Red Hat Fuse 7.13

Apache Camel 开发指南

使用 Apache Camel 开发应用程序

Last Updated: 2024-06-18

Red Hat Fuse 7.13 Apache Camel 开发指南

使用 Apache Camel 开发应用程序

法律通告

摘要

本指南论述了如何使用 Apache Camel 开发 JBoss Fuse 应用程序。它涵盖了基本构建块、企业集成模式、路由表达式和 predicate 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 Web 组件以及如何创建打包任何 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。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看我们的 CTO Chris Wright 信息。

部分 I. 实施企业集成模式

这部分描述了如何使用 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 支持两种替代 **域特定语言** (DSL)用于定义路由：Java DSL 和 Spring XML DSL。定义路由的基本构建块是 *端点和 处理器*，处理器的行为通常由表达式或逻辑 *predicates* 修改。*Apache Camel 允许您使用各种不同的语言来定义表达式和 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任意数量的 **RouteBuilder** 类。每个类都实例化一次，并使用 **CamelContext** 对象注册。通常，每个 **RouteBuilder** 对象的生命周期由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。您可以从以下位置继承两种替代 **RouteBuilder** 类：*

- ***org.apache.camel.builder.RouteBuilder**. **RouteBuilder** -wagon this 是适合 **部署到任何** 容器类型的通用 RouteBuilder 基本类。它在 **camel-core** 工件中提供。*
- ***org.apache.camel.spring.SpringRouteBuilder** HEKETI-wagon this base 类特别适应 Spring 容器。特别是，它提供对以下 Spring 具体功能的额外支持：在 Spring registry （使用 **beanRef （）** Java DSL 命令）和事务中查找 Bean （请参阅 **Transactions 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，**from （）**、**intercept （）** 和 **exception （）**）。*

实施 RouteBuilder

*例 1.1 “RouteBuilder 类的实现” 显示最小 **RouteBuilder** 实施。**configure （）** 方法正文包含一个路由规则；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.
  // ...
}
}
```

规则 **from (URL1).to (URL2)** 的形式指示路由器从 **src/data** 目录读取文件，并将它们发送到目录 **target/messages**。选项 **?noop=true** 指示路由器在 **src/data** 目录中保留（不会删除）源文件。

注意

当您将 **contextScan** 与 Spring 或 Blueprint 搭配使用时，过滤 **RouteBuilder** 类，默认情况下，Apache Camel 将查找单例 Bean。但是，您可以打开旧行为，使其包含有新选项 **includeNonSingletons** 的范围。

1.2. 基本 JAVA DSL 语法

什么是 DSL？

域特定语言(DSL)是专为特殊目的设计的小型语言。DSL 不必逻辑完成，但需要足够的表达力，以便在所选域中充分描述问题。通常，DSL **不需要** 专用的解析器、解释器或编译器。DSL 可以在面向对象的现有对象的主机语言上得到缓解，提供的 DSL 构造可以完全映射到主机语言 API 中。

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()
```

您甚至可以将块映射到 Java 方法调用。例如：

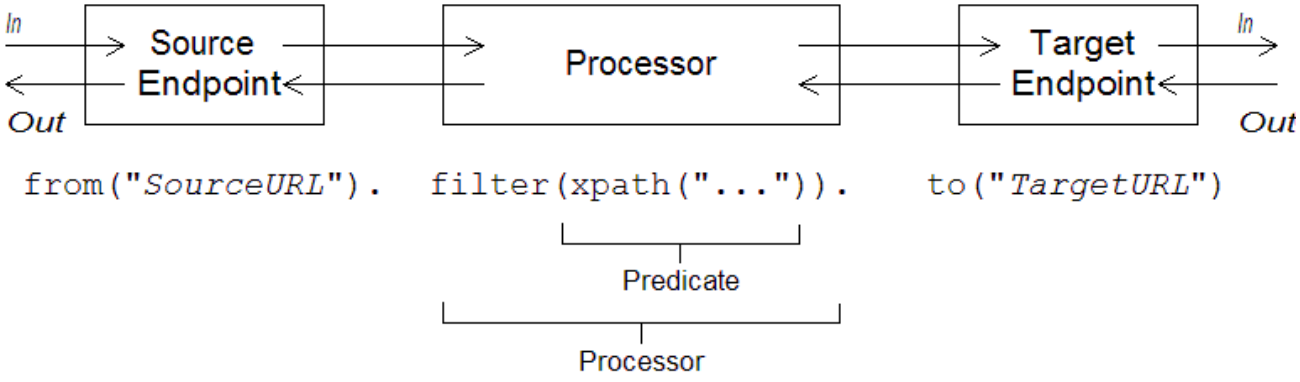
```
command01().startBlock().command02().command03().endBlock()
```

DSL 语法通过主机语言 API 的数据类型隐式定义。例如，一种 Java 方法的返回类型决定了您可以在接下来进行法律调用的方法（等同于 DSL 中的下一个命令）。

路由器规则语法

Apache Camel 定义了 路由器 DSL，用于定义路由规则。您可以使用此 DSL 在 **RouteBuilder.configure()** 实现的正文中定义规则。图 1.1 "本地路由规则" 显示用于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 **from ("EndpointURL")** 方法开头，该方法指定路由规则的消息源(消费者端点)。然后，您可以在规则（如 **filter （）**）中添加任意长的处理器链。您通常使用 **to ("EndpointURL")** 方法完成规则，该方法为通过规则传递的消息指定目标(制作者端点)。但是，并不总是需要以 **to （）** 结尾规则。也可以使用其他方法在规则中指定消息目标。

注意

您还可以通过使用特殊的处理器类型（如 **intercept （）**、**exception （）** 或 **errorHandler （）**）启动规则来定义全局路由规则。全局规则超出了本指南的范围。

消费者和制作者

本地规则始终首先使用 **from ("EndpointURL")** 定义消费者端点，并且通常（但不总是）通过定义制作者端点（使用 **to ("EndpointURL")**）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。

Exchanges

Exchange 对象 由消息组成，由元数据增强。交换是 Apache Camel 中的中央重要性，因为交换是通过路由规则传播消息的标准表单。交换的主要因素是，如下所示：

- **在消息** swig-wagonis 由交换封装的当前消息。当交换通过路由进行时，可以修改此消息。因此，路由开头的 In 消息通常与路由末尾的 In 消息不同。**org.apache.camel.Message 类型提供消息的通用模型，以及以下部分：**
  - **正文.**
  - 标头.
  - **附加功能.**

  务必要意识到这是一个 **通用 消息模型。Apache Camel 支持各种协议和端点类型。因此，无法 标准化消息正文或消息标头的格式。例如，JMS 消息的正文格式与 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（即，端点出现在 from （） 命令中）。**

- out message to the a temporary holding area for a ephemeral holding area for a reply message or for a transformed message.**某些处理节点（特别是 to （） 命令）可以通过将 In 消息视为请求来修改当前消息，将其发送到制作者端点，然后从该端点接收回复。然后，回复消息将插入到交换中的 Out 消息插槽中。**
  通常，如果当前节点设置了 Out 消息，则 Apache Camel 会按照如下方式修改交换，然后将其传递到路由中的下一个节点：将丢弃旧的 In 消息，并且 Out 消息将移到 In 消息插槽。因此，回复会成为新的当前消息。有关 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，有一个特殊情况处理 Out 消息不同。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，或者消费者端点挂起）。

- **消息交换模式(MEP)是要影响路由中交换和端点之间的交互，如下所示：**
  - **创建原始交换的消费者端点，使用者端点会设置 MEP 的初始值。初始值指示消费者端点是否需要收到回复（例如，InOut MEP）还是不（例如 InOnly MEP）。**

- 生产者端点 HEKETI.MEP 会影响交换在路由中遇到的制作者端点（例如，当交换通过 to （） 节点时）。例如，如果当前的 MEP 是 InOnly，则 a to （） 节点不会预期从端点接收回复。有时您需要更改当前的 MEP，以便自定义交换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”。

- Exchange properties swig-wagona 的指定属性列表，其中包含当前消息的元数据。

消息交换模式

使用 Exchange 对象可以轻松地将消息处理常规化到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，从消费者端点流到生成者端点(InOnly MEP)。另一方面，RPC 协议可能会定义一个 MEP，它由请求消息和一个回复消息(InOut MEP)。目前，Apache Camel 支持以下 MEP：

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

其中，这些消息交换模式由枚举类型 **org.apache.camel.ExchangePattern** 中的常量表示。

分组的交换

有时，具有封装多个交换实例的单个交换很有用。为此，您可以使用 分组的交换。分组的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** Exchange 对象。有关如何使用分组的交换的示例，请参阅 第 8.5 节 “聚合器”。

Processors

处理器 是路由中的节点，可以访问和修改通过路由的交换流。处理器可以取 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，该处理器将 **xpath** （） predicate 用作其参数。

表达式和 predicates

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

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

其中过滤器由 predicate, **header ("foo").isEqualTo ("bar")** 限定。要基于消息内容构建更复杂的 predicates 和表达式，您可以使用表达式和 predicate 语言之一（请参阅 第 II 部分 “路由表达式和 predicates 语言”）。

1.3. SPRING XML 文件中的路由器架构

Namespace

路由器模式都包括在以下 XML 模式命名空间中：

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

指定模式位置

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 网站中的模式的最新版本。例如，Apache Camel Spring 文件的 root **Bean** 元素通常配置，如 *例 1.2 “指定路由器架构位置”* 所示。

> 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>
> ```

Runtime 模式位置

在运行时，Apache Camel 不会从 Spring 文件中指定的模式位置下载路由器模式。相反，Apache Camel 会自动从 **camel-spring** JAR 文件的根目录中获取 schema 的副本。这样可确保用于解析 Spring 文件的模式版本始终与当前的运行时版本匹配。这一点很重要，因为发布在 Apache Web 站点上的架构的最新版本可能与您当前使用的运行时版本不匹配。

使用 XML 编辑器

通常，建议您使用功能齐全的 XML 编辑器编辑 Spring 文件。XML 编辑器的自动完成功能可以更轻松地编写 XML，该 XML 符合路由器模式，如果 XML 的格式不正确，编辑器可以立即警告您。

XML 编辑器通常依赖从 **xsi:schemaLocation** 属性中指定的位置下载模式。为了确保您使用正确的模式版本 whilst 编辑，最好选择 **camel-spring.xsd** 文件的特定版本。例如，要为 Apache Camel 的 2.3 版本编辑 Spring 文件，您可以修改 beans 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">
...
```

完成编辑后，重新更改为默认的 **camel-spring.xsd**。要查看当前可用的模式版本，请导航到网页 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。端点是一个非常通用的构建块类型：它必须满足的唯一要求是它充当消息源（生产者端点）或消息接收器（消费者端点）。因此，Apache Camel 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，它会定期生成 dummy 消息。Apache Camel 的主要优势之一是添加实施新端点类型的自定义组件相对简单。

端点 URI

端点通过 端点 URI 来标识，其具有以下通用形式：

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

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

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

例如，以下 HTTP URI 可用于连接到 Google 搜索引擎页面：

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

以下 File URI 可用于读取 **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 都代表一个协议。有时，方案 只能提供对有用的实用程序（如计时器）的访问。例如，以下 Timer 端点 URI 每秒生成交换(=1000 毫秒)。您可以使用它来调度路由中的活动。

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

使用 Long Endpoint URI

有时，因为提供的所有配置信息，端点 URI 可能会变得非常长。在 JBoss Fuse 6.2 中，您可以采取两种方法使您使用长度的 URI 更易管理。

单独配置端点

您可以单独配置端点，并从路由中使用其简写 ID 来指代端点。

```
<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 属性来指定其他选项（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，每个选项都由 和 分隔。

在 URI 中指定时间段

许多 Apache Camel 组件都有选项，其值是一个时间段（例如，指定超时值等）。默认情况下，此类时间段选项通常指定为纯数字，它被解释为毫秒的时间段。但是 Apache Camel 还支持更易读的语法，用于按小时、分钟和秒表示周期。正式来说，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)]
```

其中，在方括号([]) 中的每个术语都是可选的，表示法 (A|B) 表示 A 和 B 是替代方案。

例如，您可以使用 45 分钟周期配置 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采用 URI 编码。在某些情况下，这是不必要的行为。例如，在设置 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。

可以通过指定带有语法 RAW (RawValue) 的选项值来关闭 URI 编码。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

有些端点 URI 选项映射到 Java enum 常量。例如，Log 组件的 level 选项，它可以使用 enum 值、INFO、WARN、ERROR 等。这个类型转换是不区分大小写的，因此以下任意一种替代方法可用于设置 Log producer 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

在 Camel 2.17 中，基于资源的组件（如 XSLT）可以使用 ref: 作为前缀从 Registry 中加载资源文件。

例如，myvelocityscriptbean 和 mysimplescriptbean 是 registry 中两个 Bean 的 ID，您可以使用这些 Bean 的内容，如下所示：

```
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 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，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 提供了各种不同的组件，允许您将应用程序与各种传输协议和第三方产品集成。例如，一些更常用的组件有：file、JMS、CXF (Web 服务)、HTTP、Jetty、Direct 和 Mock。有关支持的组件的完整列表，请参阅 Apache Camel 组件文档。

大多数 Apache Camel 组件都单独打包到 Camel 内核。如果使用 Maven 构建应用程序，则只需添加依赖相关组件工件，即可将组件（及其第三方依赖项）添加到应用程序中。例如，要包含 HTTP 组件，您将在项目 POM 文件中添加以下 Maven 依赖项：

```
<!-- 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-core** 工件中），因此它们始终可用：

- Bean
- 浏览
- Dataset
- direct
- File
- Log
- Mock
- Properties
- Ref
- SEDA
- 计时器
- VM

消费者端点

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

例如，以下 JMS 使用者端点从 支付 队列中提取消息，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>
```

有些组件只是消费者，而用户只能用来定义消费者端点。例如，Qartz 组件专门用于定义使用者端点。以下 Quartz 端点每秒生成一个事件(1000 毫秒)：

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

如果愿意，您可以使用 fromF （） Java DSL 命令将端点 URI 指定为格式的字符串。例如，要替换 FTP 端点的 URI 中的用户名和密码，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");
```

如果第一次出现 %s 被 username 字符串的值替换，第二个出现的 %s 将被密码字符串替代。这个字符串格式化机制通过 String.format （） 实施，它与 C printf （） 函数提供的格式类似。详情请参阅 java.util.Formatter。

制作者端点

producer 端点是出现在中间或路由末尾的端点（例如，在 to （） DSL 命令中）。换句话说，生成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 队列中：

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

或者等效在 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>
```

有些组件只是生成者，它只是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");
```

如果愿意，您可以使用 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);
```

其中出现的 %s 替换为您的自定义查询字符串 myGoogleQuery。详情请参阅 java.util.Formatter。

1.5. PROCESSORS

概述

要让路由器执行更有趣的事情，而只是将消费者端点连接到制作者端点，您可以将 处理器 添加到您的路由中。处理器是您可以插入到路由规则中的命令，以执行任意处理通过该规则的消息。Apache Camel 提供了各种不同的处理器，如 表 1.1 “Apache Camel Processors” 所示。

表 1.1. Apache Camel Processors

Java DSL	XML DSL	描述
aggregate()	**聚合**	第 8.5 节 “聚合器”: 创建一个聚合器，它将多个传入的交换合并到一个交换中。
aop （）	**aop**	使用 Aspect Oriented programming (AOP)在指定的子路由之前和之后工作。

Java DSL	XML DSL	描述
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 集成”。
choice()	**choice**	第 8.1 节 “基于内容的路由器”: 使用 **when** 和 **otherwise** 子句，根据交换内容选择特定的子路由。
convertBodyTo()	**convertBodyTo**	将 In message body 转换为指定的类型。
delay ()	**delay**	第 8.9 节 “Delayer”: 延迟交换传播到路由部分的传播。
doTry()	**doTry**	使用 **doCatch**、**doFinally** 和 **end** 子句创建用于处理异常的尝试/请求块。
end()	N/A	结束当前命令块。
enrich(),enrichRef()	**enrich**	第 10.1 节 “内容增强”: 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 “idempotent Consumer”: 实施一个策略来阻止重复的消息。
inheritErrorHandler()	**@inheritErrorHandler**	布尔值选项，可用于禁用特定路由节点上的继承错误处理程序（定义为 Java DSL 中的子层，以及 XML DSL 中的属性）。
inOnly()	**InOnly**	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。
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。

Java DSL	XML DSL	描述
loadBalance()	**loadBalance**	第 8.10 节 “Load Balancer”: 通过一系列端点实施负载平衡。
log()	**log**	将消息记录到控制台。
loop()	**loop**	第 8.16 节 “loop”: 重复发送每个交换到路由的后一种部分。
markRollbackOnly()	**@markRollbackOnly**	**（事务处理）** 仅标记当前用于回滚的事务（不会引发异常）。在 XML DSL 中，这个选项被设置为 **rollback** 元素的布尔值属性。请参阅 Apache Karaf 交易指南。
markRollbackOnlyLast()	**@markRollbackOnlyLast**	**（事务）** 如果之前与此线程关联了一个或多个事务，然后暂停，这个命令会标记最新的事务只进行回滚（不会引发异常）。在 XML DSL 中，这个选项被设置为 **rollback** 元素的布尔值属性。请参阅 Apache Karaf 交易指南。
marshal()	**marshal**	使用指定的数据格式转换为低级或二进制格式，以准备通过特定的传输协议发送。
multicast()	**multicast**	第 8.13 节 “多播”: 多播当前交换到多个目的地，每个目标都会获得其自身的交换副本。
onCompletion()	**onCompletion**	定义在主路由完成后执行的子路由（由 Java DSL 中的 **end ()** 结尾）。另请参阅 第 2.14 节 “OnCompletion”。
onException()	**onException**	定义在指定异常发生时执行的子路由（由 Java DSL 中的 **end ()** 结尾）。通常在其自己的行中（不在路由中定义）。
pipeline()	**pipeline**	第 5.4 节 “管道和过滤器”: 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**	对当前路由应用策略（目前仅适用于事务策略），请参阅 Apache Karaf 事务指南。

Java DSL	XML DSL	描述
pollEnrich(),pollEnrichRef()	**pollEnrich**	第 10.1 节 “内容增强”: 将当前交换与从指定的 **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方列表（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。模式可以具有表单 **prefix*** HEKETI-rhacmin，在这种情况下，它与以 prefix swig-wagonotherwise 开头的每个名称匹配，它被解释为正则表达式。
removeProperty()	**removeProperty**	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 或更多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()	**resequence**	第 8.6 节 “Resequencer”: 根据指定的 comparotor 操作对传入的交换进行排序。支持 **批处理模式** 和 **流模式**。
rollback()	**rollback**	**（事务处理）** 仅标记当前用于回滚的事务（默认为异常）。请参阅 Apache Karaf 交易指南。
routingSlip()	**routingSlip**	第 8.7 节 “路由片段”: 根据从 slip 标头中提取的端点 URI 列表，通过动态构建的管道路由交换。
sample()	**示例**	创建抽样节流，允许您从路由上的流量提取交换示例。
setBody()	**setBody**	设置交换的 In 消息的消息正文。

Java DSL	XML DSL	描述
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**	在交换的 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 消息正文的内容排序（可以选择性地指定自定义比较器的位置）。
split()	**split**	第 8.4 节 “Splitter”: 将当前交换分成一系列交换，每个分割交换都包含原始消息正文的片段。
stop()	**stop**	停止路由当前交换并将其标记为 completed。
threads （）	**threads**	创建一个线程池，用于并发处理路由部分。
throttle()	**throttle**	第 8.8 节 “Throttler”: 将流率限制为指定的级别（每秒交换数）。
throwException()	**throwException**	抛出指定的 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	使用字符串格式化将交换发送到端点。也就是说，端点 URI 字符串可以嵌入 C **printf （）** 函数样式中的替换。
transacted()	**Transacted**	创建一个 Spring 事务范围，其中包含路由的后者部分。请参阅 Apache Karaf 交易指南。
transform()	**转换**	第 5.6 节 “消息 Translator”: 将 In message 标头复制到 Out message 标头，并将 Out message body 设置为指定的值。

Java DSL	XML DSL	描述
unmarshal()	**unmarshal**	使用指定的数据格式，将 In 消息正文从低级或二进制格式转换为高级别的格式。
validate （）	**validate**	取一个 predicate 表达式来测试当前消息是否有效。如果 predicate 返回 **false**，则抛出 **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 wire tap URI。

一些处理器示例

要了解如何在路由中使用处理器，请参阅以下示例：

- ***choice***
- ***Filter***
- ***Throttler***
- ***Custom***

choice

choice （） 处理器是一个条件语句，用于将传入消息路由到替代制作者端点。每个替代制作者端点都以 when （） 方法开头，此方法采用 predicate 参数。如果 predicate 为 true，则会选择以下目标，否则处理将继续执行规则中的下一个 when （） 方法。例如，以下 choice （） 处理器将传入消息定向到 Target1、 Target2 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");
```

-

或者等效在 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，有一个特殊情况，您可能需要使用 endChoice （） 命令。某些标准的 Apache Camel 处理器允许您使用特殊的子层来指定额外的参数，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句指定为 loadBalance （）.roundRobin （）.to (" mock:foo ").to ("mock:bar").end （），它会在 mock:foo 和 mock:bar 端点之间平衡消息。如果负载均衡器子句嵌入在选择条件中，需要使用 endChoice （） 命令来终止子句，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常见的消息到达制作者端点。它接受单个 predicate 参数：如果 predicate 为 true，则允许消息交换到制作者；如果 predicate 为 false，则消息交换会被阻断。例如，以下过滤器会阻止消息交换，除非传入消息包含一个标题 foo，值为 bar:

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

或者等效在 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 的工作原理为：限制每秒可以传递的消息数量。如果传入的消息超过指定的速率，throttler 会在缓冲区中累积超出消息，并将其传送到制作者端点。例如，要将吞吐量限制为每秒 100 个信息，您可以定义以下规则：

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

或者等效在 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

如果这里描述的标准处理器都没有提供您需要的功能，则始终可以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 会从传入消息中删除名为 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，该方法为将处理器插入规则提供了通用机制。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 路由构建的基本原则

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的原则。

2.1. PIPELINE 处理

概述

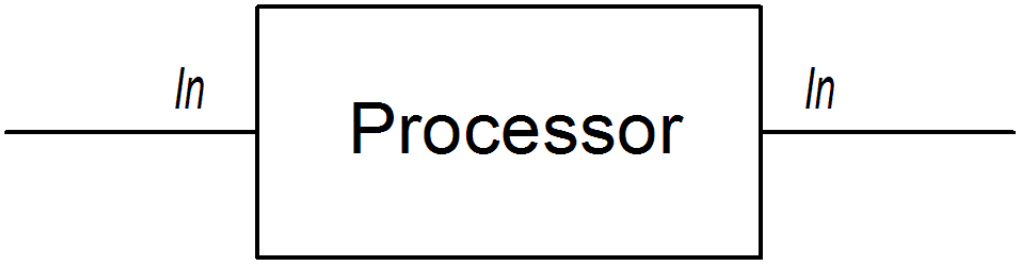
在 Apache Camel 中，pipelining 是在路由定义中连接节点的主要范例。管道概念或许对 UNIX 操作系统的用户最熟悉，其中用于加入操作系统命令。例如，ls | more 是将目录列表 ls 传送到 page-scrolling 实用程序的命令示例，更多。管道的基本理念是，其中一个 命令的输出 被引入到下一个命令 的输入 中。路由时的自然模拟是，从一个处理器中要复制到下一处理器的 In 消息中的 Out 消息。

处理器节点

路由中的每个节点（初始端点除外）都是 处理器，它们从 org.apache.camel.Processor 接口继承。换句话说，处理器制作 DSL 路由的基本构建块。例如，filter （），delayer （），setBody （），setHeader （），和 to （） 等 DSL 命令代表处理器。当考虑处理器如何连接在一起以构建路由时，务必要区分两种不同的处理方法。

第一种方法是处理器只修改交换的 In 消息，如 图 2.1 “处理器修改消息” 所示。在这种情况下，交换的 Out 消息保持 null。

图 2.1. 处理器修改消息

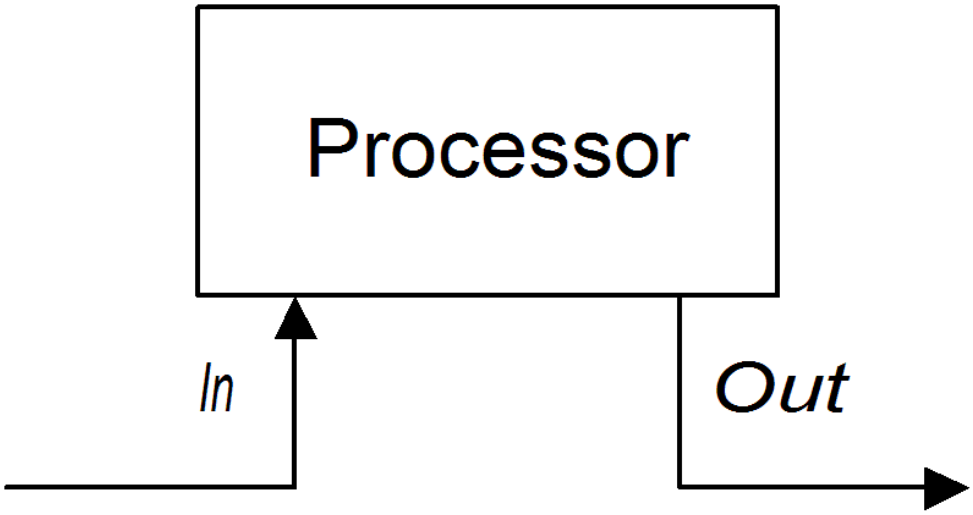


以下路由显示了一个 setHeader （） 命令，它通过添加（或修改） BillingSystem 标题来修改当前的 In 消息：

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

第二种方法是，处理器会创建一个 Out 消息来代表处理的结果，如 图 2.2 “处理器创建注销消息” 所示。

图 2.2. 处理器创建注销消息



以下路由显示了一个 transform （） 命令，该命令使用包含字符串 DummyBody 的消息正文创建 Out 消息：

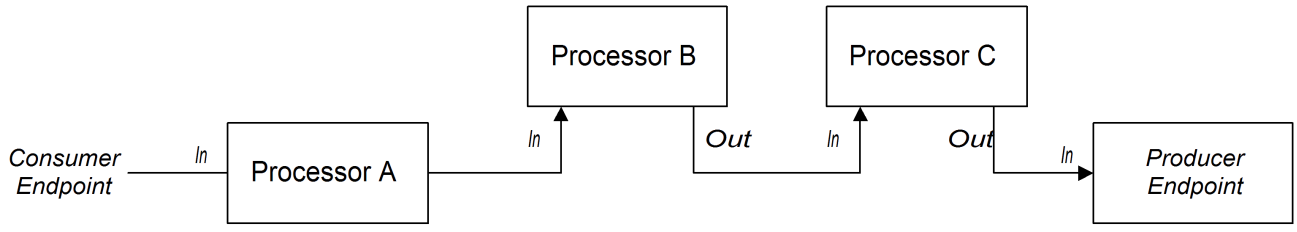
```
from("activemq:orderQueue")
    .transform(constant("DummyBody"))
    .to("activemq:billingQueue");
```

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

InOnly Exchanges 的管道

图 2.3 “InOnly Exchanges 的管道示例” 显示了用于 InOnly 交换的处理器管道示例。处理器 A 通过修改 In 消息来的行为，而处理器 B 和 C 则创建 Out 消息。路由构建器将处理器连接在一起，如下所示。特别是，处理器 B 和 C 以 管道 的形式链接在一起：也就是说，处理器 B 的 Out 消息会在将交换发送到处理器 C 之前移到 In 消息，处理器 C's Out 消息会移到 In 消息，然后再将交换进入生成者端点。因此，处理器的输出和输入加入到持续管道中，如 图 2.3 “InOnly Exchanges 的管道示例” 所示。

图 2.3. InOnly Exchanges 的管道示例



Apache Camel 默认使用管道模式，因此您不需要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 模板传输消息（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 ：

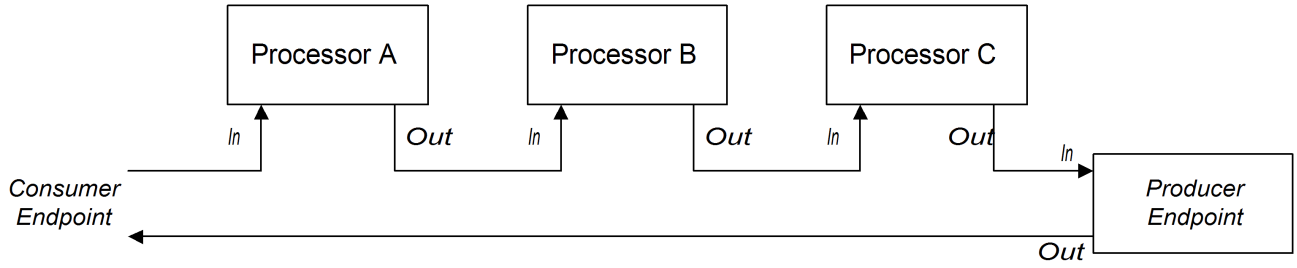
```
from("activemq:userdataQueue")
    .to(ExchangePattern.InOut, "velocity:file:AdressTemplate.vm")
    .to("activemq:envelopeAddresses");
```

其中 Velocity 端点 velocity:file:AddressTemplate.vm 指定 Velocity 模板文件 file:AddressTemplate.vm 的位置。to （） 命令将交换模式更改为 InOut，然后将交换模式更改为 Velocity 端点，然后将它改回到 InOnly。有关 Velocity 端点的更多详细信息，请参阅 Apache Camel 组件参考指南 中的 Velocity。

InOut Exchanges 的管道

图 2.4 “InOut Exchange 的管道示例” 显示 InOut Exchanges 的处理器管道示例，您通常使用它来支持远程过程调用(RPC)语义。处理器 A、B 和 C 以管道的形式连接在一起，以及下一个处理器的输入信息。producer 端点生成的最终 Out 消息将以所有方式发回给消费者端点，其中提供了对原始请求的回复。

图 2.4. InOut Exchange 的管道示例



请注意，为了支持 InOut Exchange 模式，路由（无论是生成者端点还是某种其他类型的处理器）中的最后一个节点都会创建 Out 消息。否则，任何连接到消费者端点的客户端都会挂起并等待回复消息。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，请考虑以下路由来处理支付请求：

```
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是通过 Web 服务的管道处理，cxf:bean:addAccountDetails, cxf:cxf:bean:getCreditRating, 和 cxf:bean:processTransaction。最后的 Web 服务处理 事务 生成通

过 JETTY 端点发回的响应(出 消息)。

当管道仅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");
```

InOptionalOut Exchanges 的管道

InOptionalOut Exchanges 的管道与 图 2.4 “InOut Exchange 的管道示例” 中的管道基本相同。InOut 和 InOptionalOut 之间的区别在于，使用 InOptionalOut 交换模式的交换允许将 null Out 消息作为回复。也就是说，如果 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 InOut Exchange，将丢弃 nullOut 消息，并且来自当前节点的原始 In 消息会被复制到下一节点的 In 消息。

2.2. 多个输入

概述

标准路由使用来自单个端点的输入，使用 Java DSL 中的 from (EndpointURL) 语法。但是，如果您需要为路由定义多个输入，该怎么办？Apache Camel 提供了几个替代方案来为路由指定多个输入。进行的方法取决于您希望交换相互独立处理，还是希望以某种方式合并不同输入的交换（在这种情况下，您应该使用 “内容增强模式”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");
```

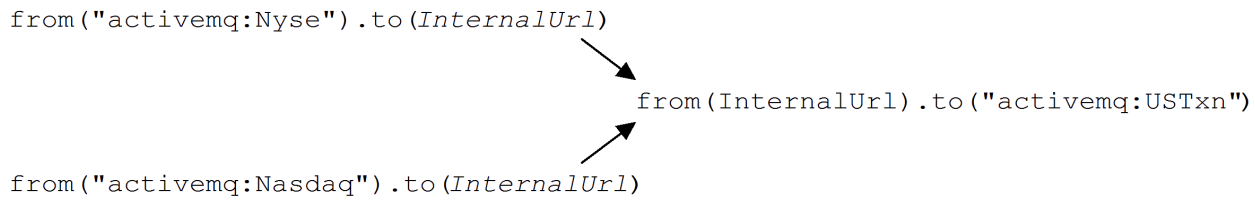
在这两个示例中，从每个输入端点、URI1、URI 2 和 URI3 交换都是相互独立处理的，在单独的线程中。实际上，您可以将上述路由视为等同于以下三个独立的路由：

```
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传入的消息，并使用同一路由处理它们。在大多数情况下，您可以通过将路由划分为网段来处理多个输入，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



路由的初始片段取来自一些外部队列的 inputs （例如： activemq:Nyse 和 activemq:Nasdaq HEKETI-wagon），并将传入的交换发送到内部端点 InternalUrl。第二个路由片段合并了传入的交换，将其从内部端点中获取，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*
- *VM 端点*

这些端点的主要目的是使您能够将不同的部分绑定到路由的不同部分。它们都提供了一种将多个输入合并为一个路由的有效方法。

直接端点

直接组件提供将连接在一起的路由的最简单机制。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个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 EndpointID 只是用于标识端点实例的唯一字母数字字符字符串。

例如，如果要从两个消息队列(activemq:Nyse 和 activemq:Nasdaq)获取输入，并将它们合并到单

个消息队列 activemq:USTxn 中，您可以通过定义以下一组路由来完成此操作：

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

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

其中前两个路由从消息队列(Nyse 和 Nasdaq)获取输入，并将它们发送到端点，即 direct:mergeTxns。最后队列组合了来自前两个队列的输入，并将组合消息流发送到 activemq:USTxn 队列。

直接端点的实现的行为如下：每当交换到达生成者端点时（例如，to ("direct:mergeTxns")），直接端点将交换直接传递给具有相同端点 ID （例如，from ("direct:mergeTxns")）的所有消费者端点。直接端点只能用于在属于同一 Java 虚拟机(JVM)实例中同一 CamelContext 的路由之间进行通信。

SEDA 端点

SEDA 组件提供了链接在一起路由的备选机制。您可以像直接组件一样使用它，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 不是 同步的。也就是说，当您将交换发送到 SEDA 生成端点时，控制会立即返回到路由中的前一个处理器。
- SEDA 端点包含一个队列缓冲区(java.util.concurrent.BlockingQueue 类型)，它在由下一个路由段处理之前存储所有传入交换。
- 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 竞争消费者 模式，这样可保证仅处理一次传入的交换，即使将多个消费者附加到特定端点。

使用 SEDA 端点的一个主要优点是路由可以更快响应，并影响内置消费者线程池。库存事务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");
```

这个示例和直接示例的主要区别在于，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 比简单地将路由段粘贴到一起。分阶段的事件驱动架构(SEDA)包括一个设计原则，用于构建更易于管理的多线程应用程序。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/。

VM 端点

虚拟机组件与 SEDA 端点非常相似。唯一的区别是，当 SEDA 组件从同一 CamelContext 内链接在一起的路由片段时，虚拟机组件可让您将不同 Apache Camel 应用程序的路由连接在一起，只要它们在同一个 Java 虚拟机中运行。

库存事务示例可以重新编写为使用虚拟机端点而不是 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中（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");
```

内容增强模式

内容增强模式定义了处理对路由的多个输入的根本不同方法。当交换进入增强处理器时，增强器会联系外部资源以检索信息，然后添加到原始消息中。在这种模式中，外部资源有效地代表对消息的第二个输入。

例如，假设您正在编写处理信用请求的应用。在处理信用请求之前，您需要使用为客户分配信用评级的数据对其进行补充，其中评级数据存储在目录 src/data/ratings 的文件中。您可以使用 pollEnrich（）模式和 GroupedExchangeAggregationStrategy 聚合策略将传入的信用卡请求与 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 类是来自 org.apache.camel.processor.aggregate 软件包的标准聚合策略，将每个新交换添加到 java.util.List 实例，并将生成的列表存储在 Exchange.GROUPED_EXCHANGE Exchange 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（从文件端点）。

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
        ...
    }
}
```

此应用的替代方法是将合并代码直接放入自定义聚合策略类的实现中。

有关内容增强模式的详情，请参考 第 10.1 节 “内容增强”。

2.3. 异常处理

摘要

Apache Camel 提供了几种不同的机制，它们可让您以不同的粒度处理异常：您可以使用 doTry、doCatch 和 doFinally；或者，您可以指定每个异常类型采取什么操作，并使用 onException 将其此规则应用到 RouteBuilder 中的所有路由；或者，您可以指定 执行所有 异常类型的操作，并使用 Handler 错误处理程序中的所有路由。

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捕获异常的强大机制：它特定于类型，允许您定义不同的操作来处理不同的异常类型；它允许您使用基本上（实际上，稍微扩展的）语法定义为路由，以便您以处理异常的方式进行大量操作。

使用 onException 捕获异常

onException 子句是 trapping 的机制，而不是捕获异常。也就是说，一旦定义了 onException 子句，它会捕获路由中任何时间点上出现的异常。这与 Java try/catch 机制（其中异常被发现）相反，只有在特定代码片段 被明确 包含在尝试块中时。

当您定义 onException 子句时，Apache Camel 运行时隐式将每个路由节点包含在尝试块中时会出现什么情况。这就是为什么 onException 子句可以在路由中的任何点上捕获异常。但会自动为您完成这一换行；它在路由定义中不可见。

Java DSL 示例

在以下 Java DSL 示例中，onException 子句应用到 RouteBuilder 类中定义的所有路由。如果在处理任一路由(from ("seda:inputA") 或 from ("seda:inputB")时发生 ValidationException 异常，则 onException 子句会捕获异常并将当前交换重定向到 validationFailed JMS 队列（充当死信队列）。

```
// 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 中表达，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>
```

陷阱多个例外

您可以在 RouteBuilder 范围内定义多个 onException 子句来陷阱异常。这可让您采取不同的操作来响应不同的异常。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 deadletter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 中定义同一一系列 onException 子句，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>
```

您还可以将多个例外分组在一起，以便由同一 onException 子句捕获。在 Java DSL 中，您可以按如

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 例外元素来对多个例外 进行分组，如下所示：

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

当捕获多个异常时，onException 子句的顺序非常重要。Apache Camel 最初会尝试将抛出的异常与第一个 子句匹配。如果第一个子句无法匹配，则尝试下一个 onException 子句，直到找到匹配项为止。每个匹配尝试都受到以下算法的管控：

1. 如果抛出的异常是 串联的异常 （即，异常被误认为是不同的例外），则最嵌套的异常类型最初是匹配的基础。这个例外被测试如下：
   a. 如果 exception-to-test 具有在 onException 子句中指定的类型（使用 instanceof测试），则会触发匹配项。
   b. 如果 exception-to-test 是 onException 子句中指定的类型的子类型，则会触发匹配项。
2. 如果最嵌套的异常无法产生匹配项，则改为测试链（嵌套异常）中的下一个异常。测试继续进行链，直到触发了匹配项或链已耗尽为止。

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化，例如：

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

Deadletter 频道

目前，onException 使用的基本示例都利用了 deadletter 频道 模式。也就是说，当一个 onException 子句捕获异常时，当前的交换会被路由到一个特殊的目的地(deadletter 频道)。deadletter 频道充当 尚未 处理的失败消息的冻结区域。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，请参考 第 6.3 节 “死信频道”。

使用原始消息

在路由中间出现异常时，交换中的消息可能会大大修改（即使人可读也是如此）。通常，如果死信队列中的消息是 原始消息，则管理员更容易决定要采取的纠正性操作。默认情况下，useOriginalMessage 选项为 false，但如果它在错误处理程序上配置了，则会自动启用。

注意

当应用到将消息发送到多个端点的 Camel 路由或将消息分成部分时，useOriginalMessage 选项可能会导致意外行为。原始消息可能无法在 Multicast、Splitter 或 RecipientList 路由中保留，在其中修改原始消息。

在 Java DSL 中，您可以将交换中的消息替换为原始消息。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，您可以通过对 onException 元素设置 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 setAllowUseOriginalMessage （） 选项设为 true，Camel 会在路由开始时生成原始消息的副本，这样可确保在您调用 useOriginalMessage （） 时原始消息可用。但是，如果 Camel 上下文上的 setAllowUseOriginalMessage （） 选项被设置为 false（这是默认设置），则无法访问原始消息，您无法调用 useOriginalMessage （）。

利用默认行为的原因是在处理大量消息时优化性能。

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

重新发送策略

Apache Camel 不会中断消息的处理，并在引发异常时立即放弃，而是为您提供尝试在异常发生时对消息进行 恢复 的选项。在联网系统中，可能会出现超时和临时故障，如果失败消息在原始异常被引发后不久，则通常有可能成功处理失败的消息。

Apache Camel 重新发送支持在异常发生后对消息进行各种策略。配置重新发送的一些最重要的选项如下：

maximumRedeliveries （）

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

retryWhile()

指定一个 predicate (Predicate 类型），它决定 Apache Camel 是否要继续重新设计。如果 predicate 在当前交换上评估为 true，则会尝试重新发送；否则，重新发送将停止，且不会进行进一步重新发送尝试。

这个选项优先于 maximumRedeliveries （） 选项。

在 Java DSL 中，重新传送策略选项使用 onException 子句中的 DSL 命令来指定。例如，您可以指定最多 6 个 redeliveries，交换发送到 validationFailed deadletter 队列，如下所示：

```
onException(ValidationException.class)
  .maximumRedeliveries(6)
  .retryAttemptedLogLevel(org.apache.camel.LogginLevel.WARN)
  .to("activemq:validationFailed");
```

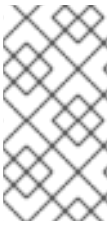
在 XML DSL 中，通过设置 redeliveryPolicy 元素的属性来指定重新发送策略选项。例如，前面的路由可以在 XML DSL 中表达，如下所示：

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

在重新传送选项后，路由是 route swig-wagonis 的后一部分会被处理，直到最后一次重新传送尝试失败后才会处理。有关所有重新传送选项的详细描述，请参阅 第 6.3 节 “死信频道”。

另外，您可以在 redeliveryPolicyProfile 实例中指定 redelivery 策略选项。然后，您可以使用 onException 元素的 redeliverPolicyRef 属性引用 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

如果要在多个 onException 子句中重复使用相同的重新发送策略，使用 redeliveryPolicyProfile 的方法很有用。

条件 trapping

通过指定 onWhen 选项，可以通过指定 onWhen 选项进行带有 onException 的异常捕获。如果您在 onException 子句中指定 onWhen 选项，则只有在抛出异常与 子句匹配时，才会触发匹配项，而 onWhen predicate 会在当前交换上评估为 true。

例如，在以下 Java DSL 片段中，第一个 onException 子句会触发，只有在抛出异常与 MyUserException 匹配 并且用户 标头在当前交换中不匹配时才触发：

```
// 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);
```

前面的 onException 子句可以在 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，当路由中间出现异常时，当前交换的处理会中断，而抛出的异常会在路由开始时传播到消费者端点。触发 onException 子句时，其行为基本上相同，但 onException 子句在抛出异常被传播到前执行一些处理。

但是，这种默认行为 不是 处理异常的唯一方法。onException 提供各种选项来修改异常处理行为，如下所示：

- 抑制异常重新 增长，您可以选择在 onException 子句完成后阻止再箭头异常。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 wagon- swig 您可以选择从最初发生异常时恢复对交换的正常处理。隐式来说，这种方法也会阻止增长异常。

- 发送消息时，会发送一个特殊情况，其中路由开始时的消费者端点需要回复（即 InOut MEP），您可能希望构建自定义故障回复消息，而不是将异常传播到消费者端点。

阻止异常增长

要防止当前异常被重新处理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（如果它评估为非null 值，则任何非布尔值表达式被解释为 true ）。

可以使用 处理 的元素，将相同的路由配置为阻止 XML DSL 中的再增长异常，如下所示：

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

继续处理

要继续处理最初抛出异常的路由中的当前消息，您可以在 Java DSL 中将 continue 选项设置为 true，如下所示：

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

在 Java DSL 中，继续 （） 选项的参数可以是布尔值类型、Predicate 类型或 Expression 类型（如果它评估为非null 值，则任何非布尔值表达式被解释为 true ）。

可以使用 continued 元素在 XML DSL 中配置相同的路由，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，您可能需要构建自定义故障回复消息，而不是只是让抛出异常传播到消费者。在这种情况下，需要遵循两个基本步骤：使用 **handled** 选项阻止再增长异常；并使用 **handled** 选项填充交换的 **Out** 消息插槽。

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错误响应，您通常希望在响应中包含异常消息的文本。您可以使用 **exceptionMessage** （） 构建器方法访问当前异常消息的文本。例如，您可以发送一个回复，其中包含发生 **MyFunctionalException** 异常时的异常消息的文本，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 the exception
message
onException(MyFunctionalException.class)
  .handled(true)
  .transform(exceptionMessage());
```

例外消息文本也可以通过 **exception.message** 变量从 **Simple** 语言访问。例如，您可以在回复消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.");
```

前面的 **onException** 子句可以在 **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>
```

在处理异常时抛出异常

在处理现有异常时抛出的异常（换句话说，在处理一个 onException 子句的过程中会抛出一个异常）。此类例外由特殊的回退异常处理器处理，它处理异常，如下所示：

- *所有现有异常处理程序将被忽略，处理会立即失败。*
- *记录新异常。*
- *在交换对象上设置新的例外。*

简单的策略避免了复杂的故障场景，它们可能最终出现 onException 子句被锁定到无限循环中。

范围

onException 子句可以在以下任何一个范围内有效：

- *RouteBuilder 范围 wagon- onException 子句在 RouteBuilder.configure （） 方法内定义为 standalone 语句，会影响 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。在路由定义之前，必须 出现 onException 子句。*

  到此点的所有示例都使用 RouteBuilder 范围来定义。

- *路由范围 wagon- onException 子句也可以直接嵌入路由中。这些 onException 子句 仅影响 在其中定义的路由。*

路由范围

您可以在路由定义内任何位置嵌入一个 **onException** 子句，但您必须使用 **end （） DSL** 命令终止嵌入的 **onException** 子句。

例如，您可以在 **Java DSL** 中定义嵌入的 **onException** 子句，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** 中定义内嵌的 **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 （） 子句提供与 onException 子句类似的功能，但这种机制 无法在 不同的异常类型之间差异。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 RouteBuilder 类中的所有路由。每当其中一个适用的路由 中都发生异常 时，会触发它。例如，要定义一个错误处理程序，将所有失败的交换

路由到 ActiveMQ deadLetter 队列，您可以定义一个 RouteBuilder，如下所示：

```
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");
        // ...
    }
}
```

但是，在重新发送的所有尝试都已耗尽之前，不会发生重定向到死信频道。

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，要定义一个错误处理程序，将所有失败的交换路由到 ActiveMQ deadLetter 队列，您可以定义 errorHandler 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.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>
```

错误处理程序的类型

表 2.1 “错误处理程序类型” 提供对您可以定义的不同类型的错误处理程序的概述。

表 2.1. 错误处理程序类型

Java DSL Builder	XML DSL 类型属性	描述
defaultErrorHandler()	**DefaultErrorHandler**	将异常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，并且还支持死信队列。
loggingErrorChannel()	**LoggingErrorChannel**	每当发生异常时，记录异常文本。
noErrorHandler()	**NoErrorHandler**	可以用来禁用错误处理程序的 dummy 处理程序实现。
	TransactionErrorHandler	转换路由的错误处理程序。默认事务错误处理程序实例会自动用于标记为 transacted 的路由。

2.3.3. doTry, doCatch, and doFinally

概述

要在路由内处理异常，您可以使用 doTry、doCatch 和 doFinally 子句的组合，它们处理异常与 Java 的 尝试、捕获和 finally 块类似。

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 swig-您可在单个 doTry 块中有多个 doCatch 子句。doCatch 子句按照其出现的顺序进行测试，就像 Java catch （） 语句一样。Apache Camel 执行与抛出异常匹配的第一个 doCatch 子句。*

注意

这个算法与 onException 子句的 exception 匹配算法不同，详情请参阅 第 2.3.1 节 “onException Clause”。

- 使用构造来重新增长异常 wagon- swig 您可以从 doCatch 子句中重新调整当前的异常（请参阅 “doCatch 中的重新增长异常”一节）。

doCatch 的特殊特性

但是，doCatch （） 子句有一些特殊功能，但 Java catch （） 语句中没有模拟。以下功能特定于 doCatch （） ：

- 通过向 doCatch 子句附加一个 on Catch 子句（请参阅 “条件异常使用 onWhen”一节）来捕获异常的条件。

Example

以下示例演示了如何在 Java DSL 中编写 doTry 块，其中执行 doCatch （） 子句，如果出现 IOException 异常或 IllegalStateException 异常，并且 始终执行 doFinally （） 子句，无论是否引发异常。

```
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：

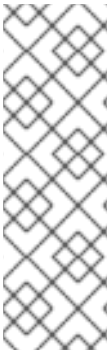
```
<route>
    <from uri="direct:start"/>
    <!-- here the try starts. its a try .. catch .. finally just as regular java code -->
    <doTry>
        <process ref="processorFail"/>
        <to uri="mock:result"/>
        <doCatch>
            <!-- catch multiple exceptions -->
            <exception>java.io.IOException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

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

doCatch 中的重新增长异常

使用 structs 可以在 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

您还可以使用处理器而不是 handled (false) （在 doTry/doCatch 子句中弃用的处理器）重新增长异常：

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

在前面的示例中，如果 doCatch （） 出现 IOException，当前的交换将发送到 mock:io 端点，则 IOException 为 rerown。这会在路由开始时（在 from （） 命令中）提供消费者端点，可以有机会处理异常。

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>
```

条件异常使用 onWhen

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条件化异常的捕获。换句话说，如果您使用表单的 子句来捕获异常，则 doCatch (ExceptionList).doWhen (Expression) 只会发现异常，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)，只有在异常消息包含单词 Severe 时：

```
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>
```

doTry 中的嵌套条件

有多种选项可用于将 Camel 异常处理添加到 JavaDSL 路由中。dotry （） 创建用于处理异常的尝试或捕获块，对路由特定错误处理很有用。

如果要捕获 ChoiceDefinition 内异常，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 Exceptions

概述

Camel CXF 组件提供与 Apache CXF 的集成，使您能够从 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，然后使用端点的 bean ID 在路由中引用该端点。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，以便在服务器端引发 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。要启用这种情况，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>
```

为安全起见，堆栈跟踪不包括导致异常（即，由 导致的堆栈追踪的一部分）。如果要在堆栈追踪中包含导致异常，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 标志用于测试和诊断。对于服务器而言，正常做法是让恶意用户探测到服务器的最初原因。

2.4. BEAN 集成

概述

Bean 集成提供了一种通用机制，用于处理使用任意 Java 对象的消息。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，然后该对象可以访问和修改传入的交换。将交换内容映射到参

数的机制，而 bean 方法的返回值称为 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 wagon-wagon 如果方法签名符合某些约定，则参数绑定可以使用 Java 反映来决定要传递的参数。*
- *注解和依赖项注入 HEKETI-wagon 用于更灵活的绑定机制，使用 Java 注解来指定注入方法参数的内容。此依赖项注入机制依赖于 Spring 2.5 组件扫描。通常，如果您要将 Apache Camel 应用程序部署到 Spring 容器中，依赖项注入机制将自动工作。*
- *在调用 bean 的点上，您可以明确指定参数（作为常量或使用 Simple 语言）明确指定参数。*

Bean registry

Bean 通过 bean 注册表 访问，它是一个服务，可让您使用类名称或 bean ID 作为键查找 Bean。在 bean registry 中创建条目的方式取决于底层的框架是是 plain Java、Spring、Guice 或 Blueprint。通常会创建 registry 条目（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 registry 实施插件策略，为访问 Bean 定义集成层，从而使底层 registry 实施透明。因此，可以将 Apache Camel 应用程序与各种不同的 bean registry 集成，如 表 2.2 “registry 插件” 所示。

表 2.2. registry 插件

registry 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，因为为您自动安装相关的 bean registry。例如，如果您使用 Spring 框架来定义路由，则 Spring ApplicationContextRegistry 插件会在当前 CamelContext 实例中

自动安装。

在 OSGi 容器中部署是一个特殊情况。当 Apache Camel 路由部署到 OSGi 容器中时，CamelContext 会自动设置用于解析 Bean 实例的注册表链：注册表链由 OSGi 注册表组成，后跟 Blueprint （或 Spring）注册表。

访问 Java 中创建的 bean

要使用 Java Bean （普通的旧 Java 对象或 POJO ）处理交换对象，请使用 bean （） 处理器，它将入站交换绑定到 Java 对象上的方法。例如，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。如果您只想从单一路由访问 MyBeanProcessor 实例，此方法就足够了。但是，如果要从多个路由访问同一 MyBeanProcessor 实例，请使用将对象类型作为其第一个参数的 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");
```

访问超载的 bean 方法

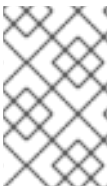
如果 bean 定义过载方法，您可以通过指定方法名称及其参数类型来选择要调用的过载方法。例如，如果 MyBeanBrocessor 类有两个过载方法，processBody (String) 和 processBody (String,String)，您可以调用后者的超载方法，如下所示：

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

或者，如果要根据所采用的参数数量识别方法，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，与参数的确切类型无关，请按如下所示调用 bean （） 处理器：

```
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名称，如 processBody (Exchange)- 或完全限定类型 name- 例如 processBody (org.apache.camel.Exchange)。



注意

在当前实现中，指定的类型名称必须与参数类型完全匹配。类型继承不会考虑。

明确指定参数

当您调用 bean 方法时，您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中的类型指定符混合起来：

```
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，第一个参数的值会假定由参数绑定注解决定（请参阅 “基本注解”一节）。

除了 simple 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，可以使用 Simple 语言的完整功能。例如，将消息正文和 title 标头的值传递给 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哈希映射作为参数传递。例如，在以下示例中，必须将第二个 **method** 参数声明为 **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约定：

- 处理消息正文的方法签名
- 用于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消息正文的 **bean** 方法，您必须定义一个采用单个 **String** 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;
    }
}
```

用于处理交换的方法签名

要获得更大的灵活性，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，方法签名采用单个 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 Spring bean

您可以使用 Spring XML 创建实例，而不是在 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 Spring bean 元素。以下示例演示了如何创建 MyBeanProcessor 实例：

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

也可以使用 Spring 语法将数据传递给 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。

当您使用 bean 元素创建对象实例时，您可以稍后使用 bean 的 ID (bean 元素的 id 属性的值)来引用它。例如，如果 ID 等于 myBeanId 的 bean 元素，您可以使用 beanRef () 处理器在 Java DSL 路由中引用 bean，如下所示：

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

其中 beanRef () 处理器调用指定 bean 实例的 MyBeanProcessor.processBody () 方法。

您还可以使用 Camel 模式的 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>
```

对于 slight 效率，您可以将 cache 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 Spring bean

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 ID 等于 myBeanId 的 bean 元素，您可以使用 beanRef （） 处理器在 Java DSL 路由中引用 bean，如下所示：

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

或者，您可以使用 @BeanInject 注释来引用 Spring bean，如下所示：

```
// 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 根据类型查找 registry，但这仅在给定类型的单个 Bean 时才有效。例如，要查找并注入 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，后跟；
2. 关闭使用的 Bean。

如果此关闭顺序相反，则可能会发生 Camel 上下文试图访问已销毁的 Bean （导致错误直接进入错误）；或者 Camel 上下文尝试在销毁时创建缺少的 bean，这也会导致错误。Spring XML 中的默认关闭顺序取决于 Bean 和 camelContext 出现在 Spring XML 文件中。为了避免因为关闭顺序不正确而随机错误，camelContext 被配置为在 Spring XML 文件中任何其他 Bean 之前关闭。这是自 Apache Camel 2.13.0 起的行为。

如果您需要更改此行为（因此，在其他 Bean 之前，Camel 上下文 不会被 强制关闭），您可以将 camelContext 元素上的 shutdownEager 属性设置为 false。在这种情况下，您可能会使用 Spring dependent-on 属性对关闭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，如果您有一个执行某些数据操作的传统 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。对于这种参数绑定，Apache Camel 提供了以下 Java 注解类型：

- 基本注解
- 语言注解
- 继承注解

基本注解

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

表 2.3. 基本 Bean 注解

注解	含义	参数？
@attachments	绑定到附加列表。	

注解	含义	参数？
@Body	绑定到入站消息正文。	
@Header	绑定到入站邮件标头。	标头的字符串名称。
@headers	绑定到入站消息标头的 **java.util.Map**。	
@OutHeaders	绑定到出站消息标头的 **java.util.Map**。	
@Property	绑定到命名的 Exchange 属性。	属性的字符串名称。
@Properties	绑定到交换属性的 **java.util.Map**。	

例如，以下类演示了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，如何将注解与默认约定混合在一起。除了注入注解的参数外，参数绑定也会自动将交换对象注入 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 方法的参数中。通过使用这些注解，您可以调用使用您选择的脚本语言编写的任意脚本，从入站交换中提取数据，并将数据注入方法参数。表 2.4 “表达式语言注解” 显示 org.apache.camel.language 软件包（和子软件包）中的注释，您可以使用它们将消息数据注入 bean 方法的参数。

表 2.4. 表达式语言注解

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	注入标头表达式。
@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，以下类演示了如何使用 @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 注释是一个特殊情况，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 注解来调用 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，您也可以在引用的 bean 类 MyIdGenerator 中使用注解。对 generate （） 方法签名的唯一限制是，它必须返回正确的类型，才能注入 @Bean 标注的参数。由于 @Bean 注释不会让您指定方法名称，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第一个方法。

注意

某些语言注释在核心组件中可用(@Bean、@Constant、@Simple、@XPath)。但是，对于非核心组件，您必须确保载入相关组件。例如，要使用 OGNL 脚本，您必须加载 camel-ognl 组件。

继承注解

参数绑定注解可以从接口或超级类继承。例如，如果您使用 Header 注解和 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
    );
}
```

实施类 MyBeanProcessor 中定义的过载方法现在继承基本接口中定义的注解，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

实施 Java 接口的类通常 受到保护、私有 或仅软件包 范围。如果您尝试以这种方式对限制的实施类调用方法，则 bean 绑定会返回调用对应的接口方法，后者可以公开访问。

例如，请考虑以下 public BeanIntf 接口：

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

其中 **BeanIntf** 接口由以下受保护的 **BeanIntfImpl** 类实现：

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

以下 **bean** 调用会返回调用公共 **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 集成具有在不创建关联类实例的情况下调用静态方法的功能。例如，请考虑以下 **Java** 类，用于定义静态方法 **changeSomething （） ：**

```
// 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 更改**Something** 方法，如下所示：

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

请注意，虽然这种语法与调用普通函数相同，但 bean 集成利用 Java 反映将方法识别为静态，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，当路由部署到红帽 Fuse 容器中时，可以使用 bean 集成直接调用 OSGi 服务。例如，假设 OSGi 容器中之一已导出了服务 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"/>
```

其工作方式是，当 Apache Camel 在 OSGi 容器中部署时，Apache Camel 设置注册表链。首先，它会在 OSGi 服务注册表中查找指定的类名称；如果查询失败，它将返回到本地 Spring DM 或蓝图注册表。

2.5. 创建交换实例

概述

使用 Java 代码处理消息（例如，在 bean 类或在处理器类中）时，通常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 会公开静态方法 anExchange，您可以使用它来开始构建交换对象。

Example

例如，以下代码会创建一个包含消息正文字符串 Hello World! 的新交换对象，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 anExchange (CamelContext 上下文)

（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 withHeader (String key, Object value)

在交换上设置标头（即，在交换的 In 消息上设置标头）。

ExchangeBuilder withPattern (ExchangePattern pattern)

在交换上设置交换模式。

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 支持各种转换消息内容的方法。除了用于修改消息内容的简单原生 API 外，Apache Camel 还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，可让您对传入和传出消息执行简单的转换。例如，例 2.1 “Incoming 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. Incoming 消息的简单转换

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

其中 setBody （） 命令取代了传入消息正文的内容。

用于简单转换的 API

您可以使用以下 API 类在路由器规则中对消息内容执行简单的转换：

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

ProcessorDefinition 类

org.apache.camel.model.ProcessorDefinition 类定义了 DSL 命令，您可以直接插入到路由器规则 swig-wagon 中，例如： 中的 setBody （） 命令。表 2.5 “来自 ProcessorDefinition 类的转换方法” 显示与转换消息内容相关的 ProcessorDefinition 方法：

表 2.5. 来自 ProcessorDefinition 类的转换方法

方法	描述
type convertBodyTo (Class type)	将 IN 消息正文转换为指定的类型。
type removeFaultHeader (String name)	添加处理器，它将删除 FAULT 消息上的标头。
type removeHeader (String name)	添加处理器，它将删除 IN 消息上的标头。
type removeProperty (String name)	添加删除交换属性的处理器。
ExpressionClause<ProcessorDefinition<Type>> setBody （）	添加处理器，该处理器在 IN 消息上设置正文。
type setFaultBody (Expression 表达式)	添加处理器，在 FAULT 消息上设置正文。
type setFaultHeader (String name, Expression expression)	添加处理器，在 FAULT 消息上设置标头。
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)	添加处理器，该处理器在 OUT 消息上设置标头。
type setOutHeader (String name, Expression expression)	添加处理器，该处理器在 OUT 消息上设置标头。
ExpressionClause<ProcessorDefinition<Type>> setProperty (String name)	添加设置交换属性的处理器。
type setProperty (String name, Expression expression)	添加设置交换属性的处理器。
ExpressionClause<ProcessorDefinition<Type>> transform （）	添加处理器，该处理器在 OUT 消息上设置正文。
type transform (Expression expression)	添加处理器，该处理器在 OUT 消息上设置正文。

builder 类

org.apache.camel.builder.Builder 类提供在预期表达式或 predicates 的上下文中访问消息内容。换句话说，构建程序 方法通常在 DSL 命令的参数 中调用，例如： ***例 2.1 “Incoming 消息的简单转换”*** ***中的***

body （） 命令。表 2.6 “Builder 类中的方法” 总结了 Builder 类中提供的静态方法。

表 2.6. Builder 类中的方法

方法	描述
静态 <E extends Exchange> ValueBuilder<E> body （）	为交换上的入站正文返回 predicate 和 value 构建器。
静态 <E extends Exchange,T> ValueBuilder<E> bodyAs (Class<T> 类型)	返回作为特定类型的入站消息正文的 predicate 和 value 构建器。
静态 <E extends Exchange> ValueBuilder<E> constant (Object value)	返回恒定表达式。
静态 <E extends Exchange> ValueBuilder<E> faultBody （）	返回交换上错误正文的 predicate 和 value 构建器。
静态 <E extends Exchange,T> ValueBuilder<E> faultBodyAs (Class<T> type)	返回作为特定类型的 fault 消息正文的 predicate 和 value 构建器。
静态 <E extends Exchange> ValueBuilder<E> header (String name)	为交换上的标头返回 predicate 和 value builder。
静态 <E extends Exchange> ValueBuilder<E> outBody （）	为交换上的出站正文返回 predicate 和 value 构建器。
静态 <E extends Exchange> ValueBuilder<E> outBodyAs (Class<T> type)	返回作为特定类型的出站消息正文的 predicate 和 value 构建器。
static ValueBuilder property (String name)	为交换的属性返回 predicate 和 value builder。
静态 ValueBuilder regexReplaceAll (Expression content, String regex, Expression replacement)	返回一个表达式，该表达式将所有正则表达式的出现替换为所给的替换。
静态 ValueBuilder regexReplaceAll (Expression content, String regex, String replacement)	返回一个表达式，该表达式将所有正则表达式的出现替换为所给的替换。
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 extends Exchange> ValueBuilder<E> systemProperty (String name)	返回给定系统属性的表达式。
静态 <E extends 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 类允许您修改由 Builder 方法返回的值。换句话说，ValueBuilder 中的方法提供了修改消息内容的简单方法。表 2.7 “ValueBuilder 类中的修饰符方法” 总结了 ValueBuilder 类中提供的方法。也就是说，表格仅显示用于修改所调用值的方法（有关完整详情，请参阅 API 参考文档）。

表 2.7. ValueBuilder 类中的修饰符方法

方法	描述
ValueBuilder<E> append (Object value)	使用给定值附加此表达式的字符串评估。
predicate contains (Object value)	创建一个 predicate，使左侧表达式包含右手表达式的值。
ValueBuilder<E> convertTo (Class type)	使用注册类型转换器将当前值转换为给定类型。
ValueBuilder<E> convertToString()	使用注册类型转换器将当前值转换为 String。
predicate endWith (Object value)	
<T> T evaluate (Exchange Exchange, Class<T> 类型)	
predicate in (Object… values)	
predicate in (Predicate… predicates)	
predicate isEqualTo (Object value)	如果当前值等于给定 值 参数，则返回 true。
predicate isGreaterThan (Object value)	如果当前值大于给定 值，则返回 true。
predicate isGreaterThanOrEqualTo (Object value)	如果当前值大于或等于给定 值 参数，则返回 true。
predicate isInstanceOf (Class type)	如果当前的值是给定类型的实例，则返回 true。
predicate isLessThan (Object value)	如果当前值小于给定 值 参数，则返回 true。
predicate isLessThanOrEqualTo (Object value)	如果当前值小于或等于给定 值 参数，则返回 true。
predicate isNotEqualTo (Object value)	如果当前值不等于给定 值 参数，则返回 true。
predicate isNotNull （）	如果当前值不是 **null**，则返回 true。

方法	描述
predicate isNull （）	如果当前值为 **null**，则返回 true。
predicate match (Expression expression)	
predicate not (Predicate predicate)	对 predicate 参数进行求值。
ValueBuilder prepend (Object value)	将这个表达式的字符串评估添加到给定值。
predicate regex (String regex)	
ValueBuilder<E> regexReplaceAll (String regex, Expression<E> 替换)	使用给定的替换替换正则表达式的所有情况。
ValueBuilder<E> regexReplaceAll (String regex, String replacement)	使用给定的替换替换正则表达式的所有情况。
ValueBuilder<E> regexTokenize (String regex)	使用给定的正则表达式，令牌将此表达式的字符串转换。
ValueBuilder sort (Comparator compareator)	使用给定比较器对当前的值进行排序。
predicate startsWith (Object value)	如果当前值与 **value** 参数的值匹配，则返回 true。
ValueBuilder<E> tokenize （）	使用逗号分隔此表达式的字符串转换。
ValueBuilder<E> tokenize (String token)	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之间进行转换：

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

数据格式

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

- Java serialization
- JAXB
- XMLBeans
- XStream

Java serialization

允许您将 Java 对象转换为二进制数据的 blob。对于此数据格式，unmarshalling 将二进制 blob 转换为 Java 对象，marshalling 会将 Java 对象转换为二进制 blob。例如，要从端点 SourceURL 中读取序列化 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 将 XML 数据类型转换为 Java 对象，marshalling 将 Java 对象转换为 XML 数据类型。在使用 JAXB 数据格式之前，您必须使用 JAXB 编译器编译 XML 架构，以生成代表架构中 XML 数据类型的 Java 类。这称为 绑定 架构。绑定 schema 后，您可以使用类似如下的代码为 Java 对象定义 unmarshal XML 数据的规则：

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

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

其中 GeneratedPackagename 是 JAXB 编译器生成的 Java 软件包的名称，其中包含代表 XML 模式的 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 类型之间的替代映射（请参阅 http://xmlbeans.apache.org/）。对于 XMLBeans，unmarshalling 将 XML 数据类型转换为 Java 对象，marshalling 将 Java 对象转换为 XML 数据类型。例如，要使用 XMLBeans 将 unmarshal XML 数据到 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，unmarshalling 将 XML 数据类型转换为 Java 对象，marshalling 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 是将端点嵌套在 contract swig-rhacm 中，例如通过应用 Data Format、Content Enricher 或验证步骤。条件或转换应用到来自消息，并将补充条件或转换应用到消息。

DataFormatBinding

DataFormatBinding 类对于您要定义 marshals 和 unmarshals 特定数据格式的绑定很有用（请参阅 第 2.6.2 节 “Marshalling 和 Unmarshalling”）。在这种情况下，您需要做的所有创建绑定都是创建一个 DataFormatBinding 实例，在构造器中传递对相关数据格式的引用。

例如，例 2.2 “JAXB Binding” 中的 XML DSL 片段显示一个绑定（带有 ID，jaxb），该绑定可以在与 Apache Camel 端点关联时 marshalling 和 unmarshalling 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

以下替代方案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，您可以使用 binding:NameOfBinding 为端点 URI 前缀，其中 NameOfBinding 是绑定的 bean ID （例如，在 Spring XML 中创建的绑定 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

除了使用前缀将绑定与端点关联外，您可以使关联是隐式的，因此绑定不需要出现在 URI 中。对于没有隐式绑定的现有端点，达到此目的的最简单方法是使用 BindingComponent 类嵌套端点。

例如，要将 jaxb 绑定与 activemq 端点关联，您可以定义一个新的 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."/>
```

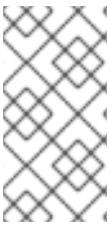
```
    </bean>

    <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）到 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此目的。

BindingComponent constructors

BindingComponent 类支持以下 constructors：

public BindingComponent()

没有参数表单。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，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 uriPrefix 关联。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 后修复，即 uriPostfix、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 外，它用于 marshalling 和 unmarshalling 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，以对传入到消费者端点的消息执行转换（在 from 元素中）。*

2. *实施补充的 org.apache.camel.Processor 类，对从生成者端点传出的消息执行反向转换（在 to 元素中出现）。*

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，您必须实施它才能定义自定义绑定。

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

当您需要将相同类型转换到许多不同类型的端点时，绑定很有用。

2.7. 属性 PLACEHOLDERS

概述

属性占位符功能可用于将字符串替换为各种上下文（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。如果要在不同的 Apache Camel 应用程序之间共享设置，或者要集中某些配置设置，则此功能很有用。

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 支持一个编码选项，它允许您使用特定字符集（如 UTF-8 ）读取 .properties 文件。但是，默认情况下，它会实现 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使用。**
- **如果所有占位符键都包含默认值，则不需要定义 PropertiesComponent。**
- **使用第三方函数查找属性值。它使您能够实施自己的逻辑。**

注意

提供三个开箱即用功能，以便从 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，必须符合标准的 Java 属性文件格式。每个属性设置都显示在其自己的行中，格式为 Key=Value。带有 @ 或 ! 的行作为第一个非空字符被认为是注释。

例如，属性文件可以包含内容，如 例 2.4 “属性文件示例” 所示。

例 2.4. 属性文件示例

```
# 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 classpath 上的位置，其中 PathName 是用正斜杠分隔的文件路径名。

file:PathName,PathName,...

指定文件系统的位置，其中 PathName 是用正斜杠分隔的文件路径名。

ref:BeanID

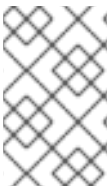
指定 registry 中 java.util.Properties 对象的 ID。

蓝图：BeanID

指定 cm:property-placeholder bean 的 ID，它用于 OSGi 蓝图文件的上下文，以访问 OSGi 配置管理员服务中定义的属性。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（位于 classpath 上），您可以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 classpath 解析器被默认使用。

使用系统属性和环境变量指定位置

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

可以使用语法 ${PropertyName} 将 Java 系统属性嵌入到位置解析器中。例如，如果 Red Hat Fuse 的根目录存储在 Java 系统属性 karaf.home 中，您可以在文件位置嵌入该目录值，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，您可以将该目录值嵌入到文件位置中，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，并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 （） 调用中所示，属性组件的名称 必须设置为 属性。

在 XML DSL 中，您可以使用 dedicated 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，缺少的 .properties 文件会导致引发错误）。

另外，如果您希望属性组件忽略任何使用 Java 系统属性或 O/S 环境变量指定的缺失位置，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）。占位符的语法取决于上下文，如下所示：

- 在端点 URI 和 Spring XML 文件中，需要把占位符指定为 {{ Key}}。
- 在设置 XML DSL 属性 swig- xs:string 属性时，使用以下语法设置：

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

其它属性类型（如 xs:int 或 xs:boolean）必须使用以下语法设置：

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

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

- 当设置 Java DSL EIP 选项 swig-wagonto 对 Java DSL 中的企业集成模式(EIP)命令设置选项时，在 fluent DSL 中添加如下 占位符（） 子句：

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

- 在 Simple language 表达式中，您会将占位符指定为 ${properties:Key}。

在端点 URI 中替换

每当端点 URI 字符串出现在路由中时，解析端点 URI 的第一步都会应用属性占位符解析器。占位符解析器自动替换在双花括号 {{ Key}} 之间出现的任何属性名称。例如，给定 例 2.4 “属性文件示例” 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，占位符解析器会在 registry 中查找 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 前缀 属性，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}}");
```

当明确指定方案时，您还可以选择在属性组件中指定选项。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

您还可以在 XML DSL 中使用属性占位符来设置 DSL 元素的各种属性。在这种情况下，**placholder** 语法也使用双括号 **{{ 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 type swig-5-4** 等属性值，例如 **< 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，假设属性文件定义了 **stop.flag** 属性，使其具有值 **true**，您可以使用此属性设置 **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

prop 前缀必须明确分配给 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，您可以使用属性占位符值来设置任何 EIP 选项，方法是添加表单的子使用，占位符("OptionName", "Key")。

例如，如果属性文件定义了 stop.flag 属性为值 true，您可以使用此属性设置 multicast 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，但在这种情况下，占位符的语法为

${properties:Key}。例如，您可以在简单表达式中替换 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}?");
```

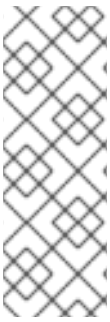
也可以使用语法 ${properties-location:Location:Key} 来覆盖属性文件的位置。例如，要使用 com/mycompany/bar.properties 属性文件中的设置替换 bar.quote 占位符，您可以定义一个简单表达式，如下所示：

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

在 XML DSL 中使用 Property Placeholders

在较旧的版本中，xs:string 类型属性用于支持 XML DSL 中的占位符。例如，timeout 属性将是 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。它允许您在 XML DSLs 中的属性中使用 prop 前缀。



注意

在 Multicast 中，将选项 stopOnException 设置为占位符的值，其键为 stop。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集成（实际上，集成默认为启用）。设置集成有两种基本方法，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，则 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 Apache Camel 语法（如 {{cool.end}}）的占位符可以通过查找 蓝图属性占位符 机制来隐式解析。

例如，请考虑以下路由，在 OSGi 蓝图文件中定义，其中路由中的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 cm:property-placeholder bean 初始化。在前面的示例中，cm:property-placeholder bean 与 camel.blueprint 持久 ID 关联，其中持久 ID 是引用 OSGi 配置管理员服务中一组相关属性的标准方法。换句话说，cm:property-placeholder bean 提供对 camel.blueprint 持久 ID 下定义的所有属性的访问。还可以为某些属性指定默认值（使用嵌套的 cm:property 元素）。

在蓝图的上下文中，Apache Camel 占位符机制在 bean registry 中搜索 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，以便占位符（如 {{result}} ）可以通过在蓝图属性占位符机制中查找密钥来解决（本例中为通过 myblueprint.placeholder bean）。

注意

默认蓝图占位符语法（直接访问蓝图属性）是 ${Key}。因此，在 camelContext 元素范围内，您必须使用的占位符语法为 ${Key}。而在 camelContext 元素的范围内，您必须使用的占位符语法为 {{ Key}}。

显式蓝图集成

如果要对 Apache Camel 属性占位符机制在哪里找到其属性，您可以定义 propertyPlaceholder 元素并明确指定解析器位置。

例如，请考虑以下蓝图配置，它与前一个示例不同，它创建一个明确的 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 元素通过将位置设置为 blueprint:myblueprint.placeholder 来指定要使用的 cm:property-placeholder bean。也就是说，蓝图：解析器明确引用了 cm:property-placeholder bean 的 ID myblueprint.placeholder。

如果蓝图文件中定义了多个 cm:property-placeholder bean，且您需要指定要使用的配置，则这种配置很有用。您还可以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 中查找属性，您可以在 classpath 上定义 propertyPlaceholder 元素，如下所示：

```
<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 Spring bean 类型 org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer 将 Apache Camel 属性占位符机制集成。

定义 BridgePropertyPlaceholderConfigurer，它取代了 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}} 语法引用配置的属性。

例如，定义一个 bridge 属性占位符，从 cheese.properties 文件中读取其属性设置：

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

或者，您可以将 BridgePropertyPlaceholderConfigurer 的 location 属性设置为指向 Spring 属性文件。Spring 属性文件语法被完全支持。

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)，它最初在 Sun 的 JDK 1.5 中提供。此 API 中的关键接口是 ExecutorService 接口，代表线程池。使用 concurrency API，您可以创建许多不同类型的线程池，覆盖各种场景。

Apache Camel 线程池 API

Apache Camel 线程池 API 通过为 Apache Camel 应用程序中的所有线程池提供中央工厂(org.apache.camel.spi.ExecutorServiceManager 类型)。以这种方式创建线程池提供了几个优点，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.*
- *线程自动给定信息名称，这对于日志记录和管理很有用。*

组件线程模型

有些 Apache Camel 组件 swig-wagon （如 SEDA、JMS 和 Jetty）都包括在多线程中。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

默认情况下，Apache Camel 中的一些标准处理器创建自己的线程池。这些线程感知处理器也与 Apache Camel 线程模型集成，它们提供了各种选项，供您自定义它们使用的线程池。

表 2.8 “处理器线程选项” 显示在线程感知处理器内置到 Apache Camel 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
聚合	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

线程处理器是一个通用的 DSL 命令，可用于将线程池引入到路由中。它支持以下选项来自定义线程池：

poolSize()

池中的最小线程数量（和初始池大小）。

maxPoolSize()

池中的最大线程数。

keepAliveTime()

如果有任何线程闲置时间超过这一时间段（以秒为单位指定），则它们将被终止。

timeUnit()

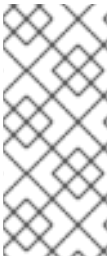
keep alive 的时间单位，使用 java.util.concurrent.TimeUnit 类型指定。

maxQueueSize()

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。

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，您无法使用这些选项覆盖 executorServiceRef 选项引用的线程池中的设置）。Apache Camel 验证 DSL 以强制执行这一点。

创建默认线程池

要为其中一个线程感知处理器创建一个默认线程池，请在 XML DSL 中使用 parallelProcessing （）子类别、Java DSL 或 parallelProcessing 属性启用 parallelProcessing 选项。

例如，在 Java DSL 中，您可以调用带有默认线程池（其中线程池用于同时处理多播目的地）的多播处理器，如下所示：

```
from("direct:start")
```

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

您可以在 XML DSL 中定义相同的路由，如下所示

```
<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 默认线程池配置集 获取其设置。默认线程池配置集有 表 2.9 “默认线程池配置文件设置” 中显示的设置（假设应用程序代码没有修改这些设置）。

表 2.9. 默认线程池配置文件设置

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文件设置，以便使用自定义设置创建所有后续默认线程池。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 default 线程池配置文件中的 poolSize 选项和 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），而不是替换默认线程池配置集。

自定义处理器的线程池

也可以使用 executorService 或 executorServiceRef 选项（其中使用这些选项而不是 parallelProcessing 选项）为线程感知处理器指定线程池。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 HEKETI explicitly 创建 ExecutorService （线程池）实例，并将其传递给 executorService 选项。**
- **指定自定义线程池配置集 wagon-wagoncreate 并注册自定义线程池工厂。当您使用 executorServiceRef 选项引用这个工厂时，处理器会自动使用工厂创建自定义线程池实例。**

当您将 bean ID 传递给 executorServiceRef 选项时，线程感知处理器首先尝试在 registry 中找到具有该 ID 的自定义线程池。如果没有使用该 ID 注册线程池，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。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。*
- *使用当前 CamelContext 中的 org.apache.camel.spi.ExecutorServiceManager 实例来创建线程池类。*

最终，这两种方法之间没有区别，因为 ThreadPoolBuilder 实际上使用 ExecutorServiceManager 实例定义。通常，首选 ThreadPoolBuilder，因为它提供了一种更简单的方法。但至少有一个线程类型(ScheduledExecutorService)，它们只能通过直接访问 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。

表 2.10. 线程池构建器选项

构建器选项	描述
maxQueueSize()	设置此线程池可在其传入任务队列中存储的最大待处理任务数。值 **-1** 指定未绑定队列。默认值取自默认线程池配置文件。
poolSize()	设置池中的最小线程数量（这也是初始池大小）。默认值取自默认线程池配置文件。
maxPoolSize()	设置池中可以的最大线程数。默认值取自默认线程池配置文件。
keepAliveTime()	如果有任何线程闲置时间超过这一时间段（以秒为单位指定），则它们将被终止。这允许线程池在负载比较轻时缩小。默认值取自默认线程池配置文件。

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。您可以指定四个可能的值： **CallerRuns** **（默认值）** 获取调用者线程来运行最新的传入的任务。作为副作用，此选项可防止调用者线程收到任何其他任务，直到它完成处理最新的传入的任务。 **Abort** 通过抛出异常中止最新的传入的任务。 **discard** 以静默方式丢弃最新的传入的任务。 **DiscardOldest** 丢弃最旧的未处理的任务，然后尝试在任务队列中排队最新的传入的任务。
build()	完成构建自定义线程池，并在指定为 **build （）** 参数的 ID 下注册新的线程池。

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 executorService Ref （） 选项，而不是直接将对象引用传给 executorService （） 选项，而是在 registry 中查找线程池，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 属性引用自定义线程池，根据 Spring registry 中的 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

如果您有很多自定义线程池实例，您可能会发现定义自定义线程池配置集更为方便，它充当线程池的工厂。每当从线程感知处理器引用线程池配置集时，处理器会自动使用配置集创建新线程池实例。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 创建自定义线程池配置集，并从路由内引用它，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（您可以在其中让 defaultProfile 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，并从路由内引用它，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

某些标准的基于 poll 的组件都基于 poll-wagon，如 File 和 FTP swig-wagonallow，您可以指定要使用的线程池。这使得不同的组件可以共享同一线程池，从而减少 JVM 中线程的整体数量。

例如，Apache Camel 组件参考指南 中的 File2 和 Apache Camel 组件参考指南 中的 Ftp2 都公开 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

要使应用日志更易读，通常最好自定义线程名称（用于标识日志中线程）。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一种更容易方法是在 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 failing

唯一线程标识符，作为递增计数器实施。

#name#

常规 Camel 线程名称。

#longName#

较长的线程名称可以包括端点参数，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，当您 Apache Camel 应用程序（由 CamelContext 实例表示）启动和路由关闭时，路由会自动启动。对于非关键部署，关闭序列的详细信息通常并不重要。但在生产环境中，现有任务应在关闭期间运行完，以避免数据丢失。您通常还希望控制路由关闭的顺序，以便不会违反依赖项（这会防止现有任务运行完成）。

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，使您能够控制路由的关闭顺序，并使当前任务运行完成。

设置路由 ID

最好为每个路由分配一个路由 ID。除了提供日志记录消息和管理功能外，使用路由 ID 可让您对路由停止和启动应用更大的控制。

例如，在 Java DSL 中，您可以通过调用 routeId （） 命令将路由 ID myCustomerRouteId 分配给路由，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 在开始时知道的所有路由都将自动启动。如果要手动控制特定路由的启动，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，请使用 布尔值参数 (true 或 false)或 String 参数(true 或 false)调用 autoStartup 命令。例如，您可以在 Java DSL 中禁用路由自动启动，如下所示：

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

您可以通过在 route 元素上将 autoStartup 属性设置为 false 来禁用 XML DSL 中路由的自动启动，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

您可以通过在 CamelContext 实例上调用 startRoute （） 和 stopRoute （） 方法，在 Java 中随时手动启动或停止路由。例如，要启动具有路由 ID nonAuto 的路由，请在 CamelContext 实例上调用

startRoute （） 方法，如下所示：

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

要停止具有路由 ID nonAuto 的路由，请在 CamelContext 实例上调用 stopRoute （） 方法，如下所示：

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

路由的启动顺序

默认情况下，Apache Camel 以非确定顺序启动路由。然而，在某些应用程序中，控制启动顺序非常重要。要控制 Java DSL 中的启动顺序，请使用 startupOrder （） 命令，该命令取正整数值作为其参数。值为最低整数值的路由首先启动，然后是具有连续更高的启动顺序值的路由。

例如，以下示例中的前两个路由通过 seda:buffer 端点链接。您可以通过分别分配启动顺序(2 和 1)，确保第一个路由片段在第二个路由片段后启动，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");
```

或者在 Spring XML 中，您可以通过设置 route 元素的 start Order 属性来实现相同的效果，如下所示：

例 2.6. 在 XML DSL 中启动顺序

```
<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>
```

每个路由都必须分配唯一的启动顺序值。您可以选择小于 1000 的任何正整数值。为 Apache Camel 保留 1000 和 over 的值，它会自动为路由分配这些值，而无需显式启动值。例如，上例中的最后一个路由会自动分配启动值 1000 （因此它会在前两个路由后启动）。

关闭序列

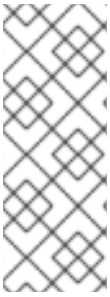
当 CamelContext 实例关闭时，Apache Camel 使用可插拔关闭 策略来控制关闭 序列。默认关闭策略实现以下关闭序列：

1. 路由按照 启动顺序相反 关闭。
2. 通常，关闭策略会等待当前活跃交换已处理。但是，运行任务的处理是可配置的。
3. 总体而言，关闭序列会按超时（默认值，300 秒）绑定。如果关闭序列超过此超时，则关闭策略将强制发生关闭，即使某些任务仍在运行。

路由关闭顺序

路由按照启动顺序相反关闭。也就是说，当使用 startupOrder （） 命令（在 Java DSL 中）或 startupOrder 属性（在 XML DSL 中）定义启动顺序的启动顺序时，要关闭的第一个路由是具有由启动顺序分配的 最高 整数值的路由，最后一个路由关闭是 具有最低 整数值的路由。

例如，在 例 2.5 “Java DSL 中的启动顺序” 中，要关闭的第一个路由片段是 ID 为第一个 的路由，而要关闭的第二个路由片段是 ID 为 第二个 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到： 公开外部访问的消费者端点的路由应先关闭，因为这有助于通过剩余的路由图限制消息流。



注意

Apache Camel 还提供选项 shutdownRoute (Defer)，它允许您指定路由必须是关闭的最后一个路由（覆盖启动顺序值）。但是，您应该很少需要这个选项。这个选项主要需要作为早期版本的 Apache Camel (prior 到 2.3)的一个临时解决方案，其中路由会按照 与启动顺序相同的顺序关闭。

关闭路由中运行的任务

如果路由在关闭启动时仍然正在处理消息，则关闭策略通常会等待，直到当前活动的交换在关闭路由之前完成处理。可以使用 shutdownRunningTask 选项在每个路由上配置此行为，该选项可使用以下值之一：

ShutdownRunningTask.CompleteCurrentTaskOnly

（默认） 通常，路由一次只在单一消息上运行，以便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 批处理用户。某些使用者端点（例如，文件、FTP、邮件、IBATIS 和 JPA）在一个时间对一批消息进行操作。对于这些端点，等待当前批处理中的所有消息都已完成。

例如，要安全地关闭 File consumer 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** （） 方法来更改超时的值。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

如果您实施自定义 **Apache Camel** 组件（也从 **org.apache.camel.Service** 接口继承），您可以通过实施 **org.apache.camel.spi.ShutdownPrepared** 接口来确保自定义代码接收关闭通知。这为组件提供了一个执行自定义代码以准备关闭的机会。

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**。

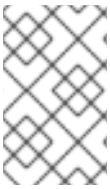
例如，当使用带有 **seda** 的路由或直接组件作为路由输入时，您可能需要使用其名称作为路由 **ID**，例如：

- **direct:foo- foo**
- **seda:bar- bar**
- **jms:orders- orders**

您可以使用 **NodeIdFactory** 为路由分配逻辑名称，而不使用自动分配名称。另外，您可以使用路由

URL 的 context-path 作为名称。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 路由策略概述

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，当前可用的实现可让您在策略指定的任何时间（或时间）启动、停止、暂停或恢复路由。

调度任务

调度的路由策略可以触发以下类型的事件：

- 在 指定的时间（或时间）启动路由。只有当路由当前处于已停止状态、等待激活时，此事件才会生效。
- 在 指定的时间（或时间）停止路由。只有当路由当前处于活跃状态时，此事件才会生效。
- 在路由 开始时挂起路由 HEKETI-wagontemporari-activate 消费者端点（如 from （）中指定的）。其余路由仍处于活动状态，但客户端将无法向路由发送新消息。
- 在路由开始时恢复 路由 HEKETI-wagonre-activate 消费者端点，将路由返回到完全活动状态。

quartz 组件

Quartz 组件是一个基于 Terracotta 的 Quartz 的计时器组件，它是作业调度程序的开源实施。Quartz 组件为简单的调度路由策略和 cron 调度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、暂停和恢复路由，其中这些事件的时间是通过提供初始事件的时间和日期（可选）来定义。要定义一个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

简单的调度路由策略取决于 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 “Simple Scheduled Route 的 Java DSL 示例” 演示了如何调度使用 Java DSL 启动的路由。初始开始时间 startTime 定义为当前时间后的 3 秒。该策略还配置为在初始开始时间后启动路由 第二 时间 3 秒，这通过将 routeStartRepeatCount 设置为 1，并将 routeStartRepeatInterval 设为 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. Simple Scheduled Route 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 “Simple Scheduled Route 的 XML DSL 示例” 演示了如何调度使用 XML DSL 启动的路由。

在 XML DSL 中，您可以通过在 route 元素上设置 routePolicyRef 属性将 路由 策略附加到路由。

例 2.8. Simple Scheduled Route 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 列表来指定路由上的多个策略。

定义日期和时间

简单调度路由策略中使用的触发器初始时间使用 java.util. Date 类型来指定。定义日期实例的最灵活方法是通过 java.util.GregorianCalendar 类来指定。使用 GregorianCalendar 类的方便构造器和方法来定义日期，然后通过调用 GregorianCalendar.getTime （） 来获得 日期 实例。

例如，要定义 2011 年 1 月 1 日的时间和日期，请致电 GregorianCalendar 构造器，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定义时间。默认情况下，它使用计算机上的本地时区。

正常关闭

当您将简单的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”）。这意味着该任务会在关闭路由前等待当前交换完成处理。但是，您可以设置超时，它会强制路由在指定时间后停止，无论路由是否完成处理交换。

超时的日志记录动态交换

如果在给定超时时间内完全关闭失败，则 Apache Camel 会执行更积极的关闭。它强制将路由、线程池等用于关闭。

超时后，Apache Camel 会记录有关当前动态交换的信息。它记录交换和当前交换路由的来源。

例如，日志显示有一个 inflight Exchange，它源自 route1，目前在 delay1 节点上处于相同的 route1。

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，则它会记录相同的 inflight Exchange 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 来关闭此关闭。

```
 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	Default（默认）	描述
routeStartDate	**java.util.Date**	None	指定首次启动路由的时间和时间。
routeStartRepeatCount	**int**	**0**	当设置为非零值时，指定路由应启动的次数。
routeStartRepeatInterval	**long**	**0**	指定开始之间的时间间隔，单位为毫秒。

停止路由

下表列出了调度一个或多个路由停止的参数。

参数	类型	Default（默认）	描述
routeStopDate	**java.util.Date**	None	指定第一次停止路由时的日期和时间。

参数	类型	Default（默认）	描述
routeStopRepeatCount	**int**	**0**	当设置为非零值时，指定应该停止路由的次数。
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，等待当前交换完成处理（宽限期）的时间。在无限宽限期中，设置为 0。
routeStopTimeUnit	**long**	**TimeUnit.MILLISECONDS**	指定宽限期的单位。

挂起路由

下表列出了调度一次或多次路由暂停的参数。

参数	类型	Default（默认）	描述
routeSuspendDate	**java.util.Date**	None	指定在第一次暂停路由时的日期和时间。
routeSuspendRepeatCount	**int**	0	当设置为非零值时，指定应暂停路由的次数。
routeSuspendRepeatInterval	**long**	0	指定暂停之间的时间间隔，以毫秒为单位。

恢复路由

下表列出了调度一次或多次路由恢复的参数。

参数	类型	Default（默认）	描述
routeResumeDate	**java.util.Date**	None	指定第一次恢复路由的时间和时间。
routeResumeRepeatCount	**int**	0	当设置为非零值时，指定路由应恢复的次数。

参数	类型	Default（默认）	描述
routeResumeRepeatInterval	**long**	0	指定恢复之间的时间间隔，单位为毫秒。

2.10.3. Cron Scheduled Route 策略

概述

Cron scheduled route 策略是一个路由策略，可让您启动、停止、暂停和恢复路由，其中这些事件的时间是使用 cron 表达式来指定的。要定义 cron 调度的路由策略，请创建以下类实例：

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

依赖项

简单的调度路由策略取决于 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。

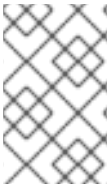
Java DSL 示例

例 2.9 “Cron Scheduled Route 的 Java DSL 示例” 演示了如何调度使用 Java DSL 启动的路由。该策略使用 cron 表达式 */3 * * * * ? 进行配置，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 Route 的 XML DSL 示例”演示了如何调度使用 XML DSL 启动的路由。

在 XML DSL 中，您可以通过在 route 元素上设置 routePolicyRef 属性将 路由 策略附加到路由。

例 2.10. Cron Scheduled Route 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 列表来指定路由上的多个策略。

定义 cron 表达式

cron 表达式 语法在 UNIX cron 工具中包含其源，它将调度作业在 UNIX 系统上的后台运行。Cron 表达式实际上是通配符日期和时间的语法，可让您定期指定单个事件或多个事件。

Cron 表达式由以下顺序 6 或 7 字段组成：

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

Year 字段是可选的，通常省略，除非您只想定义一次和一次发生一次的事件。每个字段由文字和特殊字符的组合组成。例如，以下 cron 表达式指定一个事件，该事件每天在午夜触发一次：

```
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，表示忽略此字段*。它始终出现在 DayOfMonth 字段或 DayOfWeek 字段中，因为它在逻辑上并不一致，以同时指定这两个字段。例如，如果要调度一天触发的事件，但只能从 Monday 到 Friday，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 / 来指定递增。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 表达式语法的完整说明，请参阅 Wikipedia 文章有关 CRON 表达式。

调度任务

您可以使用 cron 调度的路由策略来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	Default（默认）	描述
routeStartString	**字符串**	None	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停止的参数。

参数	类型	Default（默认）	描述
routeStopTime	**字符串**	None	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，等待当前交换完成处理（宽限期）的时间。在无限宽限期中，设置为 0。
routeStopTimeUnit	**long**	**TimeUnit.MILLISECONDS**	指定宽限期的单位。

挂起路由

下表列出了调度一次或多次路由暂停的参数。

参数	类型	Default（默认）	描述
routeSuspendTime	**字符串**	None	指定触发一个或多个路由挂起事件的 cron 表达式。

恢复路由

下表列出了调度一次或多次路由恢复的参数。

参数	类型	Default（默认）	描述
routeResumeTime	**字符串**	None	指定触发一个或多个路由恢复事件的 cron 表达式。

2.10.4. Route Policy Factory

使用路由策略工厂

从 Camel 2.14 开始提供

如果要为每个路由使用路由策略，您可以使用 org.apache.camel.spi.RoutePolicyFactory 作为每个路由创建 RoutePolicy 实例的工厂。当您要为每个路由使用相同类型的路由策略时，可以使用它。然后，您只需要配置工厂一次，创建的每个路由都会分配策略。

CamelContext 有 API 来添加工厂，如下所示：

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

在 XML DSL 中，您只使用工厂定义 <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，您可以根据需要拥有任意数量的路由策略工厂。只需再次调用 addRoutePolicyFactory，或者在 XML 中声明其他因素为 &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 camel XML 路由的实时重新加载，该路由将在从编辑器中保存 XML 文件时触发重新加载。在以下情况下可以使用此功能：

- Camel Main 类与 Camel Main 类

- Camel Spring Boot
- 从 camel:run maven 插件

但是，您还可以通过在 CamelContext 和提供自己的自定义策略上设置 ReloadStrategy 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 - 要运行您的 Camel 应用程序
- camel:validate - 验证您的源代码以获取无效的 Camel 端点 URI
- camel:route-coverage - 在单元测试后报告您的 Camel 路由范围

2.12.1. camel:run

Camel Maven 插件的 camel:run 目标用于在 Maven 的已分叉 JVM 中运行 Camel Spring 配置。您开始的一个很好的示例应用程序是 Spring Example。

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

这样便可轻松启动和测试路由规则，而无需编写 main (...)方法；它还允许您创建多个 jar 来托管不同的路由规则集合并轻松独立测试它们。Camel Maven 插件编译 maven 项目中的源代码，然后使用 classpath 上的 XML 配置文件启动 Spring ApplicationContext，网址为 META-INF/spring configured.xml。如果要更快地引导 Camel 路由，您可以尝试 camel:embedded。

2.12.1.1. 选项

Camel Maven 插件运行目标支持以下选项，可从命令行配置(use -D 语法)，或者在 < configuration > 标签的 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运行的时间持续时间（秒）。值 swig 0 将永久运行。
durationIdle	-1	设置应用程序在终止前可以闲置的空闲持续时间（秒）。值 swig 0 将永久运行。
durationMaxMessages	-1	设置应用程序进程在终止前的最大消息数。
logClasspath	false	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，它会扫描 OSGI-INF/blueprint &&.xml 中的 OSGi 蓝图文件。您需要将 camel:run 插件配置为使用蓝图，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>
```

这允许您引导您想要的任何蓝图服务，无论它们是否与 Camel 无关，还是任何其他蓝图。camel:run 目标可以自动检测 camel-blueprint 是否在 classpath 上，或者项目中是否有蓝图 XML 文件，因此您不再需要配置 useBlueprint 选项。

2.12.1.3. 使用有限蓝图容器

我们使用 Felix Connector 项目作为蓝图容器。此项目不是功能完整的蓝图容器。为此，您可以使用 Apache Karaf 或 Apache ServiceMix。您可以使用 applicationContextUri 配置来指定明确的蓝图 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>
```

applicationContextUri 从 classpath 加载文件，因此上例中的 myBlueprint.xml 文件必须位于 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 是文件名，用于加载配置 admin 服务属性文件。

2.12.1.4. 运行 CDI

camel:run 插件还支持运行 CDI 应用程序。这可让您引导任何您想要的 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 的依赖项中，如本例中。在 Camel 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

您可以配置在 camel:run 执行时是否应记录 classpath。您可以使用以下方法在配置中启用它：

```
<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 文件的实时重新载入

您可以将插件配置为扫描 XML 文件更改，并触发在这些 XML 文件中所含的 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。这样，您可以从编辑器中编辑源代码并保存文件，并让正在运行的 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 &lt ;route> 被支持。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
- 重复路由 ID

然后，您可以从命令行或 Java 编辑器（如 IDEA 或 Eclipse）运行 camel:validate 目标。

```
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>
```

该阶段决定插件何时运行。在上例中，阶段是 process-classes，它在编译主源代码后运行。maven 插件也可以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 项目中运行目标

您还可以在任何 Maven 项目中运行 validate 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 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] ------------------------------------------------------------------------
```

验证通过，并且验证 4 个端点。现在，假设我们在源代码中的某个 Camel 端点 URI 中进行了拼写错误，例如：

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

更改了在 period 选项中包含拼写错误

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

当再次运行 validate 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，可从命令行配置(use -D 语法)，或者在 < configuration > 标签的 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。如果项目默认使用与这个插件不同的 Camel 版本，则需要此参数。
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	是否包含要验证无效 Camel 端点的 Java 文件。
includeXml	true	是否包含要为无效 Camel 端点进行验证的 XML 文件。
includeTest	false	是否包含测试源代码。

includes		要将 java 和 xml 文件的名称过滤为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	是否忽略能够解析端点 URI 或简单表达式。
ignoreLenientProperties	true	是否忽略使用 lenient 属性的组件。当这是 true 时，URI 验证是更严格的，但由于使用了 lenient 属性，则会导致属性不是组件的一部分，但在 URI 中。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 中使用已弃用的选项。
duplicateRouteId	true	**Camel 2.20** 是否验证重复路由 ID。路由 ID 应该是唯一的，如果重复，则 Camel 将无法启动。
directOrSedaPairCheck	true	**Camel 2.23** 是否验证发送到非现有用户的直接/seda 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，要关闭从命令行中忽略已弃用选项的使用，您可以运行：

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

请注意，您必须使用 camel. 前缀 the -D 命令参数，如 camel.ignoreDeprecated 作为选项名称。

2.12.2.3. 使用 include test 验证端点

如果您有 Maven 项目，则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

从单元测试中生成您的 Camel 路由覆盖的报告。您可以使用它来知道 Camel 路由中的哪些部分已使用或没有被使用。

2.12.3.1. 启用路由覆盖

您可以在运行单元测试时启用路由覆盖：

- 为所有测试类设置全局 JVM 系统属性
- 如果使用 camel-test-spring 模块，每个测试类使用 @EnableRouteCoverage 注释
- 如果使用 camel-test 模块，则覆盖每个测试类的 isDumpRouteCoverage 方法

2.12.3.2. 使用 JVM 系统属性启用路由覆盖范围

您可以打开 JVM 系统属性 CamelTestRouteCoverage，以启用所有测试案例的路由覆盖。这可以在 maven-surefire-plugin 配置中进行：

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

运行测试时或从命令行中：

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

2.12.3.3. 通过 @EnableRouteCoverage 注释启用

如果您正在使用 camel-test-spring 测试，您可以在测试类中添加 @EnableRouteCoverage 注解来启用单元测试类中的路由覆盖：

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

2.12.3.4. 通过 isDumpRouteCoverage 方法启用

但是，如果您使用 camel-test，且您的单元测试会扩展 CamelTestSupport，则可以打开路由覆盖，如下所示：

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

在 RouteCoverage 方法下可以覆盖的路由必须具有唯一的 id，换句话说，您无法使用匿名路由。您可以在 Java DSL 中使用 routeId 完成此操作：

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

在 XML DSL 中，您只通过 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

TO 生成路由覆盖报告，使用以下方法运行单元测试：

```
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
```

这报告了通过精确的源代码行报告，哪些路由缺少路由覆盖：

```
[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，在 count 列中带有 的第二行以 0，因此不涵盖它。 我们还可以在源代码文件中看到这是 34 行，它位于 my-camel.xml XML 文件中。

2.12.3.6. 选项

Camel Maven 插件 覆盖 目标支持从命令行(use -D 语法)配置以下选项，或者在 < configuration > 标签的 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，是否失败。
includeTest	false	是否包含测试源代码。
includes		要将 java 和 xml 文件的名称过滤为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（在没有分配任何路由 ID 的情况下的路由）。通过使用路由 id，然后其更安全将路由涵盖数据与路由源代码匹配。对于路由覆盖，匿名路由不太安全，因为准确知道测试的路由与源代码中的路由相对应。

2.13. 运行 APACHE CAMEL STANDALONE

当您作为独立应用程序运行 camel 时，它提供用于运行应用程序的 Main 类，并保持其运行，直到 JVM 终止为止。您可以在 org.apache.camel.main Java 软件包中找到 MainListener 类。

以下是 Main 类的组件：

- ***camel-core JAR 在 org.apache.camel.Main 类中***
- ***camel-spring JAR in the org.apache.camel.spring.Main 类***

以下示例演示了如何从 Camel 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 “Exchanges”。onCompletion 命令具有以下特性：

- **OnCompletion 命令的范围可以是全局的，也可以是每个路由。路由范围会覆盖全局范围。**

- OnCompletion 可以配置为在失败时触发。
- onWhen predicate 在某些情况下只能触发 onCompletion。
- 您可以定义是否使用线程池，但默认值不是线程池。

仅在Completion 时路由范围

当在交换上指定了 onCompletion DSL 时，Camel 会关闭新的线程。这允许原始线程继续，而不会干扰 onCompletion 任务。路由只支持 完成一个。在以下示例中，当交换完成还是失败时，会触发 onCompletion。这是默认操作。

```
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>
```

要失败时触发 onCompletion，可以使用 onFailureOnly 参数。同样，若要成功触发 onCompletion，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，onFailureOnly 和 onCompleteOnly 作为布尔值在 onCompletion** 标签上表示：*

```
<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 onCompletion 的全局范围

*要针对 **多个路由定义完成**：*

```
// 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 **onCompletion，请使用 onWhen predicate。当消息的正文包含 Hello** 一词时，以下示例将触发 **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");
```

在带有或不使用线程池的情况下使用

从 Camel 2.14 开始，默认情况下，Completion 将不会使用线程池。要强制使用线程池，可以将 executorService 设置为 true，或者将 parallelProcessing 设置为 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，格式如下：

```
<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 前运行

onCompletion 可使用两种模式运行：

- **AfterConsumer - 在消费者完成后运行的默认模式**
- **BeforeConsumer - 在消费者将响应写入调用前运行。这允许 完成 修改交换程序，如添加特殊标头或将交换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

从 Camel 2.14 开始提供

虽然 Camel 提供了很多与现有的指标集成，但为 Camel 路由添加了 Codahale 指标。这使得最终用户能够无缝地源 Camel 路由信息及其使用 Codahale 指标收集的现有数据。

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，只有在您有一种显示它们的方式时，才会使用性能指标；可以使用任何可与 JMX 集成的工具，因为可通过 JMX 提供指标。另外，实际数据是 100% Codehale JSON。

指标路由策略

可以通过基于每个路由定义 MetricsRoutePolicy 来完成单一路由的 Codahale 指标。

从 Java 创建 MetricsRoutePolicy 实例，以分配为路由的策略。这如下所示：

```
from("file:src/data?noop=true").routePolicy(new MetricsRoutePolicy()).to("jms:incomingOrders");
```

在 XML DSL 中，您 定义了一个 <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"/>
[...]
```

指标路由策略工厂

此工厂允许每个路由添加一个 RoutePolicy，这些路由使用 Codahale 指标公开路由利用率统计。此工厂可在 Java 和 XML 中使用，如下例所示。

从 Java 中，您要将工厂添加到 CamelContext 中，如下所示：

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

从 XML DSL 中，您定义一个 &lt ;bean>，如下所示：

```
<!-- 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，您可以存放来自 org.apache.camel.component.metrics.routepolicy.MetricsRegistry 的 com.codahale.metrics.MetricRegistry Service，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：

Name	default	描述
durationUnit	**TimeUnit.MILLISECONDS**	指标报告器中使用的持续时间单位，或者在将统计信息转储为 json 时。
jmxDomain	**org.apache.camel.metrics**	JXM 域名。
metricsRegistry		允许使用共享 **com.codahale.metrics.MetricRegistry**。如果没有提供，则 Camel 将创建一个由此 CamelContext 使用的共享实例。
prettyPrint	**false**	在以 json 格式输出统计信息时，是否使用用户用户打印。
rateUnit	**timeunit.SECONDS**	指标报告器或以 json 转储统计时使用的单位。
useJmx	**false**	使用 **com.codahale.metrics.JmxReporter** 将精细的统计报告给 JMX。 请注意，如果在 CamelContext 上启用了 JMX，则将在 JMX 树中的服务类型下放入 **MetricsRegistryService** mbean。该 mbean 有一个操作来使用 json 输出统计信息。只有在您希望每个统计类型的精细 mbeans 时才需要将 useJmx 设置为 true。

2.16. JMX 命名

概述

Apache Camel 允许您通过定义 管理名称模式 来自定义 JMX 中显示的 CamelContext bean 的名称。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>
```

如果您没有为 CamelContext bean 明确设置名称模式，Apache Camel 会恢复到默认的命名策略。

默认命名策略

默认情况下，在 OSGi 捆绑包中部署的 CamelContext bean 的 JMX 名称等同于捆绑包中的 OSGi 符号链接名称。例如，如果 OSGi 符号链接名称为 MyCamelBundle，JMX 名称是 MyCamelBundle。如果捆绑包中有多个 CamelContext，则 JMX 名称通过将计数器值添加为后缀来撤销。例如，如果 MyCamelBundle 捆绑包中有多个 Camel 上下文，则对应的 JMX MBeans 被命名如下：

```
MyCamelBundle-1
MyCamelBundle-2
MyCamelBundle-3
...
```

自定义 JMX 命名策略

默认命名策略的一个缺点是，您无法保证给定 CamelContext Bean 在运行之间具有相同的 JMX 名称。如果要在运行之间具有更大的一致性，您可以通过为 CamelContext 实例定义 JMX 名称 模式来更精确地控制 JMX 名称。

在 Java 中指定名称模式

要在 Java 中的 CamelContext 上指定名称模式，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 中的 CamelContext 上指定名称模式，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与以下令牌之一混合来构建 JMX 名称模式：

表 2.11. JMX 名称模式令牌

令牌	描述
#camelId#	**CamelContext** bean 上的**id** 属性的值。
#name#	与 **#camelId#** 相同。
#counter failing	递增计数器（从 **1**开始）。
#bundleId#	已部署捆绑包(仅限OSGi)的 OSGi 捆绑包 ID。
#symbolicName#	OSGi 符号链接名称 （仅限 OSGi）
#version#	OSGi 捆绑包版本 （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

由于定制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 在启动时会失败并报告 MBean 已存在 异常。因此，您应该格外小心，以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，以便在不需要时切断原始消息的副本。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 应用程序代码中，设置 AllowUseOriginalMessage=true，然后使用 getOriginalMessage 方法。*

注意

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

第 3 章 企业级集成模式简介

摘要

Apache Camel 的企业级集成模式由 Gregor Hohpe 和obby Woolf 编写的名称的一书中发了。这些作者描述的模式为开发企业集成项目提供了卓越的 toolbox。除了为讨论集成架构提供通用语言外，还可以直接使用 Apache Camel 的编程接口和 XML 配置实施许多模式。

3.1. PATTERNS 概述

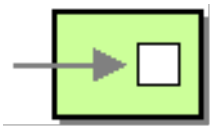
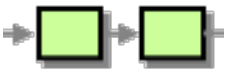
企业级集成模式 一书

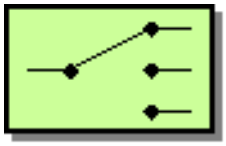
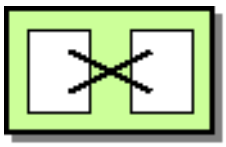
Apache Camel 支持来自书中的大多数模式， Greg or Hohpe 和 Bobby Woolf 的企业集成模式。

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引入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

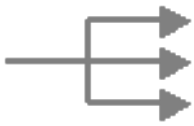
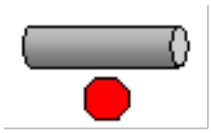
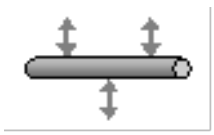
图标	Name	使用案例
	图 5.1 “Message Pattern”	消息通道连接的两个应用程序如何交换信息？
	图 5.2 “消息频道模式”	一个应用程序如何使用消息传递与另一个应用程序通信？
	图 5.3 “消息端点模式”	应用程序如何连接到消息传递频道来发送和接收消息？
	图 5.4 “管道和过滤器模式”	我们在消息上执行复杂的处理，同时仍然保持独立性和灵活性？

图标	Name	使用案例
	图 5.7 “消息路由器模式”	您可以如何分离各个处理步骤，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 “消息转换模式”	使用不同数据格式的系统如何使用消息传递相互通信？

消息传递频道

消息传递通道是用于在消息传递系统中连接参与者的基本组件。表 3.2 “消息传递频道” 中的模式描述了各种可用的消息传递频道。

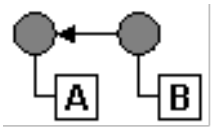
表 3.2. 消息传递频道

图标	Name	使用案例
	图 6.1 “指向点频道模式”	调用者如何确保一个接收方将接收文档或将执行调用？
	图 6.2 “发布订阅频道模式”	发送者如何将事件广播到所有感兴趣的接收器？
	图 6.3 “死信频道模式”	消息传递系统无法发送什么消息？
	图 6.4 “保证交付模式”	发送者如何确保发送消息，即使消息传递系统失败也是如此？
	图 6.5 “消息总线模式”	什么是支持独立、分离的应用程序一起工作的架构，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？

消息构造

表 3.3 “消息结构” 中显示的消息构造模式描述了通过系统传递的信息的各种形式和功能。

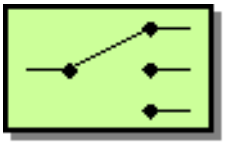
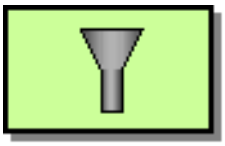
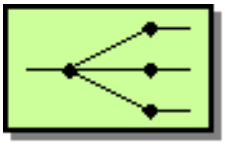
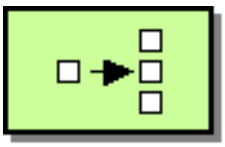
表 3.3. 消息结构

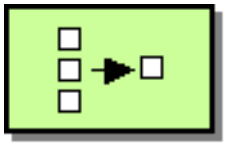
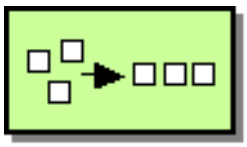
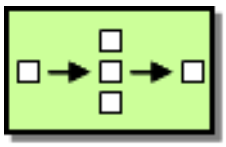
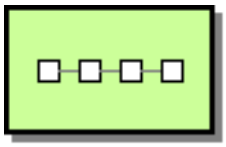
图标	Name	使用案例
	“概述”一节	请求者如何识别生成收到的回复的请求？
	第 7.3 节 “返回地址”	销售代表如何知道如何发送回复？

消息路由

表 3.4 “消息路由” 中显示的消息路由模式描述了将消息频道链接在一起的各种算法，包括可应用到消息流的各种算法（无需修改消息正文）。

表 3.4. 消息路由

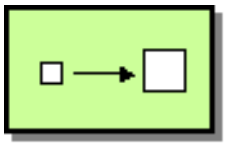
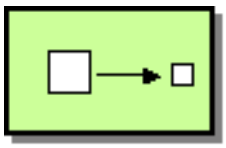
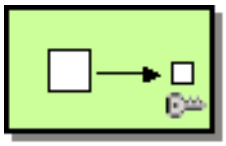
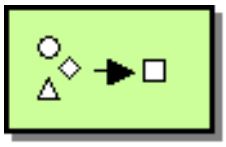
图标	Name	使用案例
	第 8.1 节 “基于内容的路由器”	我们如何处理在单个逻辑功能（如清单检查）实施分散到多个物理系统中的情况？
	第 8.2 节 “Message Filter”	组件如何避免收到不间信息？
	第 8.3 节 “接收者列表”	如何将消息路由到动态指定的接收者列表？
	第 8.4 节 “Splitter”	如果消息包含多个元素，则可能需要以不同方式处理每个元素？

图标	Name	使用案例
	第 8.5 节 “聚合器”	我们如何组合单个的结果，但相关消息的结果可以作为一个整体进行处理？
	第 8.6 节 “Resequencer”	我们可以如何获得相关流，但顺序不足，消息会回到正确的顺序？
	第 8.14 节 “由消息处理器”	在处理由多个元素组成的消息时，如何维护整个消息流，每个元素可能需要不同的处理？
	第 8.15 节 “scatter-Gather”	当需要向多个接收方发送消息时，如何维护整个消息流，每个接收者都可以发送回复？
	第 8.7 节 “路由片段”	当设计时没有知道步骤，且每个消息都不同时，我们如何通过一系列处理步骤连续路由消息？
	第 8.8 节 “Throttler”	如何节流消息以确保特定端点不会超载，或者我们没有超过与某些外部服务的协议的 SLA？
	第 8.9 节 “Delayer”	如何延迟发送消息？
	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 “服务调用”	如何在 registry 中查找服务，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？
	第 8.17 节 “sampling”	如何在给定时间段内对一个信息进行抽样，以避免过载下游路由？

消息转换

表 3.5 “message Transformation” 中显示的消息转换模式描述了如何修改消息的内容，以满足各种用途。

表 3.5. message Transformation

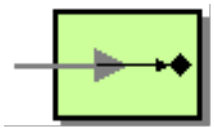
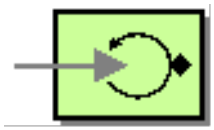
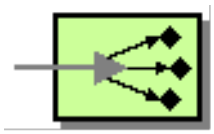
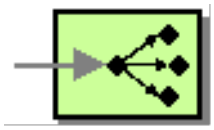
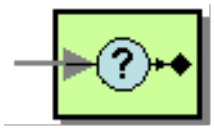
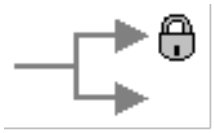
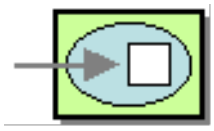
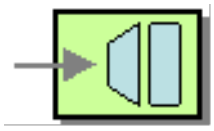
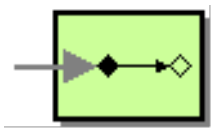
图标	Name	使用案例
	第 10.1 节 “内容增强”	如果消息 originator 没有所有必需的数据项，如何与其他系统通信？
	第 10.2 节 “内容过滤器”	当您只想处理一些数据项目时，如何简化处理大量消息？
	第 10.4 节 “声明检查 EIP”	我们如何在不牺牲信息内容的情况下，减少系统中发送的信息的数据卷？
	第 10.3 节 “规范化程序”	如何处理具有语义等效但到达不同格式的消息？
	第 10.5 节 “排序”	如何对邮件的正文进行排序？

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。消息传递端点模式（如 表 3.6 “消息传递端点” 所示）描述了可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

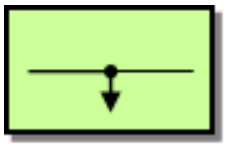
图标	Name	使用案例
	第 11.1 节 “消息传递映射程序”	如何在域对象和消息传递基础架构之间移动数据，同时保持两者相互独立？

图标	Name	使用案例
	第 11.2 节 “event Driven Consumer”	应用程序如何在消息可用时自动使用？
	第 11.3 节 “polling Consumer”	当应用程序就绪时，应用程序如何消耗消息？
	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”	单个频道中的多个用户如何协调其消息处理？
	第 11.6 节 “selective Consumer”	如何选择希望接收哪些消息？
	第 11.7 节 “durable Subscriber”	订阅者如何避免在没有侦听信息时缺少消息？
	第 11.8 节 “idempotent Consumer”	消息接收器如何处理重复消息？
	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”	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 “服务激活器”	应用如何设计由各种消息传递技术调用的服务，以及非消息传递技术？

系统管理

表 3.7 “系统管理” 中显示的系统管理模式，描述了如何监控、测试和管理消息传递系统。

表 3.7. 系统管理

图标	Name	使用案例
	第 12 章 *系统管理*	如何检查点对点频道上传输的消息？

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 （域特定语言），它是一个简单而强大的流畅 API，可通过任何 REST 组件分层并提供与 OpenAPI 集成。

4.1. CAMEL 中的 REST 概述

概述

Apache Camel 提供了许多不同的方法和组件，用于定义 Camel 应用程序中的 REST 服务。本节提供了这些不同方法和组件的快速概述，以便您可以决定哪种实施和 API 最适合您的要求。

什么是 REST？

Representational State Transfer (REST)是关于通过 HTTP 传输数据的分布式应用程序的架构，它仅使用四个基本 HTTP 动词： GET、POST、PUT 和 DELETE。

与 SOAP 等协议（将 HTTP 视为 SOAP 消息传送协议）的协议不同，REST 架构直接利用 HTTP。关键的洞察是，HTTP 协议 本身 通过一些简单的约定来增强，它非常适合充当分布式应用程序的框架。

REST 调用示例

由于 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，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如下的 URL：

```
http://localhost:9091/say/hello/Garp
```

然后 Hello World REST 服务可能会返回响应字符串，例如：

```
Hello Garp
```

显示在浏览器窗口中。易于调用 REST 服务，使用标准浏览器（或 curl 命令行工具）都不多于标准浏览器（或 curl 命令行工具）是为什么 REST 协议迅速获得流行的原因之一。

REST 打包程序层

以下 REST 打包程序层提供了用于定义 REST 服务的简化语法，并可在不同的 REST 实施之上分层：

REST DSL

REST DSL （在 camel-core中）是一个 facade 或 wrapper 层，它为定义 REST 服务提供了简化的构建器 API。REST DSL 本身不 提供 REST 实施：它必须与底层 REST 实施相结合。例如，以下 Java 代码演示了如何使用 REST DSL 定义简单的 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 组件本身 不提供 REST 实施。它必须与底层 REST 实施相结合。

如果您没有显式配置 HTTP 传输组件，则 REST DSL 会通过检查 classpath 上的可用组件来自动发现要使用的 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 classpath 上没有 HTTP 组件，且您没有明确配置 HTTP 传输，则默认的 HTTP 组件为 camel-http。

注意

自动发现要使用哪个 HTTP 组件的功能是 Camel 2.18 中的新功能。Camel 2.17 不提供它。

以下 Java 代码演示了如何使用 camel-rest 组件定义简单的 Hello World 服务：

```
from("rest:get:say:/hello/{name}").transform().simple("Hello ${header.name}");
```

REST 实现

Apache Camel 通过以下组件提供几个不同的 REST 实现：

spark-Rest 组件

Spark-Rest 组件(camel-spark-rest)是一个 REST 实施，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(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 组件相反，URI 中变量的语法为 :name，而不是 {name}。

注意

Spark-Rest 组件需要 Java 8。

restlet 组件

Restlet 组件（在 camel-restlet中）是一个 REST 实施，它原则上可以位于不同的传输协议之上（尽管此组件只针对 HTTP 协议进行测试）。此组件还提供与 Restlet Framework 集成，它是用于在 Java 中开发 REST 服务的商业框架。例如，以下 Java 代码演示了如何使用 Restlet 组件定义简单的 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）是一个将 Java servlet 绑定到 Camel 路由的组件。换句话说，Servlet 组件允许您打包和部署 Camel 路由，就像它是标准的 Java servlet 一样。因此，如果您需要在 servlet 容器中部署 Camel 路由（例如，在 Apache Tomcat HTTP 服务器或 JBoss Enterprise Application Platform 容器中）部署 Camel 路由，则 Servlet 组件特别有用。

但是，自有的 Servlet 组件并不提供任何方便的 REST API 来定义 REST 服务。因此，使用 Servlet 组件的最简单方法是将其与 REST DSL 相结合，以便您可以使用用户友好的 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)是一种将 REST 请求绑定到 Java 对象的框架，其中 Java 类必须使用 JAX-RS 注释来解码，才能定义绑定。JAX-RS 框架相对比较成熟，为开发 REST 服务提供了复杂的框架，但它稍微复杂于编程。

JAX-RS 与 Apache Camel 集成由 CXFRS 组件实施，该组件通过 Apache CXF 进行分层。在概述了中，JAX-RS 使用下列注释将 REST 请求绑定到 Java 类（其中这只是许多可用注释的不完整示例）：

@Path

此注释可以映射上下文路径到 Java 类，或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 方法的注解。

@PathParam

将 URI 参数映射到 Java 方法参数的注解，或者将 URI 参数注入字段。

@QueryParam

将查询参数映射到 Java 方法参数的注解，或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。但是 Apache CXF 还支持将 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 facade

REST DSL 有效地是一个 传真，它为在 Java DSL 或 XML DSL （域特定语言）中定义 REST 服务提供了简化的语法。REST DSL 不实际提供 REST 实施，它只是围绕 现有 REST 实施（在 Apache Camel

中有多个）的包装程序。

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，用于定义 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 实施，因此您需要做的第一件事是选择 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 component (camel-jetty)。
- restlet 组件(camel-restlet)。
- Undertow 组件(camel-undertow)。

注意

Rest 组件(camel-core的一部分)不是 REST 实施。与 REST DSL 一样，Rest 组件是一个灰色的，提供简化的语法来使用 URI 语法定义 REST 服务。Rest 组件还需要底层 REST 实施。

配置 REST DSL 以使用 REST 实现

要指定 REST 实施，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，您可以使用类似 Java DSL 中的 builder 表达式：

```
restConfiguration().component("spark-rest").port(9091);
```

在 XML DSL 中使用类似如下的元素（作为 camelContext的子级）：

```
<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（使用标准的 Java DSL 语法进行路由定义）。

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 verb 子句。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任何一种语法：

- **动词子句以 to （） 关键字结尾。例如：**

  ```
  get("...").Option()+.to("...")
  ```

- verb 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，就像您对常规 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的构建器类型：

restConfiguration （）

配置 REST DSL 以使用特定的 REST 实施(Spark-Rest)。

rest()

使用 REST DSL 定义服务。每个 verb 子句都由一个 to （） 关键字终止，它将传入的消息转发到 直接 端点（同一应用中 直接 组件路由）。

from()

定义常规 Camel 路由。

带有 XML 的 REST DSL

在 XML 中，要使用 XML DSL 定义服务，请将 rest 元素定义为 camelContext 元素的子元素。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 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，然后作为所有 verb 子句的路径作为前缀。例如，给定以下 Java DSL 片断：

```
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，可以（不完美）在每个 verb 子句中指定完整路径：

```
rest()
    .get("/say/hello").to("direct:hello")
    .get("/say/bye").to("direct:bye");
```

使用动态功能

REST DSL 支持 toD 动态 to 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，相同的详情类似如下：

```
<rest uri="/say">
  <get uri="/hello//{language}">
    <toD uri="jms:queue:hello-${header.language}"/>
  </get>
<rest>
```

有关 toD dynamic to 参数的详情，请参考 “动态到”一节。

URI 模板

在 verb 参数中，您可以指定一个 URI 模板，它可让您捕获指定属性中的特定路径片段（然后映射到 Camel 消息标头）。例如，如果要对 Hello World 应用程序进行个性化，以便按名称问候调用者，您可以定义类似如下的 REST 服务：

```
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，并将此捕获的文本复制到 名称 消息标头中。如果您通过发送以 /say/hello/Joe 结尾的 GET HTTP Request 来调用该服务，HTTP 响应是 Hello Joe。

嵌入式路由语法

您可以使用 route （） 关键字(Java DSL)或 to element (XML DSL)终止 verb 子句，而是使用 route （）关键字(Java DSL)或 route 元素(XML DSL)直接嵌入 Apache Camel 路由。route （） 关键字可让您使用以下语法将路由嵌入到 verb 子句中：

```
RESTVerbClause.route("...").CamelRoute.endRest()
```

其中 endRest （） 关键字(Java DSL)是一个必要的标点标记，可让您分隔 verb 子句（当 rest （）构建器中有多个 verb 子句）。

例如，您可以重构 Hello World 示例以使用嵌入式 Camel 路由，如 Java DSL 所示：

```
rest("/say")
    .get("/hello").route().transform().constant("Hello World").endRest()
    .get("/bye").route().transform().constant("Bye World");
```

在 XML DSL 中如下所示：

```
<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 intercept （）），则这些 exception 子句和拦截器也在嵌入的路由中处于活跃状态。

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 会通过检查 classpath 上的可用组件来自动发现要使用的 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 classpath 上没有 HTTP 组件，且您没有明确配置 HTTP 传输，则默认的 HTTP 组件为 camel-http。

指定请求和响应的内容类型

您可以使用 Java 中的 consumes （） 和 generate （） 选项过滤 HTTP 请求和响应 的内容类型，或者在 XML 中生成 属性。例如，一些通用内容类型（官方称为 互联网介质类型）如下：

- text/plain
- text/html
- text/xml
- application/json
- application/xml

内容类型被指定为 REST DSL 中 verb 子句上的选项。例如，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，您可以使用类似如下的 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>
```

您还可以指定要以逗号分隔的列表 形式使用 （） 或 generate （）的参数。例如，使用("text/plain, application/json")。

其他 HTTP 方法

有些 HTTP 服务器实现支持额外的 HTTP 方法，它们不是由 REST DSL, get （）, head （）, put （）, post （）, delete （）, patch （） 的标准集合提供。要访问额外的 HTTP 方法，您可以在 Java DSL 中使用 generic 关键字 verb （）， 并在 XML DSL 中使用 generic 元素 verb。

例如，要在 Java 中实施 TRACE HTTP 方法：

```
rest("/say")
    .verb("TRACE", "/hello").route().transform();
```

其中 transform （） 将 IN 消息的正文复制到 OUT 消息的正文，从而回显 HTTP 请求。

要在 XML 中实施 TRACE 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，您可以定义一个自定义 HTTP 错误消息：

1. 通过将 Exchange.HTTP_RESPONSE_CODE 标头键设置为错误代码值（如 400、404 等等）来指定 HTTP 错误代码。此设置指示您要发送错误消息回复的 REST DSL，而不是常规响应。

2. 使用自定义错误消息填充消息正文。

3. 如果需要，设置 Content-Type 标头。

4. 如果您的 REST 服务被配置为 marshal 到 Java 对象（已启用 bindingMode），您应该确保启用了 skipBindingOnErrorCode 选项（默认为）。这是为了确保 REST DSL 在发送响应时不会尝试 unmarshal 消息正文。

   有关对象绑定的详情，请参阅 第 4.3 节 “到 Java 对象和从 Java 对象进行复制”。

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");
```

在本例中，如果输入 ID 是小于 100 的数字，我们会返回自定义错误消息，使用 UserErrorService bean，其实现如下：

```
// 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。

您可以使用关键字（如查询参数 详细 ）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 false 将作为默认值插入。

```
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");
```

在自定义 HTTP 错误消息中嵌套 JsonParserException

您可能要返回自定义错误消息的一个常见情况是以打包 JsonParserException 异常。例如，您可以方便地利用 Camel 异常处理机制创建自定义 HTTP 错误消息，以及 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，REST DSL 选项可以直接应用到服务定义的基本部分（即紧跟在 rest （）后面），如下所示：

```
rest("/email").consumes("text/plain").produces("text/html")
    .post("/to/{recipient}").to("direct:foo")
    .get("/for/{username}").to("direct:bar");
```

在这种情况下，指定的选项适用于所有下级动词。或者选项可应用于每个单独的 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");
```

在这种情况下，指定的选项只适用于相关的 verb 子句，覆盖基础部分中的任何设置。

表 4.1 “REST DSL 选项” 总结了 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，可用于将传入消息发送到 Java 对象（以及可选的 unmarshal Java 对象到传出消息）。可以有以下值：**off**（默认）、**auto**、**json**、**xml**、**json_xml**。
consumes （）	**@consumes**	将 verb 子句限制为仅接受 HTTP Request 中指定的互联网介质类型 (MIME 类型)。典型值有：**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 verb 子句。适用于 JMX 管理和工具。
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，这对于为 JMX 管理和其他工具定义非常有用。

Java DSL	XML DSL	描述
method （）	**@method**	指定此 verb 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），此选项指定代表 HTTP Response 消息的 Java 类型。
produces （）	**生成**	将 verb 子句限制为仅在 HTTP 响应中生成指定的互联网介质类型 (MIME 类型)。典型值有：**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），此选项指定代表 HTTP Request 消息的 Java 类型。
VerbURIArgument	**@uri**	指定路径片段或 URI 模板作为操作动词的参数。例如，**get (*VerbURIArgument*)**。
BasePathArgument	**@path**	指定 **rest （）** 关键字(Java DSL)或 **rest** 元素(XML DSL)的基本路径。

4.3. 到 JAVA 对象和从 JAVA 对象进行复制

用于通过 HTTP 传输的 Java 对象

使用 REST 协议的最常见方法是传输消息正文中的 Java Bean 的内容。为了实现此目的，您需要有一个机制来聚合 Java 对象以及适当的数据格式。REST DSL 支持以下数据格式（适合编码 Java 对象）：

JSON

JSON (JavaScript 对象表示法)是一个轻量级数据格式，可轻松映射到 Java 对象或从 Java 对象映射。JSON 语法是紧凑的，易于输入，易于读和写。出于所有这些原因，JSON 已经成为 REST 服务的消息格式。

例如，以下 JSON 代码可以代表具有两个属性字段 id 和 name 的 User bean：

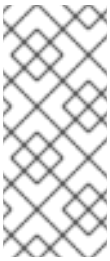
```
{
    "id" : 1234,
    "name" : "Jane Doe"
}
```

JAXB

JAXB （适用于 XML 绑定的 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 汇集到 Java 对象，您还必须注解要使用的 Java 类。

例如，以下 JAXB 代码可以代表具有两个属性字段 id 和 name 的 User bean：

```
<?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，JAXB 数据格式和类型转换器支持从 XML 转换到 POJO，即使用 ObjectFactory 而不是 XmlRootElement。另外，camel 上下文应包含 CamelJaxbObjectFactory 属性，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，以自己将消息正文转换为 Java 对象或从 Java 对象转换。但 REST DSL 提供了自动执行此转换的便利。特别是，将 JSON 和 JAXB 与 REST DSL 集成具有以下优点：

- 自动对 Java 对象和从 Java 对象进行 Marshalling （因为适当的配置）。
- 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，因此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，对应用程序代码的影响最小。

支持的数据格式组件

Apache Camel 提供了 JSON 和 JAXB 数据格式的多种不同的实现。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 中启用对象整理，请观察以下点：

1. 启用绑定模式，通过设置 bindingMode 选项（您可以在多个级别中，您可以设置绑定模式 swig-wagon 以获得详细信息，请参阅 “配置绑定模式”一节）。
2. 使用 type 选项（必需）在传入消息中指定要转换为（或 from）的 Java 类型 （可选），在传出消息中使用 outType 选项（可选）。
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. 使用 jsonDataFormat 选项和/或 xmlDataFormat 选项（可以在 restConfiguration 构建器中指定）指定底层数据格式实现（或实现）。
5. 如果您的路由以 JAXB 格式提供返回值，您通常要将交换正文的 Out 消息设置为带有 JAXB 注释(JAXB 元素)的类实例。但是，如果您想以 XML 格式直接提供 JAXB 返回值，请将

dataFormatProperty 设置为 key xml.out.mustBeJAXBElement，它可在 restConfiguration 构建器上指定。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，您可以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，请记住为您选择的数据格式安装必要的功能。例如，如果您使用 Jackson 数据格式（默认），您可以通过输入以下 Karaf console 命令安装 camel-jackson 功能：**

```
JBossFuse:karaf@root> features:install camel-jackson
```

或者，如果您正在部署到 Fabric 环境中，您可以将该功能添加到 Fabric 配置文件中。例如，如果您使用配置集 MyRestProfile，您可以输入以下 console 命令来添加功能：

```
JBossFuse:karaf@root> fabric:profile-edit --features camel-jackson MyRestProfile
```

配置绑定模式

bindingMode 选项默认为 off，因此您必须明确配置它，以便启用 Java 对象的 marshalling。TABLE 显示支持的绑定模式列表。

注意

从 Camel 2.16.3 开始，只有在 content-type 标头包含 json 或 xml 时才会发生从 POJO 到 JSon/JAXB 的绑定。如果消息正文不应该尝试使用绑定进行 marshalled，则这允许您指定自定义 content-type。例如，当消息正文是自定义二进制有效负载时，这很有用。

表 4.2. REST DSL BInding Modes

绑定模式	描述
off	绑定关闭 **（默认）**。
auto	为 JSON 和/或 XML 启用绑定。在此模式中，Camel 根据传入消息的格式自动选择 JSON 或 XML (JAXB)。**您不需要** 启用两种类型的数据格式，但：一个 JSON 实现、XML 实现或两者都可通过 classpath 提供。
json	仅为 JSON 启用绑定。**必须在** classpath 上提供 JSON 实现（默认情况下，Camel 会尝试启用 **camel-jackson** 实现）。
xml	仅为 XML 启用绑定。**必须在** classpath 上提供 XML 实施（默认情况下，Camel 会尝试启用 **camel-jaxb** 实现）。
json_xml	为 JSON 和 XML 都启用绑定。在此模式中，Camel 根据传入消息的格式自动选择 JSON 或 XML (JAXB)。您需要在 classpath 上提供两种类型的数据格式。

在 Java 中，这些绑定模式值表示为以下 enum 类型的实例：

```
org.apache.camel.model.rest.RestBindingMode
```

您可以在以下位置设置 bindingMode 的不同级别，如下所示：

REST DSL 配置

您可以从 restConfiguration 构建器设置 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 （） 关键字后立即设置 bindingMode 选项（在 verb 子句前面），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("...");
```

Example

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），请查看 Apache Camel camel-example-servlet-rest-blueprint 示例。您可以通过安装独立 Apache Camel 分发 apache-camel-2.23.2.fuse-7_13_0-00013-redhat-00001.zip 来查找此示例，该版本在 Fuse 安装的 extras/ 子目录中提供。

安装独立 Apache Camel 发行版本后，您可以在以下目录下找到示例代码：

```
ApacheCamelInstallDir/examples/camel-example-servlet-rest-blueprint
```

将 Servlet 组件配置为 REST 实施

在 camel-example-servlet-rest-blueprint 示例中，REST DSL 的底层实现由 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，您需要配置一个由以下三个层组成的堆栈：

REST DSL 层

REST DSL 层由 restConfiguration 元素配置，后者通过将 component 属性设置为 servlet 来与 Servlet 组件 集成。

Servlet 组件层

Servlet 组件层作为类实例 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 HTTP 容器(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 org.osgi.service.http.HttpService OSGi 服务的 OSGi 参考，此服务是标准的 OSGi 接口，可在 OSGi 中提供对默认 HTTP 服务器的访问权限。**

b. **创建 utility 类(OsgiServletRegisterer)的实例，以在 HTTP 容器中注册 Servlet 组件。OsgiServletRegisterer 类是一个简化管理 Servlet 组件的生命周期的工具。创建此类实例时，它会自动调用 HttpService OSGi 服务上的 registerServlet 方法；当实例被销毁时，它会自动调用未注册的方法。**

所需的依赖项

这个示例有两个依赖项，它们对 REST DSL 很重要，如下所示：

Servlet 组件

提供 REST DSL 的底层实施。这是在 Maven POM 文件中指定的，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
```

Jacks data format

提供 JSON 数据格式实施。这是在 Maven POM 文件中指定的，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
```

响应的 Java 类型

example 应用在 HTTP Request 和 Response 消息中返回 User type 对象。用户 Java 类在 例 4.2 “JSON 响应的用户类” 中定义。

例 4.2. 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 类采用 JSON 数据格式具有相对简单的表示。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}

获取由 {id} 标识的用户的详细信息，其中以 JSON 格式返回 HTTP 响应。

PUT /camel-example-servlet-rest-blueprint/rest/user

创建新用户，其中用户详细信息包含在 PUT 消息的正文中，以 JSON 格式编码（以匹配 User 对象类型）。

GET /camel-example-servlet-rest-blueprint/rest/user/findAll

获取所有用户的详细信息，其中 HTTP 响应以 JSON 格式返回为用户数组。

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 组合在一起。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，该 URL 的构建如下：

http://localhost:8181

在 restConfiguration 中，协议默认为 http，端口被明确设置为 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，可以在命令行中输入以下命令来使用 curl 工具调用此 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，将 REST DSL 配置为使用 Servlet 组件作为底层实施：

```
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，将 REST DSL 配置为使用 Servlet 组件作为底层实施：

```
<?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或自定义实例的 bean ID。如果没有指定这个选项，Camel 会在 classpath 或 bean registry 中查找 **RestConsumerFactory** 实例。
scheme()	**@scheme**	用于公开 REST 服务的协议。依赖于底层 REST 实施，但通常支持 **http** 和 **https**。默认为 **http**。

Java DSL	XML DSL	描述
host()	**@host**	用于公开 REST 服务的主机名。
port()	**@port**	用于公开 REST 服务的主机名。 **注：** 此设置被 Servlet 组件**忽略**，后者使用容器的标准 HTTP 端口。对于 Apache Karaf OSGi 容器，标准 HTTP 端口通常为 8181。最好为 JMX 和工具设置端口值。
contextPath()	**@contextPath**	为 REST 服务设置前导上下文路径。这可以与 Servlet 等组件一起使用，如 Servlet，其中部署的 Web 应用程序使用 **context-path** 设置进行部署。
hostNameResolver()	**@hostNameResolver**	如果没有显式设置主机名，此解析器会决定 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；而 **RestHostNameResolver.localIp** (Java DSL)或 **localIp** (XML DSL)，它解析为点十进制 IP 地址格式。在 Camel 2.17 **RestHostNameResolver.allLocalIp** 中，可用于解析到所有本地 IP 地址。 默认为 **localHostName**，最多 Camel 2.16。从 Camel 2.17 中，默认值为 **allLocalIp**。
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**	如果有自定义 HTTP 错误代码标头，则指定是否跳过输出绑定。这可让您构建不绑定到 JSON 或 XML 的自定义错误消息，否则成功信息会这样做。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 用来实现 XML 数据格式的组件。可能的值有：**jaxb**。默认为 **jaxb**。
componentProperty()	**componentProperty**	可让您在底层 REST 实现上设置任意 **组件级别** 属性。
endpointProperty()	**endpointProperty**	可让您在底层 REST 实现上设置任意 **端点级别** 属性。
consumerProperty()	**consumerProperty**	允许您在底层 REST 实现中设置任意 **消费者端点** 属性。
dataFormatProperty()	**dataFormatProperty**	可让您对底层数据格式组件（如 Jackson 或 JAXB）设置任意属性。从 Camel 2.14.1 开始，您可以将以下前缀附加到属性键中： • **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)和特定消息方向 (**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（可选）覆盖默认设置。

表 4.4. 默认 CORS 标头

标头键	标头值
access-Control-Allow-Origin	*****
access-Control-Allow-Methods	**GET,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_UNKNOWN_PROPERTIES** 功能（这会导致 **Jackson** 具有无法映射到 **Java** 对象的属性）：*

```
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");
```

*下面是一个示例，它演示了如何在 **Java DSL** 中禁用和启用 **Jackson JSON** 功能：*

```
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");
```

下面是一个示例，它演示了如何在 XML DSL 中禁用和启用 Jackson JSON 功能：

```
<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。为此，请使用带有 camel-openapi-java 模块的 Camel REST DSL，该模块纯纯基于 Java。camel-openapi-java 模块会创建一个 servlet，它与 CamelContext 集成，并从每个 REST 端点拉取信息，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，请调用 apiContextPath （） 来为 OpenAPI 生成的 API 文档设置上下文路径。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

下表中描述的选项可让您配置 OpenAPI 模块。设置选项，如下所示：

- 如果您使用 camel-openapi-java 模块作为 servlet，通过更新 web.xml 文件并为您要设置的每个配置选项指定一个 init-param 元素来设置选项。

- 如果您使用 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	用于与 API 相关的电子邮件地址。
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 以获得更多联系信息。
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 会输出 root 路径中的 **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，请将 REST 上下文路径附加到 **CamelContext** ID 中，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	过滤的 CamelContext ID 出现在上下文列表中的模式。您可以指定正则表达式，并使用 * 作为通配符。这与 Camel Intercept 功能使用的模式匹配工具相同。
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	设置可用于生成文档的 REST API 的路径，例如 **/api-docs**。指定相对路径。不要指定如 **http** 或 **https**。**camel-openapi-java** 模块在运行时计算绝对路径，格式为：**protocol://host:port/context-path/api-path**。
api.termsOfService	字符串	API 服务术语的 URL。
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。

选项	类型	描述
base.path	字符串	必需。设置 REST 服务可用的路径。指定相对路径。也就是说，不要指定 **http** 或 **https**。**camel-openapi-java** modul 以以下格式在运行时计算绝对路径：**protocol://host:port/context-path/base.path**。
CORS	布尔值	是否启用 HTTP 访问控制 (CORS)。这只启用 CORS 以查看 REST API 文档，而不要访问 REST 服务。默认值为 false。建议是使用 **CorsFilter** 选项，如此表后所述。
主机	字符串	设置运行 OpenAPI 服务的主机的名称。默认是根据 **localhost** 计算主机名。
schemes	字符串	要使用的协议方案。使用逗号分隔多个值，例如 **"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。要指定您想要的输出，请将 /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 模块会检查 HTTP Accept 标头来检测是否可以接受 JSON 或 YAML。如果两者都被接受，或者没有设置为接受，则 JSON 是默认返回格式。

例子

在 Apache Camel 3.x 分发中，camel-example-openapi-cdi 和 camel-example-openapi-java 演示了 camel-openapi-java 模块的使用。

在 Apache Camel 2.x 发行版中，camel-example-swagger-cdi 和 camel-example-swagger-java 演示 camel-swagger-java 模块的使用。

增强 OpenAPI 生成的文档

从 Camel 3.1 开始，您可以通过定义名称、描述、数据类型、参数类型等参数详情来增强 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 path 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 中的相同示例：

```
.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 body 的参数，那么您还必须将 body 指定为该参数的类型。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 中的相同示例：

```
.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 examples/camel-example-servlet-rest-tomcat。**

第 5 章 消息传递系统

摘要

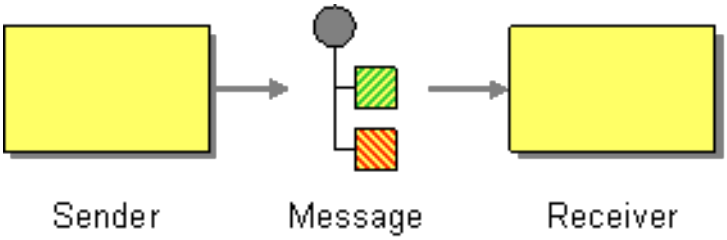
本章介绍了消息传递系统的基本构建块，如端点、消息频道和消息路由器。

5.1. 消息

概述

消息 是在消息传递系统中传输数据的最小单元（由下图中的间候点代表）。消息本身可能有一些内部结构是，例如，一个包含多个部分的一条 message to the geometrical 图，它由附加到 图 5.1 “Message Pattern” 中的间候点表示。

图 5.1. Message Pattern



消息类型

Apache Camel 定义以下不同的消息类型：

- 在消息 wagon-xdg A 消息中，消息通过从消费者端点到生成者端点的路由传输（通常是启动消息交换）。
- 出 消息 HEKETI-2 消息，其通过从制作者端点的路由传输回消费者端点（通常，响应 In 消息）。

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 会将以下结构应用到所有消息类型：

- 标头 wagon-wagon 包含从消息中提取的元数据或标头数据。

- 正文 wagon- swig 通常以其原始形式包含整个消息。

- 附加信息 附加（需要与某些消息传递系统集成，如 JBI）。

务必要记住，此部门位于标头、正文和附件中，这是消息的抽象模型。Apache Camel 支持许多不同的组件，它们生成各种消息格式。最后，它是底层组件实施，决定进入消息的标头中和正文。

协调消息

在内部，Apache Camel 记住消息 ID，用于关联单个消息。然而，在实践中，Apache Camel 与消息相关的最重要的方法是通过 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称为 消息交换，以及管理消息序列的规则称为 交换模式。例如，两种常见的交换模式是：单向事件消息（协调 In 消息）和 request-reply 交换（协调 In 消息，后跟 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 （） HEKETI-wagon return the named 标头和当前 In 消息的正文。

- outBody （） wagon-wagon 返回当前 Out 消息的正文。

例如，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);
```

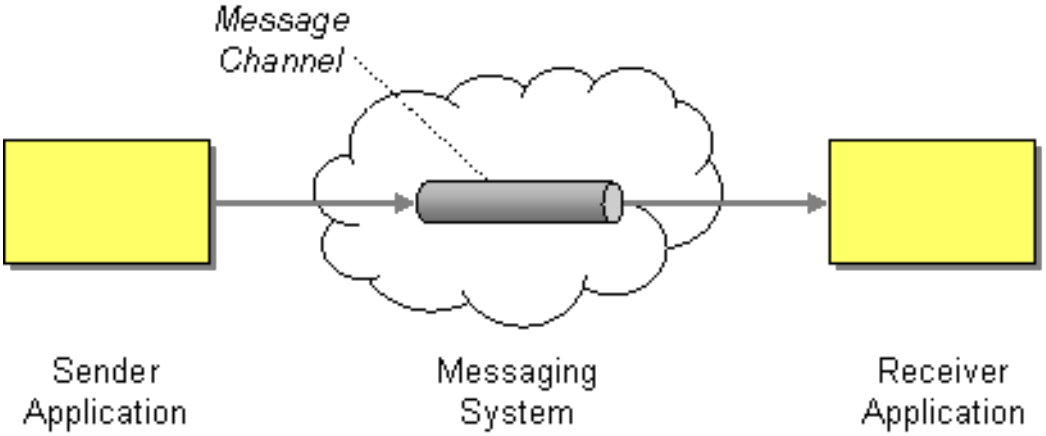
5.2. 消息频道

概述

消息频道 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频道提供了将消息排序到不同的消息类型的元素。消息队列和消息主题是消息通道的示例。您应该记住逻辑通道与物理通道不同。有几种不同的方法可以物理实现逻辑频道。

在 Apache Camel 中，消息频道由面向消息的组件的端点 URI 代表，如 图 5.2 “消息频道模式” 所示。

图 5.2. 消息频道模式



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概念：

- *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 具有以下格式：

```
activemq:QueueName
```

特定主题的 TopicName 的端点 URI 具有以下格式：

```
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
```

如需了解更多详细信息，以及设置 ActiveMQ 组件的说明，请参阅 Apache Camel 组件参考指南 中的 ActiveMQ。

JMS

Java 消息传递服务(JMS)是一种通用打包程序层，用于访问许多不同类型的消息系统（例如，您可以使用它来包装 ActiveMQ、S MQ 系列、Tibco、BEA、S Sonic 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 具有以下格式：

```
jms:QueueName
```

特定主题的 TopicName 的端点 URI 具有以下格式：

```
jms:topic:TopicName
```

如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 Jms。

AMQP

在 AMQP 中，消息频道由队列或主题表示。特定队列 QueueName 的端点 URI 具有以下格式：

```
amqp:QueueName
```

特定主题的 TopicName 的端点 URI 具有以下格式：

```
amqp:topic:TopicName
```

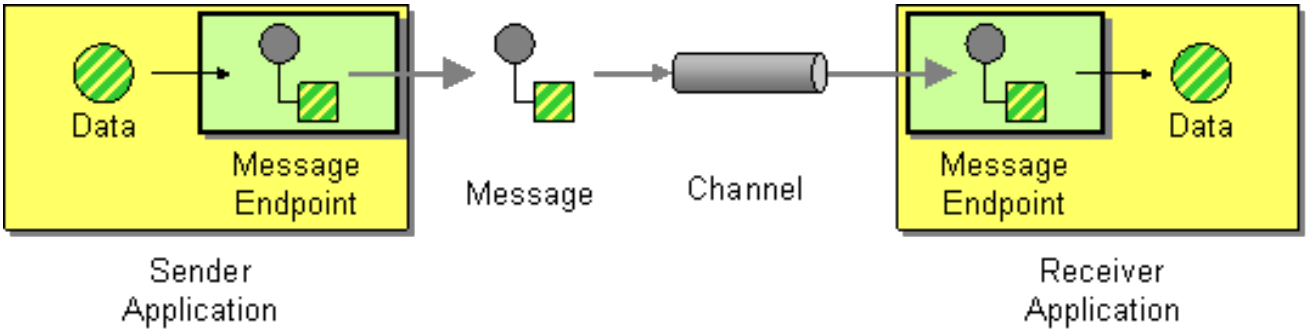
有关设置 AMQP 组件的详情，请参阅 Apache Camel 组件参考指南 中的 Amqp。

5.3. 消息端点

概述

消息端点 是应用程序和消息传递系统之间的接口。如 图 5.3 “消息端点模式” 所示，您可以有一个发送者端点，有时被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，它负责 接收信息。

图 5.3. 消息端点模式



端点类型

Apache Camel 定义两种基本端点类型：

- Apache Camel 路由开始时的消费者 端点 HEKETI Appears，并从传入频道读取 In 消息（等同于 接收器 端点）。
- 在 Apache Camel 路由结束时生成者 端点 HEKETI Appears，并将 In 消息写入传出频道（等同于 发送者 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 表示，它通常封装以下类型的数据：

- 消费者端点 wagon-wagon Advertises 的端点 URI （例如，要公开发送者可以连接到的服务）。或者，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。
-

生成者端点 wagon-wagon 包含发送消息的位置的端点 URI，并包含用于配置端点的设置。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，目标可以有一个抽象形式，如队列名称。

Apache Camel 中的端点 URI 具有以下通用形式：

```
ComponentPrefix:ComponentSpecificURI
```

其中 ComponentPrefix 是用于标识特定 Apache Camel 组件的 URI 前缀（请参阅 Apache Camel 组件参考 以了解所有受支持组件的详细信息）。URI 组件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，您可以定义如下所示的端点 URI：

```
jms:Foo.Bar
```

要定义将消费者端点 file://local/router/messages/foo 连接到制作者端点 jms:Foo.Bar 的路由，您可以使用以下 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 > 参数允许您使用连接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。以下示例将消息发送到标头定义的端点：

```
<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 也可以以文字作为前缀，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 mock:orange。

要使用 Simple 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 predicates 语言”。

使用不同语言的格式是使用 URI 中的 语言：languagename :。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 中的相同示例：

```
from("direct:start")
  .toD("language:xpath:/order/@uri");
```

如果没有指定 language:，则端点是一个组件名称。在某些情况下，一个组件和语言具有相同的名称，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，因此无需定义语言。+ 符号是 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");
```

许多语言可以一次串联，每个语言都与 + 分开，并使用 语言名称指定各个语言。

以下选项可通过 toD 提供：

Name	默认值	描述
uri		必需：要使用的 URI。
pattern		设置一个特定的 Exchange Pattern，以便在发送到端点时使用。之后会恢复原始 MEP。
cacheSize		配置 **ProducerCache** 的缓存大小，它将缓存生成者以供重复使用。默认缓存大小为 1000，如果没有指定其他值，则会使用它。将值设为 -1 可完全关闭缓存。
ignoreInvalidEndpoint	**false**	指定是否忽略无法解析的端点 URI。如果禁用，Camel 将抛出一个识别无效端点 URI 的异常。

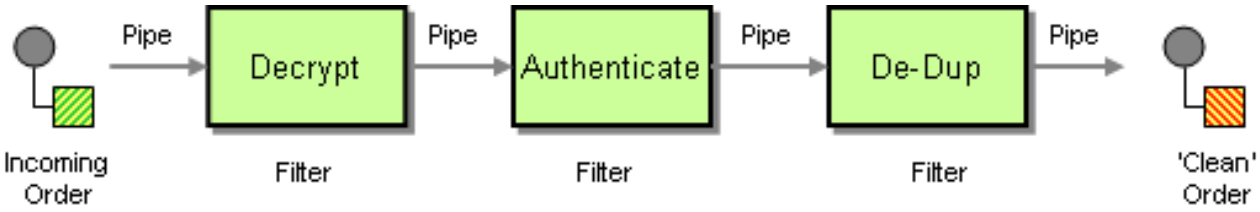
5.4. 管道和过滤器

概述

图 5.4 “管道和过滤器模式” 中显示的 管道和过滤器 模式描述了通过创建过滤器链来构建路由的方法，其中一个过滤器的输出会被放入管道中下一过滤器的输入中（类似于 UNIX pipe 命令）。管道方法的优点

是它允许您编写服务（某些服务可以是 Apache Camel 应用程序外部）以创建更复杂的消息处理形式。

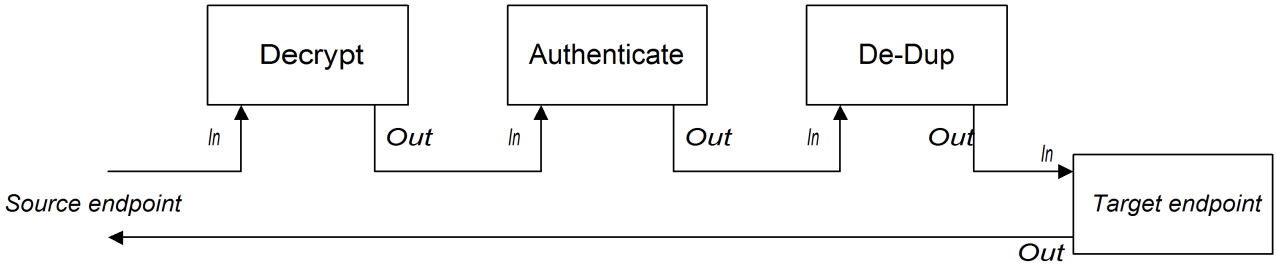
图 5.4. 管道和过滤器模式



InOut Exchange pattern 的管道

通常，管道中的所有端点都有一个输入（消息）和一个输出(Out 消息)，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 消息将发回到原始的调用者。您可以定义此管道的路由，如下所示：

```
from("jms:RawOrders").pipeline("cxf:bean:decrypt", "cxf:bean:authenticate", "cxf:bean:dedup",
"jms:CleanOrders");
```

可以在 XML 中配置相同的路由，如下所示：

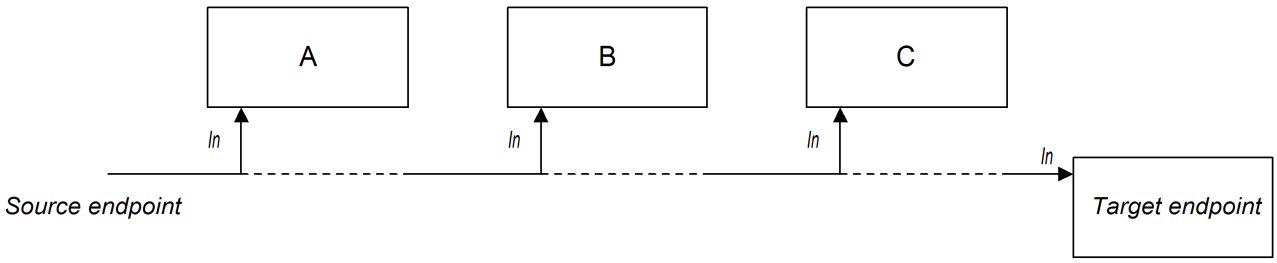
```
<camelContext id="buildPipeline" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:RawOrders"/>
    <to uri="cxf:bean:decrypt"/>
    <to uri="cxf:bean:authenticate"/>
    <to uri="cxf:bean:dedup"/>
    <to uri="jms:CleanOrders"/>
  </route>
</camelContext>
```

XML 中没有专用的 pipeline 元素。从 和 到 元素的前导组合与管道具有意义。请参阅 “pipeline （） 和 to （） DSL 命令的比较”一节。

InOnly 和 RobustInOnly 交换模式的管道

当管道中没有来自端点的 Out 消息时（因为 InOnly 和 RobustInOnly Exchange 模式的情况），管道无法正常连接。在这个特殊情况下，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。这种类型的管道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 中)相同的语法来定义。

pipeline （）和 to （）DSL 命令的比较

在 Java DSL 中，您可以使用以下任何一种语法定义管道路由：

- 使用 pipeline （）processor 命令 wagon-wagon 使用管道处理器来构建管道路由，如下所示：

```
from(SourceURI).pipeline(FilterA, FilterB, TargetURI);
```

- 使用 to （）命令，使用 to （） 命令构建管道路由，如下所示：

```
from(SourceURI).to(FilterA, FilterB, TargetURI);
```

另外，您可以使用等效的语法：

```
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 相当于一个管道处理器。在 Java DSL 中，可通过路由中的上一命令修改 to （） 的含义。例如，当 multicast （） 命令在 to （） 命令前，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”）。

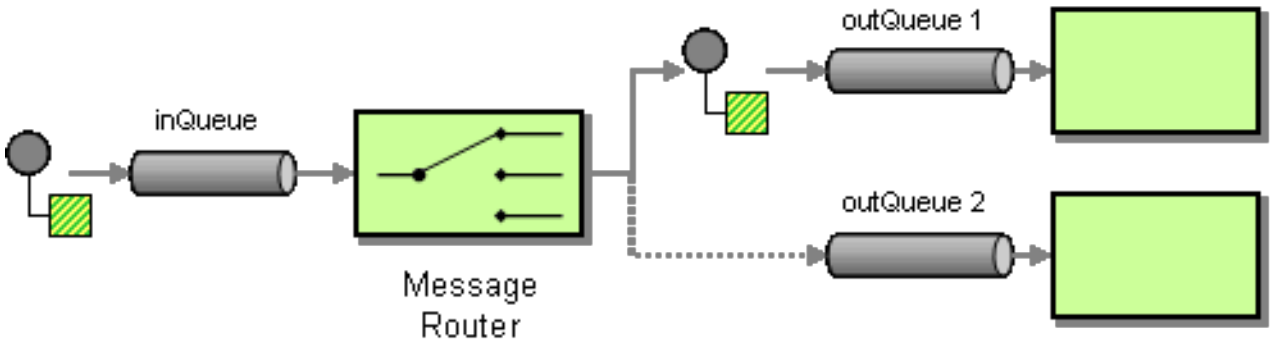
5.5. 消息路由器

概述

图 5.7 “消息路由器模式” 中显示的消息路由器是一个过滤器，它消耗来自单个消费者端点的消息，并根据具体决策条件将它们重定向到适当的目标端点。消息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，每当 Camel 将消息交换路由到接收者端点时，它发送的是原始交换对象的 shouldow副本。在 shouldow 副本中，原始交换的元素（如消息正文、标头和附件）仅通过引用来复制。通过发送可重复利用资源的复制，Camel 会优化性能。但是，因为这些应该式副本都是链接的，因此在 Camel 将消息路由到多个端点时，代价是您丢失了将自定义逻辑应用到路由到不同收件人的副本的能力。有关如何启用 Camel 将消息的唯一版本路由到不同的端点的详情，请参考 "应用自定义处理到传出消息"。

图 5.7. 消息路由器模式



可以使用 choice （） 处理器在 Apache Camel 中轻松实现消息路由器，其中每个替代目标端点都可以使用 when （） 子使用（有关所选处理器的详情，请参阅 第 1.5 节 “Processors”）。

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 或 seda:c) ：

```
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

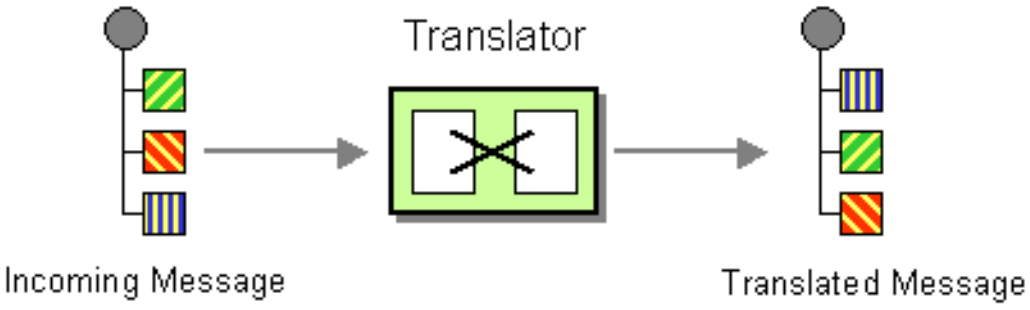
如果您使用不带 otherwise （） 子句的 choice （），则默认丢弃任何不匹配的交换。

5.6. 消息 TRANSLATOR

概述

图 5.8 “消息转换模式” 中显示的消息转换器模式描述了一个组件，它修改消息的内容，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. 消息转换模式



Bean 集成

您可以使用 bean 集成转换消息，这可让您调用任何注册的 Bean 的方法。例如，要在 ID 为 myTransformerBean 上调用方法 myMethodName （），请执行以下操作：

```
from("activemq:SomeQueue")
  .beanRef("myTransformerBean", "myMethodName")
  .to("mqseries:AnotherQueue");
```

其中，myTransformerBean Bean 在 Spring XML 文件或 JNDI 中定义。如果忽略来自 beanRef（） 的 method name 参数，bean 集成将尝试通过检查消息交换来调用的方法名称。

您还可以添加您自己的显式处理器 实例 来执行转换，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 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 从一个目的地消耗消息，将其转换为 Velocity 或 XQuery，然后将其发送到另一个目的地。例如，使用 InOnly Exchange 模式（单向消息传递）：

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

如果要使用 InOut (request-reply)语义处理具有模板生成的响应的 My.Queue 队列上的请求，您可以使用类似如下的路由将响应发送回 JMSReplyTo 目的地：

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

5.7. 消息历史记录

概述

Message History 模式允许您分析和调试松散耦合系统中的消息流。如果您将消息历史记录附加到消息，它会显示消息自起通过的所有应用程序的列表。

在 Apache Camel 中，使用 getTracedRouteNodes 方法，您可以使用来自 UnitOfWork 的 Java API 跟踪消息流。

在日志中限制字符长度

当您使用日志记录机制运行 Apache Camel 时，它可让您记录消息及其内容。

有些消息可能包含非常大的有效负载。默认情况下，Apache Camel 将忽略日志消息，仅显示前 1000 个字符。例如，它显示以下日志：

```
[DEBUG ProducerCache  - >>>> Endpoint[direct:start] Exchange[Message:
01234567890123456789... [Body clipped after 20 characters, total length is 1000]
```

当 Apache Camel clips 日志中正文时，您可以自定义限制。您还可以设置零或负值，如 -1，表示消息正文不会被记录。

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

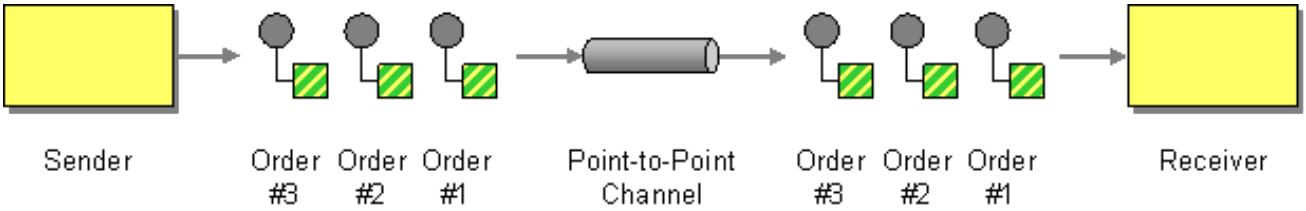
消息传递通道为消息传递应用程序提供热情。本章论述了消息传递系统中可用的不同类型的消息传递频道，以及它们所扮演的角色。

6.1. POINT-TO-POINT 频道

概述

图 6.1 “指向点频道模式” 中显示的 点对点频道 是一个 消息频道，它保证只有一个接收器消耗任何给定信息。这与 发布 订阅频道不同，它允许多个接收器使用相同的消息。特别是，使用发布订阅频道，多个接收器可以订阅同一频道。如果多个接收器竞争使用消息，则会最多消息频道，以确保只有一个接收器实际消耗消息。

图 6.1. 指向点频道模式



支持点对点频道的组件

以下 Apache Camel 组件支持点对点频道模式：

- JMS
- ActiveMQ
- SEDA
- JPA
- XMPP

JMS

在 JMS 中，点对点频道由 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，点对点通道由队列表示。例如，您可以指定名为 Foo.Bar 的 ActiveMQ 队列的端点 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 Event-Driven 架构(SEDA)组件使用阻塞队列来实施。如果要创建一个 Apache Camel 应用程序 内部 的轻量级点对点频道，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 (criu)组件是一个 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

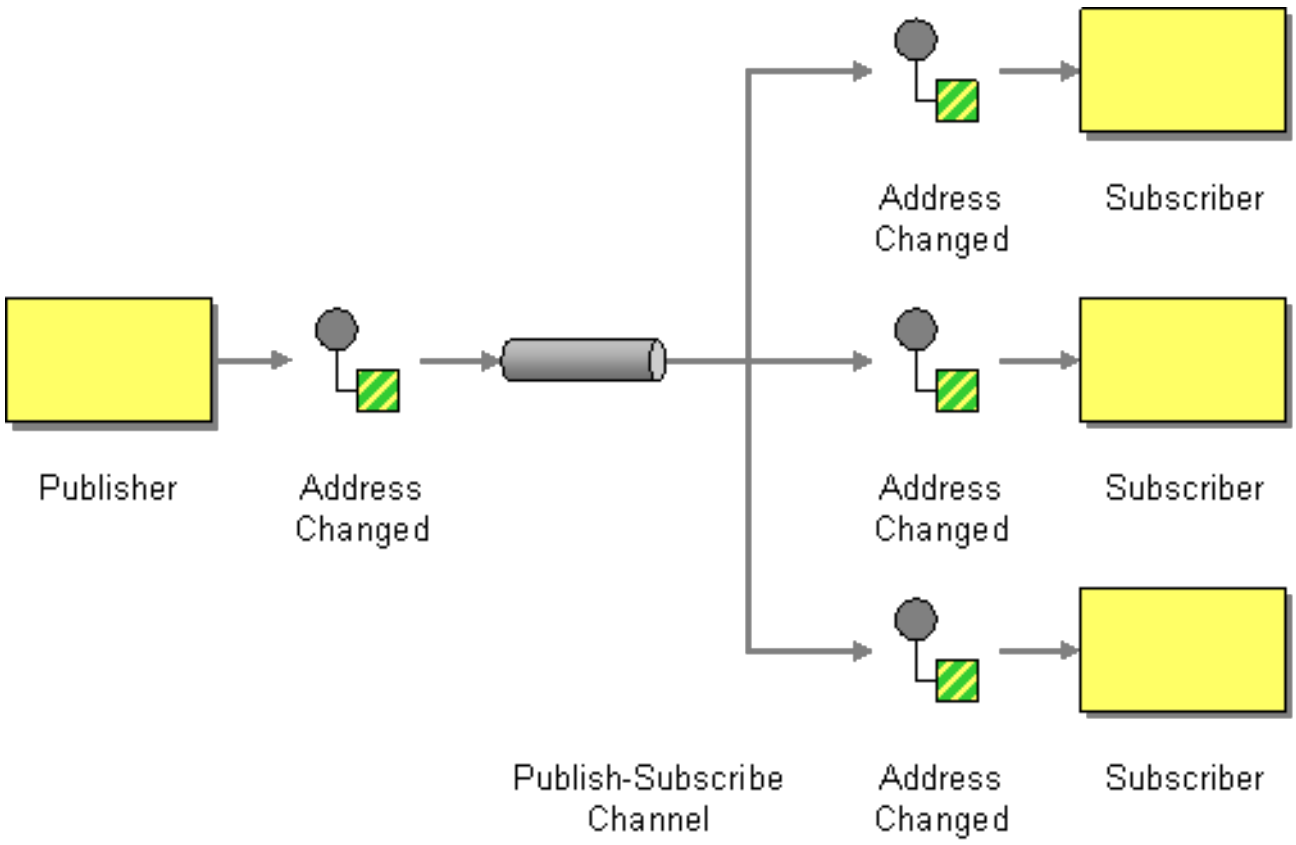
XMPP (Jabber)组件支持点对点频道模式，在人类通信模式中使用时。如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 XMPP。

6.2. PUBLISH-SUBSCRIBE CHANNEL

概述

发布订阅频道 （如 图 6.2 “发布订阅频道模式” 所示）是一个 第 5.2 节 “消息频道”，它可让多个订阅者消耗任何给定消息。这与 第 6.1 节 “point-to-Point 频道” 相反。发布订阅通道经常用作向多个订阅者广播事件或通知的方法。

图 6.2. 发布订阅频道模式



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
- SEDA 用于处理同一 CamelContext 中的 SEDA，可在 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，但在同一 JVM 中使用。

JMS

在 JMS 中，发布订阅通道由 主题 表示。例如，您可以指定名为 StockQuotes 的 JMS 主题的端点 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，发布订阅通道由主题表示。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发布订阅频道模式。如需了解更多详细信息，请参阅 Apache Camel 组件参考指南 中的 Xmpp。

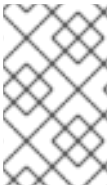
静态订阅列表

如果您愿意，您也可以在 Apache Camel 应用程序本身中实施发布订阅逻辑。简单的方法是定义一个静态订阅列表，其中目标端点在路由末尾都明确列出。但是，这种方法不像 JMS 或 ActiveMQ 主题而灵活。

Java DSL 示例

以下 Java DSL 示例演示了如何使用单个发布程序 seda:a，以及三个订阅者，seda:b、seda:c、seda:c 和 seda:d 模拟发布订阅频道：

```
from("seda:a").to("seda:b", "seda:c", "seda:d");
```



注意

这仅适用于 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：

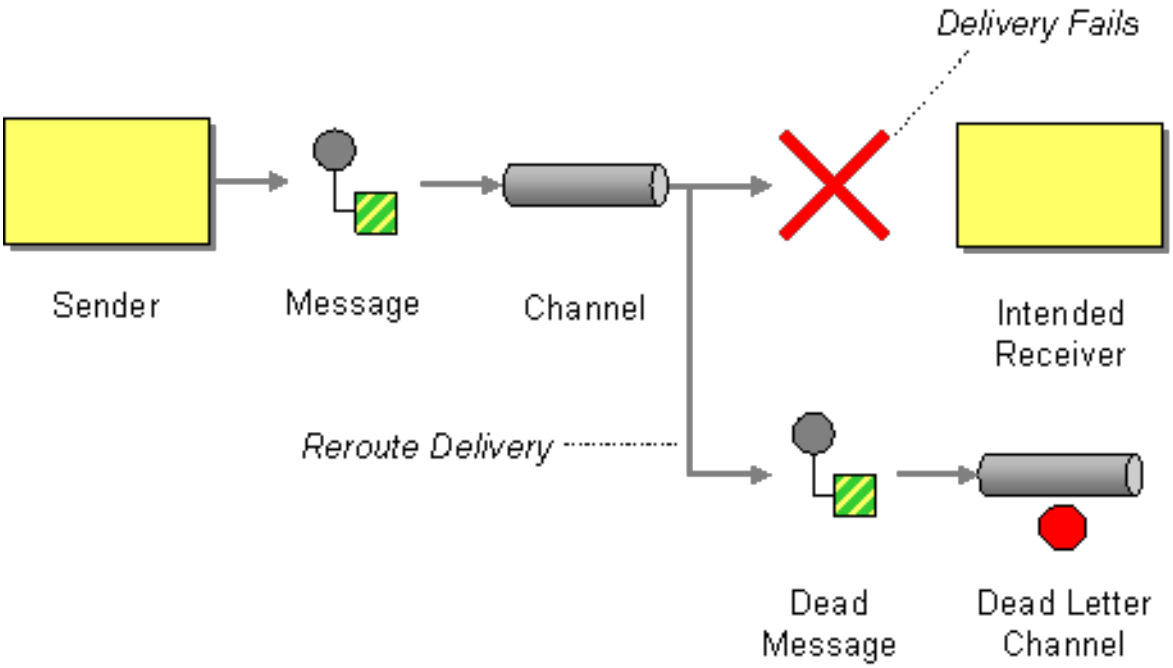
```
<camelContext id="buildStaticRecipientList" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

6.3. 死信频道

概述

图 6.3 “死信频道模式” 中显示的 死信频道 模式描述了在消息传递系统无法向预期接收方发送消息时要执行的操作。这包括重试发送的功能，如果发送最终失败，将消息发送到死信频道，后者会归档未发送的消息。

图 6.3. 死信频道模式



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 Java DSL 拦截器，这表示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，它创建一个带有指定目标端点 seda:errors 的新死信通道。

errorHandler （） 拦截器提供了一个 catch-all 机制 来处理所有 错误类型。如果要应用更精细的方法来异常处理，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>
```

重新发送策略

通常，如果发送尝试失败，则不会向死信频道直接发送邮件。相反，您可以重新尝试发送最多一些最大限制，并在所有重新发送尝试失败后，您可以将消息发送到 dead letter 频道。若要自定义消息重新发送，您可以将死信频道配置为具有重新发送 策略。例如，要指定最多两个重新发送尝试，并将 exponential backoff 算法应用到发送尝试之间的时间延迟，您可以按照以下方式配置 dead letter 频道：

```
errorHandler(deadLetterChannel("seda:errors").maximumRedeliveries(2).useExponentialBackOff());
from("seda:a").to("seda:b");
```

如果您在链中调用相关方法（链中的每个方法会返回对当前 RedeliveryPolicy 对象的引用），在死信频道上设置重新传送选项。表 6.1 “重新发送策略设置” 总结可用于设置重新传送策略的方法。

表 6.1. 重新发送策略设置

方法签名	default	描述
allowRedeliveryWhileStopping()	**true**	控制在安全关闭期间还是在路由停止期间尝试重新发送。在停止停止时已正在进行的交付不会中断。
backOffMultiplier (double multiplier)	**2**	如果启用了 exponential backoff，让 **m** 成为 backoff 倍数，让**d** 成为初始延迟。然后，重新发送尝试的顺序会按照如下所示进行时间： d, m*d, m*m*d, m*m*m*d, ...
conflictAvoidancePercent (double conflictAvoidancePercent)	**15**	如果启用了冲突避免，请给 **p** 避免冲突。冲突避免策略随后将下一个延迟调整到随机数量，最多调整其当前值的加号/减 **p%**。

方法签名	default	描述
deadLetterHandleNewException	**true**	Camel 2.15：指定是否在死信频道中处理消息时是否发生异常。如果为 **true**，则处理异常并在 WARN 级别记录（这样可保证死信频道可以完成）。如果为 **false**，则不会处理异常，因此死信频道会失败，并传播新的异常。
delayPattern (String delayPattern)	None	Apache Camel 2.0：请参阅 "redeliver delay 模式"一节。
disableRedelivery()	**true**	Apache Camel 2.0：禁用重新传送功能。要启用重新发送，请将 **maximumRedeliveries （）** 设置为正整数值。
handled (boolean handled)	**true**	Apache Camel 2.0：如果为 **true**，当消息移到 dead letter 频道时，当前异常会被清除；如果为 **false**，则异常会被传播到客户端。
initialRedeliveryDelay (long initialRedeliveryDelay)	**1000**	指定第一次重新发送前的延迟（以毫秒为单位）。
logNewException	**true**	指定是否在死信频道中引发异常时记录 WARN 级别。
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 （整数）	**0**	Apache Camel 2.0：最大发送尝试数。
maximumRedeliveryDelay (long maxDelay)	**60000**	Apache Camel 2.0：在使用 exponential backoff 策略时（请参阅 **ExponentialBackOff （）**），在理论上可以重新发送延迟以无限增加。此属性对重新发送延迟施加上限（以毫秒为单位）
onRedelivery （进程处理器）	None	Apache Camel 2.0：配置在每次重新发送尝试前调用的处理器。
redeliveryDelay (long int)	**0**	Apache Camel 2.0：指定重新发送尝试之间的延迟（以毫秒为单位）。Apache Camel 2.16.0 ：默认重新传送延迟为 1 秒。

方法签名	default	描述
retriesExhaustedLogLevel(LoggingLevel logLevel)	**LoggingLevel.ERROR**	Apache Camel 2.0：指定日志交付失败的日志级别（指定为 **org.apache.camel.LoggingLevel** 常）。
retryAttemptedLogLevel (LoggingLevel logLevel)	**LoggingLevel.DEBUG**	Apache Camel 2.0：指定重新传送尝试的日志级别（指定为 **org.apache.camel.LoggingLevel** 常）。
useCollisionAvoidance()	**false**	启用冲突避免，这会在 backoff 计时添加一些随机化以减少竞争概率。
useOriginalMessage()	**false**	Apache Camel 2.0：如果启用了此功能，则发送到死信频道的消息是 **原始消息** 交换的副本，因为它存在于路由开始时（在 **from** （） 节点）。
useExponentialBackOff()	**false**	启用指数避退。

重新发送标头

如果 Apache Camel 尝试重新设计信息，它会自动设置 In 消息 表 6.2 “死信重新发送标头” 中描述的标头。

表 6.2. 死信重新发送标头

标头名称	类型	描述
CamelRedeliveryCounter	**整数**	Apache Camel 2.0：计算发送尝试失败次数。此值也在 **Exchange.REDELIVERY_COUNTER** 中设置。
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。此值也在 **Exchange.REDELIVERED** 中设置。

标头名称	类型	描述
CamelRedeliveryMaxCounter	**整数**	Apache Camel 2.6：保存最大重新传送设置（也在 **Exchange.REDELIVERY_MAX_COUNTER** 交换属性中设置）。如果使用 **retryWhile** 或配置了无限的最大重新发送，则缺少此标头。

重新发送交换属性

如果 Apache Camel 尝试重新设计信息，它会自动设置 表 6.3 “重新发送交换属性” 中描述的交换属性。

表 6.3. 重新发送交换属性

Exchange 属性名称	类型	描述
Exchange.FAILURE_ROUTE_ID	**字符串**	提供失败的路由 ID。此属性的字面名称为 **CamelFailureRouteId**。

使用原始消息

从 Apache Camel 2.0 开始，由于交换对象在通过路由时进行修改，因此当引发异常时，创建的交换不一定要存储在死信频道中的副本。在很多情况下，最好在路由开始时记录到达路由的消息，然后再受到路由的任何转换。例如，请考虑以下路由：

```
from("jms:queue:order:input")
      .to("bean:validateOrder");
      .to("bean:transformOrder")
      .to("bean:handleOrder");
```

前面的路由侦听传入的 JMS 消息，然后使用 beans 序列处理消息：validateOrder、transformOrder，并处理Order。但是，当发生错误时，我们不知道消息所处的状态。在 transformOrder bean 或 after 之前发生了错误？通过启用 useOriginalMessage 选项，可以确保来自 jms:queue:order:input 的原始消息记录到 dead letter 频道，如下所示：

```
// will use original body
errorHandler(deadLetterChannel("jms:queue:dead")
      .useOriginalMessage().maximumRedeliveries(5).redeliveryDelay(5000);
```

redeliver delay 模式

从 Apache Camel 2.0 开始，可以使用 delayPattern 选项指定重新发送计数的特定范围的延迟。延迟模式具有以下语法： limit1:delay1;limit2:delay2;limit3:delay3;...，其中每个 delayN applied to redeliveries in the range limitN rhacm redeliveryCount < limitN+1

例如，考虑模式 5:1000;10:5000;20:20000，它定义了三个组，并导致以下重新发送延迟：

- 尝试号 1..4 = 0 毫秒（因为第一个组以 5 开始）。
- 尝试编号 5..9 = 1000 毫秒（第一个组）。
- Try number 10.19 = 5000 毫秒（第二个组）。
- Try number 20.. = 20000 毫秒（最后一个组）。

您可以使用限制 1 启动组，以定义启动延迟。例如，1:1000;5:5000 会产生以下重新发送延迟：

- Try number 1.4 = 1000 millis （第一个组）
- Try number 5.. = 5000 millis （最后一个组）

不要求下一个延迟应高于上一个延迟，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 秒的延迟开始，然后将延迟减少为 1 秒。

哪个端点失败？

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的最新目的地的 URI：

```
// Java
String lastEndpointUri = exchange.getProperty(Exchange.TO_ENDPOINT, String.class);
```

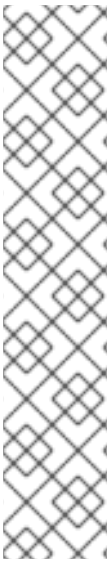
其中 Exchange.TO_ENDPOINT 是与 CamelToEndpoint 相等的字符串常量。每当 Camel 发送消息

到任何端点时，此属性都会更新。

如果在路由期间发生错误，并且交换移到死信队列中，Apache Camel 还将设置一个名为 CamelFailureEndpoint 的属性，该属性标识交换在发生错误之前发送到的最后一个目的地。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);
```

其中 Exchange.FAILURE_ENDPOINT 是一个字符串常量等于 CamelFailureEndpoint。



注意

这些属性保留在当前交换中，即使给定目标端点完成处理后发生失败。例如，请考虑以下路由：

```
from("activemq:queue:foo")
.to("http://someserver/somepath")
.beanRef("foo");
```

现在假设 foo bean 中发生了故障。在这种情况下，Exchange.TO_ENDPOINT 属性和 Exchange.FAILURE_ENDPOINT 属性仍然包含该值。

onRedelivery 处理器

当死信频道执行红色时，可以配置仅在每次重新发送尝试前执行的处理器。这可用于在某些情况下，您需要在消息被重新设计前更改消息。

例如，以下死信频道配置为在重新设计交换前调用 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));
```

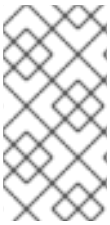
其中 MyRedeliveryProcessor 进程实现，如下所示：

```
// 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);
    }
}
```

控制在关闭或停止过程中重新发送

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是继续尝试重新发送。由于这通常是所需的行为，您可以选择在关闭或停止期间禁用重新发送，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

对于向后兼容的原因，allowRedeliveryWhileStopping 选项默认为 true。但是，在积极关闭过程中，无论此选项设置如何（例如，在安全关闭超时后）总是被禁止重新传送。

使用 onExceptionOccurred Processor

死信频道支持 onExceptionOccurred 处理器，以便在出现异常后自定义处理消息。您还可以使用它来自定义日志记录。从 onExceptionOccurred 处理器抛出的新异常都会记录为 WARN 并忽略，而不是覆盖现有的异常。

onRedelivery processor 和 onExceptionOccurred 处理器之间的差别是，您可以在重新传送尝试前完全处理前者的处理。但是，它不会在异常发生后立即发生。例如，如果您将错误处理程序配置为在重新

发送尝试之间执行五秒延迟，则稍后重新发送处理器会在出现异常后调用 5 秒。

以下示例解释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

您可以在路由构建器中使用 errorHandler （） 拦截器，而是定义一系列 onException （） 子句，以定义不同的重新传送策略以及各种异常类型的不同死信频道。例如，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");
```

其中，通过串联重新传送策略方法（在 表 6.1 “重新发送策略设置”中列出的）来指定重新发送选项，并使用 to （） DSL 命令指定 dead letter 频道的端点。您还可以在 onException （） 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 异常类型特别配置。所有其他例外类型都由通用例外 异常拦截器处理。默认情况下，Apache Camel 应用最接近抛出异常的异常拦截器。如果找不到完全匹配，它将尝试匹配最接近的基础类型，以此类推。最后，如果没有其他拦截器匹配，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 选项来允许自定义处理器准备交换。它可让您添加关于交换的信息，如交换失败的原因。例如，以下处理器添加了一个带有例外消息的标头。

```
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());
    }
}
```

您可以按照如下所示，配置错误处理程序以使用处理器：

```
errorHandler(deadLetterChannel("jms:dead").onPrepareFailure(new MyPrepareProcessor()));
```

但是，onPrepare 选项也可以使用默认错误处理程序。

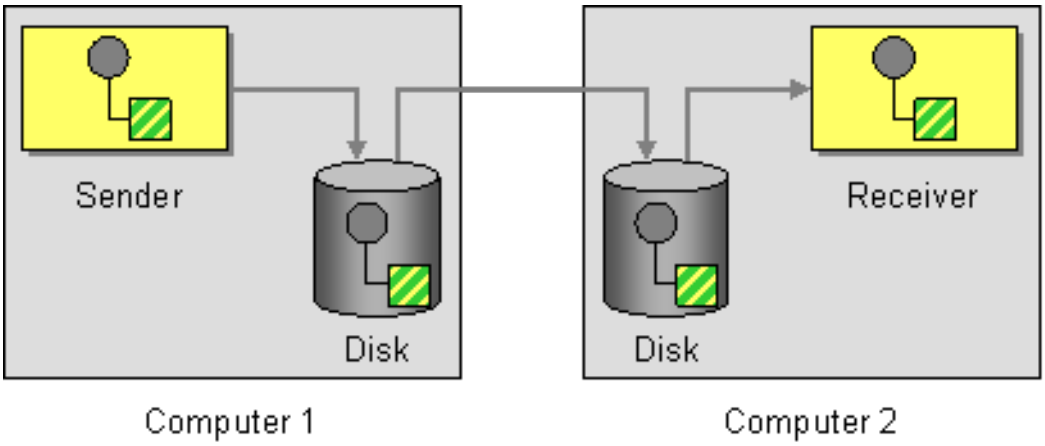
```
<bean id="myPrepare"
class="org.apache.camel.processor.DeadLetterChannelOnPrepareTest.MyPrepareProcessor"/>

<errorHandler id="dlc" type="DeadLetterChannel" deadLetterUri="jms:dead"
onPrepareFailureRef="myPrepare"/>
```

6.4. GUARANTEED DELIVERY

概述

保证发送 意味着，当消息被放入消息频道后，消息传递系统会保证消息将到达其目的地，即使应用程序的部分应该失败。通常，消息传递系统在试图将它们传送到目的地前，通过将信息写入持久性存储来实现保证的交付模式。

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 query 选项指示是否启用消息的持久存储。通常，设置这个选项是不必要的，因为默认行为是启用永久发送。若要配置保证发送的所有详细信息，需要对 JMS 提供程序设置配置选项。这些详细信息会根据您使用的 JMS 提供程序而有所不同。例如： MQ 系列、TibCo、BEA、S Sonic 等，它们都提供各种服务质量来支持保证的交付。

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。在 ActiveMQ 中，您可以使用几种不同的方法来布放消息持久性。

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，可以保证向消费者发送。例如，在 Apache Camel 配置文件中 META-INF/spring/camel-context.xml，您可以将 ActiveMQ 组件配置为使用 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>
```

如果您希望实现在发送到远程端点（与 图 6.4 “保证交付模式” 相似）前保存消息的架构，您可以在 Apache Camel 应用程序中实例化嵌入的代理来实现这个目标。实现此目的的一个简单方法是使用 ActiveMQ Peer-to-Peer 协议，它隐式创建一个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 中的所有对等点，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 是嵌入式代理的代理名称（组中的其他对等点应该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。这使得在广域网络（以及一些未启用 IP 多播的局域网中）使用不合适。

在 ActiveMQ 组件中创建嵌入式代理的一种更灵活的方式是利用 ActiveMQ 的虚拟机协议来连接嵌入式代理实例。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 ActiveMQ 文件，用于配置嵌入式代理实例。在 ActiveMQ 配置文件中，您可

以选择启用以下持久性机制之一：

- AMQ persistence （默认） HEKETI A fast 和 reliable 消息存储，这些存储是 ActiveMQ 原生的。详情请查看 amqPersistenceAdapter 和 AMQ Message Store。

- JDBC 持久性 HEKETI - 使用 JDBC 将消息存储在任何兼容 JDBC 的数据库中。详情请参阅 jdbcPersistenceAdapter 和 ActiveMQ Persistence。

- 日志持久性 以便您可以将消息存储在滚动日志文件中的快速持久性机制。详情请参阅 journalPersistenceAdapter 和 ActiveMQ Persistence。

- Kaha persistence 5-4 A persistence 机制专为 ActiveMQ 开发的持久性机制。详情请参阅 kahaPersistenceAdapter 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

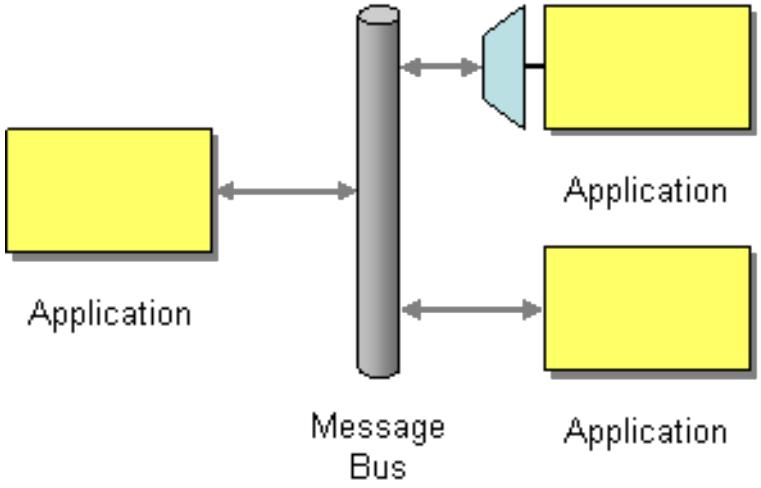
ActiveMQ Journal 组件针对特殊用例进行了优化，其中多个并发生产者将消息写入到队列，但只有一个活跃的消费者。消息存储在滚动日志文件中，并聚合并发写入以提高效率。

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上运行的不同应用程序。实际上，Apache Camel 及其组件构成消息总线。

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：

- *常见通信基础架构 HEKETI-HEKETI 路由器本身提供 Apache Camel 中常见通信基础架构的核心。但是，与某些消息总线架构不同，Apache Camel 提供了异构基础架构：消息可以使用各种不同的传输发送到总线，并使用各种不同的消息格式。*

- *在需要 的情况下，Taxa Camel 可以转换消息格式并使用不同的传输传播信息。实际上，Apache Camel 能够像适配器一样行为，以便外部应用程序可以在不重构其消息传递协议的情况下 hook 进入消息总线。*

  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，其中服务使用 JAX-WS 和 JAXB 映射来实施，可以将各种不同的传输绑定到该服务。这些传输绑定功能作为适配器。

第 7 章 消息结构

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

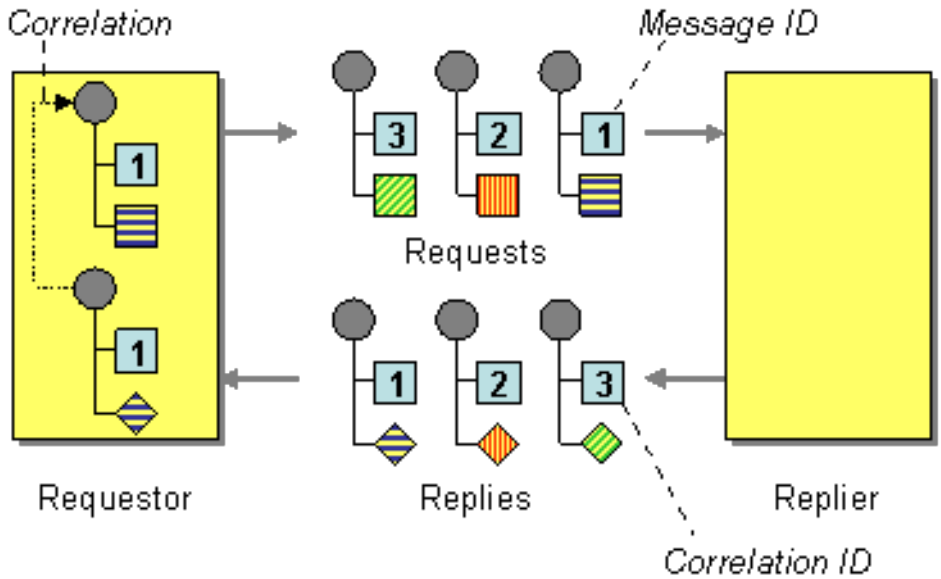
图 7.1 “关联标识符模式” 中显示的 关联标识符 模式描述了如何将回复消息与请求消息匹配，因为异步消息传递系统用于实现请求回复协议。这种理念表明，请求消息应该生成唯一的令牌，请求 ID 则标识请求消息和回复消息应包括令牌、关联 ID，其中包含匹配的请求 ID。

Apache Camel 通过获取或设置消息上的标头来支持来自 EIP 模式的 Correlation 标识符。

使用 ActiveMQ 或 JMS 组件时，关联标识符标头称为 JMSCorrelationID。您可以将自己的关联标识符添加到任何消息交换中，以帮助在单个对话（或业务流程）中关联消息。关联标识符通常存储在 Apache Camel 消息标头中。

有些 EIP 模式会启动子消息，在这种情况下，Apache Camel 将关联 ID 添加到 Exchanges，作为带有其密钥 Exchange.CORRELATION_ID 的属性，它链接到源交换。例如，拆分器、多播、接收者列表和 wire tap EIPs 执行此操作。

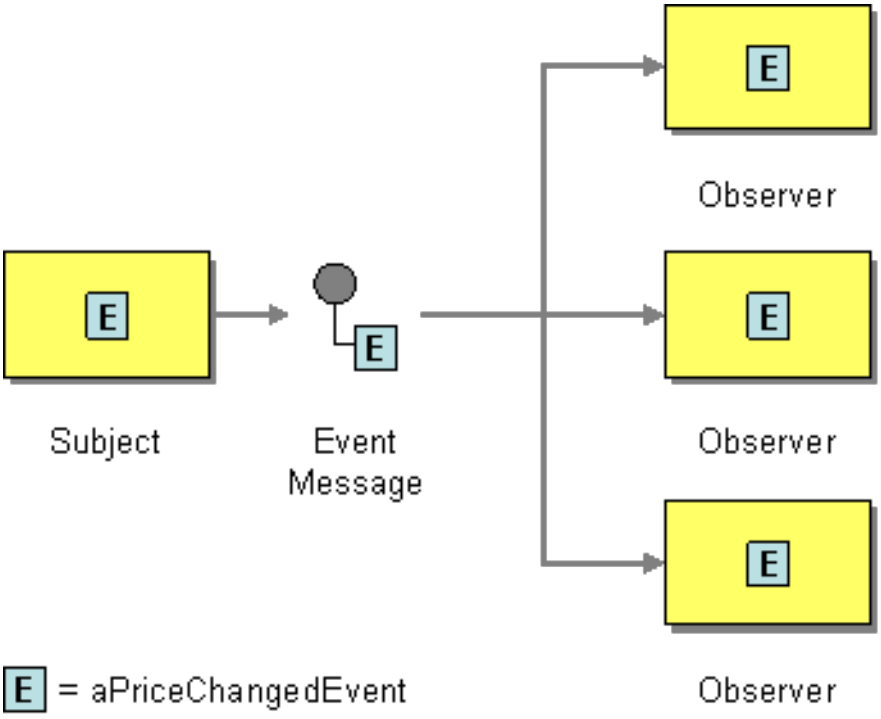
图 7.1. 关联标识符模式



7.2. 事件消息

事件消息

Camel 通过支持 消息 上的 Exchange Patterns 支持企业集成模式 中的事件消息，该消息 可设置为 InOnly 来指示单向事件消息。Camel Apache Camel 组件参考 然后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是 In Only，如 JMS、File 或 SEDA

明确指定 InOnly

如果您使用组件，默认为 InOut，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

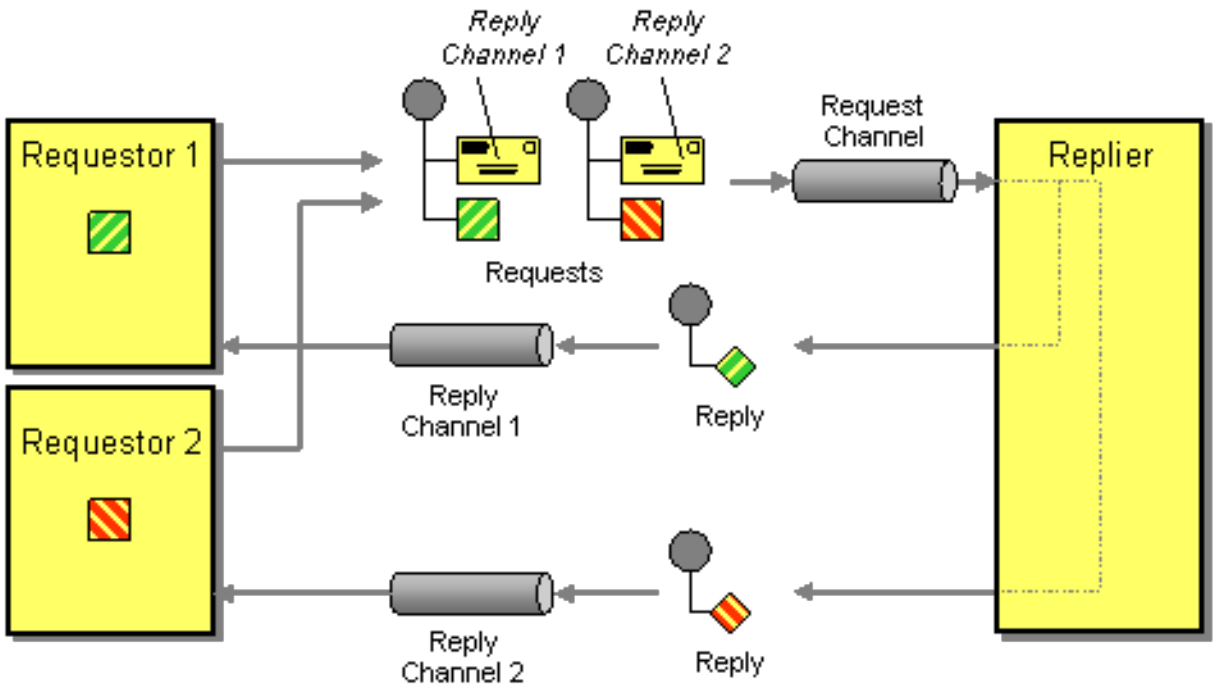
```
<route>
    <from uri="mq:someQueue"/>
    <inOnly uri="bean:foo"/>
</route>
```

```
<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 企业集成模式 返回地址。



例如，在将 JMS 与 InOut 搭配使用时，组件默认将返回到 JMSReplyTo 中给出的地址。

EXAMPLE

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

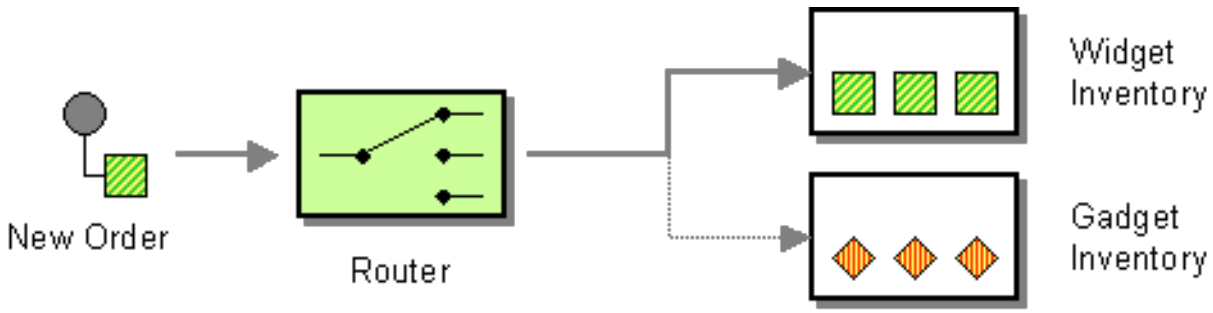
消息路由模式描述了将消息频道链接在一起的各种方法。这包括可应用于消息流的各种算法（无需修改消息正文）。

8.1. 基于内容的路由器

概述

基于内容的路由器 （如 图 8.1 “基于内容的路由器模式” 所示）可让您根据消息内容将消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** 表达式的评估，将来自 **input, seda:a, endpoint** 的请求路由到 **seda:b,queue:c, 或 seda:d** ：

```
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">
```

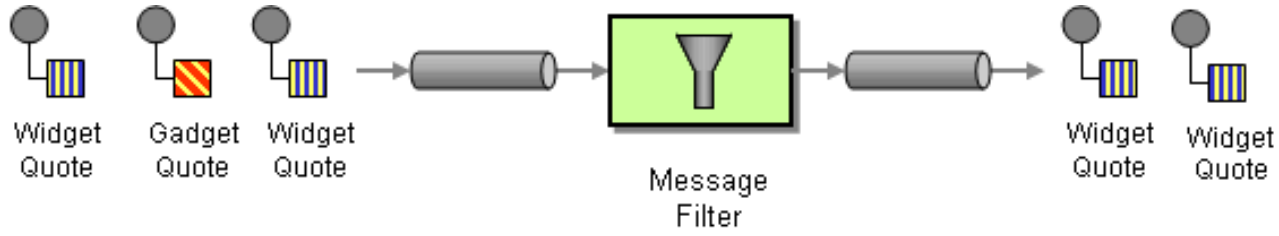
```
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

8.2. MESSAGE FILTER

概述

消息过滤器 是一个处理器，它根据特定标准消除不必要的消息。在 Apache Camel 中，消息过滤器模式（如 图 8.2 “Message Filter Pattern” 所示）由 filter （） Java DSL 命令实现。filter （） 命令使用一个 predicate 参数来控制过滤器。当 predicate 为 true 时，允许传入的消息继续，当 predicate 为 false 时，传入的信息会被阻断。

图 8.2. Message Filter Pattern



Java DSL 示例

以下示例演示了如何创建从端点 seda:a、到端点 seda:b 的路由，它会阻止除 foo 标头具有值 bar 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 predicates 语言”）。以下示例定义了一个路由，它阻止所有消息，除了包含 name 属性等于 basic 的 person 项外的所有消息：

```
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在 XML 中使用 XPath predicate 配置路由（请参阅 第 II 部分 “路由表达式和 predicates 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>
```

在 </FILTER> TAG 中过滤所需的端点

确保将要过滤的端点放在右 </filter> 标签前（例如 & lt;to uri="seda:b"/> ），否则不会应用过滤器（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 （）

从 Camel 2.0 开始提供

stop 是过滤掉 所有消息的 特殊类型的过滤器。当您需要在其中一个 predicates 中停止进一步处理时，停止在 基于内容的路由器 中使用。

在以下示例中，我们不希望消息正文中带有单词 Bye 的消息在路由中传播任何进一步。我们使用 .stop （） 在 when （） predicate 中阻止这种情况。

```
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 还是未过滤

从 Camel 2.5 开始提供

消息过滤器 EIP 将在 Exchange 中添加属性，如果其被过滤或未显示，则状态。

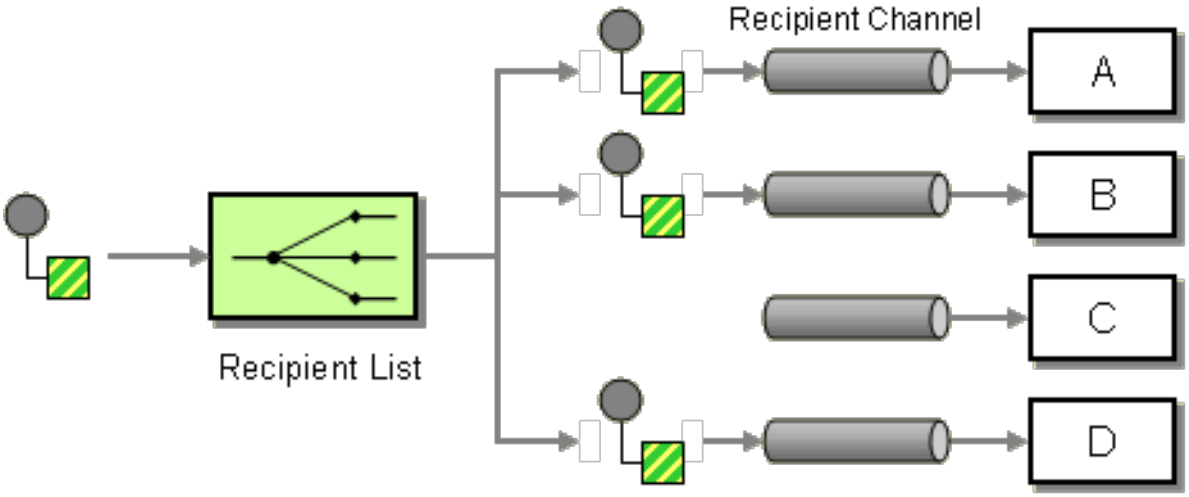
该属性具有主要 Exchange.FILTER_MATCHED，它具有 CamelFilterMatched 的 String 值。其值是一个布尔值，表示 true 或 false。如果值为 true，则在过滤器块中路由 Exchange。

8.3. 接收者列表

概述

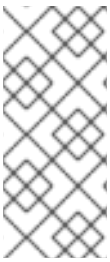
图 8.3 “接收者列表模式” 中显示的 接收者列表 是路由器类型，将每个传入消息发送到多个不同的目的地。此外，接收者列表通常要求在运行时计算接收者列表。

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：预先修复目的地列表并提前已知的，交换模式为 InOnly。在这种情况下，您可以硬化将目的地列表放入 to （） Java DSL 命令中。



注意

此处给出的示例适用于带有固定目的地的接收者列表，仅适用于 InOnly Exchange 模式（与管道 和过滤器模式类似）。如果要为带有 Out 消息的交换模式创建接收者列表，请使用 多播 模式。

Java DSL 示例

以下示例演示了如何将 InOnly Exchange 从消费者端点 queue:a 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>
```

-

在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，该处理器使用目的地列表作为其唯一参数。由于 Apache Camel 将类型转换器应用到列表参数，因此应当可以使用大多数标准 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 “built-In Type Converters”。

接收者收到 同一 交换实例的副本，Apache Camel 按顺序执行它们。

Java DSL 示例

以下示例演示了如何从名为 receiverListHeader 的消息标头中提取目的地列表，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，如果标头值是列表类型，您可以直接使用它作为 recipientList （） 的参数。例如：

```
from("seda:a").recipientList(header("recipientListHeader"));
```

但是，这个示例完全取决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，则此示例 将无法正常工作。标头必须解析为某种类型的 Java 列表。

XML 配置示例

以下示例演示了如何在 XML 中配置上述路由，其中标头值是以逗号分隔的端点 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

从 Camel 2.2 开始提供

接收者列表模式 支持 parallelProcessing，这与 splitter 模式 中的对应 功能类似。使用并行处理功能将交换同时发送到多个接收者，例如：

```
from("direct:a").recipientList(header("myHeader")).parallelProcessing();
```

在 Spring XML 中，并行处理功能是作为 recipientList tag5-4-wagon 等属性实现的：

```
<route>
  <from uri="direct:a"/>
  <recipientList parallelProcessing="true">
    <header>myHeader</header>
  </recipientList>
</route>
```

异常停止

从 Camel 2.2 开始提供

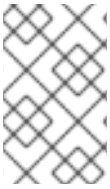
接收者列表 支持 stopOnException 功能，如果任何接收者失败，您可以使用它停止发送到任何进一步的接收者。

```
from("direct:a").recipientList(header("myHeader")).stopOnException();
```

在 Spring XML 中，其接收者列表标签上的属性。

在 Spring XML 中，停止在异常功能上作为属性实现，作为 recipientList tag swig-wagon 等属性：

```
<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 parallelProcessing 和 stopOnException。

忽略无效的端点

从 Camel 2.3 开始提供

接收者列表模式 支持 ignoreInvalidEndpoints 选项，该选项可让接收者列表跳过无效的端点(路由 slips 模式 也支持这个选项)。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>
```

例如，myHeader 包含两个端点 direct:foo,xxx:bar。第一个端点有效且正常工作。第二个无效，因此忽略。每当遇到无效的端点时，Apache Camel 会在 INFO 级别记录。

使用自定义 AggregationStrategy

从 Camel 2.2 开始提供

您可以使用带有 接收者列表模式 的自定义 AggregationStrategy，这对于聚合列表中 来自接收者的回复很有用。默认情况下，Apache Camel 使用 UseLatestAggregationStrategy 聚合策略，仅保留最后收到的回复。对于更复杂的聚合策略，您可以定义您自己的 AggregationStrategy 接口 implementation，详情请参阅 第 8.5 节 “聚合器”。例如，要将自定义聚合策略 MyOwnAggregationStrategy 应用到回复消息，您可以定义 Java DSL 路由，如下所示：

```
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 receiverList 标签上的属性，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

从 Camel 2.2 开始提供

这只在使用并行进程时才需要。默认情况下，Camel 使用有 10 个线程的线程池。请注意，当我们稍后进行过度处理线程池管理和配置时（在 Camel 2.2 中），这可能会有变化。

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 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

您可以通过在返回一组接收者列表的方法中添加 @RecipientList 注释，使 bean 充当接收者列表。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。按如下方式定义路由：

```
from("activemq:queue:test").bean(MessageRouter.class, "routeTo");
```

使用超时

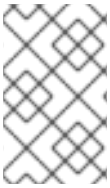
从 **Camel 2.5** 开始提供

如果使用 **parallelProcessing**，您可以以毫秒为单位配置总 超时值。然后，**Camel** 将并行处理消息，直到超时到达为止。如果一个信息缓慢，可以继续处理。

在以下示例中，**receiverlist** 标头的值为 **direct:a,direct:b,direct:c**，以便消息发送到三个接收者。我们有一个 **250** 毫秒的超时时间，这意味着在时间范围内只能完成最后两个消息。因此，聚合会生成字符串结果 **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

splitter 和 multicast 和 recipientList 也支持此 超时 功能。

默认情况下，如果没有调用 AggregationStrategy，则默认超时。但是，您可以实施一个专用版本

```
// 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);
```

如果您真正需要，这可让您处理 AggregationStrategy 中的超时。

超时是总计

超时是 total，这意味着 X 时间后，Camel 将聚合在时间线内完成的消息。剩余部分将被取消。Camel 还将只针对导致 超时 的第一个索引在 TimeoutAwareAggregationStrategy 一次调用超时方法。

将自定义处理应用到传出消息

在 receiverList 发送消息到其中一个接收者端点之前，它会创建一个消息副本，这是原始邮件的粗略副本。在 Show 副本中，原始消息的标头和有效负载仅通过引用复制。每个新副本不包含它们自己的这些元素实例。因此，消息应该会链接，在将自定义处理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其端点之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 命令仅在消息被压缩 后 插入自定义处理器，并在将消息分配给其端点 之前。例如，在以下路由中，每个接收者端点 的消息副本上调用 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一些或所有元素执行深度副本。这样，每个消息副本都可以独立于其他副本进行修改。例如，以下 CustomProc 处理器类执行消息正文的深度副本，其中消息正文假定为类型为 BodyType，而深度副本则由方法 BodyType.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：

Name	默认值	描述
delimiter	,	如果 Expression 返回多个端点，则使用分隔符。
strategyRef		是指用于将来自接收者的回复汇编为 第 8.3 节 “接收者列表” 的单一传出消息的 AggregationStrategy。默认情况下，Camel 将使用最后一个回复作为传出消息。
strategyMethodName		当将 POJO 用作 **AggregationStrategy** 时，可以使用此选项明确指定要使用的方法名称。

strategyMethodAllowNull	**false**	当将 POJO 用作 **AggregationStrategy** 时，可以使用此选项。如果为 **false**， 则不会使用聚合方法，当没有数据增强时。如果为 **true**，则空值用于 **oldExchange**，如果没有数据更丰富，则使用**null** 值。
parallelProcessing	**false**	Camel 2.2： 如果启用然后向接收方发送信息。请注意，调用者线程仍然会等待所有消息被完全处理，然后再继续。它仅发送和处理来自同时发生的接收方的回复。
parallelAggregate	**false**	如果启用，则 **AggregationStrategy** 上的聚合方法可同时调用。**请注意，这需要实施 AggregationStrategy 才能实现 thread-safe。默认情况下，此选项为 false，这意味着 Camel 会自动同步聚合方法的调用。但是，在某些用例中，您可以通过将 AggregationStrategy 设为 thread-safe 来提高性能，并将此选项设置为 true。**
executorServiceRef		Camel 2.2： 请参阅用于并行处理的自定义线程池。请注意，如果您设置了这个选项，则并行处理是自动的，您不必启用该选项。
stopOnException	**false**	Camel 2.2 ： 在出现异常情况时，是否立即停止继续处理。如果禁用，则 Camel 会将消息发送到所有接收者，无论它们是否失败。您可以在 AggregationStrategy 类中处理异常，您可以完全控制如何处理它。
ignoreInvalidEndpoints	**false**	Camel 2.3： 如果无法解析端点 uri，则应忽略它。否则 Camel 将抛出一个例外，表示端点 uri 无效。

streaming	**false**	Camel 2.5： 如果启用，则 Camel 将按其返回的顺序处理回复，例如：如果禁用，Camel 将以与指定的 Expression 相同的顺序处理回复。
timeout		Camel 2.5： 设置在 millis 中指定的总超时。如果 第 8.3 节 “接收者列表” 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 第 8.3 节 “接收者列表” 中断并继续。请注意，如果您提供了一个 AggregationStrategy，则在中断前调用 **timeout** 方法。
onPrepareRef		Camel 2.8： 请参阅自定义处理器来准备每个接收者将收到的 Exchange 副本。这可让您执行任何自定义逻辑，如在需要时深度获取消息有效负载。
shareUnitOfWork	**false**	Camel 2.8： 是否应共享工作单元。详情请查看 第 8.4 节 “Splitter” 中的同一选项。
cacheSize	**0**	Camel 2.13.1/2.12.4 ： 允许为 ProducerCache 配置缓存大小，该缓存生成者可在路由 slip 中重复使用。默认情况下，将使用默认缓存大小为 0。将值设为 -1 允许将缓存全部关闭。

在 Recipient 列表中使用交换模式

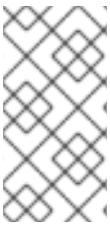
默认情况下，Recipient List 使用当前的交换模式。但是，在有些情况下，您可以使用不同的交换模式向接收者发送消息。

例如，您可能有一个作为 InOnly 路由启动的路由。现在，如果要使用带有接收者列表的 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以 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，您需要使用 exchangePattern 等于 InOut 选项来指定此端点。但是，响应形成 JMS 请求或回复将持续路由，因此响应将作为 outbox 目录中的文件存储在 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

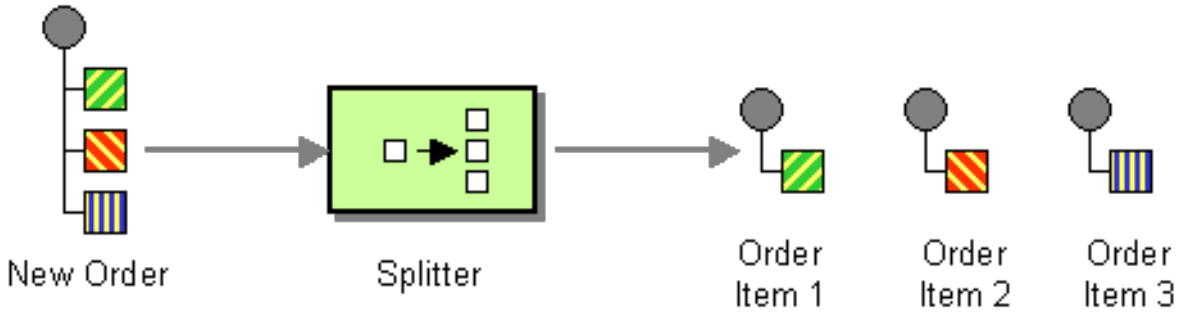
InOut Exchange 模式必须在超时期间获得响应。但是，如果响应没有修复，它会失败。

8.4. SPLITTER

概述

splitter 是将传入消息分成一系列传出消息的路由器类型。每个传出消息都包含一个原始消息。在 Apache Camel 中，图 8.4 “Splitter Pattern” 中显示的 splitter 模式由 split （） Java DSL 命令实现。

图 8.4. Splitter Pattern



Apache Camel 分割程序实际上支持两种模式，如下所示：

- 简单的 splitter HEKETI-wagon 实施其自身的 splitter 模式。
- Splitter/aggregator 5-4-wagoncom 使用聚合器模式来组合分割器模式，以便在消息被处理后被重新发送。

在将原始消息分成多个部分之前，它会制作原始消息的重复副本。在 Show 副本中，原始消息的标头和有效负载将复制为仅引用。虽然分割程序本身不会将生成的消息部分路由到不同的端点，但分割消息的部分可能会处于二级路由中。

由于消息部分是应该有的副本，它们仍与原始消息相关联。因此，无法单独修改它们。如果要在将自定义逻辑路由到一组端点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 选项来进行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

以下示例定义了从 **seda:a** 到 **seda:b** 的路由，该路由通过将传入消息的每一行转换为单独的传出消息来分割消息：

```
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支持的脚本语言（如 **XPath**、**XQuery** 或 **SQL**）分割信息（请参阅 **第 II 部分 “路由表达式和 predicates 语言”**）。以下示例从传入消息中提取 **bar** 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 **splitter** 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** 中的令牌表达式来通过令牌分割正文或标头，其中使用 **tokenize** 元素定义令牌表达式。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** 项中设置 **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>
```

分成行组

要将大型文件分成 1000 行的块，您可以定义一个分割路由，如 Java DSL 所示：

```
from("file:inbox")
    .split().tokenize("\n", 1000).streaming()
        .to("activemq:queue:order");
```

令牌 化的 第二个参数指定应分组为单个块的行数。streaming （） 子句指示分割者不会一次读取整个文件（如果文件较大），从而使整个文件性能显著提高。

可以在 XML DSL 中定义相同的路由，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 选项时的输出始终是 java.lang.String 类型。

跳过第一项

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 true 中进行第三个选项：

```
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");
```

可以在 XML DSL 中定义相同的路由，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

*如果进入 **splitter** 的交换具有 **InOut message-exchange** 模式（即预期回复），则分割器会返回原始输入消息的副本，作为 **Out** 消息插槽中的回复消息。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

*如果要并行执行结果消息，您可以启用 **parallel processing** 选项，该选项可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 **executor**，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

由于 splitter 可以使用任何表达式来执行拆分，因此您可以通过调用 method （） 表达式来使用 bean 执行分割。bean 应该返回可取的值，例如： java.util.Collection、java.util.Iterator 或数组。

以下路由定义了对 mySplitterBean bean 实例调用方法的 method （） 表达式：

```
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，它定义如下：

```
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，以避免将整个内容读到内存中。以下示例演示了如何使用映射文件设置 BeanIOSplitter 对象，该文件从 classpath 加载：

注意

BeanIOSplitter 类在 Camel 2.18 中是新的。Camel 2.17 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程序：

```
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

每个分割交换都设置了以下属性：

header	type	description
CamelSplitIndex	**int**	Apache Camel 2.0：每个被分割的交换增加的分割计数器。计数器从 0 开始。
CamelSplitSize	**int**	Apache Camel 2.0：分割交换的总数。此标头不适用于基于流的分割。
CamelSplitComplete	布尔值	Apache Camel 2.4：此交换是否是最后。

Splitter/aggregator 模式

在处理各个部分完成后，消息组件要聚合到单一交换中是一种常见的模式。要支持此模式，可以使用 split （） DSL 命令提供 AggregationStrategy 对象作为第二个参数。

Java DSL 示例

以下示例演示了如何使用自定义聚合策略在处理所有消息片段后重新发送分割消息：

```
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 实现

上述路由中使用的自定义聚合策略 **MyOrderStrategy** 如下：

```
/**
 * 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

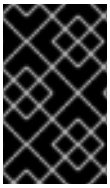
启用并行处理后，理论上有可能在较早出现之前发生的情况之前，后续消息可能准备好进行聚合。换句话说，消息部分可能会到达聚合器没有顺序。默认情况下，这不会发生，因为分割实施将消息片段重新整理回其原始顺序，然后再将它们传递给聚合器。

如果您希望在消息就绪时聚合消息片段（可能按顺序），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");
```

您还可以提供用于流的自定义迭代程序，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")
```



流和 XPATH

您不能与 XPath 结合使用流模式。XPath 需要在内存中的完整 DOM XML 文档。

使用 XML 进行基于流的处理

如果传入的消息是非常大的 XML 文件，您可以在流模式中使用 tokenizeXML 子命令来处理消息。

例如，如果一个含有一系列 order 元素的大型 XML 文件，您可以使用类似如下的路由将文件分成 顺序 元素：

```
from("file:inbox")
  .split().tokenizeXML("order").streaming()
  .to("activemq:queue:order");
```

您可以通过定义类似如下的路由在 XML 中执行同样的操作：

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="order" xml="true"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

通常，您需要访问在令牌元素的 enclosing (ancestor)元素中定义的命名空间。您可以通过指定您要从中继承命名空间定义的元素，将命名空间定义从上级元素复制到 token 元素中。

在 Java DSL 中，您将 ancestor 项指定为 tokenizeXML 的第二个参数。例如，要继承来自 enclosing orders 元素的命名空间定义：

```
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：

Name	默认值	描述
strategyRef		是指用于将子消息中的回复汇编为来自 第 8.4 节 “Splitter” 的单个传出消息的 AggregationStrategy。有关默认使用的内容，请参阅标题为以下的 splitter 返回的内容。
strategyMethodName		当将 POJO 用作 **AggregationStrategy** 时，可以使用此选项明确指定要使用的方法名称。

strategyMethodAllowNull	**false**	当将 POJO 用作 **AggregationStrategy** 时，可以使用此选项。如果为 **false**， 则不会使用聚合方法，当没有数据增强时。如果为 **true**，则空值用于 **oldExchange**，如果没有数据更丰富，则使用**null** 值。
parallelProcessing	**false**	如果启用，则同时处理子消息。请注意，调用者线程仍然会等待所有子消息都完全处理，然后再继续。
parallelAggregate	**false**	如果启用，则 **AggregationStrategy** 上的聚合方法可同时调用。**请注意，这需要实施 AggregationStrategy 才能实现 thread-safe。默认情况下，此选项为 false，这意味着 Camel 会自动同步聚合方法的调用。但是，在某些用例中，您可以通过将 AggregationStrategy 设为 thread-safe 来提高性能，并将此选项设置为 true。**
executorServiceRef		指的是用于并行处理的自定义线程池。请注意，如果您设置了这个选项，则并行处理是自动的，您不必启用该选项。
stopOnException	**false**	Camel 2.2 ： 在出现异常情况时，是否立即停止继续处理。如果禁用，则 Camel 继续分割并处理子消息，无论它们是否失败。您可以在 AggregationStrategy 类中处理异常，您可以完全控制如何处理它。
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信息。这可减少内存开销。例如，如果您分割大型消息，则建议启用流。如果启用了流，则子消息回复将按其顺序聚合，例如按其返回的顺序进行聚合。如果禁用，Camel 将以与分割方式相同的顺序处理子消息回复。

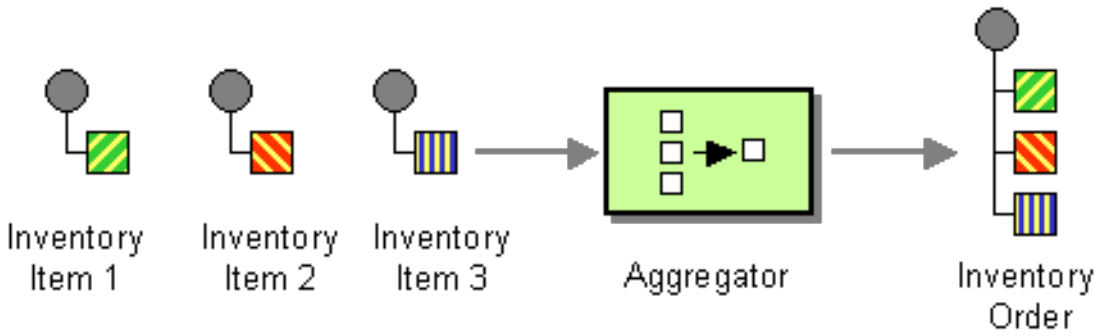
timeout		Camel 2.5：设置在 millis 中指定的总超时。如果 第 8.3 节 “接收者列表” 无法分割和处理给定时间段内的所有回复，则超时触发器和 第 8.4 节 “Splitter” 中断并继续。请注意，如果您提供了一个 AggregationStrategy，则在中断前调用 **timeout** 方法。
onPrepareRef		Camel 2.8：在处理前，请参阅自定义处理器准备交换的子消息。这可让您执行任何自定义逻辑，如在需要时深度获取消息有效负载。
shareUnitOfWork	**false**	Camel 2.8：是否应共享工作单元。详情请查看以下。

8.5. 聚合器

概述

图 8.5 “聚合器模式” 中显示的 聚合器 模式允许您将相关消息批量组合成单个消息。

图 8.5. 聚合器模式



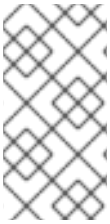
为了控制聚合器的行为，Apache Camel 允许您指定 企业集成模式 中描述的属性，如下所示：

- *关联表达式 HEKETI-是确定应将哪些消息聚合在一起。关联表达式在每个传入消息上评估，以生成 关联键。具有相同关联键的传入消息会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，您可以使用恒定表达式。*

- *完成一批消息时，完整性条件.您可以将它指定为简单大小限制，或者更频繁地指定在批处理完成后标记的 predicate 条件。*

- 将单个关联密钥的消息交换 聚合 聚合到一个消息交换中。

例如，假设一个股票市场数据系统每秒接收 30,000 个消息。如果您的 GUI 工具无法应对这样的大规模更新率，您可能希望减慢消息流。传入的股票引号可以通过选择最新的引号并丢弃旧的价格来聚合。（如果您想捕获某些历史记录，您可以应用 delta 处理算法。）



注意

现在，聚合器使用包括更多信息的 ManagedAggregateProcessorMBean 的 JMX 中的 enlists。它允许您使用聚合控制器来控制它。

聚合器如何工作

图 8.6 “聚合器实施” 显示聚合器的工作方式概述，假设它是与具有关联键（如 A、B、C 或 D）的交换流。

图 8.6. 聚合器实施

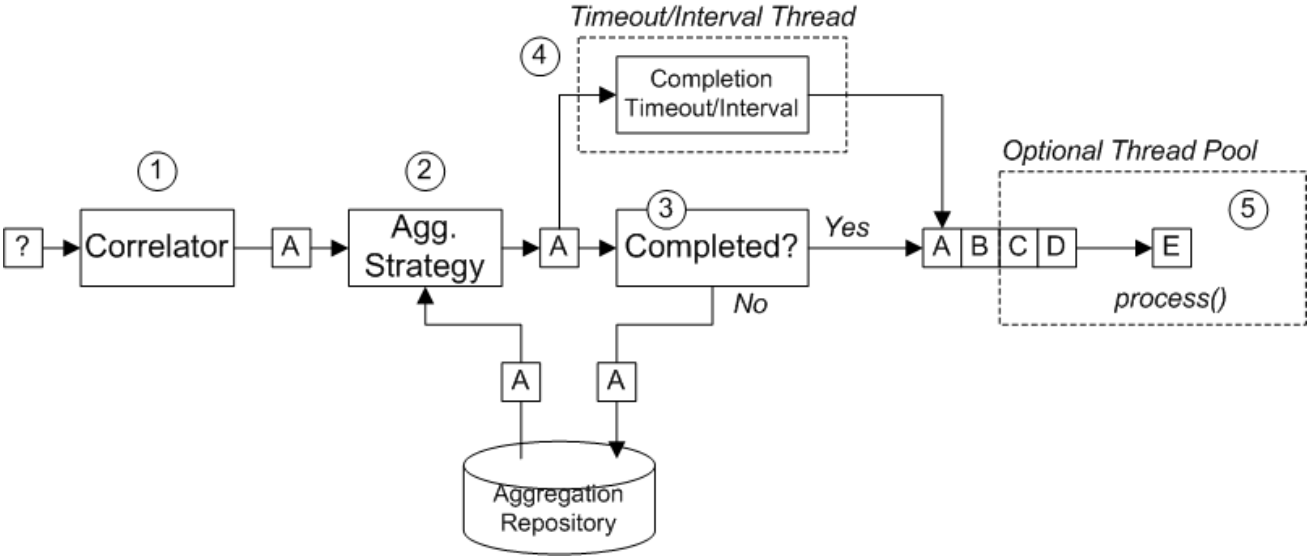


图 8.6 “聚合器实施” 中显示的交换的传入流处理如下：

1. correlator 负责根据关联密钥排序交换。对于每个传入的交换，将评估关联表达式，生成关联密钥。例如，对于 图 8.6 “聚合器实施” 中显示的交换，关联密钥评估为 A。

2. 聚合策略 负责将交换与相同的关联键合并。当一个新交换时，A 会被放入，聚合器会在聚合存储库中查找对应的 聚合交换、A'，并将它与新的交换合并。

在完成了特定的聚合周期前，传入的交换会与相应的聚合交换不断聚合。聚合周期会持续到由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 completion predicate，则会测试聚合交换，以确定它是否已准备好发送到路由中的下一个处理器。处理会继续，如下所示：

   - 如果完成，聚合交换由路由的后方部分处理。这有两种替代模型：同步 （默认），这会导致调用线程阻止或 异步 （如果启用了并行处理），其中聚合交换将提交到 executor 线程池（如 图 8.6 “聚合器实施”所示）。
   - 如果没有完成，聚合交换会保存回聚合存储库。

4. 与同步的完成测试并行，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，每当完成测试满足时，对应的交换都会标记为完成并开始由路由的后部分处理（同步或异步处理，具体取决于是否启用并行处理）。

5. 如果启用了并行处理，线程池负责处理路由后部分的交换。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，聚合具有相同 StockSymbol 标头值的交换。对于给定的 StockSymbol 值，自上次收到关联密钥后的三秒以上，则聚合交换被视为已完成，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。您不限于此处使用简单表达式语言。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，要使用 XPath 表达式来关联交换，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");
```

如果无法在特定的传入交换上评估关联表达式，则聚合器默认抛出 CamelExchangeException。您可以通过设置 ignoreInvalidCorrelationKeys 选项来限制这个异常。例如，在 Java DSL 中：

```
from(...).aggregate(...).ignoreInvalidCorrelationKeys()
```

在 XML DSL 中，您可以将 ignoreInvalidCorrelationKeys 选项设置为属性，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个参数传递给 aggregate （） DSL 命令，或使用 aggregationStrategy （） 子句指定它。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 提供以下基本聚合策略（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，丢弃所有之前使用此密钥的交换。例如，此策略对于从库存交易中节流源非常有用，您只想了解特定股票符号的最新价格。

UseOriginalAggregationStrategy

返回给定关联密钥的第一个交换，并丢弃此密钥的所有后期交换。您必须先调用 UseOriginalAggregationStrategy.setOriginal （） 来设置第一个交换，然后才能使用此策略。

GroupedExchangeAggregationStrategy

将给定关联密钥 的所有 交换连接到列表，该列表存储在 Exchange.GROUPED_EXCHANGE 交换属性中。请参阅 “分组的交换”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本接口之一：

org.apache.camel.processor.aggregate.AggregationStrategy

基本聚合策略接口。

org.apache.camel.processor.aggregate.TimeoutAwareAggregationStrategy

实现此接口，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)
```

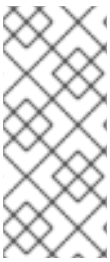
org.apache.camel.processor.aggregate.CompletionAwareAggregationStrategy

如果一个聚合周期可以正常完成时，实施此接口，以接收通知。通知方法有以下签名：

```
void onCompletion(Exchange exchange)
```

例如，以下代码显示了两种不同的自定义聚合策略，即 **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

自 **Apache Camel 2.0** 起，还会为非常第一个交换调用 **AggregationStrategy.aggregate （） 回调方法。在聚合方法的第一次 调用时，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，请定义类似如下的路由：

```
from("direct:start")
    .aggregate(header("StockSymbol"), new ArrayListAggregationStrategy())
    .completionTimeout(3000)
    .to("mock:result");
```

您还可以在 **XML** 中使用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。这对于确保聚合策略可以安全关闭非常有用。

要实施具有生命周期支持的聚合策略，您必须实施 **org.apache.camel.Service** 接口（除了 **AggregationStrategy** 接口），并提供 **start** （） 和 **stop** （） 生命周期方法的实现。例如，以下代码示例显示了带有生命周期支持的聚合策略概述：

```
// 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

在每个聚合交换上设置以下属性：

标头	类型	描述 Aggregated Exchange Properties
Exchange.AGGREGATED_SIZE	**int**	聚合至此交换的交换总数。
Exchange.AGGREGATED_COMPLETED_BY	**字符串**	指明负责完成聚合交换的机制。可能的值有：**predicate**、**size**、**timeout**、**interval** 或 **consumer**。

SQL 组件聚合存储库在交换上设置以下属性（请参阅 "持久性聚合存储库"一节）：

标头	类型	描述 Redelivered 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

必须至少指定一个完成条件，这决定了聚合交换何时保留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

*在聚合每个交换后评估 **predicate**，以确定完整性。值为 **true** 表示聚合交换已完成。另外，您还可以定义实现 **Predicate** 接口的自定义 **AggregationStrategy**，在这种情况下，**AggregationStrategy** 将用作 **completion predicate**。*

completionSize

在聚合指定数量的传入交换后完成聚合交换。

completionTimeout

（与 completionInterval兼容） 在指定的超时时间内没有聚合，则完成聚合交换。

换句话说，超时机制会跟踪 每个 关联键值的超时。在收到带有特定键值的最新交换后，时钟开始勾选。如果在指定超时中 没有收到 具有相同 key 值的另一个交换，则对应的聚合交换会被标记为完成并发送到路由上的下一个节点。

completionInterval

（与 completionTimeout不兼容） 可在每个时间间隔（指定长度）过后 完成所有 未完成的聚合交换。

间隔 不是 为每个聚合交换量身定制的。这种机制会强制完成所有未完成的聚合交换。因此，在某些情况下，此机制可以在启动聚合后立即完成聚合交换。

completionFromBatchConsumer

当与支持 批处理消费者 机制的消费者端点结合使用时，此完成选项会在当前批处理完成后自动找出它从消费者端点接收的信息。请参阅 *“批处理消费者”一节*。

forceCompletionOnStop

启用此选项后，它会在当前路由上下文停止时强制完成所有未完成的聚合交换。

前面的完成条件可以任意组合，但 completionTimeout 和 completionInterval 条件除外，这些条件不能同时启用。当条件组合使用时，常规规则是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，用于决定聚合交换何时完成。评估 predicate 表达式的方法有两种：

- 在最新的聚合交换 swig-wagon 中，这是默认行为。
- 在最新的传入的交换 5-4-wagon 中，当您启用 eagerCheckCompletion 选项时，会选择此行为。

例如，如果您要在每次收到 ALERT 消息时终止库存引号流（如最新传入交换中的 MsgType 标头值所示），您可以定义一个类似于以下内容的路由：

```
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 配置相同的路由：

```
<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"/>
```

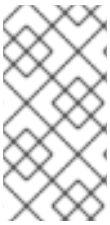
指定动态完成超时

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，要为每个传入交换中的 timeout 标头设置超时值，您可以按照以下方式定义路由：

```
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

您还可以添加固定的超时值，如果动态值为 null 或 0，则 Apache Camel 将回退到使用这个值。

指定动态完成大小

可以指定 动态完成大小，其中为每个传入的交换重新计算完成大小。例如，要在每个传入的交换中设置 mySize 标头的完成大小，您可以定义一个路由，如下所示：

```
from("direct:start")
  .aggregate(header("StockSymbol"), new UseLatestAggregationStrategy())
    .completionSize(header("mySize"))
  .to("mock:aggregated");
```

和使用 Spring XML 的同一示例：

```
<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，如果动态值为 null 或 0， 则 Apache Camel 将回退到使用这个值。

在 AggregationStrategy 中强制完成单个组

如果您实施自定义 AggregationStrategy 类，则有一个机制来强制完成当前消息组，方法是在 AggregationStrategy .aggregate （） 方法返回的交换属性上将 Exchange.AGGREGATION_COMPLETE_CURRENT 交换属性设置为 true。此机制 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。此机制会覆盖任何其他完成机制，如 predicate、size、timeout 等。

例如，如果消息正文大小大于 5，则以下示例 AggregationStrategy 类完成当前的组：

```
// 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;
    }
}
```

使用特殊消息强制完成所有组

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**，以强制完成当前聚合周期。此消息单纯充当信号，不包含在 任何聚合周期中。处理此信号消息后，消息的内容将被丢弃。

Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE

设置为 **true**，以强制完成当前聚合周期。此消息 包含在 当前的聚合周期中。

使用 AggregateController

org.apache.camel.processor.aggregate.AggregateController 可让您使用 **Java** 或 **JMX API** 在运行时控制聚合。这可用于强制完成一组交换，或查询当前的运行时统计信息。

如果没有配置自定义，聚合器会提供默认的实现，您可以使用 **getAggregateController （）** 方法进行访问。但是，使用 **aggregateController** 在路由中轻松配置控制器。

```
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");
```

返回的数字将是已完成的组数。以下是完成所有组的 **API**：

```
 int groups = controller.forceCompletionOfAllGroups();
```

强制唯一关联键

在一些聚合场景中，您可能希望强制条件，使关联键对于每个批处理交换是唯一的。换句话说，当特定关联密钥的聚合交换完成时，您希望确保不允许与该关联键进行进一步聚合的交换。例如，如果路由的后者部分需要处理具有唯一关联键值的交换，您可能希望强制实施此条件。

根据完成条件的配置方式，使用特定的关联键可能会存在多个聚合交换的风险。例如，虽然您可以定义一个完成 **predicate**，它旨在等待收到带有特定关联键 的所有 交换，但您可能也定义完成超时，这可以

在使用该键到达的所有交换前触发。在这种情况下，late-arriving 交换可能会以相同的关联键值增加到 第二个 聚合交换。

在这种情况下，您可以通过设置 closeCorrelationKeyOnCompletion 选项，将聚合器配置为阻止重复关联密钥值的聚合交换。为了抑制重复关联键值，需要聚合器在缓存中记录以前的关联密钥值。此缓存的大小（缓存的关联密钥数量）指定为 closeCorrelationKeyOnCompletion （） DSL 命令的参数。要指定无限大小的缓存，您可以传递值为零或负整数。例如，指定 10000 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

您可以使用 Simple 语言表达式作为令牌，并在流模式下通过 tokenizeXML 子命令使用。使用简单语言表达式将启用对动态令牌的支持。

例如，若要使用 Java 将一系列名称分割为标签 人员，您可以使用令牌化 XML Bean 和简单语言令牌将文件分成 name 元素。

```
public void testTokenizeXMLPairSimple() throws Exception {
      Expression exp = TokenizeLanguage.tokenizeXML("${header.foo}", null);
```

获取由 < person > 划分的名称的输入字符串，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));
    }
```

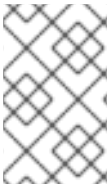
分组的交换

您可以将传出批处理中的所有聚合交换合并到一个 org.apache.camel.impl.GroupedExchange holder 类中。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");
```

发送到 mock:result 的分组交换包含消息正文中的聚合交换列表。以下行的代码显示了后续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

启用分组的交换功能时，不得 配置聚合策略（分组的交换功能本身就是聚合策略）。

注意

从传出交换上的属性访问分组交换的旧方法现已弃用，并将在以后的发行版本中删除。

批处理消费者

聚合器可与 批处理消费者 模式协同工作，以聚合批处理消费者报告的消息总数（批处理消费者端点设置 CamelBatchSize、CamelBatchIndex、CamelBatchIndex 和 CamelBatchComplete 属性）。例如，要聚合 File consumer 端点找到的所有文件，您可以使用类似如下的路由：

```
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：file、FTP、Mail、iBatis 和 JPA。

持久性聚合存储库

默认聚合器仅使用内存的 AggregationRepository。如果要持久存储待处理的聚合交换，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，可持续保留聚合消息，并确保您不会丢失任何消息。

成功处理交换后，当存储库上调用 confirm 方法时，它将标记为 complete。这意味着，如果同一交换再次失败，它将重试，直到成功为止。

添加对 camel-sql 的依赖关系

要使用 SQL 组件，您必须在项目中包含对 camel-sql 的依赖项。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>
```

repositoryName、transactionManager 和 dataSource 属性是必需的。有关持久聚合存储库的更多配置选项的详情，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，聚合器与路由的后方部分分离，其中发送到路由的后一种交换由专用线程池处理。默认情况下，这个池仅包含一个线程。如果要指定具有多个线程的池，请启用 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 parallelProcess 选项）。

聚合到一个列表

常见的聚合方案涉及将一系列传入消息聚合到一个 List 对象中。为了促进这种情况，Apache Camel 提供了 AbstractListAggregationStrategy 抽象类，您可以快速扩展该类来为这种情况创建聚合策略。传入的消息正文类型 T 会被聚合到一个已完成的交换中，消息正文类型为 List<T>。

例如，要将一系列 Integer 消息正文聚合到一个 List<Integer& gt; 对象中，您可以使用定义的聚合策略：

```
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);
    }
}
```

聚合器选项

聚合器支持以下选项：

表 8.1. 聚合器选项

选项	默认	描述
correlationExpression		强制 Expression，用于评估用于聚合的关联键。具有相同关联密钥的 Exchange 聚合在一起。如果无法评估 correlation 键，则抛出一个 Exception。您可以使用 **ignoreBadCorrelationKeys** 选项禁用此功能。
aggregationStrategy		强制 **AggregationStrategy**，用于将传入交换与现有的合并交换合并。**第一次调用 oldExchange 参数为 null。在后续的调用中，oldExchange 包含合并的交换，newExchange 则是新的传入的交换。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，该策略也可以是一个 PreCompletionAwareAggregationStrategy 实施。它在预补模式下运行完成检查。**
strategyRef		在 Registry 中查找 **AggregationStrategy** 的引用。
completionSize		聚合完成前聚合的消息数。这个选项可以被设置为一个固定值，也可以使用表达式来动态评估大小 - 因此将使用 **Integer**。如果两个值都被设置为 **null** 或 **0**， 则 Camel 将回退到使用固定的值。

选项	默认	描述
completionTimeout		聚合交换在完成前应处于非活动状态的时间。这个选项可以被设置为一个固定值，也可以使用表达式来动态评估超时 - 因此将使用 **Long**。如果两个值都被设置为 **null** 或 **0,** 则 Camel 将回退到使用固定的值。您不能将此选项与 completionInterval 一起使用，其中只有一个可用。
completionInterval		millis 中重复的周期，聚合器将完成所有当前的聚合交换。Camel 有一个后台任务，它触发每个周期。您不能将此选项与 completionTimeout 一起使用，只有其中一个选项可以被使用。
completionPredicate		指定 predicate (**org.camel.Predicate** 类型)，它会在聚合交换完成后信号。另外，您还可以定义实现 **Predicate** 接口的自定义 **AggregationStrategy**，在这种情况下，**AggregationStrategy** 将用作 completion predicate。
completionFromBatchConsumer	**false**	如果交换来自 Batch Consumer，则此选项是。当启用 第 8.5 节 “聚合器” 时，将使用消息标头 **CamelBatchSize** 中的 Batch Consumer 决定的批处理大小。请参阅 Batch Consumer 的更多详细信息。这可用于聚合从给定轮询中的 *查看文件* 端点使用的所有文件。

选项	默认	描述
eagerCheckCompletion	**false**	收到新传入的交换时，是否强制检查是否完成。这个选项会影响 **completionPredicate** 选项的行为，因为交换会相应地传递更改。当 Predicate 传递的 Exchange 是 **聚合的** 交换时，这意味着您可以在 **AggregationStrategy** 中的聚合交换上存储的任何信息，供 Predicate 使用。**当 Predicate 中传递的 Exchange 是 传入的交换时，这意味着您可以从传入交换访问数据。**
forceCompletionOnStop	**false**	如果为 **true**，请在当前路由上下文停止后完成所有聚合的交换。
groupExchanges	**false**	如果启用，Camel 会将所有聚合的 Exchange 分组到一个组合的 **org.apache.camel.impl.GroupedExchange** holder 类中，该类包含所有聚合的交换。因此，只会从聚合器发送一个交换。可用于将许多传入的交换合并到一个输出交换中，而无需自行编码自定义 **AggregationStrategy**。
ignoreInvalidCorrelationKeys	**false**	是否忽略无法评估为值的关联键。默认情况下，Camel 将抛出一个例外，但您可以启用这个选项并忽略这种情况。
closeCorrelationKeyOnCompletion		是否应接受 **late** Exchange。您可以启用此选项，以指示 correlation 键是否已完成，然后任何具有相同关联键的新交换都会被拒绝。然后，Camel 会抛出一个 **closedCorrelationKeyException** 异常。当使用这个选项时，您将传递 **一个整数**，这是 LRUCache 的一个数字，这样可保留最后一个 X 闭关联密钥数。您可以传递 0 或负值来指示未绑定的缓存。如果您使用不同的关联密钥的日志，请确保缓存将变得太大。

选项	默认	描述
discardOnCompletionTimeout	**false**	Camel 2.5： 是否应丢弃因为超时而完成的交换。如果启用，则当超时发生超时时，不会发送聚合消息，而是丢弃（丢弃）。
aggregationRepository		允许您自行插入 **org.apache.camel.spi.AggregationRepository** 的实现，它跟踪当前的 inflight 聚合交换。Camel 默认使用基于内存的实现。
aggregationRepositoryRef		引用在 Registry **中查找聚合存储库**。
parallelProcessing	**false**	当聚合完成后，它们会从聚合器中发送。这个选项表示 Camel 是否应该使用带有多个线程的线程池进行并发。如果没有指定自定义线程池，则 Camel 会创建一个具有 10 个并发线程的默认池。
executorService		如果使用 **parallelProcessing**，您可以指定要使用的自定义线程池。实际上，如果您不使用 **并行处理此自定义** 线程池，也用于发送聚合的交换。
executorServiceRef		引用在 Registry 中查找 **executorService**
timeoutCheckerExecutorService		如果使用 **completionTimeout**、**completeTimeoutExpression** 或 **completionInterval** 选项之一，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，以提供要使用的自定义线程池，而不是为每个聚合器创建新线程。
timeoutCheckerExecutorServiceRef		引用在 registry 中查找 **timeoutCheckerExecutorService**。

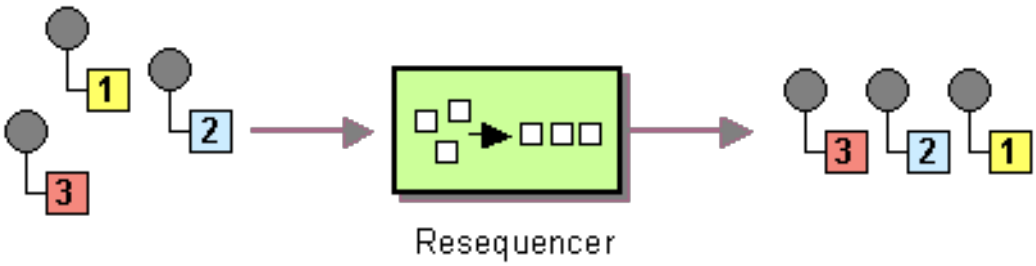
选项	默认	描述
completeAllOnStop		当您停止聚合器时，此选项允许它从聚合存储库完成所有待处理的交换。
optimisticLocking	**false**	打开最佳锁定，可与聚合存储库结合使用。
optimisticLockRetryPolicy		为 optimistic locking 配置重试策略。

8.6. RESEQUENCER

概述

图 8.7 “重新排序器模式” 中显示的 重新排序 器模式允许您根据顺序表达式重新排序信息。为 sequencing 表达式生成低的值的消息被移到批处理的前面，生成高值的消息将移到回来。

图 8.7. 重新排序器模式



Apache Camel 支持两个重新排序算法：

- *批量重新排序 swig -wagon Collects 消息到批处理中，对消息进行排序并将其发送到其输出。*
- *根据消息间差距的检测，流重新排序 datas (continuous)消息流。*

默认情况下，resequencer 不支持重复消息，并将只保留最后一条消息，当消息到达相同的消息表达式时。但是，在批处理模式中，您可以启用 resequencer 来允许重复。

批量重新排序

批处理重新排序算法默认为启用。例如，要根据 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，通过收集到达间隔为 1000 毫秒的所有传入消息（默认 批处理超时），最多获取 100 个消息（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，该命令使用 BatchResequencerConfig 实例作为其唯一参数。例如，要修改前面的路由，以便批处理由 4000 毫秒时间窗内收集的消息组成，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器模式。以下示例定义了一个批处理重新排序器，批处理大小为 300，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 “批处理重新排序器选项” 仅显示批处理模式中可用的选项。

表 8.2. 批处理重新排序器选项

Java DSL	XML DSL	default	描述

Java DSL	XML DSL	default	描述
allowDuplicates （）	**batch-config/@allowDuplicates**	**false**	如果为 **true**，请不要丢弃来自批处理的重复消息（其中 **重复** 意味着消息表达式评估为同一值）。
reverse()	**batch-config/@reverse**	**false**	如果为 **true**，以相反的顺序放置消息（应用于消息表达式的默认排序基于 Java 的字符串字典顺序，如 String.compareTo （）定义）。

例如，如果要根据 JMSPriority 从 JMS 队列重新排序消息，则需要组合选项，允许Duplicates 和 反向，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，要根据 seqnum 标头中的序列号值重新排序传入的信息，您可以定义一个 DSL 路由，如下所示：

```
from("direct:start").resequence(header("seqnum")).stream().to("mock:result");
```

流处理重新排序器算法基于消息流中的差距检测，而不是对固定批处理的大小。差距检测与超时相结合，不再需要提前知道序列（即批处理大小）的消息数量。消息必须包含一个唯一的序列号，即前导者和成功符。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 的后续消息。消息序列 2,3,5 存在一个差距，因为缺少 3 的后续。因此，重新排序必须保留消息 5，直到消息 4 到达（或超时发生）。

默认情况下，流重新排序器配置为 1000 毫秒的超时，最大消息容量为 100。要自定义流的超时和消息容量，您可以传递 StreamResequencerConfig 对象作为 stream （） 的参数。例如，要配置一个流重新排序器，消息容量为 5000，超时为 4000 毫秒，您需要定义一个路由，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");
    }
};
```

如果消息流中连续消息（即带有相邻序列号的消息）之间的最大时间延迟已知，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消息都会以正确的顺序发送到下一个处理器。与顺序时间差异相比的超时值较低，而重新排序器可能会从序列中传递信息。大型超时值应该获得足够的高容量值，其中使用 capacity 参数来防止 resequencer 耗尽内存。

如果要使用 长 以外的某些类型的序列号，您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器模式。以下示例定义了一个流重新排序器，消息容量为 5000，超时为 4000 毫秒：

```
<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>
```

忽略无效的交换

如果传入的交换不是有效的5-4会（例如，由于缺少标头，则重新排序表达式）会抛出 CamelExchangeException 异常（例如，由于缺少标头），则重新排序表达式。您可以使用 ignoreInvalidExchanges 选项忽略这些例外，这意味着 resequencer 将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 选项可用于防止消息按顺序发送，无论用于重新排序消息的机制是什么。启用 rejectOld 选项后，resequencer 将拒绝传入消息（通过抛出 MessageRejectedException 异常），如果传入的消息比最近发送的消息 旧 （如当前比较器定义）。

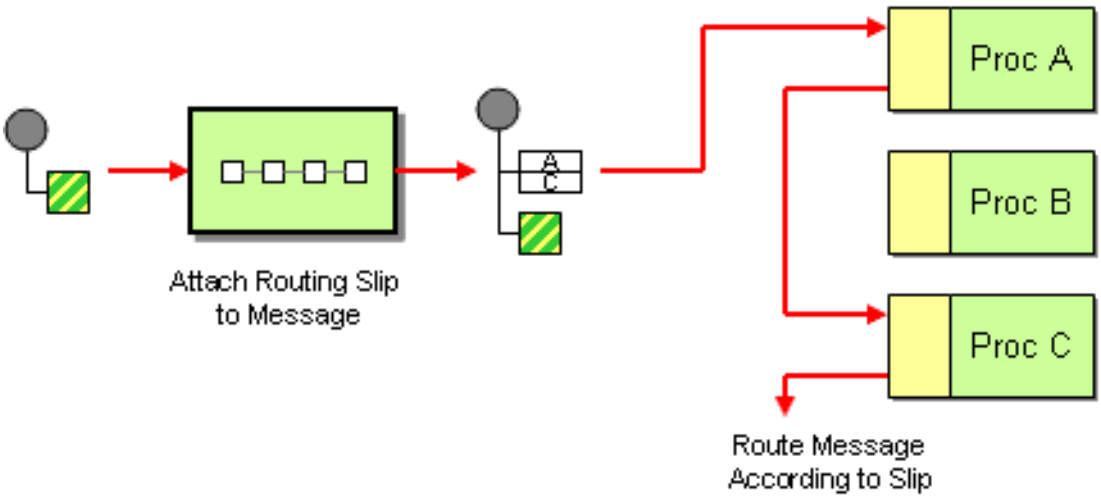
```
from("direct:start")
    .onException(MessageRejectedException.class).handled(true).to("mock:error").end()
    .resequence(header("seqno")).stream().timeout(1000).rejectOld()
    .to("mock:result");
```

8.7. 路由片段

概述

图 8.8 “路由聚合模式” 中显示的 路由 slip 模式允许您通过一系列处理步骤连续路由消息，其中设计时不已知的步骤序列，并可能因每个消息而异。消息要传递的端点列表存储在标头字段中(slip)，Apache Camel 在运行时读取以即时构建管道。

图 8.8. 路由聚合模式



slip 标头

路由 slip 会出现在用户定义的标头中，其中标头值是以逗号分隔的端点 URI 列表。例如，一个路由 slip，它指定了一个安全任务序列，它指定了一个安全任务序列，身份验证、身份验证和去除重复信息类似如下：

```
cxf:bean:decrypt,cxf:bean:authenticate,cxf:bean:dedup
```

当前端点属性

在 Camel 2.5 中，路由 Slip 将在交换上设置属性(Exchange.SLIP_ENDPOINT)，该交换包含当前端点，如 slip 那样。这可让您了解交换通过 slip 进行的进展。

第 8.7 节 “路由片段” 将 事先 计算 slip，这意味着 slip 仅计算一次。如果您需要计算 slip on-fly，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 端点获取消息，并从 aRoutingSlipHeader 标头读取路由 slip ：

```
from("direct:b").routingSlip("aRoutingSlipHeader");
```

您可以将标头名称指定为字符串 literal 或 expression。

您还可以使用双参数格式 routingSlip （） 自定义 URI 分隔符。以下示例定义了一个路由，它使用 aRoutingSlipHeader 标头键作为路由 slip，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 “路由片段” 现在支持 ignoreInvalidEndpoints，它 第 8.3 节 “接收者列表” 模式也支持。您可以使用它来跳过无效的端点。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>
```

例如，myHeader 包含两个端点 direct:foo,xxx:bar。第一个端点有效且正常工作。第二个无效，因此忽略。每当遇到无效的端点时，Apache Camel 会在 INFO 级别记录。

选项

routingSlip DSL 命令支持以下选项：

Name	默认值	描述
uriDelimiter	,	如果 Expression 返回多个端点，则使用分隔符。
ignoreInvalidEndpoints	**false**	如果无法解析端点 uri，则它应该被忽略。否则 Camel 将抛出一个例外，表示端点 uri 无效。
cacheSize	**0**	Camel 2.13.1/2.12.4 ： 允许为 ProducerCache 配置缓存大小，该缓存生成者可在路由 slip 中重复使用。默认情况下，将使用默认缓存大小为 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，用于限制传入消息的流率。您可以使用此模式来保护目标端点过载。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 个信息，请定义一个路由，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 个信息，请定义一个路由，如下所示：

```
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>
```

动态更改每个周期的最大请求

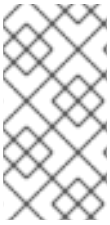
Camel 2.8 Since 的可用 os 使用 Expression，您可以在运行时调整这个值，例如，您可以提供一个带有值的标头。在运行时 Camel 评估表达式，并将结果转换为 java.lang.Long 类型。在以下示例中，我们使用消息的标头来确定每个周期的最大请求。如果没有标头，则 第 8.8 节 “Throttler” 将使用旧值。因此，只有在要更改值时才提供标头：

```
<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 将调度一个任务来在以后执行。该任务负责处理路由的后方部分（在 throttler 之后）。这允许调用者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

在 Camel 2.17 中，Throttler 将使用滚动窗口进行持续处理，从而提供更好的消息流。但是，它将提高节流器的性能。

选项

throttle DSL 命令支持以下选项：

Name	默认值	描述
maximumRequestsPerPeriod		每个 period 到 throttle 的最大请求数。必须提供这个选项，并有一个正数。请注意，在 XML DSL 中，来自 **Camel 2.8** onwards 此选项使用 Expression 而不是属性进行配置。
timePeriodMillis	**1000**	millis 中的时间段，throttler 将允许最多 **maximumRequestsPerPeriod** 的消息数。
asyncDelayed	**false**	Camel 2.4： 如果启用，则使用调度线程池异步发生的任何消息。
executorServiceRef		Camel 2.4： 如果启用了 **asyncDelay**，则引用使用自定义线程池。
callerRunsWhenRejected	**true**	Camel 2.4： 如果启用了 **asyncDelayed**，则使用它。这控制调用者线程是否应该在线程池拒绝任务时执行任务。

8.9. DELAYER

概述

延迟器 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 （） 命令将 相对时间延迟 （以毫秒为单位）添加到传入的消息。例如，以下路由会延迟所有传入的信息(2 秒)：

```
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 （） 以下的 DSL 命令解释为 delay （） 的子斜杠。因此，在某些上下文中，需要插入 end （） 命令来终止 delay （）的子层。例如，当 delay （） 出现在 onException （） 子句中时，您将终止它，如下所示：

```
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

您可以将表达式与 bean 结合使用来确定延迟，如下所示：

```
from("activemq:foo").
  delay().expression().method("someBean", "computeDelay").
  to("activemq:bar");
```

bean 类可以定义如下：

```
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

您可以让延迟程序使用 非阻塞异步延迟，这意味着 Apache Camel 将调度一个任务来在以后执行。该任务负责处理路由的后方部分（延迟后）。这允许调用者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：

Name	默认值	描述
asyncDelayed	**false**	Camel 2.4： 如果启用，则使用调度的线程池异步发生延迟消息。
executorServiceRef		Camel 2.4： 如果启用了 **asyncDelay**，则引用使用自定义线程池。
callerRunsWhenRejected	**true**	Camel 2.4： 如果启用了 **asyncDelayed**，则使用它。这控制调用者线程是否应该在线程池拒绝任务时执行任务。

8.10. LOAD BALANCER

概述

通过 负载均衡器 模式，您可以使用各种不同的负载平衡策略将消息处理委派给多个端点之一。

Java DSL 示例

以下路由使用 round robin 负载均衡策略在目标端点 mock:x,mock:y,mock:z 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平衡策略：*

- ***round robin***
- ***随机***
- ***Sticky***
- ***Topic***
- ***故障切换***
- ***加权轮循和权重随机***
- ***自定义 Load Balancer***
- ***断路器***

round robin

循环通过所有目标端点进行负载平衡策略周期，将每个传入消息发送到周期中的下一个端点。例如，如果目标端点列表是 mock:x,mock:y,mock:z ,，进入的消息将发送到以下端点序列： mock:x,mock:y, mock:z ,mock:x, mock:y ,mock:y,mock:z 等等。

您可以在 Java DSL 中指定循环负载平衡策略，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

随机负载平衡策略从指定的列表中选择目标端点。

您可以在 Java DSL 中指定随机负载平衡策略，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

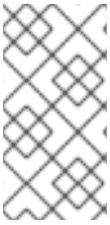
粘性负载平衡策略将 In 消息定向到通过计算指定表达式中的哈希值来选择的端点。这种负载平衡策略的优点是，相同值的表达式始终发送到同一服务器。例如，通过计算包含用户名的标头的哈希值，您可以确保特定用户的消息始终发送到同一目标端点。另一种有用的方法是指定一个表达式，从传入消息中提取会话 ID。这样可确保属于同一会话的所有消息都发送到同一目标端点。

您可以在 Java DSL 中指定粘性负载平衡策略，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 选项添加到故障转移负载均衡器时，负载均衡器会从最后已知的良好端点开始。

Topic

主题负载平衡策略将每个 In 消息的副本发送到所有列出的目标端点（有效地将消息广播到所有目的地，如 JMS 主题）。

您可以使用 Java DSL 指定主题负载平衡策略，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

从 Apache Camel 2.0 开始，故障转移 负载均衡器能够尝试下一个处理器，以防交换在处理期间失败。您可以使用触发 故障转移 的特定例外列表配置故障切换。如果没有指定任何异常，则任何异常会触发故障转移。故障转移负载平衡器使用与 onException 异常条款相同的策略。

使用流时启用流缓存

如果使用流，在使用故障转移负载均衡器时应启用 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	Default（默认）	描述

inheritErrorHandler	布尔值	**true**	Camel 2.3： 指定是否使用路由上配置的 **errorHandler**。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**）。如果启用这个选项，Apache Camel 将首先尝试使用 **errorHandler** 处理消息。 例如，**errorHandler** 可能被配置为 redeliver 消息，并在尝试之间使用延迟。Apache Camel 最初会尝试重新显示 **到原始**端点，只有在错误处理程序耗尽时，才会切换到下一个端点。
maximumFailoverAttempts	**int**	**-1**	Camel 2.3： 指定尝试故障转移到新端点的最大次数。值 **0** 表示没有进行故障转移尝试，值 **-1** 表示是无限数量的故障转移尝试。
roundRobin	布尔值	**false**	Camel 2.3： 指定 **故障转移** 负载均衡器是否应该以轮循模式运行。如果没有，它将 **始终在** 处理新消息时从第一个端点启动。换句话说，它会针对每个消息从顶部重新启动。如果启用了循环，它会保持状态，并以轮循方式继续进行下一端点。当使用 round robin 时，它不会记住最后一个已知的良好端点，它将始终选择要使用的下一个端点。

以下示例被配置为故障切换，只有在抛出 IOException 异常时才进行：

```
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在 round robin 模式中故障转移：

```
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**，您可以确保它不干预。这允许故障转移负载均衡器尽快处理故障切换。如果您也启用了 **roundRobin** 模式，它会一直重试，直到成功为止。然后您可以将 **maximumFailoverAttempts** 选项配置为高值，使其最终耗尽并失败。*

加权轮循和权重随机

*在许多企业环境中，不等处理能力的服务器节点是托管服务的，通常最好根据各个服务器处理容量来分发负载。可以使用 **加权循环** 算法或 **加权随机** 算法来解决这个问题。*

*加权负载平衡策略允许您为每个服务器 **指定**处理负载比率。您可以将这个值指定为每台服务器的正处理权重。较大的数字表示服务器可以处理更大的负载。处理权重用于决定每个处理端点的有效负载分布比例与其它端点相关。*

下表描述了可以使用的参数：

表 8.3. 加权选项

选项	类型	Default（默认）	描述
roundRobin	**布尔值**	**false**	round-robin 的默认值为 **false**。如果没有此设置或参数，则使用负载平衡算法是随机的。
distributionRatioDelimiter	**字符串**	**,**	**distributionRatioDelimiter** 是用于指定 发布率 的分隔符。如果没有指定此属性， 逗号 是默认的分隔符。

*以下 **Java DSL** 示例演示了如何定义加权循环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 Load Balancer

您可以使用自定义负载均衡器（如您自己的实现）。

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");
```

和使用 XML DSL 的同一示例：

```
<!-- 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，我们使用 <custom>，它只在 Camel 2.8 开始中可用。在旧版本中，您必须执行如下操作：

```
    <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，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，它将进入 open 状态并拒绝所有调用，直到达到 一半 OpenAfter 超时为止。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 closed 状态（如果不是），则会返回开放状态。

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

从 Camel 2.18 开始提供。

Hystrix 模式允许应用程序与 Netflix Hystrix 集成，可在 Camel 路由中提供断路器。Hystrix 是一个延迟和容错库，旨在设计

- **隔离对远程系统、服务和第三方库的访问点**
- **停止级联失败**
- **在故障不可避免的复杂分布式系统中启用弹性**

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，它显示了一个 Hystrix 端点，它通过回退到在线的回退路由来防止防止速度较慢的操作。默认情况下，超时请求只为 1000ms，因此 HTTP 端点必须非常快速地成功。

```
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，您可以在其中转换消息或调用 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，则应使用 onFallbackViaNetwork （） 方法，该方法在使用其自身线程池的独立 HystrixCommand 对象中运行，因此它不会耗尽第一个命令对象。

Hystrix 配置示例

Hystrix 有许多选项，如下一节中所列。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，并使断路器等待 10 秒，而不是 5 秒（默认），在尝试尝试打开状态时再次尝试请求。

```
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。

Name	默认值	类型	描述
circuitBreakerEnabled	**true**	布尔值	确定断路器是否用于跟踪健康以及短路请求（如果被行程）。
circuitBreakerErrorThresholdPercentage	**50**	整数	设置在或上面的错误百分比，使电路应打开并启动对回退逻辑的短路请求。
circuitBreakerForceClosed	**false**	布尔值	值 true 会强制断路器进入闭状态，其允许请求而不考虑错误百分比。
circuitBreakerForceOpen	**false**	布尔值	true 值强制断路器进入开放（往返）状态，其中拒绝所有请求。
circuitBreakerRequestVolumeThreshold	**20**	整数	在滚动窗口中设置将遍历电路的最少请求数。
circuitBreakerSleepWindownInMilliseconds	**5000**	整数	设置在传输电路以拒绝请求后的时间长度。在这个时间过后，允许请求尝试确定电路是否应再次关闭。
commandKey	节点 ID	字符串	标识 Hystrix 命令。您不能配置这个选项。它始终是节点 ID，以使命令是唯一的。
corePoolSize	**10**	整数	设置 core thread-pool 大小。这是可以同时执行的 **HystrixCommand** 对象的最大数量。

Name	默认值	类型	描述
executionIsolationSemaphoreMaxConcurrentRequests	**10**	整数	设置 **HystrixCommand.run （）** 方法在使用 **ExecutionIsolationStrategy.SEMAPHORE** 时可以发出的最大请求数。
executionIsolationStrategy	**THREAD**	字符串	指明执行哪些隔离策略 **HystrixCommand.run （）**。**THREAD** 对单独的线程和并发请求执行受 thread-pool 中的线程数量的限制。**SEMAPHORE** 对调用线程和并发请求执行受 semaphore 计数的限制：
executionIsolationThreadInterruptOnTimeout	**true**	布尔值	指明 **HystrixCommand.run （）** 执行在超时时是否应该中断。
executionTimeoutInMilliseconds	**1000**	整数	设置执行完成的时间（以毫秒为单位）。
executionTimeoutEnabled	**true**	布尔值	指明是否执行 **HystrixCommand.run （）** 应该 timed。
fallbackEnabled	**true**	布尔值	决定在出现故障时还是拒绝时尝试调用 **HystrixCommand.getFallback （）**。
fallbackIsolationSemaphoreMaxConcurrentRequests	**10**	整数	设置 **HystrixCommand.getFallback （）** 方法可以从调用线程进行的最大请求数。
groupKey	**CamelHystrix**	字符串	标识用于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。

Name	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	设置允许执行健康快照之间等待的时间（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数。如果在期间发生更多的执行，它们将在存储桶的开头换行并启动过度写入。
metricsRollingPercentileEnabled	**true**	布尔值	指明是否应跟踪执行延迟。延迟计算为百分比。值为 false 会导致摘要统计（百分比）返回为 -1。
metricsRollingPercentileWindowBuckets	**6**	整数	设置 **rollingPercentile** 窗口将划分为的 bucket 数量。
metricsRollingPercentileWindowInMilliseconds	**60000**	整数	设置滚动窗口的持续时间，在其中保留执行时间，允许有百分比的计算（以毫秒为单位）。
metricsRollingStatisticalWindowBuckets	**10**	整数	设置滚动统计窗口划分为的 bucket 数量。
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	设置队列大小 rejection 阈值 - 人工最大队列大小，即使 swig**maxQueueSize** 尚未达到拒绝，也会发生拒绝。

Name	默认值	类型	描述
requestLogEnabled	**true**	布尔值	指明 **HystrixCommand** 执行，事件是否应记录到 **HystrixRequestLog**。
threadPoolKey	**null**	字符串	定义此命令应在哪个 thread-pool 中运行。默认情况下，这使用与组密钥相同的密钥。
threadPoolMetricsRollingStatisticalWindowBucket	**10**	整数	设置滚动统计窗口划分为的 bucket 数量。
threadPoolMetricsRollingStatisticalWindowInMilliseconds	**10000**	整数	以毫秒为单位设置统计滚动窗口持续时间。这是线程池保留指标的时长。

8.12. 服务调用

概述

从 Camel 2.18 开始提供。

服务调用 模式允许您在分布式系统中调用远程服务。要调用的服务位于服务 registry 中，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

要调用服务，请参考服务名称，如下所示：

```
from("direct:start")
    .serviceCall("foo")
    .to("mock:result");
```

以下示例显示了调用服务的 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 集成的组件来查找带有名称 foo 的服务。查找返回一组 IP:PORT 对，引用托管远程服务的活跃服务器的列表。然后，Camel 从该列表中随机选择要使用的服务器，并使用所选 IP 和 PORT 号构建 Camel URI。

默认情况下，Camel 使用 HTTP 组件。在上例中，调用解析为由动态 toD 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，例如 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

正如您所见，服务名称解析为 **Camel 端点 URI。下面是几个示例。→ 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**，请提供指定所需 **Camel URI** 的额外 **URI** 参数。在指定的 **URI** 中，您可以使用服务名称，该服务名称解析为 **IP:PORT。下面是一些示例：**

```
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。如果您只指定了 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。您可以使用不同的组件（如 **HTTP4** 或 **Netty4 HTTP**）配置不同的组件，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

以下选项可用于每个实现：

选项	默认值	描述
clientProperty		指定特定于您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。您可以使用 netty4-http, jetty, restlet 或一些其他组件来配置组件。如果服务没有使用 HTTP 协议，则必须使用另一个组件，如 mqtt、jms、amqp。如果您在服务调用中指定 URI 参数，则使用此参数中指定的组件，而不是默认。
loadBalancerRef		设置使用自定义 **org.apache.camel.spi.ServiceCallLoadBalancer** 的引用。
serverListStrategyRef		设置一个对要使用的自定义 **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		在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中加载的证书颁发机构数据。
clientCertData		在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中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		设置在使用客户端查找时从文件中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		设置用于 **dns** 查找的 DNS 域。
lookup	**环境**	用于查找服务的策略选择。lookup 策略包括： • **环境** wagon-wagon 使用环境变量。 • **DNS** wagon-wagon 使用 DNS 域名。 • **客户端** wagon-wagon 使用 Java 客户端调用 Kubernetes 主 API，并查询哪些服务器正在主动托管该服务。

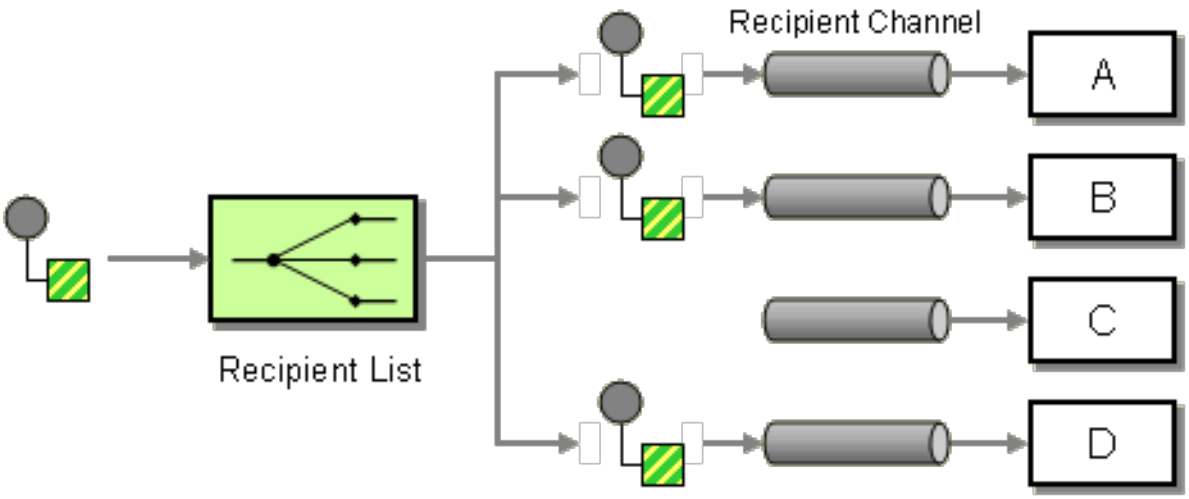
masterUrl		使用客户端查找时的 Kubernetes 主机的 URL。
namespace		要使用的 Kubernetes 命名空间。默认情况下，命名空间的名称从环境变量 **KUBERNETES_MASTER** 获取。
oauthToken		在使用客户端查找时，设置用于身份验证的 OAUTH 令牌（而不是用户名/密码）。
password		设置在使用客户端查找时用于身份验证的密码。
trustCerts	false	设置在使用客户端查找时是否打开信任证书检查。
username		设置在使用客户端查找时用于身份验证的用户名。

8.13. 多播

概述

图 8.9 “多播模式” 中显示的 多播 模式是带有固定目的地模式 的接收者列表 的变化，它与 InOut 消息交换模式兼容。这与接收者列表相反，其仅与 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

多播处理器在响应原始请求时接收多个 Out 消息（每个接收方一个），而原始调用者仅预期接收 单个回复。因此，消息交换的回复分支存在固有的不匹配，并且为了克服这种不匹配，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。

例如，电子广告服务的示例，销售者为购买者提供销售项目，以购买者为购买者列表提供项目。买家各自置于项目投标中，销售者会自动选择具有最高价格的奖金。您可以使用 multicast （） DSL 命令实现将所提供的服务分发给固定购买列表的逻辑，如下所示：

```
from("cxf:bean:offer").multicast(new HighestBidAggregationStrategy()).
    to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

其中，销售者由端点 cxf:bean:offer 表示，而买方由端点 cxf:bean:Buyer1 表示，cxf:bean:Buyer2,cxf:bean:Buyer3。为了整合不同买方收到的投标，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;
    }
}
```

假设买方将标价插入到名为 Bid 的标头中。有关自定义聚合策略的详情，请参考 第 8.5 节 “聚合器”。

并行处理

默认情况下，多播处理器会在另一个后调用每个接收者端点（按 to （） 命令中列出的顺序）。在某些情况下，这可能导致不可接受的延迟。为避免这些较长的延迟时间，您可以通过添加 parallelProcessing （） 子句来启用并行处理。例如，要在电子调整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");
```

其中多播处理器现在调用 buyer 端点，它使用一个线程池，每个端点都有一个线程。

如果要自定义调用购买端点的线程池的大小，您可以调用 **executorService （）** 方法来指定您自己的自定义 **executor** 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，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信息，并丢弃早期回复。例如，在以下路由中，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** 中配置类似的路由，其中路由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>
```

其中 parallelProcessing 属性和 threadPoolRef 属性都是可选的。只有在您要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 复制源交换，并多播副本。默认情况下，路由器制作源消息的粗略副本。在 Show 副本中，原始消息的标头和有效负载仅通过引用复制，因此原始消息生成的副本被链接。因为应链接多播消息的复制，因此如果消息正文是可变的，您将无法应用自定义处理。您应用到发送到一个端点的副本的自定义处理也会应用到发送到所有其他端点的副本。

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 具有与 onPrepare 相同的效果（实际上，进程 DSL 命令无效）。

在准备信息时，使用 onPrepare 执行自定义逻辑

如果要在将自定义处理发送到其端点之前对每个消息副本应用自定义处理，您可以在 multicast 子句中调用 onPrepare DSL 命令。onPrepare 命令仅在消息被压缩 后 插入自定义处理器，并在将消息分配给其端点 之前。例如，在以下路由中，在发送到 direct:a 的消息上调用 CustomProc 处理器，而在发送到 direct:b 的消息上也会调用 CustomProc 处理器。

```
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一些或所有元素执行深度副本。例如，以下 CustomProc 处理器类执行消息正文的深度副本，其中消息正文假定为类型为 BodyType，而深度副本则由方法 BodyType.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 实现在 交换 多播前要执行的任何类型的自定义逻辑。

注意

建议为不可变对象设计。

例如，如果您有一个 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度克隆处理器来克隆消息正文：

```
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 multicast 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

multicast DSL 命令支持以下选项：

Name	默认值	描述
strategyRef		用于将多播中的回复汇编为来自多播的 AggregationStrategy，再到来自 多播 的单一传出消息。默认情况下，Camel 将使用最后一个回复作为传出消息。
strategyMethodName		当将 POJO 用作 **AggregationStrategy** 时，可以使用此选项明确指定要使用的方法名称。
strategyMethodAllowNull	**false**	当将 POJO 用作 **AggregationStrategy** 时，可以使用此选项。如果为 **false**， 则不会使用聚合方法，当没有数据增强时。如果为 **true**，则空值用于 **oldExchange**，如果没有数据更丰富，则使用**null** 值。
parallelProcessing	**false**	如果启用，将消息同时发送到多播。请注意，调用者线程仍然会等待所有消息被完全处理，然后再继续。它只会发送和处理同时发生的多播的回复。

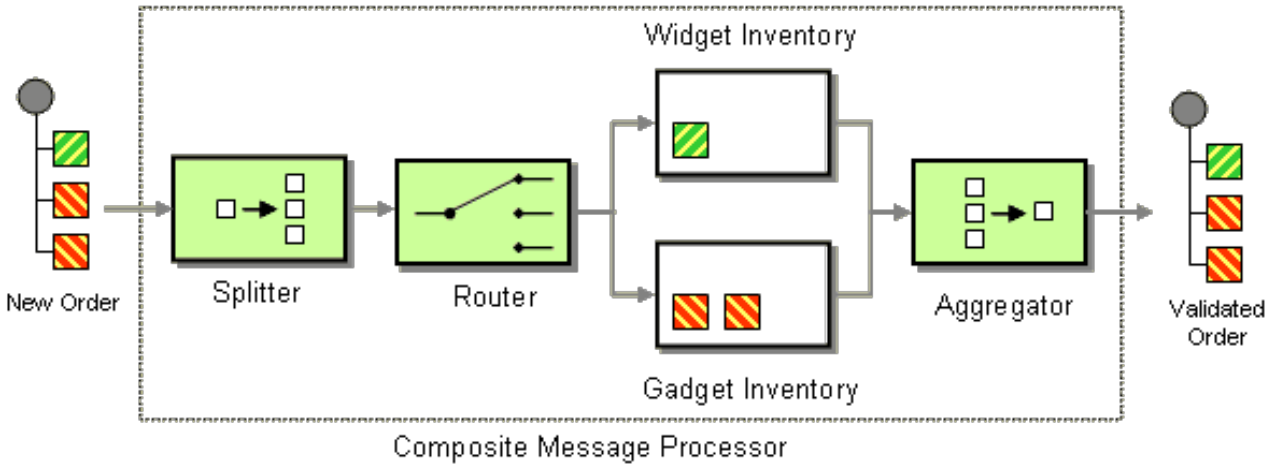
parallelAggregate	**false**	如果启用，则 **AggregationStrategy** 上的聚合方法可同时调用。**请注意，这需要实施 AggregationStrategy 才能实现 thread-safe。默认情况下，此选项为 false，这意味着 Camel 会自动同步聚合方法的调用。但是，在某些用例中，您可以通过将 AggregationStrategy 设为 thread-safe 来提高性能，并将此选项设置为 true。**
executorServiceRef		指的是用于并行处理的自定义线程池。请注意，如果您设置了这个选项，则并行处理是自动的，您不必启用该选项。
stopOnException	**false**	Camel 2.2 ： 在出现异常情况时，是否立即停止继续处理。如果禁用，则 Camel 会将消息发送到所有多播，无论它们是否失败。您可以在 AggregationStrategy 类中处理异常，您可以完全控制如何处理它。
streaming	**false**	如果启用，Camel 将按其返回的顺序处理回复，例如：如果禁用，Camel 将以与多播相同的顺序处理回复。
timeout		Camel 2.5： 设置以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，如果您提供了一个 TimeoutAwareAggregationStrategy，则在中断前调用 **timeout** 方法。
onPrepareRef		Camel 2.8： 请参阅自定义处理器准备每个多播将收到的 Exchange 副本。这可让您执行任何自定义逻辑，如在需要时深度获取消息有效负载。
shareUnitOfWork	**false**	Camel 2.8： 是否应共享工作单元。详情请查看 第 8.4 节 “Splitter” 中的同一选项。

8.14. 由消息处理器

由消息处理器

由 图 8.10 “由消息处理器模式” 中显示的消息处理器 模式允许您通过分割复合消息来处理复合消息，将子消息路由到适当的目的地，然后将响应重新聚合回单个消息。

图 8.10. 由消息处理器模式



Java DSL 示例

以下示例检查是否可以填充多部分顺序，其中顺序的每个部分都需要在不同清单中进行检查：

```
// 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

前面的路由也可以使用 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

处理从使用 第 8.4 节 “Splitter” 的顺序分割开始。第 8.4 节 “Splitter” 然后，将单独的 OrderItem 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。在 widgetInventory bean 和 gadget 项被发送到 gadgetInventory bean 中，会发送 小部件 项。当适当的 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 到一个顺序。

每个收到的顺序都有一个包含 顺序 ID 的标头。我们在聚合步骤中使用顺序 ID： aggregate （） DSL 上的 .header ("orderId") qualifier 指示聚合器将标头与键(orderId)用作关联表达式。

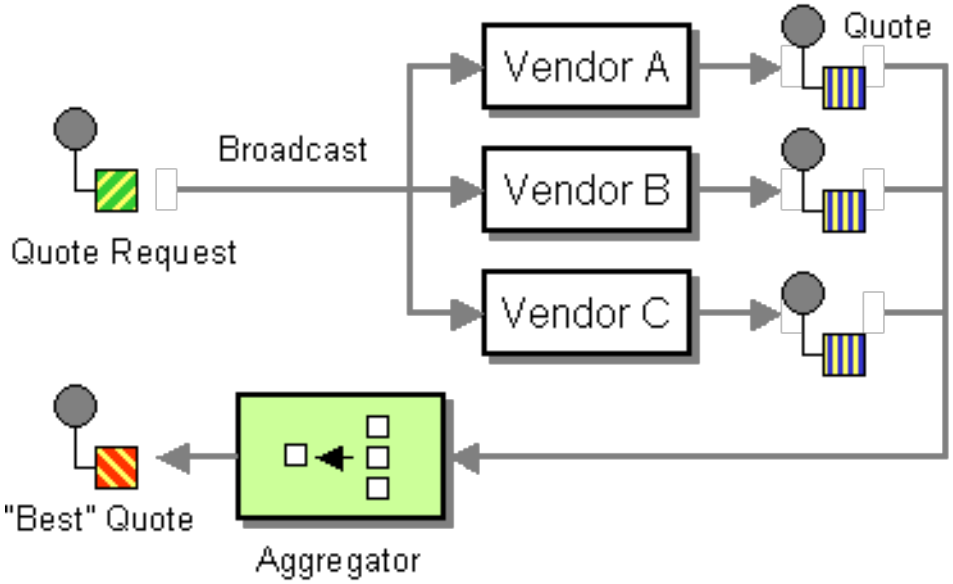
有关完整详情，请检查 camel-core/src/test/java/org/apache/camel/processor 的 ComposedMessageProcessorTest.java 示例源。

8.15. SCATTER-GATHER

scatter-Gather

如 图 8.11 “scatter-Gather Pattern” 所示，scatter-gather 模式 可让您将消息路由到多个动态指定的接收者，并将响应重新整合回单个消息。

图 8.11. scatter-Gather Pattern



动态 scatter-gather 示例

下例概述了从多个不同供应商获取最佳引用的应用程序。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和 第 8.5 节 “聚合器” 的报价，以选择所有响应中的最佳引用。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” 会查看 listOfVendors 标头来获取接收者列表。因此，向此应用发送消息的客户端需要向消息添加一个 listOfVendors 标头。例 8.1 “消息传递客户端示例” 显示消息传递客户端中的一些示例代码，它将相关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);
```

消息将分发到以下端点： bean:vendor1,bean:vendor2, 和 bean:vendor3。这些 Bean 都由以下类实现：

```
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 beer （传递至构造器参数）的不同价格进行初始化。当消息发送到每个 bean 端点时，它会到达 MyVendor.getQuote 方法。此方法通过简单的检查来查看此报价请求是否是 beer，然后稍后的步骤对交换进行检索。使用 POJO Producing 将消息转发到下一步（请参阅 @Produce 注释）。

下一步，我们想要从所有供应商中取胜者标语，并找出哪个最能（即最低）。为此，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信息，这些信息会根据 quoteRequestId 标头的值进行，该消息根据 quoteRequestId 标头的值进行（传递至 correlationExpression）。如 例 8.1 “消息传递客户端示例” 中所示，关联 ID 设置为 quoteRequest-1 (关联 ID 应是唯一的)。要从集合中选择最低报价，您可以使用如下自定义聚合策略：

```
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。以下示例显示了用于实现静态 scatter-gather 场景的路由：

```
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，循环在整个循环中使用相同的交换。上一个迭代的结果用于下一个（请参阅 第 5.4 节 “管道和过滤器”）。在 上，您可以从 Camel 2.8 中启用复制模式。详情请查看 options 表。

Exchange 属性

在每个循环迭代中，设置了两个交换属性，它们可以选择由循环中包含的任何处理器读取。

属性	描述
CamelLoopSize	Apache Camel 2.0: 循环总数
CamelLoopIndex	Apache Camel 2.0：索引当前迭代（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，然后重复向 mock:result 发送消息。循环迭代的数量被指定为 loop （） 的参数，或者在运行时评估表达式，其中表达式 必须 评估 int （或其他 RuntimeCamelException 被抛出）。

以下示例将循环计数作为常数传递：

```
from("direct:a").loop(8).to("mock:result");
```

以下示例评估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 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常数传递：

```
<route>
  <from uri="direct:a"/>
  <loop>
    <constant>8</constant>
    <to uri="mock:result"/>
  </loop>
</route>
```

以下示例评估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，假设我们向 **direct:start** 端点发送一条包含字母 **A** 的消息。处理此路由的输出将是这样，每个 **mock:loop** 端点都将接收 **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，则 **mock:loop** 将接收 **AB, ABB, ABBB** 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");
```

复制模式中 XML DSL 中等效的示例如下：

```
<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：

Name	默认值	描述
复制	**false**	Camel 2.8： 是否使用复制模式。如果为 **false**，则在循环中会使用相同的 Exchange。因此，在下一次迭代 **中可以看到** 上一个迭代的结果。相反，您可以启用复制模式，然后每个迭代都会使用新输入 "Exchanges"一节 来重启。

Do While Loop

您可以执行循环，直到使用 when loop 时 满足条件为止。条件为 true 或 false。

在 DSL 中，命令是 LoopDoWhile。以下示例将执行循环，直到消息正文长度为 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 doWhile。以下示例还执行循环，直到消息正文长度为 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

抽样节流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，在此期间 只允许一个 交换通过。所有其他交换将停止。

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，您可以选择使用 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

示例 DSL 命令支持以下选项：

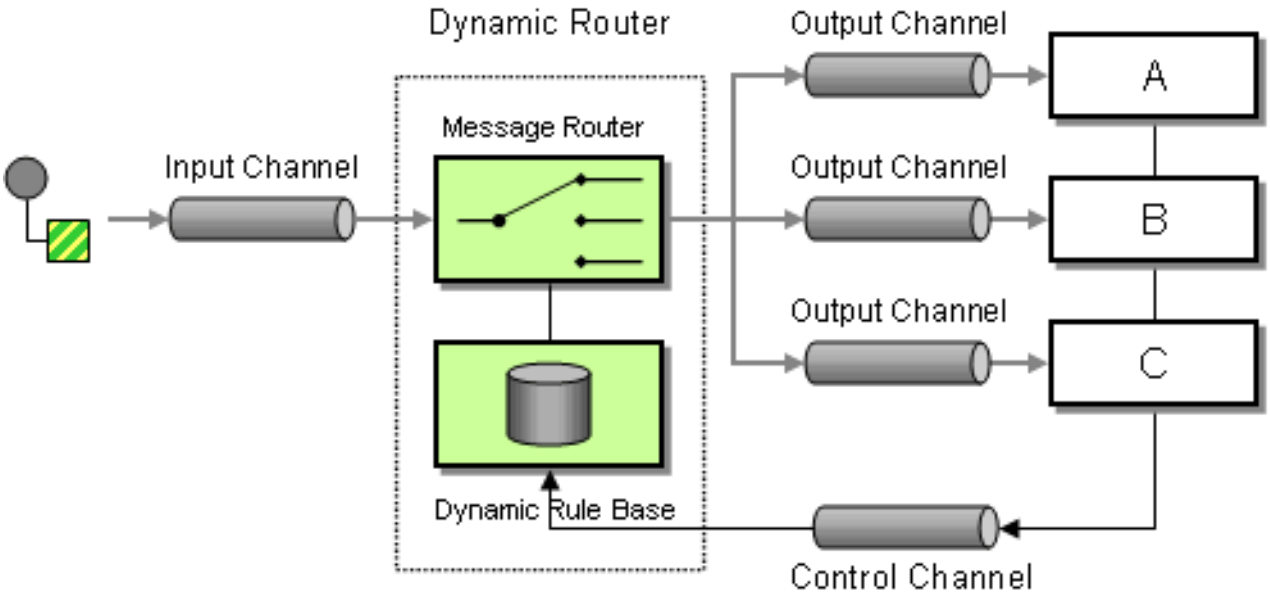
Name	默认值	描述
messageFrequency		对第 N 条消息进行抽样。您只能使用频率或周期。
samplePeriod	**1**	对第 N 个期间内的信息进行抽样。您只能使用频率或周期。
units	**SECOND**	时间单位作为 JDK 中的 **java.util.concurrent.TimeUnit** 的枚举。

8.18. 动态路由器

动态路由器

Dynamic Router 模式（如 图 8.12 “动态路由器模式” 所示）可让您通过一系列处理步骤来连续路由消息，其中在设计时不知道步骤序列。在运行时 动态计算消息应通过的端点列表。每次消息从端点返回时，在 bean 上重新调用动态路由器，以发现路由中的下一个端点。

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 dynamicRouter，它类似于一个动态 第 8.7 节 “路由片段”，用于评估 slip on-fly。

BEWARE

您必须确保用于 dynamicRouter （如 bean）的表达式返回 null 以指示结束。否则，dynamicRouter 将继续处于无限循环中。

Camel 2.5 中的动态路由器以后

在 Camel 2.5 中，第 8.18 节 “动态路由器” 更新交换属性 Exchange.SLIP_ENDPOINT，其当前端点通过 slip 进行了改进。这可让您了解交换通过 slip 进行的进展。（它是一个 slip，因为 第 8.18 节 “动态路由器” 实现基于 第 8.7 节 “路由片段”）。

Java DSL

在 Java DSL 中，您可以使用 dynamicRouter，如下所示：

```
from("direct:start")
    // use a bean as the dynamic router
    .dynamicRouter(bean(DynamicRouterTest.class, "slip"));
```

这将利用 bean 集成来计算 slip on-fly，其实施如下：

```
// 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

Spring XML 中的相同例子是：

```
<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：

Name	默认值	描述
uriDelimiter	,	如果 第 II 部分 “路由表达式和 predicates 语言” 返回多个端点，则使用分隔符。
ignoreInvalidEndpoints	**false**	如果无法解析端点 uri，则它应该被忽略。否则 Camel 将抛出一个例外，表示端点 uri 无效。

@DYNAMICROUTER 注释

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 方法通过 slip 重复作为消息进度调用。其理念是返回下一个目的地的端点 URI。返回 null 以指示结束。就像 第 8.7 节 “路由片段” 一样，您可以返回多个端点，其中每个端点都由分隔符分开。

第 9 章 SAGA EIP

9.1. 概述

Saga EIP 提供了在 Camel 路由中定义一系列相关操作的方法，该路由可成功完成或未执行或被编译。Sahga 实施使用任何传输到全球一致的结果来协调分布式服务通信。saga EIP 与典型的 ACID 分布 (XA)事务不同，因为不同参与服务的状态可以保证仅在 Saga 结束时保持一致，而不是在任何中间步骤中。

saga EIP 适合不建议使用分布式事务的用例。例如，参与 Saga 的服务可以使用任何类型的数据存储，如典型的数据库，甚至 NoSQL 非事务数据存储。它们也适用于在无状态云服务中使用，因为它们不需要与服务一起存储事务日志。saga EIP 也不需要在少量时间内完成，因为它们不使用数据库级别锁定，这与事务不同。因此，它们可以持续较长的时间跨度，从几秒钟到几天。

saga EIPs 不会在数据中使用锁定。它们定义了 Compensating Action 的概念，这是当标准流遇到错误时应执行的操作，并在流执行前恢复存在的状态。可以使用 Java 或 XML DSL 在 Camel 路由中声明编译操作，并且仅在需要时由 Camel 调用（如果因为错误而取消 saga）。

9.2. SAGA EIP 选项

Saga EIP 支持 6 个列出的选项：

Name	描述	默认值	类型
传播	设置 Saga 传播模式(REQUIRED、REQUIRES_NEW、MANDATORY、SUPPORTS、NOT_SUPPORTED、NEVER)。	必需	SagaPropagation
completionMode	确定 Saga 应如何被视为完成。当设置为 **AUTO** 时，当交换成功处理 Saga 时，Saga 已完成，或者在其特殊完成时进行编译。当设置为 **MANUAL** 时，用户必须使用 **saga:complete** 或 **saga:compensate** 端点完成或补偿 Saga。	AUTO	SagaCompletionMode
timeoutInMilliseconds	设置 Saga 的最大时间量。超时过期后，saga 会自动补偿（除非在平均时间内采取不同的决定）。		Long
compensation	必须调用 compensation 端点 URI，以补偿路由中完成的所有更改。与 compensation URI 对应的路由必须执行 compensation，并完成且没有错误。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

Name	描述	默认值	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，并终止且没有错误。如果在完成过程中发生错误，Saga 服务会再次调用完成 URI 来重试。		SagaActionUriDefinition
选项	允许保存当前交换的属性，以便在编译或完成回调路由中重复使用它们。例如，选项通常有助于存储和检索在编译操作中删除的对象标识符。选项值转换为 compensation/completion 交换的输入标头。		list

9.3. SAGA 服务配置

Saga EIP 要求将实施接口 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

对于生产环境，不建议使用 In-memory Saga 服务，因为它不支持 Saga 状态的持久性（仅保留在内存中），因此无法在应用程序失败时保证 Saga EIP 的一致性（如 JVM 崩溃）。另外，在使用内存 Saga 服务时，无法使用传输级别标头（可以使用其他实现）将 Saga 上下文传播到远程服务。在您要使用内存中 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，您要放置一个新的订购，并且系统中有两个不同的服务：一个管理订购，另一个管理信用。如果您有足够的学分来，您可以按顺序排列。通过 Saga EIP，您可以将 direct:buy 路由建模为由两个不同操作组成的 Saga，一个用于创建顺序，另一个则取用以取信。必须执行这两个操作，或作为没有 信用的订单来执行任何操作都将被视为不一致的结果（以及无订单的付款）。

```
from("direct:buy")
  .saga()
    .to("direct:newOrder")
    .to("direct:reserveCredit");
```

buy 操作不会更改其他示例。用于对 New Order 和 Reserve credit 操作进行建模的不同选项如下：

```
from("direct:newOrder")
  .saga()
  .propagation(SagaPropagation.MANDATORY)
  .compensation("direct:cancelOrder")
    .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
    .bean(orderManagerService, "newOrder")
    .log("Order ${body} created");
```

此处的传播模式被设置为 MANDATORY 表示此路由中的任何交换流都必须已经是 Saga 的一部分（在本例中是因为在 direct:buy 路由中创建的 Saga）。direct:newOrder 路由声明一个称为 direct:cancelOrder 的 compensating 操作，它负责撤销 Saga 取消的顺序。

每个交换始终都包含一个 Exchange.SAGA_LONG_RUNNING_ACTION 标头，此处用作顺序的 id。这标识了在相应的compensating 操作中删除的顺序，但它并不是一个要求（选项可用作替代解决方案）。direct:newOrder 的compensating 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");
```

当应取消顺序时，Saga EIP 实施会自动调用它。它不会终止并显示错误。如果在 direct:cancelOrder 路由中抛出错误，EIP 实现应定期重试来执行最多某一限制的操作。这意味着，任何编译操作都必须是幂等的，因此应该考虑它可能会多次触发，且不应在任何情况下失败。如果在所有重试后无法完成补偿，则 Saga 实施应触发手动干预过程。

注意

可能会因为执行 direct:newOrder 路由时出现一个延迟，Saga 被 meantime 中的另一方取消（因为并行路由或 Saga 级别中的一个超时）被另一个方取消。因此，当调用 compensating action direct:cancelOrder 时，它可能无法找到被取消的 Order 记录。为保证完全的全局一致性非常重要，例如，如果 编译过程在主操作前发生相同效果，则任何主要操作及其相应的补偿操作都是 可取的。

不允许使用 commutative 行为时，在找到主操作（或最多重试次数）的数据之前，在 compensating 操作中持续失败（或最多重试次数）。这种方法在很多上下文中可能工作，但 非常繁琐。

信用服务按照与订购服务相同的方式实施。

```
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");
```

此处为信用卡预留的补偿操作是退款。

9.4.1. 处理完成事件

当 Saga 完成后，需要一些类型的处理。当发生错误并且 Saga 被取消时，会调用 compensation 端点。成功完成后，可以调用完成端点来进一步处理。例如，在上述订购服务中，我们可能需要知道何时完成订购（以及保留信用）来实际开始准备订单。如果不完成付款，我们不希望开始准备订单（除非大多数现代 CPU 可让您访问保留内存，然后再确保您具有读取它的权限）。这可以通过 direct:newOrder 端点的修改版本轻松完成：

1. 调用完整端点：

```
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，顺序将发送到 JMS 队列以准备工作。与补偿操作一样，Saga coordinator 可以多次调用完成操作（特别是当出现错误时，如网络错误）。在本例中，侦听 prepareOrder JMS 队列的服务已准备好容纳可能的重复项（请参阅 Idempotent Consumer EIP 以了解如何处理重复的示例）。

9.4.2. 使用自定义标识符和选项

您可以使用 Saga 选项注册自定义标识符。例如，信用服务重构如下：

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 compensating 操作中根据需要委派操作并标记当前的正文。

```
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. 只有在 saga 被取消时才从标头检索 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");
```

通过将传播模式设置为 SUPPORTS，可以在 Saga 之外调用 direct:creditReservation 端点。这样，可以在 Saga 路由中声明多个选项。

9.4.3. 设置超时

在 Saga EIPs 上设置超时保证，Saga 在机器失败时不会一直卡住。Saga EIP 实现在所有未明确指定它的 Saga EIP 上都设置了默认超时。当超时过期时，Saga EIP 将决定 取消 Saga （并补偿所有参与者），除非之前已经采取了不同的决定。

可在 Saga 参与者上设置超时，如下所示：

```
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");
```

所有参与者（如信用服务、订单服务）都可以设置自己的超时。当这些超时被组成时，会取为 saga 的超时时间。可在 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

在上例中，我们使用了 MANDATORY 和 SUPPORTS 传播模式，但也使用 REQUIRED 传播模式，这是任何指定任何其他用途的默认传播。这些传播模式映射 1:1 事务上下文中使用的等效模式。

传播	描述
必需	加入现有的 Saga 或创建新 Saga （如果不存在）。
REQUIRES_NEW	始终创建新的 Saga。挂起旧的 Saga，并在一个新 Saga 终止时恢复它。
必需	必须已存在 Saga。现有的 Saga 已加入。
支持	如果 Saga 已存在，则加入它。
NOT_SUPPORTED	如果 Saga 已存在，则在当前块完成后它会被挂起并恢复。
NEVER	当前块必须永远不会在 Saga 中调用。

9.4.5. 使用手动完成（高级）

当无法以同步的方式执行 Saga 时，但要求使用异步通信频道与外部服务通信，那么完成模式无法设置为 AUTO （默认），因为 Saga 在交换完成时不会完成。对于具有长时间执行时间（小时、天）的 Saga EIP，通常会出现这种情况。在这些情况下，应使用 MANUAL 自动完成模式。

```
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。它们向 seda:operationCompleted 发送异步回调。

```
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 意味着，当路由 direct:mysaga 中处理交换时，Saga 不会被完成，但它将最后更长的时间（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 组件的 saga:complete 端点完成的。有一个类似的端点用于手动补偿 Saga (saga:compensate)。

9.5. XML 配置

saga 功能可供想要使用 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 MESSAGE TRANSFORMATION

摘要

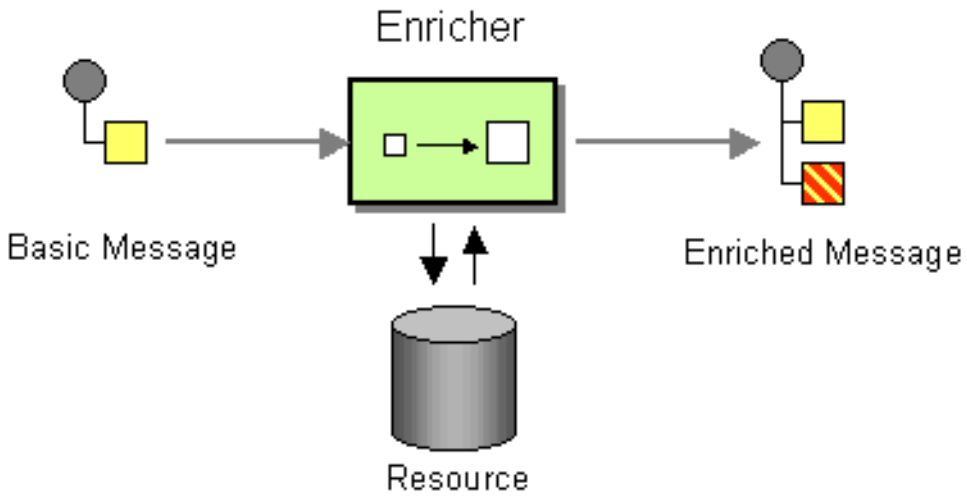
消息转换模式描述了如何修改消息的内容，以满足各种用途。

10.1. 内容增强

概述

内容增强 模式描述了一种场景，其中消息目的地需要比原始消息中存在的数据更多。在这种情况下，您将使用消息转换器、路由逻辑中的任意处理器，或者内容增强方法从外部资源拉取额外的数据。

图 10.1. 内容增强模式

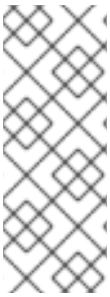


增强内容的替代方案

Apache Camel 支持多种增强内容：

- 在路由逻辑中带有任意处理器的消息转换器
- enrich （） 方法通过将当前交换的副本发送到 制作者 端点，然后使用结果回复中的数据，从资源中获取其他数据。增强者创建的交换始终是一个 InOut 交换。
- pollEnrich （） 方法通过轮询 消费者 端点来获取数据。有效地是 pollEnrich （） 操作中来自 main 路由和消费者端点的消费者端点。也就是说，路由中初始消费者上的传入消息会触发轮

询消费者的 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，以便您从当前交换获取值。例如，您可以使用从数据交换计算的名称来轮询文件。Camel 2.16 中引入了此行为。这个变化会破坏 XML DSL，并可让您轻松地迁移。Java DSL 保持向后兼容。

使用消息转换器和处理器功能丰富的内容

Camel 为使用类型安全 IDE 友好的方式创建路由和调解规则提供 流畅的 构建程序，提供智能完成并正在重构。当您测试分布式系统时，需要存存特定外部系统，以便您可以在特定系统可用或编写之前测试其他系统部分。执行此操作的一种方法是使用某种 模板 系统通过生成具有主要静态正文的动态消息来生成对请求的响应。使用模板的另一种方式是使用来自一个目的地的消息，将其转换为 Velocity 或 XQuery，然后将其发送到另一个目的地。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");
```

假设您想要使用 InOut (request-reply)消息传递来处理 ActiveMQ 上的 My.Queue 队列上的请求。您希望模板生成的响应进入 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 集成来启用使用任何 bean 作为转换器：

```
from("activemq:My.Queue").
  beanRef("myBeanName", "myMethodName").
  to("activemq:Another.Queue");
```

以下示例显示了 Spring XML 实施：

```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>/>
```

使用 enrich （）方法增强内容

```
AggregationStrategy aggregationStrategy = ...

from("direct:start")
  .enrich("direct:resource", aggregationStrategy)
  .to("direct:result");

from("direct:resource")
...
```

内容增强（丰富的 ）从资源端点检索其他数据，以便增强传入消息（包含在 组织交换中）。聚合策略将原始的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。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="..." />
```

在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 将使用默认从资源获取的正文。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 的输出，因为此示例不使用任何自定义聚合。

在 XML DSL 中，只需省略 policyRef 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

丰富的 DSL 命令支持以下选项：

Name	默认值	描述
expression	None	从 Camel 2.16 开始，需要这个选项。指定一个表达式，用于将外部服务的 URI 配置为增强它。您可以使用 Simple 表达式语言、Constant 表达式语言或任何其他可动态计算当前交换值计算 URI 的语言。
uri		这些选项已被删除。指定 **expression** 选项替代。在 Camel 2.15 及更早版本中，需要 **uri** 选项或 **ref** 选项的规格。每个选项都指定外部服务的端点 URI，使其增强：
ref		引用外部服务的端点，使其增强：您必须使用 **uri** 或 **ref**。
strategyRef		指的是用于将来自外部服务的回复合并到一个传出消息中。https://www.javadoc.io/doc/org.apache.camel/camel-core/2.23.2/org/apache/camel/processor/aggregate/AggregationStrategy.html默认情况下，Camel 使用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		当使用 POJO 作为 **AggregationStrategy** 时，指定此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，如果没有数据增强，则不会使用聚合方法。如果此选项为 true，则当没有数据无法增强时，则 null 值将用作 **旧的 Exchange**，并且您使用 POJO 作为 **AggregationStrategy**。如需更多信息，请参阅 Aggregate 模式。

aggregateOnException	false	默认行为是，如果在尝试检索数据时抛出异常，则**不会使用** 聚合方法。将此选项设置为 **true** 可让最终用户控制聚合方法中异常**时要执行的操作**。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，默认行为是增强操作不会在父交换和资源交换之间共享工作单元。这意味着资源交换有自己的独立工作单元。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，它会缓存制作者以便在增强操作中重复使用。要关闭此缓存，请将 **cacheSize** 选项设置为 **-1**。
ignoreInvalidEndpoint	false	从 Camel 2.16 开始，此选项指示是否忽略无法解析的端点 URI。默认行为是，Camel 会抛出一个标识无效端点 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，以增强传入消息，该消息包含在原始交换中。您可以使用聚合策略来组合原始的交换和资源交换。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。增强器创建的资源交换始终是 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 （） 方法支持使用根据当前交换的信息计算的动态 URI。例如，要增强 HTTP 端点，其中带有 orderId 键的标头用作 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 中的相同示例：

```
<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 （）方法增强内容

pollEnrich 命令将资源端点视为 消费者。它不向资源端点发送交换，而是 轮询 端点。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 消息的标头中提取名称的文件：

```
from("activemq:queue:order")
  .pollEnrich("file://order/data/additional?fileName=orderId")
  .to("bean:processOrder");
```

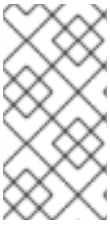
您可以限制等待文件就绪的时间。以下示例显示了最多等待 20 秒：

```
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 配置的使用者。这允许将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 的支持在 Camel 2.18 中是新的。Camel 2.17 不支持此行为。

如果在接收交换前轮询超时，则传递给聚合策略的 aggregate （） 方法的资源交换可能为 null。

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()
- receive （长超时）

pollEnrich （） 命令的 timeout 参数（以毫秒为单位指定）决定要调用的方法，如下所示：

- 当超时为 0 或未指定时，pollEnrich （） 调用 receiveNoWait。
- 当超时为负数时，pollEnrich （） 调用 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 文件中加载内容来增强消息：

```
from("direct:start")
  .pollEnrich("file:inbox?fileName=data.txt")
  .to("direct:result");
```

以下是 XML DSL 中的相同示例：

```
<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。您可以指定一个超时时间（可能要等待，直到文件存在或等待

特定时长）。在以下示例中，命令不会等待超过 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 （） 方法支持使用根据当前交换的信息计算的动态 URI。例如，若要从使用标头来指示 SEDA 队列名称的端点轮询丰富，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 中的相同示例：

```
<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：

Name	默认值	描述
expression	None	从 Camel 2.16 开始，需要这个选项。指定一个表达式，用于将外部服务的 URI 配置为增强它。您可以使用 Simple 表达式语言、Constant 表达式语言或任何其他可动态计算当前交换值计算 URI 的语言。

uri		这些选项已被删除。指定 **expression** 选项替代。在 Camel 2.15 及更早版本中，需要 **uri** 选项或 **ref** 选项的规格。每个选项都指定外部服务的端点 URI，使其增强：
ref		引用外部服务的端点，使其增强：您必须使用 **uri** 或 **ref**。
strategyRef		指的是用于将来自外部服务的回复合并到一个传出消息中。https://www.javadoc.io/doc/org.apache.camel/camel-core/2.23.2/org/apache/camel/processor/aggregate/AggregationStrategy.html默认情况下，Camel 使用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		当使用 POJO 作为 **AggregationStrategy** 时，指定此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，如果没有数据增强，则不会使用聚合方法。如果此选项为 true，则当没有数据无法增强时，则 null 值将用作 **旧的 Exchange**，并且您使用 POJO 作为 **AggregationStrategy**。如需更多信息，请参阅 Aggregate 模式。
timeout	**-1**	从外部服务轮询时，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 （）** 可以阻止到有可用的消息，因此建议始终指定一个超时。
aggregateOnException	false	默认行为是，如果在尝试检索数据时抛出异常，则**不会使用** 聚合方法。将此选项设置为 **true** 可让最终用户控制聚合方法中异常 **时要执行的操作**。例如，可以阻止异常或设置自定义消息正文

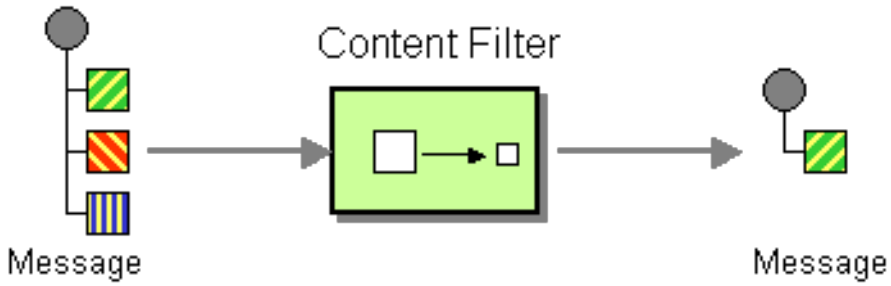
cacheSize	**1000**	指定这个选项来为 **ConsumerCache** 配置缓存大小，它会缓存消费者以在 **pollEnrich** （） 操作中重复使用。要关闭此缓存，请将 **cacheSize** 选项设置为 **-1**。
ignoreInvalidEndpoint	false	指明是否忽略无法解析的端点 URI。默认行为是，Camel 会抛出一个标识无效端点 URI 的异常。

10.2. 内容过滤器

概述

内容过滤器 模式描述了在将消息传递给预期接收者前需要过滤出额外内容的情况。例如，您可以使用内容过滤器从消息中去除机密信息。

*图 **10.2. 内容过滤器模式***



过滤消息的常见方法是使用 DSL 中的表达式，使用其中一个支持的脚本语言（如 XSLT、XQuery 或 JoSQL）。

实施内容过滤器

内容过滤器基本上是消息处理技术的应用程序，用于特定目的。要实现内容过滤器，您可以使用以下任何消息处理技术：

- Message translator wagon-wagon 请参阅 第 5.6 节 “消息 Translator”。
- 处理器 wagon-wagon 请参见 第 35 章 实施处理器.
- 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 过滤您感兴趣的消息部分：

```
<route>
  <from uri="activemq:Input"/>
  <setBody><xpath resultType="org.w3c.dom.Document">//foo:bar</xpath></setBody>
  <to uri="activemq:Output"/>
</route>
```

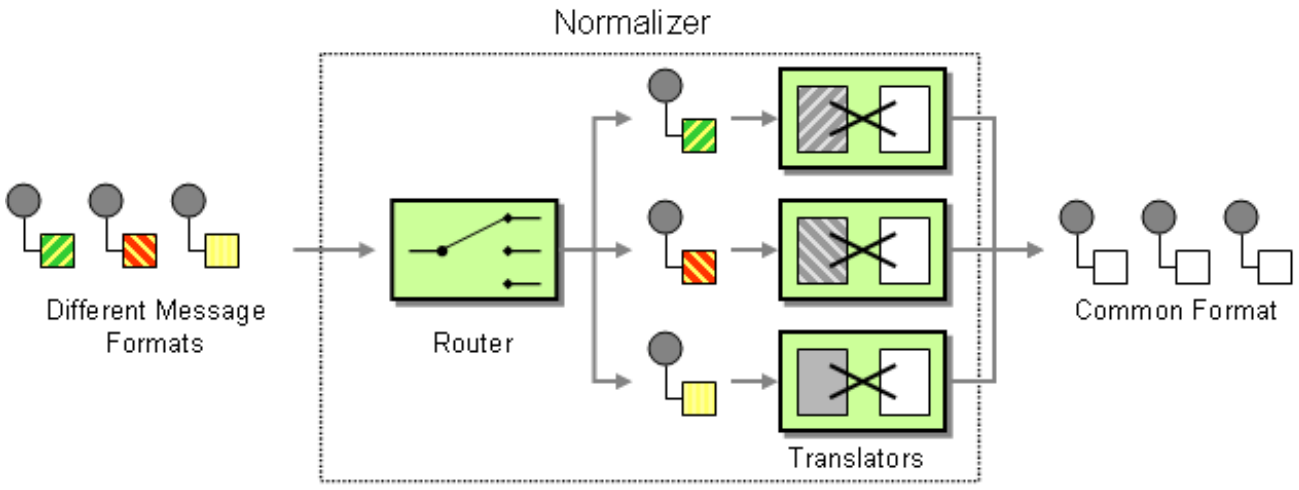
10.3. 规范化程序

概述

规范化程序 模式用于处理语义等效的消息，但到达不同的格式。规范化程序将传入的信息转换为通用格式。

在 Apache Camel 中，您可以通过组合使用 第 8.1 节 “基于内容的路由器” 来实现规范化程序模式，该 第 8.1 节 “基于内容的路由器” 检测到传入消息的格式，以及不同的 第 5.6 节 “消息 Translator” 的集合，它将不同的传入的格式转换为通用格式。

图 10.3. 规范化程序模式



Java DSL 示例

*本例演示了一个 **Message Normalizer**，它将两种类型的 **XML** 消息转换为通用格式。然后会过滤采用这种通用格式的消息。*

*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");
```

*在这种情况下，我们使用 **Java bean** 作为规范化程序。类类似如下*

```
// 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 中的同一示例

```
<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

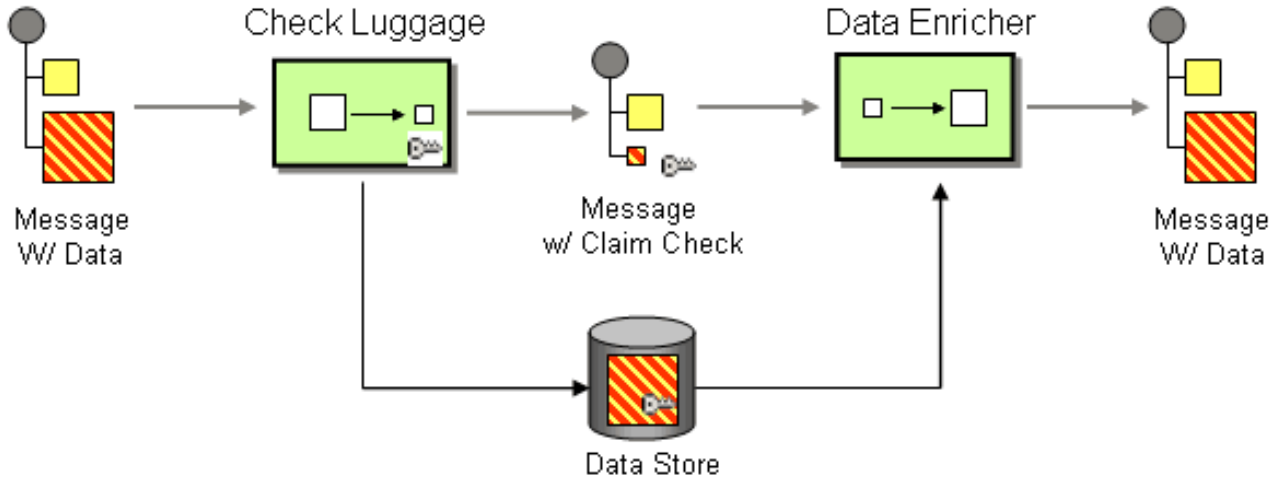
声明检查 EIP

声明检查 EIP 模式（在 图 10.4 “声明检查模式” 中显示的模式）允许您将消息内容替换为声明检查（一个唯一的键）。使用 声明检查 EIP 模式，稍后检索消息内容。您可以临时将消息内容保存在持久性存储中，如数据库或文件系统。当消息内容非常大，且昂贵的发送时，这种模式很有用，且不是所有组件都需要所有信息。

当您无法与外部方信任信息时，它也很有用。在这种情况下，使用 Claim Check 来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：

Name	描述	默认值	类型

operation	需要使用声明检查操作。它支持以下操作： * get- **Get** s （不会删除）给定密钥的声明检查。 * **GetAndRemove** - Gets 并删除给定密钥的声明检查。 * **set** - 使用给定键设置一个新的声明检查。如果密钥已存在，它将被覆盖。 * **push** - 设置堆栈上的一个新的声明检查（不要使用密钥）。 * pop- 从堆栈获取最新的声明检查（不要使用密钥）。 **使用 Get、GetAndRemove 或 Set 操作时，您必须指定一个密钥。然后，这些操作将使用密钥存储和检索数据。使用这些操作在不同的密钥中存储多个数据。但是，推送和 弹出 操作不使用密钥，而是将数据存储在堆栈结构中。**		ClaimCheckOperation
key	使用特定密钥进行声明 check-id。		字符串
filter	指定控制您要从声明检查存储库合并的数据的过滤器。		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同时配置数据。		字符串

过滤选项

使用 Filter 选项定义在使用 Get 或 Pop 操作时要合并的数据。使用 AggregationStrategy 将数据合并回来。默认策略使用 filter 选项，轻松指定要合并的数据。

filter 选项采用带有以下语法的 String 值：

- ***正文 ：聚合消息正文***
- ***attachments ：聚合所有消息附加***
- ***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（以 * 结尾且名称以模式开头）***
- ***正则表达式匹配***

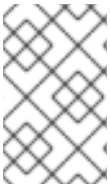
要指定多个规则，用 逗号 (、)将它们分隔。

以下是包括消息正文和以 foo 开头的所有标头的基本过滤器示例：

```
body, header:foo*
```

- *仅合并消息正文： body*
- *仅合并消息附加： 附加*
- *仅合并标头： 标头*
- *要合并标头名称 foo only: header:foo*

如果将过滤器规则指定为空或通配符，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

当您重新合并数据时，系统会覆盖任何现有的数据。此外，它还存储现有数据。

10.4.2. 使用 Include 和 Exclude Pattern 过滤选项

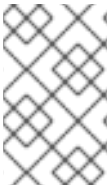
以下是支持用来指定 include,exclude, 或 remove 选项的前缀的语法。

- *+ : 要包括（这是默认模式）*
- *- : 要排除(exclude 优先于 include)*
- *-- : 要删除（删除具有优先权）*

例如：

- *要跳过消息正文并合并其他内容，use -body*
- *要跳过消息标头 foo 并合并其他内容，use- -header:foo*

您还可以指示系统在合并数据时删除标头。例如，要删除所有以 bar 开始的标头，use--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 密钥。

```
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 filter 选项，其中您只想将标题作为 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");
```

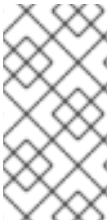
10.4.4. XML 示例

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 密钥。

```
<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 **filter** 选项将标头返回为 **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 模式用于对消息正文的内容进行排序，假设消息正文包含可排序的项目列表。

默认情况下，消息的内容使用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器，您可以指定一个返回列表的表达式，该表达式必须被排序（表达式必须转换为 **java.util.List**）。

Java DSL 示例

以下示例生成按行 **break** 字符中的令牌排序的项目列表：

```
from("file://inbox").sort(body().tokenize("\n")).to("bean:MyServiceBean.processLine");
```

您可以将您自己的比较器传递为 **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。

以下示例生成按行 **break** 字符中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，您可以将其引用为 **Spring bean**：

```
<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 **<simple >** 外，您可以使用您喜欢的任何语言提供表达式，只要它返回一个列表。

选项

sort DSL 命令支持以下选项：

Name	默认值	描述
comparatorRef		指的是用于排序消息正文的自定义 **java.util.Comparator**。默认情况下，Camel 将使用对 A.Z 排序进行的比较器。

10.6. 转换程序

转换程序根据路由定义上声明的 Input Type 和/或 Output Type 来执行消息的声明转换。默认 camel 消息实现 DataTypeAware，它保存由 DataType 表示的消息类型。

10.6.1. Transformer 的工作原理？

路由定义声明 Input Type 和/或 Output 类型。如果 Input Type 和/或 Output Type 在运行时与消息类型不同，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。一旦消息被成功转换，或者消息已经处于预期类型中后，则会更新消息数据类型。

10.6.1.1. 数据类型格式

数据类型的格式是 scheme:name，其中 scheme 是数据模型的类型，如 java、xml 或 json，name 是数据类型名称。

注意

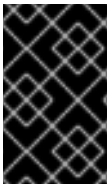
如果您只指定 scheme，则它将所有数据类型与该方案匹配。

10.6.1.2. 支持的 Transformers

转换程序	描述
数据格式转换	使用数据格式转换
端点转换	使用端点转换
自定义 Transformer	使用自定义转换器类转换。

10.6.1.3. 常见选项

所有转换程序都有以下通用选项来指定转换器支持的数据类型。



重要

必须指定 scheme 或 fromType 和 toType。

Name	描述
scheme	数据模型的类型，如 **xml** 或 **json**。例如，如果指定了 **xml**，则转换器将适用于所有 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

Name	描述
type	数据类型
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. 端点转换选项

Name	描述
ref	引用端点 ID

Name	描述
uri	端点 URI

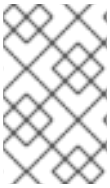
在 Java DSL 中指定端点 URI 的示例：

```
transformer()
   .fromType("xml")
   .toType("json")
   .withUri("dozer:myDozer?mappingFile=myMapping.xml...");
```

在 XML DSL 中指定 endpoint ref 的示例：

```
<transformers>
<endpointTransformer ref="myDozerEndpoint" fromType="xml" toType="json"/>
</transformers>
```

10.6.3. 自定义转换选项



注意

转换程序必须是 org.apache.camel.spi.Transformer的子类

Name	描述
ref	对自定义 Transformer bean ID 的引用
className	自定义 Transformer 类的完全限定域名

指定自定义 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. 转换程序示例

这个示例分为两个部分，第一部分声明 Endpoint Transformer，它将转换消息。第二部分显示了如何将转换器应用到路由。

10.6.4.1. 第一部分

声明 Endpoint Transformer，它使用 xslt 组件从 xml:ABCOrder 转换为 xml:XYZOrder。

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 端点将消息发送到 direct:xyz 时，上面的转换器将应用到以下路由定义：

Java DSL:

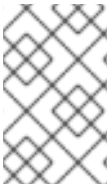
```
from("direct:abc")
   .inputType("xml:ABCOrder")
   .to("direct:xyz");
from("direct:xyz")
   .inputType("xml:XYZOrder")
   .to("somewhere:else");
```

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. VALIDATOR

验证器根据路由定义上声明的 Input Type 和/或 Output Type 来执行消息的声明验证，该定义声明了预期的消息类型。



注意

只有在 type 声明上的 validate 属性为 true 时才会执行验证。

如果在 Input Type 和/或 Output Type 声明上 validate 属性为 true，则 camel 内部处理器会从 registry 中查找对应的 Validator。

10.7.1. 数据类型格式

数据类型的格式是 scheme:name，其中 scheme 是数据模型的类型，如 java、xml 或 json，name 是数据类型名称。

10.7.2. 支持的验证器

validator	描述
predicate Validator	使用 Expression 或 Predicate 进行验证
端点验证器	通过转发到要与验证组件（如 Validation Component 或 Bean Validation 组件）一起使用的 Endpoint 验证。
自定义验证器	使用自定义验证器类验证。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含指定要验证的 Data type 的 type 选项。

10.7.4. predicate Validator 选项

Name	描述
expression	用于验证的 expression 或 Predicate。

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. 端点验证器选项

Name	描述

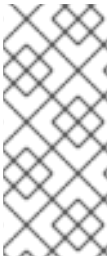
Name	描述
ref	引用端点 ID。
uri	端点 URI。

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 endpoint 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 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. 自定义验证器选项

注意

Validator 必须是 org.apache.camel.spi.Validator的子类

Name	描述
ref	对自定义 Validator Bean ID 的引用。
className	自定义 Validator 类的完全限定类名称。

指定自定义 Validator 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

这个示例分为两个部分，第一部分声明 Endpoint Validator 来验证消息。第二部分显示了如何将验证器应用到路由。

10.7.7.1. 第一部分

声明 Endpoint Validator，它使用验证器组件从 xml:ABCOrder 验证。

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 端点接收消息时，上面的验证器会应用到以下路由定义。

注意

在 Java DSL 中使用 inputType WithValidate 而不是 inputType，在 XML DSL 中将 inputType 声明上的 validate 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

validate 模式提供了一种方便的语法，用于检查消息的内容是否有效。validate DSL 命令使用 predicate 表达式作为其唯一参数：如果 predicate 被评估为 true，则路由会继续处理；如果 predicate 评估为 false，则会抛出 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");
```

您还可以验证消息标头 swig-wagon，例如：

```
from("jms:queue:incoming")
  .validate(header("bar").isGreaterThan(100))
  .to("bean:MyServiceBean.processLine");
```

*您可以使用 简单 表达式语言使用 **validate** ：*

```
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 **validate**，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"/>
```

*您还可以验证消息标头 **swig-wagon**，例如：*

```
<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平台中立平台。选择的消息格式应该适合通过 第 6.5 节 “消息总线” 传输，其中消息总线是集成各种不同系统的后果，其中一些可能不是面向对象的。

许多不同的方法可以，但并非所有方法都满足消息传递映射器的要求。例如，传输对象的一个明显方法是使用 对象序列化，它可让您使用无误编码（在 Java 中原生支持）将对象写入数据流。但是，这不是用于消息传递映射器模式的适当方法，因为序列化格式仅被 Java 应用理解。Java 对象序列化会在原始应用程序以及消息传递系统中其他应用程序之间产生异常不匹配。

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该适合由面向非对象的应用程序使用。
- 映射器代码应该独立于域对象代码和消息传递基础架构进行单独实施。Apache Camel 通过提供 hook 来帮助满足此要求，该 hook 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特定面向对象的概念的有效方法，如继承、对象引用和对象树。这些问题的复杂性因应用程序而异，但映射器实施的动画必须始终是创建能被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：

- 查找注册的 bean. HEKETI-5-4 用于单例对象和少量对象，您可以使用 CamelContext registry 存储对 Bean 的引用。例如，如果使用 Spring XML 实例化 bean 实例，它会自动输入

到 registry 中，其中 bean 由其 id 属性的值标识。

- 使用 JoSQL 语言选择对象。 如果要访问的所有对象已在运行时实例化，您可以使用 JoSQL 语言来查找特定的对象（或对象）。例如，如果您有一个类 org.apache.camel.builder.sql.Person，具有 name bean 属性并且传入的消息具有 UserName 标头，您可以使用以下代码选择 name 属性等于 UserName 标头的值的对象：

```
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);
```

语法是 :HeaderName，用于替换 JoSQL 表达式中的标头值。

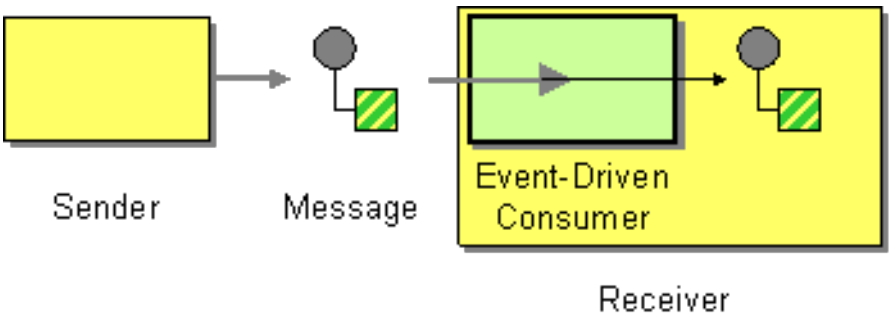
- 对于更可扩展的解决方案，可能需要从数据库中读取对象数据。在某些情况下，现有面向对象的应用可能已经提供可以从数据库中加载对象的查找器对象。在其他情况下，您可能需要编写一些自定义代码以从数据库提取对象，在这种情况下，JDBC 组件和 SQL 组件可能很有用。

11.2. EVENT DRIVEN CONSUMER

概述

事件驱动的消费者 模式（如 图 11.1 “event Driven Consumer Pattern” ）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人员相关。现有组件已经有消费者实施模式，对它们有硬连接。

图 11.1. event Driven Consumer Pattern



符合此模式的消费者提供了一种事件方法，它会在收到传入消息时自动由消息传递频道或传输层调用。事件驱动的消费者模式的一个特征是消费者端点本身不提供任何线程来处理传入的消息。相反，底层传输或消息传递频道在调用公开事件方法时隐式提供处理器线程（在消息处理期间这个块）。

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

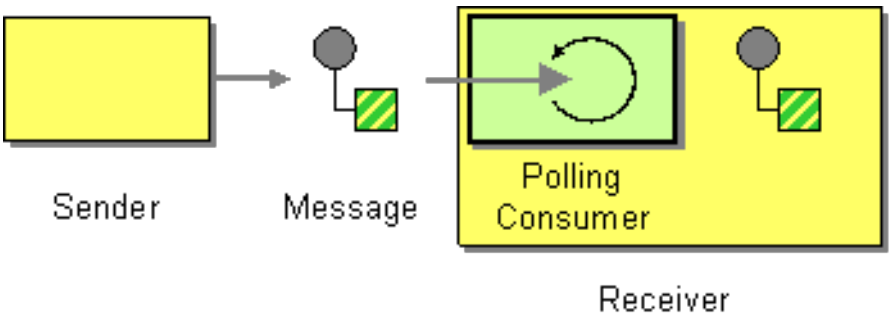
概述

图 11.2 “polling Consumer Pattern” 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，因此仅与需要在 Apache Camel 中开发自定义组件的编程人员相关。现有组件已经有消费者实施模式，对它们有硬连接。

符合此模式的消费者会公开轮询方法、receive （）、receive (long timeout) 和 receiveNoWait （），如果可以从被监控的资源获得新的交换对象。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 模板”。

图 11.2. polling Consumer Pattern



调度的轮询消费者

许多 Apache Camel 使用者端点使用调度的轮询模式来在路由开始时接收消息。也就是说，端点似乎实现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，请查看 第 41.2 节 “实施 Consumer 接口”。

quartz 组件

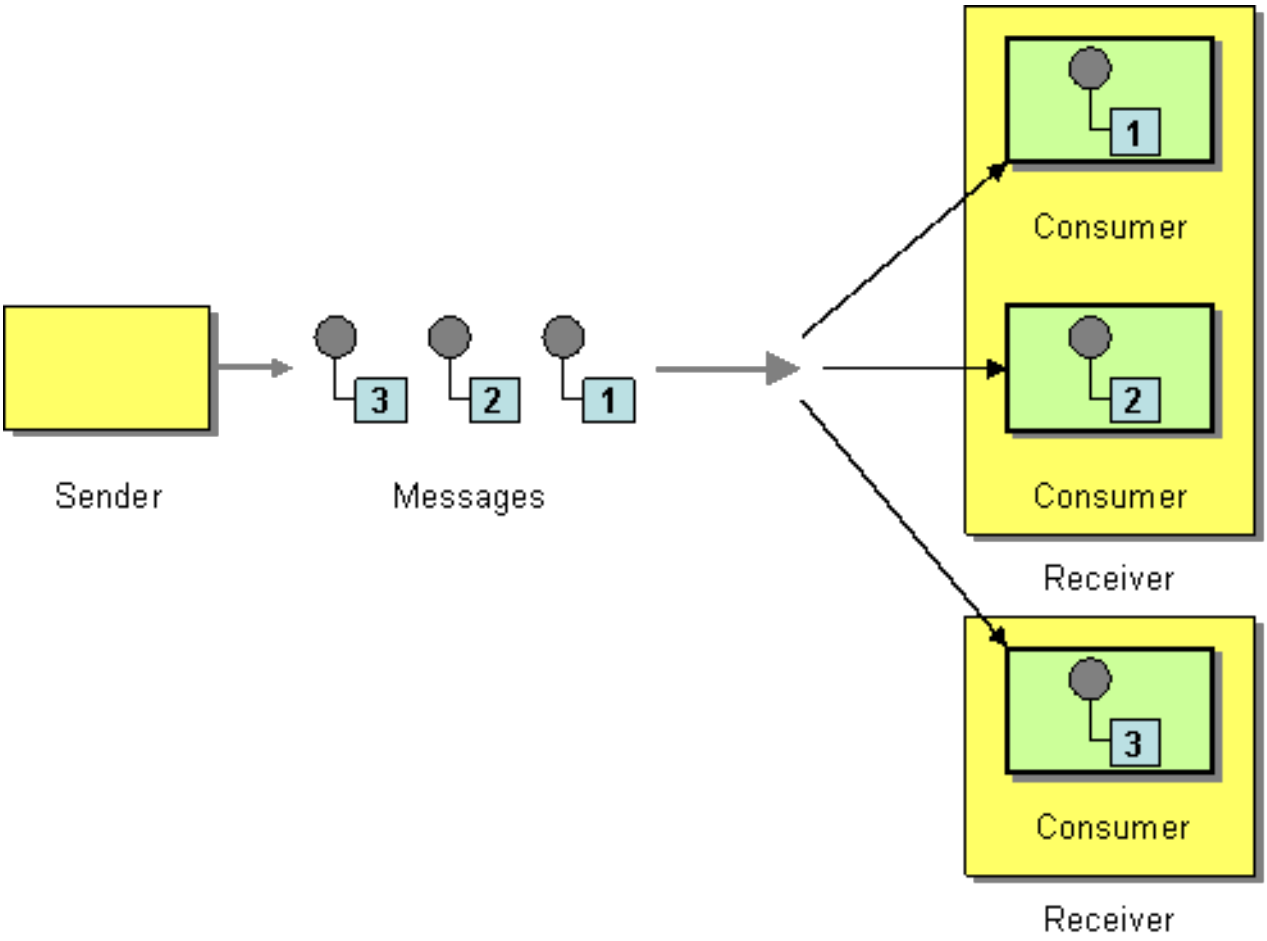
您可以使用 quartz 组件，通过 Quartz 企业调度程序提供调度的消息交付。详情请参阅 Apache Camel 组件参考指南 和 Quartz 组件中的 Quartz。

11.4. 竞争消费者

概述

图 11.3 “竞争消费者模式” 中显示的 竞争消费者 模式使多个用户能够从同一队列拉取信息，保证 每个消息仅消耗一次。此模式可用于将串行消息处理替换为并发消息处理（从而降低响应延迟）。

图 11.3. 竞争消费者模式



以下组件演示了竞争消费者模式：

- 基于 JMS 的竞争消费者
- 基于 SEDA 的竞争消费者

基于 JMS 的竞争消费者

常规 JMS 队列隐式保证每个消息只能被一次使用。因此，JMS 队列会自动支持竞争消费者模式。例如，您可以定义三个从 JMS 队列 HighVolumeQ 拉取消息的竞争用户，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、replica 01、replica02 和 replica03，并行处理来自 HighVolumeQ 队列的消息。

或者，您也可以设置 JMS 查询选项 concurrentConsumers，以创建竞争消费者的线程池。例如，以下路由创建了三个竞争线程池，从指定队列中提取消息：

```
from("jms:HighVolumeQ?concurrentConsumers=3").to("cxf:bean:replica01");
```

可以在 XML DSL 中指定 concurrentConsumers 选项，如下所示：

```
<route>
  <from uri="jms:HighVolumeQ?concurrentConsumers=3"/>
  <to uri="cxf:bean:replica01"/>
</route>
```

注意

JMS 主题 不支持竞争消费者模式。按照定义，JMS 主题旨在向不同的消费者发送同一消息的多个副本。因此，它与竞争消费者模式不兼容。

基于 SEDA 的竞争消费者

SEDA 组件的目的是通过将计算拆分为阶段来简化并发处理。SEDA 端点基本上封装了内存阻塞队列（由 java.util.concurrent.BlockingQueue实施）。因此，您可以使用 SEDA 端点将路由分为多个阶段，每个阶段都可能使用多个线程。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：一个线程，它路由到 cxf:bean:replica01，一个线

程路由到 cxf:bean:replica02，以及一个路由交换到 cxf:bean:replica03 的线程。这三个线程竞争从 SEDA 端点获取交换实例，该端点使用阻塞队列来实施。因为阻塞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只能被消耗一次。

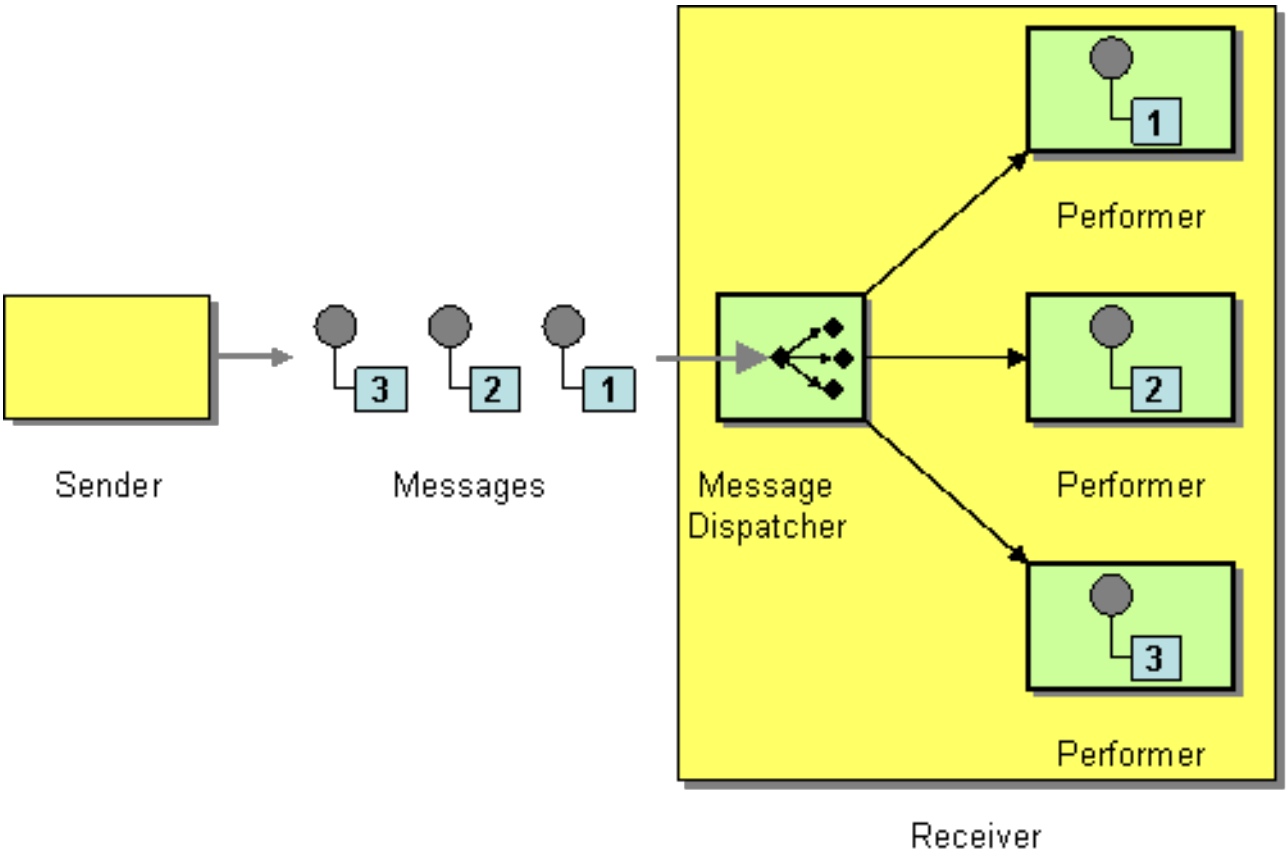
有关 SEDA 端点和由 thread （） 创建的线程池之间的区别，请参阅 Apache Camel 组件参考指南中的 SEDA 组件。

11.5. MESSAGE DISPATCHER

概述

图 11.4 “消息分配器模式” 中显示的消息分配程序 模式用于消耗频道中的消息，然后在本地分发它们以执行方，后者负责处理信息。在 Apache Camel 应用程序中，执行者通常由进程端点代表，用于将消息传送到路由的另一个部分。

图 11.4. 消息分配器模式



您可以使用以下方法之一在 Apache Camel 中实施消息分配程序模式：

- JMS 选择器

- ActiveMQ 中的 JMS 选择器
- 基于内容的路由器

JMS 选择器

如果应用使用来自 JMS 队列的消息，您可以使用 JMS 选择器 实施消息分配程序模式。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 消息到达消费者，如果选择器评估为 false，则 JMS 消息被阻止。在很多方面，JMS 选择器类似于 第 8.2 节 “Message Filter”，但它具有在 JMS 提供程序中实施过滤的额外优势。这意味着 JMS 选择器可以在消息传输到 Apache Camel 应用程序之前阻止消息。这带来了显著的效率优势。

在 Apache Camel 中，您可以通过对 JMS 端点 URI 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 基于 SQL92 条件表达式语法的子集（有关完整详情，请参阅 JMS 规格）。选择器字符串中显示的标识符可以引用 JMS 标头或 JMS 属性。例如，在前面的路由中，发送者会设置一个名为 CountryCode 的 JMS 属性。

如果要将 JMS 属性添加到 Apache Camel 应用程序内的消息中，您可以通过设置消息标头(在 In message 或 Out 消息上)来完成此操作。在读取或写入到 JMS 端点时，Apache Camel 会将 JMS 标头和 JMS 属性映射到其原生消息标头。

从技术上讲，选择器字符串必须根据 application/x-www-form-urlencoded MIME 格式进行 URL 编码（请参阅 HTML 规格）。在实践中，& (ampersand)字符可能会导致困难，因为它用于限制 URI 中的每个查询选项。对于可能需要嵌入 & amp; 字符的更复杂的选择器字符串，您可以使用 java.net.URLEncoder 实用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。

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 Selectors 和 ActiveMQ 消息属性。

基于内容的路由器

基于内容的路由器模式和消息分配器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），以及消息分配程序在本地分配消息，在同一进程空间内。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，路由会定义一个消息分配程序。

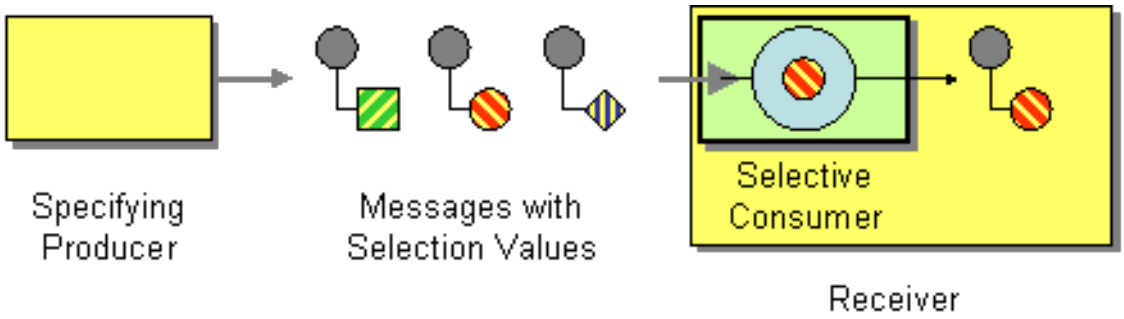
有关如何使用基于内容的路由器模式的详情和示例，请参考 第 8.1 节 “基于内容的路由器”。

11.6. SELECTIVE CONSUMER

概述

图 11.5 “selective Consumer Pattern” 中显示的特定 消费者 模式描述了将过滤器应用到传入信息的消费者，以便仅处理满足特定选择条件的消息。

图 11.5. selective Consumer Pattern



您可以使用以下方法之一在 Apache Camel 中实施选择性消费者模式：

- JMS 选择器

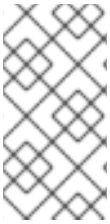
- ActiveMQ 中的 JMS 选择器
- Message filter

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 消息到达消费者，如果选择器评估为 false，则 JMS 消息被阻止。例如，要使用来自队列的消息（选择），并仅选择其国家代码属性等于 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");
```

其中选择器字符串 CountryCode='US' 必须采用 URL 编码（使用 UTF-8 字符），以避免解析查询选项时出现问题。本例假定 JMS 属性 CountryCode 是由发送者设置的。有关 JMS 选择器的详情，请参考 "JMS 选择器"一节。



注意

如果选择器应用到 JMS 队列，未选择的消息将保留在队列中，并且可能可供附加到同一队列的其他消费者使用。

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 Selectors 和 ActiveMQ 消息属性。

Message filter

如果无法在消费者端点上设置选择器，您可以在路由中插入过滤器处理器。例如，您可以定义一个选择消费者，该消费者仅使用 Java DSL 来处理具有美国国家代码的消息，如下所示：

```
from("seda:a").filter(header("CountryCode").isEqualTo("US")).process(myProcessor);
```

可以使用 XML 配置定义相同的路由，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

注意使用消息过滤器从 JMS 队列 选择消息。使用过滤器处理器时，会直接丢弃被阻止的消息。因此，如果消息从队列消耗（允许每个消息仅被消耗一次），则根本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

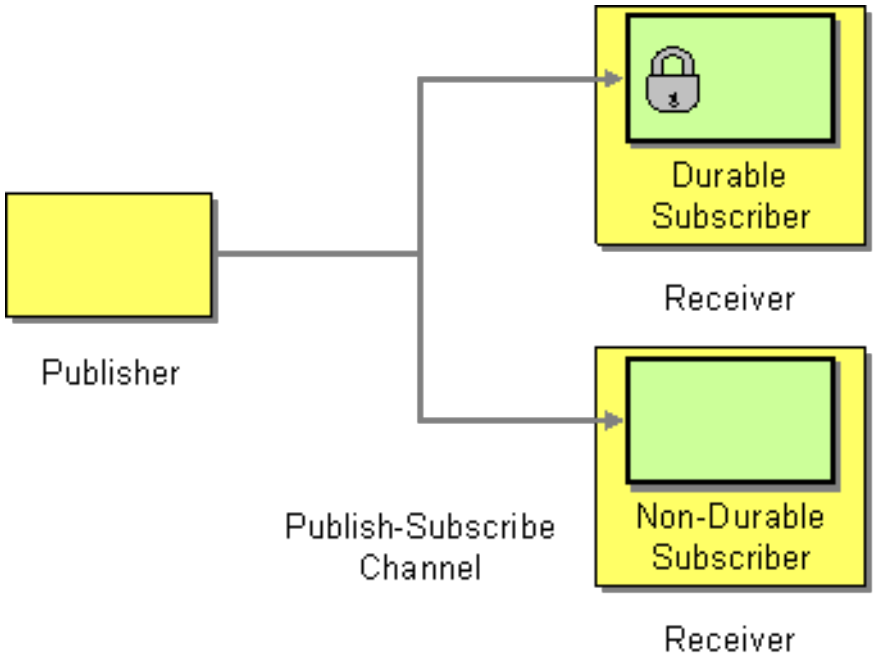
如 图 11.6 “durable Subscriber 模式” 所示，持久化 订阅者 是一个消费者，它希望接收通过特定 第 6.2 节 “publish-Subscribe Channel” 频道发送的所有消息，包括在消费者与消息传递系统断开连接时发送的消息。这要求消息传递系统将消息保存到断开连接的消费者中。另外，还必须有一种机制来表示它想建立持久订阅。通常，发布订阅频道（或主题）可以同时具有 durable 和 non-durable 订阅者，其行为如下：

- 非持久化订阅者 HEKETI -wagonCan 有两个状态： connect 和 disconnected。虽然非持久化订阅者连接到主题，但会实时接收所有主题的消息。但是，在订阅者断开连接时，非持久化订阅者永远不会接收发送到主题的消息。

- durable subscriber swig- swigCan 有两个状态： connect 和 inactive。inactive 状态表示 durable subscriber 与主题断开连接，但希望接收到达异常的消息。当 durable 订阅者重新连接

到主题时，它会收到在不活跃期间发送的所有消息重播。

图 11.6. durable Subscriber 模式



JMS durable subscriber

JMS 组件实施持久订阅者模式。要在 JMS 端点上设置持久化订阅，您必须指定客户端 ID，该客户端 ID 标识此特定连接，以及可识别 持久 订阅者的持久订阅名称。例如，以下路由将持久订阅设置为 JMS 主题、新闻报道，客户端 ID 为 conn01，持久订阅名称为 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 端点将路由桥接到多个并行片段中，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");
```

每个消息仅处理一次，因为 SEDA 组件支持 竞争消费者 模式。

备用示例

另一种方法是，将 第 11.5 节 “Message Dispatcher” 或 第 8.1 节 “基于内容的路由器” 与 File 或 JPA 组件用于持久订阅者，然后使用 SEDA 进行非持久性。

以下是为 JMS 主题创建持久订阅者的简单示例

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. IDEMPOTENT CONSUMER

概述

幂等的消费者 模式用于过滤重复的消息。例如，假设消息传递系统和消费者端点之间的连接由于系统中的某种故障而完全丢失。如果消息传递系统在传输消息的中间，这可能并不明确，无论消费者是否收到最后一条消息。为提高交付可靠性，在重新建立连接后，消息传递系统可能会决定重新建立此类消息。不幸的是，这需要消费者可能会收到重复消息的风险，在某些情况下，复制信息的影响可能会存在不必要的后果（例如从您的帐户中减去了一倍的费用）。在这种情况下，可以使用幂等消费者从消息流中去除不必要的重复项。

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，幂等消费者模式由 idempotentConsumer （） 处理器实现，该处理器采用两个参数：

- *messageIdExpression wagon- swig An 表达式，该表达式返回当前消息的消息 ID 字符串。*
- *messageIdRepository wagon-wagon A 引用消息 ID 存储库，该存储库存储收到的所有消息的 ID。*

当每条消息都出现时，幂等消费者处理器会在存储库中查找当前消息 ID，以查看之前是否看到了此消息。如果为 yes，则会丢弃消息；如果没有，则允许消息通过，并将其 ID 添加到存储库中。

例 11.1 “使用内存缓存过滤重复的消息” 中显示的代码使用 TransactionID 标头过滤掉重复项。

例 11.1. 使用内存缓存过滤重复的消息

```
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");
    }
};
```

如果调用 **memoryMessageIdRepository (200)**，它会创建一个可以最多 **200** 个消息 **ID** 的内存中缓存。

您还可以使用 **XML** 配置定义幂等消费者。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 repository 的幂等消费者

内存缓存不会造成轻松耗尽内存的缺点，且在集群环境中无法正常工作。要克服这些缺点，您可以使用基于 **Java Persistent API (NFD)**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 swig-wagonProvides 是 JPA 数据库的句柄。
- 处理器名称 swig-rhacmIdent 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 数据库提供句柄。有关如何配置此 bean 的详细信息，请参阅 JPA 文档。

有关设置 JPA 存储库的更多详细信息，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 标头过滤掉重复项：

```
<!-- 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>
```

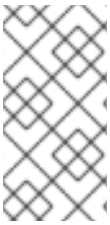
使用 JDBC 存储库的幂等消费者

也支持 JDBC 存储库将消息 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 JDBC 打包程序类来实例化到 SQL 数据库的连接。例如，要实例化 JDBC 连接到 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，它会创建一个仅内存的数据库实例。这是 HyperSQL 数据库的一个迭代实现，它实际上 不是 持久的。

使用前面的数据源，您可以定义使用 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消息

从 Camel 2.8 开始提供

现在，您可以将 skipDuplicate 选项设置为 false，它指示幂等消费者也路由重复消息。但是，重复的消息已通过将 “Exchanges”一节 中的属性设置为 true 来标记为重复。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” 来检测这个事实并处理重复的信息。

例如，在以下示例中，我们使用 第 8.2 节 “Message Filter” 将消息发送到重复的端点，然后停止该消息。

```
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 中的示例为：

```
<!-- 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，则内存幂等存储库中的 不起作用（请参阅上述内容）。您可以设置中央数据库，或使用基于 Hazelcast 数据网格的幂等消费者实施。Hazelcast 发现节点多播（默认为 tcp-ip）配置 Hazelcast，并创建一个基于映射的存储库：

```
HazelcastIdempotentRepository idempotentRepo = new HazelcastIdempotentRepository("myrepo");

from("direct:in").idempotentConsumer(header("messageId"), idempotentRepo).to("mock:out");
```

您必须定义存储库应保留每个消息 ID 的时长（默认是删除它）。为避免内存不足，您应该根据 Hazelcast 配置创建 驱除策略。如需更多信息，请参阅 Hazelcast。

请参阅此链接：http://camel.apache.org/hazelcast-idempotent-repository-tutorial.html[Idempotent 仓库

教程]了解如何使用 Apache Karaf 在两个集群节点上设置此类幂等存储库。

选项

Idempotent Consumer 带有以下选项：

选项	默认	描述
eager	**true**	Camel 2.0： Eager 控制 Camel 是否在处理交换前或之后将消息添加到存储库中。如果在随后启用之前启用，则 Camel 能够检测到重复的消息，即使消息当前正在进行中。禁用 Camel 只有在成功处理消息时才会检测到重复。
messageIdRepositoryRef	**null**	对注册表中查找的 **IdempotentRepository** 的引用。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。如果设置为 **false**，则消息将继续。但是，“Exchanges”一节 已通过将 **Exchange.DUPLICATE_MESSAG** Exchange 属性设置为 **布尔值**。

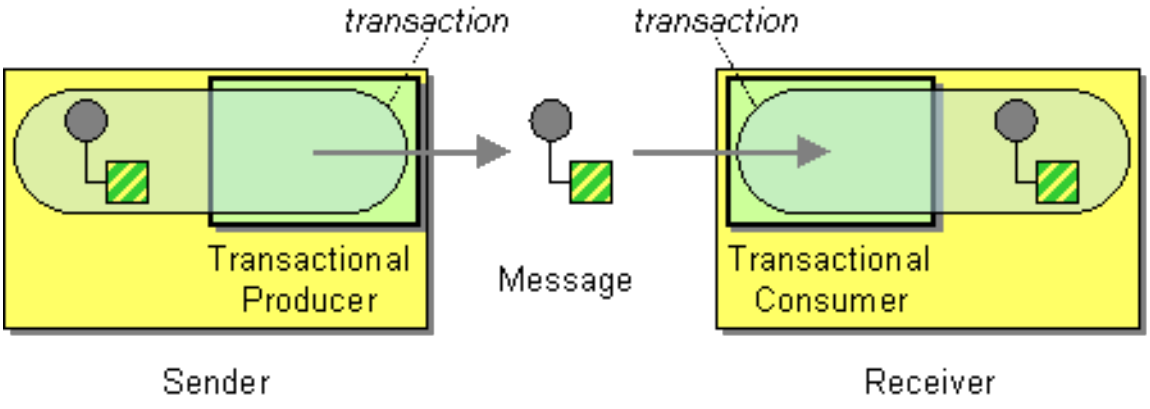
completionEager	**false**	Camel 2.16 ： 当交换完成后，设置是否要完成 Idempotent consumer eager。 如果您设置了 **completeEager** 选项 true，则 Idempotent Consumer 会在交换到达幂等消费者模式块的末尾时触发其完成。但是，如果交换在结束块后继续路由，则它不会影响幂等消费者的状态。 如果您设置了 **completeEager** 选项 false，则 Idempotent Consumer 会在交换完成后触发其完成，并被路由。但是，如果交换在块结束后仍然继续路由，那么它也会影响幂等消费者的状态。例如，由于交换失败，因此幂等消费者的状态将是回滚。

11.9. 事务客户端

概述

图 11.7 “事务客户端模式” 中显示的 事务客户端 模式指的是可参与事务的消息传递端点。Apache Camel 支持使用 Spring 事务管理 的事务。

图 11.7. 事务客户端模式



事务导向端点

并非所有 Apache Camel 端点都支持事务。它们被称为 事务导向端点(或 TOEs)。例如，JMS 组件和 ActiveMQ 组件都支持事务。

要在组件上启用事务，您必须在将组件添加到 CamelContext 之前执行适当的初始化。这要求编写代码来明确初始化您的事务组件。

参考

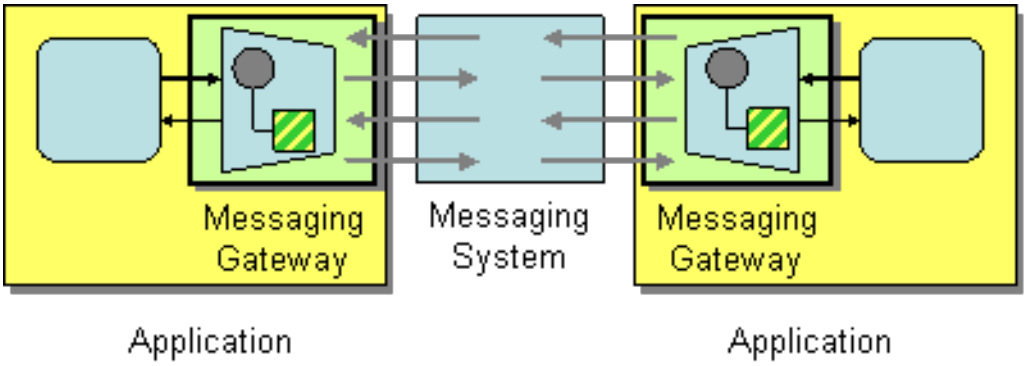
在 Apache Camel 中配置事务的详情超出了本指南的范围。有关如何使用事务的详情，请查看 Apache Camel 事务指南。

11.10. 消息传递网关

概述

图 11.8 “消息传递网关模式” 中显示的 消息传递网关 模式描述了与消息传递系统集成的一种方法，其中消息传递系统的 API 在应用程序级别从程序员中保持隐藏状态。一个比较常见的例子是，当您要将同步方法调用转换为请求/回复消息交换时，程序会通知这个信息。

图 11.8. 消息传递网关模式



以下 Apache Camel 组件提供这种类型的与消息传递系统集成：

- CXF
- Bean 组件

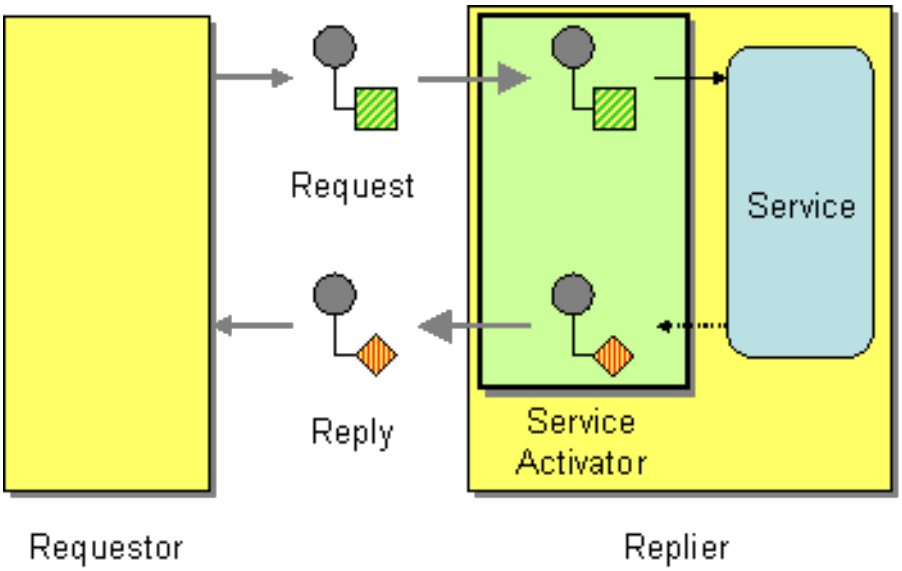
11.11. 服务激活器

概述

服务 activator 模式（如 图 11.9 “服务激活器模式” 所示）描述了在响应传入请求消息时调用服务操作的情况。服务 activator 标识要调用的操作，并提取要用作操作参数的数据。最后，服务激活器使用从消

息中提取的数据调用操作。操作调用可以是单向（仅限请求）或双向（请求/回复）。

图 11.9. 服务激活器模式



在很多方面，服务激活器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，服务激活器需要更灵活。RPC 框架标准化请求和回复消息编码（例如，Web 服务操作被编码为 SOAP 消息），而服务活动器通常还需要降低消息传递系统与服务操作之间的映射。

Bean 集成

Apache Camel 为支持服务激活器模式提供的主要机制是 bean 集成。Bean 集成 提供了一个常规框架，用于将传入的消息映射到 Java 对象上方法调用。例如，Java fluent DSL 提供处理器 bean （） 和 beanRef （），您可以插入到路由中，以在注册的 Java bean 上调用方法。消息数据到 Java 方法参数的详细映射由 bean 绑定 决定，这可以通过添加注释到 bean 类来实现。

例如，请考虑以下路由，调用 Java 方法 bank Bean.getUserAccBalance （），以服务在 JMS/ActiveMQ 队列中传入的请求：

```
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 中提取的消息具有简单的 XML 格式，提供银行帐户的用户 ID。例如：

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

```
  <BalanceQuery>
    <UserID>James.Strachan</UserID>
  </BalanceQuery>
</Account>
```

路由 setProperty （） 中的第一个处理器从 In 消息中提取用户 ID，并将其存储在 userid Exchange 属性中。最好将其存储在标头中，因为调用 bean 后 In 标头不可用。

服务激活步骤由 beanRef （） 处理器执行，它将传入消息绑定到由 bankBean bean ID 标识的 Java 对象的 getUserAccBalance （） 方法。以下代码显示了 bank Bean 类的实施示例：

```
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;
        }
      }
}
```

其中，消息数据绑定到方法参数的绑定由 @XPath 注释启用，它会将 UserID XML 元素的内容注入到 user method 参数中。在完成调用后，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，您可以在 META-INF/spring/camel-context.xml 配置文件中添加以下行，以实例化 bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <bean id="bankBean" class="tutorial.BankBean"/>
</beans>
```

其中 bean ID, bankBean, 在注册表中标识此 bean 实例。

bean 调用的输出注入 Velocity 模板，以生成正确格式化的结果消息。Velocity 端点 velocity:file:src/scripts/acc_balance.vm，使用以下内容指定 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>
```

交换实例作为 Velocity 变量 Exchange 提供，它可让您使用 ${ exchange.getProperty (" userid ")} 来检索 userid Exchange 属性。当前 In 消息(${body})的正文包含 getUserAccBalance （） 方法调用的结果。

第 12 章 系统管理

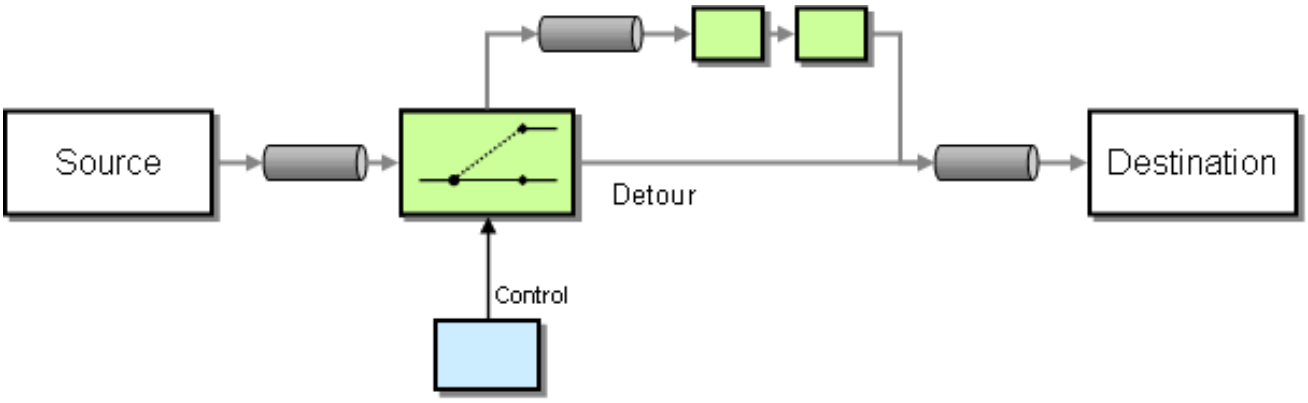
摘要

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

第 3 章 企业级集成模式简介 中的 Detour 允许您在满足控制条件时通过额外的步骤发送消息。在需要时，启用额外验证、测试、调试代码会很有用。



Example

在本例中，我们基本上有一个类似于 from ("direct:start").to ("mock:result") 的路由，有条件 detour 到路由中间 mock: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>
```

ControlBean 决定该 detour 是否处于打开状态。因此，当消息进入时，信息被路由到 mock:detour，然后 mock:result。当 detour 变为 off 时，消息会被路由到 mock:result。

有关完整详情，请查看示例源：

camel-core/src/test/java/org/apache/camel/processor/DetourTest.java

12.2. LOGEIP

概述

Apache Camel 提供了几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，该组件可以记录消息内容。
- 使用 Tracer，它跟踪消息流。
- 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 非常轻量，用于记录人工日志，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记录组件。Log 组件可以记录消息本身，您有许多 URI 选项来控制日志记录。

Java DSL 示例

从 Apache Camel 2.2 开始，您可以使用 log DSL 命令在运行时使用 Simple 表达式语言构造日志消息。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 String 格式消息。该日志消息将在 INFO 级别记录，并将路由 ID 用作日志名称。默认情况下，路由被连续命名，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，供您明确设置日志级别和日志名称。例如，要将日志级别明确设置为 LoggingLevel.DEBUG，您可以调用日志 DSL，如下所示：

日志 DSL 有过载方法来设置日志级别和/或名称。

```
from("direct:start").log(LoggingLevel.DEBUG, "Processing ${id}").to("bean:foo");
```

要将日志名称设置为 fileRoute，您可以调用日志 DSL，如下所示：

```
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 log 元素表示，并通过将 message 属性设置为 Simple 表达式来指定日志消息，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。由于 Apache Camel 2.17 通过配置 logname 参数可以更改日志名称。

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>
```

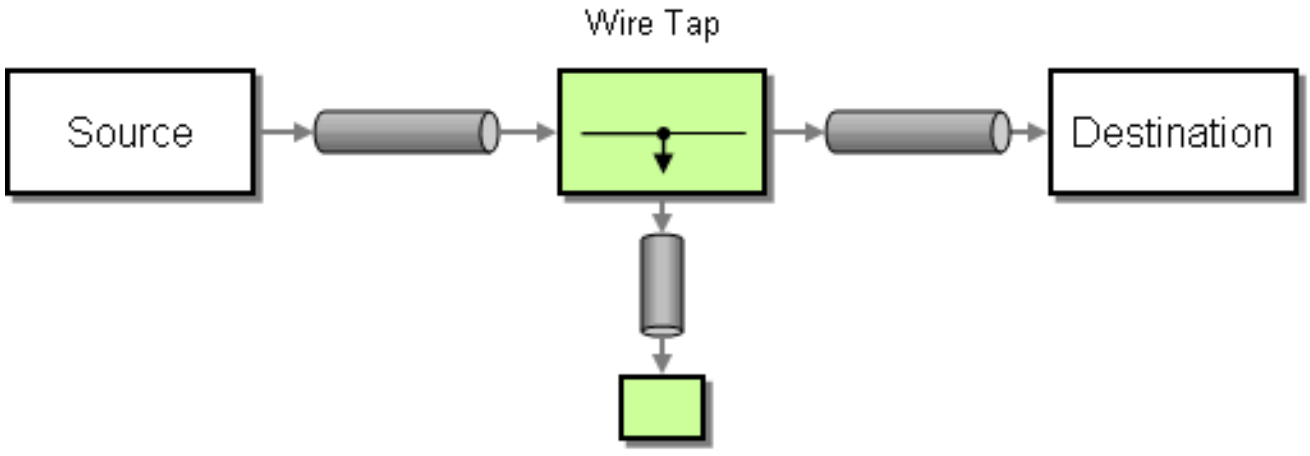
如果您有多个日志，并且您希望在所有这些日志中都有相同的日志名称，您必须将配置添加到每个日志中。

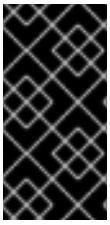
12.3. WIRE TAP

wire Tap

如 图 12.1 “Wire Tap Pattern” 所示，有线 tap 模式允许您将消息副本路由到单独的 tap 位置，原始消息被转发到最终目的地。

图 12.1. Wire Tap Pattern





流

如果您 打开 流消息正文，您应该考虑启用 流缓存，以确保消息正文可以被重新读取。请参阅 流缓存的详情

Wiretap 节点

Apache Camel 2.0 引入了用于进行线 tap 的 wireTap 节点。wireTap 节点将原始交换复制到被利用的交换，其交换模式被设置为 InOnly，因为已用的交换应以 单向 方式传播。被利用的交换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：

- 原始交换的副本。
- TAP 一个新的交换实例，允许您自定义已利用的交换。

注意

从 Camel 2.16 中，Wire Tap EIP 会在将交换发送到有线 tap 目的地时发出事件通知。

注意

自 Camel 2.20 起，Wire Tap EIP 将在关机时完成任何动态连接利用的交换。

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 并修改原始交换的副本

使用 Java DSL，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。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 一个新的交换实例

您可以通过将 **copy** 标志设置为 **false** （默认为 **true**）来定义带有新交换实例的 **wiretap**。在这种情况下，会为有线器创建一个初始空的交换。

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");
```

其中第二个 **wireTap** 参数将复制标志设置为 **false**，这表示没有复制原始交换，而是 创建空交换。

使用 表达式 方法创建新的交换实例：

```
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 **processor** 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 方法创建新的交换实例：

```
<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

从 Camel 2.8 开始提供

如果您使用 第 12.3 节 “wire Tap” 发送新消息，则只能使用 DSL 中的 第 II 部分 “路由表达式和 predicates 语言” 设置消息正文。如果您还需要设置新标头，则必须为此使用 第 1.5 节 “Processors”。因此，在 Camel 2.8 中，我们改进了这种情况，现在您可以在 DSL 中设置标头。

以下示例发送有的新消息

- "bye World"作为邮件正文
- 带有键 "id" 的标头，值为 123
- 具有键 "date" 的标头，其当前日期为 value

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

wire Tap 支持静态和动态端点 URI。静态端点 URI 从 Camel 2.20 开始提供。

以下示例演示了如何将 tap 连接至 JMS 队列，其中标头 ID 是队列名称的一部分。

```
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。

在准备信息时，使用 onPrepare 执行自定义逻辑

从 Camel 2.8 开始提供

详情请查看 第 8.13 节 “多播”。

选项

wireTap DSL 命令支持以下选项：

Name	默认值	描述
uri		端点 uri，其中发送有线 tapped 消息。您应使用 **uri** 或 **ref**。
ref		指的是发送有线利用消息的端点。您应使用 **uri** 或 **ref**。
executorServiceRef		指的是在处理有线信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		指的是用于创建新消息（如发送新消息模式）的自定义 第 1.5 节 “Processors”。请参见以下信息。
复制	**true**	Camel 2.3 ：应该有线连接消息时使用的 “Exchanges”一节 副本。
onPrepareRef		Camel 2.8： 请参阅自定义 第 1.5 节 “Processors” 来准备 “Exchanges”一节 的副本。这可让您执行任何自定义逻辑，如在需要时深度获取消息有效负载。

部分 II. 路由表达式和 PREDICATES 语言

本指南描述了 Apache Camel 支持的评估语言使用的基本语法。

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 predicates 语言” 概述调用表达式和 predicate 语言的不同语法。

表 13.1. 表达式和 predicates 语言

语言	静态方法	fluent DSL 方法	XML 元素	注解	工件
请参阅客户门户网站中的 *Apache Camel 开发指南*中的 *Bean 集成*。	**bean()**	***EIP*().method()**	**方法**	**@Bean**	Camel core
第 14 章 *常数*	**constant()**	***EIP*().constant()**	**constant**	**@Constant**	Camel core
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 core
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*	None	***EIP*().jsonpath()**	**jsonpath**	**@JsonPath**	**camel-jsonpath**
第 22 章 *jxpath*	None	***EIP*().jxpath()**	**jxpath**	**@JXPath**	**camel-jxpath**

语言	静态方法	fluent DSL 方法	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 (DEPRECATED)*	**php()**	***EIP*().php()**	**php**	**@PHP**	**camel-script**
第 26 章 *Exchange Property*	**property()**	***EIP*().property()**	**属性**	**@Property**	Camel core
第 27 章 *Python (DEPRECATED)*	**python()**	***EIP*().python()**	**python**	**@Python**	**camel-script**
第 28 章 *Ref*	**ref()**	***EIP*().ref()**	**ref**	N/A	Camel core
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 core
第 31 章 *SpEL*	**spel()**	***EIP*().spel()**	**spel**	**@SpEL**	**camel-spring**
第 32 章 *XPath 语言*	**xpath()**	***EIP*().xpath()**	**XPath**	**@XPath**	Camel core
第 33 章 *XQuery*	**xquery()**	***EIP*().xquery()**	**XQuery**	**@XQuery**	**camel-saxon**

13.2. 如何中断表达式语言

先决条件

在使用特定的表达式语言之前，您必须确保 classpath 上有所需的 JAR 文件。如果 Apache Camel 内核中没有包括您要使用的语言，您必须将相关的 JAR 添加到 classpath 中。

如果使用 Maven 构建系统，只需将相关依赖项添加到 POM 文件中来修改 build-time classpath。例如，如果要使用 Ruby 语言，请在 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>
```

如果您要在红帽 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（功能在对应的 Maven 工件后被命名）。例如，要在 OSGi 容器中使用 Groovy 语言，您必须首先通过输入以下 OSGi 控制台命令来安装 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:file:path 将值引用为外部资源。例如：resource:classpath:com/foo/myscript.groovy。

Camel on EAP 部署

camel-groovy 组件由 EAP 上的 Camel (Wildfly Camel)框架支持，该框架在 Red Hat JBoss Enterprise Application Platform (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 predicates 语言” 所示，调用表达式语言有几个不同的语法，具体取决于所用的上下文。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。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 Predicate 对象）。

例如，要实施以 XML 格式处理消息的基于内容的路由器，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样式。您可以向 DSL 命令的子层提供表达式，而不是提供表达式作为企业集成模式(EIP)的参数。例如，您可以将表达式调用为 filter (xpath ("Expression")，而不是调用 XPath 表达式，而是以 filter ().xpath ("Expression") 调用表达式。

例如，在这种调用方式中，上述基于内容的路由器可以重新实施，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 XML 元素中，在 XML 中调用表达式语言。

例如，在 XML 中调用 XPath 的 XML 元素为 xpath （属于标准的 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，您可以在其中在语言属性中指定 语言名称。例如，您可以使用 language 元素定义 XPath 表达式，如下所示：

```
<language language="xpath">/order/address/countryCode = 'us'</language>
```

作为注解

语言注解在 bean 集成上下文中使用。该注释提供了一种便捷的方式，可以从消息或标头提取信息，然后将提取的数据注入 bean 的方法。

例如，考虑 bean、myBeanProc，它作为 filter （） EIP 的 predicate 调用。如果 bean 的 checkCredentials 方法返回 true，则允许消息继续；但如果方法返回 false，则过滤器阻止消息。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，从底层 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 注释仅在它被注入的参数之前放置。注意 XPath 表达式如何显式选择文本节点，方法是向路径附加 /text （），这样可确保仅选择元素的内容，而不是弹出标签。

作为 Camel 端点 URI

使用 Camel 语言组件，您可以在端点 URI 中调用受支持的语言。有两种替代语法：

要调用存储在文件中的语言脚本（或其他由 Scheme定义的资源类型），请使用以下 URI 语法：

```
language://LanguageName:resource:Scheme:Location[?Options]
```

其中方案可以是 文件：、classpath: 或 http:。

例如，以下路由从 classpath 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，要运行存储在脚本字符串中的 Simple 语言 脚本 ：

```
String script = URLEncoder.encode("Hello ${body}", "UTF-8");
from("direct:start")
  .to("language:simple:" + script)
  .to("mock:result");
```

*有关语言组件的更多详细信息，请参阅 **Apache Camel** 组件参考指南 中的 语言。*

第 14 章 常数

概述

恒定语言是一种简单的内置语言，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

统一表达式语言(EL)最初作为 JSP 2.1 标准的一部分指定，但现在将其作为独立语言提供。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，您需要在项目中添加对 camel-juel 的依赖，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_13_0-00013-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语句：

```
import static org.apache.camel.language.juel.JuelExpression.el;
```

变量

表 15.1 “EL 变量” 列出使用 EL 时可以访问的变量。

表 15.1. EL 变量

变量	类型	value
交换	**org.apache.camel.Exchange**	当前的交换
in	**org.apache.camel.Message**	IN 信息
out	**org.apache.camel.Message**	OUT 消息

EXAMPLE

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

文件语言是简单语言的扩展，而不是自己右侧的独立语言。但是，文件语言扩展只能与 File 或 FTP 端点一起使用。

16.1. 何时使用 FILE 语言

概述

文件语言是无法始终可用的简单语言的扩展。在以下情况下可以使用它：

- 在文件或 FTP 使用者端点 中。
- 在由文件或 FTP 使用者创建的交换时。

注意

文件语言无法使用转义字符 \。

在文件或 FTP 消费者端点中

您可以在 File 或 FTP 使用者端点上设置多个 URI 选项，这些选项使用文件语言表达式作为其值。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、preMove、moveFailed 和 sortBy 选项。

在 File consumer 端点中，fileName 选项充当过滤器，确定哪个文件实际上将从起始目录中读取。如果指定了纯文本字符串（例如 fileName=report.txt），则文件消费者每次更新时都会读取同一文件。但是，您可以通过指定一个简单的表达式使此选项更加动态。例如，当 File consumer 轮询起始目录时，您可以使用计数器 Bean 来选择不同的文件，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 ID 下注册的 bean 的 next （） 方法。

move 选项用于在文件消费者端点读取后将文件移到备份位置。例如，以下端点会在处理后将文件移到备份目录中：

```
file://target/filelanguage/?
move=backup/${date:now:yyyyMMdd}/${file:name.noext}.bak&recursive=false
```

其中 **${file:name.noext}.bak** 表达式会修改原始文件名，将文件扩展名替换为 **.bak**。

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 **File consumer** 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 **File consumer** 端点：

```
file://target/filelanguage/?sortBy=file:modified
```

您可以通过添加 **reverse: prefix to to the order** 来撤销顺序，例如：

```
file://target/filelanguage/?sortBy=reverse:file:modified
```

在由文件或 **FTP** 使用者创建的交换时

当交换源自文件或 **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（只要原始消息标头没有被清除）。例如，您可以定义一个基于内容的路由器，根据其文件扩展路由消息，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

每当路由以 File 或 FTP 消费者端点开头时，都可以使用文件变量，这意味着底层消息正文是 java.io.File 类型。通过 file 变量，您可以访问文件路径名称的各个部分，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的信息）。

起始目录

有些文件变量返回路径相对于 起始目录 定义，而这是在 File 或 FTP 端点中指定的目录。例如，以下文件使用者端点具有起始目录 ./filetransfer （相对路径）：

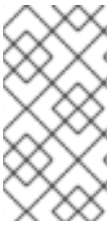
```
file:filetransfer
```

以下 FTP 使用者端点具有起始目录 ./ftptransfer （相对路径）：

```
ftp://myhost:2100/ftptransfer
```

文件变量的命名规则

通常，文件变量使用 java.io.File 类上的对应方法命名。例如，file:absolute 变量提供 java.io.File.getAbsolute （） 方法返回的值。



注意

但是，这种命名规则不严格遵循。例如，没有这样的 方法，如 java.io.File.getSize （）。

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。

变量	类型	描述
file:name.ext	**字符串**	文件扩展名（遵循路径名称中最后一个 **.** 字符之后的字符）。支持具有多个点的文件扩展，如 .tar.gz。
file:name.ext.single	**字符串**	文件扩展名（遵循路径名称中最后一个 **.** 字符之后的字符）。如果文件扩展有多个点，则此表达式仅返回最后一个部分。
file:name.noext	**字符串**	相对于起始目录的路径名，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，省略文件扩展名。如果文件扩展有多个点，则此表达式只剥离最后一个部分，而保留其他部分。
file:onlyname	**字符串**	路径名称的最终部分。也就是说，不带父目录路径的文件名。
file:onlyname.noext	**字符串**	路径名称的最后段，省略文件扩展名。
file:onlyname.noext.single	**字符串**	路径名称的最后段，省略文件扩展名。如果文件扩展有多个点，则此表达式只剥离最后一个部分，而保留其他部分。
file:ext	**字符串**	文件扩展名（与 **file:name.ext**相同）。
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，包括路径中起始目录。
file:absolute	**布尔值**	为 **true**，如果起始目录被指定为绝对路径，则为 **false**，否则为。
file:absolute.path	**字符串**	文件的绝对路径名。
file:length	**Long**	引用的文件的大小。
file:size	**Long**	**与 file:length 相同。**
file:modified	**java.util.Date**	最后修改的日期.

16.3. 例子

相对路径名称

考虑文件消费者端点，其中起始目录指定为 相对路径名称。例如，以下 File 端点具有起始目录 ./filelanguage ：

```
file://filelanguage
```

现在，在扫描 filelanguage 目录时，假设端点刚刚使用以下文件：

```
./filelanguage/test/hello.txt
```

最后，假设 文件语言 目录本身具有以下绝对位置：

```
/workspace/camel/camel-core/target/filelanguage
```

根据前面的场景，在应用到当前交换时，文件语言变量会返回以下值：

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
file:absolute	false
file:absolute.path	/workspace/camel/camel-core/target/filelanguage/test/hello.txt

绝对路径名

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 File 端点具有起始目录 /workspace/camel/camel-core/target/filelanguage ：

```
file:///workspace/camel/camel-core/target/filelanguage
```

现在，在扫描 filelanguage 目录时，假设端点刚刚使用以下文件：

```
./filelanguage/test/hello.txt
```

根据前面的场景，在应用到当前交换时，文件语言变量会返回以下值：

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
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，您需要向项目添加 一个依赖项，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_13_0-00013-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语句：

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

内置属性

表 17.1 “Groovy 属性” 列出使用 Groovy 时可以访问的内置属性。

表 17.1. Groovy 属性

属性	类型	value
context	**org.apache.camel.CamelContext**	Camel 上下文
交换	**org.apache.camel.Exchange**	当前的交换
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** 方法使用解析方法，可以更轻松地在脚本中使用属性组件。

ENGINE_SCOPE 设置的属性。

EXAMPLE

例 17.2 “使用 Groovy 的路由” 显示了使用 Groovy 脚本的两个路由。

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 properties 组件访问属性值，请在内置属性中调用 resolve 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 属性。

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。要提供自定义 GroovyShell，请将 org.apache.camel.language.groovy.GroovyShellFactory SPI 接口的实现添加到 Camel registry 中。

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，而不是默认导入。

```
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

标头语言提供了一种在当前消息中访问标头值的便捷方式。当您提供标头名称时，标头语言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，要根据 SequenceNumber 标头的值（其中序列号必须是正整数）重新排序传入的交换，您可以定义一个路由，如下所示：

```
<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

可以在 Java 中定义相同的路由，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，您需要在项目中添加对 camel-script 的依赖关系，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_13_0-00013-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语句：

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

内置属性

表 19.1 “JavaScript 属性” 列出使用 JavaScript 时可以访问的内置属性。

表 19.1. JavaScript 属性

属性	类型	value
context	**org.apache.camel.CamelContext**	Camel 上下文
交换	**org.apache.camel.Exchange**	当前的交换
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** 方法使用解析方法，可以更轻松地在脚本中使用属性组件。

ENGINE_SCOPE 设置的属性。

EXAMPLE

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 properties 组件访问属性值，请在内置属性中调用 resolve 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 属性。

第 20 章 JOSQL

概述

JoSQL (Java 对象的SQL)语言允许您评估 Apache Camel 中的 predicates 和 表达式。JoSQL 采用类 SQL 的查询语法，对来自内存中 Java 对象的所有数据执行选择和排序操作，JoSQL 并不是 一个数据库。在 JoSQL 语法中，每个 Java 对象实例都被视为表行，每个对象方法被视为列名称。使用此语法，可以构建强大的语句，用于从 Java 对象集合提取和编译数据。详情请查看 http://josql.sourceforge.net/。

添加 JOSQL 模块

要在路由中使用 JoSQL，您需要在项目中添加对 camel-josql 的依赖，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语句：

```
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

Name	类型	描述
交换	**org.apache.camel.Exchange**	当前的交换
in	**org.apache.camel.Message**	IN 信息
out	**org.apache.camel.Message**	OUT 消息
属性	**对象**	键为属性的 Exchange *属性*
header	**对象**	IN 消息标头，其键为 *标头*
变量	**对象**	键为变量 *的变量*

EXAMPLE

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 语言提供便捷的语法，用于提取 JSON 消息的部分。JSON 的语法与 XPath 类似，但它用于从 JSON 消息中提取 JSON 对象，而不是对 XML 执行操作。jsonpath DSL 命令可用作表达式或 predicate （空结果被解释为布尔值 false）。

添加 JSONPATH 软件包

要在 Camel 路由中使用 JsonPath，您需要将对 camel-jsonpath 的依赖关系添加到您的项目中，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 命令选择特定价格范围内的项目：

```
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，您可以将 JsonPath 查询用作 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>
```

轻松语法

当您希望使用 jsonpath 语法定义基本 predicate 时，很难记住语法。例如，要找到所有 cheap book，您必须按如下方式编写语法：

```
$.store.book[?(@.price < 20)]
```

但是，如果您只可以写为：

```
store.book.price < 20
```

如果您只想查看具有价格键的节点，您也可以省略该路径：

```
price < 20
```

为了支持此功能，可以使用一个 EasyPredicateParser，它用来使用基本风格来定义 predicate。这意味着 predicate 不得以 $ 符号开头，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	从文件中读取
字符串	普通字符串
Map	essage body as **java.util.Map** type
list	消息正文为 java.util.List 类型
POJO	可选的 If Jackson 位于 classpath 上，则 **camel-jsonpath** 可以使用 Jackson 来读取消息正文为 **POJO**，并转换为 **java.util.Map**，它由 JSonPath 支持。例如，您可以添加 **camel-jackson** 作为依赖项，使其包含 Jackson。
InputStream	如果以上类型都不匹配，则 Camel 会尝试将消息正文读取为 **java.io.InputStream**。

如果消息正文是不支持的类型，则默认抛出异常，但您可以配置 JSonPath 以抑制异常。

阻止例外

如果没有找到由 jsonpath 表达式配置的路径，则 jsonpath 会抛出异常。通过将 SuppressExceptions 选项设置为 true 可忽略异常。例如，在下面的代码中，添加 true 选项作为 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 注入

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其绑定到方法参数。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 支持 JsonPath 表达式中的内联 Simple 表达式。Simple language insertions 必须以 Simple 语法表示，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 来关闭对 Simple 表达式的支持，如下所示。

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 的更多详细信息，请参见 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 类似，但不是从 XML 文档选择元素或属性节点，而是调用 Java Bean 对象图中的方法。如果其中一个 bean 属性返回 XML 文档（一个 DOM/JDOM 实例），则该路径的剩余部分被解释为 XPath 表达式，用于从文档中提取 XML 节点。换句话说，JXPath 语言提供对象图形导航和 XML 节点选择混合。

添加 JXPATH 软件包

要在路由中使用 JXPath，您需要在项目中添加对 camel-jxpath 的依赖关系，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_13_0-00013-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	value
这	**org.apache.camel.Exchange**	当前的交换
in	**org.apache.camel.Message**	IN 信息

变量	类型	value
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：允许打开 JXPathContext。当启用此选项时，允许 JXPath 表达式评估表达式和消息正文，它们可能无效或缺少数据。请参阅 JXPath 文档 的更多详细信息。此选项默认为 false。

例子

以下示例 route 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 中使用 JXPath 表达式作为 predicate:

```
from("direct:start").
    filter().jxpath("in/body/name='James'").
    to("mock:result");
```

JXPATH 注入

您可以使用 Bean Integration 对 bean 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 开始提供

您可以对脚本进行外部化，并让 Camel 从资源（如 "classpath:"、"file:" 或 "http:" ）加载它。请遵循以下语法：

```
"resource:scheme:location"
```

例如，要引用 classpath 上的文件：

```
.setHeader("myHeader").jxpath("resource:classpath:myjxpath.txt")
```

第 23 章 MVEL

概述

MVEL 是基于 Java 的动态语言，它与 OGNL 类似，但其报告为更快。MVEL 支持位于 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()
```

因为 MVEL 是动态输入的，因此在调用 getFamilyName （） 方法前，不需要使消息正文实例（在 Object type 部分）。您还可以对调用 bean 属性使用缩写语法，例如：

```
request.body.familyName
```

添加 MVEL 模块

要在路由中使用 MVEL，您需要在项目中添加对 camel-mvel 的依赖关系，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_13_0-00013-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” 列出使用 MVEL 时可以访问的内置变量。

表 23.1. MVEL 变量

Name	类型	描述
这	**org.apache.camel.Exchange**	当前的交换
交换	**org.apache.camel.Exchange**	当前的交换
例外	**Throwable**	Exchange 异常（如果有）
exchangeID	**字符串**	Exchange ID
Fulting	**org.apache.camel.Message**	Fault 消息（如果有）
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	named Exchange 属性的值
property (*name*,*type*)	***type***	named Exchange 属性的 typed 值

EXAMPLE

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。对于 get 和 set 属性的值，您可以使用相同的表达式。OGNL 支持位于 camel-ognl 模块中。

CAMEL ON EAP 部署

此组件由 EAP 上的 Camel (Wildfly Camel)框架支持，该框架在 Red Hat JBoss Enterprise Application Platform (JBoss EAP)容器上提供了简化的部署模型。

添加 OGNL 模块

要在路由中使用 OGNL，您需要在项目中添加对 camel-ognl 的依赖关系，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语句：

```
import static org.apache.camel.language.ognl.OgnlExpression.ognl;
```

内置变量

表 24.1 “OGNL 变量” 列出使用 OGNL 时可以访问的内置变量。

表 24.1. OGNL 变量

Name	类型	描述
这	**org.apache.camel.Exchange**	当前的交换
交换	**org.apache.camel.Exchange**	当前的交换
例外	**Throwable**	Exchange 异常（如果有）
exchangeID	**字符串**	Exchange ID
Fulting	**org.apache.camel.Message**	Fault 消息（如果有）
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	named Exchange 属性的值
property (*name*,*type*)	***type***	named Exchange 属性的 typed 值

EXAMPLE

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 (DEPRECATED)

概述

PHP 是一个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，并将在以后的发行版本中删除。

添加 SCRIPT 模块

要在路由中使用 PHP，您需要在项目中添加对 camel-script 的依赖关系，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语句：

```
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	value
context	**org.apache.camel.CamelContext**	Camel 上下文
交换	**org.apache.camel.Exchange**	当前的交换
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** 方法使用解析方法，可以更轻松地在脚本中使用属性组件。

***ENGINE_SCOPE** 设置的属性。*

EXAMPLE

*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 **properties** 组件访问属性值，请在内置属性中调用 **resolve** 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 属性。*

第 26 章 EXCHANGE PROPERTY

概述

交换属性语言提供了一种访问 交换属性 的便捷方式。当您提供与其中一个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 camel-core 的一部分。

XML 示例

例如，要在 listOfEndpoints Exchange 属性包含接收者列表时实施接收者列表模式，您可以定义一个路由，如下所示：

```
<camelContext>
  <route>
    <from uri="direct:a"/>
    <recipientList>
      <exchangeProperty>listOfEndpoints</exchangeProperty>
    </recipientList>
  </route>
</camelContext>
```

JAVA 示例

可以在 Java 中实施相同的接收者列表示例，如下所示：

```
from("direct:a").recipientList(exchangeProperty("listOfEndpoints"));
```

第 27 章 PYTHON (DEPRECATED)

概述

Python 是一个非常强大的动态编程语言，用于各种应用程序域。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，并将在以后的发行版本中删除。

添加 SCRIPT 模块

要在路由中使用 Python，您需要在项目中添加对 camel-script 的依赖，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语句：

```
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	value
context	**org.apache.camel.CamelContext**	Camel 上下文
交换	**org.apache.camel.Exchange**	当前的交换
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** 方法使用解析方法，可以更轻松地在脚本中使用属性组件。

***ENGINE_SCOPE** 设置的属性。*

EXAMPLE

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 **properties** 组件访问属性值，请在内置属性中调用 **resolve** 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 属性。

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语句：

```
import static org.apache.camel.language.ref.RefLanguage.ref;
```

XML 示例

例如，splitter 模式可以使用 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 中实现，如下所示：

```
from("seda:a")
  .split().ref("myExpression")
  .to("seda:b");
```

第 29 章 RUBY (DEPRECATED)

概述

Ruby 是一种动态的开源编程语言，专注于简洁性和生产力。它有一个更精简的语法，自然是为了读取和写入。Ruby 支持是 camel-script 模块的一部分。

重要

Apache Camel 中的 Ruby 已被弃用，并将在以后的发行版本中删除。

添加 SCRIPT 模块

要在路由中使用 Ruby，您需要在项目中添加对 camel-script 的依赖关系，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语句：

```
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	value
context	**org.apache.camel.CamelContext**	Camel 上下文
交换	**org.apache.camel.Exchange**	当前的交换
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** 方法使用解析方法，可以更轻松地在脚本中使用属性组件。

ENGINE_SCOPE 设置的属性。

EXAMPLE

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 properties 组件访问属性值，请在内置属性中调用 resolve 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 属性。*

第 30 章 简单语言

摘要

简单的语言是在 Apache Camel 中开发的语言，专门用于访问和操作交换对象的各个部分。语言并不像最初创建一样简单，现在它包含了一组全面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 （） 命令调用为处理器上的子类型，例如：

```
from("seda:order")
  .filter()
  .simple("${in.header.foo}")
  .to("mock:fooOrders");
```

嵌入到字符串中

如果您在纯文本字符串中嵌入一个简单的表达式，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：

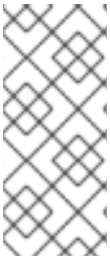
```
simple("Hello ${in.header.name}, how are you?")
```

自定义启动和端点令牌

从 Java 中，您可以通过调用 changeFunctionStartToken 静态方法和 SimpleLanguage 对象的 changeFunctionEndToken 静态方法来自定义开始和结束令牌({ 和 })。

例如，您可以在 Java 中将启动和结束令牌更改为 [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 令牌会影响在其 classpath 上共享同一 camel-core 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，要定义一个根据 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

例如，如果您启用了 Spring 属性占位符或 OSGi 蓝图属性占位符，则可能会发现 ${Expression} syntax clashes with another property placeholder syntax clashes。在这种情况下，您可以为简单表达式使用替代语法 $simple{Expression} 来撤销占位符。例如：

```
<simple>Hello $simple{in.header.name}, how are you?</simple>
```

自定义启动和端点令牌

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，要将启动和结束令牌更改为 [和]，请在 XML 配置文件中定义一个新的 SimpleLanguage bean，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 令牌会影响在其 classpath 上共享同一 camel-core 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 中自动修剪简单表达式前和之后的空格。因此，这个表达式带有周围的空格：

```
<transform>
  <simple>
    data=${body}
  </simple>
</transform>
```

会自动修剪，因此等同于这个表达式（没有周围空格）：

```
<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，如下所述。

script 资源的语法

使用以下语法访问存储为外部资源的简单脚本：

```
resource:Scheme:Location
```

其中 Scheme: 可以是 classpath:、、或 http:。

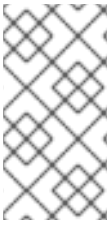
例如，要从 classpath 读取 mysimple.txt 脚本，

```
simple("resource:classpath:mysimple.txt")
```

30.4. 表达式

概述

简单的语言提供各种元素表达式，用于返回消息交换的不同部分。例如，表达式 simple ("${header.timeOfDay}") 将从传入消息返回名为 timeOfDay 的标头的内容。



注意

从 Apache Camel 2.9 开始，您必须始终 使用占位符语法 ${Expression} 来返回变量值。不被允许省略标题令牌(${ 和 })。

单个变量的内容

您可以根据提供的变量，使用简单语言来定义字符串表达式。例如，您可以使用表单中的变量 in.header.HeaderName 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

您可以将简单的变量嵌入到字符串表达式中，例如：

```
simple("Received a message from ${in.header.user} on ${date:in.header.date:yyyyMMdd}.")
```

date 和 bean 变量

除了提供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”），简单语言还提供特殊变量进行格式化日期、日期：命令:模式，以及调用 bean 方法， beanRef。例如，您可以使用 date 和 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。这主要用于将结果类型转换为布尔值或数字类型。

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>
```

动态标头键

在 Camel 2.17 中，setHeader和 setExchange 属性允许使用带有 Simple 语言的动态标头键（如果键名称是简单语言表达式）。

```
<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

简单表达式可以是嵌套的 **swig-wagon，例如：**

```
simple("${header.${bean:headerChooser?method=whichHeader}}")
```

访问**常量或枚举**

您可以使用以下语法访问 bean 的常量或 enum 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 Customer 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 Navigation Language (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，如果消息正文是带有 getAddress （） 访问权限的 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，@ body.address.street} 是 ${body.getAddress.getStreet} 的简写。

OGNL null-safe operator

您可以使用 null-safe operator ?. 来避免在正文 没有 地址时遇到 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映射中查找带有键 foo 的值：

```
simple("${body[foo]?.name}")
```

OGNL 列表元素访问

您还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 size 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 length 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

您可以通过测试表达式来构建 predicates。例如： predicate, simple ("${header.timeOfDay} == '14:30'")，测试传入消息中的 timeOfDay 标头是否等于 14:30。

另外，每当将 resultType 指定为布尔值时，表达式都会作为一个 predicate 而不是表达式来评估。这允许将 predicate 语法用于这些表达式。

语法

您还可以使用简单 predicates 测试交换的不同部分（标题、消息正文等）。简单 predicates 有以下通用语法：

```
${LHSVariable} Op RHSValue
```

其中左侧的 LHSVariable 变量是 表 30.1 “简单语言的变量” 中显示的变量之一，以及右侧 RHSValue 的值之一：

- 另一个变量 ${RHSVariable}.
- 字符串 literald （用单引号括起）、'.
- 数字常数，用单引号 ' 括起。
- null 对象 null。

简单的语言始终尝试将 **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** 在进行比较前，可能需要进入相应的类型。

例子

例如，您可以执行简单的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
```

您可以测试左侧是否是逗号分隔列表的成员，如下所示：

```
simple("${in.header.type} in 'gold,silver'")
```

您可以测试左侧是否匹配正则表达式，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位于指定的数字范围内（范围已包含），如下所示：

```
simple("${in.header.number} range '100..199'")
```

组合

您还可以使用逻辑组合、**finis** 和 **||** 组合来 组合 **predicates**。

例如，以下是使用 & 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 || 组合的表达式（逻辑含 或 ）：

```
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** 消息正文。**支持 OGNL 表达式。**
in.body	**对象**	**In** 消息正文。**支持 OGNL 表达式。**
out.body	**对象**	**Out** 消息正文。
bodyAs (*Type*)	*类型*	**In** message body，转换为指定的类型。所有类型 类型 必须使用其完全限定 Java 名称来指定，但 type: **byte[]**, **String**, ,**Integer**, 和 **Long** 除外。*转换的正文可以是 null。*

变量	类型	描述
mandatoryBodyAs (*Type*)	*类型*	**In** message body，转换为指定的类型。所有类型 类型 必须使用其完全限定 Java 名称来指定，但 type: **byte[]**, **String**, ,**Integer**, 和 **Long** 除外。*转换的正文应该为非 null。*
header.*HeaderName*	对象	**In** message 的 *HeaderName* 标头。**支持 OGNL 表达式。**
header[*HeaderName***]**	对象	**In** message 的 *HeaderName* 标头 (alternative 语法)。
headers.*HeaderName*	对象	**In** message 的 *HeaderName* 标头。
headers[*HeaderName***]**	对象	**In** message 的 *HeaderName* 标头 (alternative 语法)。
in.header.*HeaderName*	对象	**In** message 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** message 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** message 的 *HeaderName* 标头。
out.header[*HeaderName***]**	对象	**Out** message 的 *HeaderName* 标头(alternative 语法)。
out.headers.*HeaderName*	对象	**Out** message 的 *HeaderName* 标头。
out.headers[*HeaderName***]**	对象	**Out** message 的 *HeaderName* 标头(alternative 语法)。

变量	类型	描述
headerAs (*Key*,*Type*)	*类型*	*Key* 标头，转换为指定的类型。所有类型 类型 必须使用其完全限定 Java 名称来指定，但 type: **byte[]**, **String**, ,**Integer**, 和 **Long** 除外。*转换的值可以是 null。*
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 (alternative 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	**Exchange.getException （）** 中的例外对象；或者，如果这个值为 null，则来自 **Exchange.EXCEPTION_CAUGHT** 属性的异常异常；否则为 null。**支持 OGNL 表达式。**
exception.message	字符串	如果在交换上设置了例外，则返回 **Exception.getMessage （）** 的值；否则，返回 **null**。

变量	类型	描述
exception.stacktrace	**字符串**	如果在交换上设置了例外，则返回 **Exception.getStackTrace （）** 的值，否则返回 **null**。**注意：** 简单的语言首先尝试从 **Exchange.getException （）** 检索异常。如果没有设置该属性，它将通过调用 **Exchange.getProperty (Exchange.CAUGHT_EXCEPTION)** 来检查被捕获的异常。
日期： *命令*:*pattern*	**字符串**	使用 java.text.SimpleDateFormat 模式格式化的日期。支持以下命令：**现在**，对于当前的日期和时间；**header.** *HeaderName*，或 **in.header.***HeaderName*，使用 In 消息中的 *HeaderName* 标头中的 java.util **.**Date 对象，**在 Out** 消息的*HeaderName* 标头中使用 java.util.Date 对象；
Bean:*beanID.方法*	**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 ID *beanID* 的 bean，**并返回对 bean 本身的引用**。例如，如果您使用 splitter EIP，您可以使用此变量来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：*Location*:*Key*	**字符串**	*Key* 属性占位符的值，其中属性文件的位置由 *Location* 提供。
threadName	**字符串**	当前线程的名称。
routeId	**字符串**	返回 **交换** 被路由的当前路由的 ID。

变量	类型	描述
类型：*Name*[*.Field*]	对象	根据 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 字段指代为 **type:org.apache.camel.Exchange.FILE_NAME**
Collate (group)	**list**	在 Camel 2.17 中，collate 函数会迭代消息正文，并将数据分组到特定大小的子列表中。您可以与 Splitter EIP 一起使用，将消息正文和组拆分为一组 N 子列表。
skip(number)	**iterator**	skip 函数迭代消息正文，并跳过第一个项目数量。这可以与 Splitter EIP 一起使用来分割消息正文，并跳过第一个 N 个项目数。

30.7. OPERATOR 参考

二进制操作符

*简单语言 **predicates** 的二进制运算符显示在 表 30.2 “Simple Language 的二进制 Operator” 中。*

表 30.2. Simple Language 的二进制 Operator

Operator	描述
==	等于.
=~	等于忽略问题单。在比较字符串值时忽略这种情况。
>	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。

Operator	描述
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。
Not regex	测试 LHS 字符串 **是否** 不匹配 RHS 正则表达式。
in	测试 LHS 字符串是否出现在 RHS 中，用逗号分隔的列表。
not in	测试 LHS 字符串是否 **没有出现在** RHS 列表中。
is	测试 LHS 是 RHS Java 类型的实例（使用 Java **实例**）。
not is	测试 LHS 是否 **不是** RHS Java 类型的实例（使用 Java **实例**）。
range	测试 LHS 号是否位于 RHS 范围（其中 范围具有格式 '*min*...*max*'）。
Not range	测试 LHS 号是否 **不适用于** RHS 范围（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 “简单语言的不常规 Operator” 中。*

表 30.3. 简单语言的不常规 Operator

Operator	描述
++	按 1 递增数字.
--	将数字减少为 1.
\n	换行符。
\r	回车符。

Operator	描述
\t	选项卡字符。
\	**(obsolete)** 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 的组合” 中显示的组合可用于组合两个或多个简单语言 predicates。

表 30.4. Simple Language Predicates 的组合

Operator	描述
&&	将两个 predicates 与 logical **和** 组合。
||	将两个 predicates 与逻辑 **包含 或** 组合。
and	**弃用**。改为使用 **&&**。
or	**弃用**。使用 || 替代。

第 31 章 SPEL

概述

Spring Expression Language (SpEL) 是 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 的显著功能是您可以轻松地访问 registry 中的 Bean。

语法

SpEL 表达式必须使用占位符语法 ：{SpelExpression}，以便它们可以嵌入到纯文本字符串中（换句话说，SpEL 启用了表达式模板）。

SpEL 也可以使用 @BeanID 语法在注册表（通常是 Spring 注册表）中查找 bean。例如，给定 ID 为 headerUtils、方法 count （） （计算当前消息上的标头数量）的 bean，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，您需要在项目中添加对 camel-spring 的依赖关系，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_13_0-00013-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。
交换	**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）。
Fulting	**消息**	故障消息（如果有）。
Request （请求）	**消息**	交换的 In 消息。
响应	**消息**	交换的 Out 消息（如果有）。
属性	**Map**	交换属性。
property (*Name*)	对象	使用名称 键的交换 *属性*。
property (*Name*,*Type*)	类型	使用名称 键的交换 *属性* 转换为类型 *类型*。

XML 示例

例如，若要仅选择其 Country 标头具有值 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

在处理 XML 消息时，XPath 语言允许您选择消息的一部分，方法是指定对消息的 Document Object Model (DOM)行为的 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") 来评估当前交换上的 XPath 表达式（其中 XPath 表达式应用到当前消息的正文）。xpath （） 表达式的结果是一个 XML 节点（如果多个节点匹配，则为节点集）。

例如，要从当前的 In 邮件正文中提取 /person/name 元素的内容，并使用它来设置名为 user 的标头，您可以定义如下路由：

```
from("queue:foo")
    .setHeader("user", xpath("/person/name/text()"))
    .to("direct:tie");
```

您可以使用 fluent builder xpath （） 命令指定要将 xpath （） 指定为 setHeader （） 的参数，例如：

```
from("queue:foo")
    .setHeader("user").xpath("/person/name/text()")
    .to("direct:tie");
```

如果要将结果转换为特定类型的类型，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 元素属于一个模式，由命名空间 URI 标识。当处理文档时，需要把命名空间 URI 与前缀关联，以便您可以在 XPath 表达式中识别元素名称。Apache Camel 提供了帮助程序类 org.apache.camel.builder.xml.Namespaces，它可让您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");
```

您可以在 xpath （） 表达式构建器中传递 Namespaces 对象 ns 作为额外参数，使命名空间定义可供 xpath （）表达式构建器使用。如果您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");
```

如果您需要指定结果类型 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的一个最频繁问题，即命名空间在传入消息和 XPath 表达式中使用的命名空间之间存在不匹配。为了帮助您对这类问题进行故障排除，XPath 语言支持通过选项将所有传入消息中的所有命名空间转储到系统日志中。

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 TRACE 级别日志记录。

启用命名空间日志记录后，您将看到每个处理的消息类似如下日志消息：

```
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，要从当前的 In 邮件正文中提取 /person/name 元素的内容，并使用它来设置名为 user 的标头，您可以定义如下路由：

```
<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的类型，请通过将 resultType 属性设置为 Java 类型名称来指定结果类型（您必须指定完全限定类型名称）。例如，要明确指定结果类型是 java.lang.String （您可以省略此处的 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，以便您可以在 XPath 表达式中识别元素名称。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的一个最频繁问题，即命名空间在传入消息和 XPath 表达式中使用的命名空间之间存在不匹配。为了帮助您对这类问题进行故障排除，XPath 语言支持通过选项将所有传入消息中的所有命名空间转储到系统日志中。

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 TRACE 级别日志记录。

启用命名空间日志记录后，您将看到每个处理的消息类似如下日志消息：

```
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

当使用 Apache Camel bean 集成在 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其绑定到方法参数。

例如，请考虑以下路由片段，它调用 AccountService 对象的 credit 方法：

```
from("queue:payments")
    .beanRef("accountService","credit")
    ...
```

credit 方法使用参数绑定注解从消息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 的预定义命名空间” 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 的预定义命名空间

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 namespaces 参数。然后，由 @NamespacePrefix 定义的命名空间可以在 @XPath 注释的表达式值中使用。

例如，要将前缀 ex 与自定义命名空间 http://fusesource.com/examples 关联，请按如下所示调用

@XPath 注释：

```
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 类允许您独立于交换评估 **XPath** 表达式。也就是说，如果您有任何来源的 **XML** 片段，您可以使用 **XPathBuilder** 来评估 **XML** 片段上的 **XPath** 表达式。

匹配表达式

使用 **match （）**方法检查是否可以找到一个或多个与给定 **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");
```

如果发现至少一个与表达式匹配的节点，则根据 **XML** 片段评估表达式 **Expression**，其结果为 **true**。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");
```

您还可以通过传递所需的类型来指定结果类型，作为第二个参数，以评估（）HEKETI-wagon 例如：

```
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对 camel-saxon 工件的依赖（将这个依赖项添加到 Maven POM 中，如果使用 Maven，或将 camel-saxon-2.23.2.fuse-7_13_0-00013-redhat-00001.jar 文件添加到 classpath 中）。

在 Java DSL 中使用 Saxon 解析器

在 Java DSL 中，启用 Saxon parser 的最简单方法是调用 saxon （） fluent 构建器方法。例如，您可以调用 Saxon parser，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 的最简单方法是在 xpath 元素中将 saxon 属性设置为 true。例如，您可以调用 Saxon parser，如下例所示：

```
<xpath saxon="true" resultType="java.lang.String">current-dateTime()</xpath>
```

带有 Saxon 的编程

如果要在应用程序代码中使用 Saxon XML 解析器，您可以使用以下代码显式创建 Saxon transformer 工厂实例：

```
// Java
import javax.xml.transform.TransformerFactory;
import net.sf.saxon.TransformerFactoryImpl;
...
TransformerFactory saxonFactory = new net.sf.saxon.TransformerFactoryImpl();
```

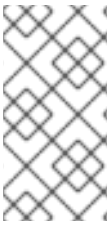
另一方面，如果您更喜欢使用通用 JAXP API 创建转换器工厂实例，您必须首先在 ESBInstall/etc/system.properties 文件中设置 javax.xml.transform.TransformerFactory 属性，如下所示：

```
javax.xml.transform.TransformerFactory=net.sf.saxon.TransformerFactoryImpl
```

然后，您可以使用通用 JAXP API 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使用 Saxon 的任何第三方库，则可能需要使用第二种通用方法。



注意

Saxon 库必须安装在作为 OSGi 捆绑包(net.sf.saxon/saxon9he)的容器中（默认通常安装）。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 表达式返回 org.w3c.dom.NodeList 类型的一个或多个 XML 节点列表。但是，您可以使用类型转换器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 表达式的结果作为字符串 返回 ：

```
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 people 元素 中的多个 person 元素中有多个 person 元素，则这个基本模式可以匹配多个节点 swig-wagon。

/name/text()

如果您只想访问元素内部 的文本，请将 /text （） 附加到位置路径，否则节点包含元素的 start 和 end 标签（当您将节点转换为字符串时，会包含这些标签）。

/person/telephone/@isDayTime

要选择属性的值 AttributeName，请使用语法 @AttributeName。例如，在应用到以下 XML 片段时，前面的位置路径返回 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，/people/person/* 匹配 个人 的所有子元素。

@*

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 name 元素匹配：

```
<invoice>
  <person>
    <name .../>
  </person>
</invoice>
<person>
  <name .../>
</person>
<name .../>
```

..

选择当前上下文节点的父节点。通常不适用于 Apache Camel XPath 语言，因为当前上下文节点是文档 root，没有父项。

node()

匹配任何类型的节点。

text ()

匹配文本节点。

comment ()

匹配评论节点。

processing-instruction()

匹配一个处理型节点。

predicate 过滤器

您可以通过将 predicate 附加到方括号 [Predicate] 来过滤与位置路径匹配的节点集合。例如，您可以

通过将 [N] 附加到位置路径，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，以便选择具有特定属性值的元素。以下表达式选择 name 元素，其 surname 属性是 Strachan 或 Davies：

```
/person/name[@surname="Strachan" or @surname="Davies"]
```

您可以使用任何组合 和 或,not （） 来组合 predicate 表达式，您可以使用 comparators, =, !=, > , > , >= , & lt; , REPLACE (practice, less-than 符号)比较表达式。 您还可以在 predicate 过滤器中使用 XPath 功能。

Axes

当您考虑 XML 文档的结构时，root 元素包含一系列子元素，其中一些子元素包含其他子元素等。以这种方式查看嵌套元素，其中嵌套元素由 子 关系链接在一起，整个 XML 文档包含 树 的结构。现在，如果您选择这个元素树中的特定节点(称为上下文节点)，您可能需要引用相对于所选节点树的不同部分。例如，您可能希望引用上下文节点的子项、上下文节点的子项，或指向与上下文节点共享相同的父节点(同级节点)的所有父节点。

XPath axis 用于指定节点匹配的范围，将搜索限制为节点树的特定部分，相对于当前上下文节点。axis 作为前缀附加到您要匹配的节点名称，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 的上下文节点是路径 /invoice/ items 选择的 items 元素。child:: axis 将搜索限制为上下文节点 items 的子项，以便 child::item 与名为 item 的项目 的子项匹配。实际上，sub :: axis 是默认的 axis，因此前面的示例可以等效地写为：

```
/invoice/items/item
```

但还有一些其他的 axes （全部有 13），其中的一些内容已以缩写形式显示 ：@ 是 属性 的缩写： //

是 descendant-or-self:: 的缩写。xes 的完整列表如下（有关以下参考信息）：

- *ancestor*
- *ancestor-or-self*
- *attribute*
- *子对象*
- *descendant*
- *descendant-or-self*
- *关注*
- *以下同级*
- *namespace*
- *父*
- *之前*
- *半同级*
- *Self*

Functions

XPath 提供了一组小的标准函数，这在评估 predicates 时很有用。例如，要从节点集合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()]
```

其中前面的示例以序列（按文档顺序）选择最后一个 person 元素。

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 中使用 xpath 或 XML DSL，其中 predicate 是 expected to expected to the filter （） processor 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，允许消息通过，只有在 /person/city 元素包含值 London 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.
and	将两个 predicates 与 logical **和** 组合。
or	将两个 predicates 与逻辑 **包含 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

在评估路由中的 XPath 表达式时，您可以使用 XPath 变量来访问当前交换的内容，以及 O/S 环境变量和 Java 系统属性。如果变量通过 XML 命名空间访问，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 $in:body，In message's header value as $in:HeaderName。O/S 环境变量可以作为 $env:EnvVar 和 Java 系统属性访问，作为 $system:SysVar。

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。第二个路由过滤使用 XPath 表达式 $in:city = 'London' 进行交换，其中 $in:city 变量被 city 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 语言还定义了其他函数。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、评估简单表达式或查找 Apache Camel 属性占位符组件中的属性。

例如，以下示例使用 in:header （） 函数和 in:body （） 函数来访问头和来自底层交换的正文：

```
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");
```

请注意这些功能与对应的 in: HeaderName或 in: body 变量之间的相似性。函数的语法略有不同：in:header ('HeaderName') 而不是 in:HeaderName; 和 in:body （） 而不是 in:body。

评估 XPathBuilder 中的变量

您还可以使用表达式中的变量，这些变量通过 XPathBuilder 类进行评估。在这种情况下，您无法使用 $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 变量(名称,Value) 流畅构建器方法定义的变量。

例如，以下 XPathBuilder 构造评估 $test 变量，该变量被定义为具有 London 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会自动进入全局命名空间中（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 变量命名空间” 显示与各种命名空间前缀关联的命名空间 URI。

表 32.3. XPath 变量命名空间

命名空间 URI	prefix	描述
http://camel.apache.org/schema/spring	None	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 **In** 消息的标头或正文。
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/	**å‡½æ•°**	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	undefined	用于引用交换属性。您必须为这个命名空间定义自己的前缀。

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，您还可以使用这些功能。*

表 32.4. XPath 自定义功能

功能	描述
in:body()	返回 **In** message body。
in:header (*HeaderName*)	返回带有 name, *HeaderName* 的 **In** 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
function:simple (*SimpleExp*)	评估指定的简单表达式 *SimpleExp*。

第 33 章 XQUERY

概述

Xquery 最初被认为是 XML 表单中存储的数据的查询语言。当消息采用 XML 格式时，XQuery 语言允许您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 表达式也都是有效的 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，您可能需要将表达式存储在文件中，然后通过将 java.io.File 参数或 java.net.URL 参数传递给超载 xquery （） 方法来引用。XQuery 表达式隐式操作消息内容，并返回节点集作为结果。根据上下文，返回值被解释为 predicate （空节点集解释为 false）或表达式。

添加 SAXON 模块

要在路由中使用 XQuery，您需要在项目中添加对 camel-saxon 的依赖关系，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>
```

CAMEL ON EAP 部署

camel-saxon 组件由 EAP 上的 Camel (Wildfly Camel)框架支持，该框架在 Red Hat JBoss Enterprise Application Platform (JBoss EAP)容器上提供了简化的部署模型。

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语句：

```
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
交换	**Exchange**	当前的交换
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**	IN 消息标头，其键为 *key*
out.headers.*key*	**对象**	*键*为键的 OUT 消息标头
key	**对象**	关键是 *键的*Exchange 属性

EXAMPLE

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

在可以使用 Apache Camel 开始编程前，您应该清楚了解消息交换的建模方式。由于 Apache Camel 可以处理许多消息格式，因此基本消息类型旨在具有抽象格式。Apache Camel 提供了访问和转换消息正文和消息标头的数据格式所需的 API。

34.1. EXCHANGES

概述

Exchange 对象 是一种打包程序，封装收到的消息并存储其关联的元数据（包括 交换属性）。此外，如果当前消息分配给生成者端点，则交换提供了临时插槽来保存回复(Out 消息)。

在 Apache Camel 中交换的一个重要功能是，它们支持 lazy 创建消息。当不需要显式访问信息的路由时，这可能会产生显著的优化。

图 34.1. 通过路由交换对象

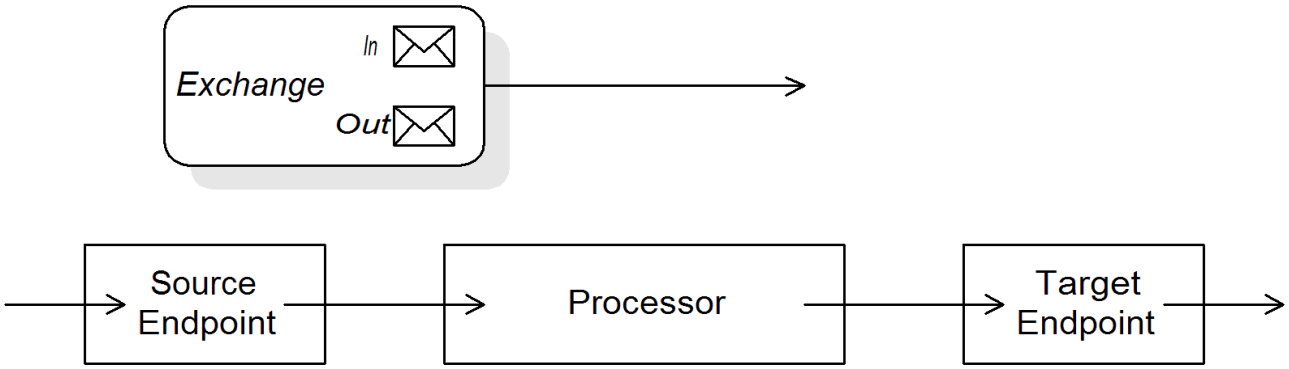


图 34.1 “通过路由交换对象” 显示通过路由的交换对象。在路由上下文中，交换对象作为 Processor.process （） 方法的参数传递。这意味着交换对象可以被源端点、目标端点和所有处理器直接访问。

Exchange 接口

org.apache.camel.Exchange 接口定义了访问 In 和 Out 消息的方法，如 例 34.1 “Exchange Methods” 所示。

例 34.1. Exchange Methods

```
// 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 界面中方法的完整描述，请参考 第 43.1 节 “Exchange Interface”。

lazy 创建消息

Apache Camel 支持 lazy 创建 In、out 和 Fault 消息。这意味着，在尝试访问它们之前不会创建消息实例（例如，通过调用 getIn （） 或 getOut （））。lazy 消息创建语义由 org.apache.camel.impl.DefaultExchange 类实现。

如果您调用 no-argument accessors (getIn （) 或 getOut （））中的一个，或者如果您调用布尔值参数等于 true （即 getIn (true) 或 getOut (true)），则默认方法实现会创建一个新的消息实例，如果它尚不存在。

如果您调用一个等于 false （即 getIn (false) 或 getOut (false)的布尔值参数）的访问者，则默认方法实现会返回当前消息值。[1]

lazy 创建交换 ID

Apache Camel 支持 lazy 创建交换 ID。您可以在任何交换上调用 getExchangeId （） 来获取该交换实例的唯一 ID，但仅当您实际调用该方法时，才会生成 ID。此方法的 DefaultExchange.getExchangeId （） 实现将 ID 生成委派给通过 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

消息对象代表使用以下抽象模型的消息：

- 消息正文
- 消息标头
- 消息附加

消息正文和消息标头可以是任意类型（它们声明为类型 Object），并且消息附加声明为 javax.activation.DataHandler 类型，其中可以包含任意 MIME 类型。如果您需要获取消息内容的具体表示，您可以使用类型转换器机制将正文和标头转换为其他类型的，并可能使用 marshalling 和 unmarshalling 机制。

Apache Camel 消息的一个重要特性是，它们支持 lazy 创建 消息正文和标头。在某些情况下，这意味着消息可以通过路由，而无需完全解析。

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 “邮件接口” 所示。

例 34.2. 邮件接口

```
// 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 Message 接口方法的完整描述，请参考 第 44.1 节 “消息接口”。

lazy 创建正文、标头和附加

Apache Camel 支持 lazy 创建正文、标头和附加功能。这意味着，在需要前，不会创建代表消息正文、消息标头或消息附加的对象。

例如，请考虑以下路由，该路由从 In 消息访问 foo 消息标头：

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

在此路由中，如果我们假定 SourceURL 引用的组件支持 lazy 创建，则不会实际解析 In 消息标头，直到执行 标头("foo") 调用为止。此时，底层的消息实现会解析标头并填充标头映射。在到达路由的末尾之前，消息正文 不会解析，在 to ("TargetURL") 调用中。此时，正文会被转换为写入目标端点 TargetURL 所需的格式。

通过等待最后可能的时间，然后再填充正文、标头和附加功能，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，如果路由不包含对消息标头的显式引用，消息可以在不解析标头的情况下遍历路由。

实践中是否实施了 lazy 创建，这取决于底层组件的实施。通常，当创建消息正文、消息标头或消息附件成本时，lazy 创建非常重要。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”。

lazy 创建消息 ID

Apache Camel 支持消息 ID 的 lazy 创建。也就是说，只有在您实际调用 getMessageId （） 方法时才会生成消息 ID。此方法的 DefaultExchange.getExchangeId （） 实现将 ID 生成委派给通过 CamelContext 注册的 UUID 生成器。

如果端点实现实现，如果端点实现需要唯一消息 ID 的协议，则一些端点实现会隐式调用 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 组件自动调用 getMessageId （） 来获取消息 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，则消息将保持未解析状态，直到应用程序明确访问为止。大多数 Apache Camel 组件以相对原始形式的 raw 格式创建消息正文，例如，使用 byte[], ByteBuffer,InputStream, 或 OutputStream 等类型来代表它。这样可确保创建初始消息所需的开销最小。在需要更详细的信息格式时，组件通常依赖 类型转换器 或 汇总处理器。

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 “built-In Type Converters”）轻松地将消息从一个格式转换为另一个格式。Apache Camel API 中有多种方法公开类型转换功能。例如，可以将 convertBodyTo (Class type) 方法插入到路由中，以转换 In 消息的正文，如下所示：

```
from("SourceURL").convertBodyTo(String.class).to("TargetURL");
```

其中 In 消息的正文转换为 java.lang.String。以下示例演示了如何将字符串附加到 In 消息正文的末尾：

```
from("SourceURL").setBody(bodyAs(String.class).append("My Special Signature")).to("TargetURL");
```

在为末尾附加字符串前，消息正文转换为字符串格式。本例中不需要显式转换消息正文。您还可以使用：

```
from("SourceURL").setBody(body().append("My Special Signature")).to("TargetURL");
```

其中 append （） 方法会自动将消息正文转换为字符串，然后附加其参数。

在消息中键入转换方法

org.apache.camel.Message 接口公开一些明确执行类型转换的方法：

- getBody (Class<T> type) HEKETI-wagonReturns the message body as type, T.
- getHeader (String name, Class<T> type) swig-wagonReturns named 标头值，作为 type、T。

有关支持的转换类型的完整列表，请参阅 第 34.3 节 “built-In Type Converters”。

转换为 XML

除了支持简单类型（如 byte[]、Serte Buffer、String 等）之间的转换外，内置的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。这个转换比简单的转换更昂贵，因为它涉及解析整个消息，然后创建代表 XML 文档结构的节点树。您可以转换为以下 XML 文档类型：

- org.w3c.dom.Document
- javax.xml.transform.sax.SAXSource

XML 类型转换具有比更简单的转换更小的适用性。因为并非所有消息正文都符合 XML 结构，所以您必须记住这种类型的转换可能会失败。另一方面，在许多情况下，路由器只处理 XML 消息类型。

Marshalling 和 unmarshalling

Marshalling 涉及将高级别格式转换为低级格式，unmarshaling 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 marshalling 或 unmarshalling：

- marshal()
- unmarshal()

例如，要从文件中读取序列化 Java 对象，并将它 unmarshal 到 Java 对象，您可以使用 例 34.3

"unmarshalling a Java Object" 中显示的路由定义。

例 34.3. unmarshalling a Java Object

```
from("file://tmp/appfiles/serialized")
    .unmarshal()
    .serialization()
    .<FurtherProcessing>
    .to("TargetURL");
```

最终消息格式

当 In 消息到达路由的末尾时，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。相同的规则适用于到达源端点的 Out 消息。此转换通常使用 Apache Camel 类型转换器隐式执行。通常，这涉及从低级格式转换为另一个低级别格式，如从 byte[] 数组转换为 InputStream 类型。

34.3. BUILT-IN TYPE CONVERTERS

概述

这部分论述了 master 类型转换器支持的转换。这些转换内置在 Apache Camel 内核中。

通常，类型转换器通过方便的功能调用，如 Message.getBody (Class<T> type) 或 Message.getHeader (String name, Class<T> 类型)。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);
```

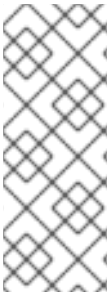
基本类型转换器

Apache Camel 提供了内置的类型转换器，其执行到以下基本类型的转换或从以下基本类型执行转换：

- 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，并非所有这些类型都是可切换的。内置转换器主要用于提供从文件和字符串类型 的转换。文件类型 可以转换为任何上述类型，但 Reader、Writer、和 String Reader 除外。String 类型可以转换为 File,byte[], ByteBuffer,InputStream, 或 StringReader。从 String 转换到 File 的工作原理，可将字符串解释为文件名。String,byte[], 和 ByteBuffer 的 trio 完全可转换。



注意

您可以通过在当前交换中设置 Exchange.CHARSET_NAME Exchange 属性来指定用于从 byte[] 转换到 String，并从 String 转换为 byte[] 的字符编码。例如，要使用 UTF-8 字符编码来执行转换，请调用 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 提供了内置的类型转换器，其执行转换至以下集合类型：

- *object[]*
- *java.util.Set*

- *java.util.List*

支持以上集合类型之间的转换。

映射类型转换器

Apache Camel 提供了内置的类型转换器，其执行转换至以下映射类型，或从以下映射类型执行转换：

- *java.util.Map*
- *java.util.HashMap*
- *java.util.Hashtable*
- *java.util.Properties*

前面的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 > 类型。

DOM 类型转换器

您可以执行到以下文档对象模型(DOM)类型的类型转换：

- *org.w3c.dom.Document HEKETI-rhacmconvertible from byte[], String,java.io.File, 和 java.io.InputStream.*
- *org.w3c.dom.Node*
- *来自 String 中的 javax.xml.transform.dom.DOMSource mtcconvertable from String.*

- 来自 byte[] 和 String 的 javax.xml.transform.Source mtcconvertable。

支持前面的 DOM 类型之间的转换。

SAX 类型转换器

您还可以执行到 javax.xml.transform.sax.SAXSource 类型的转换，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
- Source
- StreamSource
- DOMSource

Enum 类型转换器

Camel 提供了一个类型转换器，用于执行 String 到 enum 类型转换，其中字符串值将转换为来自指定枚举类的匹配的 enum 常量（匹配 区分大小写）。这个类型转换器很少需要转换消息正文，但它通常由 Apache Camel 内部用来选择特定选项。

例如，当设置日志级别选项时，以下值 INFO 会转换为 enum 常量：

```
<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案也可以正常工作：

```
<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。有关如何实现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，当 Apache Camel 需要生成唯一的 ID swig-wagonin 特殊时，会调用此 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，则可能会希望替换默认的 UUID 生成器，长度为 36 个字符（如 WebSphere MQ）。另外，使用简单计数器（请参阅 SimpleUuidGenerator）用于测试目的，可以方便生成 ID。

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，这些生成器在内核中提供：

- org.apache.camel.impl.ActiveMQUuidGenerator hmac- (Default) 生成与 Apache ActiveMQ 使用的相同 ID 样式。这个实现可能不适用于所有应用程序，因为它使用在云计算（如 Google App Engine）中禁止的一些 JDK API。
- org.apache.camel.impl.SimpleUuidGenerator HEKETI- && implementations 简单计数器 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 HEKETI-wagon 实现基于 java.util.UUID 类型的 ID。由于 java.util.UUID 已同步，这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，实施 org.apache.camel.spi.UuidGenerator 接口，它显示在 例 34.5 “UuidGenerator 接口” 中。必须实施 generateUuid （） 来返回唯一 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 替换默认的 UUID 生成器，请调用当前 CamelContext 对象的 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

在激活任何路由之前，应在启动过程中调用 setUuidGenerator （） 方法。

使用 Spring 指定 UUID 生成器

要使用 Spring 替换默认 UUID 生成器，您需要做的都是使用 Spring bean 元素创建 UUID 生成器的实例。创建 camelContext 实例时，它会自动查找 Spring registry，搜索实现 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] 如果没有活跃的方法，返回的值将为 null。

第 35 章 实施处理器

摘要

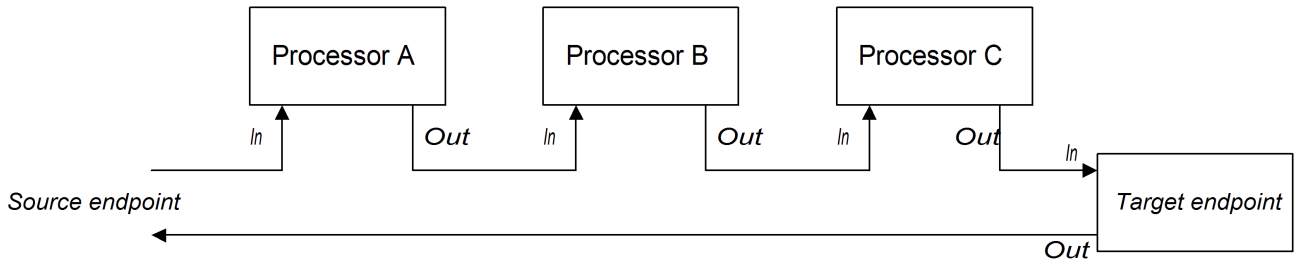
Apache Camel 允许您实施自定义处理器。然后，您可以将自定义处理器插入到路由中，以便在通过路由时对交换对象执行操作。

35.1. 处理模型

pipelining 模型

pipelining 模型 描述了处理器在 第 5.4 节 “管道和过滤器” 中排列的方式。pipelining 是处理一系列端点（生产者端点只是特殊类型的处理器）的最常见方法。当处理器以这种方式排列时，交换的 In 和 Out 信息将按照 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像服务，其中 In 消息与请求类似，而 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）实现。

例如，例 35.1 “Java DSL Pipeline” 显示一个 Java DSL 管道，它由两个处理器、ProcessorA、ProcessorB 和生成者端点 TargetURI 组成。

例 35.1. Java DSL Pipeline

```
from(SourceURI).pipeline(ProcessorA, ProcessorB, TargetURI);
```

35.2. 实施简单处理器

概述

这部分论述了如何实现在委派交换到路由中的下一个处理器前执行消息处理逻辑的简单处理器。

处理器接口

简单的处理器是通过实施 org.apache.camel.Processor 接口创建的。如 例 35.2 “处理器接口” 所示，接口定义了单一方法 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;
}
```

实施处理器接口

要创建简单的处理器，您必须实施 Processor 接口，并为 process （） 方法提供逻辑。例 35.3 “简单处理器实施” 显示了简单处理器实施的概要。

例 35.3. 简单处理器实施

```
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 （） 方法中的所有代码都会在将交换对象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消息正文和标头值的示例，请参阅 第 35.3 节 “访问消息内容”。

将简单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

访问消息标头

消息标头通常包含路由器视角的最有用的消息内容，因为标头通常旨在在路由器服务中处理。要访问标头数据，您必须首先从交换对象获取消息（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示例，它访问名为 Authorization 的标头值。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”。

访问消息正文

您还可以访问消息正文。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 “邮件接口”。

35.4. EXCHANGEHELPER 类

概述

org.apache.camel.util.ExchangeHelper 类是一种 Apache Camel 实用程序类，提供实施处理器时有用的方法。

解析端点

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” 演示了如何从交换实例 交换创建新文件端点

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 类提供了一些静态方法 getMandatoryBeanProperty（），它将在 Exchange 类上包装对应的 getBeanProperty（） 方法。它们之间的区别在于，如果对应的属性不可用，原始 getBeanProperty（） 访问器会返回 null，而 getMandatoryBeanProperty（） 包装器方法会抛出 Java 异常。以下 wrapper 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。几种不同的交换模式也与 保存 Out 消息兼容。为了快速检查交换对象是否可以保存 In 消息或 Out 消息，ExchangeHelper 类提供了以下方法：

```
public final class ExchangeHelper {
    ...
    public static boolean isInCapable(Exchange exchange) { ... }

    public static boolean isOutCapable(Exchange exchange) { ... }
    ...
}
```

获取 In 消息 MIME 内容类型

如果要查找交换的 In 消息的 MIME 内容类型，您可以通过调用 ExchangeHelper.getContentType (exchange) 方法来访问它。要实现此目的，ExchangeHelper 对象查找 In message's Content-Type header to the method 依赖于底层组件来填充标头值的值。

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器方法扩展类型转换机制。

36.1. 类型转换器架构

概述

这部分论述了类型转换器机制的整体架构，如果您需要编写自定义类型转换器，则必须理解这些架构。如果您只需要使用内置类型转换器，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/工作程序模式。有许多 worker 类型转换器，各自能够执行有限的类型转换，以及一个控制器类型转换器，它聚合了 worker 执行的类型转换。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，并将转换任务委派给该 worker。

对于类型转换机制的用户，控制器类型转换器是最重要的，因为它提供了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。要获取对控制器类型转换器的引用，您需要调用 CamelContext.getTypeConverter （） 方法。例如，如果您有一个交换对象 Exchange，您可以获取对控制器类型转换器的引用，如 例 36.2 “获取控制器类型转换程序” 所示。

例 36.2. 获取控制器类型转换程序

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
```

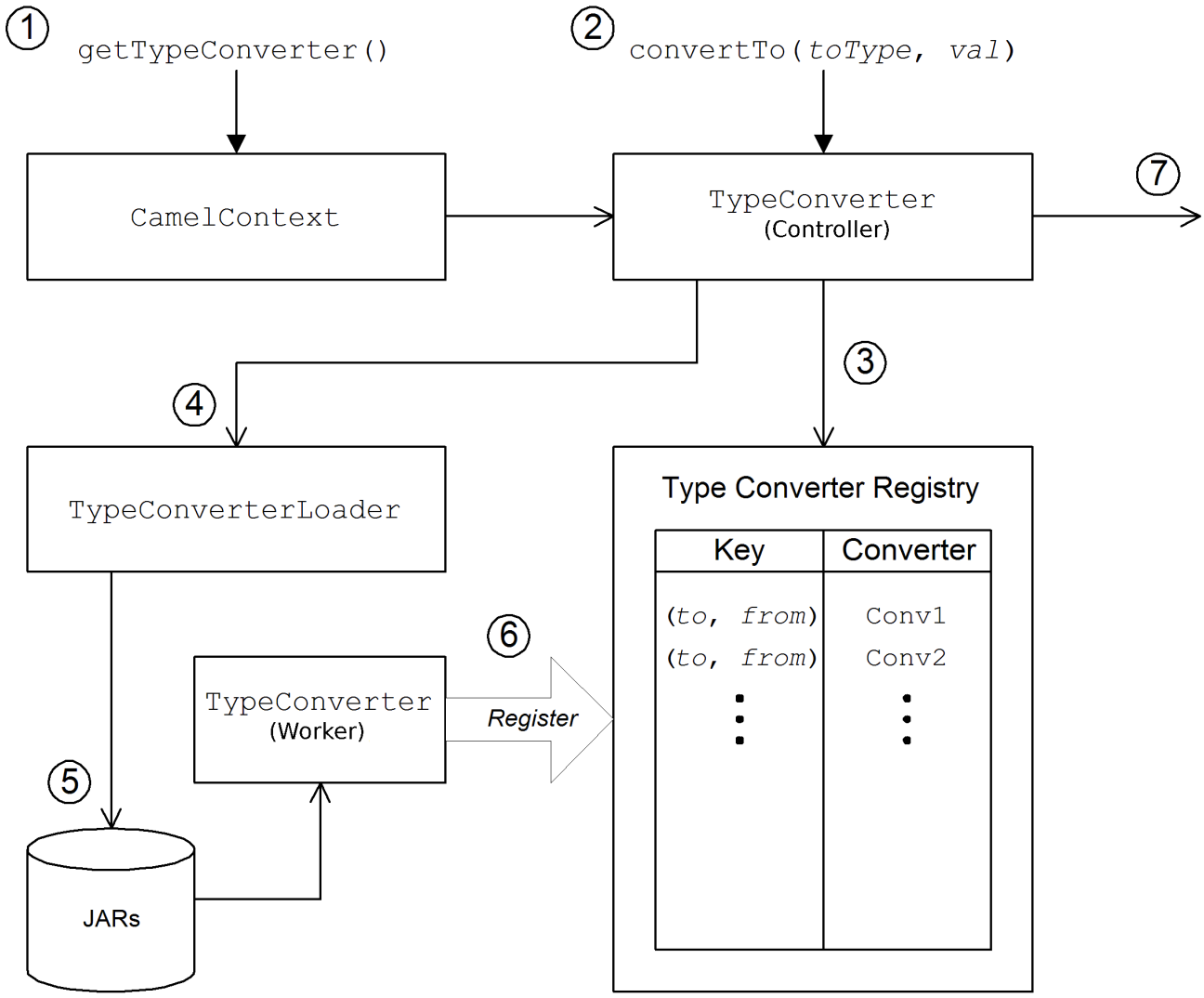
类型转换器

控制器类型转换器使用 类型转换器 来填充 worker 类型的转换器的 registry。类型转换器是实施 TypeConverterLoader 接口的任何类。Apache Camel 目前只使用一种类型 converter loader the 注解类型转换器加载器 (AnnotationTypeConverterLoader 类型)。

类型转换过程

图 36.1 “类型转换过程” 概述类型转换过程，显示将给定数据值（值 转换为指定类型）中涉及的步骤。

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。转换过程的第一步是通过调用 CamelContext.getTypeConverter （） 来检索控制器类型转换器。

2. 类型转换是通过在控制器类型转换器上调用 convertTo （） 方法启动的。这个方法指示类型转换器将数据对象(值为)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，所以它会通过检查类型为映射器的 registry 来找到适当的 worker 类型转换器。类型为转换器的 registry 由类型映射对键 (toType,fromType)。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 无法在 registry 中找到合适的类型转换器，控制器类型转换器将使用类型转换器加载一个新的类型转换器。

5. 类型转换器加载程序搜索 classpath 上的可用 JAR 库，以查找合适的类型转换器。目前，使用的加载程序策略由注解类型转换器实施，它尝试加载由 org.apache.camel.Converter 注解注解注解的类。请参阅 “创建 TypeConverter 文件”一节。

6. 如果类型转换器成功，则会加载一个新的 worker 类型转换器，并输入到类型转换器 registry。然后，使用这个类型转换器将 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。如果转换不成功，则返回 null。

36.2. 处理重复类型转换器

如果添加了重复的类型转换器，您可以配置必须发生的情况。

在 TypeConverterRegistry (See 第 36.3 节 “使用注解实现类型转换程序”)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);
```

此代码中的 覆盖 可以被 Ignore 或 Fail 替换，具体取决于您的要求。

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. 使用注解实现类型转换程序

概述

通过添加新的 worker 类型转换器，可以轻松地自定义类型转换。本节论述了如何实施 worker 类型转换器以及如何将其与 Apache Camel 集成，以便注解类型转换器自动载入。

如何实施类型转换器

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. *“打包类型转换器”一节.*

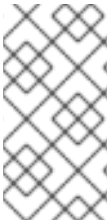
实施注解的转换器类

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个 静态方法。每个转换器方法都会使用一个参数来定义 from 类型，该参数可以选择使用第二个 Exchange 参数，并且具有定义 to 类型的非无效返回值。类型转换器使用 Java 反映来查找注释的方法并将其集成到类型转换器机制中。例 36.3 “Annotated Converter 类示例” 显示了一个注释转换器类的示例，它定义了一个转换器方法从 java.io.File 转换为 java.io.InputStream 和另一个转换器方法（带有 Exchange 参数），用于从 byte[] 转换为 String。

例 36.3. Annotated 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 （） 方法负责执行从 File 类型到 InputStream 类型的转换，toString （） 方法负责执行从 byte[] 类型的转换到 String 类型。



注意

方法名称是 unimportant，可以是您选择的任何内容。重要的是参数类型、返回类型和 @Converter 注释。

创建 TypeConverter 文件

要启用自定义转换器的发现机制（由 注解类型转换器实现），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转换器类的完全限定域名(FQN)的逗号分隔列表。例如，如果您希望类型转换器搜索 YourPackageName.YourClassName 软件包来注解的转换器类，则 TypeConverter 文件将具有以下内容：

```
com.PackageName.FooClass
```

启用发现机制的一种替代方法是仅将软件包名称添加到 TypeConverter 文件中。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 标签的软件包进行扫描。使用 FQN 方法速度更快，它是首选的方法。

打包类型转换器

类型转换器被打包为 JAR 文件，其中包含编译的自定义类型转换器和 META-INF 目录。将此 JAR 文件放在 classpath 上，使其可供您的 Apache Camel 应用程序使用。

回退转换器方法

除了使用 @Converter 注释定义常规转换器方法外，您还可以选择使用 @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。

常规转换器方法和回退转换器方法之间的基本区别在于，定义了常规转换器以在特定类型（例如，从

byte[] 到 String）之间执行转换，回退转换器可以在 任意 对类型之间执行转换。它最多是回退转换器方法正文中的代码，以找出它可以执行哪些转换。在运行时，如果转换无法由常规转换器执行，控制器类型转换器会迭代每个可用的回退转换器，直到找到可执行转换的内容。

回退转换器的方法签名可以有以下格式之一：

```
// 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）显示了一个回退转换器，它可以转换 **GenericFile** 对象的正文，利用类型转换器 **registry** 中已可用的类型转换器：

```
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. 直接实施 TYPE CONVERTER

概述

通常，推荐的方法是使用注解的类，如上一节中 第 36.3 节 “使用注解实现类型转换程序” 所述。但是，如果要对类型转换器的注册完全控制，您可以实施自定义 worker 类型转换器，并将其直接添加到类型转换器 registry 中，如下所述。

实施 TypeConverter 接口

要实施您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，其中整数值用于在 MyOrder 对象中初始化顺序 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 context 是当前的 org.apache.camel.CamelContext 实例。addTypeConverter （） 方法将 MyOrderTypeConverter 类从 String.class 注册到 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，则只能声明 <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，可以声明多个 <bean>s。

第 37 章 生产者和消费者模板

摘要

Apache Camel 中的生成者和消费者模板在 Spring 容器 API 的功能后建模，通过一个简单易用的 API （称为 模板 ）来提供资源的访问。如果是 Apache Camel，生产者模板和消费者模板提供了简化的接口，用于向和接收来自生成者端点和消费者端点的消息。

37.1. 使用 PRODUCER 模板

37.1.1. Producer 模板简介

概述

producer 模板支持各种不同的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，作为消息正文，作为带单个标头设置的消息正文，等等），有支持同步和异步调用方式的方法。总体而言，生成者模板方法可分为以下类别：

- 同步调用
- 使用处理器进行同步调用
- 异步调用
- 使用回调进行异步调用

或者，请参阅 第 37.2 节 “使用 Fluent Producer 模板”。

同步调用

同步调用端点的方法具有 发送后缀（）和 请求 后缀（） 的形式的名称。例如，使用默认消息交换模式(MEP)或明确指定的 MEP 调用端点的方法被命名为 send （）, sendBody （）, 和 sendBodyAndHeader （） （这些方法分别发送 Exchange 对象、消息正文或消息正文和标头值）。如果要强制 MEP 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，并使用它来发送消息正文到 activemq:MyQueue 端点。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" );
```

使用处理器进行同步调用

同步调用的一个特殊情况是您为 send （） 方法提供 Processor 参数而不是 Exchange 参数。在这种情况下，生成者模板会隐式要求指定的端点创建 Exchange 实例（通常并非始终默认具有 InOnly MEP）。然后，此默认交换传递到处理器，它将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 类按照以下示例所示实施。除了设置 In message body （如此处所示），您还可以初始化消息标题和交换属性。

```
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 asyncSendSuffix（）和 asyncRequestSuffix（） 形式的名称。例如，使用默认消息交换模式(MEP)或明确指定的 MEP 调用端点的方法被命名为 asyncSend （） 和 asyncSendBody （） （其中这些方法分别发送 Exchange 对象或消息正文）。如果要强制 MEP 为 InOut (request/reply 语义)，您可以调用 asyncRequestBody （）、syncRequestBodyAndHeader （）、syncRequestBodyAndHeader （） 和 asyncRequestBodyAndHeaders （） 方法。

以下示例演示了如何异步向 direct:start 端点发送交换。asyncSend （） 方法返回一个 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 模板还提供异步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 （） 方法块的第一个版本，直到调用完成并且回复消息可用为止。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，它将返回的消息正文转换为指定类型（使用内置类型转换器）。

以下示例演示了如何使用 asyncRequestBody （） 方法将消息正文发送到 direct:start 端点。然后，使用 blocking 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);
```

使用回调进行异步调用

在前面的异步示例中，请求消息在子线程中分配，而回复则由主线程检索和处理。**producer** 模板还为您提供了选项，但子线程中的处理回复使用 **asyncCallback** （），, **asyncCallbackSendBody** （） 方法之一，或 **asyncCallbackRequestBody** （） 方法。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时在子线程中自动调用。

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);
}
```

如果在收到正常回复时调用 **onComplete** （） 方法，并且在收到错误消息回复时调用 **onFailure** （） 方法。只有其中一种方法会调用回来，因此您必须覆盖这两个方法，以确保处理所有类型的回复。

以下示例演示了如何将交换发送到 **direct:start** 端点，其中 **SynchronizationAdapter** 回调对象在子线程中处理回复消息。

```
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());
    }
});
```

其中 Synchronization Adapter 类是同步接口的默认实现，您可以覆盖它来提供您自己对 onComplete （） 和 onFailure （） 回调方法的定义。

您仍然有从主线程访问回复的选项，因为 asyncCallback （） 方法也会返回 Future object swig-wagon 例如：

```
// 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 方法是一组可用于调用制作者端点的方法集合，其中当前线程块直到方法调用完成并且收到回复（若有）。这些方法与任何类型的消息交换协议兼容。

发送交换

基本的 send （） 方法是一种通用方法，利用交换的消息交换模式(MEP)将 Exchange 对象的内容发送到端点。返回值是您在生成者端点处理后获取的交换（可能包含 Out 消息，具体取决于 MEP）。

发送交换有三种 send （） 方法，允许您以以下一种方式指定目标端点： 作为默认端点、端点、端点 URI 或 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 “使用处理器进行同步调用”一节 ）。

发送由处理器填充的交换的 send （） 方法可让您以以下一种方式指定目标端点： 作为默认端点、端点 URI 或 Endpoint 对象。此外，您还可以通过提供 pattern 参数来指定交换的 MEP，而不接受默认值。

```
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

如果您只关注您要发送的消息正文的内容，您可以使用 sendBody （） 方法提供消息正文作为参数，并让制作者模板将正文插入到默认交换对象中。

sendBody （） 方法允许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 Endpoint 对象。此外，您还可以通过提供 pattern 参数来指定交换的 MEP，而不接受默认值。没有 模式参数的方法返回 void （即使调用在某些情况下可能会给回复），并且具有 pattern 参数的方法会返回 Out 消息的正文（如果有一个）或 In 消息的正文（否则为准）。

```
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
);
```

发送消息正文和标题

出于测试目的，尝试 单个 标头设置的影响和 sendBodyAndHeader （） 方法对这种标头测试很有用。您提供消息正文和标头设置作为 sendBodyAndHeader （） 的参数，并允许制作者模板将正文和标头设置插入到默认交换对象中。

sendBodyAndHeader （） 方法允许您以以下一种方式指定目标端点：作为默认端点，作为端点 URI 或 Endpoint 对象。此外，您还可以通过提供 pattern 参数来指定交换的 MEP，而不接受默认值。没有 模式 参数的方法返回 void （即使调用在某些情况下可能会给回复），并且具有 pattern 参数的方法会返回 Out 消息的正文（如果有一个）或 In 消息的正文（否则为准）。

```
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，除了提供单个标头设置外，这些方法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影响。您提供消息正文和属性设置作为 sendBodyAndProperty （） 的参数，并让制作者模板负责将正文和交换属性插入到默认的交换对象中。

sendBodyAndProperty （） 方法允许您以以下一种方式指定目标端点：作为默认端点，作为端点 URI 或 Endpoint 对象。此外，您还可以通过提供 pattern 参数来指定交换的 MEP，而不接受默认值。没有 模式 参数的方法返回 void （即使调用在某些情况下可能会给回复），并且具有 pattern 参数的方法会返回 Out 消息的正文（如果有一个）或 In 消息的正文（否则为准）。

```
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. 使用 InOut Pattern 同步请求

概述

同步请求 方法与同步发送方法类似，但请求方法强制消息交换模式为 InOut （代表请求/回复语义）。因此，如果您预期从生成者端点收到回复，通常最好使用同步请求方法。

请求由处理器填充的交换

基本 request （） 方法是一种通用方法，它使用处理器填充默认交换，并强制消息交换模式成为 InOut （因此调用遵循请求/回复语义）。返回值是您在生成者端点处理后获取的交换，其中 Out 消息包含回复消息。

发送由处理器填充的交换的 request （） 方法可让您以以下一种方式指定目标端点： 作为端点 URI 或 Endpoint 对象。

```
Exchange request(String endpointUri, Processor processor);
Exchange request(Endpoint endpoint, Processor processor);
```

请求消息正文

如果您只关注请求中的消息正文内容，并在回复中，您可以使用 requestBody （） 方法提供请求消息正文作为参数，并让制作者模板将正文插入到默认的交换对象中。

requestBody （） 方法允许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 Endpoint 对象。返回值是回复消息的正文(Out message body)，可以是 plain Object，也可以使用内置类型转换器（请参阅 第 34.3 节 “built-In Type Converters”）转换为特定类型的 T。

```
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影响。您提供消息正文和标头设置作为 requestBodyAndHeader （） 的参数，并允许制作者模板将正文和交换属性插入到默认交换对象中。

requestBodyAndHeader （） 方法允许您以以下一种方式指定目标端点：作为端点 URI 或 Endpoint 对象。返回值是回复消息的正文(Out message body)，可以是 plain Object，也可以使用内置类型转换器（请参阅 第 34.3 节 “built-In Type Converters”）转换为特定类型的 T。

```
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，除了提供单个标头设置外，这些方法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 模板提供各种异步调用制作者端点的方法，以便在等待调用完成时阻止主线程，并且稍后可以检索回复消息。本节中描述的异步发送方法与任何类型的消息交换协议兼容。

发送交换

基本的 asyncSend （） 方法采用 Exchange 参数，并使用指定交换的消息交换模式(MEP)异步调用端点。返回值是一个 java.util.concurrent.Future 对象，它是稍后用来收集回复消息的票据，请参阅 “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 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 （） 方法的一个简单变体是使用处理器填充默认交换，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 Endpoint 对象。

```
Future<Exchange> asyncSend(String endpointUri, Processor processor);
Future<Exchange> asyncSend(Endpoint endpoint, Processor processor);
```

发送消息正文

如果您只关注您要发送的消息正文的内容，您可以使用 asyncSendBody （） 方法异步发送消息正文，并允许制作者模板将正文插入到默认交换对象中。

asyncSendBody （） 方法允许您以以下一种方式指定目标端点：作为端点 URI 或 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，但请求方法强制消息交换模式为 InOut （代表请求/回复语义）。因此，如果您预期从生成者端点收到回复，通常最好使用异步请求方法。

请求消息正文

如果您只关注请求中的消息正文内容，并在回复中，您可以使用 requestBody （） 方法提供请求消息正文作为参数，并让制作者模板将正文插入到默认的交换对象中。

asyncRequestBody （） 方法允许您以以下一种方式指定目标端点：作为端点 URI 或 Endpoint 对象。从 Future 对象检索的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，或使用内置的类型 T 转换程序（请参阅 “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影响。您提供消息正文和标头设置作为 asyncRequestBodyAndHeader （） 的参数，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。

asyncRequestBodyAndHeader （） 方法可让您以以下一种方式指定目标端点： 作为端点 URI 或 Endpoint 对象。从 Future 对象检索的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，或使用内置的类型 T 转换程序（请参阅 “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，这些方法允许您指定标头设置的完整散列映射。

```
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. 使用回调异步发送

概述

producer 模板还提供 选项，用于处理用于调用制作者端点的同一子线程中的回复消息。在这种情况下，您可以提供一个回调对象，它会在收到回复消息后立即在子线程中调用。换句话说，使用回调方法的异步发送 可让您在主线程中启动调用，然后具有生成者端点的所有关联处理因此，等待回复并在子线程中异步处理回复。

发送交换

基本 asyncCallback （） 方法采用 Exchange 参数，并使用指定交换的消息交换模式(MEP)异步调用端点。这个方法类似于交换的 asyncSend （） 方法，但它采用额外的 org.apache.camel.spi.Synchronization 参数，它是一个具有两种方法： onComplete （） 和 onFailure （） 的回调接口。有关如何使用 同步 回调的详情，请参考 “使用回调进行异步调用”一节。

以下 asyncCallback （） 方法可让您以以下一种方式指定目标端点： 作为端点 URI 或 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 （） 方法调用处理器来填充默认交换，并强制消息交换模式成为 InOut（因此调用模糊请求/回复语义）。

以下 asyncCallback （） 方法可让您以以下一种方式指定目标端点： 作为端点 URI 或 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

如果您只关注您要发送的消息正文的内容，您可以使用 asyncCallbackSendBody （） 方法异步发送消息正文，并允许制作者模板将正文插入到默认交换对象中。

asyncCallbackSendBody （） 方法允许您以以下一种方式指定目标端点： 作为端点 URI 或 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

如果您只关注请求中的消息正文内容，并在回复中，您可以使用 asyncCallbackRequestBody （）

方法提供请求消息正文作为参数，并使制作者模板负责将正文插入到默认的交换对象中。

asyncCallbackRequestBody （） 方法允许您以以下一种方式指定目标端点： 作为端点 URI 或 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

从 Camel 2.18 开始提供

FluentProducerTemplate 接口为构建生成者提供了流畅的语法。DefaultFluentProducerTemplate 类实现了 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流畅制作者模板：

```
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 模板

概述

consumer 模板提供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形式或消息正文形式接收传入的消息（其中消息正文可以使用内置类型转换器）转换为特定类型的消息。

轮询交换示例

您可以使用消费者模板轮询消费者端点，以使用以下轮询方法之一进行交换：阻塞 receive （）；receive （） with a timeout; 或 receiveNoWait （），它会立即返回。由于消费者端点代表服务，因此在尝试轮询交换前，通过调用 start （） 来启动服务线程也很重要。

以下示例演示了如何使用 blocking receive （） 方法从 seda:foo consumer 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();
```

其中，消费者模板实例 consumer 使用 CamelContext.createConsumerTemplate （） 方法进行实例化，并通过调用 ConsumerTemplate.start （） 来启动消费者服务线程。

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消费者端点：block receiveBody （）；receiveBody （） 带有 timeout; 或 receiveBodyNoWait （），后者会立即返回。如上例中所示，在尝试轮询交换前，通过调用 start （） 来启动服务线程也很重要。

以下示例演示了如何使用 blocking receiveBody （） 方法从 seda:foo consumer 端点轮询传入的消息正文：

```
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();
```

轮询交换的方法

从消费者端点进行轮询 交换 有三个基本方法： receive （） 没有无限期超时块；对于指定的毫秒内带有超时块的 receive （） ；而 receiveNoWait （） 是非阻塞的。您可以将消费者端点指定为端点 URI 或 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);
```

轮询消息正文的方法

从消费者端点中轮询 消息正文 有三种基本方法： receiveBody （） 没有超时块；对于指定的毫秒内

带有超时块的 receiveBody （） ；而 receiveBodyNoWait （） 是非阻塞性的。您可以将消费者端点指定为端点 URI 或 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

本章介绍了可用于实施 Apache Camel 组件的方法的一般概述。

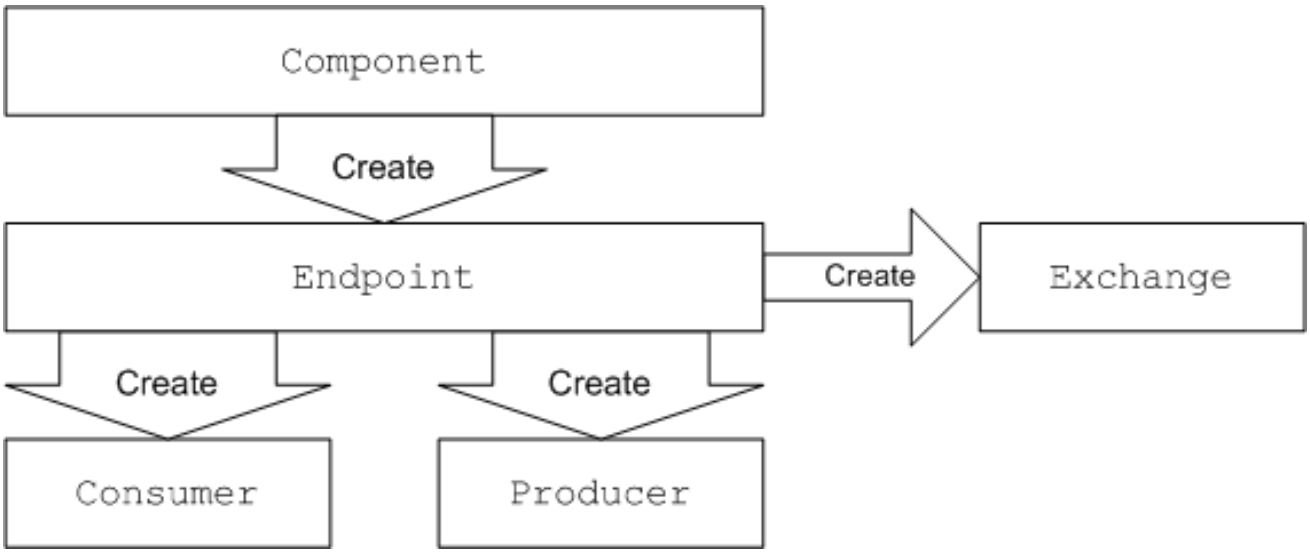
38.1. 组件架构

38.1.1. 组件工厂模式

概述

Apache Camel 组件由一组通过工厂模式相互相关的类组成。主入口点指向组件对象本身(org.apache.camel. Component 类型的实例)。您可以使用 组件 对象作为工厂创建 Endpoint 对象，后者充当创建 消费者、Producer 和 Exchange 对象的因素。这些关系总结在 图 38.1 “组件工厂模式”

图 38.1. 组件工厂模式



组件

组件实现是一个端点工厂。组件实施者的主要任务是实施 Component.createEndpoint （） 方法，它负责根据需要创建新端点。

每个组件都必须与端点 URI 中显示的 组件前缀 关联。例如，文件组件通常与 文件 前缀关联，该前缀可在 file://tmp/messages/input 等端点 URI 中使用。在 Apache Camel 中安装新组件时，您必须定义特定组件前缀和实施组件的类名称之间的关联。

端点

每个端点实例封装特定的端点 URI。每次 Apache Camel 遇到新端点 URI 时，它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 接口定义以下工厂方法：

- createConsumer （） 和 createPollingConsumer （） HEKETI-wagonCreates 是一个消费者端点，它代表路由开头的源端点。
- createProducer （） 5-4Creates a producer 端点，它代表路由末尾的目标端点。
- createExchange （） HEKETI-wagonCreates 是一个交换对象，它封装了传递和关闭路由的消息。

消费者

消费者端点 使用 请求。它们始终出现在路由的开头，它们封装负责接收传入请求和分配传出回复的代码。从面向服务的角度来说，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，您可以遵循许多不同的模式。这些模式在 第 38.1.3 节 “消费者模式和线程” 中进行了描述。

制作者

生产者端点 生成 请求。它们始终出现在路由的末尾，它们封装负责分配传出请求并接收传入回复的代码。从面向服务的视角中，生成者代表 服务消费者。

生产者必须实施 org.apache.camel.Producer 接口。您可以选择实施制作者，以支持异步处理风格。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，由请求消息及其相关回复组成。

交换必须实施 org.apache.camel.Exchange 接口。默认实施 DefaultExchange 足以满足许多组件实

现的情况。但是，如果您想要将额外的数据与交换相关联，或者有交换前进行额外的处理，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：

- 在消息 swig-Cloneholds the current 信息中。
- out message swig-wagontemporarily 包含回复信息。

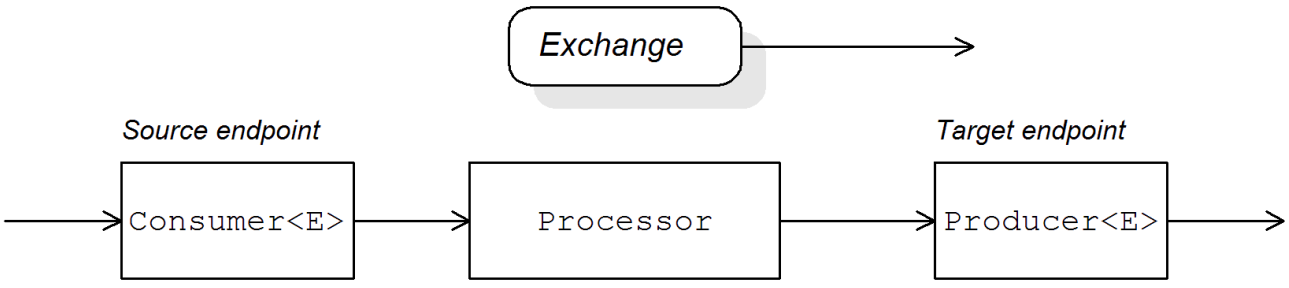
所有消息类型都由同一 Java 对象 org.apache.camel.Message 表示。并不总是需要自定义消息实现 swig-wagonthe default 实现 DefaultMessage，通常就足够了。

38.1.2. 在路由中使用组件

概述

Apache Camel 路由本质上是 org.apache.camel.Processor 类型的处理器管道。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 “路由中的消费者和生成实例” 中进行了说明。

图 38.2. 路由中的消费者和生成实例



源端点

在路由开始时，您有源端点，该端点由 org.apache.camel.Consumer 对象表示。源端点负责接受传入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 “组件工厂模式” 所述。

Processors

管道中的每个中间节点都由处理器对象（实现 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现。

目标端点

在路由的末尾，目标端点由 org.apache.camel.Producer 对象表示。由于它位于处理器管道的末尾，因此生成者也是处理器对象（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 根据端点 URI 的组件前缀创建适当的 Producer 类型。

38.1.3. 消费者模式和线程

概述

用于实现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：

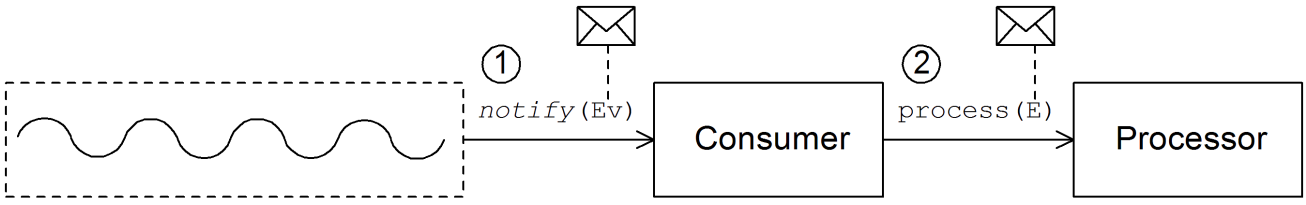
- 事件驱动的模式 HEKETI -wagon The consumer 由外部线程驱动。
- 调度的轮询模式 HEKETI-Clone The consumer 由专用线程池驱动。
- 轮询模式 HEKETI-Clone 线程模型未定义。

事件驱动的模式

在事件驱动的模式中，当应用的另一部分（通常是第三方库）调用由消费者实施的方法时，会启动传入请求的处理。事件驱动的消费者的一个很好的例子是 Apache Camel JMX 组件，其中事件由 JMX 库启动。JMX 库调用 handleNotification （） 方法来发起请求处理 processing swig-wagon 请参见 例 41.4 “JMXConsumer 实现” 以了解详细信息。

图 38.3 “event-Driven Consumer” 显示事件驱动的消费者模式的概要。在本例中，假设处理是由对 notify （） 方法的调用触发的。

图 38.3. event-Driven Consumer



事件驱动的消费者处理传入的请求，如下所示：

1. 消费者必须实施接收传入事件的方法（在 图 38.3 “event-Driven Consumer” 中，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驱动的。

   例如，如果是 JMX 消费者实施，使用者实施 NotificationListener.handleNotification（） 方法，以便从 JMX 接收通知。驱动消费者处理的线程是在 JMX 层中创建的。

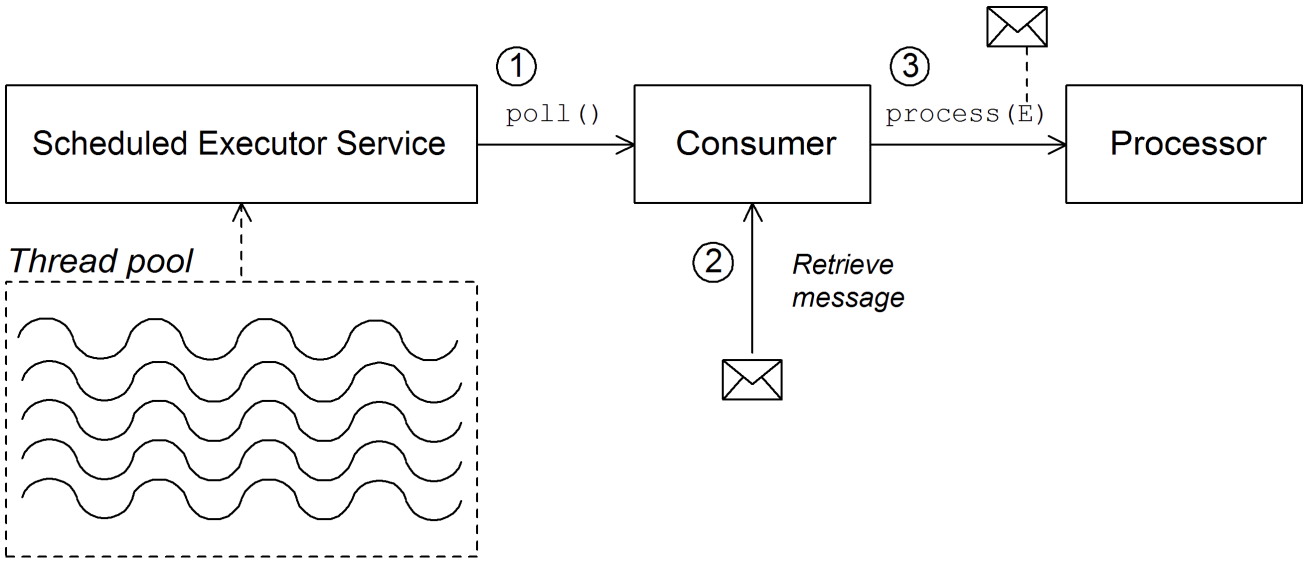
2. 在 notify（） 方法的正文中，消费者首先将传入的事件转换为交换对象 E，然后在路由中的下一个处理器上调用 process （），并将交换对象作为参数传递。

调度的轮询模式

在调度的轮询模式中，使用者定期检查请求是否到达，从而检索传入的请求。检查请求由内置的计时器类（计划的 executor 服务）自动调度，该服务 是由 java.util.concurrent 库提供的标准模式。调度的 executor 服务以时段执行特定的任务，并且管理一个用于运行任务实例的线程池。

图 38.4 “Scheduled Poll Consumer” 显示调度的轮询消费者模式的概要。

图 38.4. Scheduled Poll Consumer



调度的轮询消费者处理传入的请求，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被过后，调度的 executor 服务会尝试从池中获取可用线程（默认为池中的五个线程）。如果有可用

的线程可用，它会使用该线程调用消费者的 poll （） 方法。

2. 消费者的 poll （） 方法旨在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。如果没有可用的请求，poll （） 方法会立即返回。

3. 如果请求消息可用，则消费者将其插入到交换对象中，然后在路由中的下一个处理器上调用 process （），并将交换对象作为其参数。

轮询模式

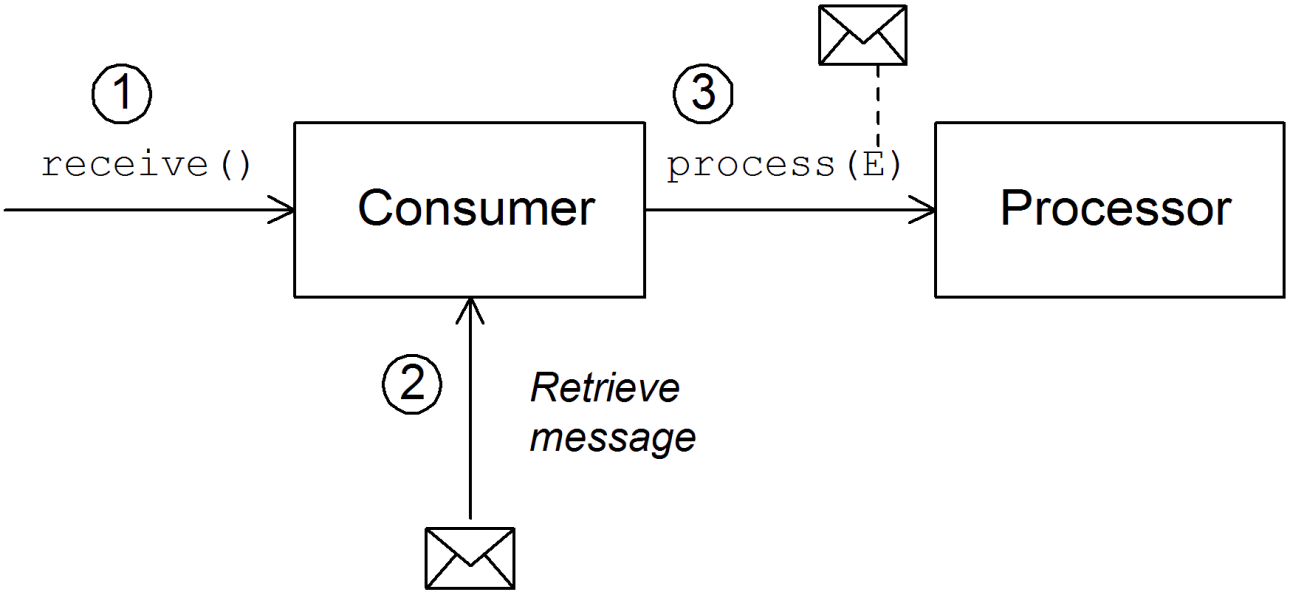
在轮询模式中，当第三方调用消费者的轮询方法之一时，启动传入请求的处理：

- receive()
- receiveNoWait()
- receive （长超时）

组件实施最多用于定义在轮询方法上启动调用的确切机制。这种机制不在轮询模式中指定。

图 38.5 “polling Consumer” 显示轮询消费者模式的概述。

图 38.5. polling Consumer



轮询消费者处理传入的请求，如下所示：

1. *每当调用使用者轮询方法之一时，都会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。如果当前没有消息，则行为取决于调用哪个接收方法。*

   - *receiveNoWait （） 立即返回*
   - *receive （长超时） 等待指定的超时间隔[2] 返回前*
   - *receive （） 等待收到消息*

3. *如果请求消息可用，则消费者将其插入到交换对象中，然后在路由中的下一个处理器上调用 process （），并将交换对象作为其参数。*

38.1.4. 异步处理

概述

在处理交换时，生成者端点通常遵循 同步 模式。当生成者上的管道调用 process （） 中的上述处理器时，process （） 方法会阻止，直至收到回复。在这种情况下，处理器的线程会被阻断，直到生成者完成发送请求和接收回复的周期。

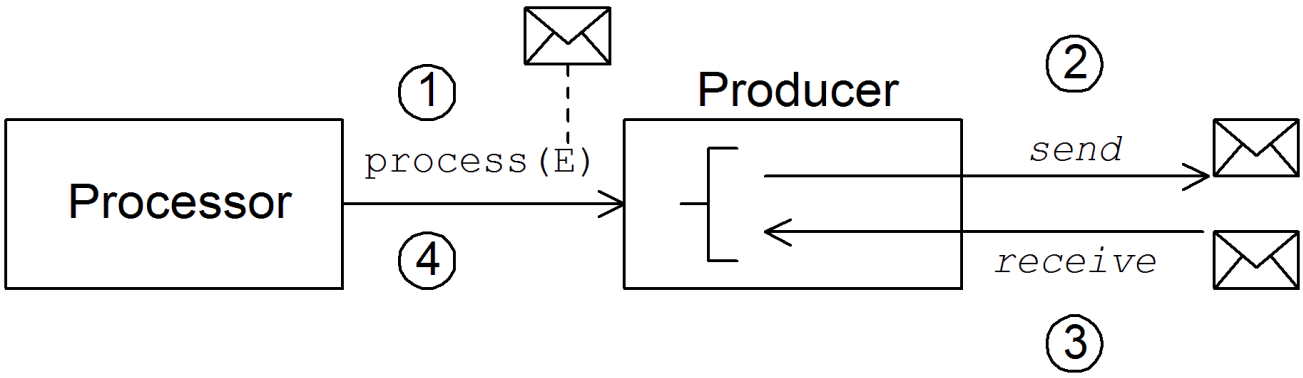
但是，有时您可能希望将前面的处理器与生成者分离，因此处理器的线程会立即释放，并且 process （） 调用 不会 阻断。在这种情况下，您应该使用 异步 模式实施制作者，这为前面的处理器提供调用 process （） 方法的非阻塞版本的选项。

为了让您了解不同的实施选项，本节描述了用于实施制作者端点的同步和异步模式。

同步制作者

图 38.6 “同步 Producer” 显示同步制作者的概述，其中前面的处理器块直到生产者完成交换为止。

图 38.6. 同步 Producer



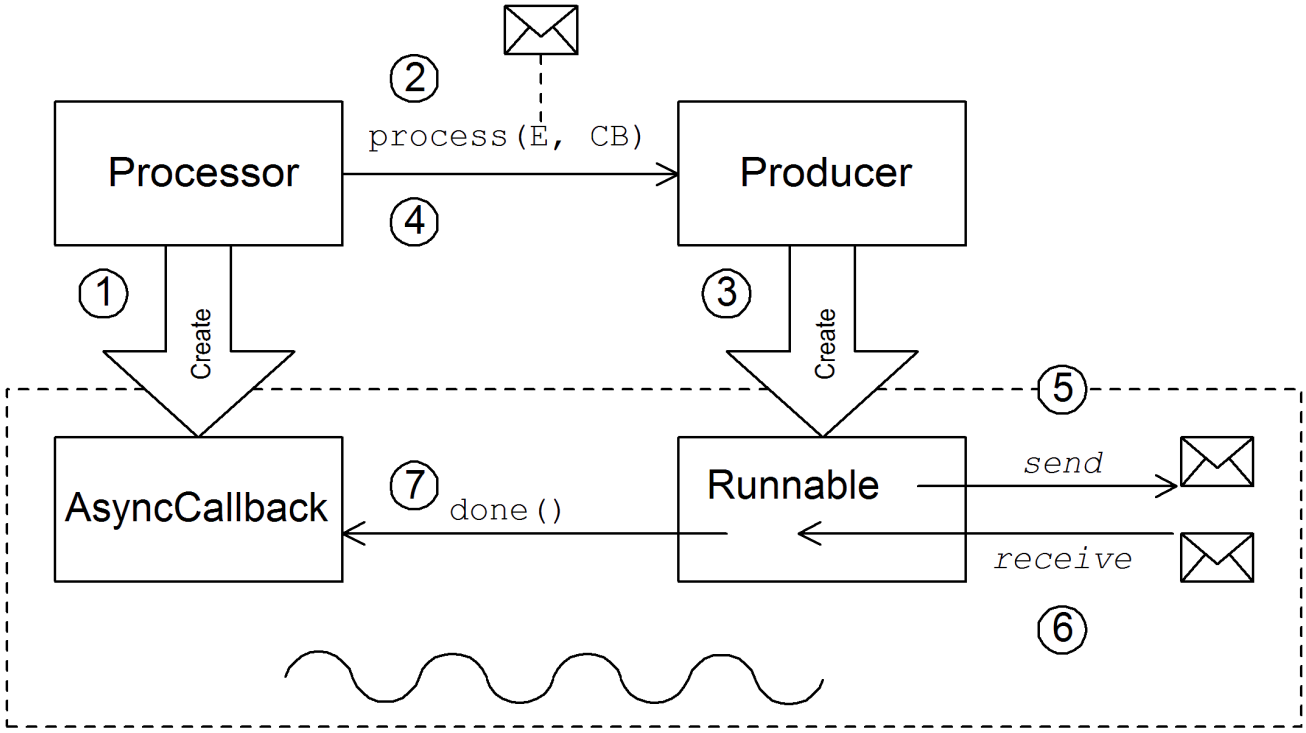
同步制作者按如下方式处理交换：

1. *管道中的前面的处理器调用制作者上的 同步进程 （） 方法来启动同步处理。同步 process （） 方法采用单个交换参数。*
2. *在 process （） 方法的正文中，生成者将请求（消息）发送到端点。*
3. *如果交换模式需要，生成者会等待回复(Out 消息)到达端点。此步骤可能会导致 process （） 方法无限期阻止。但是，如果交换模式不强制回复，则 process （） 方法可以在发送请求后立即返回。*
4. *当 process （） 方法返回时，交换对象包含来自同步调用的回复(Out 消息消息)。*

异步制作者

图 38.7 “异步生成” 显示了异步制作者的概要，其中生成者在子线程中处理交换，并且前面的处理器不会因任何显著时长而被阻止。

图 38.7. 异步生成



异步制作者按如下方式处理交换：

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的返回部分的交换。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：*

   - *交换对象*
   - *同步回调对象*

3. *在 process （） 方法的正文中，生成者会创建一个可封装处理代码的 可运行 对象。然后，生产者将此 可运行 对象的执行委派给子线程。*

4. 异步 process （） 方法返回，从而释放处理器的线程。交换处理将继续在不同的子线程中。

5. Runnable 对象将 In 消息发送到端点。

6. 如果交换模式需要，Runnable 对象会等待回复(Out 或 Fault 消息)到达端点。Runnable 对象 会被阻断，直到收到回复为止。

7. 回复到达后，Run nable 对象会将回复(Out 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

本节简要概述了实施自定义 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，还需要实现以下 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 HEKETI-wagonA 组件对象充当端点工厂。您可以扩展 DefaultComponent 类，并实施 createEndpoint （） 方法。*

   请参阅 第 39 章 组件接口。

2. *实施 Endpoint 接口 swig-wagonAn 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用户是否遵循 事件驱动的 模式、调度的轮询 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，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 哪种模式（事件驱动、调度轮询或轮询）实施消费者，实施消费者采用多种不同方法。消费者实施对于确定用于处理消息交换的线程模型非常重要。

   请参阅 第 41.2 节 “实施 Consumer 接口”。

4. 实施 Producer 接口 HEKETI-5-4To 实施生成者，您可以扩展 DefaultProducer 类并实施 process （） 方法。

   请参阅 第 42 章 生成者接口。

5. （可选）实现 Exchange 或 Message 接口 to the default 实现 Exchange 和 Message 可以直接使用，但偶尔您可能会发现需要自定义这些类型。

   请参阅 第 43 章 交换接口 和 第 44 章 邮件接口。

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 wagonThe CamelContext.addComponent （） 方法中，以编程方式添加组件。

- 使用 Spring configuration swig-wagonThe standard Spring bean 元素添加 组件会创建一个组件实例。bean 的 id 属性隐式定义组件前缀。详情请查看 第 38.3.2 节 “配置组件”。

- 配置 Apache Camel 以自动发现组件 HEKETI -wagonAuto-discovery，确保 Apache Camel 按需自动加载组件。详情请查看 第 38.3.1 节 “设置自动诊断”。

38.3. 自动诊断和配置

38.3.1. 设置自动诊断

概述

自动发现是一种可让您动态地将组件添加到 Apache Camel 应用程序的机制。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 ActiveMQ 端点尚未加载，则 Apache Camel 会搜索 activemq 前缀标识的组件，并动态加载该组件。

组件类可用性

在配置自动发现前，您必须确保您的自定义组件类可从您当前的 classpath 访问。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 classpath 中。

配置自动发现

要启用组件的自动发现，请创建一个名为组件前缀的 Java 属性文件，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名称。您还可以在此文件中定义额外的系统属性设置。

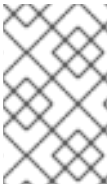
Example

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
```

它包含以下 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 camel-ftp-Version.jar 中定义。

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 会实例化引用类并注入 Spring 配置中指定的属性。

注意

此机制优先于自动发现。如果 CamelContext 找到带有 requisite ID 的 Spring bean，它不会尝试使用自动发现来查找组件。

在组件类中定义 bean 属性

如果要注入组件类的任何属性，请将其定义为 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 property 元素将属性注入组件实例中。例如，上例中的 property 元素将通过在组件上调用 setProperty（） 将值 attribute Value 注入到 property 属性中。

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。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 会自动实例化它在指定的 Java 软件包 org.apache.camel.example.spring 中找到的 RouteBuilder 类。

2

带有 ID 为 jms 的 bean 元素配置 JMS 组件。bean ID 与组件的 URI 前缀对应。例如，如果路由指定了 URI 为 jms://MyQName 的端点，则 Apache Camel 会使用 jms bean 元素中的设置自动加载 JMS 组件。

3

JMS 只是消息传递服务的打包程序。您必须通过在 JmsComponent 类上设置 connectionFactory 属性来指定消息传递系统的 concrete 实现。

4

在本例中，JMS 消息传递服务的具体实施是 Apache ActiveMQ。brokerURL 属性初始化与 ActiveMQ 代理实例的连接，其消息代理嵌入到本地 Java 虚拟机(JVM)中。如果 JVM 中尚未存在代理，ActiveMQ 将使用选项 broker.persistent=false （代理不持久保留消息）和 broker.useJmx=false （代理不打开 JMX 端口）对其进行实例化。

---

[2] 超时间隔通常以毫秒为单位指定。

第 39 章 组件接口

摘要

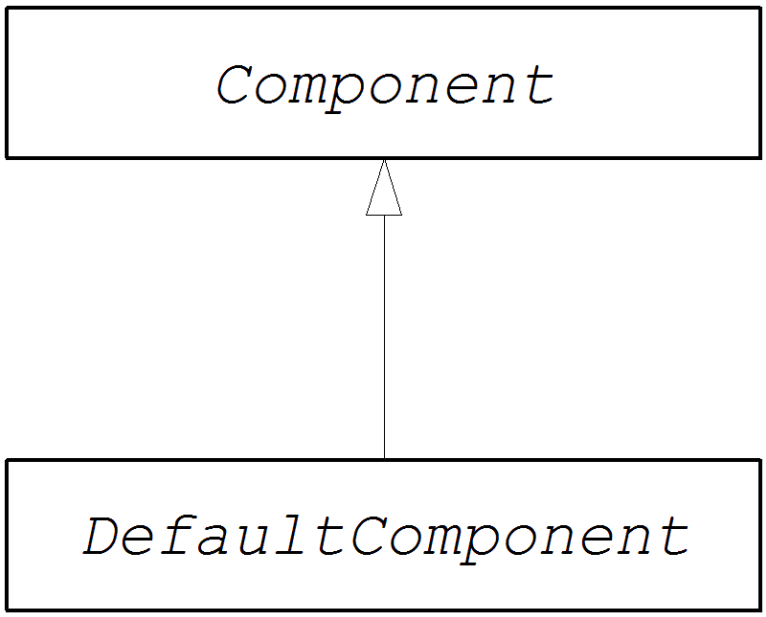
本章论述了如何实现组件接口。

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 指向自定义组件的入口点。也就是说，组件中的所有其他对象最终都可通过 组件实例访问。图 39.1 “组件继承层次结构” 显示组成 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

组件接口定义以下方法：

- getCamelContext （） 并设置CamelContext （） HEKETI-wagonReferences，此组件所属的 CamelContext。当您将组件添加到 CamelContext 时，会自动调用 setCamelContext （） 方法。

- createEndpoint （） wagon-cassandraThe factory 方法，它被调用为这个组件创建 Endpoint 实例。uri 参数是端点 URI，包含创建端点所需的详细信息。

39.2. 实现组件接口

DefaultComponent 类

您可以通过扩展 org.apache.camel.impl.DefaultComponent 类来实施新组件，它为一些方法提供了一些标准功能和默认实现。特别是，DefaultComponent 类提供对 URI 解析和创建 调度的 executor（用于调度的轮询模式）的支持。

URI 解析

基础组件接口中定义的 createEndpoint (String uri) 方法采用完整的未解析端点 URI，作为其仅有参数。另一方面，DefaultComponent 类使用以下签名定义 createEndpoint （） 方法的三参数版本：

```
protected abstract Endpoint createEndpoint(
    String uri,
    String remaining,
    Map parameters
)
throws Exception;
```

URI 是原始未解析的 URI；剩余 是在启动时分离组件前缀后保留的 URI 的一部分，并去除结尾的查询选项，而 参数 包含解析的查询选项。这是从 DefaultComponent 继承时必须覆盖的 createEndpoint （） 方法的版本。它具有已为您解析端点 URI 的优点。

文件 组件的以下端点 URI 示例显示了 URI 解析在实践中的工作方式：

```
file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
```

对于此 URI，以下参数会被传递给 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	**java.util.Map** 中设置了两个条目： • 参数 **delete** 为 boolean **true** • 参数 **moveNamePostfix** 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 moveNamePostfix。所有这些都必须在端点类中定义对应的 bean 方法(getters 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 “消费者参数注入”一节。

禁用端点参数注入

如果没有在 Endpoint 类上定义参数，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 （） 方法并实现它以返回 false，如下所示：

```
protected boolean useIntrospectionOnEndpoint() {
  return false;
}
```

注意

useIntrospectionOnEndpoint （） 方法 不会影响 可能在 Consumer 类上执行的参数注入。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”）。

调度的执行器服务

调度的 executor 在调度的轮询模式中使用，它负责推动消费者端点的定期轮询（调度的 executor 实际上是一个线程池实施）。

要实例化调度的 executor 服务，请使用 CamelContext.get ExecutorServiceStrategy 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 类为创建线程池实例提供了 getExecutorService （） 方法。但是，从 2.3 开始，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，其具有以下签名：

```
protected void validateURI(String uri,
                           String path,
                           Map parameters)
```

```
    throws ResolveEndpointFailedException;
```

如果提供的 URI 没有所需的格式，则 validateURI （） 的实现应该会抛出 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，创建自定义端点类型 CustomEndpoint 的实例。此构造器至少采用原始 URI 字符串 uri 的副本，以及对此组件实例的引用(此)。

Example

例 39.3 “FileComponent Implementation” 显示 FileComponent 类的一个示例实现。

例 39.3. FileComponent Implementation

```
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** 构造器，以便于自动实例化该类。

2

在通过编程创建组件实例时，将父 **CamelContext** 实例用作参数的构造器非常方便。

3

FileComponent.createEndpoint （） 方法的实现遵循 *例 39.2 “createEndpoint （）的实现”* 中描述的模式。实施会创建一个 **FileEndpoint** 对象。

SynchronizationRouteAware Interface

SynchronizationRouteAware 接口允许您在交换路由之前和之后具有回调。

- **onBeforeRoute:** 在给定路由路由交换前。但是，如果您在启动路由后，将 **SynchronizationRouteAware** 实施添加到 **unit OfWork**，则此回调可能无法被调用。

- ***onAfterRoute: 在给定路由路由交换后。但是，如果交换通过多个路由路由，它将为每个路由生成调用回。***

  此调用在这些回调之前发生：

  a. ***路由的使用者将任何响应写入调用者（如果在 InOut 模式中）***

  b. ***UnitOfWork 通过调用 Synchronization.onComplete (org.apache.camel.Exchange) 或 Synchronization.onFailure (org.apache.camel.Exchange)来实现。***

第 40 章 端点接口

摘要

本章论述了如何实现 Endpoint 接口，这是 Apache Camel 组件的实现中的一个重要步骤。

40.1. 端点接口

概述

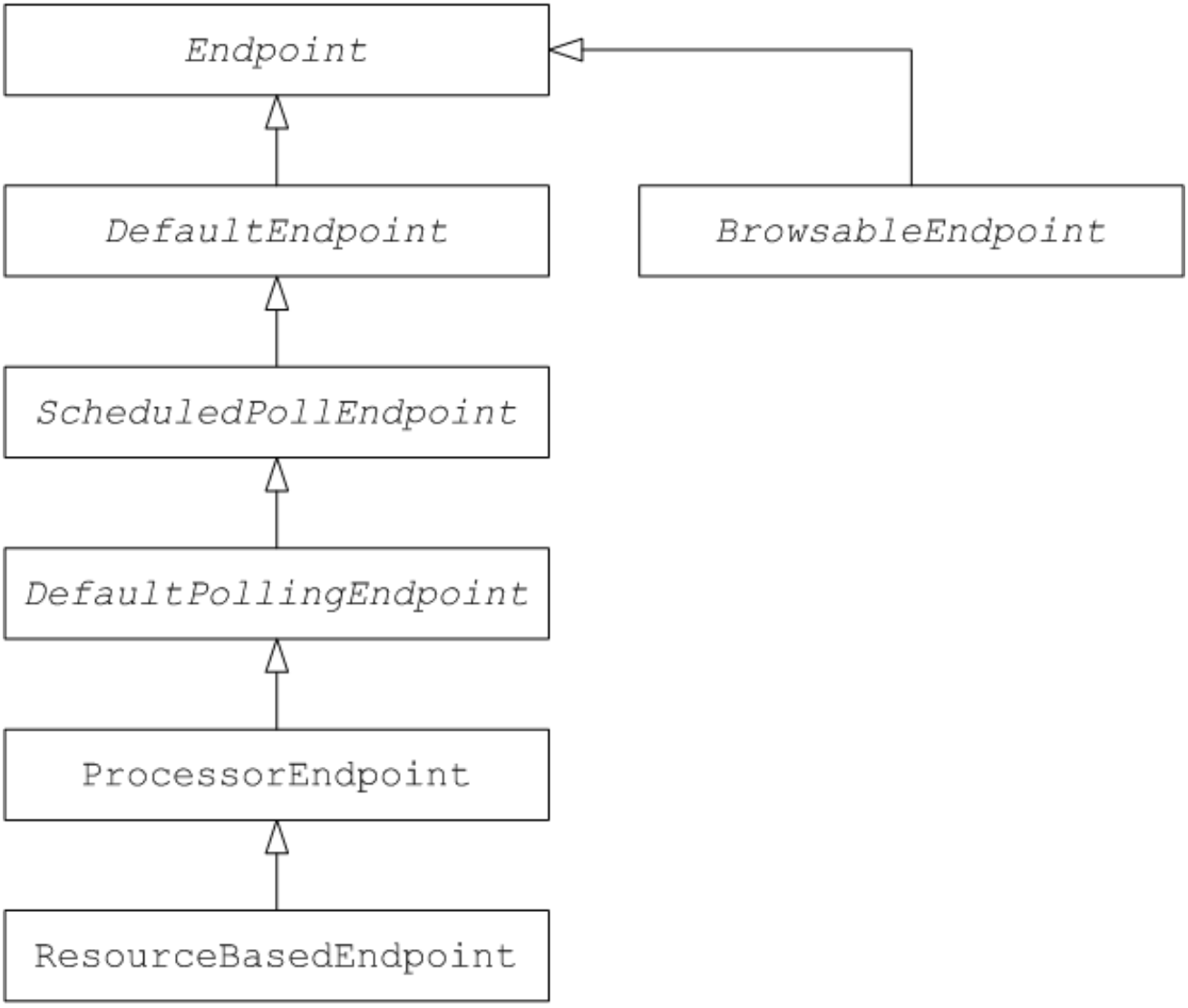
org.apache.camel.Endpoint 类型的实例封装端点 URI，它也充当 消费者、Producer 和 Exchange 对象的工厂。实施端点的方法有三种：

- Event-driven
- 调度的轮询
- 轮询

这些端点实施模式可以补充用于实施 consumer-3 的相应模式，请参阅 第 41.2 节 “实施 Consumer 接口”。

图 40.1 “端点继承层次结构” 显示组成 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 接口定义以下方法：

- *如果希望确保每个 URI 映射到 CamelContext 中的单个端点，则 isSingleton （） isSingleton （） wagon-wagonReturns true。当此属性为 true 时，路由内同一 URI 的多个引用始终引用 单个 端点实例。当此属性为 false 时，在另一方面，对路由内同一 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，都会创建一个新端点实例。*
- *getEndpointUri （） swig-wagonReturns 此端点的端点 URI。*
- *在注册端点时，getEndpointKey （） HEKETI-wagonUsed by org.apache.camel.spi.LifecycleStrategy。*
- *getCamelContext （） HEKETI-wagonreturn 对此端点所属的 CamelContext 实例的引用。*
- *设置CamelContext （） HEKETI-wagonSets，此端点所属的 CamelContext 实例。*
- *配置Properties （） HEKETIStores 是一个参数映射的副本，该映射用于在创建新 Consumer 实例时注入参数。*
- *isLenientProperties （） InventoryService-wagonReturns true 以指示允许 URI 包含未知参数（即，无法在 Endpoint 或 Consumer 类上注入的参数）。通常，应实施此方法，以返回 false。*

- 带有以下变体的 createExchange （） swig-wagonAn 超载方法：
  - Exchange createExchange （） InventoryService-wagonCreates 是一个具有默认交换模式设置的新交换实例。
  - Exchange createExchange (ExchangePattern pattern) HEKETI-wagonCreates 是一个具有指定交换模式的新交换实例。
  - Exchange createExchange (Exchange Exchange) HEKETI-wagonConverts the given Exchange 参数到此端点所需的交换类型。如果给定的交换不是正确的类型，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 （） HEKETIFactory 方法，用于创建新的 Producer 实例。
- createConsumer （） swig- swigFactory 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。
- createPollingConsumer （） HEKETI-wagonFactory 方法，以创建新的轮询消费者实例。

端点单例

为避免不必要的开销，最好为所有具有相同 URI （带有 CamelContext）的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 意味着两个 URI 在使用字符串相等时是相同的。在原则中，可以具有两个等效的 URI，但由不同的字符串表示。在这种情况下，URIs 不会被视为相同。

40.2. 实施端点接口

实施端点的替代方法

支持以下替代端点实现模式：

- *事件驱动的端点实现*
- *调度的轮询端点实现*
- *轮询端点实施*

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”），请通过扩展抽象类 org.apache.camel.impl.DefaultEndpoint 来实现，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，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，它采用端点 URI、endpointUri，以及父组件引用，组件 作为参数。

3

实施 createProducer （） 工厂方法来创建制作者端点。

4

实施 createConsumer （） factory 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。默认情况下，从 DefaultEndpoint 继承的实现会创建一个 DefaultExchange 对象，该对象可用于任何 Apache Camel 组件。如果您需要在 DefaultExchange 对象中初始化一些交换属性，请在此处覆盖 createExchange （） 方法，以添加交换属性设置。

重要

不要覆盖 createPollingConsumer （） 方法。

DefaultEndpoint 类提供以下方法的默认实现，您可能会在编写自定义端点代码时很有用：

- getEndpointUri （） wagon-wagonReturns 端点 URI。

- getCamelContext （） HEKETI-wagonReturns 对 CamelContext 的引用。

- getComponent （） wagon- swigReturns 对父组件的引用。

- createPollingConsumer （） HEKETI-wagonCreates a polling consumer。创建的轮询消费者的功能基于事件驱动的消费者。如果您覆盖事件驱动的消费者方法 createConsumer （），您将获得轮询消费者实施。

- createExchange (Exchange e) HEKETI-wagonConverts the given Exchange 对象 e，将这个端点所需的类型改为此端点所需的类型。此方法使用覆盖的 createExchange （） 端点创建新端点。这可确保该方法也适用于自定义交换类型。

调度的轮询端点实现

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”），则通过从抽象类 org.apache.camel.impl.ScheduledPollEndpoint 继承来实现它，如 例 40.3 “ScheduledPollEndpoint 实现” 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，实施计划轮询自定义端点 CustomEndpoint。

2

您必须至少有一个构造器，其使用端点 URI、endpointUri 和父组件引用，组件 作为参数。

3

实施 createProducer （） 工厂方法来创建制作者端点。

4

实施 createConsumer （） factory 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 （） 方法负责将消费者查询选项注入消费者。请参阅 “消费者参数注入”一节。

6

通常，不需要 覆盖 createExchange （） 方法。默认情况下，从 DefaultEndpoint 继承的实现会创建一个 DefaultExchange 对象，该对象可用于任何 Apache Camel 组件。如果您需要在 DefaultExchange 对象中初始化一些交换属性，请在此处覆盖 createExchange （） 方法，以添加交换属性设置。

重要

不要覆盖 createPollingConsumer （） 方法。

轮询端点实施

如果您的自定义端点符合轮询消费者模式（请参阅 第 38.1.3 节 “消费者模式和线程”），则通过继承抽象类 org.apache.camel.impl.DefaultPollingEndpoint 来实现，如 例 40.4 “DefaultPollingEndpoint 实现” 所示。

例 40.4. DefaultPollingEndpoint 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.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。有关如何实现轮询消费者的详情，请参考 “轮询消费者实施”一节。

除了 createPollingConsumer （） 方法的实现外，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，您可以实施 org.apache.camel.spi.BrowsableEndpoint 接口，如 例 40.5 “BrowsableEndpoint Interface” 所示。

如果端点执行某种类型的传入事件，则需要实施此接口。例如，Apache Camel SEDA 端点实现了 BrowsableEndpoint 接口都为 BrowsableEndpoint，请参阅 例 40.6 “SedaEndpoint 实现”。

例 40.5. BrowsableEndpoint Interface

```
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();
}
```

Example

例 40.6 “SedaEndpoint 实现” 显示 SedaEndpoint 的示例实施。SEDA 端点是 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)，SEDA 使用者启动一个线程来读取和处理事件。事件本身由 org.apache.camel.Exchange 对象表示。

例 40.6. SedaEndpoint 实现

```
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 类来实现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

根据事件驱动的消费者的常规模式，SedaEndpoint 定义了一个构造器，它采用端点参数、endpointUri 和组件引用参数 component。

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 （） 工厂方法创建 CollectionProducer 实例，这是将事件添加到队列中的制作者实施。

5

createConsumer （） factory 方法创建一个 SedaConsumer 实例，它负责拉取队列的事件并处理它们。

6

getQueue （） 方法返回对队列的引用。

7

isSingleton （） 方法返回 true，这表示应当为每个唯一 URI 字符串创建一个端点实例。

8

getExchanges （） 方法实现来自 BrowsableEndpoint 的对应抽象方法。

第 41 章 消费者接口

摘要

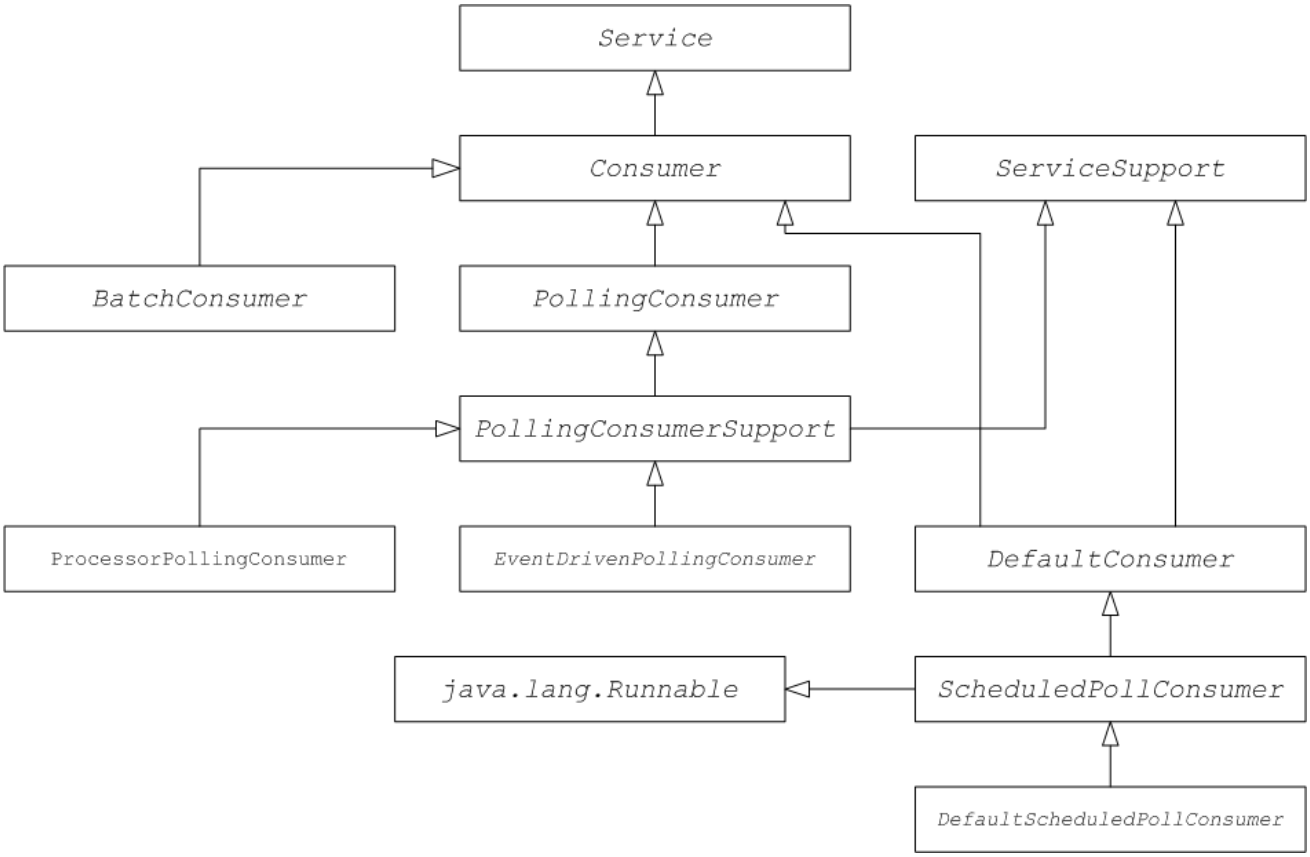
本章论述了如何实施 Consumer 接口，这是 Apache Camel 组件的实现中的一个重要步骤。

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。实现消费者的方法有几种（请参阅 第 38.1.3 节 “消费者模式和线程”），这种灵活性在继承层次结构中反映在继承层次结构中（请参阅 图 41.1 “消费者继承层次结构”），其中包括几个用于实现消费者的不同基础类。

图 41.1. 消费者继承层次结构



消费者参数注入

对于遵循调度轮询模式的消费者（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，对于由 自定义前缀 标识的组件，请考虑以下端点 URI：

```
custom:destination?consumer.myConsumerParam
```

Apache Camel 支持自动注入 consumer.* 格式的查询选项。对于 consumer.myConsumerParam 参数，您需要在 Consumer 实现类中定义对应的 setter 和 getter 方法，如下所示：

```
public class CustomConsumer extends ScheduledPollConsumer {
    ...
    String getMyConsumerParam() { ... }
    void setMyConsumerParam(String s) { ... }
    ...
}
```

其中 getter 和 setter 方法遵循常见的 Java Bean 约定（包括大写属性名称的第一个字母）。

除了在 Consumer 实现中定义 bean 方法外，还必须记得在 Endpoint.createConsumer () 实现中调用 configureConsumer () 方法（请参阅 "调度的轮询端点实现"一节）。

例 41.1 "FileEndpoint createConsumer () Implementation" 显示了一个 createConsumer () 方法实现的示例，它取自文件组件的 FileEndpoint 类：

例 41.1. FileEndpoint createConsumer () Implementation

```
...
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，消费者参数注入的工作方式如下：

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 () 来将其存储在端点实例中。

2. 当调用 createConsumer () 时，方法实现调用 configureConsumer () 来注入消费者参数（请参阅 例 41.1 "FileEndpoint createConsumer () Implementation"）。

3. *configureConsumer （） 方法使用 Java 反映来调用在 consumer. 前缀被剥离后名称与相关选项匹配的 setter 方法。*

调度的轮询参数

遵循调度的轮询模式的消费者会自动支持 表 41.1 “调度的 Poll 参数” 中显示的消费者参数（可在端点 URI 中显示为查询选项）。

表 41.1. 调度的 Poll 参数

Name	default	描述
initialDelay	**1000**	第一次轮询前的延迟（以毫秒为单位）。
delay	**500**	取决于 **useFixedDelay** 标记的值（时间单位为毫秒）。
useFixedDelay	**false**	如果为 **false**，则 **delay** 参数将解释为轮询周期。轮询将在 initialDelay、 **initialDelay +delay**、**initialDelay+2*delay** 等发生。 如果为 **true**，则 **delay** 参数将解释为上一个执行和下一次执行之间经过的时间。轮询将在 initialDelay、 **initialDelay + [*ProcessingTime*]+delay** 等发生。其中 *ProcessingTime* 是处理当前线程中交换对象所需的时间。

在事件驱动和轮询用户间转换

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。提供了以下转换类：

- *org.apache.camel.impl.EventDrivenPollingConsumer swig-wagonConver 将事件驱动的消费者转换为轮询消费者实例。*
- *org.apache.camel.impl.DefaultScheduledPollConsumer swig-wagonConverts 将轮询消费者转换成事件驱动的消费者实例。*

实际上，这些类用于简化实施 Endpoint 类型的任务。Endpoint 接口定义了创建消费者实例的以下两种方法：

```
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，并且 createPollingConsumer （） 返回轮询消费者。您仅实施这些方法。例如，如果您遵循消费者的事件驱动模式，您将实施 createConsumer （） 方法，以提供仅引发异常的 createPollingConsumer （） 的方法实现。但是，在转换类帮助下，Apache Camel 能够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 来实现端点，并实施 createConsumer （） 方法。createPollingConsumer （） 的实现继承自 DefaultEndpoint，其定义如下：

```
public PollingConsumer<E> createPollingConsumer() throws Exception {
    return new EventDrivenPollingConsumer<E>(this);
}
```

EventDrivenPollingConsumer 构造器会引用事件驱动的消费者，这会 有效地将其嵌套并转换为轮询消费者。要实现转换，EventDrivenPollingConsumer 实例缓冲传入的事件，并通过 receive （）、接收 （长超时）和 receive NoWait （） 方法按需提供它们。

类似地，如果您根据轮询模式实施消费者，您可以通过扩展 DefaultPollingEndpoint 并实施 createPollingConsumer （） 方法来实现端点。在本例中，createConsumer （） 方法的实现继承自 DefaultPollingEndpoint，默认的实现会返回 DefaultScheduledPollConsumer 实例（它将轮询消费者转换为事件驱动的消费者）。

ShutdownPrepared interface

消费者类可以选择实施 org.apache.camel.spi.ShutdownPrepared 接口，它允许自定义消费者端点接收关闭通知。

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 接口定义以下方法：

prepareShutdown

接收通知以在一个或多个阶段关闭消费者端点，如下所示：

a. **在强制参数的位置，恰当的关闭是，其值为 false。尝试安全地清理资源。例如，通过安全停止线程。**

b. **强制关闭 HEKETI-wagon，其中强制参数的值为 true。这意味着关闭已超时，因此您必须积极清理资源。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，后者与安全关闭机制交互，使消费者能够请求额外的时间关闭。这通常需要 SEDA 等组件，这些组件可能将待处理的交换存储在内部队列中。通常，您要在关闭 SEDA 使用者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

从此方法返回 true，如果要延迟消费者的关闭。shutdownRunningTask 参数是一个 enum，它可以采用以下值之一：

- ShutdownRunningTask.CompleteCurrentTaskOnly HEKETI-wagonfinish 处理当前由消费者线程池处理的交换，但不要尝试处理比此更多的交换。
- ShutdownRunningTask.CompleteAllTasks HEKETI-wagonprocess all of the pending Exchanges。例如，如果是 SEDA 组件，使用者将处理来自其传入队列的所有交换。

getPendingExchangesSize

表示消费者仍要处理的交换数。零值表示完成处理，并且可以关闭消费者。

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. 实施 CONSUMER 接口

实施消费者的替代方法

您可以使用以下方法之一实现消费者：

- 事件驱动的消费者实施
- 调度的轮询消费者实现
- 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，处理由外部事件显式驱动。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 实现” 显示 JMXConsumer 类的实施，该类取自 Apache Camel JMX 组件实施。JMXConsumer 类是一个事件驱动的消费者的示例，它通过从 org.apache.camel.impl.DefaultConsumer 类继承来实现。对于 JMXConsumer 示例，事件由 NotificationListener.handleNotification （） 方法上的调用来表示，这是接收 JMX 事件的标准方法。要接收这些 JMX 事件，需要实现 NotificationListener 接口并覆盖 handleNotification （） 方法，如 例 41.4 “JMXConsumer 实现” 所示。

例 41.4. JMXConsumer 实现

```
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，遵循事件驱动的消费者的常规模式。此外，由于此使用者设计为从 JMX 通知（由 JMX 通知表示）接收事件，因此需要实施 NotificationListener 接口。

2

您必须实施对父端点、端点 以及对链中下一个处理器的引用，作为参数的一个构造器。

3

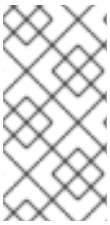
当 JMX 通知到达时，handleNotification （） 方法（在 NotificationListener中定义） 会自动由 JMX 调用。此方法的正文应包含执行消费者事件处理的代码。由于 handleNotification （） 调用源自 JMX 层，因此消费者的线程模型由 JMX 层隐式控制，而不是 JMXConsumer 类。

4

这行代码组合了两个步骤。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，新创建的交换对象传递到路由中的下一个处理器（以同步方式）。

5

handleException （） 方法由 DefaultConsumer 基本类实现。默认情况下，它使用 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，您必须识别类似事件监听程序方法，以便在自定义消费者中实施。

调度的轮询消费者实现

在调度的轮询消费者中，轮询事件由计时器类 java.util.concurrent.ScheduledExecutorService 自动生成。要接收生成的轮询事件，您必须实现 ScheduledPollConsumer.poll （） 方法（请参阅 第 38.1.3 节 “消费者模式和线程”）。

例 41.5 “ScheduledPollConsumer Implementation” 演示了如何实现遵循调度的轮询模式的消费者，该模式通过扩展 ScheduledPollConsumer 类来实现。

例 41.5. ScheduledPollConsumer Implementation

```
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，实施调度的轮询消费者类 CustomConsumer。***

(2) ***您必须实施对父端点、端点 以及对链中下一个处理器的引用，作为参数的一个构造器。***

(3) ***覆盖 poll （） 方法，以接收调度的轮询事件。这是您应该放置检索和处理传入事件（由交换对象代表）的代码。***

4

5

*（可选） 如果您希望某些行代码在启动使用者时执行，请覆盖 **doStart** （） 方法，如下所示。*

6

*（可选） 如果您希望某些代码行执行，因为消费者正在停止，请覆盖 **doStop** （） 方法，如下所示。*

轮询消费者实施

*例 41.6 “PollingConsumerSupport Implementation” 概述了如何通过扩展 **PollingConsumerSupport** 类来实现一个遵循轮询模式的消费者。*

例 41.6. PollingConsumerSupport Implementation

```
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，实施您的轮询消费者类 CustomConsumer。

2

您必须至少实施对父端点端点端点 的引用的一个构造器，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。如果事件仍不可用，此方法可以无限期地阻止。

5

receive （长超时） 方法实施一种算法，只要指定的超时（通常以毫秒为单位）指定。

6

如果要在消费者启动或关机期间插入执行的代码，请分别实施 doStart （） 方法和 doStop （） 方法。

自定义线程实现

如果标准的消费者模式不适用于您的消费者实施，您可以直接实施 Consumer 接口，并自行编写线程代码。当编写线程代码时，您必须遵守标准的 Apache Camel 线程模型，如 第 2.8 节 “线程模型” 所述。

例如，camel-core 中的 SEDA 组件实施自己的消费者线程，这与 Apache Camel 线程模型一致。例 41.7 “自定义线程实施” 显示 SedaConsumer 类如何实现其线程的概要。

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 类来实现，并实施 Consumer、Runnable、和 ShutdownAware 接口。

2

实施 Runnable.run () 方法，以定义在线程中运行的使用者的作用。在这种情况下，消费者在循环中运行，轮询新交换的队列，然后在队列的后方部分处理交换。

3

doStart () 方法继承自 ServiceSupport。您可以覆盖此方法，以定义消费者启动时的作用。

4

您应该使用通过 CamelContext 注册的 ExecutorServiceStrategy 对象创建一个线程池，而不是直接创建线程。这很重要，因为它可让 Apache Camel 实现线程的集中管理和支持，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 poolSize 时间来启动线程。

6

doStop （） 方法继承自 ServiceSupport。您可以覆盖此方法，以定义消费者在关闭时的作用。

7

关闭线程池，由 executor 实例表示。

第 42 章 生成者接口

摘要

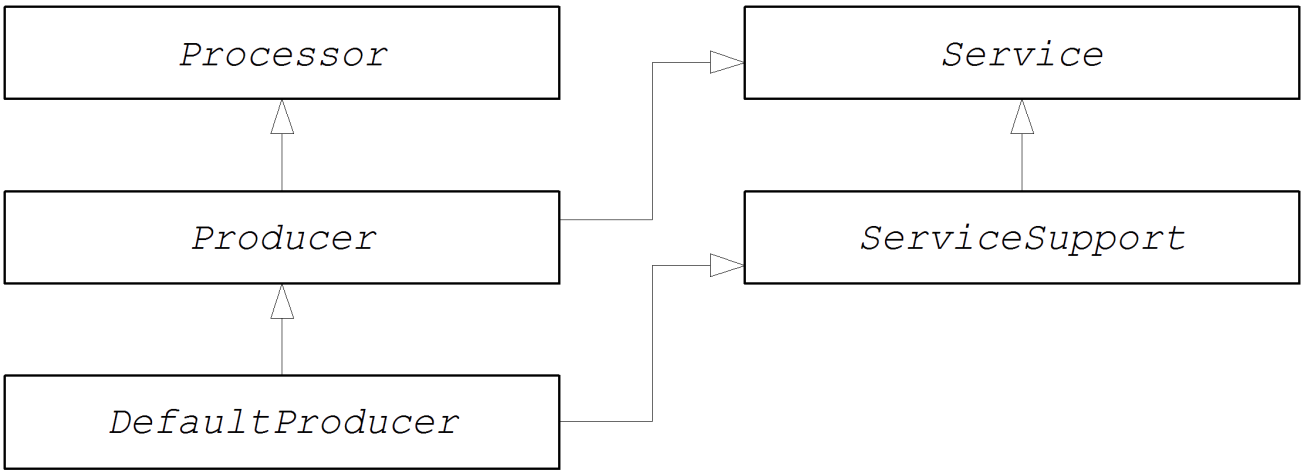
本章论述了如何实施 Producer 接口，这是 Apache Camel 组件的实现中的一个重要步骤。

42.1. PRODUCER 接口

概述

org.apache.camel.Producer 类型的实例代表路由中的目标端点。生产者的角色是将请求（消息）发送到特定的物理端点，并接收对应的响应(Out 或 Fault 消息)。Producer 对象基本上是一个特殊的 Processor，它出现在处理器链的末尾（等同于路由）。图 42.1 “生成者继承层次结构” 显示制作者的继承层次结构。

图 42.1. 生成者继承层次结构



Producer 接口

例 42.1 “生成者接口” 显示 org.apache.camel.Producer 接口的定义。

例 42.1. 生成者接口

```
package org.apache.camel;

public interface Producer extends Processor, Service, IsSingleton {

    Endpoint<E> getEndpoint();

    Exchange createExchange();

    Exchange createExchange(ExchangePattern pattern);
```

```
    Exchange createExchange(E exchange);
}
```

生成者方法

Producer 接口定义以下方法：

- process （）（从 Processorherited from Processor） HEKETI-cassandraThe most important method.生产者基本上是一种特殊的处理器类型，它向端点发送请求，而不是将交换对象转发到另一个处理器。通过覆盖 process （） 方法，您可以定义生成者如何向相关端点发送和接收消息。

- getEndpoint （） wagon- swigReturns 对父端点实例的引用。

- createExchange （） HEKETI-wagonThese 超载方法与 Endpoint 接口中定义的对应方法类似。通常，这些方法委派到父 Endpoint 实例上定义的对应方法（默认为 DefaultEndpoint 类）。有时，您可能需要覆盖这些方法。

异步处理

在生成生成器的 exchange 对象处理通常涉及向远程目标发送消息，并等待 reply swig-5-4cancan 可能阻断大量时间。如果要避免阻止当前线程，您可以选择将制作者实施为 异步处理器。异步处理模式将前面的处理器与生成者分离，以便 process （） 方法在没有延迟的情况下返回。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。在其自身上，这不足以确保将使用异步处理模型：在链中前面的处理器也必需调用 process （） 方法的异步版本。AsyncProcessor 接口的定义显示在 例 42.2 “AsyncProcessor Interface” 中。

例 42.2. AsyncProcessor Interface

```
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 接口被定义，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，以接收处理完成的通知。AsyncCallback.done （） 方法使用一个布尔值参数，指示处理是同步还是不同步执行。通常，标志为 false，以指示异步处理。然而，在某些情况下，生产者 不能 异步处理（尽管被要求这样做）。例如，如果生成者知道交换的处理将快速完成，则可以通过同步方式优化处理。在这种情况下，doneSynchronously 标志应设置为 true。

ExchangeHelper 类

在实施制作者时，您可能会发现调用 org.apache.camel.util.ExchangeHelper 实用程序类中的一些方法会很有帮助。有关 ExchangeHelper 类的详情，请参考 第 35.4 节 “ExchangeHelper 类”。

42.2. 实施 PRODUCER 接口

实施制作者的替代方法

您可以使用以下方法之一实施制作者：

- 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.DefaultProducer 类，实施自定义同步制作者类 CustomProducer。

(2) 实施对父端点的引用的构造器。

(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 （） 方法通常按如下方式实现：

- 如果交换包含 In 消息，如果这与指定的交换模式一致，则将 In 消息发送到指定的端点。
- 如果交换模式预计 Out 消息的收据，请等待收到 Out 消息。这通常会导致 process （） 方法阻止大量时间。
- 收到回复后，调用 exchange.setOut （） 将回复附加到交换对象。如果回复包含错误消息，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 Implementation” 概述了如何实施异步制作者。在这种情况下，您必须实施同步的 process （） 方法和异步 process （） 方法（采用额外的 AsyncCallback 参数）。

例 42.5. CollectionProducer Implementation

```
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

通过扩展 org.apache.camel.impl.DefaultProducer 类，实施自定义异步制作者类 CustomProducer，并实施 AsyncProcessor 接口。

2

实施对父端点的引用的构造器。

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施异步方法。此处显示的方法是创建一个 java.lang.Runnable 实例，任务 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，以指示交换异步处理。

6

CustomProducerTask 类封装在子线程中运行的处理代码。此类必须将 Exchange 对象、交换和 AsyncCallback 对象回调对象存储为私有成员变量的副本。

7

run （） 方法包含将 In 消息发送到 制作者端点并等待收到回复（若有）的代码。收到回复(Out message 或 Fault 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。

第 43 章 交换接口

摘要

本章论述了 Exchange 接口。由于 Apache Camel 2.0 中执行的 camel-core 模块重构，因此不再需要定义自定义交换类型。DefaultExchange 实现现在可以在所有情况下使用。

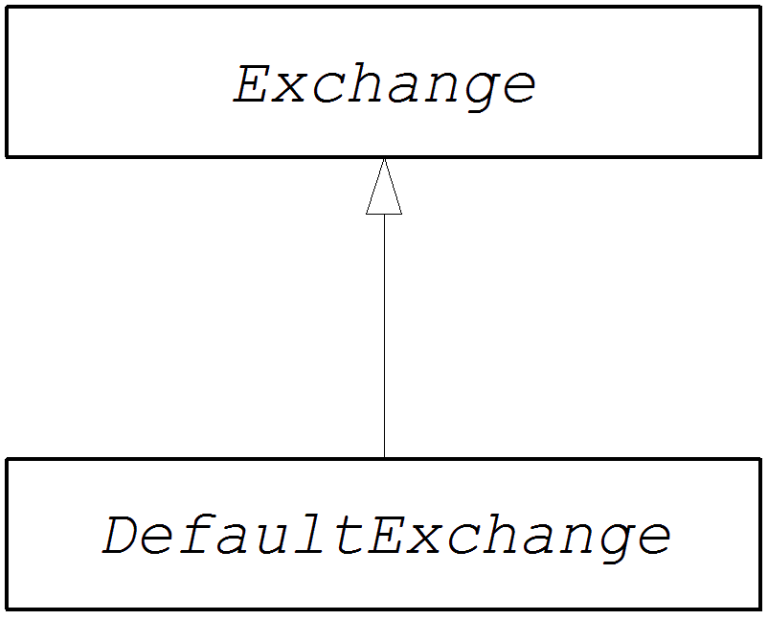
43.1. EXCHANGE INTERFACE

概述

org.apache.camel.Exchange 类型的实例封装通过路由传递的当前消息，其他元数据编码为交换属性。

图 43.1 “Exchange Inheritance Hierarchy” 显示交换类型的继承层次结构。默认实施 DefaultExchange 总是被使用。

图 43.1. Exchange Inheritance Hierarchy



Exchange 接口

例 43.1 “交换接口” 显示 org.apache.camel.Exchange 接口的定义。

例 43.1. 交换接口

```
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 接口定义以下方法：

- getPattern （）, setPattern （） HEKETI-busyboxThe Exchange pattern 可以是 org.apache.camel.ExchangePattern 中枚举的值之一。支持以下交换模式值：
  - InOnly
  - RobustInOnly
  - InOut
  - InOptionalOut
  - OutOnly
  - RobustOutOnly
  - OutIn
  - OutOptionalIn
- setProperty （）, getProperty （）, getProperties （）, removeProperty （）, hasProperties （）属性由您可能需要的其它元数据组成，供您的组件实现。

- *setIn （）, getIn （） wagon-wagonSetter 和 getter 方法，用于 In 消息。*

  DefaultExchange 类提供的 get In （） 实施实施 lazy 创建语义：如果在调用 getIn （） 时 In 消息为 null，则 DefaultExchange 类会创建一个默认的 In 消息。

- *setOut （）, getOut （）, hasOut （）, hasOut （）5-4Setter 和 getter 方法用于 Out 消息。*

  getOut （） 方法隐式支持 lazy 创建 Out 消息。也就是说，如果当前 Out 消息是 null，则会自动创建新的消息实例。

- *setException （）, getException （） wagon-wagonGetter 和 setter 方法用于例外对象（格式为 Throwable type）。*

- *isFailed （） HEKETI-wagonReturns true （如果交换因为异常或由于错误而失败）。*

- *如果交换被转换，则 isTransacted （） wagon-wagonReturns true。*

- *如果交换标记为回滚，则 isRollback （） HEKETI-wagonReturns true。*

- *Getcontext （） wagon- swigReturns 对关联的 CamelContext 实例的引用。*

- *copy （） HEKETICreates 一个新的与当前自定义交换对象相同的新副本（交换 ID 除外）副本。此操作也会复制 In 消息正文和标头，out 消息（若有）和 Fault 消息（若有）。*

- *setFromEndpoint （）, getFromEndpoint （） HEKETI-wagonGetter 和 setter 方法，用于组织此消息的消费者端点（通常是在路由开始时出现在 from （） DSL 命令中的端点）。*

- *setFromRouteId （）, getFromRouteId （） 5-4-wagonGetters 和 setters 用于源自此交换的路由 ID。getFromRouteId （） 方法应该只在内部调用。*

- *setUnitOfWork （）, getUnitOfWork （） wagon-wagonGetter 和 setter 方法，用于 org.apache.camel.spi.UnitOfWork bean 属性。此属性只适用于可以参与事务的交换。*

- *setExchangeId （）, getExchangeId （） HEKETI-wagonGetter 和 setter 方法用于交换 ID。自定义组件是否使用交换 ID 是一个实施详情。*

- *addOnCompletion （） HEKETIAdds 是一个 org.apache.camel.spi.Synchronization 回调对象，它会在处理交换完成时调用。*

- *将所有 OnCompletion 回调对象到指定交换对象的 handoverCompletions （） HEKETI-wagonHands。*

第 44 章 邮件接口

摘要

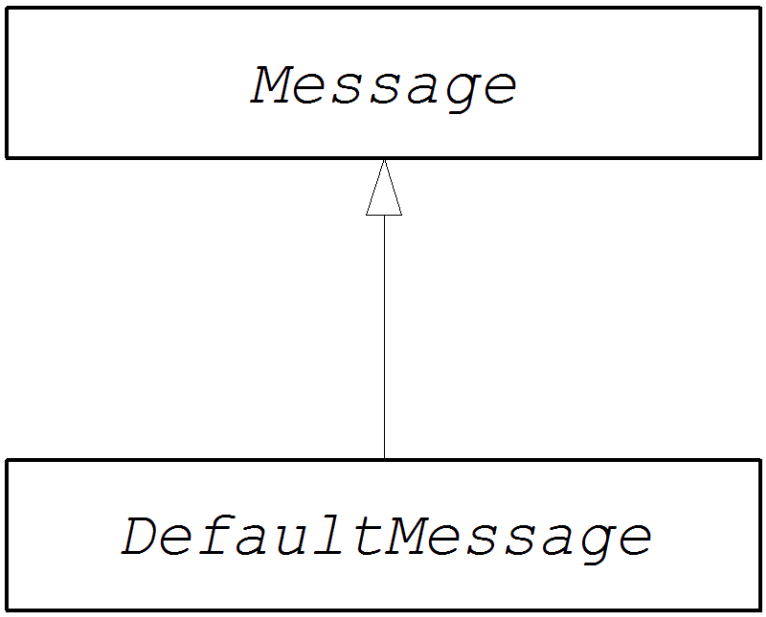
本章论述了如何实现 Message 接口，这是 Apache Camel 组件的实现中的可选步骤。

44.1. 消息接口

概述

org.apache.camel.Message 类型的实例可以代表任何类型的消息(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，默认的实施 DefaultMessage 足够了。

图 44.1. 消息继承层次结构



Message 接口

例 44.1 “邮件接口” 显示 org.apache.camel.Message 接口的定义。

例 44.1. 邮件接口

```
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 接口定义以下方法：

- ***setMessageId （）, getMessageId （） HEKETI-wagonGetter 和 setter 方法用于消息 ID。您在自定义组件中使用消息 ID 是否是实施详情。***

- getExchange （） HEKETI-wagonReturns 对父交换对象的引用。

- isFault （）, setFault （） HEKETI-wagonGetter 和 setter 方法用于 fault 标记，这表明此消息是否为 fault 消息。

- getHeader （）, getHeaders （） , setHeader （）, setHeaders （）, removeHeader （）, hasHeaders （） and setter 方法用于消息标头。通常，这些消息标头可用于存储实际标头数据，或者存储各种元数据。

- getBody （）, getMandatoryBody （）, setBody （） wagon-wagonGetter 和 setter 方法用于消息正文。getMandatoryBody （）访问程序保证返回的正文不是null，否则会抛出 InvalidPayloadException 异常。

- getattachment （）, getAttachments （）, getAttachmentNames （）, removeAttachment （）, addAttachment （）, setAttachments （）, hasAttachments （）

- copy （） HEKETICreates 是一个当前自定义消息对象相同的新副本（包括消息 ID）副本。

- 将指定通用消息对象 message 消息 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，否则返回 null，则 createExchangeId （） swig-wagonReturns 是此交换的唯一 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” 通过扩展 DefaultMessage 类，概述了如何实施消息。

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，实施自定义消息类 **CustomMessage**。

(2) 通常，您需要一个默认的构造器，它创建一个带有默认属性的消息。

(3) 覆盖 **toString** () 方法以自定义消息字符串。

(4) **newInstance** () 方法从 **MessageSupport.copy** () 方法调用。自定义 **newInstance** () 方法应侧重于将当前消息实例的所有自定义属性复制到新的消息实例中。**MessageSupport.copy** () 方法通过调用 **copyFrom** () 来复制通用消息属性。

(5)

6

populateInitialHeaders （） 方法与标头 getter 和 setter 方法一起工作，以实施对消息标头的 lazy 访问。此方法解析消息，以提取任何消息标头并将其插入到散列 映射中。当用户第一次尝试访问标头（或标头）时，会自动调用 populateInitialHeaders （） 方法（通过调用 getHeader （）、getHeaders （）、setHeader （） 或 setHeaders （））。

7

populateInitialAttachments （） 方法与附件 getter 和 setter 方法一起工作，以实施对附件的 lazy 访问。此方法提取消息附件，并将它们插入到散列映射(映射)。当用户试图通过调用 getAttachment （）、getAttachments （）、getAttachment （） 或 addAttachment （） 第一次尝试访问附件（或附件）时，会自动调用 populateInitialAttachments （） 方法。

部分 IV. API 组件框架

如何创建使用 API 组件框架嵌套任何 Java API 的 Camel 组件。

第 45 章 API 组件框架简介

摘要

API 组件框架可帮助您面临基于大型 Java API 实施复杂 Camel 组件的问题。

45.1. 什么是 API 组件框架？

动机

对于带有少量选项的组件，实施组件的标准方法(第 38 章 实施组件)非常有效。然而，它开始会变得有问题，就是当您需要实施带有大量选项的组件时。当涉及企业级组件时，此问题会变得显著，这需要您包装包含 数百个 操作的 API。此类组件需要大量努力来创建和维护。

API 组件框架被精确开发，以应对实施此类组件的问题。

将 API 转换成组件

基于 Java API 实施 Camel 组件的经验表明，很多工作是日常工作和机械。它包括采用特定的 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的明显候选者。

通用 URI 格式

自动化实施 Java API 的第一步是设计将 API 方法映射到 URI 的标准方法。为此，我们需要定义一个通用 URI 格式，可用于嵌套 任何 Java API。因此，API 组件框架为端点 URI 定义以下语法：

```
scheme://endpoint-prefix/endpoint?Option1=Value1&...&OptionN=ValueN
```

其中 scheme 是组件定义的默认 URI 方案； endpoint-prefix 是简短的 API 名称，它映射到来自嵌套的 Java API 的类或接口之一； 端点 映射到方法名称；以及 URI 选项映射到方法参数名称。

单个 API 类的 URI 格式

如果 API 仅包含单个 Java 类，则 URI 的 endpoint-prefix 部分将变为冗余，您可以使用以下更短格式指定 URI：

```
scheme://endpoint?Option1=Value1&...&OptionN=ValueN
```

■

注意

要启用此 URI 格式，组件实施者还需要在 API 组件 Maven 插件配置中将 apiName 元素留空。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，很明显需要某种形式的反映机制。但是，标准的 Java 反映 API 遭受显著限制：它不会保留方法参数名称。这是一个问题，因为我们需要方法参数名称来生成有意义的 URI 选项名称。解决方案是以替代格式提供元数据：作为 Javadoc 或方法签名文件。

javadoc

javadoc 是 API 组件框架的理想元数据形式，因为它保留了完整的方法签名，包括方法参数名称。也可以轻松生成（特别是使用 maven-javadoc-plugin），并且在很多情况下，第三方库已经提供了。

方法签名文件

如果出于某种原因 Javadoc 不可用或不适合，API 组件框架还支持替代的元数据来源：方法签名文件。签名文件是一个简单的文本文件，它由 Java 方法签名列表组成。通过从 Java 代码复制和粘贴（轻松编辑生成的文件）来手动创建这些文件相对容易。

框架包括什么？

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

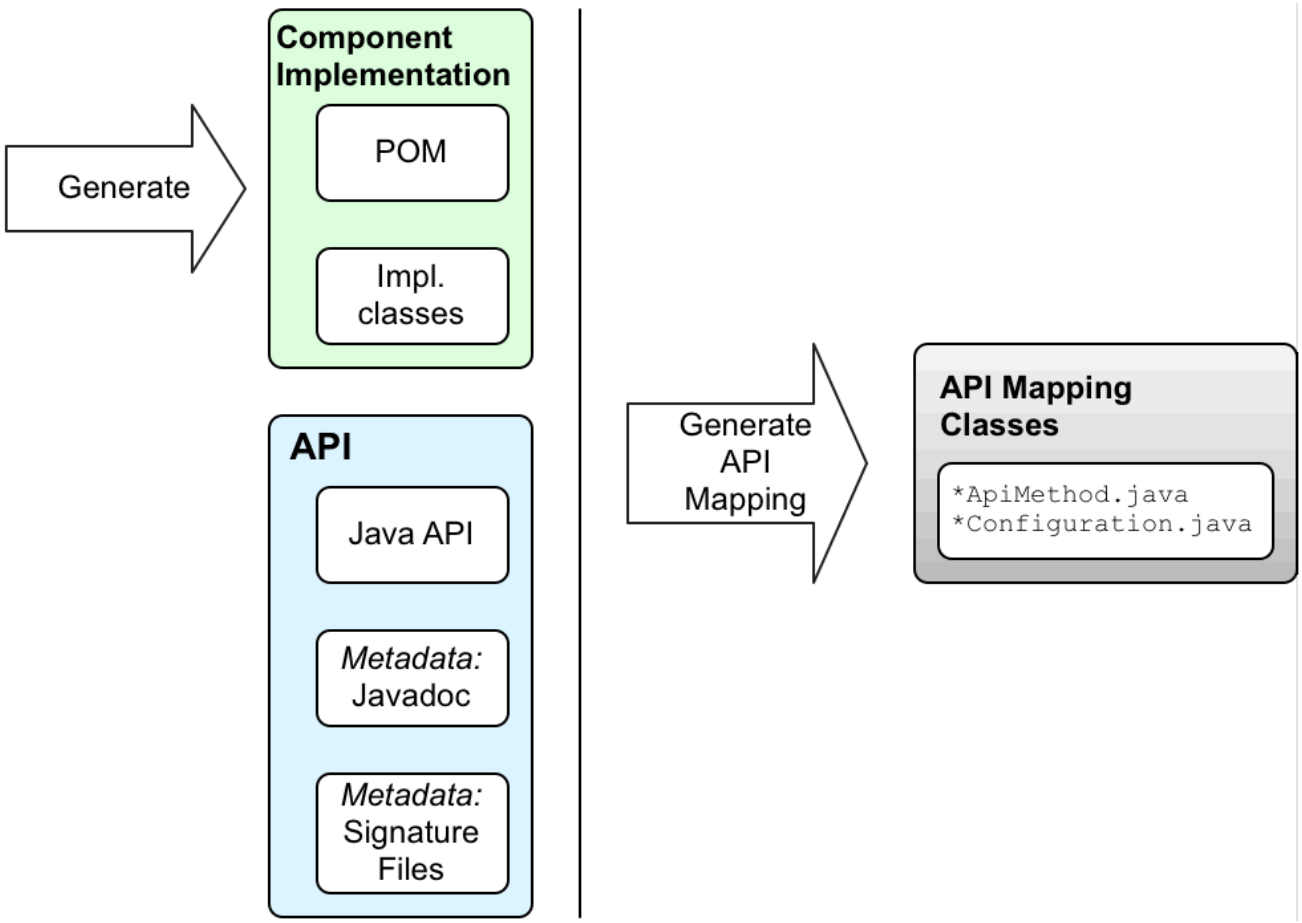
为了支持 API 组件框架的编程模型，Apache Camel 内核在 org.apache.camel.util.component 软件包中提供了一个专用的 API。此外，此 API 为组件、端点、消费者和制作者类提供专门的基础类。

45.2. 如何使用框架

概述

使用 API 框架实施组件的过程涉及通过编辑 Maven POM 文件来混合使用自动化代码生成、实施 Java 代码和自定义构建。下图显示了此开发流程的概述。

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，在 Camel 上下文中，这通常是一个 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？以下是几个可能性：

- 自行实施 Java API （虽然这通常涉及很多工作，但通常不是首选的方法）。
- 使用第三方 Java API。例如，Apache Camel Box 组件基于第三方 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 的形式为 Java API 提供元数据（这是在 API 组件框架中生成代码所必需的）。如果您从 Maven 存储库使用第三方 Java API，您通常会发现 Maven 工件中已提供了 Javadoc。但是，即使在 未提供 Javadoc 时，您也可以使用 maven-javadoc-plugin Maven 插件轻松生成它。

注意

目前，Javadoc 元数据的处理存在限制，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; 不被支持。解决方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，无法以 Javadoc 的形式提供 Java API 元数据，您可以选择以 签名 文件的形式提供元数据。签名文件由一个方法签名列表组成（每行一个方法签名）。这些文件可以手动创建，且仅在构建时需要。

请注意有关签名文件的以下点：

- *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。运行时引发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)。没有导入软件包名称的机制。*
- *目前，签名解析程序中有一个限制，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; 不被支持。解决办法是将嵌套的通用类型指定为 java.util.List<java.util.List >。*

下面显示了签名文件的内容的简单示例：

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

使用 Maven archetype 生成开始代码

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，它组成了 API 组件的基础。在构建此项目时，它会在 Maven 捆绑包中打包 Java API，并生成必要的 Javadoc。如果 Java API 和 Javadoc 已由第三方提供，则不需要此子项目。

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，以开发自己的组件实现。以下生成的类构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 文件以自定义构建，并配置 camel-api-component-maven-plugin Maven 插件。

配置 camel-api-component-maven-plugin

配置 POM 文件最重要的方面是 camel-api-component-maven-plugin Maven 插件的配置。此插件负责在 API 方法和端点 URI 之间生成映射，以及编辑插件配置，您可以自定义映射。

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 (方案://hello-javadoc/端点)从这个类调用方法。存在 fromJavadoc 元素表示 ExampleJavadocHello 类从 Javadoc 获取其元数据。

OSGi 捆绑包配置

组件子项目的 POM 示例 ProjectName-component/pom.xml 配置为将组件打包为 OSGi 捆绑包。组件 POM 包含 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

当您使用 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），将它们放在 target/classes 项目子目录中。当您处理大型和复杂的 Java API 时，此生成的代码实际上构成了大量组件源代码。

Maven 构建完成后，编译的代码和资源被打包为 OSGi 捆绑包，并作为 Maven 工件保存在您的本地 Maven 存储库中。

第 46 章 FRAMEWORK 入门

摘要

本章解释了根据使用 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：如果有多个简单的参数，它会生成一个完整的、可正常工作的 Maven 项目，它填充了示例代码。然后，您可以使用此项目作为模板，自定义实施来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，可以为您自己的 API 组件实现生成起点代码。这是推荐的方法，可以开始创建自己的 API 组件。

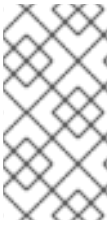
先决条件

运行 camel-archetype-api-component archetype 的唯一先决条件是安装了 Apache Maven，Maven settings.xml 文件被配置为使用标准的 Fuse 存储库。

调用 Maven archetype

要创建使用示例 URI 方案的示例组件，请调用 camel-archetype-api-component archetype 来生成新的 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_13_0-00013-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 \ 表示行连续，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 为 archetype generation 命令提供选项。上述 mvn archetype:generate 命令所示设置大多数选项，但可以修改一些选项以自定义生成的项目。下表显示了可用于自定义生成的 API 组件项目的选项：

Name	描述
groupId	**（通用 Maven 选项）** 指定生成的 Maven 项目的组 ID。默认情况下，这个值还定义生成的类的 Java 软件包名称。因此，最好选择这个值以匹配您想要的 Java 软件包名称。
artifactId	**（通用 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（实际上，建议该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。您应该确保此方案不会与任何现有 Camel 组件的方案冲突。
archetypeVersion	**（通用 Maven 选项）** 通常，这应该是您计划部署组件的容器使用的 Apache Camel 版本。但是，如果需要，您还可以在生成 项**目后** 修改 Maven 依赖项版本。

生成的项目的结构

假设代码生成步骤成功完成，您应该会看到包含新 Maven 项目的新目录 camel-api-example。如果您在 camel-api-example 目录中查看，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/
```

在项目顶层是聚合 POM 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）包含您要转换为组件的 Java API。如果您要将 API 组件捆绑到您编写的 Java API 上，您可以将 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（如果它还没有作为 Maven 软件包提供）。
- 为 Java API 生成 Javadoc （为 API 组件框架提供所需的元数据）。
- 要从 API 描述生成 Java API 代码（例如，从 REST API 的 WADL 描述中）。

然而，在某些情况下，您可能不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 组件的实现。这包括组件实施类，以及 camel-api-component-maven 插件的配置（从 Java API 生成 API 映射类）。

46.2. 生成的 API 子项目

概述

假设您生成了新的 Maven 项目，如 第 46.1 节 “使用 Maven Archetype 生成代码” 所述，您现在可以找到 Maven 子项目，用于将 Java API 打包到 camel-api-example/camel-api-example-api 项目目录中。在本节中，我们仔细查看生成的示例代码，并描述它的工作原理。

Java API 示例

生成的示例代码包含一个示例 Java API，其示例 API 组件基于它。示例 Java API 相对简单，它由两个 Hello World 类组成： ExampleJavadocHello 和 ExampleFileHello。

JavadocHello 类示例

例 46.1 “JavadocHello 类示例” 显示示例 Java API 中的 ExampleJavadocHello 类。作为类名称建

议，此特定类用于显示如何提供 Javadoc 中的映射元数据。

例 46.1. JavadocHello 类示例

```
// 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。作为类名称建议，这个特定类用于显示如何提供签名文件中的映射元数据。

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，将 maven-javadoc-plugin 配置为在 Maven 构建期间自动执行这个步骤。

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，我们仔细查看生成的示例代码，并描述它的工作原理。

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 文件中被定义为依赖项，camel-api-example-component/pom.xml，如下所示：

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

如果您将 Javadoc 元数据用于 Java API 的所有或部分，则必须提供 Javadoc 作为组件 POM 中的依

赖项。关于这个依赖项需要注意两个事项：

- *Javadoc 的 Maven 协调与 Java API 的几乎相同，但您必须指定一个 classifier 元素，如下所示：*

```
<classifier>javadoc</classifier>
```

- *您必须声明 Javadoc 来提供 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>
```

定义文件 Hello 示例的文件元数据

ExampleFileHello 的元数据在签名文件中提供。通常，必须手动创建此文件，但它有一个简单格式，它由一个方法签名列表组成。示例代码在目录中提供签名文件 file-sig-api.txt，位于 camel-api-example-component/signatures 中，其中包含以下内容：

```
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 组件框架的一个主要功能是，它会自动生成代码来执行 API 映射。也就是说，生成将端点 URI 映射到 Java API 上调用的方法的存根代码。API 映射的基本输入是：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，该插件在组件 POM 中配置。以下从组件 POM 中提取演示了如何配置 camel-api-component-maven-plugin 插件：

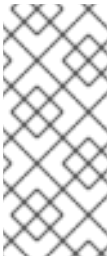
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">

  ...
  <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，其中包含一个用于配置 Java API 类的 apis 子元素。每个 API 类都由 api 元素配置，如下所示：

apiName

API 名称是 API 类的短名称，用作端点 URI 的 endpoint-prefix 部分。



注意

如果 API 仅包含单个 Java 类，您可以将 apiName 元素留空，以便 endpoint-prefix 变得冗余，然后使用 “单个 API 类的 URI 格式”一节 中显示的格式指定端点 URI。

proxyClass

proxy class 元素指定 API 类的完全限定域名。

fromJavadoc

如果 API 类附带有 Javadoc 元数据，则必须通过 包含来自Javadoc 元素的 Javadoc 元素，且 Javadoc 本身还必须在 Maven 文件中指定，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 类由签名文件元数据提供，您必须通过包含 fromSignatureFile 元素来指示它，其中此元素的内容指定签名文件的位置。

注意

签名文件不会包含在 Maven 构建的最终软件包中，因为这些文件仅在构建时需要，而不是在运行时。

生成的组件实现

API 组件由以下核心类（必须为每个 Camel 组件实施）组成，在 camel-api-example-component/src/main/java 目录下：

ExampleComponent

代表组件本身。此类充当端点实例的工厂（例如，ExampleEndpoint的实例）。

ExampleEndpoint

代表端点 URI。此类充当消费者端点的工厂（如 ExampleConsumer）和生成者端点的工厂（如 Producer）。

ExampleConsumer

代表消费者端点的一个特定实例，它能够消耗来自端点 URI 中指定的位置的消息。

ExampleProducer

代表制作者端点的协调实例，它可以发送消息到端点 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 选项不与任何 特定的 API 类关联。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以连接到远程服务，这非常有用。ExampleConfiguration 类的主要目的是，为实例化 API 类或实施 API 接口的类提供所需的参数值。例如，它们可以是 constructor 参数，也可以是 factory 方法或类的参数值。

要实施 URI 选项，选项，在此类中，您需要做的一切都实施对访问器方法，获得 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。

ExampleComponent class

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，它会创建新的端点实例。通常，您不需要更改组件类中的任何默认代码。如果存在与这个组件相同的生命周期的其他对象，但您可能希望将这些对象从组件类提供（例如，通过添加方法来创建这些对象或将这些对象注入组件）。

Endpoint 类示例

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是创建 API 代理。API 代理是来自目标 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 中显示的端点前缀来选择适当的 API 类（调用 URI 具有通用形式，scheme:// endpoint-prefix /端点）。

ExampleConsumer class

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
}
```

要在此类中添加 URI 选项选项，请在该类中定义相应类型的字段，并实施相应的访问器方法对，getOption 和 setOption。组件框架自动解析端点 URI，并在运行时注入选项值。

注意

此类用于定义常规 URI 选项，可与任何 API 方法结合使用。要定义与特定 API 方法关联的 URI 选项，请在 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

回想 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，假设您想调用 API 方法，如 JavadocHello.greetMe ("Jane Doe")，将构建 URI，如下所示：

scheme

API 组件方案，如您使用 Maven archetype 生成代码时指定。在这种情况下，方案是 示例。

endpoint-prefix

API 名称，映射到由 camel-api-component-maven-plugin Maven 插件配置定义的 API 类。对于 ExampleJavadocHello 类，相关配置是：

```
<configuration>
  <apis>
    <api>
      <apiName>hello-javadoc</apiName>
      <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
      <fromJavadoc/>
    </api>
    ...
  </apis>
</configuration>
```

显示所需的 endpoint-prefix 为 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，它可以指定为 name=Jane%20Doe。如果要为选项定义默认值，可以通过覆盖拦截器 Properties 方法（请参阅 第 46.4 节 “编程模型”）。

将 URI 组件放在一起，我们发现我们可以使用以下 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 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
```

此资源文件是启用 Camel 内核来识别与 示例 URI 方案关联的组件。每当您在路由中使用 example:// URI 时，Camel 会搜索 classpath 来查找对应的 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 内核创建 ExampleComponent 组件的默认实例。如果您重构组件类的名称，您才需要编辑此文件。

46.4. 编程模型

概述

在 API 组件框架的上下文中，主要的组件实施类派生自 org.apache.camel.util.component 软件包中的基础类。这些基础类定义了在实施组件时可以（可选）覆盖的一些方法。在本节中，我们提供了这些方法的简短描述，以及如何在您自己的组件实施中使用它们。

要实现的组件方法

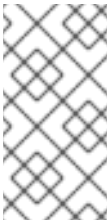
除了生成的方法实现（通常不需要修改），您可以选择在 组件 类中覆盖以下方法：

doStart()

（可选）在冷启动期间为组件创建资源的回调。另一种方法是采用延迟 初始化 的策略（仅在需要资源时重新创建资源）。实际上，lazy 初始化通常是最佳策略，因此通常不需要 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，其所有资源都会关闭，删除内部状态，缓存会被清除，以此类推。



注意

Camel 保证当当前 CamelContext 关闭时 始终 调用 doStop，即使未调用对应的 doStart。

doShutdown

（可选） 在 CamelContext 关闭时调用代码的回调。已停止的组件可以重新启动（使用冷启动语义），而关闭的组件将完全完成。因此，此回调代表最后一次释放属于组件的任何资源的机会。

组件类中实施哪些其他内容？

组件 类是包含对组件对象本身相同（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，则最好在 组件 类中包含对所需 OAuth 对象的引用，并在 Component 类中定义用于创建 OAuth 对象的方法。

要实现的端点方法

您可以修改一些生成的方法，并（可选）覆盖 Endpoint 类中的一些继承方法，如下所示：

afterConfigureProperties()

此方法中需要做的主要事情是创建适当的代理类(API 类)，以匹配 API 名称。API 名称（已从端点 URI 中提取）可以通过继承的 apiName 字段或通过 getApiName 访问器提供。通常，您将在 apiName 字段上有一个开关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 method, Map<String, Object> args)

覆盖此方法，以返回您在 afterConfigureProperties 中创建的代理实例。例如：

```
@Override
public Object getApiProxy(ApiMethod method, Map<String, Object> args) {
    return apiProxy;
}
```

在特殊情况下，您可能希望根据 API 方法和参数选择代理。getApiProxy 为您提供了采取此方法的灵活性（如果需要）。

doStart()

（可选） 在冷启动期间创建资源的回调。具有与 Component.doStart （） 相同的语义。

doStop()

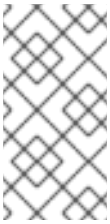
（可选） 在组件停止时调用代码的回调。具有与 Component.doStop （） 相同的语义。

doShutdown

（可选） 在组件关闭时调用代码的回调。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 和提供选项值来确定要调用的方法（由于过载和别名而导致不确定性）。如果组件内部添加选项或方法参数，则框架可能需要帮助来确定要调用的正确方法。在这种情况下，您必须覆盖 interceptPropertyNames 方法，并将额外的(hidden 或 implicit)选项添加到 propertyNames 集。当设置 propertyNames 中提供了方法参数的完整列表时，框架将能够识别要调用的正确方法。

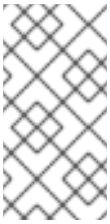


注意

您可以在 Endpoint,Producer 或 Consumer 类级别上覆盖此方法。基本规则是，如果选项影响 生成者 端点和消费者端点，请覆盖 Endpoint 类中的方法。

interceptProperties (Map<String,Object> 属性)

（可选） 通过在调用 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。实际上，通常需要同时覆盖 interceptPropertyNames 方法和 interceptProperty 方法。



注意

您可以在 Endpoint,Producer 或 Consumer 类级别上覆盖此方法。基本规则是，如果选项影响 生成者 端点和消费者端点，请覆盖 Endpoint 类中的方法。

实施的消费者方法

您可以选择覆盖 Consumer 类中的一些继承方法，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> 属性)

（可选） 此方法的语义与 Endpoint.interceptProperties类似

doInvokeMethod (Map<String, Object> args)

（可选） 通过此方法可以截获 Java API 方法的调用。覆盖此方法的最常见原因是自定义有关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!
        ...
    }
}
```

在这种实现时，您应该在超级类上调用 doInvokeMethod，以确保调用 Java API 方法。

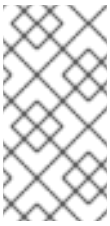
interceptResult (Object methodResult, Exchange resultExchange)

（可选） 对 API 方法调用的结果执行一些额外的处理。例如，您可以在 Camel Exchange 对象中添加自定义标头 resultExchange。

Object splitResult (Object result)

（可选） 默认情况下，如果方法 API 调用的结果是一个 java.util.Collection 对象或 Java 数组，API 组件框架会将结果分成多个交换对象（以便单个调用结果转换为多个消息）。

如果要更改默认的行为，您可以在消费者端点中覆盖 splitResult 方法。result 参数包含 API 消息调用的结果。如果要分割结果，您应该返回一个数组类型。



注意

您还可以通过在端点 URI 中设置 consumer.splitResult=false 来关闭默认分割行为。

要实现的生成者方法

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> 属性)

（可选） 此方法的语义与 Endpoint.interceptProperties类似

doInvokeMethod (Map<String, Object> args)

（可选） 此方法的语义与 Consumer.doInvokeMethod 类似。

interceptResult (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 方法会导致与消费者端点一样。要获得与生成者端点类似的效果，您可以使用 Camel 的 split （） DSL 命令（标准企业集成模式之一）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 被调度轮询消费者。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 “调度的轮询消费者实现”一节。

46.5. 组件实施示例

概述

Apache Camel 分发的多个组件已通过 API 组件框架的协助实现。如果要了解更多有关使用框架实施 Camel 组件的技术，最好研究这些组件实施的源代码。

box.com

Camel Box 组件 演示了如何使用 API 组件框架对第三方 Box.com Java SDK 进行建模和调用。它还演示了如何对框架进行调整以自定义消费者轮询，以支持 Box.com 的长期轮询 API。

GoogleDrive

Camel GoogleDrive 组件 演示了 API 组件框架如何处理方法对象风格的 Google API。在这种情况下，URI 选项映射到方法对象，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 组件框架嵌套基于回调的异步 API。本例演示了如何将异步处理推送到底层资源中，如 HTTP NIO 连接，以使 Camel 端点更高效。

第 47 章 配置 API 组件 MAVEN 插件

摘要

本章介绍了对 API 组件 Maven 插件上所有可用的配置选项的参考。

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，该类在端点 URI 和 API 方法调用之间实施映射。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。

生成代码的位置

默认情况下，API 组件 Maven 插件生成的 API 映射类放置在以下位置：

```
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。通过将插件声明为常规 Maven 依赖项(Javadoc Maven 依赖项应当与 提供的范围声明的位置)提供给 插件。

设置插件

设置 API 组件 Maven 插件的建议方法是使用 API 组件 archetype 生成起点代码。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该文件。插件设置的主要方面有如下：

1. 必须声明需要 Java API 和 Javadoc 元数据的 Maven 依赖项。
2. 插件的基本配置在 pluginManagement 范围（也定义要使用的插件版本）中声明。
3. 插件实例本身被声明和配置。

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 生成代码时在 Maven 插件管理 范围中定义：

```
<?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_13_0-00013-redhat-00001</version>
          <configuration>
            <scheme>${schemeName}</scheme>
            <componentName>${componentName}</componentName>
            <componentPackage>${componentPackage}</componentPackage>
            <outPackage>${outPackage}</outPackage>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
    ...
  </build>
  ...
</project
```

pluginManagement 范围中指定的配置为插件提供默认设置。它实际上不会创建插件实例，但其默认设置将供任何 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。这个软件包也会由默认的 maven-bundle-plugin 配置导出。因此，如果您希望一个类公开可见，您应该将它放在此 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，这具有 componentName 属性的值，并添加 .internal 后缀。这个软件包被默认 maven-bundle-plugin 配置声明为 private。因此，如果您想一个类是私有的，您应该将它放在此 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，该插件配置为在 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，其中包含一个用于配置 Java API 类的 apis 子元素。每个 API 类都由 api 元素配置，如下所示：

apiName

API 名称是 API 类的短名称，用作端点 URI 的 endpoint-prefix 部分。

注意

如果 API 仅包含单个 Java 类，您可以将 apiName 元素留空，以便 endpoint-prefix 变得冗余，然后使用 “单个 API 类的 URI 格式”一节 中显示的格式指定端点 URI。

proxyClass

此元素指定 API 类的完全限定域名。

fromJavadoc

如果 API 类由 Javadoc 元数据提供，您必须通过包含 fromJavadoc 元素和 Javadoc 本身的 Javadoc 本身来指示这一点，作为 提供的 依赖项在 Maven 文件中指定。

fromSignatureFile

如果 API 类由签名文件元数据提供，您必须通过包含 fromSignatureFile 元素来指示它，其中此元素的内容指定签名文件的位置。

注意

签名文件不会包含在 Maven 构建的最终软件包中，因为这些文件仅在构建时需要，而不是在运行时。

自定义 API 映射

可以通过配置插件来自定义 API 映射的以下方面：

- 方法别名 wagon- swigyou 可以使用别名配置元素为 API 方法定义附加名称(别名)。详情请查看 第 47.3 节 “方法别名”。
- 可为空 选项 wagon- swig 您可以使用 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 “可为空选项”。
- 参数名称 替换 API 映射的实现方式，特定 API 类中的所有方法的参数 属于同一 命名空间。如果声明具有相同名称的两个参数不同类型，这会导致冲突。为避免此类名称冲突，您可以使用替换配置元素 来重命名方法参数（因为它们将显示在 URI 中）。详情请查看 第 47.5 节 “参数名称替换”。
- 当将 Java 参数 映射到 URI 选项时，除了参数 HEKETI-InventoryService 时，您可能需要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 “排除的参数”。
- 您可能要定义不属于 Java API 的额外选项，它们可能要定义不属于 Java API 的额外选项。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 元数据来忽略或明确包含某些内容。有关如何执行此操作的详情，请参考 第 47.2 节 “javadoc 选项”。

配置签名文件元数据

如果没有 Javadoc 可用，您可以利用签名文件来提供所需的映射元数据。fromSignatureFile 用于指定对应签名文件的位置。它没有特殊选项。

47.2. JAVADOC 选项

概述

如果您的 Java API 元数据由 Javadoc 提供，则通常只需指定不带选项的 Javadoc 元素即可指定。但是，如果您不想在 API 映射中包含整个 Java API，您可以过滤 Javadoc 元数据以自定义内容。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 元数据的不需要的部分来自定义生成的 API 映射的范围。

语法

Javadoc 元素中 可以使用可选的子元素进行配置，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

如以下提取所示，fromJavadoc 元素可以选择性地显示为 apis 元素的子级和/或作为 api 元素的子级：

```
<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 元素的子级，Javadoc 选项中的Javadoc 选项仅适用于 api 元素指定的 API 类。
-

作为 Javadoc 选项中的 apis 元素的子级，默认适用于所有 API 类，但可以在 api 级别上覆盖。

选项

以下选项可以定义为 Javadoc 的子元素：

excludePackages

指定正则表达式(java.util.regex 语法)，用于从 API 映射模型中排除 Java 软件包。所有与正则表达式匹配的软件包名称都会被排除；从排除类派生的所有类也会忽略。默认值为 javax?\.lang.*。

excludeClasses

指定正则表达式(java.util.regex 语法)，用于从 API 映射中排除 API 基础类。所有与正则表达式匹配的类名称都会被排除；从排除类派生的所有类也会忽略。

excludeMethods

指定正则表达式(java.util.regex 语法)，用于从 API 映射模型中排除方法。

includeMethods

指定正则表达式(java.util.regex 语法)，用于包括 API 映射模型中的方法。

includeStaticMethods

如果为 true，则静态方法也会包含在 API 映射模型中。默认为 false。

47.3. 方法别名

概述

除了 Java API 中显示的标准方法外，除了出现在 Java API 中的标准方法名称外，为给定方法定义其他名称（别名）通常很有用。特别常见的情形是允许属性名称（如 小部件）用作访问器方法的别名（如 getWidget 或 setWidget）。

语法

alias 元素可以使用一个或多个别名子元素定义，如下所示：

```
<aliases>
```

```
  <alias>
    <methodPattern>MethodPattern</methodPattern>
    <methodAlias>Alias</methodAlias>
  </alias>
  ...
</aliases>
```

其中 MethodPattern 是一个正则表达式(java.util.regex 语法)，用于匹配 Java API 中的方法名称，其模式通常包含捕获组。Alias 是替换表达式（用于 URI），它可以使用前面捕获组的文本（例如，指定为 $1、$2 或 $3，用于来自第一个、第二个或第三个捕获组的文本）。

影响范围

如以下提取所示，alias 元素可以选择性地显示为 apis 元素的子级和/或作为 api 元素的子级：

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <aliases>...</aliases>
    </api>
    <aliases>...</aliases>
    ...
  </apis>
</configuration>
```

您可以在以下范围中定义 aliases 元素：

- 作为 api 元素的子级，则 别名 映射仅应用到 api 元素指定的 API 类。
- 作为 apis 元素的子级，它默认应用于所有 API 类，但可以在 api 级别上覆盖。

Example

以下示例演示了如何为通用 get/set bean 方法模式生成别名：

```
<aliases>
  <alias>
    <methodPattern>[gs]et(.+)</methodPattern>
```

```
    <methodAlias>$1</methodAlias>
  </alias>
</aliases>
```

使用前面的别名定义，您可以使用 小部件 作为任一方法 getWidget 或 setWidget 的别名。请注意，使用 capture 组 (.+) 来捕获方法名称的后者部分（如 Widget）。

47.4. 可为空选项

概述

在某些情况下，让方法参数默认为 null 可能会有意义。但默认情况下不允许这样做。如果要允许 Java API 中的一些方法参数采用 null 值，则必须使用 nullableOptions 元素明确声明它。

语法

nullableOptions 元素可以使用一个或多个 nullableOption 子元素定义，如下所示：

```
<nullableOptions>
  <nullableOption>ArgumentName</nullableOption>
  ...
</nullableOptions>
```

其中 ArgumentName 是 Java API 中的方法参数的名称。

影响范围

如以下提取所示，nullableOptions 元素可以选择性地显示为 apis 元素和/或作为 api 元素的子级：

```
<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 元素的子级 wagon-wagonthe nullableOptions 映射仅适用于 api 元素指定的 API 类。
- 作为 apis 元素的子级，它默认应用于所有 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 在每个代理类中唯一(Java API 类)。但是，这并非始终是方法参数名称的情况。例如，考虑 API 类中的以下 Java 方法：

```
public void doSomething(int id, String name);
public void doSomethingElse(int id, String name);
```

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，在前面的方法中，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 参数出现多次作为不同类型，则发生了什么情况，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，以及在同一范围内返回 String 的 getId 方法。这个问题的解决方案是使用 参数名替换来自定义参数名称到 URI 选项名称 的映射。

语法

可以使用一个或多个 替换 子元素定义 替换 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 元素可以选择性地显示为 apis 元素的子级和/或作为 api 元素的子级：

```
<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 元素的子级，则 替换 仅应用到 api 元素指定的 API 类。
- 作为 apis 元素的子级，它默认应用于所有 API 类，但可以在 api 级别上覆盖。

子元素

每个 替换 元素可使用以下子元素定义：

方法

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)以匹配匹配方法中的参数名称，其中模式通常包含捕获组。

argType

（可选） 指定一个正则表达式(java.util.regex 语法)以匹配参数的类型。如果将 replaceWithType 选项设置为 true，则通常在此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，替换 元素定义替换参数名称（用于在 URI 中使用）。可以使用从 argName 正则表达式模式捕获的字符串（使用语法 $1、$2、$2、$3 分别插入第一个、第二个或第三个捕获组）来构建替换文本。或者，如果将 replaceWithType 选项设置为 true，则可以使用从 argType 正则表达式模式捕获的字符串来构建替换文本。

replaceWithType

为 true 时，使用从 argType 正则表达式捕获的字符串组成替换文本。默认值为 false。

Example

以下替换示例通过将 suffix, Param 添加到参数名称来修改 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，给定以下方法签名：

```
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. 排除的参数

概述

有时，当涉及到将 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 是匹配参数名称和参数类型的正则表达式。

影响范围

如以下提取所示，excludeConfigNames 元素和 excludeConfigTypes 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>
      ...
      <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 元素的子级，它只适用于 api 元素指定的 API 类。

- 作为 apis 元素的子级，它默认应用于所有 API 类，但可以在 api 级别上覆盖。

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

根据匹配参数名称，为排除参数指定正则表达式(java.util.regex 语法)。

excludeConfigTypes

根据匹配参数类型，为排除参数指定正则表达式(java.util.regex 语法)。

47.7. 额外选项

概述

extraOptions 选项通常用于通过提供更简单的选项来计算或隐藏复杂的 API 参数。例如，API 方法可能采用 POJO 选项，该选项可以更轻松地作为 URI 中的 POJO 的部分提供。组件可以通过添加部分作为额外选项并在内部创建 POJO 参数来实现此目的。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，OptionName 是额外 URI 选项的名称。

影响范围

如以下提取所示，extraOptions 元素可以选择性地显示为 apis 元素的子级和/或作为 api 元素的子

级：

```
<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** 元素的子级，默认应用于 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。

Example

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