应用也没什么特殊之处，因此可以像任何其它 Java 程序一样运行和调试。

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

使用 Maven 安装

Spring Boot 兼容 Apache Maven 3.3 或更高版本。如果您还没有安装 Maven，可以到 maven.apache.org 上按照说明进行操作。



在许多操作系统上,可以通过软件包管理器来安装 Maven. 如果您是 OSX Homebrew 用户,请尝试使用 `brew install maven`. Ubuntu 用户可以运行 `sudo apt-get install maven`. 具有 Chocolatey 的Windows用户可以使用管理员权限(管理员)下运行 `choco install maven`.

Spring Boot 依赖使用到了 `org.springframework.boot groupId`. 通常,您的 Maven POM 文件将从 `spring-boot-starter-parent` 项目继承,并声明一个或多个“Starters”依赖. Spring Boot 还提供了一个可选的 [Maven 插件](#)来创建可执行 jar.

有关使用 Spring Boot 和 Maven 入门的更多详细信息,请参见 [Maven 插件参考指南的入门部分](#).

使用 Gradle 安装

Spring Boot 与 Gradle 6 (6.3 or later) 兼容. 还支持 Gradle 5.6.x,但已弃用,在将来的版本中将删除该支持. 如果您还没有安装 Gradle,您可以按照 [gradle.org](#) 上的说明进行操作.

Spring Boot 依赖 `org.springframework.boot group`. 通常,您的项目将声明一个或者多个“Starters”的依赖. Spring Boot 提供了一个有用的 [Gradle 插件](#),可用于简化依赖声明和创建可执行 jar 文件.

Gradle Wrapper

当您需要构建项目时,Gradle Wrapper 提供了一个用于获取 Gradle 的好方法. 它是由小脚本和库组成,您在提交的同时,您的代码将引导构建流程. 更多详细信息,请参阅 [docs.gradle.org/current/userguide/gradle_wrapper.html](#).

有关 Spring Boot 和 Gradle 入门的更多详细信息,请参见 [Gradle 插件参考指南的入门部分](#).

3.3.2. 安装 Spring Boot CLI

Spring Boot CLI 是一个命令行工具,如果您想使用 Spring 快速搭建项目原型

,您可以选择它。它允许您运行 [Groovy](#) 脚本,这意味着您有可以有类 Java 语法且没有太多样板的代码.

您不需要使用 CLI 来配合 Spring Boot,但它确实是一个入门 Spring 应用的最快方式.

手动安装

您可以从 Spring 软件仓库中下载 Spring CLI 发行版:

- [spring-boot-cli-2.4.5-bin.zip](#)
- [spring-boot-cli-2.4.5-bin.tar.gz](#)

[最新的快照发行版](#)也是可用的.

下载之后,请按照解压后文件中的 [INSTALL.txt](#) 说明进行操作. 总之: 在 [.zip](#) 文件的 bin/ 目录中有一个 [spring](#) 脚本 (在 Windows 下为 [spring.bat](#)) ,或者也可以使用 [java -jar](#) 配合 [.jar](#) 文件 (该脚本可以帮助您确保 classpath 设置正确) .

使用 **SDKMAN!** 安装

SDKMAN! (软件开发包管理器) 用于管理二进制 SDK 的多个版本,包括 Groovy 和 Spring Boot CLI. 从 [sdkman.io](#) 获取 SDKMAN! 并安装 Spring Boot:

```
$ sdk install springboot
$ spring --version
Spring Boot v2.4.5
```

如果您正在为 CLI 开发功能,并希望能够轻松地访问刚创建的版本,请参照以下指令.

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-
boot-cli-2.4.5-bin/spring-2.4.5/
$ sdk default springboot dev
$ spring --version
Spring CLI v2.4.5
```

以上操作将会安装一个名为 [dev](#) 的 [spring](#) 的本地实例. 它指向您的目标构建位置 ,因此每次重新构建 Spring Boot 时, [spring](#) 都是最新的.

您可以这样做来相关信息：

```
$ sdk ls springboot  
=====  
Available Springboot Versions  
=====  
> + dev  
* 2.4.5  
  
=====  
+ - local version  
* - installed  
> - currently in use  
=====
```

使用 OSX Homebrew 安装

如果您是在 Mac 上工作并且使用了 [Homebrew](#), 您安装 Spring Boot CLI 需要做的：

```
$ brew tap spring-io/tap  
$ brew install spring-boot
```

Homebrew 将会把 `spring` 安装在 `/usr/local/bin`.



如果您没有看到执行流程，您安装的 `brew` 可能已经过期了。执行 `brew update` 并重新尝试。

使用 MacPorts 安装

如果您是在 Mac 上工作并且使用了 [MacPorts](#), 您安装 Spring Boot CLI 所需要做的：

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

命令行完成

Spring Boot CLI 为 `BASH` 和 `zsh` 提供了命令完成脚本。您可以在任何 `shell` 中执行此脚本（也称为 `spring`）, 或将其放在您个人或系统范围的 `bash` 中完成初始化。在 Debian 系统上, 系统范围的脚本位于 `/shell-completion/bash` 中, 当新的 `shell`

启动时,该目录中的所有脚本将被执行. 要手动运行脚本, 例如: 您已经使用 `SDKMAN!` 安装了

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



如果您使用 `Homebrew` 或者 `MacPorts` 安装了 `Spring Boot CLI`, 则命令行完成脚本将自动注册到您的 `shell` 中.

Windows Scoop 安装

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

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

`Scoop` 将 `Spring` 安装在 `~/scoop/apps/springboot/current/bin`.



如果您没有看到应用清单, 则可能是因为瓢的安装已过期. 在这种情况下, 请运行 `scoop update` 更新, 然后重试.

快速入门 `Spring CLI` 示例

这是一个非常简单的 `web` 应用程序, 可以用于测试您的安装情况. 创建一个名为 `app.groovy` 的文件:

```
@RestController
class ThisWillActuallyRun {

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

}
```

之后在 `shell` 中运行它:

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



第一次运行应用的时候需要一些时间,因为需要下载依赖.
后续运行将会更快.

在您喜欢的浏览器中打开 localhost:8080,您应该会看到以下输出:

```
Hello World!
```

3.3.3. 升级旧版 Spring Boot

如果您想从 [1.x release](#) 升级Spring Boot ,升级到此版本,请查看项目 "[迁移指南](#)" .
该指南提供了详细的升级说明. 还请检查 "发行说明" 以获取每个发行版的
"新功能和值得注意的功能" 列表.

升级到新功能版本时,某些属性可能已被重命名或删除. Spring Boot
提供了一种在启动时分析应用程序环境并打印诊断的方法,还可以在运行时为您临时迁移属性.
要启用该功能,请将以下依赖添加到您的项目中:

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



因为 Properties 在环境中添加的比较晚, (例如使用
[@PropertySource](#) 时) 将不被考虑.



迁移完成后,请确保从项目的依赖中删除此模块.

要升级现有的 CLI,请使用相应的包管理器命令 (例如 [brew upgrade](#)) 或者,
如果您手动安装了 CLI,请按照[标准说明](#),记得更新您的 PATH 环境变量以删除任何旧的引用.

3.4. 开发第一个 Spring Boot 应用

让我们使用 Java 开发一个简单的 Hello World! web 应用程序,以便体现 Spring Boot 的一些关键特性. 我们将使用 Maven 构建该项目,因为大多数 IDE 都支持它.

[spring.io](#) 网站上有许多使用 Spring Boot 的入门 指南 ,如果您正在寻找具体问题的解决方案,可先从上面寻找.



您可以到 [start.spring.io](#) 使用依赖搜索功能选择 web starter 来快速完成以下步骤. 它将自动生成一个新的项目结构,以便您可以立即开始编码. 查看 [Spring Initializr](#) 文档了解更多信息.

在开始之前,打开终端检查您是否安装了符合要求的 Java 版本和 Maven 版本.

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

```
$ mvn -v
Apache Maven 3.5.4 (1eedded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-
17T14:33:14-04:00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_102, vendor: Oracle Corporation
```



此示例需要在您自己的目录中创建,后续的步骤说明假设您已经创建了这个目录,它是您的当前目录.

3.4.1. 创建 POM

我们先要创建一个 Maven `pom.xml` 文件. `pom.xml` 是用于构建项目的配方. 打开您最喜欢的编辑器并添加一下内容:

```
<?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
https://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>2.4.5</version>
</parent>

<description/>
<developers>
    <developer/>
</developers>
<licenses>
    <license/>
</licenses>
<scm>
    <url/>
</scm>
<url/>

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

<!-- (you don't need this if you are using a .RELEASE version) -->
<repositories>
    <repository>
        <id>spring-snapshots</id>
        <url>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>

```

```
</pluginRepository>
</pluginRepositories>
</project>
```

这应该会生成一个工作版本, 您可以通过运行 `mvn package` 来测试它 (此时您可以忽略 “jar will be empty - no content was marked for inclusion!” 警告信息).



此时, 您可以将项目导入 IDE (大部分的现代 Java IDE 都内置 Maven 支持) . 为了简单起见, 我们将继续在这个例子中使用纯文本编辑器.

3.4.2. 添加 Classpath 依赖

Spring Boot 提供了多个 “Starters” , 可以让您方便地将 jar 添加到 classpath 下. 我们的示例应用已经在 POM 的 `parent` 部分使用了 `spring-boot-starter-parent`. `spring-boot-starter-parent` 是一个特殊 `Starter`, 它提供了有用的 Maven 默认配置. 此外它还提供了[依赖管理](#)功能, 您可以忽略这些依赖的版本 (`version`) 标签.

其他 “Starters” 只提供在开发特定应用时可能需要到的依赖. 由于我们正在开发一个 web 应用, 因此我们将添加一个 `spring-boot-starter-web` 依赖, 但在此之前 , 让我们来看看目前拥有的.

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

`mvn dependency:tree` 命令以树的形式打印项目的依赖. 您可以看到 `spring-boot-starter-parent` 本身不提供依赖. 我们可以在 `parent` 下方立即编辑 `pom.xml` 并添加 `spring-boot-starter-web` 依赖:

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

如果您再次运行 `mvn dependency:tree`, 将会看到现在有许多附加的依赖, 包括了 Tomcat

web 服务器和 Spring Boot 本身.

3.4.3. 编码

要完成我们的应用, 我们需要创建一个 Java 文件. 默认情况下, Maven 将从 `src/main/java` 目录下编译源代码, 因此您需要创建该目录结构, 之后添加一个名为 `src/main/java/Example.java` 的文件:

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

@RestController
@EnableAutoConfiguration
public class Example {

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

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

虽然没有多少代码, 但它仍然做了很多事情. 让我们看看里面重要的部分.

`@RestController` 与 `@RequestMapping` 注解

`Example` 类中的第一个注解是 `@RestController`, 该注解被称作 *stereotype* 注解. 它能为代码阅读者提供一些提示, 对于 Spring 而言, 这个类具有特殊作用. 在本示例中, 我们的类是一个 web `@Controller`, 因此 Spring 在处理传入的 web 请求时会考虑它.

`@RequestMapping` 注解提供了 `routing` (路由) 信息. 它告诉 Spring, 任何具有路径为 `/` 的 HTTP 请求都应映射到 `home` 方法. `@RestController` 注解告知 Spring 渲染结果字符串直接返回给调用者.



`@RestController` 和 `@RequestMapping` 是 Spring MVC 注解
(它们不是 Spring Boot 特有的) . 有关更多详细信息,请参阅
Spring 参考文档中的 [MVC 章节](#)

`@EnableAutoConfiguration` 注解

第二个类级别注解是 `@EnableAutoConfiguration`. 此注解告知 Spring Boot 根据您添加的 jar 依赖来 "猜测" 您想如何配置 Spring 并进行自动配置,由于 `spring-boot-starter-web` 添加了 Tomcat 和 Spring MVC,auto-configuration (自动配置) 将假定您要开发 web 应用并相应设置了 Spring.

Starter 与自动配置

`Auto-configuration` 被设计与 `Starter` 配合使用,
但这两个概念并不是直接相关的. 您可以自由选择 `starter` 之外的 jar 依赖,Spring Boot 仍然会自动配置您的应用程序.

"main" 方法

应用的最后一部分是 `main` 方法. 这只是一个标准方法,其遵循 Java 规范中定义的应用程序入口点. 我们的 `main` 方法通过调用 `run` 来委托 Spring Boot 的 `SpringApplication` 类, `SpringApplication` 类将引导我们的应用,启动 Spring,然后启动自动配置的 Tomcat web 服务器. 我们需要将 `Example.class` 作为一个参数传递给 `run` 方法来告知 `SpringApplication`,它是 Spring 主组件. 同时还传递 `args` 数组以暴露所有命令行参数.

3.4.4. 运行示例

此时,我们的应用应该是可以工作了. 由于您使用了 `spring-boot-starter-parent` POM,因此您可以使用 `run` 来启动应用程序. 在根目录下输入 `mvn spring-boot:run` 以启动应用:

```
$ mvn spring-boot:run

.
-----
/\ / ____'-__ - _(_)- __ -- - \ \ \ \
( ( )\___ | '_ | '_| | '_ \ \ / _` | \ \ \
\ \ / ____| |_)| | | | | | | ( | | ) ) )
' | ____| .__|_|_|_|_|_\__, | / / /
=====|_|=====|_|____/_=/_/_/_/
:: Spring Boot :: (v2.4.5)
.....
..... (log output here)
.....
..... Started Example in 2.222 seconds (JVM running for 6.514)
```

如果您用浏览器打开了 `localhost:8080`, 您应该会看到以下输出:

Hello World!

要退出程序, 请按 `ctrl+c`.

3.4.5. 创建可执行 jar

我们通过创建一个完全自包含 (*self-contained*) 的可执行 jar 文件完成了示例. 该 jar 文件可以在生产环境中运行. 可执行 jar (有时又称为 **fat jars**) 是包含了编译后的类以及代码运行时所需要相关的 jar 依赖的归档文件.

可执行 jar 与 Java

Java 不提供任何标准方式来加载嵌套的 jar 文件 (比如本身包含在 jar 中的 jar 文件) . 如果您想打包一个包含Jar的应用, 这可能是个问题.

为了解决此问题, 许多开发人员使用了 **uber jar**, **uber jar** 从所有应用的依赖中打包所有的类到一个归档中. 这种方法的问题在于 , 您很难看出应用程序实际上使用到了哪些库. 如果在多个 jar 中使用了相同的文件名 (但内容不同) , 这也可能产生问题.

Spring Boot 采用了**不同方式**, 可以直接对 jar 进行嵌套.

要创建可执行 jar, 我们需要将 `spring-boot-maven-plugin` 添加到 `pom.xml` 文件中。在 `dependencies` 下方插入以下配置：

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



`spring-boot-starter-parent` POM 包含了 `<executions>` 配置, 用于绑定 `repackage`。如果您没有使用父 POM, 您需要自己声明此配置。有关详细的信息, 请参阅 [插件文档](#).

保存 `pom.xml` 并在命令行中运行 `mvn package`:

```
$ mvn package

[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] -----
[INFO] .... .
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---
[INFO] Building jar: /Users/developer/example/spring-boot-
example/target/myproject-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.4.5:repackage (default) @ myproject ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

如果您浏览 `target` 目录, 您应该会看到 `myproject-0.0.1-SNAPSHOT.jar`。该文件的大小大约为 10 MB。如果您想要查看里面的内容, 可以使用 `jar tvf`:

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

您应该还会在 `target` 目录中看到一个名为 `myproject-0.0.1-SNAPSHOT.jar.original` 的较小文件。这是在 Spring Boot 重新打包之前由 Maven 所创建的原始 jar 文件。

使用 `java -jar` 命令运行该应用：

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

. ----
/\ \ / _ _ ' _ _ _ ( _ ) _ _ _ _ _ \ \ \ \
( ( ) \_ _ _ | ' _ | ' _ | ' _ \ \ _ ` | \ \ \ \
\ \ / _ _ _ ) | ( _ ) | | | | | | ( _ | | ) ) ) )
' | _ _ _ | . _ | _ | _ | _ \ _ , | / / / /
=====|_|=====|_|=====|_|/_=/\_/_/_/
:: Spring Boot :: (v2.4.5)
.....
.....
..... (log output here)
.....
.....
..... Started Example in 2.536 seconds (JVM running for 2.864)
```

跟之前一样，要退出应用，请按 **ctrl-c**。

3.5. 下一步

希望您在本章节学到了一些 Spring Boot 的基础知识，并且开始编写自己的应用。如果您是一名面向任务 (task-oriented) 的开发人员，您可能想跳到 [spring.io](#) 并查看一些 [入门指南](#)。这些指南可以解决特定的“我该如何使用 Spring？”问题。此外我们还有 Spring Boot 专门的“[How-to](#)”参考文档。

接下来阅读的是第三部：[使用 Spring Boot](#). 如果您真的感到不耐烦了，可以跳过该部分直接阅读 [Spring Boot 特性](#).

Chapter 4. 使用 Spring Boot

本章节将详细介绍如何使用 Spring Boot.

它覆盖了诸如构建系统、自动配置和如何运行应用等主题。我们还介绍一些 Spring Boot 最佳实践。虽然 Spring Boot 并没有什么特别（它只是另一个您可以使用的类库），但仍然有一些建议可以让您的开发工作变得更加容易。

如果您是刚开始使用 Spring Boot，那么在深入本部分之前，您应该先阅读 [入门部分](#)。

4.1. 构建系统

强烈推荐您选择一个支持[依赖管理](#)的构建系统，您可以使用它将 artifact 发布到 Maven Central 仓库。我们建议您选择 Maven 或者 Gradle。虽然可以让 Spring Boot 与其它构建系统（如 Ant）配合工作，但它们不会得到特别好的支持。

4.1.1. 依赖管理

每一次 Spring Boot 发行都提供了一个它所支持的依赖清单。实际上，您不需要为构建配置提供任何依赖的版本，因为 Spring Boot 已经帮您管理这些了。当您升级 Spring Boot 时，这些依赖也将以一致的方式进行升级。



如果您觉得有必要，您仍然可以指定一个版本并覆盖 Spring Boot 所推荐的。

该清单包含了全部可以与 Spring Boot 一起使用的 spring 模块以及第三方类库，可作为标准依赖清单 ([spring-boot-dependencies](#))，并且可以与 Maven 和 Gradle 一起使用。



Spring Boot 的每一次发行都会基于一个 Spring Framework 版本，因此我们强烈建议您不要指定它的版本。

4.1.2. Maven

要了解有关将 Spring Boot 与 Maven 结合使用的信息，请参阅 Spring Boot 的 Maven 插件的文档

- 参考文档 ([HTML](#) 和 [PDF](#))
- [API](#)

4.1.3. Gradle

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

- 参考文档 ([HTML](#) 和 [PDF](#))
- [API](#)

4.1.4. Ant

可以使用 Apache Ant+Ivy 构建 Spring Boot 项目. [spring-boot-antlib](#) AntLib 模块也可以帮助 Ant 创建可执行 jar 文件.

要声明依赖, 可参考以下一个典型的 `ivy.xml` 文件内容:

```
<ivy-module version="2.0">
    <info organisation="org.springframework.boot" module="spring-boot-sample-ant" />
    <configurations>
        <conf name="compile" description="everything needed to compile this module" />
        <conf name="runtime" extends="compile" description="everything needed to run this module" />
    </configurations>
    <dependencies>
        <dependency org="org.springframework.boot" name="spring-boot-starter" rev="${spring-boot.version}" conf="compile" />
    </dependencies>
</ivy-module>
```

一个典型的 `build.xml` 大概是这样:

```

<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="2.4.5" />

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

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

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

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

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

```



如果您不想使用 `spring-boot-antlib` 模块,请参阅 [使用 Ant 构建可执行归档文件](#),无需使用 `spring-boot-antlib`.

4.1.5. Starters

`Starter` 是一组惯例依赖描述资源,可以包含在应用中. 从 `starter` 中,您可以获得所需的所有 `Spring` 和相关技术的一站式支持,无须通过示例代码和复制粘贴来获取依赖. 比如,如果您要使用 `Spring` 和 `JPA`

进行数据库访问,那么只需要在项目中包含 `spring-boot-starter-data-jpa` 依赖即可.

`starter` 包含了许多您需要用于使项目快速启动和运行
,并且需要一组受支持的可传递依赖的依赖.

命名含义

官方的所有 `starter` 都遵循类似的命名规则: `spring-boot-starter-*`,其中 `*` 是特定类型的应用. 这个命名结构旨在帮助您找到 `starter`. 许多 IDE 中 Maven 集成允许您按名称搜索依赖. 例如,安装了 Eclipse 或者 STS 插件后 ,您可以简单地在 POM 编辑器中按下 `ctrl-space` 并输入 `spring-boot-starter` 来获取完整的列表.

正如 “[创建自己的 `starter`](#)” 章节所述,第三方的 `starter` 命名不应该以 `spring-boot` 开头,因为它是官方 Spring Boot artifacts 所保留的规则. 例如 ,有一个第三方 `starter` 项目叫做 `thirdpartyproject`,它通常会命名为 `thirdpartyproject-spring-boot-starter`.

Spring Boot 在 `org.springframework.boot group` 下提供了以下应用 `starter`:

Table 1. Spring Boot 应用类 Starter

Name	Description
<code>spring-boot-starter</code>	核心的 <code>starter</code> , 包含自动配置, 日志和 YAML
<code>spring-boot-starter-activemq</code>	使用 Apache ActiveMQ 传递 JMS 消息的 Starter
<code>spring-boot-starter-amqp</code>	使用 Spring AMQP 和 Rabbit MQ Starter
<code>spring-boot-starter-aop</code>	使用 Spring AOP 和 AspectJ 的面向切面编程的 Starter
<code>spring-boot-starter-artemis</code>	使用 Apache Artemis 传递 JMS 消息的 Starter
<code>spring-boot-starter-batch</code>	使用 Spring Batch 的 Starter

Name	Description
<code>spring-boot-starter-cache</code>	使用 Spring Framework 的缓存支持的 Starter
<code>spring-boot-starter-cloud-connectors</code>	使用 Spring Cloud Connectors 的 Starter ,可简化 Cloud Foundry 和 Heroku 等云平台服务之间的连接. 弃用 Java CFEnv
<code>spring-boot-starter-data-cassandra</code>	使用 Cassandra distributed database 和 Spring Data Cassandra 的 Starter
<code>spring-boot-starter-data-cassandra-reactive</code>	使用 Cassandra distributed 和 and Spring Data Cassandra Reactive Starter
<code>spring-boot-starter-data-couchbase</code>	使用 Couchbase document-oriented database 和 Spring Data Couchbase Starter
<code>spring-boot-starter-data-couchbase-reactive</code>	使用 Couchbase document-oriented database 和 Spring Data Couchbase Reactive Starter
<code>spring-boot-starter-data-elasticsearch</code>	使用 Elasticsearch 搜索分析引擎和 Spring Data Elasticsearch 的 Starter
<code>spring-boot-starter-data-jdbc</code>	使用 Spring Data JDBC 的 Starter
<code>spring-boot-starter-data-jpa</code>	使用 Hibernate 的 Spring Data JPA 的 Starter
<code>spring-boot-starter-data-ldap</code>	使用 Spring Data LDAP 的 Starter
<code>spring-boot-starter-data-mongodb</code>	使用 MongoDB 文档数据库 和 Spring Data MongoDB 的 Starter
<code>spring-boot-starter-data-mongodb-reactive</code>	使用 MongoDB 文档数据库 和 Spring Data MongoDB Reactive Starter
<code>spring-boot-starter-data-neo4j</code>	使用 Spring Data Neo4j 支持 Neo4j graph database 的 Starter

Name	Description
<code>spring-boot-starter-data-r2dbc</code>	使用 Spring Data R2DBC 的 Starter
<code>spring-boot-starter-data-redis</code>	使用 Spring Data Redis 和 Lettuce 客户端支持 Redis key-value 数据存储的 Starter
<code>spring-boot-starter-data-reactive</code>	使用 Spring Data Redis reactive 和 Lettuce 客户端支持 Redis key-value 数据存储的 Starter
<code>spring-boot-starter-data-rest</code>	使用 Spring Data REST 将 Spring Data 存储库以 REST 形式暴露的 Starter
<code>spring-boot-starter-data-solr</code>	使用 Spring Data Solr 来支持 Apache Solr search platform. 2.3.9 后弃用
<code>spring-boot-starter-freemarker</code>	使用 FreeMarker 视图构建 MVC web 应用程序
<code>spring-boot-starter-groovy-templates</code>	使用 Groovy Templates 视图构建 MVC web 应用程序
<code>spring-boot-starter-hateoas</code>	使用 Spring MVC 和 Spring HATEOAS 构建超媒体 RESTful web 应用程序
<code>spring-boot-starter-integration</code>	使用 Spring Integration 的 Starter
<code>spring-boot-starter-jdbc</code>	使用 HikariCP 连接池的 JDBC 的 Starter
<code>spring-boot-starter-jersey</code>	使用 JAX-RS 和 Jersey 构建 RESTful web 应用程序。 <code>spring-boot-starter-web</code> 的替代方法
<code>spring-boot-starter-jooq</code>	使用 jOOQ 访问 SQL 数据库。 <code>spring-boot-starter-data-jpa</code> 或 <code>spring-boot-starter-jdbc</code> 的替代方法
<code>spring-boot-starter-json</code>	json 读写的 Starter
<code>spring-boot-starter-jta-atomikos</code>	使用 Atomikos 的 JTA transactions 的 Starter

Name	Description
spring-boot-starter-jta-bitronix	使用 Bitronix 的 JTA transactions 的 Starter. 2.3.0 已弃用
spring-boot-starter-mail	使用 Java Mail 和 Spring Framework 邮件发送的 Starter
spring-boot-starter-mustache	使用 Mustache 视图构建 web 应用程序的 Starter
spring-boot-starter-oauth2-client	使用 Spring Security 的 OAuth2/OpenID Connect 客户端功能的 Starter
spring-boot-starter-oauth2-resource-server	使用 Spring Security 的 OAuth2 资源服务器功能的 Starter
spring-boot-starter-quartz	使用 Quartz scheduler 的 Starter
spring-boot-starter-rsocket	构建 RSocket 客户端 和 服务端的 Starter.
spring-boot-starter-security	使用 Spring Security 的 Starter
spring-boot-starter-test	使用 JUnit Jupiter, Hamcrest 和 Mockito 等库测试 Spring Boot 应用程序的 Starter
spring-boot-starter-thymeleaf	使用 Thymeleaf 视图构建 MVC web 应用程序
spring-boot-starter-validation	使用 Hibernate Validator 进行 Java Bean Validation 的 Starter
spring-boot-starter-web	使用 Spring MVC 构建 web (包括 RESTful) 应用程序. 使用 Tomcat 作为默认的嵌入式容器.
spring-boot-starter-web-services	使用 Spring Web Services 的 Starter
spring-boot-starter-webflux	使用 Spring Framework 对 Reactive Web 的支持来构建 WebFlux 应用程序的 Starter

Name	Description
<code>spring-boot-starter-websocket</code>	使用 Spring Framework 对 WebSocket 的支持来构建 WebSocket 应用程序的 Starter

除了应用程序的 `starter` 外,以下 `starter` 可用于添加 [生产就绪](#) 特性:

Table 2. Spring Boot 生产类 starter

Name	Description
<code>spring-boot-starter-actuator</code>	使用 Spring Boot 的 Actuator 的 Starter,该启动器提供了生产就绪功能,可帮助您监视和管理应用程序

最后,Spring Boot 还包含以下 `starter`,如果您想要排除或切换其他特定技术,可以使用以下 `starter`:

Table 3. Spring Boot 技术类 starter

Name	Description
<code>spring-boot-starter-jetty</code>	使用 Jetty 作为嵌入式容器. <code>spring-boot-starter-tomcat</code> 的替代方法
<code>spring-boot-starter-log4j2</code>	使用 Log4j2 来记录日志. <code>spring-boot-starter-logging</code> 的替代方法
<code>spring-boot-starter-logging</code>	使用 Logback 来记录日志. 默认的日志 starter
<code>spring-boot-starter-reactor-netty</code>	使用 Reactor Netty 作为嵌入的 reactive HTTP server.
<code>spring-boot-starter-tomcat</code>	使用 Tomcat 作为嵌入 servlet 容器. <code>spring-boot-starter-web</code> 使用此作为默认 Servlet 容器
<code>spring-boot-starter-undertow</code>	使用 Undertow 作为嵌入 servlet 容器. <code>spring-boot-starter-tomcat</code> 的替代方法

要了解如何交换技术方面的信息,请参阅 [swapping web server and logging system.](#)



有关其它社区贡献的 `starter` 列表,请参阅 GitHub 上的 [spring-boot-starters](#) 模块中的 `README file` 文件.

4.2. 组织代码

Spring Boot 不需要任何特定的代码布局,但是有一些最佳实践是很有用的.

4.2.1. 使用 “default” 包

当一个类没有 `package` 声明时,它就被认为是在 `default` 包中. 通常不鼓励使用 `default` 包,应该避免使用. 对于使用 `@ComponentScan`、`@EntityScan` 或者 `@SpringBootApplication` 注解的 Spring Boot 应用,这样可能会导致特殊问题发生,因为每一个 `jar` 中的每一个类将会被读取到.



我们建议您使用 Java 推荐的包命名约定,并使用域名的反向形式命名(例如 `com.example.project`) .

4.2.2. 定位主应用类

我们通常建议您将主应用类放在其它类之上的根包中, `@SpringBootApplication` 注解通常放在主类上,它隐式定义了某些项目的包扫描的起点. 例如,如果您在编写一个 JPA 应用程序,则被 `@SpringBootApplication` 注解的类所属的包将被用于搜索标记有 `@Entity` 注解的类.

使用根包还可以允许使用没有指定 `basePackage` 属性的 `@ComponentScan` 注解. 如果您的主类在根包中,也可以使用 `@SpringBootApplication` 注解.



如果您不想使用 `@SpringBootApplication`,则可以通过导入的 `@EnableAutoConfiguration` 和 `@ComponentScan` 注解来定义该行为,因此也可以使用它们.

以下是一个经典的包结构:

```
com
+- example
  +- myapplication
    +- Application.java
    |
    +- customer
      +- Customer.java
      +- CustomerController.java
      +- CustomerService.java
      +- CustomerRepository.java
      |
    +- order
      +- Order.java
      +- OrderController.java
      +- OrderService.java
      +- OrderRepository.java
```

`Application.java` 文件声明了 `main` 方法, 附带了 `@SpringBootApplication` 注解.

```
package com.example.myapplication;

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

@SpringBootApplication
public class Application {

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

}
```

4.3. 配置类

Spring Boot 支持基于 Java 的配置. 虽然可以在 `SpringApplication` 中使用 XML 配置源, 但我们通常建议主配置源为 `@Configuration` 类. 通常, 一个很好的选择是将定义了 `main` 方法的类作为 `@Configuration`.



许多 Spring 的 XML 配置示例已经在 Internet 上发布了.
如果可能的话,您无论如何都应该尝试着使用等效的基于 Java
的配置方式,搜索 `Enable*` 注解可以帮到您不少忙.

4.3.1. 导入额外的配置类

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

4.3.2. 导入 XML 配置

如果您一定要使用基于 XML 的配置,我们建议您仍然使用 `@Configuration` 类. 您可以使用 `@ImportResource` 注解来加载 XML 配置文件.

4.4. 自动配置

Spring Boot 自动配置尝试根据您添加的 jar 依赖自动配置 Spring 应用. 例如,如果 classpath 下存在 `HSQLDB`,并且您没有手动配置任何数据库连接 bean,那么 Spring Boot 将自动配置一个内存数据库.

您需要通过将 `@EnableAutoConfiguration` 或者 `@SpringBootApplication` 注解添加到其中一个 `@Configuration` 类之上以启用自动配置.



您应该只添加一个 `@SpringBootApplication` 或 `@EnableAutoConfiguration` 注解.
我们通常建议您仅将一个或另一个添加到您的主要 `@Configuration` 类中.

4.4.1. 平滑替换自动配置

自动配置是非侵入的,您可以随时定义自己的配置来代替自动配置的特定部分. 例如,如果您添加了自己的 `DataSource` bean,默认的嵌入式数据库支持将不会自动配置.

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,请使用 `--debug` 开关启动应用.
这样做可以为核心 logger 启用调试日志,并记录到控制台.

4.4.2. 禁用指定的自动配置类

如果您发现在正在使用不需要的自动配置类,可以通过使用 `@SpringBootApplication` 的 `exclude` 属性来禁用它们.

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

@SpringBootApplication(exclude={DataSourceAutoConfiguration.class})
public class MyApplication {
}
```

如果类不在 `classpath` 下,您可以使用注解的 `excludeName` 属性并指定完全类名. 最后,您还可以通过 `spring.autoconfigure.exclude` property 控制要排除的自动配置类列表.

如果您更喜欢使用 `@EnableAutoConfiguration` 而不是 `@SpringBootApplication`,则还可以使用 `exclude` 和 `excludeName`.



您可以同时使用注解和 property 定义排除项



即使自动配置类是 `public` 的,该类的被认为是 `public API` 的唯一一方面是可用于禁用自动配置的类的名称. 这些类的实际内容(例如嵌套配置类或 Bean 方法)仅供内部使用,我们不建议直接使用它们. ,

4.5. Spring Bean 与依赖注入

您可以自由使用任何标准的 Spring Framework 技术来定义您的 bean 以及它们注入的依赖. 我们发现使用 `@ComponentScan` 来寻找 bean 和结合 `@Autowired` 构造器注入可以很好地工作.

如果您按照上述的建议 (将应用类放在根包中) 来组织代码,则可以添加无参的 `@ComponentScan`. 所有应用组件 (`@Component`、`@Service`、`@Repository`、`@Controller` 等) 将自动注册为 Spring Bean.

以下是一个 `@Service` Bean,其使用构造注入方式获取一个必需的 `RiskAssessor` bean.

```

package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...

}

```

如果 bean 中只有一个构造方法,您可以忽略掉 `@Autowired` 注解.

```

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...

}

```



请注意,构造注入允许 `riskAssessor` 字段被修饰为 `final`,这表示以后它不能被更改.

4.6. 使用 `@SpringBootApplication` 注解

很多 Spring Boot 开发者总是使用 `@Configuration`、`@EnableAutoConfiguration` 和 `@ComponentScan` 注解标记在主类上. 由于 这些注解经常一起使用

(特别是如果您遵循上述的最佳实践)。Spring Boot 提供了一个更方便的 `@SpringBootApplication` 注解可用来替代这个组合。

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

```
package com.example.myapplication;

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

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration
@ComponentScan
public class Application {

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

}
```



`@SpringBootApplication` 还提供别名以自定义 `@EnableAutoConfiguration` 和 `@ComponentScan` 的属性。

这些功能都不是强制性的,您可以选择用它启用的任何功能替换此单个注解。例如,您可能不想在应用程序中使用组件扫描或配置属性扫描:

```
package com.example.myapplication;

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

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

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

}
```



在此示例中,除了程序没有自动检测到 `@Component` 注解的类和 `@ConfigurationProperties` 注解的类和显式导入了用户定义的 Bean之外, `Application` 就像其他任何 Spring Boot 应用程序一样 (请参阅 [@Import](#)) .

4.7. 运行您的应用

将应用程序打包成 `jar` 可执行文件并使用嵌入式 HTTP 服务器的最大优点之一就是可以按照您想使用的其它方式来运行应用。调试 Spring Boot 也很简单,您不需要任何特殊的 IDE 插件或者扩展。



本章节仅涵盖基于 `jar` 的打包方式,如果您选择将应用打包为 `war` 文件,则应该参考您的服务器和 IDE 文档。

4.7.1. 使用 IDE 运行

您可以使用 IDE 运行 Spring Boot 应用,就像运行一个简单的 Java 应用程序一样,但是首先您需要导入项目,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 File 菜单中选择 Import... → Existing Maven Projects .

如果您无法将项目直接导入到 IDE 中,则可以使用构建插件生成 IDE 元数据 (metadata) . Maven 包含了 Eclipse 和 IDEA 的插件,Gradle 也为 各种 IDE 提供了插件.



如果您不小心运行了两次 web 应用,您将看到一个 Port already in use (端口已经被使用) 错误. STS 用户可以使用 Relaunch 按钮运行以确保现有的任何实例都已关闭,而不是使用 Run 按钮.

4.7.2. 作为打包应用运行

如果您使用 Spring Boot Maven 或者 Gradle 插件创建可执行 jar,可以使用 java -jar 命令运行应用. 例如:

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

也可以在运行打包应用程序时开启远程调试支持. 该功能允许您将调试器附加到打包的应用中.

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

4.7.3. 使用 Maven 插件

Spring Boot Maven 插件包含一个可用于快速编译和运行应用程序的 run goal.

应用程序以快速形式运行,就像在 IDE 中一样. 以下示例展示了运行 Spring Boot 应用程序的典型 Maven 命令:

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

您可能还想使用 MAVEN_OPTS 操作系统环境变量,如下例所示:

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

4.7.4. 使用 Gradle 插件

Spring Boot Gradle 插件包含一个 `bootRun` 任务, 可用于以快速形式运行应用程序. 每当应用 `org.springframework.boot` 和 `java` 插件时都会添加 `bootRun` 任务:

```
$ gradle bootRun
```

您可能还想使用 `JAVA_OPTS` 操作系统环境变量:

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

4.7.5. 热交换

由于 Spring Boot 应用程序只是普通的 Java 应用程序, 因此 JVM 热插拔是可以开箱即用. JVM 热插拔在可替换字节码方面有所限制. 想要更完整的解决方案, 可以使用 [JRebel](#).

`spring-boot-devtools` 模块包含了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 [开发人员工具](#) 部分以及[热插拔的 How-to](#) 部分.

4.8. 开发者工具

Spring Boot 包含了一套工具, 可以使应用开发体验更加愉快. `spring-boot-devtools` 模块可包含在任何项目中, 以提供额外的开发时 (`development-time`) 功能. 要启用 `devtools` 支持, 只需要将模块依赖添加到您的构建配置中即可:

Maven

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

Gradle

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

当运行完全打包的应用时, 开发者工具将会自动禁用. 如果您的应用使用了 `java -jar` 方式或者特殊的类加载器启动

, 那么它会被认为是一个生产级别应用. 你可以通过使用 `spring.devtools.restart.enabled` 系统属性来控制这种行为, 要启用 `devtools`, 不管用什么类加载器来启动你的应用程序, 设置 `-Dspring.devtools.restart.enabled=true` 系统属性. 在运行 `devtools` 会带来安全风险的生产环境中, 绝对不能这样做, 要禁用 `devtools`, 请排除依赖关系或设置 `-Dspring.devtools.restart.enabled = false` 系统属性.



将 Maven 的依赖标记为可选或者在 Gradle 中使用 `developmentOnly` 是防止您的项目被其他模块使用时 `devtools` 被应用到其它模块的最佳方法.



重新打包的归档默认情况下不包含 `devtools`. 如果要使用某些 `远程 devtools 功能`, 你需要禁用 `excludeDevtools` 构建属性以把 `devtools` 包含进来. 使用 Maven 插件时, 请将 `excludeDevtools` 属性设置为 `false`. 使用 Gradle 插件时, 将任务的类路径配置为包括 `"developmentOnly"` 配置.

4.8.1. Property 默认值

Spring Boot 所支持的一些库使用了缓存来提高性能. 例如, [模板引擎](#) 将缓存编译后的模板, 以避免重复解析模板文件. 此外, Spring MVC 可以在服务静态资源时添加 HTTP 缓存头.

虽然缓存在生产中非常有用,但它在开发过程可能会产生相反的效果,让您不能及时看到刚才在应用中作出的更改. 因此,`spring-boot-devtools` 将默认禁用这些缓存选项.

一般是在 `application.properties` 文件中设置缓存选项. 例如, `Thymeleaf` 提供了 `spring.thymeleaf.cache` 属性. 您不需要手动设置这些属性, `spring-boot-devtools` 会自动应用合适的开发时 (`development-time`) 配置.

由于在开发 `Spring MVC` 和 `Spring WebFlux` 应用程序时需要有关 `Web` 请求的更多信息,因此开发者工具将为 `Web` 日志记录组启用 `DEBUG` 日志记录. 这将为您提供有关传入请求,正在处理的处理程序,响应结果等的信息. 如果您希望记录所有请求详细信息(包括潜在的敏感信息), 则可以打开 `spring.mvc.log-request-details` 配置属性.



如果您不希望应用默认属性,则可以在 `application.properties` 中将 `spring.devtools.add-properties` 设置为 `false`.



有关 `devtools` 应用的属性的完整列表,请参见 [DevToolsPropertyDefaultsPostProcessor](#) .

4.8.2. 自动重启

使用 `spring-boot-devtools` 的应用在 `classpath` 下的文件发生更改时会自动重启.这对于使用 `IDE` 工作而言可能是一个非常棒的功能,因为它为代码变更提供了非常快的反馈. 默认情况下,将监视 `classpath` 指向的所有目录. 请注意,某些资源(如静态资源和视图模板)不需要重启应用.

触发重启

当 DevTools 监视 classpath 资源时, 触发重启的唯一方式是更新 classpath. 使 classpath 更新的方式取决于您使用的 IDE.

- 在 Eclipse 中, 保存修改的文件将更新 classpath, 从而触发重启.
- 在 IntelliJ IDEA 中, 构建项目 (`Build → Build Project`) 将产生相同的效果.
- 如果使用构建插件, Maven 运行 `mvn compile` 或 Gradle 运行 `gradle build` 将触发重启.



只要 `forking` 被开启, 您可以使用受支持的构建工具 (如 Maven 或

Gradle) 来重启应用, 因为 DevTools

需要隔离应用类加载器才能正常运行. 默认情况下, 当在 classpath 下检测到 DevTools 时, Gradle 和 Maven 会这么做.



自动重启功能与 LiveReload (实时重载) 一起使用效果更棒. 阅读

LiveReload 章节以获取更多信息. 如果您使用

JRebel, 自动重启将会被禁用, 以支持动态类重载, 但其他 devtools 功能 (如 LiveReload 和 property 覆盖) 仍然可以使用.



DevTools 依赖于应用上下文的关闭钩子, 以便在重启期间关闭自己.

如果禁用了关闭钩子

(`SpringApplication.setRegisterShutdownHook(false)`)

, 它将不能正常工作.



DevTools 需要自定义 `ApplicationContext` 使用到的

`ResourceLoader`. 如果您的应用已经提供了一个, 它将被包装起来

, 因为不支持在 `ApplicationContext` 上直接覆盖 `getResource` 方法.

重启 (Restart) 与重载 (Reload)

Spring Boot 通过使用两个类加载器来提供了重启技术。不改变的类（例如，第三方 jar）被加载到 `base` 类加载器中。经常处于开发状态的类被加载到 `restart` 类加载器中。当应用重启时，`restart` 类加载器将被丢弃，并重新创建一个新的。这种方式意味着应用重启比冷启动要快得多，因为省去 `base` 类加载器的处理步骤，并且可以直接使用。

如果您觉得重启还不够快，或者遇到类加载问题，您可以考虑如 ZeroTurnaround 的 [JRebel](#) 等工具。他们是通过在加载类时重写类来加快重新加载。

条件评估变更日志

默认情况下，每次应用重启时，都会记录显示条件评估增量的报告。该报告展示了在您进行更改（如添加或删除 bean 以及设置配置属性）时对应用自动配置所作出的更改。

要禁用报告的日志记录，请设置以下属性：

```
spring:
  devtools:
    restart:
      log-condition-evaluation-delta: false
```

排除资源

某些资源在更改时不一定需要触发重启。例如，Thymeleaf 模板可以实时编辑。默认情况下，更改 `/META-INF/maven`、`/META-INF/resources`、`/resources`、`/static`、`/public` 或者 `/templates` 不会触发重启，但会触发 `live reload`。如果您想自定义排除项，可以使用 `spring.devtools.restart.exclude` 属性。例如，仅排除 `/static` 和 `/public`，您可以设置以下内容：

```
spring:
  devtools:
    restart:
      exclude: "static/**,public/**"
```



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

监视附加路径

如果您想在对不在 `classpath` 下的文件进行修改时重启或重载应用，请使用
`spring.devtools.restart.additional-paths` 属性来配置监视其他路径的更改情况。
 您可以使用[上述](#)的 `spring.devtools.restart.exclude`
 属性来控制附加路径下的文件被修改时是否触发重启或只是 [live reload](#).

禁用重启

您如果不想使用重启功能，可以使用 `spring.devtools.restart.enabled` 属性来禁用它。
 一般情况下，您可以在 `application.properties` 中设置此属性
 （重启类加载器仍将被初始化，但不会监视文件更改）.

如果您需要完全禁用重启支持（例如，可能它不适用于某些类库），您需要在调用
`SpringApplication.run(...)` 之前将 `System` 属性
`spring.devtools.restart.enabled` `System` 设置为 `false`. 例如：

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

使用触发文件

如果您使用 IDE 进行开发，并且时时刻刻在编译更改的文件
 ，或许您只是希望在特定的时间内触发重启。为此，您可以使用触发文件，这是一个特殊文件
 ，您想要触发重启检查时，必须修改它。



更改文件只会触发检查，只有在 Devtools
 检查到它需要做某些操作时才会触发重启，可以手动更新触发文件，也可以通
 过 IDE 插件更新。

要使用触发文件，请设置 `spring.devtools.restart.trigger-file`
 属性指向触发文件的路径。

例如,如果您的项目具有以下结构:

```
src
+- main
  +- resources
    +- .reloadtrigger
```

然后,您的触发文件属性将是:

```
spring:
  devtools:
    restart:
      trigger-file: ".reloadtrigger"
```

现在仅在更新 `src/main/resources/.reloadtrigger` 时才发生重启.



您也许想将 `spring.devtools.restart.trigger-file` 设置成一个全局配置,以使得所有的项目都能应用此方式.

某些IDE具有使您不必手动更新触发器文件的功能. [Spring Tools for Eclipse](#) 的 Spring工具和 [IntelliJ IDEA \(Ultimate Edition\)](#) 都具有这种支持. 使用Spring Tools, 您可以从控制台视图使用 "重新加载" 按钮 (只要您的 `trigger-file` 名为 `.reloadtrigger`) . 对于IntelliJ,您可以按照其 [文档中的说明](#) 进行操作.

自定义重启类加载器

正如之前的 [重启和重载](#) 部分所述,重启功能是通过使用两个类加载器来实现的. 对于大多数应用而言,这种方式很好,然而,有时可能会导致类加载出现问题.

默认情况下,IDE 中任何打开的项目将使用 "restart" 类加载器加载,任何常规的 `.jar` 文件将使用 `base` 类加载器加载. 您如果开发的是多模块项目,而不是每一个模块都导入到 IDE 中,则可能需要自定义. 为此,您可以创建一个 `META-INF/spring-devtools.properties` 文件.

`spring-devtools.properties` 文件可以包含以 `restart.exclude.` 和 `restart.include.` 为前缀的属性. `include` 元素是加载到 `restart` 类加载器的项, `exclude` 元素是加载到 "base" 类加载器的项. 属性值是一个应用到 `classpath`

的正则表达式。例如：

```
restart:
exclude:
  companycommonlibs: "/mycorp-common-[\\w\\d-\\.]+\\.jar"
include:
  projectcommon: "/mycorp-myproj-[\\w\\d-\\.]+\\.jar"
```



所有属性键名必须是唯一的。只要有一个属性以 `restart.include`. 或 `restart.exclude`. 开头，它将被考虑。



`classpath` 下的所有 `META-INF/spring-devtools.properties` 文件将被加载，您可以将它们打包进工程或者类库中为项目所用。

已知限制

重新启动功能对使用标准 `ObjectInputStream` 反序列化的对象无效。

您如果需要反序列化数据，可能需要使用 Spring 的 `ConfigurableObjectInputStream` 配合 `Thread.currentThread().getContextClassLoader()`。

遗憾的是，一些第三方类库在没有考虑上下文类加载器的情况下使用了反序列化。

您如果遇到此问题，需要向原作者提交修复请求。

4.8.3. LiveReload

`spring-boot-devtools` 模块包括了一个内嵌 `LiveReload` 服务器，它可在资源发生更改时触发浏览器刷新。您可以从 livereload.com 上免费获取 Chrome、Firefox 和 Safari 平台下对应的 `LiveReload` 浏览器扩展程序。

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



您一次只能运行一个 `LiveReload` 服务器。在启动应用之前，请确保没有其他 `LiveReload` 服务器正在运行。如果在 IDE 中启动了多个应用，那么只有第一个应用的 `LiveReload` 生效。



要在文件更改时触发 LiveReload, 必须启用 [自动重启](#).

4.8.4. 全局设置

您可以通过将以下任何文件添加到 `$HOME/.config/spring-boot` 目录来配置全局 `devtools` 设置：

1. `spring-boot-devtools.properties`
2. `spring-boot-devtools.yaml`
3. `spring-boot-devtools.yml`

在此文件中添加的任何属性将应用到您的计算机上所有使用了 `devtools` 的 Spring Boot 应用。例如，始终使用[触发文件](#)来配置重启功能，您需要将以下属性添加到您的 `spring-boot-devtools` 文件中：：

```
spring:  
  devtools:  
    restart:  
      trigger-file: ".reloadtrigger"
```



如果在 `$HOME/.config/spring-boot` 中找不到 `devtools` 配置文件，则在 `$HOME` 目录的根目录中搜索是否存在 `.spring-boot-devtools.properties` 文件。这使您可以与不支持 `$HOME/.config/spring-boot` 位置的较旧版本的 Spring Boot 上的应用程序共享 `devtools` 全局配置。



在上述文件中激活的配置文件不会影响 [指定 profile 的配置文件](#) 的加载。.

Profiles 不支持 devtools properties/yaml 文件.



任何在 `.spring-boot-devtools.properties` 中激活的 profiles 不会影响 指定 profile 的配置文件 的加载. 不支持在 YAML 和 Properties 文件中的配置特定于配置文件的文件名(格式为 `spring-boot-devtools-<profile>.properties`) 和 `spring.config.activate.on-profile` 子文档.

配置文件系统监视器

`FileSystemWatcher` 的工作方式是按一定的时间间隔轮询更改类 , 然后等待定义好的一段时间以确保没有更多更改. 由于 Spring Boot 完全依赖 IDE 来编译文件并将其复制到 Spring Boot 可以读取文件的位置, 因此您可能会发现, 有时 devtools 重新启动应用程序时某些更改未反映出来. 如果您经常观察到此类问题, 请尝试将 `spring.devtools.restart.poll-interval` 和 `spring.devtools.restart.quiet-period` 参数增加到适合您开发环境的值:

```
spring:
  devtools:
    restart:
      poll-interval: "2s"
      quiet-period: "1s"
```

现在每 2 秒轮询一次受监视的 `classpath` 目录是否有更改, 并保持 1 秒钟的静默时间以确保没有其他类更改.

4.8.5. 远程应用

Spring Boot 开发者工具不局限于本地开发. 在远程运行应用时也可以使用许多功能. 远程支持功能是可选的, 您还可以使用多种功能. 选择启用远程支持 , 因为启用它可能会带来安全风险. 仅当在受信任的网络上运行或使用 SSL 保护时, 才应启用它. 如果这两个选项都不可用, 则不应使用 DevTools 的远程支持. 您永远不要在生产部署上启用. 如果要启用, 您需要确保在重新打包归档文件时包含 `devtools:`

```

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

```

之后您需要设置一个 `spring.devtools.remote.secret` 属性
，像任何重要的密码或机密一样，该值应唯一且强壮，以免被猜测或强行使用。

远程 `devtools` 支持分为两部分：接受连接的服务器端端点和在 IDE 中运行的客户端应用程序。设置 `spring.devtools.remote.secret` 属性后，将自动启用服务器组件。客户端组件必须手动启动。

运行远程客户端应用

假设远程客户端应用运行在 IDE 中。您需要在与要连接的远程项目相同的 `classpath` 下运行 `org.springframework.boot.devtools.RemoteSpringApplication`。
把要连接的远程 URL 作为必须参数传入。

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

- 在 Run 菜单中选择选择 `Run Configurations...`。
- 创建一个新的 `Java Application` “`launch configuration`”。
- 浏览 `my-app` 项目。
- 使用 `org.springframework.boot.devtools.RemoteSpringApplication` 作为主类。
- 将 `https://myapp.cfapps.io` 作为 `Program arguments` (或者任何远程 URL) 传入。

运行的远程客户端将如下所示：

```
2015-06-10 18:25:06.632  INFO 14938 --- [           main]
o.s.b.devtools.RemoteSpringApplication : Starting RemoteSpringApplication on
pwmpb with PID 14938 (/Users/pwebb/projects/spring-boot/code/spring-boot-
project/spring-boot-devtools/target/classes started by pwebb in
/Users/pwebb/projects/spring-boot/code)
2015-06-10 18:25:06.671  INFO 14938 --- [           main]
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@2a17b7
b6: startup date [Wed Jun 10 18:25:06 PDT 2015]; root of context hierarchy
2015-06-10 18:25:07.043  WARN 14938 --- [           main]
o.s.b.d.r.c.RemoteClientConfiguration : The connection to
http://localhost:8080 is insecure. You should use a URL starting with
'https://'.
2015-06-10 18:25:07.074  INFO 14938 --- [           main]
o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port
35729
2015-06-10 18:25:07.130  INFO 14938 --- [           main]
o.s.b.devtools.RemoteSpringApplication : Started RemoteSpringApplication in
0.74 seconds (JVM running for 1.105)
```

由于远程客户端与实际应用使用的是同一个 classpath, 因此可以直接读取应用的 properties. 这也是 `spring.devtools.remote.secret` 属性为什么能被读取和传递给服务器进行身份验证的原因.



建议使用 <https://> 作为连接协议，以便加密传输并防止密码被拦截。



如果您需要通过代理来访问远程应用，请配置
`spring.devtools.remote.proxy.host` 和
`spring.devtools.remote.proxy.port` 属性。

远程更新

远程客户端使用了与本地重启相同的方式来监控应用 `classpath` 下发生的更改。

任何更新的资源将被推送到远程应用和触发重启（如果要求）。

如果您正在迭代一个使用了本地没有的云服务的功能，这可能会非常有用。

通常远程更新和重启比完全重新构建和部署的周期要快得多。

在较慢的开发环境中，可能会发生静默期不够的情况，并且类中的更改可能会分为几批。

第一批类更改上传后，服务器将重新启动。由于服务器正在重新启动

，因此下一批不能发送到应用程序。

这通常通过 `RemoteSpringApplication` 日志中的警告来证明，

即有关上载某些类失败的消息，然后进行重试。但是，这也可能导致应用程序代码不一致

，并且在上传第一批更改后无法重新启动。

如果您经常观察到此类问题，请尝试将 `spring.devtools.restart.poll-interval` 和 `spring.devtools.restart.quiet-period` 参数增加到适合您的开发环境的值：请参阅配置文件系统监视器 部分以配置这些属性。



文件只有在远程客户端运行时才被监控。

如果您在启动远程客户端之前更改了文件，文件将不会被推送到远程服务器。

4.9. 打包生产应用

可执行 `jar` 可用于生产部署，它们是独立（`self-contained`, 独立、自包含）的，同样也适合云部署。

针对其他生产就绪功能，比如健康、审计和 REST 或者 JMX 端点指标，可以添加 `spring-boot-actuator`。有关这方面的详细信息，请参见 [Spring Boot Actuator: 生产就绪功能](#)。

4.10. 下一步

您现在应该知道如何使用 Spring Boot 以及应该遵循哪些最佳实践。

接下来您可以深入地了解 [Spring Boot 特性](#)，或者您也可以跳过下一部分直接阅读“[生产就绪功能](#)”方面的内容。

Chapter 5. Spring Boot 特性

本部分将介绍 Spring Boot 相关的细节内容。在这里，您可以学习到可能需要使用和自定义的主要功能。您如果还没有做好充分准备，可能需要阅读“[入门](#)”和“[使用 Spring Boot](#)”，以便打下前期基础。

5.1. SpringApplication

`SpringApplication` 类提供了一种可通过运行 `main()` 方法来启动 Spring 应用的简单方式。多数情况下，您只需要委托给静态的 `SpringApplication.run` 方法：

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

当应用启动时，您应该会看到类似以下的内容输出：

INFO 级别的日志信息，包括一些应用启动相关信息。如果您需要修改

INFO 日志级别,请参考[日志等级](#).

使用主应用程序类包中的实现版本来确定应用程序版本. 可以通过将 `spring.main.log-startup-info` 设置为 `false` 来关闭启动信息记录. 这还将关闭对应用程序 `active` 配置文件的日志记录.



要在启动期间添加其他日志记录,可以在 `SpringApplication` 的子类中重写 `logStartupInfo(boolean)`.

5.1.1. 启动失败

如果您的应用无法启动,注册的 `FailureAnalyzers`

可能会提供有相关的错误信息和解决问题的具体方法. 例如,如果您在已经被占用的 `8080` 端口上启动了一个 `web` 应用,会看到类似以下的错误信息:

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

Description:

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

Action:

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



Spring Boot 提供了许多的 `FailureAnalyzer` 实现,
您也可以[添加自己的实现](#).

如果没有失败分析器能够处理的异常,您仍然可以显示完整的条件报告以便更好地了解出现的问题.
. 为此,您需要针对

`org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener` 启用 `debug` 属性 或者开启 `DEBUG` 日志.

例如,如果您使用 `java -jar` 运行应用,可以按以下方式启用 `debug` 属性:

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

5.1.2. 延迟初始化

`SpringApplication` 允许延迟地初始化应用程序。启用延迟初始化后，将根据需要创建 bean，而不是在应用程序启动期间创建bean。因此，启用延迟初始化可以减少应用程序启动所需的时间。在Web应用程序中，启用延迟初始化将导致许多与Web相关的Bean直到收到HTTP请求后才被初始化。

延迟初始化的缺点是，它可能会延迟发现应用程序问题的时间。如果错误配置的Bean延迟初始化，则启动期间将不再发生故障，并且只有在初始化Bean时问题才会变得明显。还必须注意确保JVM有足够的内存来容纳所有应用程序的bean，而不仅仅是启动期间初始化的bean。由于这些原因，默认情况下不会启用延迟初始化，因此建议在启用延迟初始化之前先对JVM的堆大小进行微调。

可以使用 `SpringApplicationBuilder` 上的 `lazyInitialization` 方法或 `SpringApplication` 上的 `setLazyInitialization` 方法以编程方式启用延迟初始化。另外，可以使用 `spring.main.lazy-initialization` 属性启用它，如以下示例所示：

```
spring:
  main:
    lazy-initialization: true
```



如果要在对应用程序其余部分使用延迟初始化时禁用某些bean的延迟初始化，则可以使用 `@Lazy(false)` 注解将它们的延迟属性显式设置为 `false`。

5.1.3. 自定义 banner

可以通过在 `classpath` 下添加一个 `banner.txt` 文件，或者将 `spring.banner.location` 属性指向该文件的位置来更改启动时打印的 banner。如果文件采用了非 UTF-8 编码，您可以设置 `spring.banner.charset` 来解决。除了文本文件，您还可以将 `banner.gif`、`banner.jpg` 或者 `banner.png` 图片文件添加到 `classpath` 下，或者设置 `spring.banner.image.location` 属性。指定的图片将会被转换成 ASCII 形式并打印在 banner 文本上方。

您可以在 `banner.txt` 文件中使用以下占位符：

Table 4. Banner 变量

变量	描述
<code> \${application.version}</code>	您的应用版本号, 声明在 <code>MANIFEST.MF</code> 中. 例如, <code>Implementation-Version: 1.0</code> 将被打印为 <code>1.0</code> .
<code> \${application.formatted-version}</code>	您的应用版本号, 声明在 <code>MANIFEST.MF</code> 中 , 格式化之后打印 (用括号括起来, 以 <code>v</code> 为前缀) , 例如 <code>(v1.0)</code> .
<code> \${spring-boot.version}</code>	您使用的 Spring Boot 版本. 例如 <code>2.4.5</code> .
<code> \${spring-boot.formatted-version}</code>	您使用的 Spring Boot 版本格式化之后显示 (用括号括起来, 以 <code>v</code> 为前缀) . 例如 <code>(v2.4.5)</code> .
<code> \${Ansi.NAME} (or \${AnsiColor.NAME}, \${AnsiBackground.NAME}, \${AnsiStyle.NAME})</code>	其中 <code>NAME</code> 是 ANSI 转义码的名称. 有关详细信息, 请参阅 <code>AnsiPropertySource</code> .
<code> \${application.title}</code>	您的应用标题, 声明在 <code>MANIFEST.MF</code> 中, 例如 <code>Implementation-Title: MyApp</code> 打印为 <code>MyApp</code> .



如果您想以编程的方式生成 banner, 可以使用
 `SpringApplication.setBanner(...)` 方法. 使用
`org.springframework.boot.Banner` 接口并实现自己的
`printBanner()` 方法.

您还可以使用 `spring.main.banner-mode` 属性来确定是否必须在 `System.out`
`(console)` 上打印 banner, 还是使用日志记录器 (`log`) 或者都不打印 (`off`).

打印的 banner 的单例 bean 名为 `springBootBanner` .

只有在使用 Spring Boot 启动时, `$ {application.version}` 和 `${application.formatted-version}` 属性才可用.

如果您运行的是未打包的 jar 并以 `java -cp <classpath> <mainclass>` 开头, 则无法解析这些值.



这就是为什么我们建议您始终使用通过 `java org.springframework.boot.loader.JarLauncher` 来启动未打包的 jar 的原因. 这将在构建类路径并启动您的应用程序之前初始化 `application.*` 变量.

5.1.4. 自定义 SpringApplication

如果 `SpringApplication` 的默认设置不符合您的想法, 您可以创建本地实例进行定制化.

例如, 要关闭 banner, 您可以这样:

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



传入 `SpringApplication` 的构造参数是 spring bean 的配置源.

大多情况下是引用 `@Configuration` 类, 但您也可以引用 XML 配置或者被扫描的包.

也可以使用 `application.properties` 文件配置 `SpringApplication`. 有关详细信息, 请参见 [外部化配置](#).

关于配置选项的完整列表, 请参阅 [SpringApplication Javadoc](#).

5.1.5. Fluent Builder API

如果您需要构建一个有层级关系的 `ApplicationContext` (具有父/子关系的多上下文), 或者偏向使用 `fluent` (流式) 构建器 API, 可以使用 `SpringApplicationBuilder`.

`SpringApplicationBuilder` 允许您链式调用多个方法, 包括能创建出具有层次结构的

`parent` 和 `child` 方法.

例如:

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



创建层级的 `ApplicationContext` 时有部分限制, 比如 Web 组件必须包含在子上下文中, 并且相同的 `Environment` 将作用于父子上下文. 有关详细信息, 请参阅 [SpringApplicationBuilder Javadoc](#).

5.1.6. 应用程序的可用性

在平台上部署后, 应用程序可以使用诸如 [Kubernetes Probes](#) 之类的基础结构向平台提供有关其可用性的信息. Spring Boot 对常用的 “liveness” 和 “readiness” 可用性状态提供了开箱即用的支持. 如果您使用了 Spring Boot 的 “actuator” 支持, 则这些状态将显示为运行状况端点组.

另外, 您还可以通过将 `ApplicationAvailability` 接口注入到您自己的 bean 中来获取可用性状态.

Liveness State

应用程序的 “Liveness” 状态表明其内部是否正常运行, 或者在当前出现故障时自行恢复. 损坏的 “Liveness” 状态意味着应用程序处于无法恢复的状态, 并且应重新启动应用程序.



通常, “Liveness” 状态不应基于外部检查 (例如 [Health checks](#)). 如果确实如此, 则发生故障的外部系统 (数据库, Web API, 外部缓存) 将触发整个平台的大量重启和级联故障.

Spring Boot 应用程序的内部状态主要由 `Spring ApplicationContext` 表示. 如果应用程序上下文已成功启动, 则 Spring Boot 会假定该应用程序处于有效状态. 刷新上下文后, 应用程序即被视为活动应用程序, 请参阅 [Spring Boot](#)

应用程序生命周期和相关的应用程序事件.

Readiness State

应用程序的“Readiness”状态告诉应用程序是否已准备好处理流量。失败的“Readiness”状态告诉平台当前不应将流量路由到应用程序。这通常发生在启动过程中，正在处理 `CommandLineRunner` 和 `ApplicationRunner` 组件时，或者在应用程序认为它太忙而无法获得额外流量的情况下。

一旦调用了应用程序和命令行运行程序，就认为该应用程序已准备就绪，请参阅 [Spring Boot 应用程序生命周期和相关的应用程序事件](#)。



预期在启动期间运行的任务应由 `CommandLineRunner` 和 `ApplicationRunner` 组件执行，而不是使用 Spring 组件生命周期回调（如 `@PostConstruct`）执行。

管理应用程序可用性状态

通过注入 `ApplicationAvailability` 接口并调用其方法，应用程序组件可以随时检索当前的可用性状态。应用程序通常会希望监听状态更新或更新应用程序的状态。

例如，我们可以将应用程序的“Readiness”状态导出到文件中，以便 Kubernetes 的“exec Probe”可以查看此文件：

```

@Component
public class ReadinessStateExporter {

    @EventListener
    public void onStateChange(AvailabilityChangeEvent<ReadinessState> event) {
        switch (event.getState()) {
            case ACCEPTING_TRAFFIC:
                // create file /tmp/healthy
                break;
            case REFUSING_TRAFFIC:
                // remove file /tmp/healthy
                break;
        }
    }
}

```

当应用程序崩溃且无法恢复时，我们还可以更新应用程序的状态：

```

@Component
public class LocalCacheVerifier {

    private final ApplicationEventPublisher eventPublisher;

    public LocalCacheVerifier(ApplicationEventPublisher eventPublisher) {
        this.eventPublisher = eventPublisher;
    }

    public void checkLocalCache() {
        try {
            //...
        }
        catch (CacheCompletelyBrokenException ex) {
            AvailabilityChangeEvent.publish(this.eventPublisher, ex,
LivenessState.BROKEN);
        }
    }
}

```

Spring Boot 通过 [Kubernetes HTTP probes for "Liveness" and "Readiness"](#) with [Actuator Health Endpoints](#). 您可以在专用部分中获得[有关在 Kubernetes 上部署 Spring Boot 应用程序的更多指南](#).

5.1.7. 应用程序事件与监听器

除了常见的 Spring Framework 事件,比如 `ContextRefreshedEvent`,`SpringApplication` 还会发送其他应用程序事件.

在 `ApplicationContext` 创建之前,实际上触发了一些事件,因此您不能像 `@Bean` 一样注册监听器. 您可以通过 `SpringApplication.addListeners(...)` 或者 `SpringApplicationBuilder.listeners(...)` 方法注册它们. 如果您希望无论应用使用何种创建方式都能自动注册这些监听器,您都可以将 `META-INF/spring.factories` 文件添加到项目中,并使用 `org.springframework.context.ApplicationListener` 属性键指向您的监听器. 比如:

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

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

当您运行应用时,应用程序事件将按照以下顺序发送:

1. 在开始应用开始运行但还没有进行任何处理时(除了注册监听器和初始化器[`initializer`]),将发送 `ApplicationStartingEvent`.
2. 当 `Environment` 被上下文使用,但是在上下文创建之前,将发送 `ApplicationEnvironmentPreparedEvent`.
3. 准备 `ApplicationContext` 并调用 `ApplicationContextInitializers` 之后但在加载任何bean定义之前,将发送 `ApplicationContextInitializedEvent`.
4. 开始刷新之前,bean 定义被加载之后发送 `ApplicationPreparedEvent`.
5. 在上下文刷新之后且所有的应用和命令行运行器(`command-line runner`)被调用之前发送 `ApplicationStartedEvent`.
6. 紧随其后发送带有 `LivenessState.CORRECT` 的 `AvailabilityChangeEvent`,以指示该应用程序处于活动状态.
7. 在应用程序和命令行运行器(`command-line runner`)被调用之后,将发出,将发送

`ApplicationReadyEvent`.

8. 随即在 `ReadinessState.ACCEPTING_TRAFFIC` 之后发送 `AvailabilityChangeEvent`, 以指示该应用程序已准备就绪, 可以处理请求.
9. 如果启动时发生异常, 则发送 `ApplicationFailedEvent`.

上面的列表仅包含绑定到 `SpringApplication` 的 `SpringApplicationEvents`.

除这些以外, 以下事件也在 `ApplicationPreparedEvent` 之后和

`ApplicationStartedEvent` 之前发布:

1. `WebServer` 准备就绪后, 将发送 `WebServerInitializedEvent`.
`ServletWebServerInitializedEvent` 和
`ReactiveWebServerInitializedEvent` 分别是 `Servlet` 和 `Reactive` 变量.
2. 刷新 `ApplicationContext` 时, 将发送 `ContextRefreshedEvent` 事件.



您可能不会经常使用应用程序事件, 但了解他们的存在还是很有必要的.
在框架内部, Spring Boot 使用这些事件来处理各种任务.



默认情况下, 事件监听器不应该运行可能很长的任务, 因为它们在同一个线程中执行. 考虑改用 `application and command-line runners`.

应用程序事件发送使用了 `Spring Framework` 的事件发布机制.

该部分机制确保在子上下文中发布给监听器的事件也会发布给所有祖先上下文中的监听器. 因此, 如果您的应用程序使用有层级结构的 `SpringApplication` 实例, 则监听器可能会收到同种类型应用程序事件的多个实例.

为了让监听器能够区分其上下文事件和后代上下文事件, 您应该注入其应用程序上下文, 然后将注入的上下文与事件的上下文进行比较. 可以通过实现 `ApplicationContextAware` 来注入上下文, 如果监听器是 `bean`, 则使用 `@Autowired` 注入上下文.

5.1.8. Web 环境

`SpringApplication` 试图为您创建正确类型的 `ApplicationContext`. 确定 `WebApplicationType` 的算法非常简单:

- 如果存在 Spring MVC，则使用 `AnnotationConfigServletWebServerApplicationContext`
- 如果 Spring MVC 不存在且存在 Spring WebFlux，则使用 `AnnotationConfigReactiveWebServerApplicationContext`
- 否则，使用 `AnnotationConfigApplicationContext`

这意味着如果您在同一个应用程序中使用了 Spring MVC 和 Spring WebFlux 中的新 `WebClient`，默认情况下将使用 Spring MVC。您可以通过调用 `setWebApplicationType(WebApplicationType)` 修改默认行为。

也可以调用 `setApplicationContextClass(...)` 来完全控制 `ApplicationContext` 类型。



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

5.1.9. 访问应用程序参数

如果您需要访问从 `SpringApplication.run(...)` 传入的应用程序参数，可以注入一个 `org.springframework.boot.ApplicationArguments` bean。
`ApplicationArguments` 接口提供了访问原始 `String[]` 参数以及解析后的 `option` 和 `non-option` 参数的方法：

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

@Component
public class MyBean {

    @Autowired
    public MyBean(ApplicationArguments args) {
        boolean debug = args.containsOption("debug");
        List<String> files = args.getNonOptionArgs();
        // if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]
    }

}
```



Spring Boot 还向 Spring Environment 注册了一个 `CommandLinePropertySource`. 这允许您可以使用 `@Value` 注解注入单个应用参数.

5.1.10. 使用 `ApplicationRunner` 或 `CommandLineRunner`

如果您需要在 `SpringApplication` 启动时运行一些代码, 可以实现 `ApplicationRunner` 或者 `CommandLineRunner` 接口. 这两个接口的工作方式是一样的, 都提供了一个单独的 `run` 方法, 它将在 `SpringApplication.run(...)` 完成之前调用.



这个契约非常适合那些应该在应用程序启动后但在它开始接受流量之前运行的任务.

`CommandLineRunner` 接口提供了访问应用程序字符串数组形式参数的方法, 而 `ApplicationRunner` 则使用了上述的 `ApplicationArguments` 接口. 以下示例展示 `CommandLineRunner` 和 `run` 方法的使用:

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

@Component
public class MyBean implements CommandLineRunner {

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

如果您定义了多个 `CommandLineRunner` 或者 `ApplicationRunner` bean, 则必须指定调用顺序, 您可以实现 `org.springframework.core.Ordered` 接口, 也可以使用 `org.springframework.core.annotation.Order` 注解解决顺序问题.

5.1.11. 应用程序退出

每个 `SpringApplication` 注册了一个 JVM 关闭钩子, 以确保 `ApplicationContext` 在退出时可以优雅关闭. 所有标准的 Spring 生命周期回调 (比如 `DisposableBean` 接口, 或者 `@PreDestroy` 注解) 都可以使用.

此外,如果希望在调用 `SpringApplication.exit()` 时返回特定的退出码,则 bean 可以实现 `org.springframework.boot.ExitCodeGenerator` 接口. 之后退出码将传递给 `System.exit()` 以将其作为状态码返回,如示例所示:

```
@SpringBootApplication
public class ExitCodeApplication {

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

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

此外,`ExitCodeGenerator` 接口可以通过异常实现. 遇到这类异常时, Spring Boot 将返回实现的 `getExitCode()` 方法提供的退出码.

5.1.12. 管理功能

可以通过指定 `spring.application.admin.enabled` 属性来为应用程序启用管理相关的功能. 其将在 `MBeanServer` 平台上暴露 `SpringApplicationAdminMXBean`. 您可以使用此功能来远程管理 Spring Boot 应用. 该功能对服务包装器的实现也是非常有用的.



如果您想知道应用程序在哪一个 HTTP 端口上运行,请使用 `local.server.port` 键获取该属性.

5.1.13. Application Startup tracking

在应用程序启动期间, `SpringApplication` 和 `ApplicationContext` 执行许多与应用程序生命周期、bean 生命周期甚至处理应用程序事件相关的任务. 使用 `ApplicationStartup`, Spring 框架允许你用 `allows you to track the application startup sequence with StartupSteps` 跟踪应用程序的启动顺序.

收集这些数据可以用于分析目的，或者只是为了更好地理解应用程序启动过程。

您可以在设置 `SpringApplication` 实例时选择 `ApplicationStartup` 实现。

例如，要使用 `BufferingApplicationStartup`，您可以编写：

```
public static void main(String[] args) {
    SpringApplication app = new SpringApplication(MySpringConfiguration.class);
    app.setApplicationStartup(new BufferingApplicationStartup(2048));
    app.run(args);
}
```

第一个可用的实现是由 `Spring` 框架提供的 `FlightRecorderApplicationStartup`。

它将特定于 `Spring` 的启动事件添加到 Java Flight Recorder

会话中，用于分析应用程序，并将其 `Spring` 上下文生命周期与 JVM 事件（如

`allocations`、`gc`、类加载.....）关联起来。一旦配置好，你就可以通过启用 Flight Recorder 运行应用程序来记录数据：

```
$ java -XX:StartFlightRecording:filename=recording.jfr,duration=10s -jar
demo.jar
```

Spring Boot 附带 `BufferingApplicationStartup`

，这个实现的目的是缓冲启动步骤，并将它们抽取到外部指标系统中。

应用程序可以在任何组件中请求 `BufferingApplicationStartup` 类型的bean。

此外，Spring Boot Actuator 将暴露一个 `startup` 点以将此信息公开为 JSON 文档。

5.2. 外部化配置

Spring Boot 可以让您的配置外部化，以便可以在不同环境中使用相同的应用程序代码。

您可以使用各种外部配置源，包括 Java properties 文件、YAML 文件、环境变量或者命令行参数。

可以使用 `@Value` 注解将属性值直接注入到 bean 中，可通过 Spring 的 `Environment` 访问，或者通过 `@ConfigurationProperties` 绑定到结构化对象。

Spring Boot 使用了一个非常特别的 `PropertySource` 指令，用于智能覆盖默认值。

属性将按照以下顺序处理（下面的值覆盖前面的值）：

1. 默认属性 (通过设置 `SpringApplication.setDefaultProperties` 指定).
2. `@Configuration` 类上的 `@PropertySource` 注解.
请注意，在刷新应用程序上下文之前，不会将此类属性源添加到 `Environment` 中。
现在配置某些属性(如 `logging.*` 和 `spring.main.*`)已经太晚了。
这些属性在刷新开始之前就已读取。
3. Config data (例如 `application.properties` 文件)
4. 只有 `random.*` 属性的 `RandomValuePropertySource`.
5. 操作系统环境变量.
6. Java System 属性 (`System.getProperties()`).
7. 来自 `java:comp/env` 的 JNDI 属性 .
8. `ServletContext` 初始化参数.
9. `ServletConfig` 初始化参数.
10. 来自 `SPRING_APPLICATION_JSON` 的属性 (嵌入在环境变量或者系统属性 [system property] 中的内联 JSON) .
11. 命令行参数.
12. 测试中的 `properties`.
13. 在测试中使用到的 `properties` 属性, 可以是 `@SpringBootTest` 和
[用于测试应用程序某部分的测试注解](#).
14. 在测试中使用到的 `@TestPropertySource` 注解.
15. 当 `devtools` 被激活, `$HOME/.config/spring-boot` 目录中的 [Devtools 全局设置属性](#).

配置数据文件按以下顺序进行：

1. 在已打包的 `jar` 内部的 `Application properties` 文件
(`application.properties` 和 YAML 变量).
2. 在已打包的 `jar` 内部的指定 `profile` 的应用属性文件 (`application-{profile}.properties` 和 YAML 变量).
3. 在已打包的 `jar` 外部的 `Application properties` 文件

(`application.properties` 和 YAML 变量).

4. 在已打包的 `jar` 外部的 指定 `profile` 的应用属性文件 (`application-{profile}.properties` 和 YAML 变量).



建议您在整个应用程序中坚持使用一种格式. 如果在相同的位置有 `.properties` 和 `.yml` 格式的配置文件, 则 `.properties` 优先.

举个例子, 假设开发的 `@Component` 使用了 `name` 属性, 可以这样:

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

@Component
public class MyBean {

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

    // ...
}
```

在您的应用程序的 `classpath` 中 (比如在 `jar` 中), 您可以有一个 `application.properties`, 它为 `name` 提供了一个合适的默认属性值. 当在新环境中运行时, 您可以在 `jar` 外面提供一个 `application.properties` 来覆盖 `name`. 对于一次性测试, 您可以使用命令行指定形式启动 (比如 `java -jar app.jar --name="Spring"`).



`env` 和 `configprops` 端点在确定属性的特定值时很有用.

您可以使用这两个端点来诊断意外的属性值. 有关详细信息, 请参见 "生产就绪" 部分.

5.2.1. 访问命令行属性

默认情况下, `SpringApplication` 将任何命令行选项参数(即以 `--` 开头的参数, 例如 `--server.port=9000`)转换为属性, 并将它们添加到 Spring 环境中. 如前所述, 命令行属性总是优先于基于文件的属性源.

如果不希望将命令行属性添加到环境中，可以使用

`SpringApplication.setAddCommandLineProperties(false)` 禁用它们.

5.2.2. JSON Application Properties

环境变量和系统属性通常有限制，这意味着某些属性名不能使用。为了帮助实现这一点，`Spring Boot` 允许您将属性块编码到单个 `JSON` 结构中。

当应用程序启动时，任何 `spring.application.json` 或 `SPRING_APPLICATION_JSON` 属性都将被解析并添加到 `Environment` 中。

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

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

在此示例中，您可以在 `Spring Environment` 中使用 `acme.name=test`，也可以在系统属性（System property）中将 `JSON` 作为 `spring.application.json` 属性提供：

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

或者以命令行参数形式：

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

如果您正在部署到一个经典的应用程序服务器，您还可以使用名为 `java:comp/env/spring.application.json` 的 JNDI 变量。



尽管 `JSON` 中的 `null` 被添加到结果属性集中，但 `PropertySourcesPropertyResolver` 将 `null` 属性视为缺失值。这意味着 `JSON` 无法使用 `null` 覆盖在属性集中具有低优先级的属性。

5.2.3. 外部应用程序属性

应用程序启动时，`Spring Boot` 将自动从以下位置查找并加载 `application.properties`

和 `application.yaml` 文件：

1. `classpath` 根路径
2. `classpath` 上的 `/config` 包
3. 当前目录
4. 当前目录下的 `/config` 子目录
5. Immediate child directories of the `/config` subdirectory

该列表按优先级排序(较低项的值覆盖较早项的值). 加载文件中的文档作为 `PropertySources` 添加到 `Spring`Environment`` 中.

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` 环境属性来引用一个显式位置 (以逗号分隔的目录位置或文件路径列表) . 以下示例展示了如何指定其他文件名:

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

以下示例展示了如何指定两个位置:

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



如果位置是可选的, 可以使用前缀 `optional:`, 并且您不介意它们是否存在.



`spring.config.name` `spring.config.location` 和 `spring.config.additional-location` 在程序启动早期就用来确定哪些文件必须加载. 它们必须定义为一个环境属性(通常是一个 `OS` 环境变量、一个系统属性或一个命令行参数).

如果 `spring.config.location` 包含目录 (而不是文件), 则它们应该以 `/` 结尾

(并且在运行期间,在加载之前追加从 `spring.config.name` 生成的名称,包括指定 `profile` 的文件名)。`spring.config.location` 中指定的文件按原样使用,不支持指定 `profile` 形式,并且可被任何指定 `profile` 的文件的属性所覆盖.无论是直接指定还是包含在目录中,配置文件都必须在名称中包含文件扩展名.典型扩展名是 `.properties`,`.yaml` 和 `.yml`

当指定多个 `locations` 时,后面的 `locations` 可以覆盖前面的 `locations`.

使用了 `spring.config.location` 配置自定义配置位置时,默认位置配置将被替代.

例如,如果 `spring.config.location` 配置为 `optional:classpath:/custom-config/,optional:file:./custom-config/` 时,完整位置集是:

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

或者,当使用 `spring.config.additional-location` 配置自定义配置位置时,除了使用默认位置外,还会使用它们.例如,如果 `spring.config.additional-location` 配置为 `optional:classpath:/custom-config/,optional:file:./custom-config/`,完整位置集是:

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

该搜索顺序允许您在一个配置文件中指定默认值,然后有选择地覆盖另一个配置文件中的值.您可以在 `application.properties` (或您使用 `spring.config.name` 指定的其他文件) 中的某个默认位置为应用程序提供默认值.之后,在运行时,这些默认值将被自定义位置中的某个文件所覆盖.



如果您使用的是环境变量而不是系统属性,大部分操作系统都不允许使用 . 分隔的键名,但您可以使用下划线来代替 (例如, 使用 `SPRING_CONFIG_NAME` 而不是 `spring.config.name`) . 查看 [从环境变量绑定](#) 获取更多细节信息.



如果您的应用程序运行在 `servlet` 容器或应用服务器中,则可以使用 JNDI 属性 (`java:comp/env`) 或 `servlet` 上下文初始化参数来代替环境变量或系统属性. (in `java:comp/env`)

Optional Locations

默认情况下,当指定的配置数据位置不存在时, Spring Boot 将抛出 `ConfigDataLocationNotFoundException` 并且您的应用程序将不会启动.

如果您想指定一个位置,但如果它并不总是存在,您也不介意,可以使用 `optional:` 前缀. 你可以在 `spring.config.location` 和 `spring.config.additional-location` 属性,以及 `spring.config.import` 声明.

例如, `optional:file:./myconfig.properties` 的 `spring.config.import` 值. 属性允许您的应用程序启动,即使 `myconfig.properties` 文件丢失.

如果你想忽略所有的 `ConfigDataLocationNotFoundExceptions` 而总是继续启动你的应用程序,你可以使用 `spring.config.on-not-found` 属性. 使用 `SpringApplication.setDefaultProperties(...)` 或使用系统/环境变量将该值设置为 `ignore`.

Wildcard Locations(通配符位置)

如果配置文件的位置包含最后一个路径段的 `*`, 则将其视为通配符位置. 加载配置时, 通配符会扩展,以便检查子目录. 当存在多个配置属性源时, 通配符位置在诸如 `Kubernetes` 之类的环境中特别有用.

例如,如果您具有一些 `Redis` 配置和某些 `MySQL` 配置,则可能希望将这两个配置分开,同时要求这两个配置都存在于该应用程序可以绑定到的``application.properties`` 中. 这可能会导致两个单独的 `application.properties` 文件安装在不同的位置,例如 `/config/redis/application.properties` 和 `/config/mysql/application.property`

ies. 在这种情况下,通配符位置为 `config/*/` 将导致两个文件都被处理.

默认情况下, Spring Boot 包括默认搜索位置中的配置 `config/*/`. 这意味着将搜索 jar 之外 `/config` 目录的所有子目录.

您可以使用 `spring.config.location` 和 `spring.config.additional-location` 属性使用通配符位置.



通配符位置必须仅包含一个 并以 / 结尾 (对于目录的搜索位置) 或 `*/<filename>` (对于文件的搜索位置)
.带通配符的位置根据文件名的绝对路径按字母顺序排序.



通配符位置仅适用于外部目录. 您不能在 `classpath:` 位置中使用通配符.

特定 Profile 的属性文件

除 `application.properties` 文件外,还可以使用以下命名约定定义特定 `profile` 的属性文件:`application-{profile}`. 例如, 如果您的应用程序激活了一个名为 `prod` 的 `profile` 文件并使用 YAML 文件, 那么这两个 `application.yml` 和 `application-prod.yml` 将被加载.

特定 `profile` 的属性文件从与标准 `application.properties` 相同的位置加载, 特定 `profile` 的属性文件始终覆盖非特定文件. 如果指定了多个配置文件, 则应用 `last-wins` 策略 (优先采取最后一个). 例如, 如果由 `configprop:spring.profiles.active[]` 指定 `prod, live profiles`, `application-prod.properties` 中的属性值将被 `application-live.properties` 中的值覆盖

`Environment` 有一组默认配置文件 (默认情况下为 `default`) ,如果未设置激活的 (`active`) `profile`, 则使用这些配置文件. 换句话说,如果没有显式激活 `profile`,则会加载 `application-default` 中的属性.



属性文件只加载一次. 如果您已经直接 `imported` 了特定于配置文件的属性文件, 那么它将不会被再次导入.

导入其他数据

应用程序属性可以使用 `Spring.config.import` 属性导入来自其他位置的其他配置数据。导入是在发现它们时进行处理的，并且被视为直接插入在声明导入的文档下面的附加文档。

例如，您的 ClassPath `application.properties` 文件中可能具有以下内容：

```
spring:  
  application:  
    name: "myapp"  
  config:  
    import: "optional:file:/dev.properties"
```

这将触发当前目录中 `dev.properties` 文件的导入(如果存在这样的文件)。导入的 `dev.properties` 中的值将优先于触发导入的文件。在上面的例子中，`dev.properties` 可以将 `spring.application.name` 重定义为不同的值。

无论声明了多少次，导入都只会被导入一次。在 `properties/yaml` 文件中的单个文档中定义导入的顺序并不重要。例如，下面两个例子产生相同的结果：

```
spring:  
  config:  
    import: my.properties  
my:  
  property: value
```

```
my:  
  property: value  
spring:  
  config:  
    import: my.properties
```

在上述两个示例中，来自 `my.properties` 文件的值将优先于触发其导入的文件。

可以在单个 `spring.config.import` key 指定多个位置。位置将按照它们定义的顺序进行处理，稍后的导入优先。

Spring Boot 包括可插拔 API，允许支持各种不同的位置地址。默认情况下，您可以导入 Java 属性，yaml 和 “[configuration trees](#)”。



第三方 JAR 可以提供对附加技术的支持(不需要文件是本地的)。例如，您可以想象配置数据来自外部存储，例如 Consul，Apache Zookeeper 或 Netflix Archaius。

如果要支持自己的位置，请参阅 [org.springframework.boot.context.config](#) 包中的 `ConfigDataLocationResolver` 和 `ConfigDataLoader` 类。

Importing Extensionless Files

一些云平台不能为挂载的文件添加文件扩展名。要导入这些无扩展文件，您需要给 Spring Boot 一个提示，以便它知道如何加载它们。您可以通过将扩展提示放在方括号中来实现这一点。

例如，假设您有一个 `/etc/config/myconfig` 文件，希望导入为 yaml。您可以使用以下的 `application.properties` 导入：

```
spring:
  config:
    import: "file:/etc/config/myconfig[.yaml]"
```

使用配置树

在云平台上运行应用程序（例如Kubernetes），您通常需要阅读平台提供的配置值。使用环境变量来实现这类目的并不少见，但是这样做可能会有缺点，特别是在值应该保密的情况下。

作为环境变量的替代方案，许多云平台现在允许您将配置映射到安装的数据卷中。例如，Kubernetes 可以同时卷载 `ConfigMaps` 和 `Secrets`。

有两种常见的卷挂载模式可以使用：

1. 单个文件包含一组完整的属性(通常以YAML的形式编写)。

2. 多个文件被写入一个目录树，文件名成为 ‘key’，内容成为 ‘value’.

对于第一种情况，您可以像 [上面描述](#) 的那样直接使用 `spring.config.import` 导入 YAML 或 Properties 文件。对于第二种情况，您需要使用 `configtree:` 前缀，以便 Spring Boot 知道它需要将所有文件作为属性公开。

举个例子，假设 Kubernetes 已经挂载了下面的卷：

```
etc/
  config/
    myapp/
      username
      password
```

`username` 文件的内容将是一个配置值，`password` 的内容将是一个 secret。

要导入这些属性，可以将以下内容添加到 `application.properties` 或 `application.yaml` 文件：

```
spring:
  config:
    import: "optional:configtree:/etc/config/"
```

然后，您可以像往常一样从环境中访问或注入 `myapp.username` 和 `myapp.password`。



根据所期望的内容，配置树值可以绑定到字符串 `String` 和 `byte[]` 类型。

如果有多个配置树要从同一个父文件夹导入，可以使用通配符快捷方式。任何以 `/*` 结尾的 `configtree: location` 将导入所有直接子树作为配置树。

例如，给定以下卷：

```
etc/
  config/
    dbconfig/
      db/
        username
        password
    mqconfig/
      mq/
        username
        password
```

你可以使用 `configtree:/etc/config/*` 作为导入位置：

```
spring:
  config:
    import: "optional:configtree:/etc/config/*"
```

这将会添加 `db.username`, `db.password`, `mq.username` 和 `mq.password` 属性.



使用通配符加载的目录按字母顺序排序.

如果您需要不同的顺序, 那么您应该将每个位置作为单独的导入列出

配置树也可以用于 Docker secrets. 当一个 Docker 集群服务被授权访问一个 secrets 时, 这个 secrets 就会被安装到容器中. 例如, 如果一个 secrets 命名为 `db.password` 被挂载在 `/run/secrets/` 位置, 则可以使用以下内容使 `db.password` 可用于 Spring 环境:

```
spring:
  config:
    import: "optional:configtree:/run/secrets/"
```

属性中的占位符

`application.properties` 和 `application.yml` 中的值在使用时通过现有的 `Environment` 进行过滤, 因此您可以返回之前定义的值 (例如, 从系统属性). 标准的 `${name}` 属性占位符语法可以在值的任何地方使用.

例如, 下面的文件将 `app.description` 设置为 “MyApp is a Spring Boot

application":

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



您还可以使用此技术创建现有 Spring Boot 属性的简短形式。
有关详细信息,请参见 [使用简短命令行参数](#) .

处理多文档文件

Spring Boot 允许您将单个物理文件拆分为多个逻辑文档, 每个逻辑文件都是独立添加的.
文档按顺序处理, 从上到下处理. 后面的文档可以覆盖早期定义的属性.

对于 `application.yml` 文件, 使用标准 `yaml` 多文档语法. `---`
字符代表一个文档的结尾, 并开始下一个文档.

例如, 以下文件具有两个逻辑文件:

```
spring.application.name: MyApp  
---  
spring.config.activate.on-cloud-platform: kubernetes  
spring.application.name: MyCloudApp
```

对于 `application.properties` 文件, 特殊 `#---` 注释用于标记文档拆分:

```
spring.application.name=MyApp  
#---  
spring.config.activate.on-cloud-platform=kubernetes  
spring.application.name=MyCloudApp
```



属性文件分隔符必须没有任何前导空格, 并且必须恰好有三个连字符.
分隔符前后的行不能是注释.



多文档属性文件通常与激活属性一起使用，例如
`spring.config.activate.on-profile`. 有关详细信息，请参阅下一节.



无法使用 `@PropertySource` 或 `@TestPropertySource` 注解加载多文档属性文件.

Activation Properties

有时，只在满足某些条件时才激活给定的属性是有用的.
 例如，您可能拥有仅在特定概要文件激活时才相关的属性.

您可以使用 `spring.config.activate.*` 有条件地激活属性文档.

激活属性有以下几种：

只有在满足某些条件时，只能激活给定的属性有时有用.
 例如，您可能具有仅在特定配置文件处于活动状态时相关的属性.

您可以使用 `spring.config.activate.*` 有条件激活属性文档.

以下激活属性可用：

Table 5. activation properties

Property	Note
<code>on-profile</code>	A profile expression that must match for the document to be active.
<code>on-cloud-platform</code>	The <code>CloudPlatform</code> that must be detected for the document to be active.

例如，下面的命令指定第二个文档只有在 `Kubernetes` 上运行时是激活的，并且只有在“prod”或“staging”配置文件是激活的时候：

```

myprop:
  always-set
---
spring:
  config:
    activate:
      on-cloud-platform: "kubernetes"
      on-profile: "prod | staging"
myotherprop: sometimes-set

```

5.2.4. 加密属性

Spring Boot 没有为加密属性值提供任何内置支持, 然而, 它提供了修改 [Spring Environment](#) 包含的值所必需的钩子. [EnvironmentPostProcessor](#) 接口允许您在应用程序启动之前操作 [Environment](#). 有关详细信息, 请参见 [在启动前自定义 Environment 或 ApplicationContext](#).

如果您正在寻找一种可用于存储凭据和密码的安全方法, [Spring Cloud Vault](#) 项目支持在 [HashiCorp Vault](#) 中存储外部化配置.

5.2.5. 使用 YAML

[YAML](#) 是 JSON 的超集, 是一个可用于指定层级配置数据的便捷格式. 只要在 classpath 上有 [SnakeYAML](#) 库, [SpringApplication](#) 类就会自动支持 YAML 作为属性文件 (properties) 的替代.



如果使用 [starter](#), 则 [spring-boot-starter](#) 会自动提供 SnakeYAML.

使用 YAML 代替属性文件

YAML 文档需要从其分层格式转换为可与 [Spring Environment](#) 一起使用的平面结构. 例如, 考虑以下 YAML 文档:

```

environments:
  dev:
    url: https://dev.example.com
    name: Developer Setup
  prod:
    url: https://another.example.com
    name: My Cool App

```

为了从 **Environment** 访问这些属性，它们将被扁平化如下：

```

environments.dev.url=https://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=https://another.example.com
environments.prod.name=My Cool App

```

YAML 列表表示带有 **[index]** 下标引用的属性键。例如以下 YAML：

```

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

```

以上示例将转成以下属性：

```

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

```



使用 **[index]** 表示的属性可以通过 Spring Boot 的 **Binder** 类绑定到 Java **List** 或 **Set** 对象。有关更多细节，请参阅下面的“[类型安全的配置属性](#)”一节。



YAML文件不能通过使用 **@PropertySource** 或 **@TestPropertySource** 注解加载。
因此，在你需要以这种方式加载值的情况下，你需要使用一个属性文件。

Directly Loading YAML

Spring Framework 提供了两个方便的类，可用于加载 YAML 文档。

`YamlPropertiesFactoryBean` 以 `Properties` 的形式加载 YAML，而 `YamlMapFactoryBean` 以 `Map` 的形式加载 YAML。

你也可以使用 `YamlPropertySourceLoader` 类，如果你想加载 YAML 作为一个 Spring `PropertySource`。

5.2.6. 配置随机值

`RandomValuePropertySource` 对于注入随机值(例如，注入 `secrets` 或测试用例)很有用。它可以产生 `integers`, `longs`, `uuids`, or `strings`, 如下例所示：

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

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

5.2.7. 类型安全的配置属性

使用 `@Value("${property}")` 注解来注入配置属性有时会很麻烦, 特别是如果您使用了多个属性或者您的数据本质上是分层结构。Spring Boot 提供了另一种使用属性的方法, 该方法使用强类型的 `bean` 来管理和验证应用程序的配置, 如下所示：



另请参见[@Value 和类型安全的配置属性之间的区别](#)。

JavaBean 属性绑定

可以绑定一个声明标准 JavaBean 属性的 `bean`, 如以下示例所示：

```
package com.example;

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

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

@ConfigurationProperties("acme")
public class AcmeProperties {

    private boolean enabled;

    private InetAddress remoteAddress;

    private final Security security = new Security();

    public boolean isEnabled() { ... }

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

    public InetAddress getRemoteAddress() { ... }

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

    public Security getSecurity() { ... }

    public static class Security {

        private String username;

        private String password;

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

        public String getUsername() { ... }

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

        public String getPassword() { ... }

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

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

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

}

}
```

前面的 **POJO** 定义了以下属性：

- `acme.enabled`, 默认值为 `false`.
- `acme.remote-address`, 可以从 `String` 强制转换的类型.
- `acme.security.username`, 内嵌一个 `security` 对象, 其名称由属性名称决定.
特别是, 返回类型根本没有使用, 可能是 `SecurityProperties`.
- `acme.security.password`.
- `acme.security.roles`, `String` 集合. 默认为 `USER`.



Spring Boot 自动配置大量使用 `@ConfigurationProperties` 来轻松配置自动配置的 bean. 与自动配置类相似, Spring Boot 中可用的 `@ConfigurationProperties` 类仅供内部使用. 通过属性文件, YAML 文件, 环境变量等配置的映射到该类的属性是 public API, 但是该类本身的内容并不意味着可以直接使用.

`getter` 和 `setter` 通常是必需的, 因为绑定是通过标准的 Java Bean 属性描述符来完成, 就像在 Spring MVC 中一样. 以下情况可以省略 `setter`:



- `Map`, 只需要初始化, 就需要一个 `getter` 但不一定需要 `setter`, 因为它们可以被 `binder` 修改.
- 集合和数组可以通过一个索引 (通常使用 YAML) 或使用单个逗号分隔值 (属性) 进行访问. 最后一种情况必须使用 `setter`. 我们建议始终为此类型添加 `setter`. 如果初始化集合, 请确保它是可变的 (如上例所示) .
- 如果初始化嵌套的 POJO 属性 (如前面示例中的 `Security` 字段), 则不需要 `setter`. 如果您希望 `binder` 使用其默认构造函数动态创建实例, 则需要一个 `setter`.

有些人可能会使用 Project Lombok 来自动生成 `getter` 和 `setter`. 请确保 Lombok 不为此类型生成任何特定构造函数, 因为容器会自动使用它来实例化对象.

最后, 考虑到标准 Java Bean 属性, 不支持对静态属性的绑定.

构造函数绑定

上一节中的示例可以以不变的方式重写, 如下例所示:

```
package com.example;

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

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.ConstructorBinding;
import org.springframework.boot.context.properties.bind.DefaultValue;

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

    private final boolean enabled;
```

```

private final InetAddress remoteAddress;

private final Security security;

public AcmeProperties(boolean enabled, InetAddress remoteAddress, Security
security) {
    this.enabled = enabled;
    this.remoteAddress = remoteAddress;
    this.security = security;
}

public boolean isEnabled() { ... }

public InetAddress getRemoteAddress() { ... }

public Security getSecurity() { ... }

public static class Security {

    private final String username;

    private final String password;

    private final List<String> roles;

    public Security(String username, String password,
                    @DefaultValue("USER") List<String> roles) {
        this.username = username;
        this.password = password;
        this.roles = roles;
    }

    public String getUsername() { ... }

    public String getPassword() { ... }

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

}
}

```

在此设置中, **@ConstructorBinding** 注解用于指示应使用构造函数绑定。这意味着绑定器将期望找到带有您希望绑定的参数的构造函数。

@ConstructorBinding 类的嵌套成员 (例如上例中的 **Security**)

也将通过其构造函数进行绑定。

可以使用 `@DefaultValue` 指定默认值，并且将应用相同的转换服务将 `String` 值强制为缺少属性的目标类型。

默认情况下，如果没有属性绑定到 `Security`，则 `AcmeProperties` 实例的 `Security` 为 `null`。

如果您希望即使没有绑定任何属性都返回 `Security` 的非空实例，则可以使用空的 `@DefaultValue` 注解来这样做：

```
package com.example;
import java.net.InetAddress;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.ConstructorBinding;
import org.springframework.boot.context.properties.bind.DefaultValue;

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

    private final boolean enabled;

    private final InetAddress remoteAddress;

    private final Security security;

    public AcmeProperties(boolean enabled, InetAddress remoteAddress,
        @DefaultValue Security security) {
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
        this.security = security;
    }
}
```



要使用构造函数绑定，必须使用 `@EnableConfigurationProperties` 或配置属性扫描来启用该类。您不能对通过常规 Spring 机制创建的 bean 使用构造函数绑定（例如 `@Component` bean，通过 `@Bean` 方法创建的 bean 或使用 `@Import` 加载的 bean）



如果您的类具有多个构造函数，则还可以直接在应绑定的构造函数上使用 `@ConstructorBinding`.



不建议将 `java.util.Optional` 与 `@ConfigurationProperties` 一起使用，因为它主要是用作返回类型。因此，它不太适合配置属性注入。为了与其他类型的属性保持一致，如果确实声明了 `Optional` 属性并且没有任何值，则将绑定 `null` 而不是空的 `Optional`.

启用 `@ConfigurationProperties` 注解的类型

Spring Boot 提供了绑定 `@ConfigurationProperties` 类型并将其注册为 Bean 的基础架构。您可以逐类启用配置属性，也可以启用与组件扫描类似的方式进行配置属性扫描。

有时，用 `@ConfigurationProperties` 注解的类可能不适用于扫描，例如，如果您正在开发自己的自动配置，或者想要有条件地启用它们。在这些情况下，请使用 `@EnableConfigurationProperties` 注解 指定要处理的类型列表。可以在任何 `@Configuration` 类上完成此操作，如以下示例所示：

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

要使用配置属性扫描，请将 `@ConfigurationPropertiesScan` 注解 添加到您的应用程序。通常，它被添加到使用 `@SpringBootApplication` 注解的主应用程序类中，但可以将其添加到任何 `@Configuration` 类中。默认情况下，将从声明注解的类的包中进行扫描。如果要定义要扫描的特定程序包，可以按照以下示例所示进行操作：

```
@SpringBootApplication
@ConfigurationPropertiesScan({ "com.example.app", "org.acme.another" })
public class MyApplication { }
```



当以这种方式注册 `@ConfigurationProperties` bean 时,bean 具有一个固定格式的名称: `<prefix>-<fqn>`, 其中 `<prefix>` 是 `@ConfigurationProperties` 注解中指定的环境 key 前缀, `<fqn>` 是 bean 的完全限定类名. 如果注解未提供任何前缀, 则仅使用 bean 的完全限定类名.

上面示例中的 bean 名称为 `acme-com.example.AcmeProperties`.

即使前面的配置为 `AcmeProperties` 创建了一个 bean, 我们也建议 `@ConfigurationProperties` 只处理环境 (`environment`) , 特别是不要从上下文中注入其他 bean. 对于极端情况, 可以使用 `setter` 注入或框架提供的任何 `*Aware` 接口 (例如, 需要访问 `Environment` 的 `EnvironmentAware`) . 如果仍然想使用构造函数注入其他 bean, 则必须使用 `@Component` 注解配置属性bean, 并使用基于JavaBean的属性绑定.

使用 `@ConfigurationProperties` 注解类型

这种配置样式与 `SpringApplication` 外部 YAML 配置特别有效, 如以下示例所示:

```
acme:  
  remote-address: 192.168.1.1  
  security:  
    username: admin  
    roles:  
      - USER  
      - ADMIN
```

要使用 `@ConfigurationProperties` bean, 您可以使用与其他 bean 相同的方式注入它们, 如下所示:

```

@Service
public class MyService {

    private final AcmeProperties properties;

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

    //...

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

}

```



使用 `@ConfigurationProperties` 还可以生成元数据文件, IDE 可以通过这些文件来为您自己的 `key` 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](#).

第三方配置

`@ConfigurationProperties` 除了可以使用来注解类之外, 您还可以在公共的 `@Bean` 方法上使用. 当您想要将属性绑定到您掌控之外的第三方组件时, 这样做特别有用.

要使用 `Environment` 属性配置 bean, 请将 `@ConfigurationProperties` 添加到 bean 注册上, 如下所示:

```

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

```

使用 `another` 前缀定义的所有属性都使用与前面的 `AcmeProperties` 示例类似的方式映射到 `AnotherComponent` bean.

宽松绑定

Spring Boot 使用一些宽松的规则将 `Environment` 属性绑定到 `@ConfigurationProperties` bean, 因此 `Environment` 属性名不需要和 bean 属性名精确匹配. 常见的示例包括使用了 - 符号分割的环境属性 (例如, `context-path` 绑定到 `contextPath`) 和大写环境属性 (例如, `PORT` 绑定到 `port`) .

如下 `@ConfigurationProperties` 类:

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

    private String firstName;

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

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

}
```

在上述示例中, 同样可以使用以下属性名称:

Table 6. 宽松绑定

属性	描述
<code>acme.my-project.person.firstName</code>	Kebab 风格 (短横线命名), 建议在 <code>.properties</code> 和 <code>.yml</code> 文件中使用.
<code>acme.myProject.person.firstName</code>	标准驼峰式风格.
<code>acme.my_project.person.first_name</code>	下划线表示法, <code>.properties</code> 和 <code>.yaml</code> 文件中的另外一种格式.

属性	描述
ACME_MYPROJEC	大写风格,当使用系统环境变量时推荐使用该风格.
T_PERSON_FIRS	
TNAME	



注解的 `prefix` 值必须是 kebab (短横线命名)风格 (小写并用 `-` 分隔,例如 `acme.my-project.person`) .

Table 7. 每种属性源 (*property source*) 的宽松绑定规则

属性源	简单类型	列表集合类型
Properties 文件	驼峰式、短横线式或下划线式	标准列表语法使用 <code>[]</code> 或逗号分隔值
YAML 文件	驼峰式、短横线式或者下划线式	标准 YAML 列表语法或者逗号分隔值
环境变量	大写并且以下划线作为定界符,(查看 从环境变量绑定).	数字值两边使用下划线连接,例如 <code>MY_ACME_1_OTHER = my.acme[1].other</code> (查看 从环境变量绑定)
系统属性	驼峰式、短横线式或者下划线式	标准列表语法使用 <code>[]</code> 或逗号分隔值



我们建议,属性尽可能以小写的短横线格式存储,比如 `my.property-name=acme`.

绑定 Maps

当绑定到 `Map` 属性时,如果 `key` 包含除小写字母数字字符或 `-` 以外的任何内容,则需要使用括号表示法来保留原始值. 如果 `key` 没有使用 `[]` 包裹,则里面的任何非字母数字字符或 `-` 的字符都将被删除. 例如,将以下属性绑定到一个 `Map`:

Properties

```
acme.map./key1=value1
acme.map./key2=value2
acme.map./key3=value3
```

Yaml

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



对于 YAML 文件,方括号需要用引号引起来,以便正确解析 keys.

上面的属性将绑定到一个 `Map` 上,其中 `/key1`,`/key2` 和 `key3` 作为 `map` 的 key. 该斜杠已从 `key3` 中删除, 因为它没有被方括号包围. .

如果您的 `key` 包含 `.` 并且绑定为非数值类型, 则有可能也需要使用方括号表示法. 例如, 将 `a.b=c` 绑定到 `Map<String, Object>` 将返回一个带有 `{"a":{"b":"c"}}` entry 的 `Map`, 其中 `[a.b]=c` 会返回一个带有 `{"a.b":"c"}` entry 的 `Map`.

从环境变量绑定

大多数操作系统在对于环境变量有严格规范. 例如, `Linux shell` 变量只能包含字母(`a to z` or `A to Z`), 数字(`0 to 9`)或下划线字符(`_`). 按照约定, `Unix shell` 变量也可以用大写字母命名.

`Spring Boot` 的宽松绑定规则尽可能设计成与这些命名限制兼容.

要将规范形式的属性名称转换为环境变量名称, 可以遵循以下规则:

- 使用下划线 (`_`) 替代 `(.)`.
- 删除所有 `(-)`.
- 转换为大写.

例如, 配置属性 `spring.main.log-startup-info` 是一个名为 `SPRING_MAIN_LOGSTARTUPINFO` 的环境变量.

当绑定到对象列表时, 也可以使用环境变量. 要绑定到列表, 元素编号应在变量名称中用下划线括起来.

例如, 配置属性 `my.acme[0].other` 使用名为 `MY_ACME_0_OTHER` 的环境变量.

合并复杂类型

当列表集合 (`list`) 在多个地方配置时, 整个列表集合将被替换.

例如, 假设带有 `name` 和 `description` 属性的 `MyPojo` 对象默认为 `null`. 以下示例中, `AcmeProperties` 暴露了一个 `MyPojo` 对象列表集合:

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

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

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

}
```

配置可以如下:

```
acme:
  list:
    - name: "my name"
      description: "my description"
  ---
spring:
  config:
    activate:
      on-profile: "dev"
acme:
  list:
    - name: "my another name"
```

如果 `dev` 配置文件未激活, 则 `AcmeProperties.list` 只包含一条 `MyPojo` 条目, 如之前所述. 但是, 如果激活了 `dev` 配置文件, 列表集合仍然只包含一个条目 (`name` 属性值为 `my another name`, `description` 为 `null`). 此配置不会向列表集合中添加第二个 `MyPojo` 实例, 也不会合并条目.

在多个配置文件中指定一个 `List` 时, 最高优先级 (并且只有一个) 的列表集合将被使用. 可做如下配置:

```

acme:
  list:
    - name: "my name"
      description: "my description"
    - name: "another name"
      description: "another description"
  ---
spring:
  config:
    activate:
      on-profile: "dev"
acme:
  list:
    - name: "my another name"

```

在前面示例中,如果 `dev` 配置文件处于 `active` 状态,则 `AcmeProperties.list` 包含一个 `MyPojo` 条目 (`name` 为 `my another name`,`description` 为 `null`) . 对于 YAML 而言,逗号分隔的列表和 YAML 列表同样会完全覆盖列表集合的内容.

对于 `Map` 属性,您可以绑定来自多个源中提取的属性值. 但是,对于多个源中的相同属性,则使用高优先级最高的属性. 以下示例从 `AcmeProperties` 暴露了一个 `Map<String, MyPojo>`:

```

@ConfigurationProperties("acme")
public class AcmeProperties {

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

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

```

可以考虑以下配置:

```
acme:  
  map:  
    key1:  
      name: "my name 1"  
      description: "my description 1"  
---  
spring:  
  config:  
    activate:  
      on-profile: "dev"  
acme:  
  map:  
    key1:  
      name: "dev name 1"  
    key2:  
      name: "dev name 2"  
      description: "dev description 2"
```

如果 `dev` 配置文件未激活，则 `AcmeProperties.map` 只包含一个带 `key1` key 的条目 (`name` 为 `my name 1`, `description` 为 `my description 1`)。如果激活了 `dev` 配置文件，则 `map` 将包含两个条目，`key` 分别为 `key1` (`name` 为 `dev name 1` 和 `description` 为 `my description 1`) 和 `key2` (`name` 为 `dev name 2` 和 `description` 为 `dev description 2`)。



前面的合并规则适用于所有不同属性源的属性，而不仅仅是文件。

属性转换

当外部应用程序属性 (`application properties`) 绑定到 `@ConfigurationProperties` bean 时, Spring Boot 会尝试将其属性强制转换为正确的类型。如果需要自定义类型转换，可以提供 `ConversionService` bean (名为 `conversionService` 的 bean) 或自定义属性编辑器 (通过 `CustomEditorConfigurer` bean) 或自定义转换器 (带有注解为 `@ConfigurationPropertiesBinding` 的 bean 定义)。

由于该 `bean` 在应用程序生命周期早期就被请求，因此请限制 `ConversionService` 使用的依赖。



您在创建时可能无法完全初始化所需的依赖。如果配置 `key` 为非强制需要，您可能希望重命名自定义的 `ConversionService`，并仅依赖于使用 `@ConfigurationPropertiesBinding` 限定的自定义转换器。

转换 `duration`

Spring Boot 支持持续时间 (`duration`) 表达。如果您暴露一个 `java.time.Duration` 属性，则可以在应用程序属性中使用以下格式：

- 常规 `long` 表示（除非指定 `@DurationUnit`, 否则使用毫秒作为默认单位）
- used by `java.time.Duration` 使用的标准 ISO-8601 格式
- 一种更易读的格式，值和单位在一起（例如 `10s` 表示 10 秒）

思考以下示例：

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

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

    private Duration readTimeout = Duration.ofMillis(1000);

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

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

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

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

}
```

指定一个会话超时时间为 `30` 秒, 使用 `30`、`PT30S` 和 `30s` 等形式都是可以的.

读取超时时间设置为 `500ms`, 可以采用以下任何一种形式: `500`、`PT0.5S` 和 `500ms`.

您也可以使用任何支持的单位来标识:

- `ns` 纳秒
- `us` 微秒
- `ms` 毫秒
- `s` 秒
- `m` 分
- `h` 小时
- `d` 天

默认单位是毫秒, 可以使用 `@DurationUnit` 配合上面的单位示例重写. 请注意, 只有使用 `getter` 和 `setter` 的 JavaBean 样式的属性绑定才支持 `@DurationUnit`. 构造函数绑定不支持.

如果您更喜欢使用构造函数绑定, 则可以公开相同的属性, 如以下示例所示:

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

    private final Duration sessionTimeout;

    private final Duration readTimeout;

    public AppSystemProperties(@DurationUnit(ChronoUnit.SECONDS)
        @DefaultValue("30s") Duration sessionTimeout,
        @DefaultValue("1000ms") Duration readTimeout) {
        this.sessionTimeout = sessionTimeout;
        this.readTimeout = readTimeout;
    }

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

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

}
```

Converting periods

除了持续时间, Spring Boot 还可以使用 `java.time.Period` 类型.

可以在应用程序属性中使用以下格式:

- 常规的 `int` 表示形式 (使用天作为默认单位, 除非已指定 `@PeriodUnit`)
- `java.time.Period` 使用的标准 ISO-8601 格式 `format`
- 将值和单位对耦合在一起 (e.g. `1y3d` 表示 1 年零 3 天)

简单格式支持以下单位:

- **y** 年
- **m** 月
- **w** 周
- **d** 天



`java.time.Period` 类型从不存储事迹的星期数, 这是一个快捷方式, 表示 “7 days”.

转换 Data Size

Spring Framework 有一个 `DataSize` 值类型, 允许以字节表示大小. 如果暴露一个 `DataSize` 属性, 则可以在应用程序属性中使用以下格式:

- 常规的 `long` 表示 (使用字节作为默认单位, 除非指定了 `@DataSizeUnit`)
- 更具有可读性的格式, 值和单位在一起 (例如 `10MB` 表示 10 兆字节)

请思考以下示例:

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

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

    private DataSize sizeThreshold = DataSize.ofBytes(512);

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

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

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

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

}
```

要指定 **10** 兆字节的缓冲大小, 使用 **10** 和 **10MB** 是等效的. **256** 字节的大小可以指定为 **256** 或 **256B**.

您也可以使用任何支持的单位:

- **B** 字节
- **KB** 千字节
- **MB** 兆字节
- **GB** 千兆字节
- **TB** 兆兆字节

默认单位是字节, 可以使用 **@DataSizeUnit** 配合上面的示例单位重写.

如果您更喜欢使用构造函数绑定, 则可以公开相同的属性, 如以下示例所示:

```

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

    private final DataSize bufferSize;

    private final DataSize sizeThreshold;

    public AppIoProperties(@DataSizeUnit(DataUnit.MEGABYTES)
        @DefaultValue("2MB") DataSize bufferSize,
        @DefaultValue("512B") DataSize sizeThreshold) {
        this.bufferSize = bufferSize;
        this.sizeThreshold = sizeThreshold;
    }

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

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

}

```

@ConfigurationProperties 验证

只要使用了 Spring 的 `@Validated` 注解, Spring Boot 就会尝试验证 `@ConfigurationProperties` 类. 您可以直接在配置类上使用 JSR-303 `javax.validation` 约束注解. 为此, 请确保 JSR-303 实现在 classpath 上, 然后将约束注解添加到字段上, 如下所示:

```

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

    @NotNull
    private InetSocketAddress remoteAddress;

    // ... getters and setters

}

```



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

虽然绑定时也会验证嵌套属性，但最好的做法还是将关联字段注解上 `@Valid`。这可确保即使未找到嵌套属性也会触发验证。以下示例基于前面的 `AcmeProperties` 示例：

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

    @NotNull
    private InetSocketAddress remoteAddress;

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

    // ... getters and setters

    public static class Security {

        @NotEmpty
        public String username;

        // ... getters and setters
    }
}
```

您还可以通过创建一个名为 `configurationPropertiesValidator` 的 bean 定义来添加自定义 Spring Validator。应该将 `@Bean` 方法声明为 `static`。配置属性验证器在应用程序生命周期的早期创建，将 `@Bean` 方法声明为 `static` 可以无需实例化 `@Configuration` 类来创建 bean。这样做可以避免早期实例化可能导致的意外问题。这里有一个属性验证示例，讲解了如何设置。



`spring-boot-actuator` 模块包括一个暴露所有 `@ConfigurationProperties` bean 的端点。可将 Web 浏览器指向 `/actuator/configprops` 或使用等效的 JMX 端点。有关详细信息，请参阅 “[生产就绪功能](#)” 部分。

@ConfigurationProperties 与 @Value 对比

`@Value` 注解是核心容器功能, 它不提供与类型安全配置属性相同的功能. 下表总结了 `@ConfigurationProperties` 和 `@Value` 支持的功能:

功能	<code>@ConfigurationProperties</code>	<code>@Value</code>
宽松绑定	Yes	Limited (see note below)
元数据支持	Yes	No
SpEL 表达式	No	Yes



如果您确实想使用 `@Value`, 我们建议您以规范形式引用属性名称(kebab-case 仅使用小写字母), 这与 Spring Boot `@ConfigurationProperties` 宽松绑定使用相同的逻辑. 例如 `@Value("{demo.item-price}")` 将从 `application.properties` 文件以及 `DEMO_ITEMPRICE` 环境变量中获取 `demo.item-price` 和 `demo.itemPrice` 形式. 如果您使用的是 `@Value("{demo.itemPrice}")`, 则不会考虑 `demo.item-price` 和 `DEMO_ITEMPRICE` 环境变量.

如果您要为自己的组件定义一组配置 `key`, 我们建议您将它们分组到使用 `@ConfigurationProperties` 注解的 POJO 中. 这样做将为您提供结构化, 类型安全的对象, 您可以将其注入到自己的 bean 中.

解析这些文件并填充环境时, 不会处理来自 [应用程序属性文件](#) 的 SpEL 表达式. 但是, 可以在 `@Value` 中编写 SpEL 表达式. 如果 [应用程序属性文件](#) 中的属性值是 SpEL 表达式, 则在通过 `@Value` 进行使用时将对其进行评估.

5.3. Profiles

Spring Profile 提供了一种应用程序配置部分隔离并使其仅在特定环境中可用的方法. 可以使用 `@Profile` 来注解任何 `@Component` 或 `@Configuration` 以指定何时加载它, 如下所示:

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

    // ...

}
```



如果 `@ConfigurationProperties` Bean 是通过 `@EnableConfigurationProperties` 而非自动扫描注册的，则需要在 `@EnableConfigurationProperties` 注解的 `@Configuration` 类上指定 `@Profile` 注解。在扫描 `@ConfigurationProperties` 的情况下，可以在 `@ConfigurationProperties` 类本身上指定 `@Profile`。

您可以使用 `spring.profiles.active Environment`

属性指定哪些配置文件处于激活状态。您可以使用本章前面介绍的任何方法指定属性。例如，您可以将其包含在 `application.properties` 中，如下所示：

```
spring:
  profiles:
    active: "dev,hsqldb"
```

您还可以在命令行上使用以下开关指定它：`--spring.profiles.active=dev,hsqldb`。

5.3.1. 添加激活 Profile

`spring.profiles.active` 属性遵循与其他属性相同的排序规则：应用优先级最高的 `PropertySource`。这意味着您可以在 `application.properties` 中指定激活配置文件，然后使用命令行开关替换它们。

有时，将特定 `profile` 的属性添加到激活配置文件而不是替换它们，这种方式也是很有用的。 `SpringApplication` 入口还有一个 Java API，用于设置其他 `profile`（即，在 `spring.profiles.active` 属性激活的 `profile` 之上）。请参阅 `SpringApplication` 的 `setAdditionalProfiles()` 方法。如果给定的 `Profile` 是活动的，还可以使用 `Profile` 组（将在 [下一节中](#) 进行描述）添加活动概要文件。

5.3.2. Profile 组

有时，您在应用程序中定义和使用的 `Profile` 粒度太细，使用起来很麻烦。例如，您可以使用 `proddb` 和 `prodmq` `Profile` 独立地启用数据库和消息传递特性。

为了帮助实现这一点，Spring Boot 允许您定义 `Profile` 组。`Profile` 组允许您为相关的 `Profile` 组定义逻辑名称。

例如，我们可以创建一个由 `proddb` 和 `prodmq` 配置文件组成的 `production` 组。

```
spring:  
  profiles:  
    group:  
      production:  
        - "proddb"  
        - "prodmq"
```

我们的应用程序现在可以使用—`spring.profiles.active=production`一次性激活 `production`, `proddb` 和 `prodmq` 配置文件。

5.3.3. 以编程方式设置 Profile

您可以在应用程序运行之前通过调用 `SpringApplication.setAdditionalProfiles(...)` 以编程方式设置 `active` 配置文件。也可以使用 Spring 的 `ConfigurableEnvironment` 接口激活 `profile`。

5.3.4. 特定 Profile 的配置文件

特定 `profile` 的 `application.properties` (或 `application.yml`) 和通过 `@ConfigurationProperties` 引用的文件被当做文件并加载。有关详细信息，请参见 "[特定 Profile 的属性文件](#)"。

5.4. 日志记录

Spring Boot 使用 `Commons Logging` 记录所有内部日志，但保留底层日志实现。其为 `Java Util Logging`, `Log4J2` 和 `Logback` 提供了默认配置。

每种日志记录器都预先配置为使用控制台输出，并且还提供可选的文件输出。

默认情况下,如果您使用了 [Starter](#),则使用 [Logback](#) 进行日志记录.还包括合适的 [Logback](#) 路由,以确保在使用 [Java Util Logging](#)、[Commons Logging](#)、[Log4J](#) 或 [SLF4J](#) 的依赖库都能正常工作.



[Java](#) 有很多日志框架可供使用. 如果以上列表让您感到困惑,请不要担心.
通常,您不需要更改日志依赖,并且 [Spring Boot](#) 提供的默认配置可以保证日志正常工作.



将应用程序部署到 [Servlet](#) 容器或应用程序服务器时,通过 [Java Util Logging API](#) 执行的日志记录不会路由到应用程序的日志中.
这样可以防止容器或其他已部署到容器中的应用程序执行的日志记录出现在应用程序的日志中.

5.4.1. 日志格式

[Spring Boot](#) 默认日志输出类似于以下示例:

```
2019-03-05 10:57:51.112 INFO 45469 --- [           main]
org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache
Tomcat/7.0.52
2019-03-05 10:57:51.253 INFO 45469 --- [ost-startStop-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2019-03-05 10:57:51.253 INFO 45469 --- [ost-startStop-1]
o.s.web.context.ContextLoader           : Root WebApplicationContext:
initialization completed in 1358 ms
2019-03-05 10:57:51.698 INFO 45469 --- [ost-startStop-1]
o.s.b.c.e.ServletRegistrationBean      : Mapping servlet: 'dispatcherServlet'
to [/]
2019-03-05 10:57:51.702 INFO 45469 --- [ost-startStop-1]
o.s.b.c.embedded.FilterRegistrationBean : Mapping filter:
'hiddenHttpMethodFilter' to: [/*]
```

输出以下项:

- 日期和时间: 毫秒精度,易于排序.
- 日志级别: [ERROR](#)、[WARN](#)、[INFO](#)、[DEBUG](#) 或 [TRACE](#).
- 进程 ID.

- 一个 `---` 分隔符, 用于区分实际日志内容的开始.
- 线程名称: 在方括号中 (可能会截断控制台输出) .
- 日志记录器名称: 这通常是源类名称 (通常为缩写) .
- 日志内容.



Logback 没有 `FATAL` 级别. 该级别映射到 `ERROR`.

5.4.2. 控制台输出

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

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



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

启用调试模式后, 核心日志记录器 (内嵌容器、`Hibernate` 和 `Spring Boot`) 将被配置为输出更多日志信息. 启用调试模式不会将应用程序配置为使用 `DEBUG` 级别记录所有日志内容.

或者, 您可以通过使用 `--trace` 标志 (或在 `application.properties` 中的设置 `trace=true`) 启动应用程序来启用跟踪模式. 这样做可以为选择的核心日志记录器 (内嵌容器、`Hibernate` 模式生成和整个 `Spring` 组合) 启用日志追踪.

着色输出

如果您的终端支持 `ANSI`, 则可以使用颜色输出来提高可读性. 您可以将 `spring.output.ansi.enabled` 设置为 [受支持的值](#) 以覆盖自动检测.

可使用 `%clr` 转换字配置颜色编码. 最简单形式是, 转换器根据日志级别对输出进行着色, 如下所示:

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

下表描述日志级别与颜色的映射关系：

Level	Color
FATAL	Red
ERROR	Red
WARN	Yellow
INFO	Green
DEBUG	Green
TRACE	Green

或者，您可以通过将其作为转换选项指定应使用的颜色或样式。例如，要将文本变为黄色，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`

5.4.3. 文件输出

默认情况下，Spring Boot 仅记录到控制台，不会写入日志文件。

想除了控制台输出之外还要写入日志文件，则需要设置 `logging.file` 或 `logging.path` 属性（例如，在 `application.properties` 中）。

下表展示了如何与 `logging.*` 属性一起使用：

Table 8. Logging 属性

<code>logging.file.name</code>	<code>logging.file.path</code>	Example	Description
(none)	(none)		仅在控制台输出
指定文件	(none)	<code>my.log</code>	写入指定的日志文件。 名称可以是绝对位置或相对于当前目录。
(none)	指定目录	<code>/var/log</code>	将 <code>spring.log</code> 写入指定的目录。 名称可以是绝对位置或相对于当前目录。

日志文件在达到 **10MB** 时会轮转，并且与控制台输出一样，默认情况下会记录 **ERROR**、**WARN** 和 **INFO** 级别的内容。



日志属性独立于实际的日志底层。因此，Spring Boot 不管理特定的配置 key（例如 Logback 的 `logback.configurationFile`）。

5.4.4. File Rotation

如果您正在使用 Logback，则可以使用 `application.properties` 或 `application.yaml` 设置 log rotation。对于所有其他日志系统，您需要自己直接配置 rotation 设置（例如，如果您使用 Log4J2，那么您可以添加一个 `log4j.xml` 文件）。

支持以下 rotation 策略属性：

Name	Description
<code>configprop:logging.logback.rollingpolicy.file-name-pattern[]</code>	The filename pattern used to create log archives.
<code>configprop:logging.logback.rollingpolicy.clean-history-on-start[]</code>	If log archive cleanup should occur when the application starts.
<code>configprop:logging.logback.rollingpolicy.max-file-size[]</code>	The maximum size of log file before it's archived.

Name	Description
configprop:logging.logback.rollingpolicy.total-size-cap[]	The maximum amount of size log archives can take before being deleted.
configprop:logging.logback.rollingpolicy.max-history[]	The number of days to keep log archives (defaults to 7)

5.4.5. 日志等级

所有受支持的日志记录系统都可以使用 `logging.level.<logger-name>=<level>` 来设置 Spring Environment 中的记录器等级（例如，在 application.properties 中）。其中 level 是 TRACE、DEBUG、INFO、WARN、ERROR、FATAL 和 OFF 其中之一。可以使用 `logging.level.root` 配置 root 记录器。

以下示例展示了 application.properties 中默认的日志记录设置：

Properties

```
logging.level.root=warn
logging.level.org.springframework.web=debug
logging.level.org.hibernate=error
```

Yaml

```
logging:
  level:
    root: "warn"
    org.springframework.web: "debug"
    org.hibernate: "error"
```

也可以使用环境变量设置日志记录级别。例如，

`LOGGING_LEVEL_ORG_SPRINGFRAMEWORK_WEB=DEBUG` 会将 `org.springframework.web` 设置为 `DEBUG`。



以上方法仅适用于程序包级别的日志记录。

由于宽松的绑定总是将环境变量转换为小写，因此无法以这种方式为单个类配置日志记录。如果需要为类配置日志记录，则可以使用 [the SPRING_APPLICATION_JSON 变量](#)。

5.4.6. 日志组

将相关记录器组合在一起以便可以同时配置，这通常很有用。例如，您可以更改所有 Tomcat 相关记录器的日志记录级别，但您无法轻松记住顶层的包名。

为了解决这个问题，Spring Boot 允许您在 [Spring Environment](#) 中定义日志记录组。例如，以下通过将 `tomcat` 组添加到 `application.properties` 来定义 `tomcat` 组：

```
logging:
  group:
    tomcat: "org.apache.catalina,org.apache.coyote,org.apache.tomcat"
```

定义后，您可以使用一行配置来更改组中所有记录器的级别：

```
logging:
  level:
    tomcat: "trace"
```

Spring Boot 包含以下预定义的日志记录组，可以直接使用：

名称	日志记录器
web	<code>org.springframework.core.codec,</code> <code>org.springframework.http, org.springframework.web,</code> <code>org.springframework.boot.actuate.endpoint.web,</code> <code>org.springframework.boot.web.servlet.ServletContextInitializerBeans</code>
sql	<code>org.springframework.jdbc.core, org.hibernate.SQL,</code> <code>org.jooq.tools.LoggerListener</code>

5.4.7. 使用日志 Shutdown 钩子

为了释放日志记录资源，通常最好的做法就是在应用程序终止时停止日志记录系统。不幸的是，没有一种方法可以适用于所有应用程序类型。

如果您的应用程序具有复杂的上下文层次结构或作为 `war` 文件部署，则需要研究基础日志系统直接提供的选项。例如，`Logback` 提供了 `context selectors`，它允许在自己的上下文中创建每个 `Logger`。

对于在自己的 JVM 中部署的简单 “jar” 应用程序，可以使用 `logging.register-shutdown-hook` 属性。将 `logging.register-shutdown-hook` 设置为 `true` 将注册一个关闭钩子，当 JVM 退出时，该钩子将触发日志系统清理。

您可以在 `application.properties` 或 `application.yaml` 文件中设置属性：

```
logging:  
  register-shutdown-hook: true
```

5.4.8. 自定义日志配置

可以通过在 `classpath` 中引入适合的库来激活各种日志记录系统，并且可以通过在 `classpath` 的根目录中或在以下 `Spring Environment` 属性指定的位置提供合适的配置文件来进一步自定义：`logging.config`。

您可以使用 `org.springframework.boot.logging.LoggingSystem` 系统属性强制 Spring Boot 使用特定的日志记录系统。该值应该是一个实现了 `LoggingSystem` 的类的完全限定类名。您还可以使用 `none` 值完全禁用 Spring Boot 的日志记录配置。



由于日志记录在创建 `ApplicationContext` 之前初始化，因此无法在 `Spring @Configuration` 文件中控制来自 `@PropertySources` 的日志记录。

更改日志记录系统或完全禁用它的唯一方法是通过系统属性设置。

根据您的日志记录系统，将加载以下文件：

日志记录系统	文件
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> 或者 <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>



如果可能, 我们建议您使用 `-spring` 的形式来配置日志记录 (比如 `logback-spring.xml` 而不是 `logback.xml`) . 如果使用标准的配置位置, 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, 这些问题在以 '`executable jar`' 运行时会触发. 如果可能的话, 我们建议您在使用可执行 jar 方式运行时避免使用它. .

为了进行自定义, 部分其他属性会从 Spring Environment 传输到 System 属性, 如下表所述:

Spring Environment	系统属性	说明
<code>logging.exception-conversion-word</code>	<code>LOG_EXCEPTION_CONVERSIO</code> N_WORD	记录异常时使用的转换字.
<code>logging.file.name</code>	<code>LOG_FILE</code>	如果已定义, 则在默认日志配置中使用它.
<code>logging.file.path</code>	<code>LOG_PATH</code>	如果已定义, 则在默认日志配置中使用它.
<code>logging.pattern.console</code>	<code>CONSOLE_LOG_PATTERN</code>	要在控制台上使用的日志模式 (<code>stdout</code>) .
<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>configprop:logging.charset.console[]</code>	<code>CONSOLE_LOG_CHARSET</code>	The charset to use for console logging.

Spring Environment	系统属性	说明
<code>logging.pattern.file</code>	<code>FILE_LOG_PATTERN</code>	要在文件中使用的日志模式 (如果启用了 <code>LOG_FILE</code>) .
<code>configprop:logging.charset.file[]</code>	<code>FILE_LOG_CHARSET</code>	The charset to use for file logging (if <code>LOG_FILE</code> is enabled).
<code>logging.pattern.level</code>	<code>LOG_LEVEL_PATTERN</code>	渲染日志级别时使用的格式 (默认值为 <code>%5p</code>) .
<code>PID</code>	<code>PID</code>	当前进程 ID (如果可能, 则在未定义为 <code>OS</code> 环境变量时发现) .

如果您使用的是 Logback, 以下属性也会被转移:

Spring Environment	System Property	Comments
<code>configprop:logging.logback.rollingpolicy.filename-pattern[]</code>	<code>LOGBACK_ROLLINGPOLICY_FILENAME_PATTERN</code>	Pattern for rolled-over log file names (default <code> \${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz</code>).
<code>configprop:logging.logback.rollingpolicy.clean-history-on-start[]</code>	<code>LOGBACK_ROLLINGPOLICY_CLEAN_HISTORY_ON_START</code>	Whether to clean the archive log files on startup.
<code>configprop:logging.logback.rollingpolicy.max-file-size[]</code>	<code>LOGBACK_ROLLINGPOLICY_MAX_FILE_SIZE</code>	Maximum log file size.
<code>configprop:logging.logback.rollingpolicy.total-size-cap[]</code>	<code>LOGBACK_ROLLINGPOLICY_TOTAL_SIZE_CAP</code>	Total size of log backups to be kept.
<code>configprop:logging.logback.rollingpolicy.max-history[]</code>	<code>LOGBACK_ROLLINGPOLICY_MAX_HISTORY</code>	Maximum number of archive log files to keep.

所有受支持的日志记录系统在解析其配置文件时都可以参考系统属性。有关示例, 请参阅

`spring-boot.jar` 中的默认配置：

- [Logback](#)
- [Log4j 2](#)
- [Java Util logging](#)



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



您可以通过仅覆盖 `LOG_LEVEL_PATTERN`（或带 [Logback](#) 的 `logging.pattern.level`）将 MDC 和其他特别的内容添加到日志行。例如，如果使用 `logging.pattern.level=user:%X{user} %5p`，则默认日志格式包含 `user` MDC 项（如果存在），如下所示：

```
2019-08-30 12:30:04.031 user:someone INFO 22174 --- [nio-8080-exec-0] demo.Controller
Handling authenticated request
```

5.4.9. Logback 扩展

[Spring Boot](#) 包含许多 [Logback](#) 扩展，可用于进行高级配置。您可以在 `logback-spring.xml` 配置文件中使用这些扩展。



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



扩展不能与 [Logback](#) 的 [配置扫描](#) 一起使用。如果尝试这样做，更改配置文件会导致发生类似以下错误日志：。

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

特定 **Profile** 配置

<springProfile> 标签允许您根据激活的 Spring profile 选择性地包含或排除配置部分。在 **<configuration>** 元素中的任何位置都支持配置 profile。使用 **name** 属性指定哪个 profile 接受配置。**<springProfile>** 标记可以包含简单的 profile 名称（例如 **staging**）或 profile 表达式。profile 表达式允许表达更复杂的 profile 逻辑，例如 **production & (eu-central | eu-west)**。有关详细信息，请查阅 [参考指南](#)。

以下清单展示了三个示例 profile：

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

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

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

环境属性

使用 **<springProperty>** 标签可以让您暴露 Spring 环境 (**Environment**) 中的属性，以便在 Logback 中使用。如果在 Logback 配置中访问来自 **application.properties** 文件的值，这样做很有用。标签的工作方式与 Logback 的标准 **<property>** 标签类似。但是，您可以指定属性（来自 **Environment**）的 **source**，而不是指定直接的 **value**。如果需要将属性存储在 **local** 作用域以外的其他位置，则可以使用 **scope** 属性。如果需要回退值（如果未在 **Environment** 中设置该属性），则可以使用 **defaultValue** 属性。以下示例展示了如何暴露属性以便在 Logback 中使用：

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



必须以 kebab 风格（短横线小写风格）指定 `source`（例如 `my.property-name`）。但可以使用宽松规则将属性添加到 `Environment` 中。

5.5. 国际化

Spring Boot 支持本地化消息，因此您的应用程序可以迎合不同语言首选项的用户。默认情况下，Spring Boot 在类路径的根目录下查找 `messages` 资源包的存在。



当已配置资源束的默认属性文件可用时（即默认情况下为 `messages.properties`），将应用自动配置。

如果您的资源包仅包含特定于语言的属性文件，则需要添加默认文件。

如果找不到与任何配置的基本名称匹配的属性文件，将没有自动配置的 `MessageSource`。

可以使用 `spring.messages` 命名空间配置资源包的基本名称以及其他几个属性，如下所示：

```
spring:
  messages:
    basename: "messages,config.i18n.messages"
    fallback-to-system-locale: false
```



`spring.messages.basename` 支持以逗号分隔的位置列表，可以是包限定符，也可以是从类路径根目录解析的资源。

有关更多支持的选项，请参见 `MessageSourceProperties`

5.6. JSON

Spring Boot 为三个 JSON 映射库提供了内置集成：

- Gson
- Jackson
- JSON-B

Jackson 是首选和默认的库。

5.6.1. Jackson

Spring Boot 提供了 Jackson 的自动配置, Jackson 是 `spring-boot-starter-json` 的一部分. 当 Jackson 在 `classpath` 上时, 会自动配置 `ObjectMapper` bean. Spring Boot 提供了几个配置属性来 [自定义 ObjectMapper 的配置](#).

5.6.2. Gson

Spring Boot 提供 Gson 的自动配置. 当 `Gson` 在 `classpath` 上时, 会自动配置 `Gson` bean. Spring Boot 提供了几个 `spring.gson.*` 配置属性来自定义配置.

为了获得更多控制, 可以使用一个或多个 `GsonBuilderCustomizer` bean.

5.6.3. JSON-B

Spring Boot 提供了 JSON-B 的自动配置. 当 JSON-B API 和实现在 `classpath` 上时, 将自动配置 `Jsonb` bean. 首选的 JSON-B 实现是 Apache Johnzon, 它提供了依赖管理.

5.7. 开发 Web 应用程序

Spring Boot 非常适合用于开发 web 应用程序. 您可以使用嵌入式 Tomcat、Jetty 或者 Undertow 来创建一个独立 (`self-contained`) 的 HTTP 服务器. 大多数 web 应用程序使用 `spring-boot-starter-web` 模块来快速搭建和运行, 您也可以选择使用 `spring-boot-starter-webflux` 模块来构建响应式 (`reactive`) web 应用程序.

如果您尚未开发过 Spring Boot web 应用程序, 则可以按照 [入门](#) 章节中的 "Hello World!" 示例进行操作.

5.7.1. Spring Web MVC 框架

[Spring Web MVC 框架](#) (通常简称 "Spring MVC") 是一个富模型-视图-控制器的 web 框架。Spring MVC 允许您创建 `@Controller` 或者 `@RestController` bean 来处理传入的 HTTP 请求。控制器中的方法通过 `@RequestMapping` 注解映射到 HTTP。

以下是一个使用了 `@RestController` 来响应 JSON 数据的典型示例：

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

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

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

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

}
```

Spring MVC 是 Spring Framework 核心的一部分，详细介绍可参考其 [参考文档](#)。spring.io/guides 还提供了几个 Spring MVC 相关的指南。

Spring MVC 自动配置

Spring Boot 提供了适用于大多数 Spring MVC 应用的自动配置 (auto-configuration)。

自动配置在 Spring 默认功能上添加了以下功能：

- 引入 `ContentNegotiatingViewResolver` 和 `BeanNameViewResolver` bean。
- 支持服务静态资源，包括对 `WebJar` 的支持 ([见下文](#))。

- 自动注册 `Converter`、`GenericConverter` 和 `Formatter` bean.
- 支持 `HttpMessageConverter` (见下文) .
- 自动注册 `MessageCodesResolver` (见下文) .
- 支持静态 `index.html`.
- 自动使用 `ConfigurableWebBindingInitializer` bean (见下文) .

如果您想保留 Spring Boot MVC 的功能, 并且需要添加其他 `MVC configuration` (`interceptor`、`formatter` 和视图控制器等), 可以添加自己的 `WebMvcConfigurerAdapter` 类型的 `@Configuration` 类, 但不能带 `@EnableWebMvc` 注解.

如果您想自定义 `RequestMappingHandlerMapping`、`RequestMappingHandlerAdapter` 或者 `ExceptionHandlerExceptionResolver` 实例, 可以声明一个 `WebMvcRegistrationsAdapter` 实例来提供这些组件.

如果您想完全掌控 Spring MVC, 可以添加自定义注解了 `@EnableWebMvc` 的 `@Configuration` 配置类. 或者添加自己的 `@Configuration` 注解的 `DelegatingWebMvcConfiguration`, 如 Java 文档中的 `@EnableWebMvc` 所述. .

Spring MVC 使用了一种不同的 `ConversionService`, 该转换器用于转换 `application.properties` 或 `application.yaml` 文件中的值. 这意味着 `Period`, `Duration` 和 `DataSize` 转换器不可用, 而 `@DurationUnit` 和 `@DataSizeUnit` 注解将被忽略.



如果您想自定义 Spring MVC 使用的 `ConversionService`, 则可以为 `WebMvcConfigurer` bean 提供 `addFormatters` 方法. 通过此方法, 您可以注册所需的任何转换器, 也可以委托给 `ApplicationConversionService` 上可用的静态方法.

HttpMessageConverters

Spring MVC 使用 `HttpMessageConverter` 接口来转换 HTTP 的请求和响应. 开箱即用功能包含了合适的默认值, 比如对象可以自动转换为 JSON (使用 Jackson 库) 或者 XML (优先使用 Jackson XML 扩展, 其次为 JAXB). 字符串默认使用 `UTF-8` 编码.

如果您需要添加或者自定义转换器 (converter) , 可以使用 Spring Boot 的 [HttpMessageConverters](#) 类:

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

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

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

}
```

上下文中的所有 [HttpMessageConverter](#) bean 都将被添加到转换器列表中.

您也可以用这种方式来覆盖默认转换器.

自定义 JSON Serializer 和 Deserializer

如果您使用 Jackson 序列化和反序列化 JSON 数据, 可能需要自己编写 [JsonSerializer](#) 和 [JsonDeserializer](#) 类. 自定义序列化器 (serializer) 的做法通常是指通过一个模块来注册 Jackson, 然而 Spring Boot 提供了一个备选的 [@JsonComponent](#) 注解, 它可以更加容易地直接注册 Spring Bean.

您可以直接在 [JsonSerializer](#) 或者 [JsonDeserializer](#) 实现上使用 [@JsonComponent](#) 注解. 您也可以在将序列化器/反序列化器 (deserializer) 作为内部类的类上使用. 例如:

```

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

@JsonComponent
public class Example {

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

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

```

`ApplicationContext` 中所有的 `@JsonComponent` bean 将被自动注册到 Jackson 中, 由于 `@JsonComponent` 使用 `@Component` 注解标记, 因此组件扫描 (component-scanning) 规则将对其生效.

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

MessageCodesResolver

Spring MVC 有一个从绑定错误中生成错误码的策略, 用于渲染错误信息: `MessageCodesResolver`. 如果您设置了 `spring.mvc.message-codes-resolver.format` 属性值为 `PREFIX_ERROR_CODE` 或 `POSTFIX_ERROR_CODE`, Spring Boot 将为你创建该策略 (请参阅 `DefaultMessageCodesResolver.Format` 中的枚举).

静态内容

默认情况下, Spring Boot 将在 `classpath` 或者 `ServletContext` 根目录下从名为 `/static` (`/public`、`/resources` 或 `/META-INF/resources`) 目录中服务静态内容. 它使用了 Spring MVC 的 `ResourceHttpRequestHandler`, 因此您可以通过添加自己的 `WebMvcConfigurerAdapter` 并重写 `addResourceHandlers` 方法来修改此行为.

在一个独立的 (stand-alone) web 应用程序中,来自容器的默认 `servlet` 也是被启用的,并充当一个回退支援, Spring 决定不处理 `ServletContext` 根目录下的静态资源,容器的默认 `servlet` 也将会处理. 大多情况下,这是不会发生的 (除非您修改了默认的 MVC 配置),因为 Spring 始终能通过 `DispatcherServlet` 来处理请求.

默认情况下,资源被映射到 `/`,但可以通过 `spring.mvc.static-path-pattern` 属性调整. 比如,将所有资源重定位到 `/resources/`:

```
spring:
  mvc:
    static-path-pattern: "/resources/**"
```

您还可以使用 `spring.web.resources.static-locations` 属性来自定义静态资源的位置(使用一个目录位置列表替换默认值). 根 `Servlet context path /` 自动作为一个 `location` 添加进来.

除了上述提到的标准静态资源位置之外,还有一种特殊情况是用于 `Webjars content`. 如果以 `Webjar` 格式打包,则所有符合 `/webjars/**` 的资源都将从 `jar` 文件中服务.



如果您的应用程序要包成 `jar`,请不要使用 `src/main/webapp` 目录.

虽然此目录是一个通用标准,但它只适用于 `war` 打包,如果生成的是一个 `jar`,它将被绝大多数的构建工具所忽略.

`Spring Boot` 还支持 `Spring MVC` 提供的高级资源处理功能,允许使用例如静态资源缓存清除 (cache busting) 或者 Webjar 版本无关 URL.

要使用 Webjar 版本无关 URL 功能,只需要添加 `webjars-locator-core` 依赖. 然后声明您的 Webjar,以 `jQuery` 为例,添加的 `"/webjars/jquery/dist/jquery.min.js"` 将变成 `"/webjars/jquery/x.y.z/dist/jquery.min.js"`,其中 `x.y.z` 是 Webjar 的版本.



如果您使用 JBoss,则需要声明 `webjars-locator-jboss-vfs` 依赖,而不是 `webjars-locator-core`,否则所有 Webjar 将被解析成 `404`.

要使用缓存清除功能,以下配置为所有静态资源配置了一个缓存清除方案,实际上是在 URL

上添加了一个内容哈希,例如 `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`:

```
spring:  
  web:  
    resources:  
      chain:  
        strategy:  
          content:  
            enabled: true  
            paths: "/**"
```



模板中的资源链接在运行时被重写,这得益于 `ResourceUrlEncodingFilter` 为 `Thymeleaf` 和 `FreeMarker` 自动配置. 在使用 `JSP` 时,您应该手动声明此过滤器. 其他模板引擎现在还不会自动支持,但可以与自定义模板宏 (`macro`) /`helper` 和 `ResourceUrlProvider` 结合使用.

当使用例如 `Javascript` 模块加载器动态加载资源时,重命名文件是不可选的. 这也是为什么支持其他策略并且可以组合使用的原因. `fixed`策略将在 URL 中添加一个静态版本字符串,而不是更改文件名:

```
spring:  
  web:  
    resources:  
      chain:  
        strategy:  
          content:  
            enabled: true  
            paths: "/**"  
        fixed:  
          enabled: true  
          paths: "/js/lib/"  
          version: "v12"
```

使用此配置, `JavaScript` 模块定位在 `"/js/lib/"` 下使用固定版本策略 (`"/v12/js/lib/mymodule.js"`),而其他资源仍使用内容策略 (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`).

有关更多支持选项,请参阅 `ResourceProperties`.



该功能已经在一个专门的 [博客文章](#) 和 [Spring 框架的参考文档](#) 中进行了详细描述

欢迎页面

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

路径匹配与内容协商

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

Spring Boot 默认选择禁用后缀模式匹配，这意味着像 `"GET /projects/spring-boot.json"` 这样的请求将不会与 `@GetMapping("/projects/spring-boot")` 映射匹配。这被视为是 [Spring MVC 应用程序的最佳实践](#)。此功能在过去对于 HTTP 客户端没有发送正确的 `Accept` 请求头的情况还是很有用的，我们需要确保将正确的内容类型发送给客户端。如今，内容协商（Content Negotiation）更加可靠。

还有其他方法可以处理 HTTP 客户端发送不一致 `Accept` 请求头问题。

我们可以使用查询参数来确保像 `"GET /projects/spring-boot?format=json"` 这样的请求映射到 `@GetMapping("/projects/spring-boot")`，而不是使用后缀匹配：

```
spring:  
  mvc:  
    contentnegotiation:  
      favor-parameter: true
```

或者，如果您更喜欢使用不同的参数名称：

```
spring:  
  mvc:  
    contentnegotiation:  
      favor-parameter: true  
      parameter-name: "myparam"
```

大多数标准媒体类型都是开箱即用的，但您也可以定义新的：

```
spring:  
  mvc:  
    contentnegotiation:  
      media-types:  
        markdown: "text/markdown"
```

后缀模式匹配已被弃用，并将在未来版本中删除。

如果您了解相关注意事项但仍希望应用程序使用后缀模式匹配，则需要以下配置：

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

或者，不打开所有后缀模式，仅打开支持已注册的后缀模式更加安全：

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

从 Spring Framework 5.3 开始，Spring MVC 支持几种实现策略来将请求路径匹配到 Controller 处理程序。它以前只支持 **AntPathMatcher** 策略，但现在也提供了 **PathPatternParser**。Spring Boot 现在提供了一个可以在新策略中选择的配置属性：

```
spring:  
  mvc:  
    pathmatch:  
      matching-strategy: "path-pattern-parser"
```

有关为什么应该考虑这种新实现的更多详细信息，请查看 [专门的博客文章](#)。



`PathPatternParser` 是一个优化的实现，但限制了 [某些路径模式变体](#) 的使用，并且与后缀模式匹配(`configprop:spring.mvc.pathmatch.use-suffix-pattern[deprecated]`,`configprop:spring.mvc.pathmatch.use-registered-suffix-pattern[deprecated]`) 或将 `DispatcherServlet` 映射为 `Servlet` 前缀(`(configprop:spring.mvc.servlet.path[])`)

ConfigurableWebBindingInitializer

Spring MVC 使用一个 `WebBindingInitializer` 为特定的请求初始化 `WebDataBinder`. 如果您创建了自己的 `ConfigurableWebBindingInitializer @Bean`, Spring Boot 将自动配置 Spring MVC 使用它.

模板引擎

除了 REST web 服务之外, 您还可以使用 Spring MVC 来服务动态 HTML 内容. Spring MVC 支持多种模板技术, 包括 Thymeleaf、FreeMarker 和 JSP. 当然, 许多其他模板引擎也有自己的 Spring MVC 集成.

Spring Boot 包含了以下的模板引擎的自动配置支持:

- [FreeMarker](#)
- [Groovy](#)
- [Thymeleaf](#)
- [Mustache](#)



如果可以, 请尽量避免使用 JSP, 当使用了内嵌 `servlet` 容器, 会有几个[已知限制](#).

当您使用这些模板引擎的其中一个并附带了默认配置时, 您的模板将从 `src/main/resources/templates` 自动获取.



IntelliJ IDEA 根据您运行应用程序的方式来对 classpath 进行不同的排序。在 IDE 中通过 main 方法来运行应用程序将导致与使用 Maven 或 Gradle 或者以 jar 包方式引用程序的排序有所不同，可能会导致 Spring Boot 找不到 classpath 中的模板。如果您碰到此问题，可以重新排序 IDE 的 classpath 来放置模块的 classes 和 resources 到首位。

错误处理

默认情况下，Spring Boot 提供了一个使用了比较合理的方式来处理所有错误的 `/error` 映射，其在 `servlet` 容器中注册了一个全局错误页面。对于机器客户端而言，它将产生一个包含错误、HTTP 状态和异常消息的 JSON 响应。对于浏览器客户端而言，将以 HTML 格式呈现相同数据的 `whitelabel` 错误视图（可添加一个解析到 `error` 的 `View` 进行自定义）。

如果要自定义默认错误处理行为，可以设置许多 `server.error` 属性。请参阅附录的 “[Server Properties](#)” 部分。

要完全替换默认行为，您可以实现 `ErrorController` 并注册该类型的 bean，或者简单地添加一个类型为 `ErrorAttributes` 的 bean 来替换内容，但继续使用现用机制。



`BasicErrorController` 可以作为自定义 `ErrorController` 的基类，这非常有用，尤其是在您想添加一个新的内容类型（默认专门处理 `text/html`，并为其他内容提供后备）处理器的情况下。要做到这点，您只需要继承 `BasicErrorController` 并添加一个带有 `produces` 属性的 `@RequestMapping` 注解的公共方法，之后创建一个新类型的 bean。

您还可以定义一个带有 `@ControllerAdvice` 注解的类来自定义为特定控制器或异常类型返回的 JSON 文档：

```

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

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

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

}

```

以上示例中,如果同包下定义的控制器 `AcmeController` 抛出了 `YourException`,则将使用 `CustomerErrorType` 类型的 `POJO` 来代替 `ErrorAttributes` 做 JSON 呈现.

自定义错误页面

如果您想在自定义的 `HTML` 错误页面上显示给定的状态码,请将文件添加到 `/error` 目录中.
错误页面可以是静态 `HTML` (添加在任意静态资源目录下) 或者使用模板构建.
文件的名称应该是确切的状态码或者一个序列掩码.

例如,要将 `404` 映射到一个静态 `HTML` 文件,目录结构可以如下:

```

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

```

使用 FreeMarker 模板来映射所有 **5xx** 错误，目录的结构如下：

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

对于更复杂的映射，您还通过可以添加实现了 **ErrorViewResolver** 接口的 bean 来处理：

```
public class MyErrorViewResolver implements ErrorViewResolver {

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

}
```

您还可以使用常规的 Spring MVC 功能，比如 **@ExceptionHandler methods** 方法和 **@ControllerAdvice**。之后，**ErrorController** 将能接收任何未处理的异常。

映射到 Spring MVC 之外的错误页面

对于不使用 Spring MVC 的应用程序，您可以使用 **ErrorPageRegistrar** 接口来直接注册 **ErrorPages**。抽象部分直接与底层的内嵌 **servlet** 容器一起工作，即使您没有 Spring MVC **DispatcherServlet** 也能使用。

```

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

// ...

private static class MyErrorPageRegistrar implements ErrorPageRegistrar {

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

}

```



如果您注册了一个 `ErrorPage`, 它的路径最终由一个 `Filter` (例如, 像一些非 Spring web 框架一样, 比如 Jersey 和 Wicket) 处理, 则必须将 `Filter` 显式注册为一个 `ERROR dispatcher`, 如下示例:

```

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

```

请注意, 默认的 `FilterRegistrationBean` 不包含 `ERROR` 调度器 (`dispatcher`) 类型.

Error handling in a war deployment

当部署到 `servlet` 容器时, Spring Boot

使用其错误页面过滤器会将有错误状态的请求转发到相应的错误页面. 这是必需的, 因为 `Servlet` 规范没有提供用于注册错误页面的 API. 根据要将 `war` 文件部署到的容器以及应用程序使用的技术, 可能需要一些其他配置.

如果尚未提交响应, 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 及更高版本在成功完成 `servlet` 的 `service`

方法后提交响应。您应该将 `com.ibm.ws.webcontainer.invokeFlushAfterService` 设置为 `false` 来禁用此行为。

如果您使用的是 `Spring Security`, 并且想访问错误页面中的用户 `principal` 信息, 则必须配置 `Spring Security` 过滤器来处理的错误是分发。为此, 请将 `spring.security.filter.dispatcher-types` 属性配置为 `async, error, forward, request`

Spring HATEOAS

如果您想开发一个使用超媒体 (`hypermedia`) 的 RESTful API, Spring Boot 提供的 Spring HATEOAS 自动配置在大多数应用程序都工作得非常好。自动配置取代了 `@EnableHypermediaSupport` 的需要, 并注册了一些 bean, 以便能轻松构建基于超媒体的应用程序, 其包括了一个 `LinkDiscoverers` (用于客户端支持) 和一个用于配置将响应正确呈现的 `ObjectMapper`. `ObjectMapper` 可以通过设置 `spring.jackson.*` 属性或者 `Jackson2ObjectMapperBuilder` bean (如果存在) 自定义。

您可以使用 `@EnableHypermediaSupport` 来控制 Spring HATEOAS 的配置。请注意, 这使得上述的自定义 `ObjectMapper` 被禁用。

CORS 支持

`Cross-origin resource sharing` 跨域资源共享 (`Cross-origin resource sharing, CORS`) 是 `most browsers` 实现的一个 `W3C specification`, 其可允许您以灵活的方式指定何种跨域请求可以被授权, 而不是使用一些不太安全和不太强大的方式 (比如 `IFRAME` 或者 `JSONP`) .

Spring MVC 从 4.2 版本起开始 支持 CORS。您可在 Spring Boot 应用程序中使用 `@CrossOrigin` 注解 配置控制器方法启用 CORS。还可以通过注册一个 `WebMvcConfigurer` bean 并自定义 `addCorsMappings(CorsRegistry)` 方法来定义 全局 CORS 配置 :

```

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

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

```

5.7.2. Spring WebFlux 框架

Spring WebFlux 是 Spring Framework 5.0 中新引入的一个响应式 Web 框架。与 Spring MVC 不同，它不需要 Servlet API，完全异步且无阻塞，并通过 [Reactor 项目](#) 实现响应式流（[Reactive Streams](#)）规范。

```

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

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

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

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

```

“`WebFlux.fn`” 为函数式调用方式，它将路由配置与请求处理分开，如下所示：

```
@Configuration(proxyBeanMethods = false)
public class RoutingConfiguration {

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

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

}

@Component
public class UserHandler {

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

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

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

WebFlux 是 Spring Framework 的一部分, 详细信息可查看其 [参考文档](#).



您可以根据需要定义尽可能多的 **RouterFunction** bean
来模块化路由定义. 如果需要设定优先级, Bean 可以指定顺序.

首先, 将 **spring-boot-starter-webflux** 模块添加到您的应用程序中.



在应用程序中同时添加 `spring-boot-starter-web` 和 `spring-boot-starter-webflux` 模块会导致 Spring Boot 自动配置 Spring MVC，而不是使用 `WebFlux`。这样做的原因是因为许多 Spring 开发人员将 `spring-boot-starter-webflux` 添加到他们的 Spring MVC 应用程序中只是为了使用响应式 `WebClient`。您仍然可以通过设置 `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)` 来强制执行您选择的应用程序类型。

Spring WebFlux 自动配置

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

自动配置在 Spring 的默认基础上添加了以下功能：

- 为 `HttpMessageReader` 和 `HttpMessageWriter` 实例配置编解码器（[稍后将介绍](#)）
- 支持提供静态资源，包括对 `WebJars` 的支持（[稍后将介绍](#)）。

如果你要保留 Spring Boot WebFlux 功能并且想要添加其他 `WebFlux` 配置，可以添加自己的 `@Configuration` 类，类型为 `WebFluxConfigurer`，但不包含 `@EnableWebFlux`。

如果您想完全控制 Spring WebFlux，可以将 `@EnableWebFlux` 注解到自己的 `@Configuration`。

使用 `HttpMessageReader` 和 `HttpMessageWriter` 作为 HTTP 编解码器

Spring WebFlux 使用 `HttpMessageReader` 和 `HttpMessageWriter` 接口来转换 HTTP 的请求和响应。它们通过检测 `classpath` 中可用的类库，配置了 `CodecConfigurer` 生成合适的默认值。

Spring Boot 通过使用 `CodecCustomizer` 实例加强定制。例如，`spring.jackson.*` 配置 key 应用于 Jackson 编解码器。

如果需要添加或自定义编解码器，您可以创建一个自定义的 `CodecCustomizer` 组件，如下所示：

```

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

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

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

}

```

您还可以利用 [Boot 自定义 JSON 序列化器和反序列化器](#).

静态内容

默认情况下, Spring Boot 将在 `classpath` 或者 `ServletContext` 根目录下从名为 `/static` (`/public`、`/resources` 或 `/META-INF/resources`) 目录中服务静态内容. 它使用了 Spring WebFlux 的 `ResourceWebHandler`, 因此您可以通过添加自己的 `WebFluxConfigurer` 并重写 `addResourceHandlers` 方法来修改此行为.

默认情况下, 资源被映射到 `/`, 但可以通过 `spring.webflux.static-path-pattern` 属性调整. 比如, 将所有资源重定位到 `/resources/`:

```

spring:
  webflux:
    static-path-pattern: "/resources/**"

```

您还可以使用 `spring.web.resources.static-locations` 属性来自定义静态资源的位置 (使用一个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自定义的位置. 因此, 如果启动时有任何其中一个位置存在 `index.html`, 那么它将是应用程序的主页.

除了上述提到的标准静态资源位置之外, 还有一种特殊情况是用于 [Webjars 内容](#). 如果以 `Webjar` 格式打包, 则所有符合 `/webjars/**` 的资源都将从 `jar` 文件中服务.



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

欢迎页面

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

模板引擎

除了 REST web 服务之外, 您还可以使用 Spring WebFlux 来服务动态 HTML 内容. Spring WebFlux 支持多种模板技术, 包括 Thymeleaf、FreeMarker 和 Mustache.

Spring Boot 包含了以下的模板引擎的自动配置支持:

- [FreeMarker](#)
- [Thymeleaf](#)
- [Mustache](#)

当您使用这些模板引擎的其中一个并附带了默认配置时, 您的模板将从 `src/main/resources/templates` 自动获取.

Error Handling

Spring Boot 提供了一个 `WebExceptionHandler`, 它以合理的方式处理所有错误. 它在处理顺序中的位置紧接在 WebFlux 提供的处理器之前, 这些处理器排序是最后的. 对于机器客户端, 它会生成一个 JSON 响应, 其中包含错误详情、HTTP 状态和异常消息. 对于浏览器客户端, 有一个 `whitelabel` 错误处理器, 它以 HTML 格式呈现同样的数据. 您还可以提供自己的 HTML 模板来显示错误 (请参阅[下一节](#)).

自定义此功能的第一步通常会沿用现有机制, 但替换或扩充了错误内容. 为此, 您可以添加 `ErrorAttributes` 类型的 bean.

想要更改错误处理行为, 可以实现 `ErrorWebExceptionHandler` 并注册该类型的 bean. 因为 `WebExceptionHandler` 是一个非常底层的异常处理器, 所以 Spring Boot

还提供了一个方便的 `AbstractErrorWebExceptionHandler` 来让你以 `WebFlux` 的方式处理错误, 如下所示:

```
public class CustomErrorWebExceptionHandler extends  
AbstractErrorWebExceptionHandler {  
  
    // Define constructor here  
  
    @Override  
    protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes  
errorAttributes) {  
  
        return RouterFunctions  
            .route(aPredicate, aHandler)  
            .andRoute(anotherPredicate, anotherHandler);  
    }  
  
}
```

要获得更完整的功能, 您还可以直接继承 `DefaultErrorWebExceptionHandler` 并覆盖相关方法.

自定义错误页面

如果您想在自定义的 `HTML` 错误页面上显示给定的状态码, 请将文件添加到 `/error` 目录中. 错误页面可以是静态 `HTML` (添加在任意静态资源目录下) 或者使用模板构建. 文件的名称应该是确切的状态码或者一个序列掩码.

例如, 要将 `404` 映射到一个静态 `HTML` 文件, 目录结构可以如下:

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

使用 `Mustache` 模板来映射所有 `5xx` 错误, 目录的结构如下:

```

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

```

Web 过滤器

Spring WebFlux 提供了一个 `WebFilter` 接口, 可以通过实现该接口来过滤 HTTP 请求 / 响应消息交换. 在应用程序上下文中找到的 `WebFilter` bean 将自动用于过滤每个消息交换.

如果过滤器的执行顺序很重要, 则可以实现 `Ordered` 接口或使用 `@Order` 注解来指定顺序. Spring Boot 自动配置可能为您配置了几个 Web 过滤器. 执行此操作时, 将使用下表中的顺序:

Web Filter	Order
<code>MetricsWebFilter</code>	<code>Ordered.HIGHEST_PRECEDENCE + 1</code>
<code>WebFilterChainProxy</code> (Spring Security)	-100
<code>HttpTraceWebFilter</code>	<code>Ordered.LOWEST_PRECEDENCE - 10</code>

5.7.3. JAX-RS 与 Jersey

如果您喜欢 JAX-RS 编程模型的 REST 端点, 则可以使用一个实现来替代 Spring MVC. Jersey 和 Apache CXF 都能开箱即用. CXF 要求在应用程序上下文中以 `@Bean` 的方式将它注册为一个 `Servlet` 或者 `Filter`. Jersey 有部分原生 Spring 支持, 所以我们也在 `starter` 中提供了与 Spring Boot 整合的自动配置支持.

要使用 Jersey, 只需要将 `spring-boot-starter-jersey` 作为依赖引入, 然后您需要一个 `ResourceConfig` 类型的 `@Bean`, 您可以在其中注册所有端点:

```

@Component
public class JerseyConfig extends ResourceConfig {

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

}

```



Jersey 对于扫描可执行归档文件的支持是相当有限的。例如，它无法扫描一个完整的可执行 jar 文件中的端点，同样，当运行一个可执行的 war 文件时，它也无法扫描包中 WEB-INF/classes 下的端点。为了避免该限制，您不应该使用 packages 方法，应该使用上述的 register 方法来单独注册每一个端点。

您可以注册任意数量实现了 `ResourceConfigCustomizer` 的 bean，以实现更高级的定制化。

所有注册的端点都应注解了 `@Components` 并具有 HTTP 资源注解（`@GET` 等），例如：

```

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

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

}

```

由于 `Endpoint` 是一个 Spring `@Component`，它的生命周期由 Spring 管理，您可以使用 `@Autowired` 注入依赖并使用 `@Value` 注入外部配置。默认情况下，Jersey servlet 将被注册并映射到 `/*`。您可以通过将 `@ApplicationPath` 添加到 `ResourceConfig` 来改变此行为。

默认情况下，Jersey 在 `ServletRegistrationBean` 类型的 `@Bean` 中被设置为一个名为 `jerseyServletRegistration` 的 `Servlet`。默认情况下，该 `servlet` 将被延迟初始化，您可以使用 `spring.jersey.servlet.load-on-startup` 自定义。

您可以禁用或通过创建一个自己的同名 bean 来覆盖该 bean。您还可以通过设置

`spring.jersey.type=filter` 使用过滤器替代 `servlet` (该情况下, 替代或覆盖 `@Bean` 的为 `jerseyFilterRegistration`) . 该过滤器有一个 `@Order`, 您可以使用 `spring.jersey.filter.order` 设置. 可以使用 `spring.jersey.init.*` 指定一个 `map` 类型的 `property` 以给定 `servlet` 和过滤器的初始化参数.

这里有一个 Jersey 示例, 您可以解如何设置.

5.7.4. 内嵌 `Servlet` 容器支持

Spring Boot 包含了对内嵌 `Tomcat`, `Jetty`, 和 `Undertow` 服务器的支持. 大部分开发人员只需简单地使用对应的 `Starter` 来获取完整的配置实例. 默认情况下, 内嵌服务器将监听 `8080` 上的 HTTP 请求.

`Servlets`, `Filters`, 与 `listeners`

使用内嵌 `servlet` 容器时, 您可以使用 Spring bean 或者扫描方式来注册 Servlet 规范中的 `Servlet`、`Filter` 和所有监听器 (比如 `HttpSessionListener`) .

将 `Servlet`、`Filter` 和 `Listener` 注册为 Spring

任何 `Servlet`、`Filter` 或 `*Listener` 的 Spring bean 实例都将被注册到内嵌容器中. 如果您想引用 `application.properties` 中的某个值, 这可能会特别方便.

默认情况下, 如果上下文只包含单个 `Servlet`, 它将映射到 `/`. 在多个 `Servlet` bean 的情况下, `bean` 的名称将用作路径的前缀. `Filter` 将映射到 `/*`.

如果基于约定配置的映射不够灵活, 您可以使用 `ServletRegistrationBean`、`FilterRegistrationBean` 和 `ServletListenerRegistrationBean` 类来完全控制.

通常把过滤器 `bean` 无序是安全的. 如果需要特定的顺序, 则应使用 `@Order` 注解 `Filter` 或使其实现 `Ordered`. 您不能通过使用 `@Order` 注解 `Filter` 的 `bean` 方法来配置 `Filter` 的顺序. 如果您不能更改 `Filter` 类以添加 `@Order` 或实现 `Ordered`, 则必须为 `Filter` 定义一个 `FilterRegistrationBean` 并使用 `setOrder(int)` 方法设置注册 `bean` 的顺序. 则应避免在 `Ordered.HIGHEST_PRECEDENCE` 顺序点配置读取请求体的过滤器, 因为它的字符编码可能与应用程序的字符编码配置不一致. 如果一个 `Servlet` 过滤器包装了请求, 则应使用小于或等于 `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER` 的顺序点对其进行配置.



要查看应用程序中每个过滤器的顺序,请为 `web logging group` (`logging.level.web=debug`) 启用调试级别的日志记录. 然后,将在启动时记录已注册过滤器的详细信息,包括其顺序和URL模式. .



注册 `Filter` Bean时要小心,因为它们是在应用程序生命周期中很早就初始化的. 如果需要注册与其他bean交互的 `Filter`,请考虑改用 `DelegatingFilterProxyRegistrationBean` .

Servlet 上下文初始化

内嵌 `servlet` 容器不会直接执行 `Servlet 3.0+` 的 `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,旨在降低在 `war` 内运行时第三方类库产生的风险,防止破坏 Spring Boot 应用程序.

如果您需要在 Spring Boot 应用程序中执行 `servlet` 上下文初始化,则应注册一个实现了 `org.springframework.boot.context.embedded.ServletContextInitializer` 接口的 bean. `onStartup` 方法提供了针对 `ServletContext` 的访问入口,如果需要,它可以容易作为现有 `WebApplicationInitializer` 的适配器.

扫描 `Servlet`、`Filter` 和 `Listener`

使用内嵌容器时,可以使用 `@ServletComponentScan` 启用带 `@WebServlet`、`@WebFilter` 和 `@WebListener` 注解的类自动注册.



`@ServletComponentScan` 在独立 (standalone) 容器中不起作用,因容器将使用内置发现机制来代替.

ServletWebServerApplicationContext

Spring Boot 底层使用了一个不同的 `ApplicationContext` 类型来支持内嵌 `servlet`. `ServletWebServerApplicationContext` 是一个特殊 `WebApplicationContext` 类型,它通过搜索单个 `ServletWebServerFactory` bean 来引导自身. 通常, `TomcatServletWebServerFactory`、`JettyServletWebServerFactory` 或者

`UndertowServletWebServerFactory` 中的一个将被自动配置.



通常, 你不需要知道这些实现类. 大部分应用程序会自动配置, 并为您创建合适的 `ApplicationContext` 和 `ServletWebServerFactory`.

自定义内嵌 `Servlet` 容器

可以使用 `Spring Environment` 属性来配置通用的 `servlet` 容器设置. 通常, 您可以在 `application.properties` 或 `application.yaml` 文件中定义这些属性.

常用服务器设置包括:

- 网络设置: 监听 HTTP 请求的端口 (`server.port`) , 绑定接口地址到 `server.address` 等.
- 会话设置: 是否持久会话 (`server.session.persistence`) 、 session 超时 (`server.session.timeout`) 、 会话数据存放位置 (`server.session.store-dir`) 和 session-cookie 配置 (`server.session.cookie.*`) .
- 错误管理: 错误页面位置 (`server.error.path`) 等.
- SSL
- HTTP 压缩

`Spring Boot` 尽可能暴露通用的设置, 但并不总是都可以. 针对这些情况, 专用的命名空间为特定的服务器提供了自定义功能 (请参阅 `server.tomcat` 和 `server.undertow`). 例如, 您可以使用内嵌 `servlet` 容器的特定功能来配置 `access logs`.



有关完整的内容列表, 请参阅 `ServerProperties` 类.

以编程方式自定义

如果您需要以编程的方式配置内嵌 `servlet` 容器, 可以注册一个实现了 `WebServerFactoryCustomizer` 接口的 Spring bean. `WebServerFactoryCustomizer` 提供了对 `ConfigurableServletWebServerFactory` 的访问入口, 其中包含了许多自定义 `setter` 方法. 以下示例使用了编程方式来设置端口:

```

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

@Component
public class CustomizationBean implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

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

}

```

`TomcatServletWebServerFactory`, `JettyServletWebServerFactory` 和 `UndertowServletWebServerFactory` 是 `ConfigurableServletWebServerFactory` 的具体子类, 它们分别为 Tomcat、Jetty 和 Undertow 提供了额外的自定义 `setter` 方法。以下示例显示如何自定义 `TomcatServletWebServerFactory`, 它提供对于 Tomcat 的配置选项的访问:

```

@Component
public class TomcatServerCustomizerExample implements
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory server) {
        server.addConnectorCustomizers(
            (tomcatConnector) ->
        tomcatConnector.setAsyncTimeout(Duration.ofSeconds(20).toMillis()));
    }

}

```

直接自定义 `ConfigurableServletWebServerFactory`

For more advanced use cases that require you to extend from `ServletWebServerFactory`, you can expose a bean of such type yourself. 对于需要从 `ServletWebServerFactory` 扩展的更高级的用例, 您可以自己暴露这种类型的 bean.

`Setter` 方法提供了许多配置选项。还有几个 `hook` 保护方法供您深入定制。有关详细信息，请参阅 [源码文档](#)。



自动配置的定制器仍会应用到您的定制工厂，因此请谨慎使用该选项。

JSP 局限

当运行使用了内嵌 `servlet` 容器的 Spring Boot 应用程序时（打包为可执行归档文件），JSP 支持将存在一些限制。

- 如果您使用 `war` 打包，在 Jetty 和 Tomcat 中可以正常工作，使用 `java -jar` 启动时，可执行的 `war` 可正常使用，并且还可以部署到任何标准容器。使用可执行 `jar` 时不支持 JSP。
- Undertow 不支持 JSP。
- 创建自定义的 `error.jsp` 页面不会覆盖默认错误处理视图，应该使用 [自定义错误页面](#)来代替。

5.7.5. 内嵌响应式服务器支持

Spring Boot 包括对以下内嵌响应式 Web 服务器的支持：Reactor Netty、Tomcat、Jetty 和 Undertow。大多数开发人员使用对应的 Starter 来获取一个完全配置的实例。默认情况下，内嵌服务器在 8080 端口上监听 HTTP 请求。

5.7.6. 响应式服务器资源配置

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

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

- 用于服务器和客户端的相同技术
- 客户端实例使用了 Spring Boot 自动配置的 `WebClient.Builder` bean 构建。

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

您可以在 [WebClient Runtime](#) 章节中了解有关客户端资源配置的更多内容。

5.8. Graceful shutdown

所有四个嵌入式 Web 服务器 (Jetty, Reactor Netty, Tomcat 和 Undertow) 以及响应式和基于 Servlet 的 Web 应用程序都支持正常关机。它是关闭应用程序上下文的一部分，并且在停止 SmartLifecycle bean 的最早阶段执行。该停止处理使用一个超时，该超时提供一个宽限期，在此宽限期内，现有请求将被允许完成，而新请求将不被允许。不允许新请求的确切方式因所使用的 Web 服务器而异。Jetty, Reactor Netty 和 Tomcat 将停止在网络层接受请求。Undertow 将接受请求，但会立即以服务不可用 (503) 响应进行响应。



使用 Tomcat 正常关机需要 Tomcat 9.0.33 或更高版本。

要启用正常关机，请配置 `server.shutdown` 属性，如以下示例所示：

```
server:  
  shutdown: "graceful"
```

要配置超时时间，请配置 `spring.lifecycle.timeout-per-shutdown-phase` 属性，如以下示例所示：

```
spring:  
  lifecycle:  
    timeout-per-shutdown-phase: "20s"
```



如果 IDE 无法发送正确的 SIGTERM 信号，则在其 IDE 中使用正常关机可能无法正常工作。有关更多详细信息，请参阅 IDE 的文档。

5.9. RSocket

RSocket 是用于字节流传输的二进制协议。

它通过通过单个连接传递的异步消息来启用对称交互模型。

Spring 框架的 `spring-messaging` 模块在客户端和服务器端都支持 RSocket

请求者和响应者。有关更多详细信息,请参见 Spring Framework 参考中的 [RSocket 部分](#),其中包括 RSocket 协议的概述。

5.9.1. RSocket策略自动配置

Spring Boot 自动配置一个 `RSocketStrategies` bean,该 bean 提供了编码和解码 RSocket 有效负载所需的所有基础结构。默认情况下,自动配置将尝试(按顺序)配置以下内容:

1. Jackson 的 `CBOR` 编解码器
2. Jackson 的 `JSON` 编解码器

`spring-boot-starter-socket` 启动器提供了两个依赖。查阅 [Jackson支持部分](#),以了解有关定制可能性的更多信息。

开发人员可以通过创建实现 `RSocketStrategiesCustomizer` 接口的bean来自定义 `RSocketStrategies` 组件。请注意,它们的 `@Order` 很重要,因为它确定编解码器的顺序。

5.9.2. RSocket 服务器自动配置

Spring Boot 提供了 RSocket 服务器自动配置。所需的依赖由 `spring-boot-starter-rsocket` 提供。

Spring Boot 允许从 WebFlux 服务器通过 WebSocket 暴露 RSocket,或支持独立的 RSocket 服务器。这取决于应用程序的类型及其配置。

对于 WebFlux 应用程序(即 `WebApplicationType.REACTIVE` 类型),仅当以下属性匹配时,RSocket 服务器才会插入 Web 服务器:

```
spring:  
  rsocket:  
    server:  
      mapping-path: "/rsocket"  
      transport: "websocket"
```



由于 RSocket 本身是使用该库构建的,因此只有 Reactor Netty 支持将 RSocket 插入 Web 服务器。

另外, RSocket TCP 或 Websocket 服务器也可以作为独立的嵌入式服务器启动.

除了依赖性要求之外, 唯一需要的配置是为该服务器定义端口:

```
spring:  
  rsocket:  
    server:  
      port: 9898
```

5.9.3. Spring Messaging RSocket 支持

Spring Boot 将为 RSocket 自动配置 Spring Messaging 基础结构.

这意味着 Spring Boot 将创建一个 **RSocketMessageHandler** bean, 该 bean 将处理对您的应用程序的 RSocket 请求.

5.9.4. 使用 RSocketRequester 调用 RSocket 服务

在服务器和客户端之间建立 RSocket 通道后, 任何一方都可以向另一方发送或接收请求.

作为服务器, 您可以在 RSocket **@Controller** 的任何处理程序方法上注入 **RSocketRequester** 实例. 作为客户端, 您需要首先配置和建立 RSocket 连接. 在这种情况下, Spring Boot 会使用预期的编解码器自动配置 **RSocketRequester.Builder**.

RSocketRequester.Builder 实例是一个原型 bean, 这意味着每个注入点将为您提供一个新实例. 这样做是有目的的, 因为此构建器是有状态的, 因此您不应使用同一实例创建具有不同设置的请求者.

以下代码显示了一个典型示例:

```

@Service
public class MyService {

    private final Mono<RSocketRequester> rsocketRequester;

    public MyService(RSocketRequester.Builder rsocketRequesterBuilder) {
        this.rsocketRequester = rsocketRequesterBuilder
            .connectTcp("example.org", 9898).cache();
    }

    public Mono<User> someRSocketCall(String name) {
        return this.rsocketRequester.flatMap(req ->
            req.route("user").data(name).retrieveMono(User.class));
    }

}

```

5.10. 安全

默认情况下,如果 [Spring Security](#) 在 `classpath` 上,则 Web 应用程序是受保护的. Spring Boot 依赖 Spring Security 的内容协商策略来确定是使用 `httpBasic` 还是 `formLogin`. 要给 Web 应用程序添加方法级别的安全保护,可以使用 `@EnableGlobalMethodSecurity` 注解设置. 有关更多其他信息,您可以在 [Spring Security](#) 参考指南中找到.

默认的 `UserDetailsService` 只有一个用户. 用户名为 `user`,密码是随机的,在应用程序启动时会以 `INFO` 级别打印出来,如下所示:

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



如果您对日志配置进行微调,请确保将 `org.springframework.boot.autoconfigure.security` 的级别设置为 `INFO`. 否则,默认密码不会打印出来.

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

您在 Web 应用程序中默认会获得以下基本功能:

- 一个 `UserDetailsService` (或 `WebFlux` 应用程序中的 `ReactiveUserDetailsService`) `bean`, 采用内存存储形式, 有一个自动生成密码的用户 (有关用户属性, 请参阅 `SecurityProperties.User`) .
- 用于整个应用程序 (如果 `actuator` 在 `classpath` 上, 则包括 `actuator` 端点) 基于表单登录或 `HTTP Basic` 认证 (取决于 `Content-Type`) .
- 一个用于发布身份验证事件的 `DefaultAuthenticationEventPublisher`.

您可以通过为其添加一个 `bean` 来提供不同的 `AuthenticationEventPublisher`.

5.10.1. MVC 安全

默认的安全配置在 `SecurityAutoConfiguration` 和 `UserDetailsServiceAutoConfiguration` 中实现. `SecurityAutoConfiguration` 导入用于 Web 安全的 `SpringBootWebSecurityConfiguration`, `UserDetailsServiceAutoConfiguration` 配置身份验证, 这同样适用于非 Web 应用程序. 要完全关闭默认的 Web 应用程序安全配置, 可以添加 `SecurityFilterChain` 类型的 `bean` (这样做不会禁用 `UserDetailsService` 配置或 `Actuator` 的安全保护) .

要同时关闭 `UserDetailsService` 配置, 您可以添加 `UserDetailsService`、`AuthenticationProvider` 或 `AuthenticationManager` 类型的 `bean`. Spring Boot 示例中有几个使用了安全保护的应用程序, 他们或许可以帮助到您.

可以通过添加自定义 `SecurityFilterChain` 或 `WebSecurityConfigurerAdapter` 来重写访问规则. Spring Boot 提供了便捷方法, 可用于重写 `actuator` 端点和静态资源的访问规则. `EndpointRequest` 可用于创建一个基于 `management.endpoints.web.base-path` 属性的 `RequestMatcher`. `PathRequest` 可用于为常用位置中的资源创建一个 `RequestMatcher`.

5.10.2. WebFlux 安全

与 Spring MVC 应用程序类似, 您可以通过添加 `spring-boot-starter-security` 依赖来保护 WebFlux 应用程序. 默认的安全配置在 `ReactiveSecurityAutoConfiguration` 和 `UserDetailsServiceAutoConfiguration` 中实现. `ReactiveSecurityAutoConfiguration` 导入用于 Web 安全的 `WebFluxSecurityConfiguration`, `UserDetailsServiceAutoConfiguration`

配置身份验证,这同样适用于非 Web 应用程序. 要完全关闭默认的 Web 应用程序安全配置,可以添加 `WebFilterChainProxy` 类型的 bean (这样做不会禁用 `UserDetailsService` 配置或 `Actuator` 的安全保护) .

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

可以通过添加自定义 `SecurityWebFilterChain` 来重写访问规则. Spring Boot 提供了便捷方法,可用于重写 `actuator` 端点和静态资源的访问规则. `EndpointRequest` 可用于创建一个基于 `management.endpoints.web.base-path` 属性的 `ServerWebExchangeMatcher`.

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

例如,您可以通过添加以下内容来自定义安全配置:

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

5.10.3. OAuth2

`OAuth2` 是 Spring 支持的一种广泛使用的授权框架.

客户端

如果您的 `classpath` 上有 `spring-security-oauth2-client`, 则可以利用一些自动配置来轻松设置 OAuth2/Open ID Connect 客户端. 该配置使用 `OAuth2ClientProperties` 的属性. 相同的属性适用于 `servlet` 和响应式应用程序.

您可以在 `spring.security.oauth2.client` 前缀下注册多个 OAuth2 客户端和提供者

(provider) ,如下所示:

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

          my-client-2:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for email scope"
            provider: "my-oauth-provider"
            scope: "email"
            redirect-uri: "https://my-redirect-uri.com"
            client-authentication-method: "basic"
            authorization-grant-type: "authorization_code"

        provider:
          my-oauth-provider:
            authorization-uri: "https://my-auth-server/oauth/authorize"
            token-uri: "https://my-auth-server/oauth/token"
            user-info-uri: "https://my-auth-server/userinfo"
            user-info-authentication-method: "header"
            jwk-set-uri: "https://my-auth-server/token_keys"
            user-name-attribute: "name"
```

对于支持 [OpenID Connect discovery](#) 的 OpenID Connect 提供者，可以进一步简化配置。需要使用 `issuer-uri` 配置提供者, `issuer-uri` 是其 Issuer Identifier 的 URI。例如,如果提供的 `issuer-uri` 是 "<https://example.com>"，则将对 "<https://example.com/.well-known/openid-configuration>" 发起一个 [OpenID Provider Configuration Request](#)。期望结果是一个 [OpenID Provider Configuration Response](#)。以下示例展示了如何使用 `issuer-uri` 配置一个 OpenID Connect Provider:

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

默认情况下, Spring Security 的 `OAuth2LoginAuthenticationFilter` 仅处理与 `/login/oauth2/code/*` 相匹配的 URL. 如果要自定义 `redirect-uri` 以使用其他匹配模式, 则需要提供配置以处理该自定义模式. 例如, 对于 `servlet` 应用程序, 您可以添加类似于以下 `SecurityFilterChain`:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
    http
        .authorizeRequests()
            .anyRequest().authenticated()
            .and()
        .oauth2Login()
            .redirectionEndpoint()
                .baseUri("/custom-callback");
    return http.build();
}
```

Spring Boot 自动配置一个 `InMemoryOAuth2AuthorizedClientService`, Spring Security 使用它来管理客户端注册.



`InMemoryOAuth2AuthorizedClientService` 的功能有限, 我们建议仅将其用于开发环境. 对于生产环境, 请考虑使用 `JdbcOAuth2AuthorizedClientService` 或创建自己的 `OAuth2AuthorizedClientService` 实现.

OAuth2 客户端注册常见的提供者

对于常见的 OAuth2 和 OpenID 提供者 (provider), 包括 Google、Github、Facebook 和 Okta, 我们提供了一组提供者默认设置 (分别是 `google`、`github`、`facebook` 和 `okta`).

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

换而言之，以下示例中的两个配置使用了 Google 提供者：

```
spring:  
  security:  
    oauth2:  
      client:  
        registration:  
          my-client:  
            client-id: "abcd"  
            client-secret: "password"  
            provider: "google"  
          google:  
            client-id: "abcd"  
            client-secret: "password"
```

资源服务器

如果在 `classpath` 上有 `spring-security-oauth2-resource-server`，只要指定了 JWK Set URI 或 OIDC Issuer URI，Spring Boot 就可以设置 OAuth2 资源服务器，如下所示：

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

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



如果授权服务器不支持 JWK 设置 URI, 则可以使用用于验证 JWT 签名的公共密钥来配置资源服务器。可以使用 `spring.security.oauth2.resourceserver.jwt.public-key-location` 属性来完成此操作, 该属性值需要指向包含 PEM 编码的 x509 格式的公钥的文件。

相同的属性适用于 `servlet` 和响应式应用程序。

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

如果使用不透明令牌而不是 JWT, 则可以配置以下属性以通过自省来验证令牌:

```
spring:  
  security:  
    oauth2:  
      resourceserver:  
        opaquetoken:  
          introspection-uri: "https://example.com/check-token"  
          client-id: "my-client-id"  
          client-secret: "my-client-secret"
```

同样, 相同的属性适用于 `servlet` 和响应式应用程序。

另外, 您可以为 `Servlet` 应用程序定义自己的 `OpaqueTokenIntrospector` Bean, 或者为响应式应用程序定义 `ReactiveOpaqueTokenIntrospector`。

授权服务器

目前, Spring Security 没有提供 OAuth 2.0 授权服务器实现。但此功能可从 [Spring Security OAuth](#) 项目获得, 该项目最终会被 Spring Security 所取代。在此之前, 您可以使用 `spring-security-oauth2-autoconfigure` 模块轻松设置 OAuth 2.0 授权服务器, 请参阅 [其文档](#) 以获取详细信息。

5.10.4. SAML 2.0

依赖方

如果您在类路径中具有 `spring-security-saml2-service-provider`, 则可以利用一些自动配置功能来轻松设置 SAML 2.0 依赖方。此配置利用 `Saml2RelyingPartyProperties` 下的属性。

依赖方注册代表身份提供商 IDP 和服务提供商 SP 之间的配对配置。您可以在 `spring.security.saml2.relyingparty` 前缀下注册多个依赖方, 如以下示例所示:

```
spring:
  security:
    saml2:
      relyingparty:
        registration:
          my-relying-party1:
            signing:
              credentials:
                - private-key-location: "path-to-private-key"
                  certificate-location: "path-to-certificate"
            decryption:
              credentials:
                - private-key-location: "path-to-private-key"
                  certificate-location: "path-to-certificate"
            identityprovider:
              verification:
                credentials:
                  - certificate-location: "path-to-verification-cert"
                    entity-id: "remote-idp-entity-id1"
                    sso-url: "https://remoteidp1.sso.url"
          my-relying-party2:
            signing:
              credentials:
                - private-key-location: "path-to-private-key"
                  certificate-location: "path-to-certificate"
            decryption:
              credentials:
                - private-key-location: "path-to-private-key"
                  certificate-location: "path-to-certificate"
            identityprovider:
              verification:
                credentials:
                  - certificate-location: "path-to-other-verification-cert"
                    entity-id: "remote-idp-entity-id2"
                    sso-url: "https://remoteidp2.sso.url"
```

5.10.5. Actuator 安全

出于安全考虑,默认情况下禁用除 `/health` 和 `/info` 之外的所有 `actuator`. 可用 `management.endpoints.web.exposure.include` 属性启用 `actuator`.

如果 Spring Security 位于 `classpath` 上且没有其他 `WebSecurityConfigurerAdapter` 或 `SecurityFilterChain`, 则除了 `/health` 和 `/info` 之外的所有 `actuator` 都由 Spring Boot 自动配置保护. 如果您定义了自定义 `WebSecurityConfigurerAdapter` 或 `SecurityFilterChain`, 则 Spring Boot 自动配置将不再生效, 您可以完全控制 `actuator` 的访问规则.



在设置 `management.endpoints.web.exposure.include` 之前, 请确保暴露的 `actuator` 没有包含敏感信息和 / 或被防火墙保护亦或受 Spring Security 之类的保护.

跨站请求伪造保护

由于 Spring Boot 依赖 Spring Security 的默认值配置, 因此默认情况下会启用 CSRF 保护. 这意味着当使用默认安全配置时, 需要 `POST` (`shutdown` 和 `loggers` 端点)、`PUT` 或 `DELETE` 的 `actuator` 端点将返回 `403` 禁止访问错误.



我们建议仅在创建非浏览器客户端使用的服务时才完全禁用 CSRF 保护.

有关 CSRF 保护的其他信息, 请参阅 [Spring Security 参考指南](#)

5.11. 使用 SQL 数据库

Spring Framework 为 SQL 数据库提供了广泛的支持. 从直接使用 `JdbcTemplate` 进行 JDBC 访问到完全的对象关系映射 (object relational mapping) 技术, 比如 Hibernate. Spring Data 提供了更多级别的功能, 直接从接口创建的 `Repository` 实现, 并使用了约定从方法名生成查询.

5.11.1. 配置数据源

Java 的 `javax.sql.DataSource` 接口提供了一个使用数据库连接的标准方法. 通常, 数据源使用 URL 和一些凭据信息来建立数据库连接.



查看 “[How-to](#)” 部分 获取更多高级示例
，通常您可以完全控制数据库的配置。

内嵌数据库支持

使用内嵌内存数据库来开发应用程序非常方便的。显然，内存数据库不提供持久存储。
在应用启动时，您需要填充数据库，并在应用程序结束时丢弃数据。



[How-to](#) 部分包含了[如何初始化数据库](#)方面的内容。

Spring Boot 可以自动配置内嵌 [H2](#), [HSQL](#), 和 [Derby](#) 数据库。您不需要提供任何连接 URL，只需为您想要使用的内嵌数据库引入特定的构建依赖。



如果您在测试中使用此功能，您可能会注意到，无论使用了多少应用程序上下文，整个测试套件都会重复使用相同的数据库。

如果您想确保每个上下文都有一个单独的内嵌数据库，则应该将

`spring.datasource.generate-unique-name` 设置为 `true`。

以下是 POM 依赖示例：

```
<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>
```



要自动配置内嵌数据库，您需要一个 `spring-jdbc` 依赖。在这个例子中，它是通过 `spring-boot-starter-data-jpa` 引入。



如果出于某些原因,您需要配置内嵌数据库的连接 URL,则应注意确保禁用数据库的自动关闭功能. 如果您使用 H2,则应该使用 `DB_CLOSE_ON_EXIT=FALSE` 来设置. 如果您使用 HSQLDB,则确保不使用 `shutdown=true`. 禁用数据库的自动关闭功能允许 Spring Boot 控制数据库何时关闭,从而确保一旦不再需要访问数据库时就触发.

连接生产数据库

生产数据库连接也可以使用使用 `DataSource` 自动配置. Spring Boot 使用以下算法来选择一个特定的实现:

`DataSource` 配置

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

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
```



您至少应该使用 `spring.datasource.url` 属性来指定 URL,否则 Spring Boot 将尝试自动配置内嵌数据库.



Spring Boot 可以从 URL 推导出大多数数据库的 JDBC 驱动程序类。如果需要指定特定的类, 可以使用 `configprop:spring.datasource.driver-class-name[]` 属性。



对于要创建的池 `DataSource`, 我们需要能够验证有效的 Driver 类是否可用, 因此我们在使用之前进行检查. 例如, 如果您设置了 `spring.datasource.driver-class-name=com.mysql.jdbc.Driver`, 那么该类必须可加载.

有关更多支持选项, 请参阅 [DataSourceProperties](#) . 这些都是标准选项, 与 [实际的实现](#)

无关。还可以使用各自的前缀 (`spring.datasource.hikari.`
`、spring.datasource.tomcat.` 和 `spring.datasource.dbcp2.*`)
 微调实现特定的设置。请参考您现在使用的连接池实现的文档来获取更多信息。

例如，如果你使用 `Tomcat connection pool`，则可以自定义许多其他设置，如下：

```
spring:
  datasource:
    tomcat:
      max-wait: 10000
      max-active: 50
      test-on-borrow: true
```

如果没有可用连接，则会将连接池设置为等待 **10000ms** 的释放，然后丢弃异常，请将最大连接数限制为 **50**，并在从池中使用它之前验证连接。

连接池支持

Spring Boot 使用以下算法来选择特定实现：

- 出于性能和并发性的考虑，我们更喜欢 `HikariCP` 连接池。如果 `HikariCP` 可用，我们总是选择它。
- 否则，如果 `Tomcat` 池 `DataSource` 可用，我们将使用它。
- 如果 `HikariCP` 和 `Tomcat` 池数据源不可用，但 `Commons DBCP2` 可用，我们将使用它。
- 如果没有 `HikariCP`, `Tomcat` 和 `DBCP2`，并且如果有 `Oracle UCP`，我们将使用它。



如果使用 `spring-boot-starter-jdbc` 或 `spring-boot-starter-data-jpa` “starters”，您将自动获得对 `HikariCP` 的依赖

您完全可以绕过该算法，并通过 `configprop:spring.datasource.type[]` 属性指定要使用的连接池。如果您在 `Tomcat` 容器中运行应用程序，默认提供 `tomcat-jdbc`，这点尤其重要。

可以使用 `DataSourceBuilder` 手动配置其他连接池。如果您定义了自己的 `DataSource` bean，则自动配置将不会触发。`DataSourceBuilder` 支持以下连接池：

- HikariCP
- Tomcat pooling **Datasource**
- Commons DBCP2
- Oracle UCP & **OracleDataSource**
- Spring Framework's **SimpleDriverDataSource**
- H2 **JdbcDataSource**
- PostgreSQL **PgSimpleDataSource**

连接 **JNDI** 数据源

如果要将 Spring Boot 应用程序部署到应用服务器 (Application Server) 上, 您可能想使用应用服务器的内置功能和 JNDI 访问方式来配置和管理数据源.

`spring.datasource.jndi-name` 属性可作为 `spring.datasource.url`、`spring.datasource.username` 和 `spring.datasource.password` 属性的替代方法, 用于从特定的 JNDI 位置访问 **DataSource**. 例如, `application.properties` 中的以下部分展示了如何访问 JBoss AS 定义的 **DataSource**:

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

5.11.2. 使用 **JdbcTemplate**

Spring 的 **JdbcTemplate** 和 **NamedParameterJdbcTemplate** 类是自动配置的, 您可以使用 `@Autowired` 将它们直接注入您的 bean 中:

```

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

@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

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

    // ...
}

```

您可以使用 `spring.jdbc.template.*` 属性来自定义一些 `template` 的属性,如下:

```

spring:
  jdbc:
    template:
      max-rows: 500

```



`NamedParameterJdbcTemplate` 在底层重用了相同的 `JdbcTemplate` 实例. 如果定义了多个 `JdbcTemplate` 且没有声明 `primary` 主候选, 则不会自动配置 `NamedParameterJdbcTemplate`.

5.11.3. JPA 与 Spring Data JPA

Java Persistence API (Java 持久化 API) 是一项标准技术, 可让您将对象映射到关系数据库. `spring-boot-starter-data-jpa` POM 提供了一个快速起步的方法. 它提供了以下关键依赖:

- `Hibernate`: 最受欢迎的 JPA 实现之一.
- `Spring Data JPA`: 帮助你实现基于 JPA 的资源库.
- `Spring ORM`: Spring Framework 的核心 ORM 支持



我们不会在这里介绍太多关于 JPA 或者 Spring Data 的相关内容。
您可以在 spring.io 上查看使用“JPA 访问数据”，获取阅读 Spring Data JPA 和 Hibernate 的参考文档。

实体类

通常，JPA Entity（实体）类是在 `persistence.xml` 文件中指定的。使用了 Spring Boot，该文件将不是必需的，可以使用 Entity Scanning（实体扫描）来代替。默认情况下，将搜索主配置类（使用了 `@EnableAutoConfiguration` 或 `@SpringBootApplication` 注解）下面的所有包。

任何用了 `@Entity`、`@Embeddable` 或者 `@MappedSuperclass` 注解的类将被考虑。

一个典型的实体类如下：

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

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

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;

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

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

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

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

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

    // ... etc
}
```



您可以使用 `@EntityScan` 注解自定义实体类的扫描位置。请参见 “[从 Spring configuration 配置中分离 `@Entity` 定义](#)” 章节。

Spring Data JPA 资源库

Spring Data JPA 资源库 (repository) 是接口, 您可以定义用于访问数据. JPA 查询是根据您的方法名自动创建. 例如, `CityRepository` 接口可以声明 `findAllByState(String state)` 方法来查找指定状态下的所有城市.

对于更加复杂的查询, 您可以使用 Spring Data 的 `Query` 注解

Spring Data 资源库通常继承自 `Repository` 或者 `CrudRepository` 接口. 如果您使用了自动配置, 则将从包含主配置类 (使用了 `@EnableAutoConfiguration` 或 `@SpringBootApplication` 注解) 的包中搜索资源库:

以下是一个典型的 Spring Data 资源库接口定义:

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

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

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

Spring Data JPA 资源库支持三种不同的引导模式: `default`、`deferred` 和 `lazy`. 要启用延迟或懒惰引导, 请将 `spring.data.jpa.repositories.bootstrap-mode` 分别设置为 `deferred` 或 `lazy`. 使用延迟或延迟引导时, 自动配置的 `EntityManagerFactoryBuilder` 将使用上下文的 `AsyncTaskExecutor` (如果有) 作为 `applicationTaskExecutor`.



使用 `deferred` 或 `lazy` bootstrapping 时, 请确保在应用程序上下文阶段之后, 延迟对 JPA 的任何访问. 您可以使用 `SmartInitializingSingleton` 来调用任何需要 JPA 基础结构的初始化. 对于以 Spring Bean 创建的 JPA 组件 (例如转换器), 请使用 `ObjectProvider` 延迟对依赖项的解析 (如果有)



我们几乎没有接触到 Spring Data JPA 的表面内容。有关详细信息，请查阅 [Spring Data JPA 参考文档](#)。

创建和删除 JPA 数据库

默认情况下，仅当您使用了内嵌数据库（H2、HSQL 或 Derby）时才会自动创建 JPA 数据库。

您可以使用 `spring.jpa.*` 属性显式配置 JPA 设置。例如，要创建和删除表，您可以将以下内容添加到 `application.properties` 中：

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



关于上述功能，Hibernate 自己的内部属性名称（如果您记住更好）为 `hibernate.hbm2ddl.auto`。您可以使用 `spring.jpa.properties.*`（在添加到实体管理器之前，该前缀将被删除）来将 Hibernate 原生属性一同设置：

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

上面示例中将 `true` 值设置给 `hibernate.globally_quoted_identifiers` 属性，该属性将传给 Hibernate 实体管理器。

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

在视图中打开 EntityManager

如果您正在运行 web 应用程序，Spring Boot 将默认注册 `OpenEntityManagerInViewInterceptor` 用于在视图中打开 EntityManager 模式，即允许在 web 视图中延迟加载。如果您不想开启这个行为，则应在 `application.properties` 中将 `spring.jpa.open-in-view` 设置为 `false`。

5.11.4. Spring Data JDBC

Spring Data 包含了对 JDBC 资源库的支持，并将自动为 `CrudRepository` 上的方法生成

SQL. 对于更高级的查询, 它提供了 `@Query` 注解.

当 `classpath` 下存在必要的依赖时, Spring Boot 将自动配置 Spring Data 的 JDBC 资源库. 可以通过添加单个 `spring-boot-starter-data-jdbc` 依赖引入到项目中. 如有必要, 可通过在应用程序中添加 `@EnableJdbcRepositories` 注解或 `JdbcConfiguration` 子类来控制 Spring Data JDBC 的配置.



有关 Spring Data JDBC 的完整详细信息, 请参阅 [参考文档](#).

5.11.5. 使用 H2 的 Web 控制台

`H2 database` 数据库提供了一个 [基于浏览器的控制台](#), Spring Boot 可以为您自动配置. 当满足以下条件时, 控制台将自动配置:

- 您开发的是一个基于 `servlet` 的 web 应用程序
- `com.h2database:h2` 在 `classpath` 上
- 您使用了 [Spring Boot 的开发者工具](#).



如果您不使用 Spring Boot 的开发者工具, 但仍希望使用 H2 的控制台, 则可以通过将 `spring.h2.console.enabled` 属性设置为 `true` 来实现.



H2 控制台仅用于开发期间, 因此应注意确保 `spring.h2.console.enabled` 在生产环境中没有设置为 `true`.

更改 H2 控制台的路径

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

5.11.6. 使用 jOOQ

Java 面向对象查询 (Java Object Oriented Querying, [jOOQ](#)) 是一款广受欢迎的产品, 出自 [Data Geekery](#), 它可以通过数据库生成 Java 代码, 并允许您使用流式 API 来构建类型安全的 SQL 查询. 商业版和开源版都可以与 Spring

Boot 一起使用。

代码生成

要使用 jOOQ 的类型安全查询, 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。如果您使用了 `jooq-codegen-maven` 插件, 并且还使用了 `spring-boot-starter-parent` 父 POM, 则可以安全地省略掉插件的 `<version>` 标签。您还可以使用 Spring Boot 定义的版本变量 (例如 `h2.version`) 来声明插件的数据库依赖。以下是一个示例:

```

<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>

```

使用 `DSLContext`

jOOQ 提供的流式 API 是通过 `org.jooq.DSLContext` 接口初始化的。Spring Boot 将自动配置一个 `DSLContext` 作为 Spring Bean, 并且将其连接到应用程序的 `DataSource`。要使用 `DSLContext`, 您只需要 `@Autowired` 它:

```

@Component
public class JooqExample implements CommandLineRunner {

    private final DSLContext create;

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

}

```



jOOQ 手册建议使用名为 `create` 的变量来保存 `DSLContext`.

您可以使用 `DSLContext` 构建查询：

```

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

```

jOOQ SQL 方言

除非配置了 `spring.jooq.sql-dialect` 属性, 否则 Spring Boot 会自动判定用于数据源的 SQL 方言. 如果 Spring Boot 无法检测到方言, 则使用 `DEFAULT`.



Spring Boot 只能自动配置 jOOQ 开源版本支持的方言.

自定义 jOOQ

可通过定义自己的 `@Bean` 来实现更高级的功能, 这些自定义将在创建 jOOQ Configuration 时使用. 您可以为以下 jOOQ 类型定义 bean:

- `ConnectionProvider`
- `ExecutorProvider`

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

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

5.11.7. 使用 R2DBC

响应式关系数据库连接 (`R2DBC`) 项目将响应式编程 API 引入关系数据库. `R2DBC` 的 `io.r2dbc.spi.Connection` 提供了一种处理非阻塞数据库连接的标准方法. 通过 `ConnectionFactory` 提供连接, 类似于使用 `jdbc` 的数据源.

`ConnectionFactory` 配置由 `spring.r2dbc.*` 中的外部配置属性控制. 例如, 您可以在 `application.properties` 中声明以下部分:

```
spring:  
  r2dbc:  
    url: "r2dbc:postgresql://localhost/test"  
    username: "dbuser"  
    password: "dbpass"
```



您不需要指定驱动程序类名称, 因为 Spring Boot 从 `R2DBC` 的 `Connection Factory` 发现中获取驱动程序.



您应该至少提供 `url`. URL 中指定的信息优先于各个属性, 即 `name`, `username`, `password` 和连接池选项.



“How-to” 章节包括有关如何 [初始化数据库的部分](#)

要自定义由 `ConnectionFactory` 创建的连接, 即设置不需要 (或无法) 在中央数据库配置中配置的特定参数, 可以使用 `ConnectionFactoryOptionsBuilderCustomizer @Bean`. 以下示例显示了如何从应用程序配置中获取其余选项的同时手动覆盖数据库端口:

```
@Bean
public ConnectionFactoryOptionsBuilderCustomizer
connectionFactoryPortCustomizer() {
    return (builder) -> builder.option(PORT, 5432);
}
```

以下示例显示了如何设置一些 PostgreSQL 连接选项:

```
@Bean
public ConnectionFactoryOptionsBuilderCustomizer postgresCustomizer() {
    Map<String, String> options = new HashMap<>();
    options.put("lock_timeout", "30s");
    options.put("statement_timeout", "60s");
    return (builder) -> builder.option(OPTIONS, options);
}
```

当 `ConnectionFactory` bean 可用时, 常规 JDBC `DataSource` 自动配置将退出. 如果要保留 JDBC `DataSource` 自动配置, 并且对在响应式应用程序中使用阻塞 JDBC API 的风险感到满意, 请在应用程序的 `@Configuration` 类上添加 `@Import(DataSourceAutoConfiguration.class)` 以重新启用它

嵌入式数据库支持

与 [JDBC 支持类似](#), Spring Boot 可以自动配置嵌入式数据库进行响应式使用. 您无需提供任何连接 URL. 您只需要包括要使用的嵌入式数据库的构建依赖关系, 如以下示例所示:

```
<dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-h2</artifactId>
    <scope>runtime</scope>
</dependency>
```



如果您在测试中使用此功能，则可能会注意到，整个测试套件将重复使用同一数据库，而不管您使用的应用程序上下文有多少。

如果要确保每个上下文都有一个单独的嵌入式数据库，则应将 `spring.r2dbc.generate-unique-name` 设置为 `true`。

使用 DatabaseClient

Spring Data 的 `DatabaseClient` bean 是自动配置的，您可以将其直接 `@Autowired` 到自己的 bean 中，如以下示例所示：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.r2dbc.function.DatabaseClient;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final DatabaseClient databaseClient;

    @Autowired
    public MyBean(DatabaseClient databaseClient) {
        this.databaseClient = databaseClient;
    }

    // ...

}
```

Spring Data R2DBC Repositories

Spring Data R2DBC 存储库是可以定义以访问数据的接口。

查询是根据您的方法名称自动创建的。例如，`CityRepository` 接口可能声明了 `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` 注解对方法进行注解.

Spring Data 存储库通常从 `Repository` 或 `CrudRepository` 接口扩展.
如果您使用自动配置,则会从包含您的主要配置类 (以 `@EnableAutoConfiguration` 或
`@SpringBootApplication` 注解的类) 的包中搜索存储库.

以下示例显示了典型的 Spring Data 存储库接口定义:

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

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;
import reactor.core.publisher.Mono;

public interface CityRepository extends Repository<City, Long> {

    Mono<City> findByNameAndStateAllIgnoringCase(String name, String state);

}
```



我们只讨论了 Spring Data R2DBC 的简单的东西. 如果需要详细信息
, 请参阅 [Spring Data R2DBC reference documentation](#).

5.12. 使用 NoSQL 技术

Spring Data 提供了其他项目来帮助您访问各种NoSQL技术, 包括:

- [MongoDB](#)
- [Neo4J](#)
- [Elasticsearch](#)
- [Solr](#)
- [Redis](#)
- [GemFire or Geode](#)
- [Cassandra](#)

- [Couchbase](#)
- [LDAP](#)

Spring Boot 为 Redis, MongoDB, Neo4j, Elasticsearch, Solr, Cassandra, Couchbase 和 LDAP 提供自动配置。您可以使用其他项目，但必须自己进行配置。请参阅 spring.io/projects/spring-data 中的相应参考文档。

5.12.1. Redis

Redis 是一个集缓存、消息代理和键值存储等丰富功能的数据库。Spring Boot 为 [Lettuce](#) 和 [Jedis](#) 客户端类库提供了基本自动配置，[Spring Data Redis](#) 为他们提供了上层抽象。

使用 `spring-boot-starter-data-redis` starter 可方便地引入相关依赖。默认情况下，它使用 [Lettuce](#)。该 starter 可处理传统应用程序和响应式应用程序。



我们还提供了一个 `spring-boot-starter-data-redis-reactive` starter，以便与其他带有响应式支持的存储保持一致。

连接 Redis

您可以像所有 Spring Bean 一样注入自动配置的 `RedisConnectionFactory`、`StringRedisTemplate` 或普通的 `RedisTemplate` 实例。默认情况下，实例将尝试在 `localhost:6379` 上连接 Redis 服务器，以下是 bean 示例：

```
@Component
public class MyBean {

    private StringRedisTemplate template;

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

    // ...
}
```



您还可以注册任意数量个实现了 `LettuceClientConfigurationBuilderCustomizer` 的 bean, 以进行更高级的自定义. 如果你使用 `Jedis`, 则可以使用 `JedisClientConfigurationBuilderCustomizer`.

如果您添加了自己的任何一个自动配置类型的 `@Bean`, 它将替换默认设置 (除了 `RedisTemplate`, 由于排除是基于 bean 名称, 而 `redisTemplate` 不是它的类型) . 默认情况下, 如果 `commons-pool2` 在 `classpath` 上, 您将获得一个连接池工厂.

5.12.2. MongoDB

`MongoDB` 是一个开源的 NoSQL 文档数据库, 其使用了类似 JSON 的模式 (schema) 来替代传统基于表的关系数据. Spring Boot 为 MongoDB 提供了几种便利的使用方式, 包括 `spring-boot-starter-data-mongodb` 和 `spring-boot-starter-data-mongodb-reactive` starter.

连接 MongoDB 数据库

您可以注入一个自动配置的 `org.springframework.data.mongodb.MongoDbFactory` 来访问 MongoDB 数据库. 默认情况下, 该实例将尝试在 `mongodb://localhost/test` 上连接 MongoDB 服务器, 以下示例展示了如何连接到 MongoDB 数据库:

```

import org.springframework.data.mongodb.MongoDatabaseFactory;
import com.mongodb.client.MongoDatabase;

@Component
public class MyBean {

    private final MongoDatabaseFactory mongo;

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

    // ...

    public void example() {
        MongoDatabase db = mongo.getMongoDatabase();
        // ...
    }

}

```

如果您已经定义了自己的 `MongoClient`, 它将被用于自动配置合适的 `MongoDatabaseFactory`.

使用 `MongoClientSettings` 创建自动配置的 `MongoClient`. 要微调其配置, 请声明一个或多个 `MongoClientSettingsBuilderCustomizer` Bean. 每个命令都将与用于构建 `MongoClientSettings` 的``MongoClientSettings.Builder`` 依次调用.

您可以通过设置 `spring.data.mongodb.uri` 属性来更改 URL 和配置其他设置, 如副本集 (replica set) :

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

或者, 您可以使用 `discrete` 属性指定连接详细信息.

```
spring:  
  data:  
    mongodb:  
      host: "mongoserver.example.com"  
      port: 27017  
      database: "test"  
      username: "user"  
      password: "secret"
```



如果未指定 `spring.data.mongodb.port`, 则使用默认值 **27017**.
您可以将上述示例中的改行配置删除掉.



如果您不使用 Spring Data MongoDB, 则可以注入
`com.mongodb.MongoClient` bean 来代替
`MongoDatabaseFactory`. 如果要完全控制建立 MongoDB 连接
, 您还可以声明自己的 `MongoDatabaseFactory` 或者 `MongoClient`
bean.



如果您使用的是响应式驱动, 则 SSL 需要 Netty. 如果 Netty 可用且
`factory` 尚未自定义, 则自动配置会自动配置此 `factory`.

MongoTemplate

Spring Data MongoDB 提供了一个 `MongoTemplate` 类, 它的设计与 Spring 的 `JdbcTemplate` 非常相似. 与 `JdbcTemplate` 一样, Spring Boot 会自动配置一个 bean, 以便您能注入模板:

```

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

@Component
public class MyBean {

    private final MongoTemplate mongoTemplate;

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

    // ...
}

```

更多详细信息, 参照 [MongoOperations Javadoc](#) .

Spring Data MongoDB Repositories

Spring Data 包含了对 MongoDB 资源库 (repository) 的支持. 与之前讨论的 JPA 资源库一样, 基本原理是根据方法名称自动构建查询.

事实上, Spring Data JPA 和 Spring Data MongoDB 共享通用的底层代码, 因此你可以拿之前提到的 JPA 示例作为基础, 假设 `City` 现在是一个 MongoDB 数据类, 而不是一个 JPA `@Entity`, 他们方式工作相同:

```

package com.example.myapp.domain;

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

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);

}

```



您可以使用 `@EntityScan` 注解来自定义文档扫描位置.



有关 Spring Data MongoDB 的完整详细内容
, 包括其丰富的对象关系映射技术, 请参考其 [参考文档](#).

内嵌 Mongo

Spring Boot 提供了 [内嵌 Mongo](#) 的自动配置. 要在 Spring Boot 应用程序中使用它
, 请添加依赖 `de.flapdoodle.embed:de.flapdoodle.embed.mongo`.

可以使用 `spring.data.mongodb.port` 属性来配置 Mongo 的监听端口.

如果想随机分配空闲端口, 请把值设置为 0. `MongoAutoConfiguration` 创建的
`MongoClient` 将自动配置随机分配的端口.



如果您不配置一个自定义端口, 内嵌支持将默认使用一个随机端口 (而不是
27017) .

如果您的 `classpath` 上有 SLF4J, Mongo 产生的输出将自动路由到名为
`org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo` 的
logger.

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

可以通过声明 `DownloadConfigBuilderCustomizer` bean 来定制下载配置.

5.12.3. Neo4j

[Neo4j](#) 是一个开源的 NoSQL 图形数据库, 它使用了一个节点由关系连接的富数据模型, 比传统
RDBMS 的方式更适合连接大数据. Spring Boot 为 Neo4j 提供了便捷引入方式, 包括
`spring-boot-starter-data-neo4j` starter.

连接 Neo4j 数据库

您可以像任何 Spring Bean 一样注入一个自动配置的 `org.neo4j.driver.Driver`.
默认情况下, 该实例将尝试使用在 `localhost:7687` 上使用 Bolt 协议连接到 Neo4j

服务器,以下示例展示了如何注入一个 Neo4j Driver 它可以让你访问 Session 等:

```
@Component
public class MyBean {

    private final Driver driver;

    @Autowired
    public MyBean(Driver driver) {
        this.driver = driver;
    }

    // ...

}
```

您可以通过配置 `spring.neo4j.*` 属性来设置 uri 和凭据:

```
spring:
  neo4j:
    uri: "bolt://my-server:7687"
    authentication:
      username: "neo4j"
      password: "secret"
```

使用 `ConfigBuilder` 创建自动配置的驱动程序。要微调其配置,请声明一个或多个 `ConfigBuilderCustomizer` Bean。每个都将按顺序调用用于构建驱动程序的 `ConfigBuilder`。

Spring Data Neo4j 资源库

Spring Data 包括了对 Neo4j 资源库的支持.有关 Spring Data Neo4j 的完整细节,请参阅 [reference documentation](#).

与许多其他 Spring Data 模块一样, Spring Data Neo4j 与 Spring Data JPA 共享相同的通用底层代码。您可以采用前面的 JPA 示例,并将 `City` 定义为 Spring Data Neo4j `@Node` 而不是 JPA `@Entity`, 并且资源库抽象以相同的方式工作:

```

package com.example.myapp.domain;

import java.util.Optional;

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

public interface CityRepository extends Neo4jRepository<City, Long> {

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

}

```

`spring-boot-starter-data-neo4j` starter 支持资源库和事务管理。Spring Boot 支持使用 `Neo4JTemplate` 或 `ReactiveNeo4jTemplate` Bean 的传统的和响应式 Neo4J 存储库。当 Project Reactor 在 ClassPath 上提供时，响应式也是自动配置的。

您可以在 `@Configuration` bean 上分别使用 `@EnableNeo4jRepositories` 和 `@EntityScan` 来自定义位置以查找资源库和实体。

在使用响应式的应用程序中，无法自动配置

`ReactiveTransactionManager`。

要启用事务管理，必须在配置中定义以下 bean：



```

@Bean
public ReactiveNeo4jTransactionManager
reactiveTransactionManager(Driver driver,
                           ReactiveDatabaseSelectionProvider
databaseNameProvider) {
    return new ReactiveNeo4jTransactionManager(driver,
databaseNameProvider);
}

```

5.12.4. Solr

`Apache Solr` 是一个搜索引擎。Spring Boot 为 Solr 5 客户端类库提供了基本的自动配置，并且 `Spring Data Solr` 为其提供给了顶层抽象。相关的依赖包含在了 `spring-boot-starter-data-solr` starter 中。



从 Spring Boot 2.3 开始，已弃用对 Spring Data Solr 的支持，并将在以后的发行版中将其删除。

连接 Solr

您可以像其他 Spring Bean 一样注入一个自动配置的 `SolrClient` 实例。默认情况下，该实例将尝试通过 `localhost:8983/solr` 连接到服务器，以下示例展示了如何注入一个 Solr bean：

```
@Component
public class MyBean {

    private SolrClient solr;

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

    // ...
}
```

如果您添加了自己的 `SolrClient` 类型的 `@Bean`，它将替换掉默认配置。

Spring Data Solr 资源库

Spring Data 包含了对 Apache Solr 资源库的支持。与之前讨论的 JPA 资源库一样，基本原理是根据方法名称自动构造查询。

事实上，Spring Data JPA 和 Spring Data Solr 共享了相同的通用底层代码，因此您可以使用之前的 JPA 示例作为基础，假设 `City` 现在是一个 `@SolrDocument` 类，而不是一个 JPA `@Entity`，它的工作方式相同。



有关 Spring Data Solr 的完整详细内容，请参考其 [参考文档](#)。

5.12.5. Elasticsearch

[Elasticsearch](#) 是一个开源、分布式、RESTful 的实时搜索分析引擎。Spring Boot 为 Elasticsearch 提供了基本的自动配置。

Spring Boot 支持以下 HTTP 客户端：

- 官方 Java Low Level (低级) 和 High Level (高级) REST 客户端
- Spring Data Elasticsearch 提供的 `ReactiveElasticsearchClient`

Spring Boot 提供了一个 “Starter”。您可以使用 `spring-boot-starter-data-elasticsearch` starter 引入使用它。

使用 REST 客户端连接 Elasticsearch

Elasticsearch 提供了 [两个可用于查询集群的 REST 客户端](#)：Low Level (低级) 和 High Level (高级)。Spring Boot 提供了对 High Level (高级) 客户端的支持，客户端随 `org.elasticsearch.client:elasticsearch-rest-high-level-client` 一起提供

如果您的 classpath 上存在这个依赖，则 Spring Boot 将自动配置并注册默认目标为 `localhost:9200` 的 `RestHighLevelClient` bean。您可以进一步调整 `RestHighLevelClient` 的配置，如下所示：

```
spring:
  elasticsearch:
    rest:
      uris: "https://search.example.com:9200"
      read-timeout: "10s"
      username: "user"
      password: "secret"
```

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

如果你 classpath 上有 `org.elasticsearch.client:elasticsearch-rest-high-level-client` 依赖，Spring Boot 将自动配置一个 `RestHighLevelClient`，它利用所有现有的 `RestClientBuilder` bean 重用其 HTTP 配置。



如果您的应用程序需要访问 `Low Level` (低级) `restClient`, 则可以通过在自动配置的 `RestHighLevelClient` 上调用 `Client.getLowLevelClient()` 来获取它

使用 `Reactive REST` 客户端连接

`Spring Data Elasticsearch` 提供了 `ReactiveElasticsearchClient`, 用于以响应式查询 `Elasticsearch` 实例. 它基于 `WebFlux` 的 `WebClient` 构建, 因此 `spring-boot-starter-elasticsearch` 和 `spring-boot-starter-webflux` 依赖.

默认情况下, `Spring Boot` 将自动配置并注册一个针对 `localhost:9200` 的 `ReactiveElasticsearchClient` bean. 您可以进一步调整其配置, 如以下示例所示:

```
spring:
  data:
    elasticsearch:
      client:
        reactive:
          endpoints: "search.example.com:9200"
          use-ssl: true
          socket-timeout: "10s"
          username: "user"
          password: "secret"
```

如果配置属性不够, 并且您想完全控制客户端配置, 则可以注册自定义 `ClientConfiguration` bean.

使用 `Spring Data` 连接 `Elasticsearch`

要连接 `Elasticsearch`, 必须定义由 `Spring Boot` 自动配置或由应用程序手动提供的 `RestHighLevelClient` bean (请参阅前面的部分). 有了此配置后, 可以像其他任何 `Spring` bean 一样注入 `ElasticsearchRestTemplate`, 如以下示例所示:

```

@Component
public class MyBean {

    private final ElasticsearchRestTemplate template;

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

    // ...
}

```

如果存在 `spring-data-elasticsearch` 和使用 `WebClient` 所需的依赖（通常是 `spring-boot-starter-webflux`）的情况下，Spring Boot 还可以将 `ReactiveElasticsearchClient` 和 `ReactiveElasticsearchTemplate` 自动配置为 bean。它们与其他 REST 客户端是等效的。

Spring Data Elasticsearch 资源库

Spring Data 包含了对 Elasticsearch 资源库的支持，与之前讨论的 JPA 资源库一样，其原理是根据方法名称自动构造查询。

事实上，Spring Data JPA 与 Spring Data Elasticsearch 共享了相同的通用底层代码，因此您可以使用之前的 JPA 示例作为基础，假设 `City` 此时是一个 Elasticsearch `@Document` 类，而不是一个 JPA `@Entity`，它以相同的方式工作。



有关 Spring Data Elasticsearch 的完整详细内容，请参阅其参考文。

Spring Boot 使用 `ElasticsearchRestTemplate` 或 `ReactiveElasticsearchTemplate` bean 支持经典和响应式 Elasticsearch 资源库。给定所需的依赖，最有可能由 Spring Boot 自动配置这些 bean。

如果您希望使用自己的模板来支持 Elasticsearch 存储库，则可以添加自己的 `ElasticsearchRestTemplate` 或 `ElasticsearchOperations @Bean`，只要它名为 "elasticsearchTemplate" 即可。同样适用于 `ReactiveElasticsearchTemplate` 和 `ReactiveElasticsearchOperations`，其 bean 名称为

```
"reactiveElasticsearchTemplate".
```

您可以选择使用以下属性禁用存储库支持：

```
spring:  
  data:  
    elasticsearch:  
      repositories:  
        enabled: false
```

5.12.6. Cassandra

[Cassandra](#) 是一个开源的分布式数据库管理系统，旨在处理商用服务器上的大量数据。Spring Boot 为 Cassandra 提供了自动配置，且 [Spring Data Cassandra](#) 为其提供了顶层抽象。相关依赖包含在 `spring-boot-starter-data-cassandra` starter 中。

连接 Cassandra

您可以像其他 Spring Bean 一样注入一个自动配置的 [CassandraTemplate](#) 或 Cassandra [CqlSession](#) 实例。`spring.data.cassandra.*` 属性可用于自定义连接。通常，您会提供 `keyspace-name` 和 `contact-points` 以及 `local-datacenter` 属性：

```
spring:  
  data:  
    cassandra:  
      keyspace-name: "mykeyspace"  
      contact-points: "cassandrahost1:9042,cassandrahost2:9042"  
      local-datacenter: "datacenter1"
```

如果所有端口都相同，则可以使用快捷方式，仅指定主机名，如以下示例所示：

```
spring:  
  data:  
    cassandra:  
      keyspace-name: "mykeyspace"  
      contact-points: "cassandrahost1,cassandrahost2"  
      local-datacenter: "datacenter1"
```



这两个示例与默认端口 9042 相同。如果需要配置端口，请使用 `spring.data.cassandra.port`。

Cassandra 驱动程序具有自己的配置基础结构，该结构在类路径的根目录中加载 `application.conf`。



Spring Boot 不会查找此类文件，而是通过 `spring.data.cassandra.*` 命名空间提供了许多配置属性。对于更高级的驱动程序定制，可以注册任意数量的实现 `DriverConfigLoaderBuilderCustomizer` 的 bean。可以使用 `CqlSessionBuilderCustomizer` 类型的 bean 来定制 `CqlSession`。



如果您使用 `CqlSessionBuilder` 创建多个 `CqlSession` Bean，请记住该构建器是可变的，因此请确保为每个会话注入一个新副本。

以下代码展示了如何注入一个 Cassandra bean：

```
@Component
public class MyBean {

    private final CassandraTemplate template;

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

    // ...
}
```

如果您添加了自己的类的为 `@CassandraTemplate` 的 `@Bean`，则其将替代默认值。

Spring Data Cassandra 资源库

Spring Data 包含了基本的 Cassandra 资源库支持。目前，其限制要比之前讨论的 JPA 资源库要多，并且需要在 `finder` 方法上使用 `@Query` 注解。



有关 Spring Data Cassandra 的完整详细内容,请参阅其 [参考文档](#).

5.12.7. Couchbase

Couchbase 是一个开源、分布式多模型的 NoSQL 面向文档数据库,其针对交互式应用程序做了优化. Spring Boot 为 Couchbase 提供了自动配置, 且 Spring Data Couchbase 为其提供了顶层抽象. 相关的依赖包含在了 `spring-boot-starter-data-couchbase starter` 中.

连接 Couchbase

您可以通过添加 Couchbase SDK 和一些配置来轻松获取 `Cluster. spring.couchbase.*` 属性可用于自定义连接. 通常您会提供 `connection string username` 和 `password`:

```
spring:
  couchbase:
    connection-string: "couchbase://192.168.1.123"
    username: "user"
    password: "secret"
```

还可以自定义某些 `ClusterEnvironment` 设置. 例如,以下配置用于打开新的 Bucket 并启用SSL支持的超时:

```
spring:
  couchbase:
    env:
      timeouts:
        connect: "3s"
      ssl:
        key-store: "/location/of/keystore.jks"
        key-store-password: "secret"
```



检查 `spring.couchbase.env.*` 属性以获取更多详细信息. 为了获得更多控制权,可以使用一个或多个 `ClusterEnvironmentBuilderCustomizer bean`.

Spring Data Couchbase 资源库

Spring Data 包含了 Couchbase 资源库支持。有关 Spring Data Couchbase 的完整详细信息,请参阅其 [reference documentation](#).

您可以像使用其他 Spring Bean 一样注入自动配置的 `CouchbaseTemplate` 实例,前提是可以通过使用 `CouchbaseClientFactory` (当您 `Cluster` 可以并且指定了 bucket 名称时会发生这种情况,如之前所述) .

```
spring:  
  data:  
    couchbase:  
      bucket-name: "my-bucket"
```

以下示例展示了如何注入一个 `CouchbaseTemplate` bean:

```
@Component  
public class MyBean {  
  
    private final CouchbaseTemplate template;  
  
    @Autowired  
    public MyBean(CouchbaseTemplate template) {  
        this.template = template;  
    }  
  
    // ...  
}
```

您可以在自己的配置中定义以下几个 bean,以覆盖自动配置提供的配置:

- 一个名为 `couchbaseMappingContext` 的 `CouchbaseMappingContext @Bean`
- 一个名为 `couchbaseCustomConversions` 的 `CustomConversions @Bean`
- 一个名为 `couchbaseTemplate` 的 `CouchbaseTemplate @Bean`

为了避免在自己的配置中硬编码这些名称,您可以重用 Spring Data Couchbase 提供的 `BeanNames`,例如,您可以自定义转换器,如下:

```

@Configuration(proxyBeanMethods = false)
public class SomeConfiguration {

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

    // ...
}

```



如果您想要安全绕开 Spring Data Couchbase 的自动配置
, 请提供自己的
`org.springframework.data.couchbase.config.AbstractCouch
baseDataConfiguration` 实现.

5.12.8. LDAP

LDAP (Lightweight Directory Access Protocol, 轻量级目录访问协议) 是一个开放、厂商中立的行业标准应用协议, 其通过 IP 网络访问和维护分布式目录信息服务. Spring Boot 为兼容 LDAP 服务器提供了自动配置, 以及支持从 `UnboundID` 内嵌内存式 LDAP 服务器.

Spring Data LDAP 提供了 LDAP 抽象. 相关依赖包含在了 `spring-boot-starter-data-ldap` starter 中.

连接 LDAP 服务器

要连接 LDAP 服务器, 请确保您已经声明了 `spring-boot-starter-data-ldap` starter 或者 `spring-ldap-core` 依赖, 然后在 `application.properties` 声明服务器的 URL:

```

spring:
  ldap:
    urls: "ldap://myserver:1235"
    username: "admin"
    password: "secret"

```

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

`environment` 属性.

如果 `DirContextAuthenticationStrategy` bean 可用, 则它与自动配置的 `LdapContextSource` 相关联. `LdapContextSource` 将根据这些设置自动配置. 如果您需要自定义它, 例如使用一个 `PooledContextSource`, 则仍然可以注入自动配置的 `LdapContextSource`. 确保将自定义的 `ContextSource` 标记为 `@Primary`, 以便自动配置的 `LdapTemplate` 能使用它.

Spring Data LDAP 资源库

Spring Data 包含了 LDAP 资源库支持. 有关 Spring Data LDAP 的完整详细信息, 请参阅其 [参考文档](#).

您还可以像其他 Spring Bean 一样注入一个自动配置的 `LdapTemplate` 实例:

```
@Component
public class MyBean {

    private final LdapTemplate template;

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

    // ...
}
```

内嵌内存式 LDAP 服务器

为了测试目的, Spring Boot 支持从 `UnboundID` 自动配置一个内存式 LDAP 服务器. 要配置服务器, 请添加 `com.unboundid:unboundid-ldapsdk` 依赖并声明一个 `spring.ldap.embedded.base-dn` 属性:

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

可以定义多个 `base-dn` 值,但是,由于名称包含逗号,存在歧义,因此必须使用正确的符号来定义它们.



在 `yaml` 文件中,您可以使用 `yaml` 列表表示法,在属性文件中,必须使用索引方式:

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

默认情况下,服务器将在一个随机端口上启动,并触发常规的 LDAP 支持(不需要指定 `spring.ldap.urls` 属性).

如果您的 `classpath` 上存在一个 `schema.ldif` 文件,其将用于初始化服务器.

如果您想从不同的资源中加载脚本,可以使用 `spring.ldap.embedded.ldif` 属性.

默认情况下,将使用一个标准模式(`schema`)来校验 `LDIF` 文件. 您可以使用 `spring.ldap.embedded.validation.enabled` 属性来关闭所有校验.

如果您有自定义的属性,则可以使用 `spring.ldap.embedded.validation.schema` 来定义自定义属性类型或者对象类.

5.12.9. InfluxDB

[InfluxDB](#) 是一个开源时列数据库

,其针对运营监控、应用程序指标、物联网传感器数据和实时分析等领域中的时间序列数据在速度、高可用存储和检索方面进行了优化.

连接 [InfluxDB](#)

Spring Boot 自动配置 [InfluxDB](#) 实例,前提是 `Influxdb-java` 客户端在 `classpath` 上并且设置了数据库的 URL,如下所示:

```
spring:  
  influx:  
    url: "https://172.0.0.1:8086"
```

如果与 [InfluxDB](#) 的连接需要用户和密码,则可以相应地设置 `spring.influx.user` 和

`spring.influx.password` 属性.

InfluxDB 依赖于 OkHttp. 如果你需要调整 InfluxDB 在底层使用的 http 客户端, 则可以注册一个 `InfluxDbOkHttpClientBuilderProvider` bean.

5.13. 缓存

Spring Framework 支持以透明的方式向应用程序添加缓存. 从本质上讲, 将缓存应用于方法上, 根据缓存数据减少方法的执行次数. 缓存逻辑是透明的, 不会对调用者造成任何干扰. 通过 `@EnableCaching` 注解启用缓存支持, Spring Boot 就会自动配置缓存设置.



有关更多详细信息, 请查看 Spring Framework 参考文档的 [相关部分](#).

简而言之, 为服务添加缓存的操作就像在其方法中添加注解一样简单, 如下所示:

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class MathService {

    @Cacheable("piDecimals")
    public int computePiDecimal(int i) {
        // ...
    }

}
```

此示例展示了如何在代价可能高昂的操作上使用缓存. 在调用 `computePiDecimal` 之前, 缓存支持会在 `piDecimals` 缓存中查找与 `i` 参数匹配的项. 如果找到, 则缓存中的内容会立即返回给调用者, 并不会调用该方法. 否则, 将调用该方法, 并在返回值之前更新缓存.



您还可以使用标准 JSR-107 (JCache) 注解 (例如 `@CacheResult`). 但是, 我们强烈建议您不要将 Spring Cache 和 JCache 注解混合使用.

如果您不添加任何指定的缓存库, Spring Boot 会自动配置一个使用并发 map 的 simple provider . 当需要缓存时 (例如前面示例中的 `piDecimals`) ,该 simple provider 会为您创建缓存. 不推荐将 simple provider 用于生产环境 ,但它非常适合入门并帮助您了解这些功能. 当您决定使用缓存提供者时 ,请务必阅读其文档以了解如何配置应用程序.

几乎所有提供者都要求您显式配置应用程序中使用的每个缓存. 有些提供了自定义 `spring.cache.cache-names` 属性以定义默认缓存.



还可以透明地从缓存中 [更新](#)或 [删除](#) 数据.

5.13.1. 支持的缓存提供者

缓存抽象不提供存储实现,其依赖于 `org.springframework.cache.Cache` 和 `org.springframework.cache.CacheManager` 接口实现的抽象.

如果您未定义 `CacheManager` 类型的 bean 或名为 `cacheResolver` 的 `CacheResolver` (请参阅 `CachingConfigurer`) ,则 Spring Boot 会尝试检测以下提供者 (按序号顺序) :

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



也可以通过设置 `spring.cache.type` 属性来强制指定缓存提供者. 如果您需要在某些环境 (比如测试) 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. `starter` 引入了 `spring-context-support`. 如果手动添加依赖, 则必须包含 `spring-context-support` 才能使用 JCache、EhCache 2.x 或 Guava 支持.

如果通过 Spring Boot 自动配置 `CacheManager`, 则可以通过暴露一个实现了 `CacheManagerCustomizer` 接口的 bean, 在完全初始化之前进一步调整其配置.

以下示例设置了一个 `flag`, 表示应将 `null` 值传递给底层 map:

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager>
cacheManagerCustomizer() {
    return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {

        @Override
        public void customize(ConcurrentMapCacheManager cacheManager) {
            cacheManager.setAllowNullValues(false);
        }

    };
}
```



在前面示例中, 需要一个自动配置的 `ConcurrentMapCacheManager`.

如果不是这种情况 (您提供了自己的配置或自动配置了不同的缓存提供者), 则根本不会调用 `customizer`. 您可以拥有多个 `customizer`, 也可以使用 `@Order` 或 `Ordered` 来排序它们.

Generic

如果上下文定义了至少一个 `org.springframework.cache.Cache` bean, 则使用 Generic 缓存. 将创建一个包装所有该类型 bean 的 `CacheManager`.

JCache (JSR-107)

`JCache` 通过 classpath 上的 `javax.cache.spi.CachingProvider` (即 classpath 上存在符合 JSR-107 的缓存库) 来引导, `jCacheCacheManager` 由 `spring-boot-starter-cache` starter 提供. 您可以使用各种兼容库, Spring Boot 为 Ehcache 3、Hazelcast 和 Infinispan 提供依赖管理. 您还可以添加任何其他兼容库.

可能存在多个提供者,在这种情况下必须明确指定提供者. 即使 JSR-107 标准没有强制规定一个定义配置文件位置的标准化方法, Spring Boot 也会尽其所能设置一个包含实现细节的缓存,如下所示:

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



当缓存库同时提供原生实现和 JSR-107 支持时, Spring Boot 更倾向 JSR-107 支持,因此当您切换到不同的 JSR-107 实现时,还可以使用相同的功能.



Spring Boot 对 Hazelcast 的支持一般. 如果有一个 HazelcastInstance 可用,它也会自动为 CacheManager 复用,除非指定了 spring.cache.jcache.config 属性.

有两种方法可以自定义底层的 javax.cache.CacheManager:

- 可以通过设置 spring.cache.cache-names 属性在启动时创建缓存. 如果定义了自定义 javax.cache.configuration.Configuration bean,则会使用它来自定义.
- 使用 CacheManager 的引用调用 org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer bean 以进行完全自定义.



如果定义了一个标准的 javax.cache.CacheManager bean,它将自动包装进一个抽象所需的 org.springframework.cache.CacheManager 实现中,而不会应用自定义配置.

EhCache 2.x

如果可以在 `classpath` 的根目录中找到名为 `ehcache.xml` 的文件, 则使用 [EhCache 2.x](#). 如果找到 `EhCache 2.x`, 则使用 `spring-boot-starter-cache` starter 提供的 `EhCacheCacheManager` 来启动缓存管理器. 还可以提供其他配置文件, 如下所示:

```
spring:
  cache:
    ehcache:
      config: "classpath:config/another-config.xml"
```

Hazelcast

`Spring Boot` 对 [Hazelcast](#) 的支持一般. 如果自动配置了一个 `HazelcastInstance`, 它将自动包装进 `CacheManager` 中.

Infinispan

[Infinispan](#) 没有默认的配置文件位置, 因此必须明确指定. 否则将使用默认配置引导.

```
spring:
  cache:
    infinispan:
      config: "infinispan.xml"
```

可以通过设置 `spring.cache.cache-names` 属性在启动时创建缓存. 如果定义了自定义 `ConfigurationBuilder` bean, 则它将用于自定义缓存.



`Infinispan` 在 `Spring Boot` 中的支持仅限于内嵌模式, 非常简单. 如果你想要更多选项, 你应该使用官方的 `Infinispan Spring Boot` starter. 有关更多详细信息, 请参阅 [Infinispan 文档](#).

Couchbase

如果 `Spring Data Couchbase` 可用并且已 [配置](#) `Couchbase`, 则会自动配置 `CouchbaseCacheManager`. 通过设置 `spring.cache.cache-names` 属性可以在启动时创建其他缓存, 并且可以使用 `spring.cache.couchbase.*` 属性配置缓存默认值. 以下配置创建 `cache1` 和 `cache2` 缓存, 他们的有效时间为 10 分钟:

```
spring:
  cache:
    cache-names: "cache1,cache2"
    couchbase:
      expiration: "10m"
```

如果需要对配置进行更多控制,请考虑注册 `CouchbaseCacheManagerBuilderCustomizer` bean.以下示例显示了一个定制器,该定制器为 `cache1` 和 `cache2` 配置到期:

```
@Bean
public CouchbaseCacheManagerBuilderCustomizer
myCouchbaseCacheManagerBuilderCustomizer() {
    return (builder) -> builder
        .withCacheConfiguration("cache1",
            CouchbaseCacheConfiguration.defaultCacheConfig().entryExpiry(Duration.ofSeconds(
                10)))
        .withCacheConfiguration("cache2",
            CouchbaseCacheConfiguration.defaultCacheConfig().entryExpiry(Duration.ofMinutes(
                1)));
}
```

Redis

如果 `Redis` 可用并已经配置,则应用程序会自动配置一个 `RedisCacheManager`. 通过设置 `spring.cache.cache-names` 属性可以在启动时创建其他缓存,并且可以使用 `spring.cache.redis.*` 属性配置缓存默认值. 例如,以下配置创建 `cache1` 和 `cache2` 缓存,他们的有效时间为 10 分钟:

```
spring:
  cache:
    cache-names: "cache1,cache2"
    redis:
      time-to-live: "10m"
```



默认情况下,会添加一个 `key` 前缀
 ,这样做是因为如果两个单独的缓存使用了相同的键,Redis 不支持重叠
`key`,而缓存也不能返回无效值. 如果您创建自己的
`RedisCacheManager`,我们强烈建议您启用此设置.



您可以通过添加自己的 `RedisCacheConfiguration @Bean`
 来完全控制配置. 如果您想自定义序列化策略,这种方式可能很有用.

如果您需要控制更多的配置,请考虑注册 `RedisCacheManagerBuilderCustomizer` bean.
 以下示例显示了一个自定义的配置,该定制器配置了 `cache1` 和 `cache2` 的特定生存时间 The
 following example shows a customizer that configures a specific time to
 live for `cache1` and `cache2`:

```
@Bean
public RedisCacheManagerBuilderCustomizer myRedisCacheManagerBuilderCustomizer()
{
    return (builder) -> builder
        .withCacheConfiguration("cache1",
            RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofSeconds(10)))
        .withCacheConfiguration("cache2",
            RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofMinutes(1)));
}
```

Caffeine

`Caffeine` 是一个使用了 Java 8 重写 Guava 缓存,用于取代 Guava 支持的缓存库. 如果 `Caffeine` 存在,则应用程序会自动配置一个 `CaffeineCacheManager` (由 `spring-boot-starter-cache` starter 提供). 可以通过设置 `spring.cache.cache-names` 属性在启动时创建缓存,并且可以通过以下方式之一 (按序号顺序) 自定义缓存:

1. 一个由 `spring.cache.caffeine.spec` 定义的缓存规范
2. 一个已定义的 `com.github.benmanes.caffeine.cache.CaffeineSpec` bean
3. 一个已定义的 `com.github.benmanes.caffeine.cache.Caffeine` bean

例如,以下配置创建 `cache1` 和 `cache2` 缓存,最大大小为 500,有效时间为 10 分钟:

```
spring:
  cache:
    cache-names: "cache1,cache2"
    caffeine:
      spec: "maximumSize=500,expireAfterAccess=600s"
```

如果定义了 `com.github.benmanes.caffeine.cache.CacheLoader` bean,它将自动与 `CaffeineCacheManager` 关联. 由于 `CacheLoader` 将与缓存管理器管理的所有缓存相关联,因此必须将其定义为 `CacheLoader<Object, Object>`. 自动配置会忽略所有其他泛型类型.

Simple

如果找不到其他提供者,则配置使用一个 `ConcurrentHashMap` 作为缓存存储的简单实现. 如果您的应用程序中没有缓存库,则该项为默认值. 默认情况下,会根据需要创建缓存,但您可以通过设置 `cache-names` 属性来限制可用缓存的列表. 例如,如果只需要 `cache1` 和 `cache2` 缓存,请按如下设置 `cache-names` 属性:

```
spring:
  cache:
    cache-names: "cache1,cache2"
```

如果这样做了,并且您的应用程序使用了未列出的缓存,则运行时在它需要缓存时会触发失败,但在启动时则不会. 这类似于真实缓存提供者在使用未声明的缓存时触发的行为方式.

None

当配置中存在 `@EnableCaching` 时,也需要合适的缓存配置.

如果需要在某些环境中完全禁用缓存,请将缓存类型强制设置为 `none` 以使用 no-op 实现,如下所示:

```
spring:
  cache:
    type: "none"
```

5.14. 消息传递

Spring Framework 为消息传递系统集成提供了广泛的支持,从使用 [JmsTemplate](#) 简化 JMS API 的使用到异步接收消息的完整基础设施. Spring AMQP 为高级消息队列协议 (Advanced Message Queuing Protocol,AMQP) 提供了类似的功能集合. Spring Boot 还为 [RabbitTemplate](#) 和 RabbitMQ 提供自动配置选项. Spring WebSocket 本身包含了对 STOMP 消息传递的支持, Spring Boot 通过 `starter` 和少量自动配置即可支持它. Spring Boot 同样支持 Apache Kafka.

5.14.1. JMS

`javax.jms.ConnectionFactory` 接口提供了一种创建 `javax.jms.Connection` 的标准方法, 可与 JMS broker (代理) 进行交互. 虽然 Spring 需要一个 `ConnectionFactory` 来与 JMS 一同工作, 但是您通常不需要自己直接使用它, 而是可以依赖更高级别的消息传递抽象. (有关详细信息, 请参阅 [Spring Framework 参考文档的相关部分](#).) Spring Boot 还会自动配置发送和接收消息所需的基础设施.

`ActiveMQ` 支持

当 `ActiveMQ` 在 `classpath` 上可用时, Spring Boot 也可以配置一个 `ConnectionFactory`. 如果 `broker` 存在, 则会自动启动并配置一个内嵌式 `broker` (前提是未通过配置指定 `broder URL`) .



如果使用 `spring-boot-starter-activemq`, 则提供了连接到 ActiveMQ 实例必须依赖或内嵌一个 ActiveMQ 实例, 以及与 JMS 集成的 Spring 基础设施.

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

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

默认情况下, `CachingConnectionFactory` 将原生的 `ConnectionFactory` 使用可由

`spring.jms.*` 中的外部配置属性控制的合理设置包装起来：

```
spring:
  jms:
    cache:
      session-cache-size: 5
```

如果您更愿意使用原生池，则可以通过向 `org.messaginghub:pooled-jms` 添加一个依赖并相应地配置 `JmsPoolConnectionFactory` 来实现，如下所示：

```
spring:
  activemq:
    pool:
      enabled: true
      max-connections: 50
```



有关更多支持的选项，请参阅 [ActiveMQProperties](#).

您还可以注册多个实现了 `ActiveMQConnectionFactoryCustomizer` 的的 bean，以进行更高级的自定义。

默认情况下，`ActiveMQ` 会创建一个 `destination`（目标）（如果它尚不存在），以便根据提供的名称解析 `destination`.

ActiveMQ Artemis 支持

`Spring Boot` 可以在检测到 `Artemis` 在 `classpath` 上可用时自动配置一个 `ConnectionFactory`. 如果存在 `broker`, 则会自动启动并配置一个内嵌 `broker` (除非已明确设置 `mode` 属性) . 支持的 `mode` 为 `embedded` (明确表示需要一个内嵌 `broker`, 如果 `broker` 在 `classpath` 上不可用则发生错误) 和 `native` (使用 `netty` 传输协议连接到 `broker`) . 配置后者后, `Spring Boot` 会使用默认设置配置一个 `ConnectionFactory`, 该 `ConnectionFactory` 连接到在本地计算机上运行的 `broker`.



如果使用了 `spring-boot-starter-artemis`, 则会提供连接到现有的 `Artemis` 实例的必须依赖, 以及与 `JMS` 集成的 `Spring` 基础设施. 将 `org.apache.activemq:artemis-jms-server` 添加到您的应用程序可让您使用内嵌模式.

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

```
spring:  
  artemis:  
    mode: native  
    host: "192.168.1.210"  
    port: 9876  
    user: "admin"  
    password: "secret"
```

内嵌 broker 时，您可以选择是否要启用持久化并列出应该可用的 `destination`。可以将这些指定为以逗号分隔的列表，以使用默认选项创建它们，也可以定义类型为 `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` 或 `org.apache.activemq.artemis.jms.server.config.TopicConfiguration` 的 bean，分别用于高级队列和 topic（主题）配置。

默认情况下，`CachingConnectionFactory` 将原生的 `ConnectionFactory` 使用可由 `spring.jms.*` 中的外部配置属性控制的合理设置包装起来：

```
spring:  
  jms:  
    cache:  
      session-cache-size: 5
```

如果您更愿意使用原生池，则可以通过向 `org.messaginghub:pooled-jms` 添加一个依赖并相应地配置 `JmsPoolConnectionFactory` 来实现，如下所示：

```
spring:  
  artemis:  
    pool:  
      enabled: true  
      max-connections: 50
```

有关更多支持的选项，请参阅 [ArtemisProperties](#)。

不涉及 JNDI 查找，使用 `Artemis` 配置中的 `name` 属性或通过配置提供的名称来解析目标（destination）名称。

使用 JNDI ConnectionFactory

如果您在应用程序服务器中运行应用程序, Spring Boot 会尝试使用 JNDI 找到 JMS ConnectionFactory. 默认情况下, 将检查 `java:/JmsXA` 和 `java:/XAConnectionFactory` 这两个位置. 如果需要指定其他位置, 可以使用 `spring.jms.jndi-name` 属性, 如下所示:

```
spring:  
  jms:  
    jndi-name: "java:/MyConnectionFactory"
```

发送消息

Spring 的 `JmsTemplate` 是自动配置的, 你可以直接将它注入到你自己的 bean 中, 如下所示:

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



`JmsMessagingTemplate` 可以以类似的方式注入. 如果定义了 `DestinationResolver` 或 `MessageConverter` bean, 它将自动关联到自动配置的 `JmsTemplate`.

接收消息

当存在 JMS 基础设施时,可以使用 `@JmsListener` 对任何 bean 进行注解以创建监听器 (listener) 端点. 如果未定义 `JmsListenerContainerFactory`, 则会自动配置一个默认的 (factory). 如果定义了 `DestinationResolver` 或 `MessageConverter` bean, 它将自动关联到默认的 factory.

默认情况下,默认 factory 是具有事务特性的. 如果您在存在有 `JtaTransactionManager` 的基础设施中运行,则默认情况下它与监听器容器相关联. 如果不是,则 `sessionTransacted flag` 将为启用 (`enabled`). 在后一种情况下,您可以通过在监听器方法 (或其委托) 上添加 `@Transactional`, 将本地数据存储事务与传入消息的处理相关联.

这确保了在本地事务完成后传入消息能被告知. 这还包括了发送已在同一 JMS 会话上执行的响应消息.

以下组件在 `someQueue` destination 上创建一个监听器端点:

```
@Component
public class MyBean {

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



有关更多详细信息,请参阅 [the Javadoc of `@EnableJms`](#) 的 Javadoc.

如果需要创建更多 `JmsListenerContainerFactory` 实例或覆盖默认值, Spring Boot 会提供一个 `DefaultJmsListenerContainerFactoryConfigurer`, 您可以使用它来初始化 `DefaultJmsListenerContainerFactory`, 其设置与自动配置的 factory 设置相同.

例如,以下示例暴露了另一个使用特定 `MessageConverter` 的 factory:

```

@Configuration(proxyBeanMethods = false)
static class JmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory myFactory(
        DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory =
            new DefaultJmsListenerContainerFactory();
        configurer.configure(factory, connectionFactory());
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }

}

```

然后,您可以在任何 `@JmsListener` 注解的方法中使用该 `factory`,如下所示:

```

@Component
public class MyBean {

    @JmsListener(destination = "someQueue", containerFactory="myFactory")
    public void processMessage(String content) {
        // ...
    }

}

```

5.14.2. AMQP

高级消息队列协议 (Advanced Message Queuing Protocol,AMQP) 是一个平台无关,面向消息中间件的连接级协议. Spring AMQP 项目将核心 Spring 概念应用于基于 AMQP 消息传递解决方案的开发. Spring Boot 为通过 RabbitMQ 使用 AMQP 提供了一些快捷方法,包括 `spring-boot-starter-amqp` starter.

RabbitMQ 支持

[RabbitMQ](#) 是一个基于 AMQP 协议的轻量级、可靠、可扩展且可移植的消息代理. Spring 使用 RabbitMQ 通过 AMQP 协议进行通信.

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

```
spring:  
  rabbitmq:  
    host: "localhost"  
    port: 5672  
    username: "admin"  
    password: "secret"
```

另外,您可以配置相同 `addresses` 属性的连接:

```
spring:  
  rabbitmq:  
    addresses: "amqp://admin:secret@localhost"
```



当以这种方式指定 `addresses` 时, `host` 和 `port` 属性将被忽略.

如果地址使用 `amqps` 协议,则会自动启用 SSL 支持

如果上下文中存在 `ConnectionNameStrategy` bean, 它将自动用于命名由自动配置的 `ConnectionFactory` 所创建的连接. 有关更多支持的选项,请参阅 [RabbitProperties](#) .



有关详细信息,请参阅理解 [AMQP、RabbitMQ 使用的协议](#) f.

发送消息

Spring 的 `AmqpTemplate` 和 `AmqpAdmin` 是自动配置的, 您可以将它们直接注入自己的 bean 中,如下所示:

```

import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;
    private final AmqpTemplate amqpTemplate;

    @Autowired
    public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
        this.amqpAdmin = amqpAdmin;
        this.amqpTemplate = amqpTemplate;
    }

    // ...
}

```



`RabbitMessagingTemplate` 可以以类似的方式注入。如果定义了 `MessageConverter` bean, 它将自动关联到自动配置的 `AmqpTemplate`.

如有必要,所有定义为 bean 的 `org.springframework.amqp.core.Queue` 都会自动在 RabbitMQ 实例上声明相应的队列。

要重试操作,可以在 `AmqpTemplate` 上启用重试(例如,在 broker 连接丢失的情况下) :

```

spring:
rabbitmq:
template:
retry:
    enabled: true
    initial-interval: "2s"

```

默认情况下禁用重试。您还可以通过声明 `RabbitRetryTemplateCustomizer` bean 以编程方式自定义 `RetryTemplate`.

如果您需要创建更多的 `RabbitTemplate` 实例,或者想覆盖默认实例, Spring Boot

提供了一个 `RabbitTemplateConfigurer` bean, 您可以使用它来初始化一个 `RabbitTemplate`, 其设置与自动配置所使用的工厂相同.

接收消息

当 Rabbit 基础设施存在时, 可以使用 `@RabbitListener` 注解任何 bean 以创建监听器端点. 如果未定义 `RabbitListenerContainerFactory`, 则会自动配置一个默认的 `SimpleRabbitListenerContainerFactory`, 您可以使用 `spring.rabbitmq.listener.type` 属性切换到一个直接容器. 如果定义了 `MessageConverter` 或 `MessageRecoverer` bean, 它将自动与默认 factory 关联.

以下示例组件在 `someQueue` 队列上创建一个监听器端点:

```
@Component
public class MyBean {

    @RabbitListener(queues = "someQueue")
    public void processMessage(String content) {
        // ...
    }
}
```



有关更多详细信息, 请参阅 [the Javadoc of `@EnableRabbit`](#) 的 [Javadoc](#).

如果需要创建更多 `RabbitListenerContainerFactory` 实例或覆盖默认值, Spring Boot 提供了一个 `SimpleRabbitListenerContainerFactoryConfigurer` 和一个 `DirectRabbitListenerContainerFactoryConfigurer`, 您可以使用它来初始化 `SimpleRabbitListenerContainerFactory` 和 `DirectRabbitListenerContainerFactory`, 其设置与使用自动配置的 factory 相同.



这两个 bean 与您选择的容器类型没有关系, 它们通过自动配置暴露.

例如, 以下配置类暴露了另一个使用特定 `MessageConverter` 的 factory:

```

@Configuration(proxyBeanMethods = false)
static class RabbitConfiguration {

    @Bean
    public SimpleRabbitListenerContainerFactory myFactory(
        SimpleRabbitListenerContainerFactoryConfigurer configurer) {
        SimpleRabbitListenerContainerFactory factory =
            new SimpleRabbitListenerContainerFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }

}

```

然后，您可以在任何 `@RabbitListener` 注解的方法中使用该 `factory`，如下所示：

```

@Component
public class MyBean {

    @RabbitListener(queues = "someQueue", containerFactory="myFactory")
    public void processMessage(String content) {
        // ...
    }

}

```

您可以启用重试机制来处理监听器的异常抛出情况。默认情况下使用 `RejectAndDontRequeueRecoverer`，但您可以定义自己的 `MessageRecoverer`。如果 `broker` 配置了重试机制，当重试次数耗尽时，则拒绝消息并将其丢弃或路由到死信（dead-letter）exchange 中。默认情况下重试机制为禁用。您还可以通过声明 `RabbitRetryTemplateCustomizer` bean 以编程方式自定义 `RetryTemplate`。



默认情况下，如果禁用重试并且监听器异常抛出，则会无限期地重试传递。您可以通过两种方式修改此行为：将 `defaultRequeueRejected` 属性设置为 `false`，以便尝试零重传或抛出 `AmqpRejectAndDontRequeueException` 以指示拒绝该消息。后者是启用重试并且达到最大传递尝试次数时使用的机制。

5.14.3. Apache Kafka 支持

通过提供 `spring-kafka` 项目的自动配置来支持 Apache Kafka

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

```
spring:  
  kafka:  
    bootstrap-servers: "localhost:9092"  
    consumer:  
      group-id: "myGroup"
```



要在启动时创建主题 (topic)，请添加 `NewTopic` 类型的 Bean。
如果主题已存在，则忽略该 bean。

有关更多支持的选项，请参阅 [KafkaProperties](#)。

发送消息

Spring 的 `KafkaTemplate` 是自动配置的，您可以直接在自己的 bean 中装配它，如下所示：

```
@Component  
public class MyBean {  
  
    private final KafkaTemplate kafkaTemplate;  
  
    @Autowired  
    public MyBean(KafkaTemplate kafkaTemplate) {  
        this.kafkaTemplate = kafkaTemplate;  
    }  
  
    // ...  
}
```



如果定义了属性 `spring.kafka.producer.transaction-id-prefix`, 则会自动配置一个 `KafkaTransactionManager`. 此外, 如果定义了 `RecordMessageConverter` bean, 它将自动关联到自动配置的 `KafkaTemplate`.

接收消息

当存在 Apache Kafka 基础设施时, 可以使用 `@KafkaListener` 注解任何 bean 以创监听器端点. 如果未定义 `KafkaListenerContainerFactory`, 则会使用 `spring.kafka.listener.*` 中定义的 key 自动配置一个默认的 factory.

以下组件在 `someTopic` topic 上创建一个监听器端点:

```
@Component
public class MyBean {

    @KafkaListener(topics = "someTopic")
    public void processMessage(String content) {
        // ...
    }
}
```

如果定义了 `KafkaTransactionManager` bean, 它将自动关联到容器 factory. 同样, 如果定义了 `RecordFilterStrategy`, `ErrorHandler`, `AfterRollbackProcessor` 或 `ConsumerAwareRebalanceListener` bean, 它将自动关联到默认的 factory.

根据监听器类型, 将 `RecordMessageConverter` 或 `BatchMessageConverter` bean 与默认工厂关联. 如果对于批处理监听器仅存在一个 `RecordMessageConverter` bean, 则将其包装在 `BatchMessageConverter` 中.



自定义 `ChainedKafkaTransactionManager` 必须标记为 `@Primary`, 因为它通常引用自动配置的 `KafkaTransactionManager` bean.

Kafka Streams

Spring for Apache Kafka 提供了一个工厂 bean 来创建 `StreamsBuilder` 对象并管理其 `stream` (流) 的生命周期. 只要 `kafka-streams` 在 classpath

上并且通过 `@EnableKafkaStreams` 注解启用了 Kafka Stream, Spring Boot 就会自动配置所需的 `KafkaStreamsConfiguration` bean.

启用 Kafka Stream 意味着必须设置应用程序 id 和引导服务器 (bootstrap server) . 可以使用 `spring.kafka.streams.application-id` 配置前者, 如果未设置则默认为 `spring.application.name`. 后者可以全局设置或专门为 stream 而重写.

使用专用 `properties` 可以设置多个其他属性, 可以使用 `spring.kafka.streams.properties` 命名空间设置其他任意 Kafka 属性. 有关更多信息, 另请参见 [Kafka 属性](#) .

要使用 factory bean, 只需将 `StreamsBuilder` 装配到您的 `@Bean` 中, 如下所示:

```
@Configuration(proxyBeanMethods = false)
@EnableKafkaStreams
public static class KafkaStreamsExampleConfiguration {

    @Bean
    public KStream<Integer, String> kStream(StreamsBuilder streamsBuilder) {
        KStream<Integer, String> stream = streamsBuilder.stream("ks1In");
        stream.map((k, v) -> new KeyValue<>(k, v.toUpperCase())).to("ks10ut",
            Produced.with(Serdes.Integer(), new JsonSerde<>()));
        return stream;
    }

}
```

默认情况下, 由其创建的 `StreamBuilder` 对象管理的流会自动启动. 您可以使用 `spring.kafka.streams.auto-startup` 属性自定义此行为.

其他 Kafka 属性

自动配置支持的属性可在[常见应用程序属性](#)常见应用程序属性中找到. 请注意, 在大多数情况下, 这些属性 (连接符或驼峰命名) 直接映射到 Apache Kafka 点连形式属性. 有关详细信息, 请参阅 [Apache Kafka 文档](#).

这些属性中的前几个适用于所有组件 (生产者 [producer] 、使用者 [consumer] 、管理者 [admin] 和流 [stream]) , 但如果您希望使用不同的值, 则可以在组件级别指定. Apache Kafka 重要性 (优先级) 属性设定为 HIGH、MEDIUM 或 LOW. Spring Boot 自动配置支持所有 HIGH 重要性属性, 一些选择的 MEDIUM 和 LOW 属性

, 以及所有没有默认值的属性。

只有 Kafka 支持的属性的子集可以直接通过 **KafkaProperties** 类获得。

如果您希望使用不受支持的其他属性配置生产者或消费者, 请使用以下属性:

```
spring:  
  kafka:  
    properties:  
      "[prop.one]": "first"  
    admin:  
      properties:  
        "[prop.two)": "second"  
    consumer:  
      properties:  
        "[prop.three]": "third"  
    producer:  
      properties:  
        "[prop.four]": "fourth"  
    streams:  
      properties:  
        "[prop.five]": "fifth"
```

这将常见的 **prop.one** Kafka 属性设置为 **first** (适用于生产者、消费者和管理者)

, **prop.two** 管理者属性为 **second**, **prop.three** 消费者属性为 **third**, **prop.four**

生产者属性为 **fourth**, **prop.five** 流属性为 **fifth**.

您还可以按如下方式配置 Spring Kafka **JsonDeserializer**:

```
spring:  
  kafka:  
    consumer:  
      value-deserializer:  
        "org.springframework.kafka.support.serializer.JsonDeserializer"  
      properties:  
        "[spring.json.value.default.type)": "com.example.Invoice"  
        "[spring.json.trusted.packages)": "com.example,org.acme"
```

同样, 您可以禁用 **JsonSerializer** 在 **header** 中发送类型信息的默认行为:

```

spring:
  kafka:
    producer:
      value-serializer:
        "org.springframework.kafka.support.serializer.JsonSerializer"
        properties:
          "[spring.json.add.type.headers)": false

```



以这种方式设置的属性将覆盖 Spring Boot 明确支持的任何配置项.

使用嵌入式 Kafka 进行测试

Spring 为 Apache Kafka 提供了一种使用嵌入式 Apache Kafka 代理测试项目的便捷方法. 要使用此功能, 请在 `spring-kafka-test` 模块中使用 `@EmbeddedKafka` 注解测试类. 有关更多信息, 请参阅 [Spring for Apache Kafka 参考手册](#).

要使 Spring Boot 自动配置与上述嵌入式 Apache Kafka 代理一起使用, 您需要将嵌入式代理地址 (由 `EmbeddedKafkaBroker` 填充) 的系统属性重新映射到 Apache Kafka 的 Spring Boot 配置属性中. 有几种方法可以做到这一点:

- 提供一个系统属性, 以将嵌入式代理地址映射到测试类中的 `spring.kafka.bootstrap-servers` 中:

```

static {
  System.setProperty(EmbeddedKafkaBroker.BROKER_LIST_PROPERTY,
    "spring.kafka.bootstrap-servers");
}

```

- 在 `@EmbeddedKafka` 注解上配置属性名称:

```

@EmbeddedKafka(topics = "someTopic",
  bootstrapServersProperty = "spring.kafka.bootstrap-servers")

```

- 在配置属性中使用占位符:

```
spring:
  kafka:
    bootstrap-servers: "${spring.embedded.kafka.brokers}"
```

5.15. 使用 RestTemplate 调用 REST 服务

如果您的应用程序需要调用远程 REST 服务,这可以使用 Spring Framework 的 `RestTemplate` 类. 由于 `RestTemplate` 实例在使用之前通常需要进行自定义,因此 Spring Boot 不提供任何自动配置的 `RestTemplate` bean. 但是, 它会自动配置 `RestTemplateBuilder`,可在需要时创建 `RestTemplate` 实例. 自动配置的 `RestTemplateBuilder` 确保将合适的 `HttpMessageConverters` 应用于 `RestTemplate` 实例.

以下代码展示了一个典型示例:

```
@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate = restTemplateBuilder.build();
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details", Details.class,
name);
    }

}
```



`RestTemplateBuilder` 包含许多可用于快速配置 `RestTemplate` 的方法. 例如,要添加 BASIC auth 支持,可以使用 `builder.basicAuthentication("user", "password").build()`.

5.15.1. 自定义 RestTemplate

RestTemplate 自定义有三种主要方法, 具体取决于您希望自定义的程度.

要想自定义的作用域尽可能地窄, 请注入自动配置的 **RestTemplateBuilder**, 然后根据需要调用其方法. 每个方法调用都返回一个新的 **RestTemplateBuilder** 实例, 因此自定义只会影响当前构建器.

要在应用程序作用域内添加自定义配置, 请使用 **RestTemplateCustomizer** bean. 所有这些 bean 都会自动注册到自动配置的 **RestTemplateBuilder**, 并应用于使用它构建的所有模板.

以下示例展示了一个 **customizer**, 它为除 **192.168.0.5** 之外的所有主机配置代理:

```
static class ProxyCustomizer implements RestTemplateCustomizer {

    @Override
    public void customize(RestTemplate restTemplate) {
        HttpHost proxy = new HttpHost("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create().setRoutePlanner(new
DefaultProxyRoutePlanner(proxy) {

            @Override
            public HttpHost determineProxy(HttpHost target, HttpRequest request,
HttpContext context)
                throws HttpException {
                if (target.getHostName().equals("192.168.0.5")) {
                    return null;
                }
                return super.determineProxy(target, request, context);
            }

        }).build();
        restTemplate.setRequestFactory(new
HttpComponentsClientHttpRequestFactory(httpClient));
    }

}
```

最后, 您还可以创建自己的 **RestTemplateBuilder** bean。为了防止关闭 **RestTemplateBuilder** 的自动配置, 并防止任何 **RestTemplateCustomizer** bean 被使用, 请确保使用 **RestTemplateBuilderConfigurer** 配置您的自定义实例。下面的示例公开了一个 **RestTemplateBuilder**, Spring Boot

将自动配置它，但也指定了自定义连接和读取超时：

```
@Bean
public RestTemplateBuilder restTemplateBuilder(RestTemplateBuilderConfigurer
configurer) {
    return configurer.configure(new
RestTemplateBuilder()).setConnectTimeout(Duration.ofSeconds(5))
        .setReadTimeout(Duration.ofSeconds(2));
}
```

最极端（也很少使用）的选择是创建自己的 `RestTemplateBuilder` bean。这样做会关闭 `RestTemplateBuilder` 的自动配置，并阻止使用任何 `RestTemplateCustomizer` bean。

5.16. 使用 `WebClient` 调用 REST 服务

如果在 `classpath` 上存在 Spring WebFlux，则还可以选择使用 `WebClient` 来调用远程 REST 服务。与 `RestTemplate` 相比，该客户端更具函数式风格并且完全响应式。您可以在 [Spring Framework 文档的相关部分](#) 中了解有关 `WebClient` 的更多信息。

Spring Boot 为您创建并预配置了一个 `WebClient.Builder`。

强烈建议将其注入您的组件中并使用它来创建 `WebClient` 实例。Spring Boot 配置该构建器以共享 HTTP 资源，以与服务器相同的方式反射编解码器设置（请参阅 [WebFlux HTTP 编解码器自动配置](#)）等。

以下代码是一个典型示例：

```

@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient.Builder webClientBuilder) {
        this.webClient =
            webClientBuilder.baseUrl("https://example.org").build();
    }

    public Mono<Details> someRestCall(String name) {
        return this.webClient.get().uri("/{name}/details", name)
            .retrieve().bodyToMono(Details.class);
    }

}

```

5.16.1. WebClient 运行时

Spring Boot 将自动检测用于驱动 `WebClient` 的 `ClientHttpConnector`，具体取决于应用程序 `classpath` 上可用的类库。目前支持 Reactor Netty 和 Jetty RS 客户端。

默认情况下 `spring-boot-starter-webflux` starter 依赖于 `io.projectreactor.netty:reactor-netty`，它包含了服务器和客户端的实现。如果您选择将 Jetty 用作响应式服务器，则应添加 Jetty Reactive HTTP 客户端库依赖 `org.eclipse.jetty:jetty-reactive-httpclient`。服务器和客户端使用相同的技术具有一定优势，因为它会自动在客户端和服务器之间共享 HTTP 资源。

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

如果您只希望覆盖客户端选项，则可以定义自己的 `ClientHttpConnector` bean 并完全控制客户端配置。

您可以在 Spring Framework 参考文档中了解有关 `WebClient` 配置选项的更多信息。

5.16.2. 自定义 WebClient

`WebClient` 自定义有三种主要方法, 具体取决于您希望自定义的程度.

要想自定义的作用域尽可能地窄, 请注入自动配置的 `WebClient.Builder`, 然后根据需要调用其方法. `WebClient.Builder` 实例是有状态的: 构建器上的任何更改都会影响到之后所有使用它创建的客户端. 如果要使用相同的构建器创建多个客户端, 可以考虑使用 `WebClient.Builder other = builder.clone();` 的方式克隆构建器.

要在应用程序作用域内对所有 `WebClient.Builder` 实例添加自定义, 可以声明 `WebClientCustomizer` bean 并在注入点局部更改 `WebClient.Builder`.

最后, 您可以回退到原始 API 并使用 `WebClient.create()`. 在这种情况下, 不会应用自动配置或 `WebClientCustomizer`.

5.17. 验证

只要 classpath 上存在 JSR-303 实现 (例如 Hibernate 验证器), 就会自动启用 Bean Validation 1.1 支持的方法验证功能. 这允许 bean 方法在其参数和/或返回值上使用 `javax.validation` 约束进行注解. 带有此类注解方法的目标类需要在类级别上使用 `@Validated` 进行注解, 以便搜索其内联约束注解的方法.

例如, 以下服务触发第一个参数的验证, 确保其大小在 8 到 10 之间:

```
@Service
@Validated
public class MyBean {

    public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code,
                                         Author author) {
        ...
    }
}
```

5.18. 发送邮件

Spring Framework 提供了一个使用 `JavaMailSender` 接口发送电子邮件的简单抽象, Spring Boot 为其提供了自动配置以及一个 `starter` 模块.



有关如何使用 `JavaMailSender` 的详细说明, 请参阅 [参考文档](#).

如果 `spring.mail.host` 和相关库 (由 `spring-boot-starter-mail` 定义) 可用, 则创建默认的 `JavaMailSender` (如果不存在). 可以通过 `spring.mail` 命名空间中的配置项进一步自定义发件人. 有关更多详细信息, 请参阅 [MailProperties](#).

特别是, 某些默认超时时间的值是无限的, 您可能想更改它以避免线程被无响应的邮件服务器阻塞, 如下示例所示:

```
spring:  
  mail:  
    properties:  
      "[mail.smtp.connectiontimeout]": 5000  
      "[mail.smtp.timeout]": 3000  
      "[mail.smtp.writetimeout]": 5000
```

也可以使用 JNDI 中的现有 `Session` 配置一个 `JavaMailSender`:

```
spring:  
  mail:  
    jndi-name: "mail/Session"
```

设置 `jndi-name` 时, 它优先于所有其他与 `Session` 相关的设置.

5.19. JTA 分布式事务

Spring Boot 通过使用 `Atomikos` 嵌入式事务管理器来支持跨多个 XA 资源的分布式 JTA 事务, 并弃用了 `Bitronix` 嵌入式事务管理器的支持, 并在未来版本中删除. 部署在某些 Java EE 应用服务器 (Application Server) 上也支持 JTA 事务.

当检测到 JTA 环境时, Spring 的 `JtaTransactionManager` 将用于管理事务. 自动配置的 JMS、DataSource 和 JPA bean 已升级为支持 XA 事务. 您可以使用标准的 Spring 方式

(例如 `@Transactional`) 来使用分布式事务。如果您处于 JTA 环境中并且仍想使用本地事务，则可以将 `spring.jta.enabled` 属性设置为 `false` 以禁用 JTA 自动配置。

5.19.1. 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` 来获取相应的 Atomikos 库。Spring Boot 自动配置 Atomikos 并确保将合适的依赖设置应用于 Spring bean，以确保启动和关闭顺序正确。

默认情况下，Atomikos 事务日志将写入应用程序主目录（应用程序 jar 文件所在的目录）中的 `transaction-logs` 目录。您可以通过在 `application.properties` 文件中设置 `spring.jta.log-dir` 属性来自定义此目录的位置。也可用 `spring.jta.atomikos.properties` 开头的属性来自定义 Atomikos `UserTransactionServiceImp`。有关完整的详细信息，请参阅 [AtomikosProperties 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 Atomikos 实例配置唯一 ID。默认情况下，此 ID 是运行 Atomikos 的计算机的 IP 地址。在生产环境中要确保唯一性，应为应用程序的每个实例配置 `spring.jta.transaction-manager-id` 属性，并使用不同的值。

5.19.2. 使用 Bitronix 事务管理器



从 Spring Boot 2.3 开始，不赞成使用 Bitronix，并且在将来的版本中将删除它。

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 starter` 为您的项目添加合适的 Bitronix 依赖。与 Atomikos 一样，Spring Boot 会自动配置 Bitronix 并对 bean 进行后处理（post-processes），以确保启动和关闭顺序正确。

默认情况下，Bitronix 事务日志文件（`part1.btm` 和 `part2.btm`）将写入应用程序主目录中的 `transaction-logs` 目录。您可以通过设置 `spring.jta.log-`

`dir` 属性来自定义此目录的位置. 以 `spring.jta.bitronix.properties` 开头的属性绑定到了 `bitronix.tm.Configuration` bean, 允许完全自定义. 有关详细信息, 请参阅 [Bitronix 文档](#).



为确保多个事务管理器能够安全地协调相同的资源管理器, 必须为每个 Bitronix 实例配置唯一的 ID. 默认情况下, 此 ID 是运行 Bitronix 的计算机的 IP 地址. 生产环境要确保唯一性, 应为应用程序的每个实例配置 `spring.jta.transaction-manager-id` 属性, 并使用不同的值.

5.19.3. 使用 Java EE 管理的事务管理器

如果将 Spring Boot 应用程序打包为 `war` 或 `ear` 文件并将其部署到 Java EE 应用程序服务器, 则可以使用应用程序服务器的内置事务管理器. Spring Boot 尝试通过查找常见的 JNDI 位置 (`java:comp/UserTransaction`、`java:comp/TransactionManager` 等) 来自动配置事务管理器. 如果使用应用程序服务器提供的事务服务, 通常还需要确保所有资源都由服务器管理并通过 JNDI 暴露. Spring Boot 尝试通过在 JNDI 路径 (`java:/JmsXA` 或 `java:/JmsXA`) 中查找 `ConnectionFactory` 来自动配置 JMS, 并且可以使用 `spring.datasource.jndi-name` 属性 属性来配置 `DataSource`.

5.19.4. 混合使用 XA 与非 XA JMS 连接

使用 JTA 时, 主 JMS `ConnectionFactory` bean 可识别 XA 并参与分布式事务. 在某些情况下, 您可能希望使用非 XA `ConnectionFactory` 处理某些 JMS 消息. 例如, 您的 JMS 处理逻辑可能需要比 XA 超时时间更长的时间.

如果要使用非 XA `ConnectionFactory`, 可以注入 `nonXaJmsConnectionFactory` bean 而不是 `@Primary jmsConnectionFactory` bean. 为了保持一致性, 提供的 `jmsConnectionFactory` bean 还需要使用 `xaJmsConnectionFactory` 别名.

以下示例展示了如何注入 `ConnectionFactory` 实例:

```
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;

// Inject the XA aware ConnectionFactory (uses the alias and injects the same as
// above)
@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;

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

5.19.5. 支持嵌入式事务管理器

`XAConnectionFactoryWrapper` 和 `XADatasourceWrapper`

接口可用于支持其他嵌入式事务管理器。接口负责包装 `XAConnectionFactory` 和 `XADatasource` bean，并将它们暴露为普通的 `ConnectionFactory` 和 `DataSource` bean，它们透明地加入分布式事务。`DataSource` 和 JMS 自动配置使用 JTA 变体，前提是您需要有一个 `JtaTransactionManager` bean 和在 `ApplicationContext` 中注册有的相应 XA 包装器 (wrapper) bean.

`AtomikosXAConnectionFactoryWrapper` 和 `AtomikosXADatasourceWrapper`

为如何编写 XA 包装器提供了很好示例。

5.20. Hazelcast

如果 `Hazelcast` 在 `classpath` 上并有合适的配置，则 Spring Boot 会自动配置一个可以在应用程序中注入的 `HazelcastInstance`.

Spring Boot 首先尝试通过检查以下配置选项来创建一个客户端：

- 存在 `com.hazelcast.client.config.ClientConfig` bean.
- `spring.hazelcast.config` 属性定义的配置文件.
- 存在 `hazelcast.client.config` 系统属性.

- 工作目录中或 `classpath` 根目录下的 `hazelcast-client.xml`.
- 工作目录中或 `classpath` 根目录下的 `hazelcast-client.yaml`.



Spring Boot 同时支持 Hazelcast 4 和 Hazelcast 3. 如果降级到 Hazelcast 3, 应该将 `hazelcast-client` 添加到类路径中以配置客户端.

如果无法创建客户端, 则 Spring Boot 尝试配置嵌入式服务器.

如果定义了 `com.hazelcast.config.Config` bean, 则 Spring Boot 会使用它. 如果您的配置定义了实例名称, Spring Boot 会尝试查找现有的实例, 而不是创建新实例.

您还可以指定通过配置使用的 Hazelcast 配置文件, 如以下示例所示:

```
spring:
  hazelcast:
    config: "classpath:config/my-hazelcast.xml"
```

否则, Spring Boot 会尝试从默认位置查找 Hazelcast 配置: 工作目录或 `classpath` 根目录中的 `hazelcast.xml`, 或相同位置中的 `.yaml` 文件. 我们还检查是否设置了 `hazelcast.config` 系统属性. 有关更多详细信息, 请参见 [Hazelcast documentation](#). 如果 `classpath` 中存在 `hazelcast-client`, 则 Spring Boot 会首先尝试通过检查以下配置项来创建客户端:



Spring Boot 还为 Hazelcast 提供了缓存支持. 如果启用了缓存, `HazelcastInstance` 将自动包装在 `CacheManager` 实现中.

5.21. Quartz 调度器

Spring Boot 提供了几种使用 Quartz 调度器的便捷方式, 它们来自 `spring-boot-starter-quartz` starter. 如果 Quartz 可用, 则 Spring Boot 将自动配置 `Scheduler` (通过 `SchedulerFactoryBean` 抽象) .

自动选取以下类型的 Bean 并将其与 `Scheduler` 关联起来:

- **JobDetail**: 定义一个特定的 job. 可以使用 **JobBuilder API** 构建 **JobDetail** 实例.
- **Calendar**.
- **Trigger**: 定义何时触发 job.

默认使用内存存储方式的 **JobStore**. 但如果应用程序中有 **DataSource bean**, 并且配置了 **spring.quartz.job-store-type** 属性, 则可以配置基于 JDBC 的存储, 如下所示:

```
spring:
  quartz:
    job-store-type: "jdbc"
```

使用 JDBC 存储时, 可以在启动时初始化 **schema** (表结构), 如下所示:

```
spring:
  quartz:
    jdbc:
      initialize-schema: "always"
```



默认将使用 Quartz 库提供的标准脚本检测并初始化数据库.

这些脚本会删除现有表, 在每次重启时删除所有触发器. 可以通过设置 **spring.quartz.jdbc.schema** 属性来提供自定义脚本.

要让 Quartz 使用除应用程序主 **DataSource** 之外的 **DataSource**, 请声明一个 **DataSource bean**, 使用 **@QuartzDataSource** 注解其 **@Bean** 方法. 这样做可确保 **SchedulerFactoryBean** 和 **schema** 初始化都使用 Quartz 指定的 **DataSource**. 类似地, 要让 Quartz 使用应用程序的主 **TransactionManager** 之外的 **TransactionManager** 来声明 **TransactionManager bean**, 并用 **@QuartzTransactionManager** 注解其 **@Bean** 方法.

默认情况下, 配置创建的 job 不会覆盖已从持久 job 存储读取的已注册的 job.

要启用覆盖现有的 job 定义, 请设置 **spring.quartz.overwrite-existing-jobs** 属性.

Quartz 调取器配置可以使用 **spring.quartz** 属性和 **SchedulerFactoryBeanCustomizer bean** 进行自定义, 它们允许以编程方式的 **SchedulerFactoryBean** 自定义. 可以使用 **spring.quartz.properties.*** 自定义高级

Quartz 配置属性.



需要强调的是, `Executor bean` 与调度程序没有关联, 因为 Quartz 提供了通过 `spring.quartz.properties` 配置调度器的方法. 如果需要自定义 `Actuator`, 请考虑实现 `SchedulerFactoryBeanCustomizer`.

`job` 可以定义 `setter` 以注入数据映射属性. 也可以以类似的方式注入常规的 `bean`, 如下所示:

```
public class SampleJob extends QuartzJobBean {

    private MyService myService;

    private String name;

    // Inject "MyService" bean
    public void setMyService(MyService myService) { ... }

    // Inject the "name" job data property
    public void setName(String name) { ... }

    @Override
    protected void executeInternal(JobExecutionContext context)
        throws JobExecutionException {
        ...
    }
}
```

5.22. 任务执行与调度

在上下文中没有 `Executor bean` 的情况下, Spring Boot 会自动配置一个有合理默认值的 `ThreadPoolTaskExecutor`, 它可以自动与异步任务执行 (`@EnableAsync`) 和 Spring MVC 异步请求处理相关联.



如果您在上下文中定义了自定义 `Executor`, 则常规任务执行 (即 `@EnableAsync`) 将透明地使用它, 但不会配置 Spring MVC 支持, 因为它需要 `AsyncTaskExecutor` 实现 (名为 `applicationTaskExecutor`) . 根据您的目标安排, 您可以将 `Executor` 更改为 `ThreadPoolTaskExecutor`, 或者定义 `Executor` 的 `ThreadPoolTaskExecutor` 和 `AsyncConfigurer` 来包装自定义的 `Executor`.

您可以使用自动配置的 `TaskExecutorBuilder` 来轻松创建实例, 以复制默认的自动配置.

线程池使用 8 个核心线程, 可根据负载情况增加和减少. 可以使用 `spring.task.execution` 命名空间对这些默认设置进行微调, 如下所示:

```
spring:
  task:
    execution:
      pool:
        max-size: 16
        queue-capacity: 100
        keep-alive: "10s"
```

这会将线程池更改为使用有界队列, 在队列满 (100 个任务) 时, 线程池将增加到最多 16 个线程. 当线程在闲置 10 秒 (而不是默认的 60 秒) 时回收线程, 池的收缩更为明显.

如果需要与调度任务执行 (`@EnableScheduling`) 相关联, 可以自动配置一个 `ThreadPoolTaskScheduler`. 默认情况下, 线程池使用一个线程, 可以使用 `spring.task.scheduling` 命名空间对这些设置进行微调.

如果需要创建自定义 `Actuator` 或调度器, 则在上下文中可以使用 `TaskExecutorBuilder` bean 和 `TaskSchedulerBuilder` bean.

5.23. Spring Integration

Spring Boot 为 `Spring Integration` 提供了一些便捷的使用方式, 它们包含在 `spring-boot-starter-integration` starter 中. `Spring Integration` 为消息传递以及其他传输 (如 HTTP、TCP 等) 提供了抽象. 如果 `classpath` 上存在

Spring Integration，则 Spring Boot 会通过 `@EnableIntegration` 注解对其进行初始化。

Spring Boot 还配置了一些由其他 Spring Integration 模块触发的功能。如果 `spring-integration-jmx` 也在 `classpath` 上，则消息处理统计信息将通过 JMX 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，如下所示：

```
spring:  
  integration:  
    jdbc:  
      initialize-schema: "always"
```

如果可用 `spring-integration-rsocket`，则开发人员可以使用 `"spring.rsocket.server.*"` 属性配置 RSocket 服务器，并使其使用 `IntegrationRSocketEndpoint` 或 `RSocketOutboundGateway` 组件来处理传入的 RSocket 消息。该基础结构可以处理 Spring Integration RSocket 通道适配器和 `@MessageMapping` 处理程序（已配置 `"spring.integration.rsocket.server.message-mapping-enabled"`）。

Spring Boot 还可以使用配置属性来自动配置 `ClientRSocketConnector`：

```
# Connecting to a RSocket server over TCP  
spring:  
  integration:  
    rsocket:  
      client:  
        host: "example.org"  
        port: 9898
```

```
# Connecting to a RSocket Server over WebSocket  
spring:  
  integration:  
    rsocket:  
      client:  
        uri: "ws://example.org"
```

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 `Micrometer` 将管理 `Spring Integration` 的指标. 如果您希望使用旧版 `Spring Integration` 指标,请将 `DefaultMetricsFactory bean` 添加到应用程序上下文中.

5.24. Spring Session

`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,可以自动配置以下存储:

- JDBC
- Redis
- Hazelcast
- MongoDB

`Servlet` 的自动配置取代了使用 `@Enable*HttpSession` 的需要.

构建响应式 `Web` 应用程序时,可以自动配置以下存储:

- Redis
- MongoDB

`reactive` 的自动配置取代了使用 `@Enable*WebSession` 的需要.

如果 `classpath` 上存在单个 `Spring Session` 模块,则 `Spring Boot` 会自动使用该存储实现. 如果您有多个实现,则必须选择要用于存储会话的 `StoreType`. 例如,要使用 `JDBC` 作为后端存储,您可以按如下方式配置应用程序:

```
spring:  
  session:  
    store-type: "jdbc"
```



可以将 `store-type` 设置为 `none` 来禁用 `Spring Session`.

每个 `store` 都有自己的额外设置. 例如,可以为 `JDBC` 存储定制表的名称,如下所示:

```
spring:  
  session:  
    jdbc:  
      table-name: "SESSIONS"
```

可以使用 `spring.session.timeout` 属性来设置会话的超时时间。如果未在 `Servlet web application` 设置该属性，则自动配置将使用 `server.servlet.session.timeout` 的值。

您可以使用 `@EnableHttpSession (Servlet)` 或 `@EnableWebSession (Reactive)` 来控制 Spring Session 的配置。这将导致自动配置退出。然后，可以使用注解的属性而不是先前描述的配置属性来配置 Spring Session。

5.25. 通过 JMX 监控和管理

Java Management Extensions (JMX, Java 管理扩展)

提供了一种监视和管理应用程序的标准机制。默认情况下，Spring Boot 会创建一个 ID 为 `mbeanServer` 的 `MBeanServer` bean，并暴露使用 Spring JMX 注解 (`@ManagedResource`、`@ManagedAttribute` 或 `@ManagedOperation`) 的 bean。

有关更多详细信息，请参阅 `JmxAutoConfiguration` 类。

5.26. 测试

Spring Boot 提供了许多工具类和注解，可以在测试应用程序时提供帮助。

主要由两个模块提供：`spring-boot-test` 包含核心项，`spring-boot-test-autoconfigure` 支持测试的自动配置。

大多数开发人员都使用 `spring-boot-starter-test` “Starter”，它会导入 Spring Boot 测试模块以及 JUnit Jupiter, AssertJ, Hamcrest 和许多其他有用的库。

如果您有使用 JUnit 4 的测试，可以使用 JUnit 5 的 `vintage engine` 来运行它们。要使用 `vintage engine`，添加一个依赖 `junit-vintage-engine`，如下所示：



```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

`hamcrest-core` 被排除在外，因为支持 `org.hamcrest:hamcrest` 是 `spring-boot-starter-test` 的一部分。

5.26.1. 依赖范围测试

`spring-boot-starter-test` “Starter” (在 `test scope`) 包含以下的库：

- [JUnit 5](#): The de-facto standard for unit testing Java applications.
- [Spring 测试 & Spring Boot 测试](#): 对Spring Boot应用程序的实用程序和集成测试支持。
- [AssertJ](#): 流式的断言库。
- [Hamcrest](#): 匹配对象库（也称为约束或断言）。
- [Mockito](#): 一个Java模拟框架。
- [JSONassert](#): JSON的断言库。
- [JsonPath](#): JSON的XPath。

通常，我们发现这些通用库在编写测试时很有用。如果这些库不满足您的需求，则可以添加自己的其他测试依赖。

5.26.2. 测试 Spring 应用程序

依赖注入的主要优点之一是, 它应该使您的代码更易于进行单元测试. 您可以使用 `new` 运算符实例化对象, 甚至无需使用 Spring. 您还可以使用模拟对象而不是实际的依赖.

通常, 您需要超越单元测试并开始集成测试 (使用 Spring `ApplicationContext`) . 能够进行集成测试而无需部署应用程序或连接到其他基础结构是很有用的.

Spring 框架包括用于此类集成测试的专用测试模块. 您可以直接向 `org.springframework:spring-test` 声明依赖, 也可以使用 `spring-boot-starter-test` "Starter" 将其传递.

如果您以前没有使用过 `spring-test` 模块, 则应先阅读 [Spring Framework 参考文档的相关部分](#) .

5.26.3. 测试 Spring Boot 应用程序

Spring Boot 应用程序是 Spring `ApplicationContext`, 因此除了对普通 Spring 上下文进行常规测试以外, 无需执行任何其他特殊操作即可对其进行测试.



默认情况下, 仅当您使用 `SpringApplication` 创建 Spring Boot 的外部属性, 日志记录和其他功能时, 才将它们安装在上下文中.

Spring Boot 提供了 `@SpringBootTest` 注解, 当您需要 Spring Boot 功能时, 可以将其用作标准 `spring-test @ContextConfiguration` 注解的替代方法. [注解通过创建 `SpringApplication` 在测试中使用的 `ApplicationContext` 来起作用](#). 除了 `@SpringBootTest` 之外, 还提供了许多其他注解来测试应用程序的特定的部分.



如果您使用的是 JUnit 4, 请不要忘记也将 `@RunWith(SpringRunner.class)` 添加到测试中, 否则注解将被忽略. 如果您使用的是 JUnit 5, 则无需将等效的 `@ExtendWith(SpringExtension.class)` 添加为 `@SpringBootTest`, 而其他 `@...Test` 注解已经在其中进行了注解.

默认情况下, `@SpringBootTest` 将不会启动服务器. 您可以使用 `@SpringBootTest` 的 `webEnvironment` 属性来进一步完善测试的运行方式:

- **MOCK**(默认) : 加载 `Web ApplicationContext` 并提供模拟 Web 环境。
使用此注解时,不会启动嵌入式服务器. 如果您的类路径上没有 Web 环境
,则此模式将透明地退回到创建常规的非 Web `ApplicationContext`. 它可以与
`@AutoConfigureMockMvc` 或 `@AutoConfigureWebTestClient` 结合使用,以对 Web
应用程序进行基于模拟的测试.
- **RANDOM_PORT**: 加载 `WebServerApplicationContext` 并提供真实的 Web 环境.
在随机的端口启动并监听嵌入式服务器.
- **DEFINED_PORT**: 加载 `WebServerApplicationContext` 并提供真实的 Web 环境.
在定义的端口(来自 `application.properties`) 或 `8080`
端口启动并监听嵌入式服务器
- **NONE**: 使用 `SpringApplication` 加载 `ApplicationContext`,但不提供任何 Web
环境 (模拟或其他方式) .

如果您的测试是 `@Transactional`

,则默认情况下它将在每个测试方法的末尾回滚事务. 但是
由于将这种安排与 `RANDOM_PORT` 或 `DEFINED_PORT`
一起使用隐式提供了一个真实的 `Servlet` 环境,因此 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,服务器上启动的任何事务都不会回滚.



如果您的应用程序将不同的端口用于管理服务器,则 `@SpringBootTest`
的 `webEnvironment=WebEnvironment.RANDOM_PORT`
也将在单独的随机端口上启动管理服务器.



检测 Web 应用程序类型

如果 `Spring MVC` 可用,则配置基于常规MVC的应用程序上下文. 如果您只有 `Spring WebFlux`,我们将检测到该情况并配置基于 `WebFlux` 的应用程序上下文.

如果两者都存在,则 `Spring MVC` 优先. 如果要在这种情况下测试响应式 Web 应用程序
,则必须设置 `spring.main.web-application-type` 属性:

```
@SpringBootTest(properties = "spring.main.web-application-type=reactive")
class MyWebFluxTests { ... }
```

检测测试配置

如果您熟悉 Spring Test Framework，则可能习惯于使用 `@ContextConfiguration(classes=...)` 以指定要加载哪个 Spring `@Configuration`. 另外，您可能经常在测试中使用嵌套的 `@Configuration` 类.

在测试 Spring Boot 应用程序时，通常不需要这样做。只要您没有明确定义，Spring Boot 的 `@*Test` 注解就会自动搜索您的主要配置。

搜索算法从包含测试的程序包开始工作，直到找到带有 `@SpringBootApplication` 或 `@SpringBootConfiguration` 注解的类。只要您以合理的方式对 代码进行结构化，通常就可以找到您的主要配置。

如果您使用测试注解来测试应用程序的特定部分，，则应避免在 应用程序的 `main方法` 中添加特定于特定区域的配置设置。



`@SpringBootApplication` 的基础组件扫描配置定义了排除过滤器，这些过滤器用于确保切片按预期工作。如果在 `@SpringBootApplication` 注解的类上使用显式的 `@ComponentScan` 指令，请注意这些过滤器将被禁用。如果使用切片，则应再次定义它们。

如果要自定义主要配置，则可以使用嵌套的 `@TestConfiguration` 类。与将使用嵌套的 `@Configuration` 类代替应用程序的主要配置不同的是，在应用程序的主要配置之外还使用了嵌套的 `@TestConfiguration` 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（无论如何发现），加载上下文的潜在耗时过程就只会发生一次。

排除测试配置

如果您的应用程序使用组件扫描（例如，如果使用 `@SpringBootApplication` 或 `@ComponentScan`），则可能会发现偶然为各地创建的仅为特定测试创建的顶级配置类。

如前所述，`@TestConfiguration` 可以用于测试的内部类以自定义主要配置。当放置在顶级类上时，`@TestConfiguration` 指示不应通过扫描选择 `src/test/java` 中的类。然后，可以在需要的位置显式导入该类，如以下示例所示：

```
@SpringBootTest  
@Import(MyTestsConfiguration.class)  
class MyTests {  
  
    @Test  
    void exampleTest() {  
        ...  
    }  
  
}
```



如果直接使用 `@ComponentScan` (即不是通过 `@SpringBootApplication`) , 则需要向其中注册 `TypeExcludeFilter`. 有关详细信息, 请参见 [Javadoc](#).

使用应用程序参数

如果您的应用程序需要参数, 则可以使用 `args` 属性让 `@SpringBootTest` 注入参数.

```
@SpringBootTest(args = "--app.test=one")  
class ApplicationArgumentsExampleTests {  
  
    @Test  
    void applicationArgumentsPopulated(@Autowired ApplicationArguments args) {  
        assertThat(args.getOptionNames()).containsOnly("app.test");  
        assertThat(args.getOptionValues("app.test")).containsOnly("one");  
    }  
  
}
```

在模拟环境中进行测试

默认情况下, `@SpringBootTest` 不会启动服务器. 如果您有要在此模拟环境下进行测试的 Web 端点, 则可以另外配置 `MockMvc`, 如以下示例所示:

```
import org.junit.jupiter.api.Test;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import  
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.web.servlet.MockMvc;  
  
import static  
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;  
import static  
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;  
import static  
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;  
  
@SpringBootTest  
@AutoConfigureMockMvc  
class MockMvcExampleTests {  
  
    @Test  
    void exampleTest(@Autowired MockMvc mvc) throws Exception {  
  
        mvc.perform(get("/")).andExpect(status().isOk()).andExpect(content().string("Hel  
lo World"));  
    }  
  
}
```



如果只想关注 Web 层而不希望启动完整的 `ApplicationContext`，请考虑使用 `@WebMvcTest`.

另外，您可以配置 `WebTestClient`，如以下示例所示：

```

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestCli
ent;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest
@AutoConfigureWebTestClient
class MockWebTestClientExampleTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {

        webClient.get().uri("/").exchange().expectStatus().isOk().expectBody(String.clas
s).isEqualTo("Hello World");
    }

}

```

在模拟环境中进行测试通常比在完整的 `Servlet` 容器中运行更快。但是，由于模拟发生在 `Spring MVC` 层，因此无法使用 `MockMvc` 直接测试依赖于较低级别 `Servlet` 容器行为的代码。



例如，`Spring Boot` 的错误处理基于 `Servlet` 容器提供的“`error page`”支持。这意味着，尽管您可以按预期测试 `MVC` 层引发并处理异常，但是您无法直接测试是否呈现了特定的[自定义错误页面](#)。如果需要测试这些较低级别的问题，则可以按照下一节中的描述启动一个完全运行的服务器。

使用正在运行的服务器进行测试

如果需要启动完全运行的服务器，建议您使用随机端口。如果使用 `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`，则每次运行测试时都会随机选择一个可用端口。

`@LocalServerPort` 注解可用于将[将实际使用的端口注入](#) 测试中。为了方便起见，需要对已启动的服务器进行 `REST` 调用的测试可以 `@Autowired` 附加地使用 `WebTestClient`，该 `WebTestClient` 解析到正在运行的服务器的相对链接。

,并带有用于验证响应的专用 API,如以下示例所示:

```
import org.junit.jupiter.api.Test;

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.web.reactive.server.WebTestClient;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class RandomPortWebTestClientExampleTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {

        webClient.get().uri("/").exchange().expectStatus().isOk().expectBody(String.class)
            .isEqualTo("Hello World");
    }

}
```

这种设置需要在类路径上使用 **spring-webflux**. 如果您无法或不会添加 **webflux**, 则 Spring Boot 还提供了 **TestRestTemplate** 工具:

```
import org.junit.jupiter.api.Test;

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 static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class RandomPortTestRestTemplateExampleTests {

    @Test
    void exampleTest(@Autowired TestRestTemplate restTemplate) {
        String body = restTemplate.getForObject("/", String.class);
        assertThat(body).isEqualTo("Hello World");
    }

}
```

自定义 WebTestClient

要定制 `WebTestClient` bean, 请配置 `WebTestClientBuilderCustomizer` bean.

将使用用于创建 `WebTestClient` 的 `WebTestClient.Builder` 调用任何此类 bean.

使用 JMX

由于测试上下文框架缓存上下文, 因此默认情况下禁用 JMX 以防止相同组件在同一域上注册.

如果此类测试需要访问 `MBeanServer`, 请考虑将其标记为脏:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
class SampleJmxTests {

    @Autowired
    private MBeanServer mBeanServer;

    @Test
    void exampleTest() {
        // ...
    }

}
```

Using Metrics

无论您的类路径是什么, 在使用 `@SpringBootTest` 时, `meter` 注册表(内存中支持的除外)都不会自动配置。

如果您需要将指标作为集成测试的一部分导出到不同的后端, 请使用

`@AutoConfigureMetrics` 注解它

模拟和检测 Bean

运行测试时, 有时有必要在应用程序上下文中模拟某些组件. 例如
, 您可能在开发过程中无法使用某些远程服务的外观.

当您要模拟在实际环境中可能难以触发的故障时, 模拟也很有用.

Spring Boot 包含一个 `@MockBean` 注解, 可用于为 `ApplicationContext` 中的 bean 定义

Mockito 模拟. 您可以使用注解添加新 bean 或替换单个现有 bean 定义.

注解可以直接用于测试类, 测试中的字段或 **@Configuration** 类和字段. 在字段上使用时, 还将注入创建的模拟的实例. 每种测试方法后, 模拟 Bean 都会自动重置.

如果您的测试使用 Spring Boot 的测试注解之一 (例如 **@SpringBootTest**) , 则会自动启用此功能. 要以其他方式使用此功能, 必须显式添加监听器, 如以下示例所示:



```
@TestExecutionListeners({ MockitoTestExecutionListener.class,  
    ResetMocksTestExecutionListener.class })
```

下面的示例用模拟实现替换现有的 **RemoteService** bean:

```
import org.junit.jupiter.api.Test;  
import org.springframework.beans.factory.annotation.*;  
import org.springframework.boot.test.context.*;  
import org.springframework.boot.test.mock.mockito.*;  
  
import static org.assertj.core.api.Assertions.*;  
import static org.mockito.BDDMockito.*;  
  
@SpringBootTest  
class MyTests {  
  
    @MockBean  
    private RemoteService remoteService;  
  
    @Autowired  
    private Reverser reverser;  
  
    @Test  
    void exampleTest() {  
        // RemoteService has been injected into the reverser bean  
        given(this.remoteService.someCall()).willReturn("mock");  
        String reverse = reverser.reverseSomeCall();  
        assertThat(reverse).isEqualTo("kcom");  
    }  
}
```



`@MockBean` 不能用于模拟应用程序上下文刷新期间执行的 bean 的行为. 到执行测试时, 应用程序上下文刷新已完成, 并且配置模拟行为为时已晚. 我们建议在这种情况下使用 `@Bean` 方法创建和配置模拟.

此外, 您可以使用 `@SpyBean` 用 Mockito 间谍包装任何现有的 bean. 有关完整的详细信息, 请参见 [Javadoc](#).



CGLib代理 (例如为作用域内的Bean创建的代理) 将代理方法声明为 `final`. 这将阻止 Mockito 正常运行, 因为它无法在其默认配置中模拟或监视最终方法. 如果要模拟或监视这样的 bean, 请通过将 `org.mockito: mockito-inline` 添加到应用程序的测试依赖中, 将 Mockito 配置为使用其嵌入式模拟生成器. 这允许 Mockito 模拟和监视 `final` 方法.



Spring 的测试框架在测试之间缓存应用程序上下文, 并为共享相同配置的测试重用上下文, 而 `@MockBean` 或 `@SpyBean` 的使用会影响缓存键, 这很可能会增加上下文数量.



如果您使用 `@SpyBean` 通过 `@Cacheable` 方法监视通过名称引用参数的 bean, 则必须使用 `-parameters` 编译应用程序. 这样可以确保一旦侦察到 bean, 参数名称就可用于缓存基础结构.



当您使用 `@SpyBean` 监视由 Spring 代理的 bean 时, 在某些情况下, 例如使用 `given` 或 `when` 设置期望值时, 您可能需要删除 Spring 的代理. 使用 `AopTestUtils.getTargetObject(yourProxiedSpy)`

自动配置测试

Spring Boot 的自动配置系统适用于应用程序, 但有时对测试来说可能有点过多. 它通常仅有助于加载测试应用程序 "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, 并且您不想在这些测试中涉及数据库调用, 或者您想要测试 JPA 实体, 并且对那些 JPA 实体不感兴趣. 测试运行.

`spring-boot-test-autoconfigure` 模块包括许多注解, 可用于自动配置此类 "切片".

它们中的每一个都以相似的方式工作, 提供了一个 `@...Test` 注解 (该注解加载了 `ApplicationContext`) 以及一个或多个 `@AutoConfigure...` (可用于自定义自动配置设置的注解) .



每个 “`slicing`” 将组件扫描限制为适当的组件，并加载一组非常受限制的自动配置类. 如果您需要排除其中之一, 大多数 `@...Test` 注解提供了 `excludeAutoConfiguration` 属性. 或者, 您可以使用 `@ImportAutoConfiguration#exclude`.



不支持在一个测试中使用多个 `@...Test` 注解来包含多个 “片段”. 如果您需要多个 “`slices`”, 请选择 `@...Test` 注解之一, 并手动添加其他 “`slices`” 的 `@AutoConfigure...` 注解.



也可以将 `@AutoConfigure...` 注解与标准的 `@SpringBootTest` 注解一起使用. 如果您对 “`slicing`” 应用程序不感兴趣, 但需要一些自动配置的测试bean, 则可以使用此组合.

自动配置的 JSON 测试

要测试对象 JSON 序列化和反序列化是否按预期工作, 可以使用 `@JsonTest` 注解. `@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库之一:

- Jackson `ObjectMapper`, any `@JsonComponent` beans and any Jackson Modules
- Gson
- Jsonb



可以在[附录](#)中找到由 `@JsonTest` 启用的自动配置列表.

如果需要配置自动配置的元素, 则可以使用 `@AutoConfigureJsonTesters` 注解.

Spring Boot 包含基于 AssertJ 的帮助程序, 这些帮助程序可与 `JSONAssert` 和 `JsonPath` 库一起使用, 以检查 JSON 是否按预期方式显示. `JacksonTester`, `GsonTester`, `JsonbTester` 和 `BasicJsonTester` 类可以分别用于 Jackson, Gson, Jsonb 和

Strings. 使用 `@JsonTest` 时, 可以使用 `@Autowired` 测试类上的任何帮助程序字段.

以下示例显示了 Jackson 的测试类:

```
import org.junit.jupiter.api.Test;
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 static org.assertj.core.api.Assertions.*;

@JsonTest
class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    void testSerialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a `json` file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");

        assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make")
            .isEqualTo("Honda");
    }

    @Test
    void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\", \"model\":\"Focus\"}";
        assertThat(this.json.parse(content))
            .isEqualTo(new VehicleDetails("Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }
}
```



JSON 帮助程序类也可以直接在标准单元测试中使用. 为此, 如果不使用 `@JsonTest`, 请在 `@Before` 方法中调用帮助程序的 `initFields` 方法.

如果您使用的是 Spring Boot 基于 AssertJ 的帮助器, 以给定的 JSON 路径对数字值进行断言, 则取决于类型, 您可能无法使用 `isEqualTo`. 相反, 您可以使用

`AssertJ` 的满足条件来断言该值符合给定条件. 例如, 以下示例断言实际数是一个偏移量为 `0.01` 且接近 `0.15` 的浮点值.

```
assertThat(json.write(message))
    .extractingJsonPathNumberValue("@.test.numberValue")
    .satisfies((number) -> assertThat(number.floatValue()).isCloseTo(0.15f,
within(0.01f)));
```

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` 注解. `@WebMvcTest` 自动配置 Spring MVC 基础结构, 并将扫描的 bean 限制为 `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `HandlerInterceptor`, `WebMvcConfigurer`, 和 `HandlerMethodArgumentResolver`. 使用此注解时, 不扫描常规 `@Component`, `@ConfigurationProperties` bean. `@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean



可以在[附录中](#)找到 `@WebMvcTest` 启用的自动配置设置的列表.



如果需要注册其他组件, 例如 Jackson 模块, 则可以在测试中使用 `@Import` 导入其他配置类.

`@WebMvcTest` 通常仅限于单个控制器, 并与 `@MockBean` 结合使用, 以为所需的协作者提供模拟实现.

`@WebMvcTest` 还可以自动配置 `MockMvc`. `Mock MVC` 提供了一种强大的方法来快速测试 MVC 控制器, 而无需启动完整的 HTTP 服务器.



您还可以通过在非 `@WebMvcTest` (例如 `@SpringBootTest`) 中使用 `@AutoConfigureMockMvc` 对其进行注解来自动配置 `MockMvc`.

以下示例使用 `MockMvc`:

```
import org.junit.jupiter.api.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@WebMvcTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk()).andExpect(content().string("Honda
Civic"));
    }

}
```



如果您需要配置自动配置的元素（例如，当应该应用 `Servlet` 过滤器时），则可以使用 `@AutoConfigureMockMvc` 注解中的属性。

如果使用 `HtmlUnit` 或 `Selenium`，则自动配置还会提供 `HtmlUnit WebClient` bean 和 / 或 `Selenium WebDriver` bean。以下示例使用 `HtmlUnit`：

```

import com.gargoylesoftware.htmlunit.*;
import org.junit.jupiter.api.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@WebMvcTest(UserVehicleController.class)
class MyHtmlUnitTests {

    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
    }

}

```



默认情况下, Spring Boot 将 `WebDriver` bean 放在特殊的 “scope” 中, 以确保驱动程序在每次测试后退出并注入新实例。如果您不希望出现这种情况, 则可以将 `@Scope("singleton")` 添加到 `WebDriver @Bean` 定义中。



Spring Boot 创建的 `webDriver`

作用域将替换任何用户定义的同名作用域。如果定义自己的 `WebDriver` 作用域, 则使用 `@WebMvcTest` 时可能会发现它停止工作。

如果您在类路径上具有 `Spring Security`, 则 `@WebMvcTest` 还将扫描 `WebSecurityConfigurer Bean`。您可以使用 `Spring Security` 的测试支持来代替完全禁用此类测试的安全性。有关如何使用 `Spring Security` 的 `MockMvc` 支持的更多详细信息, 请参见 [howto.html](#) 操作方法部分。



有时编写 Spring MVC 测试是不够的。Spring Boot 可以帮助您在实际服务器上运行完整的端到端测试。

自动配置的 Spring WebFlux 测试

要测试 Spring WebFlux 控制器是否按预期工作，可以使用 `@WebFluxTest` 注解。`@WebFluxTest` 自动配置 Spring WebFlux 基础结构，并将扫描的 bean 限制为 `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `WebFilter` 和 `WebFluxConfigurer`。使用 `@WebFluxTest` 注解时，不扫描常规 `@Component`, `@ConfigurationProperties` bean。`@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean



可以在[附录中](#)找到 `@WebFluxTest` 启用的自动配置的列表。.



如果需要注册其他组件，例如 Jackson 模块，则可以在测试中使用 `@Import` 导入其他配置类。

通常，`@WebFluxTest` 仅限于单个控制器，并与 `@MockBean` 注解结合使用，以为所需的协作者提供模拟实现。

`@WebFluxTest` 还可以自动配置 `WebTestClient`，它提供了一种强大的方法来快速测试 WebFlux 控制器，而无需启动完整的 HTTP 服务器。



您还可以通过在非 `@WebFluxTest` (例如 `@SpringBootTest`) 中自动配置 `WebTestClient`，方法是使用 `@AutoConfigureWebTestClient` 对其进行注解。

下面的示例显示一个同时使用 `@WebFluxTest` 和 `WebTestClient` 的类：

```

import org.junit.jupiter.api.Test;

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.web.reactive.server.WebTestClient;

@WebFluxTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private WebTestClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN)
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Honda Civic");
    }
}

```



WebFlux 应用程序仅支持此设置,因为在模拟的 Web 应用程序中使用 `WebTestClient` 目前仅与 WebFlux 一起使用.



`@WebFluxTest` 无法检测通过功能 Web 框架注册的路由.
为了在上下文中测试 `RouterFunction` bean,请考虑自己通过 `@Import` 或使用 `@SpringBootTest` 导入 `RouterFunction`.



`@WebFluxTest` 无法检测通过 `SecurityWebFilterChain` 类型的 `@Bean` 注册的自定义安全配置. 要将其包括在测试中,您将需要通过 `@Import` 导入或使用 `@SpringBootTest` 导入用于注册 bean 的配置.



有时编写 Spring WebFlux 测试是不够的。Spring Boot 可以帮助您在[实际服务器上运行完整的端到端测试](#)。

自动配置的 Data JPA 测试

您可以使用 `@DataJpaTest` 注解来测试 JPA 应用程序。默认情况下，它将扫描 `@Entity` 类并配置 Spring Data JPA 存储库。如果在类路径上有嵌入式数据库，它也将配置一个。使用 `@DataJpaTest` 注解时，不扫描常规 `@Component`,`@ConfigurationProperties` bean。`@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean。默认情况下，通过将 `spring.jpa.show-sql` 属性设置为 `true` 来记录 SQL 查询。可以使用注解的 `showSql()` 属性禁用此功能。



可以在[附录中](#)找到由 `@DataJpaTest` 启用的自动配置设置的列表。

默认情况下，数据 JPA 测试是事务性的，并在每次测试结束时回滚。有关更多详细信息，请参见《Spring Framework 参考文档》中的 [相关部分](#)。如果这不是您想要的，则可以按以下方式禁用测试或整个类的事务管理：

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class ExampleNonTransactionalTests {

}
```

数据 JPA 测试也可以注入 `TestEntityManager` bean，它为专门为测试设计的标准 JPA `EntityManager` 提供了替代方法。如果要在 `@DataJpaTest` 实例之外使用 `TestEntityManager`，也可以使用 `@AutoConfigureTestEntityManager` 注解。如果需要，还可以使用 `JdbcTemplate`。以下示例显示了正在使用的 `@DataJpaTest` 注解：

```

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

import static org.assertj.core.api.Assertions.*;

@DataJpaTest
class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    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");
    }

}

```

内存嵌入式数据库通常运行良好, 不需要任何安装, 因此通常可以很好地进行测试. 但是, 如果您希望对真实数据库运行测试, 则可以使用 `@AutoConfigureTestDatabase` 注解, 如以下示例所示:

```

@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
class ExampleRepositoryTests {

    // ...

}

```

自动配置的 JDBC 测试

`@JdbcTest` 与 `@DataJpaTest` 相似, 但适用于仅需要数据源并且不使用 Spring Data JDBC 的测试. 默认情况下, 它配置一个内存嵌入式数据库和一个 `JdbcTemplate`. 使用 `@JdbcTest` 注解时, 不扫描常规 `@Component`, `@ConfigurationProperties` bean. `@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties`

的 bean



可以在 [附录中](#)找到 `@JdbcTest` 启用的自动配置的列表.

默认情况下, JDBC 测试是事务性的, 并在每个测试结束时回滚. 有关更多详细信息, 请参见《Spring Framework 参考文档》中的<https://docs.jcchay.com/docs/spring-framework/5.3.6/html5/zh-cn/testing.html#testcontext-tx-enabling-transactions>[相关部分]. 如果这不是您想要的, 则可以为测试或整个类禁用事务管理, 如下所示:

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Transactional(propagation = Propagation.NOT_SUPPORTED)
class ExampleNonTransactionalTests {

}
```

如果您希望测试针对真实数据库运行, 则可以使用 `@AutoConfigureTestDatabase` 注解, 其方式与 `DataJpaTest` 相同. (请参阅[自动配置的 Data JPA 测试.](#))

自动配置的 Data JDBC 测试

`@DataJdbcTest` 与 `@JdbcTest` 相似, 但适用于使用 Spring Data JDBC 存储库的测试. 默认情况下, 它配置一个内存嵌入式数据库, 一个 `JdbcTemplate` 和 Spring Data JDBC 存储库. 使用 `@DataJdbcTest` 注解时, 不扫描常规 `@Component`, `@ConfigurationProperties` bean. `@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean



可以在[附录中](#)找到由 `@DataJdbcTest` 启用的自动配置的列表.

默认情况下, Data JDBC 测试是事务性的, 并在每个测试结束时回滚. 有关更多详细信息, 请参见《Spring Framework 参考文档》中的 [相关部分](#). 如果这不是您想要的, 则可以禁用测试或整个测试类的事务管理, 如 [JDBC 示例所示](#).

如果您希望测试针对真实数据库运行，则可以使用 `@AutoConfigureTestDatabase` 注解，其方式与 `DataJpaTest` 相同。（请参阅[自动配置的 Data JPA 测试](#)。）

自动配置的 `jOOQ Tests`

您可以以与 `@JdbcTest` 类似的方式使用 `@JooqTest`，但可以用于与 jOOQ 相关的测试。由于 jOOQ 严重依赖于数据库模式相对应的基于 Java 的模式，因此将使用现有的 `DataSource`。如果要将其替换为内存数据库，则可以使用 `@AutoConfigureTestDatabase` 覆盖这些设置。（有关在 Spring Boot 中使用 jOOQ 的更多信息，请参阅本章前面的“[使用 jOOQ](#)”。）使用 `@JooqTest` 注解时，不扫描常规 `@Component`, `@ConfigurationProperties` bean。`@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean



可以在[附录中](#)找到 `@JooqTest` 启用的自动配置的列表。

`@JooqTest` 配置 `DSLContext`。常规 `@Component` bean 未加载到 `ApplicationContext` 中。以下示例显示了正在使用的 `@JooqTest` 注解：

```
import org.jooq.DSLContext;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;

@JooqTest
class ExampleJooqTests {

    @Autowired
    private DSLContext dslContext;
}
```

JOOQ 测试是事务性的，默认情况下会在每个测试结束时回滚。如果这不是您想要的，则可以禁用测试或整个测试类的事务管理，如[JDBC 示例所示](#)。

自动配置的 `Data MongoDB` 测试

您可以使用 `@DataMongoTest` 测试 MongoDB 应用程序。默认情况下，它配置内存嵌入式 MongoDB（如果可用），配置 `MongoTemplate`，扫描 `@Document` 类，并配置 Spring Data MongoDB 存储库。使用 `@DataMongoTest` 注解时，不扫描常规 `@Component`, `@ConfigurationProperties` bean。`@EnableConfigurationProperties` 可用于包含

`@ConfigurationProperties` 的 bean (有关将 MongoDB 与 Spring Boot 结合使用的更多信息,请参阅本章前面的 "[MongoDB](#)")



可以在 [附录中](#)找到由 `@DataMongoTest` 启用的自动配置设置的列表.

此类显示正在使用的 `@DataMongoTest` 注解:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;

@DataMongoTest
class ExampleDataMongoTests {

    @Autowired
    private MongoTemplate mongoTemplate;

    //
}
```

内存嵌入式 MongoDB 通常运行良好,并且不需要任何开发人员安装,因此通常可以很好地用于测试. 但是,如果您希望对真实的 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.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;

@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)
class ExampleDataMongoNonEmbeddedTests {

}
```

自动配置的 Data Neo4j 测试

您可以使用 `@DataNeo4jTest` 来测试 Neo4j 应用程序. 默认情况下, 他会扫描 `@Node` 类,并配置 Spring Data Neo4j 存储库. 使用 `@DataNeo4jTest` 注解时,不扫描常规 `@Component, @ConfigurationProperties` bean.`@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean. (有关将 Neo4J 与 Spring Boot

结合使用的更多信息,请参阅本章前面的 "[Neo4j](#)".)



可以在 [附录中](#)找到由 `@DataNeo4jTest` 启用的自动配置设置的列表.

以下示例显示了在 Spring Boot 中使用 Neo4J 测试的典型设置:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;

@DataNeo4jTest
class ExampleDataNeo4jTests {

    @Autowired
    private YourRepository repository;

    //
}
```

默认情况下, Data Neo4j 测试是事务性的,并在每次测试结束时回滚. 有关更多详细信息,请参见《Spring Framework 参考文档》中的 [相关部分](#). 如果这不是您想要的,则可以为测试或整个类禁用事务管理,如下所示:

```
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class ExampleNonTransactionalTests {

}
```



响应式访问不支持事务性测试。如果您正在使用这种样式, 您必须如上所述配置 `@DataNeo4jTest` 测试.

自动配置的 Data Redis 测试

您可以使用 `@DataRedisTest` 测试 Redis 应用程序. 默认情况下, 它会扫描 `@RedisHash` 类并配置 Spring Data Redis 存储库. 使用 `@DataRedisTest` 注解时, 不扫描常规

`@Component, @ConfigurationProperties` bean. `@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean. (有关将 Redis 与 Spring Boot 结合使用的更多信息, 请参阅本章前面的 "[Redis](#)".)



可以在 [附录中](#) 找到由 `@DataRedisTest` 启用的自动配置设置的列表.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;

@DataRedisTest
class ExampleDataRedisTests {

    @Autowired
    private YourRepository repository;

    //
}
```

自动配置的 Data LDAP 测试

您可以使用 `@DataLdapTest` 来测试 LDAP 应用程序. 默认情况下, 它配置内存嵌入式 LDAP (如果可用), 配置 `LdapTemplate`, 扫描 `@Entry` 类, 并配置 Spring Data LDAP 存储库. 使用 `@DataLdapTest` 注解时, 不扫描常规 `@Component, @ConfigurationProperties` bean. `@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean. (有关将 LDAP 与 Spring Boot 结合使用的更多信息, 请参阅本章前面的 "[LDAP](#)".)



可以在 [附录中](#) 找到由 `@DataLdapTest` 启用的自动配置设置的列表.

以下示例显示了正在使用的 `@DataLdapTest` 注解:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;

@DataLdapTest
class ExampleDataLdapTests {

    @Autowired
    private LdapTemplate ldapTemplate;

    //
}

```

内存嵌入式 LDAP 通常非常适合测试,因为它速度快并且不需要安装任何开发人员. 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.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.ldap.DataLdapTest;

@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
class ExampleDataLdapNonEmbeddedTests {

}

```

自动配置的 REST Clients

您可以使用 `@RestClientTest` 注解来测试 REST 客户端. 默认情况下,它会自动配置 Jackson, Gson 和 Jsonb 支持,配置 `RestTemplateBuilder`,并添加对 `MockRestServiceServer` 的支持. 使用 `@RestClientTest` 注解时,不扫描常规 `@Component`,`@ConfigurationProperties` bean.`@EnableConfigurationProperties` 可用于包含 `@ConfigurationProperties` 的 bean.



可以在 [附录中](#)找到由 `@RestClientTest` 启用的自动配置设置的列表.

应该使用 `@RestClientTest` 的 `value` 或 `components` 属性来指定要测试的特定 bean,如以下示例所示:

```

@RestClientTest(RemoteVehicleDetailsService.class)
class ExampleRestClientTest {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    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");
    }

}

```

自动配置的 **Spring REST Docs** 测试

您可以使用 **@AutoConfigureRestDocs** 注解在 **Mock MVC**, **REST Assured** 或 **WebTestClient** 的测试中使用 **Spring REST Docs**. 它消除了 **Spring REST Docs** 中对 **JUnit** 扩展的需求.

@AutoConfigureRestDocs 可用于覆盖默认输出目录 (如果使用 **Maven**, 则为 **target/generated-snippets** 如果使用 **Gradle**, 则为 **build/generated-snippets**) . 它也可以用于配置出现在任何记录的 **URI** 中的主机, 方案和端口.

使用 **Mock MVC** 自动配置的 **Spring REST Docs** 测试

@AutoConfigureRestDocs 自定义 **MockMvc** bean 以使用 **Spring REST Docs**. 您可以使用 **@Autowired** 注入它, 并像通常使用 **Mock MVC** 和 **Spring REST Docs** 一样, 在测试中使用它, 如以下示例所示:

```

import org.junit.jupiter.api.Test;

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.web.servlet.MockMvc;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
class UserDocumentationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    void listUsers() throws Exception {
        this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andDo(document("list-users"));
    }

}

```

如果需要对 Spring REST Docs 配置进行更多控制,而不是 [@AutoConfigureRestDocs](#) 属性提供的控制,则可以使用 [RestDocsMockMvcConfigurationCustomizer](#) bean,如以下示例所示:

```

@TestConfiguration
static class CustomizationConfiguration
    implements RestDocsMockMvcConfigurationCustomizer {

    @Override
    public void customize(MockMvcRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}

```

如果要使用 Spring REST Docs 对参数化输出目录的支持, 可以创建 `RestDocumentationResultHandler` bean. 自动配置使用此结果处理器调用 `alwaysDo`, 从而使每个 `MockMvc` 调用自动生成默认片段. 以下示例显示了定义的 `RestDocumentationResultHandler`:

```
@TestConfiguration(proxyBeanMethods = false)
static class ResultHandlerConfiguration {

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}");
    }

}
```

使用 `WebTestClient` 自动配置的Spring REST Docs测试

`@AutoConfigureRestDocs` 也可以与 `WebTestClient` 一起使用. 您可以使用 `@Autowired` 注入它, 并像通常使用 `@WebFluxTest` 和 Spring REST Docs 一样在测试中使用它, 如以下示例所示:

```

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.test.web.reactive.server.WebTestClient;

import static
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.docume
nt;

@WebFluxTest
@AutoConfigureRestDocs
class UsersDocumentationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void listUsers() {

        this.webTestClient.get().uri("/").exchange().expectStatus().isOk().expectBody()
            .consumeWith(document("list-users"));
    }

}

```

如果需要对 Spring REST Docs 配置进行更多控制,而不是 `@AutoConfigureRestDocs` 属性提供的控制,则可以使用 `RestDocsWebTestClientConfigurationCustomizer` bean,如以下示例所示:

```

@TestConfiguration(proxyBeanMethods = false)
public static class CustomizationConfiguration implements
RestDocsWebTestClientConfigurationCustomizer {

    @Override
    public void customize(WebTestClientRestDocumentationConfigurer configurer) {
        configurer.snippets().withEncoding("UTF-8");
    }

}

```

使用 `RESTAssured` 自动配置的 `Spring REST Docs` 测试

`@AutoConfigureRestDocs` 使一个 `RequestSpecification` Bean (可预配置为使用`Spring REST Docs`) 可用于您的测试. 您可以使用 `@Autowired` 注入它, 并像在使用 `REST Assured` 和 `Spring REST Docs` 时一样, 在测试中使用它, 如以下示例所示:

```
import io.restassured.specification.RequestSpecification;
import org.junit.jupiter.api.Test;

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 static io.restassured.RestAssured.given;
import static org.hamcrest.Matchers.is;
import static
org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.document;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
class UserDocumentationTests {

    @Test
    void listUsers(@Autowired RequestSpecification documentationSpec,
    @LocalServerPort int port) {
        given(documentationSpec).filter(document("list-
users")).when().port(port).get("/").then().assertThat()
            .statusCode(is(200));
    }

}
```

如果您需要对 `Spring REST Docs` 配置进行更多控制, 而不是 `@AutoConfigureRestDocs` 属性所提供的控制, 则可以使用 `RestDocsRestAssuredConfigurationCustomizer` bean, 如以下示例所示:

```
@TestConfiguration(proxyBeanMethods = false)
public static class CustomizationConfiguration implements
RestDocsRestAssuredConfigurationCustomizer {

    @Override
    public void customize(RestAssuredRestDocumentation configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

自动配置的 **Spring Web Services** 测试

您可以使用 `@WebServiceClientTest` 来通过 Spring Web Services 项目测试使用呼叫 Web 服务的应用程序。默认情况下，它配置模拟 `WebServiceServer` bean 并自动自定义 `WebServiceTemplateBuilder`。（有关在 Spring Boot 中结合使用 Web 服务的更多信息，请参阅本章前面的 "[Web Services](#)"。）



可以在 [附录中](#) 找到由 `@WebServiceClientTest` 启用的自动配置设置的列表。

以下示例显示了正在使用的 `@WebServiceClientTest` 注解：

```

@WebServiceClientTest(ExampleWebServiceClient.class)
class WebServiceClientIntegrationTests {

    @Autowired
    private MockWebServiceServer server;

    @Autowired
    private ExampleWebServiceClient client;

    @Test
    void mockServerCall() {
        this.server.expect(payload(new StringSource("<request/>"))).andRespond(
            withPayload(new
StringSource("<response><status>200</status></response>")));
        assertThat(this.client.test()).extracting(Response::getStatus).isEqualTo(200);
    }
}

```

其他的自动配置和切片

每个切片提供一个或多个 `@AutoConfigure...` 注解, 即定义应包含在切片中的自动配置.
可以通过创建自定义 `@AutoConfigure...` 注解来添加其他自动配置, 也可以简单地通过将 `@ImportAutoConfiguration` 添加到测试中来添加其他自动配置, 如以下示例所示:

```

@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration.class)
class ExampleJdbcTests {
}

```



确保不要使用常规的 `@Import` 注解导入自动配置, 因为它们是由 Spring Boot 以特定方式处理的.

或者, 可以通过在 `META-INF/spring.factories` 中注册切片注解的任何使用来添加其他自动配置, 如以下示例所示:

```
org.springframework.boot.test.autoconfigure.jdbc.JdbcTest=com.example.IntegrationAutoConfiguration
```

切片或 `@AutoConfigure...` 注解可以通过这种方式自定义,只要使用 `@ImportAutoConfiguration` 对其进行元注解即可.

用户配置和切片

如果您以合理的方式 `组织代码`,则默认情况下将 `@SpringBootApplication` 类用作测试的配置.

因此,变得重要的是,不要使用特定于其功能特定区域的配置设置来乱扔应用程序的主类.

假设您正在使用 `Spring Batch`,并且依赖于它的自动配置. 您可以如下定义 `@SpringBootApplication`:

```
@SpringBootApplication  
@EnableBatchProcessing  
public class SampleApplication { ... }
```

因为此类是测试的源配置,所以任何切片测试实际上都尝试启动 `Spring Batch`,这绝对不是您想要执行的操作.

建议的方法是将特定于区域的配置移动到与您的应用程序处于同级别的单独的 `@Configuration` 类,如以下示例所示:

```
@Configuration(proxyBeanMethods = false)  
@EnableBatchProcessing  
public class BatchConfiguration { ... }
```



根据您应用程序的复杂性,您可以为您的自定义设置一个 `@Configuration` 类,或者每个域区域一个类.

后一种方法使您可以在其中一个测试中使用 `@Import` 注解启用它.

测试片将 `@Configuration` 类从扫描中排除. 例如,对于 `@WebMvcTest`,以下配置将在测试切片加载的应用程序上下文中不包括给定的 `WebMvcConfigurer` Bean:

```

@Configuration
public class WebConfiguration {
    @Bean
    public WebMvcConfigurer testConfigurer() {
        return new WebMvcConfigurer() {
            ...
        };
    }
}

```

但是,以下配置将导致自定义 `WebMvcConfigurer` 由测试片加载.

```

@Component
public class TestWebMvcConfigurer implements WebMvcConfigurer {
    ...
}

```

混乱的另一个来源是类路径扫描. 假定在以合理的方式组织代码的同时,您需要扫描其他程序包. 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 { ... }

```

这样做有效地覆盖了默认的组件扫描指令,并且具有扫描这两个软件包的副作用,而与您选择的切片无关. 例如,`@DataJpaTest` 似乎突然扫描了应用程序的组件和用户配置. 同样,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,则可以在测试层次结构中的某个位置创建 `@SpringBootConfiguration`,以便代替使用它. 或者,您可以为测试指定一个源,从而禁用查找默认源的行为.

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,则应在应用程序的构建中添加对 Spock 的 `spock-spring` 模块的依赖. `spock-spring` 将 Spring 的测试框架集成到了 Spock 中. 建议您使用 Spock 1.2 或更高版本,以受益于 Spock 的 Spring Framework 和 Spring Boot 集成的许多改进. 有关更多详细信息,请参见 [Spock 的 Spring](#)

模块的文档.

5.26.4. 测试实用工具

一些测试实用工具类通常在测试您的应用程序时有用, 它们被打包为 `spring-boot` 的一部分.

`ConfigFileApplicationContextInitializer`

`ConfigFileApplicationContextInitializer` 是一个 `ApplicationContextInitializer`, 您可以将其应用于测试以加载 Spring Boot `application.properties` 文件. 当不需要 `@SpringBootTest` 提供的全部功能时, 可以使用它, 如以下示例所示:

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



单独使用 `ConfigFileApplicationContextInitializer` 不能提供对 `@Value("${...}")` 注入的支持. 唯一的工作就是确保将 `application.properties` 文件加载到 Spring 的环境中. 为了获得 `@Value` 支持, 您需要另外配置 `PropertySourcesPlaceholderConfigurer` 或使用 `@SpringBootTest`, 后者会为您自动配置一个.

`TestPropertyValues`

使用 `TestPropertyValues`, 可以快速将属性添加到 `ConfigurableEnvironment` 或 `ConfigurableApplicationContext`. 您可以使用 `key=value` 字符串来调用它, 如下所示:

```
TestPropertyValues.of("org=Spring", "name=Boot").applyTo(env);
```

`OutputCapture`

`OutputCapture` 是一个 JUnit 扩展, 可用于捕获 `System.out` 和 `System.err` 输出. 要使用 `add @ExtendWith(OutputCaptureExtension.class)` 并将 `CapturedOutput` 作为参数注入测试类构造函数或测试方法, 如下所示:

```
@ExtendWith(OutputCaptureExtension.class)
class OutputCaptureTests {

    @Test
    void testName(CapturedOutput output) {
        System.out.println("Hello World!");
        assertThat(output).contains("World");
    }

}
```

TestRestTemplate

TestRestTemplate 是 Spring **RestTemplate** 的一种便捷替代方案，在集成测试中非常有用。您可以使用普通模板或发送基本HTTP身份验证（带有用户名和密码）的模板。在这两种情况下，模板都不会通过在服务器端错误上引发异常来以易于测试的方式运行。



Spring Framework 5.0提供了一个新的 **WebTestClient**，可用于[WebFlux集成测试](#) 和 [WebFlux和MVC端到端测试](#)。与 **TestRestTemplate** 不同，它为声明提供了流式的API。

建议（但不是强制性的）使用 Apache HTTP Client（版本4.3.2或更高版本）。如果您在类路径中具有该名称，则 **TestRestTemplate** 会通过适当配置客户端进行响应。如果您确实使用Apache的HTTP客户端，则会启用一些其他易于测试的功能：

- 不支持重定向（因此可以断言响应位置）。
- 忽略 cookie（因此模板是无状态的）。

TestRestTemplate 可以在你的集成测试中直接实例化，如下面的例子所示：

```
public class MyTest {

    private TestRestTemplate template = new TestRestTemplate();

    @Test
    public void testRequest() throws Exception {
        HttpHeaders headers = this.template.getForEntity(
            "https://myhost.example.com/example",
            String.class).getHeaders();
        assertThat(headers.getLocation()).hasHost("other.example.com");
    }

}
```

或者,如果将 `@SpringBootTest` 注解与 `WebEnvironment.RANDOM_PORT` 或 `WebEnvironment.DEFINED_PORT` 一起使用,则可以注入完全配置的 `TestRestTemplate` 并开始使用它. 如有必要,可以通过 `RestTemplateBuilder` bean应用其他定制. 未指定主机和端口的所有 URL 都会自动连接到嵌入式服务器,如以下示例所示:

```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class SampleWebClientTests {

    @Autowired
    private TestRestTemplate template;

    @Test
    void testRequest() {
        HttpHeaders headers = this.template.getForEntity("/example",
String.class).getHeaders();
        assertThat(headers.getLocation()).hasHost("other.example.com");
    }

    @TestConfiguration(proxyBeanMethods = false)
    static class Config {

        @Bean
        RestTemplateBuilder restTemplateBuilder() {
            return new
        RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                            .setReadTimeout(Duration.ofSeconds(1));
        }

    }
}

```

5.27. WebSockets

Spring Boot 为内嵌式 Tomcat、Jetty 和 Undertow 提供了 WebSocket 自动配置。如果将 war 文件部署到独立容器，则 Spring Boot 假定容器负责配置其 WebSocket 支持。

Spring Framework 为 MVC Web 应用程序提供了 [丰富的 WebSocket 支持](#)，可以通过 `spring-boot-starter-websocket` 模块轻松访问。

WebSocket 支持也可用于 [响应式 Web 应用程序](#)，并且引入 WebSocket API 以及 `spring-boot-starter-webflux`:

```

<dependency>
    <groupId>javax.websocket</groupId>
    <artifactId>javax.websocket-api</artifactId>
</dependency>

```

5.28. Web Services

Spring Boot 提供 Web Service 自动配置,因此您要做的就是定义 Endpoints.

可以使用 `spring-boot-starter-webservices` 模块轻松访问 Spring Web Services 功能.

可以分别为 WSDL 和 XSD 自动创建 `SimpleWSDL11Definition` 和 `SimpleXsdSchema` bean. 为此,请配置其位置,如下所示:

```
spring:  
  webservices:  
    wsdl-locations: "classpath:/wsdl"
```

5.28.1. 使用 `WebServiceTemplate` 调用 Web Service

如果您需要从应用程序调用远程 Web 服务,则可以使用 `WebServiceTemplate` 类. 由于 `WebServiceTemplate` 实例在使用之前通常需要进行自定义,因此 Spring Boot 不提供任何自动配置的 `WebServiceTemplate` bean. 但是,它会自动配置 `WebServiceTemplateBuilder`,可在需要创建 `WebServiceTemplate` 实例时使用.

以下代码为一个典型示例:

```
@Service  
public class MyService {  
  
    private final WebServiceTemplate webServiceTemplate;  
  
    public MyService(WebServiceTemplateBuilder webServiceTemplateBuilder) {  
        this.webServiceTemplate = webServiceTemplateBuilder.build();  
    }  
  
    public DetailsResp someWsCall(DetailsReq detailsReq) {  
        return (DetailsResp)  
this.webServiceTemplate.marshalSendAndReceive(detailsReq, new  
SoapActionCallback(ACTION));  
    }  
}
```

默认情况下, `WebServiceTemplateBuilder` 使用 `classpath` 上的可用 HTTP 客户端库检测合适的基于 HTTP 的 `WebServiceMessageSender`.

您还可以按如下方式自定义读取和连接的超时时间:

```
@Bean  
public WebServiceTemplate webServiceTemplate(WebServiceTemplateBuilder builder)  
{  
    return builder.messageSenders(new HttpWebServiceMessageSenderBuilder()  
        .setConnectTimeout(5000).setReadTimeout(2000).build()).build();  
}
```

5.29. 创建自己的自动配置

如果您在公司负责开发公共类库, 或者如果您在开发一个开源或商业库, 您可能希望开发自己的自动配置. 自动配置类可以捆绑在外部 `jar` 中, 他仍然可以被 `Spring Boot` 获取.

自动配置可以与提供自动配置代码的 `starter` 以及您将使用的类库库相关联.

我们首先介绍构建自己的自动配置需要了解的内容, 然后我们将继续介绍创建 [自定义 starter](#) 所需的步骤.



这里有一个 [演示项目](#) 展示了如何逐步创建 `starter`.

5.29.1. 理解 自动配置的 Beans

在内部, 自动配置使用了标准的 `@Configuration` 类来实现. `@Conditional` 注解用于约束何时应用自动配置. 通常, 自动配置类使用 `@ConditionalOnClass` 和 `@ConditionalOnMissingBean` 注解. 这可确保仅在找到相关类时以及未声明您自己的 `@Configuration` 时才应用自动配置.

您可以浏览 `spring-boot-autoconfigure` 的源代码, 以查看 Spring 提供的 `@Configuration` 类 (请参阅 `META-INF/spring.factories` 文件) .

5.29.2. 找到候选的自动配置

Spring Boot 会检查已发布 `jar` 中是否存在 `META-INF/spring.factories` 文件. 该文件应列出 `EnableAutoConfiguration` key 下的配置类, 如下所示:

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



必须以这种方式加载自动配置。确保它们在特定的包空间中定义，并且它们不能是组件扫描的目标。此外，自动配置类不应启用组件扫描以查找其他组件。应该使用特定的 `@Imports` 来代替。

如果需要按特定顺序应用配置，则可以使用 `@AutoConfigureAfter` 或 `@AutoConfigureBefore` 注解。例如，如果您提供特定于 Web 的配置，则可能需要在 `WebMvcAutoConfiguration` 之后应用您的类。

如果您想排序某些不应该彼此直接了解的自动配置，您也可以使用 `@AutoConfigureOrder`。该注解与常规 `@Order` 注解有相同的语义，但它为自动配置类提供了专用顺序。

与标准的 `@Configuration` 类一样，自动配置类的应用顺序仅会影响其 `bean` 的定义顺序。随后创建这些 `bean` 的顺序不受影响，并由每个 `bean` 的依赖关系和任何 `@DependsOn` 关系确定。

5.29.3. 条件注解

您几乎总希望在自动配置类中包含一个或多个 `@Conditional` 注解。

`@ConditionalOnMissingBean` 是一个常用的注解，其允许开发人员在对您的默认值不满意用于覆盖自动配置。

Spring Boot 包含许多 `@Conditional` 注解，您可以通过注解 `@Configuration` 类或单独的 `@Bean` 方法在您自己的代码中复用它们。这些注解包括：

- [类条件](#)
- [Bean 条件](#)
- [属性条件](#)
- [资源条件](#)
- [Web 应用程序条件](#)

- [SpEL 表达式条件](#)

类条件

`@ConditionalOnClass` 和 `@ConditionalOnMissingClass`

注解允许根据特定类的是否存在来包含 `@Configuration` 类。由于使用 `ASM` 解析注解元数据，您可以使用 `value` 属性来引用真实类，即使该类实际上可能不会出现在正在运行的应用程序的 `classpath` 中。如果您希望使用 `String` 值来指定类名，也可以使用 `name` 属性。

此机制不会以相同的方式应用于返回类型是条件的目标的 `@Bean` 方法：

在方法上的条件应用之前，`JVM` 将加载类和可能处理的方法引用，如果找不到类，将发生失败。

要处理这种情况，可以使用单独的 `@Configuration` 类来隔离条件，如下所示：

```
@Configuration(proxyBeanMethods = false)
// Some conditions
public class MyAutoConfiguration {

    // Auto-configured beans

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(EmbeddedAcmeService.class)
    static class EmbeddedConfiguration {

        @Bean
        @ConditionalOnMissingBean
        public EmbeddedAcmeService embeddedAcmeService() { ... }

    }
}
```



如果使用 `@ConditionalOnClass` 或

`@ConditionalOnMissingClass`

作为元注解的一部分来组成自己的组合注解，则必须使用 `name` 来引用类，在这种情况下将不作处理。

Bean 条件

`@ConditionalOnBean` 和 `@ConditionalOnMissingBean` 注解允许根据特定 bean 是否存在来包含 bean. 您可以使用 `value` 属性按类型或使用 `name` 来指定 bean. `search` 属性允许您限制在搜索 bean 时应考虑的 `ApplicationContext` 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, 如下所示:

```
@Configuration(proxyBeanMethods = false)
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public MyService myService() { ... }

}
```

在前面的示例中, 如果 `ApplicationContext` 中不包含 `MyService` 类型的 bean, 则将创建 `myService` bean.



您需要非常小心地添加 bean 定义的顺序, 因为这些条件是根据到目前为止已处理的内容进行计算的. 因此, 我们建议在自动配置类上仅使用 `@ConditionalOnBean` 和 `@ConditionalOnMissingBean` 注解 (因为这些注解保证在添加所有用户定义的 bean 定义后加载) .



`@ConditionalOnBean` 和 `@ConditionalOnMissingBean` 不会阻止创建 `@Configuration` 类. 在类级别使用这些条件并使用注解标记每个包含 `@Bean` 方法的唯一区别是, 如果条件不匹配, 前者会阻止将 `@Configuration` 类注册为 bean.



声明 `@Bean` 方法时, 请在该方法的返回类型中提供尽可能多的类型信息. 例如, 如果您的 bean 的具体类实现一个接口, 则 bean 方法的返回类型应该是具体的类而不是接口. 使用 bean 条件时, 在 `@Bean` 方法中提供尽可能多的类型信息尤为重要, 因为它们的评估只能依靠方法签名中可用的类型信息.

属性条件

`@ConditionalOnProperty` 注解允许基于 Spring Environment 属性包含配置。使用 `prefix` 和 `name` 属性指定需要检查的属性。默认情况下，匹配存在且不等于 `false` 的所有属性。您还可以使用 `havingValue` 和 `matchIfMissing` 属性创建更高级的检查。

资源条件

`@ConditionalOnResource` 注解仅允许在存在特定资源时包含配置。可以使用常用的 Spring 约定来指定资源，如下所示：`file:/home/user/test.dat`。

Web 应用程序条件

`@ConditionalOnWebApplication` 和 `@ConditionalOnNotWebApplication` 注解在应用程序为 Web 应用程序的情况下是否包含配置。Web 应用程序是使用 Spring `WebApplicationContext`，定义一个 `session` 作用域或具有 `StandardServletEnvironment` 的任何应用程序。

通过 `@ConditionalOnWarDeployment` 注解，可以根据应用程序是否是已部署到容器的传统 WAR 应用程序进行配置。对于嵌入式服务器运行的应用程序，此条件将不匹配。

SpEL 表达式条件

`@ConditionalOnExpression` 注解允许根据 SpEL 表达式的结果包含配置。

5.29.4. 测试自动配置

自动配置可能受许多因素的影响：用户配置（`@Bean` 定义和 `Environment` 自定义）、条件评估（存在特定的类库）等。具体而言，每个测试都应该创建一个定义良好的 `ApplicationContext`，它表示这些自定义的组合。`ApplicationContextRunner` 提供了一个好的实现方法。

`ApplicationContextRunner` 通常被定义为测试类的一个字段，用于收集基本的通用配置。以下示例确保始终调用 `UserServiceAutoConfiguration`：

```
private final ApplicationContextRunner contextRunner = new  
ApplicationContextRunner()  
  
.withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class));
```



如果必须定义多个自动配置，则无需按照与运行应用程序时完全相同的顺序调用它们的声明。

每个测试都可以使用 `runner` 来表示特定的用例。例如，下面的示例调用用户配置 (`UserConfiguration`) 并检查自动配置是否正确退回。调用 `run` 提供了一个可以与 `AssertJ` 一起使用的回调上下文。

```
@Test  
void defaultServiceBacksOff() {  
  
    this.contextRunner.withUserConfiguration(UserConfiguration.class).run((context)  
-> {  
        assertThat(context).hasSingleBean(UserService.class);  
  
        assertThat(context.getBean("myUserService")).isSameAs(context.getBean(UserService.class));  
    });  
}  
  
@Configuration(proxyBeanMethods = false)  
static class UserConfiguration {  
  
    @Bean  
    UserService myUserService() {  
        return new UserService("mine");  
    }  
}
```

也可以轻松自定义 `Environment`，如下所示：

```

@Test
void serviceNameCanBeConfigured() {
    this.contextRunner.withPropertyValues("user.name=test123").run((context) ->
{
    assertThat(context).hasSingleBean(UserService.class);

    assertThat(context.getBean(UserService.class).getName()).isEqualTo("test123");
});
}

```

`runner` 还可用于展示 `ConditionEvaluationReport`. 报告可以在 `INFO` 或 `DEBUG` 级别下打印. 以下示例展示如何使用 `ConditionEvaluationReportLoggingListener` 在自动配置测试中打印报表.

```

@Test
void autoConfigTest() {
    ConditionEvaluationReportLoggingListener initializer = new
ConditionEvaluationReportLoggingListener(
    LogLevel.INFO);
    ApplicationContextRunner contextRunner = new ApplicationContextRunner()
        .withInitializer(initializer).run((context) -> {
            // Do something...
        });
}

```

模拟一个 Web 上下文

如果需要测试一个仅在 `Servlet` 或响应式 `Web` 应用程序上下文中运行的自动配置 , 请分别使用 `WebApplicationContextRunner` 或 `ReactiveWebApplicationContextRunner`.

覆盖 Classpath

还可以测试在运行时不存在特定类和/或包时发生的情况. `Spring Boot` 附带了一个可以由跑步者轻松使用的 `FilteredClassLoader`. 在以下示例中 , 我们声明如果 `UserService` 不存在, 则会正确禁用自动配置:

```

@Test
void serviceIsIgnoredIfLibraryIsNotPresent() {
    this.contextRunner.withClassLoader(new
FilteredClassLoader(UserService.class))
    .run((context) ->
assertThat(context).doesNotHaveBean("userService"));
}

```

5.29.5. 创建自己的 Starter

一个典型的 Spring Boot 启动器包含用于自动配置和使用的基础技术结构的代码, 我们称其为 "acme". 为了使其易于扩展, 可以将命名空间中的许多配置项暴露给环境. 最后, 提供了一个 "starter" 依赖, 以帮助用户尽可能轻松地入门.

具体而言, 自定义启动器可以包含以下内容:

- `autoconfigure` 模块, 为 "acme" 包含自动配置代码.
- `starter` 模块, 它为 "acme" 提供对 `autoconfigure` 模块依赖以及类库和常用的其他依赖. 简而言之, 添加 `starter` 应该提供该库开始使用所需的一切依赖.

完全没有必要将这两个模块分开. 如果 "acme" 具有多种功能, 选项或可选功能, 则最好将自动配置分开, 这样您可以清楚地表示某些功能是可选的. 此外, 您还可以制作一个启动器, 以提供有关可选的依赖.

同时, 其他人只能依靠 `autoconfigure` 模块来制作自己的具有不同选项的启动器.

如果自动配置相对简单并且不具有可选功能, 则将两个模块合并在启动器中绝对是一种选择.

命名

您应该确保为您的 `starter` 提供一个合适的命名空间. 即使您使用其他 Maven `groupId`, 也不要使用 `spring-boot` 作为模块名称的开头. 我们可能会为您以后自动配置的内容提供官方支持.

根据经验, 您应该在 `starter` 后命名一个组合模块. 例如, 假设您正在为 acme 创建一个 `starter`, 并且您将自动配置模块命名为 `acme-spring-boot`, 将 `starter` 命名为 `acme-spring-boot-starter`. 如果您只有一个组合这两者的模块, 请将其命名为 `acme-spring-`

`boot-starter.`

配置 keys

此外,如果您的 `starter` 提供配置 `key`,请为它们使用唯一的命名空间. 尤其是,不要将您的 `key` 包含在 Spring Boot 使用的命名空间中(例如 `server`、`management`、`spring` 等). 如果您使用了相同的命名空间,我们将来可能会以破坏您的模块的方式来修改这些命名空间. 根据经验,所有 `key` 都必须拥有自己的命名空间(例如 `acme`) .

确保触发元数据生成,以便为您的 `key` 提供 IDE 帮助. 您可能想查看生成的元数据 (`META-INF/spring-configuration-metadata.json`) 以确保您的 `key` 记录是否正确.

通过为每个属性添加字段 `javadoc` 来确保记录了配置 `keys`,如以下示例所示:

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

    /**
     * Whether to check the location of acme resources.
     */
    private boolean checkLocation = true;

    /**
     * Timeout for establishing a connection to the acme server.
     */
    private Duration loginTimeout = Duration.ofSeconds(3);

    // getters & setters

}
```



您仅应将简单文本与 `@ConfigurationProperties` 字段 Javadoc 一起使用,因为在将它们添加到 JSON 之前不会对其进行处理.

这是我们内部遵循的一些规则,以确保描述一致:

- 请勿以 "The" 或 "A" 头描述.
- 对于布尔类型,请从 "Whether" 或 "Enable" 开始描述.
- 对于基于集合的类型,请以 "以逗号分隔的列表" 开始描述

- 使用 `java.time.Duration` 而不是 `long`, 如果它不等于毫秒, 请说明默认单位, 例如 "如果未指定持续时间后缀, 则将使用秒".
- 除非必须在运行时确定默认值, 否则请不要在描述中提供默认值.

确保 [触发元数据生成](#), 以便为您的 `key` 提供 IDE 帮助. . 您可能需要查看生成的元数据 (`META-INF/spring-configuration-metadata.json`), 以确保您的 `key` 记录是否正确. 在兼容的IDE中使用自己的 `starter` 也是验证元数据质量的好主意.

autoconfigure 模块

`autoconfigure` 模块包含类库开始使用所需的所有内容. 它还可以包含配置 `key` 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.



您应该将类库的依赖标记为可选, 以便您可以更轻松地在项目中包含 `autoconfigure` 模块. 如果以这种方式执行, 则不提供类库, 默认情况下, Spring Boot 将会退出.

Spring Boot 使用注解处理器来收集元数据文件 (`META-INF/spring-autoconfigure-metadata.properties`) 中自动配置的条件. 如果该文件存在, 则用于快速过滤不匹配的自动配置, 缩短启动时间.

建议在包含自动配置的模块中添加以下依赖:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure-processor</artifactId>
    <optional>true</optional>
</dependency>
```

如果您直接在应用程序中定义了自动配置, 请确保配置 `spring-boot-maven-plugin`, 以防止 `repackage` 目标将依赖添加到 `fat jar` 中:

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <excludes>
            <exclude>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-autoconfigure-
processor</artifactId>
            </exclude>
          </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

使用 Gradle 4.5 及更早版本时, 应在 `compileOnly` 配置中声明依赖, 如下所示:

```

dependencies {
  compileOnly "org.springframework.boot:spring-boot-autoconfigure-processor"
}

```

使用 Gradle 4.6 或者更高的版本, 应在 `annotationProcessor` 配置中声明依赖, 如下所示:

```

dependencies {
  annotationProcessor "org.springframework.boot:spring-boot-autoconfigure-
processor"
}

```

Starter 模块

`starter` 真的是一个空 `jar`. 它的唯一目的是为使用类库提供必要的依赖. 您可以将其视为使用类库的一切基础.

不要对添加 `starter` 的项目抱有假设想法. 如果您自动配置的库经常需要其他 `starter`, 请一并声明它们. 如果可选依赖的数量很多, 则提供一组适当的默认依赖可能很难

,因为您本应该避免包含对常用库的使用不必要的依赖. 换而言之,您不应该包含可选的依赖.



无论哪种方式,您的 `starter` 必须直接或间接引用 Spring Boot 的 `core starter (spring-boot-starter)` (如果您的 `starter` 依赖于另一个 `starter`,则无需添加它). 如果只使用自定义 `starter` 创建项目,则 Spring Boot 的核心功能将通过 `core starter` 来实现.

5.30. Kotlin 支持

Kotlin 是一种针对 JVM (和其他平台) 的静态类型语言,它可编写出简洁而优雅的代码,同时提供与使用 Java 编写的现有库的 [互通性](#).

Spring Boot 通过利用其他 Spring 项目 (如 Spring Framework、Spring Data 和 Reactor) 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](#).

开始学习 Spring Boot 和 Kotlin 最简单方法是遵循这个 [全面教程](#). 您可以通过 start.spring.io 创建新的 Kotlin 项目. 如果您需要支持,请免费加入 [Kotlin Slack](#) 的 `#spring` 频道或使用 Stack Overflow 上的 `spring` 和 `kotlin` 标签提问.

5.30.1. 要求

Spring Boot 支持 Kotlin 1.3.x. 要使用 Kotlin,classpath 下必须存在 `org.jetbrains.kotlin:kotlin-stdlib` 和 `org.jetbrains.kotlin:kotlin-reflect`. 也可以使用 `kotlin-stdlib` 的变体 `kotlin-stdlib-jdk7` 和 `kotlin-stdlib-jdk8`.

由于 Kotlin 类默认为 `final`,因此您可能需要配置 `kotlin-spring` 插件以自动打开 `Spring-annotated` 类,以便可以代理它们.

在 Kotlin 中序列化/反序列化 JSON 数据需要使用 [Jackson](#) 的 [Kotlin 模块](#). 在 classpath 中找到它时会自动注册. 如果 Jackson 和 Kotlin 存在但 Jackson Kotlin 模块不存在,则会记录警告消息.



如果在 start.spring.io 上创建 Kotlin 项目,则默认提供这些依赖和插件.

5.30.2. Null 安全

Kotlin 的一个关键特性是 [null-safety](#). 它在编译时处理空值, 而不是将问题推迟到运行时并遇到 [NullPointerException](#).

这有助于消除常见的错误来源, 而无需支付像 [Optional](#) 这样的包装器的成本. Kotlin 还允许使用有可空值的, 如 [Kotlin null 安全综合指南](#) 中所述.

虽然 Java 不允许在其类型系统中表示 [null](#) 安全, 但 Spring Framework、Spring Data 和 Reactor 现在通过易于使用的工具的注解提供其 API 的安全性. 默认情况下, Kotlin 中使用的 Java API 类型被识别为放宽空检查的 [平台类型](#). Kotlin 对 [JSR 305](#) 注解的支持与可空注解相结合, 为 Kotlin 中 Spring API 相关的代码提供了空安全.

可以通过使用以下选项添加 [-Xjsr305](#) 编译器标志来配置 JSR 305 检查:

[-Xjsr305={strict|warn|ignore}](#). 默认行为与 [-Xjsr305=warn](#) 相同. 在从 Spring API 推断出的 Kotlin 类型中需要考虑 [null](#) 安全的 [strict](#) 值, 但是应该使用 Spring API 可空声明甚至可以在次要版本之间发展并且将来可能添加更多检查的方案.



尚不支持泛型类型参数、`varargs` 和数组元素可空性. 有关最新信息, 请参见 [SPR-15942](#). 另请注意, Spring Boot 自己的 API 尚未注解.

5.30.3. Kotlin API

runApplication

Spring Boot 提供了使用 [runApplication<MyApplication>\(*args\)](#) 运行应用程序的惯用方法, 如下所示:

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

这是 [SpringApplication.run\(MyApplication::class.java, *args\)](#) 的替代方式.

它还允许自定义应用程序,如下所示:

```
runApplication<MyApplication>(*args) {  
    setBannerMode(OFF)  
}
```

扩展

Kotlin 扩展 提供了使用附加功能扩展现有类的能力. Spring Boot Kotlin API 利用这些扩展为现有 API 添加新的 Kotlin 特定便利.

提供的 `TestRestTemplate` 扩展类似于 Spring Framework 为 `RestOperations` 提供的. 除此之外,扩展使得利用 Kotlin `reified` 类型参数变为可能.

5.30.4. 依赖管理

为了避免在 `classpath` 上混合不同版本的 Kotlin 依赖, Spring Boot 会导入 Kotlin BOM.

使用 Maven,可以通过 `kotlin.version` 属性自定义Kotlin版本,并且为 `kotlin-maven-plugin` 提供了插件管理. 使用 Gradle, Spring Boot 插件会自动将 `kotlin.version` 与 Kotlin 插件的版本保一致.

Spring Boot 还通过导入 Kotlin Coroutines BOM 管理Coroutines依赖的版本. 可以通过 `kotlin-coroutines.version` 属性自定义版本.



如果在 `start.spring.io` 上构建的 Kotlin 项目有至少一个响应式依赖,则默认提供 `org.jetbrains.kotlinx:kotlinx-coroutines-reactor` 依赖.

5.30.5. @ConfigurationProperties

`@ConfigurationProperties` 目前仅适用于 `lateinit` 或可空的 `var` 属性 (建议使用前者),因为尚不支持由构造函数初始化的不可变类. 与 `@ConstructorBinding` 结合使用时, `@ConfigurationProperties` 支持具有不变 `val` 属性的类,如以下示例所示:

```

@ConstructorBinding
@ConfigurationProperties("example.kotlin")
data class KotlinExampleProperties(
    val name: String,
    val description: String,
    val myService: MyService) {

    data class MyService(
        val apiToken: String,
        val uri: URI
    )
}

```



为了使用注解处理器生成自己的元数据，应该使用 `spring-boot-configuration-processor` 依赖配置 `kapt`。请注意，由于 `kapt` 提供的模型的限制，某些功能（例如检测默认值或不推荐使用的项目）无法正常工作。

5.30.6. 测试

虽然可以使用 JUnit 4 来测试 Kotlin 代码，但建议使用 JUnit 5。JUnit 5 允许测试类实例化一次，并在所有类的测试中复用。这使得可以在非静态方法上使用 `@BeforeAll` 和 `@AfterAll` 注解，这非常适合 Kotlin。

要模拟 Kotlin 类，建议使用 MockK。如果您需要与 Mockito 特定的 `@MockBean` 和 `@SpyBean` 注解相对应的 Mockk，则可以使用 SpringMockK，它提供类似的 `@MockkBean` 和 `@SpykBean` 注解。

要使用 JUnit 5，请从 `spring-boot-starter-test` 中排除 `junit:junit` 依赖，然后添加 JUnit 5 依赖，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 `per-class`。

为了模拟 Kotlin 类，建议使用 Mockk。如果需要 Mockito 特定的 `@MockBean` 和 `@SpyBean` 注解，则可以使用 SpringMockK，它提供类似的 `@MockkBean` 和 `@SpykBean` 注解。

5.30.7. 资源

进阶阅读

- [Kotlin 语言参考](#)
- [Kotlin Slack](#) (有专用的 #spring 频道)
- [Stackoverflow 上 spring 和 kotlin 标签](#)
- [在浏览器中尝试 Kotlin](#)
- [Kotlin 博客](#)
- [Awesome Kotlin](#)
- 教程：使用 Spring Boot 和 Kotlin 构建 Web 应用程序
- 使用 Kotlin 开发 Spring Boot 应用程序
- 使用 Kotlin、Spring Boot 和 PostgreSQL 开发地理信息
- 在 Spring Framework 5.0 中引入 Kotlin 支持
- [Spring Framework 5 Kotlin API 实现函数式](#)

示例

- [spring-boot-kotlin-demo](#): 常规的 Spring Boot + Spring Data JPA 项目
- [mixit](#): Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB
- [spring-kotlin-fullstack](#): WebFlux Kotlin 全栈示例, 在前端使用 Kotlin2js 代替 JavaScript 和 TypeScript
- [spring-petclinic-kotlin](#): Spring PetClinic 示例应用的 Kotlin 版本
- [spring-kotlin-deepdive](#): 将 Boot 1.0 + Java 逐步迁移到 Boot 2.0 + Kotlin
- [spring-boot-coroutines-demo](#): Coroutines 示例程序

5.31. 容器镜像

可以很容易地将 Spring Boot 的 fat jar 打包为 docker 镜像。但是, 像在 docker

镜像中一样,复制和运行 `fat jar` 还有很多弊端. 在不打包的情况下运行 `fat jar` 时,总会有一定的开销,在容器化环境中,这很明显. 另一个问题是,将应用程序的代码及其所有依赖放在 `Docker` 镜像的一层中是次优的. 由于重新编译代码的频率可能比升级所用 `Spring Boot` 的版本的频率高,因此最好将代码分开一些. 如果将 `jar` 文件放在应用程序类之前的层中,则 `Docker` 通常只需要更改最底层即可从其缓存中获取其他文件.

5.31.1. 分层 `Docker` 镜像

为了使创建的 `Docker` 镜像更加容易,`Spring Boot` 支持在 `jar` 中添加一个层索引文件. 它提供了层的列表以及应包含在其中的 `jar` 的各个部分. 索引中的层列表是根据应将层添加到 `Docker/OCI` 镜像的顺序来排序的. 现成的,支持以下层:

- `dependencies` (用于常规发布的依赖关系)
- `spring-boot-loader` (适用于 `org/springframework/boot/loader` 下的所有内容)
- `snapshot-dependencies` (用于快照依赖关系)
- `application` (用于应用程序类和资源)

下面显示了 `layers.idx` 文件的示例:

```
- "dependencies":
- BOOT-INF/lib/library1.jar
- BOOT-INF/lib/library2.jar
- "spring-boot-loader":
- org/springframework/boot/loader/JarLauncher.class
- org/springframework/boot/loader/jar/JarEntry.class
- "snapshot-dependencies":
- BOOT-INF/lib/library3-SNAPSHOT.jar
- "application":
- META-INF/MANIFEST.MF
- BOOT-INF/classes/a/b/C.class
```

此分层旨在根据应用程序构建之间更改的可能性来分离代码.

库代码不太可能在内部版本之间进行更改,因此将其放置在自己的层中,以允许工具重新使用缓存中的层. 应用程序代码更可能在内部版本之间进行更改,因此将其隔离在单独的层中.

对于 Maven, 请参阅 [packaging layered jars section](#), 以获取有关在 jar 中添加层索引的更多详细信息. 对于 Gradle, 请参阅 Gradle 插件文档的 [packaging layered jars section](#).

5.31.2. 构建容器镜像

Spring Boot 应用程序可以 使用 Dockerfiles 进行容器化, 也可以使用 Cloud Native Buildpacks 创建可以在任何地方运行的兼容 Docker 容器镜像.

Dockerfiles

虽然可以在 Dockerfile 中仅几行就将 Spring Boot jar 转换为 docker 镜像, 但我们将使用 [分层功能](#) 来创建优化的 docker 镜像. 当您创建一个包含 layers 索引文件的 jar 时, `spring-boot-jarmode-layertools` jar 将作为依赖添加到您的 jar 中. 将此 jar 放在类路径上, 您可以在特殊模式下启动应用程序, 该模式允许引导代码运行与应用程序完全不同的内容, 例如, 提取层的内容.



`layertools` 模式不能与包含启动脚本的完全可执行的 `fully executable Spring Boot archive` 一起使用. 在构建一个打算与 `layertools` 一起使用的 jar 文件时, 禁用启动脚本配置.

您可以通过以下方式使用 `layertools` jar 模式启动 jar:

```
$ java -Djarmode=layertools -jar my-app.jar
```

以下输出:

```
Usage:  
java -Djarmode=layertools -jar my-app.jar  
  
Available commands:  
list      List layers from the jar that can be extracted  
extract   Extracts layers from the jar for image creation  
help      Help about any command
```

`extract` 命令可用于轻松地将应用程序拆分为多个层, 以添加到 `dockerfile` 中. 这是一个使用 `jarmode` 的 Dockerfile 的示例.

```

FROM adoptopenjdk:11-jre-hotspot as builder
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layer-tools -jar application.jar extract

FROM adoptopenjdk:11-jre-hotspot
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]

```

假设上述 `Dockerfile` 位于当前目录中，则可以使用 `docker build`. 生成您的 `docker` 镜像，或者可以选择指定应用程序 `jar` 的路径，如以下示例所示：

```
docker build --build-arg JAR_FILE=path/to/myapp.jar .
```

这是一个多阶段的 `Dockerfile`. 构建器阶段提取以后需要的目录。每个 `COPY` 命令都与 `jarmode` 提取的层有关。

当然，无需使用 `jarmode` 即可编写 `Dockerfile`. 您可以使用 `unzip` 和 `mv` 的某种组合将内容移至正确的层，而 `jarmode` 简化了这一点.

Cloud Native Buildpacks

`Dockerfiles` 只是构建 `Docker` 镜像的一种方式。构建 `docker` 镜像的另一种方法是直接从您的 `Maven` 或 `Gradle` 插件中使用 `buildpacks`。
如果您曾经使用过 `Cloud Foundry` 或 `Heroku` 等应用程序平台，那么您可能已经使用了一个 `buildpack`. `Buildpacks` 是平台的一部分
, 可接收您的应用程序并将其转换为平台可以实际运行的内容。例如, `Cloud Foundry` 的 `Java buildpack` 将注意到您正在推送 `.jar` 文件并自动添加相关的 `JRE`.

借助 `Cloud Native Buildpacks`, 您可以创建可在任何地方运行的 `Docker` 兼容镜像。
`Spring Boot` 直接支持 `Maven` 和 `Gradle` 的 `buildpack`.
这意味着您只需输入一个命令，即可将明智的镜像快速地导入本地运行的 `Docker` 守护程序。

有关如何在 `Maven` 和 `Gradle` 中使用 `buildpack` 的信息，请参阅各个插件文档。



Paketo Spring Boot buildpack 也已更新,以支持 `layers.idx` 文件,因此,对其应用的任何自定义都将反映在 buildpack 创建的镜像中.



为了实现可重复构建和容器映像缓存, Buildpacks 可以操作应用程序资源元数据(例如文件 "last modified" 信息)。您应该确保应用程序在运行时不依赖于该元数据。Spring Boot 可以在提供静态资源时使用这些信息,但是可以用 `configprop:spring.web.resources.cache.use-last-modified[]` 禁用这些信息.

5.32. 下一步

如果您想了解本节中讨论的任何类目的更多信息,可以查看 [Spring Boot API 文档](#),也可以直接浏览 [源代码](#) . 如果您有具体问题,请查看 [how-to](#) 部分.

如果您对 Spring Boot 的核心功能感到满意,可以继续阅读有关[生产就绪功能](#)的内容.

Chapter 6. Spring Boot Actuator: 生产就绪功能

Spring Boot 包含许多其他功能，可帮助你在将应用程序推送到生产环境时监控和管理应用程序。你可以选择使用 HTTP 端点或 JMX 来管理和监控应用程序。审计、健康和指标收集也可以自动应用于你的应用程序。

每个单独的端点都可以 [启用或关闭](#) 和 [通过 HTTP 或 JMX 暴露 \(made remotely accessible\)](#)。当端点同时启用和暴露时，则视为可用。

内置端点只有在可用时才会自动配置。大多数应用程序选择通过 HTTP 进行暴露，在暴露中，`endpoint` 的 ID 与 `/actuator` 的前缀一起映射到 URL。例如，默认情况下，`health` 端点映射到 `/actuator/health`。



想了解有关 Actuator 端点及其请求和响应格式的更多信息，请参阅单独的 API 文档 ([HTML](#) 或 [PDF](#))。

6.1. 启用生产就绪功能

`spring-boot-actuator` 模块提供了 Spring Boot 的所有生产就绪功能。

启用这些功能的推荐的方法是添加 `spring-boot-starter-actuator` starter 到依赖中。

Actuator 的定义

Actuator 是制造术语，指的是用于移动或控制某物的机械装置。Actuator 可以通过一个小的变化产生大量的运动。

要将 `actuator` 添加到基于 Maven 的项目，请添加以下 `starter` 依赖：

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

对于 Gradle, 请使用以下声明:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
}
```

6.2. 端点

通过 Actuator 端点, 你可以监控应用程序并与之交互。Spring Boot 包含许多内置端点, 也允许你添加自己的端点。例如, `health` 端点提供基本的应用程序健康信息。

可以 [启用或禁用](#) 每个端点。它可以控制当其 bean 存在于应用程序上下文中是否创建端点。当端点同时启用和公开时, 它被视为可用。内置端点只有在可用时才会被自动配置。

要进行远程访问, 必须通过 [JMX 或 HTTP 暴露端点](#)。大多数应用程序选择 HTTP 方式, 通过端点的 ID 以及 `/actuator` 的前缀映射一个 URL。例如, 默认情况下, `health` 端点映射到 `/actuator/health`。

可以使用以下与技术无关的端点:

ID	描述
<code>auditevents</code>	暴露当前应用程序的审计事件信息。需要一个 <code>AuditEventRepository</code> bean.
<code>beans</code>	显示应用程序中所有 Spring bean 的完整列表。
<code>caches</code>	暴露可用的缓存。
<code>conditions</code>	显示在配置和自动配置类上评估的条件以及它们匹配或不匹配的原因。
<code>configprops</code>	显示所有 <code>@ConfigurationProperties</code> 的校对清单。
<code>env</code>	暴露 Spring <code>ConfigurableEnvironment</code> 中的属性。
<code>flyway</code>	显示已应用的 Flyway 数据库迁移, 需要一个或多个 <code>Flyway</code> beans.
<code>health</code>	显示应用程序健康信息
<code>httptrace</code>	显示 HTTP 追踪信息 (默认情况下, 最后 100 个 HTTP 请求 / 响应交换)。需要一个 <code>HttpTraceRepository</code> bean

ID	描述
info	显示应用程序信息.
integrationgraph	显示 Spring Integration 图. 需要依赖 <code>spring-integration-core</code>
loggers	显示和修改应用程序中日志记录器的配置.
liquibase	显示已应用的 Liquibase 数据库迁移. 需要一个或多个 <code>Liquibase beans</code> .
metrics	显示当前应用程序的指标指标信息.
mappings	显示所有 <code>@RequestMapping</code> 路径的整理清单.
scheduledtasks	显示应用程序中的调度任务.
sessions	允许从 <code>Spring Session</code> 支持的会话存储中检索和删除用户会话. 当使用 <code>Spring Session</code> 的响应式 Web 应用程序支持时不可用.
shutdown	正常关闭应用程序. 默认禁用
startup	显示由 <code>ApplicationStartup</code> 收集到的 启动步骤数据 . 需要使用 <code>BufferingApplicationStartup</code> 配置 <code>SpringApplication</code> .
threaddump	执行线程 dump.

如果你的应用程序是 Web 应用程序 (`Spring MVC`、`Spring WebFlux` 或 `Jersey`) , 则可以使用以下附加端点:

ID	描述
heapdump	返回一个 <code>hprof</code> 堆 dump 文件.
jolokia	通过 HTTP 暴露 JMX bean (当 Jolokia 在 classpath 上时, 不适用于 WebFlux) . 需要依赖 <code>jolokia-core</code>
logfile	返回日志文件的内容 (如果已设置 <code>logging.file</code> 或 <code>logging.path</code> 属性) . 支持使用 HTTP Range 头来检索部分日志文件的内容.
prometheus	以可以由 Prometheus 服务器抓取的格式暴露指标. 需要依赖 <code>micrometer-registry-prometheus</code>

6.2.1. 启用端点

默认情况下,Actuator 启用除 `shutdown` 之外的所有端点. 要配置端点的启用,请使用其 `management.endpoint.<id>.enabled` 属性. 以下示例展示了如何启用 `shutdown` 端点:

```
management:
  endpoint:
    shutdown:
      enabled: true
```

如果你希望端点启用是选择性加入而不是选择性退出,请将 `management.endpoints.enabled-by-default` 属性设置为 `false`,并使用各个端点的 `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
  endpoint:
    info:
      enabled: true
```



已完全从应用程序上下文中删除已禁用的端点.

如果只想更改端点所暴露的技术,请改用 `include` 和 `exclude` 属性.

6.2.2. 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暴露它们.

下表显示了内置端点和默认暴露情况:

ID	JMX	Web
<code>auditevents</code>	Yes	No
<code>beans</code>	Yes	No
<code>caches</code>	Yes	No
<code>conditions</code>	Yes	No
<code>configprops</code>	Yes	No

ID	JMX	Web
env	Yes	No
flyway	Yes	No
health	Yes	Yes
heapdump	N/A	No
httptrace	Yes	No
info	Yes	Yes
integrationgraph	Yes	No
jolokia	N/A	No
logfile	N/A	No
loggers	Yes	No
liquibase	Yes	No
metrics	Yes	No
mappings	Yes	No
prometheus	N/A	No
scheduledtasks	Yes	No
sessions	Yes	No
shutdown	Yes	No
startup	Yes	No
threaddump	Yes	No

要更改暴露的端点，请使用以下特定的 `include` 和 `exclude` 属性：

属性	默认
<code>management.endpoints.jmx.exposure.exclude</code>	
<code>management.endpoints.jmx.exposure.include</code>	*
<code>management.endpoints.web.exposure.exclude</code>	
<code>management.endpoints.web.exposure.include</code>	<code>info, health</code>

`include` 属性列出了暴露的端点的 ID `exclude` 属性列出了不应暴露的端点的 ID `exclude` 属性优先于 `include` 属性 可以使用端点 ID 列表配置 `include` 和 `exclude` 属性.

例如,要停止通过 JMX 暴露所有端点并仅暴露 `health` 和 `info` 端点,请使用以下属性:

```
management:  
  endpoints:  
    jmx:  
      exposure:  
        include: "health,info"
```

* 可用于选择所有端点. 例如,要通过 HTTP 暴露除 `env` 和 `beans` 之外的所有端点 ,请使用以下属性:

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: "*"  
        exclude: "env,beans"
```



* 在 YAML 中有特殊含义, 所以如果你想包括 (或排除) 所有端点, 请务必加引号.



如果你的应用程序是暴露的,我们强烈建议你也[保护你的端点](#).



如果要在暴露端点时实现自己的策略,可以注册一个 `EndpointFilter` bean.

6.2.3. 保护 HTTP 端点

你应该像保护所有其他敏感 URL 一样注意保护 HTTP 端点. 如果存在 Spring Security,则默认使用 Spring Security 的内容协商策略来保护端点. 例如,如果你希望为 HTTP 端点配置自定义安全策略,只允许具有特定角色身份的用户访问它们, Spring Boot 提供了方便的 `RequestMatcher` 对象,可以与 Spring Security 结合使用.

典型的 Spring Security 配置可能如下:

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {

    http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests
) ->
        requests.anyRequest().hasRole("ENDPOINT_ADMIN"));
    http.httpBasic();
    return http.build();
}

```

上面的示例使用 `EndpointRequest.toAnyEndpoint()` 将请求与所有端点进行匹配，然后确保所有端点都具有 `ENDPOINT_ADMIN` 角色。`EndpointRequest` 上还提供了其他几种匹配器方法。有关详细信息，请参阅 API 文档 ([HTML](#) 或 [PDF](#))。

如果应用程序部署在有防火墙的环境，你可能希望无需身份验证即可访问所有 `Actuator` 端点。你可以通过更改 `management.endpoints.web.exposure.include` 属性来执行此操作，如下所示：

```

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，如下所示：

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {

    http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests
) ->
        requests.anyRequest().permitAll());
    return http.build();
}

```



在上面的两个示例中，配置只适用于 `actuator` 端点。由于 Spring Boot 的安全配置在存在任何 `SecurityFilterChain` bean 时完全退出，因此您将需要配置一个附加的 `SecurityFilterChain` bean，其中包含应用于应用程序其余部分的规则。

6.2.4. 配置端点

端点对不带参数读取操作的响应自动缓存。要配置端点缓存响应的时间长度，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：

```
management:
  endpoint:
    beans:
      cache:
        time-to-live: "10s"
```



前缀 `management.endpoint.<name>` 用于唯一标识配置的端点。



在进行一个身份验证 HTTP 请求时，`Principal` 被视为端点的输入，因此不会缓存响应。

6.2.5. Actuator Web 端点超媒体

添加 `discovery page`，其包含指向所有端点的链接。默认情况下，`discovery page` 在 `/actuator` 上可访问。

配置一个自定义管理上下文（management context）路径时，`discovery page` 会自动从 `/actuator` 移动到管理上下文的根目录。例如，如果管理上下文路径是 `/management`，则可以从 `/management` 获取 `discovery page`。当管理上下文路径设置为 `/` 时，将禁用发现页面以防止与其他映射冲突。

6.2.6. 跨域支持

[Cross-origin resource sharing \(CORS\)](#) 是一个 W3C 规范，允许你以灵活的方式指定授权的跨域请求类型。如果你使用 Spring MVC 或 Spring WebFlux，则可以配置 Actuator 的 Web 端点以支持此类方案。

默认情况下 CORS 支持被禁用,仅在设置了 `management.endpoints.web.cors.allowed-origins` 属性后才启用 CORS 支持. 以下配置允许来自 `example.com` 域的 GET 和 POST 调用:

```
management:
  endpoints:
    web:
      cors:
        allowed-origins: "https://example.com"
        allowed-methods: "GET,POST"
```



有关选项的完整列表,请参阅 [CorsEndpointProperties](#)

6.2.7. 实现自定义端点

如果你添加一个使用了 `@Endpoint` 注解的 `@Bean`,则使用 `@ReadOperation`, `@WriteOperation`, 或 `@DeleteOperation` 注解的所有方法都将通过 JMX 自动暴露,并且在 Web 应用程序中也将通过 HTTP 暴露. 可以使用 Jersey、Spring MVC 或 Spring WebFlux 通过 HTTP 暴露端点.

以下示例暴露了一个 `read` 操作,该操作返回一个自定义对象:

```
@ReadOperation
public CustomData getCustomData() {
    return new CustomData("test", 5);
}
```

你还可以使用 `@JmxEndpoint` 或 `@WebEndpoint` 编写特定技术的端点. 这些端点仅限于各自的技术. 例如,`@WebEndpoint` 仅通过 HTTP 暴露,而不是 JMX.

你可以使用 `@EndpointWebExtension` 和 `@EndpointJmxExtension` 编写特定技术的扩展. 通过这些注解,你可以提供特定技术的操作来扩充现有端点.

最后,如果你需要访问特定 Web 框架的功能,则可以实现 `Servlet` 或 `Spring @Controller` 和 `@RestController` 端点,但代价是它们无法通过 JMX 或使用其他 Web 框架.

接收输入

端点上的操作通过参数接收输入。通过 Web 暴露时，这些参数的值取自 URL 的查询参数和 JSON 请求体。通过 JMX 暴露时，参数将映射到 MBean 操作的参数。

默认情况下参数是必须的。可以使用 `@javax.annotation.Nullable` 或 `@org.springframework.lang.Nullable` 对它们进行注解，使它们成为可选项。

JSON 请求体中的每个根属性都可以映射到端点的参数。考虑以下 JSON 请求体：

```
{
  "name": "test",
  "counter": 42
}
```

这可用于调用带有 `String name` 和 `int counter` 参数的写操作。

```
@WriteOperation
public void updateCustomData(String name, int counter) {
    // injects "test" and 42
}
```



由于端点与技术无关，因此只能在方法签名中指定简单类型。

特别是不支持使用定义一个 `name` 和 `counter` 属性的 `CustomData` 类型声明单个参数。



要允许将输入映射到操作方法的参数，应使用 `-parameters`

编译实现端点的 Java 代码，并且应使用 `-java-parameters`

编译实现端点的 Kotlin 代码。如果你使用的是 Spring Boot 的 Gradle 插件，或者是 Maven 和 `spring-boot-starter-parent`，则它们会自动执行此操作。

输入类型转换

如有必要，传递给端点操作方法的参数将自动转换为所需类型。在调用操作方法之前，使用 `ApplicationConversionService` 实例以及任何具有 `@EndpointConverter` 限定的 `Converter` 或 `GenericConverter` Bean，将 JMX 或 HTTP 请求接收的输入转换为所需类型。

自定义 Web 端点

`@Endpoint`、`@WebEndpoint` 或 `@EndpointWebExtension` 上的操作将使用 Jersey、Spring MVC 或 Spring WebFlux 通过 HTTP 自动暴露.

Web 端点请求断言

为 Web 暴露的端点上的每个操作自动生成请求断言

路径

断言的路径由端点的 ID 和 Web 暴露的端点的基础路径确定. 默认基础路径是 `/actuator`. 例如, 有 ID 为 `sessions` 的端点将使用 `/actuator/sessions` 作为其在断言中的路径.

通过使用 `@Selector` 注解操作方法的一个或多个参数, 可以进一步自定义路径.

这样的参数作为路径变量添加到路径断言中. 调用端点操作时, 变量的值将传递给操作方法. 如果要捕获所有剩余的路径元素, 可以将 `@Selector(Match=ALL_REMAINING)` 添加到最后一个参数, 并将其设置为与 `String []` 转换兼容的类型.

HTTP 方法

断言的 HTTP 方法由操作类型决定, 如下表所示:

Operation	HTTP method
<code>@ReadOperation</code>	GET
<code>@WriteOperation</code>	POST
<code>@DeleteOperation</code>	DELETE

Consumes

对于使用请求体的 `@WriteOperation` (HTTP POST), 断言的 `consume` 子句是 `application/vnd.spring-boot.actuator.v2+json, application/json`.

对于所有其他操作, `consume` 子句为空.

Produces

断言的 `produce` 子句可以由 `@DeleteOperation`、`@ReadOperation` 和 `@WriteOperation` 注解的 `produce` 属性确定. 该属性是可选的. 如果未使用, 则自动确定

`produce` 子句.

如果操作方法返回 `void` 或 `Void`, 则 `produce` 子句为空. 如果操作方法返回 `org.springframework.core.io.Resource`, 则 `produce` 子句为 `application/octet-stream`. 对于所有其他操作, `produce` 子句是 `application/vnd.spring-boot.actuator.v2+json, application/json`.

Web 端点响应状态

端点操作的默认响应状态取决于操作类型 (读取、写入或删除) 以及操作返回的内容 (如果有).

• `@ReadOperation` 返回一个值, 响应状态为 `200 (OK)`. 如果它未返回值, 则响应状态将为 `404 (未找到)`.

如果 `@WriteOperation` 或 `@DeleteOperation` 返回值, 则响应状态将为 `200 (OK)`. 如果它没有返回值, 则响应状态将为 `204 (无内容)`.

如果在没有必需参数的情况下调用操作, 或者使用无法转换为所需类型的参数, 则不会调用操作方法, 并且响应状态将为 `400 (错误请求)`.

Web 端点范围请求

可用 HTTP 范围请求请求部分 HTTP 资源. 使用 Spring MVC 或 Spring Web Flux 时, 返回 `org.springframework.core.io.Resource` 的操作会自动支持范围请求.



使用 Jersey 时不支持范围请求.

Web 端点安全

Web 端点或特定 Web 的端点扩展上的操作可以接收当前的 `java.security.Principal` 或 `org.springframework.boot.actuate.endpoint.SecurityContext` 作为方法参数. 前者通常与 `@Nullable` 结合使用, 为经过身份验证和未经身份验证的用户提供不同的行为. 后者通常用于使用其 `isUserInRole(String)` 方法执行授权检查.

Servlet 端点

通过实现一个带有 `@ServletEndpoint` 注解的类, `Servlet` 可以作为端点暴露, 该类也实现了

`Supplier<EndpointServlet>`. `Servlet` 端点提供了与 `Servlet` 容器更深层次的集成, 但代价是可移植性. 它们旨在用于将现有 `Servlet` 作为端点暴露. 对于新端点, 应尽可能首选 `@Endpoint` 和 `@WebEndpoint` 注解.

Controller 端点

`@ControllerEndpoint` 和 `@RestControllerEndpoint` 可用于实现仅由 Spring MVC 或 Spring WebFlux 暴露的端点. 使用 Spring MVC 和 Spring WebFlux 的标准注解(如 `@RequestMapping` 和 `@GetMapping`) 映射方法, 并将端点的 ID 用作路径的前缀. 控制器端点提供了与 Spring 的 Web 框架更深层次的集成, 但代价是可移植性. 应尽可能首选 `@Endpoint` 和 `@WebEndpoint` 注解.

6.2.8. 健康信息

你可以使用健康信息来检查正在运行的应用程序的状态.

监控软件经常在生产系统出现故障时使用它提醒某人. `health` 端点暴露的信息取决于 `management.endpoint.health.show-details` 和 `management.endpoint.health.show-components` 属性, 可以使用以下值之一配置属性:

Name	Description
<code>never</code>	永远不会显示细节.
<code>when-authorized</code>	详细信息仅向授权用户显示. 可以使用 <code>management.endpoint.health.roles</code> 配置授权角色.
<code>always</code>	向所有用户显示详细信息.

默认值为 `never`. 当用户处于一个或多个端点的角色时, 将被视为已获得授权.

如果端点没有配置角色 (默认值), 则认为所有经过身份验证的用户都已获得授权. 可以使用 `management.endpoint.health.roles` 属性配置角色.



如果你已保护应用程序并希望使用 `always`, 则安全配置必须允许经过身份验证和未经身份验证的用户对健康端点的访问.

健康信息是从 `HealthContributorRegistry` 的内容中收集的 (默认情况下, `ApplicationContext` 中定义的所有 `HealthContributor` 实例). Spring Boot

包含许多自动配置的 `HealthContributors`, 你也可以自己编写.

`HealthContributor` 可以是 `HealthIndicator`, 也可以是 `CompositeHealthContributor`. `HealthIndicator` 提供实际的健康信息, 包括 `Status`. `CompositeHealthContributor` 提供其他 `HealthContributors` 的组合. 总之, 贡献者形成了一个表示整个系统健康状况的树结构.

默认情况下, 最终系统状态由 `StatusAggregator` 扩展, 根据状态的有序列表对每个 `HealthIndicator` 的状态进行排序. 排序列表中的第一个状态作为整体健康状态. 如果没有 `HealthIndicator` 返回一个 `StatusAggregator` 已知的状态, 则使用 `UNKNOWN` 状态.



`HealthContributorRegistry` 可用于在运行时注册和注销健康指示器.

自动配置的 `HealthIndicators`

Spring Boot 会自动配置以下 `HealthIndicators`. 您也可以通过配置 `management.health.key.enabled` 并使用下表中列出的 `key` 来启用/禁用指定的指标.

Key	Name	Description
cassandra	<code>CassandraDriverHealthIndicator</code>	Checks that a Cassandra database is up.
couchbase	<code>CouchbaseHealthIndicator</code>	Checks that a Couchbase cluster is up.
datasource	<code>DataSourceHealthIndicator</code>	Checks that a connection to <code>DataSource</code> can be obtained.
diskspace	<code>DiskSpaceHealthIndicator</code>	Checks for low disk space.
elasticsearch	<code>ElasticsearchRestHealthIndicator</code>	Checks that an Elasticsearch cluster is up.
hazelcast	<code>HazelcastHealthIndicator</code>	Checks that a Hazelcast server is up.
influxdb	<code>InfluxDbHealthIndicator</code>	Checks that an InfluxDB server is up.

Key	Name	Description
jms	JmsHealthIndicator	Checks that a JMS broker is up.
ldap	LdapHealthIndicator	Checks that an LDAP server is up.
mail	MailHealthIndicator	Checks that a mail server is up.
mongo	MongoHealthIndicator	Checks that a Mongo database is up.
neo4j	Neo4jHealthIndicator	Checks that a Neo4j database is up.
ping	PingHealthIndicator	Always responds with UP.
rabbit	RabbitHealthIndicator	Checks that a Rabbit server is up.
redis	RedisHealthIndicator	Checks that a Redis server is up.
solr	SolrHealthIndicator	Checks that a Solr server is up.



你可以通过设置 `management.health.defaults.enabled` 属性来禁用它们。

其他 `HealthIndicators` 可用,但默认情况下未启用:

Key	Name	Description
livenessstate	LiveNessStateHealthIndicator	Expose the "Liveness" application availability state.
readinessstate	ReadinessStateHealthIndicator	Expose the "Readiness" application availability state.

编写自定义 `HealthIndicators`

要提供自定义健康信息,可以注册实现 `HealthIndicator` 接口的 Spring bean.

你需要提供 `health()` 方法的实现并返回一个 `Health` 响应. `Health` 响应应包括一个状态,并且可以选择包括要显示的其他详细信息. 以下代码展示了一个 `HealthIndicator` 实现示例:

```

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

@Component
public class MyHealthIndicator implements HealthIndicator {

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

}

```



给定 `HealthIndicator` 的标识符是没有 `HealthIndicator` 后缀的 `bean` 的名称（如果存在）。在前面的示例中，健康信息在名为 `my` 的条目中可用。

除了 Spring Boot 的预定义 `Status` 类型之外，`Health` 还可以返回一个表示新系统状态的自定义 `Status`。在这种情况下，还需要提供 `StatusAggregator` 接口的自定义实现，或者必须使用 `management.endpoint.health.status.order` 配置属性配置默认实现。

例如，假设在你的一个 `HealthIndicator` 实现中使用了代码为 `FATAL` 的新 `Status`。需要配置严重性顺序，请将以下属性添加到应用程序属性：

```

management:
  endpoint:
    health:
      status:
        order: "fatal,down,out-of-service,unknown,up"

```

响应中的 HTTP 状态码反映了整体运行状况（例如，`UP` 映射到 `200`，而 `OUT_OF_SERVICE` 和 `DOWN` 映射到 `503`）。任何未映射的健康状态，包括 "`UP`"，都映射为 `200`。如果通过 HTTP 访问健康端点，则可能还需要注册自定义状态映射。配置自定义映射默认会禁用 "`DOWN`" 和 "`OUT_OF_SERVICE`" 映射。如果要保留默认映射，则必须在所有自定义映射显式配置它们。

例如,以下属性将 `FATAL` 映射到 `503` (服务不可用) 并保留 "DOWN" 和 "OUT_OF_SERVICE" 的默认映射:

```
management:
  endpoint:
    health:
      status:
        http-mapping:
          down: 503
          fatal: 503
          out-of-service: 503
```



如果需要控制更多,可以定义自己的 `HttpCodeStatusMapper` bean.

下表展示了内置状态的默认状态映射:

状态	映射
DOWN	SERVICE_UNAVAILABLE (503)
OUT_OF_SERVICE	SERVICE_UNAVAILABLE (503)
UP	No mapping by default, so http status is 200
UNKNOWN	No mapping by default, so http status is 200

响应式健康指示器

对于响应式应用程序,例如使用 `Spring WebFlux` 的应用程序,
`ReactiveHealthContributor` 提供了一个非阻塞的接口来获取应用程序健康信息.
与传统的 `HealthContributor` 类似, 健康信息从
`ReactiveHealthContributorRegistry` 的内容中收集 (默认情况下,
`ApplicationContext` 中定义的所有 `HealthContributor` 和
`ReactiveHealthContributor` 实例). 不检查响应式 API 的常规
`HealthContributors` 在弹性调度程序上执行.



在响应式应用程序中, `ReactiveHealthContributorRegistry` 可用于在运行时注册和取消注册健康指示器. 如果需要注册常规的 `HealthContributor`, 则应使用 `ReactiveHealthContributor#adapt` 对其进行包装.

要从响应式 API 提供自定义健康信息,可以注册实现 `ReactiveHealthIndicator` 接口的 Spring bean. 以下代码展示了 `ReactiveHealthIndicator` 实现的示例:

```
@Component
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {

    @Override
    public Mono<Health> health() {
        return doHealthCheck() //perform some specific health check that returns
a Mono<Health>
            .onErrorResume(ex -> Mono.just(new
Health.Builder().down(ex).build()));
    }

}
```



要自动处理错误,请考虑从 `AbstractReactiveHealthIndicator` 进行扩展.

自动配置的 `ReactiveHealthIndicators`

适当时, Spring Boot 会自动配置以下 `ReactiveHealthIndicators`:

名称	描述
<code>CassandraDriverReactiveHealthIndicator</code>	检查 Cassandra 数据库是否已启动.
<code>CouchbaseReactiveHealthIndicator</code>	检查 Couchbase 集群是否已启动.
<code>MongoReactiveHealthIndicator</code>	检查 Mongo 数据库是否已启动.
<code>Neo4jReactiveHealthIndicator</code>	检查 Neo4j 数据库是否已启动.

名称	描述
RedisReactive HealthIndicator or	检查 Redis 服务器是否已启动.



必要时,响应式指示器取代常规指示器. 此外,任何未明确处理的 `HealthIndicator` 都会自动包装.

Health 组

有时候,将健康指标分为不同的组很有用. 例如,如果将应用程序部署到 `Kubernetes`,则可能需要一组不同的运行状况指示器来进行 `active` 和 "就绪" 探针.

要创建运行状况指示器组,可以使用 `management.endpoint.health.group.<name>` 属性,并使用 `include` 或 `exclude` 指定需要展示运行状况指示器ID的列表. 例如,创建仅包含数据库指示符的组,可以定义以下内容:

```
management:
  endpoint:
    health:
      group:
        custom:
          include: "db"
```

同样, 要创建一个组, 可从组中排除数据库指标, 并包含所有其他指标, 您可以定义以下内容:

```
management:
  endpoint:
    health:
      group:
        custom:
          exclude: "db"
```

默认情况下,组将继承与系统运行状况相同的 `StatusAggregator` 和 `HttpCodeStatusMapper` 设置,但是,这些设置也可以基于每个组进行定义. 如果需要,也可以覆盖 `show-details` 和 `roles` 属性:

```
management:  
  endpoint:  
    health:  
      group:  
        custom:  
          show-details: "when-authorized"  
          roles: "admin"  
          status:  
            order: "fatal,up"  
            http-mapping:  
              fatal: 500  
              out-of-service: 500
```



如果需要注册自定义 `StatusAggregator` 或 `HttpCodeStatusMapper` Bean以便与该组一起使用,则可以使用 `@Qualifier("groupname")` .

6.2.9. Kubernetes Probes

部署在 `Kubernetes` 上的应用程序可以使用 `Container Probes` 提供有关其内部状态的信息.根据 [您的Kubernetes 配置](#), `kubelet` 将调用这些探针并对结果做出反应.

`Spring Boot` 管理您的 [应用程序可用性转台](#). 如果部署在 `Kubernetes` 环境中,那么 `actuator` 将从 `ApplicationAvailability` 接口收集 "Liveness" and "Readiness" 信息,并在 `Health Indicators: LivenessStateHealthIndicator` 和 `ReadinessStateHealthIndicator` 中使用这些信息. 这些指标将显示在全局健康端点 (`"/actuator/health"`) 中. 他们还暴露了单独的 HTTP 探针,这些探针位置 `Health Groups` 中: `"/actuator/health/liveness"` 和 `"/actuator/health/readiness"`.

然后,您可以使用以下端点信息配置 `Kubernetes` 基础架构

```

livenessProbe:
  httpGet:
    path: /actuator/health/liveness
    port: <actuator-port>
  failureThreshold: ...
  periodSeconds: ...

readinessProbe:
  httpGet:
    path: /actuator/health/readiness
    port: <actuator-port>
  failureThreshold: ...
  periodSeconds: ...

```



`<actuator-port>` 应该设置为 `actuator` 端点可用的端口. 它可以是 `web` 服务器端口, 或者是单独的管理端口(如果 `"management.server.port"` 已设置)

仅当应用程序在 [Kubernetes 环境中运行时](#), 才会自动启用这些运行状况组. 您可以使用 `management.endpoint.health.probes.enabled` 配置属性在任何环境中启用它们.



如果应用程序的启动时间比配置的激活时间长, `Kubernetes` 会提及 `"startupProbe"` 作为可能的解决方案. 由于在所有启动任务完成之前 `"readinessProbe"` 将失败, 因此此处不一定需要 `"startupProbe"`, 请参阅 [探针在应用程序生命周期中的行为](#).



如果您的 `Actuator` 端点部署在单独的管理上下文中, 请注意, 端点将不使用与主应用程序相同的 Web 基础结构 (端口, 连接池, 框架组件). 在这种情况下, 即使主应用程序无法正常运行 (例如, 它不能接受新连接), 也可能会成功进行探测检查.

使用 `Kubernetes` 探针检查外部状态

`Actuator` 将 `"liveness"` 和 `"readiness"` 探针配置为 [Health Groups features](#). 这意味着所有 `Health Groups` 功能均可用. 例如, 您可以配置其他运行状况指标:

```
management:
  endpoint:
    health:
      group:
        readiness:
          include: "readinessState,customCheck"
```

默认情况下, Spring Boot 不会将其他运行状况指标添加到这些组中.

“liveness” 探针不应依赖于外部系统的运行状况检查. 如果[应用程序的 Liveness 状态](#) 被破坏, Kubernetes 将尝试通过重新启动应用程序实例来解决该问题. 这意味着, 如果外部系统发生故障 (例如数据库, Web API, 外部缓存), 则 Kubernetes 可能会重新启动所有应用程序实例并造成级联故障.

至于 “readiness” 探针, 必须由应用程序开发人员仔细选择检查外部系统的选项, 即, Spring Boot 在 readiness 探针中不包括任何其他运行状况检查. 如果[应用程序实例的 readiness 状态](#) 尚未就绪, Kubernetes 将不会将流量路由到该实例.

应用程序实例可能不会共享某些外部系统, 在这种情况下, 它们很自然地可以包含在 readiness 探针中. 其他外部系统对于该应用程序可能不是必需的 (该应用程序可能具有 circuit breakers 和 fallbacks), 在这种情况下, 绝对不应该包括它们. 不幸的是, 由所有应用程序实例共享的外部系统是常见的, 您必须做出判断调用: 将其包括在 readiness 探针中, 并期望在外部服务关闭时该应用程序退出服务, 或者退出该应用程序排除并处理更高级别的故障, 例如 在回调中使用断路器.



如果应用程序的所有实例尚未就绪, 则 type=ClusterIP 或 NodePort 服务将不接受任何传入连接. 由于没有连接, 因此没有 HTTP 错误响应 (503 等). type=LoadBalancer 的服务可能会或可能不会接受连接, 具体取决于提供程序. 具有显式 Ingress 的 Service 还将以依赖于实现的方式进行响应- Ingress Service 本身必须决定如何处理下游的 “拒绝连接”. 对于负载均衡器和入口都非常可能使用 HTTP 503.

另外, 如果应用程序正在使用 Kubernetes [autoscaling](#), 它可能会对从负载平衡中取出的应用程序做出不同的响应, 这取决于它的 autoscaler 配置.

应用程序生命周期和探针状态

Kubernetes 探针支持的一个重要方面是它与应用程序生命周期的一致性。在 [AvailabilityState](#) (应用程序的内存内部状态) 和暴露该状态的实际 Probe 之间有一个显著的区别：根据应用程序生命周期的阶段，Probe 可能不可用。

[Spring Boot](#) 在启动和关闭期间发布应用程序事件。而 Probes 可以监听此类事件并暴露给 [AvailabilityState](#) 信息。

下表显示了 [AvailabilityState](#) 和 HTTP 连接器在不同阶段的状态。

当 Spring Boot 应用程序启动时：

Startup phase	Liveness State	Readiness State	HTTP server	Notes
Starting	BROKEN	REFUSING_TRAFFIC	Not started	Kubernetes checks the "liveness" Probe and restarts the application if it takes too long.
Started	CORRECT	REFUSING_TRAFFIC	Refuses requests	The application context is refreshed. The application performs startup tasks and does not receive traffic yet.
Ready	CORRECT	ACCEPTING_TRAFFIC	Accepts requests	Startup tasks are finished. The application is receiving traffic.

当 Spring Boot 应用程序关闭时：

Shutdown phase	Liveness State	Readiness State	HTTP server	Notes
Running	CORRECT	ACCEPTING_TRAFFIC	Accepts requests	Shutdown has been requested.

Shutdown phase	Liveness State	Readiness State	HTTP server	Notes
Graceful shutdown	CORRECT	REFUSING_TRAFFIC	New requests are rejected	If enabled, graceful shutdown processes in-flight requests.
Shutdown complete	N/A	N/A	Server is shut down	The application context is closed and the application is shut down.



请查看 [Kubernetes 容器生命周期章节](#), 以获取有关 Kubernetes 部署的更多信息.

6.2.10. 应用程序信息

应用程序信息暴露从 `ApplicationContext` 中定义的所有 `InfoContributor` bean 收集的各种信息. Spring Boot 包含许多自动配置的 `InfoContributor` bean, 你可以编写自己的 bean.

自动配置的 `InfoContributors`

适当时, Spring Boot 会自动配置以下 `InfoContributor` bean:

名称	描述
<code>EnvironmentInfoContributor</code>	在 <code>info</code> key 下显示 <code>Environment</code> 中的所有 key.
<code>GitInfoContributor</code>	如果 <code>git.properties</code> 可用则暴露 git 信息.
<code>BuildInfoContributor</code>	如果 <code>META-INF/build-info.properties</code> 可用则暴露构建信息.



可以通过设置 `management.info.defaults.enabled` 属性来禁用它们.

自定义应用程序信息

你可以通过设置 `info.*` 字符串属性来自定义 `info` 端点暴露的数据. `info` key 下的所有 `Environment` 属性都会自动暴露. 例如, 你可以将以下设置添加到 `application.properties` 文件中:

```
info:
  app:
    encoding: "UTF-8"
  java:
    source: "11"
    target: "11"
```

除了对这些值进行硬编码之外, 您还可以在 [构建时扩展信息属性](#).

假设您使用 Maven, 则可以按如下所示重写前面的示例:



```
info:
  app:
    encoding: "@project.build.sourceEncoding@"
  java:
    source: "@java.version@"
    target: "@java.version@"
```

Git 提交信息

`info` 端点的另一个有用功能是它能够在构建项目时发布 `git` 源码仓库相关的信息. 如果 `GitProperties` bean 可用, 则可以使用 `info` 端点暴露这些属性.



如果 `git.properties` 文件在 `classpath` 的根目录中可用, 则会自动配置 `GitProperties` bean. 有关更多详细信息, 请参阅[生成 git 信息](#).

默认情况下, `git.branch`、`git.commit.id` 和 `git.commit.time` 属性 (如果存在) . 如果您不希望端点响应中包含任何这些属性, 则需要将它们从 `git.properties` 文件中排除. 如果要显示完整的 `git` 信息 (即 `git.properties` 的完整内容), 请使用 `management.info.git.mode` 属性, 如下所示:

```
management:  
  info:  
    git:  
      mode: "full"
```

要完全禁用来自 `info` 端点的 `git commit` 信息, 请将 `management.info.git.enabled` 属性设置为 `false`, 如下所示:

```
management.info.git.enabled=false
```

构建信息

如果 `BuildProperties` bean 可用, 则 `info` 端点还可以发布构建相关的信息. 如果 classpath 中有 `META-INF/build-info.properties` 文件, 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](#)".

编写自定义 InfoContributors

要提供自定义应用程序信息, 可以注册实现 `InfoContributor` 接口的 Spring bean.

以下示例提供了具有单个值的 `example` 条目:

```

import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example",
            Collections.singletonMap("key", "value"));
    }

}

```

如果访问 `info` 端点, 你应该能看到包含以下附加条目的响应:

```
{
  "example": {
    "key" : "value"
  }
}
```

6.3. 通过 HTTP 监控和管理

如果你正在开发 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 暴露. 默认约定是使用前缀为 `/actuator` 的端点的 `id` 作为 URL 路径. 例如, `health` 以 `/actuator/health` 暴露.



Spring MVC, Spring WebFlux 和 Jersey 本身支持 Actuator. 如果同时提供 Jersey 和 Spring MVC, 将使用 Spring MVC.



Jackson 是获取 API 文档 ([HTML](#) or [PDF](#)) 中所记录的正确 JSON 响应所必需的依赖项.

6.3.1. 自定义 Management 端点路径

有时,自定义 management 端点的前缀很有用. 例如,你的应用程序可能已将 `/actuator` 用于其他目的. 你可以使用 `management.endpoints.web.base-path` 属性更改 management 端点的前缀,如下所示:

```
management:  
  endpoints:  
    web:  
      base-path: "/manage"
```

前面的 `application.properties` 示例将端点从 `/actuator/{id}` 更改为 `/manage/{id}` (例如,`/manage/info`) .



除非已将 management 端口配置为使用[使用其他 HTTP 端口暴露端点](#),否则 `management.endpoints.web.base-path` 与 `server.servlet.context-path` 相关联 (Servlet web applications) 或 `spring.webflux.base-path` (reactive web applications). 如果配置了 `management.server.port`,则 `management.endpoints.web.base-path` 与 `management.server.base-path` 相关联.

如果要将端点映射到其他路径,可以使用 `management.endpoints.web.path-mapping` 属性.

以下示例将 `/actuator/health` 重新映射到 `/healthcheck`:

```
management:  
  endpoints:  
    web:  
      base-path: "/"  
      path-mapping:  
        health: "healthcheck"
```

6.3.2. 自定义 Management 服务器端口

使用默认 HTTP 端口暴露 management 端点是基于云部署的明智选择. 但是

,如果应用程序是在自己的数据中心内运行,你可能更喜欢使用其他 HTTP 端口暴露端点.

你可以设置 `management.server.port` 属性以更改 HTTP 端口,如下所示:

```
management:  
  server:  
    port: 8081
```



在 Cloud Foundry 上,默认情况下,应用程序仅在端口 8080 上接收 HTTP 和 TCP 路由请求. 如果要在 Cloud Foundry 上使用自定义管理端口,则需要明确设置应用程序的路由,以将流量转发到自定义端口.

6.3.3. 配置 Management 的 SSL

当配置为使用自定义端口时,还可以使用各种 `management.server.ssl.*` 属性为 `management` 服务器配置自己的 SSL. 例如,这样做可以在主应用程序使用 HTTPS 时可通过 HTTP 使用 `management` 服务器,如以下属性设置所示:

```
server:  
  port: 8443  
  ssl:  
    enabled: true  
    key-store: "classpath:store.jks"  
    key-password: secret  
management:  
  server:  
    port: 8080  
    ssl:  
      enabled: false
```

或者,主服务器和 `management` 服务器都可以使用 SSL,但他们的 `key store` 不同,如下所示:

```

server:
  port: 8443
  ssl:
    enabled: true
    key-store: "classpath:main.jks"
    key-password: "secret"
management:
  server:
    port: 8080
    ssl:
      enabled: true
      key-store: "classpath:management.jks"
      key-password: "secret"

```

6.3.4. 配置 Management 服务器地址

你可以通过设置 `management.server.address` 属性来自定义 `management` 端点可用的地址。如果你只想在内部或操作的网络上监听或仅监听来自 `localhost` 的连接，那么这样做会非常有用。



仅当端口与主服务器端口不同时，才能监听不同的地址。

以下 `application.properties` 示例不允许远程连接 `management`:

```

management:
  server:
    port: 8081
    address: "127.0.0.1"

```

6.3.5. 禁用 HTTP 端点

如果你不希望通过 HTTP 暴露端点，则可以将 `management` 端口设置为 `-1`，如下所示：

```

management:
  server:
    port: -1

```

这可以使用 `management.endpoints.web.exposure.exclude` 属性来实现，如下所示：

```
management:
  endpoints:
    web:
      exposure:
        exclude: "*"
```

6.4. 通过 JMX 监控和管理

Java 管理扩展 (Java Management Extensions, JMX)

提供了一种监控和管理应用程序的标准机制。默认情况下，此功能未启用，可以通过将配置属性 `spring.jmx.enabled` 设置为 `true` 来启用。默认情况下，Spring Boot 将 `management` 端点暴露为 `org.springframework.boot` 域下的 JMX MBean。

6.4.1. 自定义 MBean 名称

MBean 的名称通常是从端点的 `id` 生成的。例如，`health` 端点暴露为 `org.springframework.boot:type=Endpoint,name=Health`。

如果你的应用程序包含多个 Spring `ApplicationContext`，可能会发生名称冲突。

要解决此问题，可以将 `spring.jmx.unique-names` 属性设置为 `true`，以保证 MBean 名称始终唯一。

你还可以自定义暴露端点的 JMX 域。以下设置展示了在 `application.properties` 中执行此操作的示例：

```
spring:
  jmx:
    unique-names: true
management:
  endpoints:
    jmx:
      domain: "com.example.myapp"
```

6.4.2. 禁用 JMX 端点

如果你不想通过 JMX 暴露端点，可以将 `management.endpoints.jmx.exposure.exclude` 属性设置为 `*`，如下所示：

```
management:
  endpoints:
    jmx:
      exposure:
        exclude: "*"
```

6.4.3. 通过 HTTP 使用 Jolokia 访问 JMX

Jolokia 是一个 JMX-HTTP 桥, 它提供了一种访问 JMX bean 的新方式。要使用 Jolokia, 请引入依赖: `org.jolokia:jolokia-core`。例如, 使用 Maven, 你将添加以下依赖:

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

之后可以通过将 `jolokia` 或 `*` 添加到 `management.endpoints.web.exposure.include` 属性来暴露 Jolokia 端点。最后, 你可以在 management HTTP 服务器上使用 `/actuator/jolokia` 访问它。



Jolokia 端点将 Jolokia 的 servlet 公开为 `actuator endpoint`。它特定于运行于 Spring MVC 和 Jersey 的 servlet 环境。该端点在 WebFlux 应用程序中将不可用。

自定义 Jolokia

Jolokia 有许多设置, 你可以通过设置 `servlet` 参数来使用传统方式进行配置。使用 Spring Boot 时, 你可以使用 `application.properties` 文件配置。请在参数前加上 `management.endpoint.jolokia.config`。如下所示:

```
management:
  endpoint:
    jolokia:
      config:
        debug: true
```

禁用 Jolokia

如果你使用 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
```

6.5. 日志记录器

Spring Boot Actuator 有可在运行时查看和配置应用程序日志级别的功能.

你可以查看全部或单个日志记录器的配置,该配置由显式配置的日志记录级别以及日志记录框架为其提供的有效日志记录级别组成. 这些级别可以是以下之一:

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `OFF`
- `null`

`null` 表示没有显式配置.

6.5.1. 配置一个日志记录器

要配置日志记录器,请将部分实体 `POST` 到资源的 `URI`,如下所示:

```
{  
  "configuredLevel": "DEBUG"  
}
```



要重置日志记录器的特定级别（并使用默认配置代替），可以将 `null` 值作为 `configuredLevel` 传递。

6.6. 指标

Spring Boot Actuator 为 Micrometer 提供了依赖管理和自动配置，Micrometer 是一个支持 numerous monitoring systems 的应用程序指标门面，包括：

- [AppOptics](#)
- [Atlas](#)
- [Datadog](#)
- [Dynatrace](#)
- [Elastic](#)
- [Ganglia](#)
- [Graphite](#)
- [Humio](#)
- [Influx](#)
- [JMX](#)
- [KairosDB](#)
- [New Relic](#)
- [Prometheus](#)
- [SignalFx](#)
- [Simple \(in-memory\)](#)
- [Stackdriver](#)
- [StatsD](#)
- [Wavefront](#)



要了解有关 **Micrometer** 功能的更多信息,请参阅其 [参考文档](#),特别是 [概念部分](#).

6.6.1. 入门

Spring Boot 自动配置了一个组合的 **MeterRegistry**,并为 **classpath** 中每个受支持的实现向该组合注册一个注册表. 在运行时,只需要 **classpath** 中有 **micrometer-registry-{system}** 依赖即可让 **Spring Boot** 配置该注册表.

大部分注册表都有共同点 例如,即使 **Micrometer** 注册实现位于 **classpath** 上,你也可以禁用特定的注册表. 例如,要禁用 **Datadog**:

```
management:  
  metrics:  
    export:  
      datadog:  
        enabled: false
```

您也可以禁用所有注册表, 除非注册表特定属性另有说明, 如下例所示:

```
management:  
  metrics:  
    export:  
      defaults:  
        enabled: false
```

Spring Boot 还会将所有自动配置的注册表添加到 **Metrics** 类的全局静态复合注册表中,除非你明确禁止:

```
management:  
  metrics:  
    use-global-registry: false
```

在注册表中注册任何指标之前,你可以注册任意数量的 **MeterRegistryCustomizer** bean 以进一步配置注册表,例如通用标签:

```
@Bean
MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
    return registry -> registry.config().commonTags("region", "us-east-1");
}
```

你可以通过指定泛型类型,自定义注册表实现:

```
@Bean
MeterRegistryCustomizer<GraphiteMeterRegistry> graphiteMetricsNamingConvention()
{
    return registry -> registry.config().namingConvention(MY_CUSTOM_CONVENTION);
}
```

Spring Boot 还[配置内置的测量工具](#) ,你可以通过配置或专用注解标记来控制.

6.6.2. 支持的监控系统

AppOptics

默认情况下,AppOptics 注册表会定期将指标推送到 api.appoptics.com/v1/measurements. 要将指标导出到 SaaS AppOptics,你必须提供 API 令牌:

```
management:
  metrics:
    export:
      appoptics:
        api-token: "YOUR_TOKEN"
```

Atlas

默认情况下,指标标准将导出到本地的 [Atlas](#). 可以使用以下方式指定 [Atlas 服务器](#)的位置:

```
management:
  metrics:
    export:
      atlas:
        uri: "https://atlas.example.com:7101/api/v1/publish"
```

Datadog

Datadog 注册表会定期将指标推送到 [datadoghq](#). 要将指标导出到 Datadog, 你必须提供 API 密钥:

```
management:  
  metrics:  
    export:  
      datadog:  
        api-key: "YOUR_KEY"
```

你还可以更改指标标准发送到 Datadog 的间隔时间:

```
management:  
  metrics:  
    export:  
      datadog:  
        step: "30s"
```

Dynatrace

Dynatrace 注册表定期将指标推送到配置的 URI. 要将指标导出到 Dynatrace, 必须提供 API 令牌、设备 ID 和 URI:

```
management:  
  metrics:  
    export:  
      dynatrace:  
        api-token: "YOUR_TOKEN"  
        device-id: "YOUR_DEVICE_ID"  
        uri: "YOUR_URI"
```

你还可以更改指标标准发送到 Dynatrace 的间隔时间:

```
management:  
  metrics:  
    export:  
      dynatrace:  
        step: "30s"
```

Elastic

默认情况下,指标将导出到本地的 [Elastic](#). 可以使用以下属性提供 [Elastic](#) 服务器的位置:

```
management:  
  metrics:  
    export:  
      elastic:  
        host: "https://elastic.example.com:8086"
```

Ganglia

默认情况下,指标将导出到本地的 [Ganglia](#) . 可以使用以下方式提供 [Ganglia](#) 服务器主机和端口:

```
management:  
  metrics:  
    export:  
      ganglia:  
        host: "ganglia.example.com"  
        port: 9649
```

Graphite

默认情况下,指标将导出到本地的 [Graphite](#). 可以使用以下方式提供 [Graphite server](#) 主机和端口:

```
management:  
  metrics:  
    export:  
      graphite:  
        host: "graphite.example.com"  
        port: 9004
```

[Micrometer](#) 提供了一个默认的 [HierarchicalNameMapper](#), 它管理维度计数器 `id` 如何映射到平面分层名称.



要控制此行为,请定义 `GraphiteMeterRegistry` 并提供自己的 `HierarchicalNameMapper`. 除非你自己定义,否则使用自动配置的 `GraphiteConfig` 和 `Clock` bean:

```
@Bean  
public GraphiteMeterRegistry graphiteMeterRegistry(GraphiteConfig config, Clock  
clock) {  
    return new GraphiteMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);  
}
```

Humio

默认情况下,`Humio` 注册表会定期将指标推送到 cloud.humio.com. 要将指标导出到 SaaS `Humio`,你必须提供 API 令牌:

```
management:  
metrics:  
export:  
humio:  
api-token: "YOUR_TOKEN"
```

你还应配置一个或多个标记,以标识要推送指标的数据源:

```
management:  
metrics:  
export:  
humio:  
tags:  
alpha: "a"  
bravo: "b"
```

Influx

默认情况下,指标将导出到本地的 `Influx` . 要指定 `Influx server` 的位置,可以使用:

```
management:
  metrics:
    export:
      influx:
        uri: "https://influx.example.com:8086"
```

JMX

`Micrometer` 提供了与 [JMX](#) 的分层映射, 主要为了方便在本地查看指标且可移植.

默认情况下, 指标将导出到 `metrics` JMX 域. 可以使用以下方式提供要使用的域:

```
management:
  metrics:
    export:
      jmx:
        domain: "com.example.app.metrics"
```

`Micrometer` 提供了一个默认的 [HierarchicalNameMapper](#), 它管理维度计数器 `id` 如何映射到平面分层名称.



要控制此行为, 请定义 `JmxMeterRegistry` 并提供自己的 `HierarchicalNameMapper`. 除非你自己定义, 否则使用自动配置的 `JmxConfig` 和 `Clock` bean:

```
@Bean
public JmxMeterRegistry jmxMeterRegistry(JmxConfig config, Clock clock) {
  return new JmxMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

KairosDB

默认情况下, 指标将导出到本地的 [KairosDB](#) . 可以使用以下方式提供 `KairosDB server` 的位置:

```
management:  
  metrics:  
    export:  
      kairos:  
        uri: "https://kairosdb.example.com:8080/api/v1/datapoints"
```

New Relic

New Relic 注册表定期将指标推送到 New Relic。要将指标导出到 New Relic，你必须提供 API 密钥和帐户 ID：

```
management:  
  metrics:  
    export:  
      newrelic:  
        api-key: "YOUR_KEY"  
        account-id: "YOUR_ACCOUNT_ID"
```

你还可以更改将指标发送到 New Relic 的间隔时间：

```
management:  
  metrics:  
    export:  
      newrelic:  
        step: "30s"
```

默认情况下，指标标准是通过 REST 调用发布的，但是如果您在类路径中有 Java Agent API，也可以使用它：

```
management:  
  metrics:  
    export:  
      newrelic:  
        client-provider-type: "insights-agent"
```

最后，你可以完全控制你定义的 `NewRelicClientProvider` bean。

Prometheus

Prometheus 希望抓取或轮询各个应用实例以获取指标数据。Spring Boot 在 `/actuator/prometheus` 上提供 `actuator` 端点，以适当的格式呈现 **Prometheus scrape**。



默认情况下端点不可用，必须暴露，请参阅 [暴露端点](#) 以获取更多详细信息。

以下是要添加到 `prometheus.yml` 的示例 `scrape_config`：

```
scrape_configs:
- job_name: 'spring'
  metrics_path: '/actuator/prometheus'
  static_configs:
    - targets: ['HOST:PORT']
```

对于短暂的或批处理的工作，其时间可能不够长，无法被废弃，可以使用 **Prometheus Pushgateway** 支持将其指标暴露给Prometheus。要启用Prometheus Pushgateway支持，请在项目中添加以下依赖：

```
<dependency>
<groupId>io.prometheus</groupId>
<artifactId>simpleclient_pushgateway</artifactId>
</dependency>
```

当在类路径上存在 `Prometheus Pushgateway` 依赖，并且 `management.metrics.export.prometheus.pushgateway.enabled` 属性为 `true`，Spring Boot 会自动配置 `PrometheusPushGatewayManager` bean。这可以管理将指标推送到 `Prometheus Pushgateway`

可以使用 `management.metrics.export.prometheus.pushgateway` 下的属性来调整 `PrometheusPushGatewayManager`。对于高级配置，您还可以提供自己的 `PrometheusPushGatewayManager` bean。

SignalFx

`SignalFx` 注册表定期将指标推送到 `SignalFx`。要将指标导出到 `SignalFx`，你必须提供访问令牌：

```
management:  
  metrics:  
    export:  
      signalfx:  
        access-token: "YOUR_ACCESS_TOKEN"
```

你还可以更改将指标发送到 [SignalFx](#) 的间隔时间：

```
management:  
  metrics:  
    export:  
      signalfx:  
        step: "30s"
```

Simple

[Micrometer](#) 附带一个简单的内存后端,如果没有配置其他注册表,它将自动用作后备. 这使你可以查看[指标端点](#)中收集的指标信息.

只要你使用了任何其他可用的后端,内存后端就会自动禁用. 你也可以显式禁用它:

```
management:  
  metrics:  
    export:  
      simple:  
        enabled: false
```

Stackdriver

[Stackdriver](#) 注册表会定期将指标推送到 [Stackdriver](#). 要将指标导出到 SaaS [Stackdriver](#), 必须提供您的 Google Cloud 项目 ID

```
management:  
  metrics:  
    export:  
      stackdriver:  
        project-id: "my-project"
```

您还可以更改将指标发送到 [Stackdriver](#) 的时间间隔:

```
management:  
  metrics:  
    export:  
      stackdriver:  
        step: "30s"
```

StatsD

StatsD 注册表将 UDP 上的指标推送到 [StatsD](#) 代理。默认情况下，指标将导出到本地的 StatsD 代理，可以使用以下方式提供 StatsD 代理主机和端口和协议：

```
management:  
  metrics:  
    export:  
      statsd:  
        host: "statsd.example.com"  
        port: 9125  
        protocol: "udp"
```

你还可以更改要使用的 StatsD 线路协议（默认为 Datadog）：

```
management:  
  metrics:  
    export:  
      statsd:  
        flavor: "etsy"
```

Wavefront

Wavefront 注册表定期将指标推送到 [Wavefront](#)。如果要将指标直接导出到 Wavefront，则你必须提供 API 令牌：

```
management:  
  metrics:  
    export:  
      wavefront:  
        api-token: "YOUR_API_TOKEN"
```

或者，你可以在环境中使用 `Wavefront sidecar` 或内部代理设置，将指标数据转发到

Wavefront API 主机：

```
management:  
  metrics:  
    export:  
      wavefront:  
        uri: "proxy://localhost:2878"
```



如果将指标发布到 Wavefront 代理（[如文档中所述](#)），则主机必须采用 `proxy://HOST:PORT` 格式。

你还可以更改将指标发送到 Wavefront 的间隔时间：

```
management:  
  metrics:  
    export:  
      wavefront:  
        step: "30s"
```

6.6.3. 支持的指标

Spring Boot 在适当的环境注册以下核心指标：

- JVM 指标，报告利用率：
 - 各种内存和缓冲池
 - 与垃圾回收有关的统计
 - 线程利用率
 - 加载/卸载 class 的数量
- CPU 指标
- 文件描述符指标
- Kafka consumer, producer, 和 streams 指标
- Log4j2 指标：记录每个级别记录到 Log4j2 的事件数
- Logback 指标：记录每个级别记录到 Logback 的事件数

- 正常运行时间 指标：报告正常运行时间和表示应用程序绝对启动时间的固定计量值
- Tomcat 指标（必须将 `server.tomcat.mbeanregistry.enabled` 设置为 `true` 才能注册所有 Tomcat 指标）
- [Spring Integration](#) 指标

Spring MVC 指标

通过自动配置，可以检测由 Spring MVC 处理的请求。当 `management.metrics.web.server.request.autotime.enabled` 为 `true` 时，将对所有请求进行这种检测。另外，当设置为 `false` 时，可以通过将 `@Timed` 添加到请求处理方法来启用检测：

```

@RestController
@Timed ①
public class 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，为控制器中的每个请求处理程序启用计时。

② 启用单个端点。如果你在类上使用了它，就不需要在方法上再次声明，但可以用它来进一步自定义该特定端点的计时器。

③ 使用 `longTask = true` 的方法为该方法启用长任务计时器。
长任务计时器需要单独的指标名称，并且可以使用短任务计时器进行堆叠。

默认情况下，使用名称为 `http.server.requests` 生成指标指标。可以通过设置 `management.metrics.web.server.requests-metric-name` 属性来自定义名称。

默认情况下，Spring MVC 相关指标使用了以下标签标记：

标签	描述
<code>exception</code>	处理请求时抛出的异常的简单类名。

标签	描述
<code>method</code>	请求的方法 (例如, <code>GET</code> 或 <code>POST</code>)
<code>outcome</code>	根据响应状态码生成结果. <code>1xx</code> 是 <code>INFORMATIONAL</code> , <code>2xx</code> 是 <code>SUCCESS</code> , <code>3xx</code> 是 <code>REDIRECTION</code> , <code>4xx</code> 是 <code>CLIENT_ERROR</code> , <code>5xx</code> 是 <code>SERVER_ERROR</code>
<code>status</code>	响应的 HTTP 状态码 (例如, <code>200</code> 或 <code>500</code>)
<code>uri</code>	如果可能, 在变量替换之前请求 URI 模板 (例如, <code>/api/person/{id}</code>)

要添加到默认标签, 请提供一个或多个实现 `WebMvcTagsContributor` 的 `@Bean`.

要替换默认标签, 请提供实现 `WebMvcTagsProvider` 的 `@Bean`.

Spring WebFlux 指标

自动配置启用了 `WebFlux` 控制器和函数式处理程序处理的所有请求的指标记录功能.

默认情况下, 使用名为 `http.server.requests` 生成指标. 你可以通过设置 `management.metrics.web.server.requests-metric-name` 属性来自定义名称.

默认情况下, 与 `WebFlux` 相关的指标使用以下标签标记:

标签	描述
<code>exception</code>	处理请求时抛出的异常的简单类名.
<code>method</code>	请求方法 (例如, <code>GET</code> 或 <code>POST</code>)
<code>outcome</code>	根据响应状态码生成请求结果. <code>1xx</code> 是 <code>INFORMATIONAL</code> , <code>2xx</code> 是 <code>SUCCESS</code> , <code>3xx</code> 是 <code>REDIRECTION</code> , <code>4xx</code> 是 <code>CLIENT_ERROR</code> , <code>5xx</code> 是 <code>SERVER_ERROR</code>
<code>status</code>	响应的 HTTP 状态码 (例如, <code>200</code> 或 <code>500</code>)
<code>uri</code>	如果可能, 在变量替换之前请求 URI 模板 (例如, <code>/api/person/{id}</code>)

要添加到默认标签, 请提供一个或多个实现 `WebFluxTagsContributor` 的 `@Bean`.

要替换默认标签,请提供实现 `WebFluxTagsProvider` 的 `@Bean`.

Jersey Server 指标

当 `Micrometer` 的 `micrometer-jersey2` 模块位于类路径上时,自动配置将启用对 Jersey JAX-RS 实现所处理的请求的检测. 当 `management.metrics.web.server.auto-time-requests` 为 `true` 时,将对所有请求进行该项检测. 当设置为 `false` 时,你可以通过将 `@Timed` 添加到请求处理方法上来启用检测:

```
@Component
@Path("/api/people")
@Timed ①
public class Endpoint {
    @GET
    @Timed(extraTags = { "region", "us-east-1" }) ②
    @Timed(value = "all.people", longTask = true) ③
    public List<Person> listPeople() { ... }
}
```

- ① 在资源类上,为资源中的每个请求处理程序启用计时.
- ② 在方法上则启用单个端点. 如果你在类上使用了它,则不需在方法上再次声明,但可以用它来进一步自定义该特定端点的计时器.
- ③ 在有 `longTask = true` 的方法上,为该方法启用长任务计时器.
长任务计时器需要单独的指标名称,并且可以使用短任务计时器进行堆叠.

默认情况下,使用名为 `http.server.requests` 生成指标. 可以通过设置 `management.metrics.web.server.requests-metric-name` 属性来自定义名称.

默认情况下,Jersey 服务器指标使用以下标签标记:

标签	描述
<code>exception</code>	处理请求时抛出的异常的简单类名.
<code>method</code>	请求的方法 (例如, <code>GET</code> 或 <code>POST</code>)
<code>outcome</code>	根据响应状态码生成的请求结果. <code>1xx</code> 是 <code>INFORMATIONAL</code> , <code>2xx</code> 是 <code>SUCCESS</code> , <code>3xx</code> 是 <code>REDIRECTION</code> , <code>4xx</code> 是 <code>CLIENT_ERROR</code> , <code>5xx</code> 是 <code>SERVER_ERROR</code>

标签	描述
<code>status</code>	响应的 HTTP 状态码 (例如, <code>200</code> 或 <code>500</code>)
<code>uri</code>	如果可能, 在变量替换之前请求 URI 模板 (例如, <code>/api/person/{id}</code>)

要自定义标签, 请提供一个实现了 `JerseyTagsProvider` 的 `@Bean`.

HTTP Client 指标

Spring Boot Actuator 管理 `RestTemplate` 和 `WebClient` 的指标记录. 为此, 你必须注入一个自动配置的 `builder` 并使用它来创建实例:

- `RestTemplateBuilder` 用于 `RestTemplate`
- `WebClient.Builder` 用于 `WebClient`

也可以手动指定负责此指标记录的自定义程序, 即 `MetricsRestTemplateCustomizer` 和 `MetricsWebClientCustomizer`.

默认情况下, 使用名为 `http.client.requests` 生成指标. 可以通过设置 `management.metrics.web.client.requests-metric-name` 属性来自定义名称.

默认情况下, 通过检测的客户端生成的指标会标记以下信息:

标签	描述
<code>clientName</code>	<code>URI</code> 的主机部分
<code>method</code>	请求的方法 (例如, <code>GET</code> 或 <code>POST</code>).
<code>outcome</code>	根据响应状态码生成的请求结果. <code>1xx</code> 是 <code>INFORMATIONAL</code> , <code>2xx</code> 是 <code>SUCCESS</code> , <code>3xx</code> 是 <code>REDIRECTION</code> , <code>4xx</code> 是 <code>CLIENT_ERROR</code> , <code>5xx</code> 是 <code>SERVER_ERROR</code>
<code>status</code>	响应的 HTTP 状态码 (例如, <code>200</code> 或 <code>500</code>), 如果有 I/O 问题, 则为 <code>IO_ERROR</code> ; 否则为 <code>CLIENT_ERROR</code>

标签	描述
<code>uri</code>	如果可能, 在变量替换之前请求 URI 模板 (例如, <code>/api/person/{id}</code>)

要根据你选择的客户端自定义标签, 你可以提供一个实现了 `RestTemplateExchangeTagsProvider` 或 `WebClientExchangeTagsProvider` 的 `@Bean`. `RestTemplateExchangeTags` 和 `WebClientExchangeTags` 中有便捷的静态函数.

Cache 指标

在启动时, 自动配置启动所有可用 `Cache` 的指标记录功能, 指标以 `cache` 为前缀.
缓存指标记录针对一组基本指标进行了标准化. 此外, 还提供了缓存特定的指标.

支持以下缓存库:

- Caffeine
- EhCache 2
- Hazelcast
- 所有兼容 JCache (JSR-107) 的实现
- Redis

指标由缓存的名称和从 `bean` 名称扩展的 `CacheManager` 的名称标记.



只有启动时可用的缓存才会绑定到注册表.

对于未在缓存配置中定义的缓存, 例如在启动阶段之后以编程方式创建的缓存, 需要显式注册. 可用 `CacheMetricsRegistrar` bean 简化该过程.

DataSource 指标

通过自动配置, 可以使用前缀为 `jdbc.connections` 的指标来检测所有可用的 `DataSource` 对象. 数据源指标记录会生成表示池中当前 `active`、大允许和最小允许连接的计量器 (gauge). 指标还标记有基于 `bean` 名称计算的 `DataSource` 名称.

指标也由基于 `bean` 名称计算的 `DataSource` 的名称标记.



默认情况下, Spring Boot 为所有支持的数据源提供了元数据.

如果开箱即用不支持你喜欢的数据源, 则可以添加其他

`DataSourcePoolMetadataProvider` bean. 有关示例, 请参阅
`DataSourcePoolMetadataProvidersConfiguration`.

此外, Hikari 特定的指标用 `hikaricp` 前缀暴露. 每个指标都由池名称标记 (可以使用 `spring.datasource.name` 控制) .

Hibernate 指标

自动配置启用所有可用 `Hibernate EntityManagerFactory` 实例的指标记录功能, 这些实例使用名为 `hibernate` 的指标统计信息.

指标也由从 `bean` 名称扩展的 `EntityManagerFactory` 的名称标记.

要启用信息统计, 必须将标准 JPA 属性 `hibernate.generate_statistics` 设置为 `true`. 你可以在自动配置的 `EntityManagerFactory` 上启用它, 如下所示:

```
spring:  
  jpa:  
    properties:  
      "[hibernate.generate_statistics]": true
```

RabbitMQ 指标

自动配置将使用名为 `rabbitmq` 的指标启用对所有可用 RabbitMQ 连接工厂进行指标记录.

Kafka Metrics

自动配置将分别为消费者工厂和生产者工厂注册 `MicrometerConsumerListener` 和 `MicrometerProducerListener`. 它还将为 `StreamsBuilderFactoryBean` 注册一个 `KafkaStreamsMicrometerListener`. 有关更多详细信息, 请参阅 Spring Kafka 文档的 `Micrometer Native Metrics` 部分.

6.6.4. 注册自定义指标

要注册自定义指标, 请将 `MeterRegistry` 注入你的组件中, 如下所示:

```

class Dictionary {

    private final List<String> words = new CopyOnWriteArrayList<>();

    Dictionary(MeterRegistry registry) {
        registry.gaugeCollectionSize("dictionary.size", Tags.empty(),
            this.words);
    }

    // ...
}

```

如果您的指标依赖于其他 bean，则建议您使用 **MeterBinder** 进行注册，如以下示例所示：

```

@Bean
MeterBinder queueSize(Queue queue) {
    return (registry) -> Gauge.builder("queueSize",
queue::size).register(registry);
}

```

使用 **MeterBinder** 可以确保设置正确的依赖关系，并且在获取指标值时 Bean 可用。

默认情况下，所有 **MeterBinder** bean 的指标都将自动绑定到 Spring 管理的 **MeterRegistry**。如果你发现跨组件或应用程序重复记录一套指标，则 **MeterBinder** 实现也可能很有用。

6.6.5. 自定义单个指标

如果需要将自定义应用于特定的仪表实例，则可以使用

io.micrometer.core.instrument.config.MeterFilter 接口。默认情况下，所有 **MeterFilter** bean 都将自动应用于 micrometer **MeterRegistry.Config**。

例如，如果要将所有以 **com.example** 开头的仪表ID的 **mytag.region** 标签重命名为 **mytag.area**，则可以执行以下操作：

```

@Bean
public MeterFilter renameRegionTagMeterFilter() {
    return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");
}

```

常用标签

通用标签通常用于在操作环境（如主机，实例，区域，堆栈等）上进行维度深入分析。

通用标签适用于所有仪表，并可以按以下示例所示进行配置：

```
management:
  metrics:
    tags:
      region: "us-east-1"
      stack: "prod"
```

上面的示例将 `region` 和 `stack` 标签添加到所有 `meter` 中，其值分别为 `us-east-1` 和 `prod`。



如果你使用 Graphite，那么标签的顺序很重要。

由于使用此方法无法保证通用标签的顺序，因此建议 Graphite 用户定义自定义 `MeterFilter`。

Per-meter 属性

除了 `MeterFilter` bean 之外，还可以使用 `properties` 在 `per-meter` 基础上自定义。

`Per-meter` 定义适用于以给定名称开头的所有 `meter` ID。例如，以下将禁用任何以 `example.remote` 开头的 `ID` 的 `meter`：

```
management:
  metrics:
    enable:
      example:
        remote: false
```

以下属性允许 `per-meter` 自定义：

Table 9. `Per-meter` 自定义

属性	描述
<code>management.metrics.enable</code>	是否拒绝 <code>meter</code> 发布任何指标。
<code>management.metrics.distribution.per-centiles-histogram</code>	是否发布一个适用于计算可聚合（跨维度）的百分比近似柱状图。

属性	描述
<code>management.metrics.distribution.minimum-expected-value,</code> <code>management.metrics.distribution.maximum-expected-value</code>	通过限制预期值的范围来发布较少的柱状图桶.
<code>management.metrics.distribution.percentiles</code>	发布在你自己的应用程序中计算的百分比数值
<code>management.metrics.distribution.slo</code>	发布包含服务级别目标定义的存储区的累积直方图.

有关 `percentiles-histogram`、`percentiles` 和 `slo` 概念的更多详细信息, 请参阅 "柱状图与百分位数" 部分的文档.

6.6.6. 指标端点

Spring Boot 提供了一个 `metrics` 端点, 可以在诊断中用于检查应用程序收集的指标.

默认情况下端点不可用, 必须手动暴露, 请参阅 [暴露端点](#) 以获取更多详细信息.

访问 `/actuator/metrics` 会显示可用的 `meter` 名称列表. 你可以查看某一个 `meter` 的信息, 方法是将其名称作为选择器, 例如, `/actuator/metrics/jvm.memory.max`.



你在此处使用的名称应与代码中使用的名称相匹配, 而不是在命名约定规范化后的名称 — 为了发送到监控系统. 换句话说, 如果 `jvm.memory.max` 由于 Prometheus 命名约定而显示为 `jvm_memory_max`, 则在审计指标端点中的 `metrics` 时, 应仍使用 `jvm.memory.max` 作为选择器.

你还可以在 URL 的末尾添加任意数量的 `tag=KEY:VALUE` 查询参数, 以便多维度向下钻取 `meter`, 例如 `/actuator/metrics/jvm.memory.max?tag=area:nonheap`.



报告的测量值是与 `meter` 名称和已应用的任何标签匹配的所有 `meter` 的统计数据的总和。因此，在上面的示例中，返回的 `Value` 统计信息是堆的 `Code Cache`, `Compressed Class Space` 和 `Metaspace` 区域的最大内存占用量的总和。如果你只想查看 `Metaspace` 的最大大小，可以添加一个额外的 `tag=id:Metaspace`，即 `/actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace`。

6.7. 审计

一旦 Spring Security 生效，Spring Boot Actuator 就拥有一个灵活的审计框架，它可以发布事件（默认情况下，“authentication success”，“failure” 和 “access denied” 例外）。此功能对事件报告和基于身份验证失败实现一个锁定策略非常有用。

可以通过在应用程序的配置中提供类型为 `AuditEventRepository` 的 bean 来启用审核。为了方便起见，Spring Boot 提供了一个 `InMemoryAuditEventRepository`。
`InMemoryAuditEventRepository` 具有有限的功能，我们建议仅将其用于开发环境。对于生产环境，请考虑创建自己的替代 `AuditEventRepository` 实现。

6.7.1. Custom Auditing

你可以提供自己的 `AbstractAuthenticationAuditListener` 和 `AbstractAuthorizationAuditListener` 实现。

你还可以将审计服务用于自己的业务事件。为此，请将现有的 `AuditEventRepository` 注入自己的组件并直接使用它或使用 Spring `ApplicationEventPublisher`（通过实现 `ApplicationEventPublisherAware`）发布 `AuditApplicationEvent`。

6.8. HTTP 追踪

可以通过在应用程序的配置中提供 `HttpTraceRepository` 类型的 Bean 来启用 HTTP 跟踪。为了方便起见，Spring Boot 默认提供了一个 `InMemoryHttpTraceRepository`，用于存储最近 100 次请求-响应交换的跟踪。与其他跟踪解决方案相比，`InMemoryHttpTraceRepository` 受到限制，我们建议仅将其用于开发环境。对于生产环境，建议使用可用于生产的跟踪或可观察性解决方案，例如 `Zipkin` 或 `Spring Cloud Sleuth`。

或者, 创建自己的 `HttpTraceRepository` 来满足您的需求.

`httptrace` 端点可用于获取有关存储在 `HttpTraceRepository` 中的请求-响应交换的信息.

6.8.1. 自定义 HTTP 追踪

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.

对于高级定制, 请考虑注册自己的 `HttpExchangeTracer` 实现.

6.9. 进程监控

在 `spring-boot` 模块中, 你可以找到两个类来创建文件, 他们通常用于进程监控:

- `ApplicationPidFileWriter` 创建一个包含应用程序 PID 的文件
(默认在应用程序目录中, 文件名为 `application.pid`) .
- `WebServerPortFileWriter` 创建一个或多个文件, 其包含正在运行的 Web 服务器的端口 (默认在应用程序目录中, 文件名为 `application.port`) .

默认情况下, 这些 `writer` 未激活, 但你可以启用:

- 扩展配置
- 编程方式

6.9.1. 扩展配置

你可以在 `META-INF/spring.factories` 文件中激活生成和写入 PID 文件的监听器 (Listener) , 如下所示:

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.context.ApplicationPidFileWriter,\
org.springframework.boot.web.context.WebServerPortFileWriter
```

6.9.2. 编程方式

你还可以通过调用 `SpringApplication.addListeners(...)` 方法并传递相应的 `Writer` 对象来激活监听器. 此方法还允许你在 `Writer` 构造方法中自定义文件名和路径.

6.10. Cloud Foundry 支持

当你部署到一个兼容 Cloud Foundry 的实例时, Spring Boot 的 Actuator 模块包含的其他支持将被激活. `/cloudfoundryapplication` 路径为所有 `@Endpoint` bean 提供了另外一个安全路由.

该扩展支持允许使用 Spring Boot Actuator 信息扩充 Cloud Foundry 管理 UI (例如可用于查看已部署应用的 Web 应用) . 比如, 应用程序状态页面可以包括完整的健康信息而不是常见的 running 或 stop 状态.



常规用户无法直接访问 `/cloudfoundryapplication` 路径. 为了能访问端点, 你必须在请求时传递一个有效的 UAA 令牌.

6.10.1. 禁用 Cloud Foundry Actuator 扩展支持

如果要完全禁用 `/cloudfoundryapplication` 端点, 可以将以下设置添加到 `application.properties` 文件中:

`application.yaml`

```
management:  
  cloudfoundry:  
    enabled: false
```

6.10.2. Cloud Foundry 自签名证书

默认情况下, `/cloudfoundryapplication` 端点的安全验证会对各种 Cloud Foundry 服务进行 SSL 调用. 如果你的 Cloud Foundry UAA 或 Cloud Controller 服务使用自签名证书, 则需要设置以下属性:

`application.yaml`

```
management:  
  cloudfoundry:  
    skip-ssl-validation: true
```

6.10.3. 自定义上下文路径

如果服务器的 `context-path` 已配置为 `/` 以外的其他内容，则 Cloud Foundry 端点将无法在应用程序的根目录中使用。例如，如果 `server.servlet.context-path=/app`，Cloud Foundry 端点将在 `/app/cloudfoundryapplication/*` 上可用。

如果你希望 Cloud Foundry 端点始终在 `/cloudfoundryapplication/*` 上可用，则无论服务器的 `context-path` 如何，你都需要在应用程序中明确配置它。配置因使用的 Web 服务器而有所不同。针对 Tomcat，可以添加以下配置：

```

@Bean
public TomcatServletWebServerFactory servletWebServerFactory() {
    return new TomcatServletWebServerFactory() {

        @Override
        protected void prepareContext(Host host, ServletContextInitializer[]
initializers) {
            super.prepareContext(host, initializers);
            StandardContext child = new StandardContext();
            child.addLifecycleListener(new Tomcat.FixContextListener());
            child.setPath("/cloudfoundryapplication");
            ServletContainerInitializer initializer =
getServletContextInitializer(getContextPath());
            child.addServletContainerInitializer(initializer,
Collections.emptySet());
            child.setCrossContext(true);
            host.addChild(child);
        }

    };
}

private ServletContainerInitializer getServletContextInitializer(String
contextPath) {
    return (c, context) -> {
        Servlet servlet = new GenericServlet() {

            @Override
            public void service(ServletRequest req, ServletResponse res) throws
ServletException, IOException {
                ServletContext context =
req.getServletContext().getContext(contextPath);

                context.getRequestDispatcher("/cloudfoundryapplication").forward(req, res);
            }

        };
        context.addServlet("cloudfoundry", servlet).addMapping("/*");
    };
}

```

6.11. 下一步

如果你想了解本章中讨论的一些概念,你可以查看 `actuator` 示例应用程序. 或许你还想了解 `Graphite` 等图形工具的相关知识.

此外,你可以继续阅读应用‘[部署选项](#)’相关内容,或继续阅读有关[Spring Boot 构建工具插件](#)的相关内容.

Chapter 7. 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很多选择。您可以将 Spring Boot 应用程序部署到各种云平台，容器镜像（例如Docker）或虚拟机/真实机上。

本节介绍一些更常见的部署方案。

7.1. 打包成容器

如果从容器中运行应用程序，则可以使用可执行 jar，但是将其爆炸并以其他方式运行通常也是一个优点。某些 PaaS 实施也可能选择在运行存档之前将其解压缩。例如，Cloud Foundry 以这种方式运行。运行解压缩存档的最简单方法是启动相应的启动器，如下所示：

```
$ jar -xf myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```



在应用程序的 main 方法上使用 `JarLauncher` 可以预测类路径加载顺序。该 jar 包含一个 `classpath.idx` 文件，该文件由 `JarLauncher` 在构造类路径时使用。

还可以通过为依赖以及应用程序类和资源（通常会更频繁地更改）[creating separate layers](#) 来创建更有效的容器镜像。

7.2. 部署到云端

Spring Boot 的可执行 jar 已为大多数流行的云 PaaS（平台即服务）提供商提供。这些提供程序往往要求您“自带容器”。他们管理应用程序流程（不是专门用于 Java 应用程序），因此他们需要一个中间层，以使您的应用程序适应云中正在运行的流程的概念。

两家受欢迎的云提供商，Heroku 和 Cloud Foundry，采用了“buildpack”方法。buildpack 将部署的代码包装在启动应用程序所需的任何内容中。它可能是 JDK，可能是对 Java，嵌入式 Web 服务器或成熟的应用程序服务器的调用。一个 buildpack 是可插入的，但是理想情况下，您应该能够通过尽可能少的自定义来获得它。这减少了您无法控制的功能的占用空间。它使开发和生产环境之间的差异最小化。

理想情况下,您的应用程序像 Spring Boot 可执行 jar 一样,具有打包运行所需的一切.

在本节中,我们研究如何使在 “Getting Started” 部分中开发的简单应用程序启动并在云中运行.

7.2.1. Cloud Foundry

如果未指定其他构建包,Cloud Foundry 将提供默认的构建包. Cloud Foundry Java buildpack 对 Spring 应用程序 (包括Spring Boot) 提供了出色的支持.

您可以部署独立的可执行 jar 应用程序以及传统的 .war 打包应用程序.

一旦构建了应用程序 (例如,使用 `mvn clean package`) 并 安装了 cf 命令行工具,就可以使用 `cf push` 命令部署应用程序,并替换已编译的 `.jar` 的路径.

推送应用程序之前,请确保已使用 cf 命令行客户端登录. 下面的行显示了使用 `cf push` 命令部署应用程序:

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



在前面的示例中,我们用 `acloudyspringtime` 替换您给 cf 作为应用程序名称的任何值.

有关更多选项,请参阅 `cf push` 文档. 如果在同一目录中存在 Cloud Foundry `manifest.yml` 文件,则将其考虑.

此时,cf 开始上载您的应用程序,产生类似于以下示例的输出:

```
Uploading acloudyspringtime... OK
Preparing to start acloudyspringtime... OK
----> Downloaded app package (8.9M)
----> Java Buildpack Version: v3.12 (offline) |
https://github.com/cloudfoundry/java-buildpack.git#6f25b7e
----> Downloading Open Jdk JRE 1.8.0_121 from https://java-
buildpack.cloudfoundry.org/openjdk/trusty/x86_64/openjdk-1.8.0_121.tar.gz (found
in cache)
      Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.6s)
----> Downloading Open JDK Like Memory Calculator 2.0.2_RELEASE from
https://java-buildpack.cloudfoundry.org/memory-calculator/trusty/x86_64/memory-
calculator-2.0.2_RELEASE.tar.gz (found in cache)
      Memory Settings: -Xss349K -Xmx681574K -XX:MaxMetaspaceSize=104857K
-Xms681574K -XX:MetaspaceSize=104857K
----> Downloading Container Certificate Trust Store 1.0.0_RELEASE from
https://java-buildpack.cloudfoundry.org/container-certificate-trust-
store/container-certificate-trust-store-1.0.0_RELEASE.jar (found in cache)
      Adding certificates to .java-
buildpack/container_certificate_trust_store/truststore.jks (0.6s)
----> Downloading Spring Auto Reconfiguration 1.10.0_RELEASE from https://java-
buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-
1.10.0_RELEASE.jar (found in cache)
Checking status of app 'acloudyspringtime'...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  1 of 1 instances running (1 running)

App started
```

恭喜你！ 该应用程序现已上线！

应用程序上线后，可以使用 `cf apps` 命令验证已部署应用程序的状态，如以下示例所示：

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

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

一旦 Cloud Foundry 确认已部署了您的应用程序，您就应该能够在给定的 URI 上找到该应用程序。在前面的示例中，您可以在 <https://acloudyspringtime.cfapps.io/> 上找到它。

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作为环境变量（例如：`$VCAP_SERVICES`）暴露给应用程序。该架构决定是由于 Cloud Foundry 的多语言（可以将任何语言和平台支持为 buildpack）所决定的。过程范围的环境变量与语言无关。

环境变量并非总是使用最简单的 API，因此 Spring Boot 会自动提取它们并将数据平整为可通过 Spring 的 `Environment` 抽象访问的属性，如以下示例所示：

```
@Component
class MyBean implements EnvironmentAware {

    private String instanceId;

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

    // ...
}
```

所有 Cloud Foundry 属性均以 `vcap` 为前缀。您可以使用 `vcap` 属性来访问应用程序信息（例如，应用程序的公共 URL）和服务信息（例如，数据库凭据）。有关完整的详细信息，请参见

'CloudFoundryVcapEnvironmentPostProcessor' Javadoc.



Java CFEnv 项目更适合诸如配置数据源之类的任务。

7.2.2. Kubernetes

Spring Boot 通过检查环境中的 `"*_SERVICE_HOST"` 和 `"*_SERVICE_PORT"` 变量来自动检测 Kubernetes 部署环境。您可以使用 `configprop:spring.main.cloud-platform[]` 配置属性覆盖此检测。

Spring Boot 帮助您[管理应用程序的状态](#), 并使用 [Actuator](#) 通过 [HTTP Kubernetes 探针](#)将其导出。

Kubernetes 容器生命周期

当 Kubernetes 删除应用程序实例时, 关机过程会同时涉及多个子系统: `shutdown hooks`, 注销服务, 将实例从负载均衡器中删除... 因为这些关机进程并行发生 (并且由于分布式系统的性质) , 有一个窗口, 在此期间可以将流量路由到也已开始其关闭处理的 Pod。

您可以在 `preStop` 处理程序中配置睡眠执行, 以避免将请求路由到已经开始关闭的 Pod。此睡眠时间应足够长, 以使新请求停止路由到 Pod, 并且其持续时间因部署而异。可以通过 pod 配置文件中的 `PodSpec` 来配置 `preStop` 处理程序, 如下所示:

```
spec:  
  containers:  
    - name: example-container  
      image: example-image  
      lifecycle:  
        preStop:  
          exec:  
            command: ["sh", "-c", "sleep 10"]
```

一旦停止前挂钩完成, `SIGTERM` 将 被发送到容器, 并且将 [正常关机](#), 从而允许完成所有剩余的运行中请求。

7.2.3. Heroku

Heroku 是另一个流行的 PaaS 平台。要自定义 Heroku 构建，您需要提供一个 **Procfile**，该文件提供了部署应用程序所需的内容。Heroku 为 Java 应用程序分配了一个要使用的 **port**，然后确保可以路由到外部URI。

您必须配置您的应用程序以监听正确的端口。以下示例显示了我们的入门 REST 应用程序的 **Procfile**：

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

Spring Boot 使 **-D** 参数成为可从 Spring Environment 实例访问的属性。**server.port** 配置属性被馈送到嵌入式Tomcat, Jetty 或 Undertow 实例，然后在启动时使用该端口。
\$PORT 环境变量是由 Heroku PaaS 分配给我们的。

这应该是您需要的一切。Heroku 部署最常见的部署工作流程是 **git push** 将代码推送生产环境，如以下示例所示：

```
$ git push heroku master

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

----> Java app detected
----> Installing OpenJDK 1.8... done
----> Installing Maven 3.3.1... done
----> Installing settings.xml... done
----> Executing: mvn -B -DskipTests=true clean install

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

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

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

您的应用程序现在应该已经在 Heroku 上启动并运行了。有关更多详细信息,请参阅将[Spring Boot 应用程序部署到 Heroku](#)。

7.2.4. OpenShift

[OpenShift](#) 是 Kubernetes 容器编排平台的 Red Hat 公共（和企业）扩展。与 Kubernetes 相似,OpenShift 具有许多用于安装基于 Spring Boot 的应用程序的选项.

OpenShift 提供了许多资源来描述如何部署 Spring Boot 应用程序,包括:

- 使用[S2I构建器](#)
- [Architecture 指南](#)
- 在[WildFly](#)上作为传统的Web应用程序运行
- [OpenShift 公共简报](#)

7.2.5. Amazon Web Services (AWS)

Amazon Web Services 提供了多种安装基于 Spring Boot 的应用程序的方式,既可以作为传统的 Web 应用程序 (war) ,也可以作为具有嵌入式 Web 服务器的可执行 jar 文件安装. 选项包括:

- AWS Elastic Beanstalk
- AWS Code Deploy
- AWS OPS Works
- AWS Cloud Formation
- AWS Container Registry

每个都有不同的功能和定价模型. 在本文档中,我们仅描述最简单的选项: AWS Elastic Beanstalk.

AWS Elastic Beanstalk

如官方的 [Elastic Beanstalk Java 指南](#) 中所述,部署Java应用程序有两个主要选项。您可以使用 “Tomcat Platform” 或 “Java SE platform”.

使用 Tomcat 平台

该选项适用于产生 war 文件的 Spring Boot 项目。无需特殊配置。
您只需要遵循官方指南即可。

使用 Java SE 平台

此选项适用于产生 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 实例来代理在端口 5000 上运行的实际应用程序。
要对其进行配置，请将以下行添加到 `application.properties` 文件：

```
server.port=5000
```

上传二进制文件而不是源文件

默认情况下，Elastic Beanstalk 上传源码并在 AWS 中进行编译。

但是，最好改为上传二进制文件。为此，请在

`.elasticbeanstalk/config.yml` 文件中添加类似于以下内容的行：

```
deploy:  
  artifact: target/demo-0.0.1-SNAPSHOT.jar
```

通过设置环境类型来降低成本

默认情况下，Elastic Beanstalk 环境是负载平衡的。

负载均衡器的成本很高。为避免该费用，请按照 [Amazon 文档中](#) 的说明将环境类型设置为“Single instance”。您还可以使用 CLI 和以下命令来创建单实例环境：

```
eb create -s
```

简介

这是使用 AWS 的最简单方法之一，但还有更多内容需要介绍，例如如何将 Elastic Beanstalk 集成到任何 CI/CD 工具中，如何使用 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。

7.2.6. Boxfuse 和 Amazon Web Services

[Boxfuse](#) 的工作原理是将您的 Spring Boot 可执行 jar 或 war 变成一个最小的 VM 镜像, 该镜像可以在 VirtualBox 或 AWS 上不变地部署. Boxfuse 与 Spring Boot 进行了深度集成, 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 在生成的镜像以及它提供的所有资源 (实例, 安全组, 弹性负载均衡器等) 中都利用了此信息.

创建 [Boxfuse 帐户](#), 将其连接到您的 AWS 帐户, 安装 Boxfuse Client 的最新版本, 并确保该应用程序已由 Maven 或 Gradle 构建 (通过使用例如 `mvn clean package`) 后, 您可以 使用与以下类似的命令将您的 Spring Boot 应用程序部署到 AWS:

```
$ boxfuse run myapp-1.0.jar -env=prod
```

有关更多选项, 请参见 [boxfuse run 文档](#). 如果当前目录中存在 `boxfuse.conf` 文件, 则将其考虑.



默认情况下, Boxfuse 在启动时会激活一个名为 `boxfuse` 的 Spring 配置文件. 如果您的可执行 jar 或 war 包含 `application-boxfuse.properties` 文件, 则 Boxfuse 的配置将基于其包含的属性.

此时, `boxfuse` 为您的应用程序创建一个镜像, 然后上传该镜像, 并在 AWS 上配置和启动必要的资源, 其输出类似于以下示例:

```
Fusing Image for myapp-1.0.jar ...
Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0
Creating axelfontaine/myapp ...
Pushing axelfontaine/myapp:1.0 ...
Verifying axelfontaine/myapp:1.0 ...
Creating Elastic IP ...
Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...
Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in eu-central-1
(this may take up to 50 seconds) ...
AMI created in 00:23.557s -> ami-d23f38cf
Creating security group boxfuse-sg_axelfontaine/myapp:1.0 ...
Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f38cf) in eu-central-1 ...
Instance launched in 00:30.306s -> i-92ef9f53
Waiting for AWS to boot Instance i-92ef9f53 and Payload to start at
https://52.28.235.61/ ...
Payload started in 00:29.266s -> https://52.28.235.61/
Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...
Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...
Deployment completed successfully. axelfontaine/myapp:1.0 is up and running at
https://myapp-axelfontaine.boxfuse.io/
```

您的应用程序现在应该已启动并在 AWS 上运行。

请参阅有关在 [EC2上部署Spring Boot应用程序](#) 的博客文章以及 [Boxfuse Spring Boot集成的文档](#)，以开始使用 Maven 构建来运行该应用程序。

7.2.7. Google Cloud

Google Cloud 有多个选项可用于启动 Spring Boot 应用程序。最容易上手的可能是 App Engine，但您也可以找到在 Container Engine 的容器中或 Compute Engine 的虚拟机上运行 Spring Boot 的方法。

要在 App Engine 中运行，您可以先在用户界面中创建一个项目，该项目将为您设置一个唯一的标识符，并还设置 HTTP 路由。将 Java 应用程序添加到项目中，并将其保留为空，然后使用 [Google Cloud SDK](#) 从命令行或 CI 构建将 Spring Boot 应用程序推送到该插槽中。

App Engine Standard 要求您使用 WAR 包装。请按照 [这些步骤](#) 将 App Engine 标准应用程序部署到 Google Cloud。

另外,App Engine Flex 要求您创建一个 `app.yaml` 文件来描述您的应用程序所需的资源。通常,您将此文件放在 `src/main/appengine` 中,它应类似于以下文件:

```
service: default

runtime: java
env: flex

runtime_config:
  jdk: openjdk8

handlers:
- url: /.*
  script: this field is required, but ignored

manual_scaling:
  instances: 1

health_check:
  enable_health_check: False

env_variables:
  ENCRYPT_KEY: your_encryption_key_here
```

您可以通过将项目 ID 添加到构建配置中来部署应用程序 (例如, 使用 Maven 插件), 如以下示例所示:

```
<plugin>
  <groupId>com.google.cloud.tools</groupId>
  <artifactId>appengine-maven-plugin</artifactId>
  <version>1.3.0</version>
  <configuration>
    <project>myproject</project>
  </configuration>
</plugin>
```

然后使用 `mvn appengine:deploy` 进行部署 (如果您需要先进行身份验证, 则构建会失败)。

7.3. 安装 Spring Boot 应用程序

除了使用 `java -jar` 运行 Spring Boot 应用程序之外, 还可以为 Unix 系统制作完全可执行的应用程序。完全可执行的 `jar` 可以像其他可执行二进制文件一样执行

, 也可以在 `init.d` 或 `systemd` 中注册. 这使得在普通生产环境中安装和管理 Spring Boot 应用程序变得非常容易.



完全可执行的 `jar` 通过将额外的脚本嵌入文件的开头来工作. 当前, 某些工具不接受此格式, 因此您可能无法始终使用此技术. 例如, `jar -xf` 可能在无提示的情况下无法提取出已完全可执行的 `jar` 或 `war`. 建议仅当您打算直接执行 `jar` 或 `war` 时才使其完全可执行, 而不是使用 `java -jar` 来运行它或将其部署到 `servlet` 容器中.



不能使 `zip64` 格式的 `jar` 文件完全可执行.
尝试这样做将导致直接或使用 `java -jar` 执行时将 `jar` 文件报告为已损坏. 包含一个或多个 `zip64` 格式嵌套 `jar` 的标准格式 `jar` 文件可以完全执行.

要使用 Maven 创建 “完全可执行” 的 `jar`, 请使用以下插件配置:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <executable>true</executable>
    </configuration>
</plugin>
```

以下示例显示了等效的 Gradle 配置:

```
bootJar {
    launchScript()
}
```

然后, 您可以通过输入 `./my-application.jar` (其中 `my-application` 是 artifacts 的名称) 来运行您的应用程序. 包含 `jar` 的目录用作应用程序的工作目录.

7.3.1. 支持的操作系统

默认脚本支持大多数 Linux 发行版, 并已在 CentOS 和 Ubuntu 上进行了测试. 其他平台, 例如 OS X 和 FreeBSD, 则需要使用自定义的 `EmbeddedLaunchScript`.

7.3.2. Unix/Linux 服务

通过使用 `init.d` 或 `systemd`, 可以轻松地将 Spring Boot 应用程序作为 Unix/Linux 服务启动.

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, 并且不使用自定义的 `EmbeddedLaunchScript`, 则您的应用程序可以用作 `init.d` 服务. 为此, 将 jar 链接到 `init.d` 以支持标准的 `start`, `stop`, `restart`, 和 `status` 命令.

该脚本支持以下功能:

- 以拥有 jar 文件的用户身份启动服务
- 使用 `/var/run/<appname>/<appname>.pid` 跟踪应用程序的 PID
- 将控制台日志写入 `/var/log/<appname>.log`

假设您在 `/var/myapp` 中安装了 Spring Boot 应用程序, 要将 Spring Boot 应用程序安装为 `init.d` 服务, 请创建一个符号链接, 如下所示:

```
$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp
```

安装后, 您可以按照通常的方式启动和停止服务. 例如, 在基于 Debian 的系统上, 可以使用以下命令启动它:

```
$ service myapp start
```



如果您的应用程序无法启动, 请检查写入 `/var/log/<appname>.log` 的日志文件中是否有错误.

您还可以使用标准操作系统工具将应用程序标记为自动启动. 例如, 在 Debian 上, 您可以使用以下命令:

```
$ update-rc.d myapp defaults <priority>
```

保护 `init.d` 服务



以下是一组有关如何保护作为 `init.d` 服务运行的 Spring Boot 应用程序的准则.

它并不旨在详尽列出增强应用程序及其运行环境所应进行的所有工作.

当以 `root` 身份执行时（例如使用 `root` 来启动 `init.d` 服务时），默认的可执行脚本以 `RUN_AS_USER` 环境变量中指定的用户身份运行应用程序。如果未设置环境变量，则使用拥有 `jar` 文件的用户。您永远不要以 `root` 用户身份运行 Spring Boot 应用程序，因此 `RUN_AS_USER` 绝不应该是 `root` 用户，并且应用程序的 `jar` 文件也绝不应该由 `root` 用户拥有。而是创建一个特定用户来运行您的应用程序并设置 `RUN_AS_USER` 环境变量，或使用 `chown` 使其成为 `jar` 文件的所有者，如以下示例所示：

```
$ chown bootapp:bootapp your-app.jar
```

在这种情况下，默认的可执行脚本以 `bootapp` 用户身份运行该应用程序。



为了减少应用程序的用户帐户被盗的机会，您应该考虑阻止它使用登录外壳程序。例如，您可以将帐户的外壳程序设置为 `/usr/sbin/nologin`。

您还应该采取措施防止修改应用程序的 `jar` 文件。首先，配置其权限，使其不能被写入，只能由其所有者读取或执行，如以下示例所示：

```
$ chmod 500 your-app.jar
```

其次，如果您的应用程序或运行该应用程序的帐户受到威胁，您还应采取措施限制损害。

如果攻击者确实获得了访问权限，则他们可以使 `jar` 文件可写并更改其内容。

防止这种情况发生的一种方法是使用 `chattr` 使其不可变，如以下示例所示：

```
$ sudo chattr +i your-app.jar
```

这将阻止任何用户（包括 `root` 用户）修改 `jar`。

如果使用 `root` 来控制应用程序的服务，并且您使用 `使用 .conf 文件` 来自定义其启动，则 `root` 用户将读取并评估 `.conf` 文件。应该相应地对其进行保护。使用 `chmod`

,以便文件只能由所有者读取,并使用 `chown` 使 `root` 用户成为所有者,如以下示例所示:

```
$ chmod 400 your-app.conf
$ sudo chown root:root your-app.conf
```

作为 `systemd` 服务安装

`systemd` 是 `System V init` 系统的后继产品,现在被许多现代Linux发行版使用.

尽管您可以继续在 `systemd` 中使用 `init.d` 脚本,但也可以通过使用 `systemd` 的 '`service`' 脚本来启动Spring Boot应用程序.

假设您在 `/var/myapp` 中安装了 Spring Boot 应用程序,要将 Spring Boot 应用程序安装为 `systemd` 服务,请创建一个名为 `myapp.service` 的脚本并将其放在 `/etc/systemd/system` 目录中. 以下脚本提供了一个示例:

```
[Unit]
Description=myapp
After=syslog.target

[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```



切记更改应用程序的 `Description`, `User` 和 `ExecStart` 字段.



`ExecStart` 字段未声明脚本操作命令,这意味着默认情况下使用 `run` 命令.

请注意,与作为 `init.d` 服务运行时不同,运行应用程序的用户, `PID` 文件和控制台日志文件由 `systemd` 本身管理,因此必须通过在 '`service`' 脚本中使用适当的字段进行配置.
有关更多详细信息,请查阅 [服务单元配置手册页](#).

要将应用程序标记为在系统启动时自动启动,请使用以下命令:

```
$ systemctl enable myapp.service
```

有关更多详细信息,请参考 [man systemctl](#).

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设置通常就足够了.
如果发现无法自定义所需的内容,请使用 [EmbeddedLaunchScript](#) 选项完全编写自己的文件.

编写后自定义启动脚本

在将启动脚本写入 `jar` 文件时,自定义启动脚本的元素通常很有意义. 例如,[init.d](#) 脚本可以提供 “`description`”. 由于您已经预先了解了描述 (并且无需更改), 因此在生成`jar`时也可以提供它.

要自定义书面元素,请使用 [Spring Boot Maven](#) 插件的 [embeddedLaunchScriptProperties](#) 选项或 [Spring Boot Gradle](#)插件的 [launchScript](#) 的 `properties` 属性.

默认脚本支持以下属性替换:

Name	Description	Gradle default	Maven default
<code>mode</code>	The script mode.	<code>auto</code>	<code>auto</code>
<code>initIn foProv ides</code>	The <code>Provides</code> section of “INIT INFO”	<code> \${task.baseName}</code>	<code> \${project.artifactId}</code>
<code>initIn foRequ iredSt art</code>	Required-Start section of “INIT INFO”.	<code> \$remote_fs \$syslog \$network</code>	<code> \$remote_fs \$syslog \$network</code>
<code>initIn foRequ iredSt op</code>	Required-Stop section of “INIT INFO”.	<code> \$remote_fs \$syslog \$network</code>	<code> \$remote_fs \$syslog \$network</code>

Name	Description	Gradle default	Maven default
<code>initIn foDefa ultSta rt</code>	<code>Default-Start</code> <code>section of "INIT</code> <code>INFO".</code>	<code>2 3 4 5</code>	<code>2 3 4 5</code>
<code>initIn foDefa ultSto p</code>	<code>Default-Stop</code> section <code>of "INIT INFO".</code>	<code>0 1 6</code>	<code>0 1 6</code>
<code>initIn foShor tDescr ption</code>	<code>Short-Description</code> <code>section of "INIT</code> <code>INFO".</code>	Single-line version of <code> \${project.description}</code> (falling back to <code> \${task.baseName}</code>)	<code> \${project.name}</code>
<code>initIn foDesc riptio n</code>	<code>Description</code> section <code>of "INIT INFO".</code>	<code> \${project.description}</code> (falling back to <code> \${task.baseName}</code>)	<code> \${project.description}</code> (falling back to <code> \${project.name}</code>)
<code>initIn foChkc onfig</code>	<code>chkconfig</code> section of <code>"INIT INFO"</code>	<code>2345 99 01</code>	<code>2345 99 01</code>
<code>confFo lder</code>	The default value for <code>CONF_FOLDER</code>	Folder containing the jar	Folder containing the jar

Name	Description	Gradle default	Maven default
<code>inlineConfScript</code>	Reference to a file script that should be inlined in the default launch script. This can be used to set environmental variables such as <code>JAVA_OPTS</code> before any external config files are loaded		
<code>logFolder</code>	Default value for <code>LOG_FOLDER</code> . Only valid for an <code>init.d</code> service		
<code>logFilename</code>	Default value for <code>LOG_FILENAME</code> . Only valid for an <code>init.d</code> service		
<code>pidFolder</code>	Default value for <code>PID_FOLDER</code> . Only valid for an <code>init.d</code> service		
<code>pidFilename</code>	Default value for the name of the PID file in <code>PID_FOLDER</code> . Only valid for an <code>init.d</code> service		

Name	Description	Gradle default	Maven default
<code>useStartStopDaemon</code>	Whether the <code>start-stop-daemon</code> command, when it's available, should be used to control the process	<code>true</code>	<code>true</code>
<code>stopWaitTime</code>	Default value for <code>STOP_WAIT_TIME</code> in seconds. Only valid for an <code>init.d</code> service	60	60

运行时自定义脚本

对于在编写 `jar` 之后需要自定义脚本的项目, 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 <code>mode</code> . 默认值取决于 <code>jar</code> 的构建方式, 但通常是自动的 (这意味着它会通过检查 <code>init.d</code> 目录中的符号链接来尝试猜测它是否为初始化脚本). 如果要在前台运行脚本, 可以将其显式设置为服务, 以便 <code>stop start status restart</code> 命令可以运行或 <code>run</code> .
<code>RUN_AS_USER</code>	将用于运行应用程序的用户. 未设置时, 将使用拥有 <code>jar</code> 文件的用户.
<code>USE_START_STOP_DAEMON</code>	是否可以使用 <code>start-stop-daemon</code> 命令来控制该过程. 默认为 <code>true</code> .
<code>PID_FOLDER</code>	<code>pid</code> 目录的根名称 (默认为 <code>/var/run</code>) .
<code>LOG_FOLDER</code>	放置日志文件的目录的名称 (默认为 <code>/var/log</code>) .

变量	描述
CONF_FOLD ER	从中读取 <code>.conf</code> 文件的目录的名称（默认情况下与 <code>jar</code> 文件相同的目录）。
LOG_FIL ENAME	<code>LOG_FOLDER</code> 中的日志文件名（默认为 <code><appname>.log</code> ）。
APP_NAM E	应用程序的名称。如果 <code>jar</code> 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，或者您要显式设置应用程序名称，则这将很有用。
RUN_ARGS	传递给程序（Spring Boot 应用程序）的参数。
JAVA_HOME	默认情况下，使用 <code>PATH</code> 查找 <code>Java</code> 可执行文件的位置，但是如果 <code>\$JAVA_HOME/bin/java</code> 中有可执行文件，则可以显式设置它。
JAVA_OPTS	启动 <code>JVM</code> 时传递给 <code>JVM</code> 的选项。
JARFILE	<code>jar</code> 文件的显式位置，以防脚本用于启动实际上未嵌入的 <code>jar</code> 。
DEBUG	如果不为空，则在 <code>shell</code> 进程中设置 <code>-x</code> 标志，从而易于查看脚本中的逻辑。
STOP_WAIT _TIME	停止应用程序之前强制关闭的等待时间（以秒为单位）（默认为 <code>60</code> ）。



`PID_FOLDER`, `LOG_FOLDER`, 和 `LOG_FILENAME` 变量仅对 `init.d` 服务有效。对于 `systemd`, 通过使用 ‘`service`’ 脚本进行等效的自定义。有关更多详细信息，请参见 [服务单元配置手册页](#)。

除了 `JARFILE` 和 `APP_NAME`, 可以使用 `.conf` 文件配置上一节中列出的设置。该文件应位于 `jar` 文件的旁边，并且具有相同的名称，但后缀为 `.conf` 而不是 `.jar`。例如，名为 `/var/myapp/myapp.jar` 的 `jar` 使用名为 `/var/myapp/myapp.conf` 的配置文件，如以下示例所示：

`myapp.conf`

```
JAVA_OPTS=-Xmx1024M
LOG_FOLDER=/custom/log/folder
```



如果您不喜欢将配置文件放在 `jar` 文件旁边，则可以设置 `CONF_FOLDER` 环境变量以自定义配置文件的位置。

要了解有关适当保护此文件的信息,请参阅[保护init.d服务的准则](#).

7.3.3. Microsoft Windows 服务

可以使用 `winsw` 将 Spring Boot 应用程序作为 Windows 服务启动.

(一个单独维护的示例) 逐步说明了如何为 Spring Boot 应用程序创建 Windows 服务.

7.4. 下一步

请访问 [Cloud Foundry](#), [Heroku](#), [OpenShift](#) 和 [Boxfuse](#) 网站, 以获取有关 PaaS 可以提供的各种功能的更多信息. 这些只是最受欢迎的 Java PaaS 提供程序中的四个. 由于 Spring Boot 非常适合基于云的部署, 因此您也可以自由考虑其他提供商.

下一节将继续介绍 [Spring Boot CLI](#), 或者您可以继续阅读有关 [构建工具插件](#) 的信息.

Chapter 8. Spring Boot CLI

Spring Boot CLI 是一个命令行工具,如果您想快速开发 Spring 应用程序,可以使用它。它使您可以运行 Groovy 脚本,这意味着您具有类似 Java 的熟悉语法,而没有太多样板代码。您还可以引导一个新项目或为其编写自己的命令。

8.1. 安装 CLI

可以使用 SDKMAN 手动安装 Spring Boot CLI(命令行界面) (SDK Manager) 或使用 Homebrew 或 MacPorts (如果您是 OSX 用户) . 有关全面的安装说明,请参见 “Getting started” 部分中的 [getting-started.html](#) .

8.2. 使用 CLI

安装 CLI 后,可以通过输入 `spring` 并在命令行中按 Enter 来运行它。如果您不带任何参数运行 `spring`,则会显示一个简单的帮助屏幕,如下所示:

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

Available commands are:

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

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

您可以输入 `spring help` 以获取有关任何受支持命令的更多详细信息,如以下示例所示:

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

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

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

version 命令提供了一种快速的方法来检查您使用的 Spring Boot 版本,如下所示:

```
$ spring version
Spring CLI v2.4.5
```

8.2.1. 使用 CLI 运行应用程序

您可以使用 **run** 命令来编译和运行 Groovy 源代码. Spring Boot CLI 是完全独立的,因此您不需要任何外部 Groovy 安装.

以下示例显示了用 Groovy 编写的 “hello world” Web 应用程序:

hello.groovy

```
@RestController
class WebApplication {

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

要编译并运行该应用程序,请输入以下命令

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

要将命令行参数传递给应用程序,请使用 `--` 将命令与 “`spring`” 命令参数分开,如以下示例所示:

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

要设置 JVM 命令行参数,可以使用 `JAVA_OPTS` 环境变量,如以下示例所示:

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



在 Microsoft Windows 上设置 `JAVA_OPTS` 时,请确保引用整个指令,例如 `set "JAVA_OPTS=-Xms256m -Xmx2048m"`. 这样做可以确保将值正确传递给流程.

推测 “`grab`” 依赖

标准 Groovy 包含一个 `@Grab` 注解,它使您可以声明对第三方库的依赖. Groovy 可以使用这种有用的技术以与 Maven 或 Gradle 相同的方式下载 jar,而无需使用构建工具.

Spring Boot 进一步扩展了该技术,并尝试根据您的代码推断出哪些库可以“`grab`”. 例如,由于先前显示的 `WebApplication` 代码使用 `@RestController` 注解,因此 Spring Boot 会获取 “Tomcat” 和 “Spring MVC”.

以下各项用作 “`grab hints`”:

Items	Grabs
<code>JdbcTemplate</code> , <code>NamedParameterJdbcTemplate</code> , <code>DataSource</code>	JDBC 应用程序.
<code>@EnableJms</code>	JMS Application.
<code>@EnableCaching</code>	Caching 抽象.
<code>@Test</code>	JUnit.

Items	Grabs
<code>@EnableRabbit</code>	RabbitMQ.
<code>extends Specification</code>	Spock 测试.
<code>@EnableBatchProcessing</code>	Spring 批处理.
<code>@MessageEndpoint @EnableIntegration</code>	Spring 集成.
<code>@Controller @RestController</code> <code>@EnableWebMvc</code>	Spring MVC + 嵌入式 Tomcat.
<code>@EnableWebSecurity</code>	Spring Security.
<code>@EnableTransactionManagement</code>	Spring 事务管理.



请参阅 Spring Boot CLI 源代码中的 `CompilerAutoConfiguration` 的子类, 以确切地了解如何应用定制.

推测 “grab” 坐标

Spring Boot 通过允许您指定不带组或版本的依赖(例如, `@Grab('freemarker')`) 来扩展 Groovy 的标准 `@Grab` 支持. 这样做可以参考 Spring Boot 的默认依赖元数据来推断 `artifacts` 的组和版本.



默认元数据与您使用的 CLI 版本相关. 仅当您移至新版本的 CLI 时, 它才会更改, 从而使您可以控制依赖的版本何时更改. 可以在 [附录中](#)找到一个表格, 其中显示了默认元数据中包含的依赖及其版本.

默认导入语句

为了帮助减少 Groovy 代码的大小, 将自动包含几个 `import` 语句. 请注意, 前面的示例如何引用 `@Component`, `@RestController`, 和 `@RequestMapping`, 而无需使用完全限定的名称或 `import` 语句.



许多 Spring 注解无需使用 `import` 语句即可工作. 在添加导入之前, 请尝试运行您的应用程序以查看失败的原因.

自动创建 Main 方法

与等效的 Java 应用程序不同,您不需要在 Groovy 脚本中包含 `public static void main(String[] args)` 方法. `SpringApplication` 是自动创建的,其中已编译的代码作为源.

自定义依赖管理

默认情况下,在解决 `@Grab` 依赖时,CLI 使用 `spring-boot-dependencies` 中声明的依赖管理. 可以使用 `@DependencyManagementBom` 注解 来配置其他依赖管理,这些依赖管理将覆盖默认的依赖管理. 注解的值应指定一个或多个Maven BOM的坐标(`groupId:artifactId:version`) .

例如,考虑以下声明:

```
@DependencyManagementBom("com.example.custom-bom:1.0.0")
```

前面的声明在 `com/example/custom-versions/1.0.0/` 下的 Maven 仓库中选择了 `custom-bom-1.0.0.pom` .

指定多个 BOM 时,它们以声明它们的顺序应用,如下例所示:

```
@DependencyManagementBom(["com.example.custom-bom:1.0.0",
    "com.example.another-bom:1.0.0"])
```

前面的示例表明, `another-bom` 中的依赖管理会覆盖 `custom-Bom` 中 的依赖管理.

您可以在可以使用 `@Grab` 的任何地方使用 `@DependencyManagementBom`. 但是,为了确保依赖性管理的顺序一致,您可以在应用程序中最多使用一次 `@DependencyManagementBom`.

8.2.2. 具有多个源文件的应用程序

您可以对所有接受文件输入的命令使用 “shell globbing” .
这样可以使您从单个目录使用多个文件,如以下示例所示:

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

8.2.3. 打包你的应用程序

您可以使用 `jar` 命令将应用程序打包到一个独立的可执行 `jar` 文件中, 如以下示例所示:

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

生成的 `jar` 包含通过编译应用程序产生的类以及应用程序的所有依赖, 以便随后可以使用 `java -jar` 来运行它. `jar` 文件还包含来自应用程序的类路径的条目. 您可以使用 `--include` 和 `--exclude` 添加和删除 `jar` 的显式路径. 两者都用逗号分隔, 并且都接受前缀 "+" 和 "-" 形式, 以表示应将其从默认值中删除. 默认包括以下内容:

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

默认排除项如下:

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

在命令行上输入 `spring help jar` 以获取更多信息.

8.2.4. 初始化新项目

使用 `init` 命令, 可以使用 start.spring.io 创建新项目, 而无需离开 shell, 如以下示例所示:

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

前面的示例使用 `my-project` 目录创建一个基于 Maven 的项目, 该项目使用 `spring-boot-starter-web` 和 `spring-boot-starter-data-jpa`. 您可以使用 `--list` 标志列出服务的功能, 如以下示例所示:

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

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

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

...
```

`init` 命令支持许多选项。请参阅 `help` 输出以获取更多详细信息。例如
，以下命令创建一个使用 `Java 8` 和 `war` 打包的 `Gradle` 项目：

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

8.2.5. 使用嵌入式 shell

`Spring Boot` 包含用于 `BASH` 和 `zsh Shell` 的命令行完成脚本。如果您不使用这两个
`shell` 程序(也许您是Windows用户)，则可以使用 `shell` 命令启动集成 `shell` 程序
，如以下示例所示：

```
$ spring shell
Spring Boot (v2.4.5)
Hit TAB to complete. Type '\help' and hit RETURN for help, and '\exit' to quit.
```

在内部使用嵌入式 shell 程序,您可以直接运行其他命令:

```
$ version  
Spring CLI v2.4.5
```

嵌入式 shell 支持 ANSI 颜色输出以及 tab 补全. 如果需要运行本地命令,则可以使用 ! 字首. 要退出嵌入式外壳,请按 **ctrl-c**.

8.2.6. 将扩展添加到CLI

您可以使用 **install** 命令将扩展添加到 CLI. 该命令采用格式为 **group:artifact:version** 的一组或多组 artifacts 坐标,如以下示例所示:

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

除了安装由您提供的坐标标识的 artifacts 之外,还将安装所有 artifacts 的依赖.

要卸载依赖,请使用 **uninstall** 命令. 与**install**命令一样,它以 **group:artifact:version** 的格式获取一组或多组 artifacts 坐标,如以下示例所示:

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

它将卸载由您提供的坐标及其依赖标识的 artifacts .

要卸载所有其他依赖,可以使用 **--all** 选项,如以下示例所示:

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

8.3. 使用Groovy Beans DSL开发应用程序

Spring Framework 4.0 对 **beans{}** “DSL”(从 **Grails** 来) 具有本地支持 ,并且您可以使用相同的格式将 bean 定义嵌入 Groovy 应用程序脚本中. 有时这是包括外部功能(如中间件声明)的好方法,如以下示例所示:

```

@Configuration(proxyBeanMethods = false)
class Application implements CommandLineRunner {

    @Autowired
    SharedService service

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

}

import my.company.SharedService

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

```

您可以将类声明与 `beans{}` 混合在同一文件中，只要它们位于顶层即可；或者，如果愿意，可以将 bean DSL 放在单独的文件中。

8.4. 使用 `settings.xml` 配置CLI

Spring Boot CLI 使用 Maven 的依赖解析引擎 Aether 来解决依赖。CLI 使用 `~/.m2/settings.xml` 中的 Maven 配置来配置 Aether。CLI 遵循以下配置设置：

- Offline
- Mirrors
- Servers
- Proxies
- Profiles
 - Activation
 - Repositories
- Active profiles

有关更多信息,请参见 [Maven的设置文档](#).

8.5. 下一步

GitHub 仓库中提供了一些 [示例 groovy 脚本](#),您可以使用它们来试用 Spring Boot CLI.
在整个 [源代码](#)中也有大量的Javadoc.

如果发现达到了 CLI 工具的极限,则可能需要考虑将应用程序转换为完整的 Gradle 或
Maven 构建的 "Groovy项目". 下一部分将介绍Spring Boot的 "[构建工具插件](#)
",您可以将其与 Gradle 或 Maven 一起使用.

Chapter 9. 构建工具插件

Spring Boot 为 Maven 和 Gradle 提供了构建工具插件。插件提供了多种功能，包括可执行 jar 的打包。本节提供了有关这两个插件的更多详细信息，以及在扩展不受支持的构建系统时所需的一些帮助。如果您刚刚入门，则可能需要先阅读“[using-spring-boot.html](#)”部分中的“[using-spring-boot.html](#)”。

9.1. Spring Boot Maven 插件

Spring Boot Maven 插件在 Maven 中提供了 Spring Boot 支持，使您可以打包可执行 jar 或 war 归档文件并“in-place”运行应用程序。要使用它，必须使用 Maven 3.2（或更高版本）。

请参阅插件的文档以了解更多信息：

- Reference ([HTML](#) 和 [PDF](#))
- [API](#)

9.2. Spring Boot Gradle 插件

Spring Boot Gradle 插件在 Gradle 中提供了 Spring Boot 支持，可让您打包可执行 jar 或 war 归档文件，运行 Spring Boot 应用程序以及使用所提供的依赖管理 [spring-boot-dependencies](#)。它需要 Gradle 6 (6.3 或更高版本)：

还支持 Gradle 5.6.x，但不建议使用，在未来的版本中将删除该支持

.请参阅插件的文档以了解更多信息

- 参考 ([HTML](#) 和 [PDF](#))
- [API](#)

9.3. 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 jar。要使用该模块，您需要在中声明一个额外的 [spring-boot](#) 命名空间 [build.xml](#)，如以下示例所示：

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

您需要记住使用该 `-lib` 选项启动 Ant ,如以下示例所示:

```
$ ant -lib <directory containing spring-boot-antlib-2.4.5.jar>
```



使用 Spring Boot 部分包含将 Apache Ant 与结合使用 `spring-boot-antlib` 的更完整示例。

9.3.1. Spring Boot Ant 任务

一旦 `spring-boot-antlib` 命名空间已申报,以下附加任务:

- 使用 “`exejar`” Task
- 使用 “`findmainclass`” Task

使用 “`exejar`” Task

您可以使用该 `exejar` 任务创建一个 Spring Boot 可执行 jar. 任务支持以下属性:

属性	描述	是否需要
<code>destfile</code>	要创建的目标 jar 文件	Yes
<code>classes</code>	Java类文件的根目录	Yes
<code>start-class</code>	要运行的主要应用程序类	No (否 (默认为找到的第一个声明 <code>main</code> 方法的类))

以下嵌套元素可用于任务:

元素	描述
<code>resources</code>	一个或多个 资源集合,描述应添加到创建的 jar 文件内容中的一组 Resources .

元素	描述
lib	应将一个或多个 资源集合 添加到组成应用程序运行时依赖类路径的 <code>jar</code> 库集合中。

例子

本节显示了两个 Ant 任务示例。

Specify start-class

```
<spring-boot:exec jar 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:exec>
```

Detect start-class

```
<exec jar destfile="target/my-application.jar" classes="target/classes">
    <lib>
        <fileset dir="lib" />
    </lib>
</exec>
```

9.3.2. 使用 “findmainclass” Task

该 `findmainclass` 任务在内部 `exec` 用于查找声明的类 `main`。如有必要，您也可以在构建中直接使用此任务。支持以下属性：

属性	描述	是否需要
classesroot	Java类文件的根目录	Yes (除非 <code>mainclass</code> 指定)
mainclass	可用于短路 <code>main class</code> 搜索	No
property	应该与结果一起设置的Ant属性	No (如果未指定, 将记录结果)

例子

本节包含使用的三个示例 `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" />
```

9.4. 支持其他构建系统

如果要使用 Maven, Gradle 或 Ant 以外的构建工具, 则可能需要开发自己的插件. 可执行的 jar 需要遵循特定的格式, 某些条目需要以未压缩的形式编写 (有关详细信息, 请参见附录中的 “[可执行 jar 格式](#)” 部分) .

Spring Boot Maven 和 Gradle 插件都利用它们 `spring-boot-loader-tools` 来实际生成 jar. 如果需要, 可以直接使用此库.

9.4.1. 重新打包 Archives

要重新打包现有存档, 使其成为独立的可执行存档, 请使用 `org.springframework.boot.loader.tools.Repackager`. 该 Repackager class 采取的是指现有的 jar 或 war archive 单个构造函数的参数. 使用两种可用 `repackage()` 方法之一替换原始文件或写入新目标. 在重新打包程序运行之前, 还可以对其进行各种设置.

9.4.2. 嵌套库

重新打包 存档 时, 可以使用该 `org.springframework.boot.loader.tools.Libraries` 接口包括对依赖文件的引用. 我们 `Libraries` 这里不提供任何具体的实现, 因为它们通常是特定于构建系统的.

如果归档文件中已经包含库，则可以使用 `Libraries.NONE`.

9.4.3. 查找 Main Class

如果您不用于 `Repackager.setMainClass()` 指定主类，则重新包装器将使用 `ASM` 读取类文件，并尝试使用一种 `public static void main(String[] args)` 方法找到合适的类。如果找到多个候选者，则会引发异常。

9.4.4. 重新打包示例实现

以下示例显示了典型的重新打包实现：

```
Repackager repackager = new Repackager(sourceJarFile);
repackager.setBackupSource(false);
repackager.repackage(new Libraries() {
    @Override
    public void doWithLibraries(LibraryCallback callback) throws
IOException {
        // Build system specific implementation, callback for each
dependency
        // callback.library(new Library(nestedFile,
LibraryScope.COMPILE));
    }
});
```

9.5. 接下来阅读什么

如果您对构建工具插件的工作方式感兴趣，可以查看 `spring-boot-tools` 上的模块。可执行 `jar` 格式的更多技术细节在 [附录中](#) 介绍。

如果您有与构建相关的特定问题，可以查看 “[how-to](#)” 指南。

Chapter 10. “使用方法” 指南

本节提供了一些在使用 Spring Boot 时经常出现的常见 ‘how do I do that..’ 问题的答案。它的覆盖范围不是很详尽，但是确实覆盖了很多。

如果您有一个我们不在此讨论的特定问题，则可能需要查看 [stack overflow.com](https://stackoverflow.com) 以查看是否有人已经提供了答案。这也是询问新问题的好地方（请使用 `spring-boot` 标签）。

我们也很乐意扩展此部分。如果您想添加 “操作方法”，请向我们发送 [pull request](#)。

10.1. Spring Boot 应用程序

本部分包括与 Spring Boot 应用程序直接相关的主题。

10.1.1. 创建自己的FailureAnalyzer

`FailureAnalyzer` 这是拦截启动时将异常转化为人类可读消息（包装在）的一种好方法 `FailureAnalysis`。Spring Boot 为与应用程序上下文相关的异常，JSR-303 验证等提供了此类分析器。您也可以创建自己的。。

`AbstractFailureAnalyzer` 是一个方便的扩展，`FailureAnalyzer` 它检查要处理的异常中是否存在指定的异常类型。您可以对此进行扩展，以便您的实现只有在异常出现时才有机会处理该异常。如果由于某种原因无法处理该异常，请返回 `null` 以使另一个实现有机会处理该异常。

`FailureAnalyzer` 实现必须在 `META-INF/spring.factories` 中注册。以下示例注册 `ProjectConstraintViolationFailureAnalyzer`：

```
org.springframework.boot.diagnostics.FailureAnalyzer=\
com.example.ProjectConstraintViolationFailureAnalyzer
```



如果您需要访问 `BeanFactory` 或 `Environment`，则 `FailureAnalyzer` 可以分别实现 `BeanFactoryAware` 或 `EnvironmentAware`。

10.1.2. 自动配置故障排除

Spring Boot 自动配置会尽力 "做正确的事", 但有时会失败, 并且很难说出原因.

`ConditionEvaluationReport` 任何 Spring Boot 都有一个非常有用的功能 `ApplicationContext`. 如果启用 `DEBUG` 日志记录输出, 则可以看到它. 如果使用 `spring-boot-actuator` (请参阅 `Actuator` 一章), 那么还有一个 `conditions` 端点, 该端点以 JSON 形式呈现报告. 使用该端点来调试应用程序, 并在运行时查看 Spring Boot 添加了哪些功能 (尚未添加) .

通过查看源代码和 Javadoc, 可以回答更多问题. 阅读代码时, 请记住以下经验法则:

- 查找被调用的类 `AutoConfiguration` 并阅读其源代码. 特别注意 `@Conditional` 注解, 以了解它们启用了哪些功能以及何时启用. 添加 `--debug` 到命令行或系统属性 `-Ddebug` 以在控制台上获取在您的应用中做出的所有自动配置决策的日志. 在启用了 `Actuator` 的运行应用程序中, 查看 `conditions` 端点 (`/actuator/conditions` 或等效的JMX) 以获取相同信息.
- 查找属于 `@ConfigurationProperties` (例如 `ServerProperties`) 的类, 然后从中读取可用的外部配置选项. 该 `@ConfigurationProperties` 注解具有一个 `name` 充当前缀外部性能属性. 因此, `ServerProperties` 拥有 `prefix="server"` 和它的配置性能 `server.port, server.address` 以及其他. 在启用了 `Actuator` 的运行应用程序中, 查看 `configprops` 端点.
- 寻找对 `bind` 方法的使用, 以一种轻松的方式 `Binder` 将配置值明确地拉出 `Environment`. 它通常与前缀一起使用.
- 查找 `@Value` 直接绑定到的注解 `Environment`.
- 寻找 `@ConditionalOnExpression` 注解以响应 SpEL 表达式来打开或关闭功能, 这些注解通常使用从中解析的占位符进行评估 `Environment`.

10.1.3. 启动之前自定义环境或`ApplicationContext`

一个 `SpringApplication` 具有 `ApplicationListeners` 与 `ApplicationContextInitializers` 被用于应用自定义的上下文或环境. Spring Boot 加载了许多此类自定义项, 以供内部使用 `META-INF/spring.factories`. 注册其他自定义项的方法有多种:

- 在运行之前,通过对每个应用程序进行编程,方法是调用 `SpringApplication` 的 `addListeners` 和 `addInitializers` 方法.
- 通过设置 `context.initializer.classes` 或 `context.listener.classes` 属性,以声明的方式针对每个应用程序.
- 声明性地,对于所有应用程序,通过添加 `META-INF/spring.factories` 和打包一个 `jar` 文件,这些文件都被应用程序库.

该 `SpringApplication` 将一些特殊的 `ApplicationEvents` 发送给监听器(有些甚至在上下文创建之前),然后为 `ApplicationContext` 发布的事件注册监听器.有关完整列表,请参见‘`Spring Boot 特性`’部分中的“[应用程序事件和监听器](#)”.

还可以使用来自定义 `Environment` 刷新应用程序上下文之前的 `EnvironmentPostProcessor`. 每个实现都应在 `META-INF/spring.factories` 中注册,如以下示例所示:

```
org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPostProcessor
```

该实现可以加载任意文件并将其添加到中 `Environment`. 例如,以下示例从类路径加载 YAML 配置文件:

```

public class EnvironmentPostProcessorExample implements EnvironmentPostProcessor {
    private final YamlPropertySourceLoader loader = new
YamlPropertySourceLoader();

    @Override
    public void postProcessEnvironment(ConfigurableEnvironment environment,
SpringApplication application) {
        Resource path = new ClassPathResource("com/example/myapp/config.yml");
        PropertySource<?> propertySource = loadYaml(path);
        environment.getPropertySources().addLast(propertySource);
    }

    private PropertySource<?> loadYaml(Resource path) {
        if (!path.exists()) {
            throw new IllegalArgumentException("Resource " + path + " does not
exist");
        }
        try {
            return this.loader.load("custom-resource", path).get(0);
        }
        catch (IOException ex) {
            throw new IllegalStateException("Failed to load yaml configuration
from " + path, ex);
        }
    }
}

```

}

在 `Environment` 已经准备好了 Spring Boot

默认加载的所有常用属性源. 因此可以从环境中获取文件的位置.



前面的示例将 `custom-resource` 属性源添加到列表的末尾

, 以便在其他任何常见位置定义的 `key` 具有优先权.

自定义实现可以定义另一个顺序.



虽然在 `@SpringBootApplication` 上使用 `@PropertySource`

似乎是在环境中加载自定义资源的便捷方法, 但我们不建议您这样做.

在刷新应用程序上下文之前, 不会将此类属性源添加到 `Environment`

中. 现在配置某些属性 (如 `logging.` 和 `spring.main.`)

为时已晚, 这些属性在刷新开始之前已读取. .

10.1.4. 建立 ApplicationContext 层次结构 (添加父上下文或根上下文)

您可以使用 `ApplicationBuilder` 类创建父/子 `ApplicationContext` 层次结构。
有关更多信息,请参见 ‘Spring Boot 特性’ 部分中的 “[spring-boot-features.html](#)”
.

10.1.5. 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) . 如果要在 `main` 方法中执行一些代码,又要引导 Spring 应用程序以设置要使用的基础结构,则可以使用 Spring Boot 的 `SpringApplication` 功能. `SpringApplication` 根据是否认为需要 Web 应用程序来更改其 `ApplicationContext` 类. 您可以做的第一件事是让服务器相关的依赖 (例如 `Servlet API`) 脱离类路径. 如果你不能做到这一点 (例如 ,您从相同的代码库的两个应用程序) ,则可以在 `SpringApplication` 实例上显式调用 `setWebApplicationType(WebApplicationType.NONE)` 或设置 `applicationContextClass` 属性 (通过 Java API 或与外部属性) .

您可以将要作为业务逻辑运行的应用程序代码实现为 `CommandLineRunner` 并作为 `@Bean` 定义放到上下文中.

10.2. 属性和配置

本部分包括有关设置和读取属性,配置设置以及它们与 Spring Boot 应用程序的交互的主题.

10.2.1. 在构建时自动扩展属性

您可以使用现有的构建配置自动扩展它们,而不是对项目的构建配置中也指定的某些属性进行硬编码. 在 `Maven` 和 `Gradle` 中都是可能的.

使用 Maven 自动扩展属性

您可以使用资源过滤从 `Maven` 项目自动扩展属性. 如果使用 `spring-boot-starter-parent`,则可以使用 `@...@` 占位符引用 `Maven` 的 ‘`project properties`’ ,如以下示例所示:

```
app:
  encoding: "@project.build.sourceEncoding@"
  java:
    version: "@java.version@"
```



这样只会过滤生产配置（也就是说，不会对进行过滤 `src/test/resources`）。



如果启用该 `addResources` 标志，则 `spring-boot:run` 目标可以将 `src/main/resources` 直接添加到类路径中（用于热重载）。这样做避免了资源过滤和此功能。相反，您可以使用 `exec:java` 目标或自定义插件的配置。有关更多详细信息，请参见 [插件使用页面](#)。

如果您不使用入门级父级，则需要在 `<build/>` 元素中包括以下元素 `pom.xml`：

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

您还需要在其中包含以下元素 `<plugins/>`：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.7</version>
  <configuration>
    <delimiters>
      <delimiter>@</delimiter>
    </delimiters>
    <useDefaultDelimiters>false</useDefaultDelimiters>
  </configuration>
</plugin>
```



`useDefaultDelimiters` 如果在配置中使用标准的 Spring 占位符（例如 `${placeholder}` ），则该属性很重要。如果该属性未设置为 `false`，则可以通过构建扩展它们。

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，如以下示例所示：

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

然后，您可以使用占位符来引用 Gradle 项目的属性，如以下示例所示：

```
app:
  name: "${name}"
  description: "${description}"
```



Gradle 的 `expand` 方法使用 Groovy 的方法 `SimpleTemplateEngine` 来转换 `${..}` 令牌。该 `${..}` 风格与 Spring 自己的属性占位符机制冲突。要将 Spring 属性占位符与自动扩展一起使用，请按以下步骤对 Spring 属性占位符进行转义：`\${..}`。

10.2.2. 外部化配置 SpringApplication

`SpringApplication` 具有 `bean` 属性（主要是 `setter`），因此在创建应用程序时可以使用其 Java API 修改其行为。或者，您可以通过在中设置属性来外部化配置 `spring.main.*`。例如，在中 `application.properties`，您可能具有以下设置：

```
spring:
  main:
    web-application-type: "none"
    banner-mode: "off"
```

然后,启动时不会打印 Spring Boot 标语,并且应用程序也没有启动嵌入式 Web 服务器.

外部配置中定义的属性会覆盖用 Java API 指定的值,但用于创建 ApplicationContext 的数据源除外. 考虑以下应用程序:

```
new SpringApplicationBuilder()
    .bannerMode(Banner.Mode.OFF)
    .sources(demo.MyApp.class)
    .run(args);
```

现在考虑以下配置:

```
spring:
  main:
    sources: "com.acme.Config,com.acme.ExtraConfig"
    banner-mode: "console"
```

实际应用中,现在示出的 banner (如通过配置覆盖),并为 ApplicationContext 使用三个源(按以下顺序): demo.MyApp, com.acme.Config 和 com.acme.ExtraConfig.

10.2.3. 更改应用程序外部属性的位置

默认情况下,来自不同来源的属性将以定义的顺序添加到 Spring 的 Environment 中 (有关确切顺序,请参见 'Spring Boot 特性' 部分中的 [spring-boot-features.html](#))

.

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[format=envvar]`): 默认为 application 作为文件名的根.
- **spring.config.location** (`spring.config.location[format=envvar]`): 要加载的文件 (例如类路径资源或 URL). Environment 为此文档设置了单独的属性源,可以通过系统属性,环境变量或命令行来覆盖它.

无论您在环境中进行什么设置, Spring Boot 都将始终 `application.properties` 如上所述进行加载. 默认情况下,如果使用 YAML,则扩展名为 '`.yml`'

的文件也将添加到列表中。

Spring Boot 记录在该 `DEBUG` 级别加载的配置文件以及在该级别找不到的候选文件 `TRACE`。

请参阅 `ConfigFileApplicationListener` 以获取更多详细信息。

10.2.4. 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，则将的默认过滤器令牌 `maven-resources-plugins` 从更改 `${*}` 为 `@`（即，`@maven.token@` 而不是 `${maven.token}` ），以防止与 Spring 样式的占位符冲突。如果 `application.properties` 直接启用了 Maven 过滤，则可能还需要更改默认过滤器令牌以使用其他定界符。



在这种特定情况下，端口绑定可在 PaaS 环境（例如 Heroku 或 Cloud Foundry）中工作。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。

10.2.5. 对外部属性使用 YAML

YAML 是 JSON 的超集，因此是一种方便的语法，用于以分层格式存储外部属性，如以下示例所示：

```

spring:
  application:
    name: "cruncher"
  datasource:
    driver-class-name: "com.mysql.jdbc.Driver"
    url: "jdbc:mysql://localhost/test"
  server:
    port: 9000

```

创建一个名为 `application.yml` 的文件，并将其放在类路径的根目录中。然后添加 `snakeyaml` 到您的依赖（Maven 坐标 `org.yaml:snakeyaml`, 如果使用，则已经包含在内 `spring-boot-starter`）。将 YAML 文件解析为 Java `Map<String, Object>`（如 JSON 对象），然后 Spring Boot 将 Map 压平，使其只具有一层，并使用句号作为分隔键，这是许多人习惯使用 Java 中的 `Properties` 文件的原因。

前面的示例 YAML 对应于以下 `application.properties` 文件：

```

spring.application.name=cruncher
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000

```

有关 YAML 的更多信息，请参见“[Spring Boot 特性](#)”部分中的“[spring-boot-features.html](#)”。

10.2.6. 设置 Active Spring Profiles

Spring Environment 为此提供了一个 API，但是您通常会设置一个 System 属性 (`spring.profiles.active`) 或 OS 环境变量 (`SPRING_PROFILES_ACTIVE`)。另外，您可以使用 `-D` 参数启动应用程序（请记住将其放在主类或 jar 存档之前），如下所示：

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

在 Spring Boot 中，您还可以设置 Active 配置文件 `application.properties`，如以下示例所示：

```
spring:
  profiles:
    active: "production"
```

以这种方式设置的值将由系统属性或环境变量代替, 而不由 `SpringApplicationBuilder.profiles()` 方法替代. 因此, 后一种 Java API 可用于扩充配置文件, 而无需更改默认值.

有关更多信息, 请参见 “[Spring Boot 特性](#)” 部分中的 “[spring-boot-features.html](#)”

.

10.2.7. 根据环境更改配置

Spring Boot 支持多文档 YAML 和 Properties 文件(详细信息请参见 [处理多文档文件](#)), 可以根据激活的配置文件选择性的使用它们.

如果 YAML 文档包含 `spring.config.activate.on-profile` 配置, 则将配置文件值(以逗号分隔的配置文件列表或配置文件表达式) 输入到 Spring `Environment.acceptsProfiles()` 方法中. 如果配置文件表达式匹配, 则该文档将被包含在最终的合并中(否则, 则不包含), 如以下示例所示:

```
server:
  port: 9000
---
spring:
  config:
    activate:
      on-profile: "development"
server:
  port: 9001
---
spring:
  config:
    activate:
      on-profile: "production"
server:
  port: 0
```

在前面的示例中, 默认端口为 9000. 但是, 如果名为 ‘development’ 的 Spring profile 处于 `active` 状态, 则端口为 9001. 如果 ‘production’ 为 `active` 状态, 则该端口为

0.



文档按照它们遇到的顺序进行合并。以后的值将覆盖以前的值。

10.2.8. 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。在一个位置上没有 (而且从技术上来说不是) 所有受支持属性的详尽列表，因为他有可能来自类路径上的其他 `jar` 文件。

具有 `Actuator` 功能的正在运行的应用程序具有一个 `configprops` 端点，该端点显示了可通过访问的所有绑定和可绑定属性 `@ConfigurationProperties`。

附录中包含一个 `application.properties` 示例，其中列出了 Spring Boot 支持的最常见属性。最终列表来自搜索源代码中的 `@ConfigurationProperties` 和 `@Value` 注解，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 "[spring-boot-features.html](#)"。

10.3. 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。此功能导致许多方法问题，包括如何更改嵌入式服务器以及如何配置嵌入式服务器。本节回答这些问题。

10.3.1. 使用其他 Web 服务器

许多 Spring Boot starter 都包含默认的嵌入式容器。

- 对于 `servlet` 技术栈应用程序，通过 `spring-boot-starter-web` 包括来包括 `Tomcat` `spring-boot-starter-tomcat`，但是您可以使用 `spring-boot-starter-jetty` 或 `spring-boot-starter-undertow` 代替。
- 对于 `reactive` 技术栈的应用，`spring-boot-starter-webflux` 包括响应式堆栈的 `Netty` 通过包括 `spring-boot-starter-reactor-netty`，但您可以使用 `spring-boot-starter-tomcat`, `spring-boot-starter-jetty` 或 `spring-boot-starter-undertow` 代替。

切换到其他 HTTP 服务器时,您需要将默认依赖替换为所需的依赖. Spring Boot 为 HTTP 服务器提供了单独的 `starter`, 以帮助简化此过程.

以下 Maven 示例显示了如何排除 Tomcat 并包括 Spring MVC 的 Jetty:

```
<properties>
    <servlet-api.version>3.1.0</servlet-api.version>
</properties>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <!-- Exclude the Tomcat dependency -->
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!-- Use Jetty instead -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```



Servlet API 的版本已被覆盖,因为与 Tomcat 9 和 Undertow 2.0 不同,Jetty 9.4 不支持 Servlet 4.0.

以下 Gradle 示例显示了如何使用 Undertow 代替 Spring WebFlux 的 Reactor Netty:

```

configurations.all {
    resolutionStrategy.dependencySubstitution.all { dependency ->
        if (dependency.requested instanceof ModuleComponentSelector &&
dependency.requested.module == 'spring-boot-starter-reactor-netty') {
            dependency.useTarget("org.springframework.boot:spring-boot-starter-
undertow:$dependency.requested.version", 'Use Undertow instead of Reactor
Netty')
        }
    }
}

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-webflux'
    // ...
}

```



spring-boot-starter-reactor-netty 使用 WebClient

该类是必需的,因此即使您需要包括其他 HTTP 服务器,也可能需要保持对 Netty 的依赖.

10.3.2. 禁用 Web 服务器

如果您的类路径包含启动 Web 服务器所需的 bits,则 Spring Boot 将自动启动它.

要禁用此行为,请 `WebApplicationType` 在中配置 `application.properties`,如以下示例所示:

```

spring:
  main:
    web-application-type: "none"

```

10.3.3. 更改 HTTP 端口

在独立应用程序中,主 HTTP 端口默认为 8080 但可以使用 `server.port` (例如,在 `application.properties` System 属性中或作为 System 属性) 进行设置. 由于轻松地绑定了 `Environment` 值,因此还可以使用 `SERVER_PORT` (例如,作为 OS 环境变量) .

要完全关闭 HTTP 端点,但仍创建一个 `WebApplicationContext`,请使用 `server.port=-1` (这样做有时对测试很有用) .

有关更多详细信息,请参阅 ‘Spring Boot 特性’ 部分中的 “[spring-boot-features.html](#)” 或 [ServerProperties](#) 源代码.

10.3.4. 使用随机未分配的 HTTP 端口

要扫描可用端口 (使用 OS 本地来防止冲突), 请使用 `server.port=0`.

10.3.5. 在运行时发现 HTTP 端口

您可以从日志输出或 `WebServerApplicationContext`

通过其端口访问服务器正在运行的端口 `WebServer`. 最好的方法是确保它已初始化, 是添加一个 `@Bean` 类型 `ApplicationListener<WebServerApplicationContext>`, 然后在发布事件时将其从事件中拉出.

使用的测试 `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` 还可以通过使用 `@LocalServerPort` 注解将实际端口注入字段中, 如以下示例所示:

```
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

    @LocalServerPort
    int port;

    // ...

}
```



`@LocalServerPort` 是 `@Value("${local.server.port}")` 的元注解. 不要尝试在常规应用程序中注入端口. 如我们所见, 仅在初始化容器之后才设置该值. 与测试相反, 应早处理应用程序代码回调 (在值实际可用之前).

10.3.6. 启用 HTTP 响应压缩

`Jetty`, `Tomcat` 和 `Undertow` 支持 HTTP 响应压缩. 可以在 `application.properties`, 如下所示:

```
server:  
  compression:  
    enabled: true
```

默认情况下,响应的长度必须至少为 2048 个字节才能执行压缩. 您可以通过设置 `server.compression.min-response-size` 属性来配置此行为.

默认情况下,仅当响应的内容类型为以下之一时,它们才被压缩:

- `text/html`
- `text/xml`
- `text/plain`
- `text/css`
- `text/javascript`
- `application/javascript`
- `application/json`
- `application/xml`

您可以通过设置 `server.compression.mime-types` 属性来配置此行为.

10.3.7. 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`. 以下示例显示了在中设置 SSL 属性 `application.properties`:

```
server:  
  port: 8443  
  ssl:  
    key-store: "classpath:keystore.jks"  
    key-store-password: "secret"  
    key-password: "another-secret"
```

有关 `Ssl` 所有受支持属性的详细信息,请参见.

使用上述示例的配置意味着应用程序不再在端口 8080 上支持 HTTP 连接器。SpringBoot 不支持通过 `application.properties` 进行 HTTP 连接器和 HTTPS 连接器的配置。如果要同时拥有两者，则需要以编程方式配置其中之一。我们建议您使用 `application.properties` HTTPS 进行配置，因为 HTTP 连接器是两者中以编程方式进行配置的较容易方式。

10.3.8. 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。该支持取决于所选的 Web 服务器和应用程序环境，因为并非所有 JDK8 版本都支持该协议。



Spring Boot 不建议使用 HTTP/2 协议的明文版本 `h2c`。因此，下面的部分您必须先配置 SSL。如果您仍然选择使用 `h2c`，您可以查看 [特定章节](#)。

Tomcat HTTP/2

默认情况下，Spring Boot 随 Tomcat 9.0.x 一起提供，当使用 JDK 9 或更高版本时，Tomcat 9.0.x 支持 HTTP/2。另外，如果 `libtcnative` 库及其依赖已安装在主机操作系统上，则可以在 JDK 8 上使用 HTTP/2。

如果没有，则必须使库目录可用于 JVM 库路径。您可以使用 JVM 参数（例如）来执行此操作 `-Djava.library.path=/usr/local/opt/tomcat-native/lib`。有关更多信息，请参见 [Tomcat 官方文档](#)。

在没有该本地支持的情况下，在 JDK 8 上启动 Tomcat 9.0.x 会记录以下错误：

```
ERROR 8787 --- [           main] o.a.coyote.http11.Http11NioProtocol      : The
upgrade handler [org.apache.coyote.http2.Http2Protocol] for [h2] only supports
upgrade via ALPN but has been configured for the ["https-jsse-nio-8443"]
connector that does not support ALPN.
```

此错误不是致命错误，并且该应用程序仍以 HTTP/1.1 SSL 支持开头。

Jetty HTTP/2

从 Jetty 9.4.8 开始, [Conscrypt library](#) 库还支持 HTTP/2 . 要启用该支持 , 您的应用程序需要具有两个附加依赖: `org.eclipse.jetty:jetty-alpn-conscrypt-server` 和 `org.eclipse.jetty.http2:http2-server`.

要支持 HTTP/2, Jetty 需要具有 `org.eclipse.jetty.http2:http2-server` 依赖. 现在, 根据您的部署, 还需要选择其他依赖.

- `org.eclipse.jetty:jetty-alpn-java-server` 用于在 JDK9+ 上运行的应用程序
- `org.eclipse.jetty:jetty-alpn-openjdk8-server` 用于在 JDK8u252+ 上运行的应用程序
- `org.eclipse.jetty:jetty-alpn-conscrypt-server` 不需要 JDK, 使用 [Conscrypt library](#)

Reactor Netty HTTP/2

在 `spring-boot-webflux-starter` 默认情况下, Reactor Netty 作为服务器使用. 使用 JDK 9 或更高版本的 JDK 支持, 可以将 Reactor Netty 配置为 HTTP/2. 对于 JDK 8 环境或最佳运行时性能, 此服务器还支持带有本地库的 HTTP/2. 为此 , 您的应用程序需要具有其他依赖.

Spring Boot 管理 `io.netty:netty-tcnative-boringssl-static "uber jar"` 的版本, 其中包含所有平台的本地库. 开发人员可以选择使用分类器仅导入所需的依赖 (请参阅 [Netty官方文档](#)) .

HTTP/2 with Undertow

从 Undertow 1.4.0+ 开始, 在 JDK8 上毫无条件的支持 HTTP/2.

HTTP/2 Cleartext with supported servers

要启用 HTTP/2 的 Cleartext 支持, 需要将 `server.http2.enabled` 属性设置为 `false`, 并且应用您选择的特定服务器:

对于 Tomcat, 我们需要添加一个升级协议:

```
@Bean
public TomcatConnectorCustomizer connectorCustomizer() {
    return (connector) -> connector.addUpgradeProtocol(new Http2Protocol());
}
```

对于 Jetty, 我们需要向现有连接器添加连接工厂:

```
@Bean
public JettyServerCustomizer serverCustomizer() {
    return (server) -> {
        HttpConfiguration configuration = new HttpConfiguration();
        configuration.setSendServerVersion(false);
        Arrays.stream(server.getConnectors())
            .filter(connector -> connector instanceof ServerConnector)
            .map(ServerConnector.class::cast)
            .forEach(connector -> {
                connector.addConnectionFactory(new
HTTP2CServerConnectionFactory(configuration));
            });
    };
}
```

对于 Netty, 我们需要添加 h2c 作为支持的协议:

```
@Bean
public NettyServerCustomizer serverCustomizer() {
    return (server) -> server.protocol(HttpProtocol.H2C);
}
```

对于 Undertow, 我们需要启用 HTTP2 选项:

```
@Bean
public UndertowBuilderCustomizer builderCustomizer() {
    return (builder) -> {
        builder.setServerOption(ENABLE_HTTP2, true);
    };
}
```

10.3.9. 配置 Web 服务器

通常,您首先应该考虑使用许多可用的配置键之一,并通过在您的 `application.properties` (或 `application.yml`,或环境等) 中添加新条目来自定义 Web 服务器.

请参阅[发现外部属性的内置选项](#). 该 `server.` 命名空间在这里是非常有用的,它包括命名空间一样 `server.tomcat.`,`server.jetty.*` 和其他对服务器的特定功能.
请参阅 “[appendix-application-properties.html](#)” 的列表.

前面的部分已经介绍了许多常见的用例,例如压缩,SSL 或 HTTP/2. 但是,如果您的用例不存在配置密钥,则应查看 `WebServerFactoryCustomizer`. 您可以声明一个这样的组件,并访问与您选择的服务器相关的工厂: 您应该为所选服务器 (`Tomcat`,`Jetty`,`Reactor Netty`,`Undertow`) 和所选 Web 堆栈 (`Servlet` 或 `Reactive`) 选择对应的变体.

以下示例适用于带有 `spring-boot-starter-web` (Servlet 技术栈) 的 Tomcat :

```
@Component
public class MyTomcatWebServerCustomizer
    implements WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory factory) {
        // customize the factory here
    }
}
```



`Spring Boot` 在内部使用该机制来自动配置服务器. 自动配置的 `WebServerFactoryCustomizer` Bean 的顺序为 0,除非有明确说明的顺序,否则它将在任何用户定义的定制器之前进行处理.

一旦访问了 `WebServerFactory`,就可以经常向其添加定制程序,以配置特定的部分,例如连接器,服务器资源或服务器本身-全部使用服务器特定的 API .

此外,`Spring Boot` 还提供:

Server	Servlet stack	Reactive stack
Tomcat	TomcatServletWebServerFactory	TomcatReactiveWebServerFactory
Jetty	JettyServletWebServerFactory	JettyReactiveWebServerFactory
Undertow	UndertowServletWebServerFactory	UndertowReactiveWebServerFactory
Reactor	N/A	NettyReactiveWebServerFactory

最后,您还可以声明自己的 `WebServerFactory` 组件,该组件将覆盖 Spring Boot 提供的组件.这样做时,自动配置的定制程序仍会应用您定制的工厂,因此请谨慎使用该选项

10.3.10. 将 `Servlet,Filter,Listener` 添加到应用程序

在一个 `servlet` 栈的应用,即用 `spring-boot-starter-web`,有两种方法可以添加 `Servlet,Filter,ServletContextListener`,和由 `Servlet API` 到您的应用程序支持的其他听众:

- 使用 `Spring Bean` 添加 `Servlet,过滤器或监听器`
- 使用类路径扫描添加 `Servlet,过滤器和监听器`

使用 `Spring Bean` 添加 `Servlet,过滤器或监听器`

要使用 `Spring bean` 添加 `Servlet,Filter` 或 `Servlet *Listener`,必须为其提供 `@Bean` 定义. 当您要注入配置或依赖时,这样做非常有用. 但是,您必须非常小心,以免引起过多其他 `bean` 的急切初始化,因为必须在应用程序生命周期的早期就将它们安装在容器中. (例如,让它们依赖于您的 `DataSource` 或 `JPA` 配置不是一个好主意.) 您可以通过在第一次使用 `bean` 时(而不是在初始化时) 延迟初始化 `bean` 来解决这些限制.

对于过滤器和 `Servlet`,还可以通过添加 `FilterRegistrationBean` 或 `ServletRegistrationBean` 来代替基础组件或在基础组件之外添加映射和 `init` 参数.



如果在过滤器注册上未指定 `dispatcherType`, 则使用 `REQUEST`.
这符合 `Servlet` 规范的默认调度程序类型.

像其他任何 `Spring bean` 一样, 您可以定义 `Servlet` 过滤器 `bean` 的顺序. 请确保检查 “[spring-boot-features.html](#)” 部分.

禁用 `Servlet` 或过滤器的注册

如前所述, 任何 `Servlet` 或 `Filter Bean` 都会自动向 `Servlet` 容器注册. 要禁用特定 `Filter` 或 `Servlet Bean` 的注册, 请为其创建注册 `Bean` 并将其标记为已禁用, 如以下示例所示:

```
@Bean
public FilterRegistrationBean registration(MyFilter filter) {
    FilterRegistrationBean registration = new FilterRegistrationBean(filter);
    registration.setEnabled(false);
    return registration;
}
```

使用类路径扫描添加 `Servlet`, 过滤器和监听器

通过使用 `@ServletComponentScan` 注解 `@Configuration` 类并指定包含要注册的组件的软件包, 可以将 `@WebServlet`, `@WebFilter`, 和 `@WebListener` 注解的类自动注册到嵌入式 `Servlet` 容器中. 默认情况下, `@ServletComponentScan` 从带注解的类的包中进行扫描.

10.3.11. 配置访问日志

可以通过它们各自的命名空间为 `Tomcat`, `Undertow` 和 `Jetty` 配置访问日志.

例如, 以下设置使用 [自定义模式](#) 记录对 `Tomcat` 的访问.

```
server:
  tomcat:
    basedir: "my-tomcat"
    accesslog:
      enabled: true
      pattern: "%t %a %r %s (%D ms)"
```



日志的默认位置是相对于 Tomcat 基本目录的日志目录。默认情况下，`logs` 目录是一个临时目录，因此您可能需要修复 Tomcat 的基本目录或为日志使用绝对路径。在前面的示例中，相对于应用程序的工作目录，日志位于 `my-tomcat/logs` 中。

可以用类似的方式配置 Undertow 的访问日志，如以下示例所示：

```
server:
  undertow:
    accesslog:
      enabled: true
      pattern: "%t %a %r %s (%D ms)"
```

日志存储在相对于应用程序工作目录的 `logs` 目录中。您可以通过设置 `server.undertow.accesslog.dir` 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
      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，请参见 Jetty 文档。

10.3.12. 在前端代理服务器后面运行

如果您的应用程序在代理、负载均衡器之后或在云中运行，则请求信息（例如主机、端口、协议...）可能会随之变化。例如，您的应用程序可能正在 "10.10.10.10:8080" 上运行，但是 HTTP 客户端应该只能看到 "example.org"。

[RFC7239 "Forwarded Headers"](#) 定义了 "转发的" HTTP 请求头；代理可以使用此请求头提供有关原始请求的信息。您可以将应用程序配置为读取这些请求头，并在创建链接将其发送到 HTTP 302 响应、JSON 文档或 HTML 页面中的客户端时自动使用该信息。还有一些非标准的请求头，例如 "X-Forwarded-Host"、"X-Forwarded-Port"、"X-Forwarded-Proto"、"X-Forwarded-Ssl" 和 "X-

Forwarded-Prefix".

如果代理添加了常用的 `X-Forwarded-For` 和 `X-Forwarded-Proto` 请求头, 则将 `server.forward-headers-strategy` 设置为 `NATIVE` 以支持这些请求头. 使用此选项, Web 服务器本身就需要支持此功能. 您可以查看他们的特定文档以了解特定行为.

如果这还不够, Spring 框架会提供一个 `ForwardedHeaderFilter`. 您可以通过将 `server.forward-headers-strategy` 设置为 `FRAMEWORK` 来将其注册为 Servlet 过滤器.



如果您正在使用 Tomcat 并在代理处终止 SSL, 则应将 `server.tomcat.redirect-context-root` 设置为 `false`. 这允许在执行任何重定向之前遵守 `X-Forwarded-Proto` 头.



如果您的应用程序在 Cloud Foundry 或 Heroku 中运行, 则 `server.forward-headers-strategy` 属性默认为 `NATIVE`. 在所有其他情况下, 它默认为 `NONE`.

自定义 Tomcat 的代理配置

如果使用 Tomcat, 则可以另外配置用于携带 “forwarded” 信息的 header 名称, 如以下示例所示:

```
server:
  tomcat:
    remoteip:
      remote-ip-header: "x-your-remote-ip-header"
      protocol-header: "x-your-protocol-header"
```

Tomcat 还配置有一个默认正则表达式, 该正则表达式与要信任的内部代理匹配. 默认情况下, 信任 `10/8`, `192.168/16`, `169.254/16` 和 `127/8` 中的 IP 地址. 您可以通过在 `application.properties` 中添加一个条目来自定义阀门的配置, 如以下示例所示:

```
server:
  tomcat:
    remoteip:
      internal-proxies: "192\\.168\\\\.\\d{1,3}\\\\.\\d{1,3}"
```



您可以通过将 `internal-proxies` 设置为空来信任所有代理
(但在生产环境中不要这样做) .

您可以通过关闭自动功能来完全控制 Tomcat 的 `RemoteIpValve` 的配置 (为此,请设置 `server.forward-headers-strategy=NONE`) ,然后在 `WebServerFactoryCustomizer` bean 中添加新的 `Valve` 实例.

10.3.13. 使用Tomcat启用多个连接器

您可以将 `org.apache.catalina.connector.Connector` 添加到 `TomcatServletWebServerFactory`,这可以允许多个连接器,包括 HTTP 和 HTTPS 连接器,如以下示例所示:

```

@Bean
public WebServerFactoryCustomizer<TomcatServletWebServerFactory>
sslConnectorCustomizer() {
    return (tomcat) ->
tomcat.addAdditionalTomcatConnectors(createSslConnector());
}

private Connector createSslConnector() {
    Connector connector = new
Connector("org.apache.coyote.http11.Http11NioProtocol");
    Http11NioProtocol protocol = (Http11NioProtocol)
connector.getProtocolHandler();
    try {
        URL keystore = ResourceUtils.getURL("keystore");
        URL truststore = ResourceUtils.getURL("truststore");
        connector.setScheme("https");
        connector.setSecure(true);
        connector.setPort(8443);
        protocol.setSSLEnabled(true);
        protocol.setKeystoreFile(keystore.toString());
        protocol.setKeystorePass("changeit");
        protocol.setTruststoreFile(truststore.toString());
        protocol.setTruststorePass("changeit");
        protocol.setKeyAlias("apitester");
        return connector;
    }
    catch (IOException ex) {
        throw new IllegalStateException("Fail to create ssl connector", ex);
    }
}
}

```

10.3.14. 使用 Tomcat 的 LegacyCookieProcessor

默认情况下, Spring Boot 使用的嵌入式 Tomcat 不支持 Cookie 格式的 "版本 0", 因此您可能会看到以下错误:

```
java.lang.IllegalArgumentException: An invalid character [32] was present in the
Cookie value
```

如果有可能, 您应该考虑将代码更新为仅存储符合以后 Cookie 规范的值. 但是, 如果无法更改 cookie 的编写方式, 则可以将 Tomcat 配置为使用 [LegacyCookieProcessor](#). 要切换到 [LegacyCookieProcessor](#), 请使用添加了 [TomcatContextCustomizer](#) 的 [WebServerFactoryCustomizer](#) bean, 如以下示例所示:

```

@Bean
public WebServerFactoryCustomizer<TomcatServletWebServerFactory>
cookieProcessorCustomizer() {
    return (factory) -> factory
        .addContextCustomizers((context) -> context.setCookieProcessor(new
LegacyCookieProcessor()));
}

```

10.3.15. 启用 Tomcat 的 MBean 注册表

默认情况下,嵌入式 Tomcat 的 MBean 注册表是禁用的. 这样可以最大程度地减少 Tomcat 的内存占用. 例如,如果要使用 Tomcat 的 MBean,以便可以通过 Micrometer 暴露它们,则必须使用 `server.tomcat.mbeanregistry.enabled` 属性,如以下示例所示:

```

server:
  tomcat:
    mbeanregistry:
      enabled: true

```

10.3.16. 使用 Undertow 启用多个监听器

将 `UndertowBuilderCustomizer` 添加到 `UndertowServletWebServerFactory` 并将监听器添加到 `Builder`,如以下示例所示:

```

@Bean
public WebServerFactoryCustomizer<UndertowServletWebServerFactory>
undertowListenerCustomizer() {
    return (factory) -> {
        factory.addBuilderCustomizers(new UndertowBuilderCustomizer() {

            @Override
            public void customize(Builder builder) {
                builder.addHttpListener(8080, "0.0.0.0");
            }
        });
    };
}

```

10.3.17. 使用 `@ServerEndpoint` 创建 `WebSocket` 端点

如果要在使用嵌入式容器的 Spring Boot 应用程序中使用 `@ServerEndpoint`, 则必须声明一个 `ServerEndpointExporter @Bean`, 如以下示例所示:

```
@Bean
public ServerEndpointExporter serverEndpointExporter() {
    return new ServerEndpointExporter();
}
```

前面示例中显示的 Bean 将所有 `@ServerEndpoint` 注解的 Bean 注册到基础 WebSocket 容器。当部署到独立的 `servlet` 容器时, 此角色由 `servlet` 容器初始化程序执行, 并且不需要 `ServerEndpointExporter` Bean.

10.4. Spring MVC

Spring Boot 有许多启动器, 其中包括 Spring MVC。请注意, 一些入门者包括对 Spring MVC 的依赖, 而不是直接包含它。本部分回答有关 Spring MVC 和 Spring Boot 的常见问题。

10.4.1. 编写 JSON REST 服务

只要 Jackson2 在类路径上, Spring Boot 应用程序中的任何 Spring `@RestController` 默认情况下都应呈现 JSON 响应, 如以下示例所示:

```
@RestController
public class MyController {

    @RequestMapping("/thing")
    public MyThing thing() {
        return new MyThing();
    }
}
```

只要 Jackson 2 可以对 `MyThing` 进行序列化 (对于普通的 POJO 或 Groovy 对象为 `true`), 则 `localhost:8080/thing` 默认情况下将以 JSON 表示。请注意, 在浏览器中, 有时可能会看到 XML 响应, 因为浏览器倾向于发送更喜欢 XML 的接受请求头。

10.4.2. 编写 XML REST 服务

如果类路径上具有 Jackson XML 扩展名 (`jackson-dataformat-xml`) , 则可以使用它来呈现 XML 响应. 我们用于 JSON 的先前示例可以正常工作. 要使用 Jackson XML 渲染器, 请将以下依赖添加到您的项目中:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

如果 Jackson 的 XML 扩展名不可用而 JAXB 可用, 则可以将 XML 呈现为附加要求, 将 `MyThing` 注解为 `@XmlRootElement`, 如以下示例所示:

```
@XmlRootElement
public class MyThing {
    private String name;
    // .. getters and setters
}
```

JAXB 仅可与 Java 8 一起使用. 如果您使用的是较新的 Java 版本 , 请在项目中添加以下依赖:

```
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
</dependency>
```



要使服务器呈现 XML 而不是 JSON, 您可能必须发送一个 `Accept: text/xml` 请求头 (或使用浏览器) .

10.4.3. 自定义Jackson ObjectMapper

Spring MVC (客户端和服务器端) 使用 `HttpMessageConverters` 在 HTTP 交换中协商内容转换. 如果 Jackson 在类路径中, 则您已经获得了 `Jackson2ObjectMapperBuilder` 提供的默认转换器, 该转换器的实例已为您自动配置.

`ObjectMapper` (或用于 Jackson XML 转换器的 `XmlMapper`) 实例 (默认创建)

具有以下自定义属性:

- `MapperFeature.DEFAULT_VIEW_INCLUSION` 被禁言
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 被禁言
- `SerializationFeature.WRITE_DATES_AS_TIMESTAMPS` 被禁言

Spring Boot 还具有一些功能,可以更轻松地自定义此行为.

您可以通过使用环境来配置 `ObjectMapper` 和 `XmlMapper` 实例. Jackson

提供了一套广泛的简单的开/关功能,可用于配置其处理的各个方面. 在六个枚举 (在 Jackson 中) 中描述了这些功能,这些枚举映射到环境中的属性:

枚举	属性	值
<code>com.fasterxml.jackson.databind.DeserializationFeature</code>	<code>spring.jackson.deserialization.<feature_name></code>	<code>true, false</code>
<code>com.fasterxml.jackson.core.JsonGenerator.Feature</code>	<code>spring.jackson.generator.<feature_name></code>	<code>true, false</code>
<code>com.fasterxml.jackson.databind.MapperFeature</code>	<code>spring.jackson.mapper.<feature_name></code>	<code>true, false</code>
<code>com.fasterxml.jackson.core.JsonParser.Feature</code>	<code>spring.jackson.parser.<feature_name></code>	<code>true, false</code>
<code>com.fasterxml.jackson.databind.SerializationFeature</code>	<code>spring.jackson.serialization.<feature_name></code>	<code>true, false</code>
<code>com.fasterxml.jackson.annotation.JsonInclude.Include</code>	<code>spring.jackson.defaultAnnotation.JsonInclude.I</code> <code>property-inclusion</code>	<code>always, non_null, non_absent, non_default, non_empty</code>

例如,要启用漂亮打印,请设置 `spring.jackson.serialization.indent_output = true`. 请注意,由于使用了宽松绑定,因此 `indent_output` 的情况不必与相应的枚举常量 (即 `INDENT_OUTPUT`) 的情况匹配.

这种基于环境的配置将应用于自动配置的 `JacksonObjectMapperBuilder` Bean，并应用于使用该构建器创建的任何映射器，包括自动配置的 `ObjectMapper` Bean.

上下文的 `JacksonObjectMapperBuilder` 可以由一个或多个 `JacksonObjectMapperBuilderCustomizer` bean 进行自定义。可以对此类定制器 bean 进行排序（Boot 自己的定制器的顺序为 0），从而可以在 Boot 定制之前和之后应用其他定制。

任何类型为 `com.fasterxml.jackson.databind.Module` 的 bean 都会自动注册到自动配置的 `JacksonObjectMapperBuilder` 中，并应用于它创建的任何 `ObjectMapper` 实例。当您向应用程序添加新功能时，这提供了一种用于贡献自定义模块的全局机制。

如果要完全替换默认的 `ObjectMapper`，则可以定义该类型的 `@Bean` 并将其标记为 `@Primary`，或者，如果您更喜欢基于生成器的方法，则可以定义 `JacksonObjectMapperBuilder @Bean`。请注意，无论哪种情况，这样做都会禁用 `ObjectMapper` 的所有自动配置。

如果您提供任何类型为 `MappingJackson2HttpMessageConverter` 的 `@Bean`，它们将替换 MVC 配置中的默认值。此外，还提供了 `HttpMessageConverters` 类型的便捷 bean（如果使用默认的 MVC 配置，该 bean 始终可用）。
它提供了一些有用的方法来访问默认的和用户增强的消息转换器。

有关更多详细信息，请参见“[自定义 @ResponseBody 渲染](#)”部分和 `WebMvcAutoConfiguration` 源代码。

10.4.4. 自定义 `@ResponseBody` 渲染

Spring 使用 `HttpMessageConverters` 渲染 `@ResponseBody`（或 `@RestController` 的响应）。您可以通过在 Spring Boot 上下文中添加适当类型的 bean 来贡献额外的转换器。如果您添加的 Bean 仍是默认情况下将包含的类型（例如 JSON 转换的 `MappingJackson2HttpMessageConverter`），它将替换默认值。提供了 `HttpMessageConverters` 类型的便捷 bean，如果使用默认的 MVC 配置，它将始终可用。它提供了一些有用的方法来访问默认的和用户增强的消息转换器（例如，如果您想将它们手动注入到自定义的 `RestTemplate` 中，则可能会很有用）。

与正常的 MVC 用法一样，您提供的任何 `WebMvcConfigurer` Bean 也可以通过重写

`configureMessageConverters` 方法来贡献转换器。但是，与普通的 MVC 不同，您只能提供所需的其他转换器（因为 Spring Boot 使用相同的机制来提供其默认值）。最后，如果通过提供自己的 `@EnableWebMvc` 配置选择退出 Spring Boot 默认 MVC 配置，则可以完全控制并使用 `WebMvcConfigurationSupport` 中的 `getMessageConverters` 手动执行所有操作。

有关更多详细信息，请参见 `WebMvcAutoConfiguration` 源代码。

10.4.5. 处理分段文件上传

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 用单个请求将 Spring MVC 配置为每个文件最大大小为 1MB，最大文件数据为 10MB。您可以使用 `MultipartProperties` 类中暴露的属性覆盖这些值，存储中间数据的位置（例如，存储到 `/tmp` 目录）以及将数据刷新到磁盘的阈值。例如，如果要指定文件不受限制，请将 `spring.servlet.multipart.max-file-size` 属性设置为 -1。

当您想在 Spring MVC 控制器处理程序方法中以 `MultipartFile` 类型的 `@RequestParam` 注解参数接收多部分编码文件数据时，多部分支持会很有帮助。

有关更多详细信息，请参见 `MultipartAutoConfiguration` 源码。



建议使用容器的内置支持进行分段上传，而不要引入其他依赖，例如 Apache Commons File Upload。

10.4.6. 关闭 Spring MVC DispatcherServlet

默认情况下，所有内容均从应用程序（/）的根目录提供。如果您希望映射到其他路径，则可以如下配置：

```
spring:
  mvc:
    servlet:
      path: "/acme"
```

如果您有其他 `Servlet`，则可以为每个 `Servlet` 声明一个 `@Bean` 或 `ServletRegistrationBean` 类型，Spring Boot 会将它们透明地注册到容器中。因为

`servlet` 是通过这种方式注册的, 所以可以将它们映射到 `DispatcherServlet` 的子上下文, 而无需调用它.

自己配置 `DispatcherServlet` 是不寻常的, 但是如果您确实需要这样做, 则还必须提供 `DispatcherServletPath` 类型的 `@Bean`, 以提供自定义 `DispatcherServlet` 的路径.

10.4.7. 关闭默认的 MVC 配置

完全控制 MVC 配置的最简单方法是为您自己的 `@Configuration` 提供 `@EnableWebMvc` 注解. 这样做会使您掌握所有 MVC 配置.

10.4.8. 自定义 ViewResolvers

`ViewResolver` 是 Spring MVC 的核心组件, 将 `@Controller` 中的视图名称转换为实际的 `View` 实现. 请注意, `ViewResolvers` 主要用于 UI 应用程序, 而不是 REST 样式的服务 (`View` 不用于呈现 `@ResponseBody`). 有很多 `ViewResolver` 实现可供选择, Spring 本身对是否应使用哪个视图没有意见. 另一方面, Spring Boot

根据在类路径和应用程序上下文中找到的内容为您安装一个或两个. `DispatcherServlet` 使用它在应用程序上下文中找到的所有解析器, 依次尝试每个解析器, 直到获得结果为止. 如果添加自己的解析器, 则必须知道其顺序以及解析器的添加位置.

`WebMvcAutoConfiguration` 将以下 `ViewResolvers` 添加到您的上下文中:

- 一个名为 `defaultViewResolver` 的 `InternalResourceViewResolver`.
这一章查找可以通过使用 `DefaultServlet` 呈现的物理资源 (包括静态资源和 JSP 页面, 如果使用的话). 它在视图名称中应用前缀和后缀, 然后在 `Servlet` 上下文中查找具有该路径的物理资源 (默认值均为空, 但可通过 `spring.mvc.view.prefix` 和 `spring.mvc.view.suffix` 进行外部配置访问). 您可以通过提供相同类型的bean覆盖它.
- 名为 `beanNameViewResolver` 的 `BeanNameViewResolver`.
这是视图解析器链的有用成员, 可以拾取与要解析的视图同名的所有 bean.
不必重写或替换它.
- 仅当实际上存在 `View` 类型的bean时, 才添加一个名为 `viewResolver` 的 `ContentNegotiatingViewResolver`. 这是一个 'master' 解析器, 委派给所有其他解析器, 并尝试查找与客户端发送的 'Accept' HTTP 请求头匹配的内容. 您可能想学习有关 `ContentNegotiatingViewResolver`

的有用博客，以了解更多信息，并且您也可以查看源代码以获取详细信息。
您可以通过定义一个名为 `viewResolver` 的bean来关闭自动配置的
`ContentNegotiatingViewResolver`.

- 如果您使用 `Thymeleaf`, 则还有一个名为 `thymeleafViewResolver` 的 `ThymeleafViewResolver`. 它通过在视图名称前后加上前缀和后缀来查找资源. 前缀为 `spring.thymeleaf.prefix`, 后缀为 `spring.thymeleaf.suffix`.
前缀和后缀的值分别默认为 '`classpath:/templates/`' 和 '`.html`'.
您可以通过提供同名的 bean 来覆盖 `ThymeleafViewResolver`.
- 如果您使用 `FreeMarker`, 则还有一个名为 '`freeMarkerViewResolver`' 的 `FreeMarkerViewResolver`. 它通过在视图名称前加上前缀和后缀来在加载器路径 (已将其外部化为 `spring.freemarker.templateLoaderPath`, 其默认值为 '`classpath:/templates/`') 中查找资源. 前缀外部化为 `spring.freemarker.prefix`, 后缀外部化为 `spring.freemarker.suffix`.
前缀和后缀的默认值分别为空和 '`.ftlh`'. 您可以通过提供同名的bean来覆盖 `FreeMarkerViewResolver`.
- 如果您使用 `Groovy` 模板 (实际上,如果 `groovy-templates` 在类路径中)
, 则您还将有一个名为 `groovyMarkupViewResolver` 的 `GroovyMarkupViewResolver`. 它通过在视图名称前加上前缀和后缀 (在 `spring.groovy.template.prefix` 和 `spring.groovy.template.suffix` 中进行了扩展) 来在加载程序路径中查找资源. 前缀和后缀的默认值分别为 '`classpath:/templates/`' 和 '`.tpl`'. 您可以通过提供同名的bean来覆盖 `GroovyMarkupViewResolver`.
- 如果您使用 `Mustache`, 则还有一个名为 '`mustacheViewResolver`' 的 `MustacheViewResolver`. 它通过在视图名称前后加上前缀和后缀来查找资源. 前缀为 `spring.mustache.prefix`, 后缀为 `spring.mustache.suffix`.
前缀和后缀的值分别默认为 '`classpath:/templates/`' 和 '`.mustache`'.
您可以通过提供同名的bean来覆盖 `MustacheViewResolver`.

有关更多详细信息, 请参见以下部分:

- `WebMvcAutoConfiguration`
- `ThymeleafAutoConfiguration`
- `FreeMarkerAutoConfiguration`

- `GroovyTemplateAutoConfiguration`

10.5. 使用 Spring Security 进行测试

Spring Security 提供了对以特定用户身份运行测试的支持。例如，下面的代码段中的测试将与具有 `ADMIN` 角色的经过身份验证的用户一起运行。

```
@Test
@WithMockUser(roles="ADMIN")
public void requestProtectedUrlWithUser() throws Exception {
    mvc
        .perform(get("/"))
        ...
}
```

Spring Security 提供了与 Spring MVC Test 的全面集成，并且在使用 `@WebMvcTest` slice 和 `MockMvc` 测试控制器时也可以使用它。

有关 Spring Security 的测试支持的更多详细信息，请参阅 Spring Security 的参考文档。

10.6. Jersey

10.6.1. 使用 Spring Security 保护 Jersey 端点

可以使用 Spring Security 来保护基于 Jersey 的 Web 应用程序，其方式与用来保护基于 Spring MVC 的 Web 应用程序的方式几乎相同。但是，如果您想将 Spring Security 的方法级安全性与 Jersey 一起使用，则必须将 Jersey 配置为使用 `setStatus(int)` 而不是 `sendError(int)`。这可以防止 Jersey 在 Spring Security 有机会向客户端报告身份验证或授权失败之前提交响应。

必须在应用程序的 `ResourceConfig` bean 上将 `jersey.config.server.response.setStatusOverSendError` 属性设置为 `true`，如以下示例所示：

```

@Component
public class JerseyConfig extends ResourceConfig {

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

        setProperties(Collections.singletonMap("jersey.config.server.response.setStatusOn
verSendError", true));
    }

}

```

10.6.2. 与另一个 Web 框架一起使用 Jersey

要将 Jersey 与其他 Web 框架（例如 Spring MVC）一起使用，应对其进行配置，以便它将允许其他框架处理无法处理的请求。首先，通过将 `spring.jersey.type` 应用程序属性配置为 `filter` 值，将 Jersey 配置为使用 Filter 而不是 Servlet。其次，配置您的 `ResourceConfig` 以转发可能导致 404 的请求，如以下示例所示。

```

@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(Endpoint.class);
        property(ServletProperties.FILTER_FORWARD_ON_404, true);
    }

}

```

10.7. HTTP Clients

Spring Boot 提供了许多可与 HTTP Clients 一起使用的 starters。本节回答与使用它们有关的问题。

10.7.1. 配置 RestTemplate 使用代理

如 [spring-boot-features.html](#) 中所述，您可以将 `RestTemplateCustomizer` 与 `RestTemplateBuilder` 一起使用以构建自定义的 `RestTemplate`。建议使用此方法来创建配置为使用代理的 `RestTemplate`。

代理配置的确切详细信息取决于所使用的基础客户端请求工厂。下面的示例使用一个 `HttpClient` 配置 `HttpComponentsClientHttpRequestFactory`, 该 `HttpClient` 为除 `192.168.0.5` 之外的所有主机使用代理：

```
static class ProxyCustomizer implements RestTemplateCustomizer {

    @Override
    public void customize(RestTemplate restTemplate) {
        HttpHost proxy = new HttpHost("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create().setRoutePlanner(new
DefaultProxyRoutePlanner(proxy) {

            @Override
            public HttpHost determineProxy(HttpHost target, HttpRequest request,
HttpContext context)
                throws HttpException {
                if (target.getHostName().equals("192.168.0.5")) {
                    return null;
                }
                return super.determineProxy(target, request, context);
            }

        }).build();
        restTemplate.setRequestFactory(new
HttpComponentsClientHttpRequestFactory(httpClient));
    }

}
```

10.7.2. 配置基于 Reactor Netty 的 WebClient 使用的 TcpClient

当 Reactor Netty 在类路径上时, 将自动配置基于 Reactor Netty 的 `WebClient`.

要自定义客户端对网络连接的处理, 请提供一个 `ClientHttpConnector` bean.

下面的示例配置 60 秒的连接超时并添加 `ReadTimeoutHandler`:

```

@Bean
ClientHttpConnector clientHttpConnector(ReactorResourceFactory resourceFactory)
{
    HttpClient httpClient =
    HttpClient.create(resourceFactory.getConnectionProvider())

    .runOn(resourceFactory.getLoopResources()).option(ChannelOption.CONNECT_TIMEOUT_
    MILLIS, 60000)
        .doOnConnected((connection) -> connection.addHandlerLast(new
    ReadTimeoutHandler(60)));
    return new ReactorClientHttpConnector(httpClient);
}

```



注意将 `ReactorResourceFactory` 用于连接提供程序和事件循环资源。这确保了用于服务器接收请求和客户端发出请求的资源的有效共享。

10.8. Logging

除了通常由 Spring Framework 的 `spring-jcl` 模块提供的 Commons Logging API 之外, Spring Boot 没有强制性的日志记录依赖性。要使用 `Logback`, 您需要在类路径中包括它和 `spring-jcl`。最简单的方法是通过启动程序, 所有启动程序都依赖于 `spring-boot-starter-logging`。对于 Web 应用程序, 只需要 `spring-boot-starter-web`, 因为它暂时依赖于日志记录启动器。如果使用 Maven, 则以下依赖会为您添加日志记录:

```

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

```

Spring Boot 有一个 `LoggingSystem` 抽象, 它试图根据类路径的内容来配置日志。如果可以使用 `Logback`, 则它是首选。

如果您需要对日志记录进行的唯一更改是设置各种记录器的级别, 则可以使用 `"logging.level"` 前缀在 `application.properties` 中进行设置, 如以下示例所示:

```
logging:
  level:
    org.springframework.web: "debug"
    org.hibernate: "error"
```

您还可以使用 `logging.file.name` 来设置要写入日志的文件的位置（除了控制台）。

要配置日志记录系统的更细粒度的设置，您需要使用所讨论的 `LoggingSystem` 支持的本地配置格式。默认情况下，`Spring Boot` 从系统的默认位置（例如，用于 `Logback` 的 `classpath:logback.xml`）拾取本地配置，但是您可以使用 `logging.config` 属性设置配置文件的位置。

10.8.1. 配置 Logback Logging

如果您需要应用自定义项来进行登录，而不是使用 `application.properties` 可以实现的自定义项，则需要添加一个标准的登录配置文件。您可以将 `logback.xml` 文件添加到类路径的根目录中，以进行 `logback` 查找。如果要使用 `Spring Boot Logback 扩展`，也可以使用 `logback-spring.xml`。



`Logback` 文档有一个专用部分，其中 [详细介绍了配置](#)。

`Spring Boot` 提供了许多您自己的配置中 `included` 的 `logback` 配置。

这些包括旨在允许重新应用某些常见的 `Spring Boot` 约定。

`org/springframework/boot/logging/logback/` 下提供了以下文件：

- `defaults.xml` - 提供转换规则，模式属性和通用记录器配置。
- `console-appender.xml` - 使用 `CONSOLE_LOG_PATTERN` 添加一个 `ConsoleAppender`。
- `file-appender.xml` - 使用具有适当设置的 `FILE_LOG_PATTERN` 和 `ROLLING_FILE_NAME_PATTERN` 添加 `RollingFileAppender`。

另外，还提供了旧版 `base.xml` 文件，以与早期版本的 `Spring Boot` 兼容。

典型的自定义 `logback.xml` 文件如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
    <include resource="org/springframework/boot/logging/logback/console-
appender.xml" />
    <root level="INFO">
        <appender-ref ref="CONSOLE" />
    </root>
    <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

您的日志配置文件也可以利用 `LoggingSystem` 为您创建的 `System` 属性：

- `${PID}`: The 当前进程 ID.
- `${LOG_FILE}`: 是否在 Boot 的外部配置中设置了 `logging.file.name`.
- `${LOG_PATH}`: 是否在 Boot 的外部配置中设置了 `logging.file.path` (代表要存放日志文件的目录) .
- `${LOG_EXCEPTION_CONVERSION_WORD}`: 是否在 Boot 的外部配置中设置了 `logging.exception-conversion-word`.
- `${ROLLING_FILE_NAME_PATTERN}`: 是否在 Boot 的外部配置中设置了 `logging.pattern.rolling-file-name`.

通过使用自定义的 Logback 转换器, Spring Boot 还可以在控制台上提供一些不错的 ANSI 颜色终端输出 (但不在日志文件中) . 有关示例, 请参见 `defaults.xml` 配置中的 `CONSOLE_LOG_PATTERN`.

如果 Groovy 在类路径中, 则还应该能够使用 `logback.groovy` 配置 Logback. 如果存在, 则优先考虑此设置.



Groovy 配置不支持 Spring 扩展. 不会检测到任何 `logback-spring.groovy` 文件.

配置仅文件输出的 Logback

如果要禁用控制台日志记录并将输出仅写入文件, 则需要一个自定义的 `logback-spring.xml`, 该文件将导入 `file-appender.xml` 而不是 `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:-/tmp}}}/}spring.log"/>
    <include resource="org/springframework/boot/logging/logback/file-
appender.xml" />
    <root level="INFO">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

您还需要将 `logging.file.name` 添加到 `application.properties` 或 `application.yaml` 中, 如以下示例所示:

```
logging:
  file:
    name: "myapplication.log"
```

10.8.2. 配置 Log4j 日志

如果 Spring Boot 在类路径上, 则它支持 [Log4j 2](#) 进行日志记录配置. 如果使用 `starters` 来组装依赖, 则必须排除 `Logback`, 然后改为包括 `log4j 2`. 如果您不使用启动器, 则除了 `Log4j 2` 外, 还需要 (至少) 提供 [spring-jcl](#).

推荐的方法可能是通过启动程序, 即使它需要对排除对象进行微调. 以下示例显示了如何在 Maven 中设置 `starters`:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>

```

以下示例显示了在 Gradle 中设置 starters 的一种方法：

```

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
}

configurations.all {
    resolutionStrategy.dependencySubstitution.all { dependency ->
        if (dependency.requested instanceof ModuleComponentSelector &&
dependency.requested.module == 'spring-boot-starter-logging') {
            dependency.useTarget("org.springframework.boot:spring-boot-
starter-log4j2:$dependency.requested.version", 'Use Log4j2 instead of Logback')
        }
    }
}

```



Log4j 入门人员将依赖集中在一起，以满足常见的日志记录要求（例如让 Tomcat 使用 `java.util.logging`，但使用 Log4j 2 配置输出）。



为了确保将使用 `java.util.logging` 执行的调试日志记录路由到 Log4j 2, 请通过将 `java.util.logging.manager` 系统属性设置为 `org.apache.logging.log4j.jul.LogManager` 来配置其 JDK logging adapter.

使用 YAML 或 JSON 配置 Log4j 2

除了默认的 XML 配置格式外, Log4j 2 还支持 YAML 和 JSON 配置文件. 要将 Log4j 2 配置为使用备用配置文件格式, 请将适当的依赖添加到类路径中, 并命名您的配置文件以匹配您选择的文件格式, 如以下示例所示:

格式	依赖	文件名
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>

10.9. 数据访问

Spring Boot 包含许多用于处理数据源的启动器. 本节回答与这样做有关的问题.

10.9.1. 配置自定义数据源

要配置自己的数据源, 请在配置中定义该类型的 `@Bean`. Spring Boot 重用您的 `DataSource` 到任何需要的地方, 包括数据库初始化. 如果需要外部化某些设置, 则可以将 `DataSource` 绑定到环境 (请参见 "`spring-boot-features.html`".).

以下示例显示了如何在 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"
    username: "sa"
    pool-size: 30

```

假设您的 `FancyDataSource` 具有 URL, 用户名和池大小的常规 JavaBean 属性, 则在将 `DataSource` 提供给其他组件之前, 将自动绑定这些设置. 常规的[数据库初始化](#) 也会发生 (因此 `spring.datasource.*` 的相关子集仍然可以与您的自定义配置一起使用) .

`Spring Boot` 还提供了一个名为 `DataSourceBuilder` 的实用工具生成器类, 可用于创建标准数据源之一 (如果它位于类路径上) . 构建者可以根据类路径中可用的内容来检测要使用的内容. 它还基于 JDBC URL 自动检测驱动程序.

下面的示例演示如何使用 `DataSourceBuilder` 创建数据源：

```

@Bean
@ConfigurationProperties("app.datasource")
public DataSource dataSource() {
    return DataSourceBuilder.create().build();
}

```

要使用该数据源运行应用程序, 您需要的只是连接信息. 还可以提供特定于池的设置. 有关更多详细信息, 请检查将在运行时使用的实现.

以下示例显示如何通过设置属性来定义 JDBC 数据源：

```
app:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
    pool-size: 30
```

但是,有一个陷阱. 由于未暴露连接池的实际类型,因此在自定义 `DataSource` 的元数据中不会生成任何键,并且 IDE 中也无法完成操作 (因为 `DataSource` 接口未暴露任何属性) . 另外,如果您碰巧在类路径上有 `Hikari`,则此基本设置将不起作用,因为 `Hikari` 没有 `url` 属性 (但确实具有 `jdbcUrl` 属性) . 在这种情况下,您必须按照以下方式重写配置:

```
app:
  datasource:
    jdbc-url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
    pool-size: 30
```

您可以通过强制连接池使用并返回专用的实现而不是 `DataSource` 来解决此问题. 您无法在运行时更改实现,但是选项列表将是明确的.

以下示例显示了如何使用 `DataSourceBuilder` 创建 `HikariDataSource`:

```
@Bean
@ConfigurationProperties("app.datasource")
public HikariDataSource dataSource() {
    return DataSourceBuilder.create().type(HikariDataSource.class).build();
}
```

您甚至可以利用 `DataSourceProperties` 为您做的事情进一步发展-即,通过提供默认的嵌入式数据库,并在不提供 URL 的情况下提供合理的用户名和密码. 您可以从任何 `DataSourceProperties` 对象的状态轻松地初始化 `DataSourceBuilder`,因此还可以注入Spring Boot 自动创建的 `DataSource`. 但是,这会将您的配置分为两个命名空间: `spring.datasource` 上的 `url`, `username`, `password`, `type` 和 `driver`,其余部分放在您的自定义命名空间 (`app.datasource`) 上. 为避免这种情况,可以在自定义命名空间上重新定义自定义 `DataSourceProperties`

,如以下示例所示:

```
@Bean
@Primary
@ConfigurationProperties("app.datasource")
public DataSourceProperties dataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource.configuration")
public HikariDataSource dataSource(DataSourceProperties properties) {
    return
    properties.initializeDataSourceBuilder().type(HikariDataSource.class).build();
}
```

默认情况下,该设置使您与 Spring Boot 为您执行的操作保持同步,不同的是,已选择(以代码形式)专用连接池,并且其设置在 `app.datasource.configuration` 子命名空间中暴露. 由于 `DataSourceProperties` 会为您处理 `url` / `jdbcUrl` 转换,因此可以按以下方式进行配置:

```
app:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
    configuration:
      maximum-pool-size: 30
```



Spring Boot 会将针对 Hikari 的设置暴露给 `spring.datasource.hikari`. 本示例使用更通用的配置子命名空间,因为该示例不支持多个数据源实现.



由于您的自定义配置选择使用 Hikari,因此 `app.datasource.type` 无效. 实际上,构建器会使用您可以在其中设置的任何值进行初始化,然后由对 `.type()` 的调用覆盖.

有关更多详细信息,请参见 "Spring Boot 特性" 部分中的 "["spring-boot-features.html"](#)" 和 `DataSourceAutoConfiguration` 类.

10.9.2. 配置两个数据源

如果需要配置多个数据源，则可以应用上一节中介绍的相同技巧。但是，您必须将其中一个 `DataSource` 实例标记为 `@Primary`，因为将来各种自动配置都希望能够按类型获取一个。

如果您创建自己的数据源，则会取消自动配置。在以下示例中，我们提供与自动配置在主数据源上提供的功能完全相同的功能集：

```
@Bean  
@Primary  
@ConfigurationProperties("app.datasource.first")  
public DataSourceProperties firstDataSourceProperties() {  
    return new DataSourceProperties();  
}  
  
@Bean  
@Primary  
@ConfigurationProperties("app.datasource.first.configuration")  
public HikariDataSource firstDataSource() {  
    return  
        firstDataSourceProperties().initializeDataSourceBuilder().type(HikariDataSource.  
            class).build();  
}  
  
@Bean  
@ConfigurationProperties("app.datasource.second")  
public BasicDataSource secondDataSource() {  
    return DataSourceBuilder.create().type(BasicDataSource.class).build();  
}
```



必须将 `firstDataSourceProperties` 标记为 `@Primary`，以便数据库初始化程序功能使用您的副本（如果使用初始化程序）。

这两个数据源也都必须进行高级定制。例如，您可以按以下方式配置它们：

```
app:  
  datasource:  
    first:  
      url: "jdbc:mysql://localhost/first"  
      username: "dbuser"  
      password: "dbpass"  
      configuration:  
        maximum-pool-size: 30  
  
    second:  
      url: "jdbc:mysql://localhost/second"  
      username: "dbuser"  
      password: "dbpass"  
      max-total: 30
```

您也可以将相同的概念应用于辅助数据源，如以下示例所示：

```

@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first.configuration")
public HikariDataSource firstDataSource() {
    return
firstDataSourceProperties().initializeDataSourceBuilder().type(HikariDataSource.
class).build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public DataSourceProperties secondDataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource.second.configuration")
public BasicDataSource secondDataSource() {
    return
secondDataSourceProperties().initializeDataSourceBuilder().type(BasicDataSource.
class).build();
}

```

上面的示例在自定义命名空间上配置两个数据源，其逻辑与 Spring Boot 在自动配置中使用的逻辑相同。请注意，每个配置子命名空间均基于所选实现提供高级设置。

10.9.3. 使用 Spring Data Repositories

Spring Data 可以创建各种风格的 `@Repository` 接口的实现。只要那些 `@Repositories` 包含在 `@EnableAutoConfiguration` 类的同一包（或子包）中，Spring Boot 就会为您处理所有这些操作。

对于许多应用程序，您所需要做的就是在类路径上放置正确的 Spring Data 依赖。有一个用于 JPA 的 `spring-boot-starter-data-jpa`，一个用于Mongodb的 `spring-boot-starter-data-mongodb`，等等。首先，创建一些存储库接口来处理 `@Entity` 对象。

Spring Boot 会根据发现的 `@EnableAutoConfiguration` 尝试猜测 `@Repository` 定义的位置。要获得更多控制权,请使用 `@EnableJpaRepositories` 注解 (来自 Spring Data JPA)。

有关 Spring Data 的更多信息,请参见 [Spring Data 项目页面](#)。

10.9.4. 将 `@Entity` 定义与 Spring 配置分开

Spring Boot 会根据发现的 `@EnableAutoConfiguration` 尝试猜测 `@Entity` 定义的位置。要获得更多控制,可以使用 `@EntityScan` 注解,如以下示例所示:

```
@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@EntityScan(basePackageClasses=City.class)
public class Application {

    //...

}
```

10.9.5. 配置 JPA 属性

Spring Data JPA 已经提供了一些独立于供应商的配置选项 (例如用于 SQL 日志记录的那些),并且 Spring Boot 暴露了这些选项,还为 Hibernate 提供了更多选项作为外部配置属性。其中的一些会根据上下文自动检测到,因此您不必进行设置。

`spring.jpa.hibernate.ddl-auto` 是一种特殊情况,因为根据运行时条件,它具有不同的默认值。如果使用嵌入式数据库,并且没有模式管理器 (例如 Liquibase 或 Flyway) 正在处理数据源,则默认为 `create-drop`。在所有其他情况下,它默认为 `none`。

JPA 提供程序检测到要使用的方言。如果您希望自己设置方言,请设置 `spring.jpa.database-platform` 属性。

下例显示了最常用的设置选项:

```
spring:
  jpa:
    hibernate:
      naming:
        physical-strategy: "com.example.MyPhysicalNamingStrategy"
    show-sql: true
```

另外,创建本地 `EntityManagerFactory` 时,`spring.jpa.properties.*` 中的所有属性均作为普通 JPA 属性 (前缀被去除) 传递.

您需要确保在 `spring.jpa.properties.*` 下定义的名称与 JPA 提供程序期望的名称完全匹配. Spring Boot 不会尝试对这些条目进行任何形式的宽松绑定.



例如,如果要配置 `Hibernate` 的批处理大小,则必须使用 `spring.jpa.properties.hibernate.jdbc.batch_size`. 如果您使用其他形式,例如 `batchSize` 或 `batch-size`,则 `Hibernate` 将不会应用该设置.



如果您需要对 `Hibernate` 属性应用高级自定义,请考虑注册在创建 `EntityManagerFactory` 之前将被调用的 `HibernatePropertiesCustomizer` bean. 这优先于自动配置应用的任何内容.

10.9.6. 配置 `Hibernate` 命名策略

`Hibernate` 使用 [两种不同的命名策略](#) 将名称从对象模型映射到相应的数据库名称. 可以分别通过设置 `spring.jpa.hibernate.naming.physical-strategy` 和 `spring.jpa.hibernate.naming.implicit-strategy` 属性来配置物理和隐式策略实现的标准类名. 另外,如果在应用程序上下文中可以使用 `ImplicitNamingStrategy` 或 `PhysicalNamingStrategy` Bean,则 `Hibernate` 将自动配置为使用它们.

默认情况下, Spring Boot 使用 `SpringPhysicalNamingStrategy` 配置物理命名策略. 此实现提供了与 `Hibernate 4` 相同的表结构: 所有点都由下划线替换,而骆驼套也由下划线替换. 默认情况下,所有表名均以小写形式生成. 例如,一个

`TelephoneNumber` 实体被映射到 `telephone_number` 表. 如果您需要大小写混合的标识符, 请定义一个自定义 `SpringPhysicalNamingStrategy` bean, 如以下示例所示:

```
@Bean
SpringPhysicalNamingStrategy caseSensitivePhysicalNamingStrategy() {
    return new SpringPhysicalNamingStrategy() {

        @Override
        protected boolean isCaseInsensitive(JdbcEnvironment jdbcEnvironment) {
            return false;
        }

    };
}
```

如果您希望改用 `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
```

另外, 您可以配置以下 bean:

```
@Bean
public PhysicalNamingStrategy physicalNamingStrategy() {
    return new PhysicalNamingStrategyStandardImpl();
}
```

有关更多详细信息, 请参见 [HibernateJpaAutoConfiguration](#) 和 [JpaBaseConfiguration](#).

10.9.7. 配置 `Hibernate` 二级缓存

可以为一系列缓存提供程序配置 `Hibernate 二级缓存`. 与其将 `Hibernate` 配置为再次查找缓存提供程序, 不如提供尽可能在上下文中可用的缓存提供程序.

如果您使用的是 `JCache`, 这非常简单. 首先, 确保 `org.hibernate:hibernate-jcache` 在类路径上可用. 然后, 添加一个 `HibernatePropertiesCustomizer` bean, 如以下示例所示:

```

@Configuration(proxyBeanMethods = false)
public class HibernateSecondLevelCacheExample {

    @Bean
    public HibernatePropertiesCustomizer
    hibernateSecondLevelCacheCustomizer(JCacheCacheManager cacheManager) {
        return (properties) -> properties.put(ConfigSettings.CACHE_MANAGER,
cacheManager.getCacheManager());
    }

}

```

这个定制器将配置 `Hibernate` 使用与应用程序相同的 `CacheManager`. 也可以使用单独的 `CacheManager` 实例. 有关详细信息, 请参阅 [Hibernate用户指南](#).

10.9.8. 在 `Hibernate` 组件中使用依赖注入

默认情况下, `Spring Boot` 注册一个使用 `BeanFactory` 的 `BeanContainer` 实现, 以便转换器和实体监听器可以使用常规的依赖注入.

您可以通过注册删除或更改 `hibernate.resource.beans.container` 属性的 `HibernatePropertiesCustomizer` 来禁用或调整此行为.

10.9.9. 使用自定义 `EntityManagerFactory`

要完全控制 `EntityManagerFactory` 的配置, 您需要添加一个名为 '`entityManagerFactory`' 的 `@Bean`. 如果存在这种类型的 Bean, `Spring Boot` 自动配置将关闭其实体管理器.

10.9.10. 使用两个 `EntityManager`

即使默认的 `EntityManagerFactory` 正常工作, 您也需要定义一个新的 Bean. 否则, 该类型的第二个 bean 的存在将关闭默认值. 为了简化操作, 您可以使用 `Spring Boot` 提供的便捷 `EntityManagerBuilder`. 或者, 您可以直接从 `Spring ORM` 中直接访问 `LocalContainerEntityManagerManagerBean`, 如以下示例所示:

```
// add two data sources configured as above

@Bean
public LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(
    EntityManagerFactoryBuilder builder) {
    return builder
        .dataSource(customerDataSource())
        .packages(Customer.class)
        .persistenceUnit("customers")
        .build();
}

@Bean
public LocalContainerEntityManagerFactoryBean orderEntityManagerFactory(
    EntityManagerFactoryBuilder builder) {
    return builder
        .dataSource(orderDataSource())
        .packages(Order.class)
        .persistenceUnit("orders")
        .build();
}
```



当您自己为 `LocalContainerEntityManagerFactoryBean` 创建 bean 时，在自动配置的 `LocalContainerEntityManagerFactoryBean` 创建期间应用的所有自定义设置都将丢失。例如，对于 `Hibernate`, `spring.jpa.hibernate` 前缀下的任何属性都不会自动应用于您的 `LocalContainerEntityManagerManagerBean`。如果您依靠这些属性来配置诸如命名策略或 DDL 模式之类的东西，那么在创建 `LocalContainerEntityManagerManagerBean` bean 时将需要显式配置。另一方面，如果您使用自动配置的 `EntityManagerFactoryBuilder` 来构建 `LocalContainerEntityManagerFactoryBean` bean，那么将自动应用通过 `spring.jpa.properties` 指定的应用于自动配置的 `EntityManagerFactoryBuilder` 的属性。

上面的配置几乎可以独立工作。为了完成此图，您还需要为两个 `EntityManager` 配置 `TransactionManager`。如果将其中一个标记为 `@Primary`，则可以由 Spring Boot 中的默认 `JpaTransactionManager` 拾取。另一个必须显式地注入到新实例中。另外，您也许可以使用跨两个 JTA 事务管理器。

如果使用 Spring Data，则需要相应地配置 `@EnableJpaRepositories`，如以下示例所示：

```
@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = Customer.class,
    entityManagerFactoryRef = "customerEntityManagerFactory")
public class CustomerConfiguration {
    ...
}

@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = Order.class,
    entityManagerFactoryRef = "orderEntityManagerFactory")
public class OrderConfiguration {
    ...
}
```

10.9.11. 使用传统的 `persistence.xml` 文件

默认情况下，Spring Boot 不会搜索或使用 `META-INF/persistence.xml`。

如果您更喜欢使用传统的 `persistence.xml`，则需要定义自己的 `LocalBeanManagerFactoryBean` 类型的 `@Bean`（`ID` 为 ‘`entityManagerFactory`’），并在其中设置持久性单元名称。

有关默认设置，请参见 [JpaBaseConfiguration](#)。

10.9.12. 使用 Spring Data JPA 和 Mongo 存储库

Spring Data JPA 和 Spring Data Mongo 都可以为您自动创建 `Repository` 实现。如果它们都存在于类路径中，则可能必须做一些额外的配置以告诉 Spring Boot 要创建哪个存储库。最明确的方法是使用标准 Spring Data `@EnableJpaRepositories` 和 `@EnableMongoRepositories` 注解并提供 `Repository` 接口的位置。

还有一些标记（`spring.data.*.repositories.enabled` 和 `spring.data.*.repositories.type`）可用于在外部配置中打开和关闭自动配置的存储库。这样做很有用，例如，如果您想关闭 Mongo 存储库并仍然使用自动配置的 `MongoTemplate`。

对于其他自动配置的 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，请相应地更改注解和标志的名称。

10.9.13. 定制 Spring Data 的 Web 支持

Spring Data 提供了 Web 支持,简化了 Web 应用程序中 Spring Data 存储库的使用. Spring Boot 在 `spring.data.web` 命名空间中提供属性以自定义其配置. 请注意,如果您使用的是 Spring Data REST,则必须改为使用 `spring.data.rest` 命名空间中的属性.

10.9.14. 将 Spring Data Repositories 暴露为 REST 端点

如果已为应用程序启用了 Spring MVC,则 Spring Data REST 可以为您将 `Repository` 实现作为 REST 端点暴露.

Spring Boot 暴露了一组有用的属性 (来自 `spring.data.rest` 命名空间),这些属性来自定义 `RepositoryRestConfiguration`. 如果需要提供其他定制,则应使用 `RepositoryRestConfigurer` bean.



如果您未在自定义 `RepositoryRestConfigurer` 上指定任何顺序,则该顺序在一个 Spring Boot 内部使用后运行. 如果您需要指定订单,请确保该订单大于 0.

10.9.15. 配置 JPA 使用的组件

如果要配置 JPA 使用的组件,则需要确保在 JPA 之前初始化该组件. 当组件被自动配置后, Spring Boot 会为您处理. 例如,当自动配置 Flyway 时,会将 Hibernate 配置为依赖 Flyway,以便 Flyway 有机会在 Hibernate 尝试使用数据库之前对其进行初始化.

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的便捷方法. 例如,如果您将 Hibernate Search 和 Elasticsearch 用作其索引管理器,则必须将任何 `EntityManagerFactory` Bean 配置为依赖于 `elasticsearchClient` Bean,如以下示例所示:

```

/**
 * {@link EntityManagerFactoryDependsOnPostProcessor} that ensures that
 * {@link EntityManagerFactory} beans depend on the {@code elasticsearchClient}
 * bean.
 */
@Component
static class ElasticsearchEntityManagerFactoryDependsOnPostProcessor
    extends EntityManagerFactoryDependsOnPostProcessor {

    ElasticsearchEntityManagerFactoryDependsOnPostProcessor() {
        super("elasticsearchClient");
    }

}

```

10.9.16. 使用两个数据源配置 jOOQ

如果需要将 `jOOQ` 与多个数据源一起使用，则应该为每个数据源创建自己的 `DSLContext`。有关更多详细信息，请参阅 [JooqAutoConfiguration](#)。



特别是，可以重用 `JooqExceptionTranslator` 和 `SpringTransactionProvider` 以提供与自动配置对单个 `DataSource` 所做的功能相似的功能。

10.10. 初始化 Database

可以使用不同的方式初始化 SQL 数据库，具体取决于堆栈是什么。当然，如果数据库是一个单独的过程，您也可以手动执行。建议使用单一机制进行模式生成。

10.10.1. 使用 JPA 初始化数据库

JPA 具有用于 DDL 生成的功能，可以将其设置为在启动时针对数据库运行。这是通过两个外部属性控制的：

`spring.jpa.hibernate.ddl-auto` (枚举)

- `spring.jpa.generate-ddl` (boolean) 开启和关闭该功能，并且与供应商无关。
- `spring.jpa.hibernate.ddl-auto` (enum) 是一种 Hibernate 功能

, 可以更精细地控制行为. 此功能将在本指南的后面部分详细介绍.

10.10.2. 使用 `Hibernate` 初始化数据库

您可以显式设置 `spring.jpa.hibernate.ddl-auto`, 并且标准的 `Hibernate` 属性值为 `none`, 验证, 更新, 创建和创建放置. Spring Boot

根据是否认为您的数据库已嵌入为您选择默认值. 如果未检测到模式管理器, 或者在所有其他情况下均未检测到模式管理器, 则默认为 `create-drop`. 通过查看 `Connection` 类型和 `JDBC url` 可以检测到嵌入式数据库. `hsqldb`, `h2` 和 `derby` 是候选的, 其他则不是. 从内存数据库转换为 "真实" 数据库时, 请务必不要假设新平台中表和数据的存在. 您必须显式设置 `ddl-auto` 或使用其他机制之一来初始化数据库.



您可以通过启用 `org.hibernate.SQL` 记录器来输出模式创建. 如果启用 [调试模式](#), 此操作将自动为您完成.

另外, 如果 `Hibernate` 从头开始创建架构 (即, 如果 `ddl-auto` 属性设置为 `create` 或 `create-drop`) , 则在启动时会在类路径的根目录中执行一个名为 `import.sql` 的文件. 如果您小心的话, 这对于演示和测试很有用, 但可能不希望出现在生产环境的类路径中. 这是一个 `Hibernate` 功能 (与 `Spring` 无关) .

10.10.3. 使用 SQL 脚本初始化数据库

Spring Boot 可以自动创建数据源的架构 (DDL脚本) 并对其进行初始化 (DML脚本) .

它从标准根类路径位置 (分别为 `schema.sql` 和 `data.sql`) 加载SQL. 另外, Spring Boot 处理 `schema-${platform}.sql` 和 `data-${platform}.sql` 文件 (如果存在), 其中 `platform` 是 `spring.datasource.platform` 的值.

这使您可以在必要时切换到特定于数据库的脚本. 例如

, 您可以选择将其设置为数据库的供应商名称 (`hsqldb`, `h2`, `oracle`, `mysql`, `postgresql` 等)

.

Spring Boot 自动创建嵌入式数据源的架构. 可以使用 `spring.datasource.initialization-mode` 属性来自定义此行为. 例如, 如果要始终初始化数据源, 而不管其类型如何:

```
spring.datasource.initialization-mode=always
```



在基于 JPA 的应用程序中, 您可以选择让 Hibernate 创建 schema 或使用 `schema.sql`, 但您不能两者都做. 如果使用 `schema.sql`, 请确保禁用 `spring.jpa.hibernate.ddl-auto`.

```
spring.jpa.hibernate.ddl-auto=none
```

如果你正在使用 Flyway 或 Liquibase 高级数据库迁移工具, 应单独使用它们来创建和初始化 schema. 不建议将 `schema.sql` 和 `data.sql` 脚本与 Flyway 或 Liquibase 一起使用, 并且在未来的版本中删除此支持.

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, 则应用程序将无法启动. 您可以通过设置 `spring.datasource.continue-on-error` 来调整行为.

10.10.4. 使用 R2DBC 初始化数据库

如果您使用的是 R2DBC, 则常规的 `DataSource` 自动配置会退出, 因此不能使用上述任何选项.

您可以在启动时使用 SQL 脚本初始化数据库, 如以下示例所示:

```

@Configuration(proxyBeanMethods = false)
static class DatabaseInitializationConfiguration {

    @Autowired
    void initializeDatabase(ConnectionFactory connectionFactory) {
        ResourceLoader resourceLoader = new DefaultResourceLoader();
        Resource[] scripts = new Resource[] {
            resourceLoader.getResource("classpath:schema.sql"),
            resourceLoader.getResource("classpath:data.sql") };
        new
        ResourceDatabasePopulator(scripts).populate(connectionFactory).block();
    }

}

```

或者,您可以配置 [Flyway](#) 或 [Liquibase](#) 在迁移期间为您配置数据源.
.这两个库均提供用于设置要迁移的数据库的 `url`, `username` 和 `password` 的属性.



选择此选项时, `org.springframework:spring-jdbc`
仍然是必需的依赖.

10.10.5. 初始化一个 Spring Batch 数据库

如果您使用 `Spring Batch`,则它随大多数流行的数据库平台一起预包装了 SQL 初始脚本.
`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
,则默认情况下会发生这种情况. 您还可以为任何数据库类型启用它,如以下示例所示:

```

spring:
  batch:
    initialize-schema: "always"

```

您还可以通过设置 `spring.batch.initialize-schema` 为 `never` 显式关闭初始化.

10.10.6. 使用高级数据库迁移工具

`Spring Boot` 支持两个更高级别的迁移工具: [Flyway](#) 和 [Liquibase](#).

在启动时执行 Flyway 数据库迁移

要在启动时自动运行 Flyway 数据库迁移, 请将 `org.flywaydb:flyway-core` 添加到您的类路径中.

通常, 迁移是格式为 `V<VERSION>__<NAME>.sql` (带有 `<VERSION>` 下划线分隔的版本, 例如 '`1`' 或 '`2_1`') 的脚本. 默认情况下, 它们位于名为 `classpath:db/migration` 的目录中, 但是您可以通过设置 `spring.flyway.locations` 来修改该位置. 这是一个或多个 `classpath:` 或 `filesystem:` 位置的逗号分隔列表. 例如, 以下配置将在默认类路径位置和 `/opt/migration` 目录中搜索脚本:

```
spring:  
  flyway:  
    locations: "classpath:db/migration,filesystem:/opt/migration"
```

您还可以添加特殊的 `{vendor}` 占位符以使用特定于供应商的脚本. 假设以下内容:

```
spring:  
  flyway:  
    locations: "classpath:db/migration/{vendor}"
```

前面的配置没有使用 `db/migration`, 而是根据数据库的类型 (例如 MySQL 的 `db/migration/mysql`) 来设置要使用的目录. 受支持的数据库列表在 `DatabaseDriver` 中可用.

迁移也可以用 Java 编写. 将使用实现 `JavaMigration` 的任何 bean 自动配置 Flyway.

`FlywayProperties` 提供了 Flyway 的大多数设置以及少量的其他属性, 这些属性可用于禁用迁移或关闭位置检查. 如果您需要对配置进行更多控制, 请考虑注册 `FlywayConfigurationCustomizer` bean.

Spring Boot 调用 `Flyway.migrate()` 来执行数据库迁移. 如果您想要更多控制, 请提供一个实现 `FlywayMigrationStrategy` 的 `@Bean`.

Flyway 支持 SQL 和 Java `callbacks`. 要使用基于 SQL 的回调, 请将回调脚本放置在 `classpath:db/migration` 目录中. 要使用基于 Java 的回调, 请创建一个或多个实现 `Callback` 的 bean. 任何此类 bean 都会自动在 Flyway 中注册. 可以使用 `@Order`

或通过实现 `Ordered` 来排序它们。也可以检测到实现了不推荐使用的 `FlywayCallback` 接口的 Bean，但是不能与 Callback Bean 一起使用。

默认情况下，Flyway 在您的上下文中自动装配（`@Primary`）数据源，并将其用于迁移。如果您想使用其他数据源，则可以创建一个并将其 `@Bean` 标记为 `@FlywayDataSource`。如果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`。另外，您可以通过在外部属性中设置 `spring.flyway.[url,user,password]` 来使用 Flyway 的本地数据源。设置 `spring.flyway.url` 或 `spring.flyway.user` 足以使 Flyway 使用其自己的 `DataSource`。如果未设置这三个属性中的任何一个，则将使用其等效的 `spring.datasource` 属性的值。

您还可以使用 Flyway 为特定情况提供数据。例如，您可以将特定于测试的迁移放置在 `src/test/resources` 中，并且仅在您的应用程序开始进行测试时才运行它们。另外，您可以使用特定于配置文件的配置来自定义 `spring.flyway.locations`，以便仅在特定配置文件处于 `active` 状态时才运行某些迁移。例如，在 `application-dev.properties` 中，您可以指定以下设置：

```
spring:  
  flyway:  
    locations: "classpath:/db/migration,classpath:/dev/db/migration"
```

通过该设置，仅在 `dev` 环境下才运行 `dev/db/migration` 中的迁移。

在启动时执行 Liquibase 数据库迁移

要在启动时自动运行 Liquibase 数据库迁移，请将 `org.liquibase:liquibase-core` 添加到您的类路径中。

默认情况下，从 `db/changelog/db.changelog-master.yaml` 中读取主更改日志，但是您可以通过设置 `spring.liquibase.change-log` 来更改位置。除了 YAML，Liquibase 还支持 JSON、XML 和 SQL 更改日志格式。

默认情况下，Liquibase 在您的上下文中自动装配（`@Primary`）数据源，并将其用于迁移。如果需要使用其他数据源，则可以创建一个并将其 `@Bean` 标记为 `@LiquibaseDataSource`。如果这样做，并且想要两个数据源，请记住创建另一个数据源并将其标记为 `@Primary`。另外，您可以通过在外部属性中设置 `spring.liquibase.[driver-class-name,url,user,password]` 来使用 Liquibase 的本地数据源。设置

`spring.liquibase.url` 或 `spring.liquibase.user` 足以使 Liquibase 使用其自己的数据源。如果未设置这三个属性中的任何一个，则将使用其等效的 `spring.datasource` 属性的值。

有关可用设置（例如上下文，默认架构等）的详细信息，请参见 [LiquibaseProperties](#)。

10.11. 消息

Spring Boot 提供了许多包含消息传递的启动器。本部分回答了将消息与 Spring Boot 一起使用所引起的问题。

10.11.1. 禁用事务 JMS 会话

如果您的 JMS 代理不支持事务处理会话，则必须完全禁用对事务的支持。如果创建自己的 `JmsListenerContainerFactory`，则无需执行任何操作，因为默认情况下无法进行处理。

如果您想使用 `DefaultJmsListenerContainerFactoryConfigurer` 重用 Spring Boot 的默认设置，则可以按以下方式禁用事务会话：

```
@Bean
public DefaultJmsListenerContainerFactory jmsListenerContainerFactory(
    ConnectionFactory connectionFactory,
    DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory listenerFactory =
        new DefaultJmsListenerContainerFactory();
    configurer.configure(listenerFactory, connectionFactory);
    listenerFactory.setTransactionManager(null);
    listenerFactory.setSessionTransacted(false);
    return listenerFactory;
}
```

前面的示例将覆盖默认工厂，并且应将其应用于应用程序定义的任何其他工厂（如果有）。

10.12. Batch Applications

当人们从 Spring Boot 应用程序中使用 Spring Batch 时，经常会出现许多问题。本节解决这些问题。

10.12.1. 指定批处理数据源

默认情况下,批处理应用程序需要一个数据源来存储作业详细信息.

批处理在您的上下文中自动装配单个数据源,并将其用于处理.

要使批处理使用应用程序的主数据源以外的其他数据源,请声明一个数据源 `bean`,并用

`@BatchDataSource` 注解其 `@Bean` 方法. 如果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`. 为了更好地控制,请实现
`BatchConfigurer`. 有关更多详细信息,请参见 [The Javadoc of
@EnableBatchProcessing](#) 的Javadoc.

有关 `Spring Batch` 的更多信息,请参见 [Spring Batch 项目页面](#).

10.12.2. 在启动时执行 `Spring Batch` 作业

通过在 `@Configuration` 类的某个位置添加 `@EnableBatchProcessing`,可以启用 `Spring Batch` 自动配置.

默认情况下,它将在启动时在应用程序上下文中执行所有作业 (有关详细信息,请参见 `JobLauncherApplicationRunner`) . 您可以通过指定 `spring.batch.job.names` (采用作业名称模式的逗号分隔列表) 来缩小到一个或多个特定作业.

```
{spring-boot-autoconfigure-module-
code}/batch/BatchAutoConfiguration.java[BatchAutoConfiguration] 和 {spring-
batch-
api}/core/configuration/annotation/EnableBatchProcessing.html[@EnableBatchProces-
sing] 获取更多信息.
```

10.12.3. 在命令行上运行

`Spring Boot` 将任何以 `--` 开头的命令行参数转换为一个属性以添加到 `Environment` 中
,请参见 [访问命令行属性](#). 不应该将其作为参数传递给批处理作业.

要在命令行上指定批处理参数,请使用常规格式 (即不使用 `--`),如以下示例所示:

```
$ java -jar myapp.jar someParameter=someValue anotherParameter=anotherValue
```

如果在命令行上指定 `Environment` 的属性,它将被作业忽略. 考虑以下命令:

```
$ java -jar myapp.jar --server.port=7070 someParameter=someValue
```

这仅为批处理作业提供了一个参数：`someParameter=someValue`

10.12.4. 存储作业库

Spring Batch 需要一个用于 Job 仓库的数据存储。如果使用 Spring Boot，则必须使用实际的数据库。请注意，它可以是内存数据库，请参见 docs.spring.io/spring-batch/docs/4.3.x/reference/html/job.html#configuringJobRepository[配置作业存储库]。

10.13. Actuator

Spring Boot 包括 Spring Boot Actuator。本节回答了经常因使用而引起的问题。

10.13.1. 更改 Actuator 端点的HTTP端口或地址

在独立应用程序中，Actuator HTTP 端口默认与主 HTTP 端口相同。

要使应用程序在其他端口上进行监听，请设置外部属性：`management.server.port`。

要监听完全不同的网络地址（例如，

当您拥有用于管理的内部网络和用于用户应用程序的外部网络）时，还可以将 `management.server.address` 设置为服务器能够绑定到的有效IP地址。

有关更多详细信息，请参阅 生产就绪功能 部分中的 `ManagementServerProperties` 源代码和 “[production-ready-features.html](#)”。

10.13.2. 自定义 ‘whitelabel’ 错误页面

如果遇到服务器错误，Spring Boot 会安装一个‘`whitelabel`’ 错误页面，您会在浏览器客户端中看到该错误页面（使用 JSON 和其他媒体类型的机器客户端应该看到带有正确错误代码的明智响应）。



设置 `server.error.whitelabel.enabled=false`

可以关闭默认错误页面。这样做将还原您正在使用的 `servlet` 容器的默认值。请注意，`Spring Boot` 仍然尝试解决错误视图，因此您应该添加自己的错误页面，而不是完全禁用它。

用自己的方法覆盖错误页面取决于您使用的模板技术。例如，如果您使用 `Thymeleaf`，则可以添加 `error.html` 模板。如果使用 `FreeMarker`，则可以添加 `error.ftlh` 模板。通常，您需要使用错误名称解析的 `View` 或处理 `/error` 路径的 `@Controller`。除非您替换了某些默认配置，否则您应该在 `ApplicationContext` 中找到一个 `BeanNameViewResolver`，因此，以 `@Bean` 命名的错误将是一种简单的方法。有关更多选项，请参见 [ErrorMvcAutoConfiguration](#)。

有关如何在 `Servlet` 容器中注册处理程序的详细信息，另请参见“[错误处理](#)”部分。

10.13.3. Sanitize Sensitive Values

`env` 和 `configprops` 端点返回的信息可能有些敏感，因此默认情况下会清理匹配特定模式的键（即，它们的值将替换为）。

可以分别使用 `management.endpoint.env.keys-to-sanitize` 和 `management.endpoint.configprops.keys-to-sanitize` 来定制要使用的模式。

`Spring Boot` 对此类密钥使用明智的默认设置：例如，对任何以单词 “`password`”，“`secret`”，“`key`”，“`token`”，“`vcap_services`”，“`sun.java.command`” 结尾的密钥进行清理。也可以改用正则表达式，例如 `*credentials.*` 来清理任何将单词凭据作为密钥一部分保存的密钥。

此外，`Spring Boot` 只对以 “`uri`”，“`uris`”，“`address`”，或 “`addresses`” 结尾的密钥的 URI 敏感部分进行清理。URI 的敏感部分使用格式 `<scheme>://<username>:<password>@<host>:<port>/` 来标识。例如，对于属性 `myclient.uri=http://user1:password1@localhost:8081`，清理后的结果值为 `http://user1:*****@localhost:8081`。

10.13.4. 映射健康状态

`Spring Boot` 通过 `Status` 类型响应式系统运行状况。

如果要监视或警告特定应用程序的运行状况,可以通过 `Micrometer` 暴露这些状态。默认情况下, `Spring Boot` 使用状态代码 “`UP`”, “`DOWN`”, “`OUT_OF_SERVICE`” 和 “`UNKNOWN`”。要暴露这些状态,您需要将这些状态转换为一组数字,以便它们可以与 `Micrometer` 一起使用。

以下示例显示了一种方法:

```
@Configuration
public class HealthMetricsConfiguration {

    public HealthMetricsConfiguration(MeterRegistry registry, HealthEndpoint
healthEndpoint) {
        // This example presumes common tags (such as the app) are applied
elsewhere
        Gauge.builder("health", healthEndpoint,
this::getStatusCode).strongReference(true).register(registry);
    }

    private int getStatusCode(HealthEndpoint health) {
        Status status = health.health().getStatus();
        if (Status.UP.equals(status)) {
            return 3;
        }
        if (Status.OUT_OF_SERVICE.equals(status)) {
            return 2;
        }
        if (Status.DOWN.equals(status)) {
            return 1;
        }
        return 0;
    }

}
```

10.14. 安全

本部分解决有关使用 `Spring Boot` 时的安全性的问题,包括因将 `Spring Security` 与 `Spring Boot` 一起使用而引起的问题。

有关 `Spring Security` 的更多信息,请参见 [Spring Security 项目页面](#)。

10.14.1. 关闭 Spring Boot 安全性配置

如果您在应用程序中使用 `WebSecurityConfigurerAdapter` 或 `SecurityFilterChain` 定义一个 `@Configuration` Bean, 它将关闭 Spring Boot 中的默认 Webapp 安全设置.

10.14.2. 更改`UserDetailsService`并添加用户帐户

如果提供类型为 `AuthenticationManager`, `AuthenticationProvider` 或 `UserDetailsService` 的 `@Bean`, 则不会为 `InMemoryUserDetailsManager` 创建默认的 `@Bean`. 这意味着您拥有完整的 Spring Security 功能集 (例如 各种身份验证选项) .

添加用户帐户的最简单方法是提供自己的 `UserDetailsService` bean.

10.14.3. 在代理服务器后运行时启用HTTPS

对于所有应用程序而言, 确保所有主要端点仅可通过 HTTPS 进行访问都是一项重要的工作. 如果您将 Tomcat 用作 Servlet 容器, 则 Spring Boot 如果检测到某些环境设置, 则会自动添加 Tomcat 自己的 `RemoteIpValve`, 并且您应该能够依靠 `HttpServletRequest` 来报告它是否安全 (甚至在代理服务器的下游) 处理真实的 SSL 终止). 标准行为由某些请求头 (`x-forwarded-for` 和 `x-forwarded-proto`) 的存在或不存在决定, 它们的名称是常规名称, 因此它应可与大多数前端代理一起使用. 您可以通过将一些条目添加到 `application.properties` 来打开阀门, 如以下示例所示:

```
server:  
  tomcat:  
    remoteip:  
      remote-ip-header: "x-forwarded-for"  
      protocol-header: "x-forwarded-proto"
```

(这些属性中的任何一个都会在阀上切换. 或者, 您可以通过添加 `TomcatServletWebServerFactory` bean 来添加 `RemoteIpValve`.)

要将 Spring Security 配置为对所有 (或某些) 请求都需要安全通道, 请考虑添加自己的 `SecurityFilterChain`, 其中添加了以下 `HttpSecurity` 配置:

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
    // Customize the application security
    http.requiresChannel().anyRequest().requiresSecure();
    return http.build();
}

```

10.15. 热交换

Spring Boot 支持热插拔。本部分回答有关其工作方式的问题。

10.15.1. 重新加载静态内容

有几种热加载选项。推荐的方法是使用 [spring-boot-devtools](#)，因为它提供了其他开发时功能，例如对应用程序快速重启和LiveReload 的支持以及合理的开发时配置（例如模板缓存）。Devtools通过监视类路径的更改来工作。这意味着必须“构建”静态资源更改才能使更改生效。默认情况下，当您保存更改时，这在 Eclipse 中自动发生。在 IntelliJ IDEA中，“生成项目”命令将触发必要的构建。由于[默认的重新启动排除项](#)，对静态资源的更改不会触发应用程序的重新启动。但是，它们确实会触发实时重新加载。

另外，在 IDE 中运行（特别是在调试时打开）是进行开发的好方法（所有现代 IDE 都允许重新加载静态资源，并且通常还允许热交换 Java 类更改）。

最后，可以配置 [Maven](#) 和 [Gradle](#) 插件（请参见 [addResources](#) 属性）以支持从命令行运行，并直接从源代码重新加载静态文件。如果要使用高级工具编写该代码，则可以将其与外部 `css/js` 编译器进程一起使用。

10.15.2. 重新加载模板，而无需重新启动容器

Spring Boot 支持的大多数模板技术都包含用于禁用缓存的配置选项（在本文档的后面介绍）。如果您使用 [spring-boot-devtools](#) 模块，那么在开发时会自动配置这些属性。

Thymeleaf 模板

如果您使用 Thymeleaf，请将 `spring.thymeleaf.cache` 设置为 `false`。有关其他 Thymeleaf 定制选项，请参见 [ThymeleafAutoConfiguration](#)。

FreeMarker 模板

如果使用 FreeMarker, 请将 `spring.freemarker.cache` 设置为 `false`. 有关其他 FreeMarker 定制选项, 请参见 [FreeMarkerAutoConfiguration](#).

Groovy 模板

如果使用 Groovy 模板, 请将 `spring.groovy.template.cache` 设置为 `false`. 有关其他 Groovy 定制选项, 请参见 [GroovyTemplateAutoConfiguration](#).

10.15.3. 快速重启应用程序

`spring-boot-devtools` 模块包括对应用程序自动重启的支持. 尽管不如 [JRebel](#) 这样的技术快, 但通常比 “cold start” 要快得多.

在研究本文档后面讨论的一些更复杂的重载选项之前, 您可能应该先尝试一下.

有关更多详细信息, 请参见 [using-spring-boot.html](#) 部分.

10.15.4. 重新加载Java类而无需重新启动容器

许多现代的 IDE (Eclipse, IDEA等) 都支持字节码的热交换. 因此, 如果所做的更改不影响类或方法的签名, 则应干净地重新加载而没有副作用.

10.16. 构建

Spring Boot 包括 Maven 和 Gradle 的构建插件. 本部分回答有关这些插件的常见问题.

10.16.1. 生成构建信息

Maven 插件和 Gradle 插件都允许生成包含项目的坐标, 名称和版本的构建信息. 还可以将插件配置为通过配置添加其他属性. 当存在这样的文件时, Spring Boot 会自动配置 `BuildProperties` bean.

要使用 Maven 生成构建信息, 请为 `build-info` 目标添加执行, 如以下示例所示:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.4.5</version>
      <executions>
        <execution>
          <goals>
            <goal>build-info</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```



有关更多详细信息,请参见 [Spring Boot Maven 插件文档](#).

下面的示例对 `Gradle` 执行相同的操作:

```
springBoot {
  buildInfo()
}
```



有关更多详细信息,请参见 [Spring Boot Gradle 插件文档](#) .

10.16.2. 生成 Git 信息

Maven 和 Gradle 都允许生成一个 `git.properties` 文件,其中包含有关构建项目时 git 源代码仓库状态的信息.

对于 Maven 用户, `spring-boot-starter-parent` POM 包含一个预先配置的插件,用于生成 `git.properties` 文件. 要使用它,请将以下声明添加到您的 POM 中:

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

Gradle 用户可以使用 [gradle-git-properties](#) 插件获得相同的结果, 如以下示例所示:

```
plugins {
  id "com.gorylenko.gradle-git-properties" version "2.2.4"
}
```

Maven 和 Gradle 插件都允许配置 [git.properties](#) 中包含的属性.



[git.properties](#) 中的提交时间应与以下格式匹配: `yyyy-MM-`

`dd'T'HH: mm: ssZ`. 这是上面列出的两个插件的默认格式.

使用这种格式, 可以将时间解析为 [Date](#), 并将其序列化为 JSON 后
, 由杰克逊的日期序列化配置设置控制.

10.16.3. 自定义依赖版本

[spring-boot-dependencies](#) POM 管理常见依赖的版本. Maven 和 Gradle 的 Spring Boot 插件允许使用构建属性来自定义这些托管依赖版本.



每个 Spring Boot

发行版均针对这组特定的第三方依赖进行了设计和测试.

覆盖版本可能会导致兼容性问题.

要在 Maven 中覆盖依赖版本, 请参阅 Maven 插件文档的 [这一部分](#).

要在 Gradle 中覆盖依赖版本, 请参阅 Gradle 插件文档的 [这一部分](#).

10.16.4. 使用 Maven 创建可执行 JAR

`spring-boot-maven-plugin` 可用于创建可执行的 “fat” JAR. 如果使用 `spring-boot-starter-parent` 父POM, 则可以声明插件, 然后将 `jar` 重新打包, 如下所示:

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

如果您不使用父 POM, 则仍然可以使用该插件. 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>2.4.5</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

有关完整用法的详细信息, 请参阅 [插件文档](#) .

10.16.5. 使用 Spring Boot 应用程序作为依赖

像 `war` 文件一样, Spring Boot 应用程序也不打算用作依赖.

如果您的应用程序包含要与其他项目共享的类, 则建议的方法是将该代码移到单独的模块中.

然后, 您的应用程序和其他项目可以依赖单独的模块.

如果您无法按照上面的建议重新排列代码，则必须配置 Spring Boot 的 Maven 和 Gradle 插件以生成单独的 `artifacts`，该 `artifacts` 适合用作依赖。可执行存档不能用作依赖，因为可执行 jar 格式 `BOOT-INF/classes` 中的应用程序类被打包。这意味着当将可执行 jar 用作依赖时，找不到它们。

为了产生两个 `artifacts`，一个可以用作依赖，另一个可以执行，必须指定分类器。该分类器应用于可执行归档文件的名称，保留默认归档文件用作依赖。

要在 Maven 中配置 `exec` 的 `classifier`，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>
```

10.16.6. 运行可执行 jar 时提取特定的库

可执行 jar 中的大多数嵌套库不需要解压即可运行。但是，某些库可能会有问题。例如，JRuby 包含其自己的嵌套 jar 支持，它假定 `jruby-complete.jar` 始终可以直接作为文件直接使用。

为了处理任何有问题的库，您可以标记在可执行 jar 首次运行时应自动解压缩特定的嵌套 jar。这种嵌套的 jar 会写在 `java.io.tmpdir` 系统属性标识的临时目录下。



应注意确保已配置您的操作系统，以便在应用程序仍在运行时，它不会删除已解压缩到临时目录中的 jar。

例如，为了指示应该使用 Maven 插件将 JRuby 标记为要解包，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>
```

10.16.7. 创建带有排除项的不可执行的 JAR

通常,如果您具有一个可执行文件和一个不可执行的 jar 作为两个单独的构建产品,则可执行版本具有库 jar 中不需要的其他配置文件.例如,[application.yml](#) 配置文件可能被排除在不可执行的 JAR 中.

在 Maven 中,可执行 jar 必须是主要 artifacts ,您可以为库添加一个 classified jar,如下所示:

```

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <executions>
                <execution>
                    <id>lib</id>
                    <phase>package</phase>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <configuration>
                        <classifier>lib</classifier>
                        <excludes>
                            <exclude>application.yml</exclude>
                        </excludes>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

10.16.8. 远程调试以 Maven 开头的 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.

有关更多详细信息, 请参见此 [示例](#).

10.16.9. 在不使用 spring-boot-antlib 的情况下从 Ant 构建可执行归档文件

要使用 Ant 进行构建, 您需要获取依赖, 进行编译, 然后创建一个 jar 或 war 存档. 要使其可执行, 可以使用 `spring-boot-antlib` 模块, 也可以按照以下说明进行操作:

1. 如果您要构建 jar, 请将应用程序的类和资源打包在嵌套的 `BOOT-INF/classes` 目录中.
如果要打仗, 请照常将应用程序的类打包在嵌套的 `WEB-INF/classes` 目录中.

2. 将运行时依赖添加到 jar 的嵌套 **BOOT-INF/lib** 目录中, 或将其添加到 war 的 **WEB-INF/lib** 中. 切记不要压缩存档中的条目.
3. 将 **provided** 的 (嵌入式容器) 依赖添加到 jar 的嵌套 **BOOT-INF/lib** 目录中, 或将 war 添加到 **WEB-INF/lib** 提供的嵌套目录中. 切记不要压缩存档中的条目.
4. 在归档文件的根目录中添加 **spring-boot-loader** 类 (以便可以使用 **Main-Class**) .
5. 使用适当的启动器 (例如 jar 文件的 **JarLauncher**) 作为清单中的 **Main-Class** 属性, 并通过设置 **Start-Class** 属性指定它作为清单条目所需的其他属性.

以下示例显示了如何使用 Ant 构建可执行归档文件:

```
<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>
```

10.17. 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.

10.17.1. 创建可部署的 War 文件



由于 Spring WebFlux 不严格依赖 Servlet API，并且默认情况下将应用程序部署在嵌入式 Reactor Netty 服务器上，因此 WebFlux 应用程序不支持 War 部署。

产生可部署 war 文件的第一步是提供 `SpringBootServletInitializer` 子类并覆盖其 `configure` 方法。这样做可以利用 Spring Framework 的 Servlet 3.0 支持，并让您在 Servlet 容器启动应用程序时对其进行配置。通常，您应该更新应用程序的主类以扩展 `SpringBootServletInitializer`，如以下示例所示：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

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

下一步是更新构建配置，以使您的项目生成 war 文件而不是 jar 文件。如果您使用 Maven 和 `spring-boot-starter-parent`（为您配置 Maven 的 war 插件），则只需修改 `pom.xml` 即可将包装更改为 war，如下所示：

```
<packaging>war</packaging>
```

如果使用 Gradle，则需要修改 `build.gradle` 以将 war 插件应用于项目，如下所示：

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

该过程的最后一步是确保嵌入式 servlet 容器不干扰 war 文件所部署到的 servlet 容器。为此，您需要将嵌入式 Servlet 容器依赖性标记为已提供。

如果使用 `Maven`, 则以下示例将 `servlet` 容器 (在本例中为 `Tomcat`) 标记为已提供:

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <!-- ... -->
</dependencies>
```

如果使用 `Gradle`, 则以下示例将 `servlet` 容器 (在本例中为 `Tomcat`) 标记为已提供:

```
dependencies {
    // ...
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    // ...
}
```



与 `Gradle` 的 `compileOnly` 配置相比, `providerRuntime` 更受欢迎. 除其他限制外, `compileOnly` 依赖不在测试类路径上, 因此任何基于 Web 的集成测试都将失败.

如果您使用 [Spring Boot 构建工具](#), 则将提供的嵌入式 `servlet` 容器依赖标记为提供, 将生成可执行的 `war` 文件, 其中提供的依赖被打包在 `lib` 提供的目录中. 这意味着, 除了可以部署到 `Servlet` 容器之外, 还可以通过在命令行上使用 `java -jar` 运行应用程序.

10.17.2. 将现有应用程序转换为 Spring Boot

对于非 Web 应用程序, 将现有的 `Spring` 应用程序转换为 `Spring Boot` 应用程序应该很容易. 为此, 请丢弃创建您的 `ApplicationContext` 的代码, 并将其替换为对 `SpringApplication` 或 `SpringApplicationBuilder` 的调用. `Spring MVC` Web 应用程序通常适合于首先创建可部署的 `war` 应用程序, 然后再将其迁移到可执行的 `war` 或 `jar`. 请参阅有关将 [将 jar 转换为 war 的入门指南](#).

要通过扩展 `SpringBootServletInitializer` (例如, 在名为 `Application` 的类中) 并添加 `Spring Boot @SpringBootApplication` 注解来创建可部署的 `war`

,请使用类似于以下示例中所示的代码:

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        // Customize the application or call application.sources(...) to add
        sources
        // Since our example is itself a @Configuration class (via
        @SpringBootApplication)
        // we actually don't need to override this method.
        return application;
    }

}
```

请记住,无论您在源代码中放入什么内容,都仅是 `Spring ApplicationContext`. 通常,任何已经起作用的东西都应该在这里工作. 可能有些 `bean` 可以在以后删除,并让 `Spring Boot` 为它们提供自己的默认值,但是应该可以使某些东西工作,然后再执行此操作.

可以将静态资源移至类路径根目录中的 `/public` (或 `/static` 或 `/resources` 或 `/META-INF/resources`) . 这同样适用于 `messages.properties` (`Spring Boot`会在类路径的根目录中自动检测到该消息) .

在 `Spring DispatcherServlet` 和 `Spring Security` 中使用 `Vanilla` 不需要进一步更改. 如果您的应用程序中具有其他功能 (例如, 使用其他 `servlet` 或过滤器), 则可能需要通过替换 `web.xml` 中的那些元素来向 `Application` 上下文中添加一些配置, 如下所示:

- 类型为 `Servlet` 或 `ServletRegistrationBean` 的 `@Bean` 将该 `bean` 安装在容器中,就好像它是 `web.xml` 中的 `<servlet />` 和 `<servlet-mapping />` 一样.
- 类型为 `Filter` 或 `FilterRegistrationBean` 的 `@Bean` 的行为类似 (作为 `<filter />` 和 `<filter-mapping />`) .
- 可以通过应用程序中的 `@ImportResource` 添加 XML 文件中的 `ApplicationContext`. 或者 ,可以在几行中重新创建已经大量使用注解配置的简单情况作为 `@Bean` 定义.

war 文件运行后,可以通过向 `Application` 中添加 `main` 方法使其变为可执行文件,如以下示例所示:

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

如果您打算以 war 或可执行应用程序的形式启动应用程序,则需要使用 `SpringBootServletInitializer` 回调可用的方法和类似于以下类的 `main` 方法中的共享方法来共享构建器的自定义项:

```
@SpringBootApplication  
public class Application extends SpringBootServletInitializer  
{  
  
    @Override  
    protected SpringApplicationBuilder  
    configure(SpringApplicationBuilder builder) {  
        return configureApplication(builder);  
    }  
  
    public static void main(String[] args) {  
        configureApplication(new  
        SpringApplicationBuilder().run(args);  
    }  
  
    private static SpringApplicationBuilder  
    configureApplication(SpringApplicationBuilder builder) {  
        return  
        builder.sources(Application.class).bannerMode(Banner.Mode.OFF  
    );  
    }  
}
```



应用程序可以分为多个类别:

- 没有 `web.xml` 的Servlet 3.0+应用程序.
- 带有 `web.xml` 的应用程序.
- 具有上下文层次结构的应用程序.

- 没有上下文层次结构的应用程序.

所有这些都应该适合翻译,但是每种可能都需要稍微不同的技术.

如果 `Servlet 3.0+` 应用程序已经使用了 `Spring Servlet 3.0+` 初始化程序支持类,那么它们可能会很容易转换. 通常,来自现有 `WebApplicationInitializer` 的所有代码都可以移入 `SpringBootServletInitializer`. 如果您现有的应用程序具有多个 `ApplicationContext` (例如,如果使用 `AbstractDispatcherServletInitializer`) ,则您可以将所有上下文源组合到一个 `SpringApplication` 中.
您可能会遇到的主要并发症是,如果合并无效,则需要维护上下文层次结构. 有关示例,请参见有关 [构建层次结构的条目](#) . 通常需要分解包含特定于Web的功能的现有父上下文,以便所有 `ServletContextAware` 组件都位于子上下文中.

还不是 `Spring` 应用程序的应用程序可以转换为 `Spring Boot` 应用程序,前面提到的指南可能会有所帮助. 但是,您可能仍然遇到问题. 在这种情况下,我们建议 [使用 spring-boot 标签在Stack Overflow上提问](#).

10.17.3. 将 WAR 部署到 WebLogic

要将 `Spring Boot` 应用程序部署到 `WebLogic`,必须确保 `servlet` 初始化程序直接实现 `WebApplicationInitializer` (即使您从已经实现它的基类进行扩展) .

`WebLogic` 的典型初始化程序应类似于以下示例:

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements
WebApplicationInitializer {

}
```

如果使用 `Logback`,则还需要告诉 `WebLogic` 首选打包版本,而不是服务器预先安装的版本. 您可以通过添加具有以下内容的 `WEB-INF/weblogic.xml` 文件来实现:

```

<?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
        https://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
        http://xmlns.oracle.com/weblogic/weblogic-web-app
        https://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>

```

10.17.4. 使用 `Jedis` 代替 `Lettuce`

默认情况下, Spring Boot 启动器 (`spring-boot-starter-data-redis`) 使用 `Lettuce`. 您需要排除该依赖性, 而改为包含 `Jedis`. Spring Boot 管理这些依赖, 因此您无需指定版本即可切换到 `Jedis`.

以下示例显示了如何在 Maven 中执行此操作:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>

```

以下示例显示了如何在 Gradle 中执行此操作:

```

dependencies {
    implementation('org.springframework.boot:spring-boot-starter-data-redis') {
        exclude group: 'io.lettuce', module: 'lettuce-core'
    }
    implementation 'redis.clients:jedis'
    // ...
}

```

10.17.5. 在集成测试中使用 Testcontainers

[Testcontainers](#) 库提供了一种方法来管理在 Docker 容器中运行的服务。它与 JUnit 集成，允许您编写一个测试类，该类可以在运行任何测试之前启动容器。Testcontainers 在编写与真实的后端服务（例如 MySQL, MongoDB, Cassandra 等）进行通信的集成测试时特别有用。Testcontainers 可以在 Spring Boot 测试中使用，如下所示：

```

@SpringBootTest
@Testcontainers
class ExampleIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>();

}

```

在运行任何测试之前，这将启动运行 Neo4j 的 Docker 容器（如果 Docker 在本地运行）。在大多数情况下，您将需要使用正在运行的容器中的详细信息（例如容器 IP 或端口）来配置应用程序。

这可以通过静态 [@DynamicPropertySource](#) 方法完成，该方法允许向 Spring Environment 添加动态属性值。

```
@SpringBootTest
@Testcontainers
class ExampleIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>();

    @DynamicPropertySource
    static void neo4jProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.neo4j.uri", neo4j::getBoltUrl);
    }

}
```

上面的配置允许应用程序中与 Neo4j 相关的 Bean 与在 `Testcontainers` 管理的 Docker 容器中运行的 Neo4j 通信.

Chapter 11. Appendices

Appendix A: 公共应用程序属性

可以在 `application.properties` , `application.yml` 文件内或作为命令行开关指定各种属性. 本附录提供了常见的 Spring Boot 属性列表以及对使用它们的基础类的引用.



Spring Boot 提供了各种具有高级值格式的 [属性转换](#) 机制, 请务必查看属性转换部分.



属性提供者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. 另外, 您可以定义自己的属性.

11.A.1. Core Properties

Key	Default Value	Description
<code>debug</code>	<code>false</code>	Enable debug logs.
<code>info.*</code>		Arbitrary properties to add to the info endpoint.
<code>logging.charset.console</code>		Charset to use for console output.
<code>logging.charset.file</code>		Charset to use for file output.

Key	Default Value	Description
<code>logging.config</code>		Location of the logging configuration file. For instance, `classpath:logback.xml` for Logback.
<code>logging.exception-conversion-word</code>	<code>%wEx</code>	Conversion word used when logging exceptions.
<code>logging.file.name</code>		Log file name (for instance, `myapp.log`). Names can be an exact location or relative to the current directory.
<code>logging.file.path</code>		Location of the log file. For instance, `/var/log`.
<code>logging.group.*</code>		Log groups to quickly change multiple loggers at the same time. For instance, `logging.group.db =org.hibernate,org.springframework.jdbc`.

Key	Default Value	Description
<code>logging.level.*</code>		Log levels severity mapping. For instance, `logging.level.org.springframework=DEBUG`.
<code>logging.logback.rollingpolicy.clean</code> <code>-history-on-start</code>	<code>false</code>	Whether to clean the archive log files on startup.
<code>logging.logback.rollingpolicy.file-name-pattern</code>	<code> \${LOG_FILE}.%d{yy-MM-dd}.%i.gz</code>	Pattern for rolled-over log file names.
<code>logging.logback.rollingpolicy.max-file-size</code>	<code>10MB</code>	Maximum log file size.
<code>logging.logback.rollingpolicy.max-history</code>	<code>7</code>	Maximum number of days archive log files are kept.
<code>logging.logback.rollingpolicy.total-size-cap</code>	<code>0B</code>	Total size of log backups to be kept.

Key	Default Value	Description
<code>logging.pattern.console</code>	<pre>%clr(%d\${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(\${LOG_LEVEL_PATTERN:-%5p}) %clr(\${PID:-}){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n\${LOG_EXCEPTION_CONVERSION_WOR D:-%wEx}</pre>	Appender pattern for output to the console. Supported only with the default Logback setup.
<code>logging.pattern.dateformat</code>	<code>yyyy-MM-dd HH:mm:ss.SSS</code>	Appender pattern for log date format. Supported only with the default Logback setup.

Key	Default Value	Description
<code>logging.pattern.file</code>	<code>%d\${LOG_DATEFORMAT} AT_PATTERN:-yyyy- MM-dd HH:mm:ss.SSS} \${LOG_LEVEL_PATTE RN:-%5p} \${PID:- } --- [%t] %- 40.40logger{39} : %m%n\${LOG_EXCEPTI ON_CONVERSION_WOR D:-%wEx}</code>	Appender pattern for output to a file. Supported only with the default Logback setup.
<code>logging.pattern.level</code>	<code>%5p</code>	Appender pattern for log level. Supported only with the default Logback setup.
<code>logging.register-shutdown-hook</code>	<code>false</code>	Register a shutdown hook for the logging system when it is initialized.
<code>spring.aop.auto</code>	<code>true</code>	Add <code>@EnableAspectJAutoProxy</code> .
<code>spring.aop.proxy-target-class</code>	<code>true</code>	Whether subclass-based (CGLIB) proxies are to be created (<code>true</code>), as opposed to standard Java interface-based proxies (<code>false</code>).

Key	Default Value	Description
<code>spring.application.admin.enabled</code>	<code>false</code>	Whether to enable admin features for the application.
<code>spring.application.admin.jmx-name</code>	<code>org.springframework.boot:type=Admin, name=SpringApplication</code>	JMX name of the application admin MBean.
<code>spring.application.name</code>		Application name.
<code>spring.autoconfigure.exclude</code>		Auto-configuration classes to exclude.
<code>spring.banner.charset</code>	<code>UTF-8</code>	Banner file encoding.
<code>spring.banner.image.bitdepth</code>	<code>4</code>	Bit depth to use for ANSI colors. Supported values are 4 (16 color) or 8 (256 color).
<code>spring.banner.image.height</code>		Height of the banner image in chars (default based on image height).
<code>spring.banner.image.invert</code>	<code>false</code>	Whether images should be inverted for dark terminal themes.

Key	Default Value	Description
<code>spring.banner.image.location</code>	<code>classpath:banner.gif</code>	Banner image file location (jpg or png can also be used).
<code>spring.banner.image.margin</code>	2	Left hand image margin in chars.
<code>spring.banner.image.pixelmode</code>	TEXT	Pixel mode to use when rendering the image.
<code>spring.banner.image.width</code>	76	Width of the banner image in chars.
<code>spring.banner.location</code>	<code>classpath:banner.txt</code>	Banner text resource location.
<code>spring.beaninfo.ignore</code>	true	Whether to skip search of BeanInfo classes.
<code>spring.codec.log-request-details</code>	false	Whether to log form data at DEBUG level, and headers at TRACE level.

Key	Default Value	Description
<code>spring.codec.max-in-memory-size</code>		Limit on the number of bytes that can be buffered whenever the input stream needs to be aggregated. This applies only to the auto-configured WebFlux server and WebClient instances. By default this is not set, in which case individual codec defaults apply. Most codecs are limited to 256K by default.
<code>spring.config.activate.on-cloud-platform</code>		Required cloud platform for the document to be included.
<code>spring.config.activate.on-profile</code>		Profile expressions that should match for the document to be included.

Key	Default Value	Description
<code>spring.config.additional-location</code>		Config file locations used in addition to the defaults.
<code>spring.config.import</code>		Import additional config data.
<code>spring.config.location</code>		Config file locations that replace the defaults.
<code>spring.config.name</code>	<code>application</code>	Config file name.
<code>spring.config.use-legacy-processing</code>	<code>false</code>	Whether to enable configuration data processing legacy mode.
<code>spring.info.build.encoding</code>	<code>UTF-8</code>	File encoding.
<code>spring.info.build.location</code>	<code>classpath:META-INF/build-info.properties</code>	Location of the generated build-info.properties file.
<code>spring.info.git.encoding</code>	<code>UTF-8</code>	File encoding.
<code>spring.info.git.location</code>	<code>classpath:git.properties</code>	Location of the generated git.properties file.
<code>spring.jmx.default-domain</code>		JMX domain name.
<code>spring.jmx.enabled</code>	<code>false</code>	Expose management beans to the JMX domain.

Key	Default Value	Description
<code>spring.jmx.server</code>	<code>mbeanServer</code>	MBeanServer bean name.
<code>spring.jmx.unique-names</code>	<code>false</code>	Whether unique runtime object names should be ensured.
<code>spring.lifecycle.timeout-per-shutdown-phase</code>	<code>30s</code>	Timeout for the shutdown of any phase (group of SmartLifecycle beans with the same 'phase' value).
<code>spring.main.allow-bean-definition-overriding</code>	<code>false</code>	Whether bean definition overriding, by registering a definition with the same name as an existing definition, is allowed.
<code>spring.main.banner-mode</code>	<code>console</code>	Mode used to display the banner when the application runs.
<code>spring.main.cloud-platform</code>		Override the Cloud Platform auto-detection.

Key	Default Value	Description
<code>spring.main.lazy-initialization</code>	<code>false</code>	Whether initialization should be performed lazily.
<code>spring.main.log-startup-info</code>	<code>true</code>	Whether to log information about the application when it starts.
<code>spring.main.register-shutdown-hook</code>	<code>true</code>	Whether the application should have a shutdown hook registered.
<code>spring.main.sources</code>		Sources (class names, package names, or XML resource locations) to include in the ApplicationContext.
<code>spring.main.web-application-type</code>		Flag to explicitly request a specific type of web application. If not set, auto-detected based on the classpath.

Key	Default Value	Description
<code>spring.mandatory-file-encoding</code>		Expected character encoding the application must use.
<code>spring.messages.always-use-message-format</code>	<code>false</code>	Whether to always apply the <code>MessageFormat</code> rules, parsing even messages without arguments.
<code>spring.messages.basename</code>	<code>messages</code>	Comma-separated list of basenames (essentially a fully-qualified classpath location), each following the <code>ResourceBundle</code> convention with relaxed support for slash based locations. If it doesn't contain a package qualifier (such as <code>"orgmypackage"</code>), it will be resolved from the classpath root.

Key	Default Value	Description
<code>spring.messages.cache-duration</code>		Loaded resource bundle files cache duration. When not set, bundles are cached forever. If a duration suffix is not specified, seconds will be used.
<code>spring.messages.encoding</code>	UTF-8	Message bundles encoding.
<code>spring.messages.fallback-to-system-locale</code>	true	Whether to fall back to the system Locale if no files for a specific Locale have been found. if this is turned off, the only fallback will be the default file (e.g. "messages.properties" for basename "messages").

Key	Default Value	Description
<code>spring.messages.use-code-as-default-message</code>	<code>false</code>	Whether to use the message code as the default message instead of throwing a "NoSuchMessageException". Recommended during development only.
<code>spring.output.ansi.enabled</code>	<code>detect</code>	Configures the ANSI output.
<code>spring.pid.fail-on-write-error</code>		Fails if ApplicationPidFileWriter is used but it cannot write the PID file.
<code>spring.pid.file</code>		Location of the PID file to write (if ApplicationPidFileWriter is used).
<code>spring.profiles.active</code>		Comma-separated list of active profiles. Can be overridden by a command line switch.

Key	Default Value	Description
<code>spring.profiles.include</code>		Unconditionally activate the specified comma-separated list of profiles (or list of profiles if using YAML).
<code>spring.quartz.auto-startup</code>	<code>true</code>	Whether to automatically start the scheduler after initialization.
<code>spring.quartz.jdbc.comment-prefix</code>	<code>[#, --]</code>	Prefixes for single-line comments in SQL initialization scripts.
<code>spring.quartz.jdbc.initialize-schema</code>	<code>embedded</code>	Database schema initialization mode.
<code>spring.quartz.jdbc.schema</code>	<code>classpath:org/quartz/impl/jdbcjobs/tore/tables_@@platfrom@@.sql</code>	Path to the SQL file to use to initialize the database schema.
<code>spring.quartz.job-store-type</code>	<code>memory</code>	Quartz job store type.
<code>spring.quartz.overwrite-existing-jobs</code>	<code>false</code>	Whether configured jobs should overwrite existing job definitions.

Key	Default Value	Description
<code>spring.quartz.properties.*</code>		Additional Quartz Scheduler properties.
<code>spring.quartz.scheduler-name</code>	<code>quartzScheduler</code>	Name of the scheduler.
<code>spring.quartz.startup-delay</code>	<code>0s</code>	Delay after which the scheduler is started once initialization completes. Setting this property makes sense if no jobs should be run before the entire application has started up.
<code>spring.quartz.wait-for-jobs-to-complete-on-shutdown</code>	<code>false</code>	Whether to wait for running jobs to complete on shutdown.
<code>spring.reactor.debug-agent.enabled</code>	<code>true</code>	Whether the Reactor Debug Agent should be enabled when reactor-tools is present.

Key	Default Value	Description
<code>spring.task.execution.pool.allow-core-thread-timeout</code>	<code>true</code>	Whether core threads are allowed to time out. This enables dynamic growing and shrinking of the pool.
<code>spring.task.execution.pool.core-size</code>	<code>8</code>	Core number of threads.
<code>spring.task.execution.pool.keep-alive</code>	<code>60s</code>	Time limit for which threads may remain idle before being terminated.
<code>spring.task.execution.pool.max-size</code>		Maximum allowed number of threads. If tasks are filling up the queue, the pool can expand up to that size to accommodate the load. Ignored if the queue is unbounded.
<code>spring.task.execution.pool.queue-capacity</code>		Queue capacity. An unbounded capacity does not increase the pool and therefore ignores the "max-size" property.

Key	Default Value	Description
<code>spring.task.execution.shutdown.await-termination</code>	false	Whether the executor should wait for scheduled tasks to complete on shutdown.
<code>spring.task.execution.shutdown.await-termination-period</code>		Maximum time the executor should wait for remaining tasks to complete.
<code>spring.task.execution.thread-name-prefix</code>	task-	Prefix to use for the names of newly created threads.
<code>spring.task.scheduling.pool.size</code>	1	Maximum allowed number of threads.
<code>spring.task.scheduling.shutdown.await-termination</code>	false	Whether the executor should wait for scheduled tasks to complete on shutdown.
<code>spring.task.scheduling.shutdown.await-termination-period</code>		Maximum time the executor should wait for remaining tasks to complete.

Key	Default Value	Description
<code>spring.task.scheduling.thread-name-prefix</code>	<code>scheduling-</code>	Prefix to use for the names of newly created threads.
<code>trace</code>	<code>false</code>	Enable trace logs.

11.A.2. Cache Properties

Key	Default Value	Description
<code>spring.cache.cache-names</code>		Comma-separated list of cache names to create if supported by the underlying cache manager. Usually, this disables the ability to create additional caches on-the-fly.
<code>spring.cache.caffeine.spec</code>		The spec to use to create caches. See CaffeineSpec for more details on the spec format.

Key	Default Value	Description
<code>spring.cache.couchbase.expiration</code>		Entry expiration. By default the entries never expire. Note that this value is ultimately converted to seconds.
<code>spring.cache.ehcache.config</code>		The location of the configuration file to use to initialize EhCache.
<code>spring.cache.infinispan.config</code>		The location of the configuration file to use to initialize Infinispan.
<code>spring.cache.jcache.config</code>		The location of the configuration file to use to initialize the cache manager. The configuration file is dependent of the underlying cache implementation.

Key	Default Value	Description
<code>spring.cache.jcache.provider</code>		Fully qualified name of the CachingProvider implementation to use to retrieve the JSR-107 compliant cache manager. Needed only if more than one JSR-107 implementation is available on the classpath.
<code>spring.cache.redis.cache-null-values</code>	<code>true</code>	Allow caching null values.
<code>spring.cache.redis.enable-statistics</code>	<code>false</code>	Whether to enable cache statistics.
<code>spring.cache.redis.key-prefix</code>		Key prefix.
<code>spring.cache.redis.time-to-live</code>		Entry expiration. By default the entries never expire.
<code>spring.cache.redis.use-key-prefix</code>	<code>true</code>	Whether to use the key prefix when writing to Redis.
<code>spring.cache.type</code>		Cache type. By default, auto-detected according to the environment.

11.A.3. Mail Properties

Key	Default Value	Description
<code>spring.mail.default-encoding</code>	UTF-8	Default MimeMessage encoding.
<code>spring.mail.host</code>		SMTP server host. For instance, `smtp.example.com`.
<code>spring.mail.jndi-name</code>		Session JNDI name. When set, takes precedence over other Session settings.
<code>spring.mail.password</code>		Login password of the SMTP server.
<code>spring.mail.port</code>		SMTP server port.
<code>spring.mail.properties.*</code>		Additional JavaMail Session properties.
<code>spring.mail.protocol</code>	smtp	Protocol used by the SMTP server.
<code>spring.mail.test-connection</code>	false	Whether to test that the mail server is available on startup.
<code>spring.mail.username</code>		Login user of the SMTP server.
<code>spring.sendgrid.api-key</code>		SendGrid API key.

Key	Default Value	Description
<code>spring.sendgrid.proxy.host</code>		SendGrid proxy host.
<code>spring.sendgrid.proxy.port</code>		SendGrid proxy port.

11.A.4. JSON Properties

Key	Default Value	Description
<code>spring.gson.date-format</code>		Format to use when serializing Date objects.
<code>spring.gson.disable-html-escaping</code>		Whether to disable the escaping of HTML characters such as '<', '>', etc.
<code>spring.gson.disable-inner-class-serialization</code>		Whether to exclude inner classes during serialization.
<code>spring.gson.enable-complex-map-key-serialization</code>		Whether to enable serialization of complex map keys (i.e. non-primitives).

Key	Default Value	Description
<code>spring.gson.exclude-fields-without-expose-annotation</code>		Whether to exclude all fields from consideration for serialization or deserialization that do not have the "Expose" annotation.
<code>spring.gson.field-naming-policy</code>		Naming policy that should be applied to an object's field during serialization and deserialization.
<code>spring.gson.generate-non-executable-json</code>		Whether to generate non executable JSON by prefixing the output with some special text.
<code>spring.gson.lenient</code>		Whether to be lenient about parsing JSON that doesn't conform to RFC 4627.
<code>spring.gson.long-serialization-policy</code>		Serialization policy for Long and long types.

Key	Default Value	Description
<code>spring.gson.pretty-printing</code>		Whether to output serialized JSON that fits in a page for pretty printing.
<code>spring.gson.serialize-nulls</code>		Whether to serialize null fields.
<code>spring.jackson.date-format</code>		Date format string or a fully-qualified date format class name. For instance, `yyyy-MM-dd HH:mm:ss`.
<code>spring.jackson.default-property-inclusion</code>		Controls the inclusion of properties during serialization. Configured with one of the values in Jackson's <code>JsonInclude.Include</code> enumeration.
<code>spring.jackson.deserialization.*</code>		Jackson on/off features that affect the way Java objects are deserialized.
<code>spring.jackson.generator.*</code>		Jackson on/off features for generators.

Key	Default Value	Description
<code>spring.jackson.locale</code>		Locale used for formatting.
<code>spring.jackson.mapper.*</code>		Jackson general purpose on/off features.
<code>spring.jackson.parser.*</code>		Jackson on/off features for parsers.
<code>spring.jackson.property-naming-strategy</code>		One of the constants on Jackson's PropertyNamingStrategy. Can also be a fully-qualified class name of a PropertyNamingStrategy subclass.
<code>spring.jackson.serialization.*</code>		Jackson on/off features that affect the way Java objects are serialized.
<code>spring.jackson.time-zone</code>		Time zone used when formatting dates. For instance, "America/Los_Angeles" or "GMT+10".

Key	Default Value	Description
<code>spring.jackson.visibility.*</code>		Jackson visibility thresholds that can be used to limit which methods (and fields) are auto-detected.

11.A.5. Data Properties

Key	Default Value	Description
<code>spring.couchbase.connection-string</code>		Connection string used to locate the Couchbase cluster.
<code>spring.couchbase.env.io.idle-socket-connection-timeout</code>	<code>4500ms</code>	Length of time an HTTP connection may remain idle before it is closed and removed from the pool.
<code>spring.couchbase.env.io.max-endpoints</code>	<code>12</code>	Maximum number of sockets per node.
<code>spring.couchbase.env.io.min-endpoints</code>	<code>1</code>	Minimum number of sockets per node.

Key	Default Value	Description
<code>spring.couchbase.env.ssl.enabled</code>		Whether to enable SSL support. Enabled automatically if a "keyStore" is provided unless specified otherwise.
<code>spring.couchbase.env.ssl.key-store</code>		Path to the JVM key store that holds the certificates.
<code>spring.couchbase.env.ssl.key-store-password</code>		Password used to access the key store.
<code>spring.couchbase.env.timeouts.analytics</code>	75s	Timeout for the analytics service.
<code>spring.couchbase.env.timeouts.connect</code>	10s	Bucket connect timeout.
<code>spring.couchbase.env.timeouts.disconnect</code>	10s	Bucket disconnect timeout.
<code>spring.couchbase.env.timeouts.key-value</code>	2500ms	Timeout for operations on a specific key-value.
<code>spring.couchbase.env.timeouts.key-value-durable</code>	10s	Timeout for operations on a specific key-value with a durability level.

Key	Default Value	Description
<code>spring.couchbase.env.timeouts.management</code>	75s	Timeout for the management operations.
<code>spring.couchbase.env.timeouts.query</code>	75s	N1QL query operations timeout.
<code>spring.couchbase.env.timeouts.search</code>	75s	Timeout for the search service.
<code>spring.couchbase.env.timeouts.view</code>	75s	Regular and geospatial view operations timeout.
<code>spring.couchbase.password</code>		Cluster password.
<code>spring.couchbase.username</code>		Cluster username.
<code>spring.dao.exceptiontranslation.enabled</code>	true	Whether to enable the PersistenceExceptionTranslationPostProcessor.
<code>spring.data.cassandra.compression</code>	none	Compression supported by the Cassandra binary protocol.
<code>spring.data.cassandra.connection.connect-timeout</code>	5s	Timeout to use when establishing driver connections.

Key	Default Value	Description
<code>spring.data.cassandra.connection.initial-query-timeout</code>	5s	Timeout to use for internal queries that run as part of the initialization process, just after a connection is opened.
<code>spring.data.cassandra.contact-points</code>	[127.0.0.1:9042]	Cluster node addresses in the form 'host:port', or a simple 'host' to use the configured port.
<code>spring.data.cassandra.keyspace-name</code>		Keyspace name to use.
<code>spring.data.cassandra.local-datacenter</code>		Datacenter that is considered "local". Contact points should be from this datacenter.
<code>spring.data.cassandra.password</code>		Login password of the server.
<code>spring.data.cassandra.pool.heartbeat-interval</code>	30s	Heartbeat interval after which a message is sent on an idle connection to make sure it's still alive.

Key	Default Value	Description
<code>spring.data.cassandra.pool.idle-timeout</code>	5s	Idle timeout before an idle connection is removed.
<code>spring.data.cassandra.port</code>	9042	Port to use if a contact point does not specify one.
<code>spring.data.cassandra.repositories.auto-type</code>	auto	Type of Cassandra repositories to enable.
<code>spring.data.cassandra.request.consistency</code>		Queries consistency level.
<code>spring.data.cassandra.request.page-size</code>	5000	How many rows will be retrieved simultaneously in a single network roundtrip.
<code>spring.data.cassandra.request.serial-consistency</code>		Queries serial consistency level.

Key	Default Value	Description
<code>spring.data.cassandra.request.throt tler.drain-interval</code>	10ms	How often the throttler attempts to dequeue requests. Set this high enough that each attempt will process multiple entries in the queue, but not delay requests too much.
<code>spring.data.cassandra.request.throt tler.max-concurrent-requests</code>	10000	Maximum number of requests that are allowed to execute in parallel.
<code>spring.data.cassandra.request.throt tler.max-queue-size</code>	10000	Maximum number of requests that can be enqueued when the throttling threshold is exceeded.
<code>spring.data.cassandra.request.throt tler.max-requests-per-second</code>	10000	Maximum allowed request rate.
<code>spring.data.cassandra.request.throt tler.type</code>	none	Request throttling type.
<code>spring.data.cassandra.request.timeout</code>	2s	How long the driver waits for a request to complete.

Key	Default Value	Description
<code>spring.data.cassandra.schema-action</code>	<code>none</code>	Schema action to take at startup.
<code>spring.data.cassandra.session-name</code>		Name of the Cassandra session.
<code>spring.data.cassandra.ssl</code>	<code>false</code>	Enable SSL support.
<code>spring.data.cassandra.username</code>		Login user of the server.
<code>spring.data.couchbase.auto-index</code>	<code>false</code>	Automatically create views and indexes. Use the meta-data provided by "@ViewIndexed", "@N1qlPrimaryIndexed" and "@N1qlSecondaryIndexed".
<code>spring.data.couchbase.bucket-name</code>		Name of the bucket to connect to.
<code>spring.data.couchbase.field-naming-strategy</code>		Fully qualified name of the FieldNamingStrategy to use.
<code>spring.data.couchbase.repositories.auto-type</code>	<code>auto</code>	Type of Couchbase repositories to enable.

Key	Default Value	Description
<code>spring.data.couchbase.scope-name</code>		Name of the scope used for all collection access.
<code>spring.data.couchbase.type-key</code>	<code>_class</code>	Name of the field that stores the type information for complex types when using "MappingCouchbase Converter".
<code>spring.data.elasticsearch.client.reactive.connection-timeout</code>		Connection timeout.
<code>spring.data.elasticsearch.client.reactive.endpoints</code>		Comma-separated list of the Elasticsearch endpoints to connect to.
<code>spring.data.elasticsearch.client.reactive.max-in-memory-size</code>		Limit on the number of bytes that can be buffered whenever the input stream needs to be aggregated.
<code>spring.data.elasticsearch.client.reactive.password</code>		Credentials password.
<code>spring.data.elasticsearch.client.reactive.socket-timeout</code>		Read and Write Socket timeout.

Key	Default Value	Description
<code>spring.data.elasticsearch.client.reactive.use-ssl</code>	false	Whether the client should use SSL to connect to the endpoints.
<code>spring.data.elasticsearch.client.reactive.username</code>		Credentials username.
<code>spring.data.elasticsearch.repositories.enabled</code>	true	Whether to enable Elasticsearch repositories.
<code>spring.data.jdbc.repositories.enabled</code>	true	Whether to enable JDBC repositories.
<code>spring.data.jpa.repositories.bootstrap-mode</code>	default	Bootstrap mode for JPA repositories.
<code>spring.data.jpa.repositories.enabled</code>	true	Whether to enable JPA repositories.
<code>spring.data.ldap.repositories.enabled</code>	true	Whether to enable LDAP repositories.
<code>spring.data.mongodb.authentication-database</code>		Authentication database name.
<code>spring.data.mongodb.auto-index-creation</code>		Whether to enable auto-index creation.
<code>spring.data.mongodb.database</code>		Database name.
<code>spring.data.mongodb.field-naming-strategy</code>		Fully qualified name of the FieldNamingStrategy to use.

Key	Default Value	Description
<code>spring.data.mongodb.gridfs.bucket</code>		GridFS bucket name.
<code>spring.data.mongodb.gridfs.database</code>		GridFS database name.
<code>spring.data.mongodb.host</code>		Mongo server host. Cannot be set with URI.
<code>spring.data.mongodb.password</code>		Login password of the mongo server. Cannot be set with URI.
<code>spring.data.mongodb.port</code>		Mongo server port. Cannot be set with URI.
<code>spring.data.mongodb.replica-set-name</code>		Required replica set name for the cluster. Cannot be set with URI.
<code>spring.data.mongodb.repositories.type</code>	<code>auto</code>	Type of Mongo repositories to enable.
<code>spring.data.mongodb.uri</code>	<code>mongodb://localhost/test</code>	Mongo database URI. Cannot be set with host, port, credentials and replica set name.
<code>spring.data.mongodb.username</code>		Login user of the mongo server. Cannot be set with URI.

Key	Default Value	Description
<code>spring.data.mongodb.uuid-representation</code>	<code>java-legacy</code>	Representation to use when converting a UUID to a BSON binary value.
<code>spring.data.neo4j.database</code>		Database name to use. By default, the server decides the default database to use.
<code>spring.data.neo4j.repositories.type</code>	<code>auto</code>	Type of Neo4j repositories to enable.
<code>spring.data.r2dbc.repositories.enabled</code>	<code>true</code>	Whether to enable R2DBC repositories.
<code>spring.data.redis.repositories.enabled</code>	<code>true</code>	Whether to enable Redis repositories.
<code>spring.data.rest.base-path</code>		Base path to be used by Spring Data REST to expose repository resources.
<code>spring.data.rest.default-media-type</code>		Content type to use as a default when none is specified.
<code>spring.data.rest.default-page-size</code>		Default size of pages.

Key	Default Value	Description
<code>spring.data.rest.detection-strategy</code>	<code>default</code>	Strategy to use to determine which repositories get exposed.
<code>spring.data.rest.enable-enum-translation</code>		Whether to enable enum value translation through the Spring Data REST default resource bundle.
<code>spring.data.rest.limit-param-name</code>		Name of the URL query string parameter that indicates how many results to return at once.
<code>spring.data.rest.max-page-size</code>		Maximum size of pages.
<code>spring.data.rest.page-param-name</code>		Name of the URL query string parameter that indicates what page to return.
<code>spring.data.rest.return-body-on-create</code>		Whether to return a response body after creating an entity.

Key	Default Value	Description
<code>spring.data.rest.return-body-on-update</code>		Whether to return a response body after updating an entity.
<code>spring.data.rest.sort-param-name</code>		Name of the URL query string parameter that indicates what direction to sort results.
<code>spring.data.solr.host</code>	<code>http://127.0.0.1:8983/solr</code>	Solr host. Ignored if "zk-host" is set.
<code>spring.data.solr.zk-host</code>		ZooKeeper host address in the form HOST:PORT.
<code>spring.data.web.pageable.default-page-size</code>	<code>20</code>	Default page size.
<code>spring.data.web.pageable.max-page-size</code>	<code>2000</code>	Maximum page size to be accepted.
<code>spring.data.web.pageable.one-indexed-parameters</code>	<code>false</code>	Whether to expose and assume 1-based page number indexes. Defaults to "false", meaning a page number of 0 in the request equals the first page.
<code>spring.data.web.pageable.page-parameter</code>	<code>page</code>	Page index parameter name.

Key	Default Value	Description
<code>spring.data.web.pageable.prefix</code>		General prefix to be prepended to the page number and page size parameters.
<code>spring.data.web.pageable.qualifier-delimiter</code>	_	Delimiter to be used between the qualifier and the actual page number and size properties.
<code>spring.data.web.pageable.size-parameter</code>	<code>size</code>	Page size parameter name.
<code>spring.data.web.sort.sort-parameter</code>	<code>sort</code>	Sort parameter name.
<code>spring.datasource.continue-on-error</code>	<code>false</code>	Whether to stop if an error occurs while initializing the database.
<code>spring.datasource.data</code>		Data (DML) script resource references.
<code>spring.datasource.data-password</code>		Password of the database to execute DML scripts (if different).

Key	Default Value	Description
spring.datasource.data-username		Username of the database to execute DML scripts (if different).

Key	Default Value	Description
<pre>spring.datasource.dbcp2.abandoned-usage-tracking spring.datasource.dbcp2.access-to-underlying-connection-allowed spring.datasource.dbcp2.auto-commit-on-return spring.datasource.dbcp2.cache-state spring.datasource.dbcp2.clear-statement-pool-on-return spring.datasource.dbcp2.connection-factory-class-name spring.datasource.dbcp2.connection-init-sqls spring.datasource.dbcp2.default-auto-commit spring.datasource.dbcp2.default-catalog spring.datasource.dbcp2.default-query-timeout spring.datasource.dbcp2.default-read-only spring.datasource.dbcp2.default-schema spring.datasource.dbcp2.default-transaction-isolation spring.datasource.dbcp2.disconnect-on-sql-codes spring.datasource.dbcp2.driver spring.datasource.dbcp2.driver-class-name spring.datasource.dbcp2.ejection-policy-class-name spring.datasource.dbcp2.fast-fail-validation spring.datasource.dbcp2.initial-</pre>		Commons DBCP2 specific settings bound to an instance of DBCP2's BasicDataSource

Key	Default Value	Description
<code>spring.datasource.driver-class-name</code>		Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.
<code>spring.datasource.generate-unique-name</code>	<code>true</code>	Whether to generate a random datasource name.

Key	Default Value	Description
<pre>spring.datasource.hikari.allow- pool-suspension spring.datasource.hikari.auto- commit spring.datasource.hikari.catalog spring.datasource.hikari.connection- -init-sql spring.datasource.hikari.connection- -test-query spring.datasource.hikari.connection- -timeout spring.datasource.hikari.data- source-class-name spring.datasource.hikari.data- source-j-n-d-i spring.datasource.hikari.data- source-properties spring.datasource.hikari.driver- class-name spring.datasource.hikari.exception- override-class-name spring.datasource.hikari.health- check-properties spring.datasource.hikari.health- check-registry spring.datasource.hikari.idle- timeout spring.datasource.hikari.initializa- tion-fail-timeout spring.datasource.hikari.isolate- internal-queries spring.datasource.hikari.jdbc-url spring.datasource.hikari.leak- detection-threshold spring.datasource.hikari.login-</pre>		Hikari specific settings bound to an instance of Hikari's HikariDataSource

Key	Default Value	Description
<code>spring.datasource.initialization-mode</code>	<code>embedded</code>	Mode to apply when determining if DataSource initialization should be performed using the available DDL and DML scripts.
<code>spring.datasource.jndi-name</code>		JNDI location of the datasource. Class, url, username and password are ignored when set.
<code>spring.datasource.name</code>		Name of the datasource. Default to "testdb" when using an embedded database.

Key	Default Value	Description
<code>spring.datasource.oracleucp.abandoned-connection-timeout</code> <code>spring.datasource.oracleucp.connection-factory-class-name</code> <code>spring.datasource.oracleucp.connection-factory-properties</code> <code>spring.datasource.oracleucp.connection-harvest-max-count</code> <code>spring.datasource.oracleucp.connection-harvest-trigger-count</code> <code>spring.datasource.oracleucp.connection-labeling-high-cost</code> <code>spring.datasource.oracleucp.connection-pool-name</code> <code>spring.datasource.oracleucp.connection-properties</code> <code>spring.datasource.oracleucp.connection-repurpose-threshold</code> <code>spring.datasource.oracleucp.connection-validation-timeout</code> <code>spring.datasource.oracleucp.connection-wait-timeout</code> <code>spring.datasource.oracleucp.data-source-name</code> <code>spring.datasource.oracleucp.database-name</code> <code>spring.datasource.oracleucp.description</code> <code>spring.datasource.oracleucp.fast-connection-failover-enabled</code> <code>spring.datasource.oracleucp.high-cost-connection-reuse-threshold</code> <code>spring.datasource.oracleucp.inactive-connection-timeout</code> <code>spring.datasource.oracleucp.initial</code>		Oracle UCP specific settings bound to an instance of Oracle UCP's PoolDataSource

Key	Default Value	Description
<code>spring.datasource.password</code>		Login password of the database.
<code>spring.datasource.platform</code>	<code>all</code>	Platform to use in the DDL or DML scripts (such as schema- <code> \${platform}.sql</code> or data- <code> \${platform}.sql</code>).
<code>spring.datasource.schema</code>		Schema (DDL) script resource references.
<code>spring.datasource.schema-password</code>		Password of the database to execute DDL scripts (if different).
<code>spring.datasource.schema-username</code>		Username of the database to execute DDL scripts (if different).
<code>spring.datasource.separator</code>	<code>;</code>	Statement separator in SQL initialization scripts.
<code>spring.datasource.sql-script-encoding</code>		SQL scripts encoding.

Key	Default Value	Description
<pre>spring.datasource.tomcat.abandon- when-percentage-full spring.datasource.tomcat.access-to- underlying-connection-allowed spring.datasource.tomcat.alternate- username-allowed spring.datasource.tomcat.commit-on- return spring.datasource.tomcat.connection- properties spring.datasource.tomcat.data- source spring.datasource.tomcat.data- source-j-n-d-i spring.datasource.tomcat.db- properties spring.datasource.tomcat.default- auto-commit spring.datasource.tomcat.default- catalog spring.datasource.tomcat.default- read-only spring.datasource.tomcat.default- transaction-isolation spring.datasource.tomcat.driver- class-name spring.datasource.tomcat.fair-queue spring.datasource.tomcat.ignore- exception-on-pre-load spring.datasource.tomcat.init-s-q-l spring.datasource.tomcat.initial- size spring.datasource.tomcat.jdbc- interceptors spring.datasource.tomcat.jmx-</pre>		Tomcat datasource specific settings bound to an instance of Tomcat JDBC's DataSource

Key	Default Value	Description
<code>spring.datasource.type</code>		Fully qualified name of the connection pool implementation to use. By default, it is auto-detected from the classpath.
<code>spring.datasource.url</code>		JDBC URL of the database.
<code>spring.datasource.username</code>		Login username of the database.
<code>spring.datasource.xa.data-source-class-name</code>		XA datasource fully qualified name.
<code>spring.datasource.xa.properties.*</code>		Properties to pass to the XA data source.
<code>spring.elasticsearch.rest.connectio</code> <code>n-timeout</code>	1s	Connection timeout.
<code>spring.elasticsearch.rest.password</code>		Credentials password.
<code>spring.elasticsearch.rest.read-</code> <code>timeout</code>	30s	Read timeout.
<code>spring.elasticsearch.rest.uris</code>	[http://localhost:9200]	Comma-separated list of the Elasticsearch instances to use.
<code>spring.elasticsearch.rest.username</code>		Credentials username.

Key	Default Value	Description
<code>spring.h2.console.enabled</code>	<code>false</code>	Whether to enable the console.
<code>spring.h2.console.path</code>	<code>/h2-console</code>	Path at which the console is available.
<code>spring.h2.console.settings.trace</code>	<code>false</code>	Whether to enable trace output.
<code>spring.h2.console.settings.web-admin-password</code>		Password to access preferences and tools of H2 Console.
<code>spring.h2.console.settings.web-allow-others</code>	<code>false</code>	Whether to enable remote access.
<code>spring.influx.password</code>		Login password.
<code>spring.influx.url</code>		URL of the InfluxDB instance to which to connect.
<code>spring.influx.user</code>		Login user.
<code>spring.jdbc.template.fetch-size</code>	<code>-1</code>	Number of rows that should be fetched from the database when more rows are needed. Use <code>-1</code> to use the JDBC driver's default configuration.

Key	Default Value	Description
<code>spring.jdbc.template.max-rows</code>	-1	Maximum number of rows. Use -1 to use the JDBC driver's default configuration.
<code>spring.jdbc.template.query-timeout</code>		Query timeout. Default is to use the JDBC driver's default configuration. If a duration suffix is not specified, seconds will be used.
<code>spring.jooq.sql-dialect</code>		SQL dialect to use. Auto-detected by default.
<code>spring.jpa.database</code>		Target database to operate on, auto-detected by default. Can be alternatively set using the "databasePlatform" property.

Key	Default Value	Description
<code>spring.jpa.database-platform</code>		Name of the target database to operate on, auto-detected by default. Can be alternatively set using the "Database" enum.
<code>spring.jpa.generate-ddl</code>	<code>false</code>	Whether to initialize the schema on startup.
<code>spring.jpa.hibernate.ddl-auto</code>		DDL mode. This is actually a shortcut for the "hibernate.hbm2ddl.auto" property. Defaults to "create-drop" when using an embedded database and no schema manager was detected. Otherwise, defaults to "none".
<code>spring.jpa.hibernate.naming.implicit-strategy</code>		Fully qualified name of the implicit naming strategy.

Key	Default Value	Description
<code>spring.jpa.hibernate.naming.physical-strategy</code>		Fully qualified name of the physical naming strategy.
<code>spring.jpa.hibernate.use-new-id-generator-mappings</code>		Whether to use Hibernate's newer IdentifierGenerator for AUTO, TABLE and SEQUENCE. This is actually a shortcut for the "hibernate.id.new_generator_mappings" property. When not specified will default to "true".
<code>spring.jpa.mapping-resources</code>		Mapping resources (equivalent to "mapping-file" entries in persistence.xml).
<code>spring.jpa.open-in-view</code>	<code>true</code>	Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to the thread for the entire processing of the request.

Key	Default Value	Description
<code>spring.jpa.properties.*</code>		Additional native properties to set on the JPA provider.
<code>spring.jpa.show-sql</code>	<code>false</code>	Whether to enable logging of SQL statements.
<code>spring.ldap.anonymous-read-only</code>		Whether read-only operations should use an anonymous environment. Disabled by default unless a username is set.
<code>spring.ldap.base</code>		Base suffix from which all operations should originate.
<code>spring.ldap.base-environment.*</code>		LDAP specification settings.
<code>spring.ldap.embedded.base-dn</code>		List of base DNs.
<code>spring.ldap.embedded.credential.password</code>		Embedded LDAP password.
<code>spring.ldap.embedded.credential.username</code>		Embedded LDAP username.
<code>spring.ldap.embedded.ldif</code>	<code>classpath:schema.ldif</code>	Schema (LDIF) script resource reference.
<code>spring.ldap.embedded.port</code>	<code>0</code>	Embedded LDAP port.

Key	Default Value	Description
<code>spring.ldap.embedded.validation.enabled</code>	<code>true</code>	Whether to enable LDAP schema validation.
<code>spring.ldap.embedded.validation.schema</code>		Path to the custom schema.
<code>spring.ldap.password</code>		Login password of the server.
<code>spring.ldap.template.ignore-name-not-found-exception</code>	<code>false</code>	Whether NameNotFoundException should be ignored in searches via the LdapTemplate.
<code>spring.ldap.template.ignore-partial-result-exception</code>	<code>false</code>	Whether PartialResultException should be ignored in searches via the LdapTemplate.
<code>spring.ldap.template.ignore-size-limit-exceeded-exception</code>	<code>true</code>	Whether SizeLimitExceeded Exception should be ignored in searches via the LdapTemplate.
<code>spring.ldap.urls</code>		LDAP URLs of the server.
<code>spring.ldap.username</code>		Login username of the server.

Key	Default Value	Description
<code>spring.mongodb.embedded.features</code>	<code>[sync_delay]</code>	Comma-separated list of features to enable. Uses the defaults of the configured version by default.
<code>spring.mongodb.embedded.storage.database-dir</code>		Directory used for data storage.
<code>spring.mongodb.embedded.storage.oplog-size</code>		Maximum size of the oplog.
<code>spring.mongodb.embedded.storage.replica-set-name</code>		Name of the replica set.
<code>spring.mongodb.embedded.version</code>	<code>3.5.5</code>	Version of Mongo to use.
<code>spring.neo4j.authentication.kerberos-ticket</code>		Kerberos ticket for connecting to the database. Mutual exclusive with a given username.
<code>spring.neo4j.authentication.password</code>		Login password of the server.
<code>spring.neo4j.authentication.realm</code>		Realm to connect to.
<code>spring.neo4j.authentication.username</code>		Login user of the server.
<code>spring.neo4j.connection-timeout</code>	<code>30s</code>	Timeout for borrowing connections from the pool.

Key	Default Value	Description
<code>spring.neo4j.max-transaction-retry-time</code>	30s	Maximum time transactions are allowed to retry.
<code>spring.neo4j.pool.connection-acquisition-timeout</code>	60s	Acquisition of new connections will be attempted for at most configured timeout.
<code>spring.neo4j.pool.idle-time-before-connection-test</code>		Pooled connections that have been idle in the pool for longer than this threshold will be tested before they are used again.
<code>spring.neo4j.pool.log-leaked-sessions</code>	false	Whether to log leaked sessions.
<code>spring.neo4j.pool.max-connection-lifetime</code>	1h	Pooled connections older than this threshold will be closed and removed from the pool.
<code>spring.neo4j.pool.max-connection-pool-size</code>	100	Maximum amount of connections in the connection pool towards a single database.

Key	Default Value	Description
<code>spring.neo4j.pool.metrics-enabled</code>	<code>false</code>	Whether to enable metrics.
<code>spring.neo4j.security.cert-file</code>		Path to the file that holds the trusted certificates.
<code>spring.neo4j.security.encrypted</code>	<code>false</code>	Whether the driver should use encrypted traffic.
<code>spring.neo4j.security.hostname-verification-enabled</code>	<code>true</code>	Whether hostname verification is required.
<code>spring.neo4j.security.trust-strategy</code>	<code>trust-system-ca-signed-certificates</code>	Trust strategy to use.
<code>spring.neo4j.uri</code>	<code>bolt://localhost:7687</code>	URI used by the driver.
<code>spring.r2dbc.generate-unique-name</code>	<code>false</code>	Whether to generate a random database name. Ignore any configured name when enabled.
<code>spring.r2dbc.name</code>		Database name. Set if no name is specified in the url. Default to "testdb" when using an embedded database.

Key	Default Value	Description
<code>spring.r2dbc.password</code>		Login password of the database. Set if no password is specified in the url.
<code>spring.r2dbc.pool.enabled</code>		Whether pooling is enabled. Enabled automatically if "r2dbc-pool" is on the classpath.
<code>spring.r2dbc.pool.initial-size</code>	10	Initial connection pool size.
<code>spring.r2dbc.pool.max-acquire-time</code>		Maximum time to acquire a connection from the pool. By default, wait indefinitely.
<code>spring.r2dbc.pool.max-create-connection-time</code>		Maximum time to wait to create a new connection. By default, wait indefinitely.
<code>spring.r2dbc.pool.max-idle-time</code>	30m	Maximum amount of time that a connection is allowed to sit idle in the pool.

Key	Default Value	Description
<code>spring.r2dbc.pool.max-life-time</code>		Maximum lifetime of a connection in the pool. By default, connections have an infinite lifetime.
<code>spring.r2dbc.pool.max-size</code>	10	Maximal connection pool size.
<code>spring.r2dbc.pool.validation-depth</code>	local	Validation depth.
<code>spring.r2dbc.pool.validation-query</code>		Validation query.
<code>spring.r2dbc.properties.*</code>		Additional R2DBC options.
<code>spring.r2dbc.url</code>		R2DBC URL of the database, username, password and pooling options specified in the url take precedence over individual options.
<code>spring.r2dbc.username</code>		Login username of the database. Set if no username is specified in the url.

Key	Default Value	Description
<code>spring.redis.client-name</code>		Client name to be set on connections with CLIENT SETNAME.
<code>spring.redis.client-type</code>		Type of client to use. By default, auto-detected according to the classpath.
<code>spring.redis.cluster.max-redirects</code>		Maximum number of redirects to follow when executing commands across the cluster.
<code>spring.redis.cluster.nodes</code>		Comma-separated list of "host:port" pairs to bootstrap from. This represents an "initial" list of cluster nodes and is required to have at least one entry.
<code>spring.redis.connect-timeout</code>		Connection timeout.
<code>spring.redis.database</code>	0	Database index used by the connection factory.

Key	Default Value	Description
<code>spring.redis.host</code>	<code>localhost</code>	Redis server host.
<code>spring.redis.jedis.pool.max-active</code>	8	Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.
<code>spring.redis.jedis.pool.max-idle</code>	8	Maximum number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections.
<code>spring.redis.jedis.pool.max-wait</code>	-1ms	Maximum amount of time a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.

Key	Default Value	Description
<code>spring.redis.jedis.pool.min-idle</code>	0	Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if both <code>it</code> and <code>time-between-eviction-runs</code> are positive.
<code>spring.redis.jedis.pool.time-between-eviction-runs</code>		Time between runs of the idle object evictor thread. When positive, the idle object evictor thread starts, otherwise no idle object eviction is performed.
<code>spring.redis.lettuce.cluster.refresh.adaptive</code>	false	Whether adaptive topology refreshing using all available refresh triggers should be used.

Key	Default Value	Description
<code>spring.redis.lettuce.cluster.refresh.dynamic-refresh-sources</code>	<code>true</code>	Whether to discover and query all cluster nodes for obtaining the cluster topology. When set to <code>false</code> , only the initial seed nodes are used as sources for topology discovery.
<code>spring.redis.lettuce.cluster.refresh.period</code>		Cluster topology refresh period.
<code>spring.redis.lettuce.pool.max-active</code>	<code>8</code>	Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.
<code>spring.redis.lettuce.pool.max-idle</code>	<code>8</code>	Maximum number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections.

Key	Default Value	Description
<code>spring.redis.lettuce.pool.max-wait</code>	-1ms	Maximum amount of time a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.
<code>spring.redis.lettuce.pool.min-idle</code>	0	Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if both it and time between eviction runs are positive.
<code>spring.redis.lettuce.pool.time-between-eviction-runs</code>		Time between runs of the idle object evictor thread. When positive, the idle object evictor thread starts, otherwise no idle object eviction is performed.

Key	Default Value	Description
<code>spring.redis.lettuce.shutdown-timeout</code>	<code>100ms</code>	Shutdown timeout.
<code>spring.redis.password</code>		Login password of the redis server.
<code>spring.redis.port</code>	<code>6379</code>	Redis server port.
<code>spring.redis.sentinel.master</code>		Name of the Redis server.
<code>spring.redis.sentinel.nodes</code>		Comma-separated list of "host:port" pairs.
<code>spring.redis.sentinel.password</code>		Password for authenticating with sentinel(s).
<code>spring.redis.ssl</code>	<code>false</code>	Whether to enable SSL support.
<code>spring.redis.timeout</code>		Read timeout.
<code>spring.redis.url</code>		Connection URL. Overrides host, port, and password. User is ignored. Example: <code>redis://user:password@example.com:6379</code>
<code>spring.redis.username</code>		Login username of the redis server.

11.A.6. Transaction Properties

Key	Default Value	Description
<code>spring.jta.atomikos.connectionfactory.borrow-connection-timeout</code>	30	Timeout, in seconds, for borrowing connections from the pool.
<code>spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag</code>	true	Whether to ignore the transacted flag when creating session.
<code>spring.jta.atomikos.connectionfactory.local-transaction-mode</code>	false	Whether local transactions are desired.
<code>spring.jta.atomikos.connectionfactory.maintenance-interval</code>	60	Time, in seconds, between runs of the pool's maintenance thread.
<code>spring.jta.atomikos.connectionfactory.max-idle-time</code>	60	Time, in seconds, after which connections are cleaned up from the pool.
<code>spring.jta.atomikos.connectionfactory.max-lifetime</code>	0	Time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.
<code>spring.jta.atomikos.connectionfactory.max-pool-size</code>	1	Maximum size of the pool.

Key	Default Value	Description
<code>spring.jta.atomikos.connectionfactory.min-pool-size</code>	1	Minimum size of the pool.
<code>spring.jta.atomikos.connectionfactory.reap-timeout</code>	0	Reap timeout, in seconds, for borrowed connections. 0 denotes no limit.
<code>spring.jta.atomikos.connectionfactory.unique-resource-name</code>	jmsConnectionFactory	Unique name used to identify the resource during recovery.
<code>spring.jta.atomikos.connectionfactory.xa-connection-factory-class-name</code>		Vendor-specific implementation of XAConnectionFactory.
<code>spring.jta.atomikos.connectionfactory.xa-properties</code>		Vendor-specific XA properties.
<code>spring.jta.atomikos.datasource.borrow-connection-timeout</code>	30	Timeout, in seconds, for borrowing connections from the pool.
<code>spring.jta.atomikos.datasource.concurrent-validation</code>	true	Whether to use concurrent connection validation.
<code>spring.jta.atomikos.datasource.default-isolation-level</code>		Default isolation level of connections provided by the pool.

Key	Default Value	Description
<code>spring.jta.atomikos.datasource.log-in-timeout</code>	0	Timeout, in seconds, for establishing a database connection.
<code>spring.jta.atomikos.datasource.main-maintenance-interval</code>	60	Time, in seconds, between runs of the pool's maintenance thread.
<code>spring.jta.atomikos.datasource.max-idle-time</code>	60	Time, in seconds, after which connections are cleaned up from the pool.
<code>spring.jta.atomikos.datasource.max-lifetime</code>	0	Time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.
<code>spring.jta.atomikos.datasource.max-pool-size</code>	1	Maximum size of the pool.
<code>spring.jta.atomikos.datasource.min-pool-size</code>	1	Minimum size of the pool.
<code>spring.jta.atomikos.datasource.reap-timeout</code>	0	Reap timeout, in seconds, for borrowed connections. 0 denotes no limit.

Key	Default Value	Description
<code>spring.jta.atomikos.datasource.test-query</code>		SQL query or statement used to validate a connection before returning it.
<code>spring.jta.atomikos.datasource.unique-resource-name</code>	<code>dataSource</code>	Unique name used to identify the resource during recovery.
<code>spring.jta.atomikos.datasource.xa-data-source-class-name</code>		Vendor-specific implementation of XAConnectionFactory.
<code>spring.jta.atomikos.datasource.xa-properties</code>		Vendor-specific XA properties.
<code>spring.jta.atomikos.properties.allow-sub-transactions</code>	<code>true</code>	Specify whether sub-transactions are allowed.
<code>spring.jta.atomikos.properties.checkpoint-interval</code>	<code>500</code>	Interval between checkpoints, expressed as the number of log writes between two checkpoints. A checkpoint reduces the log file size at the expense of adding some overhead in the runtime.

Key	Default Value	Description
<code>spring.jta.atomikos.properties.default-jta-timeout</code>	<code>10000ms</code>	Default timeout for JTA transactions.
<code>spring.jta.atomikos.properties.default-max-wait-time-on-shutdown</code>		How long should normal shutdown (no-force) wait for transactions to complete.
<code>spring.jta.atomikos.properties.enable-logging</code>	<code>true</code>	Whether to enable disk logging.
<code>spring.jta.atomikos.properties.force-shutdown-on-vm-exit</code>	<code>false</code>	Whether a VM shutdown should trigger forced shutdown of the transaction core.
<code>spring.jta.atomikos.properties.log-base-dir</code>		Directory in which the log files should be stored. Defaults to the current working directory.
<code>spring.jta.atomikos.properties.log-base-name</code>	<code>tmlog</code>	Transactions log file base name.
<code>spring.jta.atomikos.properties.max-active</code>	<code>50</code>	Maximum number of active transactions.
<code>spring.jta.atomikos.properties.max-timeout</code>	<code>300000ms</code>	Maximum timeout that can be allowed for transactions.

Key	Default Value	Description
<code>spring.jta.atomikos.properties.recoverable.delay</code>	<code>10000ms</code>	Delay between two recovery scans.
<code>spring.jta.atomikos.properties.recoverable.forget-orphaned-log-entries-delay</code>	<code>86400000ms</code>	Delay after which recovery can cleanup pending ('orphaned') log entries.
<code>spring.jta.atomikos.properties.recoverable.max-retries</code>	<code>5</code>	Number of retry attempts to commit the transaction before throwing an exception.
<code>spring.jta.atomikos.properties.recoverable.retry-interval</code>	<code>10000ms</code>	Delay between retry attempts.
<code>spring.jta.atomikos.properties.serial-jta-transactions</code>	<code>true</code>	Whether sub-transactions should be joined when possible.
<code>spring.jta.atomikos.properties.service</code>		Transaction manager implementation that should be started.
<code>spring.jta.atomikos.properties.threaded-two-phase-commit</code>	<code>false</code>	Whether to use different (and concurrent) threads for two-phase commit on the participating resources.

Key	Default Value	Description
<code>spring.jta.atomikos.properties.transaction-manager-unique-name</code>		The transaction manager's unique name. Defaults to the machine's IP address. If you plan to run more than one transaction manager against one database you must set this property to a unique value.
<code>spring.jta.bitronix.connectionfactory.acquire-increment</code>	1	Number of connections to create when growing the pool.
<code>spring.jta.bitronix.connectionfactory.acquisition-interval</code>	1	Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.
<code>spring.jta.bitronix.connectionfactory.acquisition-timeout</code>	30	Timeout, in seconds, for acquiring connections from the pool.

Key	Default Value	Description
<code>spring.jta.bitronix.connectionfactory.allow-local-transactions</code>	false	Whether the transaction manager should allow mixing XA and non-XA transactions.
<code>spring.jta.bitronix.connectionfactory.apply-transaction-timeout</code>	false	Whether the transaction timeout should be set on the XAResource when it is enlisted.
<code>spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled</code>	true	Whether resources should be enlisted and delisted automatically.
<code>spring.jta.bitronix.connectionfactory.cache-producers-consumers</code>	true	Whether producers and consumers should be cached.
<code>spring.jta.bitronix.connectionfactory.class-name</code>		Underlying implementation class name of the XA resource.
<code>spring.jta.bitronix.connectionfactory.defer-connection-release</code>	true	Whether the provider can run many transactions on the same connection and supports transaction interleaving.

Key	Default Value	Description
<code>spring.jta.bitronix.connectionfactory.disabled</code>	<code>false</code>	Whether this resource is disabled, meaning it's temporarily forbidden to acquire a connection from its pool.
<code>spring.jta.bitronix.connectionfactory.driver-properties</code>		Properties that should be set on the underlying implementation.
<code>spring.jta.bitronix.connectionfactory.failed</code>		
<code>spring.jta.bitronix.connectionfactory.ignore-recovery-failures</code>	<code>false</code>	Whether recovery failures should be ignored.
<code>spring.jta.bitronix.connectionfactory.max-idle-time</code>	<code>60</code>	Time, in seconds, after which connections are cleaned up from the pool.
<code>spring.jta.bitronix.connectionfactory.max-pool-size</code>	<code>0</code>	Maximum size of the pool. <code>0</code> denotes no limit.
<code>spring.jta.bitronix.connectionfactory.min-pool-size</code>	<code>0</code>	Minimum size of the pool.
<code>spring.jta.bitronix.connectionfactory.password</code>		Password to use to connect to the JMS provider.

Key	Default Value	Description
<code>spring.jta.bitronix.connectionfactory.share-transaction-connections</code>	<code>false</code>	Whether connections in the ACCESSIBLE state can be shared within the context of a transaction.
<code>spring.jta.bitronix.connectionfactory.test-connections</code>	<code>false</code>	Whether connections should be tested when acquired from the pool.
<code>spring.jta.bitronix.connectionfactory.two-pc-ordering-position</code>	<code>1</code>	Position that this resource should take during two-phase commit (always first is <code>Integer.MIN_VALUE</code> , always last is <code>Integer.MAX_VALUE</code>).
<code>spring.jta.bitronix.connectionfactory.unique-name</code>	<code>jmsConnectionFactory</code>	Unique name used to identify the resource during recovery.
<code>spring.jta.bitronix.connectionfactory.use-tm-join</code>	<code>true</code>	Whether TMJOIN should be used when starting XAResources.

Key	Default Value	Description
<code>spring.jta.bitronix.connectionfactory.user</code>		User to use to connect to the JMS provider.
<code>spring.jta.bitronix.datasource.acquire-increment</code>	1	Number of connections to create when growing the pool.
<code>spring.jta.bitronix.datasource.acquisition-interval</code>	1	Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.
<code>spring.jta.bitronix.datasource.acquisition-timeout</code>	30	Timeout, in seconds, for acquiring connections from the pool.
<code>spring.jta.bitronix.datasource.allow-local-transactions</code>	false	Whether the transaction manager should allow mixing XA and non-XA transactions.
<code>spring.jta.bitronix.datasource.appl.transaction-timeout</code>	false	Whether the transaction timeout should be set on the XAResource when it is enlisted.

Key	Default Value	Description
<code>spring.jta.bitronix.datasource.automatic-enlisting-enabled</code>	true	Whether resources should be enlisted and delisted automatically.
<code>spring.jta.bitronix.datasource.class-name</code>		Underlying implementation class name of the XA resource.
<code>spring.jta.bitronix.datasource.cursor-holdability</code>		Default cursor holdability for connections.
<code>spring.jta.bitronix.datasource.deferred-connection-release</code>	true	Whether the database can run many transactions on the same connection and supports transaction interleaving.
<code>spring.jta.bitronix.datasource.disabled</code>	false	Whether this resource is disabled, meaning it's temporarily forbidden to acquire a connection from its pool.
<code>spring.jta.bitronix.datasource.driver-properties</code>		Properties that should be set on the underlying implementation.

Key	Default Value	Description
<code>spring.jta.bitronix.datasource.enabled-jdbc4-connection-test</code>	<code>false</code>	Whether <code>Connection.isValid()</code> is called when acquiring a connection from the pool.
<code>spring.jta.bitronix.datasource.failed</code>		
<code>spring.jta.bitronix.datasource.ignore-recovery-failures</code>	<code>false</code>	Whether recovery failures should be ignored.
<code>spring.jta.bitronix.datasource.isolation-level</code>		Default isolation level for connections.
<code>spring.jta.bitronix.datasource.local-auto-commit</code>		Default auto-commit mode for local transactions.
<code>spring.jta.bitronix.datasource.login-timeout</code>		Timeout, in seconds, for establishing a database connection.
<code>spring.jta.bitronix.datasource.max-idle-time</code>	<code>60</code>	Time, in seconds, after which connections are cleaned up from the pool.
<code>spring.jta.bitronix.datasource.max-pool-size</code>	<code>0</code>	Maximum size of the pool. <code>0</code> denotes no limit.

Key	Default Value	Description
<code>spring.jta.bitronix.datasource.min-pool-size</code>	0	Minimum size of the pool.
<code>spring.jta.bitronix.datasource.prepared-statement-cache-size</code>	0	Target size of the prepared statement cache. 0 disables the cache.
<code>spring.jta.bitronix.datasource.share-transaction-connections</code>	false	Whether connections in the ACCESSIBLE state can be shared within the context of a transaction.
<code>spring.jta.bitronix.datasource.test-query</code>		SQL query or statement used to validate a connection before returning it.
<code>spring.jta.bitronix.datasource.two-phase-ordering-position</code>	1	Position that this resource should take during two-phase commit (always first is Integer.MIN_VALUE, and always last is Integer.MAX_VALUE).

Key	Default Value	Description
<code>spring.jta.bitronix.datasource.unique-name</code>	<code>dataSource</code>	Unique name used to identify the resource during recovery.
<code>spring.jta.bitronix.datasource.use-tmjoin</code>	<code>true</code>	Whether TMJOIN should be used when starting XAResources.
<code>spring.jta.enabled</code>	<code>true</code>	Whether to enable JTA support.
<code>spring.jta.log-dir</code>		Transaction logs directory.
<code>spring.jta.transaction-manager-id</code>		Transaction manager unique identifier.
<code>spring.transaction.default-timeout</code>		Default transaction timeout. If a duration suffix is not specified, seconds will be used.
<code>spring.transaction.rollback-on-commit-failure</code>		Whether to roll back on commit failures.

11.A.7. Data migration Properties

Key	Default Value	Description
<code>spring.flyway.baseline-description</code>	<code><< Flyway Baseline >></code>	Description to tag an existing schema with when applying a baseline.
<code>spring.flyway.baseline-on-migrate</code>	<code>false</code>	Whether to automatically call baseline when migrating a non-empty schema.
<code>spring.flyway.baseline-version</code>	<code>1</code>	Version to tag an existing schema with when executing baseline.
<code>spring.flyway.batch</code>		Whether to batch SQL statements when executing them. Requires Flyway Teams.
<code>spring.flyway.check-location</code>	<code>true</code>	Whether to check that migration scripts location exists.

Key	Default Value	Description
<code>spring.flyway.cherry-pick</code>		Migrations that Flyway should consider when migrating or undoing. When empty all available migrations are considered. Requires Flyway Teams.
<code>spring.flyway.clean-disabled</code>	<code>false</code>	Whether to disable cleaning of the database.
<code>spring.flyway.clean-on-validation-error</code>	<code>false</code>	Whether to automatically call clean when a validation error occurs.
<code>spring.flyway.connect-retries</code>	<code>0</code>	Maximum number of retries when attempting to connect to the database.
<code>spring.flyway.create-schemas</code>	<code>true</code>	Whether Flyway should attempt to create the schemas specified in the schemas property.

Key	Default Value	Description
<code>spring.flyway.default-schema</code>		Default schema name managed by Flyway (case-sensitive).
<code>spring.flyway.enabled</code>	<code>true</code>	Whether to enable flyway.
<code>spring.flyway.encoding</code>	<code>UTF-8</code>	Encoding of SQL migrations.
<code>spring.flyway.error-overrides</code>		Rules for the built-in error handling to override specific SQL states and error codes. Requires Flyway Teams.
<code>spring.flyway.group</code>	<code>false</code>	Whether to group all pending migrations together in the same transaction when applying them.
<code>spring.flyway.ignore-future-migrations</code>	<code>true</code>	Whether to ignore future migrations when reading the schema history table.

Key	Default Value	Description
<code>spring.flyway.ignore-ignored-migrations</code>	<code>false</code>	Whether to ignore ignored migrations when reading the schema history table.
<code>spring.flyway.ignore-missing-migrations</code>	<code>false</code>	Whether to ignore missing migrations when reading the schema history table.
<code>spring.flyway.ignore-pending-migrations</code>	<code>false</code>	Whether to ignore pending migrations when reading the schema history table.
<code>spring.flyway.init-sqls</code>		SQL statements to execute to initialize a connection immediately after obtaining it.
<code>spring.flyway.installed-by</code>		Username recorded in the schema history table as having applied the migration.

Key	Default Value	Description
<code>spring.flyway.jdbc-properties.*</code>		Properties to pass to the JDBC driver. Requires Flyway Teams.
<code>spring.flyway.license-key</code>		Licence key for Flyway Teams.
<code>spring.flyway.locations</code>	[classpath:db/migration]	Locations of migrations scripts. Can contain the special "{vendor}" placeholder to use vendor-specific locations.
<code>spring.flyway.lock-retry-count</code>	50	Maximum number of retries when trying to obtain a lock.
<code>spring.flyway.mixed</code>	false	Whether to allow mixing transactional and non-transactional statements within the same migration.
<code>spring.flyway.oracle-kerberos-cache-file</code>		Path of the Oracle Kerberos cache file. Requires Flyway Teams.

Key	Default Value	Description
<code>spring.flyway.oracle-kerberos-config-file</code>		Path of the Oracle Kerberos config file. Requires Flyway Teams.
<code>spring.flyway.oracle-sqlplus</code>		Whether to enable support for Oracle SQL*Plus commands. Requires Flyway Teams.
<code>spring.flyway.oracle-sqlplus-warn</code>		Whether to issue a warning rather than an error when a not-yet-supported Oracle SQL*Plus statement is encountered. Requires Flyway Teams.
<code>spring.flyway.out-of-order</code>	<code>false</code>	Whether to allow migrations to be run out of order.
<code>spring.flyway.output-query-results</code>		Whether Flyway should output a table with the results of queries when executing migrations. Requires Flyway Teams.

Key	Default Value	Description
<code>spring.flyway.password</code>		Login password of the database to migrate.
<code>spring.flyway.placeholder-prefix</code>	<code> \${</code>	Prefix of placeholders in migration scripts.
<code>spring.flyway.placeholder-replacement</code>	<code>true</code>	Perform placeholder replacement in migration scripts.
<code>spring.flyway.placeholder-suffix</code>	<code>}</code>	Suffix of placeholders in migration scripts.
<code>spring.flyway.placeholders.*</code>		Placeholders and their replacements to apply to sql migration scripts.
<code>spring.flyway.repeatable-sql-migration-prefix</code>	<code>R</code>	File name prefix for repeatable SQL migrations.
<code>spring.flyway.schemas</code>		Schema names managed by Flyway (case-sensitive).

Key	Default Value	Description
<code>spring.flyway.skip-default-callbacks</code>	<code>false</code>	Whether to skip default callbacks. If true, only custom callbacks are used.
<code>spring.flyway.skip-default-resolvers</code>	<code>false</code>	Whether to skip default resolvers. If true, only custom resolvers are used.
<code>spring.flyway.skip-executing-migrations</code>		Whether Flyway should skip executing the contents of the migrations and only update the schema history table. Requires Flyway teams.
<code>spring.flyway.sql-migration-prefix</code>	<code>V</code>	File name prefix for SQL migrations.
<code>spring.flyway.sql-migration-separator</code>	<code>--</code>	File name separator for SQL migrations.
<code>spring.flyway.sql-migration-suffixes</code>	<code>[.sql]</code>	File name suffix for SQL migrations.

Key	Default Value	Description
<code>spring.flyway.stream</code>		Whether to stream SQL migrations when executing them. Requires Flyway Teams.
<code>spring.flyway.table</code>	<code>flyway_schema_history</code>	Name of the schema history table that will be used by Flyway.
<code>spring.flyway.tablespace</code>		Tablespace in which the schema history table is created. Ignored when using a database that does not support tablespaces. Defaults to the default tablespace of the connection used by Flyway.
<code>spring.flyway.target</code>		Target version up to which migrations should be considered.
<code>spring.flyway.url</code>		JDBC url of the database to migrate. If not set, the primary configured data source is used.

Key	Default Value	Description
<code>spring.flyway.user</code>		Login user of the database to migrate.
<code>spring.flyway.validate-migration-naming</code>	<code>false</code>	Whether to validate migrations and callbacks whose scripts do not obey the correct naming convention.
<code>spring.flyway.validate-on-migrate</code>	<code>true</code>	Whether to automatically call validate when performing a migration.
<code>spring.liquibase.change-log</code>	<code>classpath:/db/changelog/db.changelog-master.yaml</code>	Change log configuration path.
<code>spring.liquibase.clear-checksums</code>	<code>false</code>	Whether to clear all checksums in the current changelog, so they will be recalculated upon the next update.
<code>spring.liquibase.contexts</code>		Comma-separated list of runtime contexts to use.

Key	Default Value	Description
<code>spring.liquibase.database-change-log-lock-table</code>	<code>DATABASECHANGELOGLOCK</code>	Name of table to use for tracking concurrent Liquibase usage.
<code>spring.liquibase.database-change-log-table</code>	<code>DATABASECHANGELOG</code>	Name of table to use for tracking change history.
<code>spring.liquibase.default-schema</code>		Default database schema.
<code>spring.liquibase.driver-class-name</code>		Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.
<code>spring.liquibase.drop-first</code>	<code>false</code>	Whether to first drop the database schema.
<code>spring.liquibase.enabled</code>	<code>true</code>	Whether to enable Liquibase support.
<code>spring.liquibase.labels</code>		Comma-separated list of runtime labels to use.
<code>spring.liquibase.liquibase-schema</code>		Schema to use for Liquibase objects.
<code>spring.liquibase.liquibase-tablespace</code>		Tablespace to use for Liquibase objects.

Key	Default Value	Description
<code>spring.liquibase.parameters.*</code>		Change log parameters.
<code>spring.liquibase.password</code>		Login password of the database to migrate.
<code>spring.liquibase.rollback-file</code>		File to which rollback SQL is written when an update is performed.
<code>spring.liquibase.tag</code>		Tag name to use when applying database changes. Can also be used with "rollbackFile" to generate a rollback script for all existing changes associated with that tag.
<code>spring.liquibase.test-rollback-on-update</code>	<code>false</code>	Whether rollback should be tested before update is performed.
<code>spring.liquibase.url</code>		JDBC URL of the database to migrate. If not set, the primary configured data source is used.

Key	Default Value	Description
<code>spring.liquibase.user</code>		Login user of the database to migrate.

11.A.8. Integration Properties

Key	Default Value	Description
<code>spring.activemq.broker-url</code>		URL of the ActiveMQ broker. Auto-generated by default.
<code>spring.activemq.close-timeout</code>	<code>15s</code>	Time to wait before considering a close complete.
<code>spring.activemq.in-memory</code>	<code>true</code>	Whether the default broker URL should be in memory. Ignored if an explicit broker has been specified.

Key	Default Value	Description
<code>spring.activemq.non-blocking-redelivery</code>	<code>false</code>	Whether to stop message delivery before re-delivering messages from a rolled back transaction. This implies that message order is not preserved when this is enabled.
<code>spring.activemq.packages.trust-all</code>		Whether to trust all packages.
<code>spring.activemq.packages.trusted</code>		Comma-separated list of specific packages to trust (when not trusting all packages).
<code>spring.activemq.password</code>		Login password of the broker.
<code>spring.activemq.pool.block-if-full</code>	<code>true</code>	Whether to block when a connection is requested and the pool is full. Set it to false to throw a "JMSEException" instead.

Key	Default Value	Description
<code>spring.activemq.pool.block-if-full-timeout</code>	-1ms	Blocking period before throwing an exception if the pool is still full.
<code>spring.activemq.pool.enabled</code>	false	Whether a JmsPoolConnection Factory should be created, instead of a regular ConnectionFactory .
<code>spring.activemq.pool.idle-timeout</code>	30s	Connection idle timeout.
<code>spring.activemq.pool.max-connections</code>	1	Maximum number of pooled connections.
<code>spring.activemq.pool.max-sessions-per-connection</code>	500	Maximum number of pooled sessions per connection in the pool.
<code>spring.activemq.pool.time-between-expiration-check</code>	-1ms	Time to sleep between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.

Key	Default Value	Description
<code>spring.activemq.pool.use-anonymous-producers</code>	<code>true</code>	Whether to use only one anonymous "MessageProducer" instance. Set it to false to create one "MessageProducer" every time one is required.
<code>spring.activemq.send-timeout</code>	<code>0ms</code>	Time to wait on message sends for a response. Set it to 0 to wait forever.
<code>spring.activemq.user</code>		Login user of the broker.
<code>spring.artemis.embedded.cluster-password</code>		Cluster password. Randomly generated on startup by default.
<code>spring.artemis.embedded.data-directory</code>		Journal file directory. Not necessary if persistence is turned off.
<code>spring.artemis.embedded.enabled</code>	<code>true</code>	Whether to enable embedded mode if the Artemis server APIs are available.

Key	Default Value	Description
<code>spring.artemis.embedded.persistent</code>	<code>false</code>	Whether to enable persistent store.
<code>spring.artemis.embedded.queues</code>	<code>[]</code>	Comma-separated list of queues to create on startup.
<code>spring.artemis.embedded.server-id</code>	<code>0</code>	Server ID. By default, an auto-incremented counter is used.
<code>spring.artemis.embedded.topics</code>	<code>[]</code>	Comma-separated list of topics to create on startup.
<code>spring.artemis.host</code>	<code>localhost</code>	Artemis broker host.
<code>spring.artemis.mode</code>		Artemis deployment mode, auto-detected by default.
<code>spring.artemis.password</code>		Login password of the broker.
<code>spring.artemis.pool.block-if-full</code>	<code>true</code>	Whether to block when a connection is requested and the pool is full. Set it to false to throw a "JMSEException" instead.

Key	Default Value	Description
<code>spring.artemis.pool.block-if-full-timeout</code>	<code>-1ms</code>	Blocking period before throwing an exception if the pool is still full.
<code>spring.artemis.pool.enabled</code>	<code>false</code>	Whether a JmsPoolConnection Factory should be created, instead of a regular ConnectionFactory .
<code>spring.artemis.pool.idle-timeout</code>	<code>30s</code>	Connection idle timeout.
<code>spring.artemis.pool.max-connections</code>	<code>1</code>	Maximum number of pooled connections.
<code>spring.artemis.pool.max-sessions-per-connection</code>	<code>500</code>	Maximum number of pooled sessions per connection in the pool.
<code>spring.artemis.pool.time-between-expiration-check</code>	<code>-1ms</code>	Time to sleep between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.

Key	Default Value	Description
<code>spring.artemis.pool.use-anonymous-producers</code>	<code>true</code>	Whether to use only one anonymous "MessageProducer" instance. Set it to false to create one "MessageProducer" every time one is required.
<code>spring.artemis.port</code>	<code>61616</code>	Artemis broker port.
<code>spring.artemis.user</code>		Login user of the broker.
<code>spring.batch.initialize-schema</code>	<code>embedded</code>	Database schema initialization mode.
<code>spring.batch.job.enabled</code>	<code>true</code>	Execute all Spring Batch jobs in the context on startup.
<code>spring.batch.job.names</code>		Comma-separated list of job names to execute on startup (for instance, `job1,job2`). By default, all Jobs found in the context are executed.

Key	Default Value	Description
spring.batch.schema	classpath:org/springframework/batch/core/schema-@@platform@@.sql	Path to the SQL file to use to initialize the database schema.
spring.batch.table-prefix		Table prefix for all the batch meta-data tables.
spring.hazelcast.config		The location of the configuration file to use to initialize Hazelcast.
spring.integration.jdbc.initialize-schema	embedded	Database schema initialization mode.
spring.integration.jdbc.schema	classpath:org/springframework/integration/jdbc/schema-@@platform@@.sql	Path to the SQL file to use to initialize the database schema.
spring.integration.rsocket.client.host		TCP RSocket server host to connect to.
spring.integration.rsocket.client.port		TCP RSocket server port to connect to.
spring.integration.rsocket.client.uri		WebSocket RSocket server uri to connect to.

Key	Default Value	Description
<code>spring.integration.rsocket.server.message-mapping-enabled</code>	<code>false</code>	Whether to handle message mapping for RSocket via Spring Integration.
<code>spring.jms.cache.consumers</code>	<code>false</code>	Whether to cache message consumers.
<code>spring.jms.cache.enabled</code>	<code>true</code>	Whether to cache sessions.
<code>spring.jms.cache.producers</code>	<code>true</code>	Whether to cache message producers.
<code>spring.jms.cache.session-cache-size</code>	<code>1</code>	Size of the session cache (per JMS Session type).
<code>spring.jms.jndi-name</code>		Connection factory JNDI name. When set, takes precedence to others connection factory auto-configurations.
<code>spring.jms.listener.acknowledge-mode</code>		Acknowledge mode of the container. By default, the listener is transacted with automatic acknowledgment.

Key	Default Value	Description
<code>spring.jms.listener.auto-startup</code>	<code>true</code>	Start the container automatically on startup.
<code>spring.jms.listener.concurrency</code>		Minimum number of concurrent consumers.
<code>spring.jms.listener.max-concurrency</code>		Maximum number of concurrent consumers.
<code>spring.jms.listener.receive-timeout</code>	<code>1s</code>	Timeout to use for receive calls. Use <code>-1</code> for a no-wait receive or <code>0</code> for no timeout at all. The latter is only feasible if not running within a transaction manager and is generally discouraged since it prevents clean shutdown.
<code>spring.jms.pub-sub-domain</code>	<code>false</code>	Whether the default destination type is topic.

Key	Default Value	Description
<code>spring.jms.template.default-destination</code>		Default destination to use on send and receive operations that do not have a destination parameter.
<code>spring.jms.template.delivery-delay</code>		Delivery delay to use for send calls.
<code>spring.jms.template.delivery-mode</code>		Delivery mode. Enables QoS (Quality of Service) when set.
<code>spring.jms.template.priority</code>		Priority of a message when sending. Enables QoS (Quality of Service) when set.

Key	Default Value	Description
<code>spring.jms.template.qos-enabled</code>		Whether to enable explicit QoS (Quality of Service) when sending a message. When enabled, the delivery mode, priority and time-to-live properties will be used when sending a message. QoS is automatically enabled when at least one of those settings is customized.
<code>spring.jms.template.receive-timeout</code>		Timeout to use for receive calls.
<code>spring.jms.template.time-to-live</code>		Time-to-live of a message when sending. Enables QoS (Quality of Service) when set.
<code>spring.kafka.admin.client-id</code>		ID to pass to the server when making requests. Used for server-side logging.

Key	Default Value	Description
<code>spring.kafka.admin.fail-fast</code>	<code>false</code>	Whether to fail fast if the broker is not available on startup.
<code>spring.kafka.admin.properties.*</code>		Additional admin-specific properties used to configure the client.
<code>spring.kafka.admin.security.protocol</code>		Security protocol used to communicate with brokers.
<code>spring.kafka.admin.ssl.key-password</code>		Password of the private key in the key store file.
<code>spring.kafka.admin.ssl.key-store-location</code>		Location of the key store file.
<code>spring.kafka.admin.ssl.key-store-password</code>		Store password for the key store file.
<code>spring.kafka.admin.ssl.key-store-type</code>		Type of the key store.
<code>spring.kafka.admin.ssl.protocol</code>		SSL protocol to use.
<code>spring.kafka.admin.ssl.trust-store-location</code>		Location of the trust store file.

Key	Default Value	Description
<code>spring.kafka.admin.ssl.trust-store-password</code>		Store password for the trust store file.
<code>spring.kafka.admin.ssl.trust-store-type</code>		Type of the trust store.
<code>spring.kafka.bootstrap-servers</code>		Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Applies to all components unless overridden.
<code>spring.kafka.client-id</code>		ID to pass to the server when making requests. Used for server-side logging.
<code>spring.kafka.consumer.auto-commit-interval</code>		Frequency with which the consumer offsets are auto-committed to Kafka if 'enable.auto.commit' is set to true.

Key	Default Value	Description
<code>spring.kafka.consumer.auto-offset-reset</code>		What to do when there is no initial offset in Kafka or if the current offset no longer exists on the server.
<code>spring.kafka.consumer.bootstrap-servers</code>		Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for consumers.
<code>spring.kafka.consumer.client-id</code>		ID to pass to the server when making requests. Used for server-side logging.
<code>spring.kafka.consumer.enable-auto-commit</code>		Whether the consumer's offset is periodically committed in the background.

Key	Default Value	Description
<code>spring.kafka.consumer.fetch-max-wait</code>		Maximum amount of time the server blocks before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by "fetch-min-size".
<code>spring.kafka.consumer.fetch-min-size</code>		Minimum amount of data the server should return for a fetch request.
<code>spring.kafka.consumer.group-id</code>		Unique string that identifies the consumer group to which this consumer belongs.
<code>spring.kafka.consumer.heartbeat-interval</code>		Expected time between heartbeats to the consumer coordinator.
<code>spring.kafka.consumer.isolation-level</code>	<code>read-uncommitted</code>	Isolation level for reading messages that have been written transactionally.

Key	Default Value	Description
spring.kafka.consumer.key-deserializer		Deserializer class for keys.
spring.kafka.consumer.max-poll-records		Maximum number of records returned in a single call to poll().
spring.kafka.consumer.properties.*		Additional consumer-specific properties used to configure the client.
spring.kafka.consumer.security.protocol		Security protocol used to communicate with brokers.
spring.kafka.consumer.ssl.key-password		Password of the private key in the key store file.
spring.kafka.consumer.ssl.key-store-location		Location of the key store file.
spring.kafka.consumer.ssl.key-store-password		Store password for the key store file.
spring.kafka.consumer.ssl.key-store-type		Type of the key store.
spring.kafka.consumer.ssl.protocol		SSL protocol to use.
spring.kafka.consumer.ssl.trust-store-location		Location of the trust store file.

Key	Default Value	Description
<code>spring.kafka.consumer.ssl.trust-store-password</code>		Store password for the trust store file.
<code>spring.kafka.consumer.ssl.trust-store-type</code>		Type of the trust store.
<code>spring.kafka.consumer.value-deserializer</code>		Deserializer class for values.
<code>spring.kafka.jaas.control-flag</code>	<code>required</code>	Control flag for login configuration.
<code>spring.kafka.jaas.enabled</code>	<code>false</code>	Whether to enable JAAS configuration.
<code>spring.kafka.jaas.login-module</code>	<code>com.sun.security.auth.module.Krb5LoginModule</code>	Login module.
<code>spring.kafka.jaas.options.*</code>		Additional JAAS options.
<code>spring.kafka.listener.ack-count</code>		Number of records between offset commits when ackMode is "COUNT" or "COUNT_TIME".
<code>spring.kafka.listener.ack-mode</code>		Listener AckMode. See the spring-kafka documentation.

Key	Default Value	Description
<code>spring.kafka.listener.ack-time</code>		Time between offset commits when ackMode is "TIME" or "COUNT_TIME".
<code>spring.kafka.listener.client-id</code>		Prefix for the listener's consumer client.id property.
<code>spring.kafka.listener.concurrency</code>		Number of threads to run in the listener containers.
<code>spring.kafka.listener.idle-between-polls</code>	0	Sleep interval between Consumer.poll(Duration) calls.
<code>spring.kafka.listener.idle-event-interval</code>		Time between publishing idle consumer events (no data received).
<code>spring.kafka.listener.log-container-config</code>		Whether to log the container configuration during initialization (INFO level).

Key	Default Value	Description
<code>spring.kafka.listener.missing-topics-fatal</code>	<code>false</code>	Whether the container should fail to start if at least one of the configured topics are not present on the broker.
<code>spring.kafka.listener.monitor-interval</code>		Time between checks for non-responsive consumers. If a duration suffix is not specified, seconds will be used.
<code>spring.kafka.listener.no-poll-threshold</code>		Multiplier applied to "pollTimeout" to determine if a consumer is non-responsive.
<code>spring.kafka.listener.poll-timeout</code>		Timeout to use when polling the consumer.
<code>spring.kafka.listener.type</code>	<code>single</code>	Listener type.

Key	Default Value	Description
<code>spring.kafka.producer.acks</code>		Number of acknowledgments the producer requires the leader to have received before considering a request complete.
<code>spring.kafka.producer.batch-size</code>		Default batch size. A small batch size will make batching less common and may reduce throughput (a batch size of zero disables batching entirely).
<code>spring.kafka.producer.bootstrap-servers</code>		Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for producers.

Key	Default Value	Description
<code>spring.kafka.producer.buffer-memory</code>		Total memory size the producer can use to buffer records waiting to be sent to the server.
<code>spring.kafka.producer.client-id</code>		ID to pass to the server when making requests. Used for server-side logging.
<code>spring.kafka.producer.compression-type</code>		Compression type for all data generated by the producer.
<code>spring.kafka.producer.key-serializer</code>		Serializer class for keys.
<code>spring.kafka.producer.properties.*</code>		Additional producer-specific properties used to configure the client.
<code>spring.kafka.producer.retries</code>		When greater than zero, enables retrying of failed sends.
<code>spring.kafka.producer.security.protocol</code>		Security protocol used to communicate with brokers.

Key	Default Value	Description
spring.kafka.producer.ssl.key-password		Password of the private key in the key store file.
spring.kafka.producer.ssl.key-store-location		Location of the key store file.
spring.kafka.producer.ssl.key-store-password		Store password for the key store file.
spring.kafka.producer.ssl.key-store-type		Type of the key store.
spring.kafka.producer.ssl.protocol		SSL protocol to use.
spring.kafka.producer.ssl.trust-store-location		Location of the trust store file.
spring.kafka.producer.ssl.trust-store-password		Store password for the trust store file.
spring.kafka.producer.ssl.trust-store-type		Type of the trust store.
spring.kafka.producer.transaction-id-prefix		When non empty, enables transaction support for producer.
spring.kafka.producer.value-serializer		Serializer class for values.

Key	Default Value	Description
<code>spring.kafka.properties.*</code>		Additional properties, common to producers and consumers, used to configure the client.
<code>spring.kafka.security.protocol</code>		Security protocol used to communicate with brokers.
<code>spring.kafka.ssl.key-password</code>		Password of the private key in the key store file.
<code>spring.kafka.ssl.key-store-location</code>		Location of the key store file.
<code>spring.kafka.ssl.key-store-password</code>		Store password for the key store file.
<code>spring.kafka.ssl.key-store-type</code>		Type of the key store.
<code>spring.kafka.ssl.protocol</code>		SSL protocol to use.
<code>spring.kafka.ssl.trust-store-location</code>		Location of the trust store file.
<code>spring.kafka.ssl.trust-store-password</code>		Store password for the trust store file.
<code>spring.kafka.ssl.trust-store-type</code>		Type of the trust store.

Key	Default Value	Description
<code>spring.kafka.streams.application-id</code>		Kafka streams application.id property; default spring.application.name.
<code>spring.kafka.streams.auto-startup</code>	<code>true</code>	Whether or not to auto-start the streams factory bean.
<code>spring.kafka.streams.bootstrap-servers</code>		Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for streams.
<code>spring.kafka.streams.cache-max-size-buffering</code>		Maximum memory size to be used for buffering across all threads.
<code>spring.kafka.streams.cleanup.on-shutdown</code>	<code>true</code>	Cleanup the application's local state directory on shutdown.

Key	Default Value	Description
<code>spring.kafka.streams.cleanup.on-startup</code>	<code>false</code>	Cleanup the application's local state directory on startup.
<code>spring.kafka.streams.client-id</code>		ID to pass to the server when making requests. Used for server-side logging.
<code>spring.kafka.streams.properties.*</code>		Additional Kafka properties used to configure the streams.
<code>spring.kafka.streams.replication-factor</code>		The replication factor for change log topics and repartition topics created by the stream processing application.
<code>spring.kafka.streams.security.protocol</code>		Security protocol used to communicate with brokers.
<code>spring.kafka.streams.ssl.key-password</code>		Password of the private key in the key store file.
<code>spring.kafka.streams.ssl.key-store-location</code>		Location of the key store file.

Key	Default Value	Description
<code>spring.kafka.streams.ssl.key-store-password</code>		Store password for the key store file.
<code>spring.kafka.streams.ssl.key-store-type</code>		Type of the key store.
<code>spring.kafka.streams.ssl.protocol</code>		SSL protocol to use.
<code>spring.kafka.streams.ssl.trust-store-location</code>		Location of the trust store file.
<code>spring.kafka.streams.ssl.trust-store-password</code>		Store password for the trust store file.
<code>spring.kafka.streams.ssl.trust-store-type</code>		Type of the trust store.
<code>spring.kafka.streams.state-dir</code>		Directory location for the state store.
<code>spring.kafka.template.default-topic</code>		Default topic to which messages are sent.
<code>spring.rabbitmq.address-shuffle-mode</code>	none	Mode used to shuffle configured addresses.
<code>spring.rabbitmq.addresses</code>		Comma-separated list of addresses to which the client should connect. When set, the host and port are ignored.

Key	Default Value	Description
<code>spring.rabbitmq.cache.channel.check-out-timeout</code>		Duration to wait to obtain a channel if the cache size has been reached. If 0, always create a new channel.
<code>spring.rabbitmq.cache.channel.size</code>		Number of channels to retain in the cache. When "check-timeout" > 0, max channels per connection.
<code>spring.rabbitmq.cache.connection.mode</code>	<code>channel</code>	Connection factory cache mode.
<code>spring.rabbitmq.cache.connection.size</code>		Number of connections to cache. Only applies when mode is CONNECTION.
<code>spring.rabbitmq.channel-rpc-timeout</code>	<code>10m</code>	Continuation timeout for RPC calls in channels. Set it to zero to wait forever.
<code>spring.rabbitmq.connection-timeout</code>		Connection timeout. Set it to zero to wait forever.

Key	Default Value	Description
<code>spring.rabbitmq.dynamic</code>	<code>true</code>	Whether to create an AmqpAdmin bean.
<code>spring.rabbitmq.host</code>	<code>localhost</code>	RabbitMQ host. Ignored if an address is set.
<code>spring.rabbitmq.listener.direct.acknowledge-mode</code>		Acknowledge mode of container.
<code>spring.rabbitmq.listener.direct.auto-startup</code>	<code>true</code>	Whether to start the container automatically on startup.
<code>spring.rabbitmq.listener.direct.consumers-per-queue</code>		Number of consumers per queue.
<code>spring.rabbitmq.listener.direct.de-batching-enabled</code>	<code>true</code>	Whether the container should present batched messages as discrete messages or call the listener with the batch.
<code>spring.rabbitmq.listener.direct.default-requeue-rejected</code>		Whether rejected deliveries are re-queued by default.
<code>spring.rabbitmq.listener.direct.idle-event-interval</code>		How often idle container events should be published.

Key	Default Value	Description
<code>spring.rabbitmq.listener.direct.missing-queues-fatal</code>	<code>false</code>	Whether to fail if the queues declared by the container are not available on the broker.
<code>spring.rabbitmq.listener.direct.prefetch</code>		Maximum number of unacknowledged messages that can be outstanding at each consumer.
<code>spring.rabbitmq.listener.direct.retry.enabled</code>	<code>false</code>	Whether publishing retries are enabled.
<code>spring.rabbitmq.listener.direct.retry.initial-interval</code>	<code>1000ms</code>	Duration between the first and second attempt to deliver a message.
<code>spring.rabbitmq.listener.direct.retry.max-attempts</code>	<code>3</code>	Maximum number of attempts to deliver a message.
<code>spring.rabbitmq.listener.direct.retry.max-interval</code>	<code>10000ms</code>	Maximum duration between attempts.
<code>spring.rabbitmq.listener.direct.retry.multiplier</code>	<code>1</code>	Multiplier to apply to the previous retry interval.

Key	Default Value	Description
<code>spring.rabbitmq.listener.direct.retry.stateless</code>	true	Whether retries are stateless or stateful.
<code>spring.rabbitmq.listener.simple.acknowledge-mode</code>		Acknowledge mode of container.
<code>spring.rabbitmq.listener.simple.auto-startup</code>	true	Whether to start the container automatically on startup.
<code>spring.rabbitmq.listener.simple.batch-size</code>		Batch size, expressed as the number of physical messages, to be used by the container.
<code>spring.rabbitmq.listener.simple.corecurrency</code>		Minimum number of listener invoker threads.

Key	Default Value	Description
<code>spring.rabbitmq.listener.simple.consume-batch-enabled</code>	false	<p>Whether the container creates a batch of messages based on the 'receive-timeout' and 'batch-size'. Coerces 'de-batching-enabled' to true to include the contents of a producer created batch in the batch as discrete records.</p>
<code>spring.rabbitmq.listener.simple.de-batching-enabled</code>	true	<p>Whether the container should present batched messages as discrete messages or call the listener with the batch.</p>
<code>spring.rabbitmq.listener.simple.default-requeue-rejected</code>		<p>Whether rejected deliveries are re-queued by default.</p>
<code>spring.rabbitmq.listener.simple.idle-event-interval</code>		<p>How often idle container events should be published.</p>

Key	Default Value	Description
<code>spring.rabbitmq.listener.simple.max-concurrency</code>		Maximum number of Listener invoker threads.
<code>spring.rabbitmq.listener.simple.missing-queues-fatal</code>	<code>true</code>	Whether to fail if the queues declared by the container are not available on the broker and/or whether to stop the container if one or more queues are deleted at runtime.
<code>spring.rabbitmq.listener.simple.prefetch</code>		Maximum number of unacknowledged messages that can be outstanding at each consumer.
<code>spring.rabbitmq.listener.simple.retry.enabled</code>	<code>false</code>	Whether publishing retries are enabled.
<code>spring.rabbitmq.listener.simple.retry.initial-interval</code>	<code>1000ms</code>	Duration between the first and second attempt to deliver a message.

Key	Default Value	Description
<code>spring.rabbitmq.listener.simple.retry.max-attempts</code>	3	Maximum number of attempts to deliver a message.
<code>spring.rabbitmq.listener.simple.retry.max-interval</code>	10000ms	Maximum duration between attempts.
<code>spring.rabbitmq.listener.simple.retry.multiplier</code>	1	Multiplier to apply to the previous retry interval.
<code>spring.rabbitmq.listener.simple.retry.stateless</code>	true	Whether retries are stateless or stateful.
<code>spring.rabbitmq.listener.type</code>	simple	Listener container type.
<code>spring.rabbitmq.password</code>	guest	Login to authenticate against the broker.
<code>spring.rabbitmq.port</code>		RabbitMQ port. Ignored if an address is set. Default to 5672, or 5671 if SSL is enabled.
<code>spring.rabbitmq.publisher-confirm-type</code>		Type of publisher confirms to use.
<code>spring.rabbitmq.publisher-returns</code>	false	Whether to enable publisher returns.

Key	Default Value	Description
<code>spring.rabbitmq.requested-channel-max</code>	2047	Number of channels per connection requested by the client. Use 0 for unlimited.
<code>spring.rabbitmq.requested-heartbeat</code>		Requested heartbeat timeout; zero for none. If a duration suffix is not specified, seconds will be used.
<code>spring.rabbitmq.ssl.algorithm</code>		SSL algorithm to use. By default, configured by the Rabbit client library.
<code>spring.rabbitmq.ssl.enabled</code>		Whether to enable SSL support. Determined automatically if an address is provided with the protocol (amqp:// vs. amqps://).
<code>spring.rabbitmq.ssl.key-store</code>		Path to the key store that holds the SSL certificate.

Key	Default Value	Description
<code>spring.rabbitmq.ssl.key-store-password</code>		Password used to access the key store.
<code>spring.rabbitmq.ssl.key-store-type</code>	PKCS12	Key store type.
<code>spring.rabbitmq.ssl.trust-store</code>		Trust store that holds SSL certificates.
<code>spring.rabbitmq.ssl.trust-store-password</code>		Password used to access the trust store.
<code>spring.rabbitmq.ssl.trust-store-type</code>	JKS	Trust store type.
<code>spring.rabbitmq.ssl.validate-server-certificate</code>	true	Whether to enable server side certificate validation.
<code>spring.rabbitmq.ssl.verify-hostname</code>	true	Whether to enable hostname verification.
<code>spring.rabbitmq.template.default-receive-queue</code>		Name of the default queue to receive messages from when none is specified explicitly.
<code>spring.rabbitmq.template.exchange</code>		Name of the default exchange to use for send operations.

Key	Default Value	Description
<code>spring.rabbitmq.template.mandatory</code>		Whether to enable mandatory messages.
<code>spring.rabbitmq.template.receive-timeout</code>		Timeout for `receive()` operations.
<code>spring.rabbitmq.template.reply-timeout</code>		Timeout for `sendAndReceive()` operations.
<code>spring.rabbitmq.template.retry.enabled</code>	false	Whether publishing retries are enabled.
<code>spring.rabbitmq.template.retry.initial-interval</code>	1000ms	Duration between the first and second attempt to deliver a message.
<code>spring.rabbitmq.template.retry.max-attempts</code>	3	Maximum number of attempts to deliver a message.
<code>spring.rabbitmq.template.retry.max-interval</code>	10000ms	Maximum duration between attempts.
<code>spring.rabbitmq.template.retry.multiplier</code>	1	Multiplier to apply to the previous retry interval.

Key	Default Value	Description
<code>spring.rabbitmq.template.routing-key</code>		Value of a default routing key to use for send operations.
<code>spring.rabbitmq.username</code>	<code>guest</code>	Login user to authenticate to the broker.
<code>spring.rabbitmq.virtual-host</code>		Virtual host to use when connecting to the broker.

11.A.9. Web Properties

Key	Default Value	Description
<code>spring.hateoas.use-hal-as-default-json-media-type</code>	<code>true</code>	Whether application/hal+json responses should be sent to requests that accept application/json.
<code>spring.jersey.application-path</code>		Path that serves as the base URI for the application. If specified, overrides the value of <code>@ApplicationPath</code> .

Key	Default Value	Description
<code>spring.jersey.filter.order</code>	0	Jersey filter chain order.
<code>spring.jersey.init.*</code>		Init parameters to pass to Jersey through the servlet or filter.
<code>spring.jersey.servlet.load-on-startup</code>	-1	Load on startup priority of the Jersey servlet.
<code>spring.jersey.type</code>	servlet	Jersey integration type.
<code>spring.mvc.async.request-timeout</code>		Amount of time before asynchronous request handling times out. If this value is not set, the default timeout of the underlying implementation is used.
<code>spring.mvc.contentnegotiation.favor-parameter</code>	false	Whether a request parameter ("format" by default) should be used to determine the requested media type.

Key	Default Value	Description
<code>spring.mvc.contentnegotiation.media-types.*</code>		Map file extensions to media types for content negotiation. For instance, <code>yml</code> to <code>text/yaml</code> .
<code>spring.mvc.contentnegotiation.parameter-name</code>		Query parameter name to use when "favor-parameter" is enabled.
<code>spring.mvc.converters.preferred-json-mapper</code>		Preferred JSON mapper to use for HTTP message conversion. By default, auto-detected according to the environment.
<code>spring.mvc.dispatch-options-request</code>	<code>true</code>	Whether to dispatch OPTIONS requests to the <code>FrameworkServlet doService</code> method.
<code>spring.mvc.dispatch-trace-request</code>	<code>false</code>	Whether to dispatch TRACE requests to the <code>FrameworkServlet doService</code> method.
<code>spring.mvc.format.date</code>		Date format to use, for example <code>'dd/MM/yyyy'</code> .

Key	Default Value	Description
<code>spring.mvc.format.date-time</code>		Date-time format to use, for example `yyyy-MM-dd HH:mm:ss`.
<code>spring.mvc.format.time</code>		Time format to use, for example `HH:mm:ss`.
<code>spring.mvc.formcontent.filter.enabled</code>	<code>true</code>	Whether to enable Spring's FormContentFilter.
<code>spring.mvc.hiddenmethod.filter.enabled</code>	<code>false</code>	Whether to enable Spring's HiddenHttpMethodFilter.
<code>spring.mvc.ignore-default-model-on-redirect</code>	<code>true</code>	Whether the content of the "default" model should be ignored during redirect scenarios.
<code>spring.mvc.log-request-details</code>	<code>false</code>	Whether logging of (potentially sensitive) request details at DEBUG and TRACE level is allowed.

Key	Default Value	Description
<code>spring.mvc.log-resolved-exception</code>	<code>false</code>	Whether to enable warn logging of exceptions resolved by a "HandlerExceptionResolver", except for "DefaultHandlerExceptionResolver".
<code>spring.mvc.message-codes-resolver-format</code>		Formatting strategy for message codes. For instance, `PREFIX_ERROR_CODE`.
<code>spring.mvc.pathmatch.matching-strategy</code>	<code>ant-path-matcher</code>	Choice of strategy for matching request paths against registered mappings.
<code>spring.mvc.publish-request-handled-events</code>	<code>true</code>	Whether to publish a <code>ServletRequestHandledEvent</code> at the end of each request.
<code>spring.mvc.servlet.load-on-startup</code>	<code>-1</code>	Load on startup priority of the dispatcher servlet.

Key	Default Value	Description
<code>spring.mvc.servlet.path</code>	/	Path of the dispatcher servlet. Setting a custom value for this property is not compatible with the PathPatternParser matching strategy.
<code>spring.mvc.static-path-pattern</code>	/**	Path pattern used for static resources.
<code>spring.mvc.throw-exception-if-no-handler-found</code>	false	Whether a "NoHandlerFoundException" should be thrown if no Handler was found to process a request.
<code>spring.mvc.view.prefix</code>		Spring MVC view prefix.
<code>spring.mvc.view.suffix</code>		Spring MVC view suffix.
<code>spring.servlet.multipart.enabled</code>	true	Whether to enable support of multipart uploads.
<code>spring.servlet.multipart.file-size-threshold</code>	0B	Threshold after which files are written to disk.

Key	Default Value	Description
<code>spring.servlet.multipart.location</code>		Intermediate location of uploaded files.
<code>spring.servlet.multipart.max-file-size</code>	1MB	Max file size.
<code>spring.servlet.multipart.max-request-size</code>	10MB	Max request size.
<code>spring.servlet.multipart.resolve-lazily</code>	false	Whether to resolve the multipart request lazily at the time of file or parameter access.
<code>spring.session.hazelcast.flush-mode</code>	on-save	Sessions flush mode. Determines when session changes are written to the session store.
<code>spring.session.hazelcast.map-name</code>	<code>spring:session:sessions</code>	Name of the map used to store sessions.
<code>spring.session.hazelcast.save-mode</code>	on-set-attribute	Sessions save mode. Determines how session changes are tracked and saved to the session store.

Key	Default Value	Description
<code>spring.session.jdbc.cleanup-cron</code>	<code>0 * * * * *</code>	Cron expression for expired session cleanup job.
<code>spring.session.jdbc.flush-mode</code>	<code>on-save</code>	Sessions flush mode. Determines when session changes are written to the session store.
<code>spring.session.jdbc.initialize-schema</code>	<code>embedded</code>	Database schema initialization mode.
<code>spring.session.jdbc.save-mode</code>	<code>on-set-attribute</code>	Sessions save mode. Determines how session changes are tracked and saved to the session store.
<code>spring.session.jdbc.schema</code>	<code>classpath:org/springframework/session/jdbc/schema-@@platform@@.sql</code>	Path to the SQL file to use to initialize the database schema.
<code>spring.session.jdbc.table-name</code>	<code>SPRING_SESSION</code>	Name of the database table used to store sessions.
<code>spring.session.mongodb.collection-name</code>	<code>sessions</code>	Collection name used to store sessions.

Key	Default Value	Description
<code>spring.session.redis.cleanup-cron</code>	<code>0 * * * * *</code>	Cron expression for expired session cleanup job.
<code>spring.session.redis.configure-action</code>	<code>notify-keyspace-events</code>	The configure action to apply when no user defined ConfigureRedisAction bean is present.
<code>spring.session.redis.flush-mode</code>	<code>on-save</code>	Sessions flush mode. Determines when session changes are written to the session store.
<code>spring.session.redis.namespace</code>	<code>spring:session</code>	Namespace for keys used to store sessions.
<code>spring.session.redis.save-mode</code>	<code>on-set-attribute</code>	Sessions save mode. Determines how session changes are tracked and saved to the session store.
<code>spring.session.servlet.filter-dispatcher-types</code>	<code>[async, error, request]</code>	Session repository filter dispatcher types.

Key	Default Value	Description
<code>spring.session.servlet.filter-order</code>		Session repository filter order.
<code>spring.session.store-type</code>		Session store type.
<code>spring.session.timeout</code>		Session timeout. If a duration suffix is not specified, seconds will be used.
<code>spring.web.locale</code>		Locale to use. By default, this locale is overridden by the "Accept-Language" header.
<code>spring.web.locale-resolver</code>	<code>accept-header</code>	Define how the locale should be resolved.
<code>spring.web.resources.add-mappings</code>	<code>true</code>	Whether to enable default resource handling.
<code>spring.web.resources.cache.cachecontrol.cache-control.cache-private</code>		Indicate that the response message is intended for a single user and must not be stored by a shared cache.

Key	Default Value	Description
<code>spring.web.resources.cache.cachecontrol.cache-public</code>		Indicate that any cache may store the response.
<code>spring.web.resources.cache.cachecontrol.max-age</code>		Maximum time the response should be cached, in seconds if no duration suffix is not specified.
<code>spring.web.resources.cache.cachecontrol.must-revalidate</code>		Indicate that once it has become stale, a cache must not use the response without re-validating it with the server.
<code>spring.web.resources.cache.cachecontrol.no-cache</code>		Indicate that the cached response can be reused only if re-validated with the server.
<code>spring.web.resources.cache.cachecontrol.no-store</code>		Indicate to not cache the response in any case.

Key	Default Value	Description
<code>spring.web.resources.cache.cachecontrol.no-transform</code>		Indicate intermediaries (caches and others) that they should not transform the response content.
<code>spring.web.resources.cache.cachecontrol.proxy-revalidate</code>		Same meaning as the "must-revalidate" directive, except that it does not apply to private caches.
<code>spring.web.resources.cache.cachecontrol.s-max-age</code>		Maximum time the response should be cached by shared caches, in seconds if no duration suffix is not specified.
<code>spring.web.resources.cache.cachecontrol.stale-if-error</code>		Maximum time the response may be used when errors are encountered, in seconds if no duration suffix is not specified.

Key	Default Value	Description
<code>spring.web.resources.cache.cachecontrol.stale-while-revalidate</code>		Maximum time the response can be served after it becomes stale, in seconds if no duration suffix is not specified.
<code>spring.web.resources.cache.period</code>		Cache period for the resources served by the resource handler. If a duration suffix is not specified, seconds will be used. Can be overridden by the 'spring.web.resources.cache.cachecontrol' properties.
<code>spring.web.resources.cache.use-last-modified</code>	<code>true</code>	Whether we should use the "lastModified" metadata of the files in HTTP caching headers.
<code>spring.web.resources.chain.cache</code>	<code>true</code>	Whether to enable caching in the Resource chain.

Key	Default Value	Description
<code>spring.web.resources.chain.compress</code>	<code>false</code>	Whether to enable resolution of already compressed resources (gzip, brotli). Checks for a resource name with the '.gz' or '.br' file extensions.
<code>spring.web.resources.chain.enabled</code>		Whether to enable the Spring Resource Handling chain. By default, disabled unless at least one strategy has been enabled.
<code>spring.web.resources.chain.strategy.content.enabled</code>	<code>false</code>	Whether to enable the content Version Strategy.
<code>spring.web.resources.chain.strategy.content.paths</code>	<code>[/**]</code>	Comma-separated list of patterns to apply to the content Version Strategy.
<code>spring.web.resources.chain.strategy.fixed.enabled</code>	<code>false</code>	Whether to enable the fixed Version Strategy.

Key	Default Value	Description
<code>spring.web.resources.chain.strategy.fixed.paths</code>	<code>[/**]</code>	Comma-separated list of patterns to apply to the fixed Version Strategy.
<code>spring.web.resources.chain.strategy.fixed.version</code>		Version string to use for the fixed Version Strategy.
<code>spring.web.resources.static-locations</code>	<code>[classpath:/META-INF/resources/, classpath:/resources/, classpath:/static/, classpath:/public/]</code>	Locations of static resources. Defaults to classpath:[/META-INF/resources/, /resources/, /static/, /public/].
<code>spring.webflux.base-path</code>		Base path for all web handlers.
<code>spring.webflux.format.date</code>		Date format to use, for example `dd/MM/yyyy`.
<code>spring.webflux.format.date-time</code>		Date-time format to use, for example `yyyy-MM-dd HH:mm:ss`.
<code>spring.webflux.format.time</code>		Time format to use, for example `HH:mm:ss`.
<code>spring.webflux.hiddenmethod.filter.enabled</code>	<code>false</code>	Whether to enable Spring's HiddenHttpMethodFilter.

Key	Default Value	Description
<code>spring.webflux.static-path-pattern</code>	<code>/**</code>	Path pattern used for static resources.
<code>spring.webservices.path</code>	<code>/services</code>	Path that serves as the base URI for the services.
<code>spring.webservices.servlet.init.*</code>		Servlet init parameters to pass to Spring Web Services.
<code>spring.webservices.servlet.load-on-startup</code>	-1	Load on startup priority of the Spring Web Services servlet.
<code>spring.webservices.wsdl-locations</code>		Comma-separated list of locations of WSDLs and accompanying XSDs to be exposed as beans.

11.A.10. Templating Properties

Key	Default Value	Description
<code>spring.freemarker.allow-request-override</code>	<code>false</code>	Whether <code>HttpServletRequest</code> attributes are allowed to override (hide) controller generated model attributes of the same name.
<code>spring.freemarker.allow-session-override</code>	<code>false</code>	Whether <code>HttpSession</code> attributes are allowed to override (hide) controller generated model attributes of the same name.
<code>spring.freemarker.cache</code>	<code>false</code>	Whether to enable template caching.
<code>spring.freemarker.charset</code>	<code>UTF-8</code>	Template encoding.
<code>spring.freemarker.check-template-location</code>	<code>true</code>	Whether to check that the templates location exists.
<code>spring.freemarker.content-type</code>	<code>text/html</code>	Content-Type value.
<code>spring.freemarker.enabled</code>	<code>true</code>	Whether to enable MVC view resolution for this technology.

Key	Default Value	Description
<code>spring.freemarker.expose-request-attributes</code>	<code>false</code>	Whether all request attributes should be added to the model prior to merging with the template.
<code>spring.freemarker.expose-session-attributes</code>	<code>false</code>	Whether all HttpSession attributes should be added to the model prior to merging with the template.
<code>spring.freemarker.expose-spring-macro-helpers</code>	<code>true</code>	Whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".

Key	Default Value	Description
<code>spring.freemarker.prefer-file-system-access</code>	<code>false</code>	Whether to prefer file system access for template loading to enable hot detection of template changes. When a template path is detected as a directory, templates are loaded from the directory only and other matching classpath locations will not be considered.
<code>spring.freemarker.prefix</code>		Prefix that gets prepended to view names when building a URL.
<code>spring.freemarker.request-context-attribute</code>		Name of the RequestContext attribute for all views.
<code>spring.freemarker.settings.*</code>		Well-known FreeMarker keys which are passed to FreeMarker's Configuration.

Key	Default Value	Description
<code>spring.freemarker.suffix</code>	<code>.ftlh</code>	Suffix that gets appended to view names when building a URL.
<code>spring.freemarker.template-loader-path</code>	<code>[classpath:/templates/]</code>	Comma-separated list of template paths.
<code>spring.freemarker.view-names</code>		View names that can be resolved.
<code>spring.groovy.template.allow-request-override</code>	<code>false</code>	Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.
<code>spring.groovy.template.allow-session-override</code>	<code>false</code>	Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.
<code>spring.groovy.template.cache</code>	<code>false</code>	Whether to enable template caching.
<code>spring.groovy.template.charset</code>	<code>UTF-8</code>	Template encoding.

Key	Default Value	Description
spring.groovy.template.check-template-location	true	Whether to check that the templates location exists.
spring.groovy.template.configuration.auto-escape spring.groovy.template.configuration.auto-indent spring.groovy.template.configuration.auto-indent-string spring.groovy.template.configuration.auto-new-line spring.groovy.template.configuration.base-template-class spring.groovy.template.configuration.cache-templates spring.groovy.template.configuration.declaration-encoding spring.groovy.template.configuration.expand-empty-elements spring.groovy.template.configuration.locale spring.groovy.template.configuration.newLine-string spring.groovy.template.configuration.resource-loader-path spring.groovy.template.configuration.use-double-quotes		See GroovyMarkupConfigurer
spring.groovy.template.content-type	text/html	Content-Type value.

Key	Default Value	Description
<code>spring.groovy.template.enabled</code>	<code>true</code>	Whether to enable MVC view resolution for this technology.
<code>spring.groovy.template.expose-request-attributes</code>	<code>false</code>	Whether all request attributes should be added to the model prior to merging with the template.
<code>spring.groovy.template.expose-session-attributes</code>	<code>false</code>	Whether all HttpSession attributes should be added to the model prior to merging with the template.
<code>spring.groovy.template.expose-spring-macro-helpers</code>	<code>true</code>	Whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".
<code>spring.groovy.template.prefix</code>		Prefix that gets prepended to view names when building a URL.

Key	Default Value	Description
<code>spring.groovy.template.request-context-attribute</code>		Name of the RequestContext attribute for all views.
<code>spring.groovy.template.resource-loader-path</code>	<code>classpath:/templates/</code>	Template path.
<code>spring.groovy.template.suffix</code>	<code>.tpl</code>	Suffix that gets appended to view names when building a URL.
<code>spring.groovy.template.view-names</code>		View names that can be resolved.
<code>spring.mustache.allow-request-override</code>	<code>false</code>	Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.
<code>spring.mustache.allow-session-override</code>	<code>false</code>	Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.
<code>spring.mustache.cache</code>	<code>false</code>	Whether to enable template caching.

Key	Default Value	Description
<code>spring.mustache.charset</code>	<code>UTF-8</code>	Template encoding.
<code>spring.mustache.check-template-location</code>	<code>true</code>	Whether to check that the templates location exists.
<code>spring.mustache.content-type</code>	<code>text/html</code>	Content-Type value.
<code>spring.mustache.enabled</code>	<code>true</code>	Whether to enable MVC view resolution for this technology.
<code>spring.mustache.expose-request-attributes</code>	<code>false</code>	Whether all request attributes should be added to the model prior to merging with the template.
<code>spring.mustache.expose-session-attributes</code>	<code>false</code>	Whether all HttpSession attributes should be added to the model prior to merging with the template.

Key	Default Value	Description
<code>spring.mustache.expose-spring-macro-helpers</code>	<code>true</code>	Whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".
<code>spring.mustache.prefix</code>	<code>classpath:/templates/</code>	Prefix to apply to template names.
<code>spring.mustache.request-context-attribute</code>		Name of the RequestContext attribute for all views.
<code>spring.mustache.suffix</code>	<code>.mustache</code>	Suffix to apply to template names.
<code>spring.mustache.view-names</code>		View names that can be resolved.
<code>spring.thymeleaf.cache</code>	<code>true</code>	Whether to enable template caching.
<code>spring.thymeleaf.check-template</code>	<code>true</code>	Whether to check that the template exists before rendering it.
<code>spring.thymeleaf.check-template-location</code>	<code>true</code>	Whether to check that the templates location exists.

Key	Default Value	Description
<code>spring.thymeleaf.enable-spring-el-compiler</code>	<code>false</code>	Enable the SpringEL compiler in SpringEL expressions.
<code>spring.thymeleaf.enabled</code>	<code>true</code>	Whether to enable Thymeleaf view resolution for Web frameworks.
<code>spring.thymeleaf.encoding</code>	<code>UTF-8</code>	Template files encoding.
<code>spring.thymeleaf.excluded-view-names</code>		Comma-separated list of view names (patterns allowed) that should be excluded from resolution.
<code>spring.thymeleaf.mode</code>	<code>HTML</code>	Template mode to be applied to templates. See also Thymeleaf's <code>TemplateMode</code> enum.
<code>spring.thymeleaf.prefix</code>	<code>classpath:/templates/</code>	Prefix that gets prepended to view names when building a URL.

Key	Default Value	Description
<code>spring.thymeleaf.reactive.chunked-mode-view-names</code>		Comma-separated list of view names (patterns allowed) that should be the only ones executed in CHUNKED mode when a max chunk size is set.
<code>spring.thymeleaf.reactive.full-mode-view-names</code>		Comma-separated list of view names (patterns allowed) that should be executed in FULL mode even if a max chunk size is set.
<code>spring.thymeleaf.reactive.max-chunk-size</code>	0B	Maximum size of data buffers used for writing to the response. Templates will execute in CHUNKED mode by default if this is set.
<code>spring.thymeleaf.reactive.media-types</code>		Media types supported by the view technology.

Key	Default Value	Description
<code>spring.thymeleaf.render-hidden-markers-before-checkboxes</code>	<code>false</code>	Whether hidden form inputs acting as markers for checkboxes should be rendered before the checkbox element itself.
<code>spring.thymeleaf.servlet.content-type</code>	<code>text/html</code>	Content-Type value written to HTTP responses.
<code>spring.thymeleaf.servlet.produce-partial-output-while-processing</code>	<code>true</code>	Whether Thymeleaf should start writing partial output as soon as possible or buffer until template processing is finished.
<code>spring.thymeleaf.suffix</code>	<code>.html</code>	Suffix that gets appended to view names when building a URL.

Key	Default Value	Description
<code>spring.thymeleaf.template-resolver-order</code>		Order of the template resolver in the chain. By default, the template resolver is first in the chain. Order start at 1 and should only be set if you have defined additional "TemplateResolver" beans.
<code>spring.thymeleaf.view-names</code>		Comma-separated list of view names (patterns allowed) that can be resolved.

11.A.11. Server Properties

Key	Default Value	Description
<code>server.address</code>		Network address to which the server should bind.
<code>server.compression.enabled</code>	<code>false</code>	Whether response compression is enabled.

Key	Default Value	Description
<code>server.compression.excluded-user-agents</code>		Comma-separated list of user agents for which responses should not be compressed.
<code>server.compression.mime-types</code>	[<code>text/html</code> , <code>text/xml</code> , <code>text/plain</code> , <code>text/css</code> , <code>text/javascript</code> , <code>application/javascript</code> , <code>application/json</code> , <code>application/xml</code>]	Comma-separated list of MIME types that should be compressed.
<code>server.compression.min-response-size</code>	2KB	Minimum "Content-Length" value that is required for compression to be performed.
<code>server.error.include-binding-errors</code>	never	When to include "errors" attribute.
<code>server.error.include-exception</code>	false	Include the "exception" attribute.
<code>server.error.include-message</code>	never	When to include "message" attribute.
<code>server.error.include-stacktrace</code>	never	When to include the "trace" attribute.

Key	Default Value	Description
<code>server.error.path</code>	<code>/error</code>	Path of the error controller.
<code>server.error.whitelabel.enabled</code>	<code>true</code>	Whether to enable the default error page displayed in browsers in case of a server error.
<code>server.forward-headers-strategy</code>		Strategy for handling X-Forwarded-* headers.
<code>server.http2.enabled</code>	<code>false</code>	Whether to enable HTTP/2 support, if the current environment supports it.
<code>server.jetty.accesslog.append</code>	<code>false</code>	Append to log.
<code>server.jetty.accesslog.custom-format</code>		Custom log format, see <code>org.eclipse.jetty.server.CustonRequestLog</code> . If defined, overrides the "format" configuration key.
<code>server.jetty.accesslog.enabled</code>	<code>false</code>	Enable access log.

Key	Default Value	Description
server.jetty.accesslog.file-date-format		Date format to place in log file name.
server.jetty.accesslog.filename		Log filename. If not specified, logs redirect to "System.err".
server.jetty.accesslog.format	ncsa	Log format.
server.jetty.accesslog.ignore-paths		Request paths that should not be logged.
server.jetty.accesslog.retention-period	31	Number of days before rotated log files are deleted.
server.jetty.connection-idle-timeout		Time that the connection can be idle before it is closed.
server.jetty.max-http-form-post-size	200000B	Maximum size of the form content in any HTTP post request.

Key	Default Value	Description
<code>server.jetty.threads.acceptors</code>	-1	Number of acceptor threads to use. When the value is -1, the default, the number of acceptors is derived from the operating environment.
<code>server.jetty.threads.idle-timeout</code>	60000ms	Maximum thread idle time.
<code>server.jetty.threads.max</code>	200	Maximum number of threads.
<code>server.jetty.threads.max-queue-capacity</code>		Maximum capacity of the thread pool's backing queue. A default is computed based on the threading configuration.
<code>server.jetty.threads.min</code>	8	Minimum number of threads.

Key	Default Value	Description
<code>server.jetty.threads.selectors</code>	-1	Number of selector threads to use. When the value is -1, the default, the number of selectors is derived from the operating environment.
<code>server.max-http-header-size</code>	8KB	Maximum size of the HTTP message header.
<code>server.netty.connection-timeout</code>		Connection timeout of the Netty channel.
<code>server.netty.h2c-max-content-length</code>	0B	Maximum content length of an H2C upgrade request.
<code>server.netty.initial-buffer-size</code>	128B	Initial buffer size for HTTP request decoding.
<code>server.netty.max-chunk-size</code>	8KB	Maximum chunk size that can be decoded for an HTTP request.
<code>server.netty.max-initial-line-length</code>	4KB	Maximum length that can be decoded for an HTTP request's initial line.

Key	Default Value	Description
server.netty.validate-headers	true	Whether to validate headers when decoding requests.
server.port	8080	Server HTTP port.
server.server-header		Value to use for the Server response header (if empty, no header is sent).
server.servlet.application-display-name	application	Display name of the application.
server.servlet.context-parameters.*		Servlet context init parameters.
server.servlet.context-path		Context path of the application.
server.servlet.encoding.charset		
server.servlet.encoding.enabled	true	Whether to enable http encoding support.
server.servlet.encoding.force		
server.servlet.encoding.force-request		
server.servlet.encoding.force-response		
server.servlet.encoding.mapping.*		

Key	Default Value	Description
<code>server.servlet.jsp.class-name</code>	<code>org.apache.jasper.servlet.JspServlet</code>	Class name of the servlet to use for JSPs. If registered is true and this class * is on the classpath then it will be registered.
<code>server.servlet.jsp.init-parameters.*</code>		Init parameters used to configure the JSP servlet.
<code>server.servlet.jsp.registered</code>	<code>true</code>	Whether the JSP servlet is registered.
<code>server.servlet.register-default-servlet</code>	<code>false</code>	Whether to register the default Servlet with the container.
<code>server.servlet.session.cookie.comment</code>		Comment for the session cookie.
<code>server.servlet.session.cookie.domain</code>		Domain for the session cookie.
<code>server.servlet.session.cookie.http-only</code>		Whether to use "HttpOnly" cookies for session cookies.

Key	Default Value	Description
<code>server.servlet.session.cookie.max-age</code>		Maximum age of the session cookie. If a duration suffix is not specified, seconds will be used.
<code>server.servlet.session.cookie.name</code>		Session cookie name.
<code>server.servlet.session.cookie.path</code>		Path of the session cookie.
<code>server.servlet.session.cookie.secure</code>		Whether to always mark the session cookie as secure.
<code>server.servlet.session.persistent</code>	false	Whether to persist session data between restarts.
<code>server.servlet.session.store-dir</code>		Directory used to store session data.
<code>server.servlet.session.timeout</code>	30m	Session timeout. If a duration suffix is not specified, seconds will be used.
<code>server.servlet.session.tracking-modes</code>		Session tracking modes.

Key	Default Value	Description
server.shutdown	immediate	Type of shutdown that the server will support.
server.ssl.ciphers		Supported SSL ciphers.
server.ssl.client-auth		Client authentication mode. Requires a trust store.
server.ssl.enabled	true	Whether to enable SSL support.
server.ssl.enabled-protocols		Enabled SSL protocols.
server.ssl.key-alias		Alias that identifies the key in the key store.
server.ssl.key-password		Password used to access the key in the key store.
server.ssl.key-store		Path to the key store that holds the SSL certificate (typically a jks file).
server.ssl.key-store-password		Password used to access the key store.
server.ssl.key-store-provider		Provider for the key store.

Key	Default Value	Description
server.ssl.key-store-type		Type of the key store.
server.ssl.protocol	TLS	SSL protocol to use.
server.ssl.trust-store		Trust store that holds SSL certificates.
server.ssl.trust-store-password		Password used to access the trust store.
server.ssl.trust-store-provider		Provider for the trust store.
server.ssl.trust-store-type		Type of the trust store.
server.tomcat.accept-count	100	Maximum queue length for incoming connection requests when all possible request processing threads are in use.
server.tomcat.accesslog.buffered	true	Whether to buffer output such that it is flushed only periodically.

Key	Default Value	Description
<code>server.tomcat.accesslog.check-exists</code>	<code>false</code>	Whether to check for log file existence so it can be recreated if an external process has renamed it.
<code>server.tomcat.accesslog.condition-if</code>		Whether logging of the request will only be enabled if "ServletRequest.getAttribute(conditionIf)" does not yield null.
<code>server.tomcat.accesslog.condition-unless</code>		Whether logging of the request will only be enabled if "ServletRequest.getAttribute(conditionUnless)" yield null.
<code>server.tomcat.accesslog.directory</code>	<code>logs</code>	Directory in which log files are created. Can be absolute or relative to the Tomcat base dir.
<code>server.tomcat.accesslog.enabled</code>	<code>false</code>	Enable access log.

Key	Default Value	Description
<code>server.tomcat.accesslog.encoding</code>		Character set used by the log file. Default to the system default character set.
<code>server.tomcat.accesslog.file-date-format</code>	<code>.yyyy-MM-dd</code>	Date format to place in the log file name.
<code>server.tomcat.accesslog.ipv6-canonical</code>	<code>false</code>	Whether to use IPv6 canonical representation format as defined by RFC 5952.
<code>server.tomcat.accesslog.locale</code>		Locale used to format timestamps in log entries and in log file name suffix. Default to the default locale of the Java process.
<code>server.tomcat.accesslog.max-days</code>	<code>-1</code>	Number of days to retain the access log files before they are removed.
<code>server.tomcat.accesslog.pattern</code>	<code>common</code>	Format pattern for access logs.
<code>server.tomcat.accesslog.prefix</code>	<code>access_log</code>	Log file name prefix.

Key	Default Value	Description
<code>server.tomcat.accesslog.rename-on-rotate</code>	<code>false</code>	Whether to defer inclusion of the date stamp in the file name until rotate time.
<code>server.tomcat.accesslog.request-attributes-enabled</code>	<code>false</code>	Set request attributes for the IP address, Hostname, protocol, and port used for the request.
<code>server.tomcat.accesslog.rotate</code>	<code>true</code>	Whether to enable access log rotation.
<code>server.tomcat.accesslog.suffix</code>	<code>.log</code>	Log file name suffix.

Key	Default Value	Description
<code>server.tomcat.additional-tld-skip-patterns</code>		Comma-separated list of additional patterns that match jars to ignore for TLD scanning. The special '?' and '*' characters can be used in the pattern to match one and only one character and zero or more characters respectively.
<code>server.tomcat.background-processor-delay</code>	10s	Delay between the invocation of backgroundProcess methods. If a duration suffix is not specified, seconds will be used.
<code>server.tomcat.basedir</code>		Tomcat base directory. If not specified, a temporary directory is used.

Key	Default Value	Description
<code>server.tomcat.connection-timeout</code>		Amount of time the connector will wait, after accepting a connection, for the request URI line to be presented.
<code>server.tomcat.max-connections</code>	8192	Maximum number of connections that the server accepts and processes at any given time. Once the limit has been reached, the operating system may still accept connections based on the "acceptCount" property.
<code>server.tomcat.max-http-form-post-size</code>	2MB	Maximum size of the form content in any HTTP post request.
<code>server.tomcat.max-swallow-size</code>	2MB	Maximum amount of request body to swallow.
<code>server.tomcat.mbeanregistry.enabled</code>	false	Whether Tomcat's MBean Registry should be enabled.

Key	Default Value	Description
<code>server.tomcat.processor-cache</code>	200	Maximum number of idle processors that will be retained in the cache and reused with a subsequent request. When set to -1 the cache will be unlimited with a theoretical maximum size equal to the maximum number of connections.
<code>server.tomcat.redirect-context-root</code>	true	Whether requests to the context root should be redirected by appending a / to the path. When using SSL terminated at a proxy, this property should be set to false.

Key	Default Value	Description
<code>server.tomcat.relaxed-path-chars</code>		Comma-separated list of additional unencoded characters that should be allowed in URI paths. Only "< > [\] ^ ` { }" are allowed.
<code>server.tomcat.relaxed-query-chars</code>		Comma-separated list of additional unencoded characters that should be allowed in URI query strings. Only "< > [\] ^ ` { }" are allowed.
<code>server.tomcat.remoteip.host-header</code>	<code>X-Forwarded-Host</code>	Name of the HTTP header from which the remote host is extracted.

Key	Default Value	Description
server.tomcat.remoteip.internal-proxies	<code>10\\.\\.\\d{1,3}\\.\\.\\d{1,3}\\.192\\.168\\.\\d{1,3}\\.169\\.254\\.\\d{1,3}\\.127\\.\\d{1,3}\\.\\d{1,3}\\.172\\.1[6-9]{1}\\.\\d{1,3}\\.172\\.2[0-9]{1}\\.\\d{1,3}\\.172\\.3[0-1]{1}\\.\\d{1,3}\\.0:0:0:0:1::1</code>	Regular expression that matches proxies that are to be trusted.
server.tomcat.remoteip.port-header	X-Forwarded-Port	Name of the HTTP header used to override the original port value.
server.tomcat.remoteip.protocol-header		Header that holds the incoming protocol, usually named "X-Forwarded-Proto".

Key	Default Value	Description
<code>server.tomcat.remoteip.protocol-header-https-value</code>	<code>https</code>	Value of the protocol header indicating whether the incoming request uses SSL.
<code>server.tomcat.remoteip.remote-ip-header</code>		Name of the HTTP header from which the remote IP is extracted. For instance, `X-FORWARDED-FOR`.
<code>server.tomcat.resource.allow-caching</code>	<code>true</code>	Whether static resource caching is permitted for this web application.
<code>server.tomcat.resource.cache-ttl</code>		Time-to-live of the static resource cache.
<code>server.tomcat.threads.max</code>	<code>200</code>	Maximum amount of worker threads.
<code>server.tomcat.threads.min-spare</code>	<code>10</code>	Minimum amount of worker threads.
<code>server.tomcat.uri-encoding</code>	<code>UTF-8</code>	Character encoding to use to decode the URI.

Key	Default Value	Description
server.tomcat.use-relative-redirects	false	Whether HTTP 1.1 and later location headers generated by a call to sendRedirect will use relative or absolute redirects.
server.undertow.accesslog.dir		Undertow access log directory.
server.undertow.accesslog.enabled	false	Whether to enable the access log.
server.undertow.accesslog.pattern	common	Format pattern for access logs.
server.undertow.accesslog.prefix	access_log.	Log file name prefix.
server.undertow.accesslog.rotate	true	Whether to enable access log rotation.
server.undertow.accesslog.suffix	log	Log file name suffix.

Key	Default Value	Description
<code>server.undertow.allow-encoded-slash</code>	<code>false</code>	<p>Whether the server should decode percent encoded slash characters.</p> <p>Enabling encoded slashes can have security implications due to different servers interpreting the slash differently. Only enable this if you have a legacy application that requires it.</p>
<code>server.undertow.always-set-keep-alive</code>	<code>true</code>	<p>Whether the 'Connection: keep-alive' header should be added to all responses, even if not required by the HTTP specification.</p>

Key	Default Value	Description
<code>server.undertow.buffer-size</code>		Size of each buffer. The default is derived from the maximum amount of memory that is available to the JVM.
<code>server.undertow.decode-url</code>	true	Whether the URL should be decoded. When disabled, percent-encoded characters in the URL will be left as-is.
<code>server.undertow.direct-buffers</code>		Whether to allocate buffers outside the Java heap. The default is derived from the maximum amount of memory that is available to the JVM.
<code>server.undertow.eager-filter-init</code>	true	Whether servlet filters should be initialized on startup.

Key	Default Value	Description
<code>server.undertow.max-cookies</code>	200	Maximum number of cookies that are allowed. This limit exists to prevent hash collision based DOS attacks.
<code>server.undertow.max-headers</code>		Maximum number of headers that are allowed. This limit exists to prevent hash collision based DOS attacks.
<code>server.undertow.max-http-post-size</code>	-1B	Maximum size of the HTTP post content. When the value is -1, the default, the size is unlimited.
<code>server.undertow.max-parameters</code>		Maximum number of query or path parameters that are allowed. This limit exists to prevent hash collision based DOS attacks.

Key	Default Value	Description
<code>server.undertow.no-request-timeout</code>		Amount of time a connection can sit idle without processing a request, before it is closed by the server.
<code>server.undertow.options.server.*</code>		
<code>server.undertow.options.socket.*</code>		
<code>server.undertow.preserve-path-on-forward</code>	<code>false</code>	Whether to preserve the path of a request when it is forwarded.
<code>server.undertow.threads.io</code>		Number of I/O threads to create for the worker. The default is derived from the number of available processors.
<code>server.undertow.threads.worker</code>		Number of worker threads. The default is 8 times the number of I/O threads.
<code>server.undertow.url-charset</code>	<code>UTF-8</code>	Charset used to decode URLs.

11.A.12. Security Properties

Key	Default Value	Description
<code>spring.security.filter.dispatcher-types</code>	<code>[async, error, request]</code>	Security filter chain dispatcher types.
<code>spring.security.filter.order</code>	<code>-100</code>	Security filter chain order.
<code>spring.security.oauth2.client.provider.*</code>		OAuth provider details.
<code>spring.security.oauth2.client.registration.*</code>		OAuth client registrations.
<code>spring.security.oauth2.resource-server.jwt.issuer-uri</code>		URI that can either be an OpenID Connect discovery endpoint or an OAuth 2.0 Authorization Server Metadata endpoint defined by RFC 8414.
<code>spring.security.oauth2.resource-server.jwt.jwk-set-uri</code>		JSON Web Key URI to use to verify the JWT token.
<code>spring.security.oauth2.resource-server.jwt.jws-algorithm</code>	<code>RS256</code>	JSON Web Algorithm used for verifying the digital signatures.

Key	Default Value	Description
<code>spring.security.oauth2.resourceserver.jwt.public-key-location</code>		Location of the file containing the public key used to verify a JWT.
<code>spring.security.oauth2.resourceserver.opaqueToken.client-id</code>		Client id used to authenticate with the token introspection endpoint.
<code>spring.security.oauth2.resourceserver.opaqueToken.client-secret</code>		Client secret used to authenticate with the token introspection endpoint.
<code>spring.security.oauth2.resourceserver.opaqueToken.introspection-uri</code>		OAuth 2.0 endpoint through which token introspection is accomplished.
<code>spring.security.saml2.relyingparty.registration.*</code>		SAML2 relying party registrations.
<code>spring.security.user.name</code>	<code>user</code>	Default user name.
<code>spring.security.user.password</code>		Password for the default user name.
<code>spring.security.user.roles</code>		Granted roles for the default user name.

11.A.13. RSocket Properties

Key	Default Value	Description
<code>spring.rsocket.server.address</code>		Network address to which the server should bind.
<code>spring.rsocket.server.fragment-size</code>		Maximum transmission unit. Frames larger than the specified value are fragmented.
<code>spring.rsocket.server.mapping-path</code>		Path under which RSocket handles requests (only works with websocket transport).
<code>spring.rsocket.server.port</code>		Server port.
<code>spring.rsocket.server.ssl.ciphers</code>		
<code>spring.rsocket.server.ssl.client-auth</code>		
<code>spring.rsocket.server.ssl.enabled</code>		
<code>spring.rsocket.server.ssl.enabled-protocols</code>		
<code>spring.rsocket.server.ssl.key-alias</code>		
<code>spring.rsocket.server.ssl.key-password</code>		
<code>spring.rsocket.server.ssl.key-store</code>		
<code>spring.rsocket.server.ssl.key-store-password</code>		

Key	Default Value	Description
spring.rsocket.server.ssl.key-store-provider		
spring.rsocket.server.ssl.key-store-type		
spring.rsocket.server.ssl.protocol		
spring.rsocket.server.ssl.trust-store		
spring.rsocket.server.ssl.trust-store-password		
spring.rsocket.server.ssl.trust-store-provider		
spring.rsocket.server.ssl.trust-store-type		
spring.rsocket.server.transport	tcp	RSocket transport protocol.

11.A.14. Actuator Properties

Key	Default Value	Description
management.auditevents.enabled	true	Whether to enable storage of audit events.
management.cloudfoundry.enabled	true	Whether to enable extended Cloud Foundry actuator endpoints.
management.cloudfoundry.skip-ssl-validation	false	Whether to skip SSL verification for Cloud Foundry actuator endpoint security calls.

Key	Default Value	Description
management.endpoint.auditevents.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.auditevents.enabled	true	Whether to enable the auditevents endpoint.
management.endpoint.beans.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.beans.enabled	true	Whether to enable the beans endpoint.
management.endpoint.caches.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.caches.enabled	true	Whether to enable the caches endpoint.
management.endpoint.conditions.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.conditions.enabled	true	Whether to enable the conditions endpoint.
management.endpoint.configprops.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.configprops.enabled	true	Whether to enable the configprops endpoint.

Key	Default Value	Description
management.endpoint.configprops.keys-to-sanitize	[password, secret, key, token, .*credentials.* , vcap_services, sun.java.command]	Keys that should be sanitized. Keys can be simple strings that the property ends with or regular expressions.
management.endpoint.env.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.env.enabled	true	Whether to enable the env endpoint.
management.endpoint.env.keys-to-sanitize	[password, secret, key, token, .*credentials.* , vcap_services, sun.java.command]	Keys that should be sanitized. Keys can be simple strings that the property ends with or regular expressions.
management.endpoint.flyway.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.flyway.enabled	true	Whether to enable the flyway endpoint.
management.endpoint.health.cache.time-to-live	0ms	Maximum time that a response can be cached.

Key	Default Value	Description
<code>management.endpoint.health.enabled</code>	<code>true</code>	Whether to enable the health endpoint.
<code>management.endpoint.health.group.*</code>		Health endpoint groups.
<code>management.endpoint.health.probes.enabled</code>	<code>false</code>	Whether to enable liveness and readiness probes.
<code>management.endpoint.health.roles</code>		Roles used to determine whether or not a user is authorized to be shown details. When empty, all authenticated users are authorized.
<code>management.endpoint.health.show-components</code>		When to show components. If not specified the 'show-details' setting will be used.
<code>management.endpoint.health.show-details</code>	<code>never</code>	When to show full health details.

Key	Default Value	Description
<code>management.endpoint.health.status.http-mapping.*</code>		Mapping of health statuses to HTTP status codes. By default, registered health statuses map to sensible defaults (for example, UP maps to 200).
<code>management.endpoint.health.status.order</code>	<code>[DOWN, OUT_OF_SERVICE, UP, UNKNOWN]</code>	Comma-separated list of health statuses in order of severity.
<code>management.endpoint.heapdump.cache.time-to-live</code>	<code>0ms</code>	Maximum time that a response can be cached.
<code>management.endpoint.heapdump.enabled</code>	<code>true</code>	Whether to enable the heapdump endpoint.
<code>management.endpoint.httptrace.cache.time-to-live</code>	<code>0ms</code>	Maximum time that a response can be cached.
<code>management.endpoint.httptrace.enabled</code>	<code>true</code>	Whether to enable the httptrace endpoint.
<code>management.endpoint.info.cache.time-to-live</code>	<code>0ms</code>	Maximum time that a response can be cached.
<code>management.endpoint.info.enabled</code>	<code>true</code>	Whether to enable the info endpoint.

Key	Default Value	Description
<code>management.endpoint.integrationgraph.h.cache.time-to-live</code>	0ms	Maximum time that a response can be cached.
<code>management.endpoint.integrationgraph.h.enabled</code>	true	Whether to enable the integrationgraph endpoint.
<code>management.endpoint.jolokia.config.*</code>		Jolokia settings. Refer to the documentation of Jolokia for more details.
<code>management.endpoint.jolokia.enabled</code>	true	Whether to enable the jolokia endpoint.
<code>management.endpoint.liquibase.cache.time-to-live</code>	0ms	Maximum time that a response can be cached.
<code>management.endpoint.liquibase.enabled</code>	true	Whether to enable the liquibase endpoint.
<code>management.endpoint.logfile.cache.time-to-live</code>	0ms	Maximum time that a response can be cached.
<code>management.endpoint.logfile.enabled</code>	true	Whether to enable the logfile endpoint.

Key	Default Value	Description
<code>management.endpoint.logfile.external-file</code>		External Logfile to be accessed. Can be used if the logfile is written by output redirect and not by the logging system itself.
<code>management.endpoint.loggers.cache.ttl-time-to-live</code>	0ms	Maximum time that a response can be cached.
<code>management.endpoint.loggers.enabled</code>	true	Whether to enable the loggers endpoint.
<code>management.endpoint.mappings.cache.ttl-time-to-live</code>	0ms	Maximum time that a response can be cached.
<code>management.endpoint.mappings.enabled</code>	true	Whether to enable the mappings endpoint.
<code>management.endpoint.metrics.cache.ttl-time-to-live</code>	0ms	Maximum time that a response can be cached.
<code>management.endpoint.metrics.enabled</code>	true	Whether to enable the metrics endpoint.
<code>management.endpoint.prometheus.cache.ttl-time-to-live</code>	0ms	Maximum time that a response can be cached.

Key	Default Value	Description
management.endpoint.prometheus.enabled	true	Whether to enable the prometheus endpoint.
management.endpoint.scheduledtasks.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.scheduledtasks.enabled	true	Whether to enable the scheduledtasks endpoint.
management.endpoint.sessions.enabled	true	Whether to enable the sessions endpoint.
management.endpoint.shutdown.enabled	false	Whether to enable the shutdown endpoint.
management.endpoint.startup.enabled	true	Whether to enable the startup endpoint.
management.endpoint.threaddump.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.threaddump.enabled	true	Whether to enable the threaddump endpoint.
management.endpoints.enabled-by-default		Whether to enable or disable all endpoints by default.

Key	Default Value	Description
<code>management.endpoints.jmx.domain</code>	<code>org.springframework.boot</code>	Endpoints JMX domain name. Fallback to 'spring.jmx.default-domain' if set.
<code>management.endpoints.jmx.exposure.exclude</code>		Endpoint IDs that should be excluded or '*' for all.
<code>management.endpoints.jmx.exposure.include</code>	<code>*</code>	Endpoint IDs that should be included or '*' for all.
<code>management.endpoints.jmx.static-names</code>		Additional static properties to append to all ObjectNames of MBeans representing Endpoints.
<code>management.endpoints.migrate-legacy-ids</code>	<code>false</code>	Whether to transparently migrate legacy endpoint IDs.

Key	Default Value	Description
<code>management.endpoints.web.base-path</code>	<code>/actuator</code>	Base path for Web endpoints. Relative to the servlet context path (server.servlet.context-path) or WebFlux base path (spring.webflux.base-path) when the management server is sharing the main server port. Relative to the management server base path (management.server.base-path) when a separate management server port (management.server.port) is configured.
<code>management.endpoints.web.cors.allow-credentials</code>		Whether credentials are supported. When not set, credentials are not supported.

Key	Default Value	Description
<code>management.endpoints.web.cors.allowed-headers</code>		Comma-separated list of headers to allow in a request. '*' allows all headers.
<code>management.endpoints.web.cors.allowed-methods</code>		Comma-separated list of methods to allow. '*' allows all methods. When not set, defaults to GET.
<code>management.endpoints.web.cors.allowed-origins</code>		Comma-separated list of origins to allow. '*' allows all origins. When not set, CORS support is disabled.
<code>management.endpoints.web.cors.exposed-headers</code>		Comma-separated list of headers to include in a response.

Key	Default Value	Description
<code>management.endpoints.web.cors.max-age</code>	<code>1800s</code>	How long the response from a pre-flight request can be cached by clients. If a duration suffix is not specified, seconds will be used.
<code>management.endpoints.web.exposure.exclude</code>		Endpoint IDs that should be excluded or '*' for all.
<code>management.endpoints.web.exposure.include</code>	<code>[health, info]</code>	Endpoint IDs that should be included or '*' for all.
<code>management.endpoints.web.path-mapping.*</code>		Mapping between endpoint IDs and the path that should expose them.
<code>management.health.cassandra.enabled</code>	<code>true</code>	Whether to enable Cassandra health check.
<code>management.health.couchbase.enabled</code>	<code>true</code>	Whether to enable Couchbase health check.
<code>management.health.db.enabled</code>	<code>true</code>	Whether to enable database health check.

Key	Default Value	Description
<code>management.health.db.ignore-routing-data-sources</code>	<code>false</code>	Whether to ignore AbstractRoutingDataSources when creating database health indicators.
<code>management.health.defaults.enabled</code>	<code>true</code>	Whether to enable default health indicators.
<code>management.health.diskspace.enabled</code>	<code>true</code>	Whether to enable disk space health check.
<code>management.health.diskspace.path</code>		Path used to compute the available disk space.
<code>management.health.diskspace.threshold</code>	<code>10MB</code>	Minimum disk space that should be available.
<code>management.health.elasticsearch.enabled</code>	<code>true</code>	Whether to enable Elasticsearch health check.
<code>management.health.influxdb.enabled</code>	<code>true</code>	Whether to enable InfluxDB health check.
<code>management.health.jms.enabled</code>	<code>true</code>	Whether to enable JMS health check.
<code>management.health.ldap.enabled</code>	<code>true</code>	Whether to enable LDAP health check.

Key	Default Value	Description
management.health.livenessstate.enabled	false	Whether to enable Liveness state health check.
management.health.mail.enabled	true	Whether to enable Mail health check.
management.health.mongo.enabled	true	Whether to enable MongoDB health check.
management.health.neo4j.enabled	true	Whether to enable Neo4j health check.
management.health.ping.enabled	true	Whether to enable ping health check.
management.health.rabbit.enabled	true	Whether to enable RabbitMQ health check.
management.health.readinessstate.enabled	false	Whether to enable readiness state health check.
management.health.redis.enabled	true	Whether to enable Redis health check.
management.health.solr.enabled	true	Whether to enable Solr health check.
management.health.status.order	[DOWN, OUT_OF_SERVICE, UP, UNKNOWN]	

Key	Default Value	Description
<code>management.info.build.enabled</code>	<code>true</code>	Whether to enable build info.
<code>management.info.defaults.enabled</code>	<code>true</code>	Whether to enable default info contributors.
<code>management.info.env.enabled</code>	<code>true</code>	Whether to enable environment info.
<code>management.info.git.enabled</code>	<code>true</code>	Whether to enable git info.
<code>management.info.git.mode</code>	<code>simple</code>	Mode to use to expose git information.
<code>management.metrics.distribution.maximum-expected-value.*</code>		Maximum value that meter IDs starting with the specified name are expected to observe. The longest match wins. Values can be specified as a long or as a Duration value (for timer meters, defaulting to ms if no unit specified).

Key	Default Value	Description
<code>management.metrics.distribution.minimum-expected-value.*</code>		Minimum value that meter IDs starting with the specified name are expected to observe. The longest match wins. Values can be specified as a long or as a Duration value (for timer meters, defaulting to ms if no unit specified).

Key	Default Value	Description
<code>management.metrics.distribution.percentiles-histogram.*</code>		Whether meter IDs starting with the specified name should publish percentile histograms. For monitoring systems that support aggregable percentile calculation based on a histogram, this can be set to true. For other systems, this has no effect. The longest match wins, the key `all` can also be used to configure all meters.

Key	Default Value	Description
<code>management.metrics.distribution.percentiles.*</code>		Specific computed non-aggregable percentiles to ship to the backend for meter IDs starting-with the specified name. The longest match wins, the key `all` can also be used to configure all meters.
<code>management.metrics.distribution.slo.*</code>		Specific service-level objective boundaries for meter IDs starting with the specified name. The longest match wins. Counters will be published for each specified boundary. Values can be specified as a long or as a Duration value (for timer meters, defaulting to ms if no unit specified).

Key	Default Value	Description
<code>management.metrics.enable.*</code>		Whether meter IDs starting with the specified name should be enabled. The longest match wins, the key `all` can also be used to configure all meters.
<code>management.metrics.export.appoptics.api-token</code>		AppOptics API token.
<code>management.metrics.export.appoptics.batch-size</code>	500	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.appoptics.connect-timeout</code>	5s	Connection timeout for requests to this backend.
<code>management.metrics.export.appoptics.enabled</code>	true	Whether exporting of metrics to this backend is enabled.

Key	Default Value	Description
<code>management.metrics.export.appoptics.floor-times</code>	<code>false</code>	Whether to ship a floored time, useful when sending measurements from multiple hosts to align them on a given time boundary.
<code>management.metrics.export.appoptics.host-tag</code>	<code>instance</code>	Tag that will be mapped to "@host" when shipping metrics to AppOptics.
<code>management.metrics.export.appoptics.read-timeout</code>	<code>10s</code>	Read timeout for requests to this backend.
<code>management.metrics.export.appoptics.step</code>	<code>1m</code>	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.appoptics.uri</code>	<code>https://api.appoptics.com/v1/measurements</code>	URI to ship metrics to.

Key	Default Value	Description
<code>management.metrics.export.atlas.batch-size</code>	10000	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.atlas.config-refresh-frequency</code>	10s	Frequency for refreshing config settings from the LWC service.
<code>management.metrics.export.atlas.config-time-to-live</code>	150s	Time to live for subscriptions from the LWC service.
<code>management.metrics.export.atlas.config-uri</code>	<code>http://localhost:7101/lwc/api/v1/expressions/local-dev</code>	URI for the Atlas LWC endpoint to retrieve current subscriptions.
<code>management.metrics.export.atlas.connect-timeout</code>	1s	Connection timeout for requests to this backend.
<code>management.metrics.export.atlas.enabled</code>	true	Whether exporting of metrics to this backend is enabled.

Key	Default Value	Description
management.metrics.export.atlas.evaluate-uri	http://localhost:7101/lwc/api/v1/evaluate	URI for the Atlas LWC endpoint to evaluate the data for a subscription.
management.metrics.export.atlas.lwc-enabled	false	Whether to enable streaming to Atlas LWC.
management.metrics.export.atlas.meter-time-to-live	15m	Time to live for meters that do not have any activity. After this period the meter will be considered expired and will not get reported.
management.metrics.export.atlas.num-threads	4	Number of threads to use with the metrics publishing scheduler.
management.metrics.export.atlas.read-timeout	10s	Read timeout for requests to this backend.
management.metrics.export.atlas.step	1m	Step size (i.e. reporting frequency) to use.
management.metrics.export.atlas.uri	http://localhost:7101/api/v1/public	URI of the Atlas server.

Key	Default Value	Description
<code>management.metrics.export.datadog.api-key</code>		Datadog API key.
<code>management.metrics.export.datadog.application-key</code>		Datadog application key. Not strictly required, but improves the Datadog experience by sending meter descriptions, types, and base units to Datadog.
<code>management.metrics.export.datadog.batch-size</code>	10000	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.datadog.connect-timeout</code>	1s	Connection timeout for requests to this backend.

Key	Default Value	Description
<code>management.metrics.export.datadog.d escriptions</code>	<code>true</code>	Whether to publish descriptions metadata to Datadog. Turn this off to minimize the amount of metadata sent.
<code>management.metrics.export.datadog.e nabled</code>	<code>true</code>	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.datadog.h ost-tag</code>	<code>instance</code>	Tag that will be mapped to "host" when shipping metrics to Datadog.
<code>management.metrics.export.datadog.r ead-timeout</code>	<code>10s</code>	Read timeout for requests to this backend.
<code>management.metrics.export.datadog.s tep</code>	<code>1m</code>	Step size (i.e. reporting frequency) to use.

Key	Default Value	Description
<code>management.metrics.export.datadog.uri</code>	<code>https://api.datadoghq.com</code>	URI to ship metrics to. If you need to publish metrics to an internal proxy en-route to Datadog, you can define the location of the proxy with this.
<code>management.metrics.export.defaults.enabled</code>	<code>true</code>	Whether to enable default metrics exporters.
<code>management.metrics.export.dynatrace.api-token</code>		Dynatrace authentication token.
<code>management.metrics.export.dynatrace.batch-size</code>	<code>10000</code>	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.dynatrace.connect-timeout</code>	<code>1s</code>	Connection timeout for requests to this backend.

Key	Default Value	Description
<code>management.metrics.export.dynatrace.device-id</code>		ID of the custom device that is exporting metrics to Dynatrace.
<code>management.metrics.export.dynatrace.enabled</code>	<code>true</code>	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.dynatrace.group</code>		Group for exported metrics. Used to specify custom device group name in the Dynatrace UI.
<code>management.metrics.export.dynatrace.read-timeout</code>	<code>10s</code>	Read timeout for requests to this backend.
<code>management.metrics.export.dynatrace.step</code>	<code>1m</code>	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.dynatrace.technology-type</code>	<code>java</code>	Technology type for exported metrics. Used to group metrics under a logical technology name in the Dynatrace UI.

Key	Default Value	Description
<code>management.metrics.export.dynatrace.uri</code>		URI to ship metrics to. Should be used for SaaS, self managed instances or to en-route through an internal proxy.
<code>management.metrics.export.elastic.auto-create-index</code>	true	Whether to create the index automatically if it does not exist.
<code>management.metrics.export.elastic.batch-size</code>	10000	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.elastic.connect-timeout</code>	1s	Connection timeout for requests to this backend.
<code>management.metrics.export.elastic.enabled</code>	true	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.elastic.host</code>	<code>http://localhost:9200</code>	Host to export metrics to.

Key	Default Value	Description
<code>management.metrics.export.elasticsearch.index</code>	<code>micrometer-metrics</code>	Index to export metrics to.
<code>management.metrics.export.elasticsearch.index-date-format</code>	<code>yyyy-MM</code>	Index date format used for rolling indices. Appended to the index name.
<code>management.metrics.export.elasticsearch.index-date-separator</code>	<code>-</code>	Prefix to separate the index name from the date format used for rolling indices.
<code>management.metrics.export.elasticsearch.password</code>		Login password of the Elastic server.
<code>management.metrics.export.elasticsearch.pipeline</code>		Ingest pipeline name. By default, events are not pre-processed.
<code>management.metrics.export.elasticsearch.read-timeout</code>	<code>10s</code>	Read timeout for requests to this backend.
<code>management.metrics.export.elasticsearch.step</code>	<code>1m</code>	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.elasticsearch.timestamp-field-name</code>	<code>@timestamp</code>	Name of the timestamp field.
<code>management.metrics.export.elasticsearch.username</code>		Login user of the Elastic server.

Key	Default Value	Description
management.metrics.export.ganglia.addressing-mode	multicast	UDP addressing mode, either unicast or multicast.
management.metrics.export.ganglia.duration-units	milliseconds	Base time unit used to report durations.
management.metrics.export.ganglia.enabled	true	Whether exporting of metrics to Ganglia is enabled.
management.metrics.export.ganglia.host	localhost	Host of the Ganglia server to receive exported metrics.
management.metrics.export.ganglia.port	8649	Port of the Ganglia server to receive exported metrics.
management.metrics.export.ganglia.step	1m	Step size (i.e. reporting frequency) to use.
management.metrics.export.ganglia.time-to-live	1	Time to live for metrics on Ganglia. Set the multi-cast Time-To-Live to be one greater than the number of hops (routers) between the hosts.

Key	Default Value	Description
<code>management.metrics.export.graphite.duration-units</code>	<code>milliseconds</code>	Base time unit used to report durations.
<code>management.metrics.export.graphite.enabled</code>	<code>true</code>	Whether exporting of metrics to Graphite is enabled.
<code>management.metrics.export.graphite.graphite-tags-enabled</code>		Whether Graphite tags should be used, as opposed to a hierarchical naming convention. Enabled by default unless "tagsAsPrefix" is set.
<code>management.metrics.export.graphite.host</code>	<code>localhost</code>	Host of the Graphite server to receive exported metrics.
<code>management.metrics.export.graphite.port</code>	<code>2004</code>	Port of the Graphite server to receive exported metrics.
<code>management.metrics.export.graphite.pickled-protocol</code>		Protocol to use while shipping data to Graphite.
<code>management.metrics.export.graphite.rate-units</code>	<code>seconds</code>	Base time unit used to report rates.

Key	Default Value	Description
<code>management.metrics.export.graphite.step</code>	<code>1m</code>	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.graphite.tags-as-prefix</code>	<code>[]</code>	For the hierarchical naming convention, turn the specified tag keys into part of the metric prefix. Ignored if "graphiteTagsEnabled" is true.
<code>management.metrics.export.humio.api-token</code>		Humio API token.
<code>management.metrics.export.humio.batch-size</code>	<code>10000</code>	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.humio.connect-timeout</code>	<code>5s</code>	Connection timeout for requests to this backend.

Key	Default Value	Description
<code>management.metrics.export.humio.enabled</code>	<code>true</code>	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.humio.read-timeout</code>	<code>10s</code>	Read timeout for requests to this backend.
<code>management.metrics.export.humio.step</code>	<code>1m</code>	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.humio.tags.*</code>		Humio tags describing the data source in which metrics will be stored. Humio tags are a distinct concept from Micrometer's tags. Micrometer's tags are used to divide metrics along dimensional boundaries.

Key	Default Value	Description
<code>management.metrics.export.humio.uri</code>	<code>https://cloud.humio.com</code>	URI to ship metrics to. If you need to publish metrics to an internal proxy en-route to Humio, you can define the location of the proxy with this.
<code>management.metrics.export.influx.auto-create-db</code>	<code>true</code>	Whether to create the Influx database if it does not exist before attempting to publish metrics to it.
<code>management.metrics.export.influx.batch-size</code>	<code>10000</code>	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.influx.compressed</code>	<code>true</code>	Whether to enable GZIP compression of metrics batches published to Influx.

Key	Default Value	Description
management.metrics.export.influx.connect-timeout	1s	Connection timeout for requests to this backend.
management.metrics.export.influx.consistency	one	Write consistency for each point.
management.metrics.export.influx.db	mydb	Database to send metrics to.
management.metrics.export.influx.enabled	true	Whether exporting of metrics to this backend is enabled.
management.metrics.export.influx.password		Login password of the Influx server.
management.metrics.export.influx.read-timeout	10s	Read timeout for requests to this backend.
management.metrics.export.influx.retention-duration		Time period for which Influx should retain data in the current database. For instance 7d, check the influx documentation for more details on the duration format.

Key	Default Value	Description
<code>management.metrics.export.influx.retention-policy</code>		Retention policy to use (Influx writes to the DEFAULT retention policy if one is not specified).
<code>management.metrics.export.influx.retention-replication-factor</code>		How many copies of the data are stored in the cluster. Must be 1 for a single node instance.
<code>management.metrics.export.influx.retention-shard-duration</code>		Time range covered by a shard group. For instance 2w, check the influx documentation for more details on the duration format.
<code>management.metrics.export.influx.step</code>	1m	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.influx.uri</code>	<code>http://localhost:8086</code>	URI of the Influx server.
<code>management.metrics.export.influx.user-name</code>		Login user of the Influx server.
<code>management.metrics.export.jmx.domain</code>	<code>metrics</code>	Metrics JMX domain name.

Key	Default Value	Description
<code>management.metrics.export.jmx.enabled</code>	<code>true</code>	Whether exporting of metrics to JMX is enabled.
<code>management.metrics.export.jmx.step</code>	<code>1m</code>	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.kairos.batch-size</code>	<code>10000</code>	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.kairos.connect-timeout</code>	<code>1s</code>	Connection timeout for requests to this backend.
<code>management.metrics.export.kairos.enabled</code>	<code>true</code>	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.kairos.password</code>		Login password of the KairosDB server.
<code>management.metrics.export.kairos.read-timeout</code>	<code>10s</code>	Read timeout for requests to this backend.

Key	Default Value	Description
management.metrics.export.kairos.step	1m	Step size (i.e. reporting frequency) to use.
management.metrics.export.kairos.uri	http://localhost:8080/api/v1/datapoints	URI of the KairosDB server.
management.metrics.export.kairos.user-name		Login user of the KairosDB server.
management.metrics.export.newrelic.account-id		New Relic account ID.
management.metrics.export.newrelic.api-key		New Relic API key.
management.metrics.export.newrelic.batch-size	10000	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.newrelic.client-provider-type		Client provider type to use.
management.metrics.export.newrelic.connect-timeout	1s	Connection timeout for requests to this backend.

Key	Default Value	Description
<code>management.metrics.export.newrelic.enabled</code>	<code>true</code>	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.newrelic.event-type</code>	<code>SpringBootSample</code>	The event type that should be published. This property will be ignored if 'meter-name-event-type-enabled' is set to 'true'.
<code>management.metrics.export.newrelic.meter-name-event-type-enabled</code>	<code>false</code>	Whether to send the meter name as the event type instead of using the 'event-type' configuration property value. Can be set to 'true' if New Relic guidelines are not being followed or event types consistent with previous Spring Boot releases are required.
<code>management.metrics.export.newrelic.read-timeout</code>	<code>10s</code>	Read timeout for requests to this backend.

Key	Default Value	Description
management.metrics.export.newrelic.step	1m	Step size (i.e. reporting frequency) to use.
management.metrics.export.newrelic.uri	https://insights-collector.newrelic.com	URI to ship metrics to.
management.metrics.export.prometheus.descriptions	true	Whether to enable publishing descriptions as part of the scrape payload to Prometheus. Turn this off to minimize the amount of data sent on each scrape.
management.metrics.export.prometheus.enabled	true	Whether exporting of metrics to Prometheus is enabled.
management.metrics.export.prometheus.histogram-flavor	prometheus	Histogram type for backing DistributionSummary and Timer.
management.metrics.export.prometheus.pushgateway.base-url	http://localhost:9091	Base URL for the Pushgateway.
management.metrics.export.prometheus.pushgateway.enabled	false	Enable publishing via a Prometheus Pushgateway.

Key	Default Value	Description
<code>management.metrics.export.prometheus.pushgateway.grouping-key.*</code>		Grouping key for the pushed metrics.
<code>management.metrics.export.prometheus.pushgateway.job</code>		Job identifier for this application instance.
<code>management.metrics.export.prometheus.pushgateway.password</code>		Login password of the Prometheus Pushgateway.
<code>management.metrics.export.prometheus.pushgateway.push-rate</code>	1m	Frequency with which to push metrics.
<code>management.metrics.export.prometheus.pushgateway.shutdown-operation</code>	none	Operation that should be performed on shutdown.
<code>management.metrics.export.prometheus.pushgateway.username</code>		Login user of the Prometheus Pushgateway.
<code>management.metrics.export.prometheus.step</code>	1m	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.signalfx.access-token</code>		SignalFX access token.

Key	Default Value	Description
<code>management.metrics.export.signalfx.batch-size</code>	10000	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.signalfx.connect-timeout</code>	1s	Connection timeout for requests to this backend.
<code>management.metrics.export.signalfx.enabled</code>	true	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.signalfx.read-timeout</code>	10s	Read timeout for requests to this backend.
<code>management.metrics.export.signalfx.source</code>		Uniquely identifies the app instance that is publishing metrics to SignalFx. Defaults to the local host name.
<code>management.metrics.export.signalfx.step</code>	10s	Step size (i.e. reporting frequency) to use.

Key	Default Value	Description
management.metrics.export.signalfx.uri	<code>https://ingest.signdata.signalfx.com</code>	URI to ship metrics to.
management.metrics.export.simple.enabled	true	Whether, in the absence of any other exporter, exporting of metrics to an in-memory backend is enabled.
management.metrics.export.simple.mode	cumulative	Counting mode.
management.metrics.export.simple.step	1m	Step size (i.e. reporting frequency) to use.
management.metrics.export.stackdriver.batch-size	10000	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.stackdriver.connect-timeout	1s	Connection timeout for requests to this backend.
management.metrics.export.stackdriver.enabled	true	Whether exporting of metrics to this backend is enabled.

Key	Default Value	Description
management.metrics.export.stackdriver.project-id		Identifier of the Google Cloud project to monitor.
management.metrics.export.stackdriver.read-timeout	10s	Read timeout for requests to this backend.
management.metrics.export.stackdriver.resource-type	global	Monitored resource type.
management.metrics.export.stackdriver.step	1m	Step size (i.e. reporting frequency) to use.
management.metrics.export.statsd.enabled	true	Whether exporting of metrics to StatsD is enabled.
management.metrics.export.statsd.flavor	datadog	StatsD line protocol to use.
management.metrics.export.statsd.host	localhost	Host of the StatsD server to receive exported metrics.
management.metrics.export.statsd.max-packet-length	1400	Total length of a single payload should be kept within your network's MTU.

Key	Default Value	Description
<code>management.metrics.export.statsd.polling-frequency</code>	10s	How often gauges will be polled. When a gauge is polled, its value is recalculated and if the value has changed (or <code>publishUnchangedMeters</code> is true), it is sent to the StatsD server.
<code>management.metrics.export.statsd.port</code>	8125	Port of the StatsD server to receive exported metrics.
<code>management.metrics.export.statsd.protocol</code>	udp	Protocol of the StatsD server to receive exported metrics.
<code>management.metrics.export.statsd.publish-unchanged-meters</code>	true	Whether to send unchanged meters to the StatsD server.
<code>management.metrics.export.wavefront.api-token</code>		API token used when publishing metrics directly to the Wavefront API host.

Key	Default Value	Description
<code>management.metrics.export.wavefront.batch-size</code>	10000	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.
<code>management.metrics.export.wavefront.enabled</code>	true	Whether exporting of metrics to this backend is enabled.
<code>management.metrics.export.wavefront.global-prefix</code>		Global prefix to separate metrics originating from this app's white box instrumentation from those originating from other Wavefront integrations when viewed in the Wavefront UI.
<code>management.metrics.export.wavefront.sender.flush-interval</code>	1s	
<code>management.metrics.export.wavefront.sender.max-queue-size</code>	50000	
<code>management.metrics.export.wavefront.sender.message-size</code>		

Key	Default Value	Description
<code>management.metrics.export.wavefront.source</code>		Unique identifier for the app instance that is the source of metrics being published to Wavefront. Defaults to the local host name.
<code>management.metrics.export.wavefront.step</code>	1m	Step size (i.e. reporting frequency) to use.
<code>management.metrics.export.wavefront.uri</code>	https://longboard.wavefront.com	URI to ship metrics to.
<code>management.metrics.tags.*</code>		Common tags that are applied to every meter.
<code>management.metrics.use-global-registry</code>	true	Whether auto-configured MeterRegistry implementations should be bound to the global static registry on Metrics. For testing, set this to 'false' to maximize test independence.

Key	Default Value	Description
management.metrics.web.client.max-uri-tags	100	Maximum number of unique URI tag values allowed. After the max number of tag values is reached, metrics with additional tag values are denied by filter.
management.metrics.web.client.request.autotime.enabled	true	Whether to automatically time web client requests.
management.metrics.web.client.request.autotime.percentiles		Computed non-aggregable percentiles to publish.
management.metrics.web.client.request.autotime.percentiles-histogram	false	Whether percentile histograms should be published.
management.metrics.web.client.requests.metric-name	http.client.requests	Name of the metric for sent requests.

Key	Default Value	Description
management.metrics.web.server.max-uri-tags	100	Maximum number of unique URI tag values allowed. After the max number of tag values is reached, metrics with additional tag values are denied by filter.
management.metrics.web.server.request.autotime.enabled	true	Whether to automatically time web server requests.
management.metrics.web.server.request.autotime.percentiles		Computed non-aggregable percentiles to publish.
management.metrics.web.server.request.autotime.percentiles-histogram	false	Whether percentile histograms should be published.
management.metrics.web.server.request.ignore-trailing-slash	true	Whether the trailing slash should be ignored when recording metrics.
management.metrics.web.server.request.metric-name	http.server.requests	Name of the metric for received requests.

Key	Default Value	Description
<code>management.server.add-application-context-header</code>	<code>false</code>	Add the "X-Application-Context" HTTP header in each response.
<code>management.server.address</code>		Network address to which the management endpoints should bind. Requires a custom <code>management.server.port</code> .
<code>management.server.base-path</code>		Management endpoint base path (for instance, `/management`). Requires a custom <code>management.server.port</code> .
<code>management.server.port</code>		Management endpoint HTTP port (uses the same port as the application by default). Configure a different port to use management-specific SSL.
<code>management.server.ssl.ciphers</code>		Supported SSL ciphers.

Key	Default Value	Description
<code>management.server.ssl.client-auth</code>		Client authentication mode. Requires a trust store.
<code>management.server.ssl.enabled</code>	<code>true</code>	Whether to enable SSL support.
<code>management.server.ssl.enabled-protocols</code>		Enabled SSL protocols.
<code>management.server.ssl.key-alias</code>		Alias that identifies the key in the key store.
<code>management.server.ssl.key-password</code>		Password used to access the key in the key store.
<code>management.server.ssl.key-store</code>		Path to the key store that holds the SSL certificate (typically a jks file).
<code>management.server.ssl.key-store-password</code>		Password used to access the key store.
<code>management.server.ssl.key-store-provider</code>		Provider for the key store.
<code>management.server.ssl.key-store-type</code>		Type of the key store.
<code>management.server.ssl.protocol</code>	<code>TLS</code>	SSL protocol to use.

Key	Default Value	Description
<code>management.server.ssl.trust-store</code>		Trust store that holds SSL certificates.
<code>management.server.ssl.trust-store-password</code>		Password used to access the trust store.
<code>management.server.ssl.trust-store-provider</code>		Provider for the trust store.
<code>management.server.ssl.trust-store-type</code>		Type of the trust store.
<code>management.trace.http.enabled</code>	<code>true</code>	Whether to enable HTTP request-response tracing.
<code>management.trace.http.include</code>	<code>[request-headers, response-headers, errors]</code>	Items to be included in the trace. Defaults to request headers (excluding Authorization and Cookie), response headers (excluding Set-Cookie), and time taken.

11.A.15. Devtools Properties

Key	Default Value	Description
<code>spring.devtools.add-properties</code>	<code>true</code>	Whether to enable development property defaults.
<code>spring.devtools.livereload.enabled</code>	<code>true</code>	Whether to enable a livereload.com-compatible server.
<code>spring.devtools.livereload.port</code>	<code>35729</code>	Server port.
<code>spring.devtools.remote.context-path</code>	<code>/ .~spring-boot!~</code>	Context path used to handle the remote connection.
<code>spring.devtools.remote.proxy.host</code>		The host of the proxy to use to connect to the remote application.
<code>spring.devtools.remote.proxy.port</code>		The port of the proxy to use to connect to the remote application.
<code>spring.devtools.remote.restart.enabled</code>	<code>true</code>	Whether to enable remote restart.
<code>spring.devtools.remote.secret</code>		A shared secret required to establish a connection (required to enable remote support).

Key	Default Value	Description
<code>spring.devtools.remote.secret-header-name</code>	X-AUTH-TOKEN	HTTP header used to transfer the shared secret.
<code>spring.devtools.restart.additional-exclude</code>		Additional patterns that should be excluded from triggering a full restart.
<code>spring.devtools.restart.additional-paths</code>		Additional paths to watch for changes.
<code>spring.devtools.restart.enabled</code>	true	Whether to enable automatic restart.
<code>spring.devtools.restart.exclude</code>	META-INF/maven/**,META-INF/resources/**,resources/**,static/**,public/**,templates/**,**/*Test.class,**/*Test.class,git.properties,META-INF/build-info.properties	Patterns that should be excluded from triggering a full restart.
<code>spring.devtools.restart.log-condition-evaluation-delta</code>	true	Whether to log the condition evaluation delta upon restart.

Key	Default Value	Description
<code>spring.devtools.restart.poll-interval</code>	<code>1s</code>	Amount of time to wait between polling for classpath changes.
<code>spring.devtools.restart.quiet-period</code>	<code>400ms</code>	Amount of quiet time required without any classpath changes before a restart is triggered.
<code>spring.devtools.restart.trigger-file</code>		Name of a specific file that, when changed, triggers the restart check. Must be a simple name (without any path) of a file that appears on your classpath. If not specified, any classpath file change triggers the restart.

11.A.16. Testing Properties

Key	Default Value	Description
spring.test.database.replace	any	Type of existing DataSource to replace.
spring.test.mockmvc.print	default	MVC Print option.

Appendix 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. 这些文件旨在允许 IDE 开发人员在用户使用 `application.properties` 或 `application.yml` 文件时提供上下文帮助和"代码完成" .

主要的元数据文件是在编译器通过处理所有被 `@ConfigurationProperties` 注解的节点来自动生成的. 但是, 对于极端情况或更高级的用例, 可以[手动编写部分元数据](#) .

11.B.1. Metadata 格式

配置元数据文件位于 `jars` 文件中的 `META-INF/spring-configuration-metadata.json` . 它们使用一个具有 "groups" 或 "properties" 分类节点的简单 JSON 格式, 并将其他值提示归类为 "hints", 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",
      "group": "server"
    }
  ]
}
```

```
"name": "server.port",
"type": "java.lang.Integer",
"sourceType":
"org.springframework.boot.autoconfigure.web.ServerProperties"
},
{
"name": "server.address",
"type": "java.net.InetAddress",
"sourceType":
"org.springframework.boot.autoconfigure.web.ServerProperties"
},
{
"name": "spring.jpa.hibernate.ddl-auto",
"type": "java.lang.String",
"description": "DDL mode. This is actually a shortcut for the
\"hibernate.hbm2ddl.auto\" property.",
"sourceType":
"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"
}
...
],"hints": [
{
"name": "spring.jpa.hibernate.ddl-auto",
"values": [
{
"value": "none",
"description": "Disable DDL handling."
},
{
"value": "validate",
"description": "Validate the schema, make no changes to the
database."
},
{
"value": "update",
"description": "Update the schema if necessary."
},
{
"value": "create",
"description": "Create the schema and destroy previous data."
},
{
"value": "create-drop",
"description": "Create and then destroy the schema at the end of
the session."
}
]
}
```

]}

每个 “property” 都是用户使用给定值指定的配置项. 例如,可以在 `application.properties` 中指定 `server.port` 和 `server.address`,如下所示:

```
server.port=9090
server.address=127.0.0.1
```

“groups” 是高级别的节点,它们本身不指定一个值,但为 `properties` 提供一个有上下文关联的分组. 例如, `server.port` 和 `server.address` 属性是 `server` 组的一部分.



不需要每个 “property” 都有一个 “group”.
一些属性可以以自己的形式存在.

最后, “hints” 是用于帮助用户配置给定属性的其他信息. 例如,当开发人员配置 `spring.jpa.hibernate.ddl-auto` 属性时,工具可以使用提示 `none`, `validate`, `update`, `create` 和 `create-drop` 值 .

Group 属性

`groups` 数组包含的 JSON 对象可以由以下属性组成:

名称	类型	目的
<code>name</code>	<code>String</code>	组的全名. 此属性是必需的.
<code>type</code>	<code>String</code>	组数据类型的类名. <code>group</code> 数据类型的类名. 例如,如果 <code>group</code> 是基于一个被 <code>@ConfigurationProperties</code> 注解的类,该属性将包含该类的全限定名. 如果基于一个 <code>@Bean</code> 方法,它将是该方法的返回类型. 如果该类型未知,则该属性将被忽略
<code>description</code>	<code>String</code>	一个简短的 <code>group</code> 描述,用于展示给用户.如果没有可用描述,该属性将被忽略.推荐使用一个简短的段落描述,第一行提供一个简洁的总结,最后一行以句号结尾

名称	类型	目的
<code>sourceType</code>	<code>String</code>	贡献该组的来源类名.例如,如果组基于一个被 <code>@ConfigurationProperties</code> 注解的 <code>@Bean</code> 方法,该属性将包含 <code>@Configuration</code> 类的全限定名,该类包含此方法.如果来源类型未知,则该属性将被忽略
<code>sourceMethod</code>	<code>String</code>	贡献该组的方法的全名(包含括号及参数类型).例如,被 <code>@ConfigurationProperties</code> 注解的 <code>@Bean</code> 方法名.如果源方法未知,该属性将被忽略

Property 属性

`properties` 数组中包含的JSON对象可由以下属性构成:

名称	类型	目的
<code>name</code>	<code>String</code>	属性的全名.名称以小写的句点分隔(例如, <code>server.address</code>) .此属性是必需的.
<code>type</code>	<code>String</code>	<code>property</code> 数据类型的类名(例如, <code>java.lang.String</code>),还具有完整的泛型类型(例如, <code>java.util.Map<java.lang.String,acme.MyEnum></code>) . 该属性可以用来指导用户他们可以输入值的类型.为了保持一致,原生类型使用它们的包装类代替(例如, <code>boolean</code> 变为 <code>java.lang.Boolean</code>) .注意,这个类可能是个从一个字符串转换而来的复杂类型.如果类型未知则该属性会被忽略
<code>description</code>	<code>String</code>	一个简短的组的描述,用于展示给用户.如果没有描述可用则该属性会被忽略.推荐使用一个简短的段落描述,开头提供一个简洁的总结,最后一行以句号结束
<code>sourceType</code>	<code>String</code>	贡献 <code>property</code> 的来源类名.例如,如果 <code>property</code> 来自一个被 <code>@ConfigurationProperties</code> 注解的类,该属性将包括该类的全限定名.如果来源类型未知则该属性会被忽略
<code>defaultValue</code>	<code>Object</code>	当 <code>property</code> 没有定义时使用的默认值.如果 <code>property</code> 类型是个数组则该属性也可以是个数组.如果默认值未知则该属性会被忽略

名称	类型	目的
<code>deprecation</code>	<code>Deprecation</code>	<p>指定该 <code>property</code> 是否过期 .如果该字段没有过期或该信息未知则该属性会被忽略. 下表提供了有关 <code>deprecation</code> 属性的更多详细信息.</p>

每个 `properties` 元素的 `deprecation` 属性中包含的JSON对象可以包含以下属性：

名称	类型	目的
<code>level</code>	<code>String</code>	<p>弃用级别,可以是警告(默认) 或错误. 当某个属性具有警告弃用级别时,它仍应绑定在环境中. 但是 ,当它具有错误弃用级别时,该属性将不再受管理且未绑定.</p>
<code>reason</code>	<code>String</code>	<p>简短描述了该资源被弃用的原因.如果没有理由可以省略.建议 描述是一个简短的段落,第一行提供简明扼要的摘要.说明中的 最后一行应以(.) 结尾.</p>
<code>replacement</code>	<code>String</code>	<p>正在替换此不推荐使用的属性的属性的全名.如果没有替换此属性, 可以省略.</p>



在 Spring Boot 1.3 之前,可以使用单个 `deprecated`

使用的布尔属性来代替 `deprecation` 元素.

这仍然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` 应该设置一个空的对象.

也可以在代码中以声明方式指定弃用,方法是将 `@DeprecatedConfigurationProperty` 注解添加到暴露弃用属性的 `getter` 中. 例如,假设 `app.acme.target` 属性令人困惑,并将其重命名为 `app.acme.name`. 以下示例显示了如何处理这种情况:

```

@ConfigurationProperties("app.acme")
public class AcmeProperties {

    private String name;

    public String getName() { ... }

    public void setName(String name) { ... }

    @DeprecatedConfigurationProperty(replacement = "app.acme.name")
    @Deprecated
    public String getTarget() {
        return getName();
    }

    @Deprecated
    public void setTarget(String target) {
        setName(target);
    }
}

```



无法设置级别。由于代码仍在处理该属性，因此始终 **warning**。

前面的代码确保不推荐使用的属性仍然有效（将其委托给幕后的 `name` 属性）。一旦可以从公共 API 中删除 `getTarget` 和 `setTarget` 方法，元数据中的自动弃用提示也将消失。如果要保留提示，请添加具有错误弃用级别的手动元数据，以确保仍然向用户通知该属性。进行替换时，这样做特别有用。

Hint 属性

`hints` 数组中包含的 JSON 对象可以包含以下属性：

名称	类型	目的
<code>name</code>	<code>String</code>	该提示所引用的属性的全名。名称采用小写的句点分隔形式（例如 <code>spring.mvc.servlet.path</code> ）。如果属性引用映射（例如 <code>system.contexts</code> ），则提示将应用于映射的键（ <code>system.contexts.keys</code> ）或映射的值（ <code>system.contexts.values</code> ）。此属性是必需的。

名称	类型	目的
values	<code>ValueHint[]</code>	由 <code>ValueHint</code> 对象定义的有效值列表(如下表所述) . 每个条目都定义该值,并且可以具有描述.
providers	<code>ValueProvider[]</code>	由 <code>ValueProvider</code> 对象定义的提供者列表 (在本文档的后面介绍) . 每个条目定义提供者的名称及其参数(如果有) .

每个 `hint` 元素的 `values` 属性中包含的 JSON 对象可以包含下表中描述的属性:

名称	类型	目的
value	<code>Object</code>	提示所引用元素的有效值. 如果属性的类型是数组 ,则它也可以是值的数组. 此属性是必需的.
description	<code>String</code>	可以显示给用户的值的简短描述. 如果没有可用的描述 ,则可以省略. 建议使用简短的描述,第一行提供简要的摘要. 说明中的最后一行应以句点(.) 结尾.

每个 `hint` 元素的 `providers` 属性中包含的 JSON 对象可以包含下表中描述的属性:

名称	类型	目的
name	<code>String</code>	用于为提示所引用的元素提供附加内容帮助的提供者的名称.
parameters	<code>JSON object</code>	<code>provider</code> 支持的任何其他参数(有关更多详细信息,请参阅 <code>provider</code> 的文档) .

重复的元数据项

具有相同 “`property`” 和 “`group`” 名称的对象可以在元数据文件中多次出现. 例如 ,您可以将两个单独的类绑定到同一前缀,每个类具有可能重叠的属性名称.

虽然相同的名称多次出现在元数据中应该不常见,但元数据的使用者应注意确保它们支持该名称.

11.B.2. 提供手动提示

为了改善用户体验并进一步帮助用户配置给定属性,您可以提供其他元数据,这些元数据可以:

- 描述属性的潜在值列表.

- 关联提供者, 以将定义良好的语义附加到属性
, 以便工具可以根据项目的上下文来发现潜在值的列表.

Value Hint

每个提示的 `name` 属性是指属性的名称. 在[前面显示的初始示例中](#), 我们为 `spring.jpa.hibernate.ddl-auto` 属性提供了五个值: `none`, `validate`, `update`, `create`, 和 `create-drop`. 每个值也可以具有描述.

如果您的属性属于 `Map` 类型, 则可以提供键和值的提示(但不提供 `map` 本身的提示) . 特殊的 `.keys` 和 `.values` 后缀必须分别引用键和值.

假设有一个 `sample.contexts` 的 `Map<String, Integer>`, 如以下示例所示:

```
@ConfigurationProperties("sample")
public class SampleProperties {

    private Map<String, Integer> contexts;
    // getters and setters
}
```

`String` (在此示例中) 为 `sample1` 和 `sample2`. 为了为 `key` 提供其他内容提示 , 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"
            }
        ]
    }
]}
```



我们建议您对这两个值使用枚举。如果您的 IDE 支持，这是迄今为止最有效的自动完成方法。

Value Providers

Providers 是一种将语义附加到属性的强有力的方法，我们将定义可以用于您自己的提示的官方 Providers。但是，您最喜欢的 IDE 可能只实现其中一些，也可能没有实现。



由于这是一项新功能，IDE 供应商必须赶上它的工作方式。

采用时间自然会有所不同。下表总结了受支持的 provider 的列表：

名字	描述
<code>any</code>	允许提供任何附加值。
<code>class-reference</code>	自动完成项目中可用的类。通常受 <code>target</code> 参数指定的基类的约束。
<code>handle-as</code>	如同按强制 <code>target</code> 参数定义的类型定义属性一样处理属性。
<code>logger-name</code>	自动完成有效的记录器名称和 记录器组 。通常，可以自动完成当前项目中可用的包和类名以及定义的组。.
<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> 参数指定的基类的约束。
<code>spring-profile-name</code>	自动完成项目中可用的 Spring profile 名称。



对于给定的属性，只有一个 provider 可以处于 active 状态，但是如果它们都可以通过某种方式管理该属性，则可以指定多个 provider。确保将最有用的 provider 放在首位，因为 IDE 必须使用它可以处理的JSON部分中的第一个。如果不支持给定属性的 provider，则也不提供特殊的内容帮助。

Any

这个特殊的 provider 允许提供任何其他值。如果支持，则应基于属性类型进行常规值验证。

如果您具有值列表，并且任何其他值应视为有效，则通常使用此 provider。

以下示例提供了 `system.state` 的自动完成值的 `on` 和 `off`:

```
{"hints": [
  {
    "name": "system.state",
    "values": [
      {
        "value": "on"
      },
      {
        "value": "off"
      }
    ],
    "providers": [
      {
        "name": "any"
      }
    ]
  }
]}
```

注意，在前面的示例中，还允许任何其他值。

Class 引用

类引用 provider 自动完成项目中可用的类。此 provider 支持以下参数：

参数	类型	默认值	描述
<code>target</code>	<code>String</code> (<code>Class</code>)	<code>none</code>	应分配给所选值的类的完全限定名称。 通常用于过滤掉非候选类。请注意 , 可以通过暴露具有适当上限的类来由类型本身 提供此信息。
<code>concrete</code>	<code>boolean</code>	<code>true</code>	指定是否仅将具体类视为有效候选者。

以下元数据片段对应于标准 `server.servlet.jsp.class-name` 属性

，该属性定义了要使用的 `JspServlet` 类名称：

```
{"hints": [
  {
    "name": "server.servlet.jsp.class-name",
    "providers": [
      {
        "name": "class-reference",
        "parameters": {
          "target": "javax.servlet.http.HttpServlet"
        }
      }
    ]
  }
]}
```

Handle As

handle-as provider 使您可以将属性的类型替换为更高级的类型。当该属性具有 `java.lang.String` 类型时，通常会发生这种情况，因为您不希望配置类依赖于可能不在类路径中的类。此 `provider` 支持以下参数：

参数	类型	默认值	描述
target	<code>String</code> (<code>Class</code>)	<code>none</code>	要为属性考虑的类型的标准名称。 此参数是必需的。

可以使用以下类型：

- 任何 `java.lang.Enum`: 列出属性的可能值。(我们建议使用 `Enum` 类型定义属性，因为IDE不需要其他提示即可自动完成值)
- `java.nio.charset.Charset`: 支持字符集/编码值(例如 `UTF-8`) 的自动完成
- `java.util.Locale`: 语言环境的自动完成(例如 `en_US`)
- `org.springframework.util.MimeType`: 支持内容类型值(例如 `text/plain`) 的自动完成
- `org.springframework.core.io.Resource`: 支持自动完成 Spring 资源抽象以引用文件系统或类路径上的文件(例如 `classpath:/sample.properties`)



如果可以提供多个值，请使用 `Collection` 或 `Array` 类型向IDE讲解。

以下元数据片段对应于标准 `spring.liquibase.change-log` 属性
, 该属性定义了要使用的更改日志的路径. 实际上, 它在内部用作
`org.springframework.core.io.Resource`, 但不能这样暴露, 因为我们需要保留原始的String值以将其传递给Liquibase API.

```
{"hints": [
  {
    "name": "spring.liquibase.change-log",
    "providers": [
      {
        "name": "handle-as",
        "parameters": {
          "target": "org.springframework.core.io.Resource"
        }
      }
    ]
  }
]}
```

Logger 名

`logger-name provider` 会自动完成有效的记录器名称和 [记录器组](#). 通常
, 可以自动完成当前项目中可用的程序包和类名. 如果启用了组(默认)
, 并且在配置中标识了自定义记录程序组, 则应为其提供自动完成功能.
特定的框架可能还具有其他可以支持的魔法值记录器名称.

此 `provider` 支持以下参数:

参数	类型	默认值	描述
<code>group</code>	<code>boolean</code>	<code>true</code>	指定是否应考虑已知组.

由于记录器名称可以是任意名称, 因此该 `provider` 应允许使用任何值
, 但可以突出显示项目的类路径中不可用的有效程序包和类名称.

以下元数据片段对应于标准 `logging.level` 属性. 键是记录器名称
, 其值对应于标准日志级别或任何自定义级别. 当 Spring Boot
开箱即用地定义了一些记录器组时, 已经为它们添加了专用的值提示.

```
{"hints": [
```

```
{  
    "name": "logging.level.keys",  
    "values": [  
        {  
            "value": "root",  
            "description": "Root logger used to assign the default logging  
level."  
        },  
        {  
            "value": "sql",  
            "description": "SQL logging group including Hibernate SQL  
logger."  
        },  
        {  
            "value": "web",  
            "description": "Web logging group including codecs."  
        }  
    ],  
    "providers": [  
        {  
            "name": "logger-name"  
        }  
    ]  
},  
{  
    "name": "logging.level.values",  
    "values": [  
        {  
            "value": "trace"  
        },  
        {  
            "value": "debug"  
        },  
        {  
            "value": "info"  
        },  
        {  
            "value": "warn"  
        },  
        {  
            "value": "error"  
        },  
        {  
            "value": "fatal"  
        },  
        {  
            "value": "off"  
        }  
    ]  
}
```

```
],
"providers": [
{
  "name": "any"
}
]
}
]}
```

Spring Bean 引用

spring-bean-reference provider 自动完成在当前项目的配置中定义的 bean. 此 provider 支持以下参数:

参数	类型	默认值	描述
target	String (Class)	<i>none</i>	应分配给候选者的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标准 **spring.jmx.server** 属性, 该属性定义了要使用的 **MBeanServer** bean 的名称:

```
{"hints": [
{
  "name": "spring.jmx.server",
  "providers": [
    {
      "name": "spring-bean-reference",
      "parameters": {
        "target": "javax.management.MBeanServer"
      }
    }
  ]
}
]}
```



binder 不会自动装配这些元数据, 如果提供了该提示, 则仍需要使用 **ApplicationContext** 将 Bean 名称转换为实际的 Bean 引用. .

Spring Profile 名

`spring-profile-name provider` 自动完成在当前项目的配置中定义的 Spring profile.

以下元数据片段对应于标准 `spring.profiles.active` 属性, 该属性定义了要启用的 Spring profile 的名称:

```
{"hints": [
  {
    "name": "spring.profiles.active",
    "providers": [
      {
        "name": "spring-profile-name"
      }
    ]
  }
]}
```

11.B.3. 使用注解处理器生成您自己的元数据

您可以使用 `spring-boot-configuration-processor jar` 从带有 `@ConfigurationProperties` 注解的项目中轻松生成自己的配置元数据文件. 该 jar 包含一个 Java 注解处理器, 在您的项目被编译时会被调用.

配置注解处理器

要使用处理器, 请添加 `spring-boot-configuration-processor` 的依赖.

使用 Maven, 依赖应声明为可选, 如以下示例所示:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>
```

使用 Gradle, 应该在 `annotationProcessor` 配置中声明依赖, 如以下示例所示:

```
dependencies {
    compileOnly "org.springframework.boot:spring-boot-configuration-processor"
}
```

对于 Gradle 4.6 和更高版本,应在 `annotationProcessor` 配置中声明依赖,如以下示例所示:

```
dependencies {
    annotationProcessor "org.springframework.boot:spring-boot-configuration-
processor"
}
```

如果您使用的是额外的 `spring-configuration-metadata.json` 文件,则应将 `compileJava` 任务配置为依赖于 `processResources` 任务,如以下示例所示:

```
compileJava.inputs.files(processResources)
```

这种依赖确保注解处理器在编译期间运行时,其他元数据可用.

如果在项目中使用 AspectJ,则需要确保注解处理器只运行一次,有很多方法可以实现这一点.

在 Maven 中,你可以配置 `maven-apt-plugin` 并将依赖只添加到注解处理器中. 您还可以让 AspectJ 插件在 `maven-compiler-plugin` 配置中执行所有处理时禁用注解处理,如下所示



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <proc>none</proc>
    </configuration>
</plugin>
```

自动生成元数据

处理器选择用 `@ConfigurationProperties` 注解的类和方法.

如果该类也使用了 `@ConstructorBinding` 注解，则应使用单个构造函数，并为每个构造函数参数创建一个属性。否则，将通过存在标准的 `getter` 和 `setter` 并对集合和 `Map` 类型进行特殊处理来发现属性（即使仅存在 `getter` 也会被检测到）。注解处理器还支持使用 `@Data`, `@Getter` 和 `@Setter` `lombok` 注解。

考虑以下类：

```
@ConfigurationProperties(prefix="server")
public class ServerProperties {

    /**
     * Name of the server.
     */
    private String name;

    /**
     * IP address to listen to.
     */
    private String ip = "127.0.0.1";

    /**
     * Port to listener to.
     */
    private int port = 9797;

    // ... getter and setters

}
```

这暴露了三个属性，其中 `server.name` 没有默认值，`server.ip` 和 `server.port` 分别默认为 `"127.0.0.1"` 和 `9797`。字段值的 Javadoc 用于填充 `description` 属性.. 例如，`server.ip` 的描述是 `"IP address to listen to"`。



您仅应将简单文本与 `@ConfigurationProperties` 字段 Javadoc 一起使用，因为在将它们添加到 JSON 之前不会对其进行处理。

注解处理器应用多种启发式方法从源模型中提取默认值。必须静态提供默认值。

特别是不要引用另一个类中定义的常量。另外，注解处理器无法自动检测 `Enum` 和 `Collections` 的默认值。

如果该类具有一个带有至少一个参数的构造函数，则为每个构造函数参数创建一个属性。否则

,将通过存在标准的 `getter` 和 `setter` 并对集合类型进行特殊处理来发现属性(即使仅存在`getter`也会被检测到) .

对于无法检测到默认值的情况,应提供[手动元数据](#).

考慮以下类:

```
@ConfigurationProperties(prefix = "acme.messaging")
public class MessagingProperties {

    private List<String> addresses = new ArrayList<>(Arrays.asList("a", "b"));

    private ContainerType containerType = ContainerType.SIMPLE;

    // ... getter and setters

    public enum ContainerType {

        SIMPLE,
        DIRECT

    }

}
```

为了记录以上类中属性的默认值,您可以将以下内容[添加到模块的手动元数据](#):

```
{"properties": [
    {
        "name": "acme.messaging.addresses",
        "defaultValue": ["a", "b"]
    },
    {
        "name": "acme.messaging.container-type",
        "defaultValue": "simple"
    }
]}
```



只需要属性 `name` 即可记录带有手动元数据的其他字段.

嵌套属性

注解处理器自动将内部类视为嵌套属性。我们可以为它创建一个子命名空间，而不是在根命名空间的记录 `ip` 和 `port`。考虑更新后的示例：考虑以下类：

```
@ConfigurationProperties(prefix="server")
public class ServerProperties {

    private String name;

    private Host host;

    // ... getter and setters

    public static class Host {

        private String ip;

        private int port;

        // ... getter and setters

    }

}
```

前面的示例为 `server.name`, `server.host.ip` 和 `server.host.port` 属性生成元数据信息。您可以在字段上使用 `@NestedConfigurationProperty` 注解，以指示应将常规(非内部)类视为嵌套类。



这对集合和地图没有影响，因为这些类型会自动识别，并且会为每个集合生成一个元数据属性。

添加其他元数据

Spring Boot 的配置文件处理非常灵活，通常情况下可能存在未绑定到 `@ConfigurationProperties` bean 的属性。您可能还需要调整现有键的某些属性。为了支持这种情况，并允许您提供自定义的 "hints"，注解处理器会自动将 `META-INF/additional-spring-configuration-metadata.json` 中的项目合并到主元数据文件中。

如果引用了已自动检测到的属性，则如果指定了描述，默认值和弃用信息，则它们将被覆盖。
如果在当前模块中未标识手动属性声明，则将其添加为新属性。

另外，`spring-configuration-metadata.json` 文件的格式与常规的 `spring-configuration-metadata.json` 完全相同。附加属性文件是可选的。
如果没有任何其他属性，请不要添加文件。

Appendix C: 自动配置类

本附录包含 Spring Boot 提供的所有自动配置类的详细信息，以及指向文档源代码的链接。
请记住，还要查看应用程序中的 `conditions` 报告以了解有关哪些功能已打开的更多详细信息。
(为此，请使用 `--debug` 或 `-Ddebug` 启动应用程序，或者在 `Actuator` 应用程序中使用 `conditions` 端点)。

11.C.1. spring-boot-autoconfigure

以下自动配置类来自 `spring-boot-autoconfigure` 模块：

Configuration Class	Links
<code>ActiveMQAutoConfiguration</code>	javadoc
<code>AopAutoConfiguration</code>	javadoc
<code>ApplicationAvailabilityAutoConfiguration</code>	javadoc
<code>ArtemisAutoConfiguration</code>	javadoc
<code>BatchAutoConfiguration</code>	javadoc
<code>CacheAutoConfiguration</code>	javadoc
<code>CassandraAutoConfiguration</code>	javadoc
<code>CassandraDataAutoConfiguration</code>	javadoc
<code>CassandraReactiveDataAutoConfiguration</code>	javadoc
<code>CassandraReactiveRepositoriesAutoConfiguration</code>	javadoc
<code>CassandraRepositoriesAutoConfiguration</code>	javadoc
<code>ClientHttpConnectorAutoConfiguration</code>	javadoc
<code>CodecsAutoConfiguration</code>	javadoc

Configuration Class	Links
ConfigurationPropertiesAutoConfiguration	javadoc
CouchbaseAutoConfiguration	javadoc
CouchbaseDataAutoConfiguration	javadoc
CouchbaseReactiveDataAutoConfiguration	javadoc
CouchbaseReactiveRepositoriesAutoConfiguration	javadoc
CouchbaseRepositoriesAutoConfiguration	javadoc
DataSourceAutoConfiguration	javadoc
DataSourceTransactionManagerAutoConfiguration	javadoc
DispatcherServletAutoConfiguration	javadoc
ElasticsearchDataAutoConfiguration	javadoc
ElasticsearchRepositoriesAutoConfiguration	javadoc
ElasticsearchRestClientAutoConfiguration	javadoc
EmbeddedLdapAutoConfiguration	javadoc
EmbeddedMongoAutoConfiguration	javadoc
EmbeddedWebServerFactoryCustomizerAutoConfiguration	javadoc
ErrorMvcAutoConfiguration	javadoc
ErrorWebFluxAutoConfiguration	javadoc
FlywayAutoConfiguration	javadoc
FreeMarkerAutoConfiguration	javadoc
GroovyTemplateAutoConfiguration	javadoc
GsonAutoConfiguration	javadoc
H2ConsoleAutoConfiguration	javadoc
HazelcastAutoConfiguration	javadoc
HazelcastJpaDependencyAutoConfiguration	javadoc
HibernateJpaAutoConfiguration	javadoc
HttpEncodingAutoConfiguration	javadoc

Configuration Class	Links
HttpHandlerAutoConfiguration	javadoc
HttpMessageConvertersAutoConfiguration	javadoc
HypermediaAutoConfiguration	javadoc
InfluxDbAutoConfiguration	javadoc
IntegrationAutoConfiguration	javadoc
JacksonAutoConfiguration	javadoc
JdbcRepositoriesAutoConfiguration	javadoc
JdbcTemplateAutoConfiguration	javadoc
JerseyAutoConfiguration	javadoc
JmsAutoConfiguration	javadoc
JmxAutoConfiguration	javadoc
JndiConnectionFactoryAutoConfiguration	javadoc
JndiDataSourceAutoConfiguration	javadoc
JooqAutoConfiguration	javadoc
JpaRepositoriesAutoConfiguration	javadoc
JsonbAutoConfiguration	javadoc
JtaAutoConfiguration	javadoc
KafkaAutoConfiguration	javadoc
LdapAutoConfiguration	javadoc
LdapRepositoriesAutoConfiguration	javadoc
LifecycleAutoConfiguration	javadoc
liquibaseAutoConfiguration	javadoc
MailSenderAutoConfiguration	javadoc
MailSenderValidatorAutoConfiguration	javadoc
MessageSourceAutoConfiguration	javadoc
MongoAutoConfiguration	javadoc

Configuration Class	Links
MongoDataAutoConfiguration	javadoc
MongoReactiveAutoConfiguration	javadoc
MongoReactiveDataAutoConfiguration	javadoc
MongoReactiveRepositoriesAutoConfiguration	javadoc
MongoRepositoriesAutoConfiguration	javadoc
MultipartAutoConfiguration	javadoc
MustacheAutoConfiguration	javadoc
Neo4jAutoConfiguration	javadoc
Neo4jDataAutoConfiguration	javadoc
Neo4jReactiveDataAutoConfiguration	javadoc
Neo4jReactiveRepositoriesAutoConfiguration	javadoc
Neo4jRepositoriesAutoConfiguration	javadoc
OAuth2ClientAutoConfiguration	javadoc
OAuth2ResourceServerAutoConfiguration	javadoc
PersistenceExceptionTranslationAutoConfiguration	javadoc
ProjectInfoAutoConfiguration	javadoc
PropertyPlaceholderAutoConfiguration	javadoc
QuartzAutoConfiguration	javadoc
R2dbcAutoConfiguration	javadoc
R2dbcDataAutoConfiguration	javadoc
R2dbcRepositoriesAutoConfiguration	javadoc
R2dbcTransactionManagerAutoConfiguration	javadoc
RSocketMessagingAutoConfiguration	javadoc
RSocketRequesterAutoConfiguration	javadoc
RSocketSecurityAutoConfiguration	javadoc
RSocketServerAutoConfiguration	javadoc

Configuration Class	Links
RSocketStrategiesAutoConfiguration	javadoc
RabbitAutoConfiguration	javadoc
ReactiveElasticsearchRepositoriesAutoConfiguration	javadoc
ReactiveElasticsearchRestClientAutoConfiguration	javadoc
ReactiveOAuth2ClientAutoConfiguration	javadoc
ReactiveOAuth2ResourceServerAutoConfiguration	javadoc
ReactiveSecurityAutoConfiguration	javadoc
ReactiveUserDetailsServiceAutoConfiguration	javadoc
ReactiveWebServerFactoryAutoConfiguration	javadoc
RedisAutoConfiguration	javadoc
RedisReactiveAutoConfiguration	javadoc
RedisRepositoriesAutoConfiguration	javadoc
RepositoryRestMvcAutoConfiguration	javadoc
RestTemplateAutoConfiguration	javadoc
Saml2RelyingPartyAutoConfiguration	javadoc
SecurityAutoConfiguration	javadoc
SecurityFilterAutoConfiguration	javadoc
SendGridAutoConfiguration	javadoc
ServletWebServerFactoryAutoConfiguration	javadoc
SessionAutoConfiguration	javadoc
SolrAutoConfiguration	javadoc
SolrRepositoriesAutoConfiguration	javadoc
SpringApplicationAdminJmxAutoConfiguration	javadoc
SpringDataWebAutoConfiguration	javadoc
TaskExecutionAutoConfiguration	javadoc
TaskSchedulingAutoConfiguration	javadoc

Configuration Class	Links
ThymeleafAutoConfiguration	javadoc
TransactionAutoConfiguration	javadoc
UserDetailsServiceAutoConfiguration	javadoc
ValidationAutoConfiguration	javadoc
WebClientAutoConfiguration	javadoc
WebFluxAutoConfiguration	javadoc
WebMvcAutoConfiguration	javadoc
WebServiceTemplateAutoConfiguration	javadoc
WebServicesAutoConfiguration	javadoc
WebSocketMessagingAutoConfiguration	javadoc
WebSocketReactiveAutoConfiguration	javadoc
WebSocketServletAutoConfiguration	javadoc
XADataSourceAutoConfiguration	javadoc

11.C.2. spring-boot-actuator-autoconfigure

以下自动配置类来自 `spring-boot-actuator-autoconfigure` 模块：

Configuration Class	Links
AppOpticsMetricsExportAutoConfiguration	javadoc
AtlasMetricsExportAutoConfiguration	javadoc
AuditAutoConfiguration	javadoc
AuditEventsEndpointAutoConfiguration	javadoc
AvailabilityHealthContributorAutoConfiguration	javadoc
AvailabilityProbesAutoConfiguration	javadoc
BeansEndpointAutoConfiguration	javadoc
CacheMetricsAutoConfiguration	javadoc

Configuration Class	Links
CachesEndpointAutoConfiguration	javadoc
CassandraHealthContributorAutoConfiguration	javadoc
CassandraReactiveHealthContributorAutoConfiguration	javadoc
CloudFoundryActuatorAutoConfiguration	javadoc
CompositeMeterRegistryAutoConfiguration	javadoc
ConditionsReportEndpointAutoConfiguration	javadoc
ConfigurationPropertiesReportEndpointAutoConfiguration	javadoc
ConnectionFactoryHealthContributorAutoConfiguration	javadoc
ConnectionPoolMetricsAutoConfiguration	javadoc
CouchbaseHealthContributorAutoConfiguration	javadoc
CouchbaseReactiveHealthContributorAutoConfiguration	javadoc
DataSourceHealthContributorAutoConfiguration	javadoc
DataSourcePoolMetricsAutoConfiguration	javadoc
DatadogMetricsExportAutoConfiguration	javadoc
DiskSpaceHealthContributorAutoConfiguration	javadoc
DynatraceMetricsExportAutoConfiguration	javadoc
ElasticMetricsExportAutoConfiguration	javadoc
ElasticSearchReactiveHealthContributorAutoConfiguration	javadoc
ElasticSearchRestHealthContributorAutoConfiguration	javadoc
EndpointAutoConfiguration	javadoc
EnvironmentEndpointAutoConfiguration	javadoc
FlywayEndpointAutoConfiguration	javadoc
GangliaMetricsExportAutoConfiguration	javadoc
GraphiteMetricsExportAutoConfiguration	javadoc
HazelcastHealthContributorAutoConfiguration	javadoc
HealthContributorAutoConfiguration	javadoc

Configuration Class	Links
HealthEndpointAutoConfiguration	javadoc
HeapDumpWebEndpointAutoConfiguration	javadoc
HibernateMetricsAutoConfiguration	javadoc
HttpClientMetricsAutoConfiguration	javadoc
HttpTraceAutoConfiguration	javadoc
HttpTraceEndpointAutoConfiguration	javadoc
HumioMetricsExportAutoConfiguration	javadoc
InfluxDbHealthContributorAutoConfiguration	javadoc
InfluxMetricsExportAutoConfiguration	javadoc
InfoContributorAutoConfiguration	javadoc
InfoEndpointAutoConfiguration	javadoc
IntegrationGraphEndpointAutoConfiguration	javadoc
JerseyServerMetricsAutoConfiguration	javadoc
JettyMetricsAutoConfiguration	javadoc
JmsHealthContributorAutoConfiguration	javadoc
JmxEndpointAutoConfiguration	javadoc
JmxMetricsExportAutoConfiguration	javadoc
JolokiaEndpointAutoConfiguration	javadoc
JvmMetricsAutoConfiguration	javadoc
KafkaMetricsAutoConfiguration	javadoc
KairosMetricsExportAutoConfiguration	javadoc
LdapHealthContributorAutoConfiguration	javadoc
LiquibaseEndpointAutoConfiguration	javadoc
Log4J2MetricsAutoConfiguration	javadoc
LogFileWebEndpointAutoConfiguration	javadoc
LogbackMetricsAutoConfiguration	javadoc

Configuration Class	Links
LoggersEndpointAutoConfiguration	javadoc
MailHealthContributorAutoConfiguration	javadoc
ManagementContextAutoConfiguration	javadoc
ManagementWebSecurityAutoConfiguration	javadoc
MappingsEndpointAutoConfiguration	javadoc
MetricsAutoConfiguration	javadoc
MetricsEndpointAutoConfiguration	javadoc
MongoHealthContributorAutoConfiguration	javadoc
MongoReactiveHealthContributorAutoConfiguration	javadoc
Neo4jHealthContributorAutoConfiguration	javadoc
NewRelicMetricsExportAutoConfiguration	javadoc
PrometheusMetricsExportAutoConfiguration	javadoc
RabbitHealthContributorAutoConfiguration	javadoc
RabbitMetricsAutoConfiguration	javadoc
ReactiveCloudFoundryActuatorAutoConfiguration	javadoc
ReactiveManagementContextAutoConfiguration	javadoc
ReactiveManagementWebSecurityAutoConfiguration	javadoc
RedisHealthContributorAutoConfiguration	javadoc
RedisReactiveHealthContributorAutoConfiguration	javadoc
ScheduledTasksEndpointAutoConfiguration	javadoc
ServletManagementContextAutoConfiguration	javadoc
SessionsEndpointAutoConfiguration	javadoc
ShutdownEndpointAutoConfiguration	javadoc
SignalFxMetricsExportAutoConfiguration	javadoc
SimpleMetricsExportAutoConfiguration	javadoc
SolrHealthContributorAutoConfiguration	javadoc

Configuration Class	Links
<code>StackdriverMetricsExportAutoConfiguration</code>	javadoc
<code>StartupEndpointAutoConfiguration</code>	javadoc
<code>StatsdMetricsExportAutoConfiguration</code>	javadoc
<code>SystemMetricsAutoConfiguration</code>	javadoc
<code>ThreadDumpEndpointAutoConfiguration</code>	javadoc
<code>TomcatMetricsAutoConfiguration</code>	javadoc
<code>WavefrontMetricsExportAutoConfiguration</code>	javadoc
<code>WebEndpointAutoConfiguration</code>	javadoc
<code>WebFluxMetricsAutoConfiguration</code>	javadoc
<code>WebMvcMetricsAutoConfiguration</code>	javadoc

Appendix D：测试自动配置注解

本附录描述了 Spring Boot 提供的用于测试应用程序切片的 `@...Test` 自动配置注解。

11.D.1. 测试切片

下表列出了各种可用于测试应用程序片的 `@...Test` 注解以及它们默认情况下导入的自动配置：

Test slice	Imported auto-configuration
@DataCassandraTest	<code>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.cassandra.Cassandra ReactiveDataAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.cassandra.Cassandra ReactiveRepositoriesAutoConfiguration</code> <code>org.springframework.boot.autoconfigure.data.cassandra.Cassandra RepositoriesAutoConfiguration</code>

Test slice	Imported auto-configuration
@DataJdbcTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfiguration 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

Test slice	Imported auto-configuration
@DataJpaTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManagerAutoConfiguration

Test slice	Imported auto-configuration
@DataLdapTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration

Test slice	Imported auto-configuration
@DataMongoTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration

Test slice	Imported auto-configuration
@DataNeo4jTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration org.springframework.boot.autoconfigure.data.neo4j.Neo4jReactiveDataAutoConfiguration org.springframework.boot.autoconfigure.data.neo4j.Neo4jReactiveRepositoriesAutoConfiguration org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration org.springframework.boot.autoconfigure.neo4j.Neo4jAutoConfiguration org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration

Test slice	Imported auto-configuration
@DataR2dbcTest	<pre>org.springframework.boot.autoconfigure.data.r2dbc.R2dbcDataAutoConfiguration org.springframework.boot.autoconfigure.data.r2dbc.R2dbcRepositoriesAutoConfiguration org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration org.springframework.boot.autoconfigure.r2dbc.R2dbcAutoConfiguration org.springframework.boot.autoconfigure.r2dbc.R2dbcTransactionManagerAutoConfiguration org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration</pre>
@DataRedisTest	<pre>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration</pre>

Test slice	Imported auto-configuration
@JdbcTest	<code>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</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.transaction.TransactionAutoConfiguration</code> <code>org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration</code>

Test slice	Imported auto-configuration
@JooqTest	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.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.json.JsonbAutoConfiguration org.springframework.boot.test.autoconfigure.json.JsonTestersAutoConfiguration

Test slice	Imported auto-configuration
@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

Test slice	Imported auto-configuration
@WebFluxTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientAutoConfiguration org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConfiguration org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration

Test slice	Imported auto-configuration
@WebMvcTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration 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.security.oauth2.client.servlet.OAuth2ClientAutoConfiguration org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2ResourceServerAutoConfiguration org.springframework.boot.autoconfigure

Test slice	Imported auto-configuration
@WebServiceClientTest	org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration org.springframework.boot.test.autoconfigure.webservices.client.MockWebServiceServerAutoConfiguration org.springframework.boot.test.autoconfigure.webservices.client.WebServiceClientTemplateAutoConfiguration

Appendix E: 可执行Jar格式

spring-boot-loader 模块使 Spring Boot 支持可执行的 jar 和 war 文件。如果使用 Maven 插件或 Gradle 插件，则会自动生成可执行的 jar，通常不需要了解其工作方式的详细信息。

如果您需要从其他构建系统创建可执行 jar，或者您只是对基础知识感到好奇，则本附录提供了一些背景知识。

11.E.1. 嵌套 JARs

Java 没有提供任何标准的方式来装载嵌套的 jar 文件（即，它们本身包含在 jar 中的 jar 文件）。如果您需要分发一个自包含的应用程序，则该应用程序可以从命令行运行而无需解压缩，则可能会出现问题。

为了解决这个问题，许多开发人员使用“shaded” jars。将包含所有 jar 的所有类的 shaded 的 jar 打包到一个“uber jar”中。带 shaded 的 jar 的问题在于，很难查看应用程序中实际包含哪些库。如果在多个 jar 中使用相同的文件名（但具有不同的内容），也可能会产生问题。Spring Boot 采用了另一种方法，实际上允许您直接嵌套 jar。

可执行的 **Jar** 文件结构

与 Spring Boot Loader 兼容的 jar 文件的结构应采用以下方式：

```
example.jar
|
+-META-INF
|   +-MANIFEST.MF
+-org
|   +-springframework
|   |   +-boot
|   |   |   +-loader
|   |   |   +-<spring boot loader classes>
+-BOOT-INF
|   +-classes
|   |   +-mycompany
|   |   |   +-project
|   |   |   +-YourClasses.class
|   +-lib
|       +-dependency1.jar
|       +-dependency2.jar
```

应用程序类应放在嵌套的 **BOOT-INF/classes** 目录中。依赖应放在嵌套的 **BOOT-INF/lib** 目录中。

可执行 **War** 文件结构

与 Spring Boot Loader 兼容的 war 文件的结构应采用以下方式：

```

example.war
|
+-META-INF
|   +-MANIFEST.MF
+-org
|   +-springframework
|       +-boot
|           +-loader
|               +-<spring boot loader classes>
+-WEB-INF
    +-classes
        |   +-com
        |       +-mycompany
        |           +-project
        |               +-YourClasses.class
    +-lib
        |   +-dependency1.jar
        |   +-dependency2.jar
    +-lib-provided
        +-servlet-api.jar
        +-dependency3.jar

```

依赖应放在嵌套的 **WEB-INF/lib** 目录中。在运行嵌入式程序时需要但在部署到传统 Web 容器时不需要的任何依赖都应放在 **WEB-INF/lib-provided** 的文件中。

文件索引

Spring Boot Loader-compatible 的 jar 和 war 可以在 **BOOT-INF/** 目录下附加一个索引文件，可以同时为 jar 和 war 提供一个 **classpath.idx** 文件，他提供了将 jar 加载到类路径的顺序。**layers.idx** 文件只能用于 jar，它允许将 jar 进行逻辑分层，以创建 Docker/OCI 镜像。

索引文件使用兼容 YAML 的语法，以便可以由第三方工具轻松解析。但是，这些文件在内部没有作为 YAML 进行解析，因此必须按照以下所述的格式编写，才能使用。

Classpath Index

可以在 **BOOT-INF/classpath.idx** 中提供类路径索引文件。它提供了 jar 名称列表（包括目录），其顺序为应将其添加到类路径中。每行必须以破折号 (**"-."**) 开头，并且名称必须用双引号引起来。

例如，给出以下 jar：

```

example.jar
|
+-META-INF
|   +...
+-BOOT-INF
|   +-classes
|   |   +...
|   +-lib
|       +-dependency1.jar
|       +-dependency2.jar

```

索引文件应为：

```

- "BOOT-INF/lib/dependency2.jar"
- "BOOT-INF/lib/dependency1.jar"

```

Layer Index(分层索引)

可以在 `BOOT-INF/layers.idx` 中提供分层索引文件。

它提供了每一层的列表以及应包含在其中的 `jar`. 按照应将其添加到 Docker/OCI 镜像的顺序来分层. 每一层的名称以带引号的字符串编写, 前缀带有短划线空格 (`"-·"`) 和后缀冒号 (:") . 分层内容是用引号引起起来的字符串的文件名或目录名, 并带有空格短划线 (`"·-·"`). 目录名称以 `/` 结尾, 而文件名则没有. 使用目录名称时, 意味着该目录内的所有文件都在同一层.

分层索引的典型示例是：

```

- "dependencies":
  - "BOOT-INF/lib/dependency1.jar"
  - "BOOT-INF/lib/dependency2.jar"
- "application":
  - "BOOT-INF/classes/"
  - "META-INF/"

```

11.E.2. Spring Boot “JarFile” 类

用于支持加载嵌套 `jar` 的核心类是

`org.springframework.boot.loader.jar.JarFile`. 它使您可以从标准 `jar` 文件或嵌套的子 `jar` 数据加载 `jar` 内容. 首次加载时, 每个 `JarEntry` 的位置都映射到外部

`jar` 的物理文件偏移, 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
```

前面的示例显示了如何在 `myapp.jar` 的 `0063` 置的 `/BOOT-INF/classes` 中找到 `A.class`. 嵌套jar的 `B.class` 实际上可以在 `myapp.jar` 的 `3452` 位置中找到, 而 `C.class`是 在位置 `3980.`

有了这些信息, 我们可以通过查找外部 `jar` 的适当部分来加载特定的嵌套条目. 我们不需要解压缩归档文件, 也不需要将所有条目数据读入内存.

与标准Java “`JarFile`” 的兼容性

`Spring Boot Loader` 努力保持与现有代码和库的兼容性.

`org.springframework.boot.loader.jar.JarFile` 从 `java.util.jar.JarFile` 扩展而来, 应该可以作为替代产品. `getURL()` 方法返回一个 `URL`, 该 `URL` 打开一个与 `java.net.JarURLConnection` 兼容的连接, 并且可以与 Java 的 `URLClassLoader` 一起使用.

11.E.3. 运行可执行 Jars

`org.springframework.boot.loader.Launcher` 类是特殊的引导程序类, 用作可执行 `jar` 的主要入口点. 它是 `jar` 文件中的实际 `Main-Class`, 用于设置适当的 `URLClassLoader` 并最终调用 `main()` 方法.

有三个启动器子类 (`JarLauncher`, `WarLauncher` 和 `PropertiesLauncher`) . 它们的目的是从目录中的嵌套 `jar` 文件或 `war` 文件 (而不是在类路径中显式的文件) 加载资源 (`.class` 文件等) . 对于 `JarLauncher` 和 `WarLauncher`, 嵌套路径是固定的. `JarLauncher` 位于 `BOOT-INF/lib/` 中, 而 `WarLauncher` 位于 `WEB-INF/lib/` 和 `WEB-INF/lib-provided/` 中. 如果需要, 可以在这些位置添加额外的 `jar`. 默认情况下

, `PropertiesLauncher` 在您的应用程序存档中的 `BOOT-INF/lib/` 中查找. 您可以通过在 `loader.properties` (这是目录, 归档文件或归档文件中的目录的逗号分隔列表) 中设置一个称为 `LOADER_PATH` 或 `loader.path` 的环境变量来添加其他位置.

运行 Manifest

您需要指定一个适当的启动器作为 `META-INF/MANIFEST.MF` 的 `Main-Class` 属性. 您要启动的实际类 (即包含 `main` 方法的类) 应在 `Start-Class` 属性中指定.

下面的示例显示了一个可执行jar文件的典型 `MANIFEST.MF`:

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

如果是 `war` 文件, 则如下

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```



您无需在清单文件中指定 `Class-Path` 条目. 类路径是从嵌套的 jar 中推导出来的.

11.E.4. PropertiesLauncher 特性

`PropertiesLauncher` 具有一些可以通过外部属性 (系统属性, 环境变量, `manifest entries` 或 `loader.properties`) 启用的特殊功能. 下表描述了这些属性:

Key	Purpose
<code>loader.path</code>	逗号分隔的类路径, 例如 <code>lib, \${HOME}/app/lib.</code> 较早的条目具有优先权, 就像 <code>javac</code> 命令行上的常规 <code>-classpath</code> 一样. .

Key	Purpose
<code>loader.home</code>	用于解析 <code>loader.path</code> 中的相对路径。例如,给定 <code>loader.path=lib</code> ,,则 <code> \${loader.home}/lib</code> 是类路径位置(以及该目录中的所有 <code>jar</code> 文件)。此属性还用于查找 <code>loader.properties</code> 文件,如以下示例 <code>/opt/app</code> 所示。它默认为 <code> \${user.dir}</code> 。
<code>loader.args</code>	<code>main</code> 方法的默认参数 (以空格分隔) .
<code>loader.main</code>	要启动的主类的名称 (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>) . 默认为 <code>loader.properties.</code>)
<code>loader.system</code>	布尔值标志,指示应将所有属性添加到系统属性。默认为 <code>false</code> .

当指定为环境变量或 `manifest` 时,应使用以下名称:

Key	Manifest entry	Environment variable
<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 `fat jar` 时, 构建插件会自动将 `Main-Class` 属性移动到 `Start-Class`. 如果使用该名称, 请使用 `Main-Class` 属性指定要启动的类的名称, 而忽略 `Start-Class`.

以下规则适用于使用 `PropertiesLauncher`:

- 在 `loader.home` 中搜索 `loader.properties`, 然后在类路径的根目录中搜索, 然后在类路径: `/BOOT-INF/classes` 中搜索. 使用具有该名称的文件的第一个位置.
- 仅当未指定 `loader.config.location` 时, `loader.home` 是其他属性文件的目录位置 (覆盖默认值) .
- `loader.path` 可以包含目录 (对 `jar` 和 `zip` 文件进行递归扫描), 存档路径, 存档文件中的对 `jar` 文件进行扫描的目录 (例如, `dependencies.jar!/lib`) 或通配符模式 (对于 默认JVM行为). 归档路径可以相对于 `loader.home` 或文件系统中任何带有 `jar:file:` 前缀的位置.
- `loader.path` (如果为空) 默认为 `BOOT-INF/lib` (表示本地目录, 如果从归档文件运行, 则为嵌套目录) . 因此, 如果未提供其他配置, 则 `PropertiesLauncher` 的行为与 `JarLauncher` 相同.
- `loader.path` 不能用于配置 `loader.properties` 的位置 (用于启动后者的类路径是启动 `PropertiesLauncher` 时的JVM类路径) .
- 占位符的替换是使用系统变量和环境变量以及属性文件本身的所有值完成的, 然后再使用.
- 属性 (在多个位置中有意义的查找) 的搜索顺序是环境变量, 系统属性, `loader.properties`, 暴露的存档清单和存档清单.

11.E.5. 可执行 Jar 限制

使用 Spring Boot Loader 打包的应用程序时, 需要考虑以下限制:

- Zip 压缩: 必须使用 `ZipEntry.STORED` 方法保存嵌套 `jar` 的 `ZipEntry`. 这是必需的, 以便我们可以直接在嵌套 `jar` 中查找单个内容. 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.
- `System classLoader`: 启动的应用程序在加载类时应使用 `Thread.getContextClassLoader()` (默认情况下, 大多数库和框架都使用

`Thread.getContextClassLoader()` . 尝试使用
`ClassLoader.getSystemClassLoader()` 加载嵌套jar类失败.
`java.util.Logging` 始终使用系统类加载器. 因此, 您应该考虑使用其他日志记录实现.

11.E.6. 替代 单一 Jar 方案

如果上述限制意味着您不能使用 Spring Boot Loader, 请考虑以下替代方法:

- [Maven Shade Plugin](#)
- [JarClassLoader](#)
- [OneJar](#)
- [Gradle Shadow Plugin](#)

Appendix F: 依赖版本

本附录提供了 Spring Boot 管理的依赖的详细信息.

11.F.1. 管理依赖坐标

下表提供了 Spring Boot 在其 CLI(命令行界面) , Maven 依赖管理和 Gradle 插件中提供的所有依赖版本的详细信息. 当您的依赖声明了 `artifacts` 而未声明版本时, 将使用表中列出的版本.

Group ID	Artifact ID	Version
antlr	antlr	2.7.7
ch.qos.logback	logback-access	1.2.3
ch.qos.logback	logback-classic	1.2.3
ch.qos.logback	logback-core	1.2.3
com.atomikos	transactions-jdbc	4.0.6
com.atomikos	transactions-jms	4.0.6
com.atomikos	transactions-jta	4.0.6
com.couchbase.client	java-client	3.0.10

Group ID	Artifact ID	Version
com.datastax.oss	java-driver-core	4.9.0
com.datastax.oss	java-driver-core-shaded	4.9.0
com.datastax.oss	java-driver-mapper-processor	4.9.0
com.datastax.oss	java-driver-mapper-runtime	4.9.0
com.datastax.oss	java-driver-metrics-micrometer	4.9.0
com.datastax.oss	java-driver-metrics-micrometer	4.9.0
com.datastax.oss	java-driver-query-builder	4.9.0
com.datastax.oss	java-driver-shaded-guava	25.1-jre-graal-sub-1
com.datastax.oss	java-driver-test-infra	4.9.0
com.datastax.oss	native-protocol	1.4.11
com.fasterxml	classmate	1.5.1
com.fasterxml.jackson.core	jackson-annotations	2.11.4
com.fasterxml.jackson.core	jackson-core	2.11.4
com.fasterxml.jackson.core	jackson-databind	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-avro	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-cbor	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-csv	2.11.4

Group ID	Artifact ID	Version
com.fasterxml.jackson.datatype	jackson-dataformat-ion	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-properties	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-proto	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-smile	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-xml	2.11.4
com.fasterxml.jackson.datatype	jackson-dataformat-yaml	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-eclipse-collections	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-guava	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-hibernate3	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-hibernate4	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-hibernate5	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-hppc	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-jaxrs	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-jdk8	2.11.4
com.fasterxml.jackson.datatype	jackson-datatype-joda	2.11.4

Group ID	Artifact ID	Version
com.fasterxml.jackson.datatype.joda-money	jackson-datatype-joda-money	2.11.4
com.fasterxml.jackson.datatype.json-org	jackson-datatype-json-org	2.11.4
com.fasterxml.jackson.datatype.jsr310	jackson-datatype-jsr310	2.11.4
com.fasterxml.jackson.datatype.jsr353	jackson-datatype-jsr353	2.11.4
com.fasterxml.jackson.datatype-pcollections	jackson-datatype-pcollections	2.11.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-base	2.11.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-cbor-provider	2.11.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-json-provider	2.11.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-smile-provider	2.11.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-xml-provider	2.11.4
com.fasterxml.jackson.jaxrs	jackson-jaxrs-yaml-provider	2.11.4
com.fasterxml.jackson.jr	jackson-jr-all	2.11.4
com.fasterxml.jackson.jr	jackson-jr-annotation-support	2.11.4
com.fasterxml.jackson.jr	jackson-jr-objects	2.11.4
com.fasterxml.jackson.jr	jackson-jr-retrofit2	2.11.4

Group ID	Artifact ID	Version
com.fasterxml.jackson.jr	jackson-jr-tree	2.11.4
com.fasterxml.jackson.module	jackson-module-afterburner	2.11.4
com.fasterxml.jackson.module	jackson-module-guice	2.11.4
com.fasterxml.jackson.module	jackson-module-jaxb-annotations	2.11.4
com.fasterxml.jackson.module	jackson-module-jsonSchema	2.11.4
com.fasterxml.jackson.module	jackson-module-kotlin	2.11.4
com.fasterxml.jackson.module	jackson-module-mrbean	2.11.4
com.fasterxml.jackson.module	jackson-module-osgi	2.11.4
com.fasterxml.jackson.module	jackson-module-parameter-names	2.11.4
com.fasterxml.jackson.module	jackson-module-paranamer	2.11.4
com.fasterxml.jackson.module	jackson-module-scala_2.10	2.11.4
com.fasterxml.jackson.module	jackson-module-scala_2.11	2.11.4
com.fasterxml.jackson.module	jackson-module-scala_2.12	2.11.4
com.fasterxml.jackson.module	jackson-module-scala_2.13	2.11.4
com.github.ben-manes.caffeine	caffeine	2.8.8

Group ID	Artifact ID	Version
com.github.ben-manes.caffeine	guava	2.8.8
com.github.ben-manes.caffeine	jcache	2.8.8
com.github.ben-manes.caffeine	simulator	2.8.8
com.github.mxab.thymeleaf-extras-dataaf.extras	thymeleaf-extras-data-attribute	2.0.1
com.google.appengine	appengine-api-1.0-sdk	1.9.88
com.google.cloud	cloud-spanner-r2dbc	0.3.0
com.google.code.gson	gson	2.8.6
com.h2database	h2	1.4.200
com.hazelcast	hazelcast	4.0.3
com.hazelcast	hazelcast-hibernate52	2.1.1
com.hazelcast	hazelcast-hibernate53	2.1.1
com.hazelcast	hazelcast-spring	4.0.3
com.ibm.db2	jcc	11.5.5.0
com.jayway.jsonpath	json-path	2.4.0
com.jayway.jsonpath	json-path-assert	2.4.0
com.microsoft.sqlserver	mssql-jdbc	8.4.1.jre8
com.nimbusds	nimbus-jose-jwt	8.20.2
com.nimbusds	oauth2-oidc-sdk	8.36.1
com.oracle.database.deb ug	ojdbc10_g	19.8.0.0
com.oracle.database.deb ug	ojdbc10dms_g	19.8.0.0
com.oracle.database.deb ug	ojdbc8_g	19.8.0.0

Group ID	Artifact ID	Version
com.oracle.database.deb ug	ojdbc8dms_g	19.8.0.0
com.oracle.database.ha	ons	19.8.0.0
com.oracle.database.ha	simplefan	19.8.0.0
com.oracle.database.jdbc c	ojdbc10	19.8.0.0
com.oracle.database.jdbc c	ojdbc10-production	19.8.0.0
com.oracle.database.jdbc c	ojdbc8	19.8.0.0
com.oracle.database.jdbc c	ojdbc8-production	19.8.0.0
com.oracle.database.jdbc c	ucp	19.8.0.0
com.oracle.database.jdbc c.debug	ojdbc10-debug	19.8.0.0
com.oracle.database.jdbc c.debug	ojdbc10-observability- debug	19.8.0.0
com.oracle.database.jdbc c.debug	ojdbc8-debug	19.8.0.0
com.oracle.database.jdbc c.debug	ojdbc8-observability- debug	19.8.0.0
com.oracle.database.nls	orai18n	19.8.0.0
com.oracle.database.obs ervability	dms	19.8.0.0
com.oracle.database.obs ervability	ojdbc10-observability	19.8.0.0
com.oracle.database.obs ervability	ojdbc10dms	19.8.0.0

Group ID	Artifact ID	Version
com.oracle.database.observability	ojdbc8-observability	19.8.0.0
com.oracle.database.observability	ojdbc8dms	19.8.0.0
com.oracle.database.security	oraclepkicurity	19.8.0.0
com.oracle.database.security	osdt_cert	19.8.0.0
com.oracle.database.security	osdt_core	19.8.0.0
com.oracle.database.xml	xdb	19.8.0.0
com.oracle.database.xml	xmlparserv2	19.8.0.0
com.oracle.jdbc	dms	19.3.0.0
com.oracle.jdbc	ojdbc10	19.3.0.0
com.oracle.jdbc	ojdbc10_g	19.3.0.0
com.oracle.jdbc	ojdbc10dms	19.3.0.0
com.oracle.jdbc	ojdbc10dms_g	19.3.0.0
com.oracle.jdbc	ojdbc8	19.3.0.0
com.oracle.jdbc	ojdbc8_g	19.3.0.0
com.oracle.jdbc	ojdbc8dms	19.3.0.0
com.oracle.jdbc	ojdbc8dms_g	19.3.0.0
com.oracle.jdbc	ons	19.3.0.0
com.oracle.jdbc	oraclepkicurity	19.3.0.0
com.oracle.jdbc	orai18n	19.3.0.0
com.oracle.jdbc	osdt_cert	19.3.0.0
com.oracle.jdbc	osdt_core	19.3.0.0
com.oracle.jdbc	simplefan	19.3.0.0

Group ID	Artifact ID	Version
com.oracle.jdbc	ucp	19.3.0.0
com.oracle.jdbc	xdb	19.3.0.0
com.oracle.jdbc	xmlparserv2	19.3.0.0
com.querydsl	querydsl-apt	4.4.0
com.querydsl	querydsl-collections	4.4.0
com.querydsl	querydsl-core	4.4.0
com.querydsl	querydsl-jpa	4.4.0
com.querydsl	querydsl-mongodb	4.4.0
com.rabbitmq	amqp-client	5.10.0
com.samskivert	jmustache	1.15
com.sendgrid	sendgrid-java	4.6.8
com.squareup.okhttp3	logging-interceptor	3.14.9
com.squareup.okhttp3	mockwebserver	3.14.9
com.squareup.okhttp3	okcurl	3.14.9
com.squareup.okhttp3	okhttp	3.14.9
com.squareup.okhttp3	okhttp-dnsoverhttps	3.14.9
com.squareup.okhttp3	okhttp-sse	3.14.9
com.squareup.okhttp3	okhttp-testing-support	3.14.9
com.squareup.okhttp3	okhttp-tls	3.14.9
com.squareup.okhttp3	okhttp-urlconnection	3.14.9
com.sun.activation	jakarta.activation	1.2.2
com.sun.mail	jakarta.mail	1.6.7
com.sun.xml.messaging.saaj	saaj-impl	1.5.3
com.unboundid	unboundid-ldapsdk	4.0.14
com.zaxxer	HikariCP	3.4.5

Group ID	Artifact ID	Version
commons-codec	commons-codec	1.15
commons-pool	commons-pool	1.6
de.flapdoodle.embed	de.flapdoodle.embed.mongo	2.2.0
dev.miku	r2dbc-mysql	0.8.2.RELEASE
io.dropwizard.metrics	metrics-annotation	4.1.19
io.dropwizard.metrics	metrics-caffeine	4.1.19
io.dropwizard.metrics	metrics-collectd	4.1.19
io.dropwizard.metrics	metrics-core	4.1.19
io.dropwizard.metrics	metrics-ehcache	4.1.19
io.dropwizard.metrics	metrics-graphite	4.1.19
io.dropwizard.metrics	metrics-healthchecks	4.1.19
io.dropwizard.metrics	metrics-httpsyncclient	4.1.19
io.dropwizard.metrics	metrics-httpclient	4.1.19
io.dropwizard.metrics	metrics-jcache	4.1.19
io.dropwizard.metrics	metrics-jdbi	4.1.19
io.dropwizard.metrics	metrics-jdbi3	4.1.19
io.dropwizard.metrics	metrics-jersey2	4.1.19
io.dropwizard.metrics	metrics-jetty9	4.1.19
io.dropwizard.metrics	metrics-jmx	4.1.19
io.dropwizard.metrics	metrics-json	4.1.19
io.dropwizard.metrics	metrics-jvm	4.1.19
io.dropwizard.metrics	metrics-log4j2	4.1.19
io.dropwizard.metrics	metrics-logback	4.1.19
io.dropwizard.metrics	metrics-servlet	4.1.19
io.dropwizard.metrics	metrics-servlets	4.1.19

Group ID	Artifact ID	Version
io.lettuce	lettuce-core	6.0.4.RELEASE
io.micrometer	micrometer-core	1.6.6
io.micrometer	micrometer-jersey2	1.6.6
io.micrometer	micrometer-registry-appoptics	1.6.6
io.micrometer	micrometer-registry-atlas	1.6.6
io.micrometer	micrometer-registry-azure-monitor	1.6.6
io.micrometer	micrometer-registry-cloudwatch	1.6.6
io.micrometer	micrometer-registry-cloudwatch2	1.6.6
io.micrometer	micrometer-registry-datadog	1.6.6
io.micrometer	micrometer-registry-dynatrace	1.6.6
io.micrometer	micrometer-registry-elastic	1.6.6
io.micrometer	micrometer-registry-ganglia	1.6.6
io.micrometer	micrometer-registry-graphite	1.6.6
io.micrometer	micrometer-registry-health	1.6.6
io.micrometer	micrometer-registry-humio	1.6.6
io.micrometer	micrometer-registry-influx	1.6.6

Group ID	Artifact ID	Version
io.micrometer	micrometer-registry-jmx	1.6.6
io.micrometer	micrometer-registry-kairos	1.6.6
io.micrometer	micrometer-registry-newrelic	1.6.6
io.micrometer	micrometer-registry-opentsdb	1.6.6
io.micrometer	micrometer-registry-prometheus	1.6.6
io.micrometer	micrometer-registry-signalfx	1.6.6
io.micrometer	micrometer-registry-stackdriver	1.6.6
io.micrometer	micrometer-registry-statsd	1.6.6
io.micrometer	micrometer-registry-wavefront	1.6.6
io.micrometer	micrometer-test	1.6.6
io.netty	netty-all	4.1.63.Final
io.netty	netty-buffer	4.1.63.Final
io.netty	netty-codec	4.1.63.Final
io.netty	netty-codec-dns	4.1.63.Final
io.netty	netty-codec-haproxy	4.1.63.Final
io.netty	netty-codec-http	4.1.63.Final
io.netty	netty-codec-http2	4.1.63.Final
io.netty	netty-codec-memcache	4.1.63.Final
io.netty	netty-codec-mqtt	4.1.63.Final
io.netty	netty-codec-redis	4.1.63.Final

Group ID	Artifact ID	Version
io.netty	netty-codec-smtp	4.1.63.Final
io.netty	netty-codec-socks	4.1.63.Final
io.netty	netty-codec-stomp	4.1.63.Final
io.netty	netty-codec-xml	4.1.63.Final
io.netty	netty-common	4.1.63.Final
io.netty	netty-dev-tools	4.1.63.Final
io.netty	netty-example	4.1.63.Final
io.netty	netty-handler	4.1.63.Final
io.netty	netty-handler-proxy	4.1.63.Final
io.netty	netty-resolver	4.1.63.Final
io.netty	netty-resolver-dns	4.1.63.Final
io.netty	netty-resolver-dns-native-macos	4.1.63.Final
io.netty	netty-tcnative	2.0.38.Final
io.netty	netty-tcnative-boringssl-static	2.0.38.Final
io.netty	netty-transport	4.1.63.Final
io.netty	netty-transport-native-epoll	4.1.63.Final
io.netty	netty-transport-native-kqueue	4.1.63.Final
io.netty	netty-transport-native-unix-common	4.1.63.Final
io.netty	netty-transport-rxtx	4.1.63.Final
io.netty	netty-transport-sctp	4.1.63.Final
io.netty	netty-transport-udt	4.1.63.Final
io.projectreactor	reactor-core	3.4.5

Group ID	Artifact ID	Version
io.projectreactor	reactor-test	3.4.5
io.projectreactor	reactor-tools	3.4.5
io.projectreactor.addons	reactor-adapter	3.4.3
io.projectreactor.addons	reactor-extra	3.4.3
io.projectreactor.addons	reactor-pool	0.2.4
io.projectreactor.kafka	reactor-kafka	1.3.3
io.projectreactor.kotlin	reactor-kotlin-extensions	1.1.3
io.projectreactor.netty	reactor-netty	1.0.6
io.projectreactor.netty	reactor-netty-core	1.0.6
io.projectreactor.netty	reactor-netty-http	1.0.6
io.projectreactor.netty	reactor-netty-http-brave	1.0.6
io.projectreactor.rabbitmq	reactor-rabbitmq	1.5.2
io.prometheus	simpleclient_pushgateway	0.9.0
io.r2dbc	r2dbc-h2	0.8.4.RELEASE
io.r2dbc	r2dbc-mssql	0.8.5.RELEASE
io.r2dbc	r2dbc-pool	0.8.6.RELEASE
io.r2dbc	r2dbc-postgresql	0.8.7.RELEASE
io.r2dbc	r2dbc-proxy	0.8.5.RELEASE
io.r2dbc	r2dbc-spi	0.8.4.RELEASE
io.reactivex	rxjava	1.3.8
io.reactivex	rxjava-reactive-streams	1.2.1

Group ID	Artifact ID	Version
io.reactivex.rxjava2	rxjava	2.2.21
io.rest-assured	json-path	3.3.0
io.rest-assured	json-schema-validator	3.3.0
io.rest-assured	rest-assured	3.3.0
io.rest-assured	scala-support	3.3.0
io.rest-assured	spring-mock-mvc	3.3.0
io.rest-assured	spring-web-test-client	3.3.0
io.rest-assured	xml-path	3.3.0
io.rsocket	rsocket-core	1.1.0
io.rsocket	rsocket-load-balancer	1.1.0
io.rsocket	rsocket-micrometer	1.1.0
io.rsocket	rsocket-test	1.1.0
io.rsocket	rsocket-transport-local	1.1.0
io.rsocket	rsocket-transport-netty	1.1.0
io.spring.gradle	dependency-management-plugin	1.0.11.RELEASE
io.undertow	undertow-core	2.2.7.Final
io.undertow	undertow-servlet	2.2.7.Final
io.undertow	undertow-websockets-jsr	2.2.7.Final
jakarta.activation	jakarta.activation-api	1.2.2
jakarta.annotation	jakarta.annotation-api	1.3.5
jakarta.jms	jakarta.jms-api	2.0.3
jakarta.json	jakarta.json-api	1.1.6
jakarta.json.bind	jakarta.json.bind-api	1.0.2
jakarta.mail	jakarta.mail-api	1.6.7
jakarta.persistence	jakarta.persistence-api	2.2.3

Group ID	Artifact ID	Version
jakarta.servlet	jakarta.servlet-api	4.0.4
jakarta.servlet.jsp.jstl	jakarta.servlet.jsp.jstl-api	1.2.7
jakarta.transaction	jakarta.transaction-api	1.3.3
jakarta.validation	jakarta.validation-api	2.0.2
jakarta.websocket	jakarta.websocket-api	1.1.2
jakarta.ws.rs	jakarta.ws.rs-api	2.1.6
jakarta.xml.bind	jakarta.xml.bind-api	2.3.3
jakarta.xml.soap	jakarta.xml.soap-api	1.4.2
jakarta.xml.ws	jakarta.xml.ws-api	2.3.3
javax.activation	javax.activation-api	1.2.0
javax.annotation	javax.annotation-api	1.3.2
javax.cache	cache-api	1.1.1
javax.jms	javax.jms-api	2.0.1
javax.json	javax.json-api	1.1.4
javax.json.bind	javax.json.bind-api	1.0
javax.mail	javax.mail-api	1.6.2
javax.money	money-api	1.1
javax.persistence	javax.persistence-api	2.2
javax.servlet	javax.servlet-api	4.0.1
javax.servlet	jstl	1.2
javax.transaction	javax.transaction-api	1.3
javax.validation	validation-api	2.0.1.Final
javax.websocket	javax.websocket-api	1.1
javax.xml.bind	jaxb-api	2.3.1
javax.xml.ws	jaxws-api	2.3.1

Group ID	Artifact ID	Version
jaxen	jaxen	1.2.0
junit	junit	4.12
mysql	mysql-connector-java	8.0.23
net.bytebuddy	byte-buddy	1.10.22
net.bytebuddy	byte-buddy-agent	1.10.22
net.minidev	json-smart	2.3
net.sf.ehcache	ehcache	2.10.6
net.sourceforge.htmlunit	htmlunit	2.44.0
net.sourceforge.jtds	jtds	1.3.1
net.sourceforge.nekohtml	nekohtml	1.9.22
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect	2.5.2
org.apache.activemq	activemq-amqp	5.16.1
org.apache.activemq	activemq-blueprint	5.16.1
org.apache.activemq	activemq-broker	5.16.1
org.apache.activemq	activemq-camel	5.16.1
org.apache.activemq	activemq-client	5.16.1
org.apache.activemq	activemq-console	5.16.1
org.apache.activemq	activemq-http	5.16.1
org.apache.activemq	activemq-jaas	5.16.1
org.apache.activemq	activemq-jdbc-store	5.16.1
org.apache.activemq	activemq-jms-pool	5.16.1
org.apache.activemq	activemq-kahadb-store	5.16.1
org.apache.activemq	activemq-karaf	5.16.1
org.apache.activemq	activemq-leveldb-store	5.16.1

Group ID	Artifact ID	Version
org.apache.activemq	activemq-log4j-appender	5.16.1
org.apache.activemq	activemq-mqtt	5.16.1
org.apache.activemq	activemq-openwire-generator	5.16.1
org.apache.activemq	activemq-openwire-legacy	5.16.1
org.apache.activemq	activemq-osgi	5.16.1
org.apache.activemq	activemq-partition	5.16.1
org.apache.activemq	activemq-pool	5.16.1
org.apache.activemq	activemq-ra	5.16.1
org.apache.activemq	activemq-run	5.16.1
org.apache.activemq	activemq-runtime-config	5.16.1
org.apache.activemq	activemq-shiro	5.16.1
org.apache.activemq	activemq-spring	5.16.1
org.apache.activemq	activemq-stomp	5.16.1
org.apache.activemq	activemq-web	5.16.1
org.apache.activemq	artemis-amqp-protocol	2.15.0
org.apache.activemq	artemis-commons	2.15.0
org.apache.activemq	artemis-core-client	2.15.0
org.apache.activemq	artemis-jms-client	2.15.0
org.apache.activemq	artemis-jms-server	2.15.0
org.apache.activemq	artemis-journal	2.15.0
org.apache.activemq	artemis-selector	2.15.0
org.apache.activemq	artemis-server	2.15.0
org.apache.activemq	artemis-service-extensions	2.15.0
org.apache.commons	commons-dbcp2	2.8.0

Group ID	Artifact ID	Version
org.apache.commons	commons-lang3	3.11
org.apache.commons	commons-pool2	2.9.0
org.apache.derby	derby	10.14.2.0
org.apache.derby	derbyclient	10.14.2.0
org.apache.httpcomponents	fluent-hc	4.5.13
org.apache.httpcomponents	httpasyncclient	4.1.4
org.apache.httpcomponents	httpclient	4.5.13
org.apache.httpcomponents	httpclient-cache	4.5.13
org.apache.httpcomponents	httpclient-osgi	4.5.13
org.apache.httpcomponents	httpclient-win	4.5.13
org.apache.httpcomponents	httpcore	4.4.14
org.apache.httpcomponents	httpcore-nio	4.4.14
org.apache.httpcomponents	httpmime	4.5.13
org.apache.johnzon	johnzon-core	1.2.10
org.apache.johnzon	johnzon-jaxrs	1.2.10
org.apache.johnzon	johnzon-jsonb	1.2.10
org.apache.johnzon	johnzon-jsonb-extras	1.2.10
org.apache.johnzon	johnzon-jsonschema	1.2.10
org.apache.johnzon	johnzon-mapper	1.2.10

Group ID	Artifact ID	Version
org.apache.johnzon	johnzon-websocket	1.2.10
org.apache.kafka	connect-api	2.6.0
org.apache.kafka	connect-basic-auth-extension	2.6.0
org.apache.kafka	connect-file	2.6.0
org.apache.kafka	connect-json	2.6.0
org.apache.kafka	connect-runtime	2.6.0
org.apache.kafka	connect-transforms	2.6.0
org.apache.kafka	kafka-clients	2.6.0
org.apache.kafka	kafka-log4j-appender	2.6.0
org.apache.kafka	kafka-streams	2.6.0
org.apache.kafka	kafka-streams-scala_2.12	2.6.0
org.apache.kafka	kafka-streams-scala_2.13	2.6.0
org.apache.kafka	kafka-streams-test-utils	2.6.0
org.apache.kafka	kafka-tools	2.6.0
org.apache.kafka	kafka_2.12	2.6.0
org.apache.kafka	kafka_2.13	2.6.0
org.apache.logging.log4j	log4j-1.2-api	2.13.3
org.apache.logging.log4j	log4j-api	2.13.3
org.apache.logging.log4j	log4j-appserver	2.13.3
org.apache.logging.log4j	log4j-cassandra	2.13.3

Group ID	Artifact ID	Version
org.apache.logging.log4j	log4j-core	2.13.3
org.apache.logging.log4j	log4j-couchdb	2.13.3
org.apache.logging.log4j	log4j-docker	2.13.3
org.apache.logging.log4j	log4j-flume-ng	2.13.3
org.apache.logging.log4j	log4j-iostreams	2.13.3
org.apache.logging.log4j	log4j-jcl	2.13.3
org.apache.logging.log4j	log4j-jmx-gui	2.13.3
org.apache.logging.log4j	log4j-jpa	2.13.3
org.apache.logging.log4j	log4j-jpl	2.13.3
org.apache.logging.log4j	log4j-jul	2.13.3
org.apache.logging.log4j	log4j-kubernetes	2.13.3
org.apache.logging.log4j	log4j-liquibase	2.13.3
org.apache.logging.log4j	log4j-mongodb2	2.13.3
org.apache.logging.log4j	log4j-mongodb3	2.13.3
org.apache.logging.log4j	log4j-slf4j-impl	2.13.3

Group ID	Artifact ID	Version
org.apache.logging.log4j	log4j-slf4j18-impl	2.13.3
org.apache.logging.log4j	log4j-spring-cloud-config-client	2.13.3
org.apache.logging.log4j	log4j-taglib	2.13.3
org.apache.logging.log4j	log4j-to-slf4j	2.13.3
org.apache.logging.log4j	log4j-web	2.13.3
org.apache.solr	solr-analysis-extras	8.5.2
org.apache.solr	solr-analytics	8.5.2
org.apache.solr	solr-cell	8.5.2
org.apache.solr	solr-clustering	8.5.2
org.apache.solr	solr-core	8.5.2
org.apache.solr	solr-dataimporthandler	8.5.2
org.apache.solr	solr-dataimporthandler-extras	8.5.2
org.apache.solr	solr-langid	8.5.2
org.apache.solr	solr-ltr	8.5.2
org.apache.solr	solr-solrj	8.5.2
org.apache.solr	solr-test-framework	8.5.2
org.apache.solr	solr-velocity	8.5.2
org.apache.tomcat	tomcat-annotations-api	9.0.45
org.apache.tomcat	tomcat-jdbc	9.0.45
org.apache.tomcat	tomcat-jsp-api	9.0.45
org.apache.tomcat.embed	tomcat-embed-core	9.0.45

Group ID	Artifact ID	Version
org.apache.tomcat.embed	tomcat-embed-el	9.0.45
org.apache.tomcat.embed	tomcat-embed-jasper	9.0.45
org.apache.tomcat.embed	tomcat-embed-websocket	9.0.45
org.aspectj	aspectjrt	1.9.6
org.aspectj	aspectjtools	1.9.6
org.aspectj	aspectjweaver	1.9.6
org.assertj	assertj-core	3.18.1
org.awaitility	awaitility	4.0.3
org.awaitility	awaitility-groovy	4.0.3
org.awaitility	awaitility-kotlin	4.0.3
org.awaitility	awaitility-scala	4.0.3
org.codehaus.btm	btm	2.1.4
org.codehaus.groovy	groovy	2.5.14
org.codehaus.groovy	groovy-ant	2.5.14
org.codehaus.groovy	groovy-bsf	2.5.14
org.codehaus.groovy	groovy-cli-commons	2.5.14
org.codehaus.groovy	groovy-cli-picocli	2.5.14
org.codehaus.groovy	groovy-console	2.5.14
org.codehaus.groovy	groovy-datetime	2.5.14
org.codehaus.groovy	groovy-dateutil	2.5.14
org.codehaus.groovy	groovy-docgenerator	2.5.14
org.codehaus.groovy	groovy-groovydoc	2.5.14
org.codehaus.groovy	groovy-groovysh	2.5.14
org.codehaus.groovy	groovy-jaxb	2.5.14
org.codehaus.groovy	groovy-jmx	2.5.14
org.codehaus.groovy	groovy-json	2.5.14

Group ID	Artifact ID	Version
org.codehaus.groovy	groovy-json-direct	2.5.14
org.codehaus.groovy	groovy-jsr223	2.5.14
org.codehaus.groovy	groovy-macro	2.5.14
org.codehaus.groovy	groovy-nio	2.5.14
org.codehaus.groovy	groovy-servlet	2.5.14
org.codehaus.groovy	groovy-sql	2.5.14
org.codehaus.groovy	groovy-swing	2.5.14
org.codehaus.groovy	groovy-templates	2.5.14
org.codehaus.groovy	groovy-test	2.5.14
org.codehaus.groovy	groovy-test-junit5	2.5.14
org.codehaus.groovy	groovy-testng	2.5.14
org.codehaus.groovy	groovy-xml	2.5.14
org.codehaus.janino	commons-compiler	3.1.3
org.codehaus.janino	commons-compiler-jdk	3.1.3
org.codehaus.janino	janino	3.1.3
org.eclipse.jetty	apache-jsp	9.4.39.v20210325
org.eclipse.jetty	apache-jstl	9.4.39.v20210325
org.eclipse.jetty	infinispan-common	9.4.39.v20210325
org.eclipse.jetty	infinispan-embedded-query	9.4.39.v20210325
org.eclipse.jetty	infinispan-remote-query	9.4.39.v20210325
org.eclipse.jetty	jetty-alpn-client	9.4.39.v20210325
org.eclipse.jetty	jetty-alpn-conscrypt-client	9.4.39.v20210325
org.eclipse.jetty	jetty-alpn-conscrypt-server	9.4.39.v20210325
org.eclipse.jetty	jetty-alpn-java-client	9.4.39.v20210325

Group ID	Artifact ID	Version
org.eclipse.jetty	jetty-alpn-java-server	9.4.39.v20210325
org.eclipse.jetty	jetty-alpn-openjdk8-client	9.4.39.v20210325
org.eclipse.jetty	jetty-alpn-openjdk8-server	9.4.39.v20210325
org.eclipse.jetty	jetty-alpn-server	9.4.39.v20210325
org.eclipse.jetty	jetty-annotations	9.4.39.v20210325
org.eclipse.jetty	jetty-ant	9.4.39.v20210325
org.eclipse.jetty	jetty-client	9.4.39.v20210325
org.eclipse.jetty	jetty-continuation	9.4.39.v20210325
org.eclipse.jetty	jetty-deploy	9.4.39.v20210325
org.eclipse.jetty	jetty-distribution	9.4.39.v20210325
org.eclipse.jetty	jetty-hazelcast	9.4.39.v20210325
org.eclipse.jetty	jetty-home	9.4.39.v20210325
org.eclipse.jetty	jetty-http	9.4.39.v20210325
org.eclipse.jetty	jetty-http-spi	9.4.39.v20210325
org.eclipse.jetty	jetty-io	9.4.39.v20210325
org.eclipse.jetty	jetty-jaas	9.4.39.v20210325
org.eclipse.jetty	jetty-jaspi	9.4.39.v20210325
org.eclipse.jetty	jetty-jmx	9.4.39.v20210325
org.eclipse.jetty	jetty-jndi	9.4.39.v20210325
org.eclipse.jetty	jetty-nosql	9.4.39.v20210325
org.eclipse.jetty	jetty-openid	9.4.39.v20210325
org.eclipse.jetty	jetty-plus	9.4.39.v20210325
org.eclipse.jetty	jetty-proxy	9.4.39.v20210325
org.eclipse.jetty	jetty-quickstart	9.4.39.v20210325

Group ID	Artifact ID	Version
org.eclipse.jetty	jetty-reactive-httpclient	1.1.7
org.eclipse.jetty	jetty-rewrite	9.4.39.v20210325
org.eclipse.jetty	jetty-security	9.4.39.v20210325
org.eclipse.jetty	jetty-server	9.4.39.v20210325
org.eclipse.jetty	jetty-servlet	9.4.39.v20210325
org.eclipse.jetty	jetty-servlets	9.4.39.v20210325
org.eclipse.jetty	jetty-spring	9.4.39.v20210325
org.eclipse.jetty	jetty-unixsocket	9.4.39.v20210325
org.eclipse.jetty	jetty-util	9.4.39.v20210325
org.eclipse.jetty	jetty-util-ajax	9.4.39.v20210325
org.eclipse.jetty	jetty-webapp	9.4.39.v20210325
org.eclipse.jetty	jetty-xml	9.4.39.v20210325
org.eclipse.jetty.fcg	fcgi-client	9.4.39.v20210325
org.eclipse.jetty.fcg	fcgi-server	9.4.39.v20210325
org.eclipse.jetty.gclou	jetty-gcloud-session-	9.4.39.v20210325
d	manager	
org.eclipse.jetty.http2	http2-client	9.4.39.v20210325
org.eclipse.jetty.http2	http2-common	9.4.39.v20210325
org.eclipse.jetty.http2	http2-hpack	9.4.39.v20210325
org.eclipse.jetty.http2	http2-http-client-	9.4.39.v20210325
	transport	
org.eclipse.jetty.http2	http2-server	9.4.39.v20210325
org.eclipse.jetty.memca	jetty-memcached-	9.4.39.v20210325
ched	sessions	
org.eclipse.jetty.orbit	javax.servlet.jsp	2.2.0.v201112011158
org.eclipse.jetty.osgi	jetty-httpservice	9.4.39.v20210325

Group ID	Artifact ID	Version
org.eclipse.jetty.osgi	jetty-osgi-boot	9.4.39.v20210325
org.eclipse.jetty.osgi	jetty-osgi-boot-jsp	9.4.39.v20210325
org.eclipse.jetty.osgi	jetty-osgi-boot-warurl	9.4.39.v20210325
org.eclipse.jetty.websocket	javax.websocket-client-impl	9.4.39.v20210325
org.eclipse.jetty.websocket	javax.websocket-server-impl	9.4.39.v20210325
org.eclipse.jetty.websocket	websocket-api	9.4.39.v20210325
org.eclipse.jetty.websocket	websocket-client	9.4.39.v20210325
org.eclipse.jetty.websocket	websocket-common	9.4.39.v20210325
org.eclipse.jetty.websocket	websocket-server	9.4.39.v20210325
org.eclipse.jetty.websocket	websocket-servlet	9.4.39.v20210325
org.ehcache	ehcache	3.9.2
org.ehcache	ehcache-clustered	3.9.2
org.ehcache	ehcache-transactions	3.9.2
org.elasticsearch	elasticsearch	7.9.3
org.elasticsearch.client	elasticsearch-rest-client	7.9.3
org.elasticsearch.client	elasticsearch-rest-client-sniffer	7.9.3
org.elasticsearch.client	elasticsearch-rest-high-level-client	7.9.3
org.elasticsearch.client	transport	7.9.3

Group ID	Artifact ID	Version
org.elasticsearch.distribution.integ-test-zip	elasticsearch	7.9.3
org.elasticsearch.plugin	transport-netty4-client	7.9.3
org.firebirdsql.jdbc	jaybird-jdk17	3.0.11
org.firebirdsql.jdbc	jaybird-jdk18	3.0.11
org.flywaydb	flyway-core	7.1.1
org.freemarker	freemarker	2.3.31
org.glassfish	jakarta.el	3.0.3
org.glassfish.jaxb	codemodel	2.3.4
org.glassfish.jaxb	codemodel-annotation-compiler	2.3.4
org.glassfish.jaxb	jaxb-jxc	2.3.4
org.glassfish.jaxb	jaxb-runtime	2.3.4
org.glassfish.jaxb	jaxb-xjc	2.3.4
org.glassfish.jaxb	txw2	2.3.4
org.glassfish.jaxb	txwc2	2.3.4
org.glassfish.jaxb	xsom	2.3.4
org.glassfish.jersey.bundles	jaxrs-ri	2.32
org.glassfish.jersey.connectors	jersey-apache-connector	2.32
org.glassfish.jersey.connectors	jersey-grizzly-connector	2.32
org.glassfish.jersey.connectors	jersey-helidon-connector	2.32
org.glassfish.jersey.connectors	jersey-jdk-connector	2.32

Group ID	Artifact ID	Version
org.glassfish.jersey.co nnectors	jersey-jetty-connector	2.32
org.glassfish.jersey.co nnectors	jersey-netty-connector	2.32
org.glassfish.jersey.co ntainers	jersey-container- grizzly2-http	2.32
org.glassfish.jersey.co ntainers	jersey-container- grizzly2-servlet	2.32
org.glassfish.jersey.co ntainers	jersey-container-jdk- http	2.32
org.glassfish.jersey.co ntainers	jersey-container-jetty- http	2.32
org.glassfish.jersey.co ntainers	jersey-container-jetty- servlet	2.32
org.glassfish.jersey.co ntainers	jersey-container-netty- http	2.32
org.glassfish.jersey.co ntainers	jersey-container- servlet	2.32
org.glassfish.jersey.co ntainers	jersey-container- servlet-core	2.32
org.glassfish.jersey.co ntainers	jersey-container- simple-http	2.32
org.glassfish.jersey.co ntainers.glassfish	jersey-gf-ejb	2.32
org.glassfish.jersey.co re	jersey-client	2.32
org.glassfish.jersey.co re	jersey-common	2.32
org.glassfish.jersey.co re	jersey-server	2.32

Group ID	Artifact ID	Version
org.glassfish.jersey.ext	jersey-bean-validation	2.32
org.glassfish.jersey.ext	jersey-declarative-linking	2.32
org.glassfish.jersey.ext	jersey-entity-filtering	2.32
org.glassfish.jersey.ext	jersey-metainf-services	2.32
org.glassfish.jersey.ext	jersey-mvc	2.32
org.glassfish.jersey.ext	jersey-mvc-bean-validation	2.32
org.glassfish.jersey.ext	jersey-mvc-freemarker	2.32
org.glassfish.jersey.ext	jersey-mvc-jsp	2.32
org.glassfish.jersey.ext	jersey-mvc-mustache	2.32
org.glassfish.jersey.ext	jersey-proxy-client	2.32
org.glassfish.jersey.ext	jersey-servlet-portability	2.32
org.glassfish.jersey.ext	jersey-spring4	2.32
org.glassfish.jersey.ext	jersey-spring5	2.32
org.glassfish.jersey.ext	jersey-wadl-doclet	2.32
org.glassfish.jersey.ext.cdi	jersey-cdi1x	2.32

Group ID	Artifact ID	Version
org.glassfish.jersey.ext.cdi	jersey-cdi1x-ban-custom-hk2-binding	2.32
org.glassfish.jersey.ext.cdi	jersey-cdi1x-servlet	2.32
org.glassfish.jersey.ext.cdi	jersey-cdi1x-transaction	2.32
org.glassfish.jersey.ext.cdi	jersey-cdi1x-validation	2.32
org.glassfish.jersey.ext.cdi	jersey-weld2-se	2.32
org.glassfish.jersey.ext.microprofile	jersey-mp-config	2.32
org.glassfish.jersey.ext.microprofile	jersey-mp-rest-client	2.32
org.glassfish.jersey.ext.rx	jersey-rx-client-guava	2.32
org.glassfish.jersey.ext.rx	jersey-rx-client-rxjava	2.32
org.glassfish.jersey.ext.rx	jersey-rx-client-rxjava2	2.32
org.glassfish.jersey.inject	jersey-cdi2-se	2.32
org.glassfish.jersey.inject	jersey-hk2	2.32
org.glassfish.jersey.media	jersey-media-jaxb	2.32
org.glassfish.jersey.media	jersey-media-json-binding	2.32
org.glassfish.jersey.media	jersey-media-json-jackson	2.32

Group ID	Artifact ID	Version
org.glassfish.jersey.me dia	jersey-media-json-jettison	2.32
org.glassfish.jersey.me dia	jersey-media-json-processing	2.32
org.glassfish.jersey.me dia	jersey-media-kryo	2.32
org.glassfish.jersey.me dia	jersey-media-moxy	2.32
org.glassfish.jersey.me dia	jersey-media-multipart	2.32
org.glassfish.jersey.me dia	jersey-media-sse	2.32
org.glassfish.jersey.se curity	oauth1-client	2.32
org.glassfish.jersey.se curity	oauth1-server	2.32
org.glassfish.jersey.se curity	oauth1-signature	2.32
org.glassfish.jersey.se curity	oauth2-client	2.32
org.glassfish.jersey.te st-framework	jersey-test-framework-core	2.32
org.glassfish.jersey.te st-framework	jersey-test-framework-util	2.32
org.glassfish.jersey.te st-framework.providers	jersey-test-framework-provider-bundle	2.32
org.glassfish.jersey.te st-framework.providers	jersey-test-framework-provider-external	2.32
org.glassfish.jersey.te st-framework.providers	jersey-test-framework-provider-grizzly2	2.32

Group ID	Artifact ID	Version
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-inmemory	2.32
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-jdk-http	2.32
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-jetty	2.32
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-simple	2.32
org.hamcrest	hamcrest	2.2
org.hamcrest	hamcrest-core	2.2
org.hamcrest	hamcrest-library	2.2
org.hibernate	hibernate-c3p0	5.4.30.Final
org.hibernate	hibernate-core	5.4.30.Final
org.hibernate	hibernate-ehcache	5.4.30.Final
org.hibernate	hibernate-entitymanager	5.4.30.Final
org.hibernate	hibernate-envers	5.4.30.Final
org.hibernate	hibernate-hikaricp	5.4.30.Final
org.hibernate	hibernate-java8	5.4.30.Final
org.hibernate	hibernate-jcache	5.4.30.Final
org.hibernate	hibernate-jpamodelgen	5.4.30.Final
org.hibernate	hibernate-micrometer	5.4.30.Final
org.hibernate	hibernate-proxool	5.4.30.Final
org.hibernate	hibernate-spatial	5.4.30.Final
org.hibernate	hibernate-testing	5.4.30.Final
org.hibernate	hibernate-vibur	5.4.30.Final
org.hibernate.validator	hibernate-validator	6.1.7.Final

Group ID	Artifact ID	Version
org.hibernate.validator	hibernate-validator-annotation-processor	6.1.7.Final
org.hsqldb	hsqldb	2.5.2
org.infinispan	infinispan-anchored-keys	11.0.10.Final
org.infinispan	infinispan-api	11.0.10.Final
org.infinispan	infinispan-cachestore-jdbc	11.0.10.Final
org.infinispan	infinispan-cachestore-jpa	11.0.10.Final
org.infinispan	infinispan-cachestore-remote	11.0.10.Final
org.infinispan	infinispan-cachestore-rest	11.0.10.Final
org.infinispan	infinispan-cachestore-rocksdb	11.0.10.Final
org.infinispan	infinispan-cdi-common	11.0.10.Final
org.infinispan	infinispan-cdi-embedded	11.0.10.Final
org.infinispan	infinispan-cdi-remote	11.0.10.Final
org.infinispan	infinispan-checkstyle	11.0.10.Final
org.infinispan	infinispan-cli-client	11.0.10.Final
org.infinispan	infinispan-client-hotrod	11.0.10.Final
org.infinispan	infinispan-client-rest	11.0.10.Final
org.infinispan	infinispan-clustered-counter	11.0.10.Final
org.infinispan	infinispan-clustered-lock	11.0.10.Final

Group ID	Artifact ID	Version
org.infinispan	infinispan-commons	11.0.10.Final
org.infinispan	infinispan-commons-test	11.0.10.Final
org.infinispan	infinispan-component-annotations	11.0.10.Final
org.infinispan	infinispan-component-processor	11.0.10.Final
org.infinispan	infinispan-console	0.7.2.Final
org.infinispan	infinispan-core	11.0.10.Final
org.infinispan	infinispan-extended-statistics	11.0.10.Final
org.infinispan	infinispan-hibernate-cache-commons	11.0.10.Final
org.infinispan	infinispan-hibernate-cache-spi	11.0.10.Final
org.infinispan	infinispan-hibernate-cache-v51	11.0.10.Final
org.infinispan	infinispan-hibernate-cache-v53	11.0.10.Final
org.infinispan	infinispan-jboss-marshalling	11.0.10.Final
org.infinispan	infinispan-jcache	11.0.10.Final
org.infinispan	infinispan-jcache-commons	11.0.10.Final
org.infinispan	infinispan-jcache-remote	11.0.10.Final
org.infinispan	infinispan-key-value-store-client	11.0.10.Final
org.infinispan	infinispan-marshaller-kryo	11.0.10.Final

Group ID	Artifact ID	Version
org.infinispan	infinispan-marshaller-kryo-bundle	11.0.10.Final
org.infinispan	infinispan-marshaller-protostuff	11.0.10.Final
org.infinispan	infinispan-marshaller-protostuff-bundle	11.0.10.Final
org.infinispan	infinispan-multimap	11.0.10.Final
org.infinispan	infinispan-objectfilter	11.0.10.Final
org.infinispan	infinispan-osgi	11.0.10.Final
org.infinispan	infinispan-persistence-soft-index	11.0.10.Final
org.infinispan	infinispan-query	11.0.10.Final
org.infinispan	infinispan-query-core	11.0.10.Final
org.infinispan	infinispan-query-dsl	11.0.10.Final
org.infinispan	infinispan-remote-query-client	11.0.10.Final
org.infinispan	infinispan-remote-query-server	11.0.10.Final
org.infinispan	infinispan-scripting	11.0.10.Final
org.infinispan	infinispan-server-core	11.0.10.Final
org.infinispan	infinispan-server-hotrod	11.0.10.Final
org.infinispan	infinispan-server-memcached	11.0.10.Final
org.infinispan	infinispan-server-rest	11.0.10.Final
org.infinispan	infinispan-server-router	11.0.10.Final

Group ID	Artifact ID	Version
org.infinispan	infinispan-server-runtime	11.0.10.Final
org.infinispan	infinispan-spring5-common	11.0.10.Final
org.infinispan	infinispan-spring5-embedded	11.0.10.Final
org.infinispan	infinispan-spring5-remote	11.0.10.Final
org.infinispan	infinispan-tasks	11.0.10.Final
org.infinispan	infinispan-tasks-api	11.0.10.Final
org.infinispan	infinispan-tools	11.0.10.Final
org.infinispan.protobuf	protostream-eam	4.3.4.Final
org.infinispan.protobuf	protostream-processor-eam	4.3.4.Final
org.influxdb	influxdb-java	2.20
org.jboss	jboss-transaction-spi	7.6.0.Final
org.jboss.logging	jboss-logging	3.4.1.Final
org.jdom	jdom2	2.0.6
org.jetbrains.kotlin	kotlin-compiler	1.4.32
org.jetbrains.kotlin	kotlin-compiler-embeddable	1.4.32
org.jetbrains.kotlin	kotlin-daemon-client	1.4.32
org.jetbrains.kotlin	kotlin-main-kts	1.4.32
org.jetbrains.kotlin	kotlin-osgi-bundle	1.4.32
org.jetbrains.kotlin	kotlin-reflect	1.4.32
org.jetbrains.kotlin	kotlin-script-runtime	1.4.32
org.jetbrains.kotlin	kotlin-script-util	1.4.32

Group ID	Artifact ID	Version
org.jetbrains.kotlin	kotlin-scripting-common	1.4.32
org.jetbrains.kotlin	kotlin-scripting-ide-services	1.4.32
org.jetbrains.kotlin	kotlin-scripting-jvm	1.4.32
org.jetbrains.kotlin	kotlin-scripting-jvm-host	1.4.32
org.jetbrains.kotlin	kotlin-stdlib	1.4.32
org.jetbrains.kotlin	kotlin-stdlib-common	1.4.32
org.jetbrains.kotlin	kotlin-stdlib-jdk7	1.4.32
org.jetbrains.kotlin	kotlin-stdlib-jdk8	1.4.32
org.jetbrains.kotlin	kotlin-stdlib-js	1.4.32
org.jetbrains.kotlin	kotlin-test	1.4.32
org.jetbrains.kotlin	kotlin-test-annotations-common	1.4.32
org.jetbrains.kotlin	kotlin-test-common	1.4.32
org.jetbrains.kotlin	kotlin-test-js	1.4.32
org.jetbrains.kotlin	kotlin-test-junit	1.4.32
org.jetbrains.kotlin	kotlin-test-junit5	1.4.32
org.jetbrains.kotlin	kotlin-test-testng	1.4.32
org.jetbrains.kotlinx	kotlinx-coroutines-android	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-core	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-core-jvm	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-debug	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-guava	1.4.3

Group ID	Artifact ID	Version
org.jetbrains.kotlinx	kotlinx-coroutines-javafx	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-jdk8	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-jdk9	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-play-services	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-reactive	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-reactor	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-rx2	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-rx3	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-slf4j	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-swing	1.4.3
org.jetbrains.kotlinx	kotlinx-coroutines-test	1.4.3
org.jolokia	jolokia-core	1.6.2
org.jooq	jooq	3.14.8
org.jooq	jooq-codegen	3.14.8
org.jooq	jooq-kotlin	3.14.8
org.jooq	jooq-meta	3.14.8
org.junit.jupiter	junit-jupiter	5.7.1
org.junit.jupiter	junit-jupiter-api	5.7.1
org.junit.jupiter	junit-jupiter-engine	5.7.1
org.junit.jupiter	junit-jupiter-migrationsupport	5.7.1
org.junit.jupiter	junit-jupiter-params	5.7.1

Group ID	Artifact ID	Version
org.junit.platform	junit-platform-commons	1.7.1
org.junit.platform	junit-platform-console	1.7.1
org.junit.platform	junit-platform-engine	1.7.1
org.junit.platform	junit-platform-jfr	1.7.1
org.junit.platform	junit-platform-launcher	1.7.1
org.junit.platform	junit-platform-reporting	1.7.1
org.junit.platform	junit-platform-runner	1.7.1
org.junit.platform	junit-platform-suite-api	1.7.1
org.junit.platform	junit-platform-testkit	1.7.1
org.junit.vintage	junit-vintage-engine	5.7.1
org.jvnet.mimepull	mimepull	1.9.14
org.liquibase	liquibase-core	3.10.3
org.mariadb	r2dbc-mariadb	1.0.0
org.mariadb.jdbc	mariadb-java-client	2.7.2
org.messaginghub	pooled-jms	1.2.1
org.mockito	mockito-core	3.6.28
org.mockito	mockito-inline	3.6.28
org.mockito	mockito-junit-jupiter	3.6.28
org.mongodb	bson	4.1.2
org.mongodb	mongodb-driver-core	4.1.2
org.mongodb	mongodb-driver-legacy	4.1.2
org.mongodb	mongodb-driver-reactivestreams	4.1.2
org.mongodb	mongodb-driver-sync	4.1.2
org.mortbay.jasper	apache-el	8.5.54

Group ID	Artifact ID	Version
org.neo4j.driver	neo4j-java-driver	4.1.1
org.postgresql	postgresql	42.2.19
org.projectlombok	lombok	1.18.20
org.quartz-scheduler	quartz	2.3.2
org.quartz-scheduler	quartz-jobs	2.3.2
org.reactivestreams	reactive-streams	1.0.3
org.seleniumhq.selenium	htmlunit-driver	2.44.0
org.seleniumhq.selenium	selenium-api	3.141.59
org.seleniumhq.selenium	selenium-chrome-driver	3.141.59
org.seleniumhq.selenium	selenium-edge-driver	3.141.59
org.seleniumhq.selenium	selenium-firefox-driver	3.141.59
org.seleniumhq.selenium	selenium-ie-driver	3.141.59
org.seleniumhq.selenium	selenium-java	3.141.59
org.seleniumhq.selenium	selenium-opera-driver	3.141.59
org.seleniumhq.selenium	selenium-remote-driver	3.141.59
org.seleniumhq.selenium	selenium-safari-driver	3.141.59
org.seleniumhq.selenium	selenium-support	3.141.59
org.skyscreamer	jsonassert	1.5.0
org.slf4j	jcl-over-slf4j	1.7.30
org.slf4j	jul-to-slf4j	1.7.30
org.slf4j	log4j-over-slf4j	1.7.30
org.slf4j	slf4j-api	1.7.30
org.slf4j	slf4j-ext	1.7.30
org.slf4j	slf4j-jcl	1.7.30
org.slf4j	slf4j-jdk14	1.7.30
org.slf4j	slf4j-log4j12	1.7.30

Group ID	Artifact ID	Version
org.slf4j	slf4j-nop	1.7.30
org.slf4j	slf4j-simple	1.7.30
org.springframework	spring-aop	5.3.6
org.springframework	spring-aspects	5.3.6
org.springframework	spring-beans	5.3.6
org.springframework	spring-context	5.3.6
org.springframework	spring-context-indexer	5.3.6
org.springframework	spring-context-support	5.3.6
org.springframework	spring-core	5.3.6
org.springframework	spring-expression	5.3.6
org.springframework	spring-instrument	5.3.6
org.springframework	spring-jcl	5.3.6
org.springframework	spring-jdbc	5.3.6
org.springframework	spring-jms	5.3.6
org.springframework	spring-messaging	5.3.6
org.springframework	spring-orm	5.3.6
org.springframework	spring-oxm	5.3.6
org.springframework	spring-r2dbc	5.3.6
org.springframework	spring-test	5.3.6
org.springframework	spring-tx	5.3.6
org.springframework	spring-web	5.3.6
org.springframework	spring-webflux	5.3.6
org.springframework	spring-webmvc	5.3.6
org.springframework	spring-websocket	5.3.6
org.springframework.amqp	spring-amqp	2.3.6

Group ID	Artifact ID	Version
org.springframework.amqp	spring-rabbit	2.3.6
org.springframework.amqp	spring-rabbit-junit	2.3.6
org.springframework.amqp	spring-rabbit-test	2.3.6
org.springframework.batch	spring-batch-core	4.3.2
org.springframework.batch	spring-batch-infrastructure	4.3.2
org.springframework.batch	spring-batch-integration	4.3.2
org.springframework.batch	spring-batch-test	4.3.2
org.springframework.boot	spring-boot	2.4.5
org.springframework.boot	spring-boot-actuator	2.4.5
org.springframework.boot	spring-boot-actuator-autoconfigure	2.4.5
org.springframework.boot	spring-boot-autoconfigure	2.4.5
org.springframework.boot	spring-boot-autoconfigure-processor	2.4.5
org.springframework.boot	spring-boot-buildpack-platform	2.4.5
org.springframework.boot	spring-boot-configuration-metadata	2.4.5
org.springframework.boot	spring-boot-configuration-processor	2.4.5

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-devtools	2.4.5
org.springframework.boot	spring-boot-jarmode-layertools	2.4.5
org.springframework.boot	spring-boot-loader	2.4.5
org.springframework.boot	spring-boot-loader-tools	2.4.5
org.springframework.boot	spring-boot-properties-migrator	2.4.5
org.springframework.boot	spring-boot-starter	2.4.5
org.springframework.boot	spring-boot-starter-activemq	2.4.5
org.springframework.boot	spring-boot-starter-actuator	2.4.5
org.springframework.boot	spring-boot-starter-amqp	2.4.5
org.springframework.boot	spring-boot-starter-aop	2.4.5
org.springframework.boot	spring-boot-starter-artemis	2.4.5
org.springframework.boot	spring-boot-starter-batch	2.4.5
org.springframework.boot	spring-boot-starter-cache	2.4.5
org.springframework.boot	spring-boot-starter-data-cassandra	2.4.5
org.springframework.boot	spring-boot-starter-data-cassandra-reactive	2.4.5

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-data-couchbase	2.4.5
org.springframework.boot	spring-boot-starter-data-couchbase-reactive	2.4.5
org.springframework.boot	spring-boot-starter-data-elasticsearch	2.4.5
org.springframework.boot	spring-boot-starter-data-jdbc	2.4.5
org.springframework.boot	spring-boot-starter-data-jpa	2.4.5
org.springframework.boot	spring-boot-starter-data-ldap	2.4.5
org.springframework.boot	spring-boot-starter-data-mongodb	2.4.5
org.springframework.boot	spring-boot-starter-data-mongodb-reactive	2.4.5
org.springframework.boot	spring-boot-starter-data-neo4j	2.4.5
org.springframework.boot	spring-boot-starter-data-r2dbc	2.4.5
org.springframework.boot	spring-boot-starter-data-redis	2.4.5
org.springframework.boot	spring-boot-starter-data-redis-reactive	2.4.5
org.springframework.boot	spring-boot-starter-data-rest	2.4.5
org.springframework.boot	spring-boot-starter-data-solr	2.4.5
org.springframework.boot	spring-boot-starter-freemarker	2.4.5

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-groovy-templates	2.4.5
org.springframework.boot	spring-boot-starter-hateoas	2.4.5
org.springframework.boot	spring-boot-starter-integration	2.4.5
org.springframework.boot	spring-boot-starter-jdbc	2.4.5
org.springframework.boot	spring-boot-starter-jersey	2.4.5
org.springframework.boot	spring-boot-starter-jetty	2.4.5
org.springframework.boot	spring-boot-starter-jooq	2.4.5
org.springframework.boot	spring-boot-starter-json	2.4.5
org.springframework.boot	spring-boot-starter-jta-atomikos	2.4.5
org.springframework.boot	spring-boot-starter-jta-bitronix	2.4.5
org.springframework.boot	spring-boot-starter-log4j2	2.4.5
org.springframework.boot	spring-boot-starter-logging	2.4.5
org.springframework.boot	spring-boot-starter-mail	2.4.5
org.springframework.boot	spring-boot-starter-mustache	2.4.5
org.springframework.boot	spring-boot-starter-oauth2-client	2.4.5

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-oauth2-resource-server	2.4.5
org.springframework.boot	spring-boot-starter-quartz	2.4.5
org.springframework.boot	spring-boot-starter-reactor-netty	2.4.5
org.springframework.boot	spring-boot-starter-rsocket	2.4.5
org.springframework.boot	spring-boot-starter-security	2.4.5
org.springframework.boot	spring-boot-starter-test	2.4.5
org.springframework.boot	spring-boot-starter-thymeleaf	2.4.5
org.springframework.boot	spring-boot-starter-tomcat	2.4.5
org.springframework.boot	spring-boot-starter-undertow	2.4.5
org.springframework.boot	spring-boot-starter-validation	2.4.5
org.springframework.boot	spring-boot-starter-web	2.4.5
org.springframework.boot	spring-boot-starter-web-services	2.4.5
org.springframework.boot	spring-boot-starter-webflux	2.4.5
org.springframework.boot	spring-boot-starter-websocket	2.4.5
org.springframework.boot	spring-boot-test	2.4.5

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-test-autoconfigure	2.4.5
org.springframework.data	spring-data-cassandra	3.1.8
org.springframework.data	spring-data-commons	2.4.8
org.springframework.data	spring-data-couchbase	4.1.8
org.springframework.data	spring-data-elasticsearch	4.1.8
org.springframework.data	spring-data-envers	2.4.8
org.springframework.data	spring-data-geode	2.4.8
org.springframework.data	spring-data-jdbc	2.1.8
org.springframework.data	spring-data-jpa	2.4.8
org.springframework.data	spring-data-keyvalue	2.4.8
org.springframework.data	spring-data-ldap	2.4.8
org.springframework.data	spring-data-mongodb	3.1.8
org.springframework.data	spring-data-neo4j	6.0.8
org.springframework.data	spring-data-r2dbc	1.2.8
org.springframework.data	spring-data-redis	2.4.8

Group ID	Artifact ID	Version
org.springframework.dat a	spring-data-relational	2.1.8
org.springframework.dat a	spring-data-rest-core	3.4.8
org.springframework.dat a	spring-data-rest-hal- explorer	3.4.8
org.springframework.dat a	spring-data-rest-webmvc	3.4.8
org.springframework.dat a	spring-data-solr	4.3.8
org.springframework.hat eoas	spring-hateoas	1.2.5
org.springframework.int egration	spring-integration-amqp	5.4.6
org.springframework.int egration	spring-integration-core	5.4.6
org.springframework.int egration	spring-integration- event	5.4.6
org.springframework.int egration	spring-integration-feed	5.4.6
org.springframework.int egration	spring-integration-file	5.4.6
org.springframework.int egration	spring-integration-ftp	5.4.6
org.springframework.int egration	spring-integration- gemfire	5.4.6
org.springframework.int egration	spring-integration- groovy	5.4.6
org.springframework.int egration	spring-integration-http	5.4.6

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-ip	5.4.6
org.springframework.integration	spring-integration-jdbc	5.4.6
org.springframework.integration	spring-integration-jms	5.4.6
org.springframework.integration	spring-integration-jmx	5.4.6
org.springframework.integration	spring-integration-jpa	5.4.6
org.springframework.integration	spring-integration-kafka	5.4.6
org.springframework.integration	spring-integration-mail	5.4.6
org.springframework.integration	spring-integration-mongodb	5.4.6
org.springframework.integration	spring-integration-mqtt	5.4.6
org.springframework.integration	spring-integration-r2dbc	5.4.6
org.springframework.integration	spring-integration-redis	5.4.6
org.springframework.integration	spring-integration-rmi	5.4.6
org.springframework.integration	spring-integration-rsocket	5.4.6
org.springframework.integration	spring-integration-scripting	5.4.6
org.springframework.integration	spring-integration-security	5.4.6

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-sftp	5.4.6
org.springframework.integration	spring-integration-stomp	5.4.6
org.springframework.integration	spring-integration-stream	5.4.6
org.springframework.integration	spring-integration-syslog	5.4.6
org.springframework.integration	spring-integration-test	5.4.6
org.springframework.integration	spring-integration-test-support	5.4.6
org.springframework.integration	spring-integration-webflux	5.4.6
org.springframework.integration	spring-integration-websocket	5.4.6
org.springframework.integration	spring-integration-ws	5.4.6
org.springframework.integration	spring-integration-xml	5.4.6
org.springframework.integration	spring-integration-xmpp	5.4.6
org.springframework.integration	spring-integration-zeromq	5.4.6
org.springframework.integration	spring-integration-zookeeper	5.4.6
org.springframework.kafka	spring-kafka	2.6.7
org.springframework.kafka	spring-kafka-test	2.6.7

Group ID	Artifact ID	Version
org.springframework.ldap p	spring-ldap-core	2.3.3.RELEASE
org.springframework.ldap p	spring-ldap-core-tiger	2.3.3.RELEASE
org.springframework.ldap p	spring-ldap-ldif-batch	2.3.3.RELEASE
org.springframework.ldap p	spring-ldap-ldif-core	2.3.3.RELEASE
org.springframework.ldap p	spring-ldap-odm	2.3.3.RELEASE
org.springframework.ldap p	spring-ldap-test	2.3.3.RELEASE
org.springframework.restdocs tdocs	spring-restdocs- asciidoc	2.0.5.RELEASE
org.springframework.restdocs tdocs	spring-restdocs-core	2.0.5.RELEASE
org.springframework.restdocs tdocs	spring-restdocs-mockmvc	2.0.5.RELEASE
org.springframework.restdocs tdocs	spring-restdocs- restassured	2.0.5.RELEASE
org.springframework.restdocs tdocs	spring-restdocs- webtestclient	2.0.5.RELEASE
org.springframework.retry	spring-retry	1.3.1
org.springframework.security urity	spring-security-acl	5.4.6
org.springframework.security urity	spring-security-aspects	5.4.6
org.springframework.security urity	spring-security-cas	5.4.6

Group ID	Artifact ID	Version
org.springframework.security	spring-security-config	5.4.6
org.springframework.security	spring-security-core	5.4.6
org.springframework.security	spring-security-crypto	5.4.6
org.springframework.security	spring-security-data	5.4.6
org.springframework.security	spring-security-ldap	5.4.6
org.springframework.security	spring-security-messaging	5.4.6
org.springframework.security	spring-security-oauth2-client	5.4.6
org.springframework.security	spring-security-oauth2-core	5.4.6
org.springframework.security	spring-security-oauth2-jose	5.4.6
org.springframework.security	spring-security-oauth2-resource-server	5.4.6
org.springframework.security	spring-security-openid	5.4.6
org.springframework.security	spring-security-remoting	5.4.6
org.springframework.security	spring-security-rsocket	5.4.6
org.springframework.security	spring-security-saml2-service-provider	5.4.6
org.springframework.security	spring-security-taglibs	5.4.6

Group ID	Artifact ID	Version
org.springframework.security	spring-security-test	5.4.6
org.springframework.security	spring-security-web	5.4.6
org.springframework.session	spring-session-core	2.4.3
org.springframework.session	spring-session-data-geode	2.4.3
org.springframework.session	spring-session-data-mongodb	2.4.4
org.springframework.session	spring-session-data-redis	2.4.3
org.springframework.session	spring-session-hazelcast	2.4.3
org.springframework.session	spring-session-jdbc	2.4.3
org.springframework.ws	spring-ws-core	3.0.10.RELEASE
org.springframework.ws	spring-ws-security	3.0.10.RELEASE
org.springframework.ws	spring-ws-support	3.0.10.RELEASE
org.springframework.ws	spring-ws-test	3.0.10.RELEASE
org.springframework.ws	spring-xml	3.0.10.RELEASE
org.thymeleaf	thymeleaf	3.0.12.RELEASE
org.thymeleaf	thymeleaf-spring5	3.0.12.RELEASE
org.thymeleaf.extras	thymeleaf-extras-java8time	3.0.4.RELEASE
org.thymeleaf.extras	thymeleaf-extras-springsecurity5	3.0.4.RELEASE
org.webjars	hal-browser	3325375

Group ID	Artifact ID	Version
org.webjars	webjars-locator-core	0.46
org.xerial	sqlite-jdbc	3.32.3.3
org.xmlunit	xmlunit-assertj	2.7.0
org.xmlunit	xmlunit-core	2.7.0
org.xmlunit	xmlunit-legacy	2.7.0
org.xmlunit	xmlunit-matchers	2.7.0
org.xmlunit	xmlunit-placeholders	2.7.0
org.yaml	snakeyaml	1.27
redis.clients	jedis	3.3.0
wsdl4j	wsdl4j	1.6.3

11.F.2. Version Properties

下表提供了可用于覆盖 Spring Boot 管理的版本的所有属性。浏览 [spring-boot-dependencies build.gradle](#) 以获取依赖关系的完整列表。

Library	Version Property
ActiveMQ	activemq.version
ANTLR2	antlr2.version
AppEngine SDK	appengine-sdk.version
Artemis	artemis.version
AspectJ	aspectj.version
AssertJ	assertj.version
Atomikos	atomikos.version
Awaitility	awaitility.version
Bitronix	bitronix.version
Build Helper Maven Plugin	build-helper-maven-plugin.version
Byte Buddy	byte-buddy.version

Library	Version Property
Caffeine	caffeine.version
Cassandra Driver	cassandra-driver.version
Classmate	classmate.version
Commons Codec	commons-codec.version
Commons DBCP2	commons-dbcp2.version
Commons Lang3	commons-lang3.version
Commons Pool	commons-pool.version
Commons Pool2	commons-pool2.version
Couchbase Client	couchbase-client.version
DB2 JDBC	db2-jdbc.version
Dependency Management Plugin	dependency-management-plugin.version
Derby	derby.version
Dropwizard Metrics	dropwizard-metrics.version
Ehcache	ehcache.version
Ehcache3	ehcache3.version
Elasticsearch	elasticsearch.version
Embedded Mongo	embedded-mongo.version
Flyway	flyway.version
FreeMarker	freemarker.version
Git Commit ID Plugin	git-commit-id-plugin.version
Glassfish EL	glassfish-el.version
Glassfish JAXB	glassfish-jaxb.version
Groovy	groovy.version
Gson	gson.version
H2	h2.version

Library	Version Property
Hamcrest	hamcrest.version
Hazelcast	hazelcast.version
Hazelcast Hibernate5	hazelcast-hibernate5.version
Hibernate	hibernate.version
Hibernate Validator	hibernate-validator.version
HikariCP	hikaricp.version
HSQldb	hsqldb.version
HtmlUnit	htmlunit.version
HttpAsyncClient	httpasyncclient.version
HttpClient	httpclient.version
HttpCore	httpcore.version
Infinispan	infinispan.version
InfluxDB Java	influxdb-java.version
Jackson Bom	jackson-bom.version
Jakarta Activation	jakarta-activation.version
Jakarta Annotation	jakarta-annotation.version
Jakarta JMS	jakarta-jms.version
Jakarta Json	jakarta-json.version
Jakarta Json Bind	jakarta-json-bind.version
Jakarta Mail	jakarta-mail.version
Jakarta Persistence	jakarta-persistence.version
Jakarta Servlet	jakarta-servlet.version
Jakarta Servlet JSP JSTL	jakarta-servlet-jsp-jstl.version
Jakarta Transaction	jakarta-transaction.version
Jakarta Validation	jakarta-validation.version
Jakarta WebSocket	jakarta-websocket.version

Library	Version Property
Jakarta WS RS	jakarta-ws-rs.version
Jakarta XML Bind	jakarta-xml-bind.version
Jakarta XML SOAP	jakarta-xml-soap.version
Jakarta XML WS	jakarta-xml-ws.version
Janino	janino.version
Javax Activation	javax-activation.version
Javax Annotation	javax-annotation.version
Javax Cache	javax-cache.version
Javax JAXB	javax-jaxb.version
Javax JAXWS	javax-jaxws.version
Javax JMS	javax-jms.version
Javax Json	javax-json.version
Javax JsonB	javax-jsonb.version
Javax Mail	javax-mail.version
Javax Money	javax-money.version
Javax Persistence	javax-persistence.version
Javax Transaction	javax-transaction.version
Javax Validation	javax-validation.version
Javax WebSocket	javax-websocket.version
Jaxen	jaxen.version
Jaybird	jaybird.version
JBoss Logging	jboss-logging.version
JBoss Transaction SPI	jboss-transaction-spi.version
JDOM2	jdom2.version
Jedis	jedis.version
Jersey	jersey.version

Library	Version Property
Jetty	jetty.version
Jetty EL	jetty-el.version
Jetty JSP	jetty-jsp.version
Jetty Reactive HTTPClient	jetty-reactive-httpclient.version
JMustache	jmustache.version
Johnzon	johnzon.version
Jolokia	jolokia.version
jOOQ	jooq.version
Json Path	json-path.version
Json-smart	json-smart.version
JsonAssert	jsonassert.version
JSTL	jstl.version
JTDS	jtds.version
JUnit	junit.version
JUnit Jupiter	junit-jupiter.version
Kafka	kafka.version
Kotlin	kotlin.version
Kotlin Coroutines	kotlin-coroutines.version
Lettuce	lettuce.version
Liquibase	liquibase.version
Log4j2	log4j2.version
Logback	logback.version
Lombok	lombok.version
MariaDB	mariadb.version
Maven AntRun Plugin	maven-antrun-plugin.version
Maven Assembly Plugin	maven-assembly-plugin.version

Library	Version Property
Maven Clean Plugin	maven-clean-plugin.version
Maven Compiler Plugin	maven-compiler-plugin.version
Maven Dependency Plugin	maven-dependency-plugin.version
Maven Deploy Plugin	maven-deploy-plugin.version
Maven Enforcer Plugin	maven-enforcer-plugin.version
Maven Failsafe Plugin	maven-failsafe-plugin.version
Maven Help Plugin	maven-help-plugin.version
Maven Install Plugin	maven-install-plugin.version
Maven Invoker Plugin	maven-invoker-plugin.version
Maven Jar Plugin	maven-jar-plugin.version
Maven Javadoc Plugin	maven-javadoc-plugin.version
Maven Resources Plugin	maven-resources-plugin.version
Maven Shade Plugin	maven-shade-plugin.version
Maven Source Plugin	maven-source-plugin.version
Maven Surefire Plugin	maven-surefire-plugin.version
Maven War Plugin	maven-war-plugin.version
Micrometer	micrometer.version
MIMEPull	mimepull.version
Mockito	mockito.version
MongoDB	mongodb.version
MSSQL JDBC	mssql-jdbc.version
MySQL	mysql.version
NekoHTML	nekohtml.version
Neo4j OGM	neo4j-ogm.version
Netty	netty.version
Netty tcNative	netty-tcnative.version

Library	Version Property
Nimbus JOSE JWT	nimbus-jose-jwt.version
NIO Multipart Parser	nio-multipart-parser.version
OAuth2 OIDC SDK	oauth2-oidc-sdk.version
ojdbc	ojdbc.version
OkHttp3	okhttp3.version
Oracle Database	oracle-database.version
Pooled JMS	pooled-jms.version
Postgresql	postgresql.version
Prometheus PushGateway	prometheus-pushgateway.version
Quartz	quartz.version
QueryDSL	querydsl.version
R2DBC Bom	r2dbc-bom.version
Rabbit AMQP Client	rabbit-amqp-client.version
Reactive Streams	reactive-streams.version
Reactor Bom	reactor-bom.version
REST Assured	rest-assured.version
RSocket	rsocket.version
RxJava	rxjava.version
RxJava Adapter	rxjava-adapter.version
RxJava2	rxjava2.version
SAAJ Impl	saaj-impl.version
Selenium	selenium.version
Selenium HtmlUnit	selenium-htmlunit.version
SendGrid	sendgrid.version
Servlet API	servlet-api.version
SLF4J	slf4j.version

Library	Version Property
SnakeYAML	snakeyaml.version
Solr	solr.version
Spring AMQP	spring-amqp.version
Spring Batch	spring-batch.version
Spring Data Releasetrain	spring-data-releasetrain.version
Spring Framework	spring-framework.version
Spring HATEOAS	spring-hateoas.version
Spring Integration	spring-integration.version
Spring Kafka	spring-kafka.version
Spring LDAP	spring-ldap.version
Spring RESTDocs	spring-restdocs.version
Spring Retry	spring-retry.version
Spring Security	spring-security.version
Spring Session Bom	spring-session-bom.version
Spring WS	spring-ws.version
SQLite JDBC	sqlite-jdbc.version
Sun Mail	sun-mail.version
Thymeleaf	thymeleaf.version
Thymeleaf Extras Data Attribute	thymeleaf-extras-data-attribute.version
Thymeleaf Extras Java8Time	thymeleaf-extras-java8time.version
Thymeleaf Extras SpringSecurity	thymeleaf-extras-springsecurity.version
Thymeleaf Layout Dialect	thymeleaf-layout-dialect.version
Tomcat	tomcat.version
UnboundID LDAPSDK	unboundid-ldapsdk.version

Library	Version Property
Undertow	undertow.version
Versions Maven Plugin	versions-maven-plugin.version
WebJars HAL Browser	webjars-hal-browser.version
WebJars Locator Core	webjars-locator-core.version
WSDL4j	wsdl4j.version
XML Maven Plugin	xml-maven-plugin.version
XmlUnit2	xmlunit2.version