Red Hat build of Apache Camel 4.4

红帽构建的 Apache Camel for Spring Boot 入门

Last Updated: 2024-07-03

Red Hat build of Apache Camel 4.4 红帽构建的 Apache Camel for Spring Boot 入门

法律通告

摘要

本指南介绍了红帽构建的 Apache Camel，并解释了使用 Red Hat build of Apache Camel 创建和部署应用程序的各种方法。

目录

前言

使开源包含更多

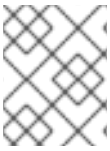
红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。详情请查看 CTO Chris Wright 的信息。

第 1 章 红帽构建的 APACHE CAMEL FOR SPRING BOOT 入门

本指南介绍了红帽构建的 Apache Camel for Spring Boot，并演示如何使用红帽构建的 Apache Camel for Spring Boot 构建应用程序。

1.1. RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT STARTERS

Camel 对 Spring Boot 的支持为许多 Camel 组件提供 Camel 和启动程序的 自动配置。在 Spring 上下文中提供的 Camel 上下文自动探测 Camel 路由的建议自动配置，并将密钥 Camel 工具（如制作者模板、消费者模板和类型转换器）注册为 beans。



注意

有关使用 Maven archtype 为 Spring Boot 应用程序生成 Camel 的详情，请参考使用 Maven 为 Spring Boot 应用程序生成 Camel。

要开始，您必须将 Camel Spring Boot BOM 添加到 Maven **pom.xml** 文件中。

```
<dependencyManagement>

  <dependencies>
    <!-- Camel BOM -->
    <dependency>
      <groupId>com.redhat.camel.springboot.platform</groupId>
      <artifactId>camel-spring-boot-bom</artifactId>
      <version>4.4.0.redhat-00014</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <!-- ... other BOMs or dependencies ... -->
  </dependencies>

</dependencyManagement>
```

camel-spring-boot-bom 是一个基本的 BOM，其中包含 Camel Spring Boot starter JAR 列表。

接下来，添加 Camel Spring Boot 初学者 以启动 Camel 上下文。

```
  <dependencies>
    <!-- Camel Starter -->
    <dependency>
      <groupId>org.apache.camel.springboot</groupId>
      <artifactId>camel-spring-boot-starter</artifactId>
    </dependency>
    <!-- ... other dependencies ... -->
  </dependencies>
```

您还必须添加 Spring Boot 应用程序 所需的组件 启动程序。以下示例演示了如何将自动配置入门添加到 MQTT5 组件。https://access.redhat.com/documentation/zh-cn/red_hat_build_of_apache_camel/4.0/html-single/red_hat_build_of_apache_camel_for_spring_boot_reference/index#spring_boot_auto_configuration

```
<dependencies>
  <!-- ... other dependencies ... -->
  <dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-paho-mqtt5</artifactId>
  </dependency>
</dependencies>
```

1.1.1. Spring Boot 配置支持

每个 入门 都列出了您可以在标准 **application.properties** 或 **application.yml** 文件中配置的配置参数。这些参数的格式为 **camel.component.[component-name].[parameter]**。例如，要配置 MQTT5 代理的 URL，您可以设置：

```
camel.component.paho-mqtt5.broker-url=tcp://localhost:61616
```

1.1.2. 添加 Camel 路由

Camel 路由 在 Spring 应用程序上下文中检测到，例如使用 **org.springframework.stereotype.Component** 注解的路由将被加载到 Camel 上下文并运行中。

```
import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MyRoute extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("...")
      .to("...");
  }

}
```

1.1.3. 使用域特定语言

Apache Camel 使用 Java 域特定语言或 DSL，在各种域特定语言(DSL)中创建企业集成模式或路由，如下所示：

- Java DSL：使用流畅的构建器风格的基于 Java 的 DSL。
- XML DSL：只有 Camel XML 文件中的基于 XML 的 DSL。
- YAML DSL 使用 YAML 格式创建路由。

1.1.3.1. DSL 的优点

与通用语言相比，使用 DSL 的优点如下：

- 更易于学习，更易于使用。您可以看到主逻辑的开始和结束位置。
- 更安全的代码。Apache Camel 中的 DSL 具有稳定的构建块，它将所有步骤绑定到一起。

- 错误是特定于域的。如果出现故障，错误描述更为明确和说明。更简单的代码也意味着不易出错的代码。
- DSL 设计为独立于平台。如果代码更改，其影响被委派给较低层。

1.1.3.2. 比较不同的 DSL

本节介绍 DSL 和您可以使用这些 DSL 的不同场景之间的区别。

	Java DSL	XML DSL	YAML DSL
开发人员工具	• 您可以将每个 IDE 与 Java 支持配合使用。 • 红帽在 VS Code 中提供 Apache Camel 的扩展包。这个软件包包含在 VS Code 中使用 Red Hat build of Apache Camel 的所有必要扩展。这包括对 Camel K Java 独立、支持 Camel URI 完成和诊断以及从源编辑器运行和调试 Camel 路由的语言支持。 • 语言支持和基本 Camel 文本路由调试。 • 它提供了代码协助，并提供路由调试器。	• 您可以将每个 IDE 与 XML 支持一起使用。 • 红帽在 VS Code 中提供 Apache Camel 的扩展包。这个软件包包含在 VS Code 中使用 Red Hat build of Apache Camel 的所有必要扩展。这包括对 Camel K Java 独立、支持 Camel URI 完成和诊断以及从源编辑器运行和调试 Camel 路由的语言支持。 • 语言支持和基本 Camel 文本路由调试 • 它提供了代码协助，并提供路由调试器。	• 您可以将每个 IDE 与 YAML 支持一起使用。 • 红帽在 VS Code 中提供 Apache Camel 的扩展包。这个软件包包含在 VS Code 中使用 Red Hat build of Apache Camel 的所有必要扩展。这包括对 Camel K Java 独立、支持 Camel URI 完成和诊断以及从源编辑器运行和调试 Camel 路由的语言支持。 • 语言支持和基本 Camel 文本路由调试。 • 它提供了代码协助，并提供路由调试器。 • 它还包括 Kaoto VS Code 扩展，它提供视觉集成设计器。
Hawtio / Fuse Console	Hawtio 从运行时检索路由作为 XML，并显示路由，而不考虑使用哪个 DSL 创建路由。	Hawtio 从运行时检索路由作为 XML，并显示路由，而不考虑使用哪个 DSL 创建路由。	Hawtio 从运行时检索路由作为 XML，并显示路由，而不考虑使用哪个 DSL 创建路由。

	Java DSL	XML DSL	YAML DSL
软件开发模型	DSL 采用流畅的构建器 API。	• 可以使用图形编辑器建模开发方法(Eclipse Desktop)。 • 允许基于拖放开发。 • 借助非常成熟的 IDE 支持，还可以进行基于文本的开发。	从头开始编写的难度。可以使用图形编辑器建模开发方法。
调试代码	• 有 IDE 插件通过 EIP 步骤 DSL 调试来提供步骤。您可以逐步进入 RouteBuilder，但仅在启动时调用，而不在处理过程中调用。 • 断点可以放入核心 Camel 类的 Java 代码中。 • 可以添加临时处理器并使用 Java 调试器。	• 有 IDE 插件通过 EIP 步骤 DSL 调试来提供步骤。 • 断点可以放入核心 Camel 类的 Java 代码中。	• 有 IDE 插件通过 EIP 步骤 DSL 调试来提供步骤。 • 断点可以放入核心 Camel 类的 Java 代码中。
与依赖项注入(DI)框架集成	更轻松地与任何 DI 框架集成。	虽然可以从 XML DSL 中的 DI 框架引用现有的 Bean，但在 XML 中声明新的 Bean 时，在 XML 中声明新的 Bean 使这些 Bean 专用于 Camel 本身，而不是 DI 框架的一部分（如 Quarkus 或 Spring Boot）。	虽然可以从 YAML DSL 中的 DI 框架引用现有的 Bean，在 YAML 中声明新 Bean 使这些 Bean 专用于 Camel 本身，而不是 DI 框架的一部分（如 Quarkus 或 Spring Boot）。
团队大小	更灵活，但更难以阅读代码。适用于长期工作和支持代码的小型共同团队。	• 对于大型和不同团队的好处。 • 更灵活，使创建复杂路由变得困难。	• 对于大型和不同团队的好处。 • 更灵活，使创建复杂路由变得困难。

	Java DSL	XML DSL	YAML DSL
团队结构	要求团队拥有开发 Camel 集成的 Java 开发人员。其他团队成员还需要了解 Java 才能读取集成流。	• XML 是广泛的语言，所有开发人员都可以在开发 Camel 时重复使用现有技能。 • 它提供更高级别的抽象，使与业务开发人员和支持团队轻松通信。	• YAML 是广泛的语言，所有开发人员都可以在开发 Camel 时重复使用现有技能。 • 它提供更高级别的抽象，使与业务开发人员和支持团队轻松通信。
开发人员体验和首选项	• 更适合 Java 经验丰富的开发人员比 XML 更加简洁，具有内部类和功能方面。 • Java 开发人员倾向于首选纯 Java 和注释，而不是 XML。	适合新用户，因为它为设计路由提供了图形化的方法。	适合新用户，因为它为设计路由提供了图形化的方法。

1.2. SPRING BOOT

Spring Boot 自动配置 Camel。在 Spring 上下文中提供了 Camel 上下文自动探测 Camel 路由的建议自动配置，并将密钥 Camel 实用程序（如制作者模板、消费者模板和类型转换器）注册为 beans。

Maven 用户需要将以下依赖项添加到其 **pom.xml** 中，才能使用此组件：

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-spring-boot</artifactId>
  <version>4.4.0.redhat-00014</version> <!-- use the same version as your Camel core version -->
</dependency>
```

camel-spring-boot jar 附带 **spring.factories** 文件，因此当您将依赖项添加到类路径后，Spring Boot 将自动为您自动配置 Camel。

1.2.1. Camel Spring Boot Starter

Apache Camel 附带了一个 Spring Boot Starter 模块，允许您使用启动程序开发 Spring Boot 应用程序。源代码中也有一个 示例应用程序。

要使用启动程序，将以下内容添加到 spring boot pom.xml 文件中：

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
```

```
    <artifactId>camel-spring-boot-bom</artifactId>
    <version>4.4.0.redhat-00014</version> <!-- use the same version as your Camel core version -->
</dependency>
```

然后，您只能使用 Camel 路由添加类，例如：

```
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo").to("log:bar");
    }
}
```

然后，这些路由会自动启动。

您可以在 **application.properties** 或 **application.yml** 文件中自定义 Camel 应用程序。

1.2.2. Spring Boot 自动配置

当在 Spring Boot 中使用 spring-boot 时，请确保使用以下 Maven 依赖项来支持自动配置：

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-spring-boot-starter</artifactId>
  <version>4.4.0.redhat-00014</version> <!-- use the same version as your Camel core version -->
</dependency>
```

1.2.3. 自动配置的 Camel 上下文

Camel 自动配置提供的最重要功能是 **CamelContext** 实例。Camel 自动配置会为您创建一个 **SpringCamelContext**，并负责正确初始化和关闭该上下文。创建的 Camel 上下文也会在 Spring 应用程序上下文中注册（在 **camelContext** bean 名称下），以便您可以像任何其他 Spring bean 一样访问它。

```
@Configuration
public class MyAppConfig {

  @Autowired
  CamelContext camelContext;

  @Bean
  MyService myService() {
    return new DefaultMyService(camelContext);
  }

}
```

1.2.4. 自动探测 Camel 路由

Camel 自动配置从 Spring 上下文收集所有 **RouteBuilder** 实例，并将它们自动注入到提供的 **CamelContext** 中。这意味着，使用 Spring Boot starter 创建新 Camel 路由非常简单，就像将 **@Component** 注解的类添加到您的类路径中一样简单：

```
@Component
public class MyRouter extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("jms:invoices").to("file:/invoices");
  }

}
```

或者在您的 **@Configuration** 类中创建新的路由 **RouteBuilder** bean：

```
@Configuration
public class MyRouterConfiguration {

  @Bean
  RoutesBuilder myRouter() {
    return new RouteBuilder() {

      @Override
      public void configure() throws Exception {
        from("jms:invoices").to("file:/invoices");
      }

    };
  }

}
```

1.2.5. Camel 属性

Spring Boot 自动配置会自动连接到 Spring Boot 外部配置 （可能包含属性占位符、OS 环境变量或系统属性）支持 Camel 属性。它基本上意味着 **application.properties** 文件中定义的任何属性：

```
route.from = jms:invoices
```

或者通过系统属性设置：

```
java -Droute.to=jms:processed.invoices -jar mySpringApp.jar
```

可用作 Camel 路由中的占位符：

```
@Component
public class MyRouter extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("{{route.from}}").to("{{route.to}}");
```

```
  }

}
```

1.2.6. 自定义 Camel 上下文配置

如果要在 Camel auto-configuration 创建的 **CamelContext** bean 上执行一些操作，请在 Spring 上下文中注册 **CamelContextConfiguration** 实例：

```
@Configuration
public class MyAppConfig {

  @Bean
  CamelContextConfiguration contextConfiguration() {
    return new CamelContextConfiguration() {
      @Override
      void beforeApplicationStart(CamelContext context) {
        // your custom configuration goes here
      }
    };
  }

}
```

只有在 Spring 上下文启动前，才会调用方法 **beforeApplicationStart**，因此传递给此回调的 **CamelContext** 实例被完全自动配置。如果您将多个 **CamelContextConfiguration** 实例添加到 Spring 上下文中，则执行每个实例。

1.2.7. 自动配置的消费者和制作者模板

Camel 自动配置提供预配置的 **ConsumerTemplate** 和 **ProducerTemplate** 实例。只需将它们注入到 Spring 管理的 Bean 中：

```
@Component
public class InvoiceProcessor {

  @Autowired
  private ProducerTemplate producerTemplate;

  @Autowired
  private ConsumerTemplate consumerTemplate;

  public void processNextInvoice() {
    Invoice invoice = consumerTemplate.receiveBody("jms:invoices", Invoice.class);
    ...
    producerTemplate.sendBody("netty-http:http://invoicing.com/received/" + invoice.id());
  }

}
```

默认情况下，消费者模板和制作者模板随端点缓存大小设置为 1000。您可以通过修改以下 Spring 属性来更改这些值：

```
camel.springboot.consumer-template-cache-size = 100
camel.springboot.producer-template-cache-size = 200
```

1.2.8. 自动配置的 TypeConverter

Camel 自动配置在 Spring 上下文中注册一个名为 **typeConverter** 的 **TypeConverter** 实例。

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

  public long parseInvoiceValue(Invoice invoice) {
    String invoiceValue = invoice.grossValue();
    return typeConverter.convertTo(Long.class, invoiceValue);
  }

}
```

1.2.8.1. Spring 类型转换 API 网桥

Spring 附带强大的 类型转换 API。Spring API 与 Camel 类型转换器 API 类似。因为两个 API 相似，Camel Spring Boot 会自动注册桥接转换器(**SpringTypeConverter**)，以委派给 Spring 转换 API。这意味着开箱即用的 Camel 将对待 Spring Converters，如 Camel。使用这个方法，您可以使用 Camel **TypeConverter** API 访问 Camel 和 Spring 转换器：

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

  public UUID parseInvoiceId(Invoice invoice) {
    // Using Spring's StringToUUIDConverter
    UUID id = invoice.typeConverter.convertTo(UUID.class, invoice.getId());
  }

}
```

在 hood Camel Spring Boot 下，将转换委托给应用程序上下文中可用的 Spring 的 **ConversionService** 实例。如果没有 **ConversionService** 实例可用，则 Camel Spring Boot 自动配置将为您创建一个。

1.2.9. 保持应用程序处于活动状态

启用了此功能的 Camel 应用程序在启动时启动一个新的线程，以便仅通过防止 JVM 终止来保持应用程序处于活动状态。这意味着，在使用 Spring Boot 启动 Camel 应用程序后，应用程序会等待 **Ctrl+C** 信号，且不会立即退出。

可以使用 **camel.springboot.main-run-controller** 到 **true** 激活控制器线程。

```
camel.springboot.main-run-controller = true
```

使用 Web 模块的应用程序（例如，导入 **org.springframework.boot:spring-boot-web-starter** 模块的应用程序通常不需要使用此功能），因为应用程序存在其他非守护进程线程。

1.2.10. 添加 XML 路由

默认情况下，您可以将 Camel XML 路由放在 camel 目录下的 classpath 中，camel-spring-boot 将自动探测到并包含。您可以使用配置选项配置目录名称或关闭这个名称：

```
# turn off
camel.springboot.routes-include-pattern = false
```

```
# scan only in the com/foo/routes classpath
camel.springboot.routes-include-pattern = classpath:com/foo/routes/*.xml
```

XML 文件应该是 Camel XML 路由(**而不是 < CamelContext>**)，例如：

```
<routes xmlns="http://camel.apache.org/schema/spring">
  <route id="test">
    <from uri="timer://trigger"/>
    <transform>
      <simple>ref:myBean</simple>
    </transform>
    <to uri="log:out"/>
  </route>
</routes>
```

1.2.11. 测试 JUnit 5 方法

为了进行测试，Maven 用户需要将以下依赖项添加到其 **pom.xml** 中：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <version>3.2.5</version> <!-- Use the same version as your Spring Boot version -->
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-spring-junit5</artifactId>
  <version>4.4.0.redhat-00019</version> <!-- use the same version as your Camel core version -->
  <scope>test</scope>
</dependency>
```

要测试 Camel Spring Boot 应用程序，请使用 **@CamelSpringBootTest** 注解您的测试类。这会为应用程序提供 Camel 的 Spring Test 支持，以便您可以使用 Spring Boot 测试惯例编写测试。

要获取 **CamelContext** 或 **ProducerTemplate**，您可以使用 **@Autowired** 以普通的 Spring 方式将它们注入到类中。

您还可以使用 camel-test-spring-junit5 来声明性配置测试。本例使用 **@MockEndpoints** 注释来自动模拟端点：

```
@CamelSpringBootTest
```

```
@SpringBootApplication
@MockEndpoints("direct:end")
public class MyApplicationTest {

    @Autowired
    private ProducerTemplate template;

    @EndpointInject("mock:direct:end")
    private MockEndpoint mock;

    @Test
    public void testReceive() throws Exception {
        mock.expectedBodiesReceived("Hello");
        template.sendBody("direct:start", "Hello");
        mock.assertIsSatisfied();
    }

}
```

1.3. 组件启动者

Camel Spring Boot 支持以下 Camel 工件作为 Spring Boot Starters：

- 表 1.1 “Camel 组件”
- 表 1.2 “Camel 数据格式”
- 表 1.3 “Camel 语言”
- 表 1.4 “其它扩展”

表 1.1. Camel 组件

组件	工件	描述	支持 IBM Power 和 IBM Z
AMQP	camel-amqp-starter	使用 Apache QPid 客户端与 AMQP 协议进行消息传递.	是
AWS Cloudwatch	camel-aws2-cw-starter	使用 AWS SDK 版本 2.x 将指标发送到 AWS CloudWatch。	是
AWS DynamoDB	camel-aws2-ddb-starter	使用 AWS SDK 版本 2.x 从 AWS DynamoDB 服务存储和检索数据。	是
AWS Kinesis	camel-aws2-kinesis-starter	使用 AWS SDK 版本 2.x 使用 AWS SDK 版本 2.x 使用和生成 AWS Kinesis Streams 的记录。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
AWS Lambda	camel-aws2-lambda-starter	使用 AWS SDK 版本 2.x 管理并调用 AWS Lambda 功能。	是
AWS S3 Storage Service	camel-aws2-s3-starter	使用 AWS SDK 版本 2.x 从 AWS S3 Storage Service 存储和检索对象。	是
AWS Simple Notification System (SNS)	camel-aws2-sns-starter	使用 AWS SDK 版本 2.x 将信息发送到 AWS Simple Notification Topic。	是
AWS Simple Queue Service (SQS)	camel-aws2-sqs-starter	使用 AWS SDK 版本 2.x 向 AWS SQS 服务发送和接收信息。	是
Azure ServiceBus	camel-azure-servicebus-starter	向 Azure 事件总线发送和接收信息。	是
Azure Storage Blob Service	camel-azure-storage-blob-starter	使用 SDK v12 从 Azure Storage Blob Service 存储和检索 Blob。	是
Azure Storage Queue Service	camel-azure-storage-queue-starter	azure-storage-queue 组件用于使用 Azure SDK v12 存储和检索信息到 Azure Storage Queue。	是
Bean	camel-bean-starter	调用存储在 Camel registry 中的 Java Bean 的方法。	是
Bean Validator	camel-bean-validator-starter	使用 Java Bean Validation API 验证消息正文。	是
浏览	camel-browse-starter	检查在支持 BrowsableEndpoint 的端点上收到的消息。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
Cassandra CQL	camel-cassandraql-starter	使用 CQL3 API （而不是 Thrift API）与 Cassandra 2.0 集成。基于 DataStax 提供的 Cassandra Java 驱动程序。	是
CICS	camel-cics-starter	与 CICS® 通用事务处理子系统交互。	否
控制总线	camel-controlbus-starter	管理和监控 Camel 路由。	是
cron	camel-cron-starter	通过 Unix cron 语法指定的时间触发事件的通用接口。	是
加密(JCE)	camel-crypto-starter	使用 Java Cryptographic 扩展(JCE)的签名服务签名并验证交换。	是
CXF	camel-cxf-soap-starter	使用 Apache CXF 公开 SOAP WebServices 或使用 CXF WS 客户端连接到外部 WebServices。	是
CXF-RS	camel-cxf-rest-starter	使用 Apache CXF 公开 JAX-RS REST 服务，或使用 CXF REST 客户端连接到外部 REST 服务。	是
数据格式	camel-dataformat-starter	使用 Camel 数据格式作为常规 Camel 组件。	是
Dataset	camel-dataset-starter	提供用于 Camel 应用程序的负载和 soak 测试的数据。	是
direct	camel-direct-starter	同步调用来自同一 Camel 上下文的另一个端点。	是
Elastic Search	camel-elasticsearch-starter	通过 Java 客户端 API 将请求发送到 ElasticSearch。	否

组件	工件	描述	支持 IBM Power 和 IBM Z
FHIR	camel-fhir-starter	使用 FHIR (Fast Healthcare Interoperability Resources)标准交换医疗域中的信息。	否
File	camel-file-starter	读写文件。	是
Flink	camel-flink-starter	将 DataSet 作业发送到 Apache Flink 集群。	是
FTP	camel-ftp-starter	上传文件并将其下载到 FTP 服务器/从 FTP 服务器下载.	是
Google BigQuery	camel-google-bigquery-starter	用于分析的 Google BigQuery 数据仓库。	是
Google Pubsub	camel-google-pubsub-starter	向 Google Cloud Platform PubSub Service 发送和接收信息。	是
gRPC	camel-grpc-starter	公开 gRPC 端点并访问外部 gRPC 端点。	是
HTTP	camel-http-starter	使用 Apache HTTP 客户端 4.x 将请求发送到外部 HTTP 服务器。	是
Infinispan	camel-infinispan-starter	从/到 Infinispan 分布式键/值存储和数据网格进行读写。	否
Infinispan Embedded	camel-infinispan-embedded-starter	从/到 Infinispan 分布式键/值存储和数据网格进行读写。	是
JDBC	camel-jdbc-starter	通过 SQL 和 JDBC 访问数据库.	是
Jira	camel-jira-starter	与 JIRA 问题跟踪器交互。	是
JMS	camel-jms-starter	从 JMS Queue 或 Topic 中发送和接收消息。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
JPA	camel-jpa-starter	使用 Java Persistence API (zFCP)从数据库存储和检索 Java 对象。	是
JSLT	camel-jslt-starter	使用 JSLT 查询或转换 JSON 有效负载。	是
Kafka	camel-kafka-starter	向 Apache Kafka 代理发送和接收信息。	是
kamelet	camel-kamelet-starter	调用 Kamelets	是
Kubernetes ConfigMap	camel-kubernetes-starter	对 Kubernetes ConfigMap 执行操作，并获取有关 ConfigMap 更改的通知。	是
Kubernetes 自定义资源	camel-kubernetes-starter	对 Kubernetes 自定义资源执行操作，并获取有关部署更改的通知。	是
Kubernetes Deployments	camel-kubernetes-starter	对 Kubernetes Deployments 执行操作，并获取有关部署更改的通知。	是
Kubernetes 事件	camel-kubernetes-starter	对 Kubernetes 事件执行操作，并获取有关事件更改的通知。	是
Kubernetes HPA	camel-kubernetes-starter	对 Kubernetes Horizontal Pod Autoscaler (HPA)执行操作，并获取有关 HPA 更改的通知。	是
Kubernetes 任务	camel-kubernetes-starter	对 Kubernetes 作业执行操作。	是
Kubernetes 命名空间	camel-kubernetes-starter	对 Kubernetes 命名空间执行操作，并获得对命名空间更改的通知。	是
Kubernetes 节点	camel-kubernetes-starter	在 Kubernetes 节点上执行操作，并获得有关节点更改的通知。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
Kubernetes 持久性卷	camel-kubernetes-starter	对 Kubernetes 持久性卷执行操作，并获得有关持久性卷更改的通知。	是
Kubernetes 持久性卷声明	camel-kubernetes-starter	对 Kubernetes 持久性卷声明执行操作，并获得有关持久性卷声明更改的通知。	是
Kubernetes Pod	camel-kubernetes-starter	对 Kubernetes Pod 执行操作并获取有关 Pod 更改的通知。	是
Kubernetes 复制控制器	camel-kubernetes-starter	是对 Kubernetes Replication Controller 的执行操作，并获得关于复制控制器更改的通知。	是
Kubernetes 资源配额	camel-kubernetes-starter	对 Kubernetes 资源配额执行操作。	是
Kubernetes Secret	camel-kubernetes-starter	对 Kubernetes Secret 执行操作。	是
Kubernetes 服务帐户	camel-kubernetes-starter	对 Kubernetes 服务帐户执行操作。	是
Kubernetes 服务	camel-kubernetes-starter	对 Kubernetes 服务执行操作并获取有关服务更改的通知。	是
Kudu	camel-kudu-starter	与 Apache Kudu 互动，Apache Hadoop 生态系统的免费、开源列导向型数据存储。	否
语言	camel-language-starter	使用 Camel 支持的任何语言执行脚本。	是
LDAP	camel-ldap-starter	在 LDAP 服务器上执行搜索。	是
Log	camel-log-starter	日志消息到底层日志记录机制。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
LRA	camel-lra-starter	Camel saga 绑定 for Long-Running-Action 框架。	是
Mail	camel-mail-starter	使用 imap、pop3 和 smtp 协议发送和接收电子邮件。	是
邮件 Microsoft OAuth	camel-mail-microsoft-oauth-starter	Camel Mail OAuth2 Authenticator for Microsoft Exchange Online.	是
MapStruct	camel-mapstruct-starter	使用 Mapstruct 键入 Conversion。	是
Master	camel-master-starter	集群中仅消耗来自给定端点的单一使用者；如果 JVM 中断，则进行自动故障转移。	是
Micrometer	camel-micrometer-starter	使用 Micrometer 库直接从 Camel 路由收集各种指标。	是
Minio	camel-minio-starter	使用 Minio SDK 从 Minio Storage Service 存储和检索对象。	是
MLLP	camel-mllp-starter	使用 MLLP 协议与外部系统通信。	是
Mock	camel-mock-starter	使用模拟测试路由和调解规则。	是
MongoDB	camel-mongodb-starter	对 MongoDB 文档和集合执行操作。	是
MyBatis	camel-mybatis-starter	使用 MyBatis 在相关数据库中执行查询、轮询、插入、更新或删除。	是
Netty	camel-netty-starter	使用带有 Netty 4.x 的 TCP 或 UDP 的套接字级别网络。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
Olingo4	camel-olingo4-starter	使用 Apache Olingo OData API 与 OData 4.0 服务通信。	是
OpenShift 构建配置	camel-kubernetes-starter	对 OpenShift 构建配置执行操作。	是
OpenShift 构建	camel-kubernetes-starter	对 OpenShift 构建执行操作。	是
OpenShift 部署配置	camel-kubernetes-starter	对 Openshift Deployment Configs 执行操作，并获得关于部署配置更改的通知。	是
Netty HTTP	camel-netty-http-starter	使用 Netty 4.x 的 Netty HTTP 服务器和客户端。	是
paho	camel-paho-starter	使用 Eclipse Paho MQTT 客户端与 MQTT 消息代理进行通信。	是
Paho MQTT 5	camel-paho-mqtt5-starter	使用 Eclipse Paho MQTT v5 客户端与 MQTT 消息代理进行通信。	是
平台 HTTP	camel-platform-http-starter	使用当前平台中的 HTTP 服务器公开 HTTP 端点。	是
quartz	camel-quartz-starter	调度使用 Quartz 2.x 调度程序发送消息。	是
Ref	camel-ref-starter	将消息路由到端点会根据 Camel Registry 中的名称动态查找。	是
REST	camel-rest-starter	公开 REST 服务或调用外部 REST 服务。	是
saga	camel-saga-starter	使用 Saga EIP 在路由中执行自定义操作。	是
Salesforce	camel-salesforce-starter	使用 Java DTO 与 Salesforce.	是

组件	工件	描述	支持 IBM Power 和 IBM Z
SAP	camel-sap-starter	使用 SAP Java Connector (SAP JCo)库来促进与 SAP 和 SAP IDoc 库的双向通信，以促进 Intermediate Document (IDoc)格式的文档传输。	是
scheduler	camel-scheduler-starter	使用 java.util.concurrent.ScheduledExecutorService 以指定间隔生成消息。	是
SEDA	camel-seda-starter	异步调用同一 JVM 中任何 Camel 上下文的另一个端点。	是
Servlet	camel-servlet-starter	由 Servlet 提供 HTTP 请求。	是
Slack	camel-slack-starter	向 Slack 发送和接收信息。	是
SMB	camel-smb-starter	从 SMB （服务器消息块）共享接收文件。	是
SNMP	camel-snmp-starter	接收陷阱和轮询 SNMP（简单网络管理协议）功能的设备。	是
Splunk	camel-splunk-starter	在 Splunk 中发布或搜索事件。	否
Spring Batch	camel-spring-batch-starter	将消息发送到 Spring Batch 以进一步处理。	是
Spring JDBC	camel-spring-jdbc-starter	使用 Spring Transaction 支持通过 SQL 和 JDBC 访问数据库。	是
Spring LDAP	camel-spring-ldap-starter	将过滤器用作消息有效负载，在 LDAP 服务器中执行搜索。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
Spring RabbitMQ	camel-spring-rabbitmq-starter	使用 Spring RabbitMQ 客户端从 RabbitMQ 发送和接收消息。	是
Spring Redis	camel-spring-redis-starter	从 Redis 发送和接收信息。	是
Spring Webservice	camel-spring-ws-starter	您可以使用此组件与 Spring Web Services 集成。它提供访问 Web 服务和服务器端支持，以便创建您的合同优先 Web 服务。	是
SQL	camel-sql-starter	使用 Spring JDBC 执行 SQL 查询。	是
SQL 存储的步骤	camel-sql-starter	使用 Spring JDBC 执行 SQL 查询作为 JDBC 存储的流程。	是
SSH	camel-ssh-starter	使用 SSH 在远程主机上执行命令。	是
Stub	camel-stub-starter	在开发或测试过程中处理任何物理端点。	是
telegram	camel-telegram-starter	发送和接收充当 Telegram Botram Bot API 的消息。	是
计时器	camel-timer-starter	使用 java.util.Timer 以指定间隔生成消息。	是
validator	camel-validator-starter	使用 XML 架构和 JAXP 验证验证载荷。	是
Velocity	camel-velocity-starter	使用 Velocity 模板转换消息。	是
Vert.x HTTP Client	camel-vertx-http-starter	使用 Vert.x 将请求发送到外部 HTTP 服务器。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
Vert.x WebSocket	camel-vertx-websocket-starter	公开 WebSocket 端点，并使用 Vert.x 连接到远程 WebSocket 服务器。	是
Webhook	camel-webhook-starter	公开 Webhook 端点以接收其他 Camel 组件的推送通知。	是
XJ	camel-xj-starter	使用 XSLT 转换 JSON 和 XML 消息。	是
XSLT	camel-xslt-starter	使用 XSLT 模板转换 XML 有效负载。	是
XSLT Saxon	camel-xslt-saxon-starter	使用 Saxon 的 XSLT 模板转换 XML 有效负载。	是

表 1.2. Camel 数据格式

组件	工件	描述	支持 IBM Power 和 IBM Z
Avro	camel-avro-starter	使用 Apache Avro 二进制数据格式序列化和反序列化消息。	是
avro Jackson	camel-jackson-avro-starter	marshal POJO 到 Avro，并使用 Jackson 返回。	是
bindy	camel-bindy-starter	使用 Camel Bindy 的 POJO 和键值对(KVP)格式之间的 marshal 和 unmarshal。	是
HL7	camel-hl7-starter	使用 HL7 MLLP codec 的 marshal 和 unmarshal HL7 (Health care)模型对象。	是
JacksonXML	camel-jacksonxml-starter	unmarshal 是一个 XML 有效负载到 POJO，并使用 Jackson 的 XMLMapper 扩展来回放。	是

组件	工件	描述	支持 IBM Power 和 IBM Z
JAXB	camel-jaxb-starter	unmarshal XML 有效负载到 POJO，使用 JAXB2 XML marshalling 标准。	是
JSON Gson	camel-gson-starter	使用 Gson marshal POJO 到 JSON 并返回	是
JSON Jackson	camel-jackson-starter	marshal POJO 到 JSON 并使用 Jackson 返回	是
protobuf Jackson	camel-jackson-protobuf-starter	使用 Jackson 对 Protobuf 和 back 进行 marshal POJO。	是
SOAP	camel-soap-starter	marshal Java 对象到 SOAP 消息和回放。	是
zip 文件	camel-zipfile-starter	使用 java.util.zip.ZipStream 压缩和解压缩流。	是

表 1.3. Camel 语言

语言	工件	描述	支持 IBM Power 和 IBM Z
常数	camel-core-starter	固定值只在路由启动期间设置一次。	是
CSimple	camel-core-starter	评估编译的简单表达式。	是
ExchangeProperty	camel-core-starter	从 Exchange 获取属性。	是
File	camel-core-starter	简单语言的文件相关功能。	是
标头	camel-core-starter	从 Exchange 获取标头。	是
JQ	camel-jq-starter	针对 JSON 消息正文评估 JQ 表达式。	是
jsonpath	camel-jsonpath-starter	针对 JSON 消息正文评估 JSONPath 表达式。	是

语言	工件	描述	支持 IBM Power 和 IBM Z
Ref	camel-core-starter	使用 registry 中的现有表达式。	是
Simple（简单）	camel-core-starter	评估 Camel 简单表达式。	是
tokenize	camel-core-starter	使用分隔符模式对文本有效负载进行令牌化。	是
XML 令牌化	camel-xml-jaxp-starter	对 XML 有效负载进行令牌化。	是
XPath	camel-xpath-starter	针对 XML 有效负载评估 XPath 表达式。	是
XQuery	camel-saxon-starter	使用 XQuery 和 Saxon 查询和/或转换 XML 有效负载。	是

表 1.4. 其它扩展

扩展	工件	描述	支持 IBM Power 和 IBM Z
Jasypt	camel-jasypt-starter	使用 Jasypt 的安全性	是
kamelet Main	camel-kamelet-main-starter	运行 Kamelet standalone	是
OpenAPI Java	camel-openapi-java-starter	使用 openapi doc 的 rest-dsl 支持	是
OpenTelemetry	camel-opentelemetry-starter	使用 OpenTelemetry 的分布式追踪	是
Spring Security	camel-spring-security-starter	使用 Spring Security 的安全性	是
YAML DSL	camel-yaml-dsl-starter	使用 YAML 的 Camel DSL	是

1.4. 入门配置

清除和可访问的配置是任何应用程序的关键部分。Camel 启动器完全支持 Spring Boot 的外部配置机制。您还可以通过 Spring Beans 为更复杂的用例配置它们。

1.4.1. 使用外部配置

在内部，每个 初学者 都通过 Spring Boot 的配置 属性进行配置。每个配置参数 可以以不同的方式 设置（**application.[properties|json|yaml]** 文件、命令行参数、环境变量等。参数的格式是 **camel.[component|language|dataformat].[name].[parameter]**

例如，要配置 MQTT5 代理的 URL，您可以设置：

```
camel.component.paho-mqtt5.broker-url=tcp://localhost:61616
```

或者要配置 CSV 数据格式的 **delimeter** 为分号 (;)，您可以设置：

```
camel.dataformat.csv.delimiter=;
```

当将属性设置为所需类型时，Camel 将使用 Type Converter 机制。

您可以使用 **#bean:name** 来引用 Registry 中的 Bean：

```
camel.component.jms.transactionManager=#bean:myjtaTransactionManager
```

Bean 通常会使用 Java 创建：

```
@Bean("myjtaTransactionManager")
public JmsTransactionManager myjtaTransactionManager(PooledConnectionFactory pool) {
    JmsTransactionManager manager = new JmsTransactionManager(pool);
    manager.setDefaultTimeout(45);
    return manager;
}
```

Bean 也可以 在配置文件中 创建，但不建议在复杂用例中使用它。

1.4.2. 使用 Beans

也可以通过 Spring Beans 创建和配置 starters。在创建初学者之前，Camel 将首先在 Registry 中查询它（如果已存在）。例如，配置 Kafka 组件：

```
@Bean("kafka")
public KafkaComponent kafka(KafkaConfiguration kafkaconfiguration){
    return ComponentsBuilderFactory.kafka()
                    .brokers("{{kafka.host}}:{{kafka.port}}")
                    .build();
}
```

Bean 名称必须与您要配置的组件、Dataformat 或 Language 相等。如果没有在注解中指定 **Bean** 名称，它将被设置为方法名称。

典型的 Camel Spring Boot 项目将使用外部配置和 Bean 的组合来配置应用程序。有关如何配置 Camel Spring Boot 项目的更多示例，请参阅示例 存储库。

1.5. 使用 MAVEN 为 SPRING BOOT 应用程序生成 CAMEL

您可以使用 Maven archetype **org.apache.camel.archetypes:camel-archetype-spring-boot:4.4.0.redhat-00014** 生成红帽构建的 Apache Camel for Spring Boot 应用程序。

步骤

1. 运行以下命令：

```
mvn archetype:generate \
 -DarchetypeGroupId=org.apache.camel.archetypes \
 -DarchetypeArtifactId=camel-archetype-spring-boot \
 -DarchetypeVersion=4.4.0.redhat-00014 \
 -DgroupId=com.redhat \
 -DartifactId=csb-app \
 -Dversion=1.0-SNAPSHOT \
 -DinteractiveMode=false
```

2. 构建应用程序：

```
mvn package -f csb-app/pom.xml
```

3. 运行应用程序：

```
java -jar csb-app/target/csb-app-1.0-SNAPSHOT.jar
```

4. 通过检查由应用生成的 *Hello World* 输出的控制台日志来验证应用是否正在运行。

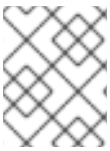
```
com.redhat.MySpringBootApplication      : Started MySpringBootApplication in 3.514
seconds (JVM running for 4.006)
Hello World
Hello World
```

1.6. 将 CAMEL SPRING BOOT 应用程序部署到 OPENSHIFT

本指南演示了如何将 Camel Spring Boot 应用程序部署到 OpenShift。

先决条件

- 您可以访问 OpenShift 集群。
- 已安装 OpenShift **oc** CLI 客户端，或者您可以访问 OpenShift Container Platform Web 控制台。



注意

经认证的 OpenShift Container Platform 在 Camel for Spring Boot 支持的配置 中列出。以下示例中使用 Red Hat OpenJDK 11 (ubi8/openjdk-11)容器镜像。

流程

1. 按照本指南的 Maven 生成 Camel for Spring Boot 应用程序一节中的内容，使用 Maven 为 Spring Boot 应用程序生成 Camel for Spring Boot 应用程序。
2. 在修改后的 pom.xml 目录下，执行以下命令。

```
mvn clean -DskipTests oc:deploy -Popenshift
```

3. 验证 CSB 应用是否在 pod 上运行。

```
oc logs -f dc/csb-app
```

1.7. 将补丁应用到红帽构建的 APACHE CAMEL FOR SPRING BOOT

使用新的 **patch-maven-plugin** 机制，您可以将补丁应用到红帽构建的 Apache Camel for Spring Boot 应用程序。此机制允许您更改由不同红帽应用程序 BOMS 提供的单个版本，例如 **camel-spring-boot-bom**。

patch-maven-plugin 的目的是将 Spring Boot BOM 上的 Camel 中列出的依赖项版本更新为您要应用到应用程序的补丁元数据中指定的版本。

patch-maven-plugin 执行以下操作：

- 检索与当前红帽应用程序 BOM 相关的补丁元数据。
- 将版本更改应用到从 BOMs 导入的 <dependencyManagement>。

在 **patch-maven-plugin** 获取元数据后，它会迭代声明插件的项目的所有受管和直接依赖项，并使用 CVE/patch 元数据替换依赖项版本（如果匹配）。在替换了版本后，Maven 构建将继续并根据标准 Maven 项目阶段进行进度。

流程

以下流程解释了如何将补丁应用到您的应用程序。

1. 将 **patch-maven-plugin** 添加到项目的 **pom.xml** 文件中。**patch-maven-plugin** 的版本必须与 Spring Boot BOM 上的 Camel 版本相同。

```
<build>
  <plugins>
    <<plugin>
      <groupId>com.redhat.camel.springboot.platform</groupId>
      <artifactId>patch-maven-plugin</artifactId>
      <version>${camel-spring-boot-version}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

2. 当您运行任何 **mvn clean deploy**,**mvn validate**, 或 **mvn dependency:tree** 命令时，插件会搜索项目模块来检查模块是否使用 Red Hat build of Apache Camel for Spring Boot BOM。只有以下内容是支持的 BOM：

   - **com.redhat.camel.springboot.platform:camel-spring-boot-bom**: 用于红帽构建的 Apache Camel for Spring Boot BOM

3. 如果插件找不到上述 BOM，插件会显示以下信息：

```
$ mvn clean install

[INFO] Scanning for projects...
[INFO]

========== Red Hat Maven patching ==========
```

```
[INFO] [PATCH] No project in the reactor uses Camel on Spring Boot product BOM. Skipping patch processing.
[INFO] [PATCH] Done in 7ms

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

4. 如果使用了正确的 BOM，则会找到补丁元数据，但没有补丁。

```
$ mvn clean install

[INFO] Scanning for projects...
[INFO]

========== Red Hat Maven patching ==========

[INFO] [PATCH] Reading patch metadata and artifacts from 2 project repositories
[INFO] [PATCH]  - redhat-ga-repository: http://maven.repository.redhat.com/ga/
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
Downloading from redhat-ga-repository: http://maven.repository.redhat.com/ga/com/redhat/camel/springboot/platform/redhat-camel-spring-boot-patch-metadata/maven-metadata.xml
Downloading from central: https://repo.maven.apache.org/maven2/com/redhat/camel/springboot/platform/redhat-camel-spring-boot-patch-metadata/maven-metadata.xml
[INFO] [PATCH] Resolved patch descriptor: /path/to/.m2/repository/com/redhat/camel/springboot/platform/redhat-camel-spring-boot-patch-metadata/3.20.1.redhat-00043/redhat-camel-spring-boot-patch-metadata-3.20.1.redhat-00043.xml
[INFO] [PATCH] Patch metadata found for com.redhat.camel.springboot.platform/camel-spring-boot-bom/[3.20,3.21)
[INFO] [PATCH] Done in 938ms

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

5. **patch-maven-plugin** 尝试获取此 Maven 元数据。

   - 对于使用 Camel Spring Boot BOM 的项目，**com.redhat.camel.springboot.platform:redhat-camel-spring-boot-patch-metadata/maven-metadata.xml** 已解决。这个 XML 数据是带有 **com.redhat.camel.springboot.platform:redhat-camel-spring-boot-patch-metadata:RELEASE** 的工件的元数据。

     Maven 生成的元数据示例

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <groupId>com.redhat.camel.springboot.platform</groupId>
  <artifactId>redhat-camel-spring-boot-patch-metadata</artifactId>
  <versioning>
    <release>3.20.1.redhat-00041</release>
    <versions>
      <version>3.20.1.redhat-00041</version>
    </versions>
```

```
    <lastUpdated>20230322103858</lastUpdated>
  </versioning>
 </metadata>
```

6. **patch-maven-plugin** 解析元数据，以选择应用到当前项目的版本。此操作只能用于带有特定版本的 Spring Boot BOM 上的 Camel Maven 项目。只有与版本范围或更新的版本匹配的元数据才适用，它只获取元数据的最新版本。

7. **patch-maven-plugin** 收集远程 Maven 存储库列表，用于下载由 **groupId**、**artifactId**、**和版本**在前面的步骤中标识的补丁元数据。这些 Maven 存储库列在活跃配置集的项目 & **lt;repositories** > 元素中，以及 **settings.xml** 文件中的存储库。

```
$ mvn clean install
[INFO] Scanning for projects...
[INFO]

========== Red Hat Maven patching ==========

[INFO] [PATCH] Reading patch metadata and artifacts from 2 project repositories
[INFO] [PATCH]  - MRRC-GA: https://maven.repository.redhat.com/ga
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
```

8. 元数据是否来自远程存储库、本地存储库还是 ZIP 文件，它由 **patch-maven-plugin** 分析。获取的元数据包含一个 CVE 列表，对于每个 CVE，我们有一个受影响的 Maven 工件列表（由 glob 模式和版本范围指定），以及一个包含给定 CVE 修复的版本。例如，

```
<?xml version="1.0" encoding="UTF-8" ?>

<<metadata xmlns="urn:redhat:patch-metadata:1">
   <product-bom groupId="com.redhat.camel.springboot.platform" artifactId="camel-spring-boot-bom" versions="[3.20,3.21)" />
   <cves>
   </cves>
   <fixes>
      <fix id="HF0-1" description="logback-classic (Example) - Version Bump">
         <affects groupId="ch.qos.logback" artifactId="logback-classic" versions="[1.0,1.3.0)" fix="1.3.0" />
      </fix>
   </fixes>
</metadata>
```

9. 最后，当迭代当前项目中的所有受管依赖项时，会查阅补丁元数据中指定的修复列表。这些匹配的依赖项（和受管依赖项）会改为固定版本。例如：

```
$ mvn dependency:tree

[INFO] Scanning for projects...
[INFO]

========== Red Hat Maven patching ==========

[INFO] [PATCH] Reading patch metadata and artifacts from 3 project repositories
[INFO] [PATCH]  - redhat-ga-repository: http://maven.repository.redhat.com/ga/
[INFO] [PATCH]  - local: file:///path/to/.m2/repository
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
```

```
[INFO] [PATCH] Resolved patch
descriptor:/path/to/.m2/repository/com/redhat/camel/springboot/platform/redhat-camel-spring-
boot-patch-metadata/3.20.1.redhat-00043/redhat-camel-spring-boot-patch-metadata-
3.20.1.redhat-00043.xml
[INFO] [PATCH] Patch metadata found for com.redhat.camel.springboot.platform/camel-
spring-boot-bom/[3.20,3.21)
[INFO] [PATCH]  - patch contains 1 patch fix
[INFO] [PATCH] Processing managed dependencies to apply patch fixes...
[INFO] [PATCH] - HF0-1: logback-classic (Example) - Version Bump
[INFO] [PATCH]   Applying change ch.qos.logback/logback-classic/[1.0,1.3.0) -> 1.3.0
[INFO] [PATCH]   Project com.test:yaml-routes
[INFO] [PATCH]    - managed dependency: ch.qos.logback/logback-classic/1.2.11 -> 1.3.0
[INFO] [PATCH] Done in 39ms

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

跳过补丁

如果您不想将特定补丁应用到项目，则 **patch-maven-plugin** 提供了一个 **skip** 选项。假设已将 **patch-maven-plugin** 添加到项目的 **pom.xml** 文件中，且您不想更改版本，您可以使用以下任一方法跳过补丁。

- 将 skip 选项添加到项目的 **pom.xml** 文件中，如下所示：

```
<build>
   <plugins>
      <plugin>
         <groupId>com.redhat.camel.springboot.platform</groupId>
         <artifactId>patch-maven-plugin</artifactId>
         <version>${camel-spring-boot-version}</version>
         <extensions>true</extensions>
         <configuration>
            <skip>true</skip>
         </configuration>
      </plugin>
   </plugins>
</build>
```

- 或者在运行 **mvn** 命令时使用 **-DskipPatch** 选项，如下所示：

```
$ mvn clean install -DskipPatch
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------< com.example:test-csb >-------------------------
[INFO] Building A Camel Spring Boot Route 1.0-SNAPSHOT
...
```

如上述输出中所示，**patch-maven-plugin** 没有被调用，这会导致补丁不会应用到应用。

1.8. CAMEL REST DSL OPENAPI MAVEN 插件

Camel REST DSL OpenApi Maven 插件支持以下目标：

- camel-restdsl-openapi:generate - 从 OpenApi 规格生成消费者 REST DSL RouteBuilder 源代码

- camel-restdsl-openapi:generate-with-dto - 从 OpenApi 规范生成消费者 REST DSL RouteBuilder 源代码，以及通过 swagger-codegen-maven-plugin 生成的 DTO 模型类。
- camel-restdsl-openapi:generate-xml - 从 OpenApi 规格生成消费者 REST DSL XML 源代码
- camel-restdsl-openapi:generate-xml-with-dto - 从 OpenApi 规范生成消费者 REST DSL XML 源代码，以及通过 swagger-codegen-maven-plugin 生成的 DTO 模型类。
- camel-restdsl-openapi:generate-yaml - 要从 OpenApi 规格生成消费者 REST DSL YAML 源代码
- camel-restdsl-openapi:generate-yaml-with-dto - 从 OpenApi 规范生成消费者 REST DSL YAML 源代码，以及通过 swagger-codegen-maven-plugin 生成的 DTO 模型类。

1.8.1. 将插件添加到 Maven pom.xml

此插件可以添加到 Maven **pom.xml** 文件中，方法是将它添加到 **plugins** 部分，例如在 Spring Boot 应用程序中：

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

    <plugin>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-restdsl-openapi-plugin</artifactId>
      <version>{CamelCommunityVersion}</version>
    </plugin>

  </plugins>
</build>
```

然后，可以使用其前缀 **camel-restdsl-openapi** 执行插件，如下所示。

```
$mvn camel-restdsl-openapi:generate
```

1.8.2. camel-restdsl-openapi:generate

Camel REST DSL OpenApi Maven 插件的目标用于从 Maven 生成 REST DSL RouteBuilder 实施源代码。

1.8.3. 选项

该插件支持从命令行（使用 **-D** 语法）配置以下选项，或者在 **配置** 标签中的 **pom.xml** 文件中定义。

参数	默认值	描述
skip	**false**	设置为 **true** 以跳过代码生成

参数	默认值	描述
filterOperation		仅用于包括指定的操作 ID。可以使用逗号分隔多个 id。可以使用通配符，例如 **find*** 以包含以 **find** 开头的所有操作。
specificationUri	**src/spec/openapi.json**	OpenApi 规格的 URI 支持文件系统路径(HTTP 和 classpath 资源)，默认为项目目录中的 **src/spec/openapi.json**。支持 JSON 和 YAML。
auth		在远程获取 OpenApi 规格定义时添加授权标头。使用以逗号分隔的多个值的 URL 编码字符串 name:header 传递。
className	来自 **title** 或 **RestDslRoute**	生成的类的名称，默认从 OpenApi 规格标题或设置为 **RestDslRoute**
packageName	从 **host** 或 **rest.dsl.generated**	生成的类的软件包名称，默认为 OpenApi 规格主机值或 **rest.dsl.generated**
indent	**" "**	默认情况下，使用缩进字符（默认为四个空格），您可以使用 **\t** 表示标签页字符
outputDirectory	**generated-sources/restdsl-openapi**	将生成的源文件放在项目目录中，**默认生成源文件/restdsl-openapi**
destinationGenerator		实现 **org.apache.camel.generator.openapi.DestinationGenerator** 接口（用于自定义目标端点）的类的完全限定域名
destinationToSyntax	**direct:${operationId}**	to uri 的默认语法是使用直接组件。
restConfiguration	**true**	是否包括要被检测到的其余组件的生成。
apiContextPath		如果 **restConfiguration** 设为 true，则定义 openapi 端点路径。

参数	默认值	描述
clientRequestValidation	**false**	是否启用请求验证。
basePath		覆盖 OpenAPI 规格中定义的 api 基础路径。
requestMappingValues	/**	允许生成自定义 RequestMapping 映射值。多个映射值可以传递： **<requestMappingValues> <param>/my-api-path/ </param> <param>/my-other-path/ </param> </requestMappingValues>**

1.8.4. 带有 Servlet 组件的 Spring Boot 项目

如果 Maven 项目是 Spring Boot 项目，并且启用了 **restConfiguration**，并且 servlet 组件被用作 REST 组件，则此插件将自动检测软件包名称（如果尚未明确配置，则 **@SpringBootApplication** 主类位于 @SpringBootApplication 主类），并使用相同的软件包名称来生成 Rest DSL 源代码和所需的 **CamelRestController** 支持类。

1.8.5. camel-restdsl-openapi:generate-with-dto

作为 **生成** 目标，还可以通过自动执行 swagger-codegen-maven-plugin 来生成 DTO 模型类来生成来自 OpenApi 规范的 DTO 模型类的 java 源代码。

此插件的范围仅限于只支持使用 swagger-codegen-maven-plugin 生成模型 DTOs 的默认值。如果您需要更多电源和灵活性，则直接使用 Swagger Codegen Maven 插件 来生成 DTO 而不是此插件。

DTO 类可能需要其他依赖项，例如：

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-core</artifactId>
  <version>2.2.8</version>
</dependency>
<dependency>
  <groupId>org.threeten</groupId>
  <artifactId>threetenbp</artifactId>
  <version>1.6.8</version>
</dependency>
```

1.8.6. 选项

插件支持以下 **附加选项**

参数	默认值	描述
swaggerCodegenMavenPluginVersion	3.0.36	要使用的 **io.swagger.codegen.v3:swagger-codegen-maven-plugin** maven 插件的版本。
modelOutput		目标输出路径（默认为 ${project.build.directory}/generated-sources/openapi）
modelPackage	**io.swagger.client.model**	用于生成的模型对象/类的软件包
modelNamePrefix		为模型类和枚举设置 pre- 或 suffix
modelNameSuffix		为模型类和枚举设置 pre- 或 suffix
modelWithXml	false	在生成的模型中启用 XML 注解（仅适用于提供对 JSON 和 XML 支持的库）
configOptions		将特定语言参数映射传递给 **swagger-codegen-maven-plugin**

1.8.7. camel-restdsl-openapi:generate-xml

Camel REST DSL OpenApi Maven 插件的 **camel-restdsl-openapi:generate-xml** 目标用于从 Maven 生成 REST DSL XML 实施源代码。

1.8.8. 选项

该插件支持从命令行（使用 **-D** 语法）配置以下选项，或者在 < **configuration** > 标签中的 **pom.xml** 文件中定义。

参数	默认值	描述
skip	**false**	设置为 **true** 以跳过代码生成。
filterOperation		仅用于包括指定的操作 ID。可以使用逗号分隔多个 id。可以使用通配符，例如 **find*** 以包含以 **find** 开头的所有操作。
specificationUri	**src/spec/openapi.json**	OpenApi 规格的 URI 支持文件系统路径(HTTP 和 classpath 资源)，默认为项目目录中的 **src/spec/openapi.json**。支持 JSON 和 YAML。

参数	默认值	描述
auth		在远程获取 OpenApi 规格定义时添加授权标头。使用以逗号分隔的多个值的 URL 编码字符串 name:header 传递。
outputDirectory	**generated-sources/restdsl-openapi**	将生成的源文件放在项目目录中，**默认生成源文件/restdsl-openapi**
fileName	**camel-rest.xml**	作为输出的 XML 文件的名称。
blueprint	**false**	如果启用，则生成 OSGi 蓝图 XML 而不是 Spring XML。
destinationGenerator		实现 **org.apache.camel.generator.openapi.DestinationGenerator** 接口（用于自定义目标端点）的类的完全限定域名
destinationToSyntax	**direct:${operationId}**	to uri 的默认语法是使用直接组件。
	restConfiguration	**true**
是否包括要被检测到的其余组件的生成。	**apiContextPath**	
如果 **restConfiguration** 设为 **true**，则定义 openapi 端点路径。	**clientRequestValidation**	**false**
是否启用请求验证。	**basePath**	
覆盖 OpenAPI 规格中定义的 api 基础路径。	**requestMappingValues**	/**

1.8.9. camel-restdsl-openapi:generate-xml-with-dto

作为 **generate-xml** 目标，还通过自动执行 swagger-codegen-maven-plugin 生成 DTO 模型类来生成来自 OpenApi 规范的 DTO 模型类的 java 源代码。

此插件的范围仅限于只支持使用 swagger-codegen-maven-plugin 生成模型 DTOs 的默认值。如果您需要更多电源和灵活性，则直接使用 Swagger Codegen Maven 插件 来生成 DTO 而不是此插件。

DTO 类可能需要其他依赖项，例如：

```
<dependency>
  <groupId>com.google.code.gson</groupId>
```

```
    <artifactId>gson</artifactId>
    <version>2.10.1</version>
  </dependency>
  <dependency>
    <groupId>io.swagger.core.v3</groupId>
    <artifactId>swagger-core</artifactId>
    <version>2.2.8</version>
  </dependency>
  <dependency>
    <groupId>org.threeten</groupId>
    <artifactId>threetenbp</artifactId>
    <version>1.6.8</version>
  </dependency>
```

1.8.10. 选项

插件支持以下 **附加**选项

参数	默认值	描述
swaggerCodegenMavenPluginVersion	3.0.36	要使用的 **io.swagger.codegen.v3:swagger-codegen-maven-plugin** maven 插件的版本。
modelOutput		目标输出路径（默认为 ${project.build.directory}/generated-sources/openapi）
modelPackage	**io.swagger.client.model**	用于生成的模型对象/类的软件包
modelNamePrefix		为模型类和枚举设置 pre- 或 suffix
modelNameSuffix		为模型类和枚举设置 pre- 或 suffix
modelWithXml	false	在生成的模型中启用 XML 注解（仅适用于提供对 JSON 和 XML 支持的库）
configOptions		将特定语言参数映射传递给 **swagger-codegen-maven-plugin**

1.8.11. camel-restdsl-openapi:generate-yaml

Camel REST DSL OpenApi Maven 插件的 **camel-restdsl-openapi:generate-yaml** 目标用于从 Maven 生成 REST DSL YAML 实现源代码。

1.8.12. 选项

该插件支持从命令行（使用 **-D** 语法）配置以下选项，或者在 < **configuration** > 标签中的 **pom.xml** 文件中定义。

参数	默认值	描述
skip	**false**	设置为 **true** 以跳过代码生成。
filterOperation		仅用于包括指定的操作 ID。可以使用逗号分隔多个 id。可以使用通配符，例如 **find*** 以包含以 **find** 开头的所有操作。
specificationUri	**src/spec/openapi.json**	OpenApi 规格的 URI 支持文件系统路径(HTTP 和 classpath 资源)，默认为项目目录中的 **src/spec/openapi.json**。支持 JSON 和 YAML。
auth		在远程获取 OpenApi 规格定义时添加授权标头。使用以逗号分隔的多个值的 URL 编码字符串 name:header 传递。
outputDirectory	**generated-sources/restdsl-openapi**	将生成的源文件放在项目目录中，**默认生成源文件/restdsl-openapi**
fileName	**camel-rest.xml**	作为输出的 XML 文件的名称。
destinationGenerator		实现 **org.apache.camel.generator.openapi.DestinationGenerator** 接口（用于自定义目标端点）的类的完全限定域名
destinationToSyntax	**direct:${operationId}**	to uri 的默认语法是使用直接组件。
	restConfiguration	**true**
是否包括要被检测到的其余组件的生成。	**apiContextPath**	
如果 **restConfiguration** 设为 **true**，则定义 openapi 端点路径。	**clientRequestValidation**	**false**
是否启用请求验证。	**basePath**	
覆盖 OpenAPI 规格中定义的 api 基础路径。	**requestMappingValues**	**/**

1.8.13. camel-restdsl-openapi:generate-yaml-with-dto

作为 **generate-yaml** 目标，还通过自动执行 swagger-codegen-maven-plugin 来生成 DTO 模型类来生成来自 OpenApi 规范的 DTO 模型类的 java 源代码。

此插件的范围仅限于只支持使用 **swagger-codegen-maven-plugin** 生成模型 DTOs 的默认值。如果您需要更多电源和灵活性，则直接使用 Swagger Codegen Maven 插件 来生成 DTO 而不是此插件。

DTO 类可能需要其他依赖项，例如：

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-core</artifactId>
  <version>2.2.8</version>
</dependency>
<dependency>
  <groupId>org.threeten</groupId>
  <artifactId>threetenbp</artifactId>
  <version>1.6.8</version>
</dependency>
```

1.8.14. 选项

插件支持以下 **附加选项**

参数	默认值	描述
swaggerCodegenMavenPluginVersion	3.0.36	要使用的 **io.swagger.codegen.v3:swagger-codegen-maven-plugin** maven 插件的版本。
modelOutput		目标输出路径（默认为 ${project.build.directory}/generated-sources/openapi）
modelPackage	**io.swagger.client.model**	用于生成的模型对象/类的软件包
modelNamePrefix		为模型类和枚举设置 pre- 或 suffix
modelNameSuffix		为模型类和枚举设置 pre- 或 suffix
modelWithXml	false	在生成的模型中启用 XML 注解（仅适用于提供对 JSON 和 XML 支持的库）

参数	默认值	描述
configOptions		将特定语言参数映射传递给 **swagger-codegen-maven-plugin**

1.9. 支持 FIPS 合规性

您可以在 x86_64 架构上安装使用 FIPS 验证的/Modules in Process 加密库的 OpenShift Container Platform 集群。

对于集群中的 Red Hat Enterprise Linux CoreOS (RHCOS)机器，当机器根据 install-config.yaml 文件中的选项状态进行部署时，会应用此更改，该文件管理用户在集群部署期间可以更改的集群选项。在 Red Hat Enterprise Linux (RHEL)机器中，您必须在计划用作 worker 机器的机器上安装操作系统时启用 FIPS 模式。这些配置方法可确保集群满足 FIPS 合规审计的要求。在初始系统引导前，只启用 FIPS 验证的/Modules in Process 加密软件包。

因为您必须在集群操作系统第一次引导前启用 FIPS，所以无法在部署集群后启用 FIPS。

1.9.1. OpenShift Container Platform 中的 FIPS 验证

OpenShift Container Platform 在 RHEL 和 RHCOS 中使用特定的 FIPS 验证的/Modules in Process 模块用于其操作系统组件。例如，当用户 SSH 到 OpenShift Container Platform 集群和容器时，这些连接会被正确加密。

OpenShift Container Platform 组件以 Go 编写，并使用红帽的 Golang 编译器构建。当您为集群启用 FIPS 模式时，需要加密签名的所有 OpenShift Container Platform 组件都会调用 RHEL 和 RHCOS 加密库。

有关 FIPS 的详情，请参阅 FIPS 模式属性和限制

有关在 OpenShift 上部署 Camel Spring Boot 的详情，请参阅 如何将 Camel Spring Boot 应用程序部署到 OpenShift？

有关支持的配置的详情，请参考 Camel for Spring Boot 支持的配置

第 2 章 在本地设置 MAVEN

Maven 是红帽构建的 Apache Camel 应用程序开发和项目管理的典型选择。

2.1. 准备设置 MAVEN

Maven 是一个来自 Apache 的免费开源构建工具。

步骤

1. 从 Maven 下载页面 下载 Maven 3.8.6 或更高版本。

   提示

   要验证您已安装了正确的 Maven 和 JDK 版本，请打开终端并输入以下命令：

   ```
   mvn --version
   ```

   检查输出，以验证 Maven 版本 3.8.6 或更新版本，并使用 OpenJDK 17。

2. 确定您的系统已连接到互联网。
   在构建项目时，默认行为是 Maven 搜索外部存储库并下载所需的工件。Maven 查找可通过互联网访问的存储库。

   您可以更改此行为，以便 Maven 仅搜索位于本地网络上的存储库。也就是说，Maven 可以在离线模式下运行。在离线模式中，Maven 会在其本地存储库中查找工件。请参阅 第 2.4 节 “使用本地 Maven 存储库”。

2.2. 将红帽软件仓库添加到 MAVEN

要访问位于 Red Hat Maven 存储库中的工件，您需要将这些存储库添加到 Maven 的 **settings.xml** 文件中。

Maven 在用户主目录的 **.m2** 目录中查找 **settings.xml** 文件。如果没有用户指定的 **settings.xml** 文件，Maven 将使用 **M2_HOME/conf/ settings.xml** 中的系统级 settings.xml 文件。

前提条件

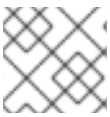
您知道要在其中添加红帽软件仓库的 **settings.xml** 文件的位置。

步骤

在 **settings.xml** 文件中，为红帽 **软件仓库** 添加软件仓库元素，如下例所示：

注意

如果您使用 **camel-jira** 组件，还要添加 atlassian 存储库。



注意

如果要使用技术预览构建，还要添加 **earlyaccess** 存储库。

```
<?xml version="1.0"?>
```

```
<settings>

 <profiles>
  <profile>
   <id>extra-repos</id>
   <activation>
    <activeByDefault>true</activeByDefault>
   </activation>
   <repositories>
    <repository>
        <id>redhat-ga-repository</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>redhat-ea-repository</id>
        <url>https://maven.repository.redhat.com/earlyaccess/all</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>atlassian</id>
        <url>https://packages.atlassian.com/maven-external/</url>
        <name>atlassian external repo</name>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <releases>
            <enabled>true</enabled>
        </releases>
    </repository>
   </repositories>
   <pluginRepositories>
    <pluginRepository>
        <id>redhat-ga-repository</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>redhat-ea-repository</id>
        <url>https://maven.repository.redhat.com/earlyaccess/all</url>
        <releases>
```

```
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>extra-repos</activeProfile>
  </activeProfiles>

</settings>
```

2.3. 构建离线 MAVEN 存储库

Red Hat build of Apache Camel for Spring Boot 用户可以构建自己的离线 Maven 存储库，该存储库在受限环境中使用。对于红帽构建的 Apache Camel for Spring Boot 用户的每个发行版本，可以从红帽客户门户网站下载 zip 文件。

步骤

1. 从客户门户网站下载 file Maven 存储库构建器。例如，对于红帽构建的 Camel Spring Boot 版本 4.4，请使用 离线 Maven 构建器。
2. 下载的文件是一个 zip 文件，其中包含为这个特定版本构建离线 Maven 存储库的所有内容。
3. 解压下载的 zip 文件。归档的目录结构如下：

```
├── README
├── build-offline-repo.sh
├── errors.log
├── logback.xml
├── maven-repositories.txt
├── offliner-2.0-sources.jar
├── offliner-2.0-sources.jar.md5
├── offliner-2.0.jar
├── offliner-2.0.jar.md5
├── offliner.log
├── rhaf-camel-offliner-4.4.0.txt
└── rhaf-camel-spring-boot-offliner-4.4.0.txt
```

这个 zip 包含以下文件：

- build-offline-repo.sh - 离线工具的打包程序脚本。
- offliner-2.0.jar - 下载清单中的工件。
- redhat-camel-4.4.0-offline-manifest.txt
  - 列出需要下载所需的工件。
- redhat-camel-spring-boot-4.4.0-offline-manifest.txt

- 列出需要下载所需的工件。

- README - 解释构建离线 Maven 存储库所需的步骤和命令。

4. 要构建离线存储库，请按照 **README** 文件中给出的说明运行 **build-offline-repo.sh** 脚本。（可选）您可以指定应下载工件的目录。如果没有指定，则会在当前工作目录中创建名为"repository"的目录。

如果需要，您可以将工具配置为使用额外的 Maven 存储库，方法是将它们添加到 **maven-repositories.txt** 文件中。这通常不需要，因为该工具已预先配置了正确的 Maven 存储库集合。

如果是 HTTP 代理以及需要通过此代理进行的任何 HTTP 调用，您可能需要更改脚本。在调用脚本中的 JVM **的行中添加 --proxy <proxy-host> --proxy-user> --proxy-pass &** lt;proxy-pass> 参数。

您可以使用 **-v** 选项打印脚本的版本号。这个版本是脚本的版本号，与 Red Hat build of Apache Camel 产品版本无关。

故障排除

您可以通过提供的 **logback.xml** 文件配置日志。执行 shell 脚本时，任何下载活动都将写入到日志文件 **offliner.log** 中，并在 **errors.log** 中列出任何下载失败。在执行 offliner 工具的末尾会显示下载和失败的工件摘要，但我们还建议通过 error **.log** 扫描任何下载失败。

如果无法下载任何工件，请针对同一目标文件夹重新运行该工具。该工具可以避免下载已下载工件，并且只尝试之前失败的工件。

2.4. 使用本地 MAVEN 存储库

如果您在没有互联网连接的情况下运行容器，并且需要部署一个具有离线依赖项的应用程序，您可以使用 Maven 依赖项插件将应用的依赖项下载到 Maven 离线存储库中。然后，您可以将此自定义 Maven 离线存储库分发到没有互联网连接的机器。

步骤

1. 在包含 **pom.xml** 文件的项目目录中，运行以下命令来下载 Maven 项目的存储库，如下所示：

```
mvn org.apache.maven.plugins:maven-dependency-plugin:3.1.0:go-offline -
Dmaven.repo.local=/tmp/my-project
```

   在本例中，构建项目所需的 Maven 依赖项和插件将下载到 **/tmp/my-project** 目录中。

2. 将此自定义 Maven 离线存储库在内部分发到没有互联网连接的任何机器。

2.5. 使用环境变量或系统属性设置 MAVEN 镜像

在运行应用程序时，您需要访问 Red Hat Maven 存储库中的工件。这些存储库添加到 Maven 的 **settings.xml** 文件中。Maven 检查 **settings.xml** 文件的以下位置：

- 查找指定的 url

- if not found looks for **${user.home}/.m2/settings.xml**

- 如果没有找到 **${maven.home}/conf/settings.xml**的查找

- 如果未找到，则查找 **${M2_HOME}/conf/settings.xml**

- 如果没有找到位置，则会创建空的 **org.apache.maven.settings.Settings** 实例。

2.5.1. 关于 Maven 镜像

Maven 使用一组远程存储库来访问工件，这些工件目前在本地存储库中不可用。存储库列表几乎始终包含 Maven Central 存储库，但对于 Red Hat Fuse，它还包含 Maven 红帽存储库。在某些情况下，无法访问不同的远程存储库，您可以使用 Maven 镜像的机制。镜像将特定的存储库 URL 替换为不同的存储库 URL，因此搜索远程工件时的所有 HTTP 流量都可以定向到单个 URL。

2.5.2. 在 **settings.xml**中添加 Maven 镜像

要设置 Maven 镜像，请在 Maven 的 **settings.xml** 中添加以下部分：

```
<mirror>
    <id>all</id>
    <mirrorOf>*</mirrorOf>
    <url>http://host:port/path</url>
</mirror>
```

如果在 **settings.xml** 文件中找不到以上部分，则不会使用 mirror。要在不提供 XML 配置的情况下指定全局镜像，您可以使用系统属性或环境变量。

2.5.3. 使用环境变量或系统属性设置 Maven 镜像

要使用环境变量或系统属性设置 Maven 镜像，您可以添加：

- 名为 **MAVEN_MIRROR_URL** 的环境变量到 **bin/setenv** 文件
- 名为 **mavenMirrorUrl** to **etc/system.properties** 文件的系统属性

2.5.4. 使用 Maven 选项指定 Maven 镜像 url

要使用替代的 Maven 镜像 url，在环境变量或系统属性之外，在运行应用程序时使用以下 maven 选项：

- **-DmavenMirrorUrl=mirrorId::mirrorUrl**
  for example, **-DmavenMirrorUrl=my-mirror::http://mirror.net/repository**
- **-DmavenMirrorUrl=mirrorUrl**
  例如，**-DmavenMirrorUrl=http://mirror.net/repository**。在本例中，<mirror> 的 <id> 只是一个镜像(mirror)。

2.6. 关于 MAVEN 工件和协调

在 Maven 构建系统中，基本构建块是一个 *工件*。构建后，工件的输出通常是一个存档，如 JAR 或 WAR 文件。

Maven 的一个关键方面是能够找到工件和管理它们之间的依赖关系。*Maven 协调* 是一组用于标识特定工件位置的值。基本协调有三个值，格式为：

groupId:artifactId:version

有时，Maven 通过 *打包值或使用打包* 值和 *分类器* 值增加一个基本的协调。*Maven 协调可以具有以下形式之一：*

```
groupId:artifactId:version
groupId:artifactId:packaging:version
groupId:artifactId:packaging:classifier:version
```

以下是值的描述：

groupdId

定义工件名称的范围。您通常使用所有或部分软件包名称作为组 ID。例如，**org.fusesource.example**。

artifactId

定义相对于组 ID 的工件名称。

version

指定工件的版本。版本号最多可有四个部分：**n.n.n.n**，其中版本号的最后一部分可以包含非数字字符。例如，**1.0-SNAPSHOT** 的最后一部分是字母数字字符子字符串 **0-SNAPSHOT**。

打包

定义构建项目时生成的打包实体。对于 OSGi 项目，打包是 **捆绑包**。默认值为 **jar**。

分类器

可让您区分从同一 POM 构建但具有不同内容的工件。

工件的 POM 文件中的元素定义工件的组 ID、工件 ID、打包和版本，如下所示：

```
<project ... >
  ...
  <groupId>org.fusesource.example</groupId>
  <artifactId>bundle-demo</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  ...
</project>
```

要定义对上述工件的依赖项，您可以在 POM 文件中添加以下 dependencies 元素：

```
<project ... >
  ...
  <dependencies>
    <dependency>
      <groupId>org.fusesource.example</groupId>
      <artifactId>bundle-demo</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
  ...
</project>
```

注意

*不需要在前面的依赖项中指定 **bundle** 软件包类型，因为捆绑包只是特定类型的 **JAR** 文件，**jar** 是默认的 **Maven** 软件包类型。但是，如果您需要在依赖项中明确指定打包类型，您可以使用 **type** 元素。*

第 3 章 监控 CAMEL SPRING BOOT 集成

本章介绍了如何在运行时监控红帽构建的 Camel Spring Boot 的集成。您可以使用已部署为 OpenShift Monitoring 一部分的 Prometheus Operator 来监控您自己的应用程序。

3.1. 在 OPENSHIFT 中启用用户工作负载监控

您可以通过在集群监控 ConfigMap 中设置 enableUserWorkload: true 字段来为用户定义的项目启用监控。

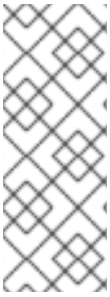
重要

在 OpenShift Container Platform 4.13 中，要为用户定义的项目启用监控，您必须先删除任何自定义 Prometheus 实例。

先决条件

您必须可以使用具有 cluster-admin 集群角色访问权限的用户访问集群，才能在 OpenShift Container Platform 中为用户定义的项目启用监控。然后，集群管理员可以选择性地授予用户权限来配置负责监控用户定义的项目的组件。

- 有集群管理员对 OpenShift 集群的访问权限。
- 已安装 OpenShift CLI（oc）。
- 您已创建 cluster-monitoring-config ConfigMap 对象。
- 可选：您已在 openshift-user-workload-monitoring 项目中创建并配置 user-workload-monitoring-config ConfigMap 对象。您可以在此 ConfigMap 中为监控用户定义的项目的组件添加配置选项。



注意

每次您将配置更改保存到 **user-workload-monitoring-config ConfigMap** 对象时，都会重新部署 **openshift-user-workload-monitoring** 项目中的 **Pod**。有时重新部署这些组件需要花费一段时间。在首次为用户定义的项目启用监控前，您可以创建和配置 **ConfigMap** 对象，以防止经常重新部署 **Pod**。

步骤

1. 使用管理员权限登录到 **OpenShift**。

```
oc login --user system:admin --token=my-token --server=https://my-cluster.example.com:6443
```

2. 编辑 **cluster-monitoring-config ConfigMap** 对象。

```
$ oc -n openshift-monitoring edit configmap cluster-monitoring-config
```

3. 在 **data/config.yaml** 部分添加 **enableUserWorkload: true**。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
    enableUserWorkload: true
```

当设置为 **true** 时，**enableUserWorkload** 参数为集群中用户定义的项目启用监控。

4. 保存文件以使改变生效。然后会自动启用对用户定义的项目的监控。

注意

将更改保存到 **cluster-monitoring-config ConfigMap** 对象时，可能会重新部署 **openshift-monitoring** 项目中的 **Pod** 和其他资源。该项目中正在运行的监控进程也可能被重启。

5.

验证 prometheus-operator、prometheus-user-workload 和 thanos-ruler-user-workload Pod 是否在 openshift-user-workload-monitoring 项目中运行。

```
$ oc -n openshift-user-workload-monitoring get pod

  Example output

  NAME                                  READY   STATUS    RESTARTS   AGE
  prometheus-operator-6f7b748d5b-t7nbg   2/2     Running   0          3h
  prometheus-user-workload-0             4/4     Running   1          3h
  prometheus-user-workload-1             4/4     Running   1          3h
  thanos-ruler-user-workload-0           3/3     Running   0          3h
  thanos-ruler-user-workload-1           3/3     Running   0          3h
```

3.2. 部署 CAMEL SPRING BOOT 应用程序

为项目启用监控后，您可以部署并监控 Camel Spring Boot 应用程序。本节使用 Camel Spring Boot 示例中列出的 monitoring-micrometrics-grafana-prometheus 示例。

步骤

1. *将 openshift-maven-plugin 添加到 monitoring-micrometrics-grafana-prometheus 示例中的 pom.xml 文件中。在 pom.xml 中，添加一个 openshift 配置集，以允许通过 openshift-maven-plugin 部署到 openshift。*

```
<profiles>
  <profile>
    <id>openshift</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.eclipse.jkube</groupId>
          <artifactId>openshift-maven-plugin</artifactId>
          <version>1.13.1</version>
          <executions>
            <execution>
              <goals>
                <goal>resource</goal>
                <goal>build</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

2.

添加 Prometheus 支持。要将 Prometheus 支持添加到 Camel Spring Boot 应用程序，请在 actuator 端点上公开 Prometheus 统计。

a. **编辑 src/main/resources/application.properties 文件。如果您有 management.endpoints.web.exposure.include 条目，请添加 prometheus、metrics 和 health。如果您没有 management.endpoints.web.exposure.include 条目，请添加一个。**

```
# expose actuator endpoint via HTTP
management.endpoints.web.exposure.include=mappings,metrics,health,shutdown,jolokia,p
ometheus
```

3. **将以下内容添加到 pom.xml 的 <dependencies /> 部分，为您的应用程序添加一些入门支持。**

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>

<dependency>
  <groupId>org.jolokia</groupId>
  <artifactId>jolokia-core</artifactId>
  <version>${jolokia-version}</version>
</dependency>

<dependency>
  <groupId>io.prometheus.jmx</groupId>
  <artifactId>collector</artifactId>
  <version>${prometheus-version}</version>
</dependency>
```

4. **将以下内容添加到 Camel Spring Boot 应用程序的 Application.java 中。**

```
import org.springframework.context.annonation.Bean;
import org.apache.camel.component.micrometer.MicrometerConstants;
import
org.apache.camel.component.micrometer.eventnotifier.MicrometerExchangeEventNoti
fier;
import
org.apache.camel.component.micrometer.eventnotifier.MicrometerRouteEventNotifier;
import
org.apache.camel.component.micrometer.messagehistory.MicrometerMessageHistory
Factory;
import
org.apache.camel.component.micrometer.routepolicy.MicrometerRoutePolicyFactory;
```

5.

更新的 Application.java 如下所示。

```
@SpringBootApplication
public class SampleCamelApplication {

@Bean(name = {MicrometerConstants.METRICS_REGISTRY_NAME,
"prometheusMeterRegistry"})
public PrometheusMeterRegistry prometheusMeterRegistry(
        PrometheusConfig prometheusConfig, CollectorRegistry collectorRegistry, Clock
clock) throws MalformedObjectNameException, IOException {

    InputStream resource = new
ClassPathResource("config/prometheus_exporter_config.yml").getInputStream();

    new JmxCollector(resource).register(collectorRegistry);
    new BuildInfoCollector().register(collectorRegistry);
    return new PrometheusMeterRegistry(prometheusConfig, collectorRegistry, clock);
}

@Bean
public CamelContextConfiguration camelContextConfiguration(@Autowired
PrometheusMeterRegistry registry) {

    return new CamelContextConfiguration() {
        @Override
        public void beforeApplicationStart(CamelContext camelContext) {
            MicrometerRoutePolicyFactory micrometerRoutePolicyFactory = new
MicrometerRoutePolicyFactory();
            micrometerRoutePolicyFactory.setMeterRegistry(registry);
            camelContext.addRoutePolicyFactory(micrometerRoutePolicyFactory);

            MicrometerMessageHistoryFactory micrometerMessageHistoryFactory = new
MicrometerMessageHistoryFactory();
            micrometerMessageHistoryFactory.setMeterRegistry(registry);
            camelContext.setMessageHistoryFactory(micrometerMessageHistoryFactory);

            MicrometerExchangeEventNotifier micrometerExchangeEventNotifier =  new
MicrometerExchangeEventNotifier();
            micrometerExchangeEventNotifier.setMeterRegistry(registry);

camelContext.getManagementStrategy().addEventNotifier(micrometerExchangeEvent
Notifier);

            MicrometerRouteEventNotifier micrometerRouteEventNotifier = new
MicrometerRouteEventNotifier();
            micrometerRouteEventNotifier.setMeterRegistry(registry);

camelContext.getManagementStrategy().addEventNotifier(micrometerRouteEventNotif
ier);

        }

        @Override
        public void afterApplicationStart(CamelContext camelContext) {
```

```
        }
    };
}
```

6. 将应用部署到 OpenShift。

```
mvn -Popenshift oc:deploy
```

7. 验证您的应用程序是否已部署。

```
oc get pods -n myapp

NAME                                    READY   STATUS      RESTARTS   AGE
camel-example-spring-boot-xml-2-deploy      0/1     Completed   0          13m
camel-example-spring-boot-xml-2-x78rk       1/1     Running     0          13m
camel-example-spring-boot-xml-s2i-2-build   0/1     Completed   0          14m
```

8. 添加此应用程序的 Service Monitor，以便 Openshift 的 prometheus 实例可以从 /actuator/prometheus 端点开始提取。

   a. 为 Service monitor 创建以下 YAML 清单。在本例中，该文件被命名为 servicemonitor.yaml。

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    k8s-app: csb-demo-monitor
  name: csb-demo-monitor
spec:
  endpoints:
  - interval: 30s
    port: http
    scheme: http
    path: /actuator/prometheus
  selector:
    matchLabels:
      app: camel-example-spring-boot-xml
```

   b. 为此应用程序添加 Service Monitor。

```
oc apply -f servicemonitor.yml
servicemonitor.monitoring.coreos.com/csb-demo-monitor "myapp" created
```

9. *验证服务监控器是否已成功部署。*

```
oc get servicemonitor

NAME                AGE
csb-demo-monitor    9m17s
```

10. *验证您可以在提取目标列表中看到服务监控器。在 Administrator 视图中，进入到 Observe → Targets。您可以在提取目标列表中找到 csb-demo-monitor。*

11. *在部署 servicemonitor 后等待大约十分钟。然后进入到 Developer 视图中的 Observe → Metrics。在下拉菜单中选择 Custom query 并键入 camel 来查看通过 /actuator/prometheus 端点公开的 Camel 指标。*

注意

红帽不支持在非 OCP 环境中安装和配置 Prometheus 和 Grafana。

第 4 章 使用带有 SPRING XML 的 CAMEL

将 Camel 与 Spring XML 文件搭配使用是一种在 Camel 中使用 XML DSL 的方法。Camel 过去一直在长期使用 Spring XML。Spring 框架启动了 XML 文件，作为构建 Spring 应用程序的常用配置。

Spring 应用程序示例

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
    ">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:a"/>
            <choice>
                <when>
                    <xpath>$foo = 'bar'</xpath>
                    <to uri="direct:b"/>
                </when>
                <when>
                    <xpath>$foo = 'cheese'</xpath>
                    <to uri="direct:c"/>
                </when>
                <otherwise>
                    <to uri="direct:d"/>
                </otherwise>
            </choice>
        </route>
    </camelContext>

</beans>
```

4.1. 在 SPRING XML 文件中使用 JAVA DSL

您可以使用 Java Code 定义 RouteBuilder 实施。它们在 spring 中被定义为 beans，然后在 camel 上下文中引用，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <routeBuilder ref="myBuilder"/>
```

```
</camelContext>

<bean id="myBuilder" class="org.apache.camel.spring.example.test1.MyRouteBuilder"/>
```

4.1.1. 配置 Spring Boot 应用程序

要将 Spring Boot Autoconfigure XML 路由用于 Bean，您需要导入 XML 资源。为此，您可以使用 Configuration 类。

例如，如果 Spring XML 文件位于 src/main/resources/camel-context.xml，您可以使用以下配置类来加载 camel-context：

示例：使用 Configuration 类

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;


/**
 * A Configuration class that import the Spring XML resource
 */
@Configuration
// load the spring xml file from classpath
@ImportResource("classpath:camel-context.xml")
public class CamelSpringXMLConfiguration {
}
```

4.2. 使用 SPRING XML 指定 CAMEL 路由

您可以使用 Spring XML 文件使用 XML DSL 指定 Camel 路由，如下所示：

```
<camelContext id="camel-A" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:start"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

4.3. 配置组件和端点

您可以在 Spring XML 中配置组件或 Endpoint 实例，如下例所示。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
</camelContext>

<bean id="jmsConnectionFactory"
class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp:someserver:61616"/>
</bean>
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp:someserver:61616"/>
    </bean>
  </property>
</bean>
```

这样，您可以使用任何名称配置组件，但通常使用相同的名称，如 jms。然后，您可以使用 jms:destinationName 引用组件。

这适用于 Camel 从用于 Endpoint URI 的方案名称的 Spring 上下文获取组件。

4.4. 使用软件包扫描

Camel 还提供强大的功能，允许在给定软件包中自动发现和初始化路由。这可以通过在 spring 上下文定义中的 camel 上下文中添加标签，指定要递归搜索 RouteBuilder 实施的软件包。要使用这个功能，请添加一个 <package></package> 标签，指定应搜索的、以逗号分隔的软件包列表。例如，

```
<camelContext>
  <packageScan>
    <package>com.foo</package>
    <excludes>**.*Excluded*</excludes>
    <includes>**.*</includes>
  </packageScan>
</camelContext>
```

这会扫描 com.foo 和 sub-packages 中的 RouteBuilder 类。

您还可以使用 includes 或 excludes 过滤类，例如：

```
<camelContext>
  <packageScan>
```

```
    <package>com.foo</package>
    <excludes>**.*Special*</excludes>
  </packageScan>
</camelContext>
```

这会跳过名称中包含 Special 的类。在包含模式前应用 exclude 模式。如果没有定义包含或排除模式，则返回软件包中发现的所有 Route 类。

? 匹配一个字符，X 匹配零个或更多字符，但匹配完全限定名称的零个或多个片段。

4.5. 使用上下文扫描

您可以允许 Camel 扫描容器上下文，例如，用于路由构建器实例的 Spring ApplicationContext。这可让您使用 Spring < component-scan& gt; 功能，并在扫描过程中选择由 Spring 创建的任何 RouteBuilder 实例。

```
<!-- enable Spring @Component scan -->
<context:component-scan base-package="org.apache.camel.spring.issues.contextscan"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- and then let Camel use those @Component scanned route builders -->
    <contextScan/>
</camelContext>
```

这可让您使用 Spring @Component 注解路由，并包含这些路由：

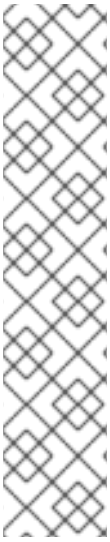
```
@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("direct:start")
            .to("mock:result");
    }
}
```

您还可以使用 ANT 样式包含和排除，如软件包扫描部分所述。

第 5 章 XML IO DSL

xml-io-dsl 是 Camel 优化的 XML DSL，具有非常快速、低的开销 XML 解析器。它是特定于 Camel 的源代码，只能解析 Camel .xml 路由文件（非经典 Spring < beans> XML 文件）。

我们建议您将 xml-io-dsl 而不是 xml-jaxb-dsl 用于 Camel XML DSL。它适用于所有 Camel 运行时。



注意

当您使用 XML IO DSL 时，camel-spring-boot 应用程序将默认在 src/main/resources/camel8:0:1::.xml 中查找 xml 文件。

您可以通过在 camel.springboot.routes-include-pattern 属性中提供不同的路径来配置此行为：

camel.springboot.routes-include-pattern=/path/to/*.xml

5.1. EXAMPLE

可以使用 Camel CLI 或 Camel K 加载并运行以下 my-route.xml 源文件：

my-route.xml

```
<routes xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="timer:tick"/>
    <setBody>
      <constant>Hello Camel K!</constant>
    </setBody>
    <to uri="log:info"/>
  </route>
</routes>
```

提示

您可以省略 xmlns 命名空间。

如果只有一个路由，您可以使用 <route> 作为 root XML 标签，而不是 < routes>。

使用 Camel K 运行

```
kamel run my-route.xml
```

使用 Camel CLI 运行

```
camel run my-route.xml
```

您可以使用 xml-io-dsl 声明要绑定到 Camel Registry 的一些 Bean。

您可以在 XML 中定义其属性（包括 嵌套属性 ）。例如：

Bean 声明和定义

```
<camel>

 <bean name="beanFromProps" type="com.acme.MyBean">
  <properties>
   <property key="field1" value="f1_p" />
   <property key="field2" value="f2_p" />
   <property key="nested.field1" value="nf1_p" />
   <property key="nested.field2" value="nf2_p" />
  </properties>
```

```
 </bean>

</camel>
```

虽然保留 xml-io-dsl 所使用的快速 XML 解析器的所有优点，但 Camel 也可以处理其他 XML 命名空间中声明的 XML 元素，并单独处理它们。使用这个机制，可以使用 Spring 的 http://www.springframework.org/schema/beans 命名空间包含 XML 元素。

这会将 Spring Beans 的灵活性引入 Camel 主，而无需实际运行任何 Spring 应用程序上下文（或 Spring Boot）。

当找到 Spring 命名空间中的元素时，它们用于填充和配置 org.springframework.beans.factory.support.DefaultListableBeanFactory 实例，并利用 Spring 依赖项注入将 Bean 链接到一起。

然后，这些 Bean 可以通过普通的 Camel Registry 公开，并可以被 Camel 路由使用。

以下是一个 camel.xml 文件示例，它定义了由路由定义使用的路由和 Bean （称为 ）：

camel.xml

```
<camel>

  <beans xmlns="http://www.springframework.org/schema/beans">
    <bean id="messageString" class="java.lang.String">
      <constructor-arg index="0" value="Hello"/>
    </bean>

    <bean id="greeter" class="org.apache.camel.main.app.Greeter">
      <description>Spring Bean</description>
      <property name="message">
        <bean class="org.apache.camel.main.app.GreeterMessage">
          <property name="msg" ref="messageString"/>
        </bean>
      </property>
    </bean>
  </beans>

  <route id="my-route">
```

```
        <from uri="direct:start"/>
        <bean ref="greeter"/>
        <to uri="mock:finish"/>
    </route>

</camel>
```

my-route 路由引用 greeter bean，该 bean 使用 Spring < bean> 元素定义。

Apache Camel JBang 页面中可以找到更多示例。

5.2. 使用带有构造器的 BEAN

当您要使用构造器参数创建 Bean 时，从 Camel 4.1 开始，您可以将其添加为 XML 标签。例如：

Camel 4.1+: Beans 带有 构造器 标签

```
<camel>

 <bean name="beanFromProps" type="com.acme.MyBean">
        <constructors>
          <constructor index="0" value="true"/>
          <constructor index="1" value="Hello World"/>
        </constructors>
        <!-- and you can still have properties -->
  <properties>
   <property key="field1" value="f1_p" />
   <property key="field2" value="f2_p" />
   <property key="nested.field1" value="nf1_p" />
   <property key="nested.field2" value="nf2_p" />
  </properties>
 </bean>

</camel>
```

如果使用 Camel 4.0，您必须将构造器参数放在 type 属性中：

Camel 4.0: Beans 在 type 属性中使用 构造器 参数

```
<bean name="beanFromProps" type="com.acme.MyBean(true, 'Hello World')">
    <properties>
        <property key="field1" value="f1_p" />
        <property key="field2" value="f2_p" />
        <property key="nested.field1" value="nf1_p" />
        <property key="nested.field2" value="nf2_p" />
    </properties>
</bean>
```

5.3. 从工厂方法创建 BEAN

Bean 也可以从 公共静态 工厂方法创建：

工厂方法 XML

```
<bean name="myBean" type="com.acme.MyBean" factoryMethod="createMyBean">
        <constructors>
          <constructor index="0" value="true"/>
          <constructor index="1" value="Hello World"/>
        </constructors>
</bean>
```

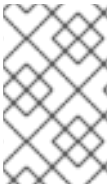
使用 factoryMethod 时，必须为参数 提供 构造标签。

例如，这意味着类 com.acme.MyBean 应如下所示：

工厂方法

```
public class MyBean {
```

```
    public static MyBean createMyBean(boolean important, String message) {
        MyBean answer = ...
        // create and configure the bean
        return answer;
    }
}
```



注意

您必须在创建的类中使工厂方法 public 静态。

5.4. 从构建器类创建 BEAN

您可以创建一个从另一个构建器类创建的 bean，如下所示：

构建器 XML

```
<bean name="myBean" type="com.acme.MyBean"
      builderClass="com.acme.MyBeanBuilder" builderMethod="createMyBean">
    <properties>
      <property key="id" value="123"/>
      <property key="name" value="Acme"/>
    </constructors>
</bean>
```

注意

您必须使用 no-arg 默认构造器 公开 构建器类。

然后，您可以使用 builder 类使用 fluent 构建器风格配置创建实际 bean。

在 builder 类上设置属性，并通过在末尾调用 builderMethod 来创建 bean。

您可以通过 Java 反映来改变此方法。

5.5. 从工厂 BEAN 创建 BEAN

您可以从工厂 Bean 创建 bean，如下所示：

工厂 XML

```
<bean name="myBean" type="com.acme.MyBean"
      factoryBean="com.acme.MyHelper" factoryMethod="createMyBean">
    <constructors>
      <constructor index="0" value="true"/>
      <constructor index="1" value="Hello World"/>
    </constructors>
</bean>
```

提示

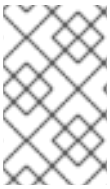
您还可以使用 factoryBean 来根据 bean id 而不是 FQN 类名称引用现有的 bean。

使用 factoryBean 时，您必须以 构造器 标签形式提供参数。

例如，类 com.acme.MyHelper 应如下所示：

factory bean

```
public class MyHelper {

    public static MyBean createMyBean(boolean important, String message) {
        MyBean answer = ...
        // create and configure the bean
        return answer;
    }
}
```



注意

您必须使工厂方法 公开静态。

5.6. 使用脚本语言创建 BEAN

如果您有高级用例，您可以内联脚本语言，如 groovy、java、Javascript 等，以创建 bean。

通过脚本脚本，您可以更灵活，并使用一些编程来创建和配置 bean：

脚本脚本

```
<bean name="myBean" type="com.acme.MyBean" scriptLanguage="groovy">
    <script>
  // some groovy script here to create the bean
  bean = ...
  ...
  return bean
    </script>
</bean>
```

注意

使用脚本时，不使用构造器、工厂 Bean 和 factory 方法。

5.7. 在 BEAN 上使用 INIT 和 DESTROY 方法

如果您需要在使用 bean 前进行初始化和清理工作，您可以使用 initMethod 和 destroyMethod （根据 Camel 触发）。

这些方法必须是 公共 void，且没有参数，如下所示：

初始化和清理方法

```
public class MyBean {

    public void initMe() {
        // do init work here
    }

    public void destroyMe() {
        // do cleanup work here
    }

}
```

您还必须在 XML DSL 中声明这些方法，如下所示：

初始化和清理 XML

```
<bean name="myBean" type="com.acme.MyBean"
        initMethod="initMe" destroyMethod="destroyMe">
      <constructors>
        <constructor index="0" value="true"/>
        <constructor index="1" value="Hello World"/>
      </constructors>
</bean>
```

initMethod 和 destroyMethod 都是可选的，因此 bean 不必同时具有这两个功能。

5.8. 同一 XML IO DSL 文件中的 REST 和路由

您可以在同一 DSL 文件中同时具有 REST 和路由：

同一 XML IO DSL 文件中的 REST 和路由

```
<camel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
           http://camel.apache.org/schema/spring
           https://camel.apache.org/schema/spring/camel-spring.xsd">
  <rest id="rest">
      <post id="post" path="start">
          <to uri="direct:start"/>
      </post>
   </rest>

   <route>
      <from uri="direct:start"/>
      <to uri="amqp:queue:Test.Broker.StreamMessage?
jmsMessageType=Stream&amp;disableReplyTo=true"/>
   </route>
</camel>
```