Red Hat Fuse 7.11

Apache Camel 开发指南

使用 Apache Camel 开发应用程序

Last Updated: 2023-09-17

Red Hat Fuse 7.11 Apache Camel 开发指南

使用 Apache Camel 开发应用程序

法律通告

摘要

本指南介绍了如何使用 Apache Camel 开发 JBoss Fuse 应用程序。它涵盖了基本的构建块、企业级集成模式、用于路由表达式和谓词语言的基本语法、使用 Apache CXF 组件创建 Web 服务、使用 Apache Camel API 创建网络服务，以及如何创建包含任何 Java API 的 Camel 组件。

目录

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。详情请查看我们的 CTO Chris Wright 信息。

部分 I. 实施企业级集成模式

这部分论述了如何使用 Apache Camel 构建路由。它涵盖了基本构建块和 EIP 组件。

第 1 章 为路由定义构建块

摘要

Apache Camel 支持两个替代 **域特定语言** (DSL)来定义路由：Java DSL 和 Spring XML DSL。定义路由的基本构建块是 *端点* 和 *处理器*，其中处理器的行为通常由表达式或逻辑 *predicates* 修改。*Apache Camel 可让您使用各种不同语言定义表达式和 predicates。*

1.1. 实施 ROUTEBUILDER 类

概述

*要使用 域特定语言 (DSL)，您可以扩展 **RouteBuilder** 类并覆盖其 **configure（）** 方法（您定义路由规则）。*

*您可以根据需要定义多个 **RouteBuilder** 类。每个类都会实例化一次，并在 **CamelContext** 对象中注册。通常，每个 **RouteBuilder** 对象的生命周期由您在其中部署路由器的容器自动管理。*

RouteBuilder 类

*作为路由器开发人员，您的核心任务是实施一个或多个 **RouteBuilder** 类。您可以从继承两种替代 **RouteBuilder** 类：*

- ***org.apache.camel.builder.RouteBuilder** 时间为适用于 **部署到任何** 容器类型的通用 **RouteBuilder** 基础类。它在 **camel-core** 工件中提供。*
- ***org.apache.camel.spring.SpringRouteBuilder**abrt-abrt这个基础类特别适用于 Spring 容器。特别是，它提供对以下 Spring 特定功能的额外支持：在 Spring registry 中查找 **BeanRef（）** Java DSL 命令，以及事务（请参阅 **交易指南** ）。它在 **camel-spring** 工件中提供。*

***RouteBuilder** 类定义用于启动路由规则的方法（例如，**来自（）**、**拦截器 （）** 和 **exception（）**）。*

实施 RouteBuilder

*例 1.1 “RouteBuilder 类的实现” 显示最小的 **RouteBuilder** 实施。**configure（）** 方法正文包含路由规则；每个规则都是单个 Java 语句。*

> ***例 1.1. RouteBuilder 类的实现***
>
> ```
> import org.apache.camel.builder.RouteBuilder;
>
> public class MyRouteBuilder extends RouteBuilder {
>
> public void configure() {
>   // Define routing rules here:
>   from("file:src/data?noop=true").to("file:target/messages");
>
>   // More rules can be included, in you like.
>   // ...
> }
> }
> ```

来自URL1.to(URL2) 的规则形式指示路由器从目录 **src/data** 读取文件，并将它们发送到目录 **目标/消息**。选项 **?noop=true** 指示路由器保留（而不是删除） **src/data** 目录中的源文件。

注意

当您将 **contextScan** 与 Spring 或 Blueprint 搭配使用以过滤 **RouteBuilder** 类时，默认情况下，Apache Camel 将查找 singleton Bean。但是，您可以打开旧行为，使其包含使用新选项 **includeNonSingletons** 的模型范围。

1.2. 基本 JAVA DSL 语法

什么是 DSL？

域特定语言(DSL)是专为特殊用途而设计的微型语言。DSL 不需要逻辑上完成，但需要足够的表达力来描述所选域中的问题。通常，DSL **不需要** 专用的解析程序、解释器或编译器。DSL 可以在现有面向对象的主机语言之上 piggyback，提供 DSL 构造在主机语言 API 中彻底构造。

在假设 DSL 中请考虑以下命令序列：

```
command01;
command02;
command03;
```

您可以将这些命令映射到 Java 方法调用，如下所示：

```
command01().command02().command03()
```

您甚至可将块映射到 Java 方法调用。例如：

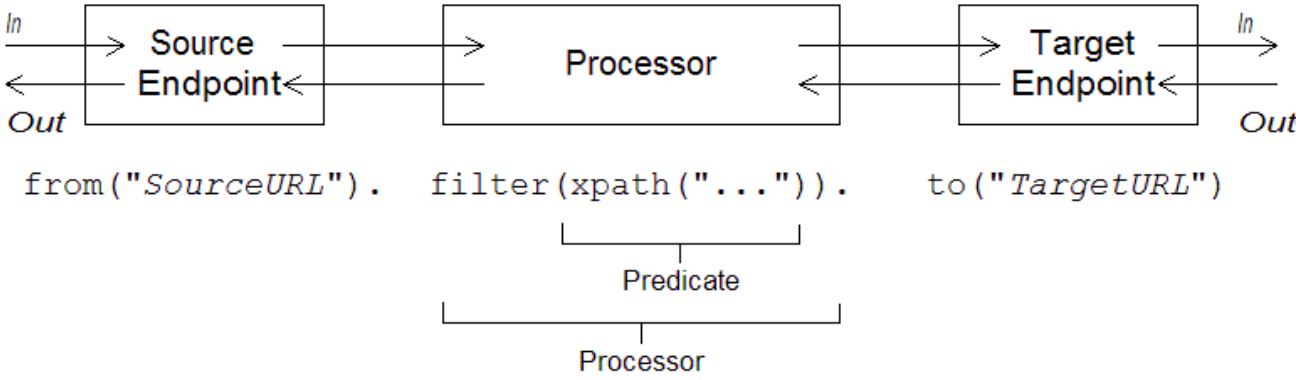
```
command01().startBlock().command02().command03().endBlock()
```

DSL 语法由主机语言 API 的数据类型隐式定义。例如，Java 方法的返回类型决定了您可以法律地调用哪些方法（与 DSL 中的下一个命令相同）。

路由器规则语法

Apache Camel 定义了一个 路由器 DSL，用于定义路由规则。您可以使用此 DSL 在 **RouteBuilder.configure（）** 实施的正文中定义规则。图 1.1 “本地路由规则” 展示了定义本地路由规则的基本语法概述。

图 1.1. 本地路由规则



本地规则始终以 **("EndpointURL")** 方法开头，该方法指定路由规则的消息（使用者端点）来源。然后您可以在规则中添加任意较长的处理器链（如 **filter（）**）。您通常通过 **to("EndpointURL")** 方法关闭规则，它指定通过规则传递的消息的目标（制作端点）。但是，始终不需要以 **to（）** 结束规则。在规则中指定消息目标的方法有其他方法。

注意

您还可以通过使用特殊处理器类型（如 intercept（）、**exception（）** 或 **errorHandler（）**）启动规则来定义全局路由规则。全局规则不在本指南的讨论范围内。

消费者和制作者

本地规则始终通过定义消费者端点（使用 **("EndpointURL")**）开始，并且通常（但并不总是）通过定义制作者端点（使用 **to("EndpointURL")**）来结束。端点 URL EndpointURL 可以使用部署时配置的任何组件。例如，您可以使用文件端点、**file:MyMessageDirectory**、Apache CXF 端点、**cxf:MyServiceName** 或 Apache ActiveMQ 端点 **activemq:queue:MyQName**。有关组件类型的完整列表，请参阅 Apache Camel 组件参考。

交换

Exchange 对象由元数据增强的消息组成。交换是 Apache Camel 中的集中重要性，因为交换是通过路由规则传播消息的标准表单。交换的主要特征是，如下所示：

- **在** message 这个过程中，使用交换封装的当前消息。在通过路由进行交换过程中，可能会修改此消息。因此，在路由开始时的 **In** 消息通常和路由末尾的 **In** 消息 **不同**。**org.apache.camel.Message** 类型提供了消息的通用模型，包括以下部分：
  - 正文.
  - 标头.
  - 附件.

  务必要意识到这是消息 **的通用** 模型。Apache Camel 支持各种协议和端点类型。因此，**无法** 标准化邮件正文或邮件标题的格式。例如，JMS 消息正文与 HTTP 消息正文或 Web 服务消息的正文具有完全不同的格式。因此，正文和标头被声明为 **对象类型**。然后，正文和标头的原始内容由创建交换实例的端点决定（即，端点会出现在 **from（）** 命令中）。

- **out** messageourier- burdenis a temporary holding area for a reply message or a transformed message.某些处理节点（特别是 **to（）** 命令）可以将 **In** 消息视为请求，将其发送到制作者端点，然后从该端点接收回复来修改当前消息。然后将回复消息插入到交换中的 **Out** message 插槽。
  通常，如果当前节点设置了 **Out** 消息，Apache Camel 会在将交换传递到路由中的下一个节点之前修改交换：旧的 **In** 消息被丢弃，并将 **Out** 消息移到 **In** message slot。因此，回复会成为新的当前消息。如需了解 Apache Camel 如何在路由中连接节点的详情，请参考 第 2.1 节 "Pipeline 处理"。

  然而，有一个特殊情形：**Out** 消息的处理方式有所不同。如果路由开头的消费者端点预期回复消息，则路由末尾的 **Out** 消息将被视为消费者端点的回复消息（在这种情况下，最终节点 **必须创建 Out** 消息，或者消费者端点会挂起）。

- 消息交换模式(MEP)admission-IFL-ffects 在路由中交换和端点之间的交互，如下所示：
  - **消费者端点** the consumer 端点创建原始交换，设置 MEP 的初始值。初始值表示消费者端点是否期望收到回复（例如，**InOut** MEP）是否不是（例如 **InOnly** MEP）。

- **生产者端点** 时间为 MEP 影响生产者端点，该端点会影响路由所遇到的生产者端点（例如，交换通过 **to（）** 节点时）。例如，如果当前 MEP 是 **InOnly**，则 **to（）** 节点不应该从端点接收回复。有时您需要更改当前的 MEP，以便自定义与制作者端点的交互。如需了解更多详细信息，请参阅 第 1.4 节 "Endpoints"。

- 交换当前消息元数据的命名属性列表。

消息交换模式

使用 **Exchange** 对象可轻松地将消息处理规范化为 不同的消息交换模式。例如，异步协议可以定义一个 MEP，它由一个消息组成，消息从消费者端点流到制作者端点（**仅适用** MEP）。另一方面，RPC 协议可能会定义由请求消息和回复消息（一个 **InOut** MEP）组成的 MEP。目前，Apache Camel 支持以下 MEPs：

- ***InOnly***
- ***RobustInOnly***
- ***InOut***
- ***InoptionalOut***
- ***OutOnly***
- ***RobustOutOnly***
- ***OutIn***
- ***OutOptionalIn***

其中，这些消息交换模式由枚举类型中的常数来表示，即 **org.apache.camel.ExchangePattern**。

分组交换

有时，有一个封装多个交换实例的单一交换很有用。为了实现此目的，您可以使用一个 分组的交换。分组交换基本上是一个交换实例，其中包含存储在 **Exchange.GROUPED_EXCHANGE** 交换属性中的 **java.util.List** 的 Exchange 对象。有关如何使用分组交换的示例，请参阅 第 8.5 节 "聚合器"。

处理器

处理器 是路由中的节点，可访问和修改通过路由进行的交换流。处理器可以使用 表达式或 predicate 参数，用于修改其行为。例如，图 1.1 "本地路由规则" 中显示的规则包含一个 **filter（）** 处理器，该处理器使用 **xpath（）** predicate 作为其参数。

表达式和 predicates

表达式（评估为字符串或其他数据类型）和 predicates（显示为 true 或 false）经常作为内置处理器类型的参数。例如，以下过滤器规则会传播 **In** 消息，只有在 **foo** 标头等于值 **bar** 时：

```
from("seda:a").filter(header("foo").isEqualTo("bar")).to("seda:b");
```

如果过滤器由 predicate、**header("foo").isEqualTo("bar")** 授权。要根据消息内容构建更加复杂的 predicates 和表达式，您可以使用以下一种表达式和谓词语言（请参阅 第 II 部分 "路由表达式和指定语言"）。

1.3. SPRING XML 文件中的路由器架构

命名空间

路由器 schema>_<- the definition XML DSL-abrtbelongs 到以下 XML 模式命名空间：

> http://camel.apache.org/schema/spring

指定 schema 位置

路由器模式的位置通常指定为 **http://camel.apache.org/schema/spring/camel-spring.xsd**，它引用 Apache Web 站点上 schema 的最新版本。例如，Apache Camel Spring 文件的 root **Bean** 元素通常配置为 *例 1.2 "指定路由器架构位置"*。

> ***例 1.2. 指定路由器架构位置***
>
> ```
> <beans xmlns="http://www.springframework.org/schema/beans"
>       xmlns:camel="http://camel.apache.org/schema/spring"
>       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-2.0.xsd
>       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
> spring.xsd">
>
>   <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
>     <!-- Define your routing rules here -->
>   </camelContext>
> </beans>
> ```

运行时模式位置

在运行时，Apache Camel **不会从** Spring 文件中指定的架构位置下载路由器模式。相反，Apache Camel 会自动从 **camel-spring** JAR 文件的根目录获取 schema 的副本。这样可确保用于解析 Spring 文件的 schema 版本始终与当前的运行时版本匹配。这很重要，因为 Apache 网站上发布的模式的最新版本可能与当前使用的运行时版本不匹配。

使用 XML 编辑器

通常，建议您使用全功能 XML 编辑器编辑 Spring 文件。借助 XML 编辑器的自动完成功能，编写符合路由器架构的 XML 更轻松，如果 XML 不正确，编辑器也会立即提醒您。

XML 编辑器 **通常** 依赖从您在 **xsi:schemaLocation** 属性中指定的位置下载 schema。为了确保您使用正确的 schema 版本，最好选择 **camel-spring.xsd** 文件的特定版本。例如，若要为 Apache Camel 的 2.3 版本编辑 Spring 文件，您可以修改 Bean 元素，如下所示：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:camel="http://camel.apache.org/schema/spring"
      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-2.0.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring-
2.3.0.xsd">
...
```

编辑完成后，请更改默认值(**camel-spring.xsd**)。要查看当前可用于下载的模式版本，请导航至网页 http://camel.apache.org/schema/spring。

1.4. ENDPOINTS

概述

Apache Camel 端点是路由中消息的源和接收器。端点是一个非常一般的构建块：它必须满足的唯一要求是它充当消息来源（制作端点）或作为消息（消费者端点）的接收器（消费者端点）。因此，Apache Camel 中支持各种不同的端点类型，范围从协议支持端点（如 HTTP）到简单的计时器端点，如 Quartz，以固定间隔生成 dummy 消息。Apache Camel 的主要优势之一是，相对容易地添加实施新端点类型的自定义组件。

端点 URI

端点通过 端点 URI 标识，其通用表单如下：

```
scheme:contextPath[?queryOptions]
```

URI 方案 标识了协议，如 **http**，contextPath 提供由协议解释的 URI 详情。另外，大多数方案都允许您定义查询选项，而 queryOptions 则采用以下格式指定：

```
?option01=value01&option02=value02&...
```

例如，以下 HTTP URI 可以用来连接到 Google 搜索引擎页面：

```
http://www.google.com
```

以下文件 URI 可用于读取在 **C:\temp\src\data** 目录下出现的所有文件：

```
file://C:/temp/src/data
```

不是每个 方案 都代表了一个协议。有时候，方案 只需提供对有用的实用程序（如计时器）的访问。例如，以下计时器端点 URI 每秒生成交换一次（=1000 毫秒）。您可以使用它来调度路由中的活动。

```
timer://tickTock?period=1000
```

使用长 Endpoint URI

有时，端点 URI 可能会因为提供的所有配置信息而变得非常长。在 JBoss Fuse 6.2 中，您可以采用两种方法使您使用冗长的 URI 更易于管理。

配置端点 9 月

您可以单独配置端点，从路由中使用简写 ID 引用端点。

```
<camelContext ...>
```

```
  <endpoint id="foo" uri="ftp://foo@myserver">
    <property name="password" value="secret"/>
    <property name="recursive" value="true"/>
    <property name="ftpClient.dataTimeout" value="30000"/>
    <property name="ftpClient.serverLanguageCode" value="fr"/>
  </endpoint>

  <route>
    <from uri="ref:foo"/>
    ...
  </route>
</camelContext>
```

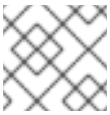
您还可以在 URI 中配置一些选项，然后使用 **property** 属性指定附加选项（或覆盖 URI 中的选项）。

```
<endpoint id="foo" uri="ftp://foo@myserver?recursive=true">
  <property name="password" value="secret"/>
  <property name="ftpClient.dataTimeout" value="30000"/>
  <property name="ftpClient.serverLanguageCode" value="fr"/>
</endpoint>
```

跨新行的分割端点配置

您可以使用新行分割 URI 属性。

```
<route>
  <from uri="ftp://foo@myserver?password=secret&amp;
        recursive=true&amp;ftpClient.dataTimeout=30000&amp;
        ftpClientConfig.serverLanguageCode=fr"/>
  <to uri="bean:doSomething"/>
</route>
```



注意

您可以在每行中指定一个或多个选项，每个选项由 **&** 分隔。

在 URI 中指定时间段

许多 Apache Camel 组件都有选项，其值是时间段（例如，指定超时值等）。默认情况下，此类时间周期选项通常指定为纯数字，被解释为毫秒的时间段。但是，Apache Camel 还支持更易读的语法，它可让您用小时、分钟和秒来表达周期。正式，人类可读的时间段是一个符合以下语法的字符串：

```
[NHour(h|hour)][NMin(m|minute)][NSec(s|second)]
```

在方括号 **[]** 中，每个术语都是可选的，并且符号 **(A|B)** 表示 **A** 和 **B** 是 alternatives。

例如，您可以使用 45 **分钟的时间配置计时器** 端点，如下所示：

```
from("timer:foo?period=45m")
  .to("log:foo");
```

您还可以使用小时、分钟和第二个单元的任意组合，如下所示：

```
from("timer:foo?period=1h15m")
  .to("log:foo");
from("timer:bar?period=2h30s")
  .to("log:bar");
from("timer:bar?period=3h45m58s")
  .to("log:bar");
```

在 URI 选项中指定原始值

默认情况下，您在 URI 中指定的选项值会自动进行 URI 编码。在某些情况下这是不需要的行为。例如，当设置 password 选项时，最好在 **没有 URI 编码的情况下** 传输原始字符字符串。

通过利用语法指定选项值，可以切换 URI 编码，即 **RAW(RawValue)**。例如，

```
from("SourceURI")
 .to("ftp:joe@myftpserver.com?password=RAW(se+re?t&23)&binary=true")
```

在本例中，密码值作为字面值传输，即 **se+re?t&23**。

不区分大小写的 enum 选项

某些端点 URI 选项映射到 Java **enum** constants。例如，Log 组件的 **level** 选项可以采用 **enum** 值 **INFO**、**WARN**、**ERROR** 等等。这个类型转换不区分大小写，因此可以使用以下任意一种选项来设置日志制作者端点的日志记录级别：

```
<to uri="log:foo?level=info"/>
<to uri="log:foo?level=INfo"/>
<to uri="log:foo?level=InFo"/>
```

指定 URI 资源

从 Camel 2.17，基于资源组件（如 XSLT）可以使用 **ref:** 作为前缀从 Registry 加载资源文件。

例如，**如果myvelocityscriptbean** 和 **mysimplescriptbean** 是 registry 中的两个 Bean 的 ID，您可以按照如下所示使用这些 Bean 的内容：

```
Velocity endpoint:
------------------
from("velocity:ref:myvelocityscriptbean").<rest_of_route>.

Language endpoint (for invoking a scripting language):
------------------------------------------------------
from("direct:start")
  .to("language:simple:ref:mysimplescriptbean")
 Where Camel implicitly converts the bean to a String.
```

Apache Camel 组件

每个 URI 方案都映射到 Apache Camel 组件，其中 Apache Camel 组件基本上是一个端点工厂。换句话说，若要使用特定类型的端点，您必须在运行时容器中部署对应的 Apache Camel 组件。例如，若要使用 JMS 端点，您要在容器中部署 JMS 组件。

Apache Camel 提供了大量不同的组件，可让您将应用程序与各种传输协议和第三方产品集成。例如，一些常用的组件有：File、JMS、CXF（Web 服务）、HTTP、Jetty、Direct 和 Mock。有关支持组件的完整列表，请参阅 Apache Camel 组件文档。

大多数 Apache Camel 组件都单独打包成 Camel 内核。如果您使用 Maven 构建应用程序，只需在相关组件构件上添加依赖项即可向应用程序中添加组件（及其第三方依赖项）。例如，要包含 HTTP 组件，您要将以下 Maven 依赖项添加到项目 POM 文件中：

```
<!-- Maven POM File -->
 <properties>
  <camel-version>{camelFullVersion}</camel-version>
  ...
 </properties>

 <dependencies>
  ...
  <dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-http</artifactId>
   <version>${camel-version}</version>
  </dependency>
  ...
 </dependencies>
```

以下组件内置到 Camel 内核（在 **camel 内核** 工件中），因此始终可用：

- bean
- 浏览
- dataset
- direct
- File
- Log
- 模拟
- Properties
- Ref
- SEDA
- timer
- VM

消费者端点

消费者端点 是在路由 **开始时** 出现的端点（即，在 **from（）** DSL 命令中）。换句话说，使用者端点负责在路由中发起处理：它创建一个新的交换实例（通常基于它收到或获取的一些消息），并提供线程来处理路由的其余部分中的交换。

例如，以下 JMS 使用者端点会拉取出 **支付** 队列的消息，并在路由中处理它们：

```
from("jms:queue:payments")
  .process(SomeProcessor)
  .to("TargetURI");
```

或者，在 Spring XML 中：

```
<camelContext id="CamelContextID"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:queue:payments"/>
    <process ref="someProcessorId"/>
    <to uri="TargetURI"/>
  </route>
</camelContext>
```

一些组件 **只是消费者**，它们只能被用来定义消费者端点。例如，Panartz 组件专门用于定义消费者端点。以下 Quartz 端点每秒钟生成一个事件（1000 毫秒）：

```
from("quartz://secondTimer?trigger.repeatInterval=1000")
  .process(SomeProcessor)
  .to("TargetURI");
```

如果您愿意，可以使用 **fromF（）** Java DSL 命令，将端点 URI 指定为格式的字符串。例如，要将用户名和密码替换为 FTP 端点的 URI，您可以使用 Java 编写路由，如下所示：

```
fromF("ftp:%s@fusesource.com?password=%s", username, password)
  .process(SomeProcessor)
  .to("TargetURI");
```

其中，第一个位置的 **%s** 替换为用户名字符串的值，第二次出现 **%s 替换为密码** 字符串。这**个字符串格式化机制由 String.format（） 实施，它类似于 C printf（） 函数提供的格式。**详情请查看 **java.util.Formatter。**

producer 端点

制作者端点 是在 中间 或路由 末尾 出现的端点（例如，在 to（） DSL 命令中）。换句话说，制作者端点接收现有的交换对象，并将交换内容发送到指定的端点。

例如，以下 JMS producer 端点将当前交换的内容推送到指定的 JMS 队列：

```
from("SourceURI")
  .process(SomeProcessor)
  .to("jms:queue:orderForms");
```

也可以在 Spring XML 中：

```
<camelContext id="CamelContextID" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURI"/>
    <process ref="someProcessorId"/>
    <to uri="jms:queue:orderForms"/>
  </route>
</camelContext>
```

某些组件只是制作者，只能用来定义制作者端点。例如，HTTP 端点专门用于定义制作者端点。

```
from("SourceURI")
  .process(SomeProcessor)
  .to("http://www.google.com/search?hl=en&q=camel+router");
```

如果您愿意，可以使用 toF（） Java DSL 命令，将端点 URI 指定为格式的字符串。例如，要将自定义 Google 查询替换为 HTTP URI，您可以使用 Java 编写路由，如下所示：

```
from("SourceURI")
  .process(SomeProcessor)
  .toF("http://www.google.com/search?hl=en&q=%s", myGoogleQuery);
```

如果出现 %s 被您的自定义查询字符串 myGoogleQuery 来替换。详情请查看 java.util.Formatter。

1.5. 处理器

概述

要让路由器执行比简单地将消费者端点连接到制作者端点更有趣的事情，您可以将 处理器 添加到路由中。处理器是一个命令，您可以插入到路由规则中，以执行任何通过规则流的消息处理。Apache Camel 提供各种处理器，如 表 1.1 “Apache Camel Processors” 所示。

表 1.1. Apache Camel Processors

Java DSL	XML DSL	描述
aggregate()	**aggregate**	第 8.5 节 “聚合器”：创建一个聚合器，将多个传入的交换组合到一个交换中。

Java DSL	XML DSL	描述
aop()	**AOP**	使用 Aspect Oriented 编程(AOP)在指定的子路由之前和之后工作。
bean(), beanRef()	**bean**	通过调用 Java 对象（或 bean）的方法来处理当前的交换。请参阅第 2.4 节 “bean Integration”。
choice()	**choice**	第 8.1 节 “基于内容的路由器”：使用 **when** 和 other 子句根据交换内容选择特定的子路由。
convertBodyTo()	**convertBodyTo**	将 In 消息正文转换为指定的类型。
delay()	**delay**	第 8.9 节 “Delayer”：将交换的传播延迟到路由后面的部分。
doTry()	**doTry**	创建用于处理异常的 try/catch 块，使用 **doCatch**、**最后执行**、和 **end** 子句。
end()	N/A	结束当前命令块。
enrich(),enrichRef()	**增强**	第 10.1 节 “内容增强器”：将当前交换与指定 **制作者** 端点 URI 请求的数据相结合。
filter（）	**filter**	第 8.2 节 “Message Filter”：使用 predicate 表达式来过滤传入的交换。
idempotentConsumer()	**idempotentConsumer**	第 11.8 节 “幂等的消费者”：实施策略以抑制重复的消息。
inheritErrorHandler()	**@inheritErrorHandler**	布尔选项可用于禁用特定路由节点上继承的错误处理程序（定义为 Java DSL 中的子词，以及 XML DSL 中的属性）。
inOnly()	**仅限**	将当前的交换的 MEP 设置为 **InOnly**（如果没有参数），或者将交换作为 **InOnly** 发送到指定的端点。

Java DSL	XML DSL	描述
inOut()	**inOut**	将当前的交换的 MEP 设置为 **InOut** （如果没有参数），或者将交换作为 **InOut** 发送到指定的端点。
loadBalance()	**loadBalance**	第 8.10 节 “Load Balancer”：在一组端点上实施负载平衡。
log()	**log**	将消息记录到控制台。
loop()	**loop**	第 8.16 节 “loop”：重复将每个交换的后端到路由中的后方。
markRollbackOnly()	**@markRollbackOnly**	**（事务）** 只标记当前回滚的事务（不会引发异常）。在 XML DSL 中，此选项设置为 **回滚** 元素上的布尔值属性。请参阅 Apache Karaf 事务指南。
markRollbackOnlyLast()	**@markRollbackOnlyLast**	**（事务）** 如果之前已与这个线程关联一个或多个事务，然后暂停，这个命令只标记回滚的最新事务（不会引发异常）。在 XML DSL 中，此选项设置为 **回滚** 元素上的布尔值属性。请参阅 Apache Karaf 事务指南。
marshal()	**marshal**	使用指定数据格式转换为低级或二进制格式，以准备通过特定传输协议发送。
multicast()	**multicast**	第 8.13 节 “多播”：将当前交换多播到多个目的地，其中每个目标获得自己的交换副本。
onCompletion()	**onCompletion**	定义在主路由完成后执行的子路由（以 **end（）** 结尾）。另请参阅第 2.14 节 “OnCompletion”。
onException()	**onException**	定义在 Java DSL 中以 **end（）** 结尾的子路由（在发生指定异常时执行的子路由）。通常在其自己的行中定义（不在路由中）。

Java DSL	XML DSL	描述
pipeline()	**pipeline**	第 5.4 节 “pipes 和 Filters”：将交换发送到一系列端点，其中一个端点的输出成为下一个端点的输入。另请参阅 第 2.1 节 “Pipeline 处理”。
policy()	**policy**	将策略应用到当前路由（目前仅用于事务性策略>_<），请参阅 Apache Karaf 事务指南。
pollEnrich(),pollEnrichRef()	**pollEnrich**	第 10.1 节 “内容增强器”：将当前交换与从指定 **消费者** 端点 URI 轮询的数据合并。
process(),processRef	**process**	在当前交换上执行自定义处理器。请参阅 “自定义处理器”一节 和 第 III 部分 “高级 Camel 编程”。
recipientList()	**recipientList**	第 8.3 节 “接收者列表”：将交换发送到在运行时计算的接收者列表（例如，基于标头的内容）。
removeHeader()	**removeHeader**	从交换的 In 消息中删除指定的标头。
removeHeaders()	**removeHeaders**	从交换的 In 消息中删除与指定模式匹配的标头。这个模式可以具有形式的 **prefix*** 替代方案，在这种情况下，它匹配以前缀为为 prefix the- theotherwise 的每个名称，它被解释为正则表达式。
removeProperty()	**删除Property**	从交换中删除指定的交换属性。
removeProperties()	**removeProperties**	从交换中删除与指定模式匹配的属性。将以逗号分隔的 1 个或多个字符串列表作为参数。第一个字符串是 模式（请参阅上面的 **removeHeaders（）** ）。后续字符串指定例外 - 这些属性保留。
resequence()	**重新排序**	第 8.6 节 “Resequencer”: 按指定比较主操作对传入的交换进行重新排序。支持 **批处理** 模式和 **流模式**。
rollback()	**rollback**	**（事务）** 只标记当前仅回滚的事务（默认情况下也增加一个异常）。请参阅 Apache Karaf 事务指南。

Java DSL	XML DSL	描述
routingSlip()	**routingSlip**	第 8.7 节 “路由 Slip”：根据从 slip 标头提取的端点 URI 列表，通过动态构建的 Pipeline 路由交换。
sample（）	**示例**	创建一个抽样节流，允许您从路由上的流量中提取交换示例。
setBody()	**setBody**	设置交换的 **消息正文**。
setExchangePattern()	**setExchangePattern**	将当前的交换的 MEP 设置为指定的值。请参阅 “消息交换模式”一节。
setHeader()	**setHeader**	在交换的 **In** 消息中设置指定的标头。
setOutHeader()	**setOutHeader**	在 Exchange's **Out** 消息中设置指定的标头。
setProperty()	**setProperty()**	设置指定的交换属性。
sort()	**排序**	对 **In** 消息正文的内容进行排序（其中可以指定自定义比较器）。
split（）	**split**	第 8.4 节 “Splitter”：将当前交换分成一系列交换，其中每个分割交换包含原始消息正文的片段。
stop()	**stop**	可停止路由当前交换，并将其标记为 completed。
threads()	**threads**	创建线程池，以并发处理路由中的后一部分。
throttle()	**throttle**	第 8.8 节 “Throttler”：将流率限制为指定级别（每秒更改）。
throwException()	**throwException**	引发指定的 Java 异常。
to()	**至**	将交换发送到一个或多个端点。请参阅 第 2.1 节 “Pipeline 处理”。
toF()	N/A	使用字符串格式将交换发送到端点。也就是说，端点 URI 字符串可以在 C **printf（）** 函数的样式中嵌入替换。

Java DSL	XML DSL	描述
transacted()	**Transacted**	创建包含路由后者一部分的 Spring 事务范围。请参阅 Apache Karaf 事务指南。
transform()	**转换**	第 5.6 节 “message Translator”：将 **In** 消息标头复制到 **Out** 消息标头，并将 **Out** 消息正文设置为指定的值。
unmarshal()	**unmarshal**	使用指定数据格式将 **In** 消息正文从低级或二进制格式转换为高级别格式。
validate()	**validate**	使用 predicate 表达式来测试当前消息是否有效。如果该 predicate 返回 **false**，则抛出一个 **PredicateValidationException** 异常。
wireTap()	**wireTap**	第 12.3 节 “wire Tap”：使用 **ExchangePattern.InOnly** MEP 将当前交换的副本发送到指定的线 tap URI。

一些示例处理器

要了解如何在路由中使用处理器，请查看以下示例：

- ***Choice***
- ***Filter***
- ***Throttler***
- ***Custom***

Choice

choice（） 处理器是一个条件语句，用于将传入的消息路由到替代制作者端点。每个替代制作者端点的前面为一个 when（） 方法，该方法使用一个 predicate 参数。如果 predicate 为 true，则会选择以下目标，否则处理继续进行规则中的下一个 when（） 方法。例如，以下 choice（） 处理器将传入消息定向到 Target1、Target 2 或 Target3，具体取决于 Predicate1 和 Predicate2 的值：

```
from("SourceURL")
   .choice()
      .when(Predicate1).to("Target1")
      .when(Predicate2).to("Target2")
      .otherwise().to("Target3");
```

也可以在 Spring XML 中：

```
<camelContext id="buildSimpleRouteWithChoice" xmlns="http://camel.apache.org/schema/spring">
 <route>
  <from uri="SourceURL"/>
  <choice>
   <when>
    <!-- First predicate -->
    <simple>header.foo = 'bar'</simple>
    <to uri="Target1"/>
   </when>
   <when>
    <!-- Second predicate -->
    <simple>header.foo = 'manchu'</simple>
    <to uri="Target2"/>
   </when>
   <otherwise>
    <to uri="Target3"/>
   </otherwise>
  </choice>
 </route>
</camelContext>
```

在 Java DSL 中，有一个特殊情形，您可能需要使用 endChoice（） 命令。某些标准 Apache Camel 处理器允许您使用特殊子使用指定额外参数，从而有效地打开额外的嵌套级别，这通常由 end（） 命令终止。例如，您可以将负载均衡器子指定为 loadBalance（）.round SDin（）.to("mock:foo").to(" mock:bar ").end（），它加载 balances 信息在 模拟：foo 和模拟：bar 端点之间进行负载平衡。但是，如果负载均衡器条款嵌入到所选条件中，需要使用 endChoice（） 命令终止子句，如下所示：

```
from("direct:start")
   .choice()
      .when(bodyAs(String.class).contains("Camel"))
         .loadBalance().roundRobin().to("mock:foo").to("mock:bar").endChoice()
      .otherwise()
         .to("mock:result");
```

Filter

filter（） 处理器可用于防止不间断消息到达制作者端点。它取一个 predicate 参数：如果 predicate 为 true，则允许消息交换给制作者；如果 predicate 为 false，则会阻止消息交换。例如，以下过滤器将阻止消息交换，除非传入消息包含标题为 foo，值设为 bar ：

```
from("SourceURL").filter(header("foo").isEqualTo("bar")).to("TargetURL");
```

也可以在 Spring XML 中：

```
<camelContext id="filterRoute" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <filter>
      <simple>header.foo = 'bar'</simple>
      <to uri="TargetURL"/>
    </filter>
  </route>
</camelContext>
```

Throttler

throttle（） 处理器确保制作者端点不会超载。throttler 通过限制每秒可传递的消息数量来实现。如果传入的消息超过指定率，则节流器会在缓冲区中累积过量消息，并将它们更慢地传输到制作者端点。例如，要将吞吐量率限制为每秒 100 个消息，您可以定义以下规则：

```
from("SourceURL").throttle(100).to("TargetURL");
```

也可以在 Spring XML 中：

```
<camelContext id="throttleRoute" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <throttle maximumRequestsPerPeriod="100" timePeriodMillis="1000">
      <to uri="TargetURL"/>
    </throttle>
  </route>
</camelContext>
```

自定义处理器

如果此处描述的标准处理器都不提供您需要的功能，您可以始终定义自己的自定义处理器。要创建自定义处理器，请定义一个实施 org.apache.camel.Processor 接口的类，并覆盖 process（） 方法。以下自定义处理器 MyProcessor 会从传入信息中删除名为 foo 的标头：

例 1.3. 实施自定义处理器类

```
public class MyProcessor implements org.apache.camel.Processor {
public void process(org.apache.camel.Exchange exchange) {
  inMessage = exchange.getIn();
  if (inMessage != null) {
      inMessage.removeHeader("foo");
  }
}
};
```

要将自定义处理器插入到路由器规则中，调用 **process（）** 方法，它为将处理器插入到规则中提供了通用机制。例如，以下规则调用 **例 1.3 “实施自定义处理器类”** 中定义的处理器：

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL").process(myProc).to("TargetURL");
```

第 2 章 ROUTE 构建的基本原则

摘要

Apache Camel 提供了几个处理器和组件，您可以在路由中连接在一起。本章介绍了使用提供的构建块构建路由原则的基础。

2.1. PIPELINE 处理

概述

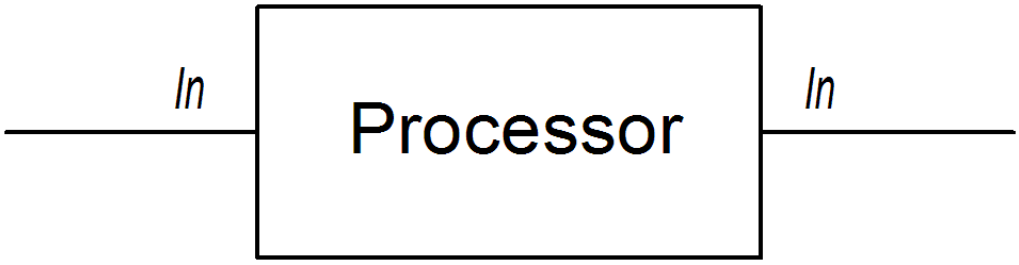
在 Apache Camel 中，pipelining 是路由定义中连接节点的主要范式。管道概念可能对 UNIX 操作系统的用户最熟悉，其中用于加入操作系统命令。例如， ls | 更多 是一个命令示例，它将目录列表传送到 page-scrolling 实用程序，更多。管道的基本概念是，一个命令的输出 将进入下一个 输入 中。如果路由的情况是，从一个处理器的 Out 消息被复制到下一处理器的 In 消息，则类似情况。

处理器节点

路由中的每个节点（除初始端点外）都是一个 处理器，它的确是从 org.apache.camel.Processor 接口继承的意义。换句话说，处理器构成 DSL 路由的基本构建块。例如： DSL 命令，如 filter（）、delayer（）、 setBody（）、 setHeader（） 和 to（） 所有代表的处理器。当考虑到处理器如何连接路由时，务必要区分两种不同的处理方法。

第一种方法是，处理器只需要修改交换的消息，如 图 2.1 “处理器修改信息” 所示。在这种情况下，交换的 Out 消息仍保持 null。

图 2.1. 处理器修改信息

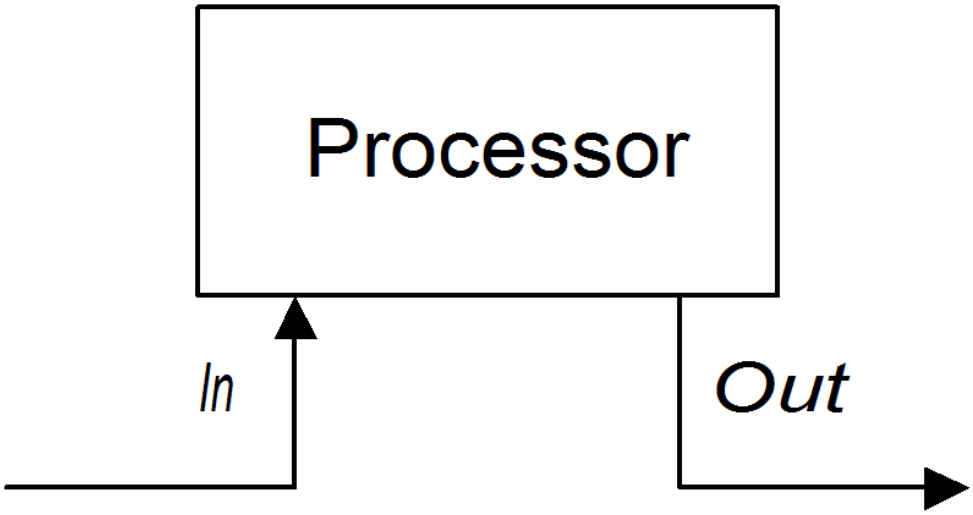


以下路由显示了一个 setHeader（） 命令，它通过添加（或修改） BillingSystem 标题来修改当前的 In 消息：

```
from("activemq:orderQueue")
    .setHeader("BillingSystem", xpath("/order/billingSystem"))
    .to("activemq:billingQueue");
```

第二种方法是，处理器创建 Out 信息来表示处理的结果，如 图 2.2 “处理器创建 Out 消息” 所示。

图 2.2. 处理器创建 Out 消息



以下路由显示了一个 transform（） 命令，该命令会 创建一个带有包含字符串 DummyBody 的消息正文：

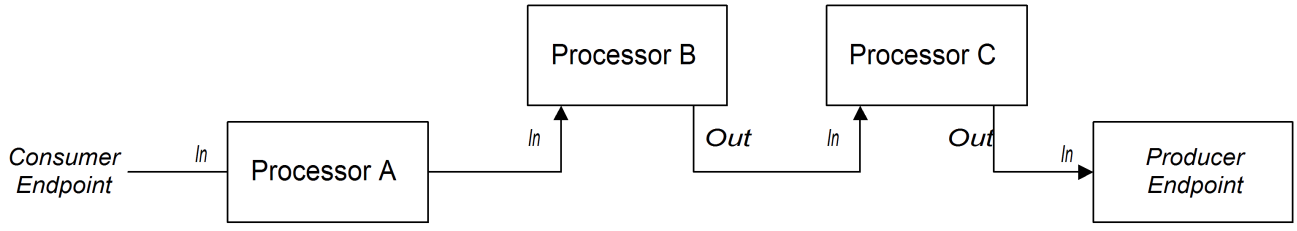
```
from("activemq:orderQueue")
    .transform(constant("DummyBody"))
    .to("activemq:billingQueue");
```

其中 常量("DummyBody") 代表一个恒定表达式。您不能直接传递字符串 DummyBody，因为对 transform（） 的参数必须是表达式类型。

用于 InOnly 交换的管道

图 2.3 “InOnly Exchanges 的 Pipeline 示例” 显示用于 InOnly 交换的处理器管道示例。处理器 A 通过修改 In 消息，而处理器 B 和 C 创建 Out 消息。路由构建器按照所示将处理器链接在一起。特别是，B 和 C 以 管道 的形式链接在一起：也就是说，处理器 B 的 Out 消息移到 In 消息，在向处理器 C 传输交换前，处理器 C 的 Out 消息移到 In 消息，然后再将处理器 C 的 Out 消息移到 In 消息中。因此，处理器的输出和输入被加入到持续管道中，如 图 2.3 “InOnly Exchanges 的 Pipeline 示例” 所示。

图 2.3. InOnly Exchanges 的 Pipeline 示例



Apache Camel 默认使用管道模式，因此您无需使用任何特殊语法在路由中创建管道。例如，以下路由从 userdataQueue 队列拉取消息，通过 Velocity 模板管道消息（以便以文本格式生成客户地址），然后将生成的文本地址发送到队列，envelopeAddresses：

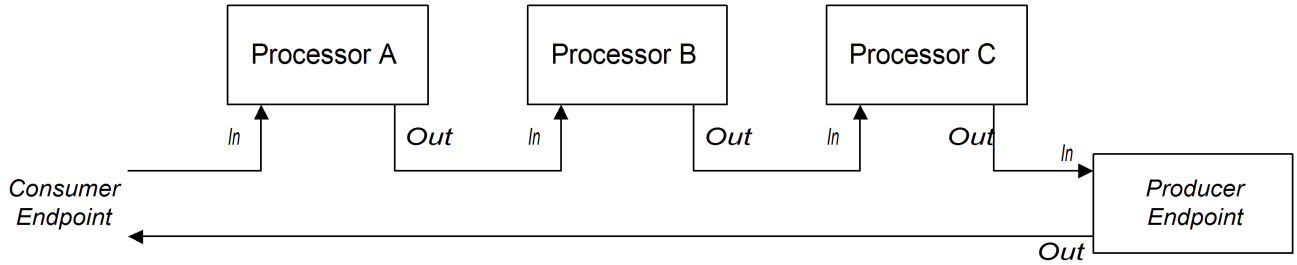
```
from("activemq:userdataQueue")
    .to(ExchangePattern.InOut, "velocity:file:AdressTemplate.vm")
    .to("activemq:envelopeAddresses");
```

在文件系统中，Velocit y:file:AddressTemplate.vm 指定 Velocity 模板文件的位置，file:AddressTemplate.vm。to（） 命令在将交换模式发送到 Velocity 端点之前，将交换模式更改为 InOut，然后将它改回到 InOnly 后。有关速率端点的详情，请参阅 Apache Camel 组件参考指南 中的 Velocity。

InOut 交换的管道

图 2.4 “InOut Exchanges 的 Pipeline 示例” 演示了 InOut 交换的处理器管道示例，您通常会用来支持远程过程调用(RPC)语义。处理器 A、B 和 C 以管道的形式链接，以及每个处理器的输出进入下一版本的输入。制作者端点生成的最终 Out 消息将将所有方式发送到消费者端点，在其中提供原始请求的回复。

图 2.4. InOut Exchanges 的 Pipeline 示例



请注意，为了支持 InOut 交换模式，重要的是，路由中最后一个节点（无论是制作者端点还是其他类型的处理器）会创建一个 Out 消息。否则，连接到消费者端点的任何客户端都会挂起并无限期等待回复消息。您应该注意，并非所有制作者端点都会创建 Out 消息。

通过处理传入的 HTTP 请求来考虑以下处理支付请求的路由：

```
from("jetty:http://localhost:8080/foo")
    .to("cxf:bean:addAccountDetails")
    .to("cxf:bean:getCreditRating")
    .to("cxf:bean:processTransaction");
```

如果传入的支付请求通过通过 Web 服务管道、cxf:bean:addAccountDetails、cxf:bean:bean:getCreditRating、cxf:bean:processTransaction 来处理。最后的 Web 服务 processTransaction 生成通过 JETTTY 端点发回的响应（Out 消息）。

当管道只由一系列端点组成时，也可以使用以下替代语法：

```
from("jetty:http://localhost:8080/foo")
    .pipeline("cxf:bean:addAccountDetails", "cxf:bean:getCreditRating",
"cxf:bean:processTransaction");
```

In OptionalOut Exchanges 的管道

In OptionalOut Exchanges 的管道基本上与 图 2.4 “InOut Exchanges 的 Pipeline 示例” 中的管道相同。In Out 和 InoptionalOut 之间的区别在于，可以使用" 可选Out 交换"模式交换作为答复。也就是说，如果是 In OptionalOut 交换，则会将 nullOut 消息复制到管道中下一节点的 In 消息。相反，如果是 In Out 交换，则会丢弃 nullOut 消息，并且当前节点中的原始 In 消息将被复制到下一节点的 In 消息中。

2.2. 多个输入

概述

标准路由仅使用 Java DSL 中的 from(EndpointURL) 语法从单一端点中获取其输入。但是，如果您需要为路由定义多个输入，该怎么办？Apache Camel 提供了一些替代方法来指定多个到路由的输入。采取的方法取决于您是否希望相互独立处理交换，还是您希望以某种方式组合使用不同输入的交换（在这种情况下，您应该使用 “内容增强器模式”一节）。

多个独立输入

指定多个输入的最简单方法是使用 from（） DSL 命令的多参数形式，例如：

```
from("URI1", "URI2", "URI3").to("DestinationUri");
```

或者您可以使用以下等效语法：

```
from("URI1").from("URI2").from("URI3").to("DestinationUri");
```

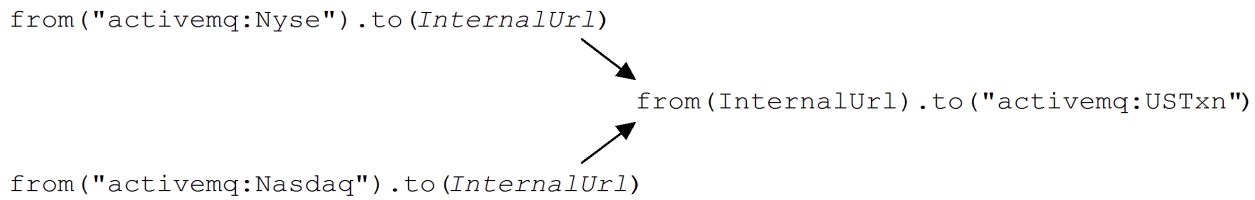
在这两个示例中，从每个输入端点、URI1、URI 2 和 URI3 中交换交换，分别相互处理，并在单独的线程中处理。实际上，您可以认为前面的路由等同于以下三个独立的路由：

```
from("URI1").to("DestinationUri");
from("URI2").to("DestinationUri");
from("URI3").to("DestinationUri");
```

分段路由

例如，您可能希望合并来自两个不同的消息传递系统的传入信息，并使用同一路由处理它们。在大多数情况下，您可以通过将路由分成片段来处理多个输入，如 图 2.5 “使用分段路由处理多个输入” 所示。

图 2.5. 使用分段路由处理多个输入



路由的初始片段从某些外部队列 3.10.0--XDP 中获取其输入，例如 activemq:Nyse 和 activemq:Nasdaq WWPN，并将传入的交换发送到内部端点 InternalUrl。第二个路由片段合并传入的交换，从内部端点中提取，并将它们发送到目标队列 activemq:USTxn。InternalUrl 是端点的 URL，仅用于在路由器应用程序中使用。以下类型的端点适合内部使用：

- 直接端点
- SEDA 端点
- 虚拟机端点

这些端点的主要目的是您可以将路由的不同片段粘滞在一起。它们都提供了将多个输入合并到一个路由的有效方法。

直接端点

直接组件提供了将路由链接最简单的机制。直接组件的事件模型是 同步的，因此后续的路由片段在与第一网段相同的线程中运行。直接 URL 的一般格式是 direct:EndpointID，其中端点 ID 为 EndpointID，它只是标识端点实例的唯一字母数字字符串。

例如，如果要从两个消息队列中使用输入 located :Nyse 和 activemq:Nasdaq，并将它们合并到一个消息队列中 Foo:USTxn，您可以通过定义以下组路由来执行此操作：

```
from("activemq:Nyse").to("direct:mergeTxns");
from("activemq:Nasdaq").to("direct:mergeTxns");
```

```
from("direct:mergeTxns").to("activemq:USTxn");
```

其中前两个路由使用来自消息队列（Nyse 和 Nasdaq）的输入，并将它们发送到端点，direct:mergeTxns。最后一个队列合并了前两个队列中的输入，并将组合消息流发送到 activemq:USTxn 队列。

直连端点的实施方式如下：每当交换到达制作者端点（例如，("direct:mergeTxns")时，直接将交换传递到具有相同端点 ID 的所有用户端点（例如，从direct:mergeTxns"））。直接端点只能用于在同一 Java 虚拟机(JVM)实例中属于同一 CamelContext 的路由之间进行通信。

SEDA 端点

SEDA 组件提供了将路由连接在一起的替代机制。您可以以类似直接组件的方式使用它，但它有不同的底层事件和线程模型，如下所示：

- 对 SEDA 端点的处理 不会 同步。也就是说，当您将交换发送到 SEDA producer 端点时，控件会立即返回到路由中的上一处理器。
- SEDA 端点包含一个队列缓冲区（包括 java.util.concurrent.BlockingQueue 类型），它将所有传入的交换存储在下一个路由网段处理之前的所有传入交换器。
- 每个 SEDA 使用者端点都会创建一个线程池（默认大小为 5），以处理来自阻塞队列的交换对象。
- SEDA 组件支持 竞争的使用者 模式，这样可保证每个传入的交换仅被处理一次，即使有多个用户附加到特定端点。

使用 SEDA 端点的一个主要优点是路由可以更快地响应，导致内置消费者线程池。库存交易示例可以重新编写为使用 SEDA 端点而不是直接端点，如下所示：

```
from("activemq:Nyse").to("seda:mergeTxns");
from("activemq:Nasdaq").to("seda:mergeTxns");

from("seda:mergeTxns").to("activemq:USTxn");
```

此示例和直接示例之间的主要区别在于，使用 SEDA 时，第二个路由片段（从 seda:mergeTxns 到 activemq:USTxn）由五个线程池处理。

注意

SEDA 不仅仅是将多个路由段过去在一起。暂存事件驱动的架构(SEDA)包含了用于构建更易管理的多线程应用程序的设计理念。Apache Camel 中的 SEDA 组件的目的是使您能够将此设计理念应用到您的应用程序。有关 SEDA 的详情，请参考 http://www.eecs.harvard.edu/~mdw/proj/seda/。

虚拟机端点

VM 组件与 SEDA 端点非常相似。唯一的不同之处在于，SEDA 组件只能将来自同一 CamelContext 中的路由片段链接在一起，VM 组件可让您将不同 Apache Camel 应用程序的路由链接在一起，只要它们在同一 Java 虚拟机中运行。

库存交易示例可以重新编写为使用 VM 端点而不是 SEDA 端点，如下所示：

```
from("activemq:Nyse").to("vm:mergeTxns");
from("activemq:Nasdaq").to("vm:mergeTxns");
```

在单独的路由器应用程序（在同一 Java 虚拟机中运行），您可以定义路由的第二个片段，如下所示：

```
from("vm:mergeTxns").to("activemq:USTxn");
```

内容增强器模式

内容增强器模式定义了处理对路由的多个输入的根本不同方式。当交换进入增强器处理器时，增强器联系一个外部资源以检索信息，然后添加到原始消息中。在此模式中，外部资源有效表示消息的第二个输入。

例如，假设您正在编写处理贡献请求的应用程序。在处理信贷请求之前，您需要用数据进行添加，从而向客户分配贡献度等级到客户，其中评分数据存储在 目录中的一个文件中，src/data/ratings。您可以使用 pollEnrich（） 模式和 GroupedExchangeAggregationStrategy 聚合策略将传入的信请求与 ratings 文件中的数据合并，如下所示：

```
from("jms:queue:creditRequests")
    .pollEnrich("file:src/data/ratings?noop=true", new GroupedExchangeAggregationStrategy())
    .bean(new MergeCreditRequestAndRatings(), "merge")
    .to("jms:queue:reformattedRequests");
```

其中 GroupedExchangeAggregationStrategy 类是一个标准的聚合策略，来自 org.apache.camel.processor.aggregate 软件包，将每个新交换添加到 java.util.List 实例，并将结果列

表存储在 Exchange.GROUPED_EXCHANGE 交换属性中。在本例中，列表中包含两个元素：原始交换（来自 creditRequests JMS 队列）；以及增强器交换（来自文件端点）。

要访问分组的交换，您可以使用类似如下的代码：

```
public class MergeCreditRequestAndRatings {
    public void merge(Exchange ex) {
        // Obtain the grouped exchange
        List<Exchange> list = ex.getProperty(Exchange.GROUPED_EXCHANGE, List.class);

        // Get the exchanges from the grouped exchange
        Exchange originalEx = list.get(0);
        Exchange ratingsEx  = list.get(1);

        // Merge the exchanges
        ...
    }
}
```

此应用的另一种方法是将合并代码直接放入自定义聚合策略类的实施中。

有关内容丰富的模式的详情，请参考 第 10.1 节 “内容增强器”。

2.3. 异常处理

摘要

Apache Camel 提供了几种不同的机制，您可以在不同的粒度级别处理异常：您可以使用 doTry、doCatch 和 最后执行 来处理路由中的异常异常，也可以使用 Exception 来指定将这个规则应用到 RouteBuilder 中的所有路由。 或者，您可以指定 所有 异常类型要执行的操作，并使用 errorHandler 将这个规则应用到 RouteBuilder 中的所有路由。

有关异常处理的详情，请参考 第 6.3 节 “死信频道”。

2.3.1. onException Clause

概述

onException 子句是一种在一个或多个路由中发生陷入异常的强大机制：它特定于类型，允许您定义处理不同异常类型的不同操作；它允许您定义使用相同（实际、稍扩展）语法作为路由作为路由的操作，您可以以处理异常的方式提供了相当的灵活性；它基于陷入模型，从而让出现异常情况的出现。

使用 onException 的 Trapping 例外

onException 子句是用于捕获异常的机制。即，当您定义 onException 子句后，它会在路由的任意时间点捕获异常。这与 Java try/catch 机制相反，只有在一个异常被发现时，才会在试块中明确包含特定的代码片段。

当您定义 onException 子句时实际上会出现什么情况，即 Apache Camel 运行时会隐式将每个路由节点包含在 try 块中。这就是为什么 onException 子句可以在路由中的任何点捕获异常。但是，这个嵌套式是自动进行的，它无法在路由定义中可见。

Java DSL 示例

在以下 Java DSL 示例中，在Exception 子句中，将应用到 RouteBuilder 类中定义的所有路由。如果在处理任何路由（从("seda:inputA")或 ("seda:inputB") ）时发生 ValidationException 异常，而 onException 子句会捕获异常并将当前交换重定向到 验证Failed JMS 队列（作为死信队列）。

```
// Java
public class MyRouteBuilder extends RouteBuilder {

  public void configure() {
    onException(ValidationException.class)
      .to("activemq:validationFailed");

    from("seda:inputA")
      .to("validation:foo/bar.xsd", "activemq:someQueue");

    from("seda:inputB").to("direct:foo")
      .to("rnc:mySchema.rnc", "activemq:anotherQueue");
  }
}
```

XML DSL 示例

前面的示例也可以在 XML DSL 中表示，使用 onException 元素来定义 exception 子句，如下所示：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:camel="http://camel.apache.org/schema/spring"
    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-2.0.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <onException>
            <exception>com.mycompany.ValidationException</exception>
            <to uri="activemq:validationFailed"/>
```

```
        </onException>
        <route>
            <from uri="seda:inputA"/>
            <to uri="validation:foo/bar.xsd"/>
            <to uri="activemq:someQueue"/>
        </route>
        <route>
            <from uri="seda:inputB"/>
            <to uri="rnc:mySchema.rnc"/>
            <to uri="activemq:anotherQueue"/>
        </route>
    </camelContext>

</beans>
```

Trapping 多例外

您可以在 Exception 子定义多个 来陷入 RouteBuilder 范围中的异常。这可让您使用不同的操作来响应不同的例外。例如，Java DSL 中定义的以下一系列 onException 子句为 ValidationException、IOException 和 Exception 定义不同的死字母目的地：

```
onException(ValidationException.class).to("activemq:validationFailed");
onException(java.io.IOException.class).to("activemq:ioExceptions");
onException(Exception.class).to("activemq:exceptions");
```

您可以在 XML DSL 中定义与 Exception 子句相同的系列，如下所示：

```
<onException>
    <exception>com.mycompany.ValidationException</exception>
    <to uri="activemq:validationFailed"/>
</onException>
<onException>
    <exception>java.io.IOException</exception>
    <to uri="activemq:ioExceptions"/>
</onException>
<onException>
    <exception>java.lang.Exception</exception>
    <to uri="activemq:exceptions"/>
</onException>
```

您也可以将多个例外分组到一起，以被同一 onException 子句捕获。在 Java DSL 中，您可以按以下方式对多个例外进行分组：

```
onException(ValidationException.class, BuesinessException.class)
  .to("activemq:validationFailed");
```

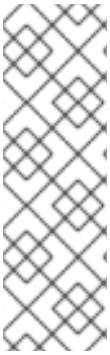
在 XML DSL 中，您可以通过在 onException 元素中定义多个 异常 元素来将多个异常元素分组在一

起，如下所示：

```
<onException>
   <exception>com.mycompany.ValidationException</exception>
   <exception>com.mycompany.BuesinessException</exception>
   <to uri="activemq:validationFailed"/>
</onException>
```

陷入多个异常时，在 Exception 子句上 的顺序非常重要。Apache Camel 最初尝试匹配对 first 子句的引发异常。如果 first 子句无法匹配，则尝试 Exception 子句上的 下一个，直到找到匹配项为止。尝试的每个匹配都受到以下算法的约束：

1. 如果抛出的异常是 链的异常 （即，一个异常已捕获并将其除以不同异常），则最嵌套的异常类型最初充当匹配的基础。这个例外被测试，如下所示：
   a. 如果 exception-to-test 具有 在Exception 子句（使用 实例测试）中指定的类型，则将触发匹配项。
   b. 如果 exception-to-test 是 onException 子句中指定的类型子类型，则会触发匹配项。
2. 如果最嵌套的例外无法产生匹配项，则通过测试链中的下一个异常。测试会继续进行链，直到触发匹配项或链用尽为止。



注意

throwException EIP 可让您从简单语言表达式创建新异常实例。您可以根据当前交换中的可用信息，使其动态。例如，

```
<throwException exceptionType="java.lang.IllegalArgumentException"
message="${body}"/>
```

Deadletter 频道

onException 使用的基本示例到目前为止都利用了 deadletter 频道 模式。也就是说，当 onException 子句捕获异常时，当前交换将路由到特殊的目的地（ deadletter 频道）。deadletter 频道充当 尚未 处理的失败消息的保存区域。管理员可以稍后检查消息，并决定需要采取什么操作。

有关 deadletter 频道模式的详情，请参阅 第 6.3 节 “死信频道”。

使用原始消息

在路由中引发异常的时间，交换中的消息可能会显著修改（而且可能被人类读取）。通常，如果死信队列中可见的消息是 原始消息，管理员更容易决定要采取的正确操作。useOriginalMessage 选项默认为 false，但如果它是在错误处理程序上配置，则会自动启用。

注意

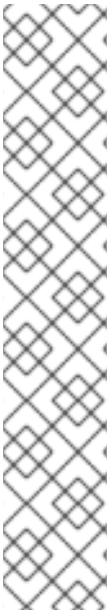
在应用到将消息发送到多个端点的 Camel 路由时，useOriginalMessage 选项可能会导致意外行为。原始消息可能不会保留在多播、Splitter 或 RecipientList 路由中，中间处理步骤会修改原始消息。

在 Java DSL 中，您可以通过原始消息替换交换中的消息。将 setAllowUseOriginalMessage（） 设置为 true，然后使用 useOriginalMessage（） DSL 命令，如下所示：

```
onException(ValidationException.class)
  .useOriginalMessage()
  .to("activemq:validationFailed");
```

在 XML DSL 中，您可以通过设置 onException 元素上的 useOriginalMessage 属性来检索原始消息，如下所示：

```
<onException useOriginalMessage="true">
    <exception>com.mycompany.ValidationException</exception>
    <to uri="activemq:validationFailed"/>
</onException>
```



注意

如果将 setAllowUseOriginalMessage（） 选项设定为 true，Camel 会在路由开始时制作原始消息的副本，这样可确保当您调用 useOriginalMessage（） 时，它会保证原始消息可用。但是，如果 Camel 上下文上的 setAllowUseOriginalMessage（） 选项被设置为 false （这是默认值），则不会 访问原始消息，您无法调用 useOriginalMessage（）。

利用默认行为的原因是在处理大型消息时优化性能。

在 2.18 之前的 Camel 版本中，allowUseOriginalMessage 的默认设置为 true。

重新传送策略

Apache Camel 允许您在出现异常时立即中断消息处理并立即放弃，Apache Camel 为您提供在发生异常情况时尝试 重 出消息的选项。在网络系统中，超时可能会出现并临时故障时，在原始异常出现后马上被处理失败的消息通常会被成功处理。

Apache Camel 重新传送支持在异常发生后对消息传输的各种策略。配置重新传送的一些最重要的选项如下：

maximumRedeliveries()

指定可尝试重新传送的次数上限（默认为 0）。负值意味着始终尝试重新传送（等同于无限值）。

retryWhile()

指定 predicate（ Predicate 类型），它决定 Apache Camel ought 是否继续重整。如果该 predicate 在当前交换上评估为 true，则会尝试重新传送；否则，将停止重新发送，且不会进行进一步重新发送尝试。

此选项优先于 maximumRedeliveries（） 选项。

在 Java DSL 中，使用 onException 子句中的 DSL 命令来指定重新传送策略选项。例如，您可以指定最大 6 个红色大小，然后该交换将发送到 验证Failed deadletter 队列，如下所示：

```
onException(ValidationException.class)
  .maximumRedeliveries(6)
  .retryAttemptedLogLevel(org.apache.camel.LogginLevel.WARN)
```

```
    .to("activemq:validationFailed");
```

在 XML DSL 中，通过在 redeliveryPolicy 元素上设置属性来指定重新传送策略选项。例如，前面的路由可以通过 XML DSL 表达，如下所示：

```
<onException useOriginalMessage="true">
    <exception>com.mycompany.ValidationException</exception>
    <redeliveryPolicy maximumRedeliveries="6"/>
    <to uri="activemq:validationFailed"/>
</onException>
```

在重新传送选项后，重新传送选项的后一部分在最后一次重新传送尝试失败后才会被处理。有关所有重新传送选项的详情，请参考 第 6.3 节 “死信频道”。

另外，您还可以在 redeliveryPolicyProfile 实例中指定重新传送策略选项。然后，您可以使用 onException 元素的 redeliverPolicyRef 属性来引用 redeliveryPolicyProfile 实例。例如，前面的路由可以表达如下：

```
<redeliveryPolicyProfile id="redelivPolicy" maximumRedeliveries="6"
retryAttemptedLogLevel="WARN"/>

<onException useOriginalMessage="true" redeliveryPolicyRef="redelivPolicy">
    <exception>com.mycompany.ValidationException</exception>
    <to uri="activemq:validationFailed"/>
</onException>
```

注意

如果您想在多个 onException 子句中重新使用同一个重新传送策略，使用 redeliveryPolicyProfile 的方法很有用。

条件捕获

通过指定 onWhen 选项，可以进行带有 onException 时出现异常捕获的问题。如果您在 onException 子句中指定 onWhen 选项，则仅在引发异常与 子句匹配时才会触发匹配项，并在 当前交换上评估为 true。

例如，在以下 Java DSL 片段中，first onException 子句触发，只有在引发n 异常与 MyUserException 匹配时，用户 标头在当前交换中是非null：

```
// Java
```

```
// Here we define onException() to catch MyUserException when
// there is a header[user] on the exchange that is not null
onException(MyUserException.class)
    .onWhen(header("user").isNotNull())
    .maximumRedeliveries(2)
    .to(ERROR_USER_QUEUE);

// Here we define onException to catch MyUserException as a kind
// of fallback when the above did not match.
// Noitce: The order how we have defined these onException is
// important as Camel will resolve in the same order as they
// have been defined
onException(MyUserException.class)
    .maximumRedeliveries(2)
    .to(ERROR_QUEUE);
```

Exception 子句的前一个可以在 XML DSL 中表达，如下所示：

```
<redeliveryPolicyProfile id="twoRedeliveries" maximumRedeliveries="2"/>

<onException redeliveryPolicyRef="twoRedeliveries">
    <exception>com.mycompany.MyUserException</exception>
    <onWhen>
        <simple>${header.user} != null</simple>
    </onWhen>
    <to uri="activemq:error_user_queue"/>
</onException>

<onException redeliveryPolicyRef="twoRedeliveries">
    <exception>com.mycompany.MyUserException</exception>
    <to uri="activemq:error_queue"/>
</onException>
```

处理异常

默认情况下，当路由中引发异常时，当前交换的处理将中断，在路由开始时将引发异常回消费者端点。触发了 **onException** 子句时，其行为基本上相同，但 **onException** 子句在引发异常前执行一些处理。

但是这种 默认行为是 处理异常的唯一方法。**onException** 提供了各种选项来修改异常处理行为，如下所示：

- 抑制异常 rethtion rethrow -abrtyou 在 **onException** 子句完成后可以阻止 **rethrown** 异常。换句话说，在这种情况下，异常 不会在 路由开始时传播到消费者端点。

- 继续处理 时间为：包含从最初发生异常的点恢复正常处理交换的选择。隐式，这种方法也会

阻止循环异常。

- 在路由开头的消费者端点时发送一个响应（即使用 InOut MEP），您可以更愿意构建自定义错误回复消息，而不是将异常请求回消费者端点。

抑制异常行

要防止当前例外被重新箭头并传播到消费者端点，您可以在 Java DSL 中将 handled（） 选项设置为 true，如下所示：

```
onException(ValidationException.class)
  .handled(true)
  .to("activemq:validationFailed");
```

在 Java DSL 中，handled（） 选项的参数可以是布尔值类型、Predicate 类型或 Expression 类型（其中任何非布尔值表达式）解释为 true，如果评估为非空值。

使用 处理 元素，可以将同一路由配置为阻止 XML DSL 中的放箭头异常，如下所示：

```
<onException>
    <exception>com.mycompany.ValidationException</exception>
    <handled>
        <constant>true</constant>
    </handled>
    <to uri="activemq:validationFailed"/>
</onException>
```

继续处理

要继续处理最初引发异常的路由中的当前消息，您可以在 Java DSL 中将持续选项设置为 true，如下所示：

```
onException(ValidationException.class)
  .continued(true);
```

在 Java DSL 中，continue（） 选项的参数可以是布尔值类型、Predicate 类型或 Expression 类型（其中任何非布尔值的表达式）解释为 true。

同一路由可以在 XML DSL 中使用 继续 元素配置，如下所示：

```
<onException>
   <exception>com.mycompany.ValidationException</exception>
   <continued>
      <constant>true</constant>
   </continued>
</onException>
```

发送响应

当启动路由的消费者端点需要回复时，您可能更愿意构建自定义错误回复消息，而不是直接让引发异常传播到消费者。在这种情况下，您需要执行两个重要步骤：使用 处理 的选项禁止放减异常；然后，使用自定义错误消息填充交换的 Out 消息。

例如，以下 Java DSL 片段演示了如何在发生 MyFunctionalException 异常时发送包含文本字符串 Sorry 的回复消息：

```
// we catch MyFunctionalException and want to mark it as handled (= no failure returned to client)
// but we want to return a fixed text response, so we transform OUT body as Sorry.
onException(MyFunctionalException.class)
   .handled(true)
   .transform().constant("Sorry");
```

如果您要向客户端发送故障响应，您通常会希望将异常消息的文本包含在响应中。您可以使用 exceptionMessage（） builder 方法访问当前异常消息的文本。例如，每当 MyFunctionalException 异常异常时，您可以发送包含异常消息文本的回复，如下所示：

```
// we catch MyFunctionalException and want to mark it as handled (= no failure returned to client)
// but we want to return a fixed text response, so we transform OUT body and return the exception
message
onException(MyFunctionalException.class)
   .handled(true)
   .transform(exceptionMessage());
```

异常消息文本也可以通过 exception.message 变量从 Simple 语言访问。例如，您可以在回复信息中嵌入当前的异常文本，如下所示：

```
// we catch MyFunctionalException and want to mark it as handled (= no failure returned to client)
// but we want to return a fixed text response, so we transform OUT body and return a nice message
// using the simple language where we want insert the exception message
onException(MyFunctionalException.class)
   .handled(true)
   .transform().simple("Error reported: ${exception.message} - cannot process this message.");
```

Exception 子句的前一个 可以在 XML DSL 中表达，如下所示：

```
<onException>
  <exception>com.mycompany.MyFunctionalException</exception>
  <handled>
    <constant>true</constant>
  </handled>
  <transform>
    <simple>Error reported: ${exception.message} - cannot process this message.</simple>
  </transform>
</onException>
```

处理异常时引发异常

在处理现有异常时（换句话说，在处理 onException 子句）的过程中会抛出异常。）会以特殊方式处理。这种异常由特殊的回退异常处理程序处理，该处理程序处理异常，如下所示：

- 所有现有的异常处理程序都会忽略，并立即处理失败。
- 新异常被记录。
- 在 exchange 对象上设置新的例外。

简单策略可避免复杂的故障场景，否则可能会以 a onException 子句结束，使锁定为死循环。

范围

onException 子句可在以下任何范围中有效：

- RouteBuilder 范围 InventoryService- onException 子句定义为 RouteBuilder.configure（） 方法内的单机语句，它们会影响该 RouteBuilder 实例中定义的所有路由。另一方面，这些 onException 子句 对任何其他 RouteBuilder 实例中定义的路由没有影响。onException 子句 必须在 路由定义之前显示。

  所有最多的示例都是使用 RouteBuilder 范围来定义的。

- 路由范围 InventoryService- onException 子句也可以直接嵌入路由内。这些 onException 子句 仅影响 定义它们的路由。

路由范围

您可以在路由定义内任何位置嵌入 onException 子句，但您必须使用 end() DSL 命令终止嵌入的Exception 子句。

例如，您可以在 Java DSL 中的 Exception 子句中定义嵌入，如下所示：

```
// Java
from("direct:start")
  .onException(OrderFailedException.class)
    .maximumRedeliveries(1)
    .handled(true)
    .beanRef("orderService", "orderFailed")
    .to("mock:error")
  .end()
  .beanRef("orderService", "handleOrder")
  .to("mock:result");
```

您可以在 XML DSL 中定义嵌入 Exception 子句，如下所示：

```
<route errorHandlerRef="deadLetter">
    <from uri="direct:start"/>
    <onException>
        <exception>com.mycompany.OrderFailedException</exception>
        <redeliveryPolicy maximumRedeliveries="1"/>
        <handled>
            <constant>true</constant>
        </handled>
        <bean ref="orderService" method="orderFailed"/>
        <to uri="mock:error"/>
    </onException>
    <bean ref="orderService" method="handleOrder"/>
    <to uri="mock:result"/>
</route>
```

2.3.2. 错误处理程序

概述

errorHandler() 子句提供类似于 onException 子句的功能，但这种机制无法区分不同的异常类型。errorHandler() 子句是 Apache Camel 提供的原始异常处理机制，在实施 onException 子句之前可用。

Java DSL 示例

errorHandler() 子句在 RouteBuilder 类中定义，并应用到该 RouteBuilder 类中的所有路由。每当在一个适用的路由中出现 任何种类的 异常时，它都会触发。例如，若要定义将所有失败交换路由到

ActiveMQ deadLetter 队列的错误处理程序，您可以按照如下所示定义 RouteBuilder ：

```
public class MyRouteBuilder extends RouteBuilder {

    public void configure() {
        errorHandler(deadLetterChannel("activemq:deadLetter"));

        // The preceding error handler applies
        // to all of the following routes:
        from("activemq:orderQueue")
          .to("pop3://fulfillment@acme.com");
        from("file:src/data?noop=true")
          .to("file:target/messages");
        // ...
    }
}
```

然而，重定向到死信频道不会发生，直到重新发送都耗尽为止。

XML DSL 示例

在 XML DSL 中，您可以使用 errorHandler 元素在 camelContext 范围内定义一个错误处理程序。例如，若要定义将所有失败交换路由到 ActiveMQ deadLetter 队列的错误处理程序，您可以按照如下所示定义 errorHandler 元素：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:camel="http://camel.apache.org/schema/spring"
    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-2.0.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <errorHandler type="DeadLetterChannel"
                      deadLetterUri="activemq:deadLetter"/>
        <route>
            <from uri="activemq:orderQueue"/>
            <to uri="pop3://fulfillment@acme.com"/>
        </route>
        <route>
            <from uri="file:src/data?noop=true"/>
            <to uri="file:target/messages"/>
        </route>
    </camelContext>

</beans>
```

错误处理器的类型

表 2.1 “错误处理程序类型” 概述您可以定义的不同类型的错误处理程序。

表 2.1. 错误处理程序类型

Java DSL Builder	XML DSL 类型属性	描述
defaultErrorHandler()	**DefaultErrorHandler**	将例外传播到调用者并支持重新传送策略，但它不支持死信队列。
deadLetterChannel()	**DeadLetterChannel**	支持与默认错误处理程序相同的功能，并且支持死信队列。
loggingErrorChannel()	**LoggingErrorChannel**	每当发生异常时，记录异常文本。
noErrorHandler()	**NoErrorHandler**	虚拟处理器实施，可用于禁用错误处理程序。
	TransactionErrorHandler	转换路由时出错处理程序。默认事务错误处理器实例会自动用于标记为转换的路由。

2.3.3. doTry、doCatch 和最后执行

概述

要在路由内处理异常，您可以使用 doTry、doCatch 和 do lastly 子句的组合，以类似 Java 的 尝试、捕获 和 最后 块处理异常。

doCatch 和 Java catch 之间的相似性

通常，路由定义中的 doCatch（） 子句的行为与 Java 代码中的 catch（） 语句类似。特别是，doCatch（） 子句支持以下功能：

- *多个 doCatch 子句 s IFL-时间可以在单个 doTry 块内拥有多个 doCatch 子句。doCatch 子句按显示的顺序进行测试，就像 Java catch（） 语句一样。Apache Camel 执行与引发 异常匹配的第一个 doCatch 子句。*

注意

这个算法与 onException 子句的匹配算法所使用的异常匹配算法不同，详情请参阅 第 2.3.1 节 “onException Clause”。

- 使用 结构（请参阅 “在 doCatch 中排放例外”一节）在 doCatch 子句中重新增加例外情况。

doCatch 的特殊特性

但是，doCatch（） 子句的一些特殊功能在 Java catch（） 语句中没有类似。以下功能特定于 doCatch（） ：

- 通过向 doCatch 子句附加 onWhen sub-clause to the doCatch 子句（请参阅 “使用 When 捕获条件异常”一节）可以有 条件捕获 异常捕获（请参阅 ）。

示例

以下示例演示了如何在 Java DSL 中写入 doTry 块，执行 doCatch（） 子句的位置，如果 IOException 异常或 IllegalStateException 异常进行引发，并且会 始终执行 doFinally（） 子句，无论是否引发异常，是否引发异常。

```
from("direct:start")
    .doTry()
        .process(new ProcessorFail())
        .to("mock:result")
    .doCatch(IOException.class, IllegalStateException.class)
        .to("mock:catch")
    .doFinally()
        .to("mock:finally")
    .end();
```

或者，在 Spring XML 中：

```
<route>
    <from uri="direct:start"/>
    <!-- here the try starts. its a try .. catch .. finally just as regular java code -->
    <doTry>
        <process ref="processorFail"/>
        <to uri="mock:result"/>
        <doCatch>
            <!-- catch multiple exceptions -->
            <exception>java.io.IOException</exception>
            <exception>java.lang.IllegalStateException</exception>
            <to uri="mock:catch"/>
        </doCatch>
        <doFinally>
            <to uri="mock:finally"/>
        </doFinally>
    </doTry>
</route>
```

-

在 doCatch 中排放例外

可以使用构造在 doCatch （） 子句中重新增加异常，如下所示：

```
from("direct:start")
   .doTry()
      .process(new ProcessorFail())
      .to("mock:result")
   .doCatch(IOException.class)
       .to("mock:io")
      // Rethrow the exception using a construct instead of handled(false) which is deprecated in a
doTry/doCatch clause.
      .throwException(new IllegalArgumentException("Forced"))
   .doCatch(Exception.class)
      // Catch all other exceptions.
      .to("mock:error")
   .end();
```

注意

您还可以使用在 doTry/doCatch 子句中弃用的处理器 (false) 来重新增加异常：

```
.process(exchange -> {throw
exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);})
```

在前面的示例中，如果 doCatch（） 可以发现 IOException，则当前交换将发送到 模拟：io 端点，然后重新进行 IOException。这会在路由开始时（在 from（） 命令中）为使用者端点提供处理异常的机会。

以下示例演示了如何在 Spring XML 中定义相同的路由：

```
<route>
   <from uri="direct:start"/>
   <doTry>
      <process ref="processorFail"/>
      <to uri="mock:result"/>
      <doCatch>
         <to uri="mock:io"/>
         <throwException message="Forced" exceptionType="java.lang.IllegalArgumentException"/>
      </doCatch>
      <doCatch>
         <!-- Catch all other exceptions. -->
         <exception>java.lang.Exception</exception>
         <to uri="mock:error"/>
```

```
        </doCatch>
    </doTry>
</route>
```

使用 When 捕获条件异常

Apache Camel doCatch（） 子句的一个特殊功能是，您可以根据运行时评估的表达式来对异常的捕获进行条件化。换而言之，如果您使用表的条款捕获一个异常，则 doCatch(ExceptionList).doWhen(Expression).doWhen(Expression)只会被发现，如果 predicate 表达式、表达式、在运行时评估为 true。

例如，以下 doTry 块将捕获异常、IOException 和 IllegalStateException，仅当异常消息包含单词时，Sever e:

```
from("direct:start")
    .doTry()
        .process(new ProcessorFail())
        .to("mock:result")
    .doCatch(IOException.class, IllegalStateException.class)
        .onWhen(exceptionMessage().contains("Severe"))
        .to("mock:catch")
    .doCatch(CamelExchangeException.class)
        .to("mock:catchCamel")
    .doFinally()
        .to("mock:finally")
    .end();
```

或者，在 Spring XML 中：

```
<route>
    <from uri="direct:start"/>
    <doTry>
        <process ref="processorFail"/>
        <to uri="mock:result"/>
        <doCatch>
            <exception>java.io.IOException</exception>
            <exception>java.lang.IllegalStateException</exception>
            <onWhen>
                <simple>${exception.message} contains 'Severe'</simple>
            </onWhen>
            <to uri="mock:catch"/>
        </doCatch>
        <doCatch>
            <exception>org.apache.camel.CamelExchangeException</exception>
            <to uri="mock:catchCamel"/>
        </doCatch>
        <doFinally>
            <to uri="mock:finally"/>
```

```
        </doFinally>
    </doTry>
</route>
```

嵌套条件

可以选择将 Camel 异常处理添加到 JavaDSL 路由。dotry（） 创建尝试或捕获块来处理异常，对于特定于路由的错误处理非常有用。

如果要捕获 ChoiceDefinition 内的异常，您可以使用以下 doTry 块：

```
from("direct:wayne-get-token").setExchangePattern(ExchangePattern.InOut)
        .doTry()
          .to("https4://wayne-token-service")
          .choice()
             .when().simple("${header.CamelHttpResponseCode} == '200'")
               .convertBodyTo(String.class)
.setHeader("wayne-token").groovy("body.replaceAll('\"','')")
               .log(">> Wayne Token : ${header.wayne-token}")
           .endChoice()

.doCatch(java.lang.Class (java.lang.Exception>)
          .log(">> Exception")
        .endDoTry();

from("direct:wayne-get-token").setExchangePattern(ExchangePattern.InOut)
        .doTry()
          .to("https4://wayne-token-service")
        .doCatch(Exception.class)
          .log(">> Exception")
        .endDoTry();
```

2.3.4. 传播 SOAP Exception

概述

Camel CXF 组件提供了与 Apache CXF 集成，可让您从 Apache Camel 端点发送和接收 SOAP 消息。您可以在 XML 中轻松定义 Apache Camel 端点，然后可使用端点的 bean ID 在路由中引用。如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 CXF。

如何传播堆栈追踪信息

可以配置 CXF 端点，以便在服务器端抛出 Java 异常时，异常的堆栈追踪会被放入故障消息并返回到客户端。要启用这个 feaure，请将 dataFormat 设置为 PAYLOAD，并在 cxfEndpoint 元素中将 faultStackTraceEnabled 属性设置为 true，如下所示：

-

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable sending the stack trace back to client; the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" />
    <entry key="dataFormat" value="PAYLOAD" />
  </cxf:properties>
</cxf:cxfEndpoint>
```

出于安全考虑，堆栈追踪不包括造成异常（即 后面的堆栈跟踪部分）。如果要在堆栈追踪中包含造成异常，请在 **cxfEndpoint** 元素中将 **exceptionMessageCauseEnabled** 属性设置为 **true**，如下所示：

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable to show the cause exception message and the default value is false -->
    <entry key="exceptionMessageCauseEnabled" value="true" />
    <!-- enable to send the stack trace back to client,  the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" />
    <entry key="dataFormat" value="PAYLOAD" />
  </cxf:properties>
</cxf:cxfEndpoint>
```

警告

您应该只启用 **exceptionMessageCauseEnabled** 标记，用于测试和诊断。服务器通常会区分原始异常原因，从而使恶意用户更难以探测服务器。

2.4. BEAN INTEGRATION

概述

Bean 集成提供了一种通用机制，用于使用任意 Java 对象处理消息。通过将 bean 引用插入到路由中，您可以在 Java 对象上调用任意方法，然后访问和修改传入的交换。将交换内容映射到参数并返回 bean 方法值的机制称为 参数绑定。参数绑定可以使用以下任一方法组合来初始化方法的参数：

-

传统的方法签名 >_<- the method 签名是否符合某些约定，则参数绑定可以使用 Java 反射来确定要通过哪些参数。

- 注解和依赖项注入 GCM- the 以获得更灵活的绑定机制，使用 Java 注解指定要注入方法的参数。这种依赖项注入机制依赖于 Spring 2.5 组件扫描。通常，如果您将 Apache Camel 应用程序部署到 Spring 容器中，则依赖项注入机制将自动工作。

- 在调用 bean 的点上，明确指定参数 InventoryService-latex，可显式指定 参数（可以是恒定常数或使用 Simple 语言）。

bean registry

Bean 是通过 bean registry 进行访问的，该服务可让您将类名称或 bean ID 用作密钥来查找 Bean。您在 bean registry 中创建条目的方式取决于底层的 framework>_<-abrt，如纯 Java、Spring、Spring、Gusice 或 Blueprint。registry 条目通常隐式创建（例如，当您在 Spring XML 文件中实例化 Spring bean 时）。

registry 插件策略

Apache Camel 为 bean 注册表实施插件策略，定义用于访问 Bean 的集成层，从而使底层 registry 的实施透明。因此，可以将 Apache Camel 应用程序与各种不同 Bean registry 集成，如 表 2.2 “registry 插件” 所示。

表 2.2. registry 插件

registry 的实现	带有 Registry 插件的 Camel 组件
Spring bean registry	**Camel-spring**
Guice bean registry	**camel-guice**
蓝图 bean registry	**Camel-blueprint**
OSGi 服务 registry	在 **OSGi 容器中部署**
JNDI registry	

通常，您不必担心配置 bean registry，因为会自动为您安装相关的 bean registry。例如，如果您使用 Spring 框架定义路由，则 Spring ApplicationContextRegistry 插件会在当前的 CamelContext 实例中自动安装。

在 OSGi 容器中部署是一个特殊情况。当将 Apache Camel 路由部署到 OSGi 容器时，CamelContext 会自动设置 registry 链以解析 Bean 实例：registry 链包括 OSGi 注册表，后面是 Blueprint（或 Spring） registry。

在 Java 中访问创建的 Bean

要使用 Java bean（一个普通旧的 Java 对象或 OVA）进程交换对象，请使用 bean（） 处理器，将入站交换绑定到 Java 对象上的方法。例如，若要使用类 MyBeanProcessor 处理入站交换，请定义如下路由：

```
from("file:data/inbound")
    .bean(MyBeanProcessor.class, "processBody")
    .to("file:data/outbound");
```

其中 bean（） 处理器创建 MyBeanProcessor 类型的实例，并调用 processBody（） 方法来处理入站交换。如果您只想从一个路由访问 MyBeanProcessor 实例，则此方法就足够了。但是，如果您要从多个路由访问相同的 MyBeanProcessor 实例，请使用将对象类型用作其第一个参数的 bean（） 变体。例如：

```
MyBeanProcessor myBean = new MyBeanProcessor();

from("file:data/inbound")
    .bean(myBean, "processBody")
    .to("file:data/outbound");
from("activemq:inboundData")
    .bean(myBean, "processBody")
    .to("activemq:outboundData");
```

访问过载的 bean 方法

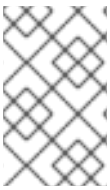
如果 bean 定义了过载方法，您可以通过指定方法名称及其参数类型来选择调用的超载方法。例如，如果 MyBeanBrocessor 类有两个超载方法： processBody(String) 和 processBody(String,String)，您可以按照如下所示调用后面的超载方法：

```
from("file:data/inbound")
  .bean(MyBeanProcessor.class, "processBody(String,String)")
  .to("file:data/outbound");
```

或者，如果您想要使用的参数数量来识别方法，而不是明确指定每个参数的类型，您可以使用通配符字符 *。例如，要调用名为 processBody 的方法，它采用两个参数(irrespective of)参数，调用 bean（） 处理器，如下所示：

```
from("file:data/inbound")
.bean(MyBeanProcessor.class, "processBody(*,*)")
.to("file:data/outbound");
```

在指定方法时，您可以使用简单的非限定类型 name- 例如，processBody(Exchange)- 或完全限定域名（如 processBody(org.apache.camel.Exchange)。



注意

在当前实现中，指定的类型名称必须是与参数类型完全匹配。类型继承不会考虑。

明确指定参数

当调用 bean 方法时，可以明确指定参数值。可以传递以下简单类型值：

- 布尔值：true 或 false。
- 数字：123、7 等。
- 字符串：'In single quotes' 或 "In double quotes"。
- null 对象：null.

以下示例演示了如何将显式参数值与同一方法调用的类型组合：

```
from("file:data/inbound")
  .bean(MyBeanProcessor.class, "processBody(String, 'Sample string value', true, 7)")
  .to("file:data/outbound");
```

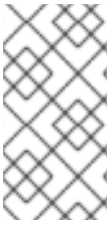
在前面的示例中，第一个参数的值可能会假定由参数绑定注解决定（请参阅 “基本注解”一节）。

除了简单类型值外，您还可以使用 Simple 语言(第 30 章 简单语言)指定参数值。这意味着在指定 参数值时提供了 Simple 语言的完整功能。例如，将邮件正文和标题 标题 的值传递给 bean 方法：

```
from("file:data/inbound")
  .bean(MyBeanProcessor.class, "processBodyAndHeader(${body},${header.title})")
  .to("file:data/outbound");
```

您还可以将整个标头散列映射作为参数传递。例如，以下示例中必须声明第二个方法参数，才能作为 **java.util.Map** 类型：

```
from("file:data/inbound")
  .bean(MyBeanProcessor.class, "processBodyAndAllHeaders(${body},${header})")
  .to("file:data/outbound");
```



注意

从 **Apache Camel 2.19** 发行版本，从 **bean** 方法调用返回 **null** 现在始终确保消息正文已设置为 **null** 值。

基本方法签名

要将交换绑定到 **bean** 方法，您可以定义符合特定约定的方法签名。特别是，方法签名有两个基本惯例：

- 处理消息正文的方法签名
- 处理交换的方法签名

处理消息正文的方法签名

如果要实施访问或修改传入的消息正文的方法，您必须定义一个使用单个 **字符串** 参数的方法签名，并返回 **String** 值。例如：

```
// Java
package com.acme;

public class MyBeanProcessor {
    public String processBody(String body) {
        // Do whatever you like to 'body'...
        return newBody;
    }
}
```

处理交换的方法签名

为获得更大的灵活性，您可以实施访问传入交换的 bean 方法。这可让您访问或修改所有标头、正文和交换属性。对于处理交换，方法签名使用一个 org.apache.camel.Exchange 参数，并返回 void。例如：

```
// Java
package com.acme;

public class MyBeanProcessor {
    public void processExchange(Exchange exchange) {
        // Do whatever you like to 'exchange'...
        exchange.getIn().setBody("Here is a new message body!");
    }
}
```

从 Spring XML 访问 Springan

您可以使用 Spring XML 创建实例，而不必在 Java 中创建 bean 实例。实际上，如果您在 XML 中定义路由，这是唯一可行的方法。要在 XML 中定义 Bean，请使用标准的 Spring Bean 元素。以下示例演示了如何创建 MyBeanProcessor 的实例：

```
<beans ...>
    ...
    <bean id="myBeanId" class="com.acme.MyBeanProcessor"/>
</beans>
```

也可以使用 Spring 语法将数据传递到 bean 的构造器参数。有关如何使用 Spring bean 元素的完整详情，请参阅 Spring 参考指南 中的 IoC 容器。

使用 bean 元素创建对象实例时，您可以使用 bean 的 ID（an 元素的 id 属性的值）来引用它。例如，如果将 ID 为等于 myBeanId 的 bean 元素，您可以使用 beanRef（） 处理器来引用 Java DSL 路由中的an，如下所示：

```
from("file:data/inbound").beanRef("myBeanId", "processBody").to("file:data/outbound");
```

其中 beanRef（） 处理器在指定的 bean 实例上调用 MyBeanProcessor.processBody（） 方法。

您还可以使用 Camel 架构的 bean 元素从 Spring XML 路由内调用 bean。例如：

```
<camelContext id="CamelContextID" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file:data/inbound"/>
```

```
    <bean ref="myBeanId" method="processBody"/>
    <to uri="file:data/outbound"/>
  </route>
</camelContext>
```

为了获得有效效率，您可以将 缓存 选项设置为 **true**，这样可避免在每次使用 **bean** 时查找 **registry**。例如，要启用缓存，您可以在 **bean** 元素上设置 **cache** 属性，如下所示：

```
<bean ref="myBeanId" method="processBody" cache="true"/>
```

从 Java 访问 Springan

使用 **Spring bean** 元素创建对象实例时，您可以使用 **bean** 的 **ID**（ **bean** 元素的 **id** 属性的值）从 **Java** 引用它。例如，如果将 **ID** 为等于 **myBeanId** 的 **bean** 元素，您可以使用 **beanRef**（） 处理器来引用 **Java DSL** 路由中的an，如下所示：

```
from("file:data/inbound").beanRef("myBeanId", "processBody").to("file:data/outbound");
```

另外，您可以使用 **@BeanInject** 注释（如下所示）来引用 **Spring bean**。

```
// Java
import org.apache.camel.@BeanInject;
...
public class MyRouteBuilder extends RouteBuilder {

  @BeanInject("myBeanId")
  com.acme.MyBeanProcessor bean;

  public void configure() throws Exception {
    ..
  }
}
```

如果省略 **@BeanInject** 注释中的 **bean ID**，**Camel** 会按照类型查找 **registry**，但只有在给定类型只有一个 **bean** 时才能发挥作用。例如，查找并注入 **com.acme.MyBeanProcessor** 类型的 **Bean**：

```
@BeanInject
com.acme.MyBeanProcessor bean;
```

Spring XML 中的 bean 关闭顺序

对于 **Camel** 上下文使用的 **Bean**，正确的关闭顺序通常是：

1. 关闭 camelContext 实例，后接；
2. 关闭使用的 Bean。

如果此关闭顺序被反向，那么 Camel 上下文会尝试访问已经被破坏的 Bean（导致直接出错）；或者 Camel 上下文尝试在销毁时创建缺失的 bean，这也会导致错误。Spring XML 中的默认关闭顺序取决于 Bean 和 camelContext 中的顺序出现在 Spring XML 文件中。为了避免因为不正确的关闭顺序导致随机错误，因此 camelContext 配置为在 Spring XML 文件中的任何其他 Bean 之前 关闭。从 Apache Camel 2.13.0 开始，这是默认的行为。

如果您需要更改此行为（因此 Camel 上下文 不会在 其他 Bean 之前关闭），您可以将 camelContext 元素上的 shutdownEager 属性设为 false。在这种情况下，您可以使用 Spring dependent-on 属性对关机顺序进行更精细的控制。

参数绑定注解

"基本方法签名"一节 中描述的基本参数绑定可能并不总是方便使用。例如，如果您有一个执行一些数据操作的旧版 Java 类，您可能希望从入站交换中提取数据并将其映射到现有方法签名的参数。对于这种类型的参数绑定，Apache Camel 提供以下 Java 注解类型：

- 基本注解
- 语言注解
- 继承的注解

基本注解

表 2.3 "基本 Bean 标注" 显示 org.apache.camel Java 软件包中的注释，您可以使用它来将消息数据注入到 bean 方法的参数中。

表 2.3. 基本 Bean 标注

注解	含义	参数？
@Attachments	绑定到附加列表。	

注解	含义	参数？
@Body	绑定到入站邮件正文。	
@Header	绑定到入站邮件标题。	标头的字符串名称。
@Headers	绑定到入站邮件标头的 **java.util.Map**。	
@OutHeaders	绑定到出站邮件标头的 **java.util.Map**。	
@property	绑定到指定交换属性。	属性的字符串名称。
@properties	绑定到交换属性的 **java.util.Map**。	

例如，以下类显示如何使用基本注解将消息数据注入 processExchange（） 方法参数。

```
// Java
import org.apache.camel.*;

public class MyBeanProcessor {
    public void processExchange(
        @Header(name="user") String user,
        @Body String body,
        Exchange exchange
    ) {
        // Do whatever you like to 'exchange'...
        exchange.getIn().setBody(body + "UserName = " + user);
    }
}
```

请注意，如何将注解与默认规则混合使用。除了注入注解的参数外，参数绑定也会自动将 exchange 对象注入到 org.apache.camel.Exchange 参数。

表达式语言注解

表达式语言注释提供了一种强大的机制，用于将消息数据注入 bean 方法的参数。通过使用这些注解，您可以调用任意脚本（使用您选择的脚本语言编写）从入站交换中提取数据，并将数据注入方法参数。表 2.4 “表达式语言注释” 显示 org.apache.camel.language 软件包（和子软件包）中的注释，供您用来将消息数据注入 bean 方法的参数中。

表 2.4. 表达式语言注释

注解	描述
@Bean	注入 Bean 表达式。
@Constant	注入 Constant 表达式
@EL	注入 EL 表达式。
@Groovy	注入 Groovy 表达式。
@Header	注入标题表达式。
@JavaScript	注入 JavaScript 表达式。
@OGNL	注入 OGNL 表达式。
@PHP	注入 PHP 表达式。
@Python	注入 Python 表达式。
@Ruby	注入 Ruby 表达式。
@Simple	注入简单表达式。
@XPath	注入 XPath 表达式。
@XQuery	注入 XQuery 表达式。

例如，以下类显示如何使用 @XPath 注释从 XML 格式的传入消息正文中提取用户名和密码：

```
// Java
import org.apache.camel.language.*;

public class MyBeanProcessor {
    public void checkCredentials(
        @XPath("/credentials/username/text()") String user,
        @XPath("/credentials/password/text()") String pass
    ) {
        // Check the user/pass credentials...
        ...
    }
}
```

@Bean 注释是一个特殊情形，因为它可让您注入调用已注册 Bean 的结果。例如，要将关联 ID 注入方法参数，您可以使用 @Bean 注释来调用 ID 生成器类，如下所示：

-

```
// Java
import org.apache.camel.language.*;

public class MyBeanProcessor {
    public void processCorrelatedMsg(
        @Bean("myCorrIdGenerator") String corrId,
        @Body String body
    ) {
        // Check the user/pass credentials...
        ...
    }
}
```

其中字符串 myCorrIdGenerator 是 ID 生成器实例的 bean ID。ID 生成器类可以使用 spring bean 元素实例化，如下所示：

```
<beans ...>
    ...
    <bean id="myCorrIdGenerator" class="com.acme.MyIdGenerator"/>
</beans>
```

其中 MyIdGenerator 类可以定义如下：

```
// Java
package com.acme;

public class MyIdGenerator {

    private UserManager userManager;

    public String generate(
        @Header(name = "user") String user,
        @Body String payload
    ) throws Exception {
        User user = userManager.lookupUser(user);
        String userId = user.getPrimaryId();
        String id = userId + generateHashCodeForPayload(payload);
        return id;
    }
}
```

请注意，您也可以在 bean 类中使用注解 MyIdGenerator。generate（） 方法签名的唯一限制是，它必须返回正确的类型来注入 @Bean 注释的参数。由于 @Bean 注释不要求您指定方法名称，因此注入机制只需在引用的 bean 中调用具有匹配返回类型的方法。

注意

某些语言注释在核心组件（@Bean、@Constant、@Simple 和 @XPath）中可用。但是，对于非内核组件，您必须确定载入相关组件。例如，要使用 OGNL 脚本，您必须加载 **camel-ognl** 组件。

继承的注解

参数绑定注解可以从接口或从超级类继承。例如，如果您使用 标头 注释和 **Body** 注解定义 **Java** 接口，如下所示：

```
// Java
import org.apache.camel.*;

public interface MyBeanProcessorIntf {
    void processExchange(
        @Header(name="user") String user,
        @Body String body,
        Exchange exchange
    );
}
```

实现类 **MyBeanProcessor** 中定义的超载方法现在继承基础接口中定义的注解，如下所示：

```
// Java
import org.apache.camel.*;

public class MyBeanProcessor implements MyBeanProcessorIntf {
    public void processExchange(
        String user,  // Inherits Header annotation
        String body,  // Inherits Body annotation
        Exchange exchange
    ) {
        ...
    }
}
```

接口实现

实现 **Java** 接口的类通常 受保护的、**private** 或在 仅软件包 范围内。如果您在以这种方式限制的实施类上调用方法，则 **bean** 绑定将返回到调用对应的接口方法，该方法可以公开访问。

例如，请考虑以下公共 **BeanIntf** 接口：

```
// Java
public interface BeanIntf {
    void processBodyAndHeader(String body, String title);
}
```

如果 BeanIntf 接口由以下受保护的 BeanIntfImpl 类实施：

```
// Java
protected class BeanIntfImpl implements BeanIntf {
    void processBodyAndHeader(String body, String title) {
        ...
    }
}
```

以下 bean 调用将回退到调用公共 BeanIntf.processBodyAndHeader 方法：

```
from("file:data/inbound")
  .bean(BeanIntfImpl.class, "processBodyAndHeader(${body}, ${header.title})")
  .to("file:data/outbound");
```

调用静态方法

Bean 集成具有调用静态方法的能力，无需创建关联类的实例。例如，请考虑以下用于定义静态方法的 Java 类，请更改Something（） ：

```
// Java
...
public final class MyStaticClass {
    private MyStaticClass() {
    }

    public static String changeSomething(String s) {
        if ("Hello World".equals(s)) {
            return "Bye World";
        }
        return null;
    }

    public void doSomething() {
        // noop
    }
}
```

您可以使用 bean 集成来调用静态的 changeSomething 方法，如下所示：

```
from("direct:a")
  *.bean(MyStaticClass.class, "changeSomething")*
  .to("mock:a");
```

请注意，虽然这种语法与普通功能的调用相同，但同时也是集成漏洞的 Java 反映出方法，以识别静态方法并继续调用方法，而无需 实例化 MyStaticClass。

调用 OSGi 服务

在特殊情况下，路由部署到红帽 Fuse 容器中，可以使用 bean 集成直接调用 OSGi 服务。例如，假设 OSGi 容器中的捆绑包已导出该服务，即 org.fusesource.example.HelloWorldOsgiService，您可以使用以下 bean 集成代码调用 sayHello 方法：

```
from("file:data/inbound")
  .bean(org.fusesource.example.HelloWorldOsgiService.class, "sayHello")
  .to("file:data/outbound");
```

您还可以使用 bean 组件从 Spring 或蓝图 XML 文件中调用 OSGi 服务，如下所示：

```
<to uri="bean:org.fusesource.example.HelloWorldOsgiService?method=sayHello"/>
```

这种方法的工作方式是，当 Apache Camel 在 OSGi 容器中部署时，会设置一个 registry 链。首先，它会在 OSGi 服务 registry 中查找指定的类名称；如果这个查找失败，它将回退到本地 Spring DM 或蓝图 registry。

2.5. 创建交换实例

概述

使用 Java 代码处理消息时（例如，在 bean 类或在处理器类中），通常会需要创建新的交换实例。如果您需要创建 Exchange 对象，最简单的方法是调用 ExchangeBuilder 类的方法，如下所述。

ExchangeBuilder 类

ExchangeBuilder 类的完全限定名称如下：

```
org.apache.camel.builder.ExchangeBuilder
```

ExchangeBuilder 公开静态方法 aExchange，可用于开始构建交换对象。

示例

例如，以下代码会创建一个新的交换对象，其中包含消息正文字符串 Hello World!，以及包含用户名和密码凭证的标头：

```
// Java
import org.apache.camel.Exchange;
import org.apache.camel.builder.ExchangeBuilder;
...
Exchange exch = ExchangeBuilder.anExchange(camelCtx)
                .withBody("Hello World!")
                .withHeader("username", "jdoe")
                .withHeader("password", "pass")
                .build();
```

ExchangeBuilder 方法

ExchangeBuilder 类支持以下方法：

ExchangeBuilder aExchange(CamelContext context)

（静态方法）初始构建交换对象。

Exchange build（）

构建交换.

ExchangeBuilder withBody(Object body)

在交换上设置消息正文（即，设置交换的 In 消息正文）。

ExchangeBuilder 带有Header(String key, Object value)

在交换上设置一个标头（即，在交换的 In 消息上设置一个标头）。

ExchangeBuilder withPattern(ExchangePattern pattern)

在交换模式上设置交换模式。

ExchangeBuilder withProperty(String key, Object value)

在交换上设置属性。

2.6. 转换消息内容

摘要

Apache Camel 支持多种方法来转换消息内容。除了用于修改消息内容的简单原生 API 外，Apache Camel 支持与几个不同的第三方库和转换标准集成。

2.6.1. 简单消息转换

概述

Java DSL 具有一个内置 API，允许您在传入和传出消息上执行简单的转换。例如，例 2.1 “引入消息的简单转换” 中显示的规则会将文本 World! 附加到传入消息正文的末尾。

例 2.1. 引入消息的简单转换

```
from("SourceURL").setBody(body().append(" World!")).to("TargetURL");
```

其中 setBody（） 命令取代了传入邮件正文的内容。

用于简单转换的 API

您可以使用以下 API 类在路由器规则中执行消息内容的简单转换：

- org.apache.camel.model.ProcessorDefinition
- org.apache.camel.builder.Builder
- org.apache.camel.builder.ValueBuilder

ProcessorDefinition class

org.apache.camel.model.ProcessorDefinition 类定义 DSL 命令，您可以在 例 2.1 “引入消息的简单转换” 中直接插入路由器 rule>_<-targetNamespaces 例如： 中的 setBody（） 命令。表 2.5 “从处理器Definition 类转换方法” 显示与转换消息内容相关的 ProcessorDefinition 方法：

表 2.5. 从处理器Definition 类转换方法

方法	描述
类型 convertBodyTo(Class type)	将 IN 消息正文转换为指定的类型。
类型 removeFaultHeader（字符串名称）	添加处理器，以删除 FAULT 消息上的标头。
type removeHeader(String name)	添加可移除 IN 消息上的标头的处理器。
键入 removeProperty(String name)	添加可移除交换属性的处理器。
ExpressionClause<ProcessorDefinition<Type>> setBody（）	添加在 IN 消息上设置正文的处理器。
键入 setFaultBody(Expression expression)	添加在 FAULT 消息上设置正文的处理器。
type setFaultHeader(String name, Expression expression)	添加在 FAULT 消息上设置标头的处理器。
ExpressionClause<ProcessorDefinition<Type>> setHeader(String name)	添加在 IN 消息上设置标头的处理器。
type setHeader(String name, Expression expression)	添加在 IN 消息上设置标头的处理器。
ExpressionClause<ProcessorDefinition<Type>> setOutHeader(String name)	添加在 OUT 消息上设置标头的处理器。
type setOutHeader(String name, Expression expression)	添加在 OUT 消息上设置标头的处理器。
ExpressionClause<ProcessorDefinition<Type>> setProperty(String name)	添加可设置 Exchange 属性的处理器。
键入 setProperty(String name, Expression expression)	添加可设置 Exchange 属性的处理器。
ExpressionClause<ProcessorDefinition<Type>> transform()	添加在 OUT 消息上设置正文的处理器。
类型转换（Expression 表达式）	添加在 OUT 消息上设置正文的处理器。

builder 类

org.apache.camel.builder.Builder 类提供对预期表达式或 predicates 的上下文中的消息内容的访问。换句话说，在 DSL 命令的参数 中通常会调用 Builder 方法。例如， *例 2.1 “引入消息的简单转换”* ***中***

的 body（） 命令。表 2.6 “Builder 类中的方法” 总结 Builder 类中的静态方法。

表 2.6. Builder 类中的方法

方法	描述
静态 <E 扩展 Exchange> ValueBuilder<E> body（）	为交换上的入站正文返回 predicate 和 value builder。
静态 <E 扩展 Exchange,T> ValueBuilder<E> bodyAs(Class<T> type)	为入站邮件正文返回 predicate 和 value builder，作为特定类型的值。
静态 <E 扩展 Exchange> ValueBuilder<E> constant(Object value)	返回恒定表达式。
静态 <E 扩展 Exchange> ValueBuilder<E> faultBody（）	为交换上的错误正文返回 predicate 和 value builder。
静态 <E 扩展 Exchange,T> ValueBuilder<E> faultBodyAs(Class<T> type)	为错误消息正文返回 predicate 和 value builder，作为特定类型的值。
静态 <E 扩展 Exchange> ValueBuilder<E> 标头（字符串名称）	为交换上的标头返回 predicate 和 value builder。
静态 <E 扩展 Exchange> ValueBuilder<E> outBody（）	为交换上的出站正文返回 predicate 和 value builder。
静态 <E 扩展 Exchange> ValueBuilder<E> outBodyAs(Class<T> type)	为出站邮件正文返回 predicate 和 value builder，作为特定类型的特定类型。
static ValueBuilder 属性（字符串名称）	为交换的属性返回 predicate 和 value builder。
static ValueBuilder regexReplaceAll(Expression content, String regex, Expression replacement)	返回表达式将所有出现正则表达式的表达式替换为给定的替换。
static ValueBuilder regexReplaceAll(Expression content, String regex, String replacement)	返回表达式将所有出现正则表达式的表达式替换为给定的替换。
static ValueBuilder sendTo(String uri)	返回将交换发送到给定端点 uri 的表达式。
静态 <E 扩展 Exchange> ValueBuilder<E> systemProperty(String name)	返回给定系统属性的表达式。
静态 <E 扩展 Exchange> ValueBuilder<E> systemProperty(String name, String defaultValue)	返回给定系统属性的表达式。

ValueBuilder 类

org.apache.camel.builder.ValueBuilder 类允许您修改 Builder 方法返回的值。换句话说，ValueBuilder 中的方法提供了一个简单的修改消息内容的方法。表 2.7 "ValueBuilder 类的修饰符方法" 总结了 ValueBuilder 类中可用的方法。也就是说，该表只显示用于修改在其所调用的值的方法（查看完整详情，请参阅 API 参考文档）。

表 2.7. ValueBuilder 类的修饰符方法

方法	描述
ValueBuilder<E> append(Object value)	使用所给值附加此表达式的字符串评估。
predicate 包含（Object 值）	创建一个 predicate，使左手表达式包含右手表达式的值。
ValueBuilder<E> convertTo(Class type)	使用已注册类型转换器将当前值转换为给定类型。
ValueBuilder<E> convertToString（）	使用注册的类型转换器转换当前值 String。
predicate 结束(Object value)	
<T> T evaluate(Exchange exchange, Class<T> type)	
predicate in(Object… values)	
predicate in(Predicate… predicates)	
predicate 是EqualTo(Object value)	返回 true，如果当前值等于给定 **值** 参数。
predicate 是GreaterThan（Object 值）	返回 true，如果当前值大于给定 **值** 参数。
predicate 是GreaterThanOrEqualTo(Object value)	返回 true，如果当前值大于或等于给定 **值** 参数。
predicate 是InstanceOf(Class type)	返回为 true，如果当前值是给定类型的实例。
predicate isLessThan(Object value)	返回为 true，如果当前值小于给定 **值** 参数。
predicate 是 LessThanOrEqualTo(Object value)	返回为 true，如果当前值小于或等于给定 **值** 参数。
predicate 是NotEqualTo(Object value)	返回 true，如果当前值不等于给定 **值** 参数。
predicate 是NotNull（）	返回 true，如果当前值不是 **null**，则返回。

方法	描述
predicate isNull（）	返回 true，如果当前值是 **null**。
predicate matches（Expression 表达式）	
predicate not(Predicate predicate)	导航 predicate 参数。
ValueBuilder prepend(Object value)	将这个表达式的字符串评估添加到所给值。
predicate regex（字符串 regex）	
ValueBuilder<E> regexReplaceAll（String regex, Expression<E> 替换）	将正则表达式的所有发生情况替换为给定的替换。
ValueBuilder<E> regexReplaceAll(String regex, String replacement)	将正则表达式的所有发生情况替换为给定的替换。
ValueBuilder<E> regexTokenize(String regex)	使用给定的正则表达式对这个表达式的字符串转换进行令牌。
ValueBuilder sort(Comparator comparator)	使用给定比较器对当前值进行排序。
predicate 启动With（Object 值）	返回 true，如果当前值与 **value** 参数的字符串值匹配。
ValueBuilder<E> tokenize（）	使用逗号令牌分隔符标记此表达式的字符串转换。
ValueBuilder<E> 令牌化（字符串令牌）	使用给定令牌分隔符对这个表达式的字符串转换进行令牌转换。

2.6.2. marshalling 和 Unmarshalling

Java DSL 命令

您可以使用以下命令在低级和高级别消息格式转换：

- ***marshal（） abrt-abrt 将高级别数据格式转换为低级数据格式。***
- ***unmarshal（） jaxb-abrt 将低级数据格式转换为高级别数据格式。***

数据格式

Apache Camel 支持以下数据格式的 marshalling 和 unmarshalling:

- Java序列化
- JAXB
- XMLBeans
- XStream

Java序列化

允许您将 Java 对象转换为二进制数据的 Blob。对于这个数据格式，取消marshall 会将二进制 blob 转换为 Java 对象，并将 Java 对象转换为二进制 blob。例如，若要从端点 SourceURL 读取序列化 Java 对象，并将其转换为 Java 对象，您可以使用类似如下的规则：

```
from("SourceURL").unmarshal().serialization()
.<FurtherProcessing>.to("TargetURL");
```

或者，在 Spring XML 中：

```
<camelContext id="serialization" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <serialization/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

JAXB

提供 XML 模式类型和 Java 类型之间的映射（请参阅 https://jaxb.dev.java.net/）。对于 JAXB，unmarshalling 会将 XML 数据类型转换为 Java 对象，并总结 Java 对象转换为 XML 数据类型。在可以使用 JAXB 数据格式之前，您必须使用 JAXB 编译器编译您的 XML 架构，以在架构中生成代表 XML 数据类型的 Java 类。这称为 绑定模式。绑定模式后，您可以使用类似如下的代码定义将 XML 数据解压缩到 Java 对象的规则：

-

```
org.apache.camel.spi.DataFormat jaxb = new
org.apache.camel.converter.jaxb.JaxbDataFormat("GeneratedPackageName");

from("SourceURL").unmarshal(jaxb)
.<FurtherProcessing>.to("TargetURL");
```

其中 generateedPackagename 是 JAXB 编译器生成的 Java 软件包的名称，其中包含代表您的 XML 架构的 Java 类。

或者，在 Spring XML 中：

```
<camelContext id="jaxb" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <jaxb prettyPrint="true" contextPath="GeneratedPackageName"/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

XMLBeans

提供 XML 模式类型和 Java 类型之间的备选映射（请参阅 http://xmlbeans.apache.org/）。对于 XMLBeans，unmarshalling 会将 XML 数据类型转换为 Java 对象，并将 Java 对象转换为 XML 数据类型。例如，要使用 XMLBeans 将 XML 数据解压缩到 Java 对象，您可以使用类似如下的代码：

```
from("SourceURL").unmarshal().xmlBeans()
.<FurtherProcessing>.to("TargetURL");
```

或者，在 Spring XML 中：

```
<camelContext id="xmlBeans" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <xmlBeans prettyPrint="true"/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

XStream

提供 XML 类型和 Java 类型之间的另一个映射（请参阅

http://www.xml.com/pub/a/2004/08/18/xstream.html）。XStream 是一个序列化库（如 Java序列化），可让您将任何 Java 对象转换为 XML。对于 XStream，解压缩将 XML 数据类型转换为 Java 对象，并汇总 Java 对象，将 Java 对象转换为 XML 数据类型。

```
from("SourceURL").unmarshal().xstream()
.<FurtherProcessing>.to("TargetURL");
```

注意

Spring XML 目前不支持 XStream 数据格式。

2.6.3. 端点绑定

什么是绑定？

在 Apache Camel 中，绑定 是一种将端点嵌套在 contract contract-sHistoryLimit 例如，通过应用数据格式、内容增强器或验证步骤的方法。条件或转换应用于传入消息的消息，并应用于消息的补充条件或转换。

DataFormatBinding

对于您要定义 marshals 和 unmarshals 特定数据格式的绑定（请参阅 第 2.6.2 节 “marshalling 和 Unmarshalling”，请参阅 在这种情况下，您需要进行创建绑定的所有操作都是创建 DataFormatBinding 实例，在 constructor 中传递对相关数据格式的引用。

例如：例 2.2 “JAXB Binding” 中的 XML DSL 片断会显示一个绑定（带有 ID、jaxb）的绑定（使用 ID, jaxb ），当它与 Apache Camel 端点关联时，能够放宽和取消汇总 JAXB 数据格式：

例 2.2. JAXB Binding

```
<beans ... >
    ...
    <bean id="jaxb" class="org.apache.camel.processor.binding.DataFormatBinding">
        <constructor-arg ref="jaxbformat"/>
    </bean>

    <bean id="jaxbformat" class="org.apache.camel.model.dataformat.JaxbDataFormat">
        <property name="prettyPrint" value="true"/>
        <property name="contextPath" value="org.apache.camel.example"/>
    </bean>

</beans>
```

将绑定与端点关联

以下的替代方法可用于将绑定与端点关联：

- *绑定 URI*
- *组件*

绑定 URI

要将绑定与端点关联，您可以将端点 URI 前缀为 binding:NameOfBinding，其中 NameOfBinding 是绑定的 bean ID（例如，在 Spring XML 中创建绑定 bean 的 ID）。

例如，以下示例演示了如何将 ActiveMQ 端点与 例 2.2 “JAXB Binding” 中定义的 JAXB 绑定关联。

```
<beans ...>
    ...
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="binding:jaxb:activemq:orderQueue"/>
            <to uri="binding:jaxb:activemq:otherQueue"/>
        </route>
    </camelContext>
    ...
</beans>
```

BindingComponent

您可以做出隐式关联绑定而不是将绑定与端点关联，而不必在 URI 中显示绑定。对于没有隐式绑定的现有端点，实现此目的的最简单方法是使用 BindingComponent 类包装端点。

例如，要将 jaxb 绑定与 activemq 端点关联，您可以定义新的 BindingComponent 实例，如下所示：

```
<beans ... >
    ...
    <bean id="jaxbmq" class="org.apache.camel.component.binding.BindingComponent">
        <constructor-arg ref="jaxb"/>
        <constructor-arg value="activemq:foo."/>
    </bean>
```

```
    <bean id="jaxb" class="org.apache.camel.processor.binding.DataFormatBinding">
        <constructor-arg ref="jaxbformat"/>
    </bean>

    <bean id="jaxbformat" class="org.apache.camel.model.dataformat.JaxbDataFormat">
        <property name="prettyPrint" value="true"/>
        <property name="contextPath" value="org.apache.camel.example"/>
    </bean>

</beans>
```

其中（可选） jaxbmq 的第二个构造器参数定义了 URI 前缀。现在，您可以使用 jaxbmq ID 作为端点 URI 的方案。例如，您可以使用这个绑定组件定义以下路由：

```
<beans ...>
    ...
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="jaxbmq:firstQueue"/>
            <to uri="jaxbmq:otherQueue"/>
        </route>
    </camelContext>
    ...
</beans>
```

前面的路由等同于以下路由，它使用绑定 URI 方法：

```
<beans ...>
    ...
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="binding:jaxb:activemq:foo.firstQueue"/>
            <to uri="binding:jaxb:activemq:foo.otherQueue"/>
        </route>
    </camelContext>
    ...
</beans>
```

注意

对于实施自定义 Apache Camel 组件的开发人员，可以通过实施从 org.apache.camel.spi.HasBinding 接口继承的端点类来实现这一点。

BindingComponent constructors

BindingComponent 类支持以下 constructors：

public BindingComponent()

无参数形式。使用属性注入配置绑定组件实例。

public BindingComponent(Binding binding)

将此绑定组件与指定的 Binding 对象关联，并且 绑定。

public BindingComponent(Binding binding, String uriPrefix)

将此绑定组件与指定的 Binding 对象、绑定和 URI 前缀关联，uriPrefix。这是最常用的构造器。

public BindingComponent(Binding binding, String uriPrefix, String uriPostfix)

此构造器支持额外的 URI post-fix, uri postfix , 参数，该参数会自动附加到使用此绑定组件定义的任何 URI。

实现自定义绑定

除了 DataFormatBinding 之外，可用于汇总和解包数据格式，您可以实施自己的自定义绑定。定义自定义绑定，如下所示：

1. 实施 org.apache.camel.Processor 类，对传入到消费者端点的消息执行转换（从 元素推断）。

2. 实施补充 org.apache.camel.Processor 类，对从制作者端点（位于元素中）发出的消息执行反向转换。

3. 实施 org.apache.camel.spi.Binding 接口，它充当处理器实例的工厂。

绑定接口

例 2.3 “org.apache.camel.spi.Binding 接口” 显示 org.apache.camel.spi.Binding 接口的定义，您必须实现它们来定义自定义绑定。

例 2.3. org.apache.camel.spi.Binding 接口

```
// Java
package org.apache.camel.spi;

import org.apache.camel.Processor;
```

```
/**
 * Represents a <a href="http://camel.apache.org/binding.html">Binding</a> or contract
 * which can be applied to an Endpoint; such as ensuring that a particular
 * <a href="http://camel.apache.org/data-format.html">Data Format</a> is used on messages in
and out of an endpoint.
 */
public interface Binding {

    /**
     * Returns a new {@link Processor} which is used by a producer on an endpoint to implement
     * the producer side binding before the message is sent to the underlying endpoint.
     */
    Processor createProduceProcessor();

    /**
     * Returns a new {@link Processor} which is used by a consumer on an endpoint to process the
     * message with the binding before its passed to the endpoint consumer producer.
     */
    Processor createConsumeProcessor();
}
```

何时使用绑定

当您需要应用相同的转换到许多不同类型的端点时，绑定很有用。

2.7. ATTRIBUTE PLACEHOLDERS

概述

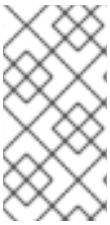
属性占位符功能可用于将字符串替换为不同的上下文（如 XML DSL 元素中的端点 URI 和属性），其中占位符设置存储在 Java 属性文件中。如果您要在不同 Apache Camel 应用程序间共享设置，或者您想要集中特定的配置设置，则这个功能很有用。

例如，以下路由将请求发送到 Web 服务器（其主机和端口由占位符替换）、{{remote.host}} 和 {{remote.port}} ：

```
from("direct:start").to("http://{{remote.host}}:{{remote.port}}");
```

占位符值在 Java 属性文件中定义，如下所示：

```
# Java properties file
remote.host=myserver.com
remote.port=8080
```

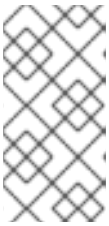


注意

属性 Placeholders 支持编码选项，可让您使用特定的字符集（如 UTF-8）读取 .properties 文件。但是，默认情况下，它会实施 ISO-8859-1 字符集。

使用 PropertyPlaceholders 的 Apache Camel 支持以下内容：

- 将默认值与要查找的键一起指定。
- 不需要定义 PropertiesComponent，如果所有占位符键都包含默认值，则会使用它们。
- 使用第三方函数查找属性值。它可让您实施自己的逻辑。



注意

提供三个开箱即用功能，以从 OS 环境变量、JVM 系统属性或服务名称 idiom 查找值。

属性文件

属性设置存储在一个或多个 Java 属性文件中，且必须符合标准的 Java 属性文件格式。每个属性设置出现在其自己的行中，格式为 Key=Value。带有 # 或 ! 作为第一个非空字符的行被视为注释。

例如，属性文件可能含有内容，如 例 2.4 “Property 文件示例” 所示。

例 2.4. Property 文件示例

```
# Property placeholder settings
# (in Java properties file format)
cool.end=mock:result
cool.result=result
cool.concat=mock:{{cool.result}}
cool.start=direct:cool
cool.showid=true

cheese.end=mock:cheese
cheese.quote=Camel rocks
cheese.type=Gouda
```

```
bean.foo=foo
bean.bar=bar
```

解决属性

属性组件必须配置有一个或多个属性文件的位置，然后才能在路由定义中使用它。您必须使用以下解析器之一提供属性值：

classpath:PathName,PathName,...

（默认） 指定类路径中的位置，其中 PathName 是使用正斜杠分隔的文件路径名。

file:PathName,PathName,...

指定文件系统中的位置，其中 PathName 是以正斜杠分隔的文件路径名。

ref:BeanID

指定 registry 中的 java.util.Properties 对象的 ID。

蓝图：BeanID

指定 cm:property-placeholder bean 的 ID，该 ID 可用于访问 OSGi 配置管理服务中定义的属性。详情请查看 “与 OSGi 蓝图属性占位符集成”一节。

例如，要指定 com/fusesource/cheese.properties 属性文件和 com/fusesource/bar.properties 属性文件，两者均位于 classpath 上，您需要使用以下位置字符串：

```
com/fusesource/cheese.properties,com/fusesource/bar.properties
```

注意

您可以省略本例中的 classpath: 前缀，因为默认情况下使用 classpath 解析器。

使用系统属性和环境变量指定位置

您可以在位置 PathName 中嵌入 Java 系统属性和 O/S 环境变量。

Java 系统属性可以使用语法 ${PropertyName} 嵌入在位置解析器中。例如，如果 Red Hat Fuse 的

根目录存储在 Java 系统属性 karaf.home 中，您可以把该目录值嵌入到文件位置中，如下所示：

```
file:${karaf.home}/etc/foo.properties
```

O/S 环境变量可以使用语法(${env:VarName})嵌入到位置解析器中。例如，如果 JBoss Fuse 的根目录存储在环境变量 SMX_HOME 中，您可以将该目录的内容嵌入到文件位置，如下所示：

```
file:${env:SMX_HOME}/etc/foo.properties
```

配置属性组件

在开始使用属性占位符前，您必须配置属性组件，指定一个或多个属性文件的位置。

在 Java DSL 中，您可以使用属性文件位置配置属性组件，如下所示：

```
// Java
import org.apache.camel.component.properties.PropertiesComponent;
...
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation("com/fusesource/cheese.properties,com/fusesource/bar.properties");
context.addComponent("properties", pc);
```

如 addComponent（） 调用所示，属性组件的名称 必须设置为 属性。

在 XML DSL 中，您可以使用专用 propertyPlacholder 元素配置属性组件，如下所示：

```
<camelContext ...>
  <propertyPlaceholder
    id="properties"
    location="com/fusesource/cheese.properties,com/fusesource/bar.properties"
  />
</camelContext>
```

如果您希望属性组件在初始化时忽略任何缺少的 .properties 文件，您可以将 ignoreMissingLocation 选项设置为 true （通常情况下，缺少 .properties 文件会导致错误引发）。

另外，如果您希望属性组件忽略使用 Java 系统属性或 O/S 环境变量指定的任何缺少的位置，您可以将 ignoreMissingLocation 选项设置为 true。

占位符语法

配置后，属性组件会自动替换占位符（在适当的上下文中）。占位符语法取决于上下文，如下所示：

- *在端点 URI 中，在 Spring XML 文件中 InventoryService-","the 占位符被指定为 {{Key}}。*
- *当设置 XML DSL 属性时，使用以下语法设置 字符串 属性：*

```
AttributeName="{{Key}}"
```

必须使用以下语法设置其他属性类型（例如 xs:int 或 xs:boolean）：

```
prop:AttributeName="Key"
```

其中 prop 与 http://camel.apache.org/schema/placeholder 命名空间关联。

- *在设置 Java DSL EIP 选项 targetNamespaces 选项 InventoryService-jaxbto 时，在 Java DSL 中对企业级集成模式(EIP)命令设置选项时，请在流畅的 DSL 中添加类似于以下内容的 占位符（） 子句：*

```
.placeholder("OptionName", "Key")
```

- *在简单语言表达式中，placeholder指定为 ${properties:Key}。*

在端点 URI 中替换

只要端点 URI 字符串出现在路由中，解析端点 URI 的第一步是应用属性占位符解析器。占位符解析器自动替换双花括号之间出现的任何属性名称 {{ Key}}。例如，如果 例 2.4 “Property 文件示例” 中显示的属性设置，您可以定义路由，如下所示：

```
from("{{cool.start}}")
    .to("log:{{cool.start}}?showBodyType=false&showExchangeId={{cool.showid}}")
    .to("mock:{{cool.result}}");
```

默认情况下，占位符解析器查找 registry 中的 属性 bean ID 来查找属性组件。如果愿意，您可以在端点 URI 中明确指定方案。例如，通过为每个端点 URI 加上前缀 属性：到每个端点 URI，您可以定义以下

等同的路由：

```
from("properties:{{cool.start}}")
    .to("properties:log:{{cool.start}}?showBodyType=false&showExchangeId={{cool.showid}}")
    .to("properties:mock:{{cool.result}}");
```

在明确指定方案时，您还可以选择指定属性组件的选项。例如，要覆盖属性文件位置，您可以按照如下所示设置 **location** 选项：

```
from("direct:start").to("properties:{{bar.end}}?location=com/mycompany/bar.properties");
```

在 **Spring XML** 文件中替换

您还可以使用 **XML DSL** 中的属性占位符，设置 **DSL** 元素的各种属性。在此上下文中，拥有者语法也使用双花括号 **{{ Key}}**。例如，您可以使用属性占位符定义 **jmxAgent** 元素，如下所示：

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

    <!-- we can use property placeholders when we define the JMX agent -->
    <jmxAgent id="agent" registryPort="{{myjmx.port}}"
              usePlatformMBeanServer="{{myjmx.usePlatform}}"
              createConnector="true"
              statisticsLevel="RoutesOnly"
            />

    <route>
        <from uri="seda:start"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

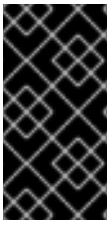
替换 **XML DSL** 属性值

您可以使用常规占位符语法来指定 **xs:string typegradle-sHistoryLimitfor example, < jmxAgent registryPort="{{myjmx.port}}" ... >**。但是，对于任何其他类型的属性（例如，**xs:int** 或 **xs:boolean**），您必须使用特殊语法 **prop:AttributeName="Key"**。

例如，如果某个属性文件定义了 **stop.flag** 属性的值为 **true**，您可以使用此属性来设置 **stopOnException** 布尔值属性，如下所示：

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:prop="http://camel.apache.org/schema/placeholder"
```

```
    ... >

  <bean id="illegal" class="java.lang.IllegalArgumentException">
    <constructor-arg index="0" value="Good grief!"/>
  </bean>

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

    <propertyPlaceholder id="properties"
                         location="classpath:org/apache/camel/component/properties/myprop.properties"
                         xmlns="http://camel.apache.org/schema/spring"/>

    <route>
      <from uri="direct:start"/>
      <multicast prop:stopOnException="stop.flag">
        <to uri="mock:a"/>
        <throwException ref="damn"/>
        <to uri="mock:b"/>
      </multicast>
    </route>

  </camelContext>

</beans>
```



重要

必须明确分配给 **Spring** 文件中的 **http://camel.apache.org/schema/placeholder** 命名空间，如上例的 **Bean** 元素中所示。

替换 Java DSL EIP 选项

在 **Java DSL** 中调用 **EIP** 命令时，您可以通过添加表单、占位符(" **options Name**", "**Key**") 值，使用属性值来设置任何 **EIP** 选项。

例如，如果某个属性文件定义了 **stop.flag** 属性的值为 **true**，您可以使用此属性设置多播 **EIP** 的 **stopOnException** 选项，如下所示：

```
from("direct:start")
  .multicast().placeholder("stopOnException", "stop.flag")
    .to("mock:a").throwException(new IllegalAccessException("Damn")).to("mock:b");
```

使用简单语言表达式替换

您还可以使用简单语言表达式替换属性占位符，但在本例中，占位符语法为 **${properties:Key}**。例如，您可以替换简单表达式内的 **cheese.quote** 占位符，如下所示：

```
from("direct:start")
   .transform().simple("Hi ${body} do you think ${properties:cheese.quote}?");
```

您可以使用语法 ${properties:Key:DefaultVal} 来指定属性的默认值。例如：

```
from("direct:start")
   .transform().simple("Hi ${body} do you think ${properties:cheese.quote:cheese is good}?");
```

也可以使用语法(${properties-location:Location:Key})覆盖属性文件的位置。例如，要使用 com/mycompany/bar.properties 属性中的设置替换 bar.quote 占位符，您可以按照如下所示定义简单表达式：

```
from("direct:start")
   .transform().simple("Hi ${body}. ${properties-location:com/mycompany/bar.properties:bar.quote}.");
```

在 XML DSL 中使用 Property Placeholders

在旧版中，xs:string 类型属性用于支持 XML DSL 中的占位符。例如，timeout 属性是一个 xs:int 类型。因此，您无法将字符串值设置为占位符键。

从 Apache Camel 2.7 开始，现在可以使用特殊的占位符命名空间来实现。以下示例演示了命名空间的 prop 前缀。它允许您使用 XML DSL 中属性中的 prop 前缀。

注意

在多播中，将选项 stopOnException 设置为带有键 stop 的占位符值。另外，在属性文件中，将值定义为

```
stop=true
```

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

   <!-- Notice in the declaration above, we have defined the prop prefix as the Camel placeholder
namespace -->
```

```
    <bean id="damn" class="java.lang.IllegalArgumentException">
      <constructor-arg index="0" value="Damn"/>
    </bean>

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

      <propertyPlaceholder id="properties"
                           location="classpath:org/apache/camel/component/properties/myprop.properties"
                           xmlns="http://camel.apache.org/schema/spring"/>

      <route>
        <from uri="direct:start"/>
        <!-- use prop namespace, to define a property placeholder, which maps to
             option stopOnException={{stop}} -->
        <multicast prop:stopOnException="stop">
          <to uri="mock:a"/>
          <throwException ref="damn"/>
          <to uri="mock:b"/>
        </multicast>
      </route>

    </camelContext>

  </beans>
```

与 OSGi 蓝图属性占位符集成

如果您将路由部署到红帽 Fuse OSGi 容器中，您可以将 Apache Camel 属性占位符机制与 JBoss Fuse 的蓝图属性占位符机制（事实上是启用集成）。设置集成的基本方法有两种，如下所示：

- *隐式蓝图集成*
- *显式蓝图集成*

隐式蓝图集成

如果您在 OSGi 蓝图文件中定义了 camelContext 元素，Apache Camel 属性占位符机制会自动与蓝图属性占位符机制集成。也就是说，在 camelContext 范围内出现的占位符遵循 Apache Camel 语法（如 {{cool.end}}）通过查找 蓝图属性占位符 机制来隐式解析。

例如，考虑在 OSGi 蓝图文件中定义以下路由，路由中最后一个端点由属性占位符定义，{{result}}：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
```

```
        xsi:schemaLocation="
        http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

  <!-- OSGI blueprint property placeholder -->
  <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint">
    <!-- list some properties for this test -->
    <cm:default-properties>
      <cm:property name="result" value="mock:result"/>
    </cm:default-properties>
  </cm:property-placeholder>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <!-- in the route we can use {{ }} placeholders which will look up in blueprint,
       as Camel will auto detect the OSGi blueprint property placeholder and use it -->
    <route>
      <from uri="direct:start"/>
      <to uri="mock:foo"/>
      <to uri="{{result}}"/>
    </route>
  </camelContext>

</blueprint>
```

*蓝图**属性占位符机制通**过创建一个 **cm:property-placeholder an** 初始化。在前面的示例中，**cm:property-placeholder bean** 与 **camel.blueprint** 持久 **ID** 关联，持久 **ID** 是引用 **OSGi** 配置管理服务中一组**相关属性的**标准方式。换句话说，**cm:property-placeholder an** 提供了对 **camel.blueprint** 持久 **ID** 中定义**的所有属性的**访问权**限。也可以为某些属性指定默**认值（使用嵌套的 **cm:property** 元素）。*

*在蓝图上下文中，**Apache Camel** 占位符机制搜索 **bean registry** 中的 **cm:property-placeholder** 实例。如果找到这样的实例，它会自动集成 **Apache Camel** 占位符机制，以便等占位符 **{{result}}** 可以通过在蓝图**属性占位符机制中查找密**钥（在这个示例中，通过 **myblueprint.placeholder bean**）来解决。*

注意

***默**认蓝图**占位符**语法（直接访问蓝图**属性**）为 **${Key}**。因此，在 **camelContext** 元素的范围 之外，您必须使用占位符语法为 **${Key}**。但在 **camelContext** 元素的范围中，您必须使用占位符语法是 **{{ Key}}**。*

显式蓝图集成

如果您想要更多控制 Apache Camel 属性占位符机制在什么位置找到其属性，您可以定义一个 propertyPlaceholder 元素并明确指定解析器位置。

例如，请考虑以下与前面示例不同的蓝图配置，它会创建一个显式 propertyPlaceholder 实例：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
         xsi:schemaLocation="
         http://www.osgi.org/xmlns/blueprint/v1.0.0
">https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

   <!-- OSGI blueprint property placeholder -->
   <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint">
      <!-- list some properties for this test -->
      <cm:default-properties>
         <cm:property name="result" value="mock:result"/>
      </cm:default-properties>
   </cm:property-placeholder>

   <camelContext xmlns="http://camel.apache.org/schema/blueprint">

      <!-- using Camel properties component and refer to the blueprint property placeholder by its id --
>
      <propertyPlaceholder id="properties" location="blueprint:myblueprint.placeholder"/>

      <!-- in the route we can use {{ }} placeholders which will lookup in blueprint -->
      <route>
         <from uri="direct:start"/>
         <to uri="mock:foo"/>
         <to uri="{{result}}"/>
      </route>

   </camelContext>

</blueprint>
```

在前面的示例中，propertyPlaceholder 元素指定通过将位置设置为 blueprint:myblueprint.placeholder 来指定使用哪个 cm:property-placeholder。该蓝图：解析器明确引用 cm:property-placeholder 的 ID、myblueprint.placeholder。

这种配置方式很有用，如果蓝图文件中定义了多个 cm:property-placeholder an，并且您需要指定要使用的选项。它还可通过指定以逗号分隔的位置列表，从多个位置的源属性。例如，如果要从 cm:property-placeholder bean 和 properties 文件(myproperties.properties)在 classpath 上查找属性，您可以按照如下所示定义 propertyPlaceholder 元素：

```
<propertyPlaceholder id="properties"
  location="blueprint:myblueprint.placeholder,classpath:myproperties.properties"/>
```

与 Spring 属性占位符集成

如果您在 Spring XML 文件中使用 XML DSL 定义 Apache Camel 应用程序，您可以通过声明类型为 Spring 片段机制来将 Apache Camel 属性占位符机制集成，org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer。

定义 BridgePropertyPlaceholderConfigurer，它替代了 Spring XML 文件中 Apache Camel 的 propertyPlaceholder 元素和 Spring 的 ctx:property-placeholder 元素。然后，您可以使用 Spring ${PropName} 语法或 Apache Camel {{ PropName}} 语法来引用配置的属性。

例如，定义从 cheese.properties 文件中读取其属性设置的桥接属性占位符：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ctx="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <!-- Bridge Spring property placeholder with Camel -->
  <!-- Do not use <ctx:property-placeholder ... > at the same time -->
  <bean id="bridgePropertyPlaceholder"
        class="org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:org/apache/camel/component/properties/cheese.properties"/>
  </bean>

  <!-- A bean that uses Spring property placeholder -->
  <!-- The ${hi} is a spring property placeholder -->
  <bean id="hello" class="org.apache.camel.component.properties.HelloBean">
    <property name="greeting" value="${hi}"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- Use Camel's property placeholder {{ }} style -->
    <route>
      <from uri="direct:{{cool.bar}}"/>
      <bean ref="hello"/>
      <to uri="{{cool.end}}"/>
    </route>
  </camelContext>

</beans>
```

注意

另外，您可以设置 BridgePropertyPlaceholderConfigurer 的 location 属性，使其指向 Spring 属性文件。完全支持 Spring 属性文件语法。

2.8. 线程模型

Java 线程池 API

Apache Camel 线程模型基于强大的 Java 并发 API Package java.util.concurrent，最初在 Sun 的 JDK 1.5 中提供。此 API 中的关键接口是 ExecutorService 接口，它代表线程池。使用并发 API，您可以创建许多不同的线程池，涵盖了广泛的场景。

Apache Camel 线程池 API

Apache Camel 线程池 API 将基于 Java 并发 API 构建，为 Apache Camel 应用程序中的所有线程池提供中心工厂（org.apache.camel.spi.ExecutorServiceManager 类型）。以这种方式进行线程池的创建提供了几个优点，包括：

- 使用实用程序类简化线程池的创建。
- 将线程池与安全关闭集成.
- 线程会自动提供信息名称，这对记录和管理有益。

组件线程模型

一些 Apache Camel 组件为 SEDA、JMS 和 Jettytty- inherent inherent多线程。这些组件已使用 Apache Camel 线程模型和线程池 API 实施。

如果您计划实施自己的 Apache Camel 组件，建议您将线程代码与 Apache Camel 线程模型集成。例如，如果您的组件需要线程池，建议您使用 CamelContext 的 ExecutorServiceManager 对象创建它。

处理器线程模型

Apache Camel 中的一些标准处理器默认创建自己的线程池。这些线程感知型处理器也与 Apache Camel 线程模型集成，它们提供各种选项，供您自定义它们使用的线程池。

表 2.8 “处理器线程选项” 显示了在内置到 Apache Camel 的线程处理器上控制和设置线程池的各种选项。

表 2.8. 处理器线程选项

处理器	Java DSL	XML DSL
aggregate	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
multicast	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
recipientList	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
split	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
threads	executorService() executorServiceRef() poolSize() maxPoolSize() keepAliveTime() timeUnit() maxQueueSize() rejectedPolicy()	@executorServiceRef @poolSize @maxPoolSize @keepAliveTime @timeUnit @maxQueueSize @rejectedPolicy
wireTap	wireTap(String uri, ExecutorService executorService) wireTap(String uri, String executorServiceRef)	@executorServiceRef

线程 DSL 选项

线程处理器是一种通用的 DSL 命令，可用于将线程池引入到路由中。它支持以下选项自定义线程池：

poolSize()

池中的最小线程数量（以及初始池大小）。

maxPoolSize()

池中的最大线程数量。

keepAliveTime()

如果任何线程闲置的时间超过这个时间段（以秒为单位指定），则这些线程将被终止。

timeUnit()

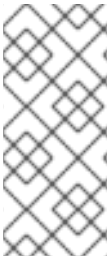
使用 java.util.concurrent.TimeUnit 类型指定的时间单位。

maxQueueSize()

此线程池可以在其传入任务队列中存储的最大待处理任务数量。

rejectedPolicy()

指定在传入的任务队列已满时要执行的操作。请查看 表 2.10 “线程池构建器选项”



注意

前面的线程池选项与 executorServiceRef 选项不兼容（例如，您不能使用这些选项覆盖 executorServiceRef 选项引用的线程池中的设置）。Apache Camel 验证 DSL 来执行此操作。

创建默认线程池

要为一个线程感知处理器创建默认线程池，启用 parallelProcessing 选项，使用 parallelProcessing（） 子使用、Java DSL 或 parallelProcessing 属性（在 XML DSL 中）。

例如，在 Java DSL 中，您可以使用默认线程池（使用线程池同时处理多播目的地）调用多播处理器：

```
from("direct:start")
  .multicast().parallelProcessing()
```

```
    .to("mock:first")
    .to("mock:second")
    .to("mock:third");
```

您可以在 XML DSL 中定义与以下方式相同的路由

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <multicast parallelProcessing="true">
      <to uri="mock:first"/>
      <to uri="mock:second"/>
      <to uri="mock:third"/>
    </multicast>
  </route>
</camelContext>
```

默认线程池配置集设置

默认线程池由线程工厂自动创建，它从 默认的线程池配置集 获取其设置。默认线程池配置集在 表 2.9 “默认线程池配置集设置” 中显示的设置（假设这些设置还没有由应用程序代码修改）。

表 2.9. 默认线程池配置集设置

线程选项	默认值
maxQueueSize	**1000**
poolSize	**10**
maxPoolSize	**20**
keepAliveTime	**60（秒）**
rejectedPolicy	**CallerRuns**

更改默认线程池配置集

可以更改默认的线程池配置集设置，以便所有后续的默认线程池都使用自定义设置来创建。您可以在 Java 或 Spring XML 中更改配置集。

例如，在 Java DSL 中，您可以自定义 poolSize 选项和默认线程池配置集中的 maxQueueSize 选项，如下所示：

```
// Java
import org.apache.camel.spi.ExecutorServiceManager;
import org.apache.camel.spi.ThreadPoolProfile;
...
ExecutorServiceManager manager = context.getExecutorServiceManager();
ThreadPoolProfile defaultProfile = manager.getDefaultThreadPoolProfile();

// Now, customize the profile settings.
defaultProfile.setPoolSize(3);
defaultProfile.setMaxQueueSize(100);
...
```

在 XML DSL 中，您可以自定义默认的线程池配置集，如下所示：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <threadPoolProfile
        id="changedProfile"
        defaultProfile="true"
        poolSize="3"
        maxQueueSize="100"/>
    ...
</camelContext>
```

请注意，在前面的 XML DSL 示例中将 defaultProfile 属性设置为 true 非常重要，否则线程池配置集将被视为自定义线程池配置集（请参阅 “创建自定义线程池配置集”一节），而不是替换默认的线程池配置集。

自定义处理器的线程池

还可以使用 executorService 或 executorServiceRef 选项（其中使用这些选项而不是 parallelProcessing 选项）指定线程池。您可以使用两种方法来自定义处理器的线程池，如下所示：

- 指定自定义线程池 是否已创建 ExecutorService （线程池）实例，并将它传递到 executorService 选项。
- 指定自定义线程池配置集 InventoryService- responsiblecreate 并注册自定义线程池工厂。当您使用 executorServiceRef 选项引用这个 factory 时，处理器会自动使用 factory 创建自定义线程池实例。

当您将 bean ID 传递给 executorServiceRef 选项时，线程识别处理器首先会尝试查找带有该 ID 在 registry 中的 ID 的自定义线程池。如果没有使用该 ID 注册了线程池，则处理器会尝试在 registry 中查找自定义线程池配置集，并使用自定义线程池配置集来实例化自定义线程池。

创建自定义线程池

自定义线程池可以是 java.util.concurrent.ExecutorService 类型的任何线程池。在 Apache Camel 中，建议使用以下创建线程池实例的方法：

- *使用 org.apache.camel.builder.ThreadPoolBuilder 实用程序构建线程池类。*
- *使用来自当前 CamelContext 的 org.apache.camel.spi.ExecutorServiceManager 实例来创建线程池类。*

最后，这两种方法之间没有区别，因为 ThreadPoolBuilder 实际使用 ExecutorServiceManager 实例定义。通常，最好使用 ThreadPoolBuilder，因为它提供了一个更简单的方法。但是，至少有一种线程(ScheduledExecutorService)只能通过直接访问 ExecutorServiceManager 实例来创建。

表 2.10 "线程池构建器选项" 显示 ThreadPoolBuilder 类支持的选项，您可以在定义新的自定义线程池时设置这些选项。

表 2.10. 线程池构建器选项

构建器选项	描述
maxQueueSize()	设置此线程池可以存储在其传入任务队列中的最大待处理任务数量。值 **-1** 指定未绑定的队列。默认值从默认的线程池配置集中提取。
poolSize()	设置池中最小线程数量（这也是初始池大小）。默认值从默认的线程池配置集中提取。
maxPoolSize()	设置池中可以的最大线程数量。默认值从默认的线程池配置集中提取。
keepAliveTime()	如果任何线程闲置的时间超过这个时间段（以秒为单位指定），则这些线程将被终止。这样，当负载较正常时，线程池可以缩小。默认值从默认的线程池配置集中提取。

构建器选项	描述
rejectedPolicy()	指定在传入的任务队列已满时要执行的操作。您可以指定 4 个可能值： **CallerRuns** **（默认值）** 获取调用器线程以运行最新的传入任务。这个问题的一个副作用是，此选项可防止调用者线程接收更多任务，直到完成最新的传入任务。 **Abort** 通过抛出异常以中止最新的传入任务。 **discard** 以静默方式丢弃最新的传入任务。 **DiscardOldest** 丢弃最旧的未处理的任务，然后尝试在任务队列中放入最新的传入的任务。
build()	完成构建自定义线程池，并将新线程池注册到 **build（）** 的 ID 下。

在 Java DSL 中，您可以使用 ThreadPoolBuilder 定义自定义线程池，如下所示：

```
// Java
import org.apache.camel.builder.ThreadPoolBuilder;
import java.util.concurrent.ExecutorService;
...
ThreadPoolBuilder poolBuilder = new ThreadPoolBuilder(context);
ExecutorService customPool =
poolBuilder.poolSize(5).maxPoolSize(5).maxQueueSize(100).build("customPool");
...

from("direct:start")
  .multicast().executorService(customPool)
    .to("mock:first")
    .to("mock:second")
    .to("mock:third");
```

您可以通过将 bean ID 传递给 executorServiceRef（） 选项，而不必将对象引用 customPool 直接传递给 executorService（） 选项，如下所示：

```
// Java
from("direct:start")
  .multicast().executorServiceRef("customPool")
    .to("mock:first")
    .to("mock:second")
    .to("mock:third");
```

在 XML DSL 中，您可以使用 threadPool 元素访问 ThreadPoolBuilder。然后，您可以使用 executorServiceRef 属性来引用自定义线程池，以便在 Spring registry 中根据 ID 查找线程池，如下所示：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <threadPool id="customPool"
                poolSize="5"
                maxPoolSize="5"
                maxQueueSize="100" />

    <route>
        <from uri="direct:start"/>
        <multicast executorServiceRef="customPool">
            <to uri="mock:first"/>
            <to uri="mock:second"/>
            <to uri="mock:third"/>
        </multicast>
    </route>
</camelContext>
```

创建自定义线程池配置集

如果您有多个要创建的自定义线程池实例，您可能会发现它更加方便地定义自定义线程池配置集，该配置集充当线程池的工厂。每当您引用线程感知处理器的线程池配置集时，处理器会自动使用配置集来创建新的线程池实例。您可以在 Java DSL 或 XML DSL 中定义自定义线程池配置集。

例如，在 Java DSL 中，您可以使用 bean ID customProfile 并从路由中引用它，如下所示：

```
// Java
import org.apache.camel.spi.ThreadPoolProfile;
import org.apache.camel.impl.ThreadPoolProfileSupport;
...
// Create the custom thread pool profile
ThreadPoolProfile customProfile = new ThreadPoolProfileSupport("customProfile");
customProfile.setPoolSize(5);
customProfile.setMaxPoolSize(5);
customProfile.setMaxQueueSize(100);
context.getExecutorServiceManager().registerThreadPoolProfile(customProfile);
...
// Reference the custom thread pool profile in a route
from("direct:start")
  .multicast().executorServiceRef("customProfile")
    .to("mock:first")
    .to("mock:second")
    .to("mock:third");
```

在 XML DSL 中，使用 threadPoolProfile 元素创建自定义池配置集（其中，默认Profile 选项默认为 false，因为这 不是默认 的线程池配置集）。您可以使用 bean ID customProfile 创建自定义线程池配置

集，并从路由中引用它，如下所示：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <threadPoolProfile
        id="customProfile"
        poolSize="5"
        maxPoolSize="5"
        maxQueueSize="100" />

  <route>
    <from uri="direct:start"/>
    <multicast executorServiceRef="customProfile">
      <to uri="mock:first"/>
      <to uri="mock:second"/>
      <to uri="mock:third"/>
    </multicast>
  </route>
</camelContext>
```

在组件间共享线程池

某些标准基于轮询的组件（如 File 和 FTP1028-jaxballow）要指定要使用的线程池。这使得不同组件可以共享同一线程池，从而减少了 JVM 中的总线程数量。

例如，Apache Camel 组件参考指南 中的 see File2 和 Apache Camel 组件参考指南 中的 Ftp2 均公开 scheduledExecutorService 属性，您可以使用它来指定组件的 ExecutorService 对象。

自定义线程名称

要使应用程序日志更易阅读，通常最好自定义线程名称（用于识别日志中的线程）。要自定义线程名称，您可以通过在 ExecutorServiceStrategy 类或 ExecutorServiceManager 类上调用 setThreadNamePattern 方法来配置线程名称 模式。另外，设置线程名称模式的更简单方法是在 CamelContext 对象中设置 threadNamePattern 属性。

以下占位符可以在线程名称模式中使用：

#camelId#

当前 CamelContext 的名称。

#counter#

唯一的线程标识符，作为递增计数器实施。

#name#

常规 Camel 线程名称。

#longName#

长线程名称 3.10.0-->_< 可以包含端点参数，以此类推。

以下是线程名称模式的典型示例：

```
Camel (#camelId#) thread #counter# - #name#
```

以下示例演示了如何使用 XML DSL 在 Camel 上下文上设置 threadNamePattern 属性：

```
<camelContext xmlns="http://camel.apache.org/schema/spring"
              threadNamePattern="Riding the thread #counter#" >
  <route>
    <from uri="seda:start"/>
    <to uri="log:result"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

2.9. 控制路由启动和关闭

概述

默认情况下，当 Apache Camel 应用程序（由 CamelContext 实例表示）启动时自动启动路由，并在您的 Apache Camel 应用程序关闭时自动关闭路由。对于非关键部署，关闭序列的详情通常并不重要。但在生产环境中，现有任务在关闭期间应该运行非常关键，以避免数据丢失。您通常还希望控制路由关闭的顺序，这样依赖项就不会被违反（这样会导致现有任务无法运行完成）。

因此，Apache Camel 提供了一组功能来支持 安全关闭 应用程序。安全关闭可让您完全控制停止和启动路由，允许您控制路由的关闭顺序，并使当前任务能够运行完成。

设置路由 ID

最好为每个路由分配一个路由 ID。除了提高日志记录消息和管理功能外，使用路由 ID 可让您对停止和启动路由应用更多的控制。

例如，在 Java DSL 中，您可以通过调用 routeId（） 命令来将路由 ID myCustomerRouteId 分配到路由，如下所示：

```
from("SourceURI").routeId("myCustomRouteId").process(...).to(TargetURI);
```

在 XML DSL 中，设置 route 元素的 id 属性，如下所示：

```
<camelContext id="CamelContextID" xmlns="http://camel.apache.org/schema/spring">
  <route id="myCustomRouteId" >
    <from uri="SourceURI"/>
    <process ref="someProcessorId"/>
    <to uri="TargetURI"/>
  </route>
</camelContext>
```

禁用自动启动路由

默认情况下，CamelContext 知道在启动时的所有路由都将自动启动。如果您想手动控制特定路由的启动启动，您可能需要为该路由禁用自动启动。

要控制 Java DSL 路由是否自动启动，可调用 autoStartup 命令，使用 布尔值 参数（true 或 false）或 String 参数（true 或 false）。例如，您可以在 Java DSL 中禁用路由的自动启动，如下所示：

```
from("SourceURI")
  .routeId("nonAuto")
  .autoStartup(false)
  .to(TargetURI);
```

您可以通过在路由元素上将 autoStartup 属性设置为 false 来禁用在 XML DSL 中自动启动 路由的 自动启动，如下所示：

```
<camelContext id="CamelContextID" xmlns="http://camel.apache.org/schema/spring">
  <route id="nonAuto" autoStartup="false">
    <from uri="SourceURI"/>
    <to uri="TargetURI"/>
  </route>
</camelContext>
```

手动启动和停止路由

您可以通过调用 CamelContext 实例的 startRoute（） 和 stopRoute（） 方法，随时在 Java 中手动启动或停止路由。例如，要启动具有路由 ID nonAuto 的路由，在 CamelContext 实例（上下文）上

调用 **startRoute（）** 方法，如下所示：

```
// Java
context.startRoute("nonAuto");
```

要停止具有路由 **ID nonAuto** 的路由，调用 **CamelContext** 实例上的 **stopRoute（）** 方法，上下文，如下所示：

```
// Java
context.stopRoute("nonAuto");
```

路由启动顺序

默认情况下，**Apache Camel** 以非确定的顺序启动路由。但是，在某些应用程序中，控制启动顺序非常重要。要在 **Java DSL** 中控制启动顺序，请使用 **startupOrder（）** 命令，该命令使用正整数值作为其参数。具有最低整数值的路由首先启动，后接具有后续启动顺序值的路由。

例如，以下示例中的前两个路由通过 **seda:buffer** 端点链接在一起。您可以通过分配启动顺序（**2** 和 **1**）来保证在第二个路由片段 后 启动第一个路由片段，如下所示：

例 2.5. Java DSL 中的启动顺序

```
from("jetty:http://fooserver:8080")
    .routeId("first")
    .startupOrder(2)
    .to("seda:buffer");

from("seda:buffer")
    .routeId("second")
    .startupOrder(1)
    .to("mock:result");

// This route's startup order is unspecified
from("jms:queue:foo").to("jms:queue:bar");
```

在 **Spring XML** 中，您可以通过设置 **route** 元素的 **startupOrder** 属性来实现相同的效果，如下所示：

例 2.6. XML DSL 中的启动顺序

```
<route id="first" startupOrder="2">
    <from uri="jetty:http://fooserver:8080"/>
    <to uri="seda:buffer"/>
</route>
```

```
<route id="second" startupOrder="1">
    <from uri="seda:buffer"/>
    <to uri="mock:result"/>
</route>

<!-- This route's startup order is unspecified -->
<route>
    <from uri="jms:queue:foo"/>
    <to uri="jms:queue:bar"/>
</route>
```

必须为每个路由分配一个唯一的启动顺序值。您可以选择小于 1000 的正整数值。为 Apache Camel 保留 1000 和 over 的值，这会自动将这些值分配给路由，而不显示显式启动值。例如，上例中最后一个路由会自动分配启动值 1000（因此在前两个路由后启动）。

关闭序列

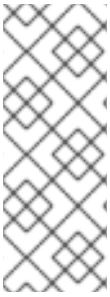
当 CamelContext 实例时，Apache Camel 使用可插入关闭 策略控制关闭 序列。默认关闭策略实施以下关闭序列：

1. 路由以 相反的 启动顺序关闭。
2. 通常，关闭策略会等待当前活动的交换发生。但是，运行任务的处理是可配置的。
3. 总体而言，关闭序列由超时（默认为 300 秒）绑定。如果关闭序列超过此超时，则关闭策略将强制关闭，即使有些任务仍在运行。

路由关闭顺序

路由以相反的启动顺序关闭。也就是说，当使用 boot-rder （） 命令（在 Java DSL 中）或 startupOrder 属性（在 XML DSL 中）定义启动顺序时，第一个关闭路由是第一个关闭的路由，由启动顺序分配 最高 的整数值。最后一个要关闭的路由是按照启动顺序分配 的最低 整数值的路由。

例如，在 例 2.5 “Java DSL 中的启动顺序” 中，需要关闭的第一个路由段是 ID、第一个 的路由，而要关闭的第二个路由段则是 ID 为 second 的路由。本例演示了一个常规规则，您应该在关闭路由时观察这些规则：公开外部可访问消费者端点的路由应首先关闭，因为这有助于通过路由图的剩余部分来节流消息流。



注意

Apache Camel 还提供选项 shutdownRoute(Defer)，它可让您指定路由必须位于关闭的最后一个路由中（覆盖了启动顺序值）。但是您很少需要此选项。这个选项主要是作为早期版本的 Apache Camel（prior 到 2.3）的一个临时解决方案，哪个路由会按照 与启动顺序相同的 顺序关闭。

关闭路由中正在运行的任务

如果关闭启动时路由仍在处理消息，关闭策略通常会等待当前活跃交换完成处理后再关闭路由。可以使用 shutdownRunningTask 选项在每个路由上配置此行为，该选项可采用以下值之一：

ShutdownRunningTask.CompleteCurrentTaskOnly

（默认） 通常，路由一次只在一条信息上运行，因此您可以在当前任务完成后安全地关闭路由。

ShutdownRunningTask.CompleteAllTasks

指定这个选项，以安全地关闭 批处理消费者。有些消费者端点（如 File、FTP、邮件、iBATIS 和 JPA）一次在批量消息上运行。对于这些端点，更适合等待当前批处理中的所有消息都已完成。

例如，要正常关闭文件消费者端点，您应该指定 CompleteAllTasks 选项，如以下 Java DSL 片段所示：

```
// Java
public void configure() throws Exception {
    from("file:target/pending")
        .routeId("first").startupOrder(2)
        .shutdownRunningTask(ShutdownRunningTask.CompleteAllTasks)
        .delay(1000).to("seda:foo");

    from("seda:foo")
        .routeId("second").startupOrder(1)
        .to("mock:bar");
}
```

同一路由可以在 XML DSL 中定义，如下所示：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <!-- let this route complete all its pending messages when asked to shut down -->
    <route id="first"
          startupOrder="2"
          shutdownRunningTask="CompleteAllTasks">
        <from uri="file:target/pending"/>
        <delay><constant>1000</constant></delay>
```

```
        <to uri="seda:foo"/>
    </route>

    <route id="second" startupOrder="1">
        <from uri="seda:foo"/>
        <to uri="mock:bar"/>
    </route>
</camelContext>
```

关闭超时

关闭超时的默认值为 300 秒。您可以通过在 shutdown 策略上调用 setTimeout（） 方法来更改超时值。例如，您可以将超时值更改为 600 秒，如下所示：

```
// Java
// context = CamelContext instance
context.getShutdownStrategy().setTimeout(600);
```

与自定义组件集成

如果您正在实施自定义 Apache Camel 组件（也会从 org.apache.camel.Service 接口继承），您可以确保自定义代码通过实施 org.apache.camel.spi.ShutdownPre ed 接口来接收关闭通知。这为组件提供了在准备关闭过程中执行自定义代码的机会。

2.9.1. RouteIdFactory

根据消费者端点，您可以添加 RouteIdFactory，以使用逻辑名称分配路由 ID。

例如，当将路由用于 seda 或 direct 组件作为路由输入时，您可能希望使用其名称作为路由 id，如：

- direct:foo- foo
- seda:bar- bar
- JMS:orders- order

您可以使用 NodeIdFactory 为路由分配逻辑名称，而不是使用自动分配的名称。另外，您可以将上下文路由 URL 用作名称。例如，执行以下命令以使用 RouteIDFactory ：

```
context.setNodeIdFactory(new RouteIdFactory());
```

注意

可以从其他端点获取自定义路由 id。

2.10. 调度的路由策略

2.10.1. Scheduled Route 策略概述

概述

调度的路由策略可用于触发在运行时影响路由的事件。特别是，目前可用的实现可让您在策略指定的任意时间（或时间）启动、停止、挂起或恢复路由。

调度任务

调度的路由策略能够触发以下事件：

- 在指定时间（或时间）时启动路由。如果路由当前处于已停止状态，则会等待激活，此事件才会生效。
- 在指定的时间（或时间）停止路由 InventoryService-操作会停止路由。只有路由当前处于活跃状态时，此事件才起作用。
- 在路由开始时（如 from（）中指定的） 挂起一个路由 时间（如 from））。其余的路由仍然活跃，但客户端将无法将新消息发送到路由。
- 在路由开始时 恢复一个路由 gradle-gradlere-activate 消费者端点，将路由返回到完全活跃状态。

Quartz 组件

Quartz 组件是基于 Terracotta 的 Quartz 的计时器组件，它是作业调度程序的开源实施。Quartz 组件为简单调度的路由策略和 cron 计划路由策略提供了底层实施。

2.10.2. 简单调度的路由策略

概述

简单的调度路由策略是一个路由策略，可让您启动、停止、挂起和恢复路由，其中通过指定一定数量的后续 repititions 定义这些事件的时间和日期（可选）。要定义简单的调度路由策略，请创建以下类实例：

```
org.apache.camel.routepolicy.quartz.SimpleScheduledRoutePolicy
```

依赖项

简单调度的路由策略取决于 Quartz 组件 camel-quartz。例如，如果您使用 Maven 作为构建系统，则需要添加对 camel-quartz 工件的依赖项。

Java DSL 示例

例 2.7 “简单调度路由的 Java DSL 示例” 演示了如何调度路由以使用 Java DSL 启动。初始的开始时间 startTime 定义为当前时间之后的 3 秒。该策略也被配置为在初始开始时间后启动路由，3 秒（通过将 routeStartRepeatCount 设置为 1）并将 routeStartRepeatInterval 设置为 3000 毫秒。

在 Java DSL 中，您可以通过在路由中调用 routePolicy（） DSL 命令将路由策略附加到路由。

例 2.7. 简单调度路由的 Java DSL 示例

```
// Java
SimpleScheduledRoutePolicy policy = new SimpleScheduledRoutePolicy();
long startTime = System.currentTimeMillis() + 3000L;
policy.setRouteStartDate(new Date(startTime));
policy.setRouteStartRepeatCount(1);
policy.setRouteStartRepeatInterval(3000);

from("direct:start")
  .routeId("test")
  .routePolicy(policy)
  .to("mock:success");
```

注意

您可以使用多个参数调用 routePolicy（） 在路由上指定多个策略。

XML DSL 示例

例 2.8 “简单调度路由的 XML DSL 示例” 演示了如何调度路由以使用 XML DSL 启动。

在 XML DSL 中，您可以通过设置路由元素上的 routePolicyRef 属性将路由策略附加到 路由。

例 2.8. 简单调度路由的 XML DSL 示例

```
<bean id="date" class="java.util.Data"/>

<bean id="startPolicy"
class="org.apache.camel.routepolicy.quartz.SimpleScheduledRoutePolicy">
    <property name="routeStartDate" ref="date"/>
    <property name="routeStartRepeatCount" value="1"/>
    <property name="routeStartRepeatInterval" value="3000"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="myroute" routePolicyRef="startPolicy">
        <from uri="direct:start"/>
        <to uri="mock:success"/>
    </route>
</camelContext>
```

注意

您可以通过将 routePolicyRef 的值设置为以逗号分隔的 bean ID 列表，在路由上指定多个策略。

定义日期和时间

使用简单调度路由策略中使用的触发器的初始时间是使用 java.util.Date 类型指定的。定义 Date 实例的最灵活方法是通过 java.util.GregorianCalendar 类。使用 GregorianCalendar 类的方便构建者和方法来定义日期，然后通过调用 GregorianCalendar.getTime（） 获取 日期。

例如，要在 2011 年 1 月 1 日之前定义时间和日期，请致电 GregorianCalendar constructor，如下所示：

```
// Java
import java.util.GregorianCalendar;
import java.util.Calendar;
...
GregorianCalendar gc = new GregorianCalendar(
```

```
    2011,
    Calendar.JANUARY,
    1,
    12,  // hourOfDay
    0,   // minutes
    0    // seconds
);

java.util.Date triggerDate = gc.getTime();
```

GregorianCalendar 类还支持在不同时区中的时间定义。默认情况下，它使用您计算机上的本地时区。

正常关闭

当您将简单调度路由策略配置为停止路由时，路由停止算法会自动与安全关闭过程集成（请参阅 第 2.9 节 “控制路由启动和关闭”）。这意味着该任务会等待当前交换完成处理，然后再关闭路由。但是，您可以设置超时，它会强制路由在指定时间后停止，无论路由是否完成交换过程。

超时记录动态交换

如果安全关闭无法在给定的超时时间内完全关闭，那么 **Apache Camel** 将执行更积极的关机。它强制关闭路由、线程池等。

在超时后，**Apache Camel** 会记录有关当前动态交换的信息。它记录交换的源和当前交换路由。

例如，以下日志显示有一个动态的交换，来自 **route1** 的起源，目前在 **delay1** 节点上的同一 **route1** 上。

在安全关闭过程中，如果您在 **org.apache.camel.impl.DefaultShutdownStrategy** 上启用 **DEBUG** 日志记录级别，则会记录相同的动态交换信息。

```
2015-01-12 13:23:23,656 [- ShutdownTask] INFO DefaultShutdownStrategy - There are 1 inflight exchanges:
InflightExchange: [exchangeId=ID-davsclaus-air-62213-1421065401253-0-3, fromRouteId=route1, routeId=route1, nodeId=delay1, elapsed=2007, duration=2017]
```

如果您不想查看这些日志，可以通过将选项 **logInflightExchangesOnTimeout** 设置为 **false** 来关闭。

```
 context.getShutdownStrategegy().setLogInflightExchangesOnTimeout(false);
```

调度任务

您可以使用简单的调度路由策略来定义一个或多个以下调度任务：

- *启动路由*
- *停止路由*
- *挂起路由*
- *恢复路由*

启动路由

下表列出了调度一个或多个路由启动的参数。

参数	类型	默认值	描述
routeStartDate	**java.util.Date**	**无**	指定首次启动路由时的日期和时间。
routeStartRepeatCount	**int**	**0**	当设置为非零值时，可指定路由应启动的次数。
routeStartRepeatInterval	**long**	**0**	指定启动之间的间隔，以毫秒为单位。

停止路由

下表列出了调度一个或多个路由的参数停止。

参数	类型	默认值	描述
routeStopDate	**java.util.Date**	**无**	指定首次停止路由时的日期和时间。

参数	类型	默认值	描述
routeStopRepeatCount	**int**	**0**	当设置为非零值时，可指定路由应该停止的次数。
routeStopRepeatInterval	**long**	**0**	指定停止之间的时间间隔，单位为毫秒。
routeStopGracePeriod	**int**	**10000**	指定在强制停止路由前等待当前交换完成（宽限期）的时长。在无限宽限期内设置为 0。
routeStopTimeUnit	**long**	**TimeUnit.MILLISECONDS**	指定宽限期的时间范围。

挂起路由

下表列出了用于调度路由启动或多次处理的参数。

参数	类型	默认值	描述
routeSuspendDate	**java.util.Date**	**无**	指定首次暂停路由的时间和时间。
routeSuspendRepeatCount	**int**	0	当设置为非零值时，可指定路由应该暂停的次数。
routeSuspendRepeatInterval	**long**	0	指定挂起之间的时间间隔，单位为毫秒。

恢复路由

下表列出了用于调度一个或多个路由恢复的参数。

参数	类型	默认值	描述
routeResumeDate	**java.util.Date**	**无**	指定首次恢复路由的时间和时间。
routeResumeRepeatCount	**int**	0	当设置为非零值时，指定应恢复路由的次数。

参数	类型	默认值	描述
routeResumeRepeatInterval	long	0	指定恢复间隔，以毫秒为单位。

2.10.3. Cron Scheduled Route 策略

概述

cron 调度路由策略是一个路由策略，可让您启动、停止、挂起和恢复路由，其中使用 cron 表达式指定这些事件的时间。要定义 cron 计划路由策略，请创建以下类实例：

```
org.apache.camel.routepolicy.quartz.CronScheduledRoutePolicy
```

依赖项

简单调度的路由策略取决于 Quartz 组件 camel-quartz。例如，如果您使用 Maven 作为构建系统，则需要添加对 camel-quartz 工件的依赖项。

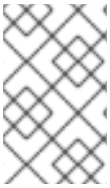
Java DSL 示例

例 2.9 “Cron Scheduled Route 的 Java DSL 示例” 演示了如何调度路由以使用 Java DSL 启动。策略使用 cron 表达式 */3 * * * ? ? 进行配置，这会每 3 秒触发一次启动事件。

在 Java DSL 中，您可以通过在路由中调用 routePolicy（） DSL 命令将路由策略附加到路由。

例 2.9. Cron Scheduled Route 的 Java DSL 示例

```
// Java
CronScheduledRoutePolicy policy = new CronScheduledRoutePolicy();
policy.setRouteStartTime("*/3 * * * * ?");

from("direct:start")
    .routeId("test")
    .routePolicy(policy)
    .to("mock:success");;
```



注意

您可以使用多个参数调用 routePolicy（） 在路由上指定多个策略。

XML DSL 示例

例 2.10 “Cron Scheduled 路由的 XML DSL 示例”演示了如何调度路由以使用 XML DSL 启动。

在 XML DSL 中，您可以通过设置路由元素上的 routePolicyRef 属性将路由策略附加到 路由。

例 2.10. Cron Scheduled 路由的 XML DSL 示例

```
<bean id="date" class="org.apache.camel.routepolicy.quartz.SimpleDate"/>

<bean id="startPolicy" class="org.apache.camel.routepolicy.quartz.CronScheduledRoutePolicy">
    <property name="routeStartTime" value="*/3 * * * * ?"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="testRoute" routePolicyRef="startPolicy">
        <from uri="direct:start"/>
        <to uri="mock:success"/>
    </route>
</camelContext>
```

注意

您可以通过将 routePolicyRef 的值设置为以逗号分隔的 bean ID 列表，在路由上指定多个策略。

定义 cron 表达式

cron 表达式语法 在 UNIX cron 实用程序中包含其原始内容，可调度在 UNIX 系统上的后台运行的作业。cron 表达式有效了用于通配符日期和时间的语法，允许您指定单个事件或定期重复发生的多个事件。

cron 表达式按以下顺序由 6 个或 7 个字段组成：

```
Seconds Minutes Hours DayOfMonth Month DayOfWeek [Year]
```

Year 字段是可选的，通常省略，除非您要定义一次发生一次的事件。每个字段由字面和特殊字符的组合组成。例如，以下 cron 表达式指定每天每天触发一次事件：

```
0 0 24 * * ?
```

** 字符是一个通配符，与字段的每个值匹配。因此，前面的表达式与每月的每天匹配。？字符是一个 dummy 占位符，这意味着 *ignore this field*。它始终出现在 DayOfMonth 字段中，或者在 DayOfWeek 字段中出现，因为它并不完全一致地指定这两个字段。例如，如果要调度一个每天触发一次的事件，但仅在周一到周五，请使用以下 cron 表达式：*

```
0 0 24 ? * MON-FRI
```

其中连字符指定范围 MON-FRI。您还可以使用正斜杠字符 / 来指定增量。例如，要指定事件每 5 分钟触发一次，请使用以下 cron 表达式：

```
0 0/5 * * * ?
```

有关 cron 表达式语法的完整解释，请参阅 CRON 表达式 中的 Wikipedia。

调度任务

您可以使用 cron 计划路由策略定义以下一个或多个调度任务：

- *启动路由*
- *停止路由*
- *挂起路由*
- *恢复路由*

启动路由

下表列出了调度一个或多个路由启动的参数。

参数	类型	默认值	描述
routeStartString	**字符串**	**无**	指定触发一个或多个路由启动事件的 cron 表达式。

停止路由

下表列出了调度一个或多个路由的参数停止。

参数	类型	默认值	描述
routeStopTime	**字符串**	**无**	指定触发一个或多个路由停止事件的 cron 表达式。
routeStopGracePeriod	**int**	**10000**	指定在强制停止路由前等待当前交换完成（宽限期）的时长。在无限宽限期内设置为 0。
routeStopTimeUnit	**long**	**TimeUnit.MILLISECONDS**	指定宽限期的时间范围。

挂起路由

下表列出了用于调度路由启动或多次处理的参数。

参数	类型	默认值	描述
routeSuspendTime	**字符串**	**无**	指定触发一个或多个路由暂停事件的 cron 表达式。

恢复路由

下表列出了用于调度一个或多个路由恢复的参数。

参数	类型	默认值	描述
routeResumeTime	**字符串**	**无**	指定触发一个或多个路由恢复事件的 cron 表达式。

2.10.4. 路由策略因素

使用路由策略 onnectionFactoryy

Camel 2.14 已提供

如果要为每个路由使用路由策略，您可以使用 org.apache.camel.spi.RoutePolicy factory 作为每个路由创建 RoutePolicy 实例的工厂。当您想为每个路由使用相同路由策略时，可以使用它。然后，您只需要配置一次工厂，并且每个创建的路由都会分配策略。

CamelContext 上的 API 来添加工厂，如下所示：

```
context.addRoutePolicyFactory(new MyRoutePolicyFactory());
```

从 XML DSL 中，您只能使用 factory 定义 <bean >

```
<bean id="myRoutePolicyFactory" class="com.foo.MyRoutePolicyFactory"/>
```

factory 包含用于创建路由策略的 createRoutePolicy 方法。

```
/**
 * Creates a new {@link org.apache.camel.spi.RoutePolicy} which will be assigned to the given route.
 *
 * @param camelContext the camel context
 * @param routeId      the route id
 * @param route        the route definition
 * @return the created {@link org.apache.camel.spi.RoutePolicy}, or <tt>null</tt> to not use a policy
for this route
 */
RoutePolicy createRoutePolicy(CamelContext camelContext, String routeId, RouteDefinition route);
```

请注意，您可以尽可能多地发生路由策略因素。只需要再次调用 addRoutePolicyFactory，或者把其他因素声明为 XML 中的 &lt ;bean& gt;。

2.11. 重新加载 CAMEL 路由

在 Apache Camel 2.19 发行版本中，您可以启用对 camel XML 路由的实时重新加载，该路由会在从编辑器保存 XML 文件时触发重新加载。在以下情况下使用此功能：

- *带有 Camel 主类的 Camel 独立（Camel 主类）*

- Camel Spring Boot
- 来自 camel:run maven 插件

但是，您还可以通过在 CamelContext 中设置 ReloadStrategy 并通过提供您自己的自定义策略来手动启用此功能。

2.12. CAMEL MAVEN 插件

Camel Maven 插件支持以下目标：

- Camel:run - 来运行 Camel 应用程序
- Camel:validate - 验证无效的 Camel 端点 URI 的源代码
- Camel:route-coverage - 在单元测试后报告您的 Camel 路由覆盖情况

2.12.1. Camel:run

Camel Maven 插件的 camel:run 目标用于在从 Maven 派生的 JVM 中运行您的 Camel Spring 配置。您启动的好示例应用程序是 Spring 示例。

```
cd examples/camel-example-spring
mvn camel:run
```

这样，可以在无需编写 main(...)方法的情况下轻松启动和测试您的路由规则；它还可让您创建多个 jar 来托管不同的路由规则，并轻松地独立测试它们。Camel Maven 插件编译了 maven 项目中的源代码，然后使用 META-INF/spring/*.xml 类路径上的 XML 配置文件引导 Spring ApplicationContext。如果要更快地引导 Camel 路由，您可以尝试 camel:embeeded。

2.12.1.1. 选项

Camel Maven 插件 运行 目标支持以下选项，它们可从命令行配置（使用 -D 语法），或在 < configuration > 标签中的 pom.xml 文件中定义。

参数	默认值	描述
duration	-1	设置应用程序在终止前所运行的时间持续时间（秒）。系统会永久运行值 InventoryService 0。
durationIdle	-1	设置应用程序在终止前可以闲置的空闲时间（秒）持续时间。系统会永久运行值 InventoryService 0。
durationMaxMessages	-1	设置应用程序进程在终止前的最大消息的持续时间。
logClasspath	false	是否在启动时记录类路径

2.12.1.2. 运行 OSGi 蓝图

camel:run 插件还支持运行 Blueprint 应用程序，默认情况下，它会在 OSGI-INF/blueprint/*.xml 中扫描 OSGi 蓝图文件。您需要将 camel:run 插件配置为使用蓝图，方法是将 useBlueprint 设置为 true，如下所示：

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <configuration>
    <useBlueprint>true</useBlueprint>
  </configuration>
</plugin>
```

这可让您引导所有您想要的蓝图服务，无论它们是否与 Camel 相关，还是其它蓝图。camel:run 目标可自动检测 camel-blueprint 位于类路径中，或者项目中是否存在蓝图 XML 文件，因此您不必配置 useBlueprint 选项。

2.12.1.3. 使用有限的蓝图容器

我们使用 Felix Connector 项目作为蓝图容器。此项目不是完整的蓝图容器。为此，您可以使用 Apache Karaf 或 Apache ServiceMix。您可以使用 应用程序ContextUri 配置指定显式蓝图 XML 文件，例如：

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <configuration>
    <useBlueprint>true</useBlueprint>
    <applicationContextUri>myBlueprint.xml</applicationContextUri>
    <!-- ConfigAdmin options which have been added since Camel 2.12.0 -->
```

```
    <configAdminPid>test</configAdminPid>
    <configAdminFileName>/user/test/etc/test.cfg</configAdminFileName>
  </configuration>
</plugin>
```

应用程序ContextUri 从 classpath 加载文件，因此在 myBlueprint.xml 文件上方的示例中必须位于 classpath 的根目录中。configAdminPid 是 pid 名称，在加载持久性属性文件时将用作配置 admin 服务的 pid 名称。configAdminFileName 是用于加载配置 admin 服务属性文件的文件名。

2.12.1.4. 运行 CDI

camel:run 插件还支持运行 CDI 应用。这可让您引导所有您想要的 CDI 服务，无论它们是与 Camel 相关的，还是任何其他 CDI 启用的服务。您应该将您选择的 CDI 容器（如 Weld 或 OpenWebBeans）添加到 camel-maven-plugin 的依赖关系，如本例中所示。在 Camel 的源中，您可以运行 CDI 示例，如下所示：

```
cd examples/camel-example-cdi
mvn compile camel:run
```

2.12.1.5. 记录 classpath

您可以配置当 camel:run 执行时，是否应该记录 classpath。您可以使用以下方法在配置中启用此功能：

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <configuration>
    <logClasspath>true</logClasspath>
  </configuration>
</plugin>
```

2.12.1.6. 使用 XML 文件实时重新载入

您可以配置插件来扫描 XML 文件更改，并触发在这些 XML 文件中包含的 Camel 路由重新载入。

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <configuration>
    <fileWatcherDirectory>src/main/resources/META-INF/spring</fileWatcherDirectory>
  </configuration>
</plugin>
```

然后，插件会监视此目录。这可让您从编辑器编辑源代码并保存文件，并使正在运行的 Camel 应用程序使用这些更改。请注意，只有 Camel 路由的更改，如 < routes> 或支持 & lt;route >。您无法更改 Spring 或 OSGi Blueprint < bean> 元素。

2.12.2. camel:validate

对于以下 Camel 功能的源代码验证：

- 端点 URI
- 简单表达式或 predicates
- 重复的路由 ID

然后，您可以从命令行运行 camel:validate 目标，或者从 IDEA 或 Eclipse 等 Java 编辑器中运行。

```
mvn camel:validate
```

您还可以启用插件作为构建的一部分自动运行，以捕获这些错误。

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <executions>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

该阶段决定了插件何时运行。在上面的示例中，阶段是 进程类，它在主源代码的编译后运行。也可将 maven 插件配置为验证测试源代码，这意味着应相应地将阶段更改为 process-test-classes，如下所示：

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>camel-maven-plugin</artifactId>
```

```
  <executions>
    <execution>
      <configuration>
        <includeTest>true</includeTest>
      </configuration>
      <phase>process-test-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

2.12.2.1. 在任何 Maven 项目上运行目标

您还可以在任何 Maven 项目中运行验证目标，而无需将插件添加到 pom.xml 文件中。这样做需要使用其完全限定名称指定插件。例如，要在来自 Apache Camel 的 camel-example-cdi 上运行目标，您可以运行

```
$cd camel-example-cdi
$mvn org.apache.camel:camel-maven-plugin:2.20.0:validate
```

然后，运行和输出以下内容：

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Camel :: Example :: CDI 2.20.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- camel-maven-plugin:2.20.0:validate (default-cli) @ camel-example-cdi ---
[INFO] Endpoint validation success: (4 = passed, 0 = invalid, 0 = incapable, 0 = unknown
components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

通过验证验证，4 个端点已被验证。现在，假设我们在源代码中的一个 Camel 端点 URI 中有拼写错误，例如：

```
@Uri("timer:foo?period=5000")
```

被修改为在 period 选项中包含拼写错误

```
@Uri("timer:foo?perid=5000")
```

在运行验证目标时，会再次报告以下内容：

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Camel :: Example :: CDI 2.20.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- camel-maven-plugin:2.20.0:validate (default-cli) @ camel-example-cdi ---
[WARNING] Endpoint validation error at:
org.apache.camel.example.cdi.MyRoutes(MyRoutes.java:32)

 timer:foo?perid=5000

                perid    Unknown option. Did you mean: [period]


[WARNING] Endpoint validation error: (3 = passed, 1 = invalid, 0 = incapable, 0 = unknown
components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

2.12.2.2. 选项

Camel Maven 插件 验证 目标支持以下选项，它们可从命令行配置（使用 -D 语法），或在 < configuration > 标签中的 pom.xml 文件中定义。

参数	默认值	描述
downloadVersion	true	是否允许从互联网下载 Camel 目录版本。项目默认使用不同于此插件的 Camel 版本，则需要此项。
failOnError	false	如果找到无效的 Camel 端点，是否失败。默认情况下，插件会在 WARN 级别上记录错误。
logUnparseable	false	是否记录不可解析的端点 URI，因此无法验证。
includeJava	true	是否包含用于无效的 Camel 端点验证的 Java 文件。
includeXml	true	是否要包含要针对无效 Camel 端点验证的 XML 文件。
includeTest	false	是否包括测试源代码。

includes		要过滤 java 和 xml 文件的名称，使其仅包含与任何给定模式列表匹配的文件（通配符和正则表达式）。可以使用逗号分隔多个值。
excludes		要过滤 java 和 xml 文件的名称，以排除与任何给定模式列表匹配的文件（通配符和正则表达式）。可以使用逗号分隔多个值。
ignoreUnknownComponent	true	是否忽略未知组件。
ignoreIncapable	true	是否忽略是否能够解析端点 URI 还是简单表达式。
ignoreLenientProperties	true	是否忽略使用 lenient 属性的组件。当此参数为 true 时，如果 URI 验证比较严格，但在 URI 中因为使用 lenient 属性而在 URI 中会失败。例如，使用 HTTP 组件在端点 URI 中提供查询参数。
ignoreDeprecated	true	Camel 2.23 是否忽略端点 URI 中使用的已弃用的选项。
duplicateRouteId	true	Camel 2.20 是否验证重复的路由 ID。路由 ID 应该是唯一的，如果有重复，Camel 将无法启动。
directOrSedaPairCheck	true	Camel 2.23 是否验证发送到非现有消费者的直接/横线端点。
showAll	false	是否显示所有端点和简单表达式（无效和有效）。

例如，要禁用在命令行中弃用的选项的使用，您可以运行：

```
$mvn camel:validate -Dcamel.ignoreDeprecated=false
```

请注意，您必须将 -D 命令参数前缀为 camel。,eg camel. ignore Deprecated 作为选项名称。

2.12.2.3. 使用验证端点包括测试

如果您有 Maven 项目，您可以运行插件来验证单元测试源代码中的端点。您可以使用 -D 风格传递选项，如下所示：

```
$cd myproject
$mvn org.apache.camel:camel-maven-plugin:2.20.0:validate -DincludeTest=true
```

2.12.3. camel:route-coverage

从单元测试开始生成 Camel 路由覆盖报告。您可以使用它来了解使用 Camel 路由哪些部分。

2.12.3.1. 启用路由覆盖范围

您可以通过以下方式在运行单元测试时启用路由覆盖范围：

- 为所有测试类设置全局 JVM 系统属性
- 如果使用 camel-test-spring 模块，在每个测试类使用 @EnableRouteCoverage 注解
- 如果使用 camel-test 模块，覆盖为每个测试类的 DumpRouteCoverage 方法

2.12.3.2. 使用 JVM 系统属性启用路由覆盖范围

您可以开启 JVM 系统属性 CamelTestRouteCoverage，以启用所有测试案例的路由覆盖范围。这可以在 maven-surefire-plugin 配置中完成：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <systemPropertyVariables>
      <CamelTestRouteCoverage>true</CamelTestRouteCoverage>
    </systemPropertyVariables>
  </configuration>
</plugin>
```

或者在运行测试时从命令行：

```
mvn clean test -DCamelTestRouteCoverage=true
```

2.12.3.3. 通过 @EnableRouteCoverage 注解启用

如果您使用 camel-test-spring 进行测试，您可以在测试类中添加 @EnableRouteCoverage 注解到测试类来启用路由覆盖：

```
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = SampleCamelApplication.class)
@EnableRouteCoverage
public class FooApplicationTest {
```

2.12.3.4. 通过DumpRouteCoverage 方法启用

但是，如果您使用 camel-test，并且您的单元测试正在扩展 CamelTestSupport，那么您可以启用路由覆盖范围，如下所示：

```
@Override
public boolean isDumpRouteCoverage() {
    return true;
}
```

可在 RouteCoverage 方法中覆盖的路由必须具有一个唯一的 id，否则您不能使用匿名路由。您可以在 Java DSL 中使用 routeId 来完成此操作：

```
from("jms:queue:cheese").routeId("cheesy")
  .to("log:foo")
  ...
```

在 XML DSL 中，您刚刚通过 id 属性分配路由 id

```
<route id="cheesy">
  <from uri="jms:queue:cheese"/>
  <to uri="log:foo"/>
  ...
</route>
```

2.12.3.5. 生成路由覆盖报告

要生成路由覆盖范围报告，使用以下命令运行单元测试：

```
mvn test
```

然后，您可以运行该目标来报告路由覆盖范围，如下所示：

```
mvn camel:route-coverage
```

报告哪个路由缺少路由覆盖率，并提供精确的源代码行报告：

```
[INFO] --- camel-maven-plugin:2.21.0:route-coverage (default-cli) @ camel-example-spring-boot-xml
---
[INFO] Discovered 1 routes
[INFO] Route coverage summary:

File: src/main/resources/my-camel.xml
RouteId: hello

  Line #      Count   Route
  ------      -----   -----
     28          1   from
     29          1     transform
     32          1     filter
     34          0       to
     36          1     to

Coverage: 4 out of 5 (80.0%)
```

在这里，我们可以看到，使用 的第二代行在 count 栏中具有 0， 因此没有涵盖。我们还可以在源代码文件中看到，这是一行 34，它位于 my-camel.xml XML 文件中。

2.12.3.6. 选项

Camel Maven 插件 覆盖 目标支持以下选项，它们可从命令行配置（使用 -D 语法），或在 < configuration > 标签中的 pom.xml 文件中定义。

参数	默认值	描述
failOnError	false	如果任何路由没有 100% 覆盖，则是否失败。
includeTest	false	是否包括测试源代码。
includes		要过滤 java 和 xml 文件的名称，使其仅包含与任何给定模式列表匹配的文件（通配符和正则表达式）。可以使用逗号分隔多个值。
excludes		要过滤 java 和 xml 文件的名称，以排除与任何给定模式列表匹配的文件（通配符和正则表达式）。可以使用逗号分隔多个值。

anonymousRoutes	false	是否允许匿名路由（没有分配任何路由 id 的路由）。通过使用路由 ID，其安全器与路由源代码中的数据匹配。匿名路由不安全地使用路由覆盖范围，因为它更难以知道哪个路由被测试与源代码中哪个路由对应。

2.13. 运行 APACHE CAMEL STANDALONE

当您作为独立应用程序运行 camel 时，它提供运行应用程序的主类，并保持其运行，直到 JVM 终止为止。您可以在 org.apache.camel.main Java 包中找到 MainListener 类。

以下是 Main 类的组件：

- ***org.apache. camel.Main 类中的 Camel-core JAR***
- ***org.apache. camel.spring.Main 类中的 Camel- spring JAR***

以下示例演示了如何从 Camel 中创建和使用 Main 类：

```
public class MainExample {

    private Main main;

    public static void main(String[] args) throws Exception {
        MainExample example = new MainExample();
        example.boot();
    }

    public void boot() throws Exception {
        // create a Main instance
        main = new Main();
        // bind MyBean into the registry
        main.bind("foo", new MyBean());
        // add routes
        main.addRouteBuilder(new MyRouteBuilder());
        // add event listener
        main.addMainListener(new Events());
        // set the properties from a file
        main.setPropertyPlaceholderLocations("example.properties");
        // run until you terminate the JVM
        System.out.println("Starting Camel. Use ctrl + c to terminate the JVM.\n");
        main.run();
```

```
    }

    private static class MyRouteBuilder extends RouteBuilder {
        @Override
        public void configure() throws Exception {
            from("timer:foo?delay={{millisecs}}")
                .process(new Processor() {
                    public void process(Exchange exchange) throws Exception {
                        System.out.println("Invoked timer at " + new Date());
                    }
                })
                .bean("foo");
        }
    }

    public static class MyBean {
        public void callMe() {
            System.out.println("MyBean.callMe method has been called");
        }
    }

    public static class Events extends MainListenerSupport {

        @Override
        public void afterStart(MainSupport main) {
            System.out.println("MainExample with Camel is now started!");
        }

        @Override
        public void beforeStop(MainSupport main) {
            System.out.println("MainExample with Camel is now being stopped!");
        }
    }
}
```

2.14. ONCOMPLETION

概述

OnCompletion DSL 名称用于定义在完成工作单元时要执行的操作。工作单元是包含整个交换的 Camel 概念。请参阅 第 34.1 节 “交换”。onCompletion 命令有以下功能：

- **OnCompletion 命令的范围可以是全局或每个路由。路由范围会覆盖全局范围。**
- **OnCompletion 可以配置为在成功时触发失败。**
- **onWhen predicate 只能在某些情况下触发该 onCompletion。**

- 您可以定义是否要使用线程池，但默认值为线程池。

在编译时只有路由范围

当在交换中指定Completion DSL 时，Camel 启动一个新的线程。这允许原始线程在不 与 Completion 任务进行干预的情况下继续。路由只支持一个 在Completion 中。在以下示例中，无论交换成功完成是成功还是失败，会触发 Completion。这是默认的操作。

```
from("direct:start")
    .onCompletion()
        // This route is invoked when the original route is complete.
        // This is similar to a completion callback.
        .to("log:sync")
        .to("mock:sync")
    // Must use end to denote the end of the onCompletion route.
    .end()
    // here the original route contiues
    .process(new MyProcessor())
    .to("mock:result");
```

对于 XML，格式如下：

```
<route>
   <from uri="direct:start"/>
   <!-- This onCompletion block is executed when the exchange is done being routed. -->
   <!-- This callback is always triggered even if the exchange fails. -->
   <onCompletion>
      <!-- This is similar to an after completion callback. -->
      <to uri="log:sync"/>
      <to uri="mock:sync"/>
   </onCompletion>
   <process ref="myProcessor"/>
   <to uri="mock:result"/>
</route>
```

要在失败时触发 Completion，可以使用 onFailureOnly 参数。同样，若要成功触发 Completion，请使用 onCompleteOnly 参数。

```
from("direct:start")
    // Here onCompletion is qualified to invoke only when the exchange fails (exception or FAULT
body).
    .onCompletion().onFailureOnly()
        .to("log:sync")
        .to("mock:sync")
    // Must use end to denote the end of the onCompletion route.
    .end()
```

```
    // here the original route continues
    .process(new MyProcessor())
    .to("mock:result");
```

对于 **XML，在 Completion tag 中将 onFailureOnly 和 onCompleteOnly 来表示：**

```
<route>
   <from uri="direct:start"/>
   <!-- this onCompletion block will only be executed when the exchange is done being routed -->
   <!-- this callback is only triggered when the exchange failed, as we have onFailure=true -->
   <onCompletion onFailureOnly="true">
      <to uri="log:sync"/>
      <to uri="mock:sync"/>
   </onCompletion>
   <process ref="myProcessor"/>
   <to uri="mock:result"/>
</route>
```

Completion 全局范围

为多个路由定义 **Completion ：**

```
// define a global on completion that is invoked when the exchange is complete
 onCompletion().to("log:global").to("mock:sync");

 from("direct:start")
    .process(new MyProcessor())
    .to("mock:result");
```

使用 OnWhen

要在某些情况下触发 Completion，请使用 onWhen predicate。当消息正文包含单词 Hello 时，以下示例会触发 onCompletion ：

```
/from("direct:start")
    .onCompletion().onWhen(body().contains("Hello"))
       // this route is only invoked when the original route is complete as a kind
       // of completion callback. And also only if the onWhen predicate is true
       .to("log:sync")
       .to("mock:sync")
    // must use end to denote the end of the onCompletion route
    .end()
    // here the original route contiues
    .to("log:original")
    .to("mock:result");
```

使用带有或没有线程池的处理

自 Camel 2.14 起，Completion 默认不使用线程池。要强制使用线程池，请将 executorService 设置为 true，或者将 parallelProcessing 设为 true。例如，在 Java DSL 中，使用以下格式：

```
onCompletion().parallelProcessing()
    .to("mock:before")
    .delay(1000)
    .setBody(simple("OnComplete:${body}"));
```

对于 XML，格式为：

```
<onCompletion parallelProcessing="true">
  <to uri="before"/>
  <delay><constant>1000</constant></delay>
  <setBody><simple>OnComplete:${body}<simple></setBody>
 </onCompletion>
```

使用 executorServiceRef 选项引用特定的线程池：

```
<onCompletion executorServiceRef="myThreadPool"
  <to uri="before"/>
  <delay><constant>1000</constant></delay>
  <setBody><simple>OnComplete:${body}</simple></setBody>
 </onCompletion>>
```

在 Consumer Sends Response 之前在Completion 上运行

在Completion 中有两种模式运行：

- AfterConsumer - 在消费者完成后运行的默认模式
- BeforeConsumer - 在消费者写出请求之前运行 给调用者。这可以在 编译时 修改 Exchange，如添加特殊标头，或将 Exchange 记录为响应日志记录器。

例如，要将标头 创建的 添加到响应中，请使用 modeBeforeConsumer（），如下所示：

```
.onCompletion().modeBeforeConsumer()
    .setHeader("createdBy", constant("Someone"))
 .end()
```

对于 XML，将 mode 属性设置为 BeforeConsumer ：

```
<onCompletion mode="BeforeConsumer">
  <setHeader headerName="createdBy">
    <constant>Someone</constant>
  </setHeader>
 </onCompletion>
```

2.15. 指标

概述

Camel 2.14 已提供

虽然 Camel 提供了很多现有指标集成，但为 Camel 路由添加了 Codahale 指标。这使得最终用户能够使用 Codahale 指标将 Camel 路由信息与所收集的现有数据连接在一起。

要使用 Codahale 指标，您需要：

1. 添加 camel-metrics 组件
2. 在 XML 或 Java 代码中启用路由指标

请注意，性能指标只在显示它们时可用；任何类型的监控工具都可以与 JMX 集成，因为指标可以通过 JMX 获得。此外，实际数据是 100% Codehale JSON。

指标路由策略

可以根据各个路由定义 MetricsRoutePolicy，获取单个路由的 Codahale 指标。

从 Java 创建 MetricsRoutePolicy 实例，作为路由的策略分配。这如下所示：

```
from("file:src/data?noop=true").routePolicy(new MetricsRoutePolicy()).to("jms:incomingOrders");
```

从 XML DSL，您将定义一个 < bean> 指定一个 <bean>，它指定为路由的策略；例如：

```
<bean id="policy" class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicy"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route routePolicyRef="policy">
    <from uri="file:src/data?noop=true"/>
[...]
```

指标路由策略 onnectionFactoryy

此工厂允许每个路由添加一个 RoutePolicy，该路由利用 Codahale 指标公开路由利用率统计信息。该工厂可以在 Java 和 XML 中使用，如下例所示。

从 Java 中，您刚刚将工厂添加到 CamelContext 中，如下所示：

```
context.addRoutePolicyFactory(new MetricsRoutePolicyFactory());
```

从 XML DSL 中，您可以按照如下所示定义 & lt;bean& gt;：

```
<!-- use camel-metrics route policy to gather metrics for all routes -->
<bean id="metricsRoutePolicyFactory"
class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

从 Java 代码中，您可以从 org.apache .camel.component.metrics.MetricsRegistry 获取 com.codahale.metrics.metrics.routepolicy.MetricsRegistryService，如下所示：

```
MetricRegistryService registryService = context.hasService(MetricsRegistryService.class);
if (registryService != null) {
  MetricsRegistry registry = registryService.getMetricsRegistry();
  ...
}
```

选项

MetricsRoutePolicyFactory 和 MetricsRoutePolicy 支持以下选项：

名称	default	描述

durationUnit	**TimeUnit.MILLISECONDS**	在指标报告器或转储统计 json 中用于持续时间的单元。
jmxDomain	**org.apache.camel.metrics**	JXM 域名。
metricsRegistry		allow 使用共享的 **com.codahale.metrics.MetricRegistry**。如果没有提供，则 Camel 将创建此 CamelContext 使用的共享实例。
prettyPrint	**false**	以 json 格式输出统计时是否使用完整的打印。
rateUnit	**TimeUnit.SECONDS**	指标报告器或转储统计为 json 的比率的单元。
useJmx	**false**	是否使用 **com.codahale.metrics.JmxReporter** 将精细统计报告为 JMX。 请注意，如果在 CamelContext 上启用了 JMX，那么在 JMX 树中的服务类型下列出了 **MetricsRegistryService** mbean。该 mbean 有单一操作，可使用 json 输出统计信息。只有在您希望每个统计类型进行精细 mbeans 时，才需要使用Jmx 设置为 true。

2.16. JMX 命名

概述

通过为它定义管理名称，Apache Camel 允许您自定义 CamelContext 的名称 （其出现在 JMX 中）。例如，您可以自定义 XML CamelContext 实例的名称模式，如下所示：

```
<camelContext id="myCamel" managementNamePattern="#name#">
  ...
</camelContext>
```

如果您没有明确为 CamelContext an 设置名称模式，Apache Camel 将恢复为默认的命名策略。

默认命名策略

默认情况下，在 OSGi 捆绑包中部署的 CamelContext 名称等于该捆绑包的 OSGi 符号链接名称。例如，如果 OSGi 符号链接名称为 MyCamelBundle，则 JMX 名称是 MyCamelBundle。如果捆绑包中有多个 CamelContext，则将计数器值添加为后缀，则 JMX 名称不会被混淆。例如，如果 MyCamelBundle 捆绑包中有多个 Camel 上下文，则对应的 JMX MBeans 命名如下：

```
MyCamelBundle-1
MyCamelBundle-2
MyCamelBundle-3
...
```

自定义 JMX 命名策略

默认命名策略的一个缺陷是，您不能保证给定 CamelContext an 在运行之间具有相同的 JMX 名称。如果您想在不同运行之间具有更高的一致性，您可以通过为 CamelContext 实例定义 JMX name 模式，以更精确地控制 JMX 名称。

在 Java 中指定名称模式

要在 Java 中为 CamelContext 指定名称模式，请调用 setNamePattern 方法，如下所示：

```
// Java
context.getManagementNameStrategy().setNamePattern("#name#");
```

在 XML 中指定名称模式

要在 XML 中为 CamelContext 指定名称模式，请在 camelContext 元素上设置 managementNamePattern 属性，如下所示：

```
<camelContext id="myCamel" managementNamePattern="#name#">
```

名称模式令牌

您可以通过将字面文本和以下令牌的任意令牌组合来构建 JMX 名称模式：

表 2.11. JMX Name Pattern Tokens

令牌	描述
#camelId#	**CamelContext** Bean 的 **id** 属性的值。
#name#	与 **#camelId#**.

令牌	描述
#counter#	递增计数器（从 **1**开始）。
#bundleId#	OSGi 部署捆绑包的 ID （仅限 OSGi）
#symbolicName#	OSGi 符号链接名称 （仅限 OSGi）
#version#	OSGi 捆绑版本 （仅限 OSGi）

例子

以下是您可以使用支持的令牌定义的 JMX 名称模式的一些示例：

```
<camelContext id="fooContext" managementNamePattern="FooApplication-#name#">
    ...
</camelContext>
<camelContext id="myCamel" managementNamePattern="#bundleID#-#symbolicName#-#name#">
    ...
</camelContext>
```

模糊名称

由于自定义命名模式会覆盖默认的命名策略，因此可以使用此方法定义模糊的 JMX MBean 名称。例如：

```
<camelContext id="foo" managementNamePattern="SameOldSameOld"> ... </camelContext>
...
<camelContext id="bar" managementNamePattern="SameOldSameOld"> ... </camelContext>
```

在这种情况下，Apache Camel 将在启动时失败，并报告 MBean 已经存在 异常。因此，您应该要格外小心，确保您没有定义模糊的名称模式。

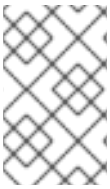
2.17. 性能和优化

消息复制

allowUseOriginalMessage 选项默认设置为 false，以便在不需要时到达原始消息的副本。要启用 allowUseOriginalMessage 选项，请使用以下命令：

- 对任何错误处理程序或 onException 元素设置 useOriginalMessage=true。

- 在 Java 应用代码中，设置 AllowUseOriginalMessage=true，然后使用 getOriginalMessage 方法。



注意

在 2.18 之前的 Camel 版本中，allowUseOriginalMessage 的默认设置为 true。

第 3 章 企业级集成模式简介

摘要

Apache Camel 的企业集成模式通过 Gregor Hohpe 和 Bobby Woolf 编写的书写。这些作者描述的模式为开发企业集成项目提供了一个卓越的 toolbox。除了提供讨论集成架构的通用语言外，许多模式可直接使用 Apache Camel 的编程界面和 XML 配置来实施。

3.1. PATTERNS 概述

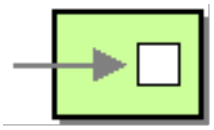
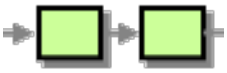
企业级集成模式书

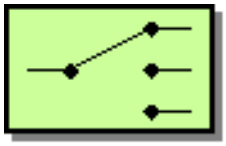
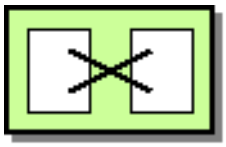
Apache Camel 支持书本中的大多数模式，即 Gregor Hohpe 和 Bobby Woolf 的 Enterprise Integration Patterns。

消息传递系统

表 3.1 “消息传递系统” 中显示的消息传递系统模式介绍了组成消息传递系统的基本概念和组件。

表 3.1. 消息传递系统

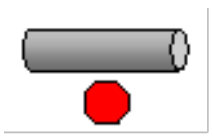
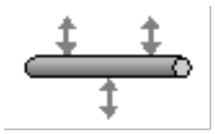
图标	名称	使用案例
	图 5.1 “消息模式”	邮件通道连接的两个应用程序如何交换信息？
	图 5.2 “Message Channel Pattern”	一个应用程序如何使用消息传递与另一个应用程序通信？
	图 5.3 “Message Endpoint Pattern”	应用程序如何连接到消息传递频道以发送和接收消息？
	图 5.4 “pipes 和 Filters Pattern”	我们可以如何对消息执行复杂的处理，同时仍然保持独立性和灵活性？

图标	名称	使用案例
	图 5.7 “Message Router Pattern”	您可以如何分离单独的处理步骤，以便可以根据一组定义的条件将消息传递给不同的过滤器？
	图 5.8 “Message Translator Pattern”	如何使用不同数据格式的系统使用消息传递相互通信？

消息传递频道

消息传递频道是连接消息传递系统中的参与者的基本组件。表 3.2 “消息传递频道” 中的模式描述了可用的不同类型的消息频道。

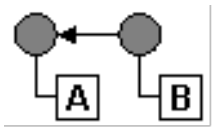
表 3.2. 消息传递频道

图标	名称	使用案例
	图 6.1 “指向点频道模式”	调用者如何确保精确的一个接收器接收文档或执行调用？
	图 6.2 “发布 Subscribe Channel Pattern”	发件人如何将事件广播到所有感兴趣的接收器？
	图 6.3 “死的频道模式”	消息传递系统与无法发送的消息有什么作用？
	图 6.4 “保证交付模式”	发件人如何确保要发送消息，即使消息传递系统失败也是如此？
	图 6.5 “消息总线模式”	哪个架构使分离应用程序可以协同工作，以便在不影响其他应用程序的情况下添加或删除一个或多个应用程序？

消息构建

消息构建模式（如 表 3.3 “消息构建” 所示）描述了通过系统传递的信息的各种形式和功能。

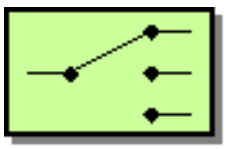
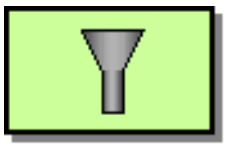
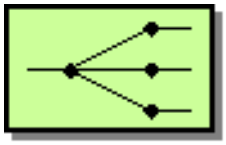
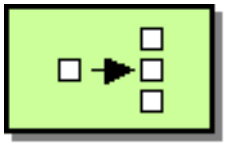
表 3.3. 消息构建

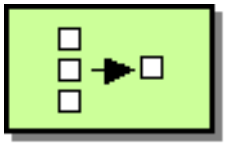
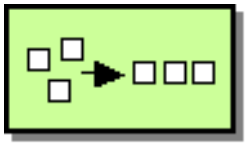
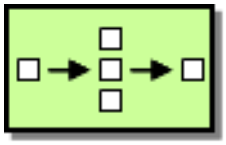
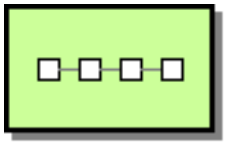
图标	名称	使用案例
	“概述”一节	请求者如何识别生成接收回复的请求？
	第 7.3 节 “返回地址”	replier 如何知道回复在哪里？

消息路由

消息路由模式（在 表 3.4 “消息路由” 所示）描述了将消息通道连接各种方法，包括可应用于消息流的各种算法（无需修改消息正文）。

表 3.4. 消息路由

图标	名称	使用案例
	第 8.1 节 “基于内容的路由器”	我们如何处理单一逻辑功能的实施（例如清单检查）到多个物理系统时？
	第 8.2 节 “Message Filter”	组件如何避免接收不必要的消息？
	第 8.3 节 “接收者列表”	如何将消息路由到动态指定收件人列表？
	第 8.4 节 “Splitter”	如果消息包含多个元素，则我们可以处理一条消息，其中每个元素可能需要以不同的方式进行处理？

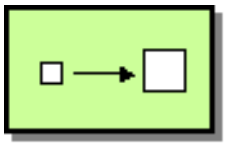
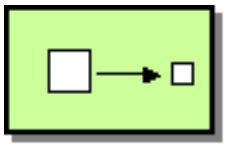
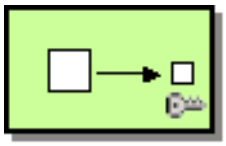
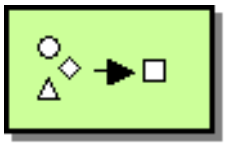
图标	名称	使用案例
	第 8.5 节 “聚合器”	我们如何组合单个结果，但相关消息以便可以作为一个整体进行处理？
	第 8.6 节 “Resequencer”	我们可以如何把相关的消息流返回至正确的顺序？
	第 8.14 节 “由消息处理器”	在处理由多个元素组成的消息时，您可以维护整个消息流，每个项可能需要不同的处理？
	第 8.15 节 “scatter-Gather”	当您需要向多个接收者发送消息时，如何维护整个消息流，每个都可能会发送回复？
	第 8.7 节 “路由 Slip”	在设计时不知道步骤时，我们如何连续通过一系列处理步骤来连续路由消息，每种消息可能会有所不同？
	第 8.8 节 “Throttler”	如何节流消息以确保特定端点不会超载，或者我们没有超出一些外部服务的 SLA？
	第 8.9 节 “Delayer”	如何延迟消息发送？
	第 8.10 节 “Load Balancer”	如何在多个端点之间平衡负载？
	第 8.11 节 “Hystrix”	在调用外部服务时，如何使用 Hystrix 断路器？Camel 2.18 中的新功能.
	第 8.12 节 “service Call”	如何在 registry 中查找该服务在分布式系统中调用远程服务？Camel 2.18 中的新功能.
	第 8.13 节 “多播”	如何同时将消息路由到多个端点？
	第 8.16 节 “loop”	如何在循环中重复处理消息？

图标	名称	使用案例
	第 8.17 节 “sampling”	我该如何把给定时间段内的许多消息进行抽样，以避免超载下游路由？

消息转换

消息转换模式（如 表 3.5 “消息转换” 所示）描述了如何根据不同的目的修改信息内容。

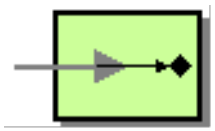
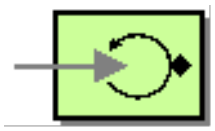
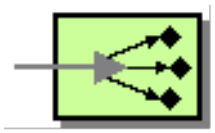
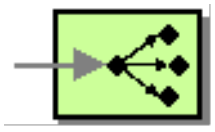
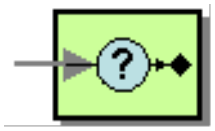
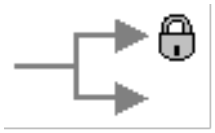
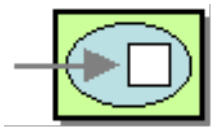
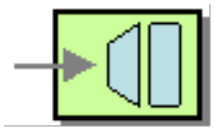
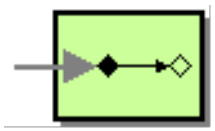
表 3.5. 消息转换

图标	名称	使用案例
	第 10.1 节 “内容增强器”	如果消息源器没有所有必需的数据项，我如何与另一个系统通信？
	第 10.2 节 “内容过滤器”	当您只对一些数据项目感兴趣时，您如何简化处理大量消息？
	第 10.4 节 “声明检查 EIP”	如何在不牺牲信息内容的情况下减少系统发送的消息量？
	第 10.3 节 “规范化程序”	您如何以其他格式格式处理语义等同的信息？
	第 10.5 节 “排序”	如何对邮件的正文进行排序？

消息传递端点

消息传递端点表示消息传递频道和应用程序之间的联系点。消息端点模式（如 表 3.6 “消息传递端点” 所示）介绍了可以在端点上配置的各种功能和服务质量。

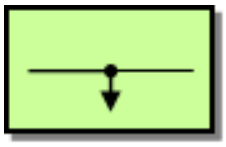
表 3.6. 消息传递端点

图标	名称	使用案例
	第 11.1 节 “消息传递映射程序”	您如何在域对象和消息传递基础架构之间移动数据，同时保持相互独立的这两个数据？
	第 11.2 节 “event Driven Consumer”	应用程序如何在消息可用时自动消耗消息？
	第 11.3 节 “polling Consumer”	应用程序如何在应用程序就绪时消耗消息？
	第 11.4 节 “竞争消费者”	消息传递客户端如何同时处理多个消息？
	第 11.5 节 “Message Dispatcher”	单个频道中的多个消费者如何协调其消息处理？
	第 11.6 节 “选择ive Consumer”	消息消费者如何选择希望接收的消息？
	第 11.7 节 “durable Subscriber”	在订阅者没有侦听它们时，如何避免缺少消息？
	第 11.8 节 “幂等的消费者”	邮件接收器如何处理重复的消息？
	第 11.9 节 “事务客户端”	客户端如何通过消息传递系统控制其事务？
	第 11.10 节 “消息传递网关”	您如何从应用的其余部分封装对消息传递系统的访问？
	第 11.11 节 “Service Activator”	应用程序如何设计要被各种消息传递技术以及非消息传递技术调用的服务？

系统管理

系统管理模式（如 表 3.7 “系统管理” 所示）介绍了如何监控、测试和管理消息传递系统。

表 3.7. 系统管理

图标	名称	使用案例
	第 12 章 *系统管理*	如何检查出差点到点频道的消息？

第 4 章 定义 REST 服务

摘要

Apache Camel 支持多种方法来定义 REST 服务。特别是，Apache Camel 提供 REST DSL(Domain Specific Language)，它是一种简单但强大的流畅 API，可以在任何 REST 组件上分层并提供与 OpenAPI 集成。

4.1. CAMEL 中的 REST 概述

概述

Apache Camel 提供了很多不同的方法和组件，用于在 Camel 应用程序中定义 REST 服务。本节概述了这些不同方法和组件，以便您可以决定哪些实施和 API 最适合您的要求。

什么是 REST？

Representational State Transfer (REST)是一个分布式应用程序的架构，它仅利用 HTTP 传输数据的中心，仅使用四个基本的 HTTP 动词： GET、POST、PUT 和 DELETE。

与 SOAP 等协议不同，它把 HTTP 视为 SOAP 消息的传输协议，REST 架构直接利用 HTTP。关键洞察是，HTTP 协议 本身 由几个简单惯例增强，它非常适合充当分布式应用程序的框架。

REST 调用示例

因为 REST 架构围绕标准 HTTP 动词构建，因此在很多情况下，您可以使用常规浏览器作为 REST 客户端。例如，若要调用在主机和端口 localhost:9091 上运行的简单 Hello World REST 服务，您可以在浏览器中导航到类似于以下 URL 的 URL：

```
http://localhost:9091/say/hello/Garp
```

Hello World REST 服务可能会返回响应字符串，例如：

```
Hello Garp
```

其显示在浏览器窗口中。您可以使用标准浏览器（或 curl 命令行实用程序）调用 REST 服务的简易性，是 REST 协议快速获得流行的原因之一。

REST 打包程序层

以下 REST 打包程序层提供了定义 REST 服务的简化语法，并可在不同 REST 实现之上分层：

REST DSL

REST DSL（在 camelcore中）是一个 facade 或 wrapper 层，它为定义 REST 服务提供了一个简化的构建器 API。REST DSL 本身不 提供 REST 实现：它必须与底层 REST 实现合并。例如，以下 Java 代码演示了如何使用 REST DSL 定义一个简单的 Hello World 服务：

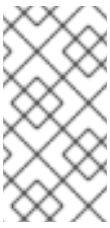
```
rest("/say")
    .get("/hello/{name}").route().transform().simple("Hello ${header.name}");
```

如需了解更多详细信息，请参阅 第 4.2 节 “使用 REST DSL 定义服务”。

REST 组件

Rest 组件（在 camel-core中）是一个打包程序层，可让您使用 URI 语法定义 REST 服务。与 REST DSL 一样，Rest(Rest)组件本身 不提供 REST 实现。它必须与底层 REST 实现相结合。

如果您没有显式配置 HTTP 传输组件，则 REST DSL 会自动发现通过检查类路径上的可用组件来使用的 HTTP 组件。REST DSL 查找任何 HTTP 组件的默认名称，并使用它找到的第一个名称。如果类路径中没有 HTTP 组件，并且您没有显式配置 HTTP 传输，则默认 HTTP 组件为 camel-http。



注意

自动发现要使用哪些 HTTP 组件的功能是 Camel 2.18 中的新功能。Camel 2.17 中不提供它。

以下 Java 代码演示了如何使用 camel-rest 组件定义一个简单的 Hello World 服务：

```
from("rest:get:say:/hello/{name}").transform().simple("Hello ${header.name}");
```

REST 实现

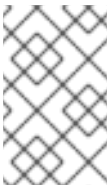
Apache Camel 通过以下组件提供多个不同的 REST 实现：

spark-Rest 组件

Spark-Rest 组件（在 camel-spark-rest）是一个 REST 实现，可让您使用 URI 语法定义 REST 服务。Spark 框架本身是一个 Java API，它基于 Sinatra 框架（一个 Python API）。例如，以下 Java 代码演示了如何使用 Spark-Rest 组件定义一个简单的 Hello World 服务：

```
from("spark-rest:get:/say/hello/:name").transform().simple("Hello ${header.name}");
```

请注意，与 Rest 组件不同，URI 中的变量语法为 :name，而不是 {name}。



注意

Spark-Rest 组件需要 Java 8。

Restlet 组件

Restlet 组件（在 camel-restlet）是一个 REST 实现，在原则上可以进行分层，但此组件仅针对 HTTP 协议进行测试。此组件还提供与 Restlet Framework 集成，它是 Java 中开发 REST 服务的一种商业框架。例如，以下 Java 代码演示了如何使用 Restlet 组件定义一个简单的 Hello World 服务：

```
from("restlet:http://0.0.0.0:9091/say/hello/{name}?restletMethod=get")
    .transform().simple("Hello ${header.name}");
```

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 Restlet。

servlet 组件

Servlet 组件（在 camel-servlet中）是一个组件，它将 Java servlet 绑定到 Camel 路由。换而言之，Servlet 组件可让您打包和部署 Camel 路由，就像它是标准的 Java servlet。因此，当您需要在 servlet 容器中部署 Camel 路由（例如，到 Apache Tomcat HTTP 服务器或 JBoss Enterprise Application Platform 容器中）中，Servlet 组件特别有用。

但是，它本身的 Servlet 组件不提供任何便捷的 REST API 来定义 REST 服务。因此，最简单的使用 Servlet 组件是将其与 REST DSL 合并，以便您可以用用户友好的 API 定义 REST 服务。

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 Servlet。

JAX-RS REST 实施

JAX-RS （RESTful Web 服务的 Java API）是一个将 REST 请求绑定到 Java 对象的框架，其中 Java 类必须使用 JAX-RS 注释进行解码，以定义绑定。JAX-RS 框架相对成熟，为开发 REST 服务提供

了复杂的框架，但它在编程上也比较复杂。

JAX-RS 与 Apache Camel 集成由 CXFRS 组件实施，该组件通过 Apache CXF 进行分层。简而言之，CIBian-RS 使用以下注释将 REST 请求绑定到 Java 类（其中，这仅是许多可用注释的不完整示例）：

@Path

标注可以映射至 Java 类的上下文路径，或者将子路径映射到特定的 Java 方法。

@GET, @POST, @PUT, @DELETE

将 HTTP 方法映射到 Java 方法的标注。

@PathParam

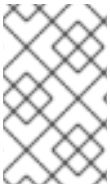
将 URI 参数映射到 Java 方法参数的注解，或者将 URI 参数注入到字段中。

@QueryParam

注释，可将查询参数映射到 Java 方法参数，或者将查询参数注入字段。

REST 请求或 REST 响应的正文通常应采用 JAXB(XML)数据格式。但 Apache CXF 还支持 JSON 格式转换为 JAXB 格式，以便也可以解析 JSON 消息。

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 和 Apache CXF 开发指南中的 CXFRS。



注意

CXFRS 组件 没有与 REST DSL 集成。

4.2. 使用 REST DSL 定义服务

REST DSL 是一个常见问题解答

REST DSL 实际上是一个 facade，它提供了在 Java DSL 或 XML DSL（域特定语言）中定义 REST 服务的简化语法。REST DSL 并不实际提供 REST 实现，它只是一个围绕 现有 REST 实施（Apache Camel 中有几个）的打包程序。

REST DSL 的优点

REST DSL 打包程序层提供以下优点：

- 现代易用的语法来定义 REST 服务。
- 与多个不同的 Apache Camel 组件兼容。
- OpenAPI 集成（通过 camel-openapi-java 组件）。

与 REST DSL 集成的组件

由于 REST DSL 不是实际的 REST 实现，所以您需要做的第一项一项是选择 Camel 组件以提供底层实施。以下 Camel 组件目前与 REST DSL 集成：

- Servlet 组件(camel-servlet)。
- spark REST 组件(camel-spark-rest)。
- Netty4 HTTP 组件(camel-netty4-http)。
- Jetty 组件(camel-jetty)。
- Restlet 组件(camel-restlet)。
- Undertow 组件(camelundertow)。

注意

Rest 组件（ camelcore的一部分）并不是 REST 的实施。与 REST DSL 一样，Reest 组件是一个 facade，它提供了一个简化的语法来定义使用 URI 语法的 REST 服务。Rest 组件还需要底层 REST 的实施。

配置 REST DSL 以使用 REST 实现

要指定 REST 的实现，您可以使用 restConfiguration() 构建器（在 Java DSL 中）或 restConfiguration 元素（在 XML DSL 中）。例如，要将 REST DSL 配置为使用 Spark-Rest 组件，您可以在 Java DSL 中使用类似以下内容的构建器表达式：

```
restConfiguration().component("spark-rest").port(9091);
```

您会在 XML DSL 中使用类似如下的元素（作为 camelContext的子项）：

```
<restConfiguration component="spark-rest" port="9091"/>
```

语法

定义 REST 服务的 Java DSL 语法如下：

```
rest("BasePath").Option().
    .Verb("Path").Option().[to() | route().CamelRoute.endRest()]
    .Verb("Path").Option().[to() | route().CamelRoute.endRest()]
    ...
    .Verb("Path").Option().[to() | route().CamelRoute];
```

其中 CamelRoute 是一个可选的嵌入式 Camel 路由（使用路由的标准 Java DSL 语法定义）。

REST 服务定义以 rest() 关键字开头，后跟处理特定 URL 路径片段的一个或多个 动词子。HTTP 动词可以是 get()、head()、put()、post()、delete()、patch() 或 verb() 之一。每个 verb 子句都可以使用以下语法之一：

- *口头以 to() 关键字结尾的动词子。例如：*

  ```
  get("...").Option()+.to("...")
  ```

- *动词 子句以 route() 关键字结尾（用于嵌入 Camel 路由）。例如：*

  ```
  get("...").Option()+.route("...").CamelRoute.endRest()
  ```

带有 Java 的 REST DSL

在 Java 中，若要使用 REST DSL 定义服务，请将 REST 定义放在 RouteBuilder.configure（） 方法的正文中，就像常规的 Apache Camel 路由一样。例如，要使用带有 Spark-Rest 组件的 REST DSL 定义一个简单的 Hello World 服务，请定义以下 Java 代码：

```
restConfiguration().component("spark-rest").port(9091);

rest("/say")
    .get("/hello").to("direct:hello")
    .get("/bye").to("direct:bye");

from("direct:hello")
    .transform().constant("Hello World");
from("direct:bye")
    .transform().constant("Bye World");
```

前面的示例具有三种不同类型的构建器：

restConfiguration()

配置 REST DSL 以使用特定的 REST 实现(Spark-Rest)。

rest()

使用 REST DSL 定义服务。各个动词条款均通过 to（） 关键字终止，该关键字可将传入的消息转发到 直接 端点（ 直接 组件转换器，在同一应用程序内路由）。

from()

定义常规 Camel 路由。

使用 XML 的 REST DSL

在 XML 中，若要使用 XML DSL 定义服务，请将 rest 元素定义为 camelContext 元素的子级。例如，若要使用带有 Spark-Rest 组件的 REST DSL 定义一个简单的 Hello World 服务，请定义以下 XML 代码（在 Blueprint 中）：

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <restConfiguration component="spark-rest" port="9091"/>

  <rest path="/say">
    <get uri="/hello">
      <to uri="direct:hello"/>
    </get>
    <get uri="/bye">
      <to uri="direct:bye"/>
    </get>
  </rest>
```

```
  <route>
    <from uri="direct:hello"/>
    <transform>
      <constant>Hello World</constant>
    </transform>
  </route>
  <route>
    <from uri="direct:bye"/>
    <transform>
      <constant>Bye World</constant>
    </transform>
  </route>
</camelContext>
```

指定基本路径

rest（） 关键字(Java DSL)或 rest 元素的 path 属性(XML DSL)允许您定义一个基本路径，然后是作为前缀放在所有 动词条款中的路径上。例如，给定以下 Java DSL 片段：

```
rest("/say")
    .get("/hello").to("direct:hello")
    .get("/bye").to("direct:bye");
```

或者给定以下 XML DSL 片段：

```
<rest path="/say">
  <get uri="/hello">
    <to uri="direct:hello"/>
  </get>
  <get uri="/bye" consumes="application/json">
    <to uri="direct:bye"/>
  </get>
</rest>
```

REST DSL 构建器为您提供了以下 URL 映射：

```
/say/hello
/say/bye
```

基本路径是可选的。如果您愿意，您可以在每个 动词条款中指定完整路径：

```
rest()
    .get("/say/hello").to("direct:hello")
    .get("/say/bye").to("direct:bye");
```

使用动态进行

REST DSL 支持使用 toD 动态参数。使用此参数指定 URI。

例如，在 JMS 中，可以通过以下方式定义动态端点 URI：

```
public void configure() throws Exception {
   rest("/say")
     .get("/hello/{language}").toD("jms:queue:hello-${header.language}");
}
```

在 XML DSL 中，同样的详情如下：

```
<rest uri="/say">
  <get uri="/hello//{language}">
    <toD uri="jms:queue:hello-${header.language}"/>
  </get>
<rest>
```

有关 toD 动态参数的更多信息，请参阅 “动态到”一节。

URI 模板

在动词参数中，您可以指定一个 URI 模板，您可以在命名的属性中捕获特定路径片段（然后映射到 Camel 消息标头）。例如，如果您想要对 Hello World 应用进行个性化，以便它根据名称设置调用者，您可以定义一个 REST 服务，如下所示：

```
rest("/say")
    .get("/hello/{name}").to("direct:hello")
    .get("/bye/{name}").to("direct:bye");

from("direct:hello")
    .transform().simple("Hello ${header.name}");
from("direct:bye")
    .transform().simple("Bye ${header.name}");
```

URI 模板捕获 {name} 路径片段的文本，并将这个捕获的文本复制到 名称 消息标头中。如果您通过发送 URL 以 /say/hello/Joe 结尾的 GET HTTP 请求调用服务，则 HTTP 响应为 Hello Joe。

嵌入式路由语法

您可以使用 to（） 关键字(Java DSL)或 to 元素(XML DSL)来终止 verb 子句，而是可以选择直接使用 Apache Camel 路由到 REST DSL 中，使用 route（） 关键字(Java DSL)或 路由 元素(XML DSL)。route（） 关键字允许您将路由嵌入到 verb 子句中，语法如下：

```
RESTVerbClause.route("...").CamelRoute.endRest()
```

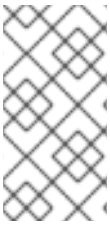
其中 endRest（） 关键字（仅限 Java DSL）是一个必要的标点符号，可让您分隔操作动词子句（当 rest（） 构建器中有多个动词子句时）。

例如，您可以重构 Hello World 示例以使用嵌入的 Camel 路由，如下为 Java DSL：

```
rest("/say")
    .get("/hello").route().transform().constant("Hello World").endRest()
    .get("/bye").route().transform().constant("Bye World");
```

如下在 XML DSL 中：

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  ...
  <rest path="/say">
    <get uri="/hello">
      <route>
        <transform>
          <constant>Hello World</constant>
        </transform>
      </route>
    </get>
    <get uri="/bye">
      <route>
        <transform>
          <constant>Bye World</constant>
        </transform>
      </route>
    </get>
  </rest>
</camelContext>
```



注意

如果您在当前 CamelContext 中定义任何异常子句（使用 Exception （））或拦截器（使用拦截器（）），这些异常子句和拦截器在嵌入式路由中也处于活动状态。

REST DSL 和 HTTP 传输组件

如果您没有显式配置 HTTP 传输组件，则 REST DSL 会自动发现通过检查类路径上的可用组件来使用的 HTTP 组件。REST DSL 查找任何 HTTP 组件的默认名称，并使用它找到的第一个名称。如果类路径中没有 HTTP 组件，并且您没有显式配置 HTTP 传输，则默认 HTTP 组件为 camel-http。

指定请求和响应的内容类型

您可以使用 consume （） 过滤 HTTP 请求和响应 的内容类型，并在 Java 中生成s（） 选项，或者在 XML 中生成属性。 例如，一些常见内容类型（官方称为 互联网介质类型）如下：

- text/plain
- text/html
- text/xml
- application/json
- application/xml

内容类型在 REST DSL 中以 verb 子句的形式提供。例如，若要将 verb 子句限制为仅接受 text/plain HTTP 请求，并且仅发送 文本/html HTTP 响应，您需要使用类似如下的 Java 代码：

```
rest("/email")
    .post("/to/{recipient}").consumes("text/plain").produces("text/html").to("direct:foo");
```

在 XML 中，您可以设置 使用 并生成 属性，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  ...
  <rest path="/email">
    <post uri="/to/{recipient}" consumes="text/plain" produces="text/html">
      <to "direct:foo"/>
    </get>
  </rest>
</camelContext>
```

您还可以指定使用 （）的参数，或以逗号分隔列表的形式 生成。例如，使用 ("text/plain,

application/json")。

其他 HTTP 方法

一些 HTTP 服务器实施支持额外的 HTTP 方法，它们不是由 REST DSL 中标准动词集合提供的，即 get（）, head（）, put （）, post （）, delete （）, patch（）.要访问其他 HTTP 方法，您可以在 XML DSL 中使用 generic 关键字 verb（） 和通用元素 动词。

例如，要在 Java 中实施 DIB HTTP 方法：

```
rest("/say")
    .verb("TRACE", "/hello").route().transform();
```

这里的 transform（） 将 IN 消息正文复制到 OUT 消息的正文，以回显 HTTP 请求。

在 XML 中实施 DIB HTTP 方法：

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  ...
  <rest path="/say">
    <verb uri="/hello" method="TRACE">
      <route>
        <transform/>
      </route>
    </get>
</camelContext>
```

定义自定义 HTTP 错误消息

如果您的 REST 服务需要发送错误消息作为其响应，您可以定义自定义 HTTP 错误消息，如下所示：

1. 通过将 Exchange.HTTP_RESPONSE_CODE 标头密钥设置为错误代码值来指定 HTTP 错误代码（例如，400、404 等）。此设置指示您要发送错误消息回复的 REST DSL，而不是定期响应。

2. 将消息正文填充自定义错误消息。

3. 如果需要，设置 Content-Type 标头。

4. 如果您的 REST 服务被配置为为来自 Java 对象和从 Java 对象（启用 绑定Mode ）,您应该确定启用了 skipBindingOnErrorCode 选项（默认为 ）。这是为了确保 REST DSL 在发送响应时不会尝试解压缩消息正文。

有关对象绑定的详情，请参阅 第 4.3 节 “摘要到 Java 对象以及”。

以下 Java 示例演示了如何定义自定义错误消息：

```
// Java
// Configure the REST DSL, with JSON binding mode
restConfiguration().component("restlet").host("localhost").port(portNum).bindingMode(RestBindingMode.json);

// Define the service with REST DSL
rest("/users/")
    .post("lives").type(UserPojo.class).outType(CountryPojo.class)
        .route()
            .choice()
                .when().simple("${body.id} < 100")
                    .bean(new UserErrorService(), "idTooLowError")
                .otherwise()
                    .bean(new UserService(), "livesWhere");
```

在这个示例中，如果输入 ID 小于 100，则返回自定义错误消息，使用 UserErrorService bean，它实现：

```
// Java
public class UserErrorService {
    public void idTooLowError(Exchange exchange) {
        exchange.getIn().setBody("id value is too low");
        exchange.getIn().setHeader(Exchange.CONTENT_TYPE, "text/plain");
        exchange.getIn().setHeader(Exchange.HTTP_RESPONSE_CODE, 400);
    }
}
```

在 UserErrorService bean 中，我们定义自定义错误消息，并将 HTTP 错误代码设置为 400。

参数默认值

可以为传入 Camel 消息的标头指定默认值。

您可以使用一个键单词（如 query 参数的 verbose ）来指定默认值。例如，在下面的代码中，默认值为 false。这意味着，如果没有为带有 verbose 键的标头提供其他值，则会将 false 插入为默认值。

```
rest("/customers/")
    .get("/{id}").to("direct:customerDetail")
    .get("/{id}/orders")
      .param()
 .name("verbose")
 .type(RestParamType.query)
 .defaultValue("false")
 .description("Verbose order details")
      .endParam()
        .to("direct:customerOrders")
    .post("/neworder").to("direct:customerNewOrder");
```

将 JsonParserException 嵌套在自定义 HTTP 错误消息中

您可能希望返回自定义错误消息的一个常见情形是嵌套 JsonParserException 异常。例如，您可以方便地利用 Camel 异常处理机制来创建自定义 HTTP 错误消息，其 HTTP 错误代码 400，如下所示：

```
// Java
onException(JsonParseException.class)
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(400))
    .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))
    .setBody().constant("Invalid json data");
```

REST DSL 选项

通常，可以直接将 REST DSL 选项应用到服务定义的基本部分（即，立即遵循 rest（））， 如下所示：

```
rest("/email").consumes("text/plain").produces("text/html")
    .post("/to/{recipient}").to("direct:foo")
    .get("/for/{username}").to("direct:bar");
```

在这种情形中，如果指定选项应用到所有从属动词子句。或者可将选项应用到每个单独的 verb 子句，如下所示：

```
rest("/email")
    .post("/to/{recipient}").consumes("text/plain").produces("text/html").to("direct:foo")
    .get("/for/{username}").consumes("text/plain").produces("text/html").to("direct:bar");
```

如果指定选项仅适用于相关的 动词子，则从基础部分覆盖任何设置。

表 4.1 “REST DSL 选项” 总结 REST DSL 支持的选项。

表 4.1. REST DSL 选项

Java DSL	XML DSL	描述
bindingMode()	**@bindingMode**	指定绑定模式，可用于将传入消息放入 Java 对象（以及可选的、unmarshal Java 对象到传出消息）。可以具有以下值：**off**（默认）、**auto**、**json**、**xml**、**json_xml**。
consumes()	**@consumes**	限制 verb 子句，以仅接受 HTTP 请求中的指定互联网介质类型（MIME 类型）。典型的值有：**text/plain**,**text/http**,**text/xml**,**application/json**,**application/xml**.
customId()	**@customId**	定义用于 JMX 管理的自定义 ID。
description()	**description**	记录 REST 服务或动词条款。对于 JMX 管理工具非常有用。
enableCORS()	**@enableCORS**	如果为 **true**，请在 HTTP 响应中启用 CORS（跨原始资源共享）标头。默认为 **false**。
id()	**@id**	为 REST 服务定义唯一 ID，它可用于定义 JMX 管理和其他工具。
method（）	**@method**	指定此动词子句处理的 HTTP 方法。通常与 generic **verb（）** 关键字一起使用。
outType()	**@outType**	启用对象绑定（即启用 **bindingMode** 选项时），这个选项指定代表 **HTTP 响应** 消息的 Java 类型。

Java DSL	XML DSL	描述
produces()	**produces**	限制 verb 子句，使其仅在 HTTP 响应中仅生成指定的互联网介质类型（MIME 类型）。典型的值有：**text/plain**,**text/http**,**text/xml**,**application/json**,**application/xml**.
type()	**@type**	启用对象绑定（即启用 **bindingMode** 选项时），这个选项指定代表 **HTTP Request** 消息的 Java 类型。
VerbURIArgument	**@uri**	指定路径片段或 URI 模板，作为动词的参数。例如，**get(*VerbURIArgument*)**。
BasePathArgument	**@path**	指定 **rest（）** 关键字(Java DSL)或 **rest** 元素(XML DSL)中的基本路径。

4.3. 摘要到 JAVA 对象以及

用于通过 HTTP 传输的 Java 对象

使用 REST 协议的最常见方式之一是传输消息正文中 Java Bean 的内容。为了实现此工作，您需要有一个机制来汇总 Java 对象，以及从适当的数据格式放入其中。REST DSL 支持下列数据格式适用于编码 Java 对象：

JSON

JSON （JavaScript 对象表示法）是一个轻量级数据格式，可轻松从 Java 对象映射。JSON 语法是紧凑、轻便键入的，方便人进行读和写。因此，JSON 已成为 REST 服务的消息格式。

例如，以下 JSON 代码可以代表用户 bean，具有两个属性字段，即 id 和 name ：

```
{
   "id" : 1234,
   "name" : "Jane Doe"
}
```

JAXB

JAXB （XML Binding 的 Java 架构）是基于 XML 的数据格式，可轻松映射到 Java 对象或从 Java 对象映射。要将 XML 接合到 Java 对象，还必须注解您要使用的 Java 类。

例如，以下 JAXB 代码可以代表用户 bean，其中有两个属性字段： id 和 name ：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<User>
  <Id>1234</Id>
  <Name>Jane Doe</Name>
</User>
```

注意

从 Camel 2.17.0 中，CIB 数据格式和类型转换支持从 XML 转换到 POJO，这些类使用 ObjectFactory 而不是 XmlRootElement。此外，camel 上下文应包含 CamelJaxbObjectFactory 属性，其值为 true。但是，由于优化，默认值为 false。

JSON 和 JAXB 与 REST DSL 集成

当然，您可以编写所需的代码，以将邮件正文转换为 Java 对象，并自行从 Java 对象转换。但是 REST DSL 提供了自动进行这个转换的便利。特别是，JSON 和 JAXB 与 REST DSL 的集成提供了以下优点：

- 自动对 Java 对象和从 Java 对象编入和执行（授予适当的配置）。
- REST DSL 可以自动检测数据格式（JSON 或 JAXB），并执行适当的转换。
- REST DSL 提供了一个抽象层，以便您编写的代码不特定于特定的 JSON 或 JAXB 实施。因此，您可以稍后切换实施，对应用程序代码产生最小影响。

支持的数据格式组件

Apache Camel 提供 JSON 和 JAXB 数据格式的多个不同实施。REST DSL 目前支持以下数据格式：

- JSON

- jackson 数据格式(camel-jackson) （默认）
- GSon 数据格式(camel-gson)
- XStream 数据格式(camel-xstream)

- JAXB
  - JAXB 数据格式(camel-jaxb)

如何启用对象 marshalling

要在 REST DSL 中启用对象摘要，请观察以下几点：

1. 启用绑定模式，通过设置 bindingMode 选项（有几个级别可设置绑定模式）。如需详情，请参阅 “配置绑定模式”一节。

2. 使用 type 选项（必需）以及带有 outType 选项（可选）的传出消息上指定要转换为的 Java 类型（或从中）。

3. 如果要将 Java 对象转换为 JAXB 数据格式，您必须记得使用适当的 JAXB 注释给 Java 类添加注解。

4. 指定底层数据格式实现（或实现），使用 jsonDataFormat 选项和/或 xmlDataFormat 选项（可以在 restConfiguration 构建器中指定）。

5. 如果您的路由以 JAXB 格式提供返回值，则通常您希望将交换正文的 Out 消息设置为带有 JAXB 注释（a JAXB 元素）的类实例。如果您希望以 XML 格式直接提供 JAXB 返回值，但是，使用键 xml.out.mustBeJAXBElement 将 dataFormatProperty 设置为 false （可以在 restConfiguration 构建器中指定）。例如，在 XML DSL 语法中：

```
<restConfiguration ...>
  <dataFormatProperty key="xml.out.mustBeJAXBElement"
                      value="false"/>
```

```
  ...
</restConfiguration>
```

6. 将所需的依赖项添加到项目构建文件中。例如，如果您使用 Maven 构建系统，且您使用 Jackson 数据格式，您要将以下依赖项添加到 Maven POM 文件中：

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependencies>
    ...
    <!-- use for json binding --> <dependency> <groupId>org.apache.camel</groupId>
<artifactId>camel-jackson</artifactId> </dependency>
    ...
  </dependencies>
</project>
```

7. 将应用程序部署到 OSGi 容器时，请记得为您的所选数据格式安装必要的功能。例如，如果您使用的是 Jackson 数据格式（默认），则通过输入以下命令来安装 camel-jackson 功能：

```
JBossFuse:karaf@root> features:install camel-jackson
```

或者，如果您部署到 Fabric 环境，您可以将该功能添加到 Fabric 配置集中。例如，如果您使用配置集 MyRestProfile，您可以通过输入以下命令来添加该功能：

```
JBossFuse:karaf@root> fabric:profile-edit --features camel-jackson MyRestProfile
```

配置绑定模式

默认情况下，bindingMode 选项是 off，因此您必须明确配置它，以便能启用 Java 对象的 marshalling。TABLE 显示支持的绑定模式列表。

注意

仅当 content-type 标头包含 json 或 xml 的绑定时，从 POJO 到 JSon/JAXB 的绑定才会进行 Camel 2.16.3。如果消息正文不应尝试使用绑定进行总结，这允许您指定自定义内容类型。例如，当消息正文是一个自定义二进制有效负载时，这很有用。

表 4.2. REST DSL BInding 模式

绑定模式	描述
off	关闭绑定 （**默认**）。
auto	为 JSON 和/或 XML 启用绑定。在这个模式中，Camel 会自动根据传入消息的格式自动选择 JSON 或 XML(JAXB)。**您不需要** 同时启用两种数据格式：JSON 实现、XML 实现，或两者都可以在类路径中提供。
json	仅为 JSON 启用了绑定。**必须在** classpath 上提供 JSON 实施（默认为 Camel 尝试启用 **camel-jackson** 实施）。
xml	仅为 XML 启用绑定。**必须在** classpath 上提供一个 XML 实现（默认为 Camel 尝试启用 **camel-jaxb** 实现）。
json_xml	为 JSON 和 XML 启用了绑定。在这个模式中，Camel 会自动根据传入消息的格式自动选择 JSON 或 XML(JAXB)。您需要在 classpath 上提供两种数据类型。

在 Java 中，这些绑定模式值表示为以下枚举类型的实例：

```
org.apache.camel.model.rest.RestBindingMode
```

您可以设置 bindingMode 的不同级别，如下所示：

REST DSL 配置

您可以在 restConfiguration 构建器中设置 bindingMode 选项，如下所示：

```
restConfiguration().component("servlet").port(8181).bindingMode(RestBindingMode.json);
```

服务定义基础部分

您可以在 rest（） 关键字后设置 bindingMode 选项（在 verb 子句之前），如下所示：

```
rest("/user").bindingMode(RestBindingMode.json).get("/{id}").VerbClause
```

verb 子句

您可以在 verb 子句中设置 bindingMode 选项，如下所示：

```
rest("/user")
    .get("/{id}").bindingMode(RestBindingMode.json).to("...");
```

示例

如需完整的代码示例，显示如何使用 REST DSL 将 Servlet 组件用作 REST 实施，了解 Apache Camel camel-example-servlet-rest-blueprint 示例。您可以通过安装独立 Apache Camel 发行版 apache-camel-2.23.2 找到这个示例。fuse-7_10_0-00018-redhat-00001.zip，它包括在 Fuse 安装的额外s / 子目录中。

安装独立 Apache Camel 发行版后，您可以在以下目录中找到示例代码：

```
ApacheCamelInstallDir/examples/camel-example-servlet-rest-blueprint
```

将 Servlet 组件配置为 REST 实现

在 camel-example-servlet-rest-blueprint 示例中，REST DSL 的底层实施由 Servlet 组件提供。Servlet 组件在 Blueprint XML 文件中配置，如 例 4.1 “为 REST DSL 配置 Servlet 组件” 所示。

例 4.1. 为 REST DSL 配置 Servlet 组件

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

  <!-- to setup camel servlet with OSGi HttpService -->
  <reference id="httpService" interface="org.osgi.service.http.HttpService"/>

  <bean class="org.apache.camel.component.servlet.osgi.OsgiServletRegisterer"
        init-method="register"
        destroy-method="unregister">
    <property name="alias" value="/camel-example-servlet-rest-blueprint/rest"/>
    <property name="httpService" ref="httpService"/>
    <property name="servlet" ref="camelServlet"/>
  </bean>

  <bean id="camelServlet"
class="org.apache.camel.component.servlet.CamelHttpTransportServlet"/>
  ...
  <camelContext xmlns="http://camel.apache.org/schema/blueprint">

    <restConfiguration component="servlet"
                       bindingMode="json"
                       contextPath="/camel-example-servlet-rest-blueprint/rest"
```

```
                port="8181">
    <dataFormatProperty key="prettyPrint" value="true"/>
   </restConfiguration>
   ...
  </camelContext>

 </blueprint>
```

要使用 REST DSL 配置 Servlet 组件，您需要配置由以下三个层组成的堆栈：

REST DSL 层

REST DSL 层由 restConfiguration 元素配置，该元素通过将 component 属性设置为值 servlet 来与 Servlet 组件集成。

servlet 组件层

Servlet 组件层作为类的实例 CamelHttpTransportServlet 实施，其中示例实例具有 bean ID(camelServlet)。

HTTP 容器层

Servlet 组件必须部署到 HTTP 容器中。Karaf 容器通常配置有默认的 HTTP 容器（a Jetty HTTP 容器），它监听端口 8181 上的 HTTP 请求。要将 Servlet 组件部署到默认的 Jetty 容器，您需要执行以下操作：

a. 获得对 org.osgi.service.http.HttpService compliant 服务（其中这个服务）是标准的 OSGi 接口，它提供了一个标准化的 OSGi 接口，提供对 OSGi 中默认 HTTP 服务器的访问。

b. 创建 utility 类的实例 OsgiServletRegisterer，以在 HTTP 容器中注册 Servlet 组件。OsgiServletRegister 类是简化 Servlet 组件生命周期的实用程序。创建此类实例时，它会自动调用 HttpService OSGi 服务中的 registerServlet 方法；在实例被销毁时，它会自动调用 unregister 方法。

所需的依赖项

这个示例有两个依赖项，它们是 REST DSL 的关键重要，如下所示：

servlet 组件

提供 REST DSL 的底层实施。这在 Maven POM 文件中指定，如下所示：

-

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-servlet</artifactId>
  <version>${camel-version}</version>
</dependency>
```

在将应用程序捆绑包部署到 OSGi 容器之前，您必须安装 Servlet 组件功能，如下所示：

```
JBossFuse:karaf@root> features:install camel-servlet
```

jackson 数据格式

提供 JSON 数据格式实施。这在 Maven POM 文件中指定，如下所示：

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jackson</artifactId>
  <version>${camel-version}</version>
</dependency>
```

在将应用程序捆绑包部署到 OSGi 容器之前，您必须安装 Jackson 数据格式功能，如下所示：

```
JBossFuse:karaf@root> features:install camel-jackson
```

Java 类型用于响应

示例应用程序会在 HTTP 请求和 Response 消息中回传递 User 类型对象。User Java 类定义为 *例 4.2 “用于 JSON 响应的用户类”* 所示。

例 4.2. 用于 JSON 响应的用户类

```
// Java
package org.apache.camel.example.rest;

public class User {

    private int id;
    private String name;

    public User() {
    }

    public User(int id, String name) {
        this.id = id;
        this.name = name;
```

```
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

User 类具有 JSON 数据格式相对简单表示。例如，此类的典型实例以 JSON 格式表示：

```
{
    "id" : 1234,
    "name" : "Jane Doe"
}
```

带有 JSON 绑定的 REST DSL 路由示例

本例中的 REST DSL 配置和 REST 服务定义显示在 例 4.3 “带有 JSON 绑定的 REST DSL 路由” 中。

例 4.3. 带有 JSON 绑定的 REST DSL 路由

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           ...>
  ...
  <!-- a bean for user services -->
  <bean id="userService" class="org.apache.camel.example.rest.UserService"/>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">

    <restConfiguration component="servlet"
                       bindingMode="json"
                       contextPath="/camel-example-servlet-rest-blueprint/rest"
                       port="8181">
      <dataFormatProperty key="prettyPrint" value="true"/>
    </restConfiguration>
```

```
    <!-- defines the REST services using the  base path, /user -->
    <rest path="/user" consumes="application/json" produces="application/json">
      <description>User rest service</description>

      <!-- this is a rest GET to view a user with the given id -->
      <get uri="/{id}" outType="org.apache.camel.example.rest.User">
        <description>Find user by id</description>
        <to uri="bean:userService?method=getUser(${header.id})"/>
      </get>

      <!-- this is a rest PUT to create/update a user -->
      <put type="org.apache.camel.example.rest.User">
        <description>Updates or create a user</description>
        <to uri="bean:userService?method=updateUser"/>
      </put>

      <!-- this is a rest GET to find all users -->
      <get uri="/findAll" outType="org.apache.camel.example.rest.User[]">
        <description>Find all users</description>
        <to uri="bean:userService?method=listUsers"/>
      </get>

    </rest>

  </camelContext>

</blueprint>
```

REST 操作

例 4.3 “带有 JSON 绑定的 REST DSL 路由” 中的 REST 服务定义以下 REST 操作：

GET /camel-example-servlet-rest-blueprint/rest/user/{id}

获取 {id} 识别的用户的详细信息，其中以 JSON 格式返回 HTTP 响应。

PUT /camel-example-servlet-rest-blueprint/rest/user

创建新用户，其中用户详情包含在 PUT 消息正文中，采用 JSON 格式编码（与 User 对象类型匹配）。

GET /camel-example-servlet-rest-blueprint/rest/user/findAll

获取 所有用户 的详情，其中 HTTP 响应作为用户数组返回，格式为 JSON 格式。

调用 REST 服务的 URL

通过检查 例 4.3 “带有 JSON 绑定的 REST DSL 路由” 中的 REST DSL 定义，您可以将调用每个

REST 操作所需的 URL 组成。例如，要调用第一个 REST 操作，它会返回具有给定 ID 的用户详情，其 URL 会按如下方式构建：

http://localhost:8181

在 restConfiguration 中，协议默认为 http，端口明确设置为 8181。

/camel-example-servlet-rest-blueprint/rest

由 restConfiguration 元素的 contextPath 属性指定。

/user

由 rest 元素的 path 属性指定。

/{id}

由 get verb 元素的 uri 属性指定。

因此，您可以在命令行中输入以下命令来使用 curl 实用程序调用此 REST 操作：

```
curl -X GET -H "Accept: application/json" http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/123
```

同样，可以通过 curl 调用剩余的 REST 操作，输入以下示例命令：

```
curl -X GET -H "Accept: application/json" http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/findAll

curl -X PUT -d "{ \"id\": 666, \"name\": \"The devil\"}" -H "Accept: application/json" http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user
```

4.4. 配置 REST DSL

使用 Java 配置

在 Java 中，您可以使用 restConfiguration（） 构建器 API 配置 REST DSL。例如，要将 REST DSL 配置为使用 Servlet 组件作为底层实现：

```
restConfiguration().component("servlet").bindingMode("json").port("8181")
    .contextPath("/camel-example-servlet-rest-blueprint/rest");
```

使用 XML 配置

在 XML 中，您可以使用 restConfiguration 元素配置 REST DSL。例如，要将 REST DSL 配置为使用 Servlet 组件作为底层实现：

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ...>
  ...
  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    ...
    <restConfiguration component="servlet"
                       bindingMode="json"
                       contextPath="/camel-example-servlet-rest-blueprint/rest"
                       port="8181">
      <dataFormatProperty key="prettyPrint" value="true"/>
    </restConfiguration>
    ...
  </camelContext>

</blueprint>
```

配置选项

表 4.3 “配置 REST DSL 的选项” 显示使用 restConfiguration（） 构建器(Java DSL)或 restConfiguration 元素(XML DSL)配置 REST DSL 的选项。

表 4.3. 配置 REST DSL 的选项

Java DSL	XML DSL	描述
component（）	**@component**	指定用作 REST 传输的 Camel 组件（如 **servlet**、**restlet**、**spark-rest** 等等）。该值可以是标准组件名称，也可以是自定义实例的 bean ID。如果没有指定这个选项，Camel 会在类路径或 bean registry 中查找 **RestConsumerFactory** 实例。
scheme()	**@scheme**	用于公开 REST 服务的协议。通常支持基于底层 REST 的实施，但 **http** 和 **https**。默认为 **http**。
host()	**@host**	用于公开 REST 服务的主机名。

Java DSL	XML DSL	描述
port()	**@port**	用于公开 REST 服务的端口号。 **注：** 此设置将被 Servlet 组件**忽略**，它使用容器的标准 HTTP 端口。对于 Apache Karaf OSGi 容器，标准 HTTP 端口通常为 8181。为采用 JMX 和工具设置端口值的好方法是个好办法。
contextPath()	**@contextPath**	为 REST 服务设置领先的上下文路径。这可以用于 Servlet 等组件，使用 **上下文路径** 设置部署部署的 Web 应用。
hostNameResolver()	**@hostNameResolver**	如果没有显式设置主机名，则此解析器决定 REST 服务的主机。可能的值有 **RestHostNameResolver.localHostName** (Java DSL)或 **localHostName** (XML DSL)，它解析为主机名格式；以及 **RestHostNameResolver.localIp** (Java DSL)或 **localIp** (XML DSL)，它解析为点十进制 IP 地址格式。从 Camel 2.17 **RestHostNameResolver.allLocalIp** 解析为所有本地 IP 地址。 默认值为 **localHostName** 最多 Camel 2.16。从 Camel 2.17，默认值 **是所有LocalIp**。
bindingMode()	**@bindingMode**	为 JSON 或 XML 格式消息启用绑定模式。可能的值有：**off**、**auto**、**json**、**xml** 或 **json_xml**。默认为 **off**。
skipBindingOnErrorCode()	**@skipBindingOnErrorCode**	指定是否在输出上跳过绑定，如果存在自定义 HTTP 错误代码标头。这可让您构建不绑定到 JSON 或 XML 的自定义错误消息，否则会成功消息。默认为 **true**。
enableCORS()	**@enableCORS**	如果为 **true**，请在 HTTP 响应中启用 CORS（跨原始资源共享）标头。默认为 **false**。

Java DSL	XML DSL	描述
jsonDataFormat()	**@jsonDataFormat**	指定 Camel 用来实施 JSON 数据格式的组件。可能的值有： **json-jackson**,**json-gson**,**json-xstream**.默认为 **json-jackson**。
xmlDataFormat()	**@xmlDataFormat**	指定 Camel 用于实施 XML 数据格式的组件。可能的值有： **jaxb**。默认为 **jaxb**。
componentProperty()	**componentProperty**	允许您在底层 REST 实现上设置任意 **组件级别** 属性。
endpointProperty()	**endpointProperty**	允许您在底层 REST 实现上设置任意 **端点级别** 属性。
consumerProperty()	**consumerProperty**	允许您在底层 REST 实现上设置任意 **使用者端点** 属性。
dataFormatProperty()	**dataFormatProperty**	允许您在底层数据格式组件上设置任意属性（如 Jackson 或 JAXB）。从 Camel 2.14.1 开始，您可以将以下前缀附加到属性键： • **json.in** • **json.out** • **xml.in** • **xml.out** 要将属性设置限制为特定格式类型（JSON 或 XML）以及特定消息方向（**IN** 或 **OUT**）。
corsHeaderProperty()	**corsHeaders**	允许您将自定义 CORS 标头指定为键/值对。

默认 CORS 标头

如果启用了 CORS（跨原始资源共享），则默认设置以下标头。您可以通过调用 corsHeaderProperty DSL 命令来选择覆盖默认设置。

表 4.4. 默认 CORS 标头

标头密钥	标头值
access-Control-Allow-Origin	*
access-Control-Allow-Methods	获取,**HEAD**,**POST**, PUT , DELETE ,**TRACE**,**OPTIONS**,**CONNECT**,**PATCH**
access-Control-Allow-Headers	**origin**,**Accept**,**X-Requested-With**,**Content-Type**,**Access-Control-Request-Method**,**Access-Control-Request-Headers**
access-Control-Max-Age	**3600**

*启用或禁用 **Jackson JSON** 功能*

*您可以通过在 **dataFormatProperty** 选项中配置以下键来启用或禁用特定的 **Jackson JSON** 功能：*

- ***json.in.disableFeatures***
- ***json.in.enableFeatures***

*例如，要禁用 **Jackson** 的 **FAIL_ON_UN_PROPERTIES** 功能（如果 **JSON** 输入包含无法映射到 **Java** 对象的属性，这会导致 **Jackson** 将失败）：*

```
restConfiguration().component("jetty")
    .host("localhost").port(getPort())
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("json.in.disableFeatures", "FAIL_ON_UNKNOWN_PROPERTIES");
```

您可以通过指定一个以逗号分隔的列表来禁用 多个功能。例如：

```
.dataFormatProperty("json.in.disableFeatures",
"FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE");
```

*以下是如何在 **Java DSL** 中禁用和启用 **Jackson JSON** 功能的示例：*

```
restConfiguration().component("jetty")
    .host("localhost").port(getPort())
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("json.in.disableFeatures",
"FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE")
    .dataFormatProperty("json.in.enableFeatures",
"FAIL_ON_NUMBERS_FOR_ENUMS,USE_BIG_DECIMAL_FOR_FLOATS");
```

以下是如何在 XML DSL 中禁用和启用 Jackson JSON 功能的示例：

```
<restConfiguration component="jetty" host="localhost" port="9090" bindingMode="json">
  <dataFormatProperty key="json.in.disableFeatures"
value="FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE"/>
  <dataFormatProperty key="json.in.enableFeatures"
value="FAIL_ON_NUMBERS_FOR_ENUMS,USE_BIG_DECIMAL_FOR_FLOATS"/>
</restConfiguration>
```

可以禁用或启用的 Jackson 功能与以下 Jackson 类中的 enum ID 对应

- *com.fasterxml.jackson.databind.SerializationFeature*
- *com.fasterxml.jackson.databind.DeserializationFeature*
- *com.fasterxml.jackson.databind.MapperFeature*

4.5. OPENAPI 集成

概述

您可以使用 OpenAPI 服务为 CamelContext 文件中的任何 REST 定义路由和端点创建 API 文档。要做到这一点，使用 Camel REST DSL 及 camel-openapi-java 模块，它完全基于 Java。camel-openapi-java 模块会创建一个 servlet，它与 CamelContext 集成，并从每个 REST 端点中提取信息，以 JSON 或 YAML 格式生成 API 文档。

如果使用 Maven，请编辑 pom.xml 文件，以添加对 camel-openapi-java 组件的依赖项：

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-openapi-java</artifactId>
```

```
    <version>x.x.x</version>
    <!-- Specify the version of your camel-core module. -->
</dependency>
```

配置 CamelContext 以启用 OpenAPI

要在 Camel REST DSL 中使用 OpenAPI，调用 apiContextPath（） 设置 OpenAPI 生成的 API 文档的上下文路径。例如：

```
public class UserRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        // Configure the Camel REST DSL to use the netty4-http component:
        restConfiguration().component("netty4-http").bindingMode(RestBindingMode.json)
            // Generate pretty print output:
            .dataFormatProperty("prettyPrint", "true")
            // Set the context path and port number that netty will use:
            .contextPath("/").port(8080)
            // Add the context path for the OpenAPI-generated API documentation:
            .apiContextPath("/api-doc")
                .apiProperty("api.title", "User API").apiProperty("api.version", "1.2.3")
                // Enable CORS:
                .apiProperty("cors", "true");

        // This user REST service handles only JSON files:
        rest("/user").description("User rest service")
            .consumes("application/json").produces("application/json")
            .get("/{id}").description("Find user by id").outType(User.class)
                .param().name("id").type(path).description("The id of the user to
get").dataType("int").endParam()
                .to("bean:userService?method=getUser(${header.id})")
            .put().description("Updates or create a user").type(User.class)
                .param().name("body").type(body).description("The user to update or create").endParam()
                .to("bean:userService?method=updateUser")
            .get("/findAll").description("Find all users").outTypeList(User.class)
                .to("bean:userService?method=listUsers");
    }
}
```

OpenAPI 模块配置选项

下表描述的选项可让您配置 OpenAPI 模块。设置如下选项：

- 如果您将 camel-openapi-java 模块用作 servlet，通过更新 web.xml 文件并为每个要设置的每个配置选项指定 init-param 元素来设置选项。

- 如果您使用来自 Camel REST 组件的 camel-openapi-java 模块，请通过调用适当的

RestConfigurationDefinition 方法来设置选项，如 enableCORS（）、 host（） 或 contextPath（）。使用 RestConfigurationDefinition.apiProperty（） 方法设置 api.xxx 选项。

选项	类型	描述
api.contact.email	字符串	用于 API 相关的对应电子邮件地址。
api.contact.name	字符串	要联系的人员或机构的名称。
api.contact.url	字符串	网站 URL 获取更多信息。
apiContextIdListing	布尔值	如果您的应用程序使用多个 **CamelContext** 对象，则默认行为是仅列出当前 **CamelContext** 中的 REST 端点。如果您希望在运行 REST 服务的 JVM 中运行的每个 **CamelContext** 中的 REST 端点列表，则此选项设置为 true。当 **apiContextIdListing** 为 true 时，OpenAPI 在根路径中输出 **CamelContext** ID，例如 **/api-docs**，作为 JSON 格式的名称列表。要访问 OpenAPI 生成的文档，请在 **CamelContext** ID 中附加 REST 上下文路径，如 **api-docs/myCamel**。您可以使用 **apiContextIdPattern** 选项过滤此输出列表中的名称。
apiContextIdPattern	字符串	过滤 CamelContext ID 出现在上下文列表中的模式。您可以指定正则表达式，并将 * 用作通配符。这与 Camel 拦截器功能所使用的模式匹配功能相同。
api.license.name	字符串	用于 API 的许可证名称。
api.license.url	字符串	用于 API 的许可证的 URL。
api.path	字符串	设置为 生成文档的 REST API 的路径，例如 **/api-docs**。指定一个相对路径。不要指定，例如 **http** 或 **https**。**camel-openapi-java** 模块以以下格式计算运行时的绝对路径：**protocol://host:port/context-path/api-path**。
api.termsOfService	字符串	API 服务条款的 URL。

选项	类型	描述
api.title	字符串	应用程序的标题。
api.version	字符串	API 的版本。默认值为 0.0.0。
base.path	字符串	必需。设置 REST 服务可用的路径。指定一个相对路径。也就是说，不要指定 **http** 或 **https**。**camel-openapi-java** modul 以以下格式计算运行时的绝对路径：**protocol://host:port/context-path/base.path**。
CORS	布尔值	是否启用 HTTP 访问控制 (CORS)。这只为查看 REST API 文档启用 CORS，而不可用于访问 REST 服务。默认值为 false。建议改为使用 **CorsFilter** 选项，如此表中所述。
主机	字符串	设置运行 OpenAPI 服务的主机的名称。默认值是根据 **localhost** 计算主机名。
方案	字符串	要使用的协议方案。使用逗号分隔多个值，如 **"http,https"**。默认值为 **http**。
opeapi.version	字符串	OpenAPI 规格版本。默认值为 3.0。

获取 JSON 或 YAML 输出

从 Camel 3.1 开始，camel-openapi-java 模块支持 JSON 和 YAML 格式的输出。要指定您需要的输出，将 /openapi.json 或 /openapi.yaml 添加到请求 URL。如果请求 URL 没有指定格式，则 camel-openapi-java 模块将检查 HTTP Accept 标头以检测 JSON 或 YAML 是否可以接受。如果两者都被接受，或者没有设置为接受状态，则 JSON 是默认的返回格式。

例子

在 Apache Camel 3.x 分发中，cal-example-openapi-cdi 和 camel-example-openapi-java 演示了使用 camel-openapi-java 模块。

在 Apache Camel 2.x 发行版中，camel-example-swagger-cdi 和 camel-example-swagger-java

演示了使用 camel-swagger-java 模块。

增强 OpenAPI 生成的文档

从 Camel 3.1 开始，您可以通过定义参数详情（如名称、描述、数据类型、参数类型等）来增强 OpenAPI 生成的文档。如果使用 XML，请指定 param 元素来添加此信息。以下示例演示了如何提供有关 ID 路径参数的信息：

```
<!-- This is a REST GET request to view information for the user with the given ID: -->
<get uri="/{id}" outType="org.apache.camel.example.rest.User">
    <description>Find user by ID.</description>
    <param name="id" type="path" description="The ID of the user to get information about."
dataType="int"/>
    <to uri="bean:userService?method=getUser(${header.id})"/>
</get>
```

以下是 Java DSL 中相同的示例：

```
.get("/{id}").description("Find user by ID.").outType(User.class)
  .param().name("id").type(path).description("The ID of the user to get information
about.").dataType("int").endParam()
  .to("bean:userService?method=getUser(${header.id})")
```

如果您定义了名称为 正文 的参数，那么您还必须将 正文 指定为这个参数的类型。例如：

```
<!-- This is a REST PUT request to create/update information about a user. -->
<put type="org.apache.camel.example.rest.User">
    <description>Updates or creates a user.</description>
    <param name="body" type="body" description="The user to update or create."/>
    <to uri="bean:userService?method=updateUser"/>
</put>
```

以下是 Java DSL 中相同的示例：

```
.put().description("Updates or create a user").type(User.class)
    .param().name("body").type(body).description("The user to update or create.").endParam()
    .to("bean:userService?method=updateUser")
```

另请参阅： Apache Camel 发行版中 的示例/camel-example-servlet-rest-tomcat。

第 5 章 消息传递系统

摘要

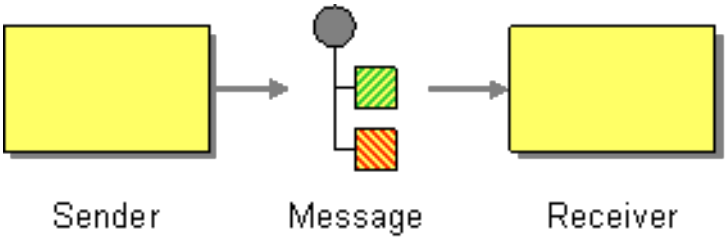
本章介绍消息传递系统的基本构建块，如端点、消息通道和消息路由器。

5.1. 消息

概述

消息 是传输消息传递系统的最小单元（由下图中的灰色点代表）。消息本身可能含有一些内部结构（即一个消息），其中包含了与 图 5.1 “消息模式” 中的 grey dot 连接的 geometrical figures 来表示的多部分（即 a geometrical figure）。

图 5.1. 消息模式



消息类型

Apache Camel 定义以下不同消息类型：

- 在消息 中，在从消费者端点通过路由传输到制作者端点的消息中（通常，启动消息交换）。
- 从生产者端点到消费者端点（通常是响应 In 消息）的消息中出发消息（通常要通过来自生产者端点的路由）。

所有这些消息类型都由 org.apache.camel.Message 接口在内部表示。

消息结构

默认情况下，Apache Camel 将以下结构应用到所有消息类型：

- 标头 相应数据包含从消息中提取的元数据或标头数据。

- 正文 abrt-sually 以原始形式包含整个消息。

- Attachments abrt-jaxb Message attachments（需要与特定的消息传递系统（如 JBI）集成。

务必要记住，这个部门分为标头、正文和附件，是消息的抽象模型。Apache Camel 支持许多不同的组件，生成各种消息格式。最终，这是底层组件实施，决定将什么放置到邮件的标头和正文中。

关联消息

在内部，Apache Camel 记住消息 ID，用于关联各个消息。但是，在实践中，Apache Camel 关联消息的最重要方法是 交换 对象。

Exchange 对象

Exchange 对象是封装相关消息的实体，其中相关消息的集合被称为 消息交换，并且管理消息序列的规则被称为 交换模式。例如，两个常见的交换模式是：单向事件消息（与 In 消息一致）和请求检查交换（记录 In 消息，后接 Out 消息）。

访问消息

在 Java DSL 中定义路由规则时，您可以使用以下 DSL 构建器方法访问消息的标头和正文：

- header(String name), body（） Curve- the returns the named header and the current In message 的正文。

- outBody（） libselinux- the return the current Out 消息的正文。

例如，若要填充 In 消息 的用户名 标头，您可以使用以下 Java DSL 路由：

```
from(SourceURL).setHeader("username", "John.Doe").to(TargetURL);
```

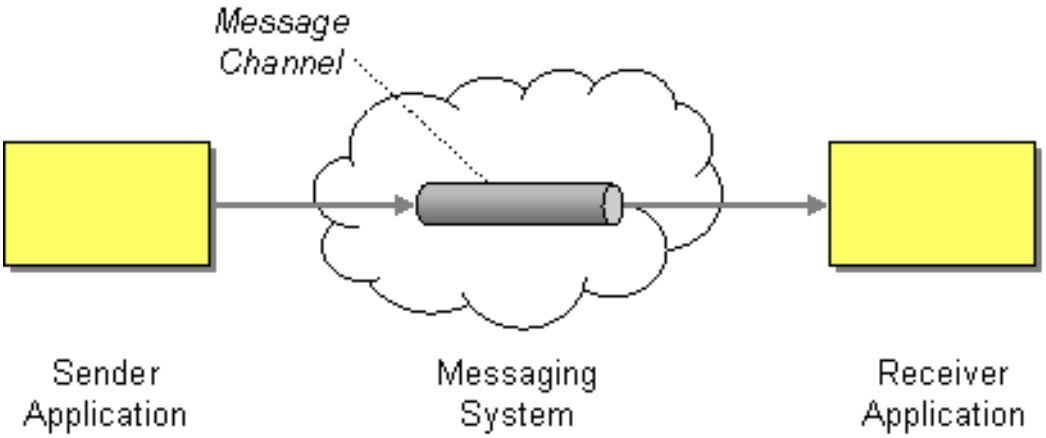
5.2. MESSAGE CHANNEL

概述

频道 是消息传递系统中的逻辑频道。也就是说，将消息发送到不同的消息通道提供了将消息排序到不同消息类型的元素方法。消息队列和消息主题是消息频道的示例。您应该记住，逻辑频道与物理频道不同。物理实现了逻辑通道的几种不同方法。

在 Apache Camel 中，消息频道由面向消息组件的端点 URI 代表，如 图 5.2 “Message Channel Pattern” 所示。

图 5.2. Message Channel Pattern



面向消息的组件

Apache Camel 中的以下面向消息的组件支持消息频道的注意：

- ActiveMQ
- JMS
- AMQP

ActiveMQ

在 ActiveMQ 中，消息通道由 队列或主题 表示。特定队列 QueueName 的端点 URI，其格式如下：

```
activemq:QueueName
```

特定主题的端点 URI TopicName 的格式如下：

```
activemq:topic:TopicName
```

例如，要将消息发送到队列 Foo.Bar，请使用以下端点 URI：

```
activemq:Foo.Bar
```

如需有关设置 ActiveMQ 组件的更多详细信息和说明，请参阅 Apache Camel 组件参考指南 中的 ActiveMQ。

JMS

Java 消息传递服务(JMS)是一个通用打包程序层，用于访问许多不同类型的消息系统（例如，您可以使用它来嵌套 ActiveMQ、MQSeries、Tibco、BEA、Sunic 等）。在 JMS 中，消息通道由队列或主题表示。特定队列 QueueName 的端点 URI，其格式如下：

```
jms:QueueName
```

特定主题的端点 URI TopicName 的格式如下：

```
jms:topic:TopicName
```

有关设置 JMS 组件的详情和说明，请参阅 Apache Camel 组件参考指南 中的 Jms。

AMQP

在 AMQP 中，消息通道由队列或主题表示。特定队列 QueueName 的端点 URI，其格式如下：

```
amqp:QueueName
```

特定主题的端点 URI TopicName 的格式如下：

```
amqp:topic:TopicName
```

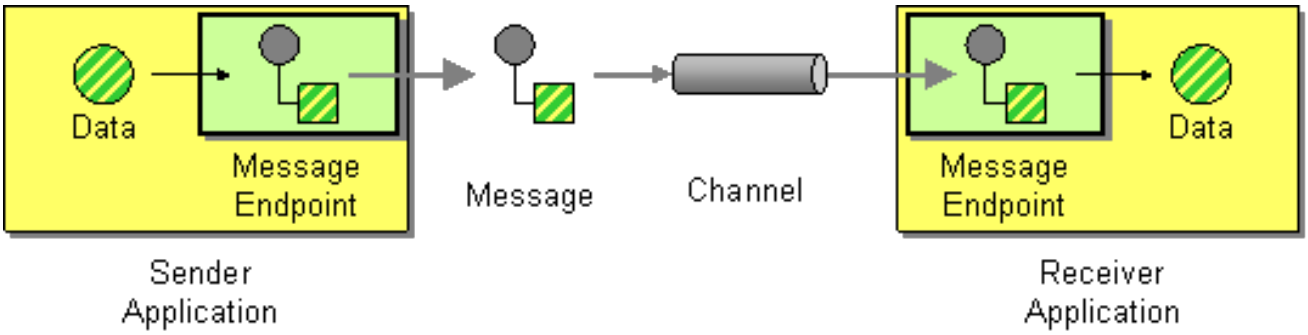
有关设置 AMQP 组件的详情和说明，请参阅 Apache Camel 组件参考指南 中的 Amqp。

5.3. MESSAGE ENDPOINT

概述

消息端点 是应用程序与消息传递系统间的接口。如 图 5.3 “Message Endpoint Pattern” 所示，您可以有一个发件人端点，有时也称为代理或服务消费者，它负责发送 In 消息和接收器端点，有时也称为端点或服务，该服务负责接收 In 消息。

图 5.3. Message Endpoint Pattern



端点类型

Apache Camel 定义两种基本端点类型：

- 消费者端点 在 Apache Camel 路由开始时出现，并读取来自传入频道的消息（等同于 接收器 端点）。
- Apache Camel 路由结束时 制作者端点 producer-latex Appears，并在消息中写入传出频道（等同于 发送方 端点）。可以使用多个制作者端点定义路由。

端点 URI

在 Apache Camel 中，端点由 端点 URI 代表，它通常会封装以下数据类型：

- 消费者端点 IFL Advertises 一个特定位置的端点 URI （例如，公开发送方可以连接到的服务）。另外，URI 可以指定消息源，如消息队列。端点 URI 可以包含用于配置端点的设置。
-

制作制作者端点的端点 URI 进行登录，包含要发送消息以及配置端点的设置的详细信息。在某些情况下，URI 指定远程接收器端点的位置；在其他情况下，目的地可以有一个抽象形式，如队列名称。

Apache Camel 中的端点 URI 有以下通用形式：

```
ComponentPrefix:ComponentSpecificURI
```

其中 component Prefix 是一个识别特定 Apache Camel 组件的 URI 前缀（有关所有支持组件的详细信息，请参阅 Apache Camel 组件参考 ）。URI(Pack SpecificURI)的其余部分具有特定组件定义的语法。例如，要连接到 JMS 队列 Foo.Bar，您可以定义类似如下的端点 URI：

```
jms:Foo.Bar
```

若要定义将消费者端点 file://local/router/messages/foo 直接连接到制作者 端点的路由，可以使用以下 Java DSL 片段：

```
from("file://local/router/messages/foo").to("jms:Foo.Bar");
```

另外，您可以在 XML 中定义相同的路由，如下所示：

```
<camelContext id="CamelContextID" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://local/router/messages/foo"/>
    <to uri="jms:Foo.Bar"/>
  </route>
</camelContext>
```

动态到

& lt;toD > 参数允许您使用串联在一起的一个或多个表达式向动态计算端点发送消息。

默认情况下，使用 Simple 语言计算端点。以下示例向标头定义的端点发送信息：

```
<route>
  <from uri="direct:start"/>
  <toD uri="${header.foo}"/>
</route>
```

在 Java DSL 中，同一命令的格式是：

```
from("direct:start")
  .toD("${header.foo}");
```

URI 也可以带有字面的前缀，如下例所示：

```
<route>
  <from uri="direct:start"/>
  <toD uri="mock:${header.foo}"/>
</route>
```

在 Java DSL 中，同一命令的格式是：

```
from("direct:start")
  .toD("mock:${header.foo}");
```

在上例中，如果 header.foo 的值为 orange，则 URI 将解析为 模拟：orange。

要使用简单以外的语言，您需要定义 language: 参数。请参阅 第 II 部分 “路由表达式和指定语言”。

使用不同语言的格式是使用语言 ：语言名称: 在 URI 中。例如，要使用 Xpath，请使用以下格式：

```
<route>
  <from uri="direct:start"/>
  <toD uri="language:xpath:/order/@uri/">
</route>
```

以下是 Java DSL 中相同的示例：

```
from("direct:start")
  .toD("language:xpath:/order/@uri");
```

如果没有指定 语言： 如果端点是一个组件名称。在某些情况下，组件和语言的名称相同，比如 xquery。

您可以使用 + 符号连接多个语言。在以下示例中，URI 是 Simple 和 Xpath 语言的组合。simple 是默

认值，因此不需要定义语言。+ 符号为 Xpath 指令后，使用 language:xpath 表示。

```
<route>
  <from uri="direct:start"/>
  <toD uri="jms:${header.base}+language:xpath:/order/@id"/>
</route>
```

在 Java DSL 中，格式如下：

```
from("direct:start")
  .toD("jms:${header.base}+language:xpath:/order/@id");
```

很多语言可以同时相互连接，只需使用 + 分开，并使用语言指定每个 语言：语言名称。

以下选项可用于 D ：

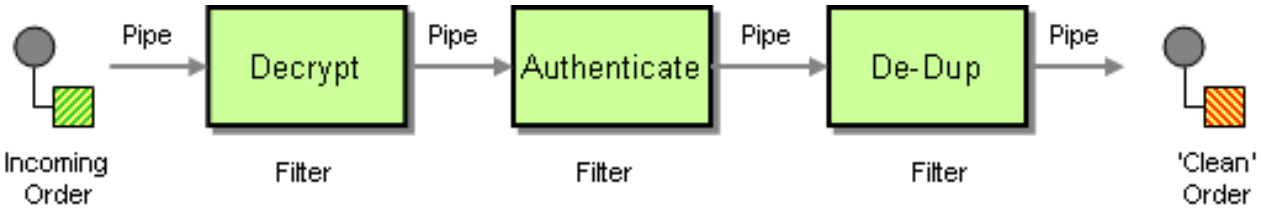
名称	默认值	描述
uri		必需：要使用的 URI。
pattern		设置一个在发送到端点时要使用的特定 Exchange Pattern。原始 MEP 随即被恢复。
cacheSize		配置 **ProducerCache** 的缓存大小，它将缓存制作者以供重复使用。默认缓存大小为 1000，如果未指定其他值，则将使用该大小。将值设为 -1 会完全关闭缓存。
ignoreInvalidEndpoint	**false**	指定是否忽略无法解析的端点 URI。如果禁用，Camel 将抛出一个指定无效端点 URI 的异常。

5.4. PIPES 和 FILTERS

概述

pipes 和过滤器 模式在 图 5.4 “pipes 和 Filters Pattern” 中显示的是，它通过创建过滤器链来构造路由的方法，其中单个过滤器的输出将反馈到管道中下一个过滤器的输入（与 UNIX pipe 命令相同）。管道方法的优点在于，它可让您编写服务（某些服务可能是 Apache Camel 应用程序外部）以创建更加复杂的消息处理。

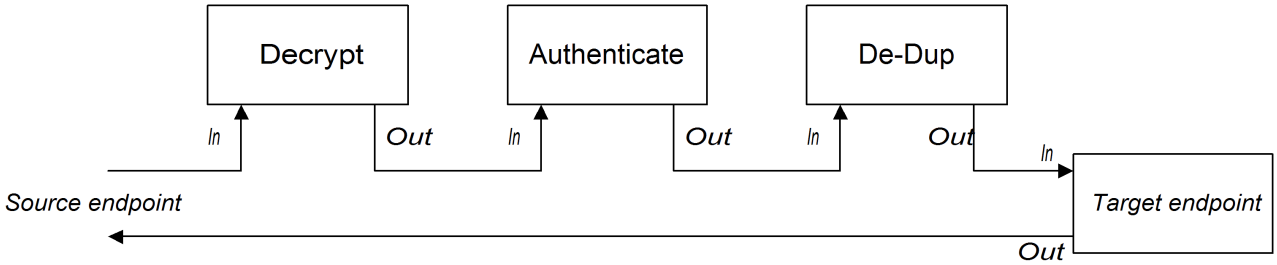
图 5.4. pipes 和 Filters Pattern



InOut Exchange 模式的管道

通常，管道中的所有端点都有一个输入（在消息中）和一个输出（输出），这意味着它们与 In Out 消息交换模式兼容。图 5.5 “InOut Exchanges 的管道” 中显示了通过 InOut 管道的典型消息流。

图 5.5. InOut Exchanges 的管道



管道将每个端点的输出连接到下一端点的输入。来自最终端点的 Out 消息会发回到原始调用者。您可以为这个管道定义路由，如下所示：

```
from("jms:RawOrders").pipeline("cxf:bean:decrypt", "cxf:bean:authenticate", "cxf:bean:dedup",
"jms:CleanOrders");
```

同一路由可以在 XML 中配置，如下所示：

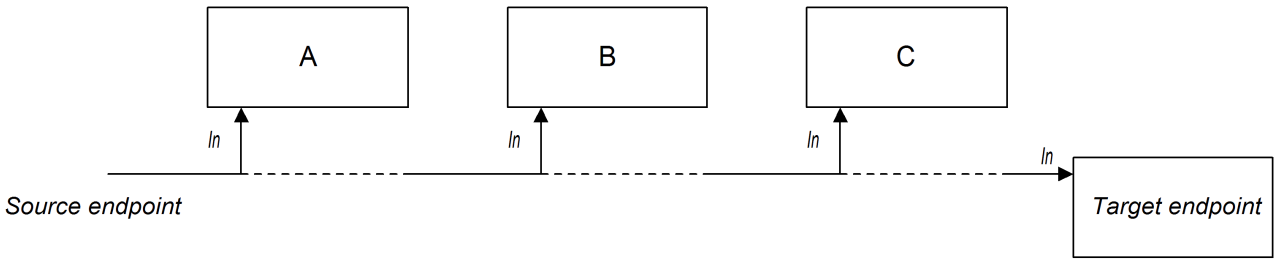
```
<camelContext id="buildPipeline" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:RawOrders"/>
    <to uri="cxf:bean:decrypt"/>
    <to uri="cxf:bean:authenticate"/>
    <to uri="cxf:bean:dedup"/>
    <to uri="jms:CleanOrders"/>
  </route>
</camelContext>
```

XML 中没有专用管道元素。和 与 元素的前一个组合有语义等同于管道。请参阅 “pipeline（）和 to（）DSL 命令的比较”一节。

InOnly 和 RobustInOnly Exchange 模式的管道

如果没有来自管道中端点的 Out 消息（如 InOnly 和 Robust InOnly 交换模式的情况），则管道不能以正常方式连接。在这种特殊情况下，管道通过将原始 In 消息的副本传递给管道中的每个端点，如图 5.6 “InOnly Exchanges 的管道” 所示。这个管道类型等同于带有固定目的地的接收者列表（请参阅第 8.3 节 “接收者列表”）。

图 5.6. InOnly Exchanges 的管道



此管道的路由使用与 InOut 管道（Java DSL 或 XML）相同的语法来定义。

pipeline（）和 to（）DSL 命令的比较

在 Java DSL 中，您可以使用以下语法之一定义管道路由：

- 使用 pipeline（）处理器命令 the the pipeline processor 使用管道处理器构建管道路由，如下所示：

```
from(SourceURI).pipeline(FilterA, FilterB, TargetURI);
```

- 使用 to（）命令 abrt- theUse the to（） 命令构建管道路由，如下所示：

```
from(SourceURI).to(FilterA, FilterB, TargetURI);
```

另外，您可以使用等同的语法：

```
from(SourceURI).to(FilterA).to(FilterB).to(TargetURI);
```

在使用 to（） 命令语法时要谨慎，因为它 并不总是 等同于管道处理器。在 Java DSL 中，可以通过以上命令在路由中修改 to（） 的含义。例如，当 multicast（） 命令前面是 to（） 命令，它会将列出的端点绑定到多播模式，而不是管道模式（请参阅 第 8.13 节 “多播”）。

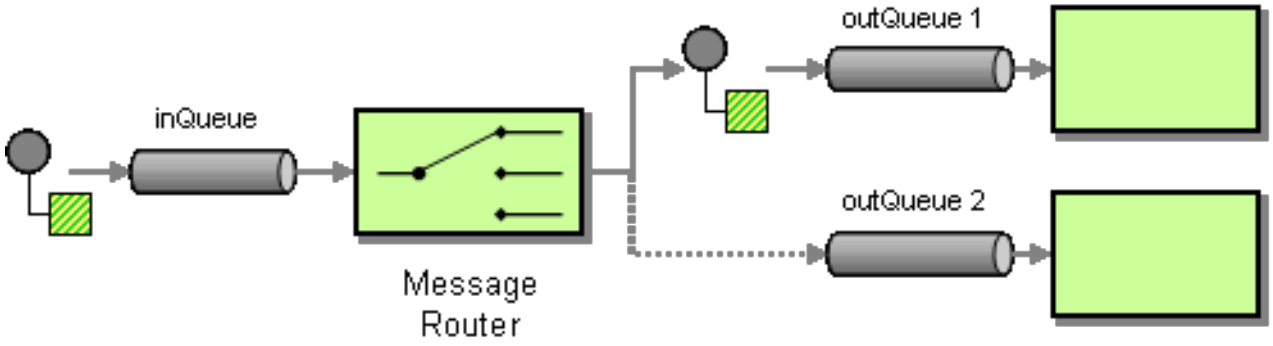
5.5. MESSAGE ROUTER

概述

图 5.7 “Message Router Pattern” 中显示的消息路由器是根据特定决策条件消耗来自单一消费者端点的消息的过滤器类型。邮件路由器仅关注重定向消息，它不会修改消息内容。

但是，默认情况下，当 Camel 将消息交换路由到接收者端点时，它发送是原始交换对象的相对副本。在虚构副本中，原始交换的元素（如消息正文、标头和附件）仅由引用进行复制。通过发送能够复制资源，Camel 可以优化性能。但是，由于这些应该副本都是链接的，因此当 Camel 将消息路由到多个端点时，折衷是，您失去应用自定义逻辑来将自定义逻辑复制到路由到不同接收方的功能。有关如何启用 Camel 将唯一消息版本路由到不同端点的信息，请参阅 "应用自定义处理到传出消息"。

图 5.7. Message Router Pattern



可以使用 choice（） 处理器在 Apache Camel 中轻松实施消息路由器，其中每个替代目标端点都可以使用 when（） 子clause（有关所选处理器的详细信息，请参阅 第 1.5 节 “处理器”）。

Java DSL 示例

以下 Java DSL 示例演示了如何根据 foo 标头的内容将消息路由到三个替代目的地（ seda:a、seda:b、seda:b ）：

```
from("seda:a").choice()
    .when(header("foo").isEqualTo("bar")).to("seda:b")
    .when(header("foo").isEqualTo("cheese")).to("seda:c")
    .otherwise().to("seda:d");
```

XML 配置示例

以下示例演示了如何在 XML 中配置相同的路由：

```
<camelContext id="buildSimpleRouteWithChoice" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
```

```
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

无其他选择

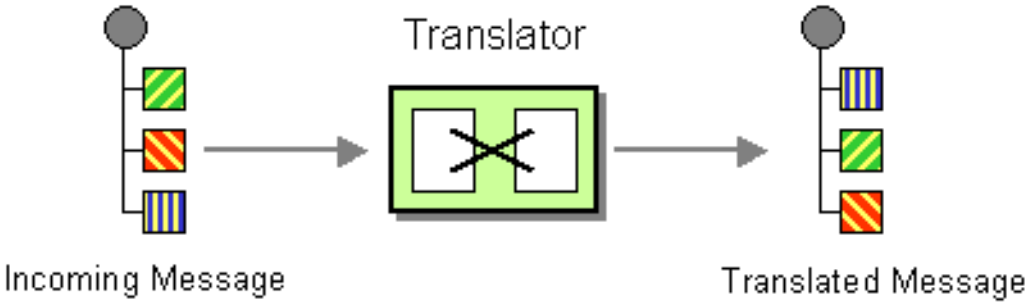
如果您使用没有 otherwise（） 子句的 choice（），则默认丢弃任何不匹配的交换。

5.6. MESSAGE TRANSLATOR

概述

图 5.8 “Message Translator Pattern” 中显示的消息转换器模式描述了一个修改信息内容的组件，将其转换为不同的格式。您可以使用 Apache Camel 的 Bean 集成功能来执行消息转换。

图 5.8. Message Translator Pattern



Bean 集成

您可以使用 bean 集成转换消息，允许您在任何注册的 Bean 上调用方法。例如，要在带有 ID 的 Bean、myTransformerBean 上调用方法 myMethodName（） ：

```
from("activemq:SomeQueue")
  .beanRef("myTransformerBean", "myMethodName")
  .to("mqseries:AnotherQueue");
```

在 Spring XML 文件或 JNDI 中定义 myTransformer Bean。如果省略 beanRef（） 中的 method name 参数，则 bean 集成将尝试推断方法名称通过检查消息交换来调用。

您还可以添加自己的显式处理器 实例 以执行转换，如下所示：

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

或者，您可以使用 DSL 来显式配置转换，如下所示：

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

您还可以使用 模板 (template)来消耗来自一个目的地的消息，将其转换为 Velocity 或 XQuery 等内容，然后将其发送到另一个目的地。例如，使用 InOnly Exchange 模式（单向消息）：

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

如果要使用 InOut （请求）语义来处理在 ActiveMQ 上的请求，使用模板生成的响应，那么您可以使用类似以下内容的路由来回 JMSReplyTo 目的地：

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

5.7. 消息历史

概述

Message History 模式允许您在松散耦合系统中分析和调试消息流。如果您在消息中附加了消息历史记录，则会显示一条从其原始消息传递的所有应用程序的列表。

在 Apache Camel 中，使用 getTracedRouteNodes 方法，您可以使用 Tracer 跟踪消息流或使用 UnitOfWork 中的 Java API 访问信息。

限制日志中的字符长度

当您使用日志记录机制运行 Apache Camel 时，它可让您及时记录消息及其内容。

有些消息可能包含非常大的有效负载。默认情况下，Apache Camel 将关闭日志消息，并只显示前 1000 个字符。例如，它以以下方式显示以下日志：

```
[DEBUG ProducerCache  - >>>> Endpoint[direct:start] Exchange[Message:
01234567890123456789... [Body clipped after 20 characters, total length is 1000]
```

当 Apache Camel clips log 中的正文时，您可以自定义限制。您还可以设置零或负值，如 -1，表示消息正文没有被记录。

使用 Java DSL 自定义限制

您可以使用 Java DSL 在 Camel 属性中设置限制。例如，

```
context.getProperties().put(Exchange.LOG_DEBUG_BODY_MAX_CHARS, "500");
```

使用 Spring DSL 自定义限制

您可以使用 Spring DSL 在 Camel 属性中设置限制。例如，

```
<camelContext>
  <properties>
    <property key="CamelLogDebugBodyMaxChars" value="500"/>
  </properties>
</camelContext>
```

第 6 章 消息传递频道

摘要

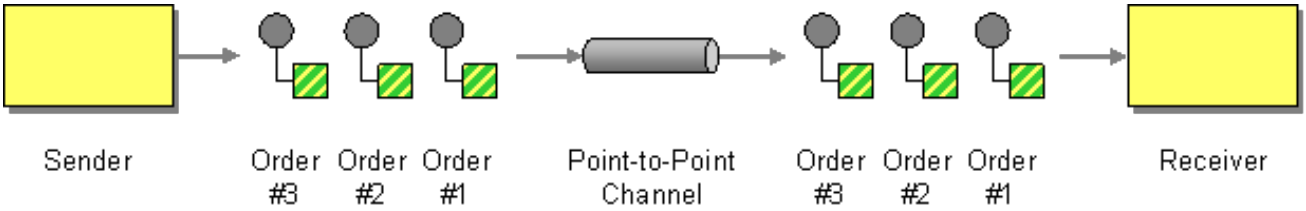
消息传递频道为消息传递应用程序提供 plumbing。本章论述了消息传递系统中可用的不同类型的消息通道，以及它们所扮演的角色。

6.1. 点到点频道

概述

在 图 6.1 “指向点频道模式” 中显示的 点对点频道 是保证只有一个接收器消耗任何给定信息的消息 频道。这与 发布与订阅频道 不同，后者允许多个接收方使用相同的消息。特别是，通过 publish-subscribe 频道，多个接收方可以订阅同一频道。如果多个接收器竞争使用一条消息，则最多可使用消息频道，以确保只有一个接收器实际消耗这个信息。

图 6.1. 指向点频道模式



支持点到点频道的组件

以下 Apache Camel 组件支持点到点频道模式：

- JMS
- ActiveMQ
- SEDA
- JPA
- XMPP

JMS

在 JMS 中，点到点频道由 队列 表示。例如，您可以为名为 Foo.Bar 的 JMS 队列指定端点 URI，如下所示：

```
jms:queue:Foo.Bar
```

限定符 queue: 是可选的，因为 JMS 组件默认创建一个队列端点。因此，您还可以指定以下等同的端点 URI：

```
jms:Foo.Bar
```

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南中的 Jms。

ActiveMQ

在 ActiveMQ 中，点到点频道由队列表示。例如，您可以为名为 Foo.Bar 的 ActiveMQ 队列指定端点 URI，如下所示：

```
activemq:queue:Foo.Bar
```

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 ActiveMQ。

SEDA

Apache Camel Staged 事件架构(SEDA)组件使用块队列实施。如果要创建一个 Apache Camel 应用程序 内部的 轻量级点到点频道，请使用 SEDA 组件。例如，您可以为名为 SedaQueue 的 SEDA 队列指定端点 URI，如下所示：

```
seda:SedaQueue
```

JPA

Java Persistence API(JPA)组件是一种 EJB 3 持久性标准，用于将实体 Bean 写入数据库。如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 JPA。

XMPP

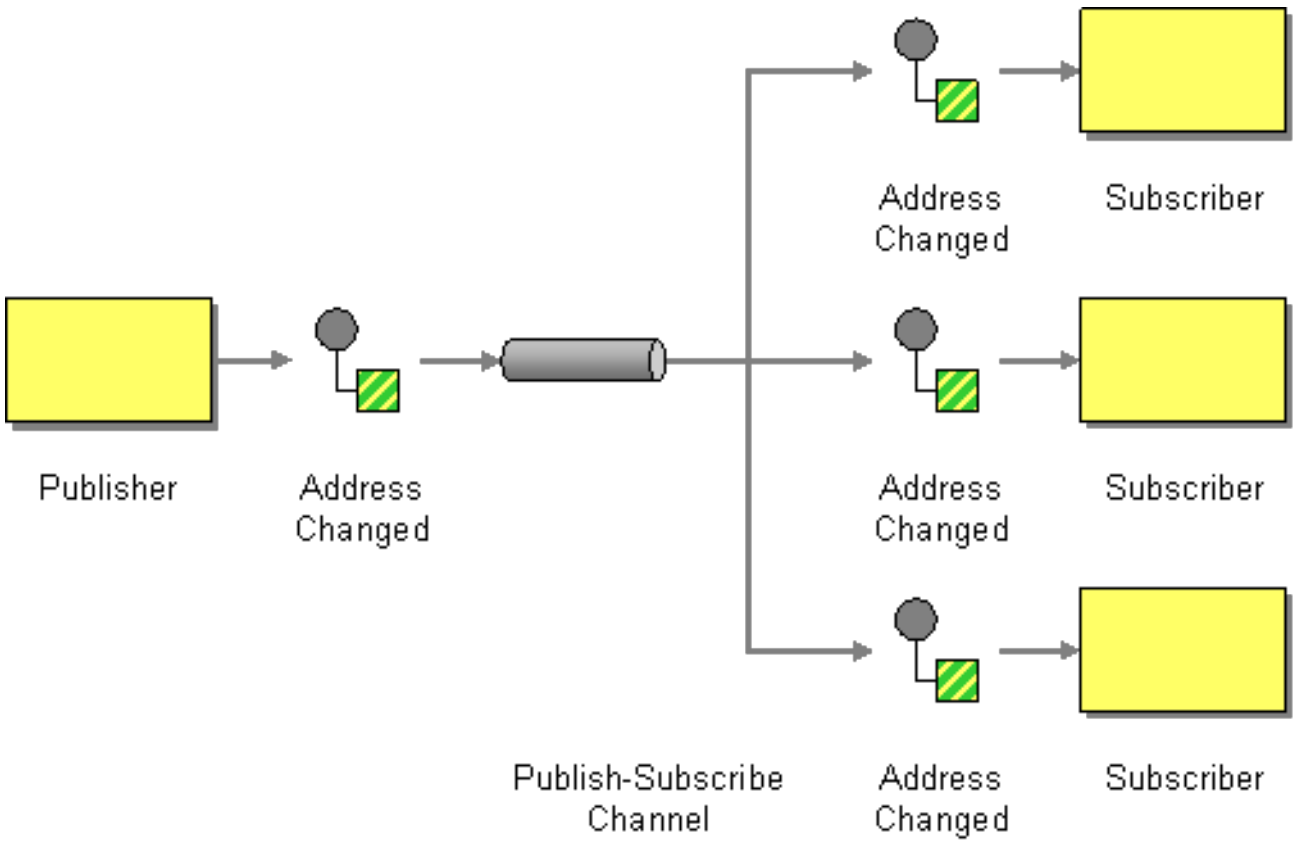
XMP(Jabber)组件支持点对点频道模式。如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 XMPP。

6.2. PUBLISH-SUBSCRIBE CHANNEL

概述

在 图 6.2 “发布 Subscribe Channel Pattern” 中显示的 发布订阅频道 是 第 5.2 节 “Message Channel”，它允许多个订阅者消耗任何给定消息。这与 第 6.1 节 “点到点频道” 不同。发布订阅频道经常用作向多个订阅者广播事件或通知的方法。

图 6.2. 发布 Subscribe Channel Pattern



支持发布订阅频道的组件

以下 Apache Camel 组件支持发布订阅频道模式：

- JMS
- ActiveMQ

- XMPP

- 适用于同一 CamelContext 的使用 SEDA 的 SEDA，可以在 pub-sub 中工作，但允许多个消费者。

- 请参阅 Apache Camel 组件参考指南 中的 VM 作为 SEDA，但可用于同一 JVM。

JMS

在 JMS 中，发布订阅频道由一个 主题 表示。例如，您可以为名为 StockQuotes 的 JMS 主题指定端点 URI，如下所示：

```
jms:topic:StockQuotes
```

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南中的 Jms。

ActiveMQ

在 ActiveMQ 中，发布订阅频道由一个主题表示。例如，您可以为名为 StockQuotes 的 ActiveMQ 主题指定端点 URI，如下所示：

```
activemq:topic:StockQuotes
```

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 ActiveMQ。

XMPP

XMPP(Jabber)组件在组通信模式中使用时支持 publish-subscribe channel 模式。如需了解更多详细信息，请参阅 Apache Camel 组件参考指南中的 Xmpp。

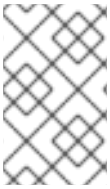
静态订阅列表

如果您愿意，也可以在 Apache Camel 应用程序本身中实施发布订阅逻辑。简单方法是定义一个 静态订阅列表，其中目标端点均在路由末尾明确列出。但是，这种方法不如 JMS 或 ActiveMQ 主题。

Java DSL 示例

以下 Java DSL 示例演示了如何使用单个发布者、seda:a 和三个订阅者、seda:b、seda:c 和 seda:d 来模拟发布订阅频道：

```
from("seda:a").to("seda:b", "seda:c", "seda:d");
```



注意

这只适用于 InOnly 消息交换模式。

XML 配置示例

以下示例演示了如何在 XML 中配置相同的路由：

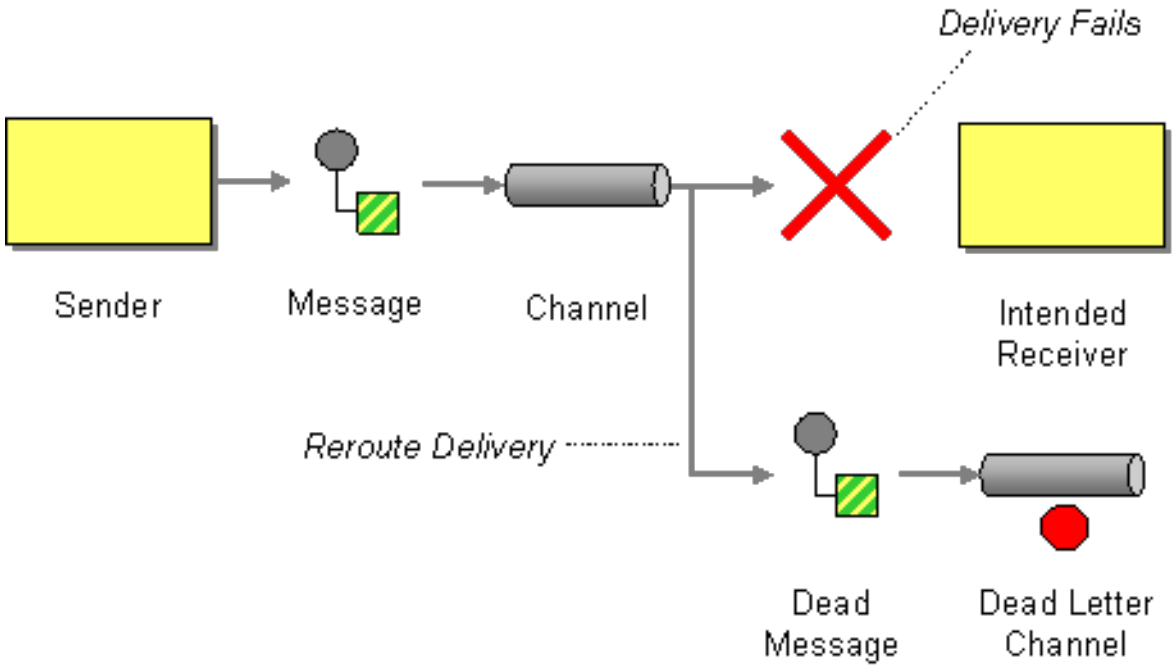
```
<camelContext id="buildStaticRecipientList" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

6.3. 死信频道

概述

死信频道 模式（如 图 6.3 “死的频道模式” 所示）描述了当消息系统无法向预期收件人发送邮件时要执行的操作。这包括重试交付等功能，如果交付最终失败，则向死信频道发送消息，以存档未发送的消息。

图 6.3. 死的频道模式



在 Java DSL 中创建死信频道

以下示例演示了如何使用 Java DSL 创建死信频道：

```
errorHandler(deadLetterChannel("seda:errors"));
from("seda:a").to("seda:b");
```

其中 errorHandler（） 方法是一个 Java DSL 拦截器，这意味着当前路由构建器中定义的所有路由都受到此设置的影响。deadLetterChannel（） 方法是一个 Java DSL 命令，它使用指定的目标端点 seda:errors 创建新的死信频道。

errorHandler（） 拦截器提供了一种 catch- all 机制，用于处理所有错误类型。如果要应用更精细的方法来例外处理，您可以使用 onException 子句（请参阅 “onException 子句”一节）。

XML DSL 示例

您可以在 XML DSL 中定义死信频道，如下所示：

```
<route errorHandlerRef="myDeadLetterErrorHandler">
  ...
</route>

<bean id="myDeadLetterErrorHandler"
class="org.apache.camel.builder.DeadLetterChannelBuilder">
    <property name="deadLetterUri" value="jms:queue:dead"/>
```

```
    <property name="redeliveryPolicy" ref="myRedeliveryPolicyConfig"/>
</bean>

<bean id="myRedeliveryPolicyConfig" class="org.apache.camel.processor.RedeliveryPolicy">
    <property name="maximumRedeliveries" value="3"/>
    <property name="redeliveryDelay" value="5000"/>
</bean>
```

重新传送策略

通常，如果发送尝试失败，您不会直接向 dead letter 频道发送邮件。相反，您会重新尝试交付到一些最大限制，且在所有重新发送尝试失败时，您会将消息发送到 dead letter 频道。要自定义消息重新传送，您可以将死信频道配置为具有 重新传送策略。例如，要指定最多两个重新传送尝试，并在交付尝试之间应用 exponential backoff 算法到时间延迟，您可以配置死信频道，如下所示：

```
errorHandler(deadLetterChannel("seda:errors").maximumRedeliveries(2).useExponentialBackOff());
from("seda:a").to("seda:b");
```

您可以通过调用链中的相关方法在 dead letter 频道上设置重新传送选项（链中每个方法都将引用当前 RedeliveryPolicy 对象）。表 6.1 “重新传送策略设置” 总结了可用于设置重新传送策略的方法。

表 6.1. 重新传送策略设置

方法签名	默认	描述
allowRedeliveryWhileStopping()	**true**	控制在安全关闭期间或路由停止期间是否尝试重新发送。在启动停止时，已经进行的交付不会中断。
BackOffMultiplier（倍数）	**2**	如果启用了 exponential backoff，让 **m** 设置为 backoff 数，并让**d** 为初始延迟。然后重新发送尝试的顺序如下： d, m*d, m*m*d, m*m*m*d, ...
collisionAvoidancePercent(double collisionAvoidancePercent)	**15**	如果启用了冲突，请让 冲突避免了百分比。冲突策略随后调整下一个延迟，最大为当前值的加号/减去 **p%**。

方法签名	默认	描述
deadLetterHandleNewException	**true**	Camel 2.15：指定是否在死信频道中处理消息时是否处理异常。如果为 **true**，则会处理异常并记录在 WARN 级别（因此保证死信频道完成）。如果为 **false**，则异常不会处理，因此死信频道失败，并传播新的异常。
delayPattern(String delayPattern)	**无**	Apache Camel 2.0：请参阅 "redeliver delay pattern"一节。
disableRedelivery()	**true**	Apache Camel 2.0：禁用重新传送功能。要启用重新传送，将 **maximumRedeliveries（）** 设置为正整数值。
handled（布尔值已处理）	**true**	Apache Camel 2.0：如果为 **true**，则当消息移动到 dead letter 频道时清除当前异常；如果为 **false**，则异常将传回给客户端。
initialRedeliveryDelay(long initialRedeliveryDelay)	**1000**	指定在尝试第一个重新发送前的延迟（以毫秒为单位）。
logNewException	**true**	指定是否在 WARN 级别登录，当在死信频道引发异常时。
logStackTrace(boolean logStackTrace)	**false**	Apache Camel 2.0：如果为 **true**，则 JVM 堆栈追踪包含在错误日志中。
maximumRedeliveries(int maximumRedeliveries)	**0**	Apache Camel 2.0：最大交付尝试次数.
maximumRedeliveryDelay(long maxDelay)	**60000**	Apache Camel 2.0：当使用指数级 backoff 策略时（请参阅 **useExponentialBackOff（）**）时，重建延迟可能会在不限制的情况下增加。此属性对重新发送延迟（以毫秒为单位）实施上限。
onRedelivery（处理器）	**无**	Apache Camel 2.0：配置在每次重新发送尝试之前调用的处理器。
redeliveryDelay(long int)	**0**	Apache Camel 2.0：指定重新传送尝试之间的延时（以毫秒为单位）。Apache Camel 2.16.0 : 默认重新传送延迟为一秒钟。

方法签名	默认	描述
retriesExhaustedLogLevel(LoggingLevel logLevel)	**LoggingLevel.ERROR**	Apache Camel 2.0：指定日志记录级别，在其中记录发送失败（指定为 **org.apache.camel.LoggingLevel** constant）。
retryAttemptedLogLevel(LoggingLevel logLevel)	**LoggingLevel.DEBUG**	Apache Camel 2.0：指定日志记录级别，在其中重新发送尝试（指定为 **org.apache.camel.LoggingLevel** constant）。
useCollisionAvoidance()	**false**	启用冲突可避免，这会在 backoff 计时中添加一些随机性，以减少竞争概率。
useOriginalMessage()	**false**	Apache Camel 2.0：如果启用了此功能，则发送到 dead letter 频道的消息是 **原始消息** 交换的副本，因为它存在于路由的开头（在 **from（）** 节点上）。
useExponentialBackOff()	**false**	启用 exponential backoff。

重新传送标头

如果 Apache Camel 尝试重新发送一条信息，它会在 In 消息中自动设置 表 6.2 “死 Letter Redelivery Headers” 中描述的标头。

表 6.2. 死 Letter Redelivery Headers

标头名称	类型	描述
CamelRedeliveryCounter	**整数**	Apache Camel 2.0：计数发送尝试失败的次数。**Exchange.REDELIVERY_COUNTER** 中也设置了这个值。
CamelRedelivered	布尔值	Apache Camel 2.0: True（如果进行了一个或多个重新发送尝试）。此值在 **Exchange.REDELIVERED** 中设定。

标头名称	类型	描述
CamelRedeliveryMaxCounter	**整数**	Apache Camel 2.6：包含最大重新传送设置（也在 **Exchange.REDELIVERY_MAX_COUNTER** 交换属性中设置）。如果使用 **retryWhile**，或者配置了无限重新传送，则缺少此标头。

重新传送交换属性

如果 Apache Camel 尝试恢复信息，它会自动设置 表 6.3 “重新传送交换属性” 中描述的交换属性。

表 6.3. 重新传送交换属性

Exchange Property Name	类型	描述
Exchange.FAILURE_ROUTE_ID	**字符串**	提供失败的路由 ID。这个属性的字面名称是 **CamelFailureRouteId**。

使用原始消息

自 Apache Camel 2.0 起，由于交换对象通过路由，因此交换对象会被修改，所以当引发异常时，这个交换不一定要存储在死信频道中的副本。在许多情况下，最好在路由开始时记录到达路由的消息，然后再记录路由受到任何转换的转换。例如，请考虑以下路由：

```
from("jms:queue:order:input")
    .to("bean:validateOrder");
    .to("bean:transformOrder")
    .to("bean:handleOrder");
```

前面的路由会侦听传入的 JMS 消息，然后使用 Beans 序列处理消息：validateOrder、transformOrder 和 handleOrder。但是，当发生错误时，我们不知道该消息所处的状态。转换 操作程序或之后是否发生错误？我们可以通过启用 useOriginalMessage 选项来确保来自 jms:queue:order:input 的原始消息记录到 dead letter 频道：

```
// will use original body
errorHandler(deadLetterChannel("jms:queue:dead")
    .useOriginalMessage().maximumRedeliveries(5).redeliveryDelay(5000);
```

redeliver delay pattern

可作为 Apache Camel 2.0 使用 delayPattern 选项为重新传送计数的特定范围指定延迟。delay 模式具有以下语法： limit1:delay1;limit2:delay2;limit3:delay3;... ，其中每个 delayN 应用到红色的 delayN PERC redeliveryCount < limitN+1

例如，假设模式为 5:1000;10:5000;20:20000，它定义了三个组，并导致以下重新发送延迟：

- 尝试编号 1.4 = 0 毫秒（作为第一个组以 5 开始）。
- 尝试编号 5..9 = 1000 毫秒（第一个组）。
- 尝试编号 10.19 = 5000 毫秒（第二个组）。
- 尝试编号 20.. = 20000 毫秒（最后一个组）。

您可以启动具有限制 1 的组来定义启动延迟。例如，1:1000;5:5000 会产生以下重新传送延迟：

- 尝试编号 1.4 = 1000 millis（第一个组）
- 尝试编号 5.. = 5000 millis（最后一个组）

不要求下一个延迟应高于上述延迟，您可以使用您喜欢的任何延迟值。例如，延迟模式 1:5000;3:1000，以 5 秒延迟开头，然后将延迟减少到 1 秒。

哪一端点失败？

当 Apache Camel 路由消息时，它会更新包含 Exchange 发送到 的最后一个 端点的 Exchange 属性。因此，您可以使用以下代码获取当前交换最新目的地的 URI：

```
// Java
String lastEndpointUri = exchange.getProperty(Exchange.TO_ENDPOINT, String.class);
```

这里的 Exchange.TO_ENDPOINT 是一个等于 CamelToEndpoint 的字符串。当 Camel 发送消息 到任何 端点时，都会更新此属性。

如果在路由过程中发生错误，并且交换被移到死信队列中，Apache Camel 将额外设置名为 CamelFailureEndpoint 的属性，该属性标识在发生错误之前交换的最后一个目的地。因此，您可以使用以下代码从死信队列中访问失败端点：

```
// Java
String failedEndpointUri = exchange.getProperty(Exchange.FAILURE_ENDPOINT, String.class);
```

这里的 Exchange.FAILURE_ENDPOINT 是一个字符串常量，相当于 CamelFailureEndpoint。

注意

这些属性保留在当前交换中设置的，即使给定目标端点完成处理后失败也是如此。例如，请考虑以下路由：

```
from("activemq:queue:foo")
.to("http://someserver/somepath")
.beanRef("foo");
```

现在假定 foo an 中出现失败。在这种情况下，Exchange.TO_ENDPOINT 属性和 Exchange.FAILURE_ENDPOINT 属性仍包含该值。

onRedelivery 处理器

当死信频道执行红色时，可以在每次重新发送尝试 前 配置只执行的处理器。这可用于您需要在消息被红色前修改消息的情况。

例如，以下死信频道被配置为在红色发生交换前调用 MyRedeliverProcessor ：

```
// we configure our Dead Letter Channel to invoke
// MyRedeliveryProcessor before a redelivery is
// attempted. This allows us to alter the message before
errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(5)
        .onRedelivery(new MyRedeliverProcessor())
        // setting delay to zero is just to make unit teting faster
        .redeliveryDelay(0L));
```

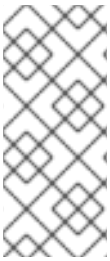
MyRedeliveryProcessor 进程在其中实现，如下所示：

```
// This is our processor that is executed before every redelivery attempt
// here we can do what we want in the java code, such as altering the message
public class MyRedeliverProcessor implements Processor {

    public void process(Exchange exchange) throws Exception {
        // the message is being redelivered so we can alter it

        // we just append the redelivery counter to the body
        // you can of course do all kind of stuff instead
        String body = exchange.getIn().getBody(String.class);
        int count = exchange.getIn().getHeader(Exchange.REDELIVERY_COUNTER, Integer.class);

        exchange.getIn().setBody(body + count);

        // the maximum redelivery was set to 5
        int max = exchange.getIn().getHeader(Exchange.REDELIVERY_MAX_COUNTER,
Integer.class);
        assertEquals(5, max);
    }
}
```

在关闭或停止过程中控制重新发送

如果您停止路由或启动安全关闭，则错误处理程序的默认行为将继续尝试重新发送。因为这通常不是所需的行为，所以可以选择通过在关闭或停止期间禁用重新传送，方法是将 allowRedeliveryWhileStopping 选项设置为 false，如下例所示：

```
errorHandler(deadLetterChannel("jms:queue:dead")
    .allowRedeliveryWhileStopping(false)
    .maximumRedeliveries(20)
    .redeliveryDelay(1000)
    .retryAttemptedLogLevel(LoggingLevel.INFO));
```



注意

出于向后兼容的原因，allowRedeliveryWhileStopping 选项默认为 true。但是，在主动关闭过程中，重新传送始终会被隐藏，这与这个选项设置无关（例如，安全关闭超时）。

使用ExceptionOccurred Processor

死 Letter 通道支持 onExceptionOccurred 处理器，允许在发生异常后对消息进行自定义处理。您还可以将其用于自定义日志记录。从 onExceptionOccurred 处理器中引发的新异常都会记录为 WARN 并忽略，而不是覆盖现有异常。

onRedelivery 处理器和在ExceptionOccurred processor 之间的区别在于，您可以在重新发送尝试

前完全处理前一个。但是，在发生异常后不会立即发生。例如，如果您将错误处理程序配置为在重新传送尝试之间执行 5 秒的延迟，则稍后在出现异常后调用重新发送处理器。

以下示例说明了如何在出现异常时执行自定义日志记录。您需要配置 onExceptionOccurred 以使用自定义处理器。

```
errorHandler(defaultErrorHandler().maximumRedeliveries(3).redeliveryDelay(5000).onExceptionOccurred(myProcessor));
```

onException 子句

您无需在路由构建器中使用 errorHandler（） 拦截器，您可以定义一系列 onException（） 子句，以定义不同的重新传送策略，以及为各种异常类型定义不同的死信频道。例如，若要为每个 NullPointerException、IOException 和 Exception 类型定义不同的行为，您可以使用 Java DSL 在路由构建器中定义以下规则：

```
onException(NullPointerException.class)
    .maximumRedeliveries(1)
    .setHeader("messageInfo", "Oh dear! An NPE.")
    .to("mock:npe_error");

onException(IOException.class)
    .initialRedeliveryDelay(5000L)
    .maximumRedeliveries(3)
    .backOffMultiplier(1.0)
    .useExponentialBackOff()
    .setHeader("messageInfo", "Oh dear! Some kind of I/O exception.")
    .to("mock:io_error");

onException(Exception.class)
    .initialRedeliveryDelay(1000L)
    .maximumRedeliveries(2)
    .setHeader("messageInfo", "Oh dear! An exception.")
    .to("mock:error");

from("seda:a").to("seda:b");
```

如果通过串联重新传送策略方法来指定重新传送选项（如 表 6.1 “重新传送策略设置”中列出的），再使用 to（） DSL 命令指定 dead letter 频道的端点。您也可以在 Exception（） 子句中调用其他 Java DSL 命令。例如，前面的示例调用 setHeader（） 在名为 messageInfo 的消息标头中记录一些错误详情。

在本例中，NullPointerException 和 IOException 异常类型是特别配置的。所有其他例外类型由通用 Exception 异常拦截器进行处理。默认情况下，Apache Camel 应用最符合所引发异常异常的异常拦截器。如果无法找到完全匹配，它将尝试匹配最接近的基本类型，以此类推。最后，如果没有其他拦截器匹配，则 Exception 类型的拦截器与所有剩余的例外匹配。

OnPrepareFailure

在将交换传递给死信队列之前，您可以使用 onPrepare 选项来允许一个自定义处理器准备交换。它可让您添加有关交换的信息，如交换失败的原因。例如，以下处理器添加带有异常消息的标头。

```
public class MyPrepareProcessor implements Processor {
    @Override
    public void process(Exchange exchange) throws Exception {
        Exception cause = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);
        exchange.getIn().setHeader("FailedBecause", cause.getMessage());
    }
}
```

您可以将错误处理程序配置为使用处理器，如下所示：

```
errorHandler(deadLetterChannel("jms:dead").onPrepareFailure(new MyPrepareProcessor()));
```

但是，Prepare 选项也可以使用默认错误处理程序来使用。

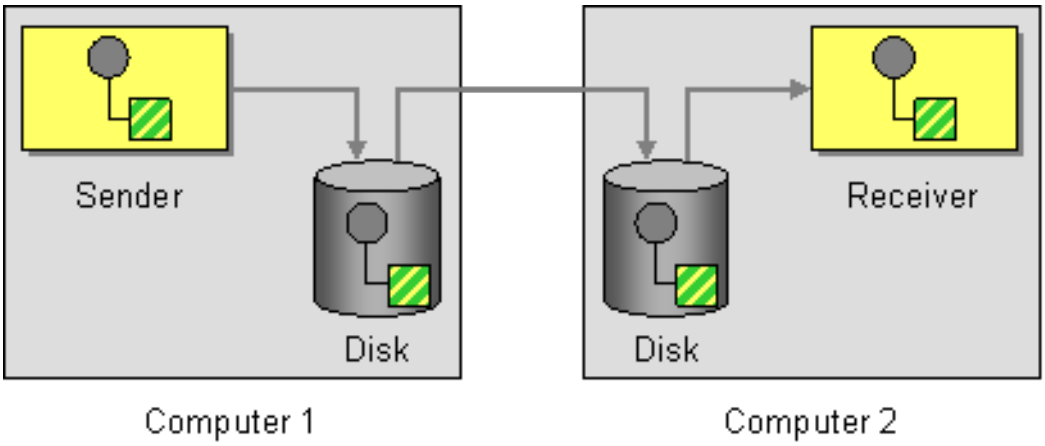
```
<bean id="myPrepare"
class="org.apache.camel.processor.DeadLetterChannelOnPrepareTest.MyPrepareProcessor"/>

<errorHandler id="dlc" type="DeadLetterChannel" deadLetterUri="jms:dead"
onPrepareFailureRef="myPrepare"/>
```

6.4. 保证交付

概述

保证交付 意味着，当消息被放入消息频道后，消息系统会保证该消息将到达其目的地，即使应用程序的一部分也失败。通常，消息传递系统实施保证交付模式（如 图 6.4 “保证交付模式”），在尝试将它们发送到目的地前，先向持久性存储写入信息。

图 6.4. 保证交付模式



支持保证交付的组件

以下 Apache Camel 组件支持保证交付模式：

- JMS
- ActiveMQ
- ActiveMQ Journal
- Apache Camel 组件 参考指南中的文件组件

JMS

在 JMS 中，deliveryPersistent 查询选项指示是否启用了消息的持久性存储。通常，设置这个选项是不需要的，因为默认行为是启用持续交付。要配置保证交付的所有详情，需要在 JMS 提供程序上设置配置选项。这些详情会根据您使用的 JMS 供应商的不同而有所不同。例如, MQSeries、TibCo、BEA、Sunic 等，它们都提供各种服务质量来支持保证交付。

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 > 中的 Jms。

ActiveMQ

在 ActiveMQ 中，消息持久性默认为启用。从版本 5 开始，ActiveMQ 使用 AMQ 消息存储作为默认的持久性机制。您可以使用几种不同方法在 ActiveMQ 中放入消息持久性。

最简单的选项（与 图 6.4 “保证交付模式”不同）是在中央代理中启用持久性，然后使用可靠的协议连接到该代理。将消息发送到中央代理后，可以保证传递给消费者。例如，在 Apache Camel 配置文件 META-INF/spring/camel-context.xml 中，您可以配置 ActiveMQ 组件以使用 OpenWire/TCP 协议连接到中央代理，如下所示：

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://somehost:61616"/>
  </bean>
  ...
</beans>
```

如果您希望实施在发送到远程端点前本地存储的信息的构架（类似于 图 6.4 “保证交付模式”），请通过在 Apache Camel 应用程序中实例化嵌入式代理来执行此操作。实现这一点的一个简单方法是使用 ActiveMQ Peer-to-Peer 协议，可隐式创建嵌入式代理来与其他对等端点通信。例如，在 camel-context.xml 配置文件中，您可以将 ActiveMQ 组件配置为连接到组 GroupA 中的所有 peer，如下所示：

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="peer://GroupA/broker1"/>
  </bean>
  ...
</beans>
```

其中 broker1 是嵌入的代理的代理名称（组中的其他对等点应使用不同的代理名称）。Peer-to-Peer 协议的一个限制功能是它依赖于 IP 多播来在其组中定位其他对等点。这使得用于广泛的区域网络（在一些没有启用 IP 多播的本地网络中）的使用变得不重要。

在 ActiveMQ 组件中创建嵌入式代理的更灵活方法是利用 ActiveMQ 的 VM 协议，它连接到嵌入式代理实例。如果所需名称的代理不存在，VM 协议会自动创建。您可以使用此机制创建带有自定义配置的嵌入式代理。例如：

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="vm://broker1?brokerConfig=xbean:activemq.xml"/>
  </bean>
  ...
</beans>
```

其中 activemq.xml 是一个 ActiveMQ 文件，用于配置嵌入的代理实例。在 ActiveMQ 配置文件中，

您可以选择启用以下持久性机制之一：

- AMQ 持久性（默认） 来快速可靠的 ActiveMQ 消息存储。详情请查看 amq PersistenceAdapter 和 AMQ Message Store。
- JDBC 持久性 abrt- JDBC 使用 JDBC 将消息存储在任何 JDBC 兼容数据库中。详情请参阅 jdbcpersistenceAdapter 和 ActiveMQ Persistence。
- 在滚动日志文件中存储消息的日志持久性机制。详情请参阅 journal PersistenceAdapter 和 ActiveMQ Persistence。
- Kaha persistence 此文件 - 为 ActiveMQ 开发的持久性机制。详情请参阅 kaha PersistenceAdapter 和 ActiveMQ Persistence。

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 ActiveMQ。

ActiveMQ Journal

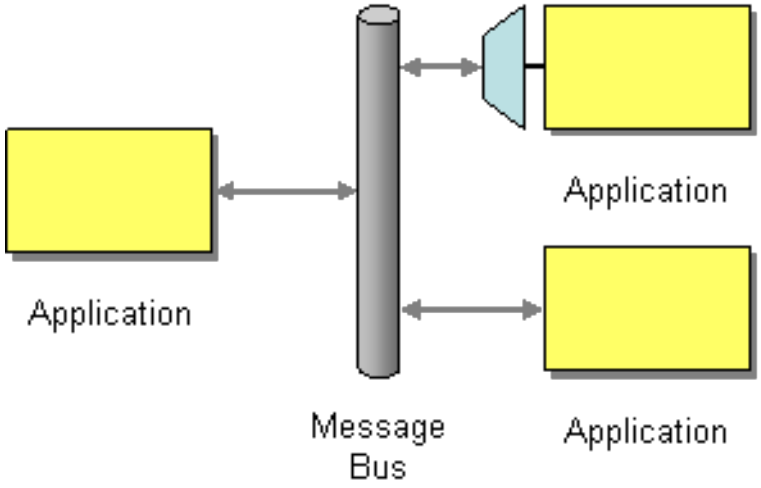
ActiveMQ Journal 组件针对特殊的用例进行了优化，其中多个并发制作者将消息写入队列，但只有一个活跃的使用者。消息存储在滚动日志文件中，并并发写入聚合在一起，以提高效率。

6.5. 消息总线

概述

消息总线 指的是 图 6.5 “消息总线模式” 中显示的消息传递架构，可让您连接在不同计算平台中运行的不同应用程序。实际上，Apache Camel 及其组件构成了消息总线。

图 6.5. 消息总线模式



消息总线模式的以下功能反映在 Apache Camel 中：

- *通用通信基础架构 10.10.10.2- the 路由器本身提供 Apache Camel 中常见通信基础架构的核心。但是，与一些消息总线架构不同，Apache Camel 提供了一个异构基础架构：消息可以使用各种不同传输方式发送到总线，并使用各种不同消息格式。*

- *Apache Camel 可以转换消息格式并使用不同传输来传播信息。实际上，Apache Camel 能够像适配器一样工作，以便外部应用程序可以 hook 到消息总线中，而不必重构它们的消息传递协议。*

  在某些情况下，也可以将适配器直接集成到外部应用程序中。例如，如果您使用 Apache CXF 开发应用，该服务使用 JAX-WS 和 JAXB 映射来实施，则可以将各种不同传输绑定到该服务。这些传输绑定功能作为适配器。

第 7 章 消息构建

摘要

消息构建模式描述了通过系统传递的消息的各种形式和功能。

7.1. 关联标识符

概述

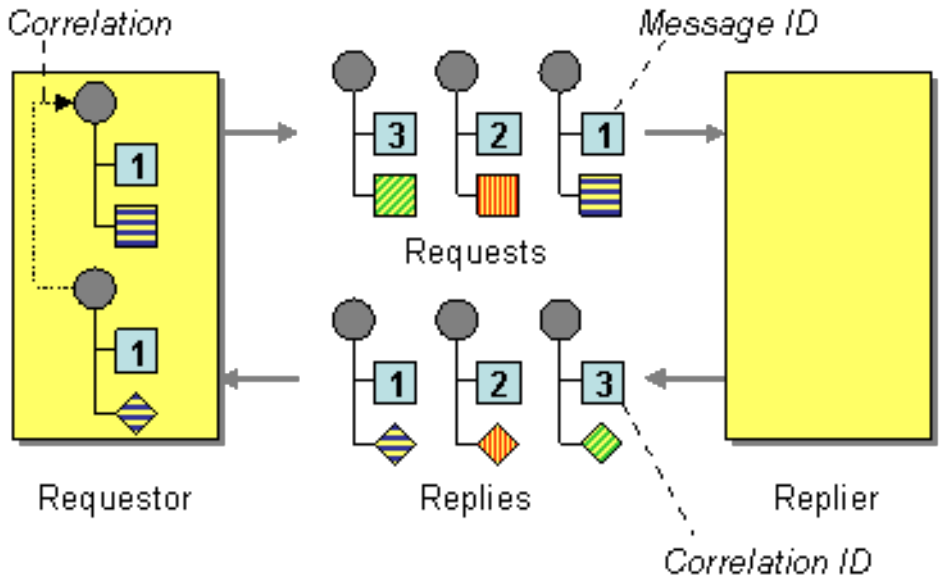
图 7.1 “关联标识符模式” 所示的 关联标识符 模式描述了如何匹配带有请求消息的回复信息，因为使用了异步消息系统来实施请求备份协议。这种理念的本质在于，请求消息应当通过唯一令牌生成，请求 ID，用于标识请求消息并回复消息应包含令牌、关联 ID（包含匹配请求 ID ）的关联 ID。

Apache Camel 通过获取或设置消息上的标头，支持来自 EIP 模式的识别标识符。

在使用 ActiveMQ 或 JMS 组件时，关联标识符标识符标头被称为 JMSCorrelationID。您可以在任何消息交换中添加您自己的关联标识符，以帮助将消息与单一对话（或业务流程）相关联。关联标识符通常存储在 Apache Camel 消息标头中。

有些 EIP 模式会加快子消息，在这种情况下，Apache Camel 会在 Exchanges 中添加一个关联 ID，作为其键为 Exchange.CORRELATION_ID 的属性，该模式将链接回源 Exchanges。例如，分割器、多播、接收者列表 和 线 tap EIP 执行此操作。

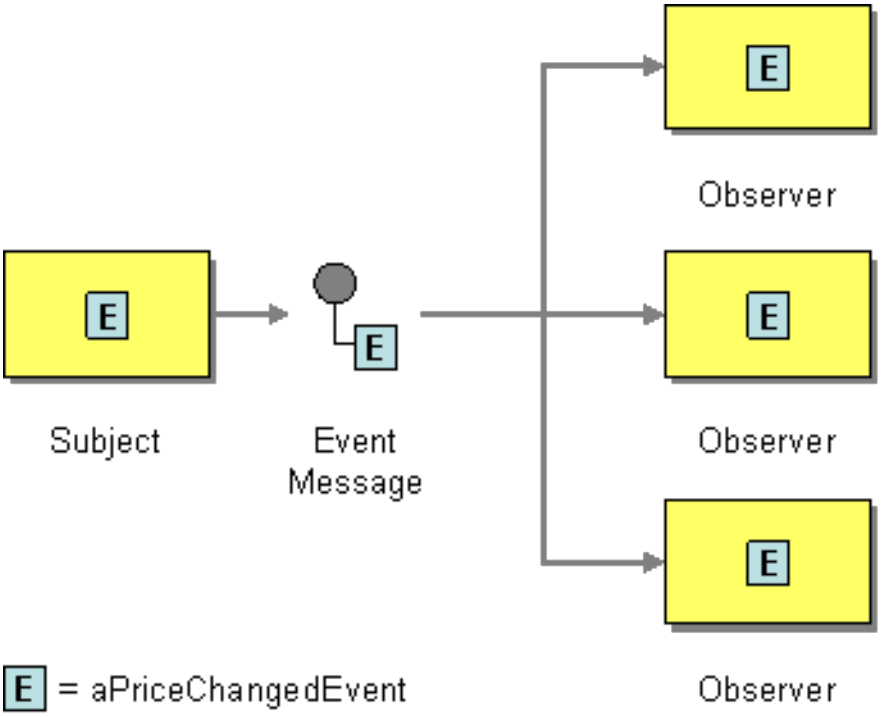
图 7.1. 关联标识符模式



7.2. 事件消息

事件消息

Camel 支持 企业集成模式 中的事件 消息，方法是支持 Exchange Pattern on a message，该消息可设为 InOnly 以指示单向事件信息。然后，Camel Apache Camel 组件参考 使用底层传输或协议实施此模式。



许多 Apache Camel 组件参考 的默认行为只是用于 JMS、file 或 SEDA 等。https://access.redhat.com/documentation/zh-cn/red_hat_fuse/7.11/html-single/apache_camel_component_reference/index#file-component

明确指定 InOnly

如果您使用默认为 InOut 的组件，您可以使用 pattern 属性覆盖端点 的消息交换 模式。

```
foo:bar?exchangePattern=InOnly
```

从 Camel 上的 2.0 开始，您可以使用 DSL 指定 消息交换模式。

使用 Fluent Builders

```
from("mq:someQueue").
  inOnly().
  bean(Foo.class);
```

或者，您可以使用显式模式调用端点

```
from("mq:someQueue").
  inOnly("mq:anotherQueue");
```

使用 Spring XML 扩展

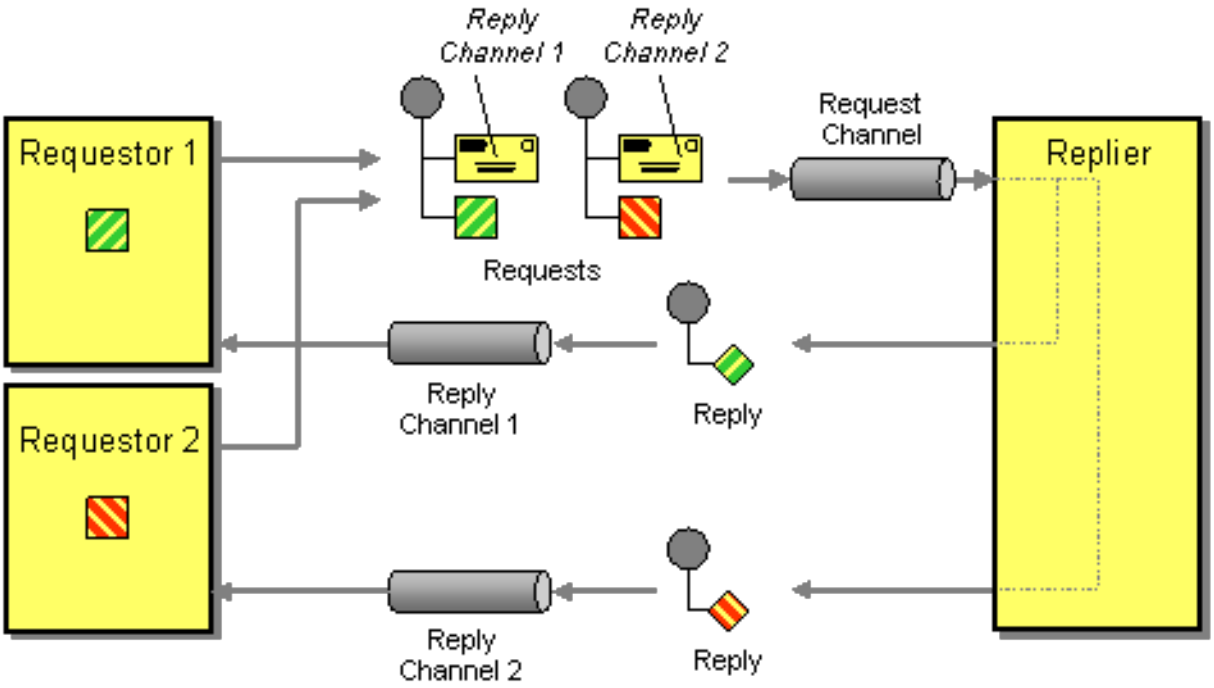
```
<route>
    <from uri="mq:someQueue"/>
    <inOnly uri="bean:foo"/>
</route>
```

```
<route>
    <from uri="mq:someQueue"/>
    <inOnly uri="mq:anotherQueue"/>
</route>
```

7.3. 返回地址

返回地址

Apache Camel 支持使用 JMSReplyTo 标头从 Enterprise Integration Patterns 中的返回地址。



例如，在将 JMS 与 InOut 搭配使用时，组件默认返回到 JMSReplyTo 中给出的地址。

示例

请求者代码

```
getMockEndpoint("mock:bar").expectedBodiesReceived("Bye World");
template.sendBodyAndHeader("direct:start", "World", "JMSReplyTo", "queue:bar");
```

使用 Fluent Builder 的路由

```
from("direct:start").to("activemq:queue:foo?preserveMessageQos=true");
from("activemq:queue:foo").transform(body().prepend("Bye "));
from("activemq:queue:bar?disableReplyTo=true").to("mock:bar");
```

使用 Spring XML 扩展的路由

```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:queue:foo?preserveMessageQos=true"/>
</route>

<route>
```

```
  <from uri="activemq:queue:foo"/>
  <transform>
     <simple>Bye ${in.body}</simple>
  </transform>
</route>

<route>
  <from uri="activemq:queue:bar?disableReplyTo=true"/>
  <to uri="mock:bar"/>
</route>
```

有关此模式的完整示例，请参阅此 JUnit 测试案例

第 8 章 消息路由

摘要

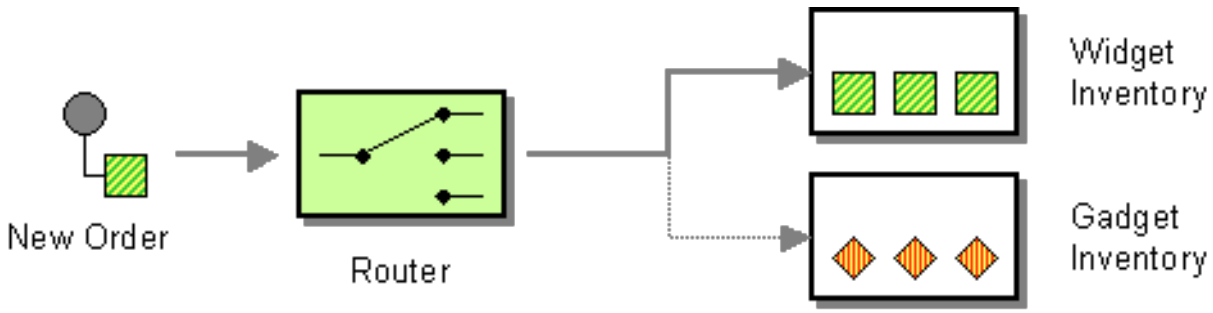
消息路由模式描述了将消息通道链接到一起的各种方法。这包括可应用于消息流的各种算法（无需修改消息的正文）。

8.1. 基于内容的路由器

概述

一个 基于内容的路由器 （在 图 8.1 “基于内容的路由器模式” 中）可让您根据消息内容将信息路由到适当的目的地。

图 8.1. 基于内容的路由器模式



Java DSL 示例

以下示例演示了如何根据各种 **predicate** 表达式评估从输入、**seda:a**、端点路由到 **seda:b**、**queue:c** 或 **seda:d** 的请求：

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").choice()
          .when(header("foo").isEqualTo("bar")).to("seda:b")
          .when(header("foo").isEqualTo("cheese")).to("seda:c")
          .otherwise().to("seda:d");
    }
};
```

XML 配置示例

以下示例演示了如何在 XML 中配置相同的路由：

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

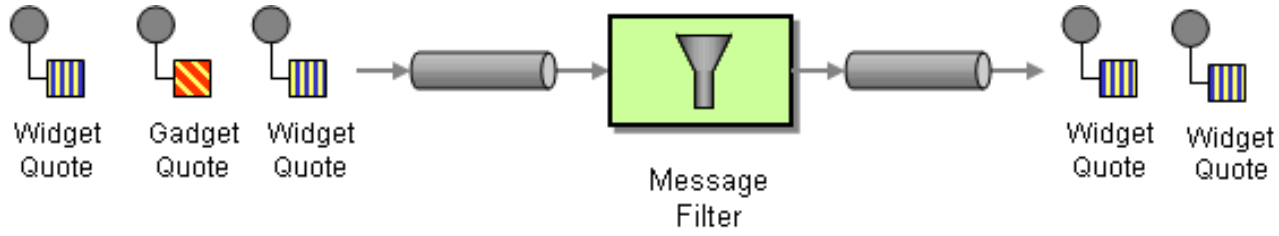
```
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

8.2. MESSAGE FILTER

概述

消息过滤器是一个处理器，用于根据特定标准消除不需要的消息。在 Apache Camel 中，消息过滤器特征（如 图 8.2 “消息过滤器模式”）通过 filter() Java DSL 命令实现。filter() 命令使用一个控制过滤器的 predicate 参数。当 predicate 为 true 时，允许传入的信息继续，如果 predicate 为 false，则会阻断传入的信息。

图 8.2. 消息过滤器模式



Java DSL 示例

以下示例演示了如何从端点(seda:a)创建路由到端点, seda:b, 它将阻断除 foo 标头具有值、bar 的消息之外的所有消息：

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").filter(header("foo").isEqualTo("bar")).to("seda:b");
    }
};
```

要评估更复杂的过滤器 predicates，您可以调用其中一个受支持的脚本语言，如 XPath、XQuery 或

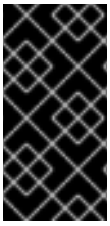
SQL（请参阅 第 II 部分 “路由表达式和指定语言”）。以下示例定义了一个路由，它阻止所有包含 name 属性等于 James 的个人元素外的所有消息：

```
from("direct:start").
        filter().xpath("/person[@name='James']").
        to("mock:result");
```

XML 配置示例

以下示例演示了如何使用 XML 中的 XPath predicate 配置路由（请参阅 第 II 部分 “路由表达式和指定语言”）：

```
<camelContext id="simpleFilterRoute" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$foo = 'bar'</xpath>
      <to uri="seda:b"/>
    </filter>
  </route>
  </camelContext>
```



过滤 </FILTER> 标签中所需的端点

在关闭 </filter> 标签前，确保您将要过滤的端点（例如，&lt ;to uri="seda:b"/>），或者过滤器不会被应用（在 2.8+ 中，省略此操作会导致错误）。

使用 Bean 过滤

以下是使用 bean 定义过滤器行为的示例：

```
from("direct:start")
    .filter().method(MyBean.class, "isGoldCustomer").to("mock:result").end()
    .to("mock:end");

public static class MyBean {
    public boolean isGoldCustomer(@Header("level") String level) {
        return level.equals("gold");
    }
}
```

使用 stop（）

可作为 Camel 2.0 提供

stop 是过滤 所有消息的 特殊过滤器。当您需要在其中一个 predicates 中停止进一步处理时，停止便可在 基于内容的路由器 中使用。

在以下示例中，我们不需要在邮件正文中传递单词的消息，以便在路由中进一步传播信息。我们防止在 when（） predicate 中使用 .stop（）。

```
from("direct:start")
    .choice()
        .when(bodyAs(String.class).contains("Hello")).to("mock:hello")
        .when(bodyAs(String.class).contains("Bye")).to("mock:bye").stop()
        .otherwise().to("mock:other")
    .end()
    .to("mock:result");
```

了解是否过滤 Exchange

可作为 Camel 2.5 提供。

消息过滤器 EIP 将添加 Exchange 的属性，该属性将说明（如果已过滤或未过滤）。

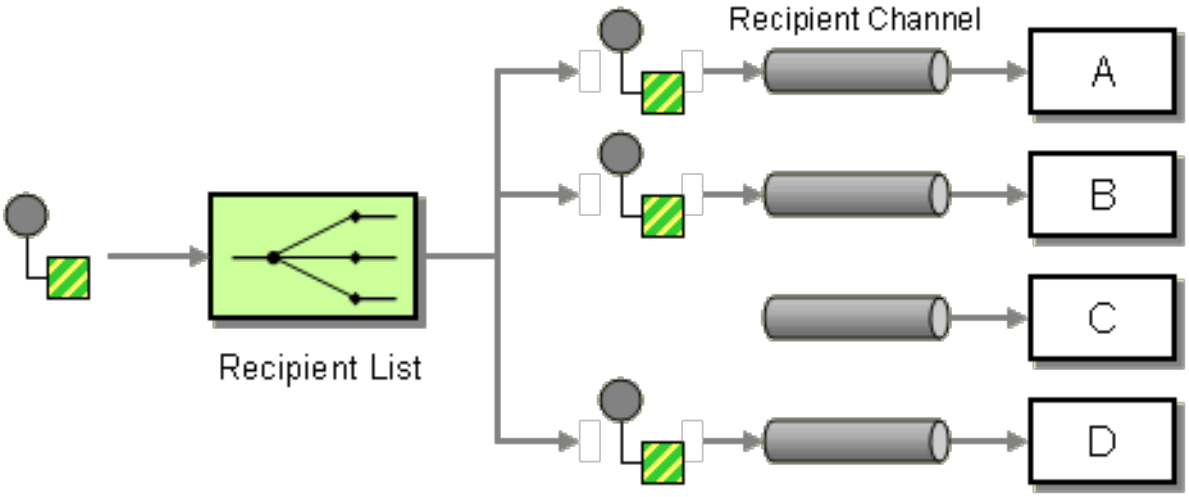
该属性具有关键 Exchange.FILTER_MATCHED，其字符串值为 CamelFilterMatched。其值是一个布尔值，代表 true 或 false。如果值为 true，则在 filter 块中路由 Exchange。

8.3. 接收者列表

概述

一个 接收者列表 （在 图 8.3 “接收者列表模式” 中显示的是路由器类型），将每个传入的信息发送到多个不同的目的地。此外，接收者列表通常要求在运行时计算接收者列表。

图 8.3. 接收者列表模式



带有固定目的地的接收者列表

最简单的接收者列表是预先固定并已知的目的地列表，而交换模式是 Only。在这种情况下，您可以硬编码到 to（） Java DSL 命令中的目的地列表。

注意

此处提供的示例，对于具有固定目的地的接收者列表，仅适用于 InOnly Exchange 模式（类似于 管道和过滤器模式）。如果要创建用于使用 Out 消息交换模式的接收者列表，请使用 多播 模式。

Java DSL 示例

以下示例演示了如何将来自消费者端点 queue:a 的 InOnly exchange 路由到固定目的地列表：

```
from("seda:a").to("seda:b", "seda:c", "seda:d");
```

XML 配置示例

以下示例演示了如何在 XML 中配置相同的路由：

```
<camelContext id="buildStaticRecipientList" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

运行时计算的接收者列表

在大多数情况下，当您使用接收者列表模式时，应在运行时计算接收者列表。为此，可使用 recipientList（） 处理器，它将目的地列表视为其唯一参数。由于 Apache Camel 将类型转换器应用到列表参数，因此应该使用大多数标准 Java 列表类型（如集合、列表或数组）。有关类型转换器的详情，请参考 第 34.3 节 “内置(In Type Converters)”。

接收者收到 同一 交换实例的副本，而 Apache Camel 会按顺序执行它们。

Java DSL 示例

以下示例演示了如何从名为 recipientListHeader 的消息标头提取目的地列表，其中标头值是以逗号分隔的端点 URI 列表：

```
from("direct:a").recipientList(header("recipientListHeader").tokenize(","));
```

在某些情况下，如果标头值是一个列表类型，您可以将其直接用作 接收者List（） 的参数。例如：

```
from("seda:a").recipientList(header("recipientListHeader"));
```

但是，这个示例完全依赖于底层组件如何解析这个特定标头。如果组件将标头解析为简单字符串，则该示例 将无法工作。标头必须解析到某种类型的 Java 列表。

XML 配置示例

以下示例演示了如何在 XML 中配置前面的路由，其中标头值是一个用逗号分开的端点 URI 列表：

```
<camelContext id="buildDynamicRecipientList" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <recipientList delimiter=",">
      <header>recipientListHeader</header>
    </recipientList>
  </route>
</camelContext>
```

并行发送到多个接收者

可作为 Camel 2.2 提供

接收者列表模式 支持 parallelProcessing，这与 splitter 模式中的相应功能类似。使用并行处理功能将交换发送到多个接收方，同时为 example.

```
from("direct:a").recipientList(header("myHeader")).parallelProcessing();
```

在 Spring XML 中，并行处理功能作为 接收者List tag>_<-abrtfor 示例上的属性实现：

```
<route>
  <from uri="direct:a"/>
  <recipientList parallelProcessing="true">
    <header>myHeader</header>
  </recipientList>
</route>
```

异常停止

可作为 Camel 2.2 提供

接收者列表 支持 stopOnException 功能，如果任何接收者失败，您可以使用它停止发送到任何进一步的接收者。

```
from("direct:a").recipientList(header("myHeader")).stopOnException();
```

以及在 Spring XML 中，其接收者列表标签上的属性。

在 Spring XML 中，异常功能的停止是作为 接收者List tag>_<-abrtfor 示例的属性实现：

```
<route>
  <from uri="direct:a"/>
  <recipientList stopOnException="true">
    <header>myHeader</header>
  </recipientList>
</route>
```

注意

您可以在同一 路由中组合并行处理 和停止OnException。

忽略无效的端点

从 Camel 2.3 开始可用

接收者列表模式 支持 ignoreInvalidEndpoints 选项，它允许接收者列表跳过无效的端点（路由 slips 模式 也支持此选项）。例如：

```
from("direct:a").recipientList(header("myHeader")).ignoreInvalidEndpoints();
```

在 Spring XML 中，您可以通过在 recipientList 标签上设置 ignoreInvalidEndpoints 属性来启用这个选项，如下所示

```
<route>
  <from uri="direct:a"/>
  <recipientList ignoreInvalidEndpoints="true">
    <header>myHeader</header>
  </recipientList>
</route>
```

请考虑 myHeader 包含两个端点的情况，direct:foo,xxx:bar。第一个端点有效且可以正常工作。第二个是无效，因此忽略。当遇到无效的端点时，Apache Camel 都会在 INFO 级别记录。

使用自定义 AggregationStrategy

可作为 Camel 2.2 提供

您可以使用带有 接收者列表模式 的自定义 AggregationStrategy，这对于从列表中接收方发出回复很有用。默认情况下，Apache Camel 使用 UseLatestAggregationStrategy 聚合策略，以仅保留上次收到的回复。对于更复杂的聚合策略，您可以自行定义 AggregationStrategy 接口的实施。有关详细信息，请参阅 第 8.5 节 “聚合器”。例如，若要将自定义聚合策略 MyOwnAggregationStrategy 应用到答复消息，您可以按照如下所示定义 Java DSL 路由：

```
from("direct:a")
    .recipientList(header("myHeader")).aggregationStrategy(new MyOwnAggregationStrategy())
    .to("direct:b");
```

在 Spring XML 中，您可以将自定义聚合策略指定为 接收者List 标签的属性，如下所示：

```
<route>
  <from uri="direct:a"/>
  <recipientList strategyRef="myStrategy">
    <header>myHeader</header>
  </recipientList>
  <to uri="direct:b"/>
</route>

<bean id="myStrategy" class="com.mycompany.MyOwnAggregationStrategy"/>
```

使用自定义线程池

可作为 Camel 2.2 提供

这只在使用 并行处理时才需要。默认情况下，Camel 使用有 10 个线程的线程池。请注意，在我们稍后进行线程管理和配置时（在 Camel 2.2 中可能发生）时，这可能会改变。

您像使用自定义聚合策略一样配置它。

使用方法调用作为接收者列表

您可以使用 bean 集成来提供接收者，例如：

```
from("activemq:queue:test").recipientList().method(MessageRouter.class, "routeTo");
```

其中，MessageRouter bean 的定义如下：

```
public class MessageRouter {

    public String routeTo() {
        String queueName = "activemq:queue:test2";
        return queueName;
    }
}
```

bean 作为接收者列表

您可以通过将 @RecipientList 注解添加到返回接收者列表的方法，使 bean 的行为作为接收者列表。

例如：

```
public class MessageRouter {

    @RecipientList
    public String routeTo() {
        String queueList = "activemq:queue:test1,activemq:queue:test2";
        return queueList;
    }
}
```

在这种情况下，不要在 路由中包含 **recipientList DSL** 命令。定义路由，如下所示：

```
from("activemq:queue:test").bean(MessageRouter.class, "routeTo");
```

使用超时

可作为 **Camel 2.5** 提供。

如果使用 并行处理，则可以以毫秒为单位配置总 超时值。然后，**Camel** 将并行处理消息，直到超时达到为止。这可让您在一条信息较慢时继续处理。

在以下示例中，**recipientlist** 标头的值为 **direct: a**、**direct:b**、**direct:c**，因此该邮件会发送到三个接收方。我们有 **250**毫秒的超时，这意味着在时间段内只有最后两条消息即可完成。因此，聚合会产生字符串结果 **BC**。

```
from("direct:start")
    .recipientList(header("recipients"), ",")
    .aggregationStrategy(new AggregationStrategy() {
            public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
                if (oldExchange == null) {
                    return newExchange;
                }

                String body = oldExchange.getIn().getBody(String.class);
                oldExchange.getIn().setBody(body + newExchange.getIn().getBody(String.class));
                return oldExchange;
            }
        })
        .parallelProcessing().timeout(250)
    // use end to indicate end of recipientList clause
    .end()
    .to("mock:result");

from("direct:a").delay(500).to("mock:A").setBody(constant("A"));
```

```
from("direct:b").to("mock:B").setBody(constant("B"));

from("direct:c").to("mock:C").setBody(constant("C"));
```

注意

这种 超时 功能也由 splitter 和 receiver List 支持。

默认情况下，如果超时发生 AggregationStrategy 不会被调用。但是，您可以实施专用版本

```
// Java
public interface TimeoutAwareAggregationStrategy extends AggregationStrategy {

    /**
     * A timeout occurred
     *
     * @param oldExchange  the oldest exchange (is <tt>null</tt> on first aggregation as we only have
the new exchange)
     * @param index       the index
     * @param total       the total
     * @param timeout     the timeout value in millis
     */
    void timeout(Exchange oldExchange, int index, int total, long timeout);
```

这允许您在 AggregationStrategy 中处理超时（如果您确实需要）。

TIMEOUT 是总计

超时为总计，这意味着在 X 时间后，Camel 将聚合在时间段内完成的消息。剩余部分将被取消。Camel 还将只调用 TimeoutAwareAggregationStrategy 中的 timeout 方法，用于第一个导致超时的索引。

将自定义处理应用到传出消息

在 recipientList 会向接收者端点发送一条信息前，它会创建一个消息副本，这是原始消息的绝对副本。在应该复制中，原始消息的标头和有效负载仅通过引用来复制。每个新副本不包含这些元素自己的实例。因此，消息的绝对副本会链接，当将其路由到不同的端点时，您无法应用自定义处理。

如果要在副本发送到端点前对每个消息副本执行一些自定义处理，您可以在 recipientList 子句中调用 onPrepare DSL 命令。onPrepare 命令只在消息被放送到其端点 前 插入自定义处理器。例如，在以下路由中，为每个接收者端点 在消息副本上调用 CustomProc 处理器：

```
from("direct:start")
  .recipientList().onPrepare(new CustomProc());
```

onPrepare DSL 命令的常见用例是对消息的部分或所有元素进行深入副本。这允许独立于其他消息副本修改每个消息副本。例如，以下 CustomProc 处理器类执行消息正文的深层副本，其中消息正文假定为 type、Bdy Type，而深度副本则由方法、Bdy Type.deepCopy（） 执行。

```
// Java
import org.apache.camel.*;
...
public class CustomProc implements Processor {

    public void process(Exchange exchange) throws Exception {
        BodyType body = exchange.getIn().getBody(BodyType.class);

        // Make a _deep_ copy of of the body object
        BodyType clone =  BodyType.deepCopy();
        exchange.getIn().setBody(clone);

        // Headers and attachments have already been
        // shallow-copied. If you need deep copies,
        // add some more code here.
    }
}
```

选项

recipientList DSL 命令支持以下选项：

名称	默认值	描述
delimiter	,	表达式返回的多个端点时使用分隔符。
strategyRef		指的是 AggregationStrategy，用于将接收方的回复组合成来自第 8.3 节 “接收者列表” 的单个传出消息。默认情况下，Camel 将使用最后的回复作为传出消息。
strategyMethodName		这个选项可用于明确指定要使用的方法名称，当 OVAs 用作 **AggregationStrategy** 时。

strategyMethodAllowNull	**false**	当将 POJOs 用作 **AggregationStrategy** 时，可以使用这个选项。如果为 **false**，则不会使用聚合方法，如果没有数据丰富。如果为 **true**，则使用空值作为 **oldExchange**，如果没有要增强数据，则使用**null** 值。
parallelProcessing	**false**	Camel 2.2：如果启用然后同时向接收者发送信息。请注意，调用者线程仍会等待所有消息都完全处理，然后再继续。它只发送和处理来自同时发生的收件人的回复。
parallelAggregate	**false**	如果启用，则 **AggregationStrategy** 上的聚合方法可以同时调用。**请注意，这需要实施 AggregationStrategy 为 thread-safe。默认情况下，此选项为 false，这表示 Camel 会自动同步对聚合方法的调用。然而，在一些用例中，您可以通过将 AggregationStrategy 作为 thread-safe，并将此选项设置为 true 来提高性能。**
executorServiceRef		Camel 2.2：请参阅自定义线程池，以用于并行处理。请注意，如果您设定了这个选项，则并行处理会被自动表示，您也不必启用该选项。
stopOnException	**false**	Camel 2.2：出现异常时是否立即停止持续处理。如果禁用，则 Camel 会将信息发送到所有收件人，无论它们是否失败。您可在完全控制如何处理它的 AggregationStrategy 类中处理异常。
ignoreInvalidEndpoints	**false**	Camel 2.3：如果无法解析端点 uri，它会被忽略。否则，Camel 会抛出一个异常，说明 endpoint uri 无效。

streaming	**false**	**Camel 2.5：** 如果启用，Camel 将处理顺序的回复，例如他们返回的顺序。如果禁用，Camel 将按照与指定的表达式相同的顺序进行回复。
timeout		**Camel 2.5：** 设置 millis 中指定的总超时。如果 第 8.3 节 “接收者列表” 没有可以在指定时间段内发送和处理所有回复，则超时触发器和 第 8.3 节 “接收者列表” 会中断并继续。请注意，如果您提供 AggregationStrategy，则会在中断前调用 **超时** 方法。
onPrepareRef		**Camel 2.8：** 请参阅自定义处理器准备每个接收方的 Exchange 副本。这可让您进行任何自定义逻辑，如 deep-cloning（如果需要）信息有效负载。
shareUnitOfWork	**false**	**Camel 2.8：** 是否应共享工作单元。详情请查看 第 8.4 节 “Splitter” 上的相同选项。
cacheSize	**0**	**Camel 2.13.1/2.12.4：** 允许配置 **ProducerCache** 的缓存大小，该缓存制作者在路由 slip 中重复使用。默认情况下，将使用默认的缓存大小，即 0。将值设为 -1 允许将缓存全部关闭。

在 Recipient 列表中使用 Exchange Pattern

默认情况下，Recipient List 使用当前的交换模式。但是，在有些情况下，您可以使用不同的交换模式向接收者发送信息。

例如，您可能有一个作为 InOnly 路由启动的路由。现在，如果您要通过接收者列表使用 InOut Exchange 模式，则需要直接在接收者端点中配置交换模式。

以下示例演示了新文件将作为 InOnly 开头，然后路由到接收者列表的路由。如果要将 InOut 与 ActiveMQ(JMS)端点搭配使用，则需要使用 exchangePattern 等于 InOut 选项指定该端点。但是，响应形式随后将继续路由 JMS 请求或回复，因此响应将作为文件存储在 outbox 目录中。

```
from("file:inbox")
  // the exchange pattern is InOnly initially when using a file route
  .recipientList().constant("activemq:queue:inbox?exchangePattern=InOut")
```

```
.to("file:outbox");
```

注意

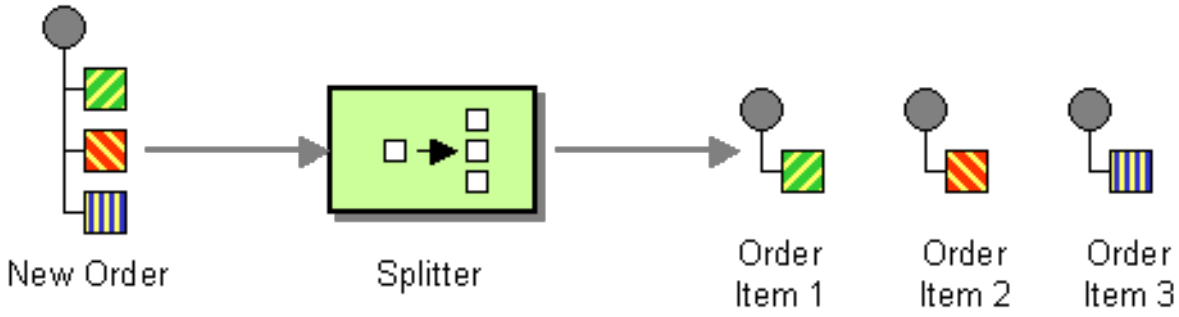
InOut Exchange 模式必须在超时时间内获得响应。但是，如果响应未接收，则会失败。

8.4. SPLITTER

概述

splitter 是将传入的消息拆分为一系列传出消息的路由器。每个传出消息均包含一条原始消息。在 Apache Camel 中，在 图 8.4 “Splitter Pattern” 中显示的 splitter 模式通过 split（） Java DSL 命令实现。

图 8.4. Splitter Pattern



Apache Camel splitter 实际上支持两种模式，如下所示：

- 简单的 splitter NETWORK-puppet 实施其本身的分割器模式。
- Splitter/aggregator fsanitize->_<com 组合了带聚合器模式的 splitter 模式，以便在处理后对消息的片段重新组合。

在分割器将原始消息分成几部分之前，它会制作原始消息的绝对副本。在应该复制中，原始消息的标头和有效负载仅复制为参考。虽然 splitter 本身不会将生成的消息部分路由到不同的端点，但拆分消息的部分可能会处于二级路由下。

由于消息部分是可浏览的副本，它们仍然与原始消息相关联。因此，无法独立修改它们。如果要在将自定义逻辑路由到端点集之前，将自定义逻辑应用到消息部分的不同副本，您必须在 splitter 子句中使用 onPrepareRef DSL 选项，以制作原始消息的深度副本。有关使用选项的详情请参考 “选项”一节。

Java DSL 示例

以下示例定义了一个来自 **seda:a** 到 **seda:b** 的路由，它将传入的消息一行转换为单独的传出消息：

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a")
          .split(bodyAs(String.class).tokenize("\n"))
          .to("seda:b");
    }
};
```

splitter 可以使用任何表达式语言，因此您可以使用任何受支持的脚本语言（如 **XPath**、**XQuery** 或 **SQL**）分割消息（请参阅 第 II 部分 “路由表达式和指定语言”）。以下示例从传入信息中提取 条 元素，并将它们插入到单独的传出消息中：

```
from("activemq:my.queue")
  .split(xpath("//foo/bar"))
  .to("file://some/directory")
```

XML 配置示例

以下示例演示了如何使用 **XPath** 脚本语言在 **XML** 中配置分割路由：

```
<camelContext id="buildSplitter" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="seda:a"/>
      <split>
        <xpath>//foo/bar</xpath>
        <to uri="seda:b"/>
      </split>
    </route>
</camelContext>
```

您可以使用 **XML DSL** 中的令牌化表达式来通过令牌分割样或标头，其中使用 **tokenize** 元素定义令牌化表达式。在以下示例中，消息正文使用 **\n** 分隔符字符进行令牌。要使用正则表达式模式，请在 **tokenize** 元素中设置 **regex=true**。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split>
            <tokenize token="\n"/>
            <to uri="mock:result"/>
```

```
        </split>
    </route>
    </camelContext>
```

分割成一组行

要将大型文件分成 1000 行块，您可以在 Java DSL 中按照如下所示定义分割路由：

```
from("file:inbox")
    .split().tokenize("\n", 1000).streaming()
        .to("activemq:queue:order");
```

用来 令牌的 第二个参数指定应分组到一个块的行数。streaming（） 子句会指示 splitter 不会一次读取整个文件（如果文件较大，则性能更高）。

同一路由可以在 XML DSL 中定义，如下所示：

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="\n" group="1000"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

使用 group 选项时的输出始终为 java.lang.String 类型。

跳过第一个项目

要跳过消息中的第一个项目，您可以使用 skipFirst 选项。

在 Java DSL 中，在 tokenize 参数中使用第三个选项 true ：

```
from("direct:start")
 // split by new line and group by 3, and skip the very first element
      .split().tokenize("\n", 3, true).streaming()
          .to("mock:group");
```

同一路由可以在 XML DSL 中定义，如下所示：

-

```
<route>
  <from uri="file:inbox"/>
    <split streaming="true">
    <tokenize token="\n" group="1000" skipFirst="true" />
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

Splitter reply

*如果输入 **splitter** 的交换具有 **InOut** 消息更改模式（即预期为回复），则分割者将原始输入消息的副本作为 **Out message** 插槽中的回复消息。您可以通过实施自己的 聚合策略来覆盖此默认行为。*

并行执行

*如果要并行执行生成的消息片段，您可以启用 **parallel processing** 选项，它会实例化线程池来处理消息片段。例如：*

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
from("activemq:my.queue").split(xPathBuilder).parallelProcessing().to("activemq:my.parts");
```

*您可以自定义并行分割器中使用的底层 **ThreadPoolExecutor**。例如，您可以在 **Java DSL** 中指定自定义 **executor**，如下所示：*

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(8, 16, 0L,
TimeUnit.MILLISECONDS, new LinkedBlockingQueue());
from("activemq:my.queue")
  .split(xPathBuilder)
  .parallelProcessing()
  .executorService(threadPoolExecutor)
  .to("activemq:my.parts");
```

*您可以在 **XML DSL** 中指定自定义执行器，如下所示：*

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:parallel-custom-pool"/>
    <split executorServiceRef="threadPoolExecutor">
      <xpath>/invoice/lineItems</xpath>
      <to uri="mock:result"/>
    </split>
  </route>
</camelContext>

<bean id="threadPoolExecutor" class="java.util.concurrent.ThreadPoolExecutor">
```

```
  <constructor-arg index="0" value="8"/>
  <constructor-arg index="1" value="16"/>
  <constructor-arg index="2" value="0"/>
  <constructor-arg index="3" value="MILLISECONDS"/>
  <constructor-arg index="4"><bean class="java.util.concurrent.LinkedBlockingQueue"/>
</constructor-arg>
</bean>
```

使用 bean 执行分割

因为 **splitter** 可以使用任何表达式进行拆分，您可以通过调用 **method（）** 表达式来使用 **bean** 执行拆分。**bean** 应返回可迭代的值，如 **java.util.Collection**、**java.util.Iterator** 或一个数组。

以下路由定义了一个 **method（）** 表达式，它调用 **mySplitterBean** 实例上的方法：

```
from("direct:body")
      // here we use a POJO bean mySplitterBean to do the split of the payload
      .split()
      .method("mySplitterBean", "splitBody")
      .to("mock:result");
from("direct:message")
      // here we use a POJO bean mySplitterBean to do the split of the message
      // with a certain header value
      .split()
      .method("mySplitterBean", "splitMessage")
      .to("mock:result");
```

其中 **mySplitterBean** 是 **MySplitterBean** 类的实例，其定义如下：

```
public class MySplitterBean {

   /**
    * The split body method returns something that is iteratable such as a java.util.List.
    *
    * @param body the payload of the incoming message
    * @return a list containing each part split
    */
   public List<String> splitBody(String body) {
      // since this is based on an unit test you can of couse
      // use different logic for splitting as {router} have out
      // of the box support for splitting a String based on comma
      // but this is for show and tell, since this is java code
      // you have the full power how you like to split your messages
      List<String> answer = new ArrayList<String>();
      String[] parts = body.split(",");
      for (String part : parts) {
         answer.add(part);
      }
      return answer;
```

```
    }

    /**
     * The split message method returns something that is iteratable such as a java.util.List.
     *
     * @param header the header of the incoming message with the name user
     * @param body the payload of the incoming message
     * @return a list containing each part split
     */
    public List<Message> splitMessage(@Header(value = "user") String header, @Body String body) {
        // we can leverage the Parameter Binding Annotations
        // http://camel.apache.org/parameter-binding-annotations.html
        // to access the message header and body at same time,
        // then create the message that we want, splitter will
        // take care rest of them.
        // *NOTE* this feature requires {router} version >= 1.6.1
        List<Message> answer = new ArrayList<Message>();
        String[] parts = header.split(",");
        for (String part : parts) {
            DefaultMessage message = new DefaultMessage();
            message.setHeader("user", part);
            message.setBody(body);
            answer.add(message);
        }
        return answer;
    }
}
```

您可以使用带有 Splitter EIP 的BeanIOSplitter 对象来分割大型有效负载，通过使用流模式以避免将整个内容读取在内存中。以下示例演示了如何使用从 classpath 加载的映射文件设置 BeanIOSplitter 对象：

注意

BeanIOSplitter 类是 Camel 2.18 中的新类。Camel 2.17 中不提供它。

```
BeanIOSplitter splitter = new BeanIOSplitter();
    splitter.setMapping("org/apache/camel/dataformat/beanio/mappings.xml");
    splitter.setStreamName("employeeFile");

    // Following is a route that uses the beanio data format to format CSV data
    // in Java objects:
    from("direct:unmarshal")
        // Here the message body is split to obtain a message for each row:
        .split(splitter).streaming()
        .to("log:line")
        .to("mock:beanio-unmarshal");
```

以下示例添加了一个错误处理器：

```
BeanIOSplitter splitter = new BeanIOSplitter();
  splitter.setMapping("org/apache/camel/dataformat/beanio/mappings.xml");
  splitter.setStreamName("employeeFile");
  splitter.setBeanReaderErrorHandlerType(MyErrorHandler.class);
  from("direct:unmarshal")
    .split(splitter).streaming()
    .to("log:line")
    .to("mock:beanio-unmarshal");
```

Exchange 属性

在每个分割交换上设置以下属性：

header	type	description
CamelSplitIndex	**int**	Apache Camel 2.0：为每个被分割交换的增加的分割计数器。计数器从 0 开始。
CamelSplitSize	**int**	Apache Camel 2.0：被分割的 Exchanges 总数。此标头不适用于基于流的拆分。
CamelSplitComplete	布尔值	Apache Camel 2.4：是否是最后的 Exchange。

Splitter/aggregator 模式

在处理各个组件完成后，消息组件的常见模式会聚合到单个交换中。为了支持这个模式， **split（） DSL** *命令可让您提供* **AggregationStrategy** *对象作为第二个参数。*

Java DSL 示例

以下示例演示了如何在处理完所有消息片段后使用自定义聚合策略重新组合分割消息：

```
from("direct:start")
   .split(body().tokenize("@"), new MyOrderStrategy())
       // each split message is then send to this bean where we can process it
       .to("bean:MyOrderService?method=handleOrder")
       // this is important to end the splitter route as we do not want to do more routing
       // on each split message
   .end()
   // after we have split and handled each message we want to send a single combined
```

```
    // response back to the original caller, so we let this bean build it for us
    // this bean will receive the result of the aggregate strategy: MyOrderStrategy
    .to("bean:MyOrderService?method=buildCombinedResponse")
```

AggregationStrategy 实施

*前面路由中使用的自定义聚合策略 **MyOrderStrategy** 被实施，如下所示：*

```
/**
 * This is our own order aggregation strategy where we can control
 * how each split message should be combined. As we do not want to
 * lose any message, we copy from the new to the old to preserve the
 * order lines as long we process them
 */
public static class MyOrderStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        // put order together in old exchange by adding the order from new exchange

        if (oldExchange == null) {
            // the first time we aggregate we only have the new exchange,
            // so we just return it
            return newExchange;
        }

        String orders = oldExchange.getIn().getBody(String.class);
        String newLine = newExchange.getIn().getBody(String.class);

        LOG.debug("Aggregate old orders: " + orders);
        LOG.debug("Aggregate new order: " + newLine);

        // put orders together separating by semi colon
        orders = orders + ";" + newLine;
        // put combined order back on old to preserve it
        oldExchange.getIn().setBody(orders);

        // return old as this is the one that has all the orders gathered until now
        return oldExchange;
    }
}
```

基于流的处理

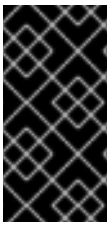
启用并行处理后，理论上可以在之前的某一部分之前为聚合做好准备。换句话说，消息片段可能会在聚合器中去除顺序。默认情况下，不会出现此情况，因为分离的实现会在将消息片段传递到聚合器之前将其重新整理到其原始顺序中。

*如果您希望在消息发布就绪后立即聚合消息片段（可能使用顺序），您可以启用 **streaming** 选项，如下所示：*

```
from("direct:streaming")
  .split(body().tokenize(","), new MyOrderStrategy())
    .parallelProcessing()
    .streaming()
    .to("activemq:my.parts")
  .end()
  .to("activemq:all.parts");
```

您还可以提供自定义迭代程序以用于流，如下所示：

```
// Java
import static org.apache.camel.builder.ExpressionBuilder.beanExpression;
...
from("direct:streaming")
    .split(beanExpression(new MyCustomIteratorFactory(), "iterator"))
    .streaming().to("activemq:my.parts")
```



STREAMING 和 XPATH

您不能与 XPath 一起使用 streaming 模式。XPath 需要内存中的完整 DOM XML 文档。

使用 XML 的流处理

如果传入的消息是一个非常大的 XML 文件，您可以在流传输模式中使用 tokenizeXML 子命令处理消息。

例如，如果一个包含一系列 顺序 元素的大型 XML 文件，您可以使用类似如下的路由将该文件分成 顺序 元素：

```
from("file:inbox")
  .split().tokenizeXML("order").streaming()
  .to("activemq:queue:order");
```

您可以在 XML 中进行同样的操作，方法是定义如下路由：

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="order" xml="true"/>
```

```
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

通常情况下，您需要访问在令牌元素的某一部分（原来）元素中定义的命名空间。您可以通过指定您要从哪个元素继承命名空间定义，将命名空间定义从其中一个级元素复制到 token 元素。

在 Java DSL 中，您要将 ancestor 元素指定为 tokenizeXML 的第二个参数。例如，从包含的 order 元素中继承 命名空间定义 ：

```
from("file:inbox")
  .split().tokenizeXML("order", "orders").streaming()
  .to("activemq:queue:order");
```

在 XML DSL 中，您可以使用 inheritNamespaceTagName 属性指定 ancestor 元素。例如：

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="order"
              xml="true"
              inheritNamespaceTagName="orders"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

选项

split DSL 命令支持以下选项：

名称	默认值	描述
strategyRef		指的是 AggregationStrategy，用于将子消息中的回复组合成来自第 8.4 节 “Splitter” 的单个传出消息。有关默认使用的内容，请参阅标题为**下面的 splitter 返回** 的内容。
strategyMethodName		这个选项可用于明确指定要使用的方法名称，当 OVAs 用作 **AggregationStrategy** 时。

strategyMethodAllowNull	**false**	当将 POJOs 用作 **AggregationStrategy** 时，可以使用这个选项。如果为 **false**，则不会使用聚合方法，如果没有数据丰富。如果为 **true**，则使用空值作为 **oldExchange**，如果没有要增强数据，则使用**null** 值。
parallelProcessing	**false**	如果启用，则同时处理子消息。请注意，调用者线程仍会等待所有子消息已完全处理，然后再继续处理。
parallelAggregate	**false**	如果启用，则 **AggregationStrategy** 上的聚合方法可以同时调用。**请注意，这需要实施 AggregationStrategy 为 thread-safe。默认情况下，此选项为 false，这表示 Camel 会自动同步对聚合方法的调用。然而，在一些用例中，您可以通过将 AggregationStrategy 作为 thread-safe，并将此选项设置为 true 来提高性能。**
executorServiceRef		指的是用于并行处理的自定义线程池。请注意，如果您设定了这个选项，则并行处理会被自动表示，您也不必启用该选项。
stopOnException	**false**	Camel 2.2： 出现异常时是否立即停止持续处理。如果禁用，则 Camel 会继续分割并处理子消息，无论它们之一是否失败。您可在完全控制如何处理它的 AggregationStrategy 类中处理异常。

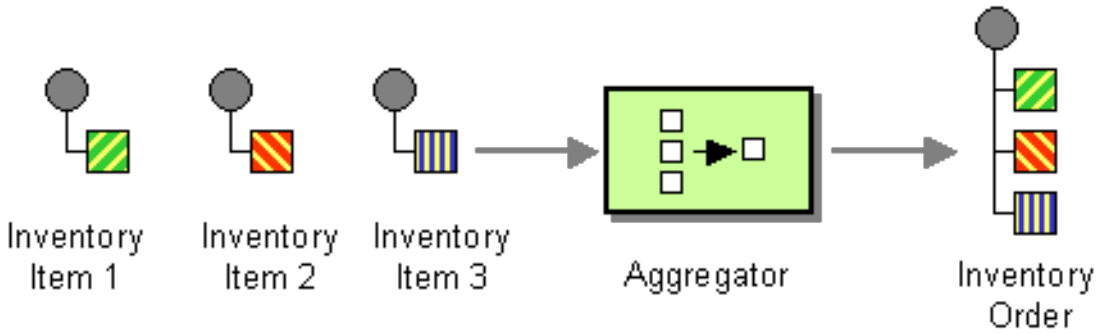
streaming	**false**	如果启用，Camel 将以流方式分割，这意味着它将以块的形式分割输入消息。这可减少内存开销。例如，如果您分离了建议启用流的大量消息。如果启用了流处理，则子消息回复将按其返回的顺序聚合。如果禁用，Camel 将按照子消息的回复处理与被分割的顺序相同。
timeout		Camel 2.5： 设置 millis 中指定的总超时。如果 第 8.3 节 "接收者列表" 无法分割并处理给定时间段内的所有回复，则超时触发器和 第 8.4 节 "Splitter" 会中断并继续。请注意，如果您提供 AggregationStrategy，则会在中断前调用 **超时** 方法。
onPrepareRef		Camel 2.8： 请参阅自定义处理器，在处理前准备 Exchange 的子消息。这可让您进行任何自定义逻辑，如 deep-cloning（如果需要）信息有效负载。
shareUnitOfWork	**false**	Camel 2.8： 是否应共享工作单元。详情请查看以下信息。

8.5. 聚合器

概述

借助在 图 8.5 "聚合器模式" 中显示的 聚合器 模式，您可以将相关消息批处理合并到单个消息中。

图 8.5. 聚合器模式



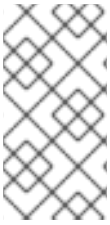
要控制聚合器的行为，Apache Camel 允许您指定 Enterprise Integration Patterns 中描述的属性，如下所示：

- *correlation 表达式 libselinux-eviction 确定应聚合哪些消息。在每条传入消息上评估关联表达式，以生成 关联键。然后，具有相同关联密钥的传入消息被分组到同一个批处理中。例如，如果要将 所有传入 的信息聚合到一个消息中，您可以使用一个恒定表达式。*

- *完整的信息完成时，完整性条件 10.10.10.2-eviction 确定了。您可以将此设置指定为一个简单的大小限制，或者更一般，您可以指定在批处理完成后标记的 predicate 条件。*

- *聚合算法 InventoryService-jaxb Com 组合了单一关联密钥的消息交换功能到单个消息交换中。*

例如，考虑一个可每秒接收 30,000 条消息的库存市场数据系统。如果您的 GUI 工具无法与如此大规模的更新率合作，则您可能希望减慢消息流。只需选择最新的报价并丢弃旧的价格，即可聚合传入的库存报价。（如果您愿意捕获历史信息，可以应用 delta 处理算法。）



注意

现在，聚合器现在使用包含更多信息的 ManagedAggregateProcessorMBean 形式列出 JMX。它允许您使用聚合控制器来控制它。

聚合器的工作方式

图 8.6 “聚合器实施” 展示了聚合器如何工作的概览，假设它通过带有关联键的交换流（如 A、B、C 或 D）进行交换流。

图 8.6. 聚合器实施

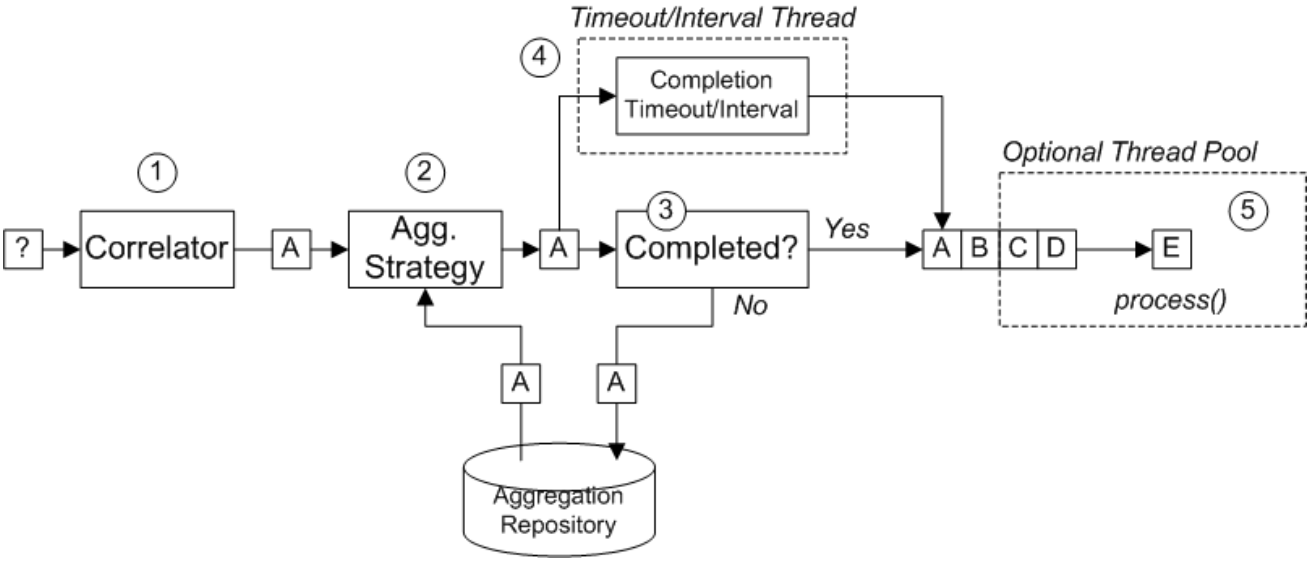


图 8.6 “聚合器实施” 中显示的传入的交换流如下：

1. 关联器负责根据关联密钥对交换进行排序。对于每个传入的交换，评估了关联表达式，从而生成关联密钥。例如，对于 图 8.6 “聚合器实施” 中显示的交换，关联键将评估为 A。

2. 聚合策略负责与相同关联密钥进行合并交换。当新的交换 A 处于 A 时，聚合器会在聚合存储库中查找对应的 聚合交换、A'，并将其与新交换合并。

   在完成特定的聚合周期前，传入的交换将继续与对应的聚合交换一起聚合。聚合周期持续到其中一个完成机制终止为止。

   注意

   从 Camel 2.16，新的 XSLT 聚合策略允许您将两个消息与 XSLT 文件合并。您可以从 toolbox 访问 AggregationStrategies.xslt（） 文件。

3. 如果在聚合器上指定了完成 predicate，则会测试聚合交换，以确定是否准备好发送到路由中的下一个处理器。处理继续，如下所示：

   - 如果完成，聚合交换由路由中的后方处理。这里有两种替代模型：同步 （默认），这会导致调用线程块或 异步 （如果启用并行处理），其中将聚合交换提交至 executor 线程池（如 图 8.6 “聚合器实施”）。
   - 如果没有完成，聚合交换将返回到聚合存储库。

4. 与同步完成测试并行，可以通过启用 completionTimeout 选项或 completionInterval 选项来启用异步完成测试。这些完成测试在单独的线程中运行，每当完成测试满意时，对应的交换都会标记为完成，并开始由路由后面的部分处理（同步或异步，具体取决于是否启用并行处理）。

5. 如果启用了并行处理，一个线程池负责在路由后面的部分处理交换。默认情况下，这个线程池包含十个线程，但您可以选择自定义池(“线程选项”一节)。

Java DSL 示例

以下示例使用 UseLatestAggregationStrategy 聚合策略来聚合具有相同 VDDK Symbol 标头值的交

换。对于给定的 prepare Symbol 值，如果收到了与该关联密钥的最后三秒钟以上，则聚合的交换被视为完成状态并发送到 模拟 端点。

```
from("direct:start")
    .aggregate(header("id"), new UseLatestAggregationStrategy())
        .completionTimeout(3000)
    .to("mock:aggregated");
```

XML DSL 示例

以下示例演示了如何在 XML 中配置相同的路由：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <aggregate strategyRef="aggregatorStrategy"
                completionTimeout="3000">
            <correlationExpression>
                <simple>header.StockSymbol</simple>
            </correlationExpression>
            <to uri="mock:aggregated"/>
        </aggregate>
    </route>
</camelContext>

<bean id="aggregatorStrategy"
      class="org.apache.camel.processor.aggregate.UseLatestAggregationStrategy"/>
```

指定关联表达式

在 Java DSL 中，关联表达式始终作为第一个参数传递给 aggregate（） DSL 命令。这里没有限制使用 Simple 表达式语言。您可以使用任何表达式语言或脚本语言（如 XPath、XQuery、SQL 等）指定关联表达式。

对于考试，若要使用 XPath 表达式关联交换，您可以使用以下 Java DSL 路由：

```
from("direct:start")
    .aggregate(xpath("/stockQuote/@symbol"), new UseLatestAggregationStrategy())
        .completionTimeout(3000)
    .to("mock:aggregated");
```

如果无法在特定的交换中评估关联表达式，聚合器默认会抛出 CamelExchangeException。您可以通过设置 ignoreCorrelationKeys 选项来限制这个异常。例如，在 Java DSL 中：

```
from(...).aggregate(...).ignoreInvalidCorrelationKeys()
```

在 XML DSL 中，您可以设置 ignoreInvalidCorrelationKeys 选项作为属性，如下所示：

```
<aggregate strategyRef="aggregatorStrategy"
        ignoreInvalidCorrelationKeys="true"
        ...>
    ...
</aggregate>
```

指定聚合策略

在 Java DSL 中，您可以将聚合策略作为第二参数传递给 aggregate（） DSL 命令，或使用 aggregationStrategy（） 子句指定。例如，您可以使用 aggregationStrategy（） 子句，如下所示：

```
from("direct:start")
    .aggregate(header("id"))
        .aggregationStrategy(new UseLatestAggregationStrategy())
        .completionTimeout(3000)
    .to("mock:aggregated");
```

Apache Camel 提供以下基本的聚合策略（即类属于 org.apache.camel.processor.aggregate Java 软件包）：

UseLatestAggregationStrategy

返回给定关联密钥的最后交换，用该密钥丢弃之前的所有交换。例如，此策略对于对来自股票交易所馈送的换算者来说非常有用，其中您只想了解特定库存符号的最新价格。

UseOriginalAggregationStrategy

返回给定关联密钥的第一个交换，用此密钥丢弃所有之后的交换。您必须在可以使用此策略前调用 UseOriginalAggregationStrategy.setOriginal（） 来设置第一个交换。

GroupedExchangeAggregationStrategy

将给定关联键 的所有 交换串联为列表，该列表存储在 Exchange.GROUPED_EXCHANGE 交换属性中。请参阅 “分组交换”一节。

实施自定义聚合策略

如果要应用不同的聚合策略，可以实施以下聚合策略基础接口之一：

org.apache.camel.processor.aggregate.AggregationStrategy

基本聚合策略接口。

org.apache.camel.processor.aggregate.TimeoutAwareAggregationStrategy

实施这个接口，如果您希望实施在聚合周期超时时收到通知。超时 通知方法有以下签名：

```
void timeout(Exchange oldExchange, int index, int total, long timeout)
```

org.apache.camel.processor.aggregate.CompletionAwareAggregationStrategy

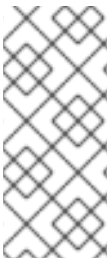
实施这个接口，如果您要在聚合周期正常完成时收到通知，则实施此接口。通知方法有以下签名：

```
void onCompletion(Exchange exchange)
```

例如，以下代码显示了两个不同的自定义聚合策略，StringAggregationStrategy 和 ArrayListAggregationStrategy::

```
//simply combines Exchange String body values using '' as a delimiter
class StringAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        if (oldExchange == null) {
            return newExchange;
        }

        String oldBody = oldExchange.getIn().getBody(String.class);
        String newBody = newExchange.getIn().getBody(String.class);
        oldExchange.getIn().setBody(oldBody + "" + newBody);
        return oldExchange;
    }
}

//simply combines Exchange body values into an ArrayList<Object>
class ArrayListAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
     Object newBody = newExchange.getIn().getBody();
     ArrayList<Object> list = null;
        if (oldExchange == null) {
      list = new ArrayList<Object>();
      list.add(newBody);
      newExchange.getIn().setBody(list);
      return newExchange;
        } else {
         list = oldExchange.getIn().getBody(ArrayList.class);
      list.add(newBody);
```

```
        return oldExchange;
      }
     }
  }
```



注意

从 Apache Camel 2.0 开始，在第一个交换中也调用 AggregationStrategy.aggregate（） 回调方法。在 aggregate 方法的第一个调用中，oldExchange 参数为 null，newExchange 参数包含第一个传入的交换。

要使用自定义策略类 ArrayListAggregationStrategy 来聚合消息，请按如下所示定义一个路由：

```
from("direct:start")
    .aggregate(header("StockSymbol"), new ArrayListAggregationStrategy())
    .completionTimeout(3000)
    .to("mock:result");
```

您还可以使用 XML 中的自定义聚合策略配置路由，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregate strategyRef="aggregatorStrategy"
            completionTimeout="3000">
      <correlationExpression>
        <simple>header.StockSymbol</simple>
      </correlationExpression>
      <to uri="mock:aggregated"/>
    </aggregate>
  </route>
</camelContext>

<bean id="aggregatorStrategy" class="com.my_package_name.ArrayListAggregationStrategy"/>
```

控制自定义聚合策略的生命周期

您可以实施自定义聚合策略，以便其生命周期与控制它的企业集成模式的生命周期一致。这可用于确保聚合策略可以安全地关闭。

要实现具有生命周期支持的聚合策略，您必须实现 org.apache.camel.Service 接口（除 AggregationStrategy 接口外），并提供 start（） 和 stop（） 生命周期方法的实现。例如，以下代码示例演示了具有生命周期支持的聚合策略：

-

```
// Java
import org.apache.camel.processor.aggregate.AggregationStrategy;
import org.apache.camel.Service;
import java.lang.Exception;
...
class MyAggStrategyWithLifecycleControl
      implements AggregationStrategy, Service {

   public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
      // Implementation not shown...
      ...
   }

   public void start() throws Exception {
      // Actions to perform when the enclosing EIP starts up
      ...
   }

   public void stop() throws Exception {
      // Actions to perform when the enclosing EIP is stopping
      ...
   }
}
```

Exchange 属性

每个聚合交换中会设置以下属性：

标头	类型	描述聚合交换属性
Exchange.AGGREGATED_SIZE	**int**	整合到此交换的交换总数。
Exchange.AGGREGATED_COMPLETED_BY	**字符串**	表示负责完成聚合交换的机制。可能的值有：**predicate**、**size**、**timeout**、**interval** 或 **consumer**。

以下属性在由 SQL 组件聚合存储库进行交换上设置（请参阅 "持久性聚合存储库"一节）：

标头	类型	描述 Red Hatlivered Exchange Properties
Exchange.REDELIVERY_COUNTER	**int**	当前重新传送尝试的序列号（从 **1**开始）。

指定完成条件

至少需要指定一个 完成条件，它决定聚合交换离开聚合器并继续路由上的下一个节点。可以指定以下完成条件：

completionPredicate

聚合每个交换后评估 predicate，以确定完整性。true 表示聚合交换已完成。另外，您可以定义一个自定义 AggregationStrategy 来实施 Predicate 接口，在这种情况下，AggregationStrategy 将用作 completion predicate。

completionSize

聚合指定数量的传入的交换后，完成聚合的交换。

completionTimeout

（与 completionInterval兼容） 在指定的超时时间内没有聚合交换时完成聚合交换。

换句话说，超时机制会跟踪 每个 关联键值的超时。在收到带有特定密钥值的最新交换后，时钟开始选择。如果指定超时内 未收到 具有相同键值的另一个交换，则对应的聚合交换标记为完成并发送到路由上的下一个节点。

completionInterval

（与 completionTimeout兼容 ） 将 完成所有 未完成的聚合交换，经过每个时间间隔（指定长度）过后。

间隔 不会 为每个聚合交换量身定制。这种机制会强制同步完成所有未完成的聚合交换。因此，在某些情况下，这种机制可在聚合后立即完成聚合交换。

completionFromBatchConsumer

与支持 批处理消费者 机制的消费者端点结合使用时，此完成选项会根据从消费者端点接收的信息，在当前批量交换完成后自动找出出。请参阅 “批量消费者”一节。

forceCompletionOnStop

启用此选项后，它会强制在当前路由上下文停止后完成所有未完成的聚合交换。

除了 completionTimeout 和 completionInterval 条件（无法同时启用）外，前面的完成条件可以任意组合使用。当条件组合使用时，触发的第一个完成条件是有效的完成条件。

指定完成 predicate

您可以指定一个任意 **predicate** 表达式，决定聚合交换完成后决定。评估 **predicate** 表达式的方法有两种：

- 在最新的聚合交换 中，**the default behavior**是默认行为。
- 当您启用 **eagerCheckCompletion** 选项时，会选择最新传入的 **Exchange** 您要将这个行为被选择。

例如，如果您想要在每次收到 **ALERT** 消息时终止库存引号流（如最新传入的交换中 **MsgType** 标头的值所示），您可以定义一个类似如下的路由：

```
from("direct:start")
   .aggregate(
     header("id"),
     new UseLatestAggregationStrategy()
   )
       .completionPredicate(
         header("MsgType").isEqualTo("ALERT")
        )
       .eagerCheckCompletion()
   .to("mock:result");
```

以下示例演示了如何使用 **XML** 配置同一路由：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregate strategyRef="aggregatorStrategy"
               eagerCheckCompletion="true">
      <correlationExpression>
         <simple>header.StockSymbol</simple>
      </correlationExpression>
      <completionPredicate>
         <simple>$MsgType = 'ALERT'</simple>
      </completionPredicate>
      <to uri="mock:result"/>
    </aggregate>
  </route>
</camelContext>

<bean id="aggregatorStrategy"
      class="org.apache.camel.processor.aggregate.UseLatestAggregationStrategy"/>
```

指定动态完成超时

可以指定 动态完成超时，其中为每个传入的交换重新计算超时值。例如，若要从每个传入交换中的 超时标头设置超时值，您可以按照如下所示定义路由：

```
from("direct:start")
    .aggregate(header("StockSymbol"), new UseLatestAggregationStrategy())
        .completionTimeout(header("timeout"))
    .to("mock:aggregated");
```

您可以在 XML DSL 中配置相同的路由，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <aggregate strategyRef="aggregatorStrategy">
            <correlationExpression>
                <simple>header.StockSymbol</simple>
            </correlationExpression>
            <completionTimeout>
                <header>timeout</header>
            </completionTimeout>
            <to uri="mock:aggregated"/>
        </aggregate>
    </route>
</camelContext>

<bean id="aggregatorStrategy"
      class="org.apache.camel.processor.UseLatestAggregationStrategy"/>
```

注意

您还可以添加固定超时值，如果动态值是 null 或 0， 则 Apache Camel 会返回使用这个值。

指定动态完成大小

可以指定 动态完成大小，在其中为每个传入的交换重新计算完成大小。例如，要在每个传入交换中从 mySize 标头设置完成大小，您可以按照如下所示定义路由：

```
from("direct:start")
    .aggregate(header("StockSymbol"), new UseLatestAggregationStrategy())
        .completionSize(header("mySize"))
    .to("mock:aggregated");
```

以及使用 Spring XML 相同的示例：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregate strategyRef="aggregatorStrategy">
      <correlationExpression>
        <simple>header.StockSymbol</simple>
      </correlationExpression>
      <completionSize>
        <header>mySize</header>
      </completionSize>
      <to uri="mock:aggregated"/>
    </aggregate>
  </route>
</camelContext>

<bean id="aggregatorStrategy"
    class="org.apache.camel.processor.UseLatestAggregationStrategy"/>
```

注意

您还可以添加固定大小值，如果动态值是 null 或 0，则 Apache Camel 会返回使用这个值。

从 AggregationStrategy 内强制完成单个组

如果实施自定义 AggregationStrategy 类，可以通过一种机制来强制完成当前消息组，方法是将 Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP Exchange 属性设为 true。这种机制仅影响当前组：其他消息组（具有不同关联 ID）不会被强制完成。这种机制会覆盖任何其他完成机制，如 predicate、大小、超时等。

例如，如果消息正文大小大于 5，以下示例 AggregationStrategy 类将完成当前的组：

```
// Java
public final class MyCompletionStrategy implements AggregationStrategy {
  @Override
  public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    if (oldExchange == null) {
      return newExchange;
    }
    String body = oldExchange.getIn().getBody(String.class) + "+"
      + newExchange.getIn().getBody(String.class);
    oldExchange.getIn().setBody(body);
    if (body.length() >= 5) {
      oldExchange.setProperty(Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP,
true);
```

```
        }
        return oldExchange;
    }
}
```

强制使用特殊消息完成所有组

通过向路由发送带有特殊标头的消息，可以强制完成所有未完成的聚合消息。您可以使用两种替代标头设置来强制完成：

Exchange.AGGREGATION_COMPLETE_ALL_GROUPS

设置为 **true**，以强制完成当前的聚合周期。此消息仅以信号形式运作，不包含在 任何聚合周期内。处理此信号消息后，会丢弃消息的内容。

Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE

设置为 **true**，以强制完成当前的聚合周期。此消息 包含在 当前聚合周期中。

使用 AggregateController

org.apache.camel.processor.aggregate.AggregateController 可让您使用 **Java** 或 **JMX API** 控制运行时的聚合。这可用于强制完成交换组，或者查询当前运行时统计信息。

如果没有配置自定义，聚合器提供了一个默认的实现，您可以使用 **getAggregateController（）** 方法访问它。但是，使用 **aggregateController** 轻松地在路由中配置控制器。

```
private AggregateController controller = new DefaultAggregateController();

from("direct:start")
  .aggregate(header("id"), new MyAggregationStrategy()).completionSize(10).id("myAggregator")
    .aggregateController(controller)
    .to("mock:aggregated");
```

另外，您可以使用 **AggregateController**上的 **API** 来强制完成。例如，使用键 **foo** 完成组

```
int groups = controller.forceCompletionOfGroup("foo");
```

数值返回将是已完成的组数。以下是完成所有组的 **API**：

```
 int groups = controller.forceCompletionOfAllGroups();
```

强制唯一关联密钥

在一些聚合场景中，您可能希望强制关联密钥对每批交换是唯一的条件。换而言之，当特定关联键的聚合交换完成时，您希望确保不进一步的聚合交换，不能与这个关联键进行进一步的交换。例如，如果路由中的后者部分与唯一关联键值进行交换，您可能希望强制执行此条件。

根据配置完成条件的方式，可能会遇到使用特定关联密钥生成多个聚合交换的风险。例如，虽然您可以定义一个完成 predicate，以便等到接收具有特定关联密钥 的所有 交换前，您也可以定义完成超时，这可以在与该密钥到达的所有交换之前触发。在这种情况下，较晚的交换可能会给 第二个 聚合交换带来的增加，具有相同关联键的值。

对于这样的场景，您可以通过设置 closeCorrelationKeyOnCompletion 选项，将聚合器配置为绕过重复之前关联键值的聚合交换。为绕过重复的关联键值，聚合器需要在缓存中记录之前的关联键值。此缓存的大小（缓存的关联键的数量）被指定为 closeCorrelationKeyOnCompletion（） DSL 命令的参数。要指定无限大小的缓存，您可以传递一个零个或一个负整数。例如，指定 10000 个键值的缓存大小：

```
from("direct:start")
    .aggregate(header("UniqueBatchID"), new MyConcatenateStrategy())
        .completionSize(header("mySize"))
        .closeCorrelationKeyOnCompletion(10000)
    .to("mock:aggregated");
```

如果聚合交换以重复的关联键值完成，聚合器会抛出 ClosedCorrelationKeyException 异常。

使用简单表达式的基于流的处理

您可以将 Simple 语言表达式用作使用流模式的 tokenizeXML 子命令的令牌。使用简单语言表达式将支持动态令牌。

例如，要使用 Java 将一系列名称分割成标签用户角色，您可以使用 令牌izeXML bean 和简单语言令牌将 文件分成 名称 元素。

```
public void testTokenizeXMLPairSimple() throws Exception {
        Expression exp = TokenizeLanguage.tokenizeXML("${header.foo}", null);
```

获取由 < person > 划分的名称输入字符串，并将 &lt ;person& gt; 设置为令牌。

```
        exchange.getIn().setHeader("foo", "<person>");
        exchange.getIn().setBody("<persons><person>James</person><person>Claus</person>
<person>Jonathan</person><person>Hadrian</person></persons>");
```

-

列出从输入中分割的名称。

```
        List<?> names = exp.evaluate(exchange, List.class);
        assertEquals(4, names.size());

        assertEquals("<person>James</person>", names.get(0));
        assertEquals("<person>Claus</person>", names.get(1));
        assertEquals("<person>Jonathan</person>", names.get(2));
        assertEquals("<person>Hadrian</person>", names.get(3));
    }
```

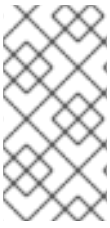
分组交换

您可以将传出批处理中的所有聚合交换组合为一个 org.apache.camel.impl.GroupedExchange holder 类。要启用分组的交换，请指定 groupExchanges（） 选项，如以下 Java DSL 路由中所示：

```
from("direct:start")
    .aggregate(header("StockSymbol"))
        .completionTimeout(3000)
        .groupExchanges()
    .to("mock:result");
```

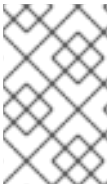
发送至 模拟：result 的分组交换列表包含消息正文中的聚合交换列表。以下行显示后续的处理器如何以列表的形式访问分组交换的内容：

```
// Java
List<Exchange> grouped = ex.getIn().getBody(List.class);
```



注意

当您启用分组的交换功能时，不得 配置聚合策略（分组交换功能本身就是一个聚合策略）。



注意

从传出交换的属性访问分组交换的旧方法是已弃用，并将在以后的发行版本中删除。

批量消费者

聚合器可以和 批处理消费者 模式一起工作，以汇总批处理消费者报告的消息总数（批处理端点设置 CamelBatchSize、CamelBatchIndex 和 CamelBatchComplete 属性）。例如，若要聚合由文件消费者

端点找到的所有文件，您可以按照以下方式使用路由：

```
from("file://inbox")
    .aggregate(xpath("//order/@customerId"), new AggregateCustomerOrderStrategy())
    .completionFromBatchConsumer()
    .to("bean:processOrder");
```

目前，以下端点支持批处理消费者机制：文件、FTP、邮件、iBatis 和 JPA。

持久性聚合存储库

默认聚合器只使用内存的 AggregationRepository。如果要永久存储待处理聚合交换，您可以使用 SQL 组件 作为持久聚合存储库。SQL 组件包含一个 JdbcAggregationRepository，可持续保留聚合的消息，并确保您不会丢失任何消息。

当成功处理交换后，当存储库上调用 确认方法时，它将标记为 complete。这意味着，如果同样的交换再次失败，它将重试，直到成功为止。

添加对 camel-sql 的依赖

要使用 SQL 组件，您必须在项目中包含对 camel-sql 的依赖。例如，如果您使用 Maven pom.xml 文件：

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sql</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

创建聚合数据库表

您必须创建一个单独的聚合和已完成的数据库表以实现持久性。例如，以下查询会为名为 my_aggregation_repo 的数据库创建表：

```
CREATE TABLE my_aggregation_repo (
 id varchar(255) NOT NULL,
 exchange blob NOT NULL,
 constraint aggregation_pk PRIMARY KEY (id)
);

CREATE TABLE my_aggregation_repo_completed (
 id varchar(255) NOT NULL,
 exchange blob NOT NULL,
```

```
 constraint aggregation_completed_pk PRIMARY KEY (id)
);
}
```

配置聚合存储库

您还必须在框架 XML 文件中配置聚合存储库（如 Spring 或 Blueprint）：

```
<bean id="my_repo"
  class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
  <property name="repositoryName" value="my_aggregation_repo"/>
  <property name="transactionManager" ref="my_tx_manager"/>
  <property name="dataSource" ref="my_data_source"/>
  ...
</bean>
```

repositories Name、transactionManager 和 dataSource 属性是必需的。有关持久性聚合存储库的更多信息，请参阅 Apache Camel 组件参考指南 中的 SQL 组件。

线程选项

如 图 8.6 “聚合器实施” 所示，聚合器与路由中的后方分离，后者将发送到路由后面的部分，后者由专用线程池处理。默认情况下，这个池仅包含单个线程。如果要指定多个线程的池，启用 parallelProcessing 选项，如下所示：

```
from("direct:start")
  .aggregate(header("id"), new UseLatestAggregationStrategy())
      .completionTimeout(3000)
      .parallelProcessing()
  .to("mock:aggregated");
```

默认情况下，这会创建一个有 10 个 worker 线程的池。

如果要对创建的线程池进行更多控制，请使用 executorService 选项指定自定义 java.util.concurrent.ExecutorService 实例（在这种情况下，不需要启用 parallelProcessing 选项）。

聚合到列表中

常见的聚合场景涉及将一系列传入的消息正文聚合到一个 List 对象中。为便于这种情况，Apache Camel 提供了 AbstractListAggregationStrategy 抽象类，您可以快速扩展来为本例创建聚合策略。传入类型为 T 的消息正文，聚合到一个完成的交换中，以及类型为 List<T> 的消息正文。

例如，若要将一系列 Integer 消息正文聚合到一个 List<Integer > 对象中，您可以使用以下定义的聚合策略：

```
import org.apache.camel.processor.aggregate.AbstractListAggregationStrategy;
...
/**
 * Strategy to aggregate integers into a List<Integer>.
 */
public final class MyListOfNumbersStrategy extends AbstractListAggregationStrategy<Integer> {

    @Override
    public Integer getValue(Exchange exchange) {
        // the message body contains a number, so just return that as-is
        return exchange.getIn().getBody(Integer.class);
    }
}
```

aggregator 选项

聚合器支持以下选项：

表 8.1. 聚合器选项

选项	默认值	描述
correlationExpression		强制表达式，用于评估用于聚合的关联密钥。具有相同关联密钥的 Exchange 会聚合在一起。如果无法评估关联密钥，则引发 Exception。您可以使用 **ignoreBadCorrelationKeys** 选项禁用此功能。

选项	默认值	描述
aggregationStrategy		mandatory **AggregationStrategy**，用于将传入的 Exchange 与现有的合并交换合并。**首先调用 oldExchange 参数是 null。在随后的调用中，oldExchange 包含合并的交换，newExchange 则属于新传入的 Exchange。从 Camel 2.9.2 开始，该策略可以选择是一个 TimeoutAwareAggregationStrategy 实现，它支持超时回调。从 Camel 2.16 开始，该策略也可以是 PreCompletionAwareAggregationStrategy 实施。它可在预补全模式下运行完成检查。**
strategyRef		在 registry 中查询 **AggregationStrategy** 的引用。
completionSize		聚合完成前聚合的消息数量。这个选项可以被设置为固定值或使用表达式（允许您动态评估大小）将使用 **Integer**。如果两者均被设置为 **null** 或 **0，** 则 Camel 将回退设置为使用固定的值。
completionTimeout		聚合交换的时间在完成前应不活跃。这个选项可以设置为固定值或使用允许您动态评估超时的表达式 - 将因此使用 长长。如果两者均被设置为 **null** 或 **0，** 则 Camel 将回退设置为使用固定的值。您不能将此选项与 completionInterval 一起使用，只能同时使用这两者之一。
completionInterval		在聚合器中重复一个期间，聚合器将完成所有当前的聚合交换。Camel 有一个后台任务，每个任务都会触发。您不能将此选项与 completionTimeout 一起使用，只能同时使用其中之一。

选项	默认值	描述
completionPredicate		指定 predicate（**org.apache.camel.Predicate** 类型），它会在聚合交换完成后发出信号。另外，您可以定义一个自定义 **AggregationStrategy** 来实施 **Predicate** 接口，在这种情况下，**AggregationStrategy** 将用作 completion predicate。
completionFromBatchConsumer	**false**	这个选项是，如果交换来自一个 Batch Consumer。然后，当启用 第 8.5 节 “聚合器” 时，将使用在消息标头 **CamelBatchSize** 中由 Batch Consumer 决定的批处理大小。如需更多相关信息，请参阅 Batch Consumer。这可用于汇总给定轮询中来自 *查看文件* 端点的所有文件。
eagerCheckCompletion	**false**	在收到新传入的 Exchange 时，是否检查是否完成。这个选项会影响 **completionPredicate** 选项的行为，因为 Exchange 会相应地传递更改。为**防止** 在 Predicate 传递的 Exchange 是 **聚合的** Exchange 时，这意味着您可以在 **AggregationStrategy** 聚合的 Exchange 中存储任何信息。为使该前文中传递的 Exchange 是 **传入** Exchange，这意味着您可以从进入的 Exchange 访问数据。
forceCompletionOnStop	**false**	如果为 **true**，则在停止当前路由上下文时完成所有聚合的交换。
groupExchanges	**false**	如果启用，Camel 将所有聚合的 Exchanges 分组到一个整合的 **org.apache.camel.impl.GroupedExchange** holder 类中，该类包含所有聚合的 Exchanges。因此，只有一个 Exchange 从聚合器中发出。可用于将许多进入的 Exchanges 组合为一个输出 Exchange，而无需自己编写自定义 **AggregationStrategy**。

选项	默认值	描述
ignoreInvalidCorrelationKeys	**false**	是否忽略无法被评估到值的关联键。默认情况下，Camel 将抛出 Exception，但您可以启用这个选项并忽略这种情况。
closeCorrelationKeyOnCompletion		是否应该接受 **后期** 的 Exchanges。您可以启用此项来指示是否已完成关联密钥，然后与相同关联密钥的任何新交换都将被拒绝。然后，CamelationKeyException 异常会引发 **closedCorrelationKeyException** 异常。当使用这个选项时，会将一个整数传递，这是 LRUCache 的数字，保留最后 X 号右关联键。**您可以传递 0 或负数值来指示未绑定缓存。通过使用不同的关联密钥的日志，通过通过数字，确保缓存不会变得太大。**
discardOnCompletionTimeout	**false**	Camel 2.5： 由于超时而完成的交换应该被丢弃。如果启用，则当超时发生时，聚合的消息 **不会** 发出出去但丢弃（无意图）。
aggregationRepository		允许您自行插入 **org.apache.camel.spi.AggregationRepository** 实施，用于跟踪当前动态的聚合交换。Camel 默认使用基于内存的实施。
aggregationRepositoryRef		引用在 registry **中查询聚合 Repository**。
parallelProcessing	**false**	当聚合完成后，它们将从聚合器中发送。此选项指明 Camel 是否应使用带有多个线程的线程池来实现并发性。如果没有指定自定义线程池，Camel 会创建一个带有 10 个并发线程的默认池。

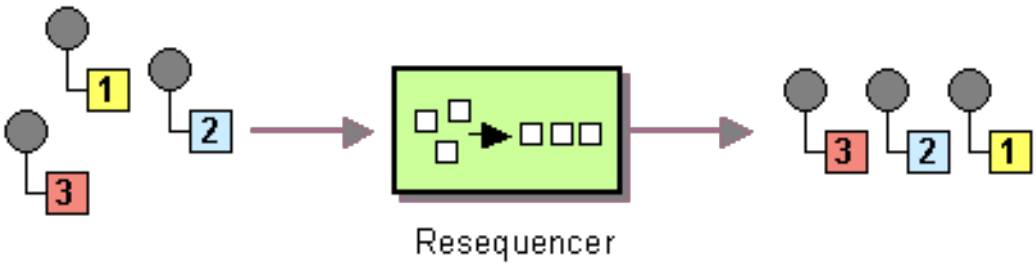
选项	默认值	描述
executorService		如果使用 **并行处理** 功能，您可以指定要使用的自定义线程池。实际上，如果您不使用 **并行处理** 这一自定义线程池，也用于发送聚合的交换。
executorServiceRef		在 Registry 中查询 **executorService** 的引用
timeoutCheckerExecutorService		如果使用一个 **completionTimeout**、**completionTimeoutExpression** 或 **completionInterval** 选项，则会创建一个后台线程来检查每个聚合器的完成情况。设置这个选项，以提供使用自定义线程池，而不是为每个聚合器创建新线程。
timeoutCheckerExecutorServiceRef		在 registry 中查找 **timeoutCheckerExecutorService** 的引用。
completeAllOnStop		当您停止聚合器时，这个选项允许它完成来自聚合存储库的所有待处理交换。
optimisticLocking	**false**	打开开放式锁定，它可以与聚合存储库结合使用。
optimisticLockRetryPolicy		为选择锁定配置重试策略。

8.6. RESEQUENCER

概述

***resequencer** 模式（如 图 8.7 “重新排序模式”）可让您根据顺序表达式重新排序信息。为 **sequencing** 表达式生成 **low** 值的消息将移到生成高值的批处理和消息前面的消息。*

图 8.7. 重新排序模式



Apache Camel 支持两种重新排序算法：

- 批处理重新排序 InventoryService-10.10.10.2 Collects 消息进入批处理中，对消息进行排序，并将它们发送到其输出。
- 根据消息间差距的检测，流重新排序 IFL 排序（持续）消息流。

默认情况下，resequencer 不支持重复消息，在消息到达相同的消息时，将仅保留最后一条消息。但是，在批处理模式下，您可以启用重新排序以允许重复。

批处理重新排序

默认启用批处理重新排序算法。例如，根据 TimeStamp 标头中包含的时间戳值重新排序传入的消息，您可以在 Java DSL 中定义以下路由：

```
from("direct:start").resequence(header("TimeStamp")).to("mock:result");
```

默认情况下，通过收集所有传入消息，以间隔 1000 毫秒（默认 批处理超时），最多获取批处理信息（默认 批处理大小）。您可以通过附加 batch（） DSL 命令来自定义批处理超时和批处理大小的值，该命令使用 BatchResequencerConfig 实例作为唯一参数。例如，若要修改前面的路由，使批处理由 4000 毫秒时间内收集的消息组成，最多 300 个消息，您可以定义 Java DSL 路由，如下所示：

```
import org.apache.camel.model.config.BatchResequencerConfig;

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start").resequence(header("TimeStamp")).batch(new
BatchResequencerConfig(300,4000L)).to("mock:result");
    }
};
```

您还可以使用 XML 配置指定批处理重新排序模式。以下示例定义了批处理大小为 300 的批处理重新排序，批处理超时为 4000 毫秒：

```
<camelContext id="resequencerBatch" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start" />
    <resequence>
      <!--
        batch-config can be omitted for default (batch) resequencer settings
      -->
```

```
      <batch-config batchSize="300" batchTimeout="4000" />
      <simple>header.TimeStamp</simple>
      <to uri="mock:result" />
    </resequence>
  </route>
</camelContext>
```

批处理选项

表 8.2 “批处理重新排序选项” 显示仅在批处理模式下可用的选项。

表 8.2. 批处理重新排序选项

Java DSL	XML DSL	默认	描述
allowDuplicates()	**batch-config/@allowDuplicates**	**false**	如果为 **true**，则不丢弃批处理中的重复消息（其中 **重复** 意味着消息表达式评估为相同的值）。
reverse()	**batch-config/@reverse**	**false**	如果为 **true**，则以相反顺序放置消息（其中应用于消息表达式的默认排序基于 Java 的字符串字典顺序，由 String.compareTo（）定义）。

例如，如果要根据 JMSPriority 重新排序来自 JMS 队列的消息，则需要组合选项、allowDuplicat 和 reverse，如下所示：

```
from("jms:queue:foo")
      // sort by JMSPriority by allowing duplicates (message can have same JMSPriority)
      // and use reverse ordering so 9 is first output (most important), and 0 is last
      // use batch mode and fire every 3th second
      .resequence(header("JMSPriority")).batch().timeout(3000).allowDuplicates().reverse()
      .to("mock:result");
```

流重新排序

要启用流重新排序算法，您必须将 stream（） 附加到 resequence（） DSL 命令中。例如，根据 seqnum 标头中的序列号值重新排序传入的信息，您需要定义一个 DSL 路由，如下所示：

```
from("direct:start").resequence(header("seqnum")).stream().to("mock:result");
```

流处理重新排序算法基于消息流中空缺的检测，而不是固定批量大小。差错检测与超时相结合，无需事先知道序列数（即批处理大小）的约束。消息必须包含唯一序列号，即 predecessor 和 a successor is known。例如，带有序列号 3 的消息带有序列号 2 的前身消息，以及序列号为 4 的后序消息。由于缺少 3 的后续情况，消息序列 2,3,5 有差距。因此，resequencer 必须保留消息 5，直到消息 4 到达（或超时发生）。

默认情况下，流 resequencer 配置超时为 1000 毫秒，最大消息容量为 100。要自定义流的超时和消息容量，您可以将 StreamResequencerConfig 对象作为参数传递到 stream（）。例如，若要将消息容量为 5000 且超时为 4000 毫秒配置流重新排序器，您需要定义路由，如下所示：

```
// Java
import org.apache.camel.model.config.StreamResequencerConfig;

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start").resequence(header("seqnum")).
            stream(new StreamResequencerConfig(5000, 4000L)).
            to("mock:result");
    }
};
```

如果成功消息之间的最大时间延迟（即，消息流中带有相邻顺序的消息）是已知的，则 resequencer 的 timeout 参数应设置为这个值。在这种情况下，您可以保证流中的所有信息都会被正确传送到下一个处理器。与不不透明时间的差值相比，超时值越低，可能更有可能将消息传送出序列。大型超时值应该获得足够高的容量值的支持，其中使用 capacity 参数以防止重新排序内存不足。

如果要使用 长期 某些类型的序列号，则必须定义自定义比较器，如下所示：

```
// Java
ExpressionResultComparator<Exchange> comparator = new MyComparator();
StreamResequencerConfig config = new StreamResequencerConfig(5000, 4000L, comparator);
from("direct:start").resequence(header("seqnum")).stream(config).to("mock:result");
```

您还可以使用 XML 配置指定流重新排序模式。以下示例定义了消息容量为 5000 且超时为 4000 毫秒的流重新排序：

```
<camelContext id="resequencerStream" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <resequence>
      <stream-config capacity="5000" timeout="4000"/>
      <simple>header.seqnum</simple>
      <to uri="mock:result" />
    </resequence>
  </route>
</camelContext>
```

忽略无效交换

如果由于某种原因无法评估 **sequencing** 表达式，**requencer EIP** 会抛出 **CamelExchangeException** 异常。如果传入交换无效，如果由于某种原因无法评估不同的表达式。您可以使用 **ignoreInvalidExchanges** 选项忽略这些异常，这意味着重新排序器会跳过任何无效的交换。

```
from("direct:start")
  .resequence(header("seqno")).batch().timeout(1000)
    // ignore invalid exchanges (they are discarded)
    .ignoreInvalidExchanges()
  .to("mock:result");
```

拒绝旧消息

rejectOld 选项可用于防止以任何顺序发送的消息，而不考虑用于重新排序消息的机制。启用 **rejectOld** 选项时，重新排序器拒绝传入的消息（通过抛出 **MessageRejectedException** 异常），如果传入的消息是 旧的 （由当前比较器定义）而不是上次发送的消息。

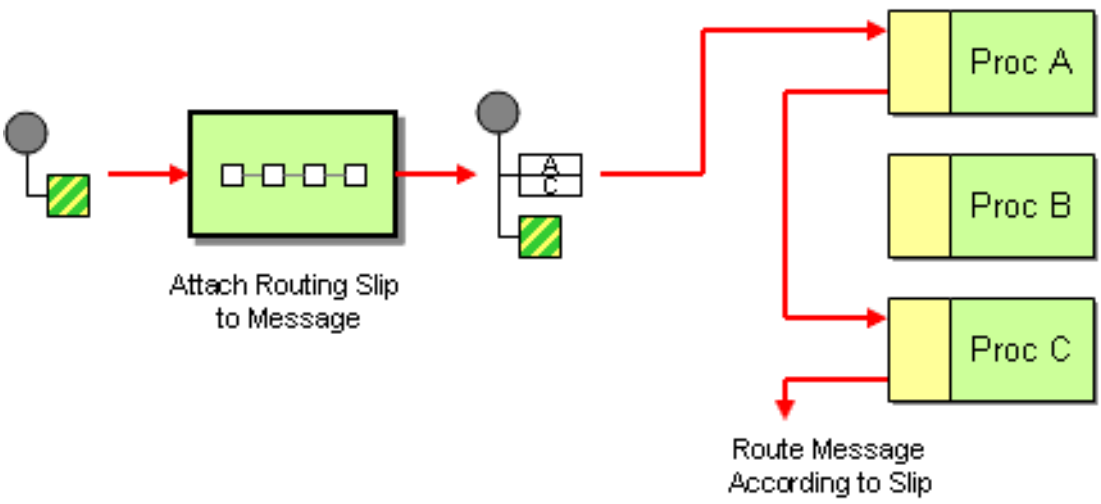
```
from("direct:start")
    .onException(MessageRejectedException.class).handled(true).to("mock:error").end()
    .resequence(header("seqno")).stream().timeout(1000).rejectOld()
    .to("mock:result");
```

8.7. 路由 SLIP

概述

路由 slip 模式（如 *图 8.8 “路由 Slip Pattern”* 所示）可让您连续通过一系列处理步骤路由消息，在设计时不已知这些步骤序列，并可能因每条消息而异。消息应传递的端点列表存储在标头字段中(**slip**)，**Apache Camel** 会在运行时读取以即时构建管道。

图 8.8. 路由 Slip Pattern



slip 标头

路由 slip 出现在用户定义的标头中，其中标头值是一个用逗号分开的端点 URI 列表。例如，一个路由 slip 用于指定一系列安全任务用于对：验证、验证和删除重复数据信息，类似于以下内容：

```
cxf:bean:decrypt,cxf:bean:authenticate,cxf:bean:dedup
```

当前的端点属性

从 Camel 2.5 中，路由 Slip 将在交换上设置一个属性(Exchange.SLIP_ENDPOINT)，该交换中包含了当前端点，这可以通过 slip 实现高级。这可让您了解交换是通过 slip 进行的。

第 8.7 节 “路由 Slip” 将预先计算一个滑动，即仅计算一次滑动。如果您需要计算 slip on-the-f，则使用 第 8.18 节 “动态路由器” 模式。

Java DSL 示例

以下路由从 direct:a 端点获取信息，并从 aRoutingSlipHeader 标头中读取路由 slip：

```
from("direct:b").routingSlip("aRoutingSlipHeader");
```

您可以将标头名称指定为字符串文字或表达式。

您也可以使用 routingSlip（） 的双参数形式自定义 URI 分隔符。以下示例定义了路由 slip 使用 aRoutingSlipHeader 标头密钥的路由，并使用 # 字符作为 URI 分隔符：

```
from("direct:c").routingSlip("aRoutingSlipHeader", "#");
```

XML 配置示例

以下示例演示了如何在 XML 中配置相同的路由：

```
<camelContext id="buildRoutingSlip" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:c"/>
    <routingSlip uriDelimiter="#">
      <headerName>aRoutingSlipHeader</headerName>
```

```
    </routingSlip>
  </route>
</camelContext>
```

忽略无效的端点

第 8.7 节 “路由 Slip” 现在支持 忽略InvalidEndpoints，第 8.3 节 “接收者列表” 模式支持它。您可以使用它来跳过无效端点。例如：

```
from("direct:a").routingSlip("myHeader").ignoreInvalidEndpoints();
```

在 Spring XML 中，通过在 < routingSlip> 标签上设置 ignoreInvalidEndpoints 属性来启用这个功能：

```
<route>
  <from uri="direct:a"/>
  <routingSlip ignoreInvalidEndpoints="true">
    <headerName>myHeader</headerName>
  </routingSlip>
</route>
```

请考虑 myHeader 包含两个端点的情况，direct:foo,xxx:bar。第一个端点有效且可以正常工作。第二个是无效，因此忽略。当遇到无效的端点时，Apache Camel 都会在 INFO 级别记录。

选项

routingSlip DSL 命令支持以下选项：

名称	默认值	描述
uriDelimiter	,	表达式返回的多个端点时使用分隔符。
ignoreInvalidEndpoints	**false**	如果无法解析 endpoint uri，它应该被忽略。否则，Camel 会抛出一个异常，说明 endpoint uri 无效。
cacheSize	**0**	Camel 2.13.1/2.12.4：允许配置 ProducerCache 的缓存大小，该缓存制作者在路由 slip 中重复使用。默认情况下，将使用默认的缓存大小，即 0。将值设为 -1 允许将缓存全部关闭。

8.8. THROTTLER

概述

throttler 是一个处理器，限制了传入消息的流率。您可以使用此模式来保护目标端点不受过载。在 Apache Camel 中，您可以使用 throttle（） Java DSL 命令实施节流模式。

Java DSL 示例

要将流率限制为每秒 100 个消息，请定义路由，如下所示：

```
from("seda:a").throttle(100).to("seda:b");
```

如果需要，您可以使用 timePeriodMillis（） DSL 命令自定义管理流率的时间段。例如，要将流率限制为每 30000 毫秒的 3 个消息，请按如下所示定义路由：

```
from("seda:a").throttle(3).timePeriodMillis(30000).to("mock:result");
```

XML 配置示例

以下示例演示了如何在 XML 中配置前面的路由：

```
<camelContext id="throttleRoute" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <!-- throttle 3 messages per 30 sec -->
    <throttle timePeriodMillis="30000">
      <constant>3</constant>
      <to uri="mock:result"/>
    </throttle>
  </route>
</camelContext>
```

动态更改每个期间的最大请求

可用的 Camel 2.8 Since 使用 Expression，您可以在运行时调整这个值，例如，为标头提供值。运行时 Camel 评估表达式并将结果转换为 java.lang.Long 类型。在以下示例中，我们使用邮件中的标头来决定每个期间的最大请求。如果缺少标头，则 第 8.8 节 “Throttler” 使用旧值。因此，在要更改值时，只允许提供标头：

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

```
    <from uri="direct:expressionHeader"/>
    <throttle timePeriodMillis="500">
      <!-- use a header to determine how many messages to throttle per 0.5 sec -->
      <header>throttleValue</header>
      <to uri="mock:result"/>
    </throttle>
  </route>
</camelContext>
```

异步延迟

throttler 可以启用 非阻塞异步延迟，这意味着 Apache Camel 将计划以后要执行的任务。该任务负责处理路由的后方（throttler 后）。这允许调用器线程取消阻塞和服务进一步传入的信息。例如：

```
from("seda:a").throttle(100).asyncDelayed().to("seda:b");
```

注意

从 Camel 2.17 中，Throttler 将使用滚动窗口进行时间，以便更好地处理消息。但是，它将提高节流的性能。

选项

throttle DSL 命令支持以下选项：

名称	默认值	描述
maximumRequestsPerPeriod		节流时的最大请求数。必须提供这个选项和正数。注意在 XML DSL 中，从 Camel 2.8 开始使用此选项时，使用 Expression 而不是属性来配置。
timePeriodMillis	**1000**	millis 中的时间周期，在 millis 中，节流允许 **最多允许最大 RequestsPerPeriod** 的消息数。
asyncDelayed	**false**	Camel 2.4： 如果启用，则任何使用调度的线程池延迟的消息都会进行。
executorServiceRef		Camel 2.4： 如果已启用 **asyncDelay**，请参阅使用自定义线程池。

callerRunsWhenRejected	**true**	Camel 2.4： 如果启用了 **asyncDelayed**，则会使用它。这将控制在线程池中拒绝该任务时调用器线程是否应该执行该任务。

8.9. DELAYER

概述

***delayer** 是一个处理器，可让您对传入消息应用 相对 时间延迟。*

Java DSL 示例

*您可以使用 **delay（） 命令**为传入消息添加 相对 时间延迟（以毫秒为单位）。例如，以下路由会根据 2 秒将所有传入的信息延迟：*

```
from("seda:a").delay(2000).to("mock:result");
```

另外，您可以使用表达式指定时间延迟：

```
from("seda:a").delay(header("MyDelay")).to("mock:result");
```

*以下 **delay（） 的 DSL 命令**解释为 **delay（） 的子clauses。因此，在某些上下文中，需要通过插入 end（） 命令来终止 delay（）的子使用。例如，当 delay（） 出现在 onException（） 子句内时，您将按以下方式将它终止：***

```
from("direct:start")
    .onException(Exception.class)
        .maximumRedeliveries(2)
        .backOffMultiplier(1.5)
        .handled(true)
        .delay(1000)
            .log("Halting for some time")
            .to("mock:halt")
        .end()
    .end()
    .to("mock:result");
```

XML 配置示例

*以下示例演示了 **XML DSL** 中的延迟：*

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <delay>
            <header>MyDelay</header>
        </delay>
        <to uri="mock:result"/>
    </route>
    <route>
        <from uri="seda:b"/>
        <delay>
            <constant>1000</constant>
        </delay>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

创建自定义延迟

您可以使用一个表达式与 bean 合并来确定延迟，如下所示：

```
from("activemq:foo").
  delay().expression().method("someBean", "computeDelay").
  to("activemq:bar");
```

这里可以定义 bean 类，如下所示：

```
public class SomeBean {
  public long computeDelay() {
     long delay = 0;
     // use java code to compute a delay value in millis
     return delay;
  }
}
```

异步延迟

您可以让延迟者使用 非阻塞异步延迟，这意味着 Apache Camel 将调度以后要执行的任务。该任务负责处理路由的后方（延迟者之后）。这允许调用器线程取消阻塞和服务进一步传入的信息。例如：

```
from("activemq:queue:foo")
    .delay(1000)
    .asyncDelayed()
    .to("activemq:aDelayedQueue");
```

同一路由可以使用 XML DSL 编写，如下所示：

```
<route>
  <from uri="activemq:queue:foo"/>
  <delay asyncDelayed="true">
     <constant>1000</constant>
  </delay>
  <to uri="activemq:aDealyedQueue"/>
  </route>
```

选项

delayer 模式支持以下选项：

名称	默认值	描述
asyncDelayed	**false**	Camel 2.4： 如果启用，则使用调度的线程池以异步方式进行延迟。
executorServiceRef		Camel 2.4： 如果已启用 **asyncDelay**，请参阅使用自定义线程池。
callerRunsWhenRejected	**true**	Camel 2.4： 如果启用了 **asyncDelayed**，则会使用它。这将控制在线程池中拒绝该任务时调用器线程是否应该执行该任务。

8.10. LOAD BALANCER

概述

负载均衡器模式允许您使用各种不同负载平衡策略将消息处理委派给多个端点之一。

Java DSL 示例

以下路由利用轮询负载平衡策略在目标端点和模拟 :x、模拟：y、模拟：z 之间分发传入的消息：

```
from("direct:start").loadBalance().roundRobin().to("mock:x", "mock:y", "mock:z");
```

XML 配置示例

以下示例演示了如何在 XML 中配置相同的路由：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <roundRobin/>
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

负载均衡策略

Apache Camel 负载均衡器支持以下负载均衡策略：

- 轮循
- 随机
- sticky
- Topic
- 故障切换
- 加权轮循，权重为随机
- 自定义负载均衡器
- 断路器

轮循

轮循负载平衡策略循环遍历所有目标端点，并将每个传入的消息发送到循环中的下一个端点。例如，如果目标端点列表为 模拟：x、模拟、模拟、模拟：z，则传入的消息将发送到以下端点序列： 模拟：x、模拟、模拟、模拟、模拟、模拟、模拟、模拟：z 等。

您可以在 Java DSL 中指定循环负载均衡策略，如下所示：

```
from("direct:start").loadBalance().roundRobin().to("mock:x", "mock:y", "mock:z");
```

另外，您可以在 XML 中配置相同的路由，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <roundRobin/>
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

随机

随机负载均衡策略从指定的列表随机选择目标端点。

您可以在 Java DSL 中指定随机负载均衡策略，如下所示：

```
from("direct:start").loadBalance().random().to("mock:x", "mock:y", "mock:z");
```

另外，您可以在 XML 中配置相同的路由，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <random/>
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
```

```
    </loadBalance>
  </route>
</camelContext>
```

sticky

粘性负载均衡策略将 In 消息定向到通过从指定表达式计算哈希值来选择的端点。此负载均衡策略的优点在于，相同值的表达式始终发送到同一服务器。例如，通过计算包含用户名的标头的 hash 值，可确保特定用户的消息始终发送到同一目标端点。另一个有用的方法是指定从传入消息中提取会话 ID 的表达式。这样可确保属于同一会话的所有信息都发送到同一目标端点。

您可以在 Java DSL 中指定粘性负载均衡策略，如下所示：

```
from("direct:start").loadBalance().sticky(header("username")).to("mock:x", "mock:y", "mock:z");
```

另外，您可以在 XML 中配置相同的路由，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <sticky>
        <correlationExpression>
          <simple>header.username</simple>
        </correlationExpression>
      </sticky>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

注意

当您将 sticky 选项添加到故障切换负载均衡器时，负载均衡器将从上次已知良好的端点启动。

Topic

主题负载平衡策略会将每个 In 消息的副本发送到所有列出的目的地端点（以经济的方式将消息广播到所有目的地，如 JMS 主题）。

您可以使用 Java DSL 指定主题负载均衡策略，如下所示：

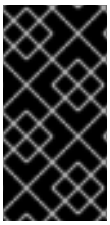
```
from("direct:start").loadBalance().topic().to("mock:x", "mock:y", "mock:z");
```

另外，您可以在 XML 中配置相同的路由，如下所示：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <topic/>
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

故障切换

自 Apache Camel 2.0 起，当 Exchange 失败时，故障转移 负载均衡器可以在处理过程中 出现异常时尝试下一个处理器。您可以使用触发 故障切换 的特定例外列表配置故障切换。如果您没有指定任何例外，则异常会触发故障切换。故障转移负载均衡器使用与 onException 异常相同的策略来匹配异常。



如果使用流启用流缓存

如果使用 streaming，您应该在使用 故障切换负载均衡器时启用流缓存。这是必要的，以便在失败时重新读取流。

故障转移 负载均衡器支持以下选项：

选项	类型	默认值	描述

inheritErrorHandler	布尔值	**true**	Camel 2.3： 指定是否使用路由中配置的 **错误 Handler**。如果要立即切换到下一个端点，您应该禁用这个选项（值为 **false**）。如果您启用这个选项，Apache Camel 将首先尝试使用 **错误 Handler** 来处理消息。 例如，**错误Handler** 可能会配置为红色等消息，并在尝试间使用延迟。Apache Camel 最初会 **尝试恢复原始** 端点，并且仅在 **错误Handler** 耗尽时切换到下一个端点。
maximumFailoverAttempts	**int**	**-1**	Camel 2.3： 指定切换到新端点的最大尝试次数。值 **0** 表示 **不会进行故障转移** 尝试，并且值 **-1** 表示故障转移尝试无限数。
roundRobin	布尔值	**false**	Camel 2.3： 指定 **故障转移** 负载均衡器是否以轮循模式运行。否则，**它将在** 处理新消息时始终从第一个端点启动。换句话说，它会在每次消息的顶部重新启动。如果启用了轮循，它会保持状态并以轮循方式继续。使用 round robin 时，**它将不** 切换为最后已知的好端点，它将始终选择要使用的下一个端点。

只有在引发 IOException 异常时，以下示例才会切换到 fail。

```
from("direct:start")
    // here we will load balance if IOException was thrown
    // any other kind of exception will result in the Exchange as failed
    // to failover over any kind of exception we can just omit the exception
    // in the failOver DSL
    .loadBalance().failover(IOException.class)
        .to("direct:x", "direct:y", "direct:z");
```

您可以选择指定多个例外来故障切换，如下所示：

```
// enable redelivery so failover can react
errorHandler(defaultErrorHandler().maximumRedeliveries(5));

from("direct:foo")
    .loadBalance()
    .failover(IOException.class, MyOtherException.class)
    .to("direct:a", "direct:b");
```

您可以在 XML 中配置相同的路由，如下所示：

```
<route errorHandlerRef="myErrorHandler">
    <from uri="direct:foo"/>
    <loadBalance>
        <failover>
            <exception>java.io.IOException</exception>
            <exception>com.mycompany.MyOtherException</exception>
        </failover>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

以下示例演示了如何以 round robin 模式进行故障切换：

```
from("direct:start")
    // Use failover load balancer in stateful round robin mode,
    // which means it will fail over immediately in case of an exception
    // as it does NOT inherit error handler. It will also keep retrying, as
    // it is configured to retry indefinitely.
    .loadBalance().failover(-1, false, true)
    .to("direct:bad", "direct:bad2", "direct:good", "direct:good2");
```

您可以在 XML 中配置相同的路由，如下所示：

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <!-- failover using stateful round robin,
        which will keep retrying the 4 endpoints indefinitely.
        You can set the maximumFailoverAttempt to break out after X attempts -->
        <failover roundRobin="true"/>
        <to uri="direct:bad"/>
        <to uri="direct:bad2"/>
        <to uri="direct:good"/>
```

```
        <to uri="direct:good2"/>
    </loadBalance>
</route>
```

如果要尽快切换到下一个端点，您可以通过配置 **inheritErrorHandler =false** 来禁用 **inheritErrorHandler**。通过禁用 **Error Handler**，您可以确保它不会干预。这样，故障转移负载均衡器可以尽快处理故障切换。如果您还要启用 **roundRobin** 模式，则会一直重试，直到成功为止。然后，您可以将 **maximumFailoverAttempts** 选项配置为高值，以便其最终耗尽和失败。

加权轮循，权重为随机

在许多企业环境中，如果无数处理能力的服务器节点正在托管服务，通常最好根据各个服务器处理容量来分发负载。可以使用 **加权轮循** 算法或 **加权随机** 算法来解决这个问题。

通过加权负载平衡策略，您可以为其他每台服务器指定处理负载 **分布比**。您可以将这个值指定为每台服务器的正处理权重。大于数字表示服务器可以处理更大的负载。处理 **weight** 用于确定与其他人相关的每个处理端点的有效负载分发率。

下表中描述了可以使用的参数：

表 8.3. 加权选项

选项	类型	默认值	描述
roundRobin	**布尔值**	**false**	round-robin 的默认值为 **false**。如果没有这一设置或参数，使用的负载平衡算法是随机的。
distributionRatioDelimiter	**字符串**	**,**	**distributionRatioDelimiter** 是用于指定 **distributionRatio** 的分隔符。如果没有指定此属性， 逗号是默认分隔符。

以下 **Java DSL** 示例演示了如何定义加权轮循路由和加权随机路由：

```
// Java
// round-robin
from("direct:start")
  .loadBalance().weighted(true, "4:2:1" distributionRatioDelimiter=":")
  .to("mock:x", "mock:y", "mock:z");
```

```
//random
from("direct:start")
  .loadBalance().weighted(false, "4,2,1")
  .to("mock:x", "mock:y", "mock:z");
```

您可以在 XML 中配置循环路由，如下所示：

```
<!-- round-robin -->
<route>
  <from uri="direct:start"/>
  <loadBalance>
    <weighted roundRobin="true" distributionRatio="4:2:1" distributionRatioDelimiter=":" />
    <to uri="mock:x"/>
    <to uri="mock:y"/>
    <to uri="mock:z"/>
  </loadBalance>
</route>
```

自定义负载均衡器

您还可以使用自定义负载均衡器（如您自己的实施）。

使用 Java DSL 的示例：

```
from("direct:start")
    // using our custom load balancer
    .loadBalance(new MyLoadBalancer())
    .to("mock:x", "mock:y", "mock:z");
```

以及使用 XML DSL 相同的示例：

```
<!-- this is the implementation of our custom load balancer -->
 <bean id="myBalancer"
class="org.apache.camel.processor.CustomLoadBalanceTest$MyLoadBalancer"/>

 <camelContext xmlns="http://camel.apache.org/schema/spring">
   <route>
     <from uri="direct:start"/>
     <loadBalance>
       <!-- refer to my custom load balancer -->
       <custom ref="myBalancer"/>
       <!-- these are the endpoints to balancer -->
       <to uri="mock:x"/>
       <to uri="mock:y"/>
       <to uri="mock:z"/>
```

```
    </loadBalance>
  </route>
</camelContext>
```

请注意，在上面的 XML DSL 中，使用 <custom>，它只在 Camel 2.8 起提供。在旧的版本中，您必须按如下操作：

```
<loadBalance ref="myBalancer">
  <!-- these are the endpoints to balancer -->
  <to uri="mock:x"/>
  <to uri="mock:y"/>
  <to uri="mock:z"/>
</loadBalance>
```

要实现自定义负载均衡器，您可以扩展一些支持类，如 LoadBalancerSupport 和 SimpleLoadBalancerSupport。前者支持异步路由引擎，而后者则不支持。下面是一个示例：

```
public static class MyLoadBalancer extends LoadBalancerSupport {

    public boolean process(Exchange exchange, AsyncCallback callback) {
        String body = exchange.getIn().getBody(String.class);
        try {
            if ("x".equals(body)) {
                getProcessors().get(0).process(exchange);
            } else if ("y".equals(body)) {
                getProcessors().get(1).process(exchange);
            } else {
                getProcessors().get(2).process(exchange);
            }
        } catch (Throwable e) {
            exchange.setException(e);
        }
        callback.done(true);
        return true;
    }
}
```

断路器

Circuit Breaker 负载均衡器是一个有状态模式，用于监控特定异常的所有调用。最初，断路器处于关闭状态并传递所有消息。如果失败并且达到阈值，则会进入 open 状态并拒绝所有调用，直到达到 一半 OpenAfter 超时。在超时后，如果存在新调用，则 Circuit Breaker 将传递所有消息。如果结果成功，则 Circuit Breaker 将变为闭路状态（如果不是），它会重新变为开状态。

Java DSL 示例：

```
from("direct:start").loadBalance()
    .circuitBreaker(2, 1000L, MyCustomException.class)
    .to("mock:result");
```

Spring XML 示例：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
    <from uri="direct:start"/>
    <loadBalance>
        <circuitBreaker threshold="2" halfOpenAfter="1000">
            <exception>MyCustomException</exception>
        </circuitBreaker>
        <to uri="mock:result"/>
    </loadBalance>
</route>
</camelContext>
```

8.11. HYSTRIX

概述

由 Camel 2.18 提供。

Hystrix 模式可让应用程序与 Netflix Hystrix 集成，这可以在 Camel 路由中提供断路器。Hystrix 是一个延迟和容错库，旨在

- **隔离对远程系统、服务和第三方库的访问点**
- **停止级联失败**
- **在复杂的分布式系统中启用弹性，其中故障不可避免**

如果您使用 maven，请在 pom.xml 文件中添加以下依赖项以使用 Hystrix：

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hystrix</artifactId>
```

```
    <version>x.x.x</version>
    <!-- Specify the same version as your Camel core version. -->
</dependency>
```

Java DSL 示例

以下是一个示例路由，它演示了 Hystrix 端点，通过回退到行的回退路由来防止运行缓慢。默认情况下，超时请求只是 1000ms，因此 HTTP 端点必须相当快速成功。

```
from("direct:start")
    .hystrix()
        .to("http://fooservice.com/slow")
    .onFallback()
        .transform().constant("Fallback message")
    .end()
    .to("mock:result");
```

XML 配置示例

以下是相同的示例，但在 XML 中：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <hystrix>
      <to uri="http://fooservice.com/slow"/>
      <onFallback>
        <transform>
          <constant>Fallback message</constant>
        </transform>
      </onFallback>
    </hystrix>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

使用 Hystrix 回退功能

onFallback（） 方法用于本地处理，您可以转换消息或调用 bean 或其他内容作为回退。如果您需要通过网络调用外部服务，那么您应该使用 onFallbackViaNetwork（） 方法，它在使用其自己的线程池的独立 HystrixCommand 对象中运行，使它不会耗尽第一个命令对象。

Hystrix 配置示例

Hystrix 有许多选项，如下一节中所述。以下示例显示，用于将执行超时设置为 5 秒的 Java DSL，而不是默认的 1 秒，并让断路器等待 10 秒，而不是 5 秒（默认），在状态停止打开前再次尝试请求。

```
from("direct:start")
    .hystrix()
        .hystrixConfiguration()
            .executionTimeoutInMilliseconds(5000).circuitBreakerSleepWindowInMilliseconds(10000)
        .end()
        .to("http://fooservice.com/slow")
    .onFallback()
        .transform().constant("Fallback message")
    .end()
    .to("mock:result");
```

以下是相同的示例，但在 XML 中：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <hystrix>
      <hystrixConfiguration executionTimeoutInMilliseconds="5000"
circuitBreakerSleepWindowInMilliseconds="10000"/>
      <to uri="http://fooservice.com/slow"/>
      <onFallback>
        <transform>
          <constant>Fallback message</constant>
        </transform>
      </onFallback>
    </hystrix>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

You can also configure Hystrix globally and then refer to that configuration. For example:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
   <!-- This is a shared config that you can refer to from all Hystrix patterns. -->
   <hystrixConfiguration id="sharedConfig" executionTimeoutInMilliseconds="5000"
circuitBreakerSleepWindowInMilliseconds="10000"/>

   <route>
        <from uri="direct:start"/>
        <hystrix hystrixConfigurationRef="sharedConfig">
        <to uri="http://fooservice.com/slow"/>
        <onFallback>
          <transform>
            <constant>Fallback message</constant>
          </transform>
        </onFallback>
      </hystrix>
```

```
    <to uri="mock:result"/>
  </route>
</camelContext>
```

选项

ths Hystrix 组件支持以下选项：Hystrix 提供默认值。

名称	默认值	类型	描述
circuitBreakerEnabled	**true**	布尔值	决定断路器是否用于跟踪健康和短路请求（如果它出差）。
circuitBreakerErrorThresholdPercentage	**50**	整数	设置在 或以上处的错误百分比，电路应开路并启动短路到回退逻辑。
circuitBreakerForceClosed	**false**	布尔值	值 true 会强制断路器进入闭路状态，而无论错误百分比如何，它都允许请求请求。
circuitBreakerForceOpen	**false**	布尔值	值 true 会强制断路器进入 open（出）状态，在其中拒绝所有请求。
circuitBreakerRequestVolumeThreshold	**20**	整数	设置出电路在滚动窗口中的最小请求数。
circuitBreakerSleepWindownInMilliseconds	**5000**	整数	设置断路器以拒绝请求的时间。在此时间过后，允许请求尝试来确定电路是否应该再次关闭。
commandKey	节点 ID	字符串	标识 Hystrix 命令。您无法配置这个选项。它始终是节点 ID，以使命令唯一。
corePoolSize	**10**	整数	设置核心 thread-pool 大小。这是可以同时执行的最大 **HystrixCommand** 对象数。

名称	默认值	类型	描述
executionIsolationSemaphoreMaxConcurrentRequests	**10**	整数	在使用 **ExecutionIsolationStrategy.SEMAPHORE** 时，设置 **HystrixCommand.run（）** 方法可以进行的最大请求数。
executionIsolationStrategy	**线程**	字符串	指示通过以下哪个隔离策略 **HystrixCommand.run（）** 执行。**THREAD** 在单独的线程上执行，并发请求受 thread-pool 中的线程数量的限制。**SEMAPHORE** 在调用线程上执行，且并发请求受 semaphore 数量的限制：
executionIsolationThreadInterruptOnTimeout	**true**	布尔值	指明在超时发生时是否应该中断 **HystrixCommand.run（）** 执行。
executionTimeoutInMilliseconds	**1000**	整数	为执行完成设置超时（以毫秒为单位）。
executionTimeoutEnabled	**true**	布尔值	指明是否应该 timed **HystrixCommand.run（）** 执行。
fallbackEnabled	**true**	布尔值	决定在发生故障或拒绝时是否尝试调用 **HystrixCommand.getFallback（）**。
fallbackIsolationSemaphoreMaxConcurrentRequests	**10**	整数	设置 **HystrixCommand.getFallback（）** 方法从调用线程发出的最大请求数。
groupKey	**CamelHystrix**	字符串	标识正在用来关联统计数据和断路器属性的 Hystrix 组。
keepAliveTime	**1**	整数	以分钟为单位设置 keep-alive 时间。

名称	默认值	类型	描述
maxQueueSize	**-1**	整数	设置 **BlockingQueue** 实施的最大队列大小。
metricsHealthSnapshotIntervalInMilliseconds	**500**	整数	在允许执行运行状况快照之间设置要等待的时间（以毫秒为单位）。健康快照计算成功和错误百分比，并影响断路器状态。
metricsRollingPercentileBucketSize	**100**	整数	设置每个存储桶保留的最大执行时间。如果在这段时间内进行更多执行，它们将换行，并在存储桶开始时进行覆盖。
metricsRollingPercentileEnabled	**true**	布尔值	指明是否应该跟踪执行延迟。延迟计算为百分比。值 false 会导致概述统计（mean，百分比数）返回为 -1。
metricsRollingPercentileWindowBuckets	**6**	整数	设置 **rollingPercentile** 窗口将被分成的 bucket 数量。
metricsRollingPercentileWindowInMilliseconds	**60000**	整数	设置执行时间的滚动窗口的持续时间，以便允许达到 50% 的计算时间（以毫秒为单位）。
metricsRollingStatisticalWindowBuckets	**10**	整数	设置滚动统计窗口的存储桶数量被分成.
metricsRollingStatisticalWindowInMilliseconds	**10000**	整数	这个选项和以下选项适用于从 **HystrixCommand** 和 **HystrixObservableCommand** 执行捕获指标。
queueSizeRejectionThreshold	**5**	整数	设定队列大小拒绝阈值 - 智能的最大队列大小，即使尚未达到**maxQueueSize** 也是如此。

名称	默认值	类型	描述
requestLogEnabled	**true**	布尔值	指明 **HystrixCommand** 执行和事件应记录到 **HystrixRequestLog**。
threadPoolKey	**null**	字符串	定义此命令应在其中运行的 thread-pool。默认情况下，这使用与 group 键相同的密钥。
threadPoolMetricsRollingStatisticalWindowBucket	**10**	整数	设置滚动统计窗口的存储桶数量被分成.
threadPoolMetricsRollingStatisticalWindowInMilliseconds	**10000**	整数	设置统计滚动窗口的持续时间，以毫秒为单位。这是为线程池保留指标的时长。

8.12. SERVICE CALL

概述

由 Camel 2.18 提供。

服务调用 模式允许您在分布式系统中调用远程服务。要调用的服务会在服务 registry 中查找，如 Kubernetes、Consul、etcd 或 Zookeeper。模式将服务 registry 的配置与调用服务分开。

Maven 用户必须为要使用的服务 registry 添加依赖项。可能性包括：

- *camel-consul*
- *camel-etcd*
- *camel-kubenetes*

- ***camel-ribbon***

调用服务的语法

要调用一个服务，请参考服务的名称，如下所示：

```
from("direct:start")
    .serviceCall("foo")
    .to("mock:result");
```

以下示例显示了用于调用服务的 XML DSL：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <serviceCall name="foo"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

在这些示例中，Camel 使用与服务 registry 集成的组件来查找服务名为 foo。lookup 返回一组 IP:PORT 对，指向托管远程服务的活跃服务器列表。然后，Camel 会随机选择该服务器，以使用该列表并使用所选 IP 和 PORT 编号构建 Camel URI。

默认情况下，Camel 使用 HTTP 组件。在上例中，调用将解析为由动态 到D 端点调用的 Camel URI，如下所示：

```
toD("http://IP:PORT")
```

```
<toD uri="http:IP:port"/>
```

您可以使用 URI 参数来调用服务，如 beer=yes ：

```
serviceCall("foo?beer=yes")
```

```
<serviceCall name="foo?beer=yes"/>
```

您还可以提供上下文路径，例如：

```
serviceCall("foo/beverage?beer=yes")
```

```
<serviceCall name="foo/beverage?beer=yes"/>
```

将服务名称转换为 URI

如您所见，服务名称会解析为 Camel 端点 URI。以下是更多示例。→ 显示 Camel URI 的解析）：

```
serviceCall("myService") -> http://hostname:port
serviceCall("myService/foo") -> http://hostname:port/foo
serviceCall("http:myService/foo") -> http:hostname:port/foo
```

```
<serviceCall name="myService"/> -> http://hostname:port
<serviceCall name="myService/foo"/> -> http://hostname:port/foo
<serviceCall name="http:myService/foo"/> -> http:hostname:port/foo
```

要完全控制已解析的 URI，请提供一个额外的 URI 参数，指定所需的 Camel URI。在指定的 URI 中，您可以使用服务名称，它解析为 IP:PORT。以下是一些示例：

```
serviceCall("myService", "http:myService.host:myService.port/foo") -> http:hostname:port/foo
serviceCall("myService", "netty4:tcp:myService?connectTimeout=1000") -> netty:tcp:hostname:port?
connectTimeout=1000
```

```
<serviceCall name="myService" uri="http:myService.host:myService.port/foo"/> ->
http:hostname:port/foo
<serviceCall name="myService" uri="netty4:tcp:myService?connectTimeout=1000"/> ->
netty:tcp:hostname:port?connectTimeout=1000
```

上面的示例调用名为 myService 的服务。第二个参数控制已解析的 URI 的值。请注意，第一个示例使用 serviceName.host 和 serviceName.port 来指代 IP 或 PORT。如果您只指定 serviceName，则会解析为 IP:PORT。

配置调用该服务的组件

默认情况下，Camel 使用 HTTP 组件调用该服务。您可以配置使用不同的组件（如 HTTP4 或 Netty4 HTTP）的使用，如下例所示：

```
KubernetesConfigurationDefinition config = new KubernetesConfigurationDefinition();
config.setComponent("netty4-http");

// Register the service call configuration:
context.setServiceCallConfiguration(config);
```

```
from("direct:start")
    .serviceCall("foo")
    .to("mock:result");
```

以下是 XML DSL 中的示例：

```
&lt;camelContext xmlns="http://camel.apache.org/schema/spring">
  &lt;kubernetesConfiguration id="kubernetes" component="netty4-http"/>
  &lt;route>
    &lt;from uri="direct:start"/>
    &lt;serviceCall name="foo"/>
    &lt;to uri="mock:result"/>
  &lt;/route>
&lt;/camelContext>
```

所有实现共享的选项

每个实现都可使用以下选项：

选项	默认值	描述
clientProperty		指定特定于您要使用的服务调用实现的属性。例如，如果您使用 ribbon 实现，则客户端属性在 **com.netflix.client.config.CommonClientConfigKey** 中定义。
component	**http**	设置用于调用远程服务的默认 Camel 组件。您可以配置使用组件，如 netty4-http、jetty、restlet 或一些其他组件。如果服务不使用 HTTP 协议，则必须使用另一个组件，如 mqtt、jms 和 amqp。如果您在 service 调用中指定 URI 参数，则会使用此参数中指定的组件而不是默认值。
loadBalancerRef		设置要使用的自定义 **org.apache.camel.spi.ServiceCallLoadBalancer** 的引用。
serverListStrategyRef		设置要使用的自定义 **org.apache.camel.spi.ServiceCallServerListStrategy** 的引用。

使用 Kubernetes 时的服务调用选项

Kubernetes 实现支持以下选项：

选项	默认值	描述
apiVersion		使用客户端查找时的 Kubernetes API 版本。
caCertData		使用客户端查找时设置证书颁发机构数据。
caCertFile		设置在使用客户端查找时从文件加载的证书颁发机构数据。
clientCertData		使用客户端查找时设置客户端证书数据。
clientCertFile		设置在使用客户端查找时从文件加载的客户端证书数据。
clientKeyAlgo		在使用客户端查找时设置客户端密钥存储算法，如 RSA。
clientKeyData		在使用客户端查找时设置客户端密钥存储数据。
clientKeyFile		使用客户端查找时，设置从文件加载的客户端密钥存储数据。
clientKeyPassphrase		在使用客户端查找时设置客户端密钥存储密码短语。
dnsDomain		设置 DNS 域，以用于 **dns** 查找。
lookup	**环境**	用于查找该服务的策略选择。lookup 策略包括： ● **环境** abrt-admission 使用环境变量。 ● **DNS** 效果为 use DNS 域名。 ● **客户端** abrt-jaxb 使用 Java 客户端调用 Kubernetes 主机 API 并查询哪些服务器正在主动托管该服务。

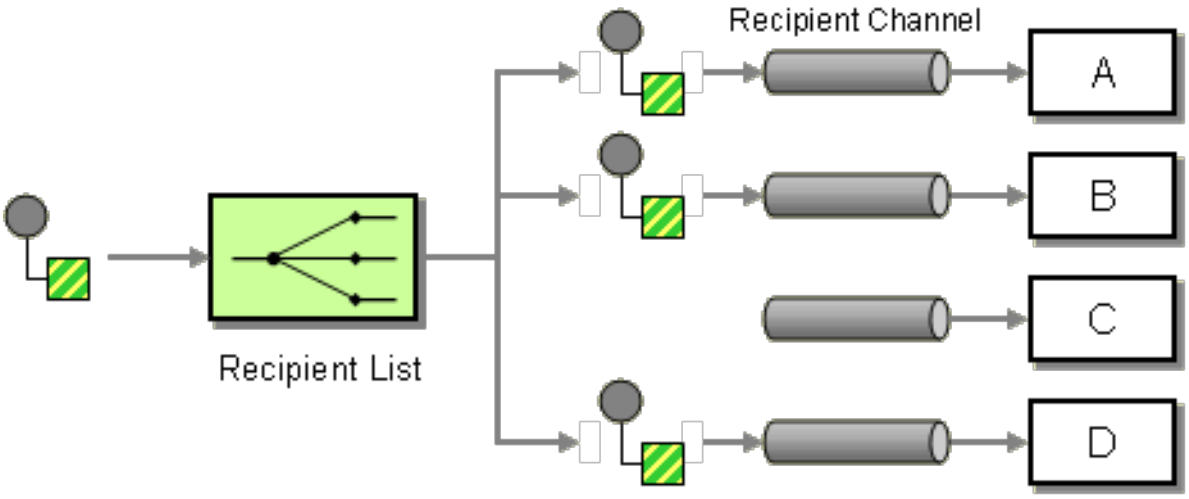
masterUrl		使用客户端查找时 Kubernetes 主机的 URL。
namespace		要使用的 Kubernetes 命名空间。默认情况下，命名空间的名称从环境变量 **KUBERNETES_MASTER** 中获取。
oauthToken		使用客户端查找时为身份验证设置 OAUTH 令牌（而不是用户名/密码）。
password		在使用客户端查找时设置密码以进行身份验证。
trustCerts	false	设置是否在使用客户端查找时打开信任证书检查。
username		在使用客户端查找时设置用于身份验证的用户名。

8.13. 多播

概述

多播 模式在 图 8.9 “多播模式” 中显示，它是带有固定目的地模式 的接收者列表 的一种变体，它与 InOut 消息交换模式兼容。这与接收者列表不同，它只与 InOnly Exchange 模式兼容。

图 8.9. 多播模式



使用自定义聚合策略的多播

多播处理器接收多个 Out 消息，以响应原始请求（来自每个接收方的一个请求），原始的调用者仅希望收到一个回复。因此，消息交换的回复图存在固有不匹配，并且要克服这种不匹配，您必须为多播处理器提供自定义 聚合策略。聚合策略类负责将所有 Out 消息聚合到一个回复消息中。

考虑电子 auction 服务示例，其中销售者提供销售商的一个项目，供购买者列表销售。买家对参与项目而言，销售者自动选择具有最高价格的投标。您可以使用 multicast（） DSL 命令实施将提供的项分发到固定购买者列表，如下所示：

```
from("cxf:bean:offer").multicast(new HighestBidAggregationStrategy()).
    to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

这里的 seller 由端点 cxf:bean:offer 代表，而 buyers 由端点、cxf:bean:Buyer1、cxf:bean:Buyer2、cxf:bean:bean:Buyer3 表示。为了整合从各种买方接收的 bid，多播处理器使用聚合策略 HighestBidAggregationStrategy。您可以在 Java 中实施 HighestBidAggregationStrategy，如下所示：

```
// Java
import org.apache.camel.processor.aggregate.AggregationStrategy;
import org.apache.camel.Exchange;

public class HighestBidAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        float oldBid = oldExchange.getOut().getHeader("Bid", Float.class);
        float newBid = newExchange.getOut().getHeader("Bid", Float.class);
        return (newBid > oldBid) ? newExchange : oldExchange;
    }
}
```

假设购买者将 bid 价格插入到名为 Bid 的标头中。有关自定义聚合策略的详情，请参阅 第 8.5 节 “聚合器”。

并行处理

默认情况下，多播处理器在另一个接收方（根据 to（） 命令中列出的顺序）调用每个接收者端点。在某些情况下，这可能导致不接受长延迟。为了避免这些长延迟时间，您可以选择通过添加 parallelProcessing（） 子句来启用并行处理。例如，要在电子 auction 示例中启用并行处理，请按如下所示定义路由：

```
from("cxf:bean:offer")
    .multicast(new HighestBidAggregationStrategy())
        .parallelProcessing()
        .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

在多播处理器现在调用购买器端点时，使用每个端点有一个线程的线程池。

如果要自定义调用购买器端点的线程池的大小，可以调用 **executorService（）** 方法来指定您自己的自定义执行者服务。例如：

```
from("cxf:bean:offer")
   .multicast(new HighestBidAggregationStrategy())
      .executorService(MyExecutor)
      .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

其中 **MyExecutor** 是 **java.util.concurrent.ExecutorService** 类型的实例。

当交换具有 **InOut** 模式时，会使用聚合策略来聚合回复消息。默认聚合策略采用最新的回复消息，并丢弃早期的回复。例如，以下路由（自定义策略 **MyAggregationStrategy** ）用于聚合来自端点的回复、**direct:a**、**direct:b** 和 **direct:c**：

```
from("direct:start")
  .multicast(new MyAggregationStrategy())
     .parallelProcessing()
     .timeout(500)
     .to("direct:a", "direct:b", "direct:c")
  .end()
  .to("mock:result");
```

XML 配置示例

以下示例演示了如何在 **XML** 中配置类似路由，路由会使用自定义聚合策略和自定义线程执行器：

```
<?xml version="1.0" encoding="UTF-8"?>
<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-3.0.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="cxf:bean:offer"/>
      <multicast strategyRef="highestBidAggregationStrategy"
                 parallelProcessing="true"
                 threadPoolRef="myThreadExcutor">
        <to uri="cxf:bean:Buyer1"/>
        <to uri="cxf:bean:Buyer2"/>
```

```
        <to uri="cxf:bean:Buyer3"/>
      </multicast>
    </route>
  </camelContext>

  <bean id="highestBidAggregationStrategy"
class="com.acme.example.HighestBidAggregationStrategy"/>
  <bean id="myThreadExcutor" class="com.acme.example.MyThreadExcutor"/>

</beans>
```

当 parallelProcessing 属性和 threadPoolRef 属性都是可选的。只有想自定义多播处理器的线程行为时，才需要设置它们。

将自定义处理应用到传出消息

多播模式 复制源 Exchange 和多播。默认情况下，路由器制作源消息的绝对副本。在应该复制中，原始消息的标头和有效负载只通过引用复制，以便生成原始消息的副本。由于多播消息的绝对副本已链接，因此如果消息正文器是可变的，则无法应用自定义处理。适用于发送到一个端点的副本的自定义处理也应用于发送到每个端点的副本。

注意

虽然 多播 语法允许您在 multicast 子句中调用 process DSL 命令，但这并不有意义上的意义，而且它 没有与 Prepare （实际上，在本例中为 ）相同的效果。

在准备消息时使用 onPrepare 执行自定义逻辑

如果要在将其发送到端点前将自定义处理应用到每个消息副本，您可以在 multicast 子句中调用 Prepare DSL 命令。onPrepare 命令只在消息被放送到其端点 前 插入自定义处理器。例如，在以下路由中调用 CustomProc 处理器，消息发送到 direct:a，自定义Proc 处理器也会在发送到 direct:b 的消息上调用。

```
from("direct:start")
  .multicast().onPrepare(new CustomProc())
  .to("direct:a").to("direct:b");
```

onPrepare DSL 命令的常见用例是对消息的部分或所有元素进行深入副本。例如，以下 CustomProc 处理器类执行消息正文的深层副本，其中消息正文假定为 type、Bdy Type，而深度副本则由方法、Bdy Type.deepCopy（） 执行。

```
// Java
import org.apache.camel.*;
...
```

```
public class CustomProc implements Processor {

    public void process(Exchange exchange) throws Exception {
        BodyType body = exchange.getIn().getBody(BodyType.class);

        // Make a _deep_ copy of of the body object
        BodyType clone =  BodyType.deepCopy();
        exchange.getIn().setBody(clone);

        // Headers and attachments have already been
        // shallow-copied. If you need deep copies,
        // add some more code here.
    }
}
```

您可以使用 onPrepare 来实现在 Exchange 多播前要执行的任何自定义逻辑。

注意

建议为不可变对象设计。

例如，如果您有 mutable 消息正文，作为这个 Animal 类：

```
public class Animal implements Serializable {

    private int id;
    private String name;

    public Animal() {
    }

    public Animal(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public Animal deepClone() {
        Animal clone = new Animal();
        clone.setId(getId());
        clone.setName(getName());
        return clone;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
```

```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return id + " " + name;
    }
}
```

然后，我们可以创建一个深层克隆消息正文：

```
public class AnimalDeepClonePrepare implements Processor {

    public void process(Exchange exchange) throws Exception {
        Animal body = exchange.getIn().getBody(Animal.class);

        // do a deep clone of the body which wont affect when doing multicasting
        Animal clone = body.deepClone();
        exchange.getIn().setBody(clone);
    }
}
```

然后，我们可以使用 onPrepare 选项在 多播路由 中使用 AnimalDeepClonePrepare 类，如下所示：

```
from("direct:start")
    .multicast().onPrepare(new AnimalDeepClonePrepare()).to("direct:a").to("direct:b");
```

和 XML DSL 中的相同示例

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <!-- use on prepare with multicast -->
        <multicast onPrepareRef="animalDeepClonePrepare">
            <to uri="direct:a"/>
            <to uri="direct:b"/>
        </multicast>
    </route>

    <route>
        <from uri="direct:a"/>
        <process ref="processorA"/>
```

```
        <to uri="mock:a"/>
    </route>
    <route>
        <from uri="direct:b"/>
        <process ref="processorB"/>
        <to uri="mock:b"/>
    </route>
 </camelContext>

 <!-- the on prepare Processor which performs the deep cloning -->
 <bean id="animalDeepClonePrepare"
class="org.apache.camel.processor.AnimalDeepClonePrepare"/>

 <!-- processors used for the last two routes, as part of unit test -->
 <bean id="processorA" class="org.apache.camel.processor.MulticastOnPrepareTest$ProcessorA"/>
 <bean id="processorB" class="org.apache.camel.processor.MulticastOnPrepareTest$ProcessorB"/>
```

选项

多播 DSL 命令支持以下选项：

名称	默认值	描述
strategyRef		指的是 AggregationStrategy，用于将多播的回复组合成来自多播的单一传出消息。???默认情况下，Camel 将使用最后的回复作为传出消息。
strategyMethodName		这个选项可用于明确指定要使用的方法名称，当 OVAs 用作 **AggregationStrategy** 时。
strategyMethodAllowNull	**false**	当将 POJOs 用作 **AggregationStrategy** 时，可以使用这个选项。如果为 **false**，则不会使用聚合方法，如果没有数据丰富。如果为 **true**，则使用空值作为 **oldExchange**，如果没有要增强数据，则使用**null** 值。
parallelProcessing	**false**	如果启用，则会同时将消息发送到多播。请注意，调用者线程仍会等待所有消息都完全处理，然后再继续。它只发送和处理来自同时发生的多播的回复。

parallelAggregate	**false**	如果启用，则 **AggregationStrategy** 上的聚合方法可以同时调用。**请注意，这需要实施 AggregationStrategy 为 thread-safe。默认情况下，此选项为 false，这表示 Camel 会自动同步对聚合方法的调用。然而，在一些用例中，您可以通过将 AggregationStrategy 作为 thread-safe，并将此选项设置为 true 来提高性能。**
executorServiceRef		指的是用于并行处理的自定义线程池。请注意，如果您设定了这个选项，则并行处理会被自动表示，您也不必启用该选项。
stopOnException	**false**	**Camel 2.2：** 出现异常时是否立即停止持续处理。如果禁用，则 Camel 会将消息发送到所有多播，无论它们之一是否失败。您可在完全控制如何处理它的 AggregationStrategy 类中处理异常。
streaming	**false**	如果启用，Camel 将按照其返回的顺序处理查询的查询。如果禁用，Camel 将按照与多播一样处理回复。
timeout		**Camel 2.5：** 设置以毫秒为单位指定的总超时。如果 多播 无法发送和处理给定时间段内的所有回复，则超时触发器和 多播 中断并继续。请注意，如果您提供 TimeoutAwareAggregationStrategy，则会在中断前调用 **超时** 方法。
onPrepareRef		**Camel 2.8：** 请参阅自定义处理器准备每个多播的副本。这可让您进行任何自定义逻辑，如 deep-cloning（如果需要）信息有效负载。

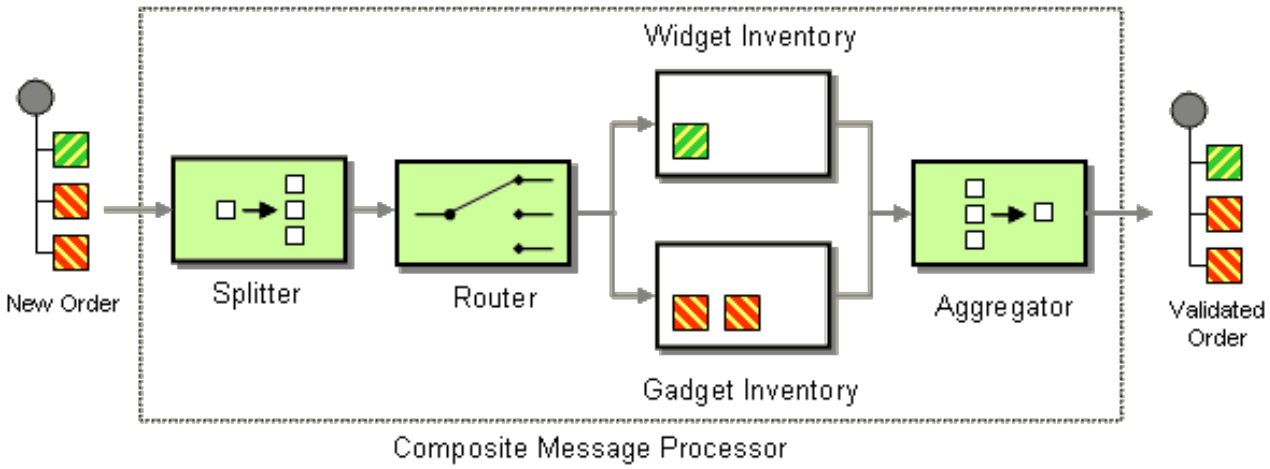
shareUnitOfWork	**false**	Camel 2.8： 是否应共享工作单元。详情请查看 第 8.4 节 “Splitter” 上的相同选项。

8.14. 由消息处理器

由消息处理器

如 图 8.10 “由消息处理器模式组成” 所示，组成的消息处理器 模式允许您通过分割来处理复合消息，将子消息路由到适当的目的地，然后将响应重新聚合到一个消息中。

图 8.10. 由消息处理器模式组成



Java DSL 示例

以下示例检查可以填写多部分顺序，其中每个顺序都要求检查不同的清单：

```
// split up the order so individual OrderItems can be validated by the appropriate bean
from("direct:start")
    .split().body()
    .choice()
        .when().method("orderItemHelper", "isWidget")
            .to("bean:widgetInventory")
        .otherwise()
            .to("bean:gadgetInventory")
    .end()
    .to("seda:aggregate");

// collect and re-assemble the validated OrderItems into an order again
from("seda:aggregate")
    .aggregate(new MyOrderAggregationStrategy())
    .header("orderId")
    .completionTimeout(1000L)
    .to("mock:result");
```

XML DSL 示例

以上路由也可以使用 XML DSL 编写，如下所示：

```
<route>
  <from uri="direct:start"/>
  <split>
    <simple>body</simple>
    <choice>
      <when>
        <method bean="orderItemHelper" method="isWidget"/>
 <to uri="bean:widgetInventory"/>
      </when>
      <otherwise>
 <to uri="bean:gadgetInventory"/>
      </otherwise>
    </choice>
    <to uri="seda:aggregate"/>
  </split>
</route>

<route>
  <from uri="seda:aggregate"/>
  <aggregate strategyRef="myOrderAggregatorStrategy" completionTimeout="1000">
    <correlationExpression>
      <simple>header.orderId</simple>
    </correlationExpression>
    <to uri="mock:result"/>
  </aggregate>
</route>
```

处理步骤

处理首先使用 第 8.4 节 “Splitter” 分割顺序。然后 第 8.4 节 “Splitter” 将独立的 OrderItems 发送到 第 8.1 节 “基于内容的路由器”，它根据项目类型路由信息。向 gadget Inventory bean 和 gadget 项中发送 小工具 项以检查项将发送到 gadgetInventory bean。在适当的 Bean 验证这些 OrderItems 后，它们会被发送到 第 8.5 节 “聚合器”，它会收集并重新集合验证的 OrderItems 作为顺序。

每个接收的顺序都有一个标头，其中包含一个 订购 ID。我们在聚合步骤中使用顺序 ID：aggregate（） DSL 命令上的 .header("orderId") 限定符，指示聚合器使用带有键 orderId 的标头，作为关联表达式。

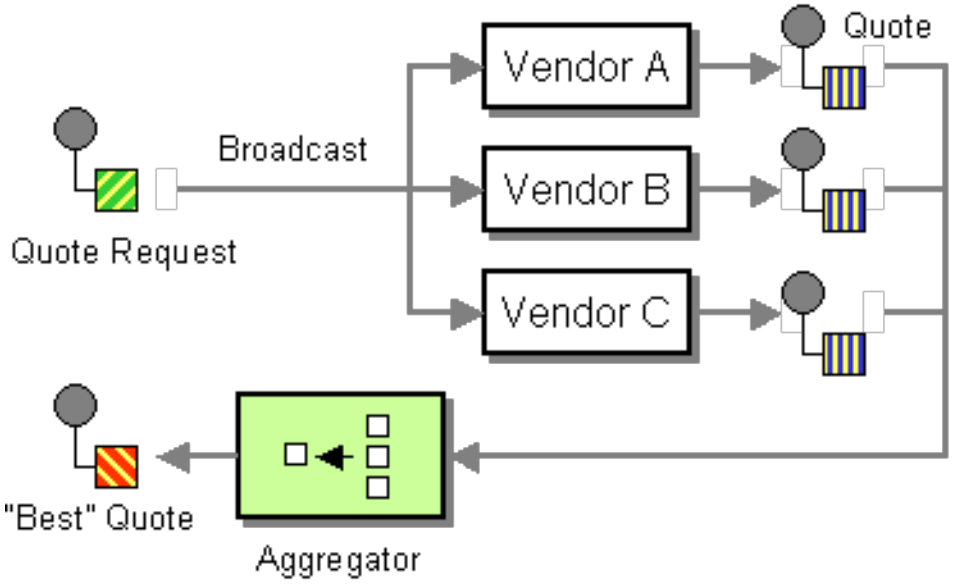
有关完整详情，请检查 camel-core/src/test/java/org/apache/camel/processor 中的 ComposedMessageProcessorTest.java 示例源。

8.15. SCATTER-GATHER

scatter-Gather

scatter-gather 模式 （如 图 8.11 “scatter-Gather Pattern” 所示）可让您将信息路由到多个动态指定接收方，并将响应重新分配给单一消息。

图 8.11. scatter-Gather Pattern



动态 scatter-gather 示例

以下示例概述了从多个不同供应商获得最佳报价的应用程序。这个示例使用动态 第 8.3 节 “接收者列表” 来请求来自所有供应商的引用和 第 8.5 节 “聚合器”，以选择所有响应的最佳引用。此应用程序的路由定义如下：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <recipientList>
      <header>listOfVendors</header>
    </recipientList>
  </route>
  <route>
    <from uri="seda:quoteAggregator"/>
    <aggregate strategyRef="aggregatorStrategy" completionTimeout="1000">
      <correlationExpression>
        <header>quoteRequestId</header>
      </correlationExpression>
      <to uri="mock:result"/>
    </aggregate>
  </route>
</camelContext>
```

在第一个路由中，第 8.3 节 “接收者列表” 查看 listOfVendors 标头以获取接收者列表。因此，向此应用发送消息的客户端需要在消息中添加 listOfVendors 标头。例 8.1 “消息传递客户端示例” 显示消息传递

客户端的一些示例代码，用于将相关的标头数据添加到传出消息中。

例 8.1. 消息传递客户端示例

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("listOfVendors", "bean:vendor1, bean:vendor2, bean:vendor3");
headers.put("quoteRequestId", "quoteRequest-1");
template.sendBodyAndHeaders("direct:start", "<quote_request item=\"beer\"/>", headers);
```

该消息将分发到以下端点：an :vendor1、bean:vendor2 和 bean:vendor3。这些 Bean 都由以下类实施：

```
public class MyVendor {
    private int beerPrice;

    @Produce(uri = "seda:quoteAggregator")
    private ProducerTemplate quoteAggregator;

    public MyVendor(int beerPrice) {
        this.beerPrice = beerPrice;
    }

    public void getQuote(@XPath("/quote_request/@item") String item, Exchange exchange) throws
Exception {
        if ("beer".equals(item)) {
            exchange.getIn().setBody(beerPrice);
            quoteAggregator.send(exchange);
        } else {
            throw new Exception("No quote available for " + item);
        }
    }
}
```

bean 实例、vendor1、 vendor2 和 vendor3 使用 Spring XML 语法进行实例化，如下所示：

```
<bean id="aggregatorStrategy"
class="org.apache.camel.spring.processor.scattergather.LowestQuoteAggregationStrategy"/>

<bean id="vendor1" class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>1</value>
  </constructor-arg>
</bean>

<bean id="vendor2" class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>2</value>
  </constructor-arg>
```

```
</bean>

<bean id="vendor3" class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>3</value>
  </constructor-arg>
</bean>
```

每个 bean 都使用不同的价格进行初始化（传递至 constructor 参数）。当消息发送到每个 bean 端点时，它会到达 MyVendor.getQuote 方法。这个方法确实会进行一次检查，查看此报价请求是否是 beer，然后在稍后的步骤上设置交换上的价格。该消息使用 HIPAA Prod ion 转发到下一步（请参阅 @Produce 注释）。

下一步，我们希望向所有供应商采用引号，并找出哪个是最佳（即最低）的引号。因此，我们使用带有自定义聚合策略的 第 8.5 节 “聚合器”。第 8.5 节 “聚合器” 需要识别与当前引用相关的消息，这通过根据 quoteRequestId 标头的值的关联信息（传递给 correlationExpression）来实现。如 例 8.1 “消息传递客户端示例” 所示，关联 ID 被设置为 quoteRequest-1（关联 ID 应该是唯一的）。要从集合中选择最低报价，您可以使用类似如下的自定义聚合策略：

```
public class LowestQuoteAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        // the first time we only have the new exchange
        if (oldExchange == null) {
            return newExchange;
        }

        if (oldExchange.getIn().getBody(int.class) < newExchange.getIn().getBody(int.class)) {
            return oldExchange;
        } else {
            return newExchange;
        }
    }
}
```

静态 scatter-gather 示例

您可以使用静态 第 8.3 节 “接收者列表” 在 scatter-gather 应用程序中明确指定接收者。以下示例显示了实施静态 scatter-gather 情境的路由：

```
from("direct:start").multicast().to("seda:vendor1", "seda:vendor2", "seda:vendor3");

from("seda:vendor1").to("bean:vendor1").to("seda:quoteAggregator");
from("seda:vendor2").to("bean:vendor2").to("seda:quoteAggregator");
from("seda:vendor3").to("bean:vendor3").to("seda:quoteAggregator");

from("seda:quoteAggregator")
    .aggregate(header("quoteRequestId"), new LowestQuoteAggregationStrategy()).to("mock:result")
```

8.16. LOOP

loop

loop 模式允许您多次处理消息。它主要用于测试。

默认情况下，循环在循环过程中使用相同的交换。上一个迭代的结果用于下一个（请参阅 第 5.4 节 “pipes 和 Filters”）。您可以从 上的 Camel 2.8 来启用复制模式。详情请查看 options 表。

Exchange 属性

在每个循环迭代中，会设置两个交换属性，可选择性地读取循环中包含的任何处理器。

属性	描述
CamelLoopSize	Apache Camel 2.0: 总数
CamelLoopIndex	Apache Camel 2.0：当前迭代的索引（基于 0）

Java DSL 示例

以下示例演示了如何从 direct:x 端点获取请求，然后重复将消息发送到 模拟结果。循环迭代数量被指定为 loop（） 的参数，或者通过评估表达式在运行时评估表达式，其中表达式 必须 评估到 int （或者其它 运行时操作Exception 被抛出）。

以下示例将循环计数作为恒定进行传递：

```
from("direct:a").loop(8).to("mock:result");
```

以下示例评估了一个简单的表达式来确定循环计数：

```
from("direct:b").loop(header("loop")).to("mock:result");
```

以下示例评估了一个 XPath 表达式来确定循环计数：

```
from("direct:c").loop().xpath("/hello/@times").to("mock:result");
```

XML 配置示例

您可以在 Spring XML 中配置相同的路由。

以下示例将循环计数作为恒定进行传递：

```
<route>
  <from uri="direct:a"/>
  <loop>
    <constant>8</constant>
    <to uri="mock:result"/>
  </loop>
</route>
```

以下示例评估了一个简单的表达式来确定循环计数：

```
<route>
  <from uri="direct:b"/>
  <loop>
    <header>loop</header>
    <to uri="mock:result"/>
  </loop>
</route>
```

使用复制模式

现在假定我们发送一条消息以包括字母 A。处理此路由的输出将是该路由，每个 模拟：loop 端点将接收 AB 作为消息。

```
from("direct:start")
    // instruct loop to use copy mode, which mean it will use a copy of the input exchange
    // for each loop iteration, instead of keep using the same exchange all over
    .loop(3).copy()
        .transform(body().append("B"))
        .to("mock:loop")
    .end()
    .to("mock:result");
```

但是，如果没有 启用复制模式，那么 loop 将接收 ABB、ABB 消息。

```
from("direct:start")
    // by default loop will keep using the same exchange so on the 2nd and 3rd iteration its
    // the same exchange that was previous used that are being looped all over
    .loop(3)
```

```
        .transform(body().append("B"))
        .to("mock:loop")
    .end()
    .to("mock:result");
```

复制模式的 XML DSL 中的等效示例如下：

```
<route>
  <from uri="direct:start"/>
  <!-- enable copy mode for loop eip -->
  <loop copy="true">
    <constant>3</constant>
    <transform>
      <simple>${body}B</simple>
    </transform>
    <to uri="mock:loop"/>
  </loop>
  <to uri="mock:result"/>
</route>
```

选项

loop DSL 命令支持以下选项：

名称	默认值	描述
复制	**false**	Camel 2.8： 是否使用复制模式。若为 **false**，则在循环过程中都会使用相同的 Exchange。因此，下一个迭代的结果会显示。**相反，您可以启用复制模式，然后每个迭代都会使用输入 "交换"一节 的新副本重启。**

Dole Loop

您可以执行循环，直到在 loop 使用一个条件满足条件。该条件可以是 true 或 false。

在 DSL 中，命令为 Loop DoWhile。以下示例将执行循环，直到邮件正文长度为 5 个字符或更少：

```
from("direct:start")
    .loopDoWhile(simple("${body.length} <= 5"))
        .to("mock:loop")
```

```
        .transform(body().append("A"))
    .end()
    .to("mock:result");
```

在 XML 中，命令是 loop 的 doWhile。以下示例还执行循环，直到邮件正文长度为 5 个字符或更少：

```
<route>
  <from uri="direct:start"/>
  <loop doWhile="true">
    <simple>${body.length} <= 5</simple>
    <to uri="mock:loop"/>
    <transform>
      <simple>A${body}</simple>
    </transform>
  </loop>
  <to uri="mock:result"/>
</route>
```

8.17. SAMPLING

sampling Throttler

sampling throttler 允许您通过路由从流量中提取交换示例。它被配置为一个抽样周期，期间仅允许单个 交换进行传递。将停止所有其他交换。

默认情况下，示例周期为 1 秒。

Java DSL 示例

使用 sample（） DSL 命令调用 sampler，如下所示：

```
// Sample with default sampling period (1 second)
from("direct:sample")
    .sample()
    .to("mock:result");

// Sample with explicitly specified sample period
from("direct:sample-configured")
    .sample(1, TimeUnit.SECONDS)
    .to("mock:result");

// Alternative syntax for specifying sampling period
from("direct:sample-configured-via-dsl")
    .sample().samplePeriod(1).timeUnits(TimeUnit.SECONDS)
    .to("mock:result");
```

```
from("direct:sample-messageFrequency")
    .sample(10)
    .to("mock:result");

from("direct:sample-messageFrequency-via-dsl")
    .sample().sampleMessageFrequency(5)
    .to("mock:result");
```

Spring XML 示例

在 Spring XML 中，使用 sample 元素调用 sampler，其中可以选择使用 samplePeriod 和 units 属性指定抽样周期：

```
<route>
    <from uri="direct:sample"/>
    <sample samplePeriod="1" units="seconds">
        <to uri="mock:result"/>
    </sample>
</route>
<route>
    <from uri="direct:sample-messageFrequency"/>
    <sample messageFrequency="10">
        <to uri="mock:result"/>
    </sample>
</route>
<route>
    <from uri="direct:sample-messageFrequency-via-dsl"/>
    <sample messageFrequency="5">
        <to uri="mock:result"/>
    </sample>
</route>
```

选项

DSL 命令示例支持以下选项：

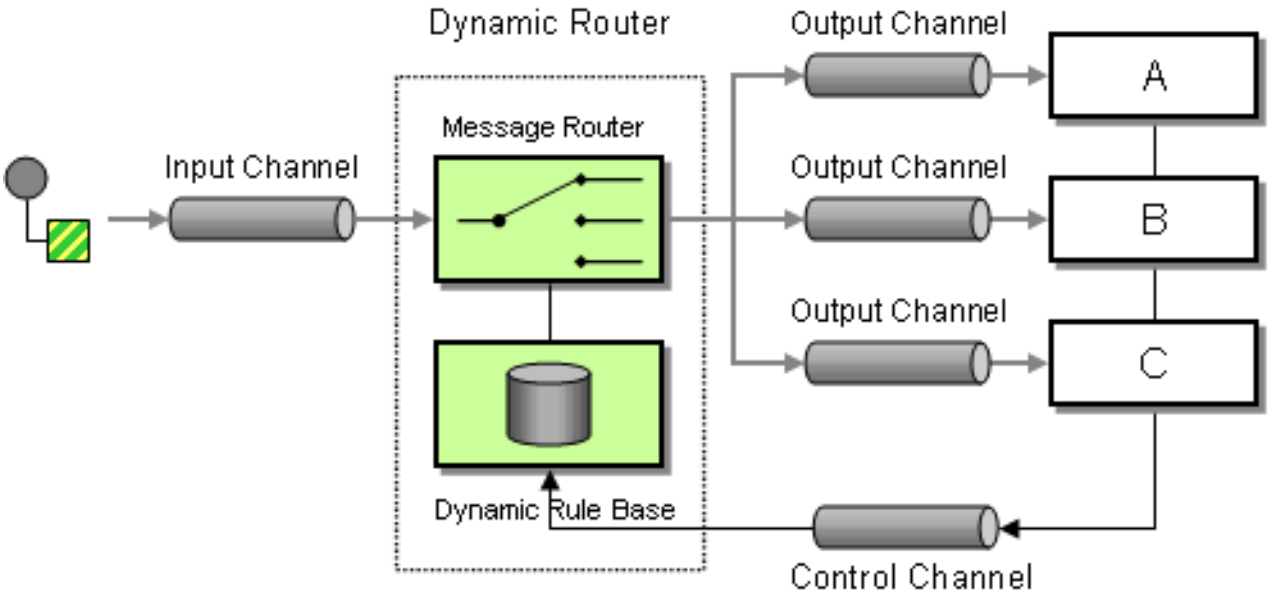
名称	默认值	描述
messageFrequency		每 N 的消息示例示例。只能使用频率或周期。
示例Period	**1**	每 N 期示例消息。只能使用频率或周期。
单位	**SECOND**	从 JDK，作为 **java.util.concurrent.TimeUnit** 的枚举单位。

8.18. 动态路由器

动态路由器

如 图 8.12 “动态路由器模式” 所示，动态路由器 模式允许您通过一系列处理步骤连续路由消息，其中设计时不已知步骤序列。在运行时动态计算消息的端点列表。每次消息从端点返回时，在 bean 上返回动态路由器调用，以便在路由中发现下一个端点。

图 8.12. 动态路由器模式



在 Camel 2.5 中，我们在 DSL 中引入了一个动态 路由器，类似于评估 slip on-fly 的动态 第 8.7 节 “路由 Slip”。

BEWARE

您必须确保用于 dynamicRouter 的表达式（如 bean）返回 null 以表示结尾。否则，dynamicRouter 会继续进行无限循环。

Camel 2.5 中的动态路由器开始

从 Camel 2.5 中，第 8.18 节 “动态路由器” 更新交换属性 Exchange.SLIP_ENDPOINT，其当前的端点通过 slip 推进。这可让您了解交换是通过 slip 进行的。（它是滑动，因为 第 8.18 节 “动态路由器” 的实现基于 第 8.7 节 “路由 Slip”。）

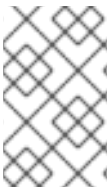
Java DSL

*在 **Java DSL** 中，您可以使用 动态**Router**，如下所示：*

```
from("direct:start")
    // use a bean as the dynamic router
    .dynamicRouter(bean(DynamicRouterTest.class, "slip"));
```

*它们利用 **bean** 集成来计算滑动 (**fly**)上的，该集成可以按以下方式实施：*

```
// Java
/**
 * Use this method to compute dynamic where we should route next.
 *
 * @param body the message body
 * @return endpoints to go, or <tt>null</tt> to indicate the end
 */
public String slip(String body) {
    bodies.add(body);
    invoked++;

    if (invoked == 1) {
        return "mock:a";
    } else if (invoked == 2) {
        return "mock:b,mock:c";
    } else if (invoked == 3) {
        return "direct:foo";
    } else if (invoked == 4) {
        return "mock:result";
    }

    // no more so return null
    return null;
    }
```



注意

前面的示例 不是 线程安全。您必须将状态存储在 Exchange 上，以确保线程安全。

Spring XML

*与 **Spring XML** 中的相同示例是：*

```
<bean id="mySlip" class="org.apache.camel.processor.DynamicRouterTest"/>

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

```
    <route>
      <from uri="direct:start"/>
      <dynamicRouter>
        <!-- use a method call on a bean as dynamic router -->
        <method ref="mySlip" method="slip"/>
      </dynamicRouter>
    </route>

    <route>
      <from uri="direct:foo"/>
      <transform><constant>Bye World</constant></transform>
      <to uri="mock:foo"/>
    </route>

  </camelContext>
```

选项

dynamicRouter DSL 命令支持以下选项：

名称	默认值	描述
uriDelimiter	**,**	第 II 部分 “路由表达式和指定语言” 返回多个端点时使用的分隔符。
ignoreInvalidEndpoints	**false**	如果无法解析 endpoint uri，它应该被忽略。否则，Camel 会抛出一个异常，说明 endpoint uri 无效。

@DYNAMICROUTER ANNOTATION

您还可以使用 @DynamicRouter 注释。例如：

```
// Java
public class MyDynamicRouter {

    @Consume(uri = "activemq:foo")
    @DynamicRouter
    public String route(@XPath("/customer/id") String customerId, @Header("Location") String
location, Document body) {
        // query a database to find the best match of the endpoint based on the input parameteres
        // return the next endpoint uri, where to go. Return null to indicate the end.
    }
}
```

路由 方法会在消息通过 slip 时重复调用。其理念是返回下一个目的地的端点 URI。返回 null 以表示结尾。如果您像 第 8.7 节 “路由 Slip” 一样，您可以返回多个端点，其中每个端点通过分隔符分隔。

第 9 章 SAGA EIP

9.1. 概述

Saga EIP 提供了一种方法，可以在 Camel 路由中定义一系列相关操作，这些操作可以成功完成或不能执行或编译。Saga 实施协调使用任何传输的分布式服务进行通信，达到全球一致性结果。Saga EIP 与典型的 ACID 分布式(XA)事务不同，因为保证不同参与服务的状态只在 Saga 结束时处于一致状态，而不是在任何中间步骤中保持一致。

Saga EIP 适用于不建议使用分布式事务的用例。例如，参与 Saga 的服务允许使用任何类型的数据存储，如类数据库，甚至 NoSQL 非事务数据存储。它们也适合用于无状态云服务，因为它们不需要将事务日志与服务一起存储。Saga EIP 也不需要在短时间内完成，因为它们不使用数据库级别的锁定，这与事务不同。因此，他们可以进行更长的时间，从几秒到几天。

Saga EIP 不使用数据上的锁定。相反，它们定义 Compensating Action 的概念，这是当标准流遇到错误时应执行的操作，以及恢复流执行前出现的状态的目的。可使用 Java 或 XML DSL 在 Camel 路由中声明 Compensating 操作，并且仅在需要时由 Camel 调用（如果 saga 被因为错误而被取消）。

9.2. SAGA EIP 选项

Saga EIP 支持以下列出的 6 个选项：

名称	描述	默认	类型
Propagation	设置 Saga propagation 模式（REQUIRED、REQUIRES_NEW、MANDATORY、SUPPORTS、NOT_SUPPORTED、NEVER）。	必需	SagaPropagation
completionMode	确定 Saga 如何处理完成。当设置为 **AUTO** 时，当成功处理启动 Saga 的交换时，Saga 会被完成，或者在特别处理后进行修复。当设置为 **MANUAL** 时，用户必须使用 **saga:complete** 或 **saga:compensate** 端点完成或编译 Saga。	AUTO	SagaCompletionMode
timeoutInMilliseconds	设置 Saga 的最大时间。在超时到期后，saga 会自动合并（除非在平均时间有不同的决定）。		Long
补偿	必须调用的过滤端点 URI，以弥合路由中执行的所有更改。与 compensation URI 对应的路由必须执行编译并完成且无错误。如果在编译过程中发生错误，Saga 服务会再次调用 compensation URI 来重试。		SagaActionUriDefinition

名称	描述	默认	类型
completion	成功完成 Saga 时调用的完成端点 URI。与完成 URI 对应的路由必须执行完成任务，并无错误终止。如果完成错误，Saga 服务会再次调用完成 URI 以重试。		SagaActionUriDefinition
选项	允许保存当前交换的属性，以便在合并或完成回调路由中重复使用它们。在编译操作中，选项通常很有用，用于存储和检索删除对象的标识符。选项值转换为合并/完成交换的输入标头。		list

9.3. SAGA SERVICE 配置

Saga EIP 要求实施接口 org.apache.camel.saga.CamelSagaService 的服务添加到 Camel 上下文中。Camel 目前支持以下 Saga 服务：

- *InMemorySagaService ：这是 Saga EIP 的基本 实现，它不支持高级功能（没有远程上下文传播，在应用程序失败时不能保证一致性）。*

9.3.1. 使用 In-Memory Saga 服务

不建议将 In-memory Saga 服务用于生产环境，因为它不支持 Saga 状态的持久性（只保留在内存中），因此它不能保证应用程序故障时 Saga EIP 的一致性（如 JVM 崩溃）。另外，在使用内存 Saga 服务时，Saga 上下文无法使用传输级别标头传播到远程服务（可以通过其他实现操作）。当使用内存 saga 服务时，您可以添加以下代码来自定义 Camel 上下文。该服务属于 camel-core 模块。

```
context.addService(new org.apache.camel.impl.saga.InMemorySagaService());
```

9.4. 例子

例如，您要放置一个新顺序，并在系统中有两个不同的服务：一个管理订单和一个管理学点。如果您有足够的学分，以逻辑方式排列一个订单。通过 Saga EIP，您可以将直接建模为 直接：buy route as a Saga aga aga aga aga EIP，一个用于创建顺序，另一个用于推行学。必须执行这两个操作，或记录 它们没有顺序，而没有字母顺序就可能被视为不一致的结果（以及没有订单的支付）。

```
from("direct:buy")
  .saga()
    .to("direct:newOrder")
    .to("direct:reserveCredit");
```

对于剩余的示例，购买操作不会改变。用于为 New Order 和 Reserve Point 操作建模不同的选项如下：

```
from("direct:newOrder")
  .saga()
  .propagation(SagaPropagation.MANDATORY)
  .compensation("direct:cancelOrder")
    .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
    .bean(orderManagerService, "newOrder")
    .log("Order ${body} created");
```

这里的 propagation 模式被设置为 MANDATORY。此路由中的任何交换流都必须已经属于 Saga（在此示例中是该例中，因为 Saga 在 direct:buy 路由中创建）。direct:newOrder 路由声明了一个名为 direct:cancelOrder 的编译操作，负责撤销 Saga 取消的顺序。

每个交换总是包含一个 Exchange.SAGA_LONG_RUNNING_ACTION 标头，这里用作订单的 id。这标识在相应的编译操作中要删除的顺序，但它不是要求（选项可用作替代解决方案）。跳过 direct:newOrder 的过滤操作是 direct:cancelOrder，它如下所示：

```
from("direct:cancelOrder")
  .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
  .bean(orderManagerService, "cancelOrder")
  .log("Order ${body} cancelled");
```

当订购应该被取消时，Saga EIP 实施会自动调用。它不会以错误结尾。如果在 direct:cancelOrder 路由中引发错误，EIP 实施应定期重试，以将操作编译到特定限制。这意味着，任何处理操作都必须是幂等的，因此应该考虑它可以被多次触发，且不应在任何情况下失败。如果在所有重试后无法进行编译，则 Saga 实施应触发手动干预过程。

注意

这可能是因为执行 直接：newOrder 路由中的延迟而发生，因此 Saga 在平均时间里被另一方取消（因为并行路由出现错误或 Saga 一级的超时）。因此，当调用编译操作 direct:cancelOrder 时，它可能无法找到被取消的 Order 记录。为了保证完全全局一致性，任何主要操作及其对应的过滤操作都是合情，例如，在主操作 应有相同的影响前，如果对操作进行编译，这很重要。

另一种可能的解决方法是不可能在编译操作中持续失败，直到找到主操作生成的数据（或者重试次数上限）。这种方法可以在很多上下文中工作，但这是一个极端 的。

支持服务的实施方式几乎与订购服务相同。

```
from("direct:reserveCredit")
  .saga()
  .propagation(SagaPropagation.MANDATORY)
  .compensation("direct:refundCredit")
    .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
    .bean(creditService, "reserveCredit")
    .log("Credit ${header.amount} reserved in action ${body}");
```

调用 compensation 操作：

```
from("direct:refundCredit")
  .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
  .bean(creditService, "refundCredit")
  .log("Credit for action ${body} refunded");
```

这里，对信用卡保留的换算行动是一个退还。

9.4.1. 处理编译事件

当 Saga 完成后，需要进行一些处理。当发生错误并取消 Saga 时，会调用相应的端点。可以调用 完成端点，以便在成功完成 Saga 时进一步处理。例如，根据以上顺序服务，可能需要知道订单已完成（保留的信度）以实际开始准备订单。如果没有完成支付，我们不想开始准备订单（与大多数现代 CPU 一样，可让您访问保留内存后再确保您有权读它）。这可以通过直接修改的 version :newOrder 端点轻松完成 ：

1. 调用完整的端点：

```
from("direct:newOrder")
  .saga()
  .propagation(SagaPropagation.MANDATORY)
  .compensation("direct:cancelOrder")
  .completion("direct:completeOrder")
    .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
    .bean(orderManagerService, "newOrder")
    .log("Order ${body} created");
```

1. direct:cancelOrder 与上例中的相同。在成功完成时调用，如下所示：

```
from("direct:completeOrder")
  .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
  .bean(orderManagerService, "findExternalId")
  .to("jms:prepareOrder")
  .log("Order ${body} sent for preparation");
```

Saga 完成后，订购将发送到 JMS 队列以进行准备。与处理操作一样，Saga 协调器（特别是针对错误，比如网络错误）可以多次调用完成操作。在本例中，侦听 prepareOrder JMS 队列的服务已准备好容纳可能的重复状态（请参阅 Idempent Consumer EIP EIP 如何处理重复）。

9.4.2. 使用自定义标识符和选项

您可以使用 Saga 选项来注册自定义标识符。例如，贡献度服务被重构，如下所示：

1. 生成自定义 ID 并在正文中设置它，如下所示：

```
from("direct:reserveCredit")
  .bean(idService, "generateCustomId")
  .to("direct:creditReservation")
```

1. 在编写操作中需要委派操作，并根据需要标记当前的正文。

```
from("direct:creditReservation")
  .saga()
  .propagation(SagaPropagation.SUPPORTS)
  .option("CreditId", body())
  .compensation("direct:creditRefund")
    .bean(creditService, "reserveCredit")
    .log("Credit ${header.amount} reserved. Custom Id used is ${body}");
```

1. 仅在 saga 被取消时从标头检索 creditId 选项。

```
from("direct:creditRefund")
  .transform(header("CreditId")) // retrieve the CreditId option from headers
  .bean(creditService, "refundCredit")
  .log("Credit for Custom Id ${body} refunded");
```

可以在 Saga 外部调用 direct:creditReservation 端点，方法是将 propagation 模式设置为 SUPPORTS。这样，可以在 Saga 路由中声明多个选项。

9.4.3. 设置超时

在 Saga EIP 中设置超时可确保 Saga 在机器发生故障时不会一直处于卡住状态。Saga EIP 实施在未明确指定它的所有 Saga EIP 上设置了默认超时。当超时过期时，Saga EIP 将决定 取消 Saga （并合约所有参与者），除非之前已进行了不同的决定。

在 Saga 参与者中可以按照如下方式设置超时：

```
from("direct:newOrder")
  .saga()
  .timeout(1, TimeUnit.MINUTES) // newOrder requires that the saga is completed within 1
minute
  .propagation(SagaPropagation.MANDATORY)
  .compensation("direct:cancelOrder")
  .completion("direct:completeOrder")
    // ...
    .log("Order ${body} created");
```

所有报名者（如服务、订单服务）都可以设置自己的超时。当由 saga 组成时，这些超时的最小值会作为超时。也可以在 Saga 级别指定超时，如下所示：

```
from("direct:buy")
  .saga()
  .timeout(5, TimeUnit.MINUTES) // timeout at saga level
    .to("direct:newOrder")
    .to("direct:reserveCredit");
```

9.4.4. 选择传播

在上面的示例中，我们使用了 MANDATORY 和 SUPPORTS 传播模式，也使用了 REQUIRED 传播模式，这是没有指定其他时使用的默认传播模式。这些传播模式映射 1:1 在事务上下文中使用的等效模式。

Propagation	描述
必需	加入现有 Saga 或创建新 Saga（如果不存在）。
REQUIRES_NEW	始终创建一个新的 Saga。挂起旧的 Saga，并在新卷终止时恢复它。
必需	Saga 必须已经存在。现有 Saga 已加入。
支持	如果 Saga 已存在，请加入它。
NOT_SUPPORTED	如果 Saga 已存在，它会在当前块完成后暂停并恢复。
NEVER	在 Saga 中不能调用当前块。

9.4.5. 使用手动完成（高级）

当 Saga 无法以同步的方式执行，但需要它，例如：使用异步通信频道与外部服务通信时，无法将完成模式设置为 AUTO （默认），因为 Saga 在创建交换时没有完成。这通常是具有长执行时间（小时、天）的 Saga EIP。在这些情况下，应使用 MANUAL 完成模式。

```
from("direct:mysaga")
  .saga()
  .completionMode(SagaCompletionMode.MANUAL)
  .completion("direct:finalize")
  .timeout(2, TimeUnit.HOURS)
    .to("seda:newOrder")
    .to("seda:reserveCredit");
```

为 seda:newOrder 和 seda:reserveCredit 添加异步处理。这些将异步回调发送到 seda:operationCompleted。

```
from("seda:operationCompleted") // an asynchronous callback
  .saga()
  .propagation(SagaPropagation.MANDATORY)
    .bean(controlService, "actionExecuted")
    .choice()
      .when(body().isEqualTo("ok"))
        .to("saga:complete") // complete the current saga manually (saga component)
    .end()
```

您可以添加 direct:finalize 端点来执行最终操作。

将完成模式设置为 MANUAL 表示，当路由 direct:mysaga 中处理交换时，Saga 不会被完成，但最后会更长时间（最大持续时间设置为 2 小时）。当两个异步操作都完成后，Saga 已完成。要完成的调用是使用 Camel Saga Component saga:complete 端点完成的。有一个类似的端点用于手动处理 Saga(saga:compensate)。

9.5. XML 配置

Saga 功能可供希望使用 XML 配置的用户使用。以下片段显示了一个示例：

```
<route>
  <from uri="direct:start"/>
  <saga>
    <compensation uri="direct:compensation" />
    <completion uri="direct:completion" />
    <option optionName="myOptionKey">
      <constant>myOptionValue</constant>
    </option>
    <option optionName="myOptionKey2">
```

```
      <constant>myOptionValue2</constant>
    </option>
  </saga>
  <to uri="direct:action1" />
  <to uri="direct:action2" />
</route>
```

第 10 章 消息转换

摘要

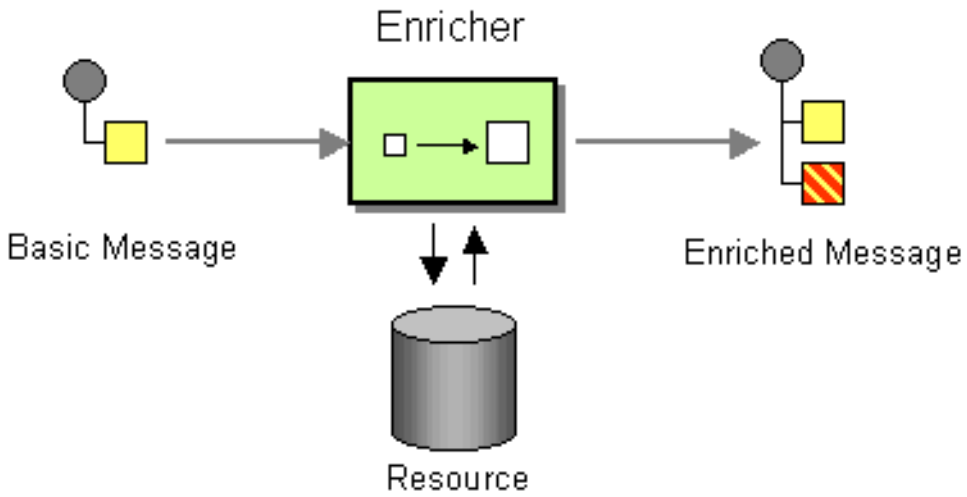
消息转换模式描述了如何为各种目的修改消息的内容。

10.1. 内容增强器

概述

内容增强器 模式描述了消息目的地需要超过原始消息中的数据的场景。在这种情况下，您要使用消息转换器、路由逻辑中的任意处理器，或者内容丰富的方法从外部资源拉取到额外数据。

图 10.1. 内容增强模式

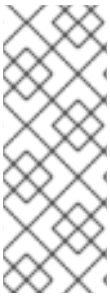


丰富内容的替代方案

Apache Camel 支持多种方式丰富内容：

- 带有路由逻辑中任意处理器的消息转换器
- enrich（） 方法通过向 制作者 端点发送当前交换的副本来获取来自资源的额外数据，然后在生成的回复中使用数据。由增强器创建的交换始终是一个 InOut 交换。
- pollEnrich（） 方法通过轮询 消费者 端点来获取额外的数据。实际上，来自主路由的消费者端点和 pollEnrich（） 操作中的使用者端点是结合的。也就是说，路由初始消费者上的传入消息

会触发用户轮询的 pollEnrich（） 方法。



注意

enrich（） 和 pollEnrich（） 方法支持动态端点 URI。您可以通过指定一个表达式来计算 URI，您可以从当前交换中获取值。例如，您可以使用从数据交换中计算的名称来轮询文件。Camel 2.16 中引入了此行为。这个更改会破坏 XML DSL，并可让您轻松迁移。Java DSL 保持向后兼容。

使用消息转换器和处理器来增强内容

Camel 提供 流畅的构建器，利用一种安全型 IDE 友好的方式创建路由和调解规则，从而提供智能完成并安全重构。当您测试分布式系统时，必须存根特定外部系统，以便可以在特定系统可用或编写特定系统之前测试系统的其他部分。执行此操作的一种方法是通过生成具有大部分静态正文的动态消息来生成对请求的响应。https://camel.apache.org/templating.html使用模板的另一种方法是使用来自一个目的地的消息，将其转换为 Velocity 或 XQuery 等内容，然后将其发送到另一目的地。以下示例显示了一个 InOnly （单向）消息：

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

假设您想使用 InOut （请求）消息传递来处理 ActiveMQ 上的 My.Queue 队列上的请求。您需要一个模板生成的响应，它指向 JMSReplyTo 目的地。以下示例演示了如何进行此操作：

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

以下简单示例演示了如何使用 DSL 转换消息正文：

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

以下示例使用显式 Java 代码来添加处理器：

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

下一个示例使用 bean 集成，使任何 bean 的使用作为转换器：

```
from("activemq:My.Queue").
  beanRef("myBeanName", "myMethodName").
  to("activemq:Another.Queue");
```

以下示例显示了一个 Spring XML 的实现：

```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>/>
```

使用 enrich（）方法丰富内容

```
AggregationStrategy aggregationStrategy = ...

from("direct:start")
  .enrich("direct:resource", aggregationStrategy)
  .to("direct:result");

from("direct:resource")
...
```

内容增强器（丰富）从 资源端点 检索额外的数据，以增强传入的消息（在机构 交换中包含）。聚合策略将原始交换与资源交换相结合。AggregationStrategy.aggregate（Exchange、Exchange） 方法的第一个参数与原始交换对应，第二个参数则对应于资源交换。资源端点的结果存储在资源交换的 Out 消息中。以下是实施您自己的聚合策略类的示例模板：

```
public class ExampleAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange original, Exchange resource) {
        Object originalBody = original.getIn().getBody();
        Object resourceResponse = resource.getOut().getBody();
        Object mergeResult = ... // combine original body and resource response
        if (original.getPattern().isOutCapable()) {
            original.getOut().setBody(mergeResult);
        } else {
            original.getIn().setBody(mergeResult);
        }
        return original;
    }

}
```

使用此模板时，原始交换可以具有任何交换模式。由增强器创建的资源交换始终是一个 InOut 交换。

Spring XML 丰富示例

前面的示例也可以在 Spring XML 中实施：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <enrich strategyRef="aggregationStrategy">
      <constant>direct:resource</constant>
    <to uri="direct:result"/>
  </route>
  <route>
    <from uri="direct:resource"/>
    ...
  </route>
</camelContext>
 <bean id="aggregationStrategy" class="..." />
```

增强内容时的默认聚合策略

聚合策略是可选的。如果没有提供它，Apache Camel 将默认使用从资源获取的正文。例如：

```
from("direct:start")
  .enrich("direct:resource")
  .to("direct:result");
```

在前面的路由中，发送到 direct:result 端点的消息包含来自 direct:resource 的消息，因为本例不使用任何自定义聚合。

在 XML DSL 中，只省略 strategyRef 属性，如下所示：

```
<route>
    <from uri="direct:start"/>
    <enrich uri="direct:resource"/>
    <to uri="direct:result"/>
</route>
```

enrich（）方法支持的选项

enrich DSL 命令支持以下选项：

名称	默认值	描述
expression	无	从 Camel 2.16 开始，需要这个选项。指定一个表达式，用于配置外部服务的 URI，以便从中丰富。您可以使用 Simple 表达式语言、Constant 表达式语言或任何可动态地从当前交换中的值动态计算 URI 的语言。
uri		这些选项已被删除。改为指定 **expression** 选项。在 Camel 2.15 及更早版本中，需要规格 **uri** 选项或 **ref** 选项。每个选项都指定要从中丰富的外部服务的端点 URI。
Ref		代表外部服务的端点，以丰富.您必须使用 **uri** 或 **ref**。
strategyRef		指的是 AggregationStrategy，用于将外部服务的回复合并到单一传出消息中。默认情况下，Camel 使用来自外部服务的回复作为传出消息。您可以使用 POJO 作为 **AggregationStrategy**。如需更多信息，请参阅 Aggregate 模式的文档。
strategyMethodName		使用 OVAs 作为 **AggregationStrategy** 时，指定这个选项来显式声明聚合方法的名称。详情请查看 Aggregate 模式。
strategyMethodAllowNull	false	默认行为是，如果没有要增强数据，则聚合方法不使用。如果这个选项为 true，则当没有数据丰富且您用作 **AggregationStrategy** 时，会使用 null 值作为 **旧的 Exchange**。如需更多信息，请参阅 Aggregate 模式。

aggregateOnException	false	默认行为是，如果试图从资源检索数据丰富的数据时，**不使用** 聚合方法。如果将此选项设置为 **true**，最终用户可以控制聚合方法中是否有异常时要做什么。**例如，可以阻止异常或设置自定义消息正文**
shareUntOfWork	false	从 Camel 2.16 开始，默认行为是，丰富的操作不共享父交换与资源交换之间的工作单元。这意味着资源交换都有自己的工作单元。如需更多信息，请参阅 Splitter 模式的文档。
cacheSize	**1000**	从 Camel 2.16 开始，指定这个选项来为 **ProducerCache** 配置缓存大小，这将缓存制作者以便在增强操作中重复使用。要关闭这个缓存，将 **cacheSize** 选项设置为 **-1**。
ignoreInvalidEndpoint	false	从 Camel 2.16 开始，这个选项指示是否忽略无法解析的端点 URI。默认行为是 Camel 引发标识无效端点 URI 的异常。

使用 enrich（）方法指定聚合策略

enrich（） 方法从资源端点检索额外的数据来增强传入的消息，该消息包含在原始交换中。您可以使用聚合策略来合并原始交换和资源交换。AggregationStrategy.aggregate（Exchange、Exchange）方法的第一个参数与原始交换对应。第二个参数对应于资源交换。资源端点的结果存储在资源交换的 Out 消息中。例如：

```
AggregationStrategy aggregationStrategy = ...

  from("direct:start")
  .enrich("direct:resource", aggregationStrategy)
  .to("direct:result");

  from("direct:resource")
...
```

以下代码是实施聚合策略的模板。在使用此模板的实现中，原始交换可以是任何消息交换模式。由 enricher 创建的资源交换始终是一个 InOut 消息交换模式。

```
public class ExampleAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange original, Exchange resource) {
        Object originalBody = original.getIn().getBody();
        Object resourceResponse = resource.getIn().getBody();
        Object mergeResult = ... // combine original body and resource response
        if (original.getPattern().isOutCapable()) {
            original.getOut().setBody(mergeResult);
        } else {
            original.getIn().setBody(mergeResult);
        }
        return original;
    }

}
```

以下示例显示了使用 Spring XML DSL 来实现聚合策略：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <enrich strategyRef="aggregationStrategy">
      <constant>direct:resource</constant>
    </enrich>
    <to uri="direct:result"/>
  </route>
  <route>
    <from uri="direct:resource"/>
    ...
  </route>
</camelContext>

<bean id="aggregationStrategy" class="..." />
```

使用带有 enrich（）的动态 URI

从 Camel 2.16 开始，enrich（） 和 pollEnrich（） 方法支持使用基于当前交换信息计算的动态 URI。例如，要从 HTTP 端点中丰富，带有 orderId 键的标头被用作 HTTP URL 的内容路径的一部分，您可以执行以下操作：

```
from("direct:start")
  .enrich().simple("http:myserver/${header.orderId}/order")
  .to("direct:result");
```

以下是 XML DSL 中相同的示例：

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

```
  <from uri="direct:start"/>
  <enrich>
    <simple>http:myserver/${header.orderId}/order</simple>
  </enrich>
  <to uri="direct:result"/>
</route>
```

使用 pollEnrich（）方法丰富内容

pollEnrich 命令会将资源端点视为 消费者。它将 轮询 端点，而不是向资源端点发送交换。默认情况下，如果资源端点没有可用的交换，则轮询会立即返回。例如，以下路由读取从传入 JMS 消息标头中提取其名称的文件：

```
from("activemq:queue:order")
  .pollEnrich("file://order/data/additional?fileName=orderId")
  .to("bean:processOrder");
```

您可以限制等待文件就绪的时间。以下示例显示了最大等待 20 秒：

```
from("activemq:queue:order")
  .pollEnrich("file://order/data/additional?fileName=orderId", 20000) // timeout is in milliseconds
  .to("bean:processOrder");
```

您还可以为 pollEnrich（） 指定聚合策略，例如：

```
  .pollEnrich("file://order/data/additional?fileName=orderId", 20000, aggregationStrategy)
```

pollEnrich（） 方法支持使用 consumer.bridgeErrorHandler=true 配置的用户。这样，可以从轮询传播到路由错误处理程序的任何异常（例如，重试轮询）。

注意

对 consumer.bridgeErrorHandler=true 的支持是 Camel 2.18 中的新功能。Camel 2.17 不支持此行为。

传递给聚合策略的 aggregate（） 方法的资源交换可能是 null （如果轮询超时）在收到交换前超时。

pollEnrich（）使用的轮询方法

pollEnrich() 方法通过调用以下轮询方法之一来轮询消费者端点：

- receiveNoWait ()（这是默认值。）
- receive()
- 接收（长超时）

pollEnrich() 命令的 timeout 参数（以毫秒为单位）决定要调用的方法，如下所示：

- 如果超时为 0 或未指定，则 pollEnrich() 调用 receiveNoWait。
- 当超时为负数时，pollEnrich() 调用会接收。
- 否则，pollEnrich() 调用 receive(timeout)。

如果没有数据，则聚合策略中的 newExchange 为 null。

使用 pollEnrich() 方法的示例

以下示例显示，通过从 inbox/data.txt 文件中载入内容来增强消息：

```
from("direct:start")
  .pollEnrich("file:inbox?fileName=data.txt")
  .to("direct:result");
```

以下是 XML DSL 中相同的示例：

```
<route>
  <from uri="direct:start"/>
  <pollEnrich>
    <constant>file:inbox?fileName=data.txt"</constant>
```

```
    </pollEnrich>
    <to uri="direct:result"/>
</route>
```

如果指定的文件不存在，则消息为空。您可以指定要等待的超时时间（永久），直到文件存在或等待到特定的时长。在以下示例中，命令会等待不超过 5 秒：

```
<route>
    <from uri="direct:start"/>
    <pollEnrich timeout="5000">
        <constant>file:inbox?fileName=data.txt"</constant>
    </pollEnrich>
    <to uri="direct:result"/>
</route>
```

使用带有 pollEnrich（）的动态 URI

从 Camel 2.16 开始，enrich（） 和 pollEnrich（） 方法支持使用基于当前交换信息计算的动态 URI。例如，要从使用标头来指示 SEDA 队列名称的端点中轮询 enrich，您可以执行以下操作：

```
from("direct:start")
  .pollEnrich().simple("seda:${header.name}")
  .to("direct:result");
```

以下是 XML DSL 中相同的示例：

```
<route>
    <from uri="direct:start"/>
    <pollEnrich>
        <simple>seda${header.name}</simple>
    </pollEnrich>
    <to uri="direct:result"/>
</route>
```

pollEnrich（）方法支持的选项

pollEnrich DSL 命令支持以下选项：

名称	默认值	描述

expression	无	从 Camel 2.16 开始，需要这个选项。指定一个表达式，用于配置外部服务的 URI，以便从中丰富。您可以使用 Simple 表达式语言、Constant 表达式语言或任何可动态地从当前交换中的值动态计算 URI 的语言。
uri		这些选项已被删除。改为指定 **expression** 选项。在 Camel 2.15 及更早版本中，需要规格 **uri** 选项或 **ref** 选项。每个选项都指定要从中丰富的外部服务的端点 URI。
Ref		代表外部服务的端点，以丰富.您必须使用 **uri** 或 **ref**。
strategyRef		指的是 AggregationStrategy，用于将外部服务的回复合并到单一传出消息中。默认情况下，Camel 使用来自外部服务的回复作为传出消息。您可以使用 POJO 作为 **AggregationStrategy**。如需更多信息，请参阅 Aggregate 模式的文档。
strategyMethodName		使用 OVAs 作为 **AggregationStrategy** 时，指定这个选项来显式声明聚合方法的名称。详情请查看 Aggregate 模式。
strategyMethodAllowNull	false	默认行为是，如果没有要增强数据，则聚合方法不使用。如果这个选项为 true，则当没有数据丰富且您用作 **AggregationStrategy** 时，会使用 null 值作为 **旧的 Exchange**。如需更多信息，请参阅 Aggregate 模式。
timeout	**-1**	从外部服务轮询时要等待响应的最长时间（以毫秒为单位）。默认行为是 **pollEnrich（）** 方法调用 **receive（）** 方法。因为 **receive（）** 可能会阻断，直到有可用的消息，所以建议始终指定超时。

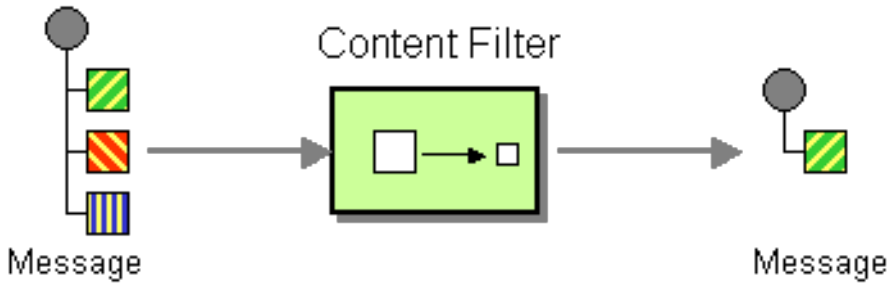
aggregateOnException	false	默认行为是，如果试图从资源检索数据丰富的数据时，**不使用** 聚合方法。如果将此选项设置为 **true**，最终用户可以控制聚合方法中是否有异常时要做什么。**例如，可以阻止异常或设置自定义消息正文**
cacheSize	**1000**	指定这个选项来为 **ConsumerCache** 配置缓存大小，该缓存消费者在 **pollEnrich（）** 操作中重复使用。要关闭这个缓存，将 **cacheSize** 选项设置为 **-1**。
ignoreInvalidEndpoint	false	指明是否忽略了无法解析的端点 URI。默认行为是 Camel 引发标识无效端点 URI 的异常。

10.2. 内容过滤器

概述

内容过滤器 模式描述了在将消息传送到预期收件人前需要从消息过滤额外内容的情况。例如，您可能使用内容过滤器从消息中剥离机密信息。

*图 **10.2. 内容过滤器模式***



过滤消息的常用方法是使用 DSL 中的表达式，以一种受支持的脚本语言（例如： XSLT、XQuery 或 JoSQL）编写。

实施内容过滤器

内容过滤器基本上是消息处理技术以特定用途的应用。要实施内容过滤器，您可以使用以下任何消息处理技术：

- **message translator** 时间为 第 5.6 节 “message Translator”。
- 处理器 **abrt-abrt** 请参阅 第 35 章 实现处理器。
- Bean 集成.

XML 配置示例

以下示例演示了如何在 XML 中配置相同的路由：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="xslt:classpath:com/acme/content_filter.xsl"/>
    <to uri="activemq:Another.Queue"/>
  </route>
</camelContext>
```

使用 XPath 过滤器

您还可以使用 XPath 过滤出您感兴趣的消息的一部分：

```
<route>
  <from uri="activemq:Input"/>
  <setBody><xpath resultType="org.w3c.dom.Document">//foo:bar</xpath></setBody>
  <to uri="activemq:Output"/>
</route>
```

10.3. 规范化程序

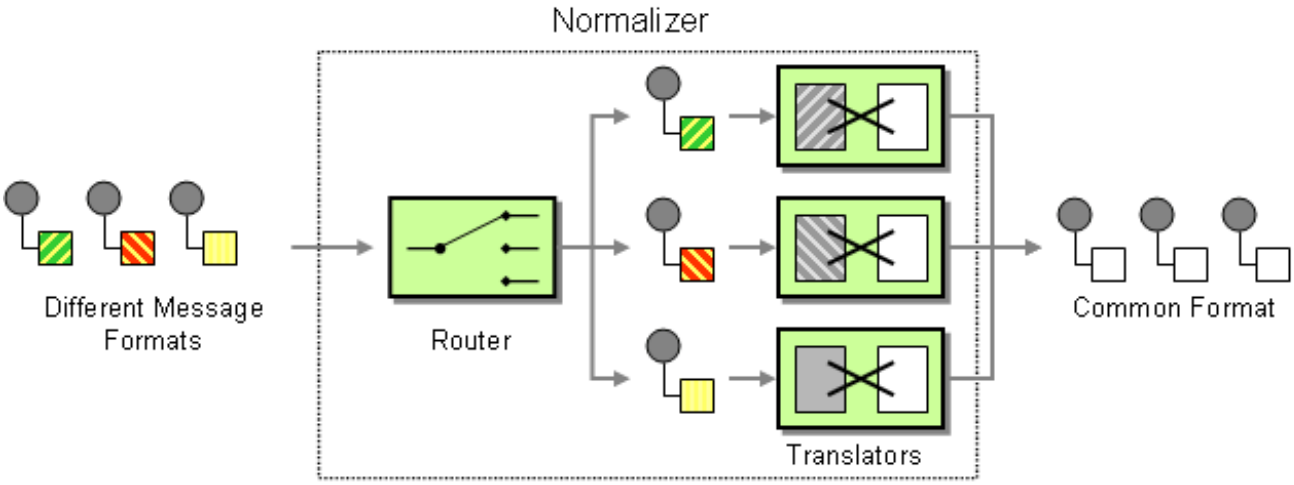
概述

规范化程序 模式用于处理语义等同的信息，但采用不同格式。规范化程序将传入的信息转换为常见格式。

在 Apache Camel 中，您可以合并 第 8.1 节 “基于内容的路由器” 来实现规范化程序模式，它检测到传入的消息的格式，使用不同的 第 5.6 节 “message Translator” 集合，它会将不同的传入的格式转换为

常见格式。

图 10.3. 规范化程序模式



Java DSL 示例

本例演示了 Message Normalizer，它将两种 XML 消息转换为常见格式。然后会过滤此通用格式的消息。

使用 Fluent Builders

```
// we need to normalize two types of incoming messages
from("direct:start")
    .choice()
        .when().xpath("/employee").to("bean:normalizer?method=employeeToPerson")
        .when().xpath("/customer").to("bean:normalizer?method=customerToPerson")
    .end()
    .to("mock:result");
```

在本例中，我们将 Java bean 用作规范化程序。类如下所示

```
// Java
public class MyNormalizer {
    public void employeeToPerson(Exchange exchange, @XPath("/employee/name/text()") String
name) {
        exchange.getOut().setBody(createPerson(name));
    }

    public void customerToPerson(Exchange exchange, @XPath("/customer/@name") String name) {
        exchange.getOut().setBody(createPerson(name));
    }

    private String createPerson(String name) {
```

```
        return "<person name=\"" + name + "\"/>";
    }
}
```

XML 配置示例

XML DSL 中的相同示例

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <xpath>/employee</xpath>
        <to uri="bean:normalizer?method=employeeToPerson"/>
      </when>
      <when>
        <xpath>/customer</xpath>
        <to uri="bean:normalizer?method=customerToPerson"/>
      </when>
    </choice>
    <to uri="mock:result"/>
  </route>
</camelContext>

<bean id="normalizer" class="org.apache.camel.processor.MyNormalizer"/>
```

10.4. 声明检查 EIP

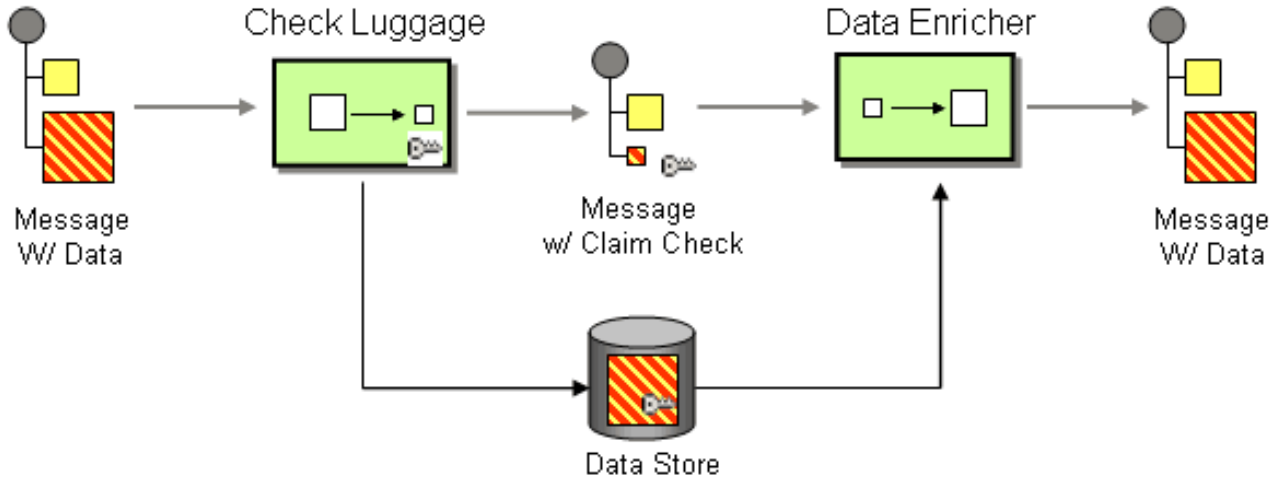
声明检查 EIP

通过 声明检查 EIP 模式，如 图 10.4 “声明检查模式” 所示，您可以将消息内容替换为一个声明检查（唯一键）。使用 声明检查 EIP 模式来稍后检索消息内容。您可以将消息内容临时存储在像数据库或文件系统这样的持久性存储中。当消息内容非常大（且要发送时，不是所有组件都需要所有信息时，这个模式非常有用。

当您无法信任外部的信息时，它也很有用。在这种情况下，使用 Claim Check 隐藏数据的敏感部分。

EIP 模式的 Camel 实施将消息内容临时存储在内部内存存储中。

图 10.4. 声明检查模式



10.4.1. 声明检查 EIP 选项

Claim Check EIP 支持下表中列出的选项：

名称	描述	默认	类型

operation	需要使用声明检查操作。它支持以下操作： * get - **获取** （不删除）给定密钥的声明检查。 * **GetAndRemove** - Gets 并删除给定键的声明检查。 * set - 使用给定键设置新的声明检查。**如果密钥已存在，它将被覆盖。** *** push - 在堆栈上设置新的声明检查（不要使用密钥）。** *** 弹出 - 从堆栈获取最新的声明检查（不要使用该密钥）。** **使用 Get、GetAndRemove 或 Set 操作时，您必须指定一个密钥。然后，这些操作将使用该密钥存储和检索数据。使用这些操作将多个数据存储在不同的密钥中。但是，推送和 弹出 操作不使用密钥，而是将数据存储到堆栈结构中。**		ClaimCheckOperation
key	使用特定密钥进行声明检查。		字符串

filter	指定过滤器来控制要从声明检查仓库中合并的数据。		字符串
strategyRef	使用自定义 **AggregationStrategy** 而不是默认的实现。您不能同时使用自定义聚合策略并配置数据。		字符串

过滤器选项

使用 Filter 选项定义在使用 Get 或 Pop 操作时重新合并的数据。使用 AggregationStrategy 来重新合并数据。默认策略使用过滤器选项轻松指定要合并的数据。

过滤器选项 ***采用带有以下语法的 String 值：***

- ***正文 ：聚合消息正文***
- ***Attachments ：聚合所有消息附件***
- *标头* ***：聚合所有消息标头***
- ***header:pattern: 聚合与模式匹配的所有消息标头***

模式规则支持通配符和正则表达式。

- ***通配符匹配项（模式以 * 结尾， 名称以模式开头）***
- ***正则表达式匹配***

要指定多个规则，用 逗号 （、 ）分开它们。

以下是包含邮件正文和以 foo 开始的所有标头的基本过滤器示例：

```
body, header:foo*
```

- 只合并消息正文： 正文
- 只合并消息附加： 附加
- 只合并标头： 标头
- 要合并标题名称 foo only: header:foo

如果您将过滤器规则指定为空的或通配符，您可以合并所有内容。如需更多信息，请参阅 过滤要合并的数据。

注意

当合并数据时，系统会覆盖任何现有的数据。它还会存储现有数据。

10.4.2. 使用 Include 和 Exclude Pattern 过滤选项

以下是支持用来指定 include、exclude 或 remove 选项的语法。

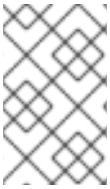
- + : 以包含 （默认模式）
- - : 要排除（排除的优先级高于 include）
- -- : 删除（删除优先）

例如：

- 要跳过消息正文并合并所有内容，请使用 -body

- 要跳过消息标头 foo 并合并所有其他内容，使用- -header:foo

您还可以指示系统在合并数据时删除标头。例如，要删除以条形开始的所有标题，使用- --headers:bar*。



注意

不要同时使用 include(+)和 exclude(-) header:pattern。

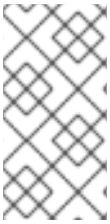
10.4.3. Java 示例

以下示例显示了操作中的 Push 和 Pop 操作：

```
from("direct:start")
   .to("mock:a")
   .claimCheck(ClaimCheckOperation.Push)
   .transform().constant("Bye World")
   .to("mock:b")
   .claimCheck(ClaimCheckOperation.Pop)
   .to("mock:c");
```

以下是使用 Get 和 Set 操作的示例：这个示例使用 foo 键。

```
from("direct:start")
   .to("mock:a")
   .claimCheck(ClaimCheckOperation.Set, "foo")
   .transform().constant("Bye World")
   .to("mock:b")
   .claimCheck(ClaimCheckOperation.Get, "foo")
   .to("mock:c")
   .transform().constant("Hi World")
   .to("mock:d")
   .claimCheck(ClaimCheckOperation.Get, "foo")
   .to("mock:e");
```



注意

您可以使用 **Get** 操作获取相同的数据两次，因为它不会删除数据。但是，如果您只想获取一次数据，请使用 **GetAndRemove** 操作。

以下示例演示了如何使用过滤器选项，其中您只想返回标题为 **foo** 或 **bar**。

```
from("direct:start")
    .to("mock:a")
    .claimCheck(ClaimCheckOperation.Push)
    .transform().constant("Bye World")
    .setHeader("foo", constant(456))
    .removeHeader("bar")
    .to("mock:b")
    // only merge in the message headers foo or bar
    .claimCheck(ClaimCheckOperation.Pop, null, "header:(foo|bar)")
    .to("mock:c");
```

10.4.4. XML 示例

以下示例显示了操作中的 **Push** 和 **Pop** 操作。

```
<route>
  <from uri="direct:start"/>
  <to uri="mock:a"/>
  <claimCheck operation="Push"/>
  <transform>
    <constant>Bye World</constant>
  </transform>
  <to uri="mock:b"/>
  <claimCheck operation="Pop"/>
  <to uri="mock:c"/>
</route>
```

以下是使用 **Get** 和 **Set** 操作的示例：这个示例使用 **foo** 键。

```
<route>
  <from uri="direct:start"/>
  <to uri="mock:a"/>
  <claimCheck operation="Set" key="foo"/>
  <transform>
    <constant>Bye World</constant>
  </transform>
  <to uri="mock:b"/>
  <claimCheck operation="Get" key="foo"/>
  <to uri="mock:c"/>
```

```
  <transform>
    <constant>Hi World</constant>
  </transform>
  <to uri="mock:d"/>
  <claimCheck operation="Get" key="foo"/>
  <to uri="mock:e"/>
</route>
```

注意

您可以使用 **Get** 操作获取相同的数据两次，因为它不会删除数据。但是，如果您要获取一次数据，可以使用 **GetAndRemove** 操作。

以下示例演示了如何使用过滤器选项将标头重新为 **foo** 或 **bar**。

```
<route>
  <from uri="direct:start"/>
  <to uri="mock:a"/>
  <claimCheck operation="Push"/>
  <transform>
    <constant>Bye World</constant>
  </transform>
  <setHeader headerName="foo">
    <constant>456</constant>
  </setHeader>
  <removeHeader headerName="bar"/>
  <to uri="mock:b"/>
  <!-- only merge in the message headers foo or bar -->
  <claimCheck operation="Pop" filter="header:(foo|bar)"/>
  <to uri="mock:c"/>
</route>
```

10.5. 排序

排序

sort 模式用于对邮件正文的内容进行排序，假设消息正文包含可以排序的项目列表。

默认情况下，消息的内容按照处理数字值或字符串的默认比较器进行排序。您可以提供自己的比较程序，您可以指定一个表达式，返回要排序的列表（表达式必须转换为 **java.util.List**）。

Java DSL 示例

以下示例生成按换行符字符的令牌排序的项目列表：

```
from("file://inbox").sort(body().tokenize("\n")).to("bean:MyServiceBean.processLine");
```

您可以将自己的比较程序传递给 **sort（） 的第二个参数：**

```
from("file://inbox").sort(body().tokenize("\n"), new
MyReverseComparator()).to("bean:MyServiceBean.processLine");
```

XML 配置示例

您可以在 **Spring XML** 中配置相同的路由。

以下示例生成按换行符字符的令牌排序的项目列表：

```
<route>
  <from uri="file://inbox"/>
  <sort>
    <simple>body</simple>
  </sort>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>
```

要使用自定义比较器，您可以将它作为 **Springan** 来引用：

```
<route>
  <from uri="file://inbox"/>
  <sort comparatorRef="myReverseComparator">
    <simple>body</simple>
  </sort>
  <beanRef ref="MyServiceBean" method="processLine"/>
</route>

<bean id="myReverseComparator" class="com.mycompany.MyReverseComparator"/>
```

除了 **& lt;simple > 外，您可以使用您喜欢的任何语言提供表达式，只要它返回列表。**

选项

sort DSL 命令支持以下选项：

名称	默认值	描述
comparatorRef		指的是自定义 **java.util.Comparator** 用于对消息正文进行排序。Camel 默认将使用执行 A.Z 排序的比较器。

10.6. TRANSFORMER

转换器根据路由定义上的声明的输入类型和/或输出类型执行消息声明性转换。默认 camel 消息实现 DataTypeAware，其中包含 DataType 表示的消息类型。

10.6.1. Transformer 的工作原理？

路由定义声明 输入类型和 /或 输出类型。如果输入类型和/或 输出类型 与运行时的消息类型不同，则 camel 内部处理器会查找 Transformer。Transformer 将当前消息类型转换为预期的消息类型。当消息被成功转换后，或者消息已处于预期类型后，则会更新消息数据类型。

10.6.1.1. 数据类型格式

数据类型格式为 scheme:name，其中 scheme 是 java、xml 或 json 等数据模型类型，name 是数据类型名称。

注意

如果您只指定 方案，则它会与该方案的所有数据类型匹配。

10.6.1.2. 支持的 Transformer

transformer	描述
Data Format Transformer	使用 Data Format 转换
endpoint Transformer	使用 Endpoint 转换
custom Transformer	使用自定义转换器类转换。

10.6.1.3. 常见选项

所有转换器都有以下常用选项，通过转换器指定受支持的数据类型。

重要

必须指定 从Type 和 toType 的 方案。

名称	描述
scheme	**xml** 或 **json** 等数据模型类型。例如，如果指定了 **xml**，则转换程序将适用于所有 java -> xml 和 xml -> java 转换。
fromType	从中 转换的数据类型。
toType	要 转换为的数据类型。

10.6.1.4. DataFormat Transformer 选项

名称	描述
type	数据格式类型
ref	引用数据格式 ID

指定 bindy DataFormat 类型的示例：

Java DSL：

```
BindyDataFormat bindy = new BindyDataFormat();
bindy.setType(BindyType.Csv);
bindy.setClassType(com.example.Order.class);
transformer()
   .fromType(com.example.Order.class)
   .toType("csv:CSVOrder")
   .withDataFormat(bindy);
```

XML DSL：

```
<dataFormatTransformer fromType="java:com.example.Order" toType="csv:CSVOrder">
    <bindy id="csvdf" type="Csv" classType="com.example.Order"/>
</dataFormatTransformer>
```

10.6.2. endpoint Transformer 选项

名称	描述
ref	引用端点 ID
uri	端点 URI

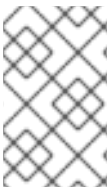
在 Java DSL 中指定端点 URI 的示例：

```
transformer()
    .fromType("xml")
    .toType("json")
    .withUri("dozer:myDozer?mappingFile=myMapping.xml...");
```

在 XML DSL 中指定端点 ref 的示例：

```
<transformers>
<endpointTransformer ref="myDozerEndpoint" fromType="xml" toType="json"/>
</transformers>
```

10.6.3. custom Transformer 选项



注意

转换器必须是 org.apache.camel.spi.Transformer的子类

名称	描述
ref	引用自定义 Transformer bean ID
className	自定义 Transformer 类的完全限定类名称

指定 custom Transformer 类的示例：

Java DSL：

```
transformer()
   .fromType("xml")
   .toType("json")
   .withJava(com.example.MyCustomTransformer.class);
```

XML DSL：

```
<transformers>
<customTransformer className="com.example.MyCustomTransformer" fromType="xml"
toType="json"/>
</transformers>
```

10.6.4. 转换器示例

这个示例分为两个部分，第一部分声明了 Endpoint Transformer，它转换了消息。第二部分演示了转换程序如何应用到路由。

10.6.4.1. 第 I 部分

声明使用 xslt 组件从 xml:ABCOrder 转换为 xml:XYZOrder 的 Endpoint Transformer。

Java DSL：

```
transformer()
   .fromType("xml:ABCOrder")
   .toType("xml:XYZOrder")
   .withUri("xslt:transform.xsl");
```

XML DSL：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
   <transformers>
      <endpointTransformer uri="xslt:transform.xsl" fromType="xml:ABCOrder"
toType="xml:XYZOrder"/>
   </transformers>
   ....
</camelContext>
```

10.6.4.2. 第 II 部分

当 direct:abc 端点发送消息来 direct:abc 端点将上述转换程序应用于以下路由定义：

Java DSL：

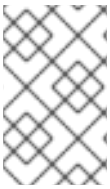
```
from("direct:abc")
   .inputType("xml:ABCOrder")
   .to("direct:xyz");
from("direct:xyz")
   .inputType("xml:XYZOrder")
   .to("somewhere:else");
```

XML DSL：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="direct:abc"/>
      <inputType urn="xml:ABCOrder"/>
      <to uri="direct:xyz"/>
   </route>
   <route>
      <from uri="direct:xyz"/>
      <inputType urn="xml:XYZOrder"/>
      <to uri="somewhere:else"/>
   </route>
</camelContext>
```

10.7. 验证器

验证器会根据声明的输入类型和/或输出类型对声明性消息执行声明性验证，该定义声明了预期的消息类型。



注意

只有在 type 声明中的 validate 属性为 true 时才执行验证。

如果在输入类型和/或输出类型声明上 验证 属性为 true，则 camel 内部处理器会从 registry 查找对应的验证器。

10.7.1. 数据类型格式

数据类型格式为 scheme:name，其中 scheme 是 Java、xml 或 json，name 是数据类型名称。

10.7.2. 支持的验证器

验证器	描述
predicate Validator	使用 Expression 或 Predicate 进行验证
endpoint Validator	通过转发到与验证组件（如 Validation 组件或 Bean Validation 组件）搭配使用的 Endpoint 验证组件进行验证。
自定义验证器	使用自定义验证器类进行验证。验证器必须是 **org.apache.camel.spi.Validator**的子类

10.7.3. 常见选项

所有验证器都必须包括指定 Data type to validate 的 type 选项。

10.7.4. predicate 验证器选项

名称	描述
expression	用于验证的表达式或优先级。

指定验证 predicate 的示例：

Java DSL：

```
validator()
   .type("csv:CSVOrder")
   .withExpression(bodyAs(String.class).contains("{name:XOrder}"));
```

XML DSL：

```
<predicateValidator Type="csv:CSVOrder">
   <simple>${body} contains 'name:XOrder'</simple>
</predicateValidator>
```

10.7.5. 端点验证选项

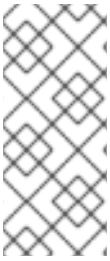
名称	描述
ref	引用端点 ID。
uri	端点 URI.

在 Java DSL 中指定端点 URI 的示例：

```
validator()
   .type("xml")
   .withUri("validator:xsd/schema.xsd");
```

在 XML DSL 中指定端点 ref 的示例：

```
<validators>
<endpointValidator uri="validator:xsd/schema.xsd" type="xml"/>
</validators>
```



注意

Endpoint Validator 将消息转发到指定的端点。在上例中，camel 会将消息转发到 validator: 端点，它是一个 Validation 组件。您还可以使用不同的验证组件，如 Bean Validation 组件。

10.7.6. 自定义验证选项

注意

Validator 必须是 org.apache.camel.spi.Validator的子类

名称	描述
ref	引用自定义验证器 bean ID。
className	自定义验证类类的全限定类名称。

指定自定义验证类的示例：

Java DSL：

```
validator()
  .type("json")
  .withJava(com.example.MyCustomValidator.class);
```

XML DSL：

```
<validators>
<customValidator className="com.example.MyCustomValidator" type="json"/>
</validators>
```

10.7.7. 验证器示例

此示例分为两个部分，第一部分声明了用于验证消息的 Endpoint Validator。第二部分显示验证器如何应用到路由。

10.7.7.1. 第 I 部分

声明使用验证器组件从 xml:ABCOrder 进行身份验证的 Endpoint Validator。

Java DSL：

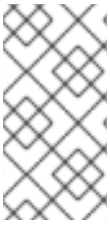
```
validator()
  .type("xml:ABCOrder")
  .withUri("validator:xsd/schema.xsd");
```

XML DSL：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <validators>
    <endpointValidator uri="validator:xsd/schema.xsd" type="xml:ABCOrder"/>
  </validators>
</camelContext>
```

10.7.7.2. 第 II 部分

当 direct:abc 端点收到消息时，上述验证器会应用到以下路由定义。



注意

inputTypeWithValidate 被使用在 Java DSL 中而不是 inputType，在 XML DSL 中将 inputType 声明中的 verify 属性设置为 true ：

Java DSL ：

```
from("direct:abc")
  .inputTypeWithValidate("xml:ABCOrder")
  .log("${body}");
```

XML DSL ：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:abc"/>
    <inputType urn="xml:ABCOrder" validate="true"/>
    <log message="${body}"/>
  </route>
</camelContext>
```

10.8. VALIDATE

概述

验证模式提供了方便的语法，来检查消息的内容是否有效。验证 DSL 命令采用 predicate 表达式作为其唯一参数：如果该 predicate 评估为 true，则路由会正常处理；如果该 predicate 评估为 false，则抛出 PredicateValidationException。

Java DSL 示例

以下路由使用正则表达式验证当前消息的正文：

```
from("jms:queue:incoming")
  .validate(body(String.class).regex("^\\w{10}\\,\\d{2}\\,\\w{24}$"))
  .to("bean:MyServiceBean.processLine");
```

您还可以验证消息 header>_<- the the example:

```
from("jms:queue:incoming")
  .validate(header("bar").isGreaterThan(100))
  .to("bean:MyServiceBean.processLine");
```

您可以将 validate 与 简单表达式语言 一起使用：

```
from("jms:queue:incoming")
  .validate(simple("${in.header.bar} == 100"))
  .to("bean:MyServiceBean.processLine");
```

XML DSL 示例

要在 XML DSL 中使用验证，推荐的方法是使用 简单 表达式语言：

```
<route>
  <from uri="jms:queue:incoming"/>
  <validate>
    <simple>${body} regex ^\\w{10}\\,\\d{2}\\,\\w{24}$</simple>
  </validate>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>

<bean id="myServiceBean" class="com.mycompany.MyServiceBean"/>
```

您还可以验证消息 header>_<- the the example:

```
<route>
  <from uri="jms:queue:incoming"/>
  <validate>
    <simple>${in.header.bar} == 100</simple>
  </validate>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>

<bean id="myServiceBean" class="com.mycompany.MyServiceBean"/>
```

第 11 章 消息传递端点

摘要

消息传递端点模式描述了可以在端点上配置的各种功能和服务质量。

11.1. 消息传递映射程序

概述

消息传递映射器 模式描述了如何将域对象映射到规范消息格式，其中消息格式被选为可中立的平台。选择的消息格式应适用于通过 第 6.5 节 “消息总线” 进行传输，其中消息总线是集成各种不同系统的后端，其中有些可能不是面向对象。

可能有很多不同的方法，但并非所有方法都满足消息传递映射程序的要求。例如，传输对象的一个明显方法是使用 对象序列化，它允许您使用非模糊编码（在 Java 中支持原生支持的）将对象写入数据流。但是，这不是 用于 messaging 映射器模式的适当方法，因为序列化格式只被 Java 应用程序理解。Java 对象序列化会在原始应用程序和消息传递系统中的其他应用程序间造成不匹配的情况。

消息传递映射程序的要求总结如下：

- 用于传输域对象的规范消息格式应适用于非面向对象的应用程序使用。
- 映射程序代码应该独立于域对象代码和消息传递基础架构实现。Apache Camel 通过提供 hook 来实现此要求，可用于将映射程序代码插入到路由中。
- 映射器可能需要找到处理某些面向对象的概念（如继承、对象引用和对象树）的有效方法。这些问题的复杂性因应用程序而异，但映射程序实施必须始终能够创建能由非面向对象的应用程序有效地处理的消息。

查找要映射的对象

您可以使用以下机制之一查找要映射的对象：

- 找到注册的 bean. baseDomain for singleton 对象和少量对象，您可以使用 CamelContext 注册表存储对 Bean 的引用。例如，如果一个 bean 实例使用 Spring XML 进行实例化，它将自

动输入到 registry 中，其中 bean 被其 id 属性的值来标识。

- 使用 JoSQL 语言选择对象。 需要的所有对象都在运行时都已被实例化，您可以使用 JoSQL 语言定位特定的对象（或对象）。例如，如果您有一个类，则 org.apache.camel.builder.sql.Person，其名称为 bean 属性，并且传入的消息具有 UserName 标头，则可以选择其 name 属性等于 UserName 标头的值：

```
import static org.apache.camel.builder.sql.SqlBuilder.sql;
import org.apache.camel.Expression;
...
Expression expression = sql("SELECT * FROM org.apache.camel.builder.sql.Person where
name = :UserName");
Object value = expression.evaluate(exchange);
```

语法 :HeaderName 用于在 JoSQL 表达式中替换标头值。

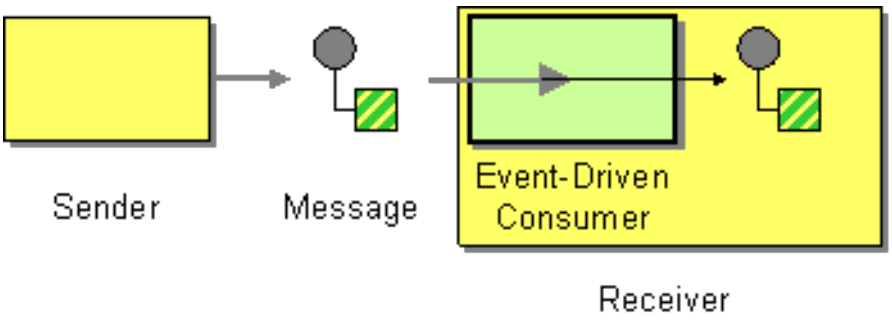
- 动态 admission-对更具扩展性的解决方案，可能需要从数据库读取对象数据。在某些情况下，现有面向对象的应用程序可能会已经提供一个 finder 对象，可以从数据库加载对象。在其他情况下，您可能必须编写一些自定义代码以从数据库提取对象，在这种情况下，JDBC 组件和 SQL 组件可能会很有用。

11.2. EVENT DRIVEN CONSUMER

概述

在 图 11.1 “event Driven Consumer Pattern” 中显示的 事件驱动的消费者 模式是在 Apache Camel 组件中实施消费者端点的模式，仅与需要在 Apache Camel 中开发自定义组件的编程者相关。现有组件已经具有使用者实施模式，难以连接它们。

图 11.1. event Driven Consumer Pattern



符合此模式的用户提供了在收到传入消息时由消息通道或传输层自动调用的事件方法。事件驱动的消费者模式的一种特征是，使用者端点本身不提供处理传入消息的任何线程。相反，底层传输或消息通道在调用公开事件方法时隐式提供处理器线程（消息处理期间的块）。

有关此实现模式的详情，请参阅 第 38.1.3 节 “消费者模式和线程” 和 第 41 章 消费者接口。

11.3. POLLING CONSUMER

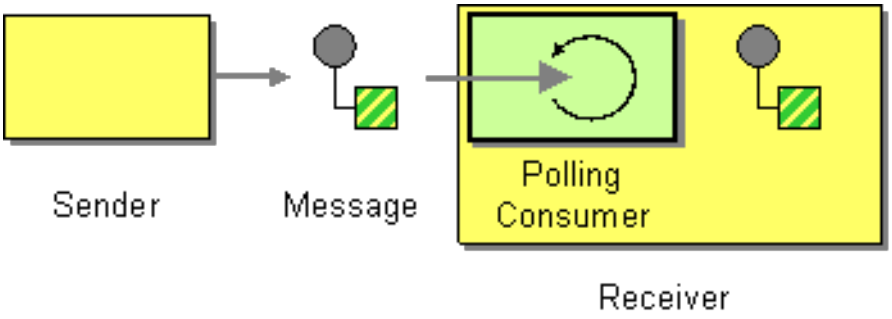
概述

在 图 11.2 “轮询消费者模式” 中显示的 轮询消费者 模式是在 Apache Camel 组件中实施消费者端点的模式，因此它仅与需要在 Apache Camel 中开发自定义组件的编程者相关。现有组件已经具有使用者实施模式，难以连接它们。

符合此模式的用户会公开轮询方法、receive（）、接收 （长超时） 和，并接收 NoWait（） 返回新的交换对象（若从被监控资源中可用）。轮询消费者实施必须提供自己的线程池来执行轮询。

有关此实现模式的详情，请参阅 第 38.1.3 节 “消费者模式和线程”、第 41 章 消费者接口 和 第 37.3 节 “使用 Consumer Template”。

图 11.2. 轮询消费者模式



调度的轮询消费者

许多 Apache Camel 消费者端点采用调度的轮询模式在路由开始时接收消息。也就是说，端点似乎实施事件驱动的消费者接口，但在调度的轮询内部用于监控为端点提供传入消息的资源。

有关如何实现此模式的详情，请参阅 第 41.2 节 “实现使用者接口”。

Quartz 组件

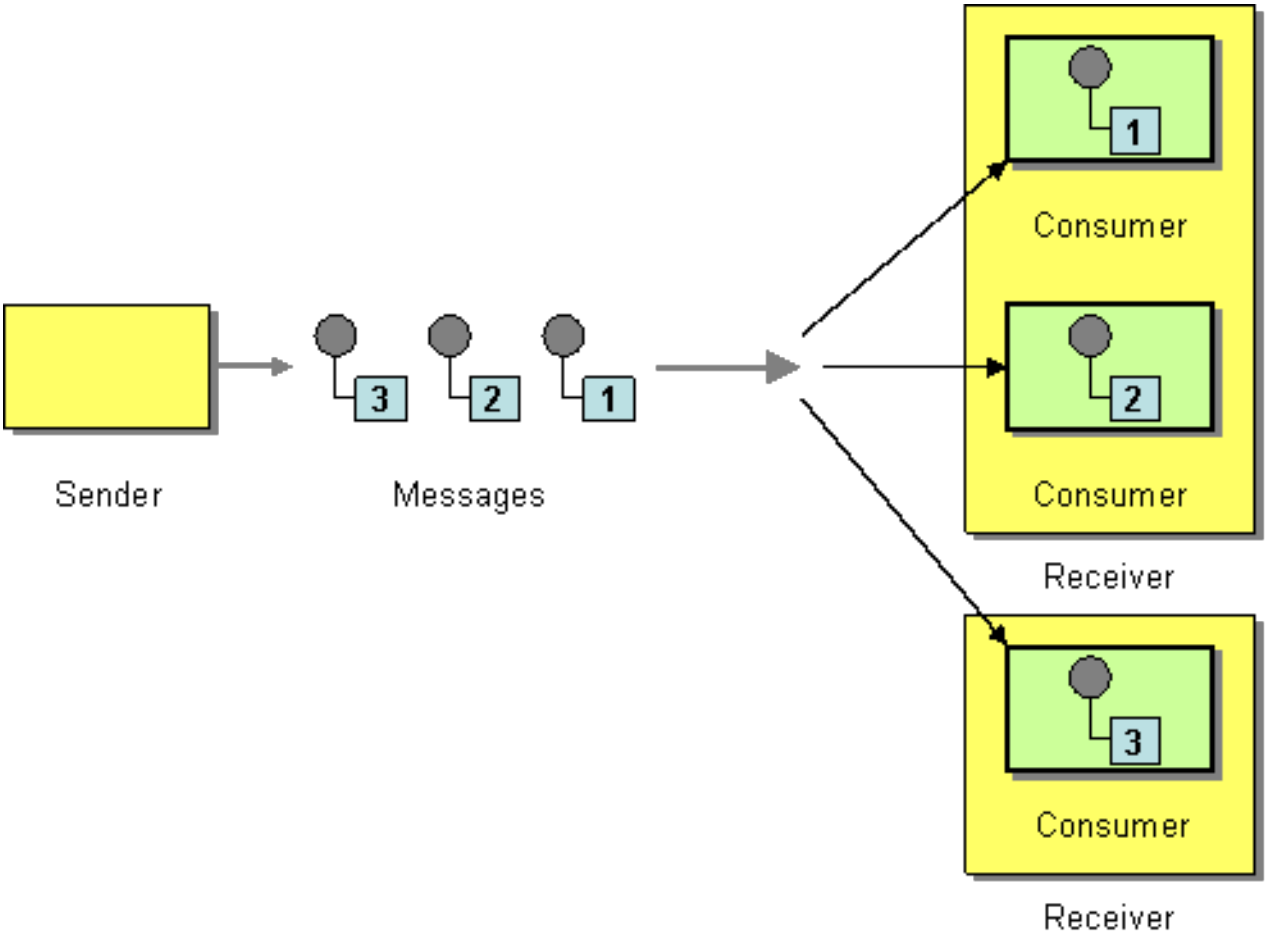
您可以使用 quartz 组件使用 Quartz 企业调度程序提供调度的消息交付。有关详细信息，请参阅 Apache Camel 组件参考指南 和 Quartz 组件 中的 Quartz。

11.4. 竞争消费者

概述

与 图 11.3 “竞争使用者模式” 中显示的 竞争消费者 模式可让多个消费者从同一队列拉取信息，保证 每个消息仅被使用一次。此模式可用于将串行消息处理替换为并发消息处理（降低响应延迟）。

图 11.3. 竞争使用者模式



以下组件展示竞争的使用者模式：

- 基于 JMS 的竞争消费者
- 基于竞争消费者的 SEDA

基于 JMS 的竞争消费者

常规 JMS 队列隐式保证每个消息一次只能被使用。因此，JMS 队列会自动支持竞争的使用者模式。例如，您可以定义三个来自 JMS 队列 HighVolumeQ 的消息的竞争消费者，如下所示：

```
from("jms:HighVolumeQ").to("cxf:bean:replica01");
from("jms:HighVolumeQ").to("cxf:bean:replica02");
from("jms:HighVolumeQ").to("cxf:bean:replica03");
```

如果 CXF（Web 服务）端点、replica01、Replica02 和 replica03，它并行处理来自 HighVolumeQ 队列的消息。

另外，您可以设置 JMS 查询选项，以便创建有竞争消费者的线程池。例如，以下路由创建了一个三个竞争的线程池，从指定队列获取消息：

```
from("jms:HighVolumeQ?concurrentConsumers=3").to("cxf:bean:replica01");
```

和 concurrentConsumers 选项也可以在 XML DSL 中指定，如下所示：

```
<route>
  <from uri="jms:HighVolumeQ?concurrentConsumers=3"/>
  <to uri="cxf:bean:replica01"/>
</route>
```

注意

JMS 主题不支持竞争的消费者模式。按照定义，JMS 主题旨在向不同的消费者发送相同消息的多个副本。因此，它与竞争的使用者模式不兼容。

基于竞争消费者的 SEDA

SEDA 组件的目的是通过将计算划分为多个阶段来简化并发处理。SEDA 端点实际上封装了内存块队列（由 java.util.concurrent.BlockingQueue 实施）。因此，您可以使用 SEDA 端点将路由划分为多个阶段，每个阶段都可能会使用多个线程。例如，您可以定义由两个阶段组成的 SEDA 路由，如下所示：

```
// Stage 1: Read messages from file system.
from("file://var/messages").to("seda:fanout");

// Stage 2: Perform concurrent processing (3 threads).
from("seda:fanout").to("cxf:bean:replica01");
from("seda:fanout").to("cxf:bean:replica02");
from("seda:fanout").to("cxf:bean:replica03");
```

其中，第一个阶段包含一个线程，它消耗来自文件端点 file://var/messages 的消息，并将它们路由到 SEDA 端点，seda:fanout。第二个阶段包含三个线程：路由交换到 cxf:bean:replica01 的线程，路由交

换到 cxf:bean:replica02 的线程，以及路由交换到 cxf:bean:replica03 的线程。这三个线程与来自 SEDA 端点的实例相比，该端点使用阻止队列实施。由于块队列使用锁定来防止多个线程一次访问队列，所以您可以保证每个交换实例一次只能使用一次。

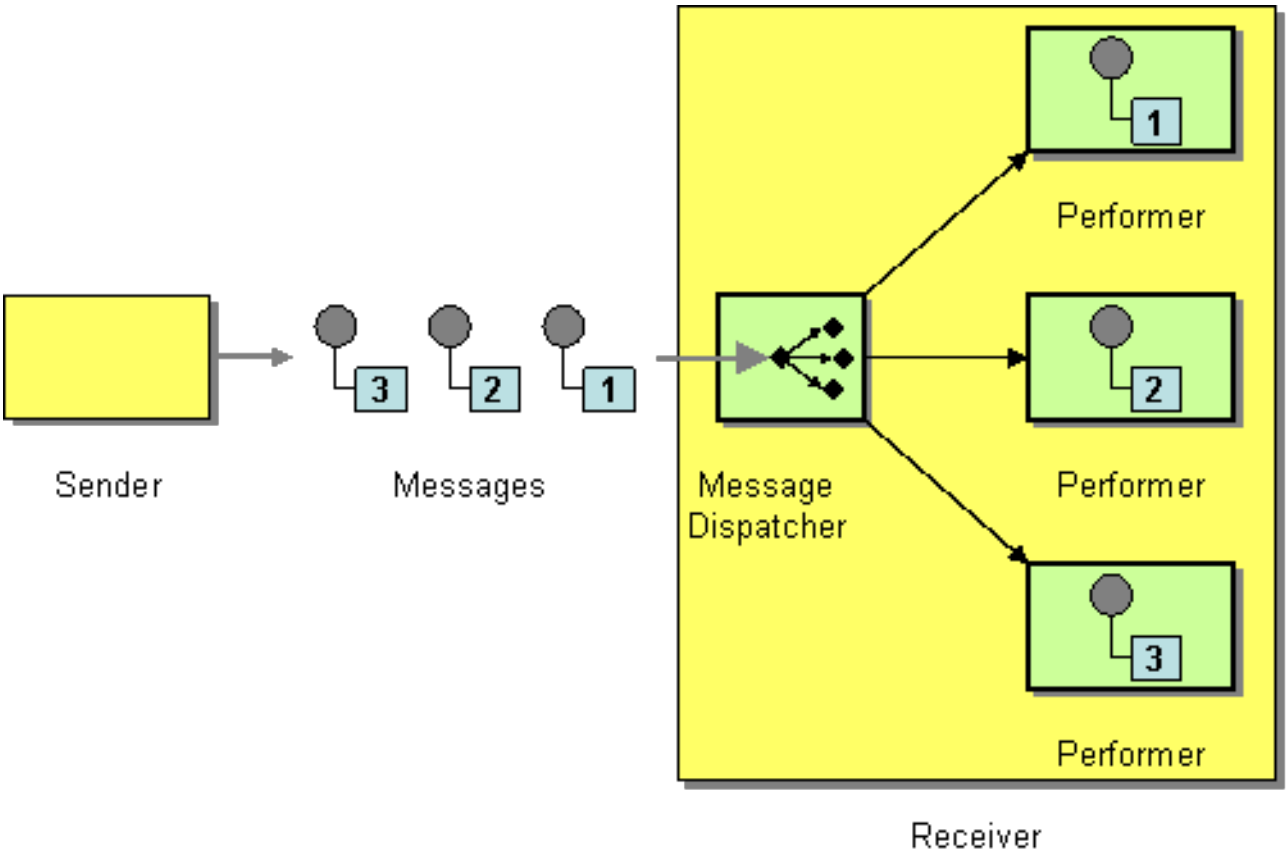
有关 SEDA 端点和 thread（） 创建的线程池之间的区别，请参阅 Apache Camel 组件参考指南 中的 SEDA 组件。

11.5. MESSAGE DISPATCHER

概述

消息分配程序 模式（如 图 11.4 “消息 Dispatcher Pattern” ）用于消耗来自某个频道的信息，然后在本地分发它们以执行它，该模式负责处理信息。在 Apache Camel 应用程序中，执行者通常由进程中的端点表示，这些端点用于将消息传送到路由的另一个部分。

图 11.4. 消息 Dispatcher Pattern



您可以使用以下方法之一在 Apache Camel 中实施消息分配模式：

- *JMS 选择器*

- ActiveMQ 中的 JMS 选择器
- 基于内容的路由器

JMS 选择器

如果您的应用程序消耗来自 JMS 队列的消息，您可以使用 JMS 选择器 实施消息分配模式。JMS 选择器是涉及 JMS 标头和 JMS 属性的 predicate 表达式。如果选择器评估为 true，则允许 JMS 消息访问使用者，如果选择器评估为 false，则 JMS 消息会被阻止。在很多方面，JMS 选择器类似于 第 8.2 节 “Message Filter”，但在 JMS 提供程序中实施过滤具有额外的优点。这意味着，JMS 选择器可以在将消息传输到 Apache Camel 应用程序前阻止消息。这带来了显著的效率优势。

在 Apache Camel 中，您可以通过在 JMS 端点上设置 选择器 查询选项，在消费者端点上定义 JMS 选择器。例如：

```
from("jms:dispatcher?selector=CountryCode='US'").to("cxf:bean:replica01");
from("jms:dispatcher?selector=CountryCode='IE'").to("cxf:bean:replica02");
from("jms:dispatcher?selector=CountryCode='DE'").to("cxf:bean:replica03");
```

在选择器字符串中显示的 predicates 是基于 SQL92 条件表达式语法的子集（用于完整详情，请参阅 JMS 规格）。选择器字符串中显示的标识符可以指代 JMS 标头或 JMS 属性。例如，在前面的路由中，发送者会设置名为 CountryCode 的 JMS 属性。

如果要向 Apache Camel 应用中的消息添加 JMS 属性，您可以通过设置消息标头或 Out 消息来完成此操作。在读取或写入 JMS 端点时，Apache Camel 会将 JMS 标头和 JMS 属性映射到其原生邮件标题。

从技术上讲，选择器字符串必须根据 application/x-www-form-url 编码的 MIME 格式进行 URL 编码（请参阅 HTML 规格）。在实践中，&(ampersand)字符可能会造成困难，因为它用于限定 URI 中的每个查询选项。对于可能需要嵌入 & amp; 字符的复杂选择器字符串，您可以使用 java.net.URLEncoder 程序类对字符串进行编码。例如：

```
from("jms:dispatcher?selector=" + java.net.URLEncoder.encode("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

必须使用 UTF-8 编码的位置。

ActiveMQ 中的 JMS 选择器

您还可以在 ActiveMQ 端点上定义 JMS 选择器。例如：

```
from("activemq:dispatcher?selector=CountryCode='US'").to("cxf:bean:replica01");
from("activemq:dispatcher?selector=CountryCode='IE'").to("cxf:bean:replica02");
from("activemq:dispatcher?selector=CountryCode='DE'").to("cxf:bean:replica03");
```

如需了解更多详细信息，请参阅 ActiveMQ: JMS Selector 和 ActiveMQ Message Properties。

基于内容的路由器

基于内容的路由器模式和消息分配模式之间的基本区别在于，基于内容的路由器将消息分配给物理独立的目的地（远程端点），以及分配本地的、在同一进程空间中的消息分配消息。在 Apache Camel 中，这两种模式之间的区别由目标端点决定。相同的路由器逻辑同时用于实施基于内容的路由器和消息分配程序。当目标端点为远程端点时，路由会定义一个基于内容的路由器。当目标端点处于进程中时，路由定义了一个消息分配程序。

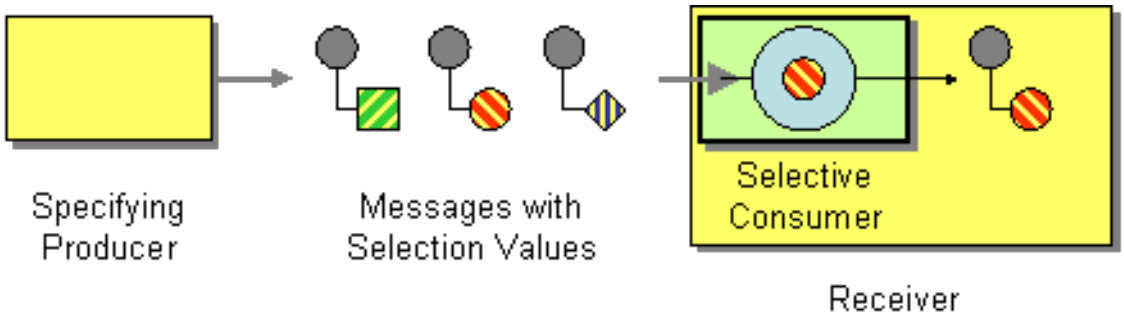
有关如何使用内容的路由器模式的详情和示例请查看 第 8.1 节 “基于内容的路由器”。

11.6. 选择IVE CONSUMER

概述

图 11.5 “选择ive Consumer Pattern” 中显示的选择 消费者 模式描述了向传入信息应用过滤器的使用者，以便仅处理满足特定选择条件的消息。

图 11.5. 选择ive Consumer Pattern



您可以使用以下方法之一在 Apache Camel 中实施选择消费者模式：

- JMS 选择器

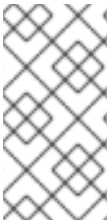
- ActiveMQ 中的 JMS 选择器
- Message filter

JMS 选择器

JMS 选择器是涉及 JMS 标头和 JMS 属性的 predicate 表达式。如果选择器评估为 true，则允许 JMS 消息访问使用者，如果选择器评估为 false，则 JMS 消息会被阻止。例如，若要消耗来自队列的消息、选择性，并且仅选择那些国家代码属性等于 US 的消息，您可以使用以下 Java DSL 路由：

```
from("jms:selective?selector=" + java.net.URLEncoder.encode("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

如果选择器字符串 America Americacode='US' 是 URL 编码（使用 UTF-8 字符），以避免在解析查询选项时出现问题。本例假定 JMS 属性 国家代码 是由发件人设置。有关 JMS 选择器的详情，请参阅 “JMS 选择器”一节。



注意

如果选择器应用到 JMS 队列，则不会选择的消息保留在队列中，并且有可能可供附加到同一队列的其他用户使用。

ActiveMQ 中的 JMS 选择器

您还可以在 ActiveMQ 端点上定义 JMS 选择器。例如：

```
from("acivemq:selective?selector=" + java.net.URLEncoder.encode("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

如需了解更多详细信息，请参阅 ActiveMQ: JMS Selector 和 ActiveMQ Message Properties。

Message filter

如果无法在消费者端点上设置选择器，您可以将过滤器处理器插入到您的路由中。例如，您可以定义一个选择使用者，它仅使用 Java DSL 处理带有美国国家代码的消息，如下所示：

```
from("seda:a").filter(header("CountryCode").isEqualTo("US")).process(myProcessor);
```

同一路由可以使用 XML 配置来定义，如下所示：

```
<camelContext id="buildCustomProcessorWithFilter"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$CountryCode = 'US'</xpath>
      <process ref="#myProcessor"/>
    </filter>
  </route>
</camelContext>
```

有关 Apache Camel 过滤器处理器的详情请参考 第 8.2 节 “Message Filter”。

警告

请注意，使用消息过滤器选择来自 JMS 队列 的消息。使用过滤器处理器时，阻止的消息会被简单地丢弃。因此，如果消息来自一个队列（这允许每个消息被使用一次，则每条消息就可以被使用一次： 第 11.4 节 “竞争消费者”），则不会处理阻止的消息。这可能不是您想要的行为。

11.7. DURABLE SUBSCRIBER

概述

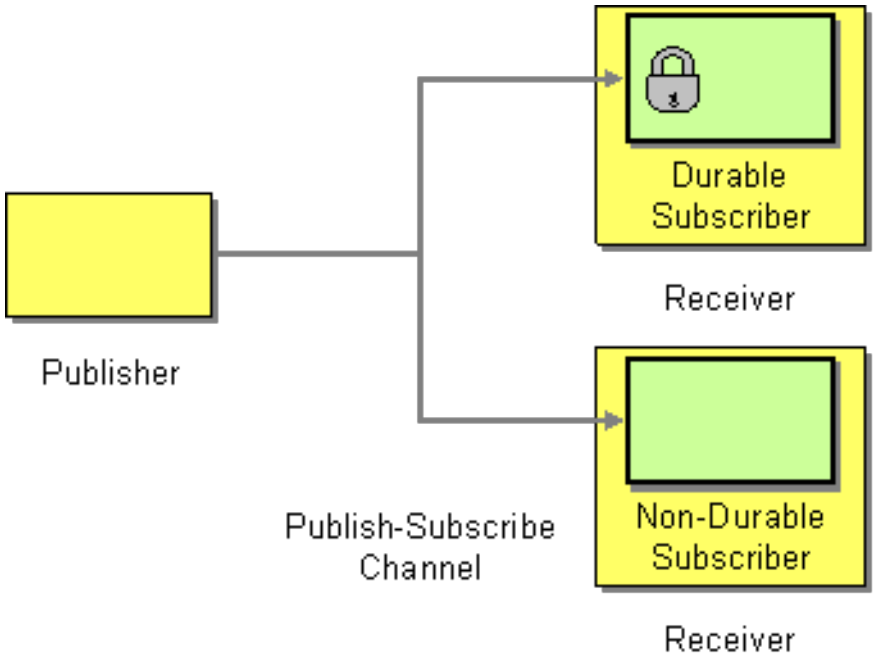
一个 可靠的订阅者 （如 图 11.6 “durable Subscriber Pattern” 所示），它希望接收通过特定 第 6.2 节 “publish-Subscribe Channel” 频道发送的所有消息，包括消费者从消息传递系统断开连接时发送的消息。这要求消息传递系统将消息存储到断开连接的消费者中。对于消费者而言，还必须是一种机制，指明它要建立持久的订阅。通常，发布订阅频道（或主题）可以同时有持久的订阅者，其行为如下：

- 不 可持续的订阅者 Inktank - theCan 有两个状态：已连接和断开连接的。 虽然一个不可避免的订阅者连接到一个主题，但它会实时接收所有主题的消息。但是，在订阅者断开连接时，不可避免的订阅者不会接收发送到主题的消息。

- durable subscriber abrt-abrtCan 有两个状态： connected 和 inactive。inactive 状态意味着 durable 订阅者与主题断开连接，但希望接收到达 interim 的消息。当 durable 订阅者重新

连接到该主题时，它会收到发送的所有消息的回放。

图 11.6. durable Subscriber Pattern



JMS durable subscriber

JMS 组件实施 durable 订阅者模式。要在 JMS 端点上设置持久化订阅，您必须指定 客户端 ID，它用于标识此特定连接，以及确定 durable 订阅名称。例如，以下路由设置了一个持久订阅到 JMS 主题(news)，其客户端 ID 为 conn01，一个持久的订阅名称为 John.Doe ：

```
from("jms:topic:news?clientId=conn01&durableSubscriptionName=John.Doe").
    to("cxf:bean:newsprocessor");
```

您还可以使用 ActiveMQ 端点设置持久订阅：

```
from("activemq:topic:news?clientId=conn01&durableSubscriptionName=John.Doe").
    to("cxf:bean:newsprocessor");
```

如果要同时处理传入的信息，您可以使用 SEDA 端点将路由填充到多个并行片段中，如下所示：

```
from("jms:topic:news?clientId=conn01&durableSubscriptionName=John.Doe").
    to("seda:fanout");

from("seda:fanout").to("cxf:bean:newsproc01");
from("seda:fanout").to("cxf:bean:newsproc02");
from("seda:fanout").to("cxf:bean:newsproc03");
```

每个消息都仅处理一次，因为 SEDA 组件支持 有竞争的使用者 模式。

其他示例

另一个选择是将 第 11.5 节 “Message Dispatcher” 或 第 8.1 节 “基于内容的路由器” 与 文件 或 JPA 组件合并，用于临时订阅者，类似非持久性的 SEDA。

以下是创建可靠订阅者到 JMS 主题的简单示例

使用 Fluent Builders

```
from("direct:start").to("activemq:topic:foo");

from("activemq:topic:foo?clientId=1&durableSubscriptionName=bar1").to("mock:result1");

from("activemq:topic:foo?clientId=2&durableSubscriptionName=bar2").to("mock:result2");
```

使用 Spring XML 扩展

```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:topic:foo"/>
</route>

<route>
  <from uri="activemq:topic:foo?clientId=1&durableSubscriptionName=bar1"/>
  <to uri="mock:result1"/>
</route>

<route>
  <from uri="activemq:topic:foo?clientId=2&durableSubscriptionName=bar2"/>
  <to uri="mock:result2"/>
</route>
```

以下是 JMS durable 订阅者的另一个示例，但这一次使用 虚拟主题 （由 AMQ over durable 订阅推荐）

使用 Fluent Builders

```
from("direct:start").to("activemq:topic:VirtualTopic.foo");
```

```
from("activemq:queue:Consumer.1.VirtualTopic.foo").to("mock:result1");

from("activemq:queue:Consumer.2.VirtualTopic.foo").to("mock:result2");
```

使用 Spring XML 扩展

```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:topic:VirtualTopic.foo"/>
</route>

<route>
  <from uri="activemq:queue:Consumer.1.VirtualTopic.foo"/>
  <to uri="mock:result1"/>
</route>

<route>
  <from uri="activemq:queue:Consumer.2.VirtualTopic.foo"/>
  <to uri="mock:result2"/>
</route>
```

11.8. 幂等的消费者

概述

幂等的使用者 模式用于过滤重复的消息。例如，在消息传递系统和消费者端点之间进行连接的情况会因为系统中的一些故障而丢失。如果消息传递系统位于传输消息的中间，这可能并不明确，消费者是否收到了最后一条消息。为提高交付可靠性，消息传递系统可能决定在重新建立连接后尽快进行此类消息。不幸的是，这要求消费者可能会收到重复消息的风险，在某些情况下，重复消息的结果可能会无法产生不可取的后果（例如，从您的帐户中取消了两次的资金总和两次）。在这种情况下，可以使用幂等的消费者来从消息流中意外重复。

Camel 提供以下 Idempotent Consumer 实现：

- MemoryIdempotentRepository
- KafkaIdempotentRepository
- File

- *Hazelcast*
- *SQL*
- *JPA*

带有内存缓存的幂等消费者

在 Apache Camel 中，幂等的使用者模式由 idempotentConsumer（） 处理器实施，它采用两个参数：

- *messageIdExpression abrt-abrt An 表达式为当前消息返回消息 ID 字符串。*
- *messageIdRepository >_< A reference to a message ID repository，它存储收到的所有消息的 ID。*

当每条消息出现时，幂等使用者处理器会在存储库中查找当前消息 ID，以查看此消息之前是否已看到。如果 yes，则会丢弃消息；如果没有，则消息被允许通过，并将其 ID 添加到存储库中。

例 11.1 “使用内存缓存过滤 Duplicate 消息” 中显示的代码使用 TransactionID 标头来过滤掉重复。

例 11.1. 使用内存缓存过滤 Duplicate 消息

```
import static
org.apache.camel.processor.idempotent.MemoryMessageIdRepository.memoryMessageIdRepository;
...
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a")
          .idempotentConsumer(
            header("TransactionID"),
            memoryMessageIdRepository(200)
          ).to("seda:b");
    }
};
```

其中调用 memoryMessageIdRepository(200) 会创建一个内存缓存，它最多可容纳 200 消息 ID。

您还可以使用 XML 配置来定义幂等消费者。例如，您可以在 XML 中定义前面的路由，如下所示：

```
<camelContext id="buildIdempotentConsumer" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <idempotentConsumer messageIdRepositoryRef="MsgIDRepos">
      <simple>header.TransactionID</simple>
      <to uri="seda:b"/>
    </idempotentConsumer>
  </route>
</camelContext>

<bean id="MsgIDRepos"
class="org.apache.camel.processor.idempotent.MemoryMessageIdRepository">
    <!-- Specify the in-memory cache size. -->
    <constructor-arg type="int" value="200"/>
</bean>
```

注意

从 Camel 2.17 中，Idempotent Repository 支持可选的序列化标头。

使用 JPA 存储库的幂等消费者

内存缓存的缺陷影响内存不足，且不能在集群环境中工作。要克服这些缺点，您可以使用基于 Java Persistent API(JPA)的存储库。JPA 消息 ID 存储库使用面向对象的数据库来存储消息 ID。例如，您可以定义将 JPA 存储库用于幂等消费者的路由，如下所示：

```
import org.springframework.orm.jpa.JpaTemplate;

import org.apache.camel.spring.SpringRouteBuilder;
import static
org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository.jpaMessageIdRepository;
...
RouteBuilder builder = new SpringRouteBuilder() {
    public void configure() {
        from("seda:a").idempotentConsumer(
          header("TransactionID"),
          jpaMessageIdRepository(bean(JpaTemplate.class), "myProcessorName")
        ).to("seda:b");
    }
};
```

JPA 消息 ID 存储库使用两个参数初始化：

- JpaTemplate instance>_<-jaxbProvides the JPA 数据库的句柄。
- 处理器 name 3.10.0--jaxbIdent 表示当前的幂等消费者处理器。

SpringRouteBuilder.bean（） 方法是引用 Spring XML 文件中定义的 bean 的快捷方式。JpaTemplate bean 为底层 JPA 数据库提供了句柄。有关如何配置此 bean 的详情，请查看 JPA 文档。

有关设置 JPA 存储库的详情，请参阅 JPA 组件 文档、Spring JPA 文档和 Camel JPA 单元测试 中的示例代码。

Spring XML 示例

以下示例使用 myMessageId 标头过滤掉重复：

```
<!-- repository for the idempotent consumer -->
<bean id="myRepo"
class="org.apache.camel.processor.idempotent.MemoryIdempotentRepository"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <idempotentConsumer messageIdRepositoryRef="myRepo">
      <!-- use the messageId header as key for identifying duplicate messages -->
      <header>messageId</header>
      <!-- if not a duplicate send it to this mock endpoint -->
      <to uri="mock:result"/>
    </idempotentConsumer>
  </route>
</camelContext>
```

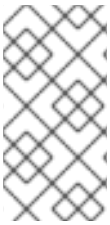
具有 JDBC 存储库的幂等消费者

JDBC 存储库也支持将消息 ID 存储在幂等消费者模式中。JDBC 存储库的实现由 SQL 组件提供。因此，如果您使用 Maven 构建系统，请添加对 camel-sql 工件的依赖项。

您可以使用 Spring persistence API 中的 SingleConnectionDataSource wrapper 类来实例化与 SQL 数据库的连接。例如，若要将 JDBC 连接实例化到 HyperSQL 数据库实例，您可以定义以下 JDBC

数据源：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
   <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
   <property name="url" value="jdbc:hsqldb:mem:camel_jdbc"/>
   <property name="username" value="sa"/>
   <property name="password" value=""/>
</bean>
```



注意

前面的 JDBC 数据源使用 HyperSQL mem 协议，该协议会创建一个仅内存的数据库实例。这是尚未持久的 HyperSQL 数据库的实施。

使用前面的数据源，您可以定义一个使用 JDBC 消息 ID 存储库的幂等消费者模式，如下所示：

```
<bean id="messageIdRepository"
class="org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository">
 <constructor-arg ref="dataSource" />
 <constructor-arg value="myProcessorName" />
</bean>

<camel:camelContext>
 <camel:errorHandler id="deadLetterChannel" type="DeadLetterChannel"
deadLetterUri="mock:error">
  <camel:redeliveryPolicy maximumRedeliveries="0" maximumRedeliveryDelay="0"
logStackTrace="false" />
 </camel:errorHandler>

 <camel:route id="JdbcMessageIdRepositoryTest" errorHandlerRef="deadLetterChannel">
  <camel:from uri="direct:start" />
  <camel:idempotentConsumer messageIdRepositoryRef="messageIdRepository">
   <camel:header>messageId</camel:header>
   <camel:to uri="mock:result" />
  </camel:idempotentConsumer>
 </camel:route>
 </camel:camelContext>
```

如何在路由中处理重复的信息

可作为 Camel 2.8 提供。

现在，您可以将 skipDuplicate 选项设置为 false，它会指示幂等的使用者也用于路由重复的消息。但是，重复的消息已通过将 "交换"一节 中的属性设为 true 来被标记为重复。我们可以使用 第 8.1 节 "基于内容的路由器" 或 第 8.2 节 "Message Filter" 检测此消息并处理重复的信息来利用这个事实。

例如，在以下示例中，我们使用 第 8.2 节 “Message Filter” 发送消息到重复的端点，然后停止继续路由该消息。

```
from("direct:start")
    // instruct idempotent consumer to not skip duplicates as we will filter then our self
    .idempotentConsumer(header("messageId")).messageIdRepository(repo).skipDuplicate(false)
    .filter(property(Exchange.DUPLICATE_MESSAGE).isEqualTo(true))
        // filter out duplicate messages by sending them to someplace else and then stop
        .to("mock:duplicate")
        .stop()
    .end()
    // and here we process only new messages (no duplicates)
    .to("mock:result");
```

XML DSL 中的示例如下：

```
<!-- idempotent repository, just use a memory based for testing -->
<bean id="myRepo"
class="org.apache.camel.processor.idempotent.MemoryIdempotentRepository"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <!-- we do not want to skip any duplicate messages -->
        <idempotentConsumer messageIdRepositoryRef="myRepo" skipDuplicate="false">
            <!-- use the messageId header as key for identifying duplicate messages -->
            <header>messageId</header>
            <!-- we will to handle duplicate messages using a filter -->
            <filter>
                <!-- the filter will only react on duplicate messages, if this property is set on the Exchange --
>
                <property>CamelDuplicateMessage</property>
                <!-- and send the message to this mock, due its part of an unit test -->
                <!-- but you can of course do anything as its part of the route -->
                <to uri="mock:duplicate"/>
                <!-- and then stop -->
                <stop/>
            </filter>
            <!-- here we route only new messages -->
            <to uri="mock:result"/>
        </idempotentConsumer>
    </route>
</camelContext>
```

如何使用数据网格处理集群环境中的重复消息

如果您在集群环境中运行 Camel，则需要一个内存幂等性存储库无法工作（请参阅上面的操作）。您可以根据 Hazelcast 数据网格设置中央数据库或使用幂等的消费者实施。Hazelcast 通过多播查找节点（默认 - 为 tcp-ip 配置 Hazelcast），并创建基于映射的存储库：

```
HazelcastIdempotentRepository idempotentRepo = new HazelcastIdempotentRepository("myrepo");

from("direct:in").idempotentConsumer(header("messageId"), idempotentRepo).to("mock:out");
```

您必须定义存储库应当容纳每个消息 id 的时长（默认为从不删除）。为了避免内存不足，您应该基于 Hazelcast 配置创建 驱除策略。如需了解更多信息，请参阅 Hazelcast。

请参阅此链接：http://camel.apache.org/hazelcast-idempotent-repository-tutorial.html[Idempotent 软件仓库

教程] 了解更多有关如何使用 Apache Karaf 在两个集群节点上设置此类幂等存储库。

选项

Idempotent Consumer 有以下选项：

选项	默认值	描述
eager	**true**	Camel 2.0： Eager 控制 Camel 是否在处理交换之前或之后将消息添加到存储库中。如果在之前启用，Camel 将能够检测到重复的消息，即使消息当前正在进行时也是如此。通过禁用 Camel，只有在成功处理消息时才会检测到重复的。
messageIdRepositoryRef	**null**	对 **IdempotentRepository** 的引用，以便在 registry 中查询。使用 XML DSL 时，这个选项是必须的。
skipDuplicate	**true**	Camel 2.8： 设置是否跳过重复消息。如果设为 **false**，则消息将继续。但是，“交换”一节 已通过将 **Exchange.DUPLICATE_MESSAG** Exchange 属性设置为 **Boolean.TRUE** 值来被标记为一个重复的。

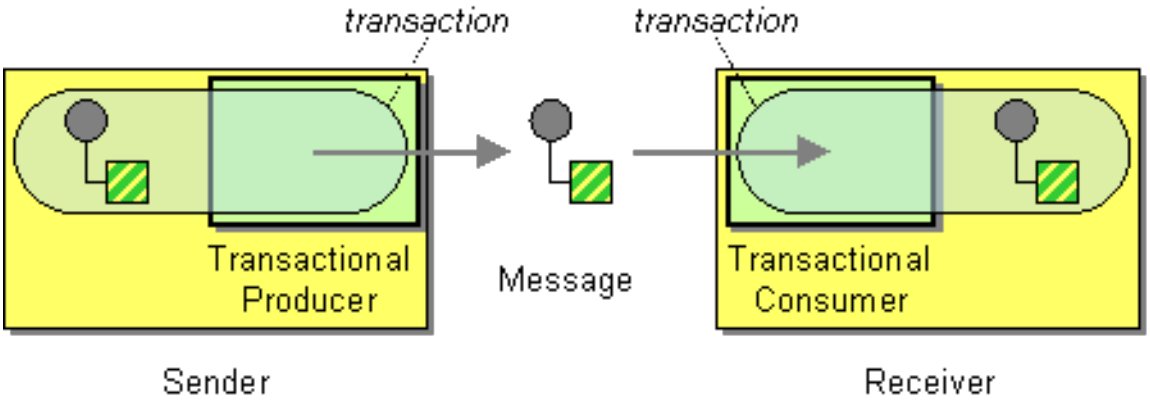
completionEager	**false**	Camel 2.16： 完成交换时是否需要完成 Idempotent eager。 如果您设置了 **completeEager** 选项 true，那么当交换到达幂等消费者块的末尾时，Idempotent Consumer 会触发其完成。但是，如果交换在最终块后仍然可以被路由，那么它不会影响幂等消费者的状态。 如果您设置了 **completeEager** 选项 false，则 Idempotent Consumer 会在交换完成后触发其完成，并被路由。但是，如果交换在块结束后仍继续路由，那么它也会影响幂等消费者的状态。例如，由于交换失败导致异常，因此幂等消费者的状态将是回滚。

11.9. 事务客户端

概述

事务性客户端 模式（如 图 11.7 “事务性客户端模式” 所示）指的是可以参与事务的消息传递端点。Apache Camel 支持使用 Spring 事务管理 进行事务。

图 11.7. 事务性客户端模式



事务型端点

并非所有 Apache Camel 端点都支持事务。这样做称为 面向事务的端点 （或 TOE）。例如，JMS 组件和 ActiveMQ 组件都支持事务。

要在组件上启用事务，必须在将组件添加到 CamelContext 之前执行适当的初始化。这需要编写代码

来明确初始化您的事务组件。

参考信息

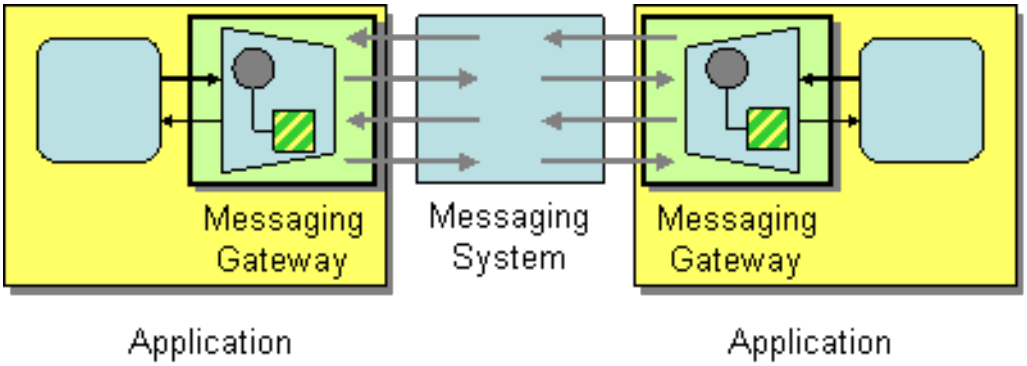
在 Apache Camel 中配置事务的详情超出了本指南的范围。有关如何使用事务的完整详情，请参阅 Apache Camel 事务指南。

11.10. 消息传递网关

概述

消息网关 模式（在 图 11.8 “消息传递网关模式” 所示）描述了与消息传递系统集成的方法，其中消息传递系统的 API 在应用程序级别被隐藏。更常见的例子之一就是，您希望将同步方法调用转换为 request/reply 消息交换，而无需了解它。

图 11.8. 消息传递网关模式



以下 Apache Camel 组件提供以下与消息传递系统的集成：

- CXF
- bean 组件

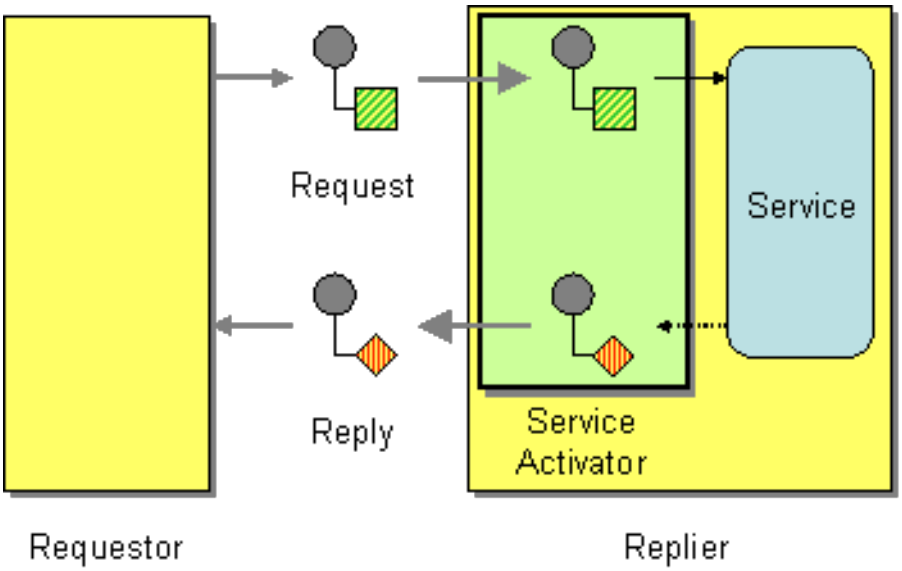
11.11. SERVICE ACTIVATOR

概述

service activator 模式在 图 11.9 “Service Activator Pattern” 中显示的是，描述在响应传入请求信息时调用该服务的操作的情况。服务激活器标识要调用哪些操作并提取要用作操作参数的数据。最后，服

务激活程序利用从消息中提取的数据来调用操作。操作调用可以是单向（请求）或双向(request/reply)。

图 11.9. Service Activator Pattern



在很多方面，服务激活程序类似于传统的远程过程调用(RPC)，其中操作调用被编码为消息。主要区别在于，服务激活者需要更灵活。RPC 框架对请求和回复消息编码（例如，Web 服务操作编码为 SOAP 消息），而服务激活器通常需要在消息传递系统和服务操作之间提供映射。

Bean 集成

Apache Camel 为支持服务提供的主要机制是 集成。bean 集成 提供了一个通用框架，用于将传入消息映射到 Java 对象上方法调用。例如，Java fluent DSL 提供了处理器 bean（） 和 beanRef（），您可以插入到路由中以调用已注册 Java Bean 上的方法。消息数据到 Java 方法参数的详细映射由 bean 绑定 决定，可以通过向 bean 类添加注解来实现。

例如，请考虑以下路由，该路由调用 Java 方法 BankBean.getUserAccBalance（），以服务请求传入 JMS/ActiveMQ 队列：

```
from("activemq:BalanceQueries")
  .setProperty("userid", xpath("/Account/BalanceQuery/UserID").stringResult())
  .beanRef("bankBean", "getUserAccBalance")
  .to("velocity:file:src/scripts/acc_balance.vm")
  .to("activemq:BalanceResults");
```

从 ActiveMQ 端点 activemq:BalanceQueries 中拉取的消息具有一个简单的 XML 格式，可提供银行账户的用户 ID。例如：

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

```
  <BalanceQuery>
    <UserID>James.Strachan</UserID>
  </BalanceQuery>
</Account>
```

路由中的第一个处理器 设置Property（），从 In 消息中提取用户 ID，并将其存储在 userid Exchange 属性中。这最好将其存储在标头中，因为调用 bean 后无法使用 In 标头。

服务激活步骤由 beanRef（） 处理器执行，它会将传入的消息绑定到由 银行 Bean 识别的 Java 对象上的 getUserAccBalance（） 方法。以下代码显示了 BankBean 类实施示例：

```
package tutorial;

import org.apache.camel.language.XPath;

public class BankBean {
    public int getUserAccBalance(@XPath("/Account/BalanceQuery/UserID") String user) {
        if (user.equals("James.Strachan")) {
            return 1200;
        }
        else {
            return 0;
        }
      }
}
```

如果 @XPath 注释启用消息数据到方法参数的绑定，它会将 UserID XML 元素的内容注入到 用户 方法参数中。在完成调用时，返回值将插入到 Out 消息的正文中，然后复制到路由中下一步的 In 消息中。要让 bean 可以被 beanRef（） 处理器访问，您必须在 Spring XML 中实例化实例。例如，您可以将以下行添加到 META-INF/spring/camel-context.xml 配置文件以实例化 bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <bean id="bankBean" class="tutorial.BankBean"/>
</beans>
```

在 registry 中，bean ID、tailBean 标识了此 bean 实例。

bean 调用的输出注入到 Velocity 模板中，以生成正确格式的结果消息。Velocity 端点 velocity:file:src/scripts/acc_balance.vm 用于指定 velocity 脚本的位置：

```
<?xml version='1.0' encoding='UTF-8'?>
<Account>
  <BalanceResult>
```

```
    <UserID>${exchange.getProperty("userid")}</UserID>
    <Balance>${body}</Balance>
  </BalanceResult>
</Account>
```

Exchange 实例作为 Velocity 变量 交换 提供，它可让您使用 ${exchange.getProperty(" userid ")} 获取 userid Exchange 属性。当前 In 消息的正文 ${body} 中包含 getUserAccBalance（） 方法调用的结果。

第 12 章 系统管理

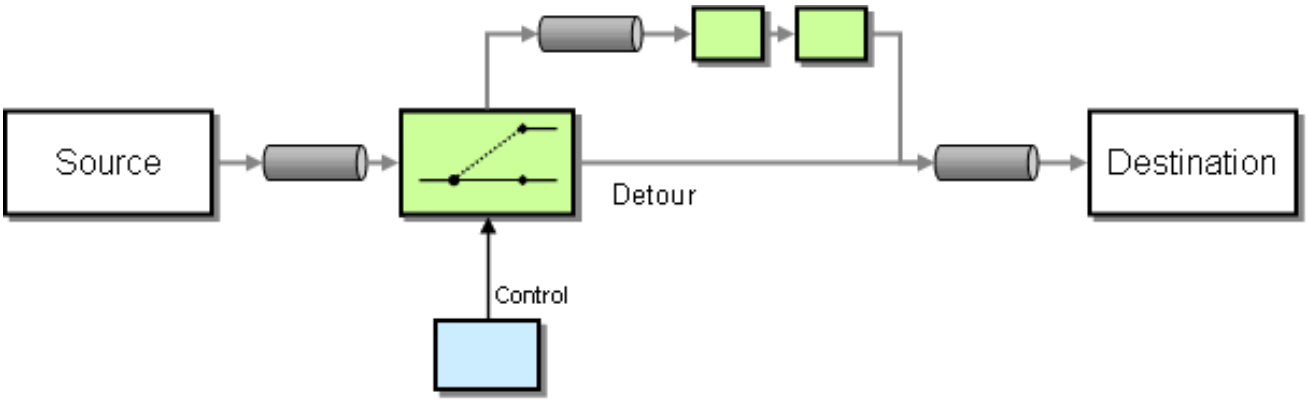
摘要

系统管理模式描述了如何监控、测试和管理消息传递系统。

12.1. DETOUR

detour

如果满足控制条件，可以通过 第 3 章 企业级集成模式简介 取消使用额外的步骤来发送信息。http://www.enterpriseintegrationpatterns.com/Detour.html它可用于在需要时打开额外的验证、测试、调试代码。



示例

在本例中，我们本质上有一个 来自("direct:start").to("mock:result") 的路由，其有条件 指示到路由中间的模拟：detour 端点。

```
from("direct:start").choice()
    .when().method("controlBean", "isDetour").to("mock:detour").end()
    .to("mock:result");
```

使用 Spring XML 扩展

```
<route>
  <from uri="direct:start"/>
    <choice>
```

```
      <when>
        <method bean="controlBean" method="isDetour"/>
  <to uri="mock:detour"/>
      </when>
    </choice>
    <to uri="mock:result"/>
  </split>
</route>
```

债务部是否由 ControlBean 决定。因此，当消息上路由到 模拟：detour， 然后 模拟结果。当 detour 被关闭时，消息会被路由到 模拟：result。

有关详情请查看示例源：

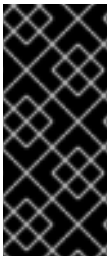
camel-core/src/test/java/org/apache/camel/processor/DetourTest.java

12.2. LOGEIP

概述

Apache Camel 提供多种在路由中执行日志记录的方法：

- *使用 log DSL 命令。*
- *使用 Log 组件，它可以记录消息内容。*
- *使用 Tracer，跟踪消息流。*
- *使用 处理器 或 Bean 端点在 Java 中执行日志记录。*



日志 DSL 命令和日志组件之间的区别

日志 DSL 更加简洁，用于记录人类日志，如 Starting to do ...。它只能根据 Simple 语言记录消息。相反，Log 组件是一个功能齐全的日志组件。Log 组件能够记录消息本身，并且有许多 URI 选项来控制日志记录。

Java DSL 示例

自 Apache Camel 2.2 起，您可以使用 log DSL 命令，在运行时使用 Simple 表达式语言构建日志消息。例如，您可以在路由中创建日志消息，如下所示：

```
from("direct:start").log("Processing ${id}").to("bean:foo");
```

此路由在运行时构造字符串格式消息。日志消息记录在 INFO 级别，将路由 ID 用作日志名称。默认情况下，路由会连续命名，分别为 route-1、route-2 等。但是，您可以使用 DSL 命令 routeId("myCoolRoute") 来指定自定义路由 ID。

日志 DSL 还提供变体，可让您明确设置日志级别和日志名称。例如，要将日志记录级别明确设置为 LoggingLevel.DEBUG，您可以按照如下所示调用日志 DSL：

日志 DSL 具有超载的方法，可用于设置日志级别和/或名称。

```
from("direct:start").log(LoggingLevel.DEBUG, "Processing ${id}").to("bean:foo");
```

要将日志名称设置为 fileRoute，您可以按照以下方法调用日志 DSL：

```
from("file://target/files").log(LoggingLevel.DEBUG, "fileRoute", "Processing file
${file:name}").to("bean:foo");
```

XML DSL 示例

在 XML DSL 中，日志 DSL 由 日志 元素表示，日志消息通过将 message 属性设置为 Simple 表达式来指定，如下所示：

```
<route id="foo">
    <from uri="direct:foo"/>
    <log message="Got ${body}"/>
    <to uri="mock:foo"/>
</route>
```

log 元素支持 消息、loggingLevel 和 logName 属性。例如：

```
<route id="baz">
    <from uri="direct:baz"/>
    <log message="Me Got ${body}" loggingLevel="FATAL" logName="cool"/>
```

```
    <to uri="mock:baz"/>
</route>
```

全局日志名称

路由 ID 用作默认日志名称。自 Apache Camel 2.17 起，可以通过配置 logname 参数来更改日志名称。

Java DSL，根据以下示例配置日志名称：

```
CamelContext context = ...
context.getProperties().put(Exchange.LOG_EIP_NAME, "com.foo.myapp");
```

在 XML 中，使用以下方法配置日志名称：

```
<camelContext ...>
  <properties>
    <property key="CamelLogEipName" value="com.foo.myapp"/>
  </properties>
```

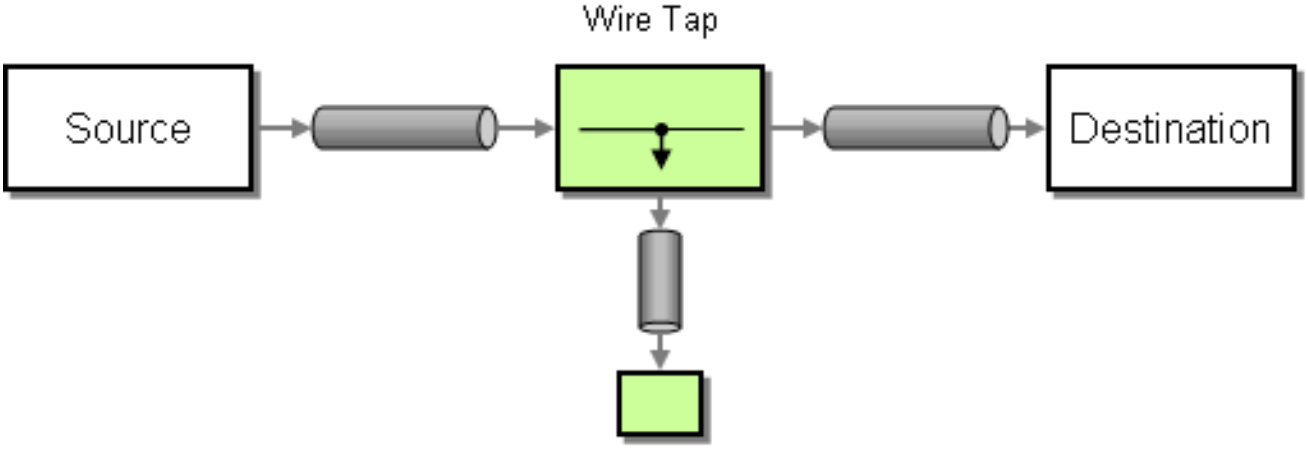
如果您有多个日志，且您希望在所有日志中都有相同的日志名称，您必须将配置添加到每个日志中。

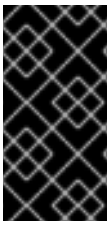
12.3. WIRE TAP

wire Tap

与 图 12.1 “wire Tap Pattern” 所示，有线 tap 模式允许您将消息的副本路由到单独的 tap 位置，而原始消息转发到最终目的地。

图 12.1. wire Tap Pattern





流

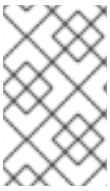
如果连接了流消息正文，您应该考虑启用 流缓存，以确保邮件正文可以重新读取。如需更多有关 流缓存的详细信息

Wiretap 节点

Apache Camel 2.0 引入了用于线 taps 的 wireTap 节点。wireTap 节点将原始交换复制到 tap 的交换中，其交换模式被设置为 InOnly，因为被交换交换应该以 一种方式 传播。交换会在单独的线程中处理，以便它可以与主路由同时运行。

wireTap 支持两种不同的方法来利用交换：

- 排除原始交换的副本。
- 使用一个新的交换实例，您可以自定义被利用的交换。



注意

从 Camel 2.16，Wire Tap EIP 会在将交换发送到线 tap 目的地时发出事件通知。

注意

自 Camel 2.20 起，Wire Tap EIP 在关闭时将完成任意有线流交换。

TAP 原始交换的副本

使用 Java DSL：

```
from("direct:start")
    .to("log:foo")
    .wireTap("direct:tap")
    .to("mock:result");
```

使用 Spring XML 扩展：

```
<route>
  <from uri="direct:start"/>
  <to uri="log:foo"/>
  <wireTap uri="direct:tap"/>
  <to uri="mock:result"/>
</route>
```

TAP 和修改原始交换的副本

使用 Java DSL 时，Apache Camel 支持使用处理器或表达式来修改原始交换的副本。使用处理器可以让您完全了解交换的填充方式，因为您可以设置属性、标头等等。表达式方法只能用于修改 In 消息正文。

例如，使用 处理器 方法修改原始交换的副本：

```
from("direct:start")
  .wireTap("direct:foo", new Processor() {
    public void process(Exchange exchange) throws Exception {
      exchange.getIn().setHeader("foo", "bar");
    }
  }).to("mock:result");

from("direct:foo").to("mock:foo");
```

使用 表达式 方法修改原始交换的副本：

```
from("direct:start")
  .wireTap("direct:foo", constant("Bye World"))
  .to("mock:result");

from("direct:foo").to("mock:foo");
```

使用 Spring XML 扩展，您可以使用 处理器 方法修改原始交换的副本，其中 processorRef 属性引用带有 myProcessor ID 的 spring bean:

```
<route>
  <from uri="direct:start2"/>
  <wireTap uri="direct:foo" processorRef="myProcessor"/>
  <to uri="mock:result"/>
</route>
```

使用 表达式 方法修改原始交换的副本：

```
<route>
   <from uri="direct:start"/>
   <wireTap uri="direct:foo">
      <body><constant>Bye World</constant></body>
   </wireTap>
   <to uri="mock:result"/>
</route>
```

TAP 新交换实例

您可以通过将 **copy** 标记设置为 **false** （默认为 **true**），使用新的交换实例定义 **wiretap**。在本例中，会为 **wiretap** 创建一个初始空交换。

例如，使用 处理器 方法创建新交换实例：

```
from("direct:start")
   .wireTap("direct:foo", false, new Processor() {
      public void process(Exchange exchange) throws Exception {
         exchange.getIn().setBody("Bye World");
         exchange.getIn().setHeader("foo", "bar");
      }
   }).to("mock:result");

from("direct:foo").to("mock:foo");
```

如果第二个 **wireTap** 参数将 **copy** 标记设置为 **false**，这表示 不会 复制原始交换，而是创建一个空交换。

使用 表达式 方法创建新交换实例：

```
from("direct:start")
   .wireTap("direct:foo", false, constant("Bye World"))
   .to("mock:result");

from("direct:foo").to("mock:foo");
```

使用 **Spring XML** 扩展，您可以通过将 **wireTap** 元素的 **copy** 属性设置为 **false** 来指示要创建新的交换。

要使用 处理器 方法创建新交换实例，其中 **processorRef** 属性引用带有 **myProcessor ID** 的 **spring bean**，如下所示：

```
<route>
  <from uri="direct:start2"/>
  <wireTap uri="direct:foo" processorRef="myProcessor" copy="false"/>
  <to uri="mock:result"/>
</route>
```

使用 表达式 方法创建新交换实例：

```
<route>
  <from uri="direct:start"/>
  <wireTap uri="direct:foo" copy="false">
    <body><constant>Bye World</constant></body>
  </wireTap>
  <to uri="mock:result"/>
</route>
```

在 DSL 中发送一个新的 Exchange 并设置标头

可作为 Camel 2.8 提供。

如果您使用 第 12.3 节 “wire Tap” 发送新信息，那么您只能使用 DSL 中的 第 II 部分 “路由表达式和指定语言” 来设置消息正文。如果您需要设置新标头，则必须将该标头使用 第 1.5 节 “处理器”。因此，在 Camel 2.8 中，我们提高了这种情况，因此您可以在 DSL 中设置标头和 DSL。

以下示例发送一个新信息，其中包含

- "通过 World"作为邮件正文
- 带有键 "id" 的标头，值为 123
- 带有键"date"的标题，其当前日期为值

Java DSL

```
from("direct:start")
  // tap a new message and send it to direct:tap
  // the new message should be Bye World with 2 headers
  .wireTap("direct:tap")
    // create the new tap message body and headers
    .newExchangeBody(constant("Bye World"))
```

```
        .newExchangeHeader("id", constant(123))
        .newExchangeHeader("date", simple("${date:now:yyyyMMdd}"))
    .end()
    // here we continue routing the original messages
    .to("mock:result");

// this is the tapped route
from("direct:tap")
    .to("mock:tap");
```

XML DSL

XML DSL 与 Java DSL 稍有不同，因为您如何配置消息正文和标头。在 XML 中，您可以使用 <body> 和 <setHeader>，如下所示：

```
<route>
    <from uri="direct:start"/>
    <!-- tap a new message and send it to direct:tap -->
    <!-- the new message should be Bye World with 2 headers -->
    <wireTap uri="direct:tap">
        <!-- create the new tap message body and headers -->
        <body><constant>Bye World</constant></body>
        <setHeader headerName="id"><constant>123</constant></setHeader>
        <setHeader headerName="date"><simple>${date:now:yyyyMMdd}</simple></setHeader>
    </wireTap>
    <!-- here we continue routing the original message -->
    <to uri="mock:result"/>
</route>
```

使用 URI

*线 **Tap 支持静态和动态端点 URI。静态端点 URI 可从 Camel 2.20 开始使用。***

以下示例演示了如何将 tap 与标头 ID 是队列名称一部分的 JMS 队列连接。

```
from("direct:start")
  .wireTap("jms:queue:backup-${header.id}")
  .to("bean:doSomething");
```

***有关动态端点 URI 的更多信息，**请参阅 “动态到”一节。*

在准备消息时使用 onPrepare 执行自定义逻辑

可作为 Camel 2.8 提供。

详情请查看 第 8.13 节 “多播”。

选项

wireTap DSL 命令支持以下选项：

名称	默认值	描述
uri		用于发送有线条消息的 endpoint uri。您应该使用 **uri** 或 **ref**。
Ref		代表发送有线消息的端点。您应该使用 **uri** 或 **ref**。
executorServiceRef		指的是处理线 tapped 消息时要使用的自定义 第 2.8 节 “线程模型”。如果没有设置，则 Camel 将使用默认线程池。
processorRef		请参考自定义 第 1.5 节 “处理器”以创建新消息（例如发送新消息模式）。请参见以下内容。
复制	**true**	Camel 2.3: 应该是 “交换”一节 的一个副本，以便在有线利用消息时使用。
onPrepareRef		Camel 2.8： 请参阅自定义 第 1.5 节 “处理器”，以准备将 “交换”一节 的副本进行线路。这可让您进行任何自定义逻辑，如 deep-cloning（如果需要）信息有效负载。

部分 II. 路由表达式和指定语言

本指南描述了 Apache Camel 支持的评估语言所用的基本语法。

第 13 章 简介

摘要

本章概述 Apache Camel 支持的所有表达式语言。

13.1. 语言概述

表达式和 predicate 语言表

表 13.1 “表达式和优先级语言” 概述了调用表达式和 predicate 语言的不同语法。

表 13.1. 表达式和优先级语言

语言	静态方法	Fluent DSL Method	XML 元素	注解	工件
请参阅客户门户网站的 *Apache Camel 开发指南*中的 *Bean 集成*。	**bean()**	***EIP*().method()**	**method**	**@Bean**	Camel 内核
第 14 章 *常数*	**constant()**	***EIP*().constant()**	**constant**	**@Constant**	Camel 内核
第 15 章 *EL*	**el()**	***EIP*().el()**	**el**	**@EL**	**Camel-juel**
第 17 章 *groovy*	**groovy()**	***EIP*().groovy()**	**groovy**	**@Groovy**	**Camel-groovy**
第 18 章 *标头*	**header()**	***EIP*().header()**	**header**	**@Header**	Camel 内核
第 19 章 *JavaScript*	**javaScript()**	***EIP*().javaScript()**	**javaScript**	**@JavaScript**	**camel-script**
第 20 章 *JoSQL*	**sql()**	***EIP*().sql()**	**sql**	**@SQL**	**camel-josql**
第 21 章 *JsonPath*	无	***EIP*().jsonpath()**	**jsonpath**	**@JsonPath**	**camel-jsonpath**
第 22 章 *JXPath*	无	***EIP*().jxpath()**	**jxpath**	**@JXPath**	**camel-jxpath**

语言	静态方法	Fluent DSL Method	XML 元素	注解	工件
第 23 章 *MVEL*	**mvel()**	***EIP*().mvel()**	**mvel**	**@MVEL**	**camel-mvel**
第 24 章 *Object-Graph 导航语言 (OGNL)*	**ognl()**	***EIP*().ognl()**	**ognl**	**@OGNL**	**camel-ognl**
第 25 章 *PHP（已弃用）*	**php()**	***EIP*().php()**	**php**	**@PHP**	**camel-script**
第 26 章 *Exchange Property*	**attribute（）**	***EIP*().property()**	**属性**	**@property**	Camel 内核
第 27 章 *Python(DEPRECATED)*	**python()**	***EIP*().python()**	**Python**	**@Python**	**camel-script**
第 28 章 *Ref*	**ref()**	***EIP*().ref()**	**Ref**	N/A	Camel 内核
第 29 章 *Ruby(DEPRECATED)*	**ruby()**	***EIP*().ruby()**	**Ruby**	**@Ruby**	**camel-script**
第 30 章 *简单语言*/第 16 章 *文件语言*	**simple()**	***EIP*().simple()**	**simple**	**@Simple**	Camel 内核
第 31 章 *SpEL*	**spel()**	***EIP*().spel()**	**spel**	**@SpEL**	**Camel-spring**
第 32 章 *XPath 语言*	**xpath()**	***EIP*().xpath()**	**XPath**	**@XPath**	Camel 内核
第 33 章 *XQuery*	**XQuery（）**	***EIP*().xquery()**	**XQuery**	**@XQuery**	**Camel-saxon**

13.2. 如何检查表达式语言

先决条件

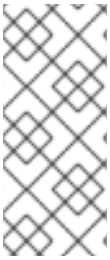
在可以使用特定的表达式语言前，您必须确保 classpath 上有所需的 JAR 文件。如果 Apache Camel 核心中没有包括您要使用的语言，您必须将相关的 JAR 添加到您的类路径中。

如果使用 Maven 构建系统，只需将相关依赖项添加到 POM 文件即可修改构建时类路径。例如，如果您使用 Ruby 语言，请在您的 POM 文件中添加以下依赖项：

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-groovy</artifactId>
  <!-- Use the same version as your Camel core version -->
  <version>${camel.version}</version>
</dependency>
```

如果要在红帽 Fuse OSGi 容器中部署应用程序，您还需要确保安装了相关的语言功能（功能在相应的 Maven 工件后被命名）。例如，若要在 OSGi 容器中使用 Groovy 语言，您必须首先通过输入以下 OSGi 控制台来安装 camel-groovy 功能：

```
karaf@root> features:install camel-groovy
```



注意

如果您在路由中使用表达式或 predicate，请使用 **resource:classpath:path** 或 **resource: path** 来引用值作为外部资源。例如，**resource:classpath:com/foo/myscript.groovy**。

EAP 部署上的 Camel

camel-groovy 组件由 Camel on EAP(Wildfly Camel)框架支持，它在红帽 JBoss 企业应用平台 (JBoss EAP)容器上提供了简化的部署模型。

调用方法

如 表 13.1 “表达式和优先级语言” 所示，根据所使用的上下文，调用表达式语言有几种不同的语法。您可以调用表达式语言：

- 作为静态方法
- 作为流畅的 DSL 方法
- 作为 XML 元素

- *作为注解*

作为静态方法

大多数语言都定义了静态方法，可在任何上下文中使用 org.apache.camel.Expression 类型或 org.apache.camel.Predicate 类型是正常的。静态方法使用字符串表达式（或 predicate）作为其参数，并返回 Expression 对象（通常是一个 Predicate 对象）。

例如，要实施基于内容的路由器以 XML 格式处理消息，您可以根据 /order/address/countryCode 元素的值路由消息，如下所示：

```
from("SourceURL")
  .choice
    .when(xpath("/order/address/countryCode = 'us'"))
      .to("file://countries/us/")
    .when(xpath("/order/address/countryCode = 'uk'"))
      .to("file://countries/uk/")
    .otherwise()
      .to("file://countries/other/")
  .to("TargetURL");
```

作为流畅的 DSL 方法

Java fluent DSL 支持另一种调用表达式语言。您可以将表达式作为 Enterprise Integration Pattern(EIP)的参数提供，而是作为 DSL 命令的子使用提供表达式。例如，不将 XPath 表达式作为 filter（xpath("Expression")）调用，您可以将表达式称为 filter（）.xpath("Expression")。

例如，前面的基于内容的路由器可以在这个调用中重新实施，如下所示：

```
from("SourceURL")
  .choice
    .when().xpath("/order/address/countryCode = 'us'")
      .to("file://countries/us/")
    .when().xpath("/order/address/countryCode = 'uk'")
      .to("file://countries/uk/")
    .otherwise()
      .to("file://countries/other/")
  .to("TargetURL");
```

作为 XML 元素

您还可以通过将表达式字符串放在相关的 XML 元素中，在 XML 中调用表达式语言。

例如，在 XML 中调用 XPath 的 XML 元素是 xpath （属于标准 Apache Camel 命名空间）。您可以在基于 XML DSL 内容的路由器中使用 XPath 表达式，如下所示：

```
<from uri="file://input/orders"/>
<choice>
  <when>
    <xpath>/order/address/countryCode = 'us'</xpath>
    <to uri="file://countries/us/"/>
  </when>
  <when>
    <xpath>/order/address/countryCode = 'uk'</xpath>
    <to uri="file://countries/uk/"/>
  </when>
  <otherwise>
    <to uri="file://countries/other/"/>
  </otherwise>
</choice>
```

或者，您可以使用 language 元素指定语言表达式，您可以在其中指定 language 属性中的语言名称。例如，您可以使用 语言 元素定义 XPath 表达式，如下所示：

```
<language language="xpath">/order/address/countryCode = 'us'</language>
```

作为注解

语言注释在 bean 集成的上下文中使用。该注释提供了一种便捷方式，用于从消息或标头提取信息，然后将提取的数据注入 bean 的方法解析器。

例如，请考虑 bean myBeanProc，它作为 filter（） EIP 的 predicate 调用。如果 bean 的 checkCredentials 方法返回 true，则消息被允许继续；但如果方法返回 false，则消息会被阻止。过滤器模式实施如下：

```
// Java
MyBeanProcessor myBeanProc = new MyBeanProcessor();

from("SourceURL")
  .filter().method(myBeanProc, "checkCredentials")
  .to("TargetURL");
```

MyBeanProcessor 类的实现利用 @XPath 注释从基础 XML 消息中提取 用户名和密码 ，如下所示：

-

```
// Java
import org.apache.camel.language.XPath;

public class MyBeanProcessor {
    boolean void checkCredentials(
        @XPath("/credentials/username/text()") String user,
        @XPath("/credentials/password/text()") String pass
    ) {
        // Check the user/pass credentials...
        ...
    }
}
```

@XPath 注释放在它要注入的参数之前。请注意，XPath 表达式如何显式选择文本节点，方法是将 /text（） 附加到路径中，这样可确保仅选择元素的内容，而不是包含标签。

作为 Camel 端点 URI

通过使用 Camel 语言组件，您可以在端点 URI 中调用支持的语言。有两种替代语法。

要调用存储在文件（或者由 Scheme定义的其他资源类型）中的语言脚本，请使用以下 URI 语法：

```
language://LanguageName:resource:Scheme:Location[?Options]
```

其中的方案可以是 file：、classpath: 或 http:。

例如，以下路由从 classpath 中执行 mysimplescript.txt ：

```
from("direct:start")
  .to("language:simple:classpath:org/apache/camel/component/language/mysimplescript.txt")
  .to("mock:result");
```

要调用嵌入的语言脚本，请使用以下 URI 语法：

```
language://LanguageName[:Script][?Options]
```

例如，运行存储在 script 字符串中的 Simple 语言 脚本 ：

```
String script = URLEncoder.encode("Hello ${body}", "UTF-8");
from("direct:start")
```

```
.to("language:simple:" + script)
.to("mock:result");
```

*有关语言组件的详情，请参阅 **Apache Camel** 组件参考指南 中的 语言。*

第 14 章 常数

概述

恒定语言是一个简单的内置语言，用于指定纯文本字符串。这样便可在预期表达式类型的任何上下文中提供纯文本字符串。

XML 示例

在 XML 中，您可以将 用户名 标头设置为值 Jane Doe，如下所示：

```
<camelContext>
  <route>
    <from uri="SourceURL"/>
    <setHeader headerName="username">
      <constant>Jane Doe</constant>
    </setHeader>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

JAVA 示例

在 Java 中，您可以将 用户名 标头设置为值 Jane Doe，如下所示：

```
from("SourceURL")
  .setHeader("username", constant("Jane Doe"))
  .to("TargetURL");
```

第 15 章 EL

概述

Unified Expression Language(EL)最初被指定为 JSP 2.1 标准(JSR-245)的一部分，但现在可作为独立语言使用。Apache Camel 与 JUEL(http://juel.sourceforge.net/)集成，它是 EL 语言的开源实现。

添加 JUEL 软件包

要在路由中使用 EL，您需要将对 camel-juel 的依赖添加到项目，如 例 15.1 “添加 camel-juel 依赖项” 所示。

例 15.1. 添加 camel-juel 依赖项

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-7_10_0-00018-redhat-00001</camel-version>
  ...
</properties>

<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-juel</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静态导入

要在应用程序代码中使用 el（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.language.juel.JuelExpression.el;
```

变量

表 15.1 “EL 变量” 列出使用 EL 时可访问的变量。

表 15.1. EL 变量

变量	类型	值
Exchange	**org.apache.camel.Exchange**	当前交换
in	**org.apache.camel.Message**	IN 信息
out	**org.apache.camel.Message**	OUT 消息

示例

例 15.2 “使用 EL 的路由” 显示使用 EL 的两个路由。

例 15.2. 使用 EL 的路由

```
<camelContext>
  <route>
    <from uri="seda:foo"/>
    <filter>
      <language language="el">${in.headers.foo == 'bar'}</language>
      <to uri="seda:bar"/>
    </filter>
  </route>
  <route>
    <from uri="seda:foo2"/>
    <filter>
      <language language="el">${in.headers['My Header'] == 'bar'}</language>
      <to uri="seda:bar"/>
    </filter>
  </route>
</camelContext>
```

第 16 章 文件语言

摘要

文件语言是对简单语言的扩展，而不是独立语言。但是，文件语言扩展只能与文件或 FTP 端点结合使用。

16.1. 使用文件语言时

概述

文件语言是简单语言的扩展，它并非始终可用。您可以在以下情况下使用它：

- 在文件或 FTP 使用者端点 中。
- 在交换上，由文件或 FTP 消费者 创建。

注意

转义字符 \ 在文件语言中不可用。

在文件或 FTP 消费者端点中

您可以在 File 或 FTP 消费者端点上设置多个 URI 选项，使用文件语言表达式作为值。例如，在文件消费者端点 URI 中，您可以使用文件表达式设置 fileName、move、preMove、moveFailed 和 sortBy 选项。

在文件消费者端点中，fileName 选项充当过滤器，确定哪个文件实际上会从起始目录中读取。如果指定了纯文本字符串（例如 fileName=report.txt），则文件消费者在每次更新时读取同一文件。您可通过指定简单表达式使此选项更加动态。例如，您可以在每次文件消费者轮询起始目录时选择不同的文件，如下所示：

```
file://target/filelanguage/bean/?fileName=${bean:counter.next}.txt&delete=true
```

其中 ${bean:counter.next} 表达式调用在 ID 下注册的 bean 上的 next（） 方法，计数器 为。

move 选项用于在文件消费者端点读取后将文件移动到备份位置。例如，以下端点会在处理后将文件移动到备份目录中：

```
file://target/filelanguage/?
move=backup/${date:now:yyyyMMdd}/${file:name.noext}.bak&recursive=false
```

其中 **${file:name.noext}.bak** 表达式修改原始文件，用 **.bak** 替换文件扩展名。

您可以使用 **sortBy** 选项指定应处理文件的顺序。例如，要根据文件名的字母顺序处理文件，您可以使用以下文件消费者端点：

```
file://target/filelanguage/?sortBy=file:name
```

要根据上次修改的顺序处理文件，您可以使用以下文件消费者端点：

```
file://target/filelanguage/?sortBy=file:modified
```

您可以通过添加 **reverse: prefix**的 **prefix: prefix>_<-abrtfor** 示例来反转顺序：

```
file://target/filelanguage/?sortBy=reverse:file:modified
```

在由文件或 FTP 使用者创建的交换中

当交换来自文件或 **FTP** 消费者端点时，可以将文件语言表达式应用到整个路由中的交换（只要未清除原始消息标头）。例如，您可以定义基于内容的路由器，该路由器根据其文件扩展名路由消息，如下所示：

```
<from uri="file://input/orders"/>
<choice>
  <when>
    <simple>${file:ext} == 'txt'</simple>
    <to uri="bean:orderService?method=handleTextFiles"/>
  </when>
  <when>
    <simple>${file:ext} == 'xml'</simple>
    <to uri="bean:orderService?method=handleXmlFiles"/>
  </when>
  <otherwise>
    <to uri="bean:orderService?method=handleOtherFiles"/>
  </otherwise>
</choice>
```

16.2. 文件变量

概述

只要路由以文件或 FTP 消费者端点启动，都可以使用文件变量，这意味着底层消息正文为 java.io.File 类型。文件变量可让您访问文件路径名称的各种部分，几乎就像您调用 java.io.File 类的方法（实际上，文件语言从文件或 FTP 端点设置的消息标头中提取所需信息）。

开始目录

一些文件变量返回路径相对于 起始目录 而定义，这只是在 File 或 FTP 端点中指定的目录。例如，以下文件消费者端点具有起始目录 ./filetransfer （相对路径）：

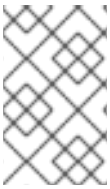
```
file:filetransfer
```

以下 FTP 使用者端点具有起始目录 ./ftptransfer （相对路径）：

```
ftp://myhost:2100/ftptransfer
```

文件变量命名规则

通常，文件变量使用 java.io.File 类上对应的方法命名。例如，file:absolute 变量提供 java.io.File.getAbsolute（） 方法返回的值。



注意

但是，这个命名规则不严格。例如，没有 像 java.io.File.getSize（） 这样的方法。

变量表

表 16.1 “文件语言的变量” 显示文件语言支持的所有变量。

表 16.1. 文件语言的变量

变量	类型	描述
file:name	**字符串**	相对于起始目录的路径名称.

变量	类型	描述
file:name.ext	**字符串**	文件扩展名（在路径名称中的最后一个 **.** 字符后面加上字符）。支持多个点的文件扩展，如 .tar.gz。
file:name.ext.single	**字符串**	文件扩展名（在路径名称中的最后一个 **.** 字符后面加上字符）。如果文件扩展名有 mutiple 点，则此表达式仅返回最后一部分。
file:name.noext	**字符串**	相对于起始目录的路径名称，省略文件扩展名。
file:name.noext.single	**字符串**	相对于起始目录的路径名称，省略文件扩展名。如果文件扩展具有多个点，则此表达式仅剥离最后一个部分，并保留其他内容。
file:onlyname	**字符串**	路径名称的最终部分。也就是说，文件名没有父目录路径。
file:onlyname.noext	**字符串**	路径名称的最后一个部分，省略文件扩展名。
file:onlyname.noext.single	**字符串**	路径名称的最后一个部分，省略文件扩展名。如果文件扩展具有多个点，则此表达式仅剥离最后一个部分，并保留其他内容。
file:ext	**字符串**	文件扩展名（与 **file:name.ext**一样）。
file:parent	**字符串**	父目录的路径名，包括路径中的起始目录。
file:path	**字符串**	文件路径名称，包括路径中的起始目录。
file:absolute	**布尔值**	**true**，如果将起始目录指定为绝对路径，则为 **false**，否则为 false。
file:absolute.path	**字符串**	文件的绝对路径名。
file:length	**Long**	所引用文件的大小。
file:size	**Long**	**与 文件相同：长度.**

变量	类型	描述
file:modified	**java.util.Date**	上次修改日期.

16.3. 例子

相对路径名

请考虑文件消费者端点，其中起始目录指定为 相对路径名。例如，以下 File 端点包含起始目录 ./filelanguage ：

```
file://filelanguage
```

现在，在扫描 文件语言 目录时，假设端点只消耗了以下文件：

```
./filelanguage/test/hello.txt
```

最后，假设 filelanguage 目录本身具有以下绝对位置：

```
/workspace/camel/camel-core/target/filelanguage
```

根据上述情况，文件语言变量会在应用到当前交换时返回以下值：

表达式	结果
file:name	**test/hello.txt**
file:name.ext	**txt**
file:name.noext	**test/hello**
file:onlyname	**hello.txt**
file:onlyname.noext	**您好**
file:ext	**txt**
file:parent	**filelanguage/test**
file:path	**filelanguage/test/hello.txt**

表达式	结果
file:absolute	**false**
file:absolute.path	**/workspace/camel/camel-core/target/filelanguage/test/hello.txt**

绝对路径名称

请考虑文件消费者端点，其中起始目录指定为 绝对路径名。例如，以下文件端点包含起始目录 /workspace/camel/camel-core/target/filelanguage ：

```
file:///workspace/camel/camel-core/target/filelanguage
```

现在，在扫描 文件语言 目录时，假设端点只消耗了以下文件：

```
./filelanguage/test/hello.txt
```

根据上述情况，文件语言变量会在应用到当前交换时返回以下值：

表达式	结果
file:name	**test/hello.txt**
file:name.ext	**txt**
file:name.noext	**test/hello**
file:onlyname	**hello.txt**
file:onlyname.noext	**您好**
file:ext	**txt**
file:parent	**/workspace/camel/camel-core/target/filelanguage/test**
file:path	**/workspace/camel/camel-core/target/filelanguage/test/hello.txt**
file:absolute	**true**

表达式	结果
file:absolute.path	**/workspace/camel/camel-core/target/filelanguage/test/hello.txt**

第 17 章 GROOVY

概述

Groovy 是基于 Java 的脚本语言，允许快速解析对象。Groovy 支持是 camel-groovy 模块的一部分。

添加 SCRIPT 模块

要在路由中使用 Groovy，您需要在您的项目中添加 camel-groovy 的依赖关系，如 例 17.1 “添加 camel-groovy 依赖项” 所示。

例 17.1. 添加 camel-groovy 依赖项

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-7_10_0-00018-redhat-00001</camel-version>
  ...
</properties>

<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-groovy</artifactId>
    <version>${camel-version}</version>
  </dependency>
</dependencies>
```

静态导入

要在应用程序代码中使用 groovy（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

内置属性

表 17.1 “Groovy 属性” 列出在使用 Groovy 时可以访问的内置属性。

表 17.1. Groovy 属性

属性	类型	值
context	**org.apache.camel.CamelContext**	Camel 上下文
Exchange	**org.apache.camel.Exchange**	当前交换
Request（请求）	**org.apache.camel.Message**	IN 信息
响应	**org.apache.camel.Message**	OUT 消息
属性	**org.apache.camel.builder.script.PropertiesFunction**	通过 **解析** 方法运行，可以更轻松地在脚本中使用属性组件。

在 ENGINE_SCOPE 设置的属性。

示例

例 17.2 “使用 Groovy 的路由” 显示两个使用 Groovy 脚本的路由。

例 17.2. 使用 Groovy 的路由

```
<camelContext>
  <route>
    <from uri="direct:items" />
    <filter>
      <language language="groovy">request.lineItems.any { i -> i.value > 100 }</language>
      <to uri="mock:mock1" />
    </filter>
  </route>
  <route>
    <from uri="direct:in"/>
    <setHeader headerName="firstName">
      <language language="groovy">$user.firstName $user.lastName</language>
    </setHeader>
    <to uri="seda:users"/>
  </route>
</camelContext>
```

使用属性组件

要从属性组件访问属性值，调用内置 属性属性 上的 解析 方法，如下所示：

```
.setHeader("myHeader").groovy("properties.resolve(PropKey)")
```

其中 PropKey 是您要解析的属性的键，其中键值是 String 类型。

有关属性组件的详情，请参阅 Apache Camel 组件参考指南 中的 Properties。

自定义 GROOVY SHELL

有时，您可能需要在 Groovy 表达式中使用自定义 GroovyShell 实例。为了提供自定义 GroovyShell，请在 Camel 注册表中添加 org.apache.camel.language.groovy.Groovy SPI 接口的实施。

例如，当您将以下 bean 添加到 Spring 上下文中时，Apache Camel 将使用包含自定义静态导入的自定义 GroovyShell 实例，而不是默认值。

```
public class CustomGroovyShellFactory implements GroovyShellFactory {

  public GroovyShell createGroovyShell(Exchange exchange) {
    ImportCustomizer importCustomizer = new ImportCustomizer();
    importCustomizer.addStaticStars("com.example.Utils");
    CompilerConfiguration configuration = new CompilerConfiguration();
    configuration.addCompilationCustomizers(importCustomizer);
    return new GroovyShell(configuration);
  }
 }
```

第 18 章 标头

概述

标头语言提供了在当前消息中轻松访问标头值的便捷方式。提供标头名称时，标头语言将执行不区分大小写的查找，并返回对应的标头值。

标头语言是 **camel-core** 的一部分。

XML 示例

例如，根据 **SequenceNumber** 标头的值重新排序传入的交换（其中序列数必须是正整数），您可以按照如下所示定义路由：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <resequence>
      <language language="header">SequenceNumber</language>
    </resequence>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

JAVA 示例

同一路由可以在 **Java** 中定义，如下所示：

```
from("SourceURL")
  .resequence(header("SequenceNumber"))
  .to("TargetURL");
```

第 19 章 JAVASCRIPT

概述

JavaScript（也称为 ECMAScript）是基于 Java 的脚本语言，允许快速解析对象。JavaScript 支持是 camel-script 模块的一部分。

添加 SCRIPT 模块

要在路由中使用 JavaScript，您需要在您的项目中添加对 camelscript 的依赖项，如 例 19.1 “添加 camel-script 依赖项” 所示。

例 19.1. 添加 camel-script 依赖项

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-7_10_0-00018-redhat-00001</camel-version>
  ...
</properties>

<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静态导入

要在应用程序代码中使用 javaScript（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

内置属性

表 19.1 “JavaScript 属性” 列出在使用 JavaScript 时可以访问的内置属性。

表 19.1. JavaScript 属性

属性	类型	值
context	**org.apache.camel.CamelContext**	Camel 上下文
Exchange	**org.apache.camel.Exchange**	当前交换
Request（请求）	**org.apache.camel.Message**	IN 信息
响应	**org.apache.camel.Message**	OUT 消息
属性	**org.apache.camel.builder.script.PropertiesFunction**	通过 **解析** 方法运行，可以更轻松地在脚本中使用属性组件。

在 ENGINE_SCOPE 设置的属性。

示例

例 19.2 “使用 JavaScript 的路由” 显示使用 JavaScript 的路由。

例 19.2. 使用 JavaScript 的路由

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <langauge langauge="javaScript">request.headers.get('user') == 'admin'</langauge>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

使用属性组件

要从属性组件访问属性值，调用内置 属性属性 上的 解析 方法，如下所示：

```
.setHeader("myHeader").javaScript("properties.resolve(PropKey)")
```

其中 PropKey 是您要解析的属性的键，其中键值是 String 类型。

有关属性组件的详情，请参阅 Apache Camel 组件参考指南 中的 Properties。

第 20 章 JOSQL

概述

JoSQL（用于 Java 对象的SQL）语言可让您评估 Apache Camel 中的 predicates 和表达式。JoSQL 使用类似 SQL 的查询语法，对来自内存 Java 对象的 data 执行选择和排序操作，JoSQL 不是一个 数据库。在 JoSQL 语法中，每个 Java 对象实例被视为表行，每个对象方法被视为列名称。使用这个语法，可以构造强大的语句，以便从 Java 对象的集合中提取和编译数据。详情请查看 http://josql.sourceforge.net/。

添加 JOSQL 模块

要在路由中使用 JoSQL，您需要在您的项目中添加对 camel-josql 的依赖项，如 例 20.1 “添加 camel-josql 依赖项” 所示。

例 20.1. 添加 camel-josql 依赖项

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-josql</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静态导入

要在应用程序代码中使用 sql（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.builder.sql.SqlBuilder.sql;
```

变量

表 20.1 “SQL 变量” 列出使用 JoSQL 时可访问的变量。

表 20.1. SQL 变量

名称	类型	描述
Exchange	**org.apache.camel.Exchange**	当前交换
in	**org.apache.camel.Message**	IN 信息
out	**org.apache.camel.Message**	OUT 消息
属性	**对象**	Exchange 属性，其键是 *属性*
header	**对象**	键为标头的 IN 消息 *标头*
变量	**对象**	其键为 *变量的变量*

示例

例 20.2 “使用 JoSQL 的路由” 显示使用 JoSQL 的路由。

例 20.2. 使用 JoSQL 的路由

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <setBody>
      <language language="sql">select * from MyType</language>
    </setBody>
    <to uri="seda:regularQueue"/>
  </route>
</camelContext>
```

第 21 章 JSONPATH

概述

JsonPath 语言提供了一种方便的语法，用于提取 JSON 消息的部分。JSON 的语法与 XPath 类似，但它用于从 JSON 消息提取 JSON 对象，而不是对 XML 执行操作。jsonpath DSL 命令可用作表达式或 predicate（空结果解释为布尔值 false）。

添加 JSONPATH 软件包

要在 Camel 路由中使用 JsonPath，您需要将对 camel-jsonpath 的依赖添加到项目中，如下所示：

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jsonpath</artifactId>
  <version>${camel-version}</version>
</dependency>
```

JAVA 示例

以下 Java 示例演示了如何使用 jsonpath（） DSL 命令在特定价格范围内选择项目：

```
from("queue:books.new")
  .choice()
    .when().jsonpath("$.store.book[?(@.price < 10)]")
      .to("jms:queue:book.cheap")
    .when().jsonpath("$.store.book[?(@.price < 30)]")
      .to("jms:queue:book.average")
    .otherwise()
      .to("jms:queue:book.expensive")
```

如果 JsonPath 查询返回空集，则结果将解释为 false。这样，您可以使用 JsonPath 查询作为 predicate。

XML 示例

以下 XML 示例演示了如何使用 jsonpath DSL 元素在路由中定义 predicates：

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
```

```
    <choice>
      <when>
        <jsonpath>$.store.book[?(@.price < 10)]</jsonpath>
        <to uri="mock:cheap"/>
      </when>
      <when>
        <jsonpath>$.store.book[?(@.price < 30)]</jsonpath>
        <to uri="mock:average"/>
      </when>
      <otherwise>
        <to uri="mock:expensive"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

简单的语法

您希望使用 jsonpath 语法定义基本 predicate 时，很难记住语法。例如，要找到所有无线图书，您必须按如下方式编写语法：

```
$.store.book[?(@.price < 20)]
```

但是，如果您可以按以下方式写入它：

```
store.book.price < 20
```

如果只想使用价格键查看节点，您也可以省略该路径：

```
price < 20
```

为了支持此操作，有一个 EasyPredicateParser，您可以使用基本样式来定义 predicate。这意味着 predicate 不能使用 $ 符号启动，且只能包含一个 operator。简单语法如下：

```
left OP right
```

您可以在正确的 Operator 中使用 Camel 简单语言，例如：

```
store.book.price < ${header.limit}
```

支持的消息正文类型

Camel JSonPath 支持使用以下类型的消息正文：

类型	描述
File	从文件读取
字符串	普通字符串
map	将正文作为 **java.util.Map** 类型
list	java.util.List 类型的消息正文
POJO	可选的 Jackson 位于类路径上，那么 **camel-jsonpath** 可以使用 Jackson 将消息正文读为 **POJO**，并转换为 **java.util.Map**，它由 JSonPath 支持。例如，您可以将 **camel-jackson** 作为依赖项添加，以包含 Jackson。
InputStream	如果上述类型都不匹配，则 Camel 会尝试将消息正文读为 **java.io.InputStream**。

如果消息正文不受支持，则默认抛出异常，但您可以将 JSonPath 配置为禁止异常。

隐藏异常

如果没有找到 jsonpath 表达式配置的路径，JSONPath 将抛出异常。通过将 SuppressExceptions 选项设置为 true 来忽略异常。例如，在下面的代码中，将 true 选项添加为 jsonpath 参数的一部分：

```
from("direct:start")
    .choice()
        // use true to suppress exceptions
        .when().jsonpath("person.middlename", true)
            .to("mock:middle")
        .otherwise()
            .to("mock:other");
```

在 XML DSL 中，使用以下语法：

```
<route>
  <from uri="direct:start"/>
  <choice>
    <when>
      <jsonpath suppressExceptions="true">person.middlename</jsonpath>
```

```
      <to uri="mock:middle"/>
    </when>
    <otherwise>
      <to uri="mock:other"/>
    </otherwise>
  </choice>
</route>
```

JSONPATH INJECTION

在使用 bean 集成来调用 bean 方法时，您可以使用 JsonPath 从消息中提取值，并将它绑定到 method 参数。例如：

```
// Java
public class Foo {

    @Consume(uri = "activemq:queue:books.new")
    public void doSomething(@JsonPath("$.store.book[*].author") String author, @Body String json) {
      // process the inbound message here
    }
}
```

内联简单表达式

Camel 2.18 中的新功能.

Camel 在 JsonPath 表达式中支持内联 简单 表达式。简单 语法必须用简单语法表达 简单 语言，如下所示：

```
from("direct:start")
  .choice()
    .when().jsonpath("$.store.book[?(@.price < `${header.cheap}`)]")
      .to("mock:cheap")
    .when().jsonpath("$.store.book[?(@.price < `${header.average}`)]")
      .to("mock:average")
    .otherwise()
      .to("mock:expensive");
```

通过设置 选项 allow Simple =false （如下所示）关闭对 Simple 表达式的支持。

Java：

```
// Java DSL
.when().jsonpath("$.store.book[?(@.price < 10)]", false, false)
```

XML DSL：

```
// XML DSL
<jsonpath allowSimple="false">$.store.book[?(@.price &lt; 10)]</jsonpath>
```

参考

有关 JsonPath 的详情，请查看 JSonPath 项目页面。

第 22 章 JXPATH

概述

JXPath 语言允许您使用 Apache Commons JXPath 语言调用 Java Bean。JXPath 语言的语法与 XPath 相似，但不是从 XML 文档选择元素或属性节点，而是调用 Java Bean 对象图的方法。如果其中一个 bean 属性返回 XML 文档（ DOM/JDOM 实例），但路径的其余部分将解释为 XPath 表达式，用于从文档中提取 XML 节点。换句话说，JXPath 语言提供混合对象图形导航和 XML 节点选择。

添加 JXPATH 软件包

要在路由中使用 JXPath，您需要将有关 camel-jxpath 的依赖项添加到项目，如 例 22.1 “添加 camel-jxpath 依赖项” 所示。

例 22.1. 添加 camel-jxpath 依赖项

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-7_10_0-00018-redhat-00001</camel-version>
  ...
</properties>

<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jxpath</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

变量

表 22.1 “JXPath 变量” 列出使用 JXPath 时可访问的变量。

表 22.1. JXPath 变量

变量	类型	值
这	**org.apache.camel.Exchange**	当前交换
in	**org.apache.camel.Message**	IN 信息

变量	类型	值
out	**org.apache.camel.Message**	OUT 消息

选项

表 22.2 “JXPath 选项” 描述 JXPath 的选项。

表 22.2. JXPath 选项

选项	类型	描述
lenient	**布尔值**	Camel 2.11/2.10.5：允许其打开 JXPathContext。当启用此选项时，JXPath 表达式可以针对表达式和消息正文（可能无效或缺失的数据）评估。请参阅 JXPath 文档。此选项默认为 false。

例子

以下示例路由使用 JXPath：

```
<camelContext>
  <route>
    <from uri="activemq:MyQueue"/>
    <filter>
      <jxpath>in/body/name = 'James'</xpath>
      <to uri="mqseries:SomeOtherQueue"/>
    </filter>
  </route>
</camelContext>
```

以下简单示例在 Message Filter 中将 JXPath 表达式用作 predicate：

```
from("direct:start").
    filter().jxpath("in/body/name='James'").
    to("mock:result");
```

JXPATH 注入

您可以使用 Bean 在 Bean 上调用方法并使用各种语言（如 JXPath）从消息中提取值并将其绑定到方

法参数。

例如：

```
public class Foo {
    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@JXPath("in/body/foo") String correlationID, @Body String body)
    { // process the inbound message here }
   }
```

从外部资源载入脚本

从 Camel 2.11 开始可用

您可以对脚本进行外部化，并使 Camel 从资源（如 "classpath:"、"file:" 或 "http:" ）加载。请遵循以下语法：

```
"resource:scheme:location"
```

例如，要引用类路径上的文件：

```
.setHeader("myHeader").jxpath("resource:classpath:myjxpath.txt")
```

第 23 章 MVEL

概述

MVEL 是一个基于 Java 的动态语言，与 OGNL 类似，但报告速度要快得多。MVEL 支持在 camel-mvel 模块中。

语法

您可以使用 MVEL dot 语法调用 Java 方法，例如：

```
getRequest().getBody().getFamilyName()
```

由于 MVEL 是动态键入的，所以在调用 getFamilyName（） 方法前，无需广播消息正文实例（对象类型）。您还可以使用缩写语法调用 Bean 属性，例如：

```
request.body.familyName
```

添加 MVEL 模块

要在路由中使用 MVEL，您需要将对 camel-mvel 的依赖添加到项目，如 例 23.1 “添加 camel-mvel 依赖项” 所示。

例 23.1. 添加 camel-mvel 依赖项

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-7_10_0-00018-redhat-00001</camel-version>
  ...
</properties>

<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mvel</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

内置变量

表 23.1 “MVEL 变量” 列出在使用 MVEL 时可以访问的内置变量。

表 23.1. MVEL 变量

名称	类型	描述
这	**org.apache.camel.Exchange**	当前交换
Exchange	**org.apache.camel.Exchange**	当前交换
例外	**Throwable**	Exchange 例外（若有）
exchangeID	**字符串**	Exchange ID
fault	**org.apache.camel.Message**	失败消息（若有）
Request（请求）	**org.apache.camel.Message**	IN 信息
响应	**org.apache.camel.Message**	OUT 消息
属性	**map**	Exchange 属性
property(*name*)	对象	命名的 Exchange 属性的值
属性（*名称*，*键入*）	*类型*	名为 Exchange 属性的输入值

示例

例 23.2 “使用 MVEL 的路由” 显示使用 MVEL 的路由。

例 23.2. 使用 MVEL 的路由

```
<camelContext>
  <route>
    <from uri="seda:foo"/>
    <filter>
      <language langauge="mvel">request.headers.foo == 'bar'</language>
      <to uri="seda:bar"/>
    </filter>
  </route>
</camelContext>
```

第 24 章 OBJECT-GRAPH 导航语言(OGNL)

概述

OGNL 是用于获取和设置 Java 对象属性的表达式语言。为 get 和 设置 属性的值都使用相同的表达式。OGNL 支持在 camel-ognl 模块中。

EAP 部署上的 CAMEL

此组件由 Camel on EAP(Wildfly Camel)框架支持，它可在红帽 JBoss 企业应用平台(JBoss EAP)容器上提供简化的部署模型。

添加 OGNL 模块

要在路由中使用 OGNL，您需要将有关 camel-ognl 的依赖项添加到您的项目中，如 例 24.1 “添加 camel-ognl 依赖项” 所示。

例 24.1. 添加 camel-ognl 依赖项

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ognl</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静态导入

要在应用程序代码中使用 ognl() 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.language.ognl.OgnlExpression.ognl;
```

内置变量

表 24.1 “OGNL 变量” 列出在使用 OGNL 时可以访问的内置变量。

表 24.1. OGNL 变量

名称	类型	描述
这	**org.apache.camel.Exchange**	当前交换
Exchange	**org.apache.camel.Exchange**	当前交换
例外	**Throwable**	Exchange 例外（若有）
exchangeID	**字符串**	Exchange ID
fault	**org.apache.camel.Message**	失败消息（若有）
Request（请求）	**org.apache.camel.Message**	IN 信息
响应	**org.apache.camel.Message**	OUT 消息
属性	**map**	Exchange 属性
property(*name*)	对象	命名的 Exchange 属性的值
属性（*名称*，*键入*）	*类型*	名为 Exchange 属性的输入值

示例

例 24.2 “使用 OGNL 的路由” 显示使用 OGNL 的路由。

例 24.2. 使用 OGNL 的路由

```
<camelContext>
  <route>
    <from uri="seda:foo"/>
    <filter>
      <language langauge="ognl">request.headers.foo == 'bar'</language>
      <to uri="seda:bar"/>
    </filter>
  </route>
</camelContext>
```

第 25 章 PHP（已弃用）

概述

PHP 是广泛使用的通用脚本语言，特别适用于 Web 开发。PHP 支持是 camel-script 模块的一部分。

重要

Apache Camel 中的 PHP 已被弃用，并将在以后的版本中删除。

添加 SCRIPT 模块

要在路由中使用 PHP，您需要在您的项目中添加对 camelscript 的依赖项，如 例 25.1 “添加 camel-script 依赖项” 所示。

例 25.1. 添加 camel-script 依赖项

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静态导入

要在应用程序代码中使用 php（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

内置属性

表 25.1 “PHP 属性” 列出使用 PHP 时可以访问的内置属性。

表 25.1. PHP 属性

属性	类型	值
context	**org.apache.camel.CamelContext**	Camel 上下文
Exchange	**org.apache.camel.Exchange**	当前交换
Request（请求）	**org.apache.camel.Message**	IN 信息
响应	**org.apache.camel.Message**	OUT 消息
属性	**org.apache.camel.builder.script.PropertiesFunction**	通过 **解析** 方法运行，可以更轻松地在脚本中使用属性组件。

在 ENGINE_SCOPE 设置的属性。

示例

例 25.2 “使用 PHP 的路由” 显示使用 PHP 的路由。

例 25.2. 使用 PHP 的路由

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <language language="php">strpos(request.headers.get('user'), 'admin')!==
FALSE</language>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

使用属性组件

要从属性组件访问属性值，调用内置 属性属性 上的 解析 方法，如下所示：

```
.setHeader("myHeader").php("properties.resolve(PropKey)")
```

*其中 **PropKey** 是您要解析的属性的键，其中键值是 **String** 类型。*

*有关属性组件的详情，请参阅 **Apache Camel** 组件参考指南 中的 **Properties**。*

第 26 章 EXCHANGE PROPERTY

概述

交换属性语言提供了一种便捷方式来访问 交换属性。当您提供与某一交换属性名称匹配的键时，交换属性语言会返回对应的值。

Exchange 属性语言是 camelcore 的一部分。

XML 示例

例如，要在 listOfEndpoints 交换属性包含接收者列表时实施接收者列表模式，您可以按照如下所示定义路由：

```
<camelContext>
  <route>
    <from uri="direct:a"/>
    <recipientList>
      <exchangeProperty>listOfEndpoints</exchangeProperty>
    </recipientList>
  </route>
</camelContext>
```

JAVA 示例

同一接收者列表示例可以在 Java 中按照如下方式实施：

```
from("direct:a").recipientList(exchangeProperty("listOfEndpoints"));
```

第 27 章 PYTHON(DEPRECATED)

概述

Python 是一个极其强大的动态编程语言，可在各种应用程序域中使用。Python 通常与 Tcl、Perl、Ruby、Scheme 或 Java 进行比较。Python 支持是 camel-script 模块的一部分。

重要

Apache Camel 中的 Python 已被弃用，并将在以后的版本中删除。

添加 SCRIPT 模块

要在路由中使用 Python，您需要将 camel-script 的依赖项添加到您的项目中，如 例 27.1 “添加 camel-script 依赖项” 所示。

例 27.1. 添加 camel-script 依赖项

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静态导入

要在应用程序代码中使用 python（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

内置属性

表 27.1 “Python 属性” 列出使用 Python 时可以访问的内置属性。

表 27.1. Python 属性

属性	类型	值
context	**org.apache.camel.CamelContext**	Camel 上下文
Exchange	**org.apache.camel.Exchange**	当前交换
Request（请求）	**org.apache.camel.Message**	IN 信息
响应	**org.apache.camel.Message**	OUT 消息
属性	**org.apache.camel.builder.script.PropertiesFunction**	通过 **解析** 方法运行，可以更轻松地在脚本中使用属性组件。

在 ENGINE_SCOPE 设置的属性。

示例

例 27.2 “使用 Python 的路由” 显示使用 Python 的路由。

例 27.2. 使用 Python 的路由

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <langauge langauge="python">if request.headers.get('user') = 'admin'</langauge>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

使用属性组件

要从属性组件访问属性值，调用内置 属性属性 上的 解析 方法，如下所示：

```
.setHeader("myHeader").python("properties.resolve(PropKey)")
```

其中 **PropKey** 是您要解析的属性的键，其中键值是 **String** 类型。

有关属性组件的详情，请参阅 **Apache Camel** 组件参考指南 中的 **Properties**。

第 28 章 REF

概述

Ref 表达式语言实际上只是从 Registry 查找自定义表达式 的方法。这在 XML DSL 中使用特别方便。

Ref 语言是 camel-core 的一部分。

静态导入

要在 Java 应用程序代码中使用 Ref 语言，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.language.ref.RefLanguage.ref;
```

XML 示例

例如，splitter 模式可使用 Ref 语言引用自定义表达式，如下所示：

```
<beans ...>
  <bean id="myExpression" class="com.mycompany.MyCustomExpression"/>
  ...
  <camelContext>
    <route>
      <from uri="seda:a"/>
      <split>
        <ref>myExpression</ref>
        <to uri="mock:b"/>
      </split>
    </route>
  </camelContext>
</beans>
```

JAVA 示例

前面的路由也可以在 Java DSL 中实施，如下所示：

```
from("seda:a")
  .split().ref("myExpression")
  .to("seda:b");
```

第 29 章 RUBY(DEPRECATED)

概述

Ruby 是一个动态的开源编程语言，专注于简洁和生产力。它有一个明确的语法，是方便读和易写的。Ruby 支持是 camel-script 模块的一部分。

重要

Apache Camel 中的 Ruby 已被弃用，并将在以后的版本中删除。

添加 SCRIPT 模块

要在路由中使用 Ruby，需要在您的项目中添加有关 camel-script 的依赖项，如 例 29.1 “添加 camel-script 依赖项” 所示。

例 29.1. 添加 camel-script 依赖项

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静态导入

要在应用程序代码中使用 ruby（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

内置属性

表 29.1 “Ruby 属性” 列出使用 Ruby 时可以访问的内置属性。

表 29.1. Ruby 属性

属性	类型	值
context	**org.apache.camel.CamelContext**	Camel 上下文
Exchange	**org.apache.camel.Exchange**	当前交换
Request（请求）	**org.apache.camel.Message**	IN 信息
响应	**org.apache.camel.Message**	OUT 消息
属性	**org.apache.camel.builder.script.PropertiesFunction**	通过 **解析** 方法运行，可以更轻松地在脚本中使用属性组件。

在 ENGINE_SCOPE 设置的属性。

示例

例 29.2 “使用 Ruby 的路由” 显示使用 Ruby 的路由。

例 29.2. 使用 Ruby 的路由

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <langauge langauge="ruby">$request.headers['user'] == 'admin'</langauge>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

使用属性组件

要从属性组件访问属性值，调用内置 属性属性 上的 解析 方法，如下所示：

```
.setHeader("myHeader").ruby("properties.resolve(PropKey)")
```

其中 PropKey 是您要解析的属性的键，其中键值是 String 类型。

有关属性组件的详情，请参阅 Apache Camel 组件参考指南 中的 Properties。

第 30 章 简单语言

摘要

简单的语言是在 Apache Camel 中开发的一种语言，专门用于访问和操作交换对象的各个部分。其语言就像最初创建之时一样简单，它现在包括了一组完整的逻辑运算符和组合。

30.1. JAVA DSL

Java DSL 中的简单表达式

在 Java DSL 中，路由中使用 simple（） 命令有两种样式。您可以将 simple（） 命令作为参数传递给处理器，如下所示：

```
from("seda:order")
  .filter(simple("${in.header.foo}"))
  .to("mock:fooOrders");
```

或者，您可以将 simple（） 命令作为处理器的子使用调用，例如：

```
from("seda:order")
  .filter()
  .simple("${in.header.foo}")
  .to("mock:fooOrders");
```

在字符串中嵌入

如果您要在纯文本字符串中嵌入一个简单表达式，则必须使用占位符语法 ${Expression}。例如，要在字符串中嵌入 in.header.name 表达式：

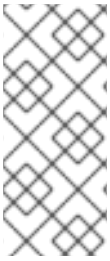
```
simple("Hello ${in.header.name}, how are you?")
```

自定义开始和访问令牌

从 Java，您可以通过调用 changeFunctionStartToken 静态方法和 SimpleLanguage 对象上的 changeFunctionEndToken 静态方法来自定义启动和结束令牌（默认为{ 和 } ）。

例如，您可以在 Java 中将开始和最终令牌更改为 [和]，如下所示：

```
// Java
import org.apache.camel.language.simple.SimpleLanguage;
...
SimpleLanguage.changeFunctionStartToken("[");
SimpleLanguage.changeFunctionEndToken("]");
```



注意

自定义 start 和 end 令牌会影响在类路径上共享相同的 camelcore 库的所有 Apache Camel 应用程序。例如，在 OSGi 服务器中，这可能会影响许多应用程序，而在 Web 应用程序（WAR 文件中）中，它只会影响 Web 应用程序本身。

30.2. XML DSL

XML DSL 中的简单表达式

在 XML DSL 中，您可以通过将表达式放在一个简单元素中来使用 简单的 表达式。例如，定义要根据 foo 标头的内容执行过滤的路由：

```
<route id="simpleExample">
  <from uri="seda:orders"/>
  <filter>
    <simple>${in.header.foo}</simple>
    <to uri="mock:fooOrders"/>
  </filter>
</route>
```

其他占位符语法

有时，如果您启用了 Spring 属性占位符，或者 OSGi 蓝图占位符会发现 ${Expression} 语法 clashes with another property placeholders clashes。在这种情况下，您可以用替代语法消除占位符，$simple{Expression} 用于简单表达式。例如：

```
<simple>Hello $simple{in.header.name}, how are you?</simple>
```

自定义开始和访问令牌

在 XML 配置中，您可以通过覆盖 SimpleLanguage 实例来自定义开始和结束令牌（默认为{ 和 } ）。例如，要将 start 和 end tokens 更改为 [and]，在 XML 配置文件中定义一个新的 SimpleLanguage an，如下所示：

```
<bean id="simple" class="org.apache.camel.language.simple.SimpleLanguage">
  <constructor-arg name="functionStartToken" value="["/>
```

```
  <constructor-arg name="functionEndToken" value="]"/>
</bean>
```

注意

自定义 start 和 end 令牌会影响在类路径上共享相同的 camelcore 库的所有 Apache Camel 应用程序。例如，在 OSGi 服务器中，这可能会影响许多应用程序，而在 Web 应用程序（WAR 文件中）中，它只会影响 Web 应用程序本身。

XML DSL 中的空格和自动修剪

默认情况下，在 XML DSL 中自动修剪简单的表达式前和之后的空格。因此，使用周围空白的表达式：

```
<transform>
  <simple>
    data=${body}
  </simple>
</transform>
```

自动修剪，使其等同于此表达式（不含空格）：

```
<transform>
  <simple>data=${body}</simple>
</transform>
```

如果要在表达式之前或之后包含换行符，您可以显式添加换行符，如下所示：

```
<transform>
  <simple>data=${body}\n</simple>
</transform>
```

或者，您可以通过将 trim 属性设置为 false 来关闭自动修剪，如下所示：

```
<transform trim="false">
  <simple>data=${body}
</simple>
</transform>
```

30.3. 调用外部脚本

概述

可以执行存储在外部资源中的简单脚本，如此处所述。

script 资源的语法

使用以下语法访问作为外部资源存储的简单脚本：

```
resource:Scheme:Location
```

其中 Scheme: 可以是 classpath:、file: 或 http:。

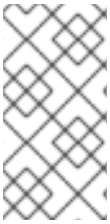
例如，要从 classpath 中读取 mysimple.txt 脚本，

```
simple("resource:classpath:mysimple.txt")
```

30.4. 表达式

概述

简单语言提供了返回消息交换不同部分的各种元素表达式。例如，表达式 （ simple("${header.timeOfDay}") 会从传入消息返回名为 timeOfDay 的标头的内容。



注意

自 Apache Camel 2.9 起，您必须始终 使用占位符语法 ${Expression}，返回变量值。省略包含令牌（${ 和 }）不可预见。

单个变量的内容

您可以根据提供的变量，使用简单的语言来定义字符串表达式。例如，您可以使用表单(in.header.HeaderName)的变量来获取 HeaderName 标头的值，如下所示：

```
simple("${in.header.foo}")
```

嵌入在字符串中的变量

您可以将简单变量嵌入到字符串表达式中，如：

```
simple("Received a message from ${in.header.user} on ${date:in.header.date:yyyyMMdd}.")
```

日期和 bean 变量

还提供了访问交换的所有不同部分的变量（请参阅 表 30.1 “简单语言的变量”），简单语言还提供用于格式日期、日期：模式，以及调用 Bean 方法的变量，bean :bean Ref。例如，您可以使用日期以及 bean 变量，如下所示：

```
simple("Todays date is ${date:now:yyyyMMdd}")
simple("The order type is ${bean:orderService?method=getOrderType}")
```

指定结果类型

您可以明确指定表达式的结果类型。这在将结果类型转换为布尔值或数字类型时特别有用。

在 Java DSL 中，将结果类型指定为 simple（） 的额外参数。例如，要返回整数结果，您可以评估一个简单的表达式，如下所示：

```
...
.setHeader("five", simple("5", Integer.class))
```

在 XML DSL 中，使用 resultType 属性指定结果类型。例如：

```
<setHeader headerName="five">
  <!-- use resultType to indicate that the type should be a java.lang.Integer -->
  <simple resultType="java.lang.Integer">5</simple>
</setHeader>
```

动态标头密钥

在 Camel 2.17 中，如果键的名称为 Simple 语言，则 setHeader和 setExchange 属性允许使用动态标头键。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
      <setHeader
headerName="$simple{type:org.apache.camel.spring.processor.SpringSetPropertyNameDynamicTest$
```

```
TestConstans.EXCHANGE_PROP_TX_FAILED}">
      <simple>${type:java.lang.Boolean.TRUE}</simple>
    </setHeader>
   <to uri="mock:end"/>
  </route>
</camelContext>
```

嵌套表达式

简单的表达式可以是嵌套的、deploy-的、例如：

```
simple("${header.${bean:headerChooser?method=whichHeader}}")
```

访问常数或枚举

您可以使用以下语法访问 bean 的常数或枚举字段：

```
type:ClassName.Field
```

例如，请考虑以下 Java enum 类型：

```
package org.apache.camel.processor;
...
public enum Customer {
    GOLD, SILVER, BRONZE
}
```

您可以访问 客户 enum 字段，如下所示：

```
 from("direct:start")
    .choice()
        .when().simple("${header.customer} ==
          ${type:org.apache.camel.processor.Customer.GOLD}")
            .to("mock:gold")
        .when().simple("${header.customer} ==
          ${type:org.apache.camel.processor.Customer.SILVER}")
            .to("mock:silver")
        .otherwise()
            .to("mock:other");
```

OGNL 表达式

Object Graph 导航语言(OGNL)是以类似链的方式调用 Bean 方法的表示法。如果消息正文包含 Java

bean，您可以使用 OGNL 表示法轻松访问其 bean 属性。例如，如果消息正文是具有 getAddress（） accessor 的 Java 对象，您可以访问 Address 对象和 Address 对象的属性，如下所示：

```
simple("${body.address}")
simple("${body.address.street}")
simple("${body.address.zip}")
simple("${body.address.city}")
```

其中表示法 ${body.address.street} 是 ${body.getAddress.getStreet} 的简写。

OGNL null-safe operator

您可以使用 null-safe 运算符 ?。 如果正文 没有 地址，则可以避免遇到 null-pointer 异常。例如：

```
simple("${body?.address?.street}")
```

如果正文是 java.util.Map 类型，您可以使用以下表示法在 map 中查找值 foo ：

```
simple("${body[foo]?.name}")
```

OGNL list 元素访问权限

您也可以使用方括号表示法 [k] 来访问列表的元素。例如：

```
simple("${body.address.lines[0]}")
simple("${body.address.lines[1]}")
simple("${body.address.lines[2]}")
```

last 关键字返回列表的最后一个元素的索引。例如，您可以访问列表 的第二个最后一个 元素，如下所示：

```
simple("${body.address.lines[last-1]}")
```

您可以使用 大小 方法查询列表的大小，如下所示：

```
simple("${body.address.lines.size}")
```

OGNL 阵列长度访问

您可以通过 长度 方法访问 Java 阵列的长度，如下所示：

```
String[] lines = new String[]{"foo", "bar", "cat"};
exchange.getIn().setBody(lines);

simple("There are ${body.length} lines")
```

30.5. PREDICATES

概述

您可以根据测试表达式来构造 predicates，以实现相等性。例如： predicate, simple("${header.timeOfDay} == '14:30'"), 测试传入消息中的 timeOfDay 标头等于 14:30。

另外，每当将 resultType 指定为布尔值时，表达式会被评估为 predicate 而不是一个表达式。这允许这些表达式使用 predicate 语法。

语法

您还可以使用简单的 predicates 测试交换（headers、消息正文等）的不同部分。简单 predicates 具有以下通用语法：

```
${LHSVariable} Op RHSValue
```

在左侧的 LHSVariable 中，变量是 表 30.1 “简单语言的变量” 中显示的变量之一，右侧是 RHSValue 的值：

- 另一个变量 ${RHSVariable}.
- 字符串文字，用单引号 ' 括起。
- 一个数字常量，用单引号括起来，'。
- null 对象 null。

简单语言始终会尝试将 RHS 值转换为 LHS 值的类型。

注意

尽管简单的语言将尝试转换 RHS，但根据 LHS 可能需要在进行比较前，可能需要转换为相应的类型。

例子

例如，您可以执行简单字符串比较和数字比较，如下所示：

```
simple("${in.header.user} == 'john'")

simple("${in.header.number} > '100'")  // String literal can be converted to integer
```

您可以测试左侧是用逗号分开的列表的成员，如下所示：

```
simple("${in.header.type} in 'gold,silver'")
```

您可以测试左侧是否与正则表达式匹配，如下所示：

```
simple("${in.header.number} regex '\d{4}'")
```

您可以使用 **is operator** 测试左侧的类型，如下所示：

```
simple("${in.header.type} is 'java.lang.String'")
simple("${in.header.type} is 'String'") // You can abbreviate java.lang. types
```

您可以测试左侧是否在指定数字范围内（范围是 **inclusive**），如下所示：

```
simple("${in.header.number} range '100..199'")
```

Conjunctions

您还可以使用逻辑联合、& amp;& 和 || 将 predicates 组合。

例如，以下是一个使用 & amp;& 组合（逻辑和）的表达式：

```
simple("${in.header.title} contains 'Camel' && ${in.header.type} == 'gold'")
```

以下是使用 || 组合（逻辑包或包）的表达式：

```
simple("${in.header.title} contains 'Camel' || ${in.header.type} == 'gold'")
```

30.6. 变量参考

变量表

表 30.1 “简单语言的变量” 显示简单语言支持的所有变量。

表 30.1. 简单语言的变量

变量	类型	描述
camelContext	**对象**	Camel 上下文。**支持 OGNL 表达式。**
camelId	**字符串**	Camel 上下文的 ID 值。
exchangeId	**字符串**	交换的 ID 值。
id	**字符串**	**In** message ID 值。
正文（body）	**对象**	**In** message body。**支持 OGNL 表达式。**
in.body	**对象**	**In** message body。**支持 OGNL 表达式。**
out.body	**对象**	**Out** 消息正文。
bodyAs(*Type*)	*类型*	**In** 消息正文，转换为指定的类型。所有类型（*类型*）必须使用其完全限定 Java 名称进行指定，但类型除外：**byte[]**、**String**、**Integer** 和 **Long**。转换的正文可以是 null。

变量	类型	描述
mandatoryBodyAs(*Type*)	*类型*	In 消息正文，转换为指定的类型。所有类型（*类型*）必须使用其完全限定 Java 名称进行指定，但类型除外：**byte[]**、**String**、**Integer** 和 **Long**。转换的正文应该是非null。
header.*HeaderName*	对象	In 消息的 *HeaderName* 标头。**支持 OGNL 表达式。**
header[*HeaderName***]**	对象	In message 的 *HeaderName* 标头（alternative 语法）。
标头。*HeaderName*	对象	In 消息的 *HeaderName* 标头。
headers[*HeaderName***]**	对象	In message 的 *HeaderName* 标头（alternative 语法）。
in.header.*HeaderName*	对象	In 消息的 *HeaderName* 标头。**支持 OGNL 表达式。**
in.header[*HeaderName***]**	对象	In message 的 *HeaderName* 标头（alternative 语法）。
in.headers.*HeaderName*	对象	In 消息的 *HeaderName* 标头。**支持 OGNL 表达式。**
in.headers[*HeaderName***]**	对象	In message 的 *HeaderName* 标头（alternative 语法）。
out.header.*HeaderName*	对象	Out 消息的 *HeaderName* 标头。
out.header[*HeaderName***]**	对象	Out 消息的 *HeaderName* 标头（alternative 语法）。
out.headers.*HeaderName*	对象	Out 消息的 *HeaderName* 标头。
out.headers[*HeaderName***]**	对象	Out 消息的 *HeaderName* 标头（alternative 语法）。
headerAs(*Key*,*Type*)	*类型*	*Key* 标头，转换为指定类型。所有类型（*类型*）必须使用其完全限定 Java 名称进行指定，但类型除外：**byte[]**、**String**、**Integer** 和 **Long**。转换的值可以是 null。

变量	类型	描述
标头	**map**	所有 In 标头（作为 **java.util.Map** 类型）。
in.headers	**map**	所有 In 标头（作为 **java.util.Map** 类型）。
exchangeProperty.*PropertyName*	**对象**	交换上的 *PropertyName* 属性。
exchangeProperty[*PropertyName***]**	**对象**	交换上的 *PropertyName* 属性（临时语法）。
exchangeProperty.*PropertyName*.OGNL	**对象**	交换上的 *PropertyName* 属性，并使用 Camel OGNL 表达式调用其值。
sys.*SysPropertyName*	**字符串**	*SysPropertyName* Java 系统属性。
sysenv.*SysEnvVar*	**字符串**	*SysEnvVar* 系统环境变量。
例外	**字符串**	来自 **Exchange.getException（）** 异常对象；如果此值为空，来自 **Exchange.EXCEPTION_CAUGHT** 属性的 caught 除外，否则为空。**支持 OGNL 表达式。**
exception.message	**字符串**	如果在交换上设置了一个异常，则返回 **Exception.getMessage（）** 的值；否则返回 **null**。
exception.stacktrace	**字符串**	如果交换中设置了异常，则返回 **Exception.getStackTrace（）** 的值；否则返回 **null**。**注意：** 简单语言首先尝试从 **Exchange.getException（）** 检索异常。如果未设置该属性，它会通过调用 **Exchange.getProperty(Exchange.CAUGHT_EXCEPTION)** 检查发现异常。

变量	类型	描述
日期： *命令*:*模式*	**字符串**	使用 java.text.SimpleDateFormat 模式格式的日期。以下命令被支持：**现在**，对于当前日期和时间；**header.***HeaderName*，或 **in.header.***HeaderName*，**在** In 消息的 *HeaderName* 标头中使用 java.util.Date 对象; **out.header.***HeaderName* 以使用 java.util.Date 对象到 **Out** 消息中的 *HeaderName* 标头；
bean:*beanID.Method*	**对象**	在引用的 Bean 上调用一个方法，**并返回方法调用 的结果**。要指定方法名称，您可以使用 ***beanID.Method*** 语法；或者，您可以使用 ***beanID*?method=*method* Name**语法。
Ref:*beanID*	**对象**	在注册表中查找带有 ID *beanID* 的 bean，**并返回对 bean 本身的引用**。例如，如果您使用 splitter EIP，您可以使用此变量引用实现拆分算法的 bean。
属性：*key*	**字符串**	*Key* 属性占位符 的值。
属性：*位置* ：*密钥*	**字符串**	*Key* 属性占位符的值，其中属性文件的位置由 *Location* 指定。
threadName	**字符串**	当前线程的名称。
routeId	**字符串**	返回交换要通过的当前路由的 ID。
类型：*名称*[.*Field*]	**对象**	通过其 Fully-Qualified-Name(FQN)引用类型或字段。要引用字段，请附加 **.*Field***。例如，您可以将 **Exchange** 类中的 **FILE_NAME** constant 字段引用为 **type:org.apache.camel.Exchange.FILE_NAME**

变量	类型	描述
collate(group)	**list**	在 Camel 2.17 中，collate 函数将迭代邮件正文，并将数据分组到特定大小的子列表中。您可以使用 和 Splitter EIP 将消息正文和组分组，或者将子消息分组到 N 子列表中。
skip(number)	**ITerator**	跳过函数会迭代消息正文，并跳过第一个项目数。这可以与 Splitter EIP 分割消息正文，并跳过第 N 个项目数。

30.7. OPERATOR 参考

二进制运算符

*简单语言 **predicates** 的二进制运算符显示在 表 30.2 “简单语言的二进制 Operator” 中。*

表 30.2. 简单语言的二进制 Operator

Operator	描述
==	等于。
=~	等于忽略大小写。比较字符串值时忽略大小写。
>	大于.
>=	大于或等于。
<	小于.
⇐	小于或等于。
!=	不等于.
contains	测试 LHS 字符串是否包含 RHS 字符串。
不包含	测试 LHS 字符串 **不包含** RHS 字符串。
regex	测试 LHS 字符串是否与 RHS 正则表达式匹配。
不是正则表达式	测试 LHS 字符串与 RHS 正则表达式 **不匹配**。

Operator	描述
in	测试 LHS 字符串是否出现在 RHS comma- separated 列表中。
not in	测试 LHS 字符串是否 **没有出现在** RHS comma- separated 列表中。
is	测试 LHS 是否是 RHS Java 类型的实例（使用 Java **实例** ）。
不是	测试 LHS **不是** RHS Java 类型的实例（使用 Java **实例** ）。
range	测试 LHS 编号是否在 RHS 范围内（其中范围具有格式，"*min*...*max*'）。
不是范围	测试 LHS 编号在 RHS 范围内的位置是否不同（其中范围具有格式，"*min*...*max*'）。
从开始	Camel 2.18 中的新功能.测试 LHS 字符串是否以 RHS 字符串开头。
结束	Camel 2.18 中的新功能.测试 LHS 字符串是否以 RHS 字符串结尾。

元运算符和字符转义

*简单语言 **predicates** 的二进制运算符显示在 表 30.3 “用于简单语言的 unary Operator” 中。*

表 30.3. 用于简单语言的 unary Operator

Operator	描述
++	按 1 递增数字.
--	数字减少 1.
\n	换行字符。
\r	回车符返回字符。
\t	标签字符。

Operator	描述
\	（过时） Since Camel 版本 2.11，不支持反斜杠转义字符。

组合 predicates

表 30.4 “Simple Language Predicates 的 Conjunctions” 所示的联合可用于组合两个或者多个简单语言 predicates。

表 30.4. Simple Language Predicates 的 Conjunctions

Operator	描述
&&	将两个 predicates 与逻辑 **和** 组合。
||	将两个 predicates 与逻辑 inclusive **或** 组合。
和	**弃用**。使用 **&&** 替代。
或者	**弃用**。使用 || 替代。

第 31 章 SPEL

概述

Spring Expression Language(SpEL) 是随 Spring 3 提供的对象图形导航语言，可用于在路由中构造 predicates 和表达式。SpEL 的显著功能是如何从 registry 访问 Bean 的简易功能。

语法

SpEL 表达式必须使用占位符语法 #{SpelExpression} 以便将其嵌入到纯文本字符串中（换句话说，SpEL 已启用表达式模板）。

SpEL 也可以在 registry 中查找 Bean（通常是 Spring registry），使用 @BeanID 语法。例如，给定一个带有 ID、headerUtils 和 方法的 bean（它计算当前消息中的标头数量），您可以在 SpEL predicate 中使用 headerUtils bean，如下所示：

```
#{@headerUtils.count > 4}
```

添加 SPEL 软件包

要在路由中使用 SpEL，您需要在您的项目中添加对 camel-spring 的依赖，如 例 31.1 “添加 camel-spring 依赖项” 所示。

例 31.1. 添加 camel-spring 依赖项

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-7_10_0-00018-redhat-00001</camel-version>
  ...
</properties>

<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

变量

表 31.1 “SpEL 变量” 列出在使用 SpEL 时可访问的变量。

表 31.1. SpEL 变量

变量	类型	描述
这	**Exchange**	当前交换是 root 对象。
Exchange	**Exchange**	当前交换。
exchangeId	**字符串**	当前交换的 ID。
例外	**Throwable**	交换例外（如果有）。
fault	**消息**	故障消息（若有）。
Request（请求）	**消息**	交换的 In 消息。
响应	**消息**	交换的 Out 消息（若有）。
属性	**map**	交换属性。
属性（*名称*）	对象	交换属性由 *名称* 键。
property(*Name*,*Type*)	类型	由 *名称* 键的交换属性转换为类型，*Type*。

XML 示例

例如，要仅选择其 country 标头拥有值 USA 的消息，您可以使用以下 SpEL 表达式：

```
<route>
  <from uri="SourceURL"/>
  <filter>
    <spel>#{request.headers['Country'] == 'USA'}</spel>
    <to uri="TargetURL"/>
  </filter>
</route>
```

JAVA 示例

您可以在 Java DSL 中定义相同的路由，如下所示：

```
from("SourceURL")
  .filter().spel("#{request.headers['Country'] == 'USA'}")
  .to("TargetURL");
```

以下示例演示了如何在纯文本字符串中嵌入 SpEL 表达式：

```
from("SourceURL")
  .setBody(spel("Hello #{request.body}! What a beautiful #{request.headers['dayOrNight']}"))
  .to("TargetURL");
```

第 32 章 XPATH 语言

摘要

当处理 XML 消息时，XPath 语言允许您选择消息的一部分，方法是通过指定适用于消息的 Document Object Model(DOM)的 XPath 表达式。您还可以定义 XPath predicates 来测试元素或属性的内容。

32.1. JAVA DSL

基本表达式

您可以使用 xpath("Expression") 来评估当前交换中的 XPath 表达式（其中 XPath 表达式被应用于当前消息的正文）。xpath（） 表达式的结果是一个 XML 节点（如果有多个节点匹配，则为节点集）。

例如，要从当前 In 消息正文中提取 /person/name 元素的内容，并使用它来设置名为 user 的标头，您可以定义一个类似如下的路由：

```
from("queue:foo")
    .setHeader("user", xpath("/person/name/text()"))
    .to("direct:tie");
```

除了将 xpath（） 指定为 setHeader（） 的参数外，您可以使用 fluent builder xpath（） command>_<-对 example:

```
from("queue:foo")
    .setHeader("user").xpath("/person/name/text()")
    .to("direct:tie");
```

如果要将结果转换为特定的类型，请将结果类型指定为 xpath（） 的第二个参数。例如，要明确指定结果类型是 String ：

```
xpath("/person/name/text()", String.class)
```

命名空间

通常，XML 元素属于一个 schema，它通过命名空间 URI 进行标识。当处理诸如此类文档时，需要将命名空间 URI 与前缀关联，以便您可以在 XPath 表达式中明确识别元素名称。Apache Camel 提供帮助程序类 org.apache.camel.builder.xml.Namespaces，用于定义命名空间和前缀之间的关联。

例如，要将前缀 cust 与命名空间 http://acme.com/customer/record 关联，然后提取元素的内容 /cust:person/cust:name，您可以定义一个类似如下的路由：

```
import org.apache.camel.builder.xml.Namespaces;
...
Namespaces ns = new Namespaces("cust", "http://acme.com/customer/record");

from("queue:foo")
    .setHeader("user", xpath("/cust:person/cust:name/text()", ns))
    .to("direct:tie");
```

您可以通过传递 Namespaces 对象 ns 作为额外参数，将命名空间定义提供给 xpath（） 表达式构建器。如果需要定义多个命名空间，请使用 Namespace.add（） 方法，如下所示：

```
import org.apache.camel.builder.xml.Namespaces;
...
Namespaces ns = new Namespaces("cust", "http://acme.com/customer/record");
ns.add("inv", "http://acme.com/invoice");
ns.add("xsi", "http://www.w3.org/2001/XMLSchema-instance");
```

如果需要指定结果类型 并定义 命名空间，您可以使用三参数形式 xpath（），如下所示：

```
xpath("/person/name/text()", String.class, ns)
```

审计命名空间

使用 XPath 表达式时，可能会出现最频繁的问题之一是命名空间在传入消息中出现不匹配的情况和 XPath 表达式中使用的命名空间。为了帮助您排除此类问题，XPath 语言支持 选项将所有命名空间从所有传入消息转储到系统日志中。

要在 INFO 日志级别启用命名空间日志记录，请在 Java DSL 中启用 logNamespaces 选项，如下所示：

```
xpath("/foo:person/@id", String.class).logNamespaces()
```

或者，您也可以将日志记录系统配置为在 org.apache.camel.builder.xml.XPathBuilder 日志记录器上启用 MERGE 级别日志记录。

启用命名空间日志记录后，您会看到如下日志消息，以了解每个处理的消息：

```
2012-01-16 13:23:45,878 [stSaxonWithFlag] INFO  XPathBuilder  -
```

```
Namespaces discovered in message: {xmlns:a=[http://apache.org/camel],
DEFAULT=[http://apache.org/default],
xmlns:b=[http://apache.org/camelA, http://apache.org/camelB]}
```

32.2. XML DSL

基本表达式

要评估 XML DSL 中的 XPath 表达式，请将 XPath 表达式放在 xpath 元素中。XPath 表达式应用于当前 In 消息的正文，并返回 XML 节点（或节点集）。通常，返回的 XML 节点会自动转换为字符串。

例如，要从当前 In 消息正文中提取 /person/name 元素的内容，并使用它来设置名为 user 的标头，您可以定义一个类似如下的路由：

```
<beans ...>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="queue:foo"/>
      <setHeader headerName="user">
        <xpath>/person/name/text()</xpath>
      </setHeader>
      <to uri="direct:tie"/>
    </route>
  </camelContext>

</beans>
```

如果要将结果转换为特定类型，请通过将 resultType 属性设置为 Java 类型名称（其中必须指定完全限定类型名称）来指定结果类型。例如，要明确指定结果类型是 java.lang.String （您可以在此处省略 java.lang. 前缀）：

```
<xpath resultType="String">/person/name/text()</xpath>
```

命名空间

当处理元素属于一个或多个 XML 模式的文档时，通常需要将命名空间 URI 与前缀关联，以便您可以识别 XPath 表达式中的元素名称。可以使用标准 XML 机制将前缀与命名空间 URI 关联。也就是说，您可以设置类似如下的属性：xmlns:Prefix="NamespaceURI"。

例如，要将前缀 cust 与命名空间 http://acme.com/customer/record 关联，然后提取元素的内容 /cust:person/cust:name，您可以定义一个类似如下的路由：

```
<beans ...>

  <camelContext xmlns="http://camel.apache.org/schema/spring"
                xmlns:cust="http://acme.com/customer/record" >
    <route>
      <from uri="queue:foo"/>
      <setHeader headerName="user">
        <xpath>/cust:person/cust:name/text()</xpath>
      </setHeader>
      <to uri="direct:tie"/>
    </route>
  </camelContext>

</beans>
```

审计命名空间

使用 XPath 表达式时，可能会出现最频繁的问题之一是命名空间在传入消息中出现不匹配的情况和 XPath 表达式中使用的命名空间。为了帮助您排除此类问题，XPath 语言支持 选项将所有命名空间从所有传入消息转储到系统日志中。

要在 INFO 日志级别启用命名空间日志记录，请在 XML DSL 中启用 logNamespaces 选项，如下所示：

```
<xpath logNamespaces="true" resultType="String">/foo:person/@id</xpath>
```

或者，您也可以将日志记录系统配置为在 org.apache.camel.builder.xml.XPathBuilder 日志记录器上启用 MERGE 级别日志记录。

启用命名空间日志记录后，您会看到如下日志消息，以了解每个处理的消息：

```
2012-01-16 13:23:45,878 [stSaxonWithFlag] INFO  XPathBuilder  -
Namespaces discovered in message: {xmlns:a=[http://apache.org/camel],
DEFAULT=[http://apache.org/default],
xmlns:b=[http://apache.org/camelA, http://apache.org/camelB]}
```

32.3. XPATH INJECTION

参数绑定注解

在使用 Apache Camel bean 集成时，在 Java bean 上调用方法时，您可以使用 @XPath 注释从交换中提取值，并将它绑定到 method 参数。

例如，请考虑以下路由片段，它调用 AccountService 对象的贡献方法：

```
from("queue:payments")
    .beanRef("accountService","credit")
    ...
```

贡献 方法使用参数绑定注解从邮件正文中提取相关数据并将其注入其参数，如下所示：

```
public class AccountService {
    ...
    public void credit(
            @XPath("/transaction/transfer/receiver/text()") String name,
            @XPath("/transaction/transfer/amount/text()") String amount
            )
    {
        ...
    }
    ...
}
```

如需更多信息，请参阅客户门户网站的 Apache Camel 开发指南中的 Bean 集成。

命名空间

表 32.1 “@XPath 预定义的命名空间” 显示 XPath 预定义的命名空间。您可以在 @XPath 注释中显示的 XPath 表达式中使用这些命名空间前缀。

表 32.1. @XPath 预定义的命名空间

命名空间 URI	prefix
http://www.w3.org/2001/XMLSchema	xsd
http://www.w3.org/2003/05/soap-envelope	SOAP

自定义命名空间

您可以使用 @NamespacePrefix 注释来定义自定义 XML 命名空间。调用 @NamespacePrefix 注释，以初始化 @XPath 注释 的命名空间 参数。然后，可以在 @XPath 注释的表达式值中使用 @NamespacePrefix 定义的命名空间。

例如，要将前缀 ex 与自定义命名空间 http://fusesource.com/examples 关联，调用 @XPath 注释，

如下所示：

```
public class AccountService {
  ...
  public void credit(
    @XPath(
      value = "/ex:transaction/ex:transfer/ex:receiver/text()",
      namespaces = @NamespacePrefix( prefix = "ex", uri = "http://fusesource.com/examples"
      )
    ) String name,
    @XPath(
      value = "/ex:transaction/ex:transfer/ex:amount/text()",
      namespaces = @NamespacePrefix( prefix = "ex", uri = "http://fusesource.com/examples"
      )
    ) String amount,
  )
  {
    ...
  }
  ...
}
```

32.4. XPATH BUILDER

概述

org.apache.camel.builder.xml.XPathBuilder 类允许您独立评估 **XPath** 表达式。也就是说，如果您拥有来自任何源的 **XML** 片段，您可以使用 **XPathBuilder** 来评估 **XML** 片段中的 **XPath** 表达式。

匹配表达式

使用 **matches**（） 方法检查一个或多个 **XML** 节点是否可以找到与给定 **XPath** 表达式匹配的 **XML** 节点。使用 **XPathBuilder** 匹配 **XPath** 表达式的基本语法如下：

```
boolean matches = XPathBuilder
                .xpath("Expression")
                .matches(CamelContext, "XMLString");
```

如果至少有一个节点与表达式匹配，则给定表达式、表达式 会根据 **XML** 片段、**XMLString** 和结果为 **true** 进行评估。例如，以下示例返回 **true**，因为 **XPath** 表达式在 **xyz** 属性中找到匹配项。

```
boolean matches = XPathBuilder
                .xpath("/foo/bar/@xyz")
                .matches(getContext(), "<foo><bar xyz='cheese'/></foo>"));
```

评估表达式

使用 evaluate（） 方法返回与给定 XPath 表达式匹配的第一个节点的内容。使用 XPathBuilder 评估 XPath 表达式的基本语法如下：

```
String nodeValue = XPathBuilder
                .xpath("Expression")
                .evaluate(CamelContext, "XMLString");
```

您也可以通过传递所需的类型作为第二个参数来指定结果类型，以评估（） targetNamespaces-InventoryServicefor 示例：

```
String name = XPathBuilder
                .xpath("foo/bar")
                .evaluate(context, "<foo><bar>cheese</bar></foo>", String.class);
Integer number = XPathBuilder
                .xpath("foo/bar")
                .evaluate(context, "<foo><bar>123</bar></foo>", Integer.class);
Boolean bool = XPathBuilder
                .xpath("foo/bar")
                .evaluate(context, "<foo><bar>true</bar></foo>", Boolean.class);
```

32.5. 启用 SAXON

先决条件

使用 Saxon 解析器的先决条件是，您将依赖 camel-saxon 工件（请将这个依赖项添加到您的 Maven POM 中，如果使用 Maven，或者添加 camel-saxon-2.23.2.fuse-7_10_0-00018-redhat-00001.jar 文件到您的类路径中）。

在 Java DSL 中使用 Saxon parser

在 Java DSL 中，启用 Saxon 解析器的最简单方法是调用 saxon（） fluent builder 方法。例如，您可以调用 Saxon 解析器，如下例所示：

```
// Java
// create a builder to evaluate the xpath using saxon
XPathBuilder builder = XPathBuilder.xpath("tokenize(/foo/bar, '_')[2]").saxon();

// evaluate as a String result
String result = builder.evaluate(context, "<foo><bar>abc_def_ghi</bar></foo>");
```

在 XML DSL 中使用 Saxon parser

在 XML DSL 中，启用 Saxon parser 最简单的方法是在 xpath 元素中将 saxon 属性设置为 true。例如，您可以调用 Saxon 解析器，如下例所示：

```
<xpath saxon="true" resultType="java.lang.String">current-dateTime()</xpath>
```

使用 Saxon 编程

如果要在应用程序代码中使用 Saxon XML 解析程序，您可以使用以下代码明确创建 Saxon 转换器实例：

```
// Java
import javax.xml.transform.TransformerFactory;
import net.sf.saxon.TransformerFactoryImpl;
...
TransformerFactory saxonFactory = new net.sf.saxon.TransformerFactoryImpl();
```

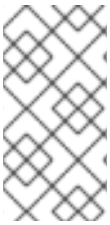
另一方面，如果您更喜欢使用通用 JAXP API 创建转换器工厂实例，您必须首先在 ESBInstall/etc/system.properties 文件中设置 javax.xml.transform.TransformerFactory 属性，如下所示：

```
javax.xml.transform.TransformerFactory=net.sf.saxon.TransformerFactoryImpl
```

然后，您可以使用通用 JAXP API 来实例化 Saxon 工厂，如下所示：

```
// Java
import javax.xml.transform.TransformerFactory;
...
TransformerFactory factory = TransformerFactory.newInstance();
```

如果您的应用程序依赖于任何使用 Saxon 的第三方库，则可能需要使用第二个通用方法。



注意

Saxon 库必须作为 OSGi 捆绑包安装，即 net.sf.saxon/saxon9he （通常为默认安装）。在 7.1 之前的 Fuse ESB 版本中，无法使用通用 JAXP API 加载 Saxon。

32.6. 表达式

结果类型

默认情况下，XPath 表达式会返回一个或多个 XML 节点的列表，即 org.w3c.dom.NodeList 类型。但是，您可以使用类型转换机制将结果转换为不同的类型。在 Java DSL 中，您可以在 xpath（） 命令的第二个参数中指定结果类型。例如，要将 XPath 表达式的结果返回为 String：

```
xpath("/person/name/text()", String.class)
```

在 XML DSL 中，您可以在 resultType 属性中指定结果类型，如下所示：

```
<xpath resultType="java.lang.String">/person/name/text()</xpath>
```

位置路径中的模式

您可以在 XPath 位置路径中使用以下模式：

/people/person

基本位置路径指定特定元素的嵌套位置。也就是说，前面的位置路径与以下 XML 片段中的 person 元素匹配：

```
<people>
  <person>...</person>
</people>
```

请注意，如果 一个人元素中有多个人元素，则此基本模式可以匹配 多个 nodes>_<->_& lt;。

/name/text()

如果您只想访问元素内的 文本，请将 /text（） 附加到位置路径，否则节点会包含该元素的开始和最终标签（当您将节点转换为字符串时，这些标签将被包含）。

/person/telephone/@isDayTime

要选择属性的值 AttributeName，可使用语法 @AttributeName。例如，在应用到以下 XML 片段时，前面的位置路径会返回 true ：

```
<person>
  <telephone isDayTime="true">1234567890</telephone>
</person>
```

*

与指定范围中的所有元素匹配的通配符。例如，/body/person/* 与 个人 的所有子元素匹配。

@*

匹配匹配元素的所有属性的通配符。例如： /person/name/@* 匹配每个匹配 name 元素的所有属性。

//

匹配每个嵌套级别的位置路径。例如，//name 模式与以下 XML 片段中突出显示的每个 名称 元素匹配：

```
<invoice>
  <person>
    <name .../>
  </person>
</invoice>
<person>
  <name .../>
</person>
<name .../>
```

..

选择当前上下文节点的父级。通常不适用于 Apache Camel XPath 语言，因为当前上下文节点是文档根目录，而这没有父项。

node()

匹配任何种类的节点。

text()

匹配文本节点。

comment()

匹配注释节点。

processing-instruction()

匹配处理结构节点。

predicate 过滤器

您可以通过在方括号中附加 predicate [Predicate] 来过滤匹配路径的节点集合。例如，您可以通过将

[N] 到位置路径，从匹配项列表中选择Nth 节点。以下表达式选择第一个匹配的 person 元素：

```
/people/person[1]
```

以下表达式选择 second-last person 元素：

```
/people/person[last()-1]
```

您可以测试属性值，以选择具有特定属性值的元素。以下表达式选择 名称 元素，其 surname 属性可以是 Strachan 或 Davies：

```
/person/name[@surname="Strachan" or @surname="Davies"]
```

您可以使用任何组合（或,not（） ）组合 predicate 表达式，您可以使用比较器、= !=、>=、& gt;=、& lt;、表示(practice)替代 less-than 符号的表达式。 您还可以使用 predicate 过滤器中的 XPath 功能。

Axes

当您考虑 XML 文档的结构时，根元素包含一系列子项，其中一些子元素包含更多子项等。通过这种方式，嵌套元素的位置由子关系链接，整个 XML 文档具有 树结构。现在，如果您选择此元素树中的特定节点（将其命名为 节点），您可能想引用与所选节点相关的树的不同部分。例如，您可能想要引用上下文节点的子项、上下文节点的父项，或与上下文节点共享相同父节点（同级节点）的所有节点上。

XPath axis 用于指定节点匹配范围，相对于当前上下文节点，将搜索限制为节点树的特定部分。axis 作为一个前缀附加到您要匹配的节点名称的前缀，其语法为 AxisType::MatchingNode。例如，您可以使用 child:: axis 搜索当前上下文节点的子项，如下所示：

```
/invoice/items/child::item
```

child::item 的上下文节点是由路径 /invoice/ items 选择的 items 元素。子：: axis 将搜索范围限制为上下文节点的子项，以便 child:: item 与名为 items 的项目 的子项匹配。实际上，子级：: axis 是默认的 axis，因此前面的示例可以等效于：

```
/invoice/items/item
```

但是，还有其他一些的 axes（全部），您已被缩写为缩写形式的一些内容 ：@ 是 属性 的缩写：而 // 是 descend -or-self:: 的缩写。如下为 axes 的完整列表（有关详细信息请参考下面的信息）：

- *ancestor*
- *监管者*
- *attribute*
- *child*
- *descendant*
- *descendant-or-self*
- *关注*
- *following-sibling*
- *namespace*
- *父*
- *前面的*
- *前面的同级*
- *self*

Functions

XPath 提供一组较小的标准函数，在评估 predicates 时很有用。例如，要从节点集中选择最后一个匹

配节点，您可以使用 last（）函数，该函数返回节点集合中最后一个节点的索引，如下所示：

```
/people/person[last()]
```

在前面的例子中，选择序列中的最后一个 人 元素（按文档顺序）。

有关 XPath 提供的所有功能的详情，请参考下面的参考。

参考

有关 XPath grammar 的完整详情，请参阅 XML 路径语言 Version 1.0 规格。

32.7. PREDICATES

基本 predicates

您可以在 Java DSL 中，或者在 predicate 期望的上下文中使用 xpath，作为 filter（） 处理器的参数，或作为 when（） 子句的参数。

例如，以下路由过滤传入的消息，允许一条信息传递，仅当 /person/city 元素包含值 伦敦 时：

```
from("direct:tie")
   .filter().xpath("/person/city = 'London'").to("file:target/messages/uk");
```

以下路由评估 when（） 子句中的 XPath predicate：

```
from("direct:tie")
   .choice()
      .when(xpath("/person/city = 'London'")).to("file:target/messages/uk")
      .otherwise().to("file:target/messages/others");
```

XPath predicate operator

XPath 语言支持标准 XPath predicate 运算符，如 表 32.2 “XPath 语言的 Operator” 所示。

表 32.2. XPath 语言的 Operator

Operator	描述
=	等于。
!=	不等于.
>	大于.
>=	大于或等于。
<	小于.
⇐	小于或等于。
和	将两个 predicates 与逻辑 **和** 组合。
或者	将两个 predicates 与逻辑 inclusive **或** 组合。
not()	negate predicate 参数.

32.8. 使用变量和函数

评估路由中的变量

当评估路由中的 XPath 表达式时，您可以使用 XPath 变量访问当前交换的内容，以及 O/S 环境变量和 Java 系统属性。如果通过 XML 命名空间访问变量，则访问变量值 的语法为 $VarName 或 $Prefix：VarName。

例如，您可以将 In 消息的正文访问为 $in:body，In message 的标题值为 $in:HeaderName。O/S 环境变量可作为 $env:EnvVar 和 Java 系统属性进行访问，可作为 $system:SysVar 进行访问。

在以下示例中，第一个路由提取 /person/city 元素的值，并将其插入到 城市 标头中。第二个路由过滤器使用 XPath 表达式 $in:city = 'London' 进行交换，其中 $in:city 变量被 城市 标头的值替代。

```
from("file:src/data?noop=true")
    .setHeader("city").xpath("/person/city/text()")
    .to("direct:tie");

from("direct:tie")
    .filter().xpath("$in:city = 'London'").to("file:target/messages/uk");
```

评估路由中的功能

除了标准 XPath 功能外，XPath 语言还定义了其他功能。这些附加功能（在 表 32.4 “XPath 自定义功能”中列出的）可用于访问底层交换，用于评估简单表达式或在 Apache Camel 属性占位符组件中查找属性。

例如，以下示例使用 in:header（） 函数和 in:body（） 函数来访问底层交换中的头条和正文：

```
from("direct:start").choice()
  .when().xpath("in:header('foo') = 'bar'").to("mock:x")
  .when().xpath("in:body() = '<two/>'").to("mock:y")
  .otherwise().to("mock:z");
```

注意这些功能和对应在 HeaderName或 in: body 变量之间的相似性。功能有稍有不同的语法：在：header('HeaderName') 中，而不是 in:HeaderName; 和 in:body（） 而不是 in:body （）。

评估 XPathBuilder 中的变量

您还可以使用使用 XPathBuilder 类评估的表达式中的变量。在这种情况下，您不能使用 $in:body 或 $in:HeaderName 等变量，因为没有要评估的交换对象。但是，您可以使用 内嵌 定义的变量（名称、值） fluent 构建器方法。

例如，以下 XPathBuilder 构造将评估 $test 变量，该变量被定义为 伦敦 的值：

```
String var = XPathBuilder.xpath("$test")
             .variable("test", "London")
             .evaluate(getContext(), "<name>foo</name>");
```

请注意，以这种方式定义的变量将自动输入到全局命名空间中（例如，变量 $test 不使用前缀）。

32.9. 变量命名空间

命名空间表

表 32.3 “XPath 变量值” 显示与各种命名空间前缀关联的命名空间 URI。

表 32.3. XPath 变量值

命名空间 URI	prefix	描述
http://camel.apache.org/schema/spring	无	默认命名空间（与没有命名空间前缀的变量关联）。
http://camel.apache.org/xml/in/	in	用于引用当前交换的标题或 正文。
http://camel.apache.org/xml/out/	out	用于引用当前交换的 Out 消息的标头或正文。
http://camel.apache.org/xml/functions/	功能	用于引用一些自定义功能。
http://camel.apache.org/xml/variables/environment-variables	env	用于引用 O/S 环境变量。
http://camel.apache.org/xml/variables/system-properties	system	用于引用 Java 系统属性。
http://camel.apache.org/xml/variables/exchange-property	未定义	用于引用交换属性。您必须为这个命名空间定义您自己的前缀。

32.10. 功能参考

自定义功能表

表 32.4 “XPath 自定义功能” 显示您可以在 Apache Camel XPath 表达式中使用的自定义功能。除了标准 XPath 功能外，这些功能还可使用。

表 32.4. XPath 自定义功能

功能	描述
in:body（）	返回 邮件正文。
in:header(*HeaderName*)	返回包含 name, *HeaderName* 的消息标头。
out:body()	返回 Out 消息正文。
out:header(*HeaderName*)	返回带有 name, *HeaderName* 的 Out message 标头。
function:properties(*PropKey*)	使用密钥 *PropKey* 查找属性。

功能	描述
功能：simple(*SimpleExp*)	评估指定的简单表达式(*SimpleExp*)。

第 33 章 XQUERY

概述

XQuery 最初被设计为作为存储在数据库中 XML 表单的数据的查询语言。XQuery 语言可让您在消息采用 XML 格式时选择当前消息的部分。XQuery 是 XPath 语言的超集，因此，任何有效的 XPath 表达式也是有效的 XQuery 表达式。

JAVA 语法

您可以通过几种方法将 XQuery 表达式传递给 xquery（）。对于简单表达式，您可以将 XQuery 表达式作为字符串传递(java.lang.String)。对于较长的 XQuery 表达式，您可能想将表达式存储在文件中，然后通过将 java.io.File 参数或 java. net.URL 参数传递给 overloaded xquery（） 方法来引用。XQuery 表达式隐式操作消息内容，并返回节点集。根据上下文，返回值将解释为 predicate（空节点集被解释为 false）或为一个表达式。

添加 SAXON 模块

要在路由中使用 XQuery，您需要将有关 camel-saxon 的依赖项添加到您的项目中，如 例 33.1 “添加 camel-saxon 依赖项” 所示。

例 33.1. 添加 camel-saxon 依赖项

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-saxon</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

EAP 部署上的 CAMEL

camel-saxon 组件由 Camel on EAP(Wildfly Camel)框架支持，该框架可在红帽 JBoss 企业应用平台 (JBoss EAP)容器上提供简化的部署模型。

静态导入

要在应用程序代码中使用 xquery（） 静态方法，请在 Java 源文件中包含以下导入声明：

```
import static org.apache.camel.component.xquery.XQueryBuilder.xquery;
```

变量

表 33.1 “XQuery 变量” 列出使用 XQuery 时可访问的变量。

表 33.1. XQuery 变量

变量	类型	描述
Exchange	**Exchange**	当前交换
in.body	**对象**	IN 消息的正文
out.body	**对象**	OUT 消息的正文
in.headers.*key*	**对象**	键是 *密钥的*N 消息标头
out.headers.*key*	**对象**	键为密钥的 OUT 消息标头
key	**对象**	Exchange 属性，*其密钥*是

示例

例 33.2 “使用 XQuery 的路由” 显示使用 XQuery 的路由。

例 33.2. 使用 XQuery 的路由

```
<camelContext>
  <route>
    <from uri="activemq:MyQueue"/>
    <filter>
      <language langauge="xquery">/foo:person[@name='James']</language>
      <to uri="mqseries:SomeOtherQueue"/>
    </filter>
  </route>
</camelContext>
```

部分 III. 高级 CAMEL 编程

本指南论述了如何使用 Apache Camel API。

第 34 章 了解消息格式

摘要

在开始使用 Apache Camel 编程之前，您需要明确了解消息和消息交换如何建模。由于 Apache Camel 可以处理许多消息格式，因此基本消息类型设计为具有抽象格式。Apache Camel 提供了访问和转换数据格式所需的 API，这些格式是消息正文和消息标头下。

34.1. 交换

概述

exchange 对象 是封装接收的消息并存储其关联的元数据（包括 交换属性）的打包程序。另外，如果当前消息被分配给制作者端点，则交换会提供一个临时插槽来保存回复（ Out 消息）。

Apache Camel 中交换的一个重要特点是它们支持创建消息的速度。如果不需要明确访问消息的路由，这可以提供显著的优化功能。

图 34.1. 通过路由交换对象传输

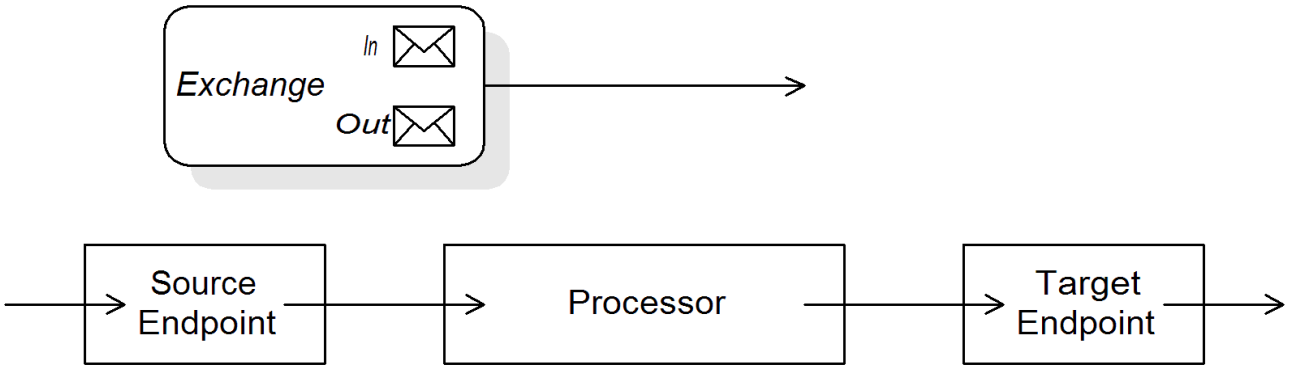


图 34.1 “通过路由交换对象传输” 显示通过路由的交换对象。在路由上下文中，交换对象将作为 Processor.process（） 方法的参数传递。这意味着，交换对象可以被源端点、目标端点和两者中的所有处理器直接访问。

Exchange 接口

org.apache.camel.Exchange 接口定义了用于访问 In 和 Out 消息的方法，如 例 34.1 “Exchange method” 所示。

例 34.1. Exchange method

```
// Access the In message
```

```
Message getIn();
void    setIn(Message in);

// Access the Out message (if any)
Message getOut();
void    setOut(Message out);
boolean hasOut();

// Access the exchange ID
String  getExchangeId();
void    setExchangeId(String id);
```

有关 Exchange 界面中的方法的完整描述，请参阅 第 43.1 节 “Exchange Interface”。

延迟创建消息

Apache Camel 支持在、Out 和 故障消息中创建 延迟。这意味着在您尝试访问它们之前不会创建消息实例（例如，调用 getIn（） 或 getOut（））。lazy 消息创建语义由 org.apache.camel.impl.DefaultExchange 类实施。

如果您调用一个 no-argument accessors（getIn（） 或 getOut（）），或者调用等于 true 的 accessor（即 getIn(true) 或 getOut(true)），则默认方法实施会创建一个新的消息实例（如果其中一个实例尚不存在）。

如果您调用了布尔值参数的 accessor 等于 false （即 getIn(false) 或 getOut(false)），则默认方法实施会返回当前消息值。[1]

延迟创建交换 ID

Apache Camel 支持创建交换 ID。您可以在任何交换上调用 getExchangeId（），以获取该交换实例的唯一 ID，但只有您真正调用这个方法时才会生成 ID。这个方法的 DefaultExchange.getExchangeId（） 实现将 ID 生成委托给使用 CamelContext 注册的 UUID 生成器。

有关如何使用 CamelContext 注册 UUID 生成器的详情，请参考 第 34.4 节 “built-In UUID Generators”。

34.2. 消息

概述

Message 对象使用以下抽象模型代表信息：

- 邮件正文
- **Message headers**
- 消息附加

消息正文和消息标头可以是任意类型（它们被声明为类型 对象）并将消息附件声明为类型为 javax.activation.DataHandler，其中可以包含任意 MIME 类型。如果您需要获取消息内容的声明，您可以使用类型转换器机制将正文和标头转换为另一个类型，可能使用 marshalling 和 unmarshalling 机制。

Apache Camel 消息的一个重要功能是它们支持对消息正文和标头 的创建。在某些情况下，这意味着消息可以通过路由，而无需解析所有消息。

Message 接口

org.apache.camel.Message 接口定义了访问消息正文、消息标头和消息附加的方法，如 例 34.2 “消息接口” 所示。

例 34.2. 消息接口

```
// Access the message body
Object getBody();
<T> T  getBody(Class<T> type);
void   setBody(Object body);
<T> void setBody(Object body, Class<T> type);

// Access message headers
Object getHeader(String name);
<T> T  getHeader(String name, Class<T> type);
void   setHeader(String name, Object value);
Object removeHeader(String name);
Map<String, Object> getHeaders();
void setHeaders(Map<String, Object> headers);

// Access message attachments
javax.activation.DataHandler getAttachment(String id);
java.util.Map<String, javax.activation.DataHandler> getAttachments();
java.util.Set<String> getAttachmentNames();
void addAttachment(String id, javax.activation.DataHandler content)
```

```
// Access the message ID
String getMessageId();
void   setMessageId(String messageId);
```

有关消息接口方法的完整描述，请参阅 第 44.1 节 “消息接口”。

lazy creation of bodies、headers 和 attachments

Apache Camel 支持创建正文、标头和附加内容。这意味着，代表邮件正文、邮件标题或消息附加的对象不会被创建，直到需要它们。

例如，请考虑以下路由，它从 In 消息访问 foo 邮件标头：

```
from("SourceURL")
    .filter(header("foo")
    .isEqualTo("bar"))
    .to("TargetURL");
```

在本路由中，如果假设 SourceURL 引用的组件支持 lazy 创建，In message 标头不会被实际解析，直到执行 header("foo") 调用为止。此时，底层消息实现会解析标头并填充标头映射。在到达路由结束前，邮件正文 不会在路由末尾（位于 ("TargetURL") 调用前解析。此时，正文将转换为写成目标端点 TargetURL 所需的格式。

在填充正文、标头和附加前等待最后一个可能的时间，您可以确保避免不必要的类型转换。在某些情况下，您可以完全避免解析。例如，如果路由没有对消息标头的显式引用，则消息可以在不需要解析标头的情况下遍历路由。

是否在实践中实施是否实施是否取决于底层组件的实施。通常，当创建消息正文、邮件标题或消息附件是代价昂贵的时，会非常宝贵的创建。有关实现支持延迟创建的消息类型的详情，请参考 第 44.2 节 “实施消息接口”。

消息 ID 的 lazy 创建

Apache Camel 支持创建消息 ID。也就是说，只有在您实际调用 getMessageId（） 方法时才会生成消息 ID。这个方法的 DefaultExchange.getExchangeId（） 实现将 ID 生成委托给使用 CamelContext 注册的 UUID 生成器。

如果端点实施了需要唯一消息 ID 的协议，一些端点实施会隐式调用 getMessageId（） 方法。特别是，JMS 消息通常包含一个包含唯一消息 ID 的标头，因此 JMS 组件会自动调用 getMessageId（） 获取消息 ID（这由 JMS 端点上的 messageIdEnabled 选项控制）。

有关如何使用 CamelContext 注册 UUID 生成器的详情，请参考 第 34.4 节 “built-In UUID Generators”。

初始消息格式

In 消息的初始格式由源端点决定，Out 消息的初始格式由目标端点决定。如果底层组件支持 lazy 创建，则该消息将保持不变，直到应用程序明确访问为止。大部分 Apache Camel 组件以相对的原始格式是：使用 byte[] 等类型（如 byte[]、ste Buffer、InputStream 或 OutputStream ）创建消息正文。这样可确保创建初始信息所需的开销最小。当需要更协作的消息格式时，通常依赖于 类型转换器 或 总结处理器。

类型转换器

消息的初始格式无关紧要，因为您可以使用内置的类型转换器（请参阅 第 34.3 节 “内置(In Type Converters)”）轻松地将信息从一个格式转换为另一个格式（请参阅 ）。Apache Camel API 中有多种方法可公开类型转换功能。例如，可以将 convertBodyTo(Class type) 方法插入路由中，以转换 In 消息的正文，如下所示：

```
from("SourceURL").convertBodyTo(String.class).to("TargetURL");
```

In 邮件的正文将转换为 java.lang.String。以下示例演示了如何在 In 消息正文末尾附加字符串：

```
from("SourceURL").setBody(bodyAs(String.class).append("My Special Signature")).to("TargetURL");
```

在将字符串附加到结尾前，将消息正文转换为字符串格式。本例中并不需要显式转换邮件正文。您还可以使用：

```
from("SourceURL").setBody(body().append("My Special Signature")).to("TargetURL");
```

其中 append（） 方法会在附加参数前自动将消息正文转换为字符串。

消息中的类型转换方法

org.apache.camel.Message 接口会公开一些方法来明确执行类型转换：

- getBody(Class<T> type) the message body as type, T.
- getHeader(String name, Class<T> type) abrt-abrtRe returns named header value as type, T.

有关支持的转换类型的完整列表，请参阅 第 34.3 节 “内置(In Type Converters)”。

转换为 XML

除了支持简单类型（如 byte[]、steBuffer、String 等等），内置类型转换器还支持转换为 XML 格式。例如，您可以将消息正文转换为 org.w3c.dom.Document 类型。这个转换比简单转换更为昂贵，因为它涉及解析整个消息，然后创建一个节点来代表 XML 文档结构。您可以转换为以下 XML 文档类型：

- org.w3c.dom.Document
- javax.xml.transform.sax.SAXSource

XML 类型转换的适用性比更简单的转换更小。因为并非每个消息正文都符合 XML 结构，所以您必须记住这种类型转换可能会失败。另一方面，有许多路由器仅涉及 XML 消息类型。

marshalling 和 unmarshalling

摘要 涉及将高级别格式转换为低级格式，而 unmarshalling 涉及将低级格式转换为高级别格式。以下两个处理器用于在路由中执行摘要或解压缩：

- marshal()
- unmarshal()

例如，要从文件中读取序列化 Java 对象并将其解包到 Java 对象，您可以使用 例 34.3 “解包 Java 对象” 中显示的路由定义。

例 34.3. 解包 Java 对象

```
from("file://tmp/appfiles/serialized")
    .unmarshal()
    .serialization()
    .<FurtherProcessing>
    .to("TargetURL");
```

最终消息格式

当 In 消息到达路由末尾时，目标端点必须能够将消息正文转换为可写入物理端点的格式。同一规则适用于 注销 到达源端点的消息。这个转换通常是隐式执行，使用 Apache Camel 类型转换器。通常，这涉及从低级格式转换为另一个低级格式，例如从 byte[] 数组转换为 InputStream 类型。

34.3. 内置(IN TYPE CONVERTERS)

概述

本节介绍了 master 类型转换器支持的转换。这些转换内置到 Apache Camel 内核中。

通常，通过方便功能调用类型转换器，如 Message.getBody(Class<T> type) 或 Message.getHeader(String name, Class<T> type)。也可以直接调用 master 类型转换器。例如，如果您有一个交换对象 Exchange，您可以将给定值转换为 String，如 例 34.4 “将值转换为字符串” 所示。

例 34.4. 将值转换为字符串

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
String str_value = tc.convertTo(String.class, value);
```

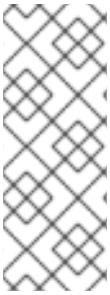
基本类型转换器

Apache Camel 提供内置类型转换器，用于执行转换至以下基本类型以及以下基本类型：

- java.io.File
- 字符串
- byte[] and java.nio.ByteBuffer

- *java.io.InputStream and java.io.OutputStream*
- *java.io.Reader and java.io.Writer*
- *java.io.BufferedReader and java.io.BufferedWriter*
- *java.io.StringReader*

但是，不是所有这些类型都是相互依存关系。内置转换器主要侧重于提供来自文件和 字符串类型 的转换。文件类型 可以转换为上述任意类型，但 Reader、作者 和 StringReader 除外。String 类型可以转换为 文件,byte[], ByteBuffer,InputStream, 或 StringReader。通过将字符串解释为文件名，从 String 到 File 可转换。字符串、字节[] 和 ByteBuffer 的 trio 完全可见。



注意

您可以通过在当前交换中设置 Exchange.CHARSET_NAME Exchange 属性，明确指定 用于从字节[] 转换的字符编码，从 String 转换为 byte[] 。例如，要使用 UTF-8 字符编码执行转换，调用 exchange.setProperty("Exchange.CHARSET_NAME", "UTF-8"). 支持的字符集在 java.nio.charset.Charset 类中描述。

集合类型转换器

Apache Camel 提供内置类型转换器，用于对以下集合类型执行转换：

- *Object[]*
- *java.util.Set*
- *java.util.List*

支持在上述集合类型之间进行转换的所有变化。

映射类型转换器

Apache Camel 提供内置类型转换器，用于执行转换，或从以下映射类型进行转换：

- java.util.Map
- java.util.HashMap
- java.util.Hashtable
- java.util.Properties

以上映射类型也可以转换为 java.util.Set 类型的集合，其中 set 元素是 MapEntry<K,V& gt; 类型的集合。

DOM 类型转换器

您可以对以下 Document Object Model(DOM)类型进行类型转换：

- org.w3c.dom.Document InventoryService-jaxbconvertible from byte[]、String、java.io.File 和 java.io.InputStream.
- org.w3c.dom.Node
- javax.xml.transform.dom.DOMSource 匹配项(String)。
- javax.xml.transform.Source 匹配项（从 byte[] 和 String ）中可见。

支持上述 DOM 类型之间的转换的所有变化。

SAX 类型转换器

您还可以对 javax.xml.transform.sax.SAXSource 类型执行转换，该类型支持 SAX 事件驱动的 XML

解析器（请参阅 SAX 网站 了解详细信息）。您可以从以下类型转换为 SAXSource ：

- 字符串
- InputStream
- 源
- StreamSource
- DOMSource

enum 类型 converter

Camel 提供了一个类型转换器，用于执行 num 类型转换的字符串，其中字符串值将转换为来自指定枚举类的匹配枚举数（匹配是 区分大小写）。 转换消息正文，很少需要这个类型转换程序，但 Apache Camel 在内部使用它来选择特定选项。

例如，当设置日志级别选项时，以下值 INFO 将被转换为 enum constant ：

```
<to uri="log:foo?level=INFO"/>
```

因为 enum 类型转换器不区分大小写，所以以下任意一种替代方法也可以正常工作 ：

```
<to uri="log:foo?level=info"/>
<to uri="log:foo?level=INfo"/>
<to uri="log:foo?level=InFo"/>
```

自定义类型转换器

Apache Camel 还允许您实施自己的自定义类型转换器。有关如何实施自定义类型转换器的详情，请参考 第 36 章 类型转换器。

34.4. BUILT-IN UUID GENERATORS

概述

Apache Camel 允许您在 CamelContext 中注册 UUID 生成器。然后，每当 Apache Camel 需要生成唯一 ID 确认时，调用该 UUID 生成器来生成 Exchange.getExchangeId（） 和 Message.getMessageId（） 方法返回的 ID。

例如，如果应用程序的一部分不支持 ID 且长度为 36 个字符（如 Websphere MQ），则您可能更愿意替换默认的 UUID 生成器。另外，使用简单计数器生成 ID 可以方便（请参阅 SimpleUuidGenerator）进行测试。

提供的 UUID 生成器

您可以将 Apache Camel 配置为使用以下 UUID 生成器之一，该生成器在内核中提供：

- *org.apache.camel.impl.ActiveMQUuidGenerator abrt-（默认） 生成与 Apache ActiveMQ 使用的相同 ID 样式。这种实现可能不适用于所有应用程序，因为它使用云计算环境中禁止的一些 JDK API（如 Google App Engine）。*
- *org.apache.camel.impl.SimpleUuidGenerator KUBECONFIG-KUBECONFIGimplements a simple counter ID，从 1 开始。底层实施使用 java.util.concurrent.atomic.AtomicLong 类型，因此它是 thread-safe。*
- *org.apache.camel.impl.JavaUuidGenerator >_<-sHistoryLimit 实施基于 java.util.UUID 类型的 ID。由于 java.util.UUID 已同步，因此可能会影响某些高度并发系统的性能。*

自定义 UUID 生成器

要实施自定义 UUID 生成器，请实施 org.apache.camel.spi.UuidGenerator 接口，它出现在 例 34.5 “UuidGenerator 接口” 中。必须实施 generateUuid（） 才能返回唯一 ID 字符串。

例 34.5. UuidGenerator 接口

```
// Java
package org.apache.camel.spi;

/**
 * Generator to generate UUID strings.
 */
public interface UuidGenerator {
    String generateUuid();
}
```

使用 Java 指定 UUID 生成器

要使用 Java 替换默认 UUID 生成器，请调用当前 CamelContext 对象上的 setUuidGenerator（） 方法。例如，您可以使用当前的 CamelContext 注册 SimpleUuidGenerator 实例，如下所示：

```
// Java
getContext().setUuidGenerator(new org.apache.camel.impl.SimpleUuidGenerator());
```

注意

在激活任何路由 前，应在启动时调用 setUuidGenerator（） 方法。

使用 Spring 指定 UUID 生成器

要使用 Spring 来替换默认 UUID 生成器，您要做的都是使用 Spring bean 元素创建 UUID 生成器实例。创建 camelContext 实例时，它会自动查找 Spring registry，搜索实施 org.apache.camel.spi.UuidGenerator 的 Bean。例如，您可以使用 CamelContext 注册 SimpleUuidGenerator 实例，如下所示：

```
<beans ...>
  <bean id="simpleUuidGenerator"
        class="org.apache.camel.impl.SimpleUuidGenerator" />

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

---

[1] 如果没有活动方法，返回的值将为 null。

第 35 章 实现处理器

摘要

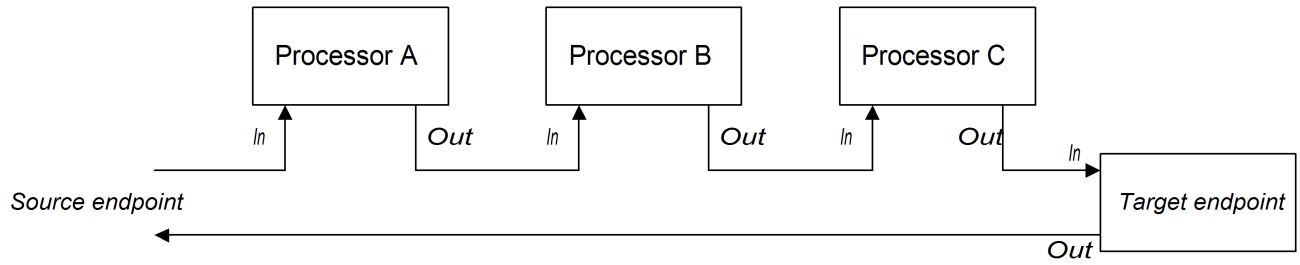
Apache Camel 允许您实施自定义处理器。然后，您可以将自定义处理器插入到路由中，以便在通过路由时对交换对象执行操作。

35.1. 处理模型

pipelining model

pipelining 模型 描述了处理器以 第 5.4 节 “pipes 和 Filters” 排列的方式。pipelining 是处理一系列端点（制作端点只是一个特殊类型处理器）的最常用方法。当处理器以这种方式排列时，交换的 In 和 Out 消息将按照 图 35.1 “pipelining Model” 所示进行处理。

图 35.1. pipelining Model



管道中的处理器看起来类似于服务，其中 In 消息类似于请求，Out 消息类似于回复。实际上，在现实管道中，管道中的节点通常由 Web 服务端点（如 CXF 组件）实施。

例如，例 35.1 “Java DSL Pipeline” 显示由两个处理器、ProcessorA、ProcessorB 和 producer 端点(TargetURI)组成的 Java DSL 管道。

例 35.1. Java DSL Pipeline

```
from(SourceURI).pipeline(ProcessorA, ProcessorB, TargetURI);
```

35.2. 实施简单处理器

概述

这部分论述了如何在将交换分配给路由中的下一个处理器前实施一个执行消息处理逻辑的简单处理器。

处理器接口

通过实施 **org.apache.camel.Processor** 接口来创建简单的处理器。如 *例 35.2 “处理器接口”* 所示，接口定义了一个方法 **process（），** 它处理交换对象。

例 35.2. 处理器接口

```
package org.apache.camel;

public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

实现处理器接口

要创建简单的处理器，您必须实现处理器接口，并提供 **process（）** 方法的逻辑。*例 35.3 “简单处理器实现”* 显示了简单的处理器实施概述。

例 35.3. 简单处理器实现

```
import org.apache.camel.Processor;

public class MyProcessor implements Processor {
    public MyProcessor() { }

    public void process(Exchange exchange) throws Exception
    {
        // Insert code that gets executed *before* delegating
        // to the next processor in the chain.
        ...
    }
}
```

process（） 方法中的所有代码在 **exchange** 对象被委派给链中的下一个处理器之前执行。

有关如何访问简单处理器中的邮件正文和标头值的示例，请参考 *第 35.3 节 “访问消息内容”*。

将简单的处理器插入到路由中

使用 **process（）** DSL 命令，将简单的处理器插入到路由中。创建自定义处理器实例，然后将此实例

作为参数传递给 **process（） 方法，如下所示：**

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL").process(myProc).to("TargetURL");
```

35.3. 访问消息内容

访问邮件标头

邮件标题通常包含路由器视角中最有用的消息内容，因为标头通常在路由器服务中处理。要访问标头数据，您必须首先从 **Exchange** 对象获取消息（例如，使用 **Exchange.getIn（）），然后使用 Message** 接口来检索各个标头（例如，使用 **Message.getHeader（）**。

例 35.4 “访问授权标头” 显示了一个自定义处理器的示例，它访问名为 **Authorization** 的标头的值。这个示例使用 **ExchangeHelper.getMandatoryHeader（） 方法，它无需测试 null** 标头值。

例 35.4. 访问授权标头

```
import org.apache.camel.*;
import org.apache.camel.util.ExchangeHelper;

public class MyProcessor implements Processor {
  public void process(Exchange exchange) {
    String auth = ExchangeHelper.getMandatoryHeader(
                  exchange,
                  "Authorization",
                  String.class
                );
    // process the authorization string...
    // ...
  }
}
```

有关 **Message** 接口的详情，请参考 第 34.2 节 “消息”。

访问邮件正文

您还可以访问邮件正文。例如，要将字符串附加到 **In** 消息的末尾，您可以使用 例 35.5 “访问消息正文” 中显示的处理器。

例 35.5. 访问消息正文

```
_
```

```
import org.apache.camel.*;
import org.apache.camel.util.ExchangeHelper;

public class MyProcessor implements Processor {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}
```

访问消息附加

您可以使用 Message.getAttachment（） 方法或 Message.getAttachments（） 方法访问消息的附件。详情请查看 例 34.2 “消息接口”。

35.4. EXCHANGEHELPER 类

概述

org.apache.camel.util.ExchangeHelper 类是 Apache Camel 实用程序类，提供在实施处理器时可以使用的方法。

解决端点

静态 resolveEndpoint（） 方法是 ExchangeHelper 类中最有用的方法之一。您可以在处理器中使用它来即时创建新的 Endpoint 实例。

例 35.6. resolveEndpoint（） 方法

```
public final class ExchangeHelper {
    ...
    @SuppressWarnings({"unchecked" })
    public static Endpoint
    resolveEndpoint(Exchange exchange, Object value)
        throws NoSuchEndpointException { ... }
    ...
}
```

resolveEndpoint（） 的第一个参数是一个交换实例，第二个参数通常是端点 URI 字符串。例 35.7

"创建文件端点" 显示如何从交换实例 交换中创建新的文件端点

> ***例 35.7. 创建文件端点***
>
> ```
> Endpoint file_endp = ExchangeHelper.resolveEndpoint(exchange, "file://tmp/messages/in.xml");
> ```

嵌套交换访问器

ExchangeHelper 类提供了多种静态方法，其格式为 getMandatoryBeanProperty（），它将 对应的 getBeanProperty（） 方法包装在 Exchange 类上。两者之间的区别在于，原始 getBeanProperty（） accessors 返回 null （如果对应的属性不可用），而 getMandatoryBeanProperty（） 打包程序方法会引发 Java 异常。以下打包程序方法在 ExchangeHelper 类中实施：

```
public final class ExchangeHelper {
    ...
    public static <T> T getMandatoryProperty(Exchange exchange, String propertyName, Class<T>
type)
        throws NoSuchPropertyException { ... }

    public static <T> T getMandatoryHeader(Exchange exchange, String propertyName, Class<T>
type)
        throws NoSuchHeaderException { ... }

    public static Object getMandatoryInBody(Exchange exchange)
        throws InvalidPayloadException { ... }

    public static <T> T getMandatoryInBody(Exchange exchange, Class<T> type)
        throws InvalidPayloadException { ... }

    public static Object getMandatoryOutBody(Exchange exchange)
        throws InvalidPayloadException { ... }

    public static <T> T getMandatoryOutBody(Exchange exchange, Class<T> type)
        throws InvalidPayloadException { ... }
    ...
}
```

测试交换模式

几种不同的交换模式与保留 In 消息兼容。几个不同的交换模式也与按住 Out 消息兼容。为提供检查交换对象是否能够持有 In 消息或 Out 消息的快速方法，ExchangeHelper 类将提供以下方法：

```
public final class ExchangeHelper {
    ...
    public static boolean isInCapable(Exchange exchange) { ... }
```

```
    public static boolean isOutCapable(Exchange exchange) { ... }
    ...
}
```

获取 In 消息的 MIME 内容类型

如果要找出交换的 MIME 内容类型，可以通过调用 ExchangeHelper.getContentType(exchange) 方法来访问它。要实现此实施，ExchangeHelper 对象会查找 In 消息的 Content-Type header>_<-jaxb this 方法的值，它依赖于底层组件来填充标头值。

第 36 章 类型转换器

摘要

Apache Camel 具有一个内置类型转换机制，用于将消息正文和消息标头转换为不同的类型。本章介绍了如何通过添加您自己的自定义转换方法来扩展类型转换机制。

36.1. 类型转换器架构

概述

本节描述了类型转换器机制的整体架构（如果您想要编写自定义类型转换器），您必须了解它。如果您只需要使用内置的类型转换器，请参阅 第 34 章 了解消息格式。

类型转换器接口

例 36.1 “TypeConverter Interface” 显示 org.apache.camel.TypeConverter 接口的定义，所有类型转换器都必须实现。

例 36.1. TypeConverter Interface

```
package org.apache.camel;

public interface TypeConverter {
    <T> T convertTo(Class<T> type, Object value);
}
```

控制器类型转换器

Apache Camel 类型转换器机制遵循控制器/worker 模式。有许多 worker 类型转换器，各自能够执行有限数量的转换，以及一个 控制器类型 转换器，用于聚合 worker 执行的类型。控制器类型转换器充当 worker 类型转换器的前端。当您请求控制器执行类型转换时，它会选择适当的 worker，并将转换任务委托给该 worker。

对于类型转换机制的用户，控制器类型转换更为重要，因为它提供了用于访问转换机制的入口点。在启动过程中，Apache Camel 会自动将控制器类型转换器实例与 CamelContext 对象关联。要获得对控制器类型转换器的引用，调用 CamelContext.getTypeConverter（） 方法。例如，如果您有一个交换对象交换对象，您可以获得对控制器类型转换器的引用，如 例 36.2 “获取控制器类型 Converter” 所示。

例 36.2. 获取控制器类型 Converter

```
-
```

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
```

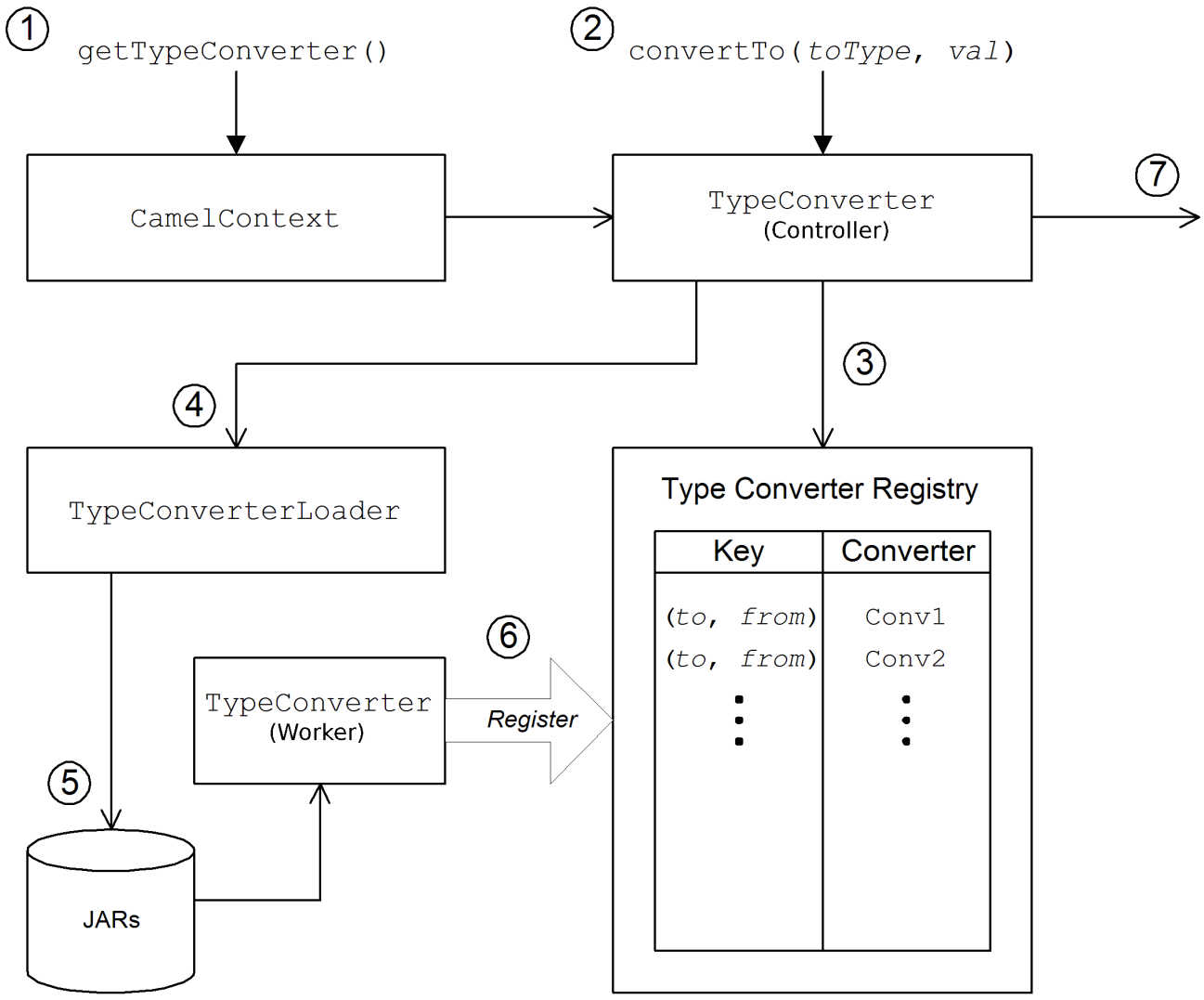
类型 converter loader

控制器类型转换器使用 类型转换程序 来填充 worker 类型转换器的 registry。类型转换程序是任何实现 TypeConverterLoader 接口的类。Apache Camel 目前只使用一种类型 converter loader","-abrtthe 注解类型 converter loader （ AnnotationTypeConverterLoader 类型）。

类型转换过程

图 36.1 “类型转换过程” 概述了类型转换过程，显示将给定数据值转换为指定类型到指定类型所涉及的步骤。

图 36.1. 类型转换过程



类型转换机制如下：

1. CamelContext 对象包含对控制器 TypeConverter 实例的引用。转换流程中的第一步是通过调用 CamelContext.getTypeConverter（） 检索控制器类型转换器。

2. 通过调用 controller 类型 converter 上的 convertTo（） 方法来启动类型转换。此方法指示类型转换器，将数据对象 value 从原始类型转换为 toType 参数指定的类型。

3. 因为控制器类型转换器是很多不同的 worker 类型转换器的前端，它通过检查类型映射的 registry 来查找适当的 worker 类型转换器。type converters 的 registry 由类型映射对（从Type ） 进行密钥。如果在 registry 中找到合适的类型转换器，控制器类型转换器会调用 worker 的 convertTo（） 方法，并返回结果。

4. 如果 registry 中没有找到合适的类型转换器，控制器类型转换器会加载新的类型转换程序，使用类型转换程序。

5. type converter loader 搜索 classpath 上的可用 JAR 库来查找合适的类型转换器。目前，使用的 loader 策略由注解类型转换程序实施，它会尝试加载由 org.apache.camel.Converter 注解标注的类。请参阅 *“创建 TypeConverter 文件”一节*。

6. 如果类型 converter loader 成功，则会载入一个新的 worker 类型转换程序，并进入类型转换器 registry。然后，使用这个类型 converter 将 value 参数转换为 toType 类型。

7. 如果数据成功转换，则返回转换的数据值。如果转换没有成功，则返回 null。

36.2. 处理 DUPLICATE TYPE CONVERTERS

如果添加了重复类型转换器，您可以配置必须发生的情况。

在 TypeConverterRegistry (See *第 36.3 节 “使用 Annotations 实现类型转换器”*)中，您可以使用以下代码将操作设置为 Override、Ignore 或 Fail ：

```
typeconverterregistry = camelContext.getTypeConverter()
// Define the behaviour if the TypeConverter already exists
typeconverterregistry.setTypeConverterExists(TypeConverterExists.Override);
```

根据您的要求，此代码中 覆盖 可以被 Ignore 或 Fail 替代。

TypeConverterExists Class

TypeConverterExists 类由以下命令组成：

```
package org.apache.camel;

import javax.xml.bind.annotation.XmlEnum;

/**
 * What to do if attempting to add a duplicate type converter
 *
 * @version
 */
@XmlEnum
public enum TypeConverterExists {

    Override, Ignore, Fail

}
```

36.3. 使用 ANNOTATIONS 实现类型转换器

概述

类型转换机制可以通过添加新的 worker 类型转换程序来轻松自定义。本节论述了如何实现 worker 类型转换器以及如何将其与 Apache Camel 集成，以便注解类型转换程序自动加载它。

如何实现类型转换器

要实现自定义类型转换器，请执行以下步骤：

1. “实施注解的转换器类”一节。
2. “创建 TypeConverter 文件”一节。
3. “软件包类型转换器”一节。

实施注解的转换器类

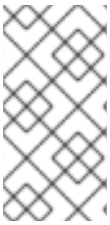
您可以使用 @Converter 注释实施自定义类型转换器类。您必须注解类本身，以及用于执行类型转换

的每种 静态方法。每个转换器方法都使用一个参数来定义 来自 类型，可选择使用第二个 Exchange 参数，并且具有 定义要 类型的非void 返回值。类型转换程序使用 Java 反应来查找注解的方法，并将它们集成到类型转换器机制中。例 36.3 “注解的 Converter 类示例” 显示注解的转换器类示例，它定义了转换器方法，用于从 java.io.File 转换为 java.io.InputStream 和另一个转换器方法（使用 Exchange 参数），用于将 from byte[] 转换为 String。

例 36.3. 注解的 Converter 类示例

```
package com.YourDomain.YourPackageName;

import org.apache.camel.Converter;

import java.io.*;

@Converter
public class IOConverter {
    private IOConverter() {
    }

    @Converter
    public static InputStream toInputStream(File file) throws FileNotFoundException {
        return new BufferedInputStream(new FileInputStream(file));
    }

    @Converter
    public static String toString(byte[] data, Exchange exchange) {
        if (exchange != null) {
            String charsetName = exchange.getProperty(Exchange.CHARSET_NAME, String.class);
            if (charsetName != null) {
                try {
                    return new String(data, charsetName);
                } catch (UnsupportedEncodingException e) {
                    LOG.warn("Can't convert the byte to String with the charset " + charsetName, e);
                }
            }
        }
        return new String(data);
    }
}
```

toInputStream（） 方法负责将从 File 类型转换为 InputStream 类型，而 toString（） 方法负责执行从 byte[] 类型到 String 类型的转换。



注意

方法名称不重要，可以是您选择的任何内容。重要的是参数类型、返回类型和是否存在 @Converter 注释。

创建 TypeConverter 文件

要为自定义转换器启用发现机制（由 注解类型转换程序实现），请在以下位置创建一个 TypeConverter 文件：

```
META-INF/services/org/apache/camel/TypeConverter
```

TypeConverter 文件必须包含类型为 converter 类的 Fully Qualified Names(FQN)列表。例如，如果您想要类型转换程序加载程序来搜索 您的PackageName.YourClassName 软件包，用于注解的转换器类，TypeConverter 文件将包含以下内容：

```
com.PackageName.FooClass
```

启用发现机制的替代方法是将软件包名称添加到 TypeConverter 文件。例如，TypeConverter 文件将包含以下内容：

```
com.PackageName
```

这将导致软件包扫描程序通过 @Converter 标签的软件包扫描。使用 FQN 方法更快，是首选的方法。

软件包类型转换器

类型转换器被打包为 JAR 文件，其中包含自定义类型转换器的编译类和 META-INF 目录。将这个 JAR 文件放在您的类路径上，使其可用于 Apache Camel 应用程序。

fallback converter 方法

除了使用 @Converter 注释定义常规转换器方法外，您还可以使用 @FallbackConverter 注释来定义回退转换器方法。只有在控制器类型转换器无法在类型 registry 中找到常规转换器方法时，才会尝试回退转换器方法。

常规转换方法和回退转换方法之间基本区别是，而常规转换器则定义了在特定对类型的转换（例如，从 字节[] 到 String），而回退转换器 可以在任何 对类型间执行转换。最多是回退转换器方法正文中的代码，以找出它能够执行的转换。在运行时，如果转换无法通过常规转换器执行转换，控制器类型转换器会迭代每个可用的回退转换器，直到它找到可以执行转换的转换。

回退转换器的方法签名可以采用以下格式之一：

```
// 1. Non-generic form of signature
@FallbackConverter
public static Object MethodName(
    Class type,
    Exchange exchange,
    Object value,
    TypeConverterRegistry registry
)

// 2. Templating form of signature
@FallbackConverter
public static <T> T MethodName(
    Class<T> type,
    Exchange exchange,
    Object value,
    TypeConverterRegistry registry
)
```

其中 MethodName 是回退转换器的任意方法名称。

例如，以下代码提取（来自 File 组件的实现中）会显示一个回退转换器，它可转换 GenericFile 对象的正文，利用类型转换器在类型转换器中已有的转换器：

```
package org.apache.camel.component.file;

import org.apache.camel.Converter;
import org.apache.camel.FallbackConverter;
import org.apache.camel.Exchange;
import org.apache.camel.TypeConverter;
import org.apache.camel.spi.TypeConverterRegistry;

@Converter
public final class GenericFileConverter {

    private GenericFileConverter() {
        // Helper Class
    }

    @FallbackConverter
    public static <T> T convertTo(Class<T> type, Exchange exchange, Object value,
TypeConverterRegistry registry) {
        // use a fallback type converter so we can convert the embedded body if the value is GenericFile
        if (GenericFile.class.isAssignableFrom(value.getClass())) {
            GenericFile file = (GenericFile) value;
            Class from = file.getBody().getClass();
            TypeConverter tc = registry.lookup(type, from);
            if (tc != null) {
                Object body = file.getBody();
                return tc.convertTo(type, exchange, body);
            }
        }
```

```
        return null;
    }
    ...
}
```

36.4. 直接实现类型转换

概述

通常，实现类型转换器的建议方法是使用注解的类，如上一节中所述，第 36.3 节 “使用 Annotations 实现类型转换器”。但是，如果要完全控制类型转换器，您可以实施自定义 worker 类型转换器，直接将其添加到类型转换器 registry 中，如下所述。

实施 TypeConverter 接口

要实现自己的类型转换器类，请定义一个实施 TypeConverter 接口的类。例如，以下 MyOrderTypeConverter 类将整数值转换为 MyOrder 对象，其中整数值用于初始化 MyOrder 对象中的顺序 ID。

```
import org.apache.camel.TypeConverter

private class MyOrderTypeConverter implements TypeConverter {

    public <T> T convertTo(Class<T> type, Object value) {
        // converter from value to the MyOrder bean
        MyOrder order = new MyOrder();
        order.setId(Integer.parseInt(value.toString()));
        return (T) order;
    }

    public <T> T convertTo(Class<T> type, Exchange exchange, Object value) {
        // this method with the Exchange parameter will be preferd by Camel to invoke
        // this allows you to fetch information from the exchange during convertions
        // such as an encoding parameter or the likes
        return convertTo(type, value);
    }

    public <T> T mandatoryConvertTo(Class<T> type, Object value) {
        return convertTo(type, value);
    }

    public <T> T mandatoryConvertTo(Class<T> type, Exchange exchange, Object value) {
        return convertTo(type, value);
    }
}
```

将类型转换器添加到 registry

您可以使用类似如下的代码将自定义类型转换器直接添加到类型转换器 registry 中：

```
// Add the custom type converter to the type converter registry
context.getTypeConverterRegistry().addTypeConverter(MyOrder.class, String.class, new
MyOrderTypeConverter());
```

其中上下文是当前的 org.apache.camel.CamelContext 实例。addTypeConverter（） 方法根据特定类型转换注册 MyOrderTypeConverter 类，从 String.class 到 MyOrder.class。

您可以将自定义类型转换器添加到 Camel 应用程序，而无需使用 META-INF 文件。如果您使用 Spring 或 Blueprint，则可以只声明一个 <bean>。CamelContext 自动发现 Bean 并添加转换器。

```
<bean id="myOrderTypeConverters" class="..."/>
 <camelContext>
   ...
</camelContext>
```

如果您有更多类，您可以声明多个 <bean>s。

第 37 章 PRODUCER 和 CONSUMER 模板

摘要

Apache Camel 中的制作者和消费者模板按照 Spring 容器 API 的功能建模，其中可通过简化、易用的 API（称为 模板 ）来提供对资源的访问。对于 Apache Camel，生产者模板和使用者模板提供了从生产者端点和消费者端点发送到和接收消息的简化接口。

37.1. 使用 PRODUCER 模板

37.1.1. Producer 模板介绍

概述

producer 模板支持多种不同方法来调用制作者端点。有方法支持请求消息的不同格式（作为 Exchange 对象、作为邮件正文、带有单一标头设置的消息正文，等等）的方法支持同步和异步调用方式。总体而言，制作者模板方法可分为以下类别：

- 同步调用
- 使用处理器同步调用
- 异步调用
- 使用回调的异步调用

或者，请参阅 第 37.2 节 “使用 Fluent Producer 模板”。

同步调用

异步调用端点的方法具有 发送Suffix（） 形式的名称 并请求Suffix （）。例如，使用默认消息交换模式(MEP)或明确指定 MEP 的调用端点的方法命名为 send（）、 sendBody（） 和 sendBodyAndHeader（） （其中这些方法分别发送 Exchange 对象、邮件正文或消息正文和标题值）。如果您想要强制 MEP 成为 InOut （请求/回复语义），您可以改为调用 request（）、requestBody（） 和 requestBodyAndHeader（） 方法。

以下示例演示了如何创建 ProducerTemplate 实例，并使用它来向 activemq:MyQueue 端点发送消息正文。这个示例还演示了如何使用 sendBodyAndHeader（） 发送消息正文和标头值。

```
import org.apache.camel.ProducerTemplate
import org.apache.camel.impl.DefaultProducerTemplate
...
ProducerTemplate template = context.createProducerTemplate();

// Send to a specific queue
template.sendBody("activemq:MyQueue", "<hello>world!</hello>");

// Send with a body and header
template.sendBodyAndHeader(
    "activemq:MyQueue",
    "<hello>world!</hello>",
    "CustomerRating", "Gold" );
```

使用处理器同步调用

一个特殊的同步调用，当您通过 Processor 参数（而非 Exchange 参数）提供 send（） 方法。在这种情况下，制作者模板隐式询问指定的端点来创建 Exchange 实例（通常通常不会默认具有 InOnly MEP）。然后，此默认交换会传递到处理器，初始化交换对象的内容。

以下示例演示了如何将 MyProcessor 处理器初始化的交换发送到 activemq:MyQueue 端点。

```
import org.apache.camel.ProducerTemplate
import org.apache.camel.impl.DefaultProducerTemplate
...
ProducerTemplate template = context.createProducerTemplate();

// Send to a specific queue, using a processor to initialize
template.send("activemq:MyQueue", new MyProcessor());
```

MyProcessor 类按照下例中所示实施。除了设置 In 消息正文（如此处所示），您也可以初始化邮件标题和交换属性。

```
import org.apache.camel.Processor;
import org.apache.camel.Exchange;
...
public class MyProcessor implements Processor {
    public MyProcessor() { }

    public void process(Exchange ex) {
        ex.getIn().setBody("<hello>world!</hello>");
    }
}
```

异步调用

*异步 调用端点的方法具有格式 **asyncSendSuffix（） 和 asyncRequestSuffix（）。例如，使用默认消息交换模式(MEP)或明确指定 MEP 的调用端点的方法命名为 asyncSend（） 和 asyncSendBody（） （其中，这两种方法分别发送 Exchange 对象或消息正文）。如果您想要强制 MEP 成为 InOut （请求/reply 语义），您可以调用 asyncRequestBody（）、asyncRequestBodyAndHeader（） 以及 asyncRequestBodyAndHeaders（） 方法。***

*以下示例演示了如何异步向 **direct:start 端点发送交换。asyncSend（） 方法返回 java.util.concurrent.Future 对象，用于稍后检索调用结果。***

```
import java.util.concurrent.Future;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultExchange;
...
Exchange exchange = new DefaultExchange(context);
exchange.getIn().setBody("Hello");

Future<Exchange> future = template.asyncSend("direct:start", exchange);

// You can do other things, whilst waiting for the invocation to complete
...
// Now, retrieve the resulting exchange from the Future
Exchange result = future.get();
```

*producer 模板还提供了以异步方式发送消息正文的方法（例如，使用 **asyncSendBody（） 或 asyncRequestBody（））。在这种情况下，您可以使用以下帮助程序方法之一从 Future 对象中提取返回的消息正文：***

```
<T> T extractFutureBody(Future future, Class<T> type);
<T> T extractFutureBody(Future future, long timeout, TimeUnit unit, Class<T> type) throws
TimeoutException;
```

extractFutureBody（） 方法的第一个版本将阻止，直到调用完成并且回复消息可用。extractFutureBody（） 方法的第二个版本允许您指定超时。两种方法都具有类型参数，键入，它使用内置的类型转换器将返回的消息正文转换为指定的类型。

*以下示例演示了如何使用 **asyncRequestBody（） 方法向 direct:start 端点发送消息正文。然后，使用块 extractFutureBody（） 方法从 Future 对象检索回复消息正文。***

```
Future<Object> future = template.asyncRequestBody("direct:start", "Hello");

// You can do other things, whilst waiting for the invocation to complete
```

```
...
// Now, retrieve the reply message body as a String type
String result = template.extractFutureBody(future, String.class);
```

使用回调的异步调用

在前面的异步示例中，请求消息会在子线程中发送，同时由主线程检索和处理回复。制作者模板还为您提供 选项；但是，使用 asyncCallback（）、asyncCallbackSendBody（） 或 asyncCallbackRequestBody（） 方法来处理回复。在这种情况下，您可以提供一个回调对象（ org.apache.camel.impl.SynchronizationAdapter 类型），它会在回复消息到达后立即在子线程中调用它。

同步 回调接口定义如下：

```
package org.apache.camel.spi;

import org.apache.camel.Exchange;

public interface Synchronization {
    void onComplete(Exchange exchange);
    void onFailure(Exchange exchange);
}
```

在收到普通回复时调用 onComplete（） 方法，并在收到错误消息回复时调用 onFailure（） 方法。只有其中一种方法可以返回，因此您必须覆盖这两者，以确保处理所有类型的答复。

以下示例演示了如何将交换发送到 direct:start 端点，该端点由 SynchronizationAdapter 回调对象在子线程中处理答复信息。

```
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultExchange;
import org.apache.camel.impl.SynchronizationAdapter;
...
Exchange exchange = context.getEndpoint("direct:start").createExchange();
exchange.getIn().setBody("Hello");

Future<Exchange> future = template.asyncCallback("direct:start", exchange, new
SynchronizationAdapter() {
    @Override
    public void onComplete(Exchange exchange) {
        assertEquals("Hello World", exchange.getIn().getBody());
    }
});
```

如果 SynchronizationAdapter 类是 Synchronization 界面的默认实现，您可以覆盖它以提供 Complete（）和 onFailure（） 回调方法自己的定义。

您仍然可以选择从主线程访问回复，因为 asyncCallback（） 方法也返回 Future 对象是：

```
// Retrieve the reply from the main thread, specifying a timeout
Exchange reply = future.get(10, TimeUnit.SECONDS);
```

37.1.2. 同步发送

概述

同步发送 方法是您可以用来调用制作者端点的方法集合，即当前线程块，直到方法调用完成并且收到回复（若有）。这些方法与任何消息交换协议兼容。

发送交换

基本 send（） 方法是一种通用的方法，它利用交换模式(MEP)将 Exchange 对象的内容发送到端点。返回值是您由制作者端点处理后获得的交换（可能含有 Out 消息，具体取决于 MEP）。

发送交换的 send（） 方法有三个变体，可让您使用以下方法之一指定目标端点：作为默认端点，作为端点 URI，或作为 Endpoint 对象。

```
Exchange send(Exchange exchange);
Exchange send(String endpointUri, Exchange exchange);
Exchange send(Endpoint endpoint, Exchange exchange);
```

发送由处理器填充的交换

常规 send（） 方法的一个简单变体是使用处理器来填充默认交换，而不是显式提供交换对象（详情请参阅 “使用处理器同步调用”一节）。

发送由处理器填充的交换的 send（） 方法可让您使用以下方法之一指定目标端点：作为默认端点，作为端点 URI，或作为 Endpoint 对象。另外，您可以通过提供 模式 参数来指定交换的 MEP，而不必接受默认值。

```
Exchange send(Processor processor);
Exchange send(String endpointUri, Processor processor);
Exchange send(Endpoint endpoint, Processor processor);
```

```
Exchange send(
    String endpointUri,
    ExchangePattern pattern,
    Processor processor
);
Exchange send(
    Endpoint endpoint,
    ExchangePattern pattern,
    Processor processor
);
```

发送消息正文

如果您只关注要发送的消息正文内容，您可以使用 **sendBody（）** 方法将消息正文作为参数提供，并让制作者模板将正文插入到默认交换对象中。

sendBody（） 方法可让您使用以下方法之一指定目标端点：作为默认端点，作为端点 **URI**，或作为 **Endpoint** 对象。另外，您可以通过提供 **模式** 参数来指定交换的 **MEP**，而不必接受默认值。没有 **模式** 参数返回 **void** 的方法（即使调用可能会引发回复）；并且带有 **pattern** 参数的方法返回 **Out** 消息的正文（如果存在一个）或 **In** 消息的正文（否则为跳过）。

```
void sendBody(Object body);
void sendBody(String endpointUri, Object body);
void sendBody(Endpoint endpoint, Object body);
Object sendBody(
    String endpointUri,
    ExchangePattern pattern,
    Object body
);
Object sendBody(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body
);
```

发送邮件正文和标头。

出于测试目的，尝试单个标头设置的影响通常是对于这种标头测试很有用的，而 **sendBodyAndHeader（）** 方法也很有用。您将消息正文和标头设置作为环境变量提供 **sendBodyAndHeader（）**，并让制作者模板负责将 **body** 和 **header** 设置插入到默认交换对象中。

sendBodyAndHeader（） 方法可让您使用以下方法之一指定目标端点：作为默认端点，作为端点 **URI**，或作为 **Endpoint** 对象。另外，您可以通过提供 **模式** 参数来指定交换的 **MEP**，而不必接受默认值。没有 **模式** 参数返回 **void** 的方法（即使调用可能会引发回复）；并且带有 **pattern** 参数的方法返回 **Out** 消息的正文（如果存在一个）或 **In** 消息的正文（否则为跳过）。

```
void sendBodyAndHeader(
    Object body,
    String header,
    Object headerValue
);
void sendBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
void sendBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
Object sendBodyAndHeader(
    String endpointUri,
    ExchangePattern pattern,
    Object body,
    String header,
    Object headerValue
);
Object sendBodyAndHeader(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body,
    String header,
    Object headerValue
);
```

sendBodyAndHeaders（） 方法与 sendBodyAndHeader（） 方法类似，除了除了仅提供单个标头设置外，这两种方法都允许您指定标头设置的完整散列映射。

```
void sendBodyAndHeaders(
    Object body,
    Map<String, Object> headers
);
void sendBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers
);
void sendBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers
);
Object sendBodyAndHeaders(
    String endpointUri,
    ExchangePattern pattern,
    Object body,
```

```
    Map<String, Object> headers
);
Object sendBodyAndHeaders(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body,
    Map<String, Object> headers
);
```

发送消息正文和交换属性

您可以使用 sendBodyAndProperty（） 方法尝试设置单个交换属性的效果。您将消息正文和属性设置作为环境变量提供 sendBodyAndProperty（），并让制作者模板负责将正文和交换属性插入到默认交换对象中。

sendBodyAndProperty（） 方法可让您使用以下方法之一指定目标端点：作为默认端点，作为端点 URI，或作为 Endpoint 对象。另外，您可以通过提供 模式 参数来指定交换的 MEP，而不必接受默认值。没有 模式 参数返回 void 的方法（即使调用可能会引发回复）；并且带有 pattern 参数的方法返回 Out 消息的正文（如果存在一个）或 In 消息的正文（否则为跳过）。

```
void sendBodyAndProperty(
    Object body,
    String property,
    Object propertyValue
);
void sendBodyAndProperty(
    String endpointUri,
    Object body,
    String property,
    Object propertyValue
);
void sendBodyAndProperty(
    Endpoint endpoint,
    Object body,
    String property,
    Object propertyValue
);
Object sendBodyAndProperty(
    String endpoint,
    ExchangePattern pattern,
    Object body,
    String property,
    Object propertyValue
);
Object sendBodyAndProperty(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body,
    String property,
    Object propertyValue
);
```

37.1.3. 通过 InOut Pattern 进行同步请求

概述

同步请求 方法与同步发送方法类似，除了请求方法强制消息交换模式强制为 InOut （与 request/reply 语义一致）。因此，如果您期望从制作者端点接收回复，通常使用同步请求方法。

请求一个由处理器填充的交换

basic request（） 方法是一个通用的通用方法，它使用处理器填充默认交换方法，并强制消息交换模式是 InOut （因此调用 obeys 请求/reply语义）。return 值是您由制作者端点处理后获取的交换，其中 Out 消息包含回复消息。

用于发送交换的 request（） 方法可让您使用以下方法之一指定目标端点： 作为端点 URI，或作为 Endpoint 对象。

```
Exchange request(String endpointUri, Processor processor);
Exchange request(Endpoint endpoint, Processor processor);
```

请求消息正文

如果您只关注请求和回复中消息正文的内容，您可以使用 requestBody（） 方法以参数形式提供请求消息正文，并让制作者模板负责将正文插入到默认交换对象中。

requestBody（） 方法可让您使用以下方法之一指定目标端点： 作为默认端点，作为端点 URI，或作为 Endpoint 对象。返回值是回复消息的正文(Out message body)，它可以作为普通 对象 返回或转换为特定类型的 T，使用内置的类型转换器（请参阅 第 34.3 节 “内置(In Type Converters)”）。

```
Object requestBody(Object body);
<T> T requestBody(Object body, Class<T> type);
Object requestBody(
    String endpointUri,
    Object body
);
<T> T requestBody(
    String endpointUri,
    Object body,
    Class<T> type
);
Object requestBody(
    Endpoint endpoint,
    Object body
);
<T> T requestBody(
```

```
    Endpoint endpoint,
    Object body,
    Class<T> type
);
```

请求消息正文和标头。

您可以使用 requestBodyAndHeader（） 方法尝试设置单个标头值的效果。您将消息正文和标头设置作为环境变量提供 requestBodyAndHeader（），并让制作者模板负责将正文和交换属性插入到默认交换对象中。

requestBodyAndHeader（） 方法可让您使用以下方法之一指定目标端点：作为端点 URI，或作为 Endpoint 对象。返回值是回复消息的正文(Out message body)，它可以作为普通 对象 返回或转换为特定类型的 T，使用内置的类型转换器（请参阅 第 34.3 节 “内置(In Type Converters)”）。

```
Object requestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
<T> T requestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
Object requestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
<T> T requestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
```

requestBodyAndHeaders（） 方法与 requestBodyAndHeader（） 方法类似，除了除了仅提供一个标头设置外，这两种方法都允许您指定标头设置的完整散列映射。

```
Object requestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers
```

```
);
<T> T requestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
Object requestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers
);
<T> T requestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
```

37.1.4. 异步发送

概述

producer 模板提供了各种方法来异步调用制作者端点，因此主线程在等待调用完成并且稍后可以检索回复消息时不会阻断。本节中描述的异步发送方法与任何消息交换协议兼容。

发送交换

基本 asyncSend（） 方法采用 Exchange 参数，并异步调用端点，并使用指定交换的消息交换模式 (MEP)。返回值是一个 java.util.concurrent.Future 对象，它是一个 ticket，您可以在以后用来收集回复信息。有关如何从 Future 对象获取返回值的详情，请参考 “异步调用”一节。

以下 asyncSend（） 方法可让您使用以下方法之一指定目标端点：作为端点 URI，或作为 Endpoint 对象。

```
Future<Exchange> asyncSend(String endpointUri, Exchange exchange);
Future<Exchange> asyncSend(Endpoint endpoint, Exchange exchange);
```

发送由处理器填充的交换

常规 asyncSend（） 方法的简单变体是使用处理器来填充默认交换，而不是显式提供交换对象。

以下 asyncSend（） 方法可让您使用以下方法之一指定目标端点：作为端点 URI，或作为 Endpoint 对象。

```
Future<Exchange> asyncSend(String endpointUri, Processor processor);
Future<Exchange> asyncSend(Endpoint endpoint, Processor processor);
```

发送消息正文

如果您只关注要发送的消息正文内容，您可以使用 asyncSendBody（） 方法异步发送消息正文，并让制作者模板将正文插入到默认交换对象中。

asyncSendBody（） 方法可让您使用以下方法之一指定目标端点：作为端点 URI，或作为 Endpoint 对象。

```
Future<Object> asyncSendBody(String endpointUri, Object body);
Future<Object> asyncSendBody(Endpoint endpoint, Object body);
```

37.1.5. 使用 InOut Pattern 的异步请求

概述

异步请求 方法与异步发送方法类似，除了请求方法强制消息交换模式强制为 InOut （显示 request/reply语义）。因此，如果您期望从制作者端点接收回复，通常会方便使用异步请求方法。

请求消息正文

如果您只关注请求和回复中消息正文的内容，您可以使用 requestBody（） 方法以参数形式提供请求消息正文，并让制作者模板负责将正文插入到默认交换对象中。

asyncRequestBody（） 方法可让您使用以下方法之一指定目标端点：作为端点 URI，或作为 Endpoint 对象。从 Future 对象检索的返回值是回复消息的正文（传出消息正文），可以作为纯文本对象或转换为特定类型的 T （使用内置类型转换器）返回（请参见 “异步调用”一节）。

```
Future<Object> asyncRequestBody(
    String endpointUri,
    Object body
);
<T> Future<T> asyncRequestBody(
    String endpointUri,
    Object body,
    Class<T> type
);
Future<Object> asyncRequestBody(
    Endpoint endpoint,
    Object body
);
```

```
<T> Future<T> asyncRequestBody(
    Endpoint endpoint,
    Object body,
    Class<T> type
);
```

请求消息正文和标头。

您可以使用 asyncRequestBodyAndHeader（） 方法尝试设置单个标头值的效果。您将消息正文和标头设置作为环境变量提供给 asyncRequestBodyAndHeader（），并让制作者模板负责将正文和交换属性插入默认交换对象。

asyncRequestBodyAndHeader（） 方法可让您使用以下方法之一指定目标端点：作为端点 URI，或作为 Endpoint 对象。从 Future 对象检索的返回值是回复消息的正文（传出消息正文），可以作为纯文本对象或转换为特定类型的 T （使用内置类型转换器）返回（请参见 “异步调用”一节）。

```
Future<Object> asyncRequestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
<T> Future<T> asyncRequestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
Future<Object> asyncRequestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
<T> Future<T> asyncRequestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
```

asyncRequestBodyAndHeaders（） 方法与 asyncRequestBodyAndHeader（） 方法类似，除了提供单个标头设置外，这两种方法都允许您指定完整的哈希映射。

```
Future<Object> asyncRequestBodyAndHeaders(
    String endpointUri,
    Object body,
```

```
    Map<String, Object> headers
);
<T> Future<T> asyncRequestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
Future<Object> asyncRequestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers
);
<T> Future<T> asyncRequestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
```

37.1.6. 使用回调进行异步发送

概述

制作者模板也提供用于调用制作者端点的同一子线程中处理回复消息的选项。在这种情况下，您提供一个回调对象，当收到回复消息时，该对象会在子线程中自动调用。换而言之，使用回调方法的异步发送可让您在主线程中发起调用，然后让制作者端点的所有关联处理在制作器端点的关联处理会，等待回复和处理在子线程中异步处理回复。

发送交换

基本 asyncCallback（） 方法采用 交换 参数，并异步调用端点，并使用指定交换的消息交换模式 (MEP)。此方法类似于用于交换的 asyncSend（） 方法，但它使用额外的 org.apache.camel.spi.Synchronization 参数，它是一个带有以下两个方法的回调接口：在 Complete（） 和 onFailure（） 上。有关如何使用 Synchronization 回调的详情，请参考 “使用回调的异步调用”一节。

以下 asyncCallback（） 方法可让您使用以下方法之一指定目标端点：作为端点 URI，或作为 Endpoint 对象。

```
Future<Exchange> asyncCallback(
    String endpointUri,
    Exchange exchange,
    Synchronization onCompletion
);
Future<Exchange> asyncCallback(
    Endpoint endpoint,
```

```
    Exchange exchange,
    Synchronization onCompletion
);
```

发送由处理器填充的交换

*处理器的 **asyncCallback（） 方法调用处理器以填充默认交换，并强制消息交换模式为 InOut （因此调用 obeys 请求/reply 语义）。***

*以下 **asyncCallback（） 方法可让您使用以下方法之一指定目标端点： 作为端点 URI，或作为 Endpoint 对象。***

```
Future<Exchange> asyncCallback(
    String endpointUri,
    Processor processor,
    Synchronization onCompletion
);
Future<Exchange> asyncCallback(
    Endpoint endpoint,
    Processor processor,
    Synchronization onCompletion
);
```

发送消息正文

*如果您只关注要发送的消息正文内容，您可以使用 **asyncCallbackSendBody（） 方法以异步发送消息正文，并让制作者模板负责将正文插入默认交换对象。***

asyncCallbackSendBody（） 方法可让您使用以下方法之一指定目标端点： 作为端点 URI 或一个 Endpoint 对象。

```
Future<Object> asyncCallbackSendBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackSendBody(
    Endpoint endpoint,
    Object body,
    Synchronization onCompletion
);
```

请求消息正文

*如果您只关注请求和回复中消息正文的内容，您可以使用 **asyncCallbackRequestBody（） 方法将*

请求消息正文作为参数提供，并让制作者模板将正文插入默认交换对象。

asyncCallbackRequestBody（） 方法可让您使用以下方法之一指定目标端点： 作为端点 URI，或作为 Endpoint 对象。

```
Future<Object> asyncCallbackRequestBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackRequestBody(
    Endpoint endpoint,
    Object body,
    Synchronization onCompletion
);
```

37.2. 使用 FLUENT PRODUCER 模板

由 Camel 2.18 提供

FluentProducerTemplate 界面为构建制作者提供了一个流畅的语法。DefaultFluentProducerTemplate 类实施 FluentProducerTemplate。

以下示例使用 DefaultFluentProducerTemplate 对象来设置标头和正文：

```
Integer result = DefaultFluentProducerTemplate.on(context)
    .withHeader("key-1", "value-1")
    .withHeader("key-2", "value-2")
    .withBody("Hello")
    .to("direct:inout")
    .request(Integer.class);
```

以下示例演示了如何在 DefaultFluentProducerTemplate 对象中指定处理器：

```
Integer result = DefaultFluentProducerTemplate.on(context)
    .withProcessor(exchange -> exchange.getIn().setBody("Hello World"))
    .to("direct:exception")
    .request(Integer.class);
```

下一个示例演示了如何自定义默认的 fluent producer 模板：

```
Object result = DefaultFluentProducerTemplate.on(context)
```

```
    .withTemplateCustomizer(
        template -> {
            template.setExecutorService(myExecutor);
            template.setMaximumCacheSize(10);
        }
    )
    .withBody("the body")
    .to("direct:start")
    .request();
```

要创建 FluentProducerTemplate 实例，请在 Camel 上下文上调用 createFluentProducerTemplate（） 方法。例如：

```
FluentProducerTemplate fluentProducerTemplate = context.createFluentProducerTemplate();
```

37.3. 使用 CONSUMER TEMPLATE

概述

使用者模板提供了轮询消费者端点的方法，以接收传入的消息。您可以选择以交换对象的形式接收传入的消息，也可以采用消息正文格式（其中消息正文可以使用内置类型转换器来广播到特定类型的消息）。

轮询交换示例

您可以使用以下轮询方法之一使用消费者模板来轮询消费者端点进行交换：块 receive（）； receive（） 带有超时；或 接收NoWait（），该端点会立即返回。由于消费者端点代表服务，因此在尝试轮询交换之前，还要通过调用 start（） 来启动服务线程。

以下示例演示了如何使用 blocking receive（） 方法从 seda:foo 消费者端点轮询交换：

```
import org.apache.camel.ProducerTemplate;
import org.apache.camel.ConsumerTemplate;
import org.apache.camel.Exchange;
...
ProducerTemplate template = context.createProducerTemplate();
ConsumerTemplate consumer = context.createConsumerTemplate();

// Start the consumer service
consumer.start();
...
template.sendBody("seda:foo", "Hello");
Exchange out = consumer.receive("seda:foo");
```

```
...
// Stop the consumer service
consumer.stop();
```

如果使用者模板实例 消费者，使用 CamelContext.createConsumerTemplate（） 方法进行实例化，并通过调用 ConsumerTemplate.start（） 启动使用者服务线程。

轮询消息正文示例

您还可以使用以下方法之一为传入消息正文轮询使用者端点：blocking receiveBody（）；receiveBody（）, receiveBody（）；或 receiveBodyNoWait（），这将立即返回。如上例所示，在尝试轮询交换之前，还要通过调用 start（） 来启动服务线程。

以下示例演示了如何使用块 receiveBody（） 方法从 seda:foo 消费者端点轮询传入的邮件正文：

```
import org.apache.camel.ProducerTemplate;
import org.apache.camel.ConsumerTemplate;
...
ProducerTemplate template = context.createProducerTemplate();
ConsumerTemplate consumer = context.createConsumerTemplate();

// Start the consumer service
consumer.start();
...
template.sendBody("seda:foo", "Hello");
Object body = consumer.receiveBody("seda:foo");
...
// Stop the consumer service
consumer.stop();
```

轮询交换方法

从消费者端点轮询 交换 有三种基本方法： receive（） 无限，没有超时块；接收（），指定毫秒的超时块；接收NoWait（） 是非阻塞的。您可以将使用者端点指定为端点 URI，也可以指定为 Endpoint 实例。

```
Exchange receive(String endpointUri);
Exchange receive(String endpointUri, long timeout);
Exchange receiveNoWait(String endpointUri);

Exchange receive(Endpoint endpoint);
Exchange receive(Endpoint endpoint, long timeout);
Exchange receiveNoWait(Endpoint endpoint);
```

轮询消息正文方法

从消费者端点轮询 消息正文 有三种基本方法： receiveBody（） 无限，没有超时块，receiveBody（） 且具有指定毫秒的超时块； 接收BodyNoWait（） 是非阻塞的。您可以将使用者端点指定为端点 URI，也可以指定为 Endpoint 实例。此外，通过调用这些方法的模板形式，您可以使用内置的类型转换器将返回的正文转换为特定类型的 T。

```
Object receiveBody(String endpointUri);
Object receiveBody(String endpointUri, long timeout);
Object receiveBodyNoWait(String endpointUri);

Object receiveBody(Endpoint endpoint);
Object receiveBody(Endpoint endpoint, long timeout);
Object receiveBodyNoWait(Endpoint endpoint);

<T> T receiveBody(String endpointUri, Class<T> type);
<T> T receiveBody(String endpointUri, long timeout, Class<T> type);
<T> T receiveBodyNoWait(String endpointUri, Class<T> type);

<T> T receiveBody(Endpoint endpoint, Class<T> type);
<T> T receiveBody(Endpoint endpoint, long timeout, Class<T> type);
<T> T receiveBodyNoWait(Endpoint endpoint, Class<T> type);
```

第 38 章 实施组件

摘要

本章一般概述可用于实施 Apache Camel 组件。

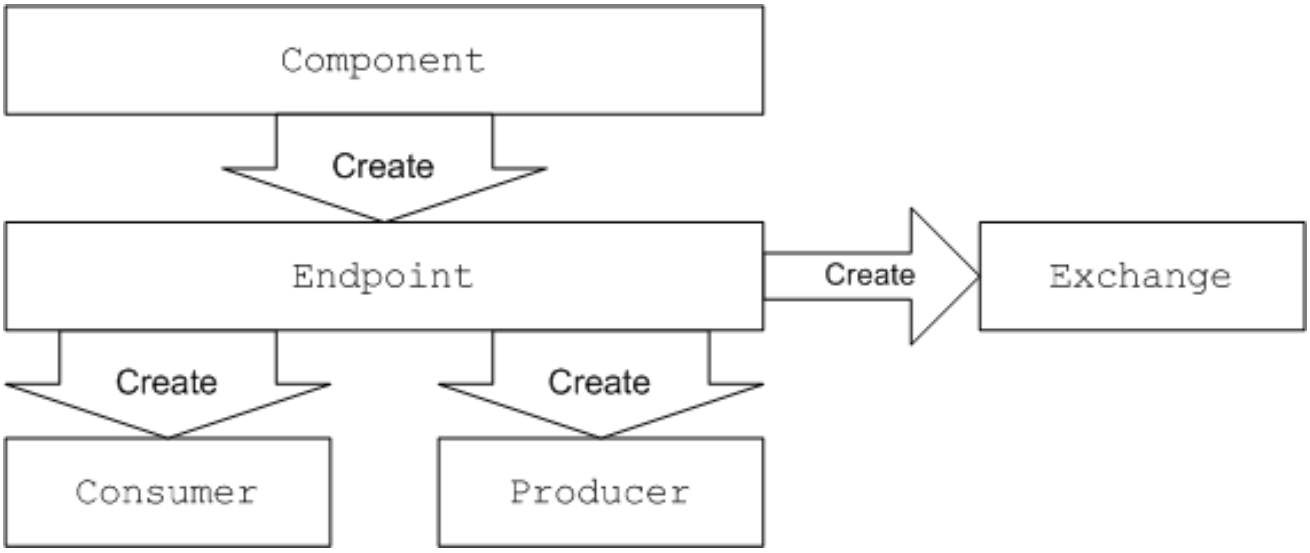
38.1. 组件架构

38.1.1. 组件的工厂模式

概述

Apache Camel 组件由一组通过工厂模式相互关联的类组成。组件的主要入口点是 Component 对象本身（ org.apache.camel.Component 类型的实例）。您可以使用 component 对象作为工厂来创建 Endpoint 对象，后者又是创建 使用者、Producer 和 Exchange 对象的工厂。这些关系在 中进行了概述 图 38.1 “组件 onnectionFactoryy Patterns”

图 38.1. 组件 onnectionFactoryy Patterns



组件

组件实施是一个端点工厂。组件实施器的主要任务是实施 component .createEndpoint（） 方法，它负责根据需要创建新端点。

每种组件必须与出现在端点 URI 中的组件前缀 关联。例如，文件组件通常与 文件 前缀关联，该前缀可以在 file://tmp/messages/input 等端点 URI 中使用。当您在 Apache Camel 中安装新组件时，您必须定义特定组件前缀和实现组件的类名称之间的关联。

端点

每个端点实例封装了特定的端点 URI。每次 Apache Camel 遇到新端点 URI 时，它都会创建一个新的端点实例。端点对象也是创建消费者端点和制作者端点的工厂。

端点必须实施 org.apache.camel.Endpoint 接口。Endpoint 接口定义了以下工厂方法：

- createConsumer（） 和 createPollingConsumer（） abrt-活动Creates 是一个消费者端点，代表路由开头的源端点。
- createProducer（） >_<-jaxbCreates a producer 端点，它代表路由末尾的目标端点。
- createExchange（） InventoryService-jaxbCreates a exchange 对象，用于封装通过和关闭路由的消息。

消费者

消费者端点 会消耗 请求。它们始终显示在路由开始时，并封装负责接收传入请求和分配传出回复的代码。从面向服务的视角来看，使用者代表 服务。

消费者必须实施 org.apache.camel.Consumer 接口。在实施消费者时，可以遵循多种不同模式。这些模式在 第 38.1.3 节 “消费者模式和线程” 中所述。

producer

生产者端点 生成 请求。它们始终出现在路由的末尾，并封装负责分配传出请求和接收传入的回复的代码。从面向服务的视角来看，生产者代表服务消费者 。

制作者必须实施 org.apache.camel.Producer 接口。您可以选择实施制作者来支持异步处理方式。详情请查看 第 38.1.4 节 “异步处理”。

Exchange

Exchange 对象封装一组相关的消息。例如，一种消息交换是同步调用，它由请求消息和相关回复组成。

Exchanges 必须实施 org.apache.camel.Exchange 接口。默认的实现(DefaultExchange)足以满

足许多组件实施。但是，如果您要将额外的数据与交换相关联，或者交换了额外处理，那么自定义交换实施会很有用。

消息

Exchange 对象中有两个不同的消息插槽：

- 在 message 这个过程中保留当前消息。
- out message>_<-abrttemporarily 包含回复信息。

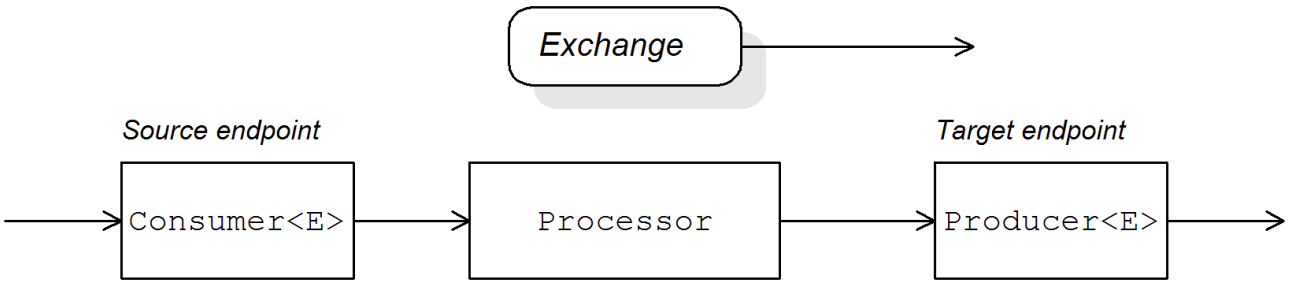
所有消息类型都由同一 Java 对象 org.apache.camel.Message 来表示。并非始终需要自定义消息的实施，即 default Message 通常会足够。

38.1.2. 在路由中使用组件

概述

Apache Camel 路由基本上是一个处理器的管道，即 org.apache.camel.Processor 类型。消息封装在交换对象 E 中，通过调用 process（） 方法从节点传递到节点。处理器管道的架构在 图 38.2 “Route 中的使用者和 Producer 实例” 中进行了说明。

图 38.2. Route 中的使用者和 Producer 实例



源端点

在路由开始时，您有源端点，由 org.apache.camel.Consumer 对象表示。源端点负责接受传入的请求消息并分配回复。在构建路由时，Apache Camel 根据端点 URI 的组件前缀创建适当的 Consumer 类型，如 第 38.1.1 节 “组件的工厂模式” 所述。

处理器

管道中的每个中间节点由处理器对象（实施 org.apache.camel.Processor 接口）表示。您可以插入标准处理器（例如，过滤、throttler 或 delayer）或者插入您自己的自定义处理器实施。

目标端点

路由的末尾是目标端点，由 org.apache.camel.Producer 对象代表。由于它源自处理器管道，因此制作者也是处理器对象（实施 org.apache.camel.Processor 接口）。目标端点负责发送传出请求消息并接收传入的回复。在构建路由时，Apache Camel 根据来自端点 URI 的组件前缀来创建适当的 Producer 类型。

38.1.3. 消费者模式和线程

概述

用于实施使用者的模式决定了处理传入交换时使用的线程模型。消费者可使用以下模式之一实施：

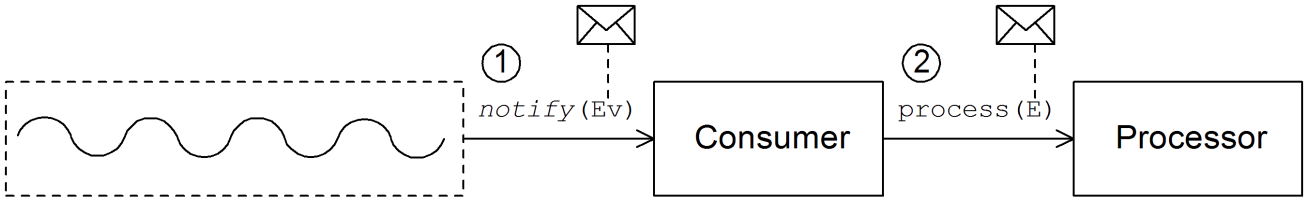
- 事件驱动的模式 abrt- the consumer 由外部线程驱动。
- 调度轮询模式 InventoryService ->_<The consumer is by a dedicated thread pool（一个专用的线程池）。
- Polling pattern InventoryService- the threading 模型保留为未定义状态。

事件驱动的模式

在事件驱动的模式中，当应用程序的另一个部分（通常是第三方库）调用消费者实施的方法时，发起传入请求的处理。事件驱动的消费者是一个很好的例子，即 Apache Camel JMX 组件，其中的事件由 JMX 库启动。JMX 库调用 handleNotification（） 方法，以启动 requests processing>_<-abrt，有关详细信息，请参阅 例 41.4 “JMXConsumer 实施”

图 38.3 “event-Driven Consumer” 显示了事件驱动的消费者模式的概要。在本例中，假定调用 notify（） 方法触发了处理。

图 38.3. event-Driven Consumer



事件驱动的消费者按如下方式处理传入请求：

1. *使用者必须实施一种方法才能接收传入事件（在 图 38.3 “event-Driven Consumer” 中，这由 notify（） 方法表示）。调用 notify（） 的线程通常是应用的独立部分，因此消费者的线程策略是外部的。*

   例如，在 JMX 消费者实施的情况下，使用者实施 NotificationListener.handleNotification（） 方法，以接收来自 JMX 的通知。驱动消费者处理的线程在 JMX 层内创建。

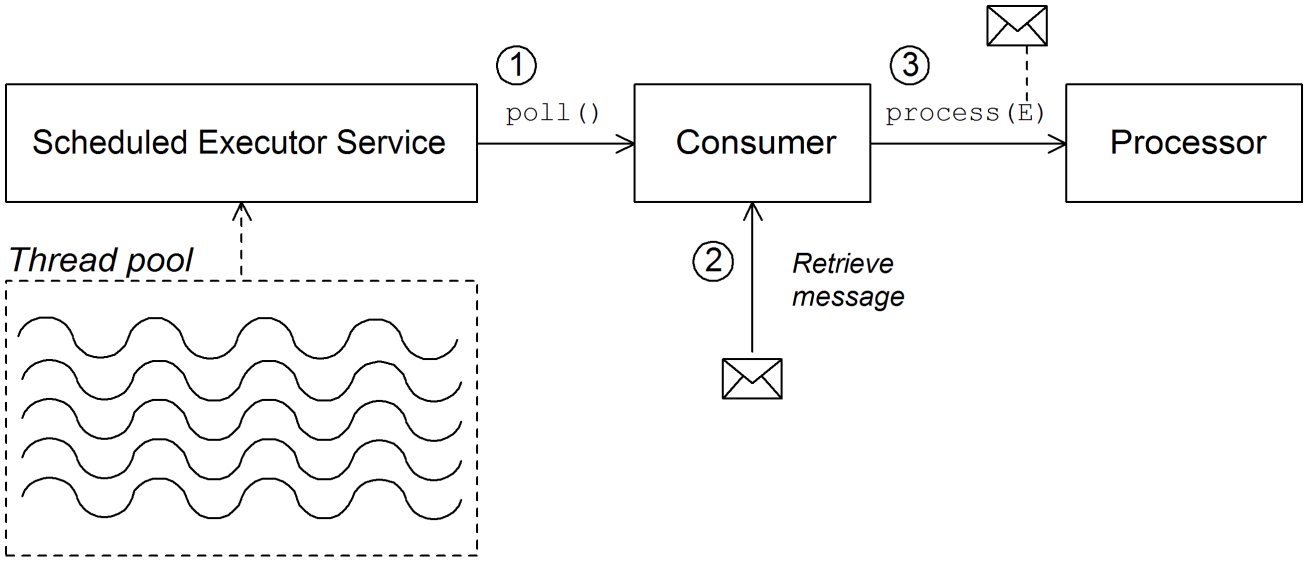
2. *在 notify（） 方法的正文中，使用者首先将传入的事件转换为交换对象 E，然后在路由中的下一个处理器调用 process（），并将交换对象作为参数传递。*

调度的轮询模式

在计划的轮询模式中，使用者通过定期检查间隔来检索传入的请求，无论请求是否到达。内置的计时器类（调度的 executor 服务）会自动检查请求，这是 java.util.concurrent 库提供的标准模式。调度的 executor 服务在固定间隔时执行特定的任务，还可管理一个线程池，用于运行任务实例。

图 38.4 “调度的 Poll Consumer” 显示计划的轮询消费者模式的概要。

图 38.4. 调度的 Poll Consumer



调度的轮询消费者处理传入请求，如下所示：

1. *调度的 executor 服务有一个线程池，可用于启动消费者处理。在各个调度的时间间隔过后，调度的 executor 服务会尝试从池中获取可用线程（默认情况下，池中有五个线程）。如果可用线程可用，它将使用该线程在消费者上调用 poll（） 方法。*

2. *使用者的 poll（） 方法用于触发传入请求的处理。在 poll（） 方法的正文中，使用者会尝试检索传入的消息。如果没有请求可用，则 poll（） 方法会立即返回。*

3. *如果请求消息可用，则使用者将其插入到交换对象中，然后在路由中的下一个处理器调用 process（），并将交换对象传递至其参数。*

轮询模式

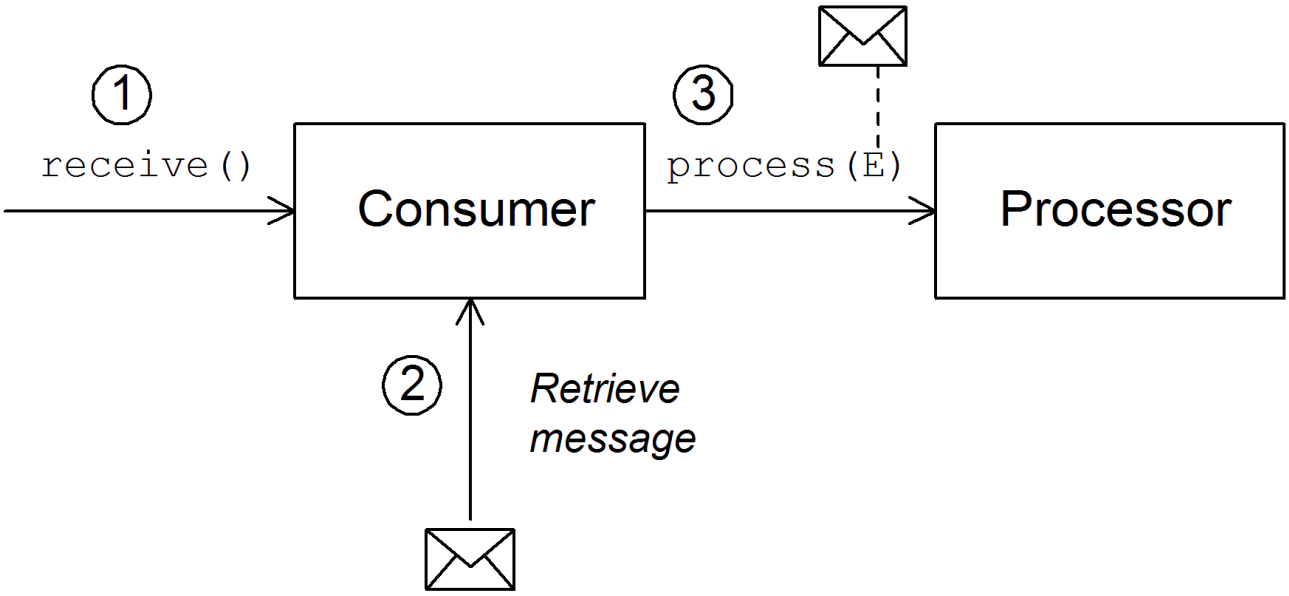
在轮询模式中，当第三方调用一个消费者轮询方法时，启动传入请求的处理：

- *receive()*
- *receiveNoWait()*
- *接收（长超时）*

组件实施过程由组件实施来定义精确的机制，以发起对轮询方法的调用。这种机制在轮询模式中没有指定。

图 38.5 “polling Consumer” 显示了轮询消费者模式的概要。

图 38.5. polling Consumer



轮询消费者按如下方式处理传入请求：

1. 每当称为消费者的轮询方法之一时，启动传入请求的处理。调用这些轮询方法的机制由组件实施定义。

2. 在 receive() 方法的正文中，使用者会尝试检索传入的请求消息。如果目前没有可用的消息，则行为取决于调用哪个接收方法。

   - receiveNoWait() returns immediately
   - 接收（长超时） 等待指定的超时间隔[2] 在返回前
   - receive() 等待直到收到消息

3. 如果请求消息可用，则使用者将其插入到交换对象中，然后在路由中的下一个处理器调用

process（），并将交换对象传递至其参数。

38.1.4. 异步处理

概述

在处理交换时，制作者端点通常遵循 同步 模式。当制作者上的管道调用 process（） 中的前面的处理器时，process（） 方法将阻止，直到收到回复。在这种情况下，处理器的线程将会被阻止，直到制作者完成发送请求并接收回复的生命周期。

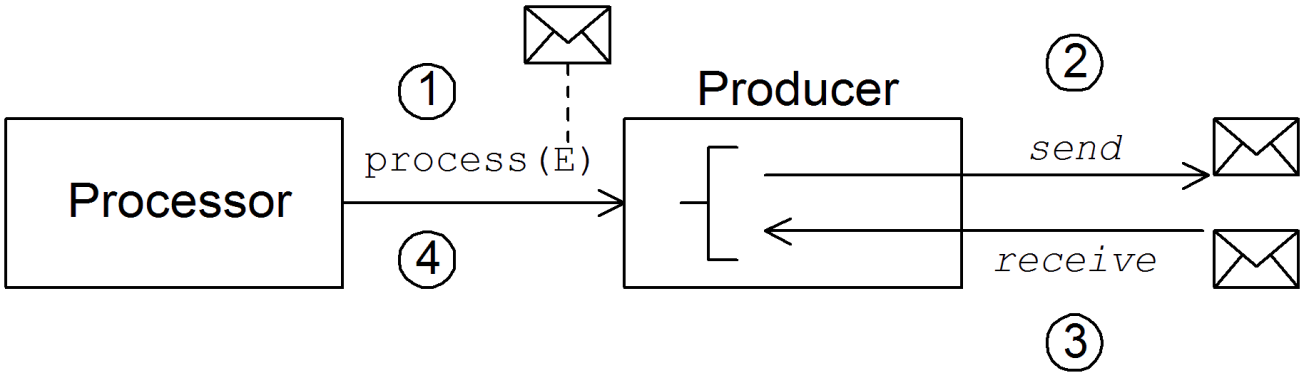
但有时候，您可能更喜欢将前面的处理器与制作者分离，从而使处理器的线程会立即释放，并且 process（） 调用 不会阻止。在这种情况下，您应该使用 异步 模式实施制作者，为前面的处理器提供调用 process（） 方法的非阻塞版本的选项。

为了让您了解不同的实现选项，本节描述了实施制作者端点的同步和异步模式。

同步制作者

图 38.6 “synchronous Producer” 显示同步制作者的概要，其中前面的处理器块直到制作者完成交换过程。

图 38.6. synchronous Producer



同步制作者会按如下方式处理交换：

1. 管道中的前面的处理器调用制作者上的 synchronous process（） 方法，以启动同步处理。同步 process（） 方法采用单一交换参数。

2. 在 process（） 方法的正文中，生产者将请求（在消息中 ）发送到端点。

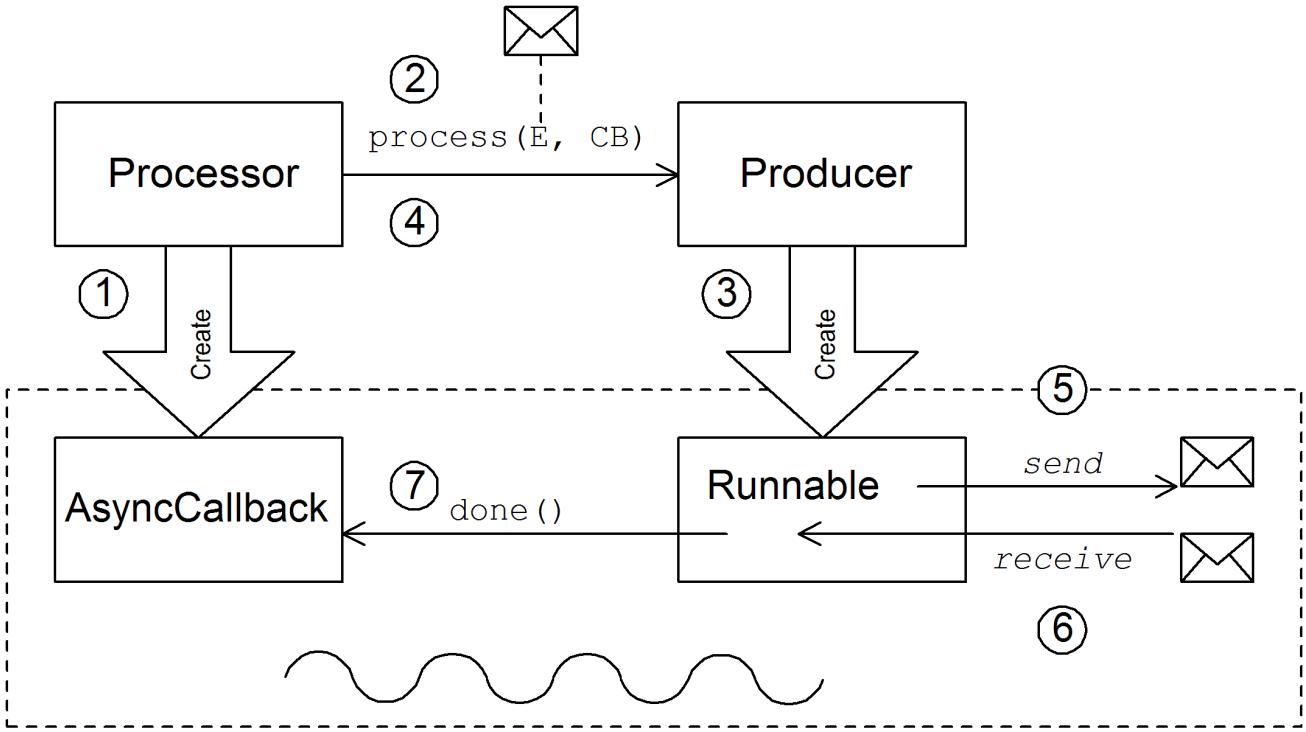
3. 如果交换模式需要，生产者会等待回复（出 消息）从端点到达。此步骤可能会导致 process（） 方法无限期阻止。但是，如果交换模式没有强制回复，process（） 方法可以在发送请求后立即返回。

4. 当 process（） 方法返回时，exchange 对象包含来自同步调用（ Out 消息消息）的回复。

异步制作者

图 38.7 “asynchronous Producer” 展示了异步制作者的概要，其中生产者在子线程中处理交换，而前面的处理器在一定时间内不会被阻止。

图 38.7. asynchronous Producer



异步制作者以如下方式处理交换：

1. 在处理器可以调用异步 process（） 方法之前，它必须创建一个 异步回调 对象，该对象负责处理路由返回部分的交换。对于异步回调，处理器必须实施从 AsyncCallback 接口继承的类。

2. 处理器调用制作者上的异步 process（） 方法，以启动异步处理。异步 process（） 方法采用两个参数：

- *Exchange 对象*
- *同步回调对象*

3. *在 process（） 方法的正文中，制作者会创建一个 可运行的 对象来封装处理代码。然后，制作者将这个 Runnable 对象的 执行委托给子线程。*

4. *异步 process（） 方法返回，从而释放处理器线程。交换处理将继续处于单独的子线程中。*

5. *Runnable 对象将 In 消息发送到端点。*

6. *如果交换模式需要，Run nable 对象会等待回复（Out 或 Failure 消息）来到达端点。Runnable 对象 会一直被阻止，直到收到回复为止。*

7. *在回复到达后，Run nable 对象会将回复（传出消息）插入到交换对象中，然后在异步回调对象上调用 done（）。然后，异步回调会负责处理回复消息（在子线程中执行）。*

38.2. 如何实施组件

概述

本节概述了实施自定义 Apache Camel 组件所需的步骤。

您需要实施哪些接口？

在实施组件时，通常需要实施以下 Java 接口：

- *org.apache.camel.Component*
- *org.apache.camel.Endpoint*

- *org.apache.camel.Consumer*
- *org.apache.camel.Producer*

另外，也需要实施以下 Java 接口：

- *org.apache.camel.Exchange*
- *org.apache.camel.Message*

实施步骤

您通常实施自定义组件，如下所示：

1. *实施组件接口 abrt- theA 组件对象充当端点工厂。您扩展 DefaultComponent 类，并实施 createEndpoint（） 方法。*

   请参阅 第 39 章 组件接口。

2. *实施 Endpoint 接口 10.10.10.2- theAn 端点代表由特定 URI 标识的资源。实施端点时采取的方法取决于消费者是否遵循 事件驱动的 模式、调度轮询 模式或 轮询 模式。 对于事件驱动的模式，通过扩展 DefaultEndpoint 类并实施以下方法来实现端点：*

   - *createProducer()*
   - *createConsumer()*

   对于调度的轮询模式，通过扩展 ScheduledPollEndpoint 类并实施以下方法来实现端点：

   - *createProducer()*

- *createConsumer()*

  对于轮询模式，通过扩展 DefaultPollingEndpoint 类并实施以下方法来实现端点：

- *createProducer()*

- *createPollConsumer()*

  请参阅 第 40 章 端点接口。

3. *根据您需要 实施 的模式（事件驱动的、调度轮询或轮询），实施消费者，实施使用者可能要实施以下几项不同的方法。消费者实施对于确定用于处理消息交换的线程模型而言也非常重要。*

   请参阅 第 41.2 节 “实现使用者接口”。

4. *实施 Producer 接口 & gt;_<-jaxbTo 实施制作者，您要扩展 DefaultProducer 类并实施 process（） 方法。*

   请参阅 第 42 章 producer Interface。

5. *（可选）实现 Exchange 或 Message 接口 的 Message 接口 the default implementations of Exchange and Message 可以直接使用，但偶尔，您可能会发现有必要自定义这些类型。*

   请参阅 第 43 章 Exchange Interface 和 第 44 章 消息接口。

安装和配置组件

您可以使用以下方法之一安装自定义组件：

- *将组件直接添加到 CamelContext - the CamelContext.addComponent（） 方法，以编程方式添加组件。*

- *使用 Spring 配置 IFL-The standard Spring bean 元素创建组件实例，添加 组件实例。bean 的 id 属性隐式地定义组件前缀。详情请查看 第 38.3.2 节 “配置组件”。*

- 将 Apache Camel 配置为自动发现组件 KUBECONFIG -abrtAuto-discovery，确保 Apache Camel 根据需要自动加载组件。详情请查看 第 38.3.1 节 “设置 Auto-Discovery”。

38.3. AUTO-DISCOVERY 和 CONFIGURATION

38.3.1. 设置 Auto-Discovery

概述

自动发现是一种机制，可让您动态将组件添加到 Apache Camel 应用程序。组件 URI 前缀用作按需加载组件的密钥。例如，如果 Apache Camel 遇到端点 URI、activemq://MyQName，并且尚未加载 ActiveMQ 端点，Apache Camel 将搜索由 activemq 前缀标识的组件，并且动态加载组件。

组件类的可用性

在配置自动发现前，您必须确保可以从当前类路径访问您的自定义组件类。通常，您可以将自定义组件类捆绑到 JAR 文件中，并将 JAR 文件添加到您的 classpath 中。

配置自动发现

要启用组件的自动发现，请创建一个名为 的 Java 属性文件，并以 组件前缀、组件前缀 并将该文件存储在以下位置：

```
/META-INF/services/org/apache/camel/component/component-prefix
```

component-prefix 属性文件必须包含以下属性设置：

```
class=component-class-name
```

其中 component-class-name 是自定义组件类的完全限定域名。您还可以在此文件中定义额外的系统属性设置。

示例

例如，您可以通过创建以下 Java 属性文件来为 Apache Camel FTP 组件启用自动发现：

```
/META-INF/services/org/apache/camel/component/ftp
```

包含以下 Java 属性设置：

```
class=org.apache.camel.component.file.remote.RemoteFileComponent
```

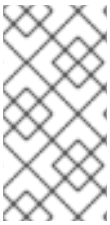
注意

FTP 组件的 Java 属性文件已在 JAR 文件中定义为 camel-ftp-Version.jar。

38.3.2. 配置组件

概述

您可以通过在 Apache Camel Spring 配置文件 META-INF/spring/camel-context.xml 中配置组件来添加组件。要查找组件，组件的 URI 前缀与 Spring 配置中 bean 元素的 ID 属性匹配。如果组件前缀与 bean 元素 ID 匹配，Apache Camel 会实例化引用的类，并注入 Spring 配置中指定的属性。



注意

这种机制的优先级高于自动发现。如果 CamelContext 找到具有 requisite ID 的 Spring bean，则不会尝试使用自动发现来查找组件。

在组件类上定义 bean 属性

如果要注入组件类的任何属性，请将它们定义为 bean 属性。例如：

```
public class CustomComponent extends
  DefaultComponent<CustomExchange> {
    ...
    PropType getProperty() { ... }
    void setProperty(PropType v) { ... }
}
```

getProperty（） 方法和 setProperty（） 方法访问 属性 的值。

在 Spring 中配置组件

要在 Spring 中配置组件，请编辑配置文件 META-INF/spring/camel-context.xml，如 例 38.1 “在

Spring 中配置组件”所示。

例 38.1. 在 Spring 中配置组件

```
<?xml version="1.0" encoding="UTF-8"?>
<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-2.0.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

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

  <bean id="component-prefix" class="component-class-name">
    <property name="property" value="propertyValue"/>
   </bean>
</beans>
```

带有 ID component-prefix 的 bean 元素配置 component-class-name 组件。您可以使用属性元素将属性注入到组件实例中。例如，上例中的 attribute 元素通过在组件上调用 setProperty（） 来将 值 property Value 注入属性。

例子

例 38.2 “JMS 组件 Spring 配置”演示了如何通过定义 ID 等于 jms 的 Bean 元素来配置 Apache Camel 的 JMS 组件的示例。这些设置添加到 Spring 配置文件 camel-context.xml 中。

例 38.2. JMS 组件 Spring 配置

```
<?xml version="1.0" encoding="UTF-8"?>
<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-2.0.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <package>org.apache.camel.example.spring</package> (1)
  </camelContext>

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent"> (2)
    <property name="connectionFactory" (3)
```

```
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL"
                value="vm://localhost?broker.persistent=false&amp;broker.useJmx=false"/> 4
    </bean>
  </property>
 </bean>
</beans>
```

1

CamelContext 自动实例化在指定 Java 软件包中找到的任何 RouteBuilder 类，即 org.apache.camel.spring。

2

带有 ID 为 jms 的 bean 元素配置 JMS 组件。bean ID 对应于组件的 URI 前缀。例如，如果路由指定带有 URI、jms://MyQName 的端点，Apache Camel 将使用 jms bean 元素中的设置自动加载 JMS 组件。

3

JMS 只是用于消息传递服务的打包程序。您必须通过设置 JmsComponent 类的 connectionFactory 属性来指定消息传递系统的聚合实施。

4

在本例中，JMS 消息服务的整合实施是 Apache ActiveMQ。brokerURL 属性初始化与 ActiveMQ 代理实例的连接，其中的消息代理嵌入到本地 Java 虚拟机(JVM)中。如果 JVM 中还没有代理，ActiveMQ 将使用 options .persistent=false（代理 没有保留消息）和 broker.useJmx=false （代理不会打开 JMX 端口）对其实例化。

---

[2] *超时间隔通常以毫秒为单位指定。*

第 39 章 组件接口

摘要

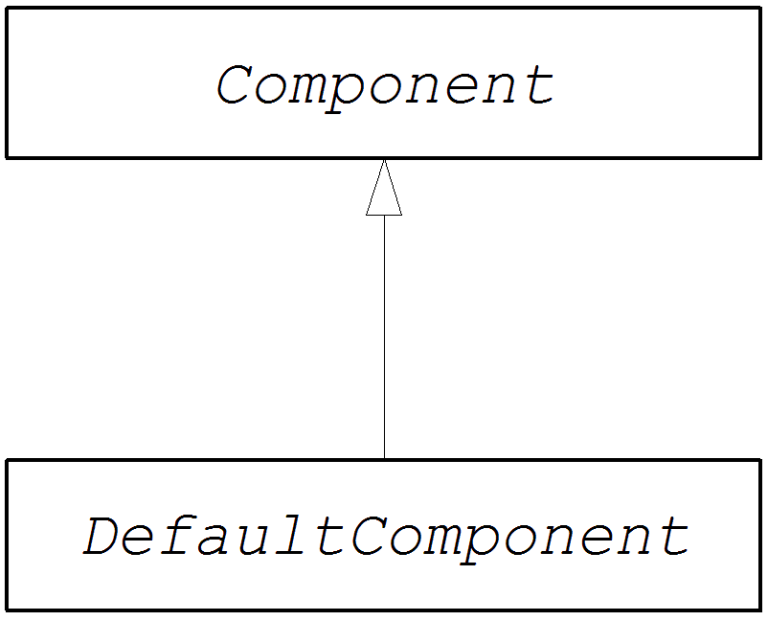
本章论述了如何实现组件接口。

39.1. 组件接口

概述

要实施 Apache Camel 组件，您必须实施 org.apache.camel.Component 接口。组件类型 的实例 提供自定义组件的入口点。也就是说，组件中的其他所有对象最终都可通过 component 实例访问。图 39.1 “组件继承层次结构” 显示构成 组件 继承层次结构的相关 Java 接口和类。

图 39.1. 组件继承层次结构



组件接口

例 39.1 “组件接口” 显示 org.apache.camel.Component 接口的定义。

例 39.1. 组件接口

```
package org.apache.camel;

public interface Component {
    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);
```

```
    Endpoint createEndpoint(String uri) throws Exception;
}
```

组件方法

组件接口定义了以下方法：

- getCamelContext（） 和 setCamelContext（） 时间为这个组件所属的 CamelContext References。当您将组件添加到 CamelContext 时，会自动调用 setCamelContext（） 方法。

- createEndpoint（） the factory 方法被调用，为这个组件创建 Endpoint 实例。uri 参数是端点 URI，其中包含创建端点所需的详细信息。

39.2. 实施组件接口

DefaultComponent 类

您可以通过扩展 org.apache.camel.impl.DefaultComponent 类来实施新的组件，该类为某些方法提供了一些标准功能和默认实施。特别是，DefaultComponent 类支持 URI 解析和创建 调度的 executor（用于调度的轮询模式）。

URI 解析

基本组件接口中定义的 createEndpoint(String uri) 方法使用一个完整的、未解析的端点 URI，作为其唯一参数。另一方面，DefaultComponent 类定义了带有以下签名的 createEndpoint（） 方法的三参数版本：

```
protected abstract Endpoint createEndpoint(
    String uri,
    String remaining,
    Map parameters
)
throws Exception;
```

URI 是原始的、未解析的 URI；剩余的 是 URI 的一部分，该 URI 在开始剥离组件前缀后，并削减查询选项末尾的查询选项；参数 包含解析的查询选项。这是从 DefaultComponent 继承时必须覆盖的 createEndpoint（） 方法的这个版本。具有已为您解析端点 URI 的优点。

以下文件组件的端点 URI 示例显示了 URI 解析在实践中如何工作：

```
file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
```

对于此 URI，以下参数将传递给 createEndpoint（） 的三参数版本：

参数	示例值
uri	file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
剩余	**/tmp/messages/foo**
parameters	在 **java.util.Map** 中设置两个条目： • 参数 **删除** 是布尔值 **true** • 参数 **moveName Postfix** 具有字符串值 **.old**。

参数注入

默认情况下，从 URI 查询选项中提取的参数会在端点的 bean 属性中注入。DefaultComponent 类会自动注入您的参数。

例如，如果要定义支持两个 URI 查询选项的自定义端点：delete 和 moveName Postfix。您必须执行的所有操作是在端点类中定义对应的 bean 方法（getter 和 setters）：

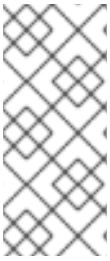
```
public class FileEndpoint extends ScheduledPollEndpoint {
    ...
    public boolean isDelete() {
        return delete;
    }
    public void setDelete(boolean delete) {
        this.delete = delete;
    }
    ...
    public String getMoveNamePostfix() {
        return moveNamePostfix;
    }
    public void setMoveNamePostfix(String moveNamePostfix) {
        this.moveNamePostfix = moveNamePostfix;
    }
}
```

也可以将 URI 查询选项注入 消费者 参数。详情请查看 “consumer 参数注入”一节。

禁用端点参数注入

如果您的 Endpoint 类上没有定义参数，您可以通过禁用端点参数注入来优化端点创建过程。要在端点中禁用参数注入，请覆盖 useIntrospectionOnEndpoint（） 方法并把它返回 false，如下所示：

```
protected boolean useIntrospectionOnEndpoint() {
  return false;
}
```



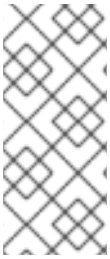
注意

useIntrospectionOnEndpoint（） 方法 不会影响在 Consumer 类上执行的参数注入。位于该级别的参数注入由 Endpoint.configureProperties（） 方法控制（请参阅 第 40.2 节 “实施端点接口”）。

调度的 executor 服务

调度的 executor 在调度的轮询模式中使用，该模式负责推动消费者端点的定期轮询（调度的 executor 实际上是线程池实施）。

要实例化调度的 executor 服务，请使用 CamelContext.getExecutorServiceStrategy（） 方法返回的 ExecutorServiceStrategy 对象。有关 Apache Camel 线程模型的详情，请参考 第 2.8 节 “线程模型”。



注意

在 Apache Camel 2.3 之前，DefaultComponent 类提供了一个 getExecutorService（） 方法来创建线程池实例。从 2.3 起，创建线程池现在由 ExecutorServiceStrategy 对象集中管理。

验证 URI

如果要在创建端点实例前验证 URI，您可以覆盖 DefaultComponent 类中的 validateURI（） 方法，该类具有以下签名：

```
protected void validateURI(String uri,
                           String path,
```

```
                Map parameters)
    throws ResolveEndpointFailedException;
```

如果提供的 URI 没有所需格式，则 validateURI（） 的实施应抛出 org.apache.camel.ResolveEndpointFailedException 异常。

创建端点

例 39.2 “createEndpoint（）的实现” 概述了如何实施 DefaultComponent.createEndpoint（） 方法，它负责根据需要创建端点实例。

例 39.2. createEndpoint（）的实现

```
public class CustomComponent extends DefaultComponent { 1
    ...
    protected Endpoint createEndpoint(String uri, String remaining, Map parameters) throws
Exception { 2
        CustomEndpoint result = new CustomEndpoint(uri, this); 3
        // ...
        return result;
    }
}
```

1

CustomComponent 是自定义组件类的名称，通过扩展 DefaultComponent 类来定义。

2

在扩展 DefaultComponent 时，您必须使用三个参数实施 createEndpoint（） 方法（请参阅 “URI 解析”一节）。

3

通过调用其构造器来创建自定义端点类型 CustomEndpoint 的实例。至少，此构造器获取原始 URI 字符串、uri 以及对此组件实例的引用。

示例

例 39.3 “文件编译实施” 显示了一个 FileComponent 类实施示例。

例 39.3. 文件编译实施

```
```

```
package org.apache.camel.component.file;

import org.apache.camel.CamelContext;
import org.apache.camel.Endpoint;
import org.apache.camel.impl.DefaultComponent;

import java.io.File;
import java.util.Map;

public class FileComponent extends DefaultComponent {
    public static final String HEADER_FILE_NAME = "org.apache.camel.file.name";

    public FileComponent() { (1)
    }

    public FileComponent(CamelContext context) { (2)
        super(context);
    }

    protected Endpoint createEndpoint(String uri, String remaining, Map parameters) throws
Exception { (3)
        File file = new File(remaining);
        FileEndpoint result = new FileEndpoint(file, uri, this);
        return result;
    }
}
```

(1) 始终为组件类定义 **no-argument constructor**，以便自动实例化该类。

(2) 在创建组件实例时，使用父 **CamelContext** 实例的构造器可以方便使用。

(3) **FileComponent.createEndpoint（）** 方法的实现遵循 *例 39.2 “createEndpoint（）的实现”* 中描述的模式。该实施将创建 **FileEndpoint** 对象。

SynchronizationRouteAware Interface

SynchronizationRouteAware 接口允许您在交换之前和路由后具有回调。

- **onBeforeRoute**：在给定路由路由前检查交换。但是，如果在启动路由后，如果您将 **SynchronizationRouteAware** 实现添加到 单元OfWork，则此回调可能无法被调用。

- ***onAfterRoute: Invoked** 在给定路由被路由交换后。但是，如果交换通过多个路由进行路由，它将为每个路由生成调用后端。*

  这个调用会在这些回调前发生：

  a. *该路由的使用者将任何响应写入调用者（如果为 **InOut** 模式）*

  b. ***unit OfWork** 通过调用 **Synchronization.onComplete(org.apache.camel.Exchange)** 或 **Synchronization.onFailure(org.apache.camel.Exchange)***

第 40 章 端点接口

摘要

本章论述了如何实施 Endpoint 接口，这是实施 Apache Camel 组件中的基本步骤。

40.1. 端点接口

概述

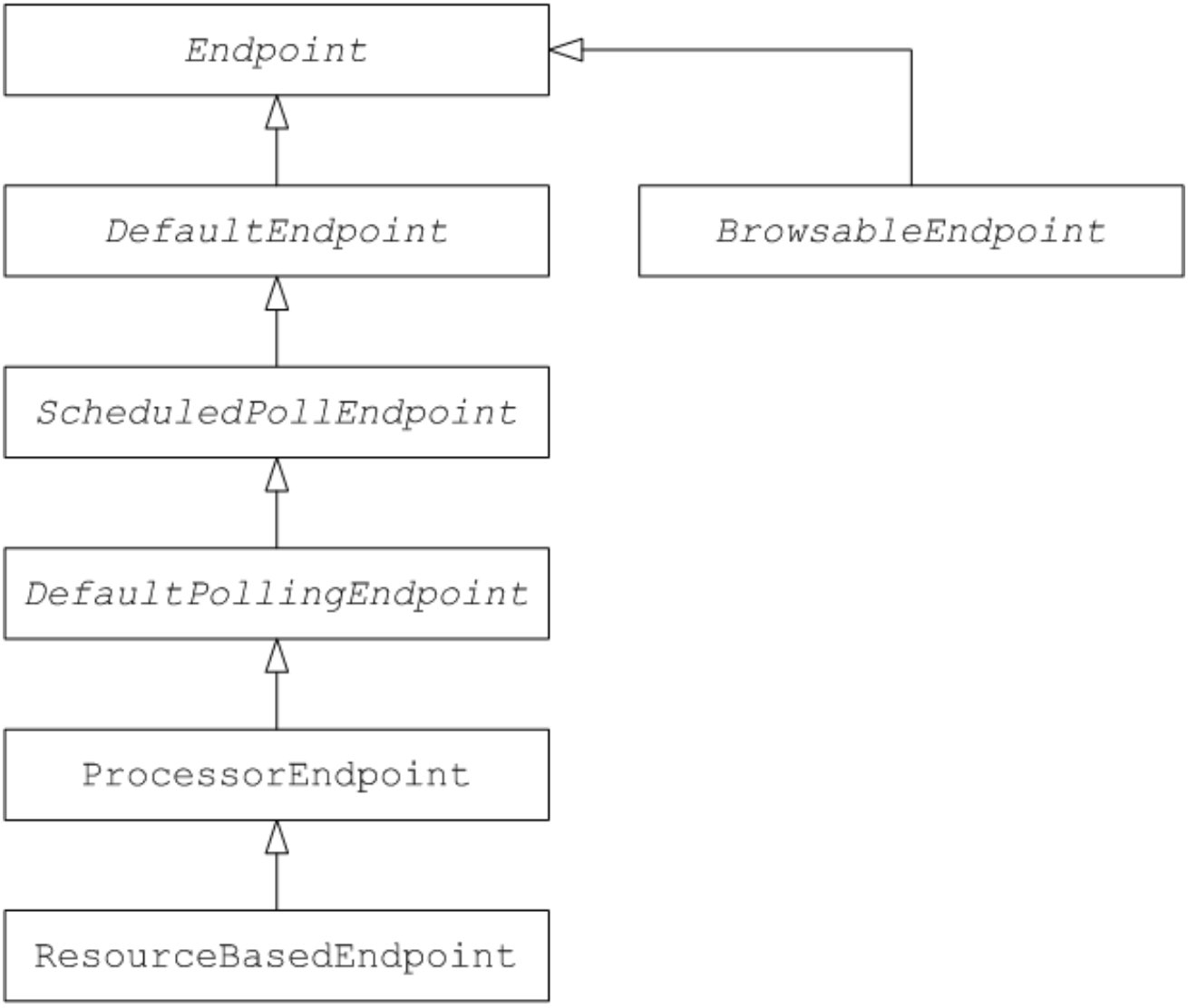
org.apache.camel.Endpoint 类型的实例封装了端点 URI，它也充当 消费者、Producer 和 Exchange 对象的工厂。实施端点的方法有三种：

- Event-driven
- 计划轮询
- 轮询

这些端点实施模式与对应的模式补充，用于实施使用者","-abrtsee 第 41.2 节 “实现使用者接口”。

图 40.1 “端点继承层次结构” 显示构成 端点 继承层次结构的相关 Java 接口和类。

图 40.1. 端点继承层次结构



Endpoint 接口

例 40.1 “端点接口” 显示 **org.apache.camel.Endpoint 接口的定义。**

例 40.1. 端点接口

```
package org.apache.camel;

public interface Endpoint {
    boolean isSingleton();

    String getEndpointUri();

    String getEndpointKey();

    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    void configureProperties(Map options);
```

```
    boolean isLenientProperties();

    Exchange createExchange();
    Exchange createExchange(ExchangePattern pattern);
    Exchange createExchange(Exchange exchange);

    Producer createProducer() throws Exception;

    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

端点方法

Endpoint 接口定义了以下方法：

- 代表Singleton（） InventoryService-abrtRe returns true，如果您要确保每个 URI 映射到 CamelContext 中的一个端点。当此属性为 true 时，指向路由中相同 URI 的多个引用始终引用单个 端点实例。另一方面，当此属性为 false 时，对路由中同一 URI 的多个引用指的是 不同的端点实例。每次引用路由中的 URI 时，都会创建一个新的端点实例。
- getEndpointUri（） abrt- the endpointRe 返回此端点的端点 URI。
- 在注册端点时，由 org.apache.camel.spi.LifecycleStrategy 的 getEndpointKey（） Infoblox- InfobloxUsed by org.apache.camel.spi.LifecycleStrategy。
- getCamelContext（）进行对该端点所属的 CamelContext 实例的引用。
- 设置此端点 所属的 CamelContext Set。
- 配置Properties（） >_<-jaxbStores 是参数映射的副本，用于在创建新 Consumer 实例时注入参数。
- isLenientProperties（） 时间为 true，表示允许 URI 包含未知参数（即，不能在 Endpoint 或 Consumer 类上注入的参数）。通常，应实施此方法来返回 假。
- 使用以下变体 创建Exchange（） 3.10.0-- theAn overloaded 方法：

- Exchange createExchange（） Infoblox-jaxbCreates a new exchange pattern 设置。

- Exchange createExchange(ExchangePattern pattern) abrt-abrtCreates a new exchange instance with specified exchange pattern.

- Exchange createExchange(Exchange exchange) the given exchange 参数到此端点所需的交换类型。如果给定交换尚未正确类型，则此方法将其复制到正确类型的新实例中。DefaultEndpoint 类中提供了此方法的默认实现。

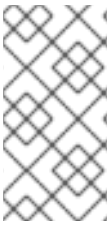
- createProducer（） >_<->_<Factory 方法 用于创建新的 Producer 实例。

- createConsumer（） 10.10.10.2-CustomizationFactory 方法，以创建新的事件驱动的消费者实例。processor 参数是对路由中的第一个处理器的引用。

- 创建PollingConsumer（）10.10.10.2 ->_<Factory 方法，以创建新的轮询消费者实例。

端点单例

为了避免不必要的开销，最好为具有相同 URI（通过 CamelContext）的所有端点创建一个端点实例。您可以通过实施 isSingleton（） 来返回 true 来强制实施此条件。



注意

在这个上下文中，同一 URI 意味着，使用字符串相等时两个 URI 相同。在原则上，可以有两个等效的 URI，虽然由不同的字符串表示。在这种情况下，URI 不会像一样对待。

40.2. 实施端点接口

实现端点的替代方法

支持以下替代端点实现模式：

- 事件驱动的端点实现

- *调度轮询端点实现*
- *轮询端点实现*

事件驱动的端点实现

*如果您的自定义端点符合事件驱动的模式（请参阅 第 38.1.3 节 “消费者模式和线程”），它通过扩展抽象类来实现，则 **org.apache.camel.impl.DefaultEndpoint**，如 例 40.2 “实施 DefaultEndpoint” 所示。*

例 40.2. 实施 DefaultEndpoint

```
import java.util.Map;
import java.util.concurrent.BlockingQueue;

import org.apache.camel.Component;
import org.apache.camel.Consumer;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;
import org.apache.camel.impl.DefaultExchange;

public class CustomEndpoint extends DefaultEndpoint { (1)

    public CustomEndpoint(String endpointUri, Component component) { (2)
        super(endpointUri, component);
        // Do any other initialization...
    }

    public Producer createProducer() throws Exception { (3)
        return new CustomProducer(this);
    }

    public Consumer createConsumer(Processor processor) throws Exception { (4)
        return new CustomConsumer(this, processor);
    }

    public boolean isSingleton() {
        return true;
    }

    // Implement the following methods, only if you need to set exchange properties.
    //
    public Exchange createExchange() { (5)
        return this.createExchange(getExchangePattern());
    }

    public Exchange createExchange(ExchangePattern pattern) {
```

```
        Exchange result = new DefaultExchange(getCamelContext(), pattern);
        // Set exchange properties
        ...
        return result;
    }
}
```

1

通过扩展 DefaultEndpoint 类来实施事件驱动的自定义端点 CustomEndpoint。

2

您必须至少有一个构造器使用端点 URI、endpointUri 和父组件引用、组件 作为参数。

3

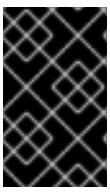
实施 createProducer（） factory 方法来创建制作者端点。

4

实施 createConsumer（） 工厂方法来创建事件驱动的消费者实例。

5

通常，不需要 覆盖 createExchange（） 方法。从 DefaultEndpoint 继承的实现会默认创建一个 DefaultExchange 对象，它可以用于任何 Apache Camel 组件。然而，如果您需要在 DefaultExchange 对象中初始化一些交换属性，则最好覆盖此处的 createExchange（） 方法来添加 Exchange 属性设置。



重要

不要 覆盖 createPollingConsumer（） 方法。

DefaultEndpoint 类提供以下方法的默认实现，您可以在编写自定义端点代码时发现这些实施可能很有用：

- getEndpointUri（） the endpoint URI。
- getCamelContext（） 进行对 CamelContext 的引用。

- *getComponent（） 3.10.0-- the return returns 引用父组件。*

- *创建PollingConsumer（）>_<-对称Creates 轮询消费者。创建的轮询消费者功能取决于事件驱动的消费者。如果覆盖事件驱动的消费者方法，请创建Consumer（），您可以获得轮询消费者的实施。*

- *CreateExchange(Exchange e) the given exchange 对象 e to the given exchange 对象 e，指向该端点所需的类型。此方法使用覆盖的 createExchange（） 端点来创建新的端点。这可确保该方法也可用于自定义交换类型。*

调度轮询端点实现

如果您的自定义端点符合调度的轮询模式（请参阅 第 38.1.3 节 “消费者模式和线程”），它通过从抽象类继承来实现（请参阅 例 40.3 “ScheduledPollEndpoint 实现”，如 所示。

例 40.3. ScheduledPollEndpoint 实现

```
import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.ExchangePattern;
import org.apache.camel.Message;
import org.apache.camel.impl.ScheduledPollEndpoint;

public class CustomEndpoint extends ScheduledPollEndpoint { (1)

  protected CustomEndpoint(String endpointUri, CustomComponent component) { (2)
    super(endpointUri, component);
    // Do any other initialization...
  }

  public Producer createProducer() throws Exception { (3)
    Producer result = new CustomProducer(this);
    return result;
  }

  public Consumer createConsumer(Processor processor) throws Exception { (4)
    Consumer result = new CustomConsumer(this, processor);
    configureConsumer(result); (5)
    return result;
  }

  public boolean isSingleton() {
    return true;
  }

  // Implement the following methods, only if you need to set exchange properties.
```

```
    //
    public Exchange createExchange() { 6
        return this.createExchange(getExchangePattern());
    }

    public Exchange createExchange(ExchangePattern pattern) {
        Exchange result = new DefaultExchange(getCamelContext(), pattern);
        // Set exchange properties
        ...
        return result;
    }
}
```

1

通过扩展 ScheduledPollEndpoint 类来实施调度的轮询自定义端点 CustomEndpoint。

2

您必须至少有一个构造器使用端点 URI、endpointUri 和父组件引用、组件 作为参数。

3

实施 createProducer（） factory 方法来创建制作者端点。

4

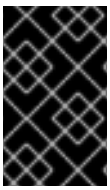
实施 createConsumer（） 工厂方法来创建调度的轮询消费者实例。

5

ScheduledPollEndpoint 基本类中定义的 configureConsumer（） 方法负责将使用者查询选项注入使用者。请参阅 “consumer 参数注入”一节。

6

通常，不需要 覆盖 createExchange（） 方法。从 DefaultEndpoint 继承的实现会默认创建一个 DefaultExchange 对象，它可以用于任何 Apache Camel 组件。然而，如果您需要在 DefaultExchange 对象中初始化一些交换属性，则最好覆盖此处的 createExchange（） 方法来添加 Exchange 属性设置。



重要

不要 覆盖 createPollingConsumer（） 方法。

轮询端点实现

如果您的自定义端点遵循轮询消费者模式（请参阅 第 38.1.3 节 “消费者模式和线程”），它通过从抽象类继承来实现的，则 org.apache.camel.impl.DefaultPollingEndpoint，如 例 40.4 “DefaultPollingEndpoint Implementation” 所示。

例 40.4. DefaultPollingEndpoint Implementation

```
import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.ExchangePattern;
import org.apache.camel.Message;
import org.apache.camel.impl.DefaultPollingEndpoint;

public class CustomEndpoint extends DefaultPollingEndpoint {
    ...
    public PollingConsumer createPollingConsumer() throws Exception {
        PollingConsumer result = new CustomConsumer(this);
        configureConsumer(result);
        return result;
    }

    // Do NOT implement createConsumer(). It is already implemented in DefaultPollingEndpoint.
    ...
}
```

由于此 CustomEndpoint 类是一个轮询端点，您必须实施 createPollingConsumer（） 方法，而不是 createConsumer（） 方法。从 createPollingConsumer（） 返回的消费者实例必须从 PollingConsumer 接口继承。有关如何实施轮询消费者的详情，请参考 “轮询消费者实施”一节。

除了 createPollingConsumer（） 方法的实施步骤外，实施 DefaultPollingEndpoint 的步骤与实施 ScheduledPollEndpoint 的步骤类似。详情请查看 例 40.3 “ScheduledPollEndpoint 实现”。

实施 BrowsableEndpoint 接口

如果要公开当前端点中待处理的交换实例列表，您可以实现 org.apache.camel.spi.BrowsableEndpoint 接口，如 例 40.5 “BrowsableEndpoint 接口” 所示。如果端点执行某种形式的传入事件，则需要实施这个接口。例如，Apache Camel SEDA 端点实施 BrowsableEndpoint 接口>_<-KUBECONFIGsee 例 40.6 “SedaEndpoint 实施”。

例 40.5. BrowsableEndpoint 接口

```
package org.apache.camel.spi;
```

```
import java.util.List;

import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;

public interface BrowsableEndpoint extends Endpoint {
    List<Exchange> getExchanges();
}
```

示例

例 40.6 “SedaEndpoint 实施” 显示 **SedaEndpoint** 的示例实现。**SEDA** 端点是一个由 事件驱动的端点的示例。传入的事件存储在 **FIFO** 队列中（ **java.util.concurrent.BlockingQueue**的实例）和 **SEDA** 使用者启动线程以读取和处理事件。事件本身由 **org.apache.camel.Exchange** 对象表示。

例 40.6. SedaEndpoint 实施

```
package org.apache.camel.component.seda;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.BlockingQueue;

import org.apache.camel.Component;
import org.apache.camel.Consumer;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;
import org.apache.camel.spi.BrowsableEndpoint;

public class SedaEndpoint extends DefaultEndpoint implements BrowsableEndpoint { (1)
    private BlockingQueue<Exchange> queue;

    public SedaEndpoint(String endpointUri, Component component, BlockingQueue<Exchange>
queue) { (2)
        super(endpointUri, component);
        this.queue = queue;
    }

    public SedaEndpoint(String uri, SedaComponent component, Map parameters) { (3)
        this(uri, component, component.createQueue(uri, parameters));
    }

    public Producer createProducer() throws Exception { (4)
        return new CollectionProducer(this, getQueue());
    }

    public Consumer createConsumer(Processor processor) throws Exception { (5)
        return new SedaConsumer(this, processor);
```

```
    }

    public BlockingQueue<Exchange> getQueue() { 6
        return queue;
    }

    public boolean isSingleton() { 7
        return true;
    }

    public List<Exchange> getExchanges() { 8
        return new ArrayList<Exchange> getQueue());
    }
}
```

1

SedaEndpoint 类遵循通过扩展 DefaultEndpoint 类来实施事件驱动的端点的模式。SedaEndpoint 类还实施 BrowsableEndpoint 接口，它提供对队列中交换对象列表的访问。

2

按照事件驱动的消费者的常用模式，SedaEndpoint 定义了一个使用端点参数、端点Uri 以及组件引用参数 的构造器。

3

提供了另一个构造器，它将队列创建委派给父组件实例。

4

createProducer（） factory 方法创建 CollectionProducer 的实例，这是向队列添加事件的生产者实现。

5

createConsumer（） 工厂方法创建 SedaConsumer 的实例，它负责从队列拉取事件并进行处理。

6

getQueue（） 方法返回对队列的引用。

7

isSingleton（） 方法返回 true，表示应为每个唯一 URI 字符串创建一个端点实例。

8

第 41 章 消费者接口

摘要

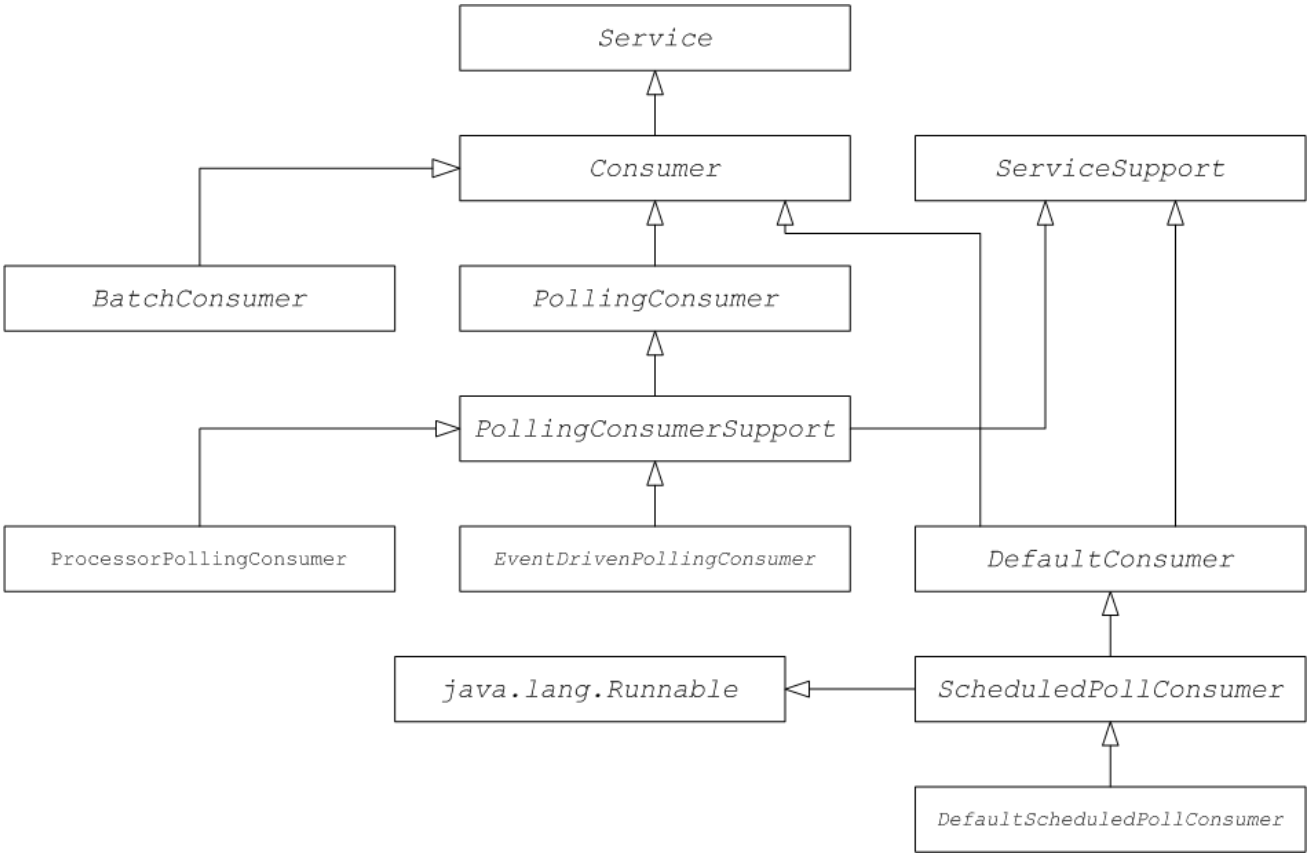
本章论述了如何实施使用者接口，这是实施 Apache Camel 组件中的基本步骤。

41.1. CONSUMER 接口

概述

org.apache.camel.Consumer 类型的实例代表路由中的源端点。实施消费者的方式有多种不同方法（请参阅 第 38.1.3 节 “消费者模式和线程”），这种灵活性在继承分级（查看 图 41.1 “消费者继承层次结构”）中有所反映，其中包括多个不同的基本类来实施消费者。

图 41.1. 消费者继承层次结构



consumer 参数注入

对于遵循调度轮询模式的用户（请参阅 “调度的轮询模式”一节），Apache Camel 为将参数注入消费者实例提供支持。例如，考虑由 自定义 前缀标识的组件的以下端点 URI：

```
custom:destination?consumer.myConsumerParam
```

Apache Camel 支持自动注入 使用者形式的查询选项。*。对于 consumer.myConsumerParam 参数，您需要在 Consumer 实施类上定义对应的 setter 和 getter 方法，如下所示：

```
public class CustomConsumer extends ScheduledPollConsumer {
    ...
    String getMyConsumerParam() { ... }
    void setMyConsumerParam(String s) { ... }
    ...
}
```

如果 getter 和 setter 方法遵循常见的 Java Bean 规范（包括属性名的第一个字母的大写）。

除了在消费者实施中定义 bean 方法外，还必须记住在 Endpoint.createConsumer() 实施中调用 configureConsumer() 方法（请参阅 "调度轮询端点实现"一节。

例 41.1 "FileEndpoint createConsumer() 实施" 显示了一个 createConsumer() 方法实现示例，它取自文件组件中的 FileEndpoint 类：

例 41.1. FileEndpoint createConsumer() 实施

```
...
public class FileEndpoint extends ScheduledPollEndpoint {
    ...
    public Consumer createConsumer(Processor processor) throws Exception {
        Consumer result = new FileConsumer(this, processor);
        configureConsumer(result);
        return result;
    }
    ...
    }
```

在运行时，使用者参数注入功能如下：

1. 创建端点时，DefaultComponent.createEndpoint(String uri) 的默认实现会解析 URI 以提取消费者参数，并通过调用 ScheduledPollEndpoint.configureProperties() 来把它们存储在端点实例中。

2. 调用 createConsumer() 时，方法实施调用 configureConsumer() 以注入消费者参数（请参阅 例 41.1 "FileEndpoint createConsumer() 实施"）。

3. *configureConsumer（） 方法使用 Java 反应来调用名称与 消费者后的相关选项匹配的 setter 方法。 前缀被剥离。*

调度的轮询参数

按照调度的轮询模式的用户自动支持 表 41.1 “调度的轮询参数” 中显示的消费者（它们可能显示为端点 URI 中的查询选项）。

表 41.1. 调度的轮询参数

名称	default	描述
initialDelay	**1000**	延迟（以毫秒为单位）到第一次轮询前。
delay	**500**	取决于 **useFixedDelay** 标志的值（时间单位为毫秒）。
useFixedDelay	**false**	如果为 **false**，则 **delay** 参数被解释为轮询周期。轮询将在 **initialDelay**、**initialDelay+delay**、**initialDelay+2*delay** 等等。 如果为 **true**，则 **delay** 参数将解释为上一执行与下一次执行之间所经过的时间。轮询将在 **initialDelay**、**initialDelay+[*ProcessingTime*]+delay** 等等。其中 *ProcessingTime* 是处理当前线程中的交换对象所需时间。

在事件驱动和轮询消费者间转换

Apache Camel 提供了两种特殊的消费者实施，可用于在事件驱动的消费者和轮询消费者之间转换和发送。提供了以下转换类：

- *org.apache.camel.impl.EventDrivenPollingConsumer 进行在轮询消费者实例中一个事件驱动的消费者。*
- *org.apache.camel.impl.DefaultScheduledPollConsumer abrt-abrtConverts一个轮询消费者到事件驱动的消费者实例中。*

在实践中，这些类用于简化实施端点类型的任务。Endpoint 接口定义了以下两种方法来创建消费者实例：

```
package org.apache.camel;

public interface Endpoint {
    ...
    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

createConsumer（） 返回事件驱动的消费者，创建PollingConsumer（） 返回轮询消费者。您只能实施这些方法。例如，如果您遵循消费者的事件驱动的模式，则实施 createConsumer（） 方法，为创建 PollingConsumer（） 提供方法实施，以简单地引发异常。但是，通过转换类，Apache Camel 可以提供更有用的默认实施。

例如，如果要根据事件驱动的模式实施消费者，您可以通过扩展 DefaultEndpoint 并实施 createConsumer（） 方法来实施端点。createPollingConsumer（） 的实现从 DefaultEndpoint 继承，其定义如下：

```
public PollingConsumer<E> createPollingConsumer() throws Exception {
    return new EventDrivenPollingConsumer<E>(this);
}
```

EventDrivenPollingConsumer 构造器引用事件驱动的消费者，这 实际上将其嵌套，并将其转换为轮询消费者。要实现转换，EventDrivenPollingConsumer 实例缓冲传入的事件，并通过 receive （长超时）和 receive NoWait（） 方法根据需要提供它们。

类似地，如果您按照轮询模式实施消费者，通过扩展 DefaultPollingEndpoint 和实施 createPollingConsumer（） 方法来实施端点。在本例中，createConsumer（） 方法的实现从 DefaultPollingEndpoint 继承，默认的实现会返回一个 DefaultScheduledPollConsumer 实例（将轮询消费者转换为事件驱动的消费者）。

ShutdownPrepared 接口

消费者类可以选择实施 org.apache.camel.spi.ShutdownPrepared 接口，它使您的自定义消费者端点能够接收关闭通知。

例 41.2 “ShutdownPrepared Interface” 显示 ShutdownPrepared 接口的定义。

例 41.2. ShutdownPrepared Interface

```
package org.apache.camel.spi;

public interface ShutdownPrepared {

    void prepareShutdown(boolean forced);

}
```

ShutdownPrepared 接口定义了以下方法：

prepareShutdown

接收通知以一个或两个阶段关闭消费者端点，如下所示：

a. **正常关闭 10.10.10.2 - thewherewhere the parameter 的值为 false。尝试正常清理资源。例如，通过正常停止线程。**

b. **强制关闭 InventoryService-West where the force argument 的值为 true。这意味着关闭超时，因此您必须更加积极地清理资源。这是在进程退出前清理资源的最后一个机会。**

ShutdownAware 接口

消费者类可以选择实施 org.apache.camel.spi.ShutdownAware 接口，该界面与安全关闭机制交互，让消费者要求额外的时间关机。这通常需要这个组件，比如 SEDA，它们可能会使存储在内部队列中的待处理交换。通常，您希望在关闭 SEDA 消费者之前处理队列中的所有交换。

例 41.3 “ShutdownAware Interface” 显示 ShutdownAware 接口的定义。

例 41.3. ShutdownAware Interface

```
// Java
package org.apache.camel.spi;

import org.apache.camel.ShutdownRunningTask;

public interface ShutdownAware extends ShutdownPrepared {

    boolean deferShutdown(ShutdownRunningTask shutdownRunningTask);

    int getPendingExchangesSize();
}
```

ShutdownAware 接口定义以下方法：

deferShutdown

如果您想要延迟用户关闭，请从此方法返回 true。shutdownRunningTask 参数是一个 enum，可取以下值之一：

- ShutdownRunningTask.CompleteCurrentTaskOnly abrt-abrtfinish 处理目前由消费者的线程池处理的交换，但不试图处理超过该交换的交换。
- ShutdownRunningTask.CompleteAllTasks abrt- 时间 处理所有待处理的交换。例如，如果 SEDA 组件，使用者将处理来自其传入队列的所有交换。

getPendingExchangesSize

指明消费者用于处理多少交换。零值表示完成处理，可关闭消费者。

有关如何定义 ShutdownAware 方法的示例，请参考 例 41.7 “自定义线程实施”。

41.2. 实现使用者接口

实施消费者的替代方法

您可以通过以下任一方法实现消费者：

- 事件驱动的消费者实施
- 计划轮询消费者实施
- 轮询消费者实施
- 自定义线程实现

事件驱动的消费者实施

在事件驱动的消费者中，外部事件明确驱动处理。事件通过 event-listener 接口接收，其中监听程序接口特定于特定事件源。

例 41.4 “JMXConsumer 实施” 演示了 JMXConsumer 类的实施，该类取自 Apache Camel JMX 组件实施。JMXConsumer 类是事件驱动的消费者的示例，通过从 org.apache.camel.impl.DefaultConsumer 类继承来实施该类。在 JMXConsumer 示例示例中，事件由 notification Listener.handleNotification（） 方法上的调用表示，这是接收 JMX 事件的标准方法。要接收这些 JMX 事件，需要实施 NotificationListener 接口并覆盖 handleNotification（） 方法，如 例 41.4 “JMXConsumer 实施” 所示。

例 41.4. JMXConsumer 实施

```
package org.apache.camel.component.jmx;

import javax.management.Notification;
import javax.management.NotificationListener;
import org.apache.camel.Processor;
import org.apache.camel.impl.DefaultConsumer;

public class JMXConsumer extends DefaultConsumer implements NotificationListener { 1

    JMXEndpoint jmxEndpoint;

    public JMXConsumer(JMXEndpoint endpoint, Processor processor) { 2
        super(endpoint, processor);
        this.jmxEndpoint = endpoint;
    }

    public void handleNotification(Notification notification, Object handback) { 3
        try {
            getProcessor().process(jmxEndpoint.createExchange(notification)); 4
        } catch (Throwable e) {
            handleException(e); 5
        }
    }
}
```

1

JMXConsumer 模式通过扩展 DefaultConsumer 类，遵循事件驱动的用户的常见模式。此外，由于此使用者旨在接收来自 JMX 通知的事件（由 JMX 通知表示），因此实施 NotificationListener 接口是必须的。

2

3

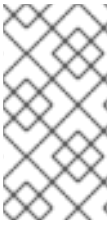
当 JMX 通知到达时，handleNotification（） 方法（在 NotificationListener中定义）会自动调用 JMX。这个方法的正文应该包含执行消费者的事件处理的代码。由于 handleNotification（）调用源自 JMX 层，因此消费者的线程模型由 JMX 层隐式控制，而不是由 JMXConsumer 类控制。

4

这种代码行组合了两个步骤。首先，JMX 通知对象转换为交换对象，这是 Apache Camel 中事件的通用表示。然后，新创建的交换对象会被传递给路由中的下一个处理器（同步）。

5

handleException（） 方法由 DefaultConsumer 基本类实施。默认情况下，它利用 org.apache.camel.impl.LoggingExceptionHandler 类来处理异常。



注意

handleNotification（） 方法特定于 JMX 示例。在实施您自己的事件驱动的消费者时，您必须确定要在自定义消费者中实施的类似事件监听程序方法。

计划轮询消费者实施

在计划的轮询消费者中，轮询事件由计时器类自动生成，java.util.concurrent.ScheduledExecutorService。要接收生成的轮询事件，您必须实施 ScheduledPollConsumer.poll（） 方法（请参阅 第 38.1.3 节 “消费者模式和线程”。

例 41.5 “ScheduledPollConsumer 实现” 演示了如何实施遵循调度轮询模式的使用者，具体是通过扩展 ScheduledPollConsumer 类来实现的。

例 41.5. ScheduledPollConsumer 实现

```
import java.util.concurrent.ScheduledExecutorService;

import org.apache.camel.Consumer;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.PollingConsumer;
import org.apache.camel.Processor;

import org.apache.camel.impl.ScheduledPollConsumer;

public class pass:quotes[CustomConsumer] extends ScheduledPollConsumer { (1)
    private final pass:quotes[CustomEndpoint] endpoint;
```

```
    public pass:quotes[CustomConsumer](pass:quotes[CustomEndpoint] endpoint, Processor
processor) { 2
        super(endpoint, processor);
        this.endpoint = endpoint;
    }

    protected void poll() throws Exception { 3
        Exchange exchange = /* Receive exchange object ... */;

        // Example of a synchronous processor.
        getProcessor().process(exchange); 4
    }

    @Override
    protected void doStart() throws Exception { 5
        // Pre-Start:
        // Place code here to execute just before start of processing.
        super.doStart();
        // Post-Start:
        // Place code here to execute just after start of processing.
    }

    @Override
    protected void doStop() throws Exception { 6
        // Pre-Stop:
        // Place code here to execute just before processing stops.
        super.doStop();
        // Post-Stop:
        // Place code here to execute just after processing stops.
    }
}
```

1

通过扩展 **org.apache.camel.impl.ScheduledPollConsumer** 类来实施计划的轮询类。

2

您必须实施一个构造器，它引用了父端点、端点，以及对链的下一个处理器（处理器）的引用。

3

覆盖 **poll**（） 方法，以接收调度的轮询事件。这是您应该将检索和处理传入事件的代码位置（由交换对象代表）。

4

在本例中，事件被同步处理。如果要异步处理事件，您应该通过调用 **getAsyncProcessor**（） 来使用对异步处理器的引用。有关如何异步处理事件的详情，请参考 第 38.1.4 节 “异步处理”。

5

（可选） 如果要在消费者启动时执行某些代码行，请按如下所示覆盖 doStart（） 方法。

6

（可选） 如果您希望以消费者的身份执行某些代码行是停止的，请按如下所示覆盖 doStop（） 方法。

轮询消费者实施

例 41.6 “PollingConsumerSupport 实现” 概述了如何实施遵循轮询模式的消费者，该模式通过扩展 Polling ConsumerSupport 类来实现。

例 41.6. PollingConsumerSupport 实现

```
import org.apache.camel.Exchange;
import org.apache.camel.RuntimeCamelException;
import org.apache.camel.impl.PollingConsumerSupport;

public class pass:quotes[CustomConsumer] extends PollingConsumerSupport { 1
    private final pass:quotes[CustomEndpoint] endpoint;

    public pass:quotes[CustomConsumer](pass:quotes[CustomEndpoint] endpoint) { 2
        super(endpoint);
        this.endpoint = endpoint;
    }

    public Exchange receiveNoWait() { 3
        Exchange exchange = /* Obtain an exchange object. */;
        // Further processing ...
        return exchange;
    }

    public Exchange receive() { 4
        // Blocking poll ...
    }

    public Exchange receive(long timeout) { 5
        // Poll with timeout ...
    }

    protected void doStart() throws Exception { 6
        // Code to execute whilst starting up.
    }

    protected void doStop() throws Exception {
```

```
        // Code to execute whilst shutting down.
    }
}
```

1

通过扩展 org.apache.camel.impl.PollingConsumerSupport 类来实施轮询消费者类。

2

您必须实施一个构造器，它将引用父端点 端点端点端点 作为参数。轮询消费者不需要对处理器实例的引用。

3

receiveNoWait（） 方法应该实施用于检索事件（exchange 对象）的非阻塞算法。如果没有可用的事件，它应该返回 null。

4

receive（） 方法应该实施用于检索事件的阻塞算法。如果事件不可用，此方法可以无限期地阻止。

5

receive（长超时） 方法实施一个算法，只要指定的超时（通常在毫秒单位指定）。

6

如果要在启动或关闭消费者时插入执行的代码，请分别实施 doStart（） 方法和 doStop（） 方法。

自定义线程实现

如果标准消费者模式不适用于您的消费者实施，您可以直接实现 Consumer 接口，并自行编写线程代码。但是，在编写线程代码时，务必要遵守标准 Apache Camel 线程模型，如 第 2.8 节 “线程模型” 所述。

例如，来自 camel-core 的 SEDA 组件实施自己的消费者线程处理，与 Apache Camel 线程模型一致。例 41.7 “自定义线程实施” 展示了 SedaConsumer 类如何实施其线程的概要。

例 41.7. 自定义线程实施

```
package org.apache.camel.component.seda;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

import org.apache.camel.Consumer;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.ShutdownRunningTask;
import org.apache.camel.impl.LoggingExceptionHandler;
import org.apache.camel.impl.ServiceSupport;
import org.apache.camel.util.ServiceHelper;
...
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * A Consumer for the SEDA component.
 *
 * @version $Revision: 922485 $
 */
public class SedaConsumer extends ServiceSupport implements Consumer, Runnable,
ShutdownAware { 1
    private static final transient Log LOG = LogFactory.getLog(SedaConsumer.class);

    private SedaEndpoint endpoint;
    private Processor processor;
    private ExecutorService executor;
    ...
    public SedaConsumer(SedaEndpoint endpoint, Processor processor) {
        this.endpoint = endpoint;
        this.processor = processor;
    }
    ...

    public void run() { 2
        BlockingQueue<Exchange> queue = endpoint.getQueue();
        // Poll the queue and process exchanges
        ...
    }

    ...
    protected void doStart() throws Exception { 3
        int poolSize = endpoint.getConcurrentConsumers();
        executor = endpoint.getCamelContext().getExecutorServiceStrategy()
            .newFixedThreadPool(this, endpoint.getEndpointUri(), poolSize); 4
        for (int i = 0; i < poolSize; i++) { 5
            executor.execute(this);
        }
        endpoint.onStarted(this);
    }
```

```
    protected void doStop() throws Exception { 6
        endpoint.onStopped(this);
        // must shutdown executor on stop to avoid overhead of having them running
        endpoint.getCamelContext().getExecutorServiceStrategy().shutdownNow(executor); 7

        if (multicast != null) {
            ServiceHelper.stopServices(multicast);
        }
    }
    ...
    //----------
    // Implementation of ShutdownAware interface

    public boolean deferShutdown(ShutdownRunningTask shutdownRunningTask) {
        // deny stopping on shutdown as we want seda consumers to run in case some other queues
        // depend on this consumer to run, so it can complete its exchanges
        return true;
    }

    public int getPendingExchangesSize() {
        // number of pending messages on the queue
        return endpoint.getQueue().size();
    }

}
```

1

SedaConsumer 类通过扩展 org.apache.camel.impl.ServiceSupport 类来实施，并实施 使用者、Runnable 和 ShutdownAware 接口。

2

实施 Runnable.run（） 方法，以定义使用者在线程中运行时所执行的操作。在这种情况下，使用者在循环中运行，轮询新交换的队列，然后在队列的后一部分处理交换。

3

doStart（） 方法继承自 ServiceSupport。您覆盖这种方法，以定义消费者启动时的作用。

4

您不必直接创建线程，而是使用在 CamelContext 中注册的 ExecutorServiceStrategy 对象创建一个线程池。这很重要，因为它可让 Apache Camel 实现对线程的集中管理并支持如安全关闭。详情请查看 第 2.8 节 “线程模型”。

5

通过调用 ExecutorService.execute（） 方法 池Size 时间来启动线程。

6

doStop（） 方法从 ServiceSupport 继承。您可以覆盖这个方法，以定义消费者在关闭时做什么。

7

关闭线程池，该池由 executor 实例表示。

第 42 章 PRODUCER INTERFACE

摘要

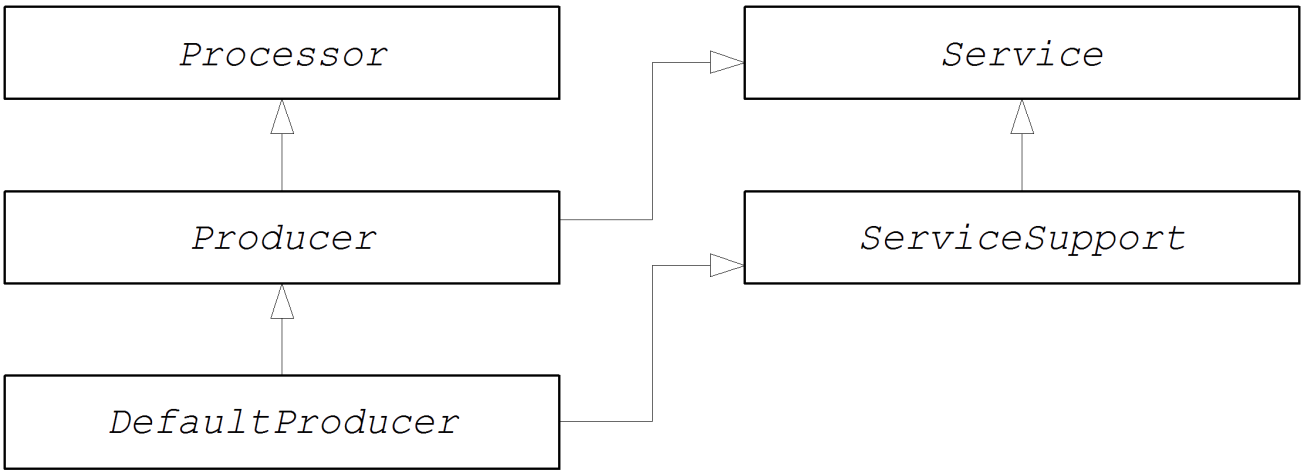
本章论述了如何实施 Producer 接口，这是实施 Apache Camel 组件的基本步骤。

42.1. PRODUCER 接口

概述

org.apache.camel.Producer 类型的实例代表路由中的一个目标端点。producer 的角色是将请求（传入消息）发送到特定的物理端点，并接收对应的响应（Out 或 Failure 消息）。 Producer 对象基本上是一个特殊的处理器，它出现在处理器链的末尾（与路由相同）。图 42.1 “producer Initance Hierarchy” 显示制作者的继承层次结构。

图 42.1. producer Initance Hierarchy



Producer 接口

例 42.1 “producer Interface” 显示 org.apache.camel.Producer 接口的定义。

例 42.1. producer Interface

```
package org.apache.camel;

public interface Producer extends Processor, Service, IsSingleton {

    Endpoint<E> getEndpoint();

    Exchange createExchange();

    Exchange createExchange(ExchangePattern pattern);
```

```
    Exchange createExchange(E exchange);
}
```

producer 方法

Producer 接口定义以下方法：

- process （）（从 Processor 中完全ited）需要注意最重要的方法。制作者基本上是一个特殊的处理器类型，它向端点发送请求，而不是将交换对象转发到另一处理器。通过覆盖 process（） 方法，您可以定义制作者如何发送和接收消息到相关的端点。

- getEndpoint（） 权利- the parent endpoint 实例引用。

- createExchange（） InfobloxThese overloaded 方法与 Endpoint 接口中定义的对应方法类似。通常，这些方法委托给父 Endpoint 实例上定义的对应方法（这是 DefaultEndpoint 类的作用）。有时，您可能需要覆盖这些方法。

异步处理

在制作的 producer producer-seconds 中处理交换对象通常涉及向远程目的地发送消息，并等待一个 reply-abrtcan 可能会大量时间块。如果您想避免阻塞当前线程，可以选择将生产者作为 异步处理器实施。异步处理模式将前面的处理器与制作者分离，以便 process（） 方法在不延迟的情况下返回。请参阅 第 38.1.4 节 “异步处理”。

在实施制作者时，您可以通过实施 org.apache.camel.AsyncProcessor 接口来支持异步处理模型。自行利用此功能是不够的，以确保将使用异步处理模型：另外，链中的前面的处理器还需要调用 process（） 方法的异步版本。AsyncProcessor 接口的定义显示在 例 42.2 “AsyncProcessor 接口” 中。

例 42.2. AsyncProcessor 接口

```
package org.apache.camel;

public interface AsyncProcessor extends Processor {
    boolean process(Exchange exchange, AsyncCallback callback);
}
```

process（） 方法的异步版本采用 org.apache.camel.AsyncCallback 类型的额外参数 回调。对应的

AsyncCallback 接口定义，如 例 42.3 “AsyncCallback Interface” 所示。

例 42.3. AsyncCallback Interface

```
package org.apache.camel;

public interface AsyncCallback {
    void done(boolean doneSynchronously);
}
```

AsyncProcessor.process（） 的调用者必须提供 AsyncCallback 的实现，以接收处理过程的通知。AsyncCallback.done（） 方法取一个布尔值参数，指明处理是同步的执行。通常，标志将是 false，用于指示异步处理。然而，在一些情况下，制作者 并不 以异步处理（尽管被要求这样做）。例如，如果制作者知道交换的处理将迅速完成，它可以通过同步方式来优化处理。在这种情况下，/ networkhron ous 标记应设置为 true。

ExchangeHelper 类

在实施制作者时，您可能会发现调用 org.apache.camel.util.ExchangeHelper 实用程序类中的某些方法会很有帮助。有关 ExchangeHelper 类的详情，请参考 第 35.4 节 “ExchangeHelper 类”。

42.2. 实施 PRODUCER 接口

实现制作者的替代方法

您可以通过以下方法之一实现制作者：

- 如何实施同步制作者
- 如何实施异步制作者

如何实施同步制作者

例 42.4 “DefaultProducer 实现” 概述了如何实施同步制作者。在这种情况下，调用 Producer.process（） 块，直到收到回复为止。

例 42.4. DefaultProducer 实现

```
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultProducer;

public class CustomProducer extends DefaultProducer { (1)

    public CustomProducer(Endpoint endpoint) { (2)
        super(endpoint);
        // Perform other initialization tasks...
    }

    public void process(Exchange exchange) throws Exception { (3)
        // Process exchange synchronously.
        // ...
    }
}
```

(1) 通过扩展 org.apache.camel.impl.Default Producer 类来实施自定义 同步制作者类。

(2) 实施对父端点的引用。

(3) process（） 方法实施代表制作者代码的核心。process（） 方法的实现完全取决于您实施的组件类型。

在概述中，process（） 方法通常实施如下：

- 如果交换包含 In 消息，如果这与指定的交换模式一致，请将 In 消息发送到指定的端点。
- 如果交换模式预测出 Out 消息的收据，请等待 Out 消息收到。这通常会导致 process（） 方法阻止了非常长的时间。
- 收到回复后，调用 exchange.setOut（） 将回复附加到交换对象。如果回复包含故障消息，请使用 Message.setFault(true) 在 Out 消息上设置 fault 标志。

如何实施异步制作者

例 42.5 “CollectionProducer 实现” 概述了如何实施异步制作者。在这种情况下，您必须同时实施同步 process（） 方法和异步 process（） 方法（需要使用额外的 AsyncCallback 参数）。

例 42.5. CollectionProducer 实现

```
import org.apache.camel.AsyncCallback;
import org.apache.camel.AsyncProcessor;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultProducer;

public class _CustomProducer_ extends DefaultProducer implements AsyncProcessor {
(1)

    public _CustomProducer_(Endpoint endpoint) { (2)
        super(endpoint);
        // ...
    }

    public void process(Exchange exchange) throws Exception { (3)
        // Process exchange synchronously.
        // ...
    }

    public boolean process(Exchange exchange, AsyncCallback callback) { (4)
        // Process exchange asynchronously.
        CustomProducerTask task = new CustomProducerTask(exchange, callback);
        // Process 'task' in a separate thread...
        // ...
        return false; (5)
    }
}

public class CustomProducerTask implements Runnable { (6)
    private Exchange exchange;
    private AsyncCallback callback;

    public CustomProducerTask(Exchange exchange, AsyncCallback callback) {
        this.exchange = exchange;
        this.callback = callback;
    }

    public void run() { (7)
        // Process exchange.
        // ...
        callback.done(false);
    }
}
```

(1)

2

实施对父端点的引用。

3

实施同步 process（） 方法。

4

实施异步 process（） 方法。您可以通过几种方法实现异步方法。此处所示的方法是创建一个 java.lang.Runnable 实例 task，它代表在子线程中运行的代码。然后，您可以使用 Java 线程 API 在子线程中运行任务（例如，通过创建新线程，或通过将任务分配给现有线程池）。

5

通常，您从异步 process（） 方法返回 false，以指示交换是异步处理的。

6

CustomProducerTask 类封装了在子线程中运行的处理代码。此类必须存储 Exchange 对象、交换 和 AsyncCallback 对象的副本，作为专用成员变量。

7

run（） 方法包含将 In 消息发送到 制作者端点的代码，并等待收到回复（若有）。收到回复（传出消息或 故障 消息）并将其插入到交换对象后，您必须调用 callback.done（） 以通知处理完成的调用者。

第 43 章 EXCHANGE INTERFACE

摘要

本章论述了 Exchange 界面。由于重构了在 Apache Camel 2.0 中执行的 camelcore 模块，因此不再需要定义自定义交换类型。现在，在所有情况下都可以使用 DefaultExchange 实现。

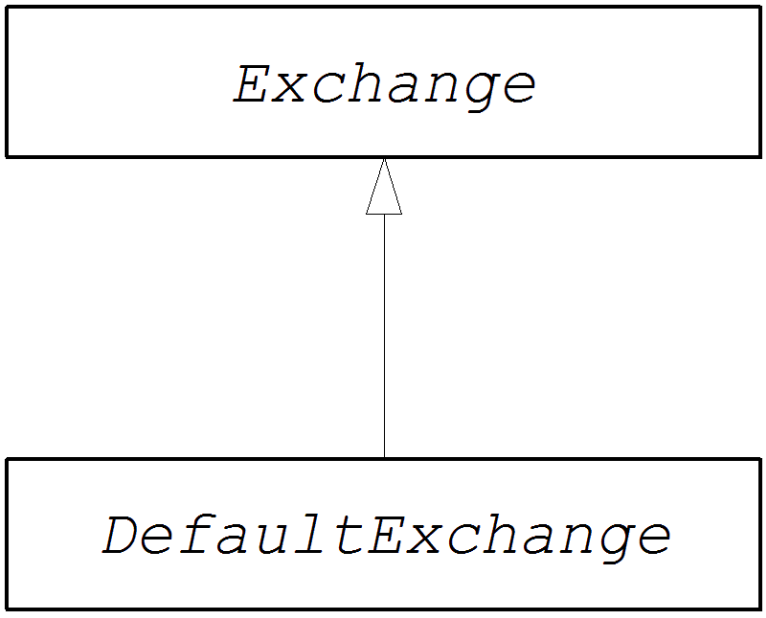
43.1. EXCHANGE INTERFACE

概述

org.apache.camel.Exchange 类型的实例封装当前通过路由传递的消息，提供额外的元数据编码为交换属性。

图 43.1 “交换继承层次结构” 显示交换类型的继承层次结构。默认实现(DefaultExchange)是始终使用。

图 43.1. 交换继承层次结构



Exchange 接口

例 43.1 “Exchange Interface” 显示 org.apache.camel.Exchange 接口的定义。

例 43.1. Exchange Interface

```
package org.apache.camel;
```

```
import java.util.Map;

import org.apache.camel.spi.Synchronization;
import org.apache.camel.spi.UnitOfWork;

public interface Exchange {
    // Exchange property names (string constants)
    // (Not shown here)
    ...

    ExchangePattern getPattern();
    void setPattern(ExchangePattern pattern);

    Object getProperty(String name);
    Object getProperty(String name, Object defaultValue);
    <T> T  getProperty(String name, Class<T> type);
    <T> T  getProperty(String name, Object defaultValue, Class<T> type);
    void   setProperty(String name, Object value);
    Object removeProperty(String name);
    Map<String, Object> getProperties();
    boolean hasProperties();

    Message getIn();
    <T> T   getIn(Class<T> type);
    void    setIn(Message in);

    Message getOut();
    <T> T   getOut(Class<T> type);
    void    setOut(Message out);
    boolean hasOut();

    Throwable getException();
    <T> T     getException(Class<T> type);
    void      setException(Throwable e);

    boolean isFailed();

    boolean isTransacted();

    boolean isRollbackOnly();

    CamelContext getContext();

    Exchange copy();

    Endpoint getFromEndpoint();
    void     setFromEndpoint(Endpoint fromEndpoint);

    String getFromRouteId();
    void   setFromRouteId(String fromRouteId);

    UnitOfWork getUnitOfWork();
    void setUnitOfWork(UnitOfWork unitOfWork);

    String getExchangeId();
    void setExchangeId(String id);
```

```
    void addOnCompletion(Synchronization onCompletion);
    void handoverCompletions(Exchange target);
}
```

Exchange 方法

Exchange 接口定义了以下方法：

- *getPattern（）, setPattern（） the exchange pattern 可以是 org.apache.camel.ExchangePatmel.ExchangePattern.ExchangePattern.ExchangePattern. ExchangePattern.ExchangePattern. .支持以下交换模式值：*
  - *InOnly*
  - *RobustInOnly*
  - *InOut*
  - *InoptionalOut*
  - *OutOnly*
  - *RobustOutOnly*
  - *OutIn*
  - *OutOptionalIn*
- *setProperty（）, getProperty（）, getProperties （）, removeProperty （）, have Properties（）有Properties（） 使用属性 setter 和 getter 方法将命名的属性与交换实例相关联。属性包含您的组件实施可能需要的其它元数据。*

- *setIn（）, getIn（） Infoblox->_<Setter 和 getter 方法用于 In 消息。*

  DefaultExchange 类提供的 getIn（） 实施实施 lazy creation语义：如果 In message 为 null，则 DefaultExchange 类会创建一个默认的 In 消息。

- *setOut（）, getOut（） , , 具有Out（） abrt- theSetter 和 getter 方法（用于 Out 消息）。*

  getOut（） 方法隐式支持创建 Out 消息。也就是说，如果当前 Out 消息 为空，则会自动创建新的消息实例。

- *setException（）, getException（） 10.10.10.2- theGetter 和 setter 方法用于异常对象（元 类型）。*

- *对于因为异常或故障而失败的交换失败，则 isFailed（） 3.10.0--abrtRe returns true。*

- *如果交换被转换，则 isTransacted（） to the 时间为 to- the the Transli's true。*

- *如果交换被标记为回滚，则 isRollback（） toRollback（）to and-abrtRe returns true。*

- *Getcontext（） InventoryService-evictionRe returns 对关联的 CamelContext 实例的引用。*

- *copy（） Equal-jaxbCreates a new identical（来自交换 ID 的部分）副本当前自定义交换对象。In 消息的正文 和标头、Out 消息（若有）和故障消息（若有）也被这个操作复制。*

- *setFromEndpoint（）, getFromEndpoint（） 时间为发布此消息的消费者端点的 setter 和 setter 方法（这通常是端点在路由启动时出现在 from（） DSL 命令中）。*

- *setFromRouteId（）, getFromRouteId（） 时间（）和 setters，查找源自此交换的路由 ID。getFromRouteId（） 方法应该只在内部调用。*

- *setUnitOfWork（）, getUnitOfWork（） int-3Getter 和 setter method for the org.apache.camel.spi.UnitOfWork an 属性。这个属性只适用于参与事务的交换。*

- *setExchangeId（）, getExchangeId（） abrt-abrtGetter 和 setter 方法用于交换 ID。自定义组件是否使用交换 ID 是实现详情。*

- *addOnCompletion（） Infoblox->_<Adds a org.apache.camel.spi.Synchronization 回调对象（在处理交换完成后会调用）。*

- *在所有 OnCompletion callback 对象到指定的交换对象上， handoverCompletions（） abrt-abrt- the callbackHands on the all of the all of the all of the OnCompletion callback 对象。*

第 44 章 消息接口

摘要

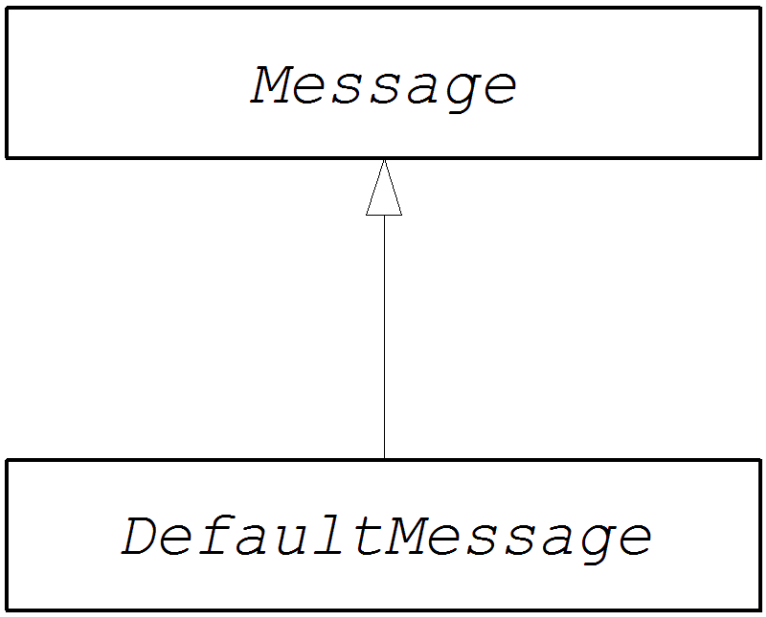
本章论述了如何实施 Message 接口，这是实施 Apache Camel 组件的可选步骤。

44.1. 消息接口

概述

org.apache.camel.Message 类型的实例可以表示任何类型的消息（In 或 Out）。图 44.1 “消息继承层次结构” 显示消息类型的继承层次结构。您不需要为组件实施自定义消息类型。在很多情况下，默认的实现(DefaultMessage)是适当的。

图 44.1. 消息继承层次结构



Message 接口

例 44.1 “消息接口” 显示 org.apache.camel.Message 接口的定义。

例 44.1. 消息接口

```
package org.apache.camel;

import java.util.Map;
import java.util.Set;

import javax.activation.DataHandler;
```

```
public interface Message {

    String getMessageId();
    void setMessageId(String messageId);

    Exchange getExchange();

    boolean isFault();
    void    setFault(boolean fault);

    Object getHeader(String name);
    Object getHeader(String name, Object defaultValue);
    <T> T getHeader(String name, Class<T> type);
    <T> T getHeader(String name, Object defaultValue, Class<T> type);
    Map<String, Object> getHeaders();
    void setHeader(String name, Object value);
    void setHeaders(Map<String, Object> headers);
    Object  removeHeader(String name);
    boolean removeHeaders(String pattern);
    boolean hasHeaders();

    Object getBody();
    Object getMandatoryBody() throws InvalidPayloadException;
    <T> T  getBody(Class<T> type);
    <T> T  getMandatoryBody(Class<T> type) throws InvalidPayloadException;
    void     setBody(Object body);
    <T> void setBody(Object body, Class<T> type);

    DataHandler getAttachment(String id);
    Map<String, DataHandler> getAttachments();
    Set<String> getAttachmentNames();
    void removeAttachment(String id);
    void addAttachment(String id, DataHandler content);
    void setAttachments(Map<String, DataHandler> attachments);
    boolean hasAttachments();

    Message copy();

    void copyFrom(Message message);

    String createExchangeId();
}
```

消息方法

Message 接口定义了以下方法：

- ***setMessageId（）, getMessageId（） InventoryService-","Getter 和 setter 方法用于消息 ID。是否需要在自定义组件中使用消息 ID 是一个实现详情。***

- getExchange（） admission- the parent exchange 对象的引用。
- isFault（）, setFault（） abrt-jaxbGetter 和 setter method for the fault 标记（指明此消息是否是一个错误消息）。
- getHeader（）, getHeaders（）, setHeader（） , setHeaders （）, removeHeader （）, 带有Headers（） protobuf-对消息标头的 setter 方法。通常，这些消息标头可用于存储实际标头数据，或者用于存储各种元数据。
- getBody（）, getMandatoryBody（）, setBody（） and setBody（）and setter method for the message body.getMandatoryBody（）accessor 保证返回的正文不是null, 否则会引发 InvalidPayloadException 异常。
- getAttachment（）, getAttachment（）, getAttachmentNames（）, removeAttachment（） , add Attachment（）, setAttachments（） , haveAttachments（）, withAttachments（）获取、设置、添加和删除附加。
- copy（） InventoryService-apiVersionCreates a new, identical（包括消息 ID）副本当前自定义消息对象。
- 将 指定通用消息对象（包括消息 ID）的完整内容（包括消息 ID）复制到当前消息实例中。由于此方法必须能够从任何消息类型复制，因此它会复制通用消息属性，但不能复制自定义属性。
- 如果消息实施能够提供 ID，则 createExchangeId（） ， 会返回 null 的唯一 ID。

44.2. 实施消息接口

如何实施自定义消息

例 44.2 “自定义消息实施” 概述了如何通过扩展 DefaultMessage 类来实施消息。

例 44.2. 自定义消息实施

```
import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultMessage;

public class CustomMessage extends DefaultMessage { (1)
```

```
    public CustomMessage() { (2)
        // Create message with default properties...
    }

    @Override
    public String toString() { (3)
        // Return a stringified message...
    }

    @Override
    public CustomMessage newInstance() { (4)
        return new CustomMessage( ... );
    }

    @Override
    protected Object createBody() { (5)
        // Return message body (lazy creation).
    }

    @Override
    protected void populateInitialHeaders(Map&lt;String, Object&gt; map) { (6)
        // Initialize headers from underlying message (lazy creation).
    }

    @Override
    protected void populateInitialAttachments(Map&lt;String, DataHandler&gt; map) { (7)
        // Initialize attachments from underlying message (lazy creation).
    }
}
```

(1) 通过扩展 org.apache.camel.impl.DefaultMessage 类来实施自定义消息类 CustomMessage。

(2) 通常，您需要一种默认的构造器来创建具有默认属性的消息。

(3) 覆盖 toString（） 方法以自定义消息字符串ification。

(4) newInstance（） 方法从 MessageSupport.copy（） 方法中调用。自定义 newInstance（） 方法应该侧重于将当前消息实例的所有自定义属性复制到新消息实例中。MessageSupport.copy（） 方法通过调用 copyFrom（） 来复制通用消息属性。

5

createBody（） 方法与 MessageSupport.getBody（） 方法配合使用，以实施对消息正文的 lazy 访问权限。默认情况下，消息正文为 null。只有在应用程序代码试图访问正文（调用 getBody（））时，才应创建正文。当第一次访问消息正文时，MessageSupport.getBody（） 会自动调用 createBody（）。

6

populateInitialHeaders（） 方法与标头 getter 和 setter 方法配合使用，以实施对消息标头的 lazy access。此方法解析消息以提取任何消息标头，并将它们插入到散列映射中。当用户首次尝试访问标头（或标头）时（调用 getHeader（）、getHeaders（）、setHeaders（）或 set Headers（） ）时，会自动调用 populate InitialHeaders（）。

7

populateInitialAttachments（） 方法与附件 getter 和 setter 方法配合使用，来实现对附件的访问。此方法提取消息附件，并将它们插入到散列映射中。当用户首次通过调用 getAttachment（）、getAttachment（） 或 addAttachment（）时，会自动调用 populateInitial Attachment s（） 或 addAttachment（）。

部分 IV. API 组件框架

如何使用 API 组件框架创建嵌套任何 Java API 的 Camel 组件。

第 45 章 API 组件框架简介

摘要

API 组件框架可帮助您根据大型 Java API 实施复杂 Camel 组件。

45.1. 什么是 API 组件框架？

动机

对于包含少量选项的组件，实施组件的标准方法(第 38 章 实施组件)非常有效。然而，它开始会遇到问题，当您需要实施大量选项的组件时。当涉及企业级组件时，这个问题会非常显著，这需要您打包由 数百个 操作组成的 API。此类组件需要大量精力来创建和维护。

API 组件框架被精确开发，以应对实施此类组件所带来的挑战。

将 API 转换为组件

基于 Java API 实施 Camel 组件的经验表明，许多工作是日常工作。它包括使用特定的 Java 方法，将其映射到特定的 URI 语法，并允许用户通过 URI 选项设置方法参数。这种类型的工作是自动化和代码生成不可或缺的候选者。

通用 URI 格式

自动化实施 Java API 的第一步是设计一种将 API 方法映射到 URI 的标准方法。为此，我们需要定义通用 URI 格式，可用于嵌套 任何 Java API。因此，API 组件框架定义了端点 URI 的以下语法：

```
scheme://endpoint-prefix/endpoint?Option1=Value1&...&OptionN=ValueN
```

其中 scheme 是组件定义的默认 URI 方案； endpoint-prefix 是一个简短的 API 名称，它映射到嵌套的 Java API 中的类或接口之一； 端点 映射到方法名称； URI 选项映射到方法参数名称。

单个 API 类的 URI 格式

如果 API 只包含一个 Java 类，则 URI 的端点前缀 部分会具有冗余，并且您可以使用以下更短的格式指定 URI：

```
scheme://endpoint?Option1=Value1&...&OptionN=ValueN
```

■

注意

要启用此 URI 格式，组件实施器还需要将 apiName 元素留在 API 组件 Maven 插件的配置中。如需更多信息，请参阅 “配置 API 映射”一节 部分。

反映和元数据

要将 Java 方法调用映射到 URI 语法，显然需要某种形式的反射机制。但是，标准的 Java 反映 API 的性能有显著限制：它不保留方法参数名称。这是一个问题，因为我们需要方法参数名称才能生成有意义的 URI 选项名称。解决办法是提供替代格式的元数据：作为 Javadoc 或方法签名文件。

javadoc

javadoc 是 API 组件框架的一个理想的元数据形式，因为它会保留完整的方法签名，包括方法参数名称。它还容易生成（使用 maven-javadoc-plugin）和很多情况下，已在第三方库中提供。

方法签名文件

如果出于某种原因，如果 Javadoc 不可用或不明确，API 组件框架还支持备选元数据来源：方法签名文件。签名文件是一个简单文本文件，它由 Java 方法签名列表组成。从 Java 代码复制和粘贴（并最好编辑得到的文件）相对容易地创建这些文件。

框架包含什么？

从组件开发人员的角度来看，API 组件框架由多个不同的元素组成，如下所示：

Maven archetype

camel-archetype-api-component Maven archetype 用于生成组件实施的框架代码。

Maven 插件

camel-api-component-maven-plugin Maven 插件负责生成在 Java API 和端点 URI 语法之间实现映射的代码。

专用基础类

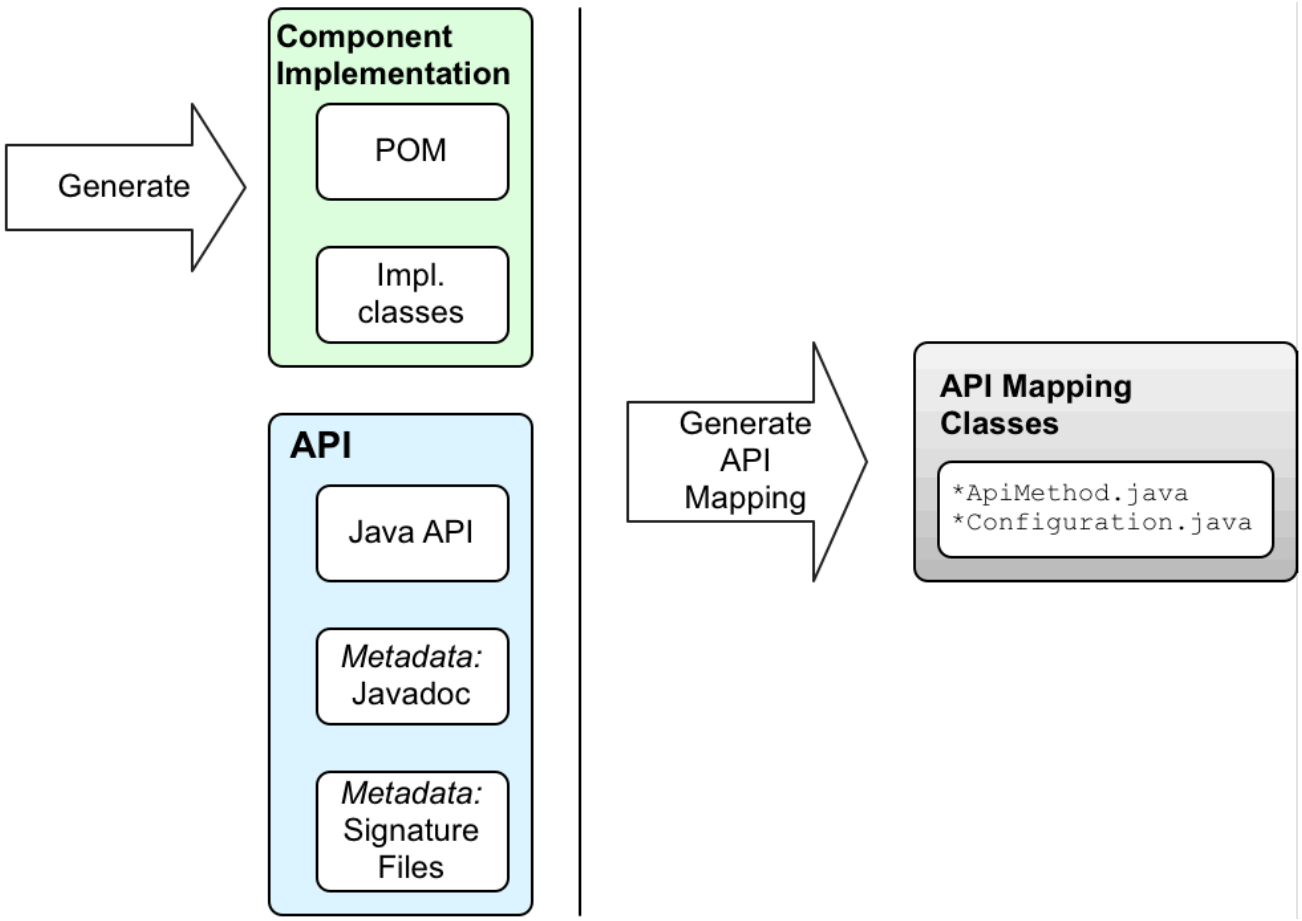
为了支持 API 组件框架的编程模型，Apache Camel 内核在 org.apache.camel.util.component 软件包中提供了一个专用 API。另外，此 API 为组件、端点、消费者和制作者类提供专业的基础类。

45.2. 如何使用框架

概述

使用 API 框架实施组件的过程涉及混合了自动化代码生成、实施 Java 代码以及通过编辑 Maven POM 文件来自定义构建。下图提供了此开发过程的概述。

图 45.1. 使用 API 组件框架



Java API

API 组件的起点始终是一个 Java API。通常而言，在 Camel 的上下文中，这通常意味着 Java 客户端 API，它连接到远程服务器端点。第一个问题是，Java API 来自哪里？以下是一些可能性：

- 自行实施 Java API（尽管这通常涉及许多工作，通常不是首选方法）。
- 使用第三方 Java API。例如，Apache Camel Box 组件基于第三方 Box Java SDK 库。
- 从语言中立接口生成 Java API。

javadoc 元数据

您可以选择以 Javadoc（在 API 组件框架中生成代码）的形式为 Java API 提供元数据。如果您从 Maven 存储库使用第三方 Java API，则通常会发现 Maven 工件中已提供 Javadoc。但即使在 未提供 Javadoc 的情况下，您仍可使用 maven-javadoc-plugin Maven 插件轻松生成它。

注意

目前，处理 Javadoc 元数据存在限制，因此不支持通用嵌套。例如，支持 java.util.List<String>，但 java.util.List<java.util.List<String>& gt; 不是。解决办法是在签名文件中将嵌套的通用类型指定为 java.util.List<java.util.List >。

签名文件元数据

如果出于某种原因无法方便地以 Javadoc 的形式提供 Java API 元数据，您可以选择以 签名文件形式提供元数据。签名文件包含方法签名列表（每行一个方法签名）。这些文件可以手动创建，且仅在构建时才需要。

请注意以下关于签名文件的内容：

- *您必须为每个代理类（Java API 类）创建一个签名文件。*
- *方法签名 不应 抛出异常。运行时出现的所有异常都嵌套在 RuntimeCamelException 中并从端点返回。*
- *指定参数类型的类名称必须是完全限定的类名称（ java.lang.* 类型除外）。没有用于导入软件包名称的机制。*
- *目前，签名解析器存在限制，因此不支持通用嵌套。例如，支持 java.util.List<String>，而 java.util.List<java.util.List<String>& gt; 不是。解决办法是将嵌套的通用类型指定为 java.util.List<java.util.List>。*

下面显示了一个签名文件内容的简单示例：

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

使用 Maven archetype 生成启动代码

开始开发 API 组件的最简单方法是使用 camel-archetype-api-component Maven archetype 生成初始 Maven 项目。有关如何运行 archetype 的详情，请参考 第 46.1 节 “使用 Maven Archetype 生成代码”。

运行 Maven archetype 后，您将在生成的 ProjectName 目录下找到两个子项目：

ProjectName-api

此项目包含 Java API，它形成了 API 组件的基础。构建此项目时，它会将 Java API 打包到 Maven 捆绑包中，并生成必要的 Javadoc。如果 Java API 和 Javadoc 已经由第三方提供，但您不需要这个子项目。

ProjectName-component

此项目包含 API 组件的框架代码。

编辑组件类

您可以编辑 ProjectName-component 中的框架代码，以开发您自己的组件实施。以下生成的类组成了框架实施的核心：

```
ComponentNameComponent
ComponentNameEndpoint
ComponentNameConsumer
ComponentNameProducer
ComponentNameConfiguration
```

自定义 POM 文件

您还需要编辑 Maven POM 文件来自定义构建，并配置 camel-api-component-maven-plugin Maven 插件。

配置 camel-api-component-maven-plugin

配置 POM 文件的最重要方面是 camel-api-component-maven-plugin Maven 插件的配置。此插件负责生成 API 方法和端点 URI 之间的映射，并通过编辑插件配置来自定义映射。

例如，在 ProjectName-component/pom.xml 文件中，以下 camel-api-component-maven-plugin 插件配置显示了名为 ExampleJavadocHello 的 API 类的最小配置。

```
<configuration>
  <apis>
    <api>
      <apiName>hello-javadoc</apiName>
      <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
      <fromJavadoc/>
    </api>
  </apis>
</configuration>
```

在本例中，hello-javadoc API 名称映射到 ExampleJavadocHello 类，这意味着您可以使用表单的 URI 从这个类调用方法，方案://hello-javadoc/端点。存在 fromJavadoc 元素表示 ExampleJavadocHello 类从 Javadoc 获取其元数据。

OSGi 捆绑包配置

组件 sub-project、ProjectName-component/pom.xml 的 POM 示例配置为将组件打包为 OSGi 捆绑包。组件 POM 包括 maven-bundle-plugin 的示例配置。您应该自定义 maven-bundle-plugin 插件的配置，以确保 Maven 为您的组件生成正确配置的 OSGi 捆绑包。

构建组件

使用 Maven 构建组件时（例如，使用 mvn clean 软件包），camel-api-component-maven-plugin 插件会自动生成 API 映射类（定义 Java API 和端点 URI 语法之间的映射），将它们放入 target/classes 项目子目录。当您处理大型和复杂的 Java API 时，这个生成的代码实际上构成了大量组件源代码。

Maven 构建完成后，编译的代码和资源打包为 OSGi 捆绑包，并将其作为 Maven 工件存储在本地 Maven 存储库中。

第 46 章 FRAMEWORK 入门

摘要

本章论述了基于使用 camel-archetype-api-component Maven archetype 生成的代码，使用 API 组件框架实施 Camel 组件的基本原则。

46.1. 使用 MAVEN ARCHETYPE 生成代码

Maven archetypes

Maven archetype 与代码向导类似：给定几个简单参数，它会生成一个完整的、可正常工作的 Maven 项目，填充示例代码。然后，您可以将此项目用作模板，自定义实施以创建自己的应用程序。

API 组件 Maven archetype

API 组件框架提供了一个 Maven archetype(camel-archetype-api-component)，可为您自己的 API 组件实施生成起始点代码。这是开始创建自己的 API 组件的建议方法。

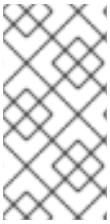
先决条件

运行 camel-archetype-api-component archetype 的唯一先决条件是安装 Apache Maven，并且将 Maven settings.xml 文件配置为使用标准 Fuse 存储库。

调用 Maven archetype

若要创建 示例 组件，它使用 示例 URI 方案，调用 camel-archetype-api-component archetype 以生成新的 Maven 项目，如下所示：

```
mvn archetype:generate \
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-api-component \
-DarchetypeVersion=2.23.2.fuse-7_10_0-00018-redhat-00001 \
-DgroupId=org.jboss.fuse.example \
-DartifactId=camel-api-example \
-Dname=Example \
-Dscheme=example \
-Dversion=1.0-SNAPSHOT \
-DinteractiveMode=false
```



注意

每行末尾的反斜杠字符 \ 代表行继续，这仅适用于 Linux 和 UNIX 平台。在 Windows 平台上，删除反斜杠并将所有参数放在一行中。

选项

使用语法 -DName=Value 提供给 archetype generation 命令的选项。大多数选项应按照前面的 mvn archetype:generate 命令设置，但可以修改一些选项，以自定义生成的项目。下表显示了可用来自定义生成的 API 组件项目的选项：

名称	描述
groupId	（generic Maven 选项） 指定生成的 Maven 项目的组 ID。默认情况下，这个值也定义所生成的类的 Java 软件包名称。因此，选择这个值要与您想要的 Java 软件包名称匹配是一个好主意。
artifactId	（generic Maven 选项） 指定生成的 Maven 项目的构件 ID。
name	API 组件的名称。这个值用于在生成的代码中生成类名称（提示，建议名称应该以大写字母开头）。
scheme	此组件的 URI 中使用的默认方案。您应该确保此方案与现有 Camel 组件的方案不会冲突。
archetypeVersion	（通用 Maven 选项） 是计划部署组件的容器应使用的 Apache Camel 版本。但是，在生成项目后，您也可以修改 Maven 依赖项的版本。

生成的项目的结构

假设代码生成步骤成功完成，您应该会看到一个新的目录 camel-api-example，其中包含新的 Maven 项目。如果您在 camel-api-example 目录内部，您会看到它有以下通用结构：

```
camel-api-example/
    pom.xml
    camel-api-example-api/
    camel-api-example-component/
```

在项目顶端是一个聚合 POM, pom.xml，它被配置为构建两个子项目，如下所示：

camel-api-example-api

API 子项目（名为 ArtifactId-api）包含您要转变为组件的 Java API。如果您自己编写了 Java API 上的 API 组件，可以将 Java API 代码直接放入此项目。

API 子项目可用于以下一个或多个目的：

- 打包 Java API 代码（如果它尚未作为 Maven 软件包可用）。
- 为 Java API 生成 Javadoc（提供 API 组件框架所需的元数据）。
- 要从 API 描述生成 Java API 代码（例如，来自 REST API 的 WADL 描述）。

然而，在有些情况下，您可能需要执行任何这些任务。例如，如果 API 组件基于第三方 API，它已在 Maven 软件包中提供 Java API 和 Javadoc。在这种情况下，您可以删除 API 子项目。

camel-api-example-component

组件子项目（名为 ArtifactId-component）包含新 API 组件的实施。这包括组件实施类和配置 camel-api-component-maven 插件（用于从 Java API 生成 API 映射类）。

46.2. 生成的 API 子项目

概述

假设您在 第 46.1 节 “使用 Maven Archetype 生成代码” 中生成了新的 Maven 项目，您现在可以找到用于在 camel-api-example/camel-api-example-api 项目目录下打包 Java API 的 Maven 子项目。在本节中，我们会仔细查看生成的示例代码，并描述其工作原理。

Java API 示例

生成的示例代码包含了一个 Java API 示例，它基于该示例 API 组件。示例 Java API 相对简单，仅包含两个 Hello World 类： ExampleJavadocHello 和 ExampleFileHello。

ExampleJavadocHello 类

例 46.1 “ExampleJavadocHello 类” 显示示例 Java API 中的 ExampleJavadocHello 类。根据课程

的名称，此特定类用于显示如何从 Javadoc 提供映射元数据。

例 46.1. ExampleJavadocHello 类

```
// Java
package org.jboss.fuse.example.api;

/**
 * Sample API used by Example Component whose method signatures are read from Javadoc.
 */
public class ExampleJavadocHello {

    public String sayHi() {
        return "Hello!";
    }

    public String greetMe(String name) {
        return "Hello " + name;
    }

    public String greetUs(String name1, String name2) {
            return "Hello " + name1 + ", " + name2;
    }
}
```

ExampleFileHello class

例 46.2 “ExampleFileHello class” 显示示例 Java API 中的 ExampleFileHello 类。根据分类的名称，此特定类用于显示如何从签名文件中提供映射元数据。

例 46.2. ExampleFileHello class

```
// Java
package org.jboss.fuse.example.api;

/**
 * Sample API used by Example Component whose method signatures are read from File.
 */
public class ExampleFileHello {

    public String sayHi() {
        return "Hello!";
    }

    public String greetMe(String name) {
        return "Hello " + name;
    }

    public String greetUs(String name1, String name2) {
```

```
        return "Hello " + name1 + ", " + name2;
    }
}
```

为 **ExampleJavadocHello 生成 Javadoc 元数据**

由于 ExampleJavadocHello 的元数据已作为 Javadoc 提供，因此需要为示例 Java API 生成 Javadoc，并将它安装到 camel-api-example-api Maven 工件中。API POM 文件(camel-api-example-api/pom.xml)配置 maven-javadoc-plugin，以在 Maven 构建过程中自动执行此步骤。

46.3. 生成的组件子项目

概述

用于构建新组件的 Maven 子项目位于 camel-api-example/camel-api-example-component 项目目录下。在本节中，我们会仔细查看生成的示例代码，并描述其工作原理。

在组件 POM 中提供 Java API

Java API 必须作为组件 POM 中的依赖项提供。例如，示例 Java API 在组件 POM 文件中定义为依赖项，cal -api-example-component/pom.xml 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.fuse.example</groupId>
      <artifactId>camel-api-example-api</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

在组件 POM 中提供 Javadoc 元数据

如果您使用 Java API 的所有或部分的 Javadoc 元数据，则必须在组件 POM 中以依赖项形式提供

Javadoc。关于这个依赖关系，有两个问题：

- *Javadoc 的 Maven 协调几乎和 Java API 相同，但您还必须指定类 符 元素，如下所示：*

```
<classifier>javadoc</classifier>
```

- *您必须声明 Javadoc 以 提供范围，如下所示：*

```
<scope>provided</scope>
```

例如，在组件 POM 中，Javadoc 依赖项定义如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">

  ...
  <dependencies>
    ...
    <!-- Component API javadoc in provided scope to read API signatures -->
    <dependency>
      <groupId>org.jboss.fuse.example</groupId>
      <artifactId>camel-api-example-api</artifactId>
      <version>1.0-SNAPSHOT</version>
      <classifier>javadoc</classifier>
      <scope>provided</scope>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

为示例文件 Hello 定义文件元数据

ExampleFileHello 的元数据在签名文件中提供。通常，必须手动创建此文件，但是它的格式非常简单，它由一个方法签名列表（每行一个）组成。示例代码在 目录中提供了签名文件 file-sig -api.txt，它包含以下内容：

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

有关签名文件格式的详情，请参考 “签名文件元数据”一节。

配置 API 映射

API 组件框架的主要功能之一是它自动生成代码来执行 API 映射。也就是说，生成存根代码，将端点 URI 映射到 Java API 上调用的方法。API 映射的基本输入有：Java API、Javadoc 元数据和/或签名文件元数据。

执行 API 映射的组件是 camel-api-component-maven-plugin Maven 插件，它在组件 POM 中配置。以下从组件 POM 中提取的显示了如何配置 camel-api-component-maven-plugin 插件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">

  ...
  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>
      ...
      <!-- generate Component source and test source -->
      <plugin>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-api-component-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>generate-test-component-classes</id>
            <goals>
              <goal>fromApis</goal>
            </goals>
            <configuration>
              <apis>
                <api>
                  <apiName>hello-file</apiName>
                  <proxyClass>org.jboss.fuse.example.api.ExampleFileHello</proxyClass>
                  <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile>
                </api>
                <api>
                  <apiName>hello-javadoc</apiName>
                  <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
                  <fromJavadoc/>
                </api>
              </apis>
            </configuration>
          </execution>
        </executions>
      </plugin>
```

```
    ...
  </plugins>
  ...
 </build>
 ...
</project>
```

该插件由 configuration 元素配置，其中包含一个 apis 子元素来配置 Java API 类。每个 API 类都由一个 api 元素配置，如下所示：

apiName

API 名称是 API 类的简短名称，用作 端点 URI 的端点前缀 部分。

注意

如果 API 只包含一个 Java 类，您可以将 apiName 元素留空，以便 endpoint-prefix 变为冗余，您可以使用 “单个 API 类的 URI 格式”一节 所示的格式指定端点 URI。

proxyClass

proxy class 元素指定 API 类的完全限定名称。

fromJavadoc

如果 API 类由 Javadoc 元数据提供，您必须通过包含 fromJavadoc 元素和 Javadoc 本身在 Maven 文件中指定它，作为 提供的 依赖项（请参阅 “在组件 POM 中提供 Javadoc 元数据”一节）。

fromSignatureFile

如果 API 类使用签名文件元数据，您必须通过包含 fromSignatureFile 元素来指明这一点，此元素的内容指定了签名文件的位置。

注意

签名文件不包括在 Maven 构建的最终软件包中，因为这些文件仅在构建时不需要。

生成的组件实施

API 组件由以下核心类（必须为每个 Camel 组件实施）位于 camel-api-example-component/src/main/java 目录下：

Component 示例

代表组件本身。此类充当端点实例的工厂（如 ExampleEndpoint的实例）。

示例Endpoint

代表端点 URI。此类充当消费者端点的工厂（例如，ExampleConsumer）和制作者端点的工厂（如 示例Producer）。

Consumer 示例

代表消费者端点的声明实例，该端点能够使用来自端点 URI 中指定的位置的消息。

Producer 示例

代表制作者端点的 concrete 实例，它能够将消息发送到端点 URI 中指定的位置。

ExampleConfiguration

可用于定义端点 URI 选项。此配置类定义的 URI 选项 不 绑定到任何特定的 API 类。也就是说，您可以将这些 URI 选项与任何 API 类或方法组合。例如，如果您需要声明用户名和密码凭证才能连接到远程服务，这将会很有用。ExampleConfiguration 类的主要目的是提供实例化 API 类或实施 API 接口所需的参数的值。例如，这些可以是构造参数，也可以是工厂方法或类的参数值。

要实现 URI 选项（在这一类中），您需要做的所有选项都是实施访问器方法对，获取选项和设置选项。 组件框架自动解析端点 URI，并在运行时注入选项值。

Component 类示例

生成的 ExampleComponent 类定义如下：

```
// Java
package org.jboss.fuse.example;

import org.apache.camel.CamelContext;
import org.apache.camel.Endpoint;
import org.apache.camel.spi.UriEndpoint;
import org.apache.camel.util.component.AbstractApiComponent;

import org.jboss.fuse.example.internal.ExampleApiCollection;
import org.jboss.fuse.example.internal.ExampleApiName;

/**
```

```
 * Represents the component that manages {@link ExampleEndpoint}.
 */
@UriEndpoint(scheme = "example", consumerClass = ExampleConsumer.class, consumerPrefix =
"consumer")
public class ExampleComponent extends AbstractApiComponent<ExampleApiName,
ExampleConfiguration, ExampleApiCollection> {

    public ExampleComponent() {
        super(ExampleEndpoint.class, ExampleApiName.class, ExampleApiCollection.getCollection());
    }

    public ExampleComponent(CamelContext context) {
        super(context, ExampleEndpoint.class, ExampleApiName.class,
ExampleApiCollection.getCollection());
    }

    @Override
    protected ExampleApiName getApiName(String apiNameStr) throws IllegalArgumentException {
        return ExampleApiName.fromValue(apiNameStr);
    }

    @Override
    protected Endpoint createEndpoint(String uri, String methodName, ExampleApiName apiName,
                                      ExampleConfiguration endpointConfiguration) {
        return new ExampleEndpoint(uri, this, apiName, methodName, endpointConfiguration);
    }
}
```

此类中的重要方法是 createEndpoint，它用于创建新的端点实例。通常，您不需要更改组件类中的任何默认代码。但是，如果存在与这个组件相同的生命周期其他任何对象，您可能想让这些对象从组件类访问（例如，通过添加方法来创建这些对象或通过将这些对象注入组件）。

ExampleEndpoint 类

生成的 ExampleEndpoint 类定义如下：

```
// Java
package org.jboss.fuse.example;

import java.util.Map;

import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.spi.UriEndpoint;
import org.apache.camel.util.component.AbstractApiEndpoint;
import org.apache.camel.util.component.ApiMethod;
import org.apache.camel.util.component.ApiMethodPropertiesHelper;

import org.jboss.fuse.example.api.ExampleFileHello;
import org.jboss.fuse.example.api.ExampleJavadocHello;
```

```
import org.jboss.fuse.example.internal.ExampleApiCollection;
import org.jboss.fuse.example.internal.ExampleApiName;
import org.jboss.fuse.example.internal.ExampleConstants;
import org.jboss.fuse.example.internal.ExamplePropertiesHelper;

/**
 * Represents a Example endpoint.
 */
@UriEndpoint(scheme = "example", consumerClass = ExampleConsumer.class, consumerPrefix =
"consumer")
public class ExampleEndpoint extends AbstractApiEndpoint<ExampleApiName,
ExampleConfiguration> {

    // TODO create and manage API proxy
    private Object apiProxy;

    public ExampleEndpoint(String uri, ExampleComponent component,
                           ExampleApiName apiName, String methodName, ExampleConfiguration
endpointConfiguration) {
        super(uri, component, apiName, methodName,
ExampleApiCollection.getCollection().getHelper(apiName), endpointConfiguration);

    }

    public Producer createProducer() throws Exception {
        return new ExampleProducer(this);
    }

    public Consumer createConsumer(Processor processor) throws Exception {
        // make sure inBody is not set for consumers
        if (inBody != null) {
            throw new IllegalArgumentException("Option inBody is not supported for consumer
endpoint");
        }
        final ExampleConsumer consumer = new ExampleConsumer(this, processor);
        // also set consumer.* properties
        configureConsumer(consumer);
        return consumer;
    }

    @Override
    protected ApiMethodPropertiesHelper<ExampleConfiguration> getPropertiesHelper() {
        return ExamplePropertiesHelper.getHelper();
    }

    protected String getThreadProfileName() {
        return ExampleConstants.THREAD_PROFILE_NAME;
    }

    @Override
    protected void afterConfigureProperties() {
        // TODO create API proxy, set connection properties, etc.
        switch (apiName) {
            case HELLO_FILE:
                apiProxy = new ExampleFileHello();
                break;
```

```
            case HELLO_JAVADOC:
                apiProxy = new ExampleJavadocHello();
                break;
            default:
                throw new IllegalArgumentException("Invalid API name " + apiName);
        }
    }

    @Override
    public Object getApiProxy(ApiMethod method, Map<String, Object> args) {
        return apiProxy;
    }
}
```

在 API 组件框架的上下文中，端点类执行的一个关键步骤之一是创建 API 代理。API 代理是一个来自目标 Java API 的实例，其方法由端点调用。因为 Java API 通常由多个类组成，因此需要根据 URI 中出现的端点 前缀 来选择适当的 API 类（所有 URI 具有常规形式，方案://端点前缀/端点）。

Consumer 类示例

生成的 ExampleConsumer 类定义如下：

```
// Java
package org.jboss.fuse.example;

import org.apache.camel.Processor;
import org.apache.camel.util.component.AbstractApiConsumer;

import org.jboss.fuse.example.internal.ExampleApiName;

/**
 * The Example consumer.
 */
public class ExampleConsumer extends AbstractApiConsumer<ExampleApiName,
ExampleConfiguration> {

    public ExampleConsumer(ExampleEndpoint endpoint, Processor processor) {
        super(endpoint, processor);
    }

}
```

Producer 类示例

生成的 ExampleProducer 类定义如下：

```
// Java
package org.jboss.fuse.example;
```

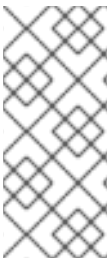
```
import org.apache.camel.util.component.AbstractApiProducer;

import org.jboss.fuse.example.internal.ExampleApiName;
import org.jboss.fuse.example.internal.ExamplePropertiesHelper;

/**
 * The Example producer.
 */
public class ExampleProducer extends AbstractApiProducer<ExampleApiName,
ExampleConfiguration> {

    public ExampleProducer(ExampleEndpoint endpoint) {
        super(endpoint, ExamplePropertiesHelper.getHelper());
    }
}
```

ExampleConfiguration 类

生成的 ExampleConfiguration 类定义如下：

```
// Java
package org.jboss.fuse.example;

import org.apache.camel.spi.UriParams;

/**
 * Component configuration for Example component.
 */
@UriParams
public class ExampleConfiguration {

    // TODO add component configuration properties
}
```

要添加 URI 选项（选择）到此类中，定义相应类型的字段，并实施对应的访问器方法、获取选项和设置选项。组件框架自动解析端点 URI，并在运行时注入选项值。



注意

这个类用于定义常规 URI 选项，可以与任何 API 方法相结合。要定义与特定 API 方法关联的 URI 选项，在 API 组件 Maven 插件中配置额外的选项。详情请查看 第 47.7 节 “额外选项”。

URI 格式

重新调用 API 组件 URI 的一般格式：

```
scheme://endpoint-prefix/endpoint?Option1=Value1&...&OptionN=ValueN
```

通常，URI 映射到 Java API 上的特定方法调用。例如，假设您要调用 API 方法，ExampleJavadocHello.greetMe("Jane Doe") 将被构造 URI，如下所示：

scheme

API 组件方案，如使用 Maven archetype 生成代码时所指定。在这种情况下，方案是 示例。

endpoint-prefix

API 名称，它映射到由 camel-api-component-maven-plugin Maven 插件配置定义的 API 类。对于 ExampleJavadocHello 类，相关配置为：

```
<configuration>
  <apis>
    <api>
      <apiName>hello-javadoc</apiName>
      <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
      <fromJavadoc/>
    </api>
    ...
  </apis>
</configuration>
```

这表明所需的 endpoint-prefix 是 hello-javadoc。

端点

端点 映射到方法名称，即 greetMe。

Option1=Value1

URI 选项指定方法参数。greetMe(String name) 方法采用单一参数，即 名称，它可以指定为 name=Jane%20Doe。如果要为选项定义默认值，可以通过覆盖 interceptProperties 方法（请参阅 第 46.4 节 “编程模型”）。

将 URI 的片段放在一起，我们可以看到您可以使用以下 URI 调用 ExampleJavadocHello.greetMe("Jane Doe")：

```
example://hello-javadoc/greetMe?name=Jane%20Doe
```

默认组件实例

要将 示例 URI 方案映射到默认组件实例，Maven archetype 会在 camel-api-example-component 子项目下创建以下文件：

```
src/main/resources/META-INF/services/org/apache/camel/component/example
```

此资源文件是什么使 Camel 内核能够识别与 示例 URI 方案关联的组件。每当您在路由中使用 example:// URI 时，Camel 都会搜索 classpath 以查找对应的 示例 资源文件。示例文件 包含以下内容：

```
class=org.jboss.fuse.example.ExampleComponent
```

这可让 Camel 核心创建 ExampleComponent 组件的默认实例。如果您重构了组件类的名称，您需要编辑此文件的唯一时间。

46.4. 编程模型

概述

在 API 组件框架的上下文中，主要组件实施类派生自 org.apache.camel.util.component 软件包中的基础类。这些基础类定义在实施组件时可以（可选）覆盖的一些方法。在本节中，我们提供了对这些方法的简单描述，以及如何在自己的组件实施中使用它们。

来实现的组件方法

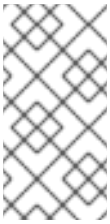
除了生成的方法实施（通常不需要修改），您也可以选择覆盖 组件 类中的以下方法：

doStart()

（可选）在冷启动期间为组件创建资源的回调。另一种方法是采用 lazy 初始化 策略（仅在需要资源时重新分配资源）。实际上，Lazy 初始化通常是最佳策略，因此经常不需要 doStart 方法。

doStop()

（可选）在组件停止时调用代码的回调。停止组件意味着其所有资源都关闭，内部状态会被删除，缓存会被清除，以此类推。



注意

当当前的 CamelContext 关闭时，即使没有调用对应的 doStart，Camel 保证在当前 CamelContext 关闭时始终会调用 doStop。

doShutdown

（可选）在 CamelContext 关闭时调用代码的回调。停止的组件可以重新启动（带有冷启动语义），一个被完全关闭的组件。因此，这个回调代表最后一次释放属于组件的资源的几率。

在组件类中实施哪些其他事项？

组件 类是存放对组件对象本身具有相同（或类似）生命周期的对象的引用。例如，如果组件使用 OAuth 安全性，在组件类中保留对所需 OAuth 对象的引用，并在 组件 类中定义用于创建 OAuth 对象的方法。

执行的端点方法

您可以修改一些生成的方法，并选择性地覆盖 Endpoint 类中的一些继承方法，如下所示：

afterConfigureProperties()

此方法中需要执行的主要操作是创建适当类型的代理类（API 类），以匹配 API 名称。API 名称（已从端点 URI 提取）可通过继承的 apiName 字段或通过 getApiName 访问器获得。通常，您要对 apiName 字段执行交换机来创建对应的代理类。例如：

```
// Java
private Object apiProxy;
...
@Override
protected void afterConfigureProperties() {
    // TODO create API proxy, set connection properties, etc.
    switch (apiName) {
        case HELLO_FILE:
            apiProxy = new ExampleFileHello();
            break;
        case HELLO_JAVADOC:
            apiProxy = new ExampleJavadocHello();
            break;
        default:
            throw new IllegalArgumentException("Invalid API name " + apiName);
    }
}
```

getApiProxy（ApiMethod 方法, Map<String, Object> args）

覆盖此方法，返回您在 ConfigureProperties 后创建的代理 实例。例如：

```
@Override
public Object getApiProxy(ApiMethod method, Map<String, Object> args) {
    return apiProxy;
}
```

特殊情况下，您可能想要选择取决于 API 方法和参数的代理。对于需要，getApiProxy 为您提供了使用此方法的灵活性。

doStart()

（可选） 在冷启动过程中创建资源的回调。具有与 Component.doStart（） 相同的语义。

doStop()

（可选） 在组件停止时调用代码的回调。具有与 Component.doStop（） 相同的语义。

doShutdown

（可选） 在关闭组件时调用代码的回调。具有与 Component.doShutdown（） 相同的语义。

interceptPropertyNames(Set<String> propertyNames)

（可选） API 组件框架使用端点 URI 和提供的选项值来确定调用的方法（可能源自于过载和别名）。如果组件在内部添加了选项或方法参数，但框架可能需要帮助来确定调用的正确方法。在这种情况下，您必须覆盖 interceptPropertyNames 方法，并为 set 的 propertyNames 添加 extra（hidden 或 implicit）选项。当设置了 propertyNames 中的完整方法参数列表时，框架将能够识别要调用的正确方法。

注意

您可以在 Endpoint、Producer 或 Consumer 类的级别上覆盖此方法。如果某个选项影响了制作者端点和消费者端点，则基本规则会覆盖 Endpoint 类中的方法。

interceptProperties(Map<String,Object> properties)

（可选） 通过覆盖此方法，您可以在调用 API 方法前修改或设置选项的实际值。例如，如果需要，您可以使用此方法为某些选项设置默认值。在实践中，通常需要覆盖 intercept Property Names 方法和 interceptProperty 方法。

注意

您可以在 Endpoint、Producer 或 Consumer 类的级别上覆盖此方法。如果某个选项影响了制作者端点和消费者端点，则基本规则会覆盖 Endpoint 类中的方法。

消费者方法实施

您可以选择在 Consumer 类中覆盖一些继承的方法，如下所示：

interceptPropertyNames(Set<String> propertyNames)

（可选） 此方法的语义与 Endpoint.interceptPropertyNames类似

interceptProperties(Map<String,Object> properties)

（可选） 此方法的语义类似于 Endpoint.interceptProperties

doInvokeMethod(Map<String, Object> args)

（可选） 使用此方法可以截获 Java API 方法的调用。覆盖此方法的最常见原因是自定义方法调用的错误处理。例如，以下代码片段中显示了覆盖 doInvokeMethod 的典型方法：

```
// Java
@Override
protected Object doInvokeMethod(Map<String, Object> args) {
    try {
        return super.doInvokeMethod(args);
    } catch (RuntimeCamelException e) {
        // TODO - Insert custom error handling here!
        ...
    }
}
```

在本次实施中的某个时刻，您应该调用 doInvokeMethod，以确保调用 Java API 方法。

拦截器Result(Object methodResult, Exchange resultExchange)

（可选） 对 API 方法调用的结果进行一些额外的处理。例如，您可以在 Camel Exchange 对象(resultExchange)中添加自定义标头。

对象分割结果（Object 结果）

（可选） 默认情况下，如果方法 API 调用的结果是 java.util.Collection 对象或 Java 数组，API 组件框架会将结果拆分为 多个 交换对象（因此单个调用结果转换为多个消息）。

如果要更改默认行为，您可以覆盖消费者端点中的 splitResult 方法。result 参数包含 API 消息调用的结果。如果要分割结果，您应该返回数组类型。

注意

您还可以通过在端点 URI 中设置 consumer.splitResult=false 来关闭默认分割。

实现生产者方法

您可以选择在 Producer 类中覆盖一些继承的方法，如下所示：

interceptPropertyNames(Set<String> propertyNames)

（可选） 此方法的语义与 Endpoint.interceptPropertyNames类似

interceptProperties(Map<String,Object> properties)

（可选） 此方法的语义类似于 Endpoint.interceptProperties

doInvokeMethod(Map<String, Object> args)

（可选） 此方法的语义与 Consumer.doInvokeMethod 类似。

拦截器Result(Object methodResult, Exchange resultExchange)

（可选） 此方法的语义与 Consumer.interceptResult 类似。

注意

Producer.splitResult（） 方法 不会被 调用，因此无法分割 API 方法，从而造成与消费者端点相同的方式。若要获得制作者端点的类似效果，您可以使用 Camel 的 split（） DSL 命令（标准企业级集成模式之一）来分割 集合 或数组结果。

消费者轮询和线程模型

API 组件框架中消费者端点的默认线程模型 会被调度来轮询消费者。这意味着，消费者端点中的 API 方法会定期调用调度的时间间隔。如需了解更多详细信息，请参阅 “计划轮询消费者实施”一节。

46.5. 组件实现示例

概述

由 Apache Camel 分发的几个组件已使用 API 组件框架帮助实现。如果您想要进一步了解使用框架实施 Camel 组件的技术，则最好学习这些组件实施的源代码。

Box.com

Camel Box 组件 演示了如何使用 API 组件框架建模和调用第三方 Box.com Java SDK。它还演示了框架如何能够自定义消费者轮询，从而支持 Box.com 的长轮询 API。

GoogleDrive

Camel GoogleDrive 组件 演示了 API 组件框架如何处理一种方法对象风格的 Google API。在本例中，URI 选项映射到一个方法对象，然后通过覆盖消费者和制作者中的 doInvoke 方法来调用。

Olingo2

Camel Olingo2 组件 演示了如何使用 API 组件框架包装基于回调的异步 API。本例演示了如何将异步处理推送到底层资源（如 HTTP NIO 连接）以使 Camel 端点更具资源效率。

第 47 章 配置 API 组件 MAVEN 插件

摘要

本章介绍了 API 组件 Maven 插件中所有可用的配置选项。

47.1. 插件配置概述

概述

API 组件 Maven 插件的主要目的是 camel-api-component-maven-plugin 来生成 API 映射类，该类实现端点 URI 和 API 方法调用之间的映射。通过编辑 API 组件 Maven 插件的配置，您可以自定义 API 映射的各个方面。

生成的代码的位置

API 组件 Maven 插件生成的 API 映射类默认放置在以下位置：

```
ProjectName-component/target/generated-sources/camel-component
```

先决条件

API 组件 Maven 插件的主要输入是 Java API 类和 Javadoc 元数据。这些插件可通过声明为常规 Maven 依赖项（其中 Javadoc Maven 依赖项应当声明为 所提供的 范围）来提供给插件。

设置插件

设置 API 组件 Maven 插件的建议方法是通过 API 组件 archetype 生成起始点代码。这会在 ProjectName-component/pom.xml 文件中生成默认插件配置，然后您可以为项目自定义这些插件。插件设置的主要方面包括：

1. 必须为 requisite Java API 和 Javadoc 元数据声明 Maven 依赖项。
2. 插件的基本配置在 pluginManagement 范围内声明（也会定义要使用的插件版本）。
3. 插件实例本身会被声明和配置。

4. ***build-helper-maven 插件配置为从 target/generated-sources/camel-component 目录中获取生成的源，并将它们包含在 Maven 构建中。***

基本配置示例

以下 POM 文件提取显示 API 组件 Maven 插件的基本配置，如使用 API 组件 archetype 生成代码时 Maven pluginManagement 范围中所定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <build>
    ...
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.camel</groupId>
          <artifactId>camel-api-component-maven-plugin</artifactId>
          <version>2.23.2.fuse-7_10_0-00018-redhat-00001</version>
          <configuration>
            <scheme>${schemeName}</scheme>
            <componentName>${componentName}</componentName>
            <componentPackage>${componentPackage}</componentPackage>
            <outPackage>${outPackage}</outPackage>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
    ...
  </build>
  ...
</project
```

pluginManagement 范围中指定的配置提供了插件的默认设置。它实际上不会创建插件实例，但其默认设置将被任何 API 组件插件实例使用。

基本配置

除了指定插件版本（在 version 元素中），前面的基本配置指定了以下配置属性：

scheme

此 API 组件的 URI 方案。

componentName

此 API 组件的名称（也用作生成的类名称的前缀）。

componentPackage

指定包含 API 组件 Maven archetype 生成的类的 Java 软件包。此软件包也会由默认的 maven-bundle-plugin 配置导出。因此，如果您想要一个类公开可见，您应该将其放置在这个 Java 软件包中。

outPackage

指定放置生成的 API 映射类的 Java 软件包（当它们由 API 组件 Maven 插件生成时）。默认情况下，它具有 componentName 属性的值，增加了 .internal 后缀。默认 maven-bundle-plugin 配置声明这个软件包为私有。因此，如果您希望某一类是私有的，您应该将其放置在这个 Java 软件包中。

实例配置示例

以下 POM 文件提取显示了 API 组件 Maven 插件的示例实例，它配置为在 Maven 构建期间生成 API 映射：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">

  ...
  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>
      ...
      <!-- generate Component source and test source -->
      <plugin>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-api-component-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>generate-test-component-classes</id>
            <goals>
              <goal>fromApis</goal>
            </goals>
            <configuration>
              <apis>
                <api>
                  <apiName>hello-file</apiName>
                  <proxyClass>org.jboss.fuse.example.api.ExampleFileHello</proxyClass>
                  <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile>
                </api>
                <api>
                  <apiName>hello-javadoc</apiName>
```

```
            <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
            <fromJavadoc/>
          </api>
        </apis>
      </configuration>
    </execution>
  </executions>
 </plugin>
 ...
</plugins>
...
</build>
...
</project>
```

基本映射配置

该插件由 configuration 元素配置，其中包含一个 apis 子元素来配置 Java API 类。每个 API 类都由一个 api 元素配置，如下所示：

apiName

API 名称是 API 类的简短名称，用作 端点 URI 的端点前缀 部分。

注意

如果 API 只包含一个 Java 类，您可以将 apiName 元素留空，以便 endpoint-prefix 变为冗余，您可以使用 “单个 API 类的 URI 格式”一节 所示的格式指定端点 URI。

proxyClass

此元素指定 API 类的完全限定名称。

fromJavadoc

如果 API 类由 Javadoc 元数据提供，您必须通过包含 fromJavadoc 元素和 Javadoc 本身在内的 Javadoc 本身作为 提供的 依赖项来指明这一点。

fromSignatureFile

如果 API 类使用签名文件元数据，您必须通过包含 fromSignatureFile 元素来指明这一点，此元素的内容指定了签名文件的位置。

注意

签名文件不包括在 Maven 构建的最终软件包中，因为这些文件仅在构建时不需要。

自定义 API 映射

可通过配置插件来自定义 API 映射的以下方面：

- 可以使用 别名 配置元素为 API 方法定义额外名称（ 别名 ）。详情请查看 第 47.3 节 “方法别名”。
- nullable options you 可以使用 nullableOptions 配置元素来声明默认为 null 的方法参数。详情请查看 第 47.4 节 “nullable 选项”。
- 解决 API 映射的实现方式的参数 替换 InventoryService- theue，特定 API 类中所有方法的参数 属于同一 命名空间。如果将具有相同名称的两个参数声明为不同的类型，则会导致冲突。要避免这种名称冲突，您可以使用 替换 配置元素来重命名方法参数（与 URI 中显示的是）。详情请查看 第 47.5 节 “参数名称替换”。
- 当 参数 InventoryService-时间将 Java 参数映射到 URI 选项时，有时您可能想从映射中排除某些参数。您可以通过指定 excludeConfigNames 元素或 excludeConfigTypes 元素来过滤不需要的参数。详情请查看 第 47.6 节 “排除参数”。
- 您可能想定义额外选项（不是 Java API 的一部分）时，您可能需要定义额外选项。您可以使用 extraOptions 配置元素进行此操作。

配置 Javadoc 元数据

可以过滤 Javadoc 元数据以忽略或明确包含某些内容。有关如何进行此操作的详情，请参考 第 47.2 节 “javadoc 选项”。

配置签名文件元数据

如果没有可用的 Javadoc，您可以使用签名文件来提供所需的映射元数据。fromSignatureFile 用于指定相应签名文件的位置。它没有特殊选项。

47.2. JAVADOC 选项

概述

如果 Javadoc 提供了 Java API 元数据，通常足以指定没有选项的 fromJavadoc 元素。但是，如果您不想在 API 映射中包含整个 Java API，您可以过滤 Javadoc 元数据来自定义内容。换句话说，因为 API 组件 Maven 插件通过迭代 Javadoc 元数据来生成 API 映射，因此可以通过过滤 Javadoc 元数据中的不需要的部分来自定义所生成的 API 映射的范围。

语法

fromJavadoc 元素可使用可选的子元素配置，如下所示：

```
<fromJavadoc>
  <excludePackages>PackageNamePattern</excludePackages>
  <excludeClasses>ClassNamePattern</excludeClasses>
  <excludeMethods>MethodNamePattern</excludeMethods>
  <includeMethods>MethodNamePattern</includeMethods>
  <includeStaticMethods>[true|false]<includeStaticMethods>
</fromJavadoc>
```

影响范围

如以下摘录所示，fromJavadoc 元素可选择性地显示为 apis 元素的子项，/或作为 api 元素的子项：

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <fromJavadoc>...</fromJavadoc>
    </api>
    <fromJavadoc>...</fromJavadoc>
    ...
  </apis>
</configuration>
```

您可以在以下范围中定义 fromJavadoc 元素：

- 作为 api 元素 targetNamespaces- the fromJavadoc 选项的子项，仅适用于 api 元素指定的 API 类。
-

作为 apis 元素 的子项，Javadoc 选项中的 Javadoc 选项默认应用于所有 API 类，但可以在 api 级别覆盖。

选项

以下选项可以定义为 Javadoc 的子元素：

excludePackages

指定用于从 API 映射模型中排除 Java 软件包的正则表达式（java.util.regex 语法）。所有与正则表达式匹配的软件包名称都被排除，并且所有 从排除类派生的类都会被忽略。默认值为 javax?\.lang.*。

excludeClasses

指定用于从 API 映射中排除 API 基本类的正则表达式（java.util.regex 语法）。所有与正则表达式匹配的类名称都不包括；并且所有 从排除类派生的类都会被忽略。

excludeMethods

指定一个正则表达式（java.util.regex 语法），用于排除 API 映射模型中的方法。

includeMethods

指定正则表达式（java.util.regex 语法），用于包括来自 API 映射模型的方法。

includeStaticMethods

如果为 true，则 API 映射模型中也会包含静态方法。默认为 false。

47.3. 方法别名

概述

除了 Java API 中显示的标准方法名称外，通常对于定义给定方法的其他名称（别名）也很有用。某种常见情形是，您允许将属性名称（如 小部件）用作 accessor 方法的别名（如 getWidget 或 setWidget）。

语法

alias 元素可使用一个或多个别名子元素来定义，如下所示：

```
<aliases>
  <alias>
    <methodPattern>MethodPattern</methodPattern>
    <methodAlias>Alias</methodAlias>
  </alias>
  ...
</aliases>
```

这里的 MethodPattern 是一个正则表达式（java.util.regex 语法），用于与 Java API 中的匹配方法名称，模式通常包括捕获组。Alias 是替换表达式（用于在 URI 中使用），它使用前面捕获组中的文本（例如，以 $1、$2 或 $3 ）用于第一个、第二个或第三个捕获组中的文本。

影响范围

如以下摘录所示，别名 元素可以选择性地显示为 apis 元素的子项，和/或作为 api 元素的子项：

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <aliases>...</aliases>
    </api>
    <aliases>...</aliases>
    ...
  </apis>
</configuration>
```

您可以在以下范围中定义 别名 元素：

- *作为 api 元素 的子项 the alias 映射只适用于 api 元素指定的 API 类。*
- *作为 apis 元素的子项 the aliases 映射默认应用到所有 API 类，但可以在 api 级别上覆盖。*

示例

以下示例演示了如何为 common get/set Bean 方法模式生成别名：

```
<aliases>
  <alias>
    <methodPattern>[gs]et(.+)</methodPattern>
```

```
    <methodAlias>$1</methodAlias>
  </alias>
</aliases>
```

使用上述别名定义时，您可以使用 widget 作为方法 getWidget 或 setWidget 的别名。请注意，使用捕获组 (.+) 来捕获方法名称的后一种部分（例如，Bloy s）。

47.4. NULLABLE 选项

概述

在某些情况下，让方法参数默认为 null。但默认情况下不允许这样做。如果要允许 Java API 中的一些方法参数使用 null 值，则必须使用 nullableOptions 元素明确声明它。

语法

nullableOptions 元素可使用一个或多个 nullableOption 子元素定义，如下所示：

```
<nullableOptions>
  <nullableOption>ArgumentName</nullableOption>
  ...
</nullableOptions>
```

其中 ArgumentName 是 Java API 中的 method 参数的名称。

影响范围

如以下摘录所示，nullableOptions 元素可以选择性地显示为 apis 元素的子项，/或作为 api 元素的子项：

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <nullableOptions>...</nullableOptions>
    </api>
    ...
    <nullableOptions>...</nullableOptions>
  </apis>
</configuration>
```

您可以在以下范围中定义 **nullableOptions** 元素：

- 作为 **api** 元素的子项 **the nullableOptions** 映射只适用于 **api** 元素指定的 **API** 类。
- 作为 **apis** 元素的子项 **the nullableOptions** 映射默认应用到所有 **API** 类，但可以在 **api** 级别上覆盖。

47.5. 参数名称替换

概述

API 组件框架要求 **URI** 选项名称 在每个代理类（**Java API** 类）中是唯一的。然而，方法参数名称并非始终如此。例如，在 **API** 类中考虑以下 **Java** 方法：

```
public void doSomething(int id, String name);
public void doSomethingElse(int id, String name);
```

当您构建 **Maven** 项目时，**camel-api-component-maven-plugin** 生成配置类，**ProxyClassEndpointConfiguration**，其中包含 **ProxyClass** 类中 所有参数 的 **getter** 和 **setter** 方法。例如，根据前面的方法，插件会在配置类中生成以下 **getter** 和 **setter** 方法：

```
public int  getId();
public void setId(int id);
public String getName();
public void   setName(String name);
```

但是，如果 **id** 参数以不同的类型出现多次，如下例所示：

```
public void doSomething(int id, String name);
public void doSomethingElse(int id, String name);
public String lookupByID(String id);
```

在这种情况下，代码生成会失败，因为您无法定义返回 **int** 的 **getId** 方法以及返回相同范围内的 **String** 方法。此问题的解决方案是使用参数名称替换 来定制参数名称 到 **URI** 选项名称的映射。

语法

替换 元素可以定义为一个或多个 替换 子元素，如下所示：

```
<substitutions>
  <substitution>
    <method>MethodPattern</method>
    <argName>ArgumentNamePattern</argName>
    <argType>TypeNamePattern</argType>
    <replacement>SubstituteArgName</replacement>
    <replaceWithType>[true|false]</replaceWithType>
  </substitution>
  ...
</substitutions>
```

其中 argType 元素和 replaceWithType 元素是可选的，可以省略。

影响范围

如以下提取所示，替换 项可以选择性地显示为 apis 元素的子项，/或作为 api 元素的子项：

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <substitutions>...</substitutions>
    </api>
    <substitutions>...</substitutions>
    ...
  </apis>
</configuration>
```

您可以在以下范围中定义 替换 元素：

- *作为 api 元素的子项 the the substitutions 只适用于 api 元素指定的 API 类。*
- *作为 apis 元素的子项 the substitutions 默认应用于所有 API 类，但可以在 api 级别上覆盖。*

子元素

每个 替换 元素均可通过以下子元素定义：

method

指定正则表达式（java.util.regex 语法），以匹配 Java API 中的方法名称。

argName

指定正则表达式（java.util.regex 语法），以匹配匹配匹配方法的参数名称，其中模式通常包含捕获组。

argType

（可选）指定正则表达式（java.util.regex 语法）以匹配参数的类型。如果将 replaceWithType 选项设置为 true，则通常会在这个正则表达式中使用捕获组。

替换

根据 方法 模式、argName 模式和（可选） argType 模式的特定匹配项，替换 元素定义了替换参数名称（用于 URI 中使用）。可使用从 argName 正则表达式模式捕获的字符串来构建替换文本（使用语法 $1、$2、$3 来分别插入第一个、第二个或第三个捕获组）。或者，如果使用从 argType 正则表达式模式捕获的字符串来构建替换文本，如果您将 replaceWithType 选项设置为 true。

replaceWithType

为 true 时，指定使用从 argType 正则表达式捕获的字符串来构建替换文本。默认值为 false。

示例

以下替换示例通过在参数名称中添加后缀 Param 来修改 java.lang.String 类型的每个参数：

```
<substitutions>
  <substitution>
    <method>^.+$</method>
    <argName>^.+$</argName>
    <argType>java.lang.String</argType>
    <replacement>$1Param</replacement>
    <replaceWithType>false</replaceWithType>
  </substitution>
</substitutions>
```

例如，给出以下方法签名：

```
public String greetUs(String name1, String name2);
```

此方法的参数将通过端点 URI 中的选项、name1Param 和 name2Param 来指定。

47.6. 排除参数

概述

有时，在将 Java 参数映射到 URI 选项时，您可能需要排除某些参数。您可以通过在 camel-api-component-maven-plugin 插件配置中指定 excludeConfigNames 元素或 excludeConfigTypes 元素来过滤不需要的参数。

语法

excludeConfigNames 元素和 excludeConfigTypes 元素指定如下：

```
<excludeConfigNames>ArgumentNamePattern</excludeConfigNames>
<excludeConfigTypes>TypeNamePattern</excludeConfigTypes>
```

其中 ArgumentNamePattern 和 TypeNamePattern 是与参数名称和参数类型匹配的正则表达式。

影响范围

如以下摘录所示，excludeConfigNames 元素和 excludeConfigTypes 元素可以选择性地显示为 apis 元素和/或 api 元素的子项：

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <excludeConfigNames>...</excludeConfigNames>
      <excludeConfigTypes>...</excludeConfigTypes>
    </api>
    <excludeConfigNames>...</excludeConfigNames>
    <excludeConfigTypes>...</excludeConfigTypes>
    ...
  </apis>
</configuration>
```

您可以在以下范围中定义 excludeConfigNames 元素和 excludeConfigTypes 元素：

- 作为 api 元素的子项 operators，仅应用于 api 元素指定的 API 类。

- 作为 apis 元素的子项 targetNamespaces-excludethe exclusions 默认应用到所有 API 类，但可以在 api 级别覆盖。

元素

以下元素可用于从 API 映射中排除参数（因此它们作为 URI 选项不可用）：

excludeConfigNames

根据匹配的参数，指定用于排除参数的正则表达式（java.util.regex 语法）。

excludeConfigTypes

根据匹配的参数类型，指定用于排除参数的正则表达式（java.util.regex 语法）。

47.7. 额外选项

概述

通过提供更简单的选项，通常使用 extraOptions 选项计算或隐藏复杂的 API 参数。例如，API 方法可能会采用 POJO 选项，它可以作为 URI 中的 OVA 部分来更轻松地提供。组件可以通过添加部分作为额外选项来实现此目的，并在内部创建 OVA 参数。要完成这些额外选项的实现，您还需要覆盖 EndpointConsumer 和/或 EndpointProducer 类中的 interceptProperties 方法（请参阅 第 46.4 节 “编程模型”。

语法

extraOptions 元素可使用一个或多个 extraOption 子元素定义，如下所示：

```
<extraOptions>
  <extraOption>
    <type>TypeName</type>
    <name>OptionName</name>
  </extraOption>
</extraOptions>
```

其中 TypeName 是额外选项的完全限定类型名称，option optionName 是额外 URI 选项的名称。

影响范围

如以下摘录所示，extraOptions 元素可以选择性地显示为 apis 元素的子项，/或作为 api 元素的子项：

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <extraOptions>...</extraOptions>
    </api>
    <extraOptions>...</extraOptions>
    ...
  </apis>
</configuration>
```

您可以在以下范围中定义 extraOptions 元素：

- *作为 api 元素 的子级，extraOptions 仅适用于 api 元素指定的 API 类。*
- *作为 apis 元素 的子项，extraOptions 默认应用到所有 API 类，但可以在 api 级别上覆盖。*

子元素

每个 extraOptions 元素可使用以下子元素定义：

type

指定额外选项的完全限定类型名称。

name

指定选项名称，因为它将显示在端点 URI 中。

示例

以下示例定义了一个额外的 URI 选项 customOption，它是 java.util.list<String> 类型：

```
<extraOptions>
```

```
  <extraOption>
    <type>java.util.List<String></type>
    <name>customOption</name>
  </extraOption>
</extraOptions>
```

索引