Red Hat Fuse 7.9

Apache Camel 開発ガイド

Apache Camel を使用したアプリケーションの開発

Last Updated: 2023-08-02

Red Hat Fuse 7.9 Apache Camel 開発ガイド

Apache Camel を使用したアプリケーションの開発

法律上の通知

概要

本ガイドでは、Apache Camel で Red Hat Fuse アプリケーションを開発する方法を説明します。ここでは、基本的なビルディングブロック、エンタープライズ統合パターン、ルーティング式および述語言語の基本的な構文、Apache CXF コンポーネントを使った Web サービスの作成、Apache Camel API の使用、Java API をラップする Camel コンポーネントの作成方法について説明します。

目次

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、CTO である Chris Wright のメッセージ をご覧ください。

パート I. エンタープライズ統合パターンの実装

ここでは、Apache Camel を使用してルートを構築する方法を説明します。基本的なビルディングブロックおよび EIP コンポーネントをカバーします。

第1章 ルート定義のためのビルディングブロック

概要

Apache Camel はルートを定義するために、Java DSL と Spring XML DSL の 2 つの **ドメイン固有言語 (DSL)** をサポートします。ルートを定義するための基本的なビルディングブロックは **エンドポイント** および **プロセッサー** で、プロセッサーの動作は通常 **式** または論理 **述語** によって変更されます。Apache Camel は、さまざまな言語を使用して式や述語を定義できます。

1.1. ROUTEBUILDER クラスの実装

概要

ドメイン固有言語 (DSL) を使用するには、**RouteBuilder** クラスを拡張し、その **configure()** メソッド (ここにルーティングルールを定義します) を上書きします。

必要な数だけ **RouteBuilder** クラスを定義できます。各クラスは一度インスタンス化され、**CamelContext** オブジェクトに登録されます。通常、各 **RouteBuilder** オブジェクトのライフサイクルは、ルーターをデプロイするコンテナーによって自動的に管理されます。

RouteBuilder クラス

ルート開発者の主なロールは、1 つ以上の **RouteBuilder** クラスを実装することです。以下の 2 つの代替 **RouteBuilder** クラスを継承できます。

- **org.apache.camel.builder.RouteBuilder**: これは、**あらゆる** コンテナータイプへのデプロイに適した汎用の **RouteBuilder** ベースクラスです。**camel-core** アーティファクトで提供されます。
- **org.apache.camel.spring.SpringRouteBuilder**: このベースクラスは、Spring コンテナーに特別に適合されています。特に、以下の Spring 固有の機能に対する追加サポートが提供されます。追加サポートの対象となる機能は、Spring レジストリーでの Bean 検索 (**beanRef()** Java DSL コマンドを使用) およびトランザクション (詳細は **トランザクションガイド** を参照) です。**camel-spring** アーティファクトで提供されます。

RouteBuilder クラスは、ルーティングルールの開始に使用されるメソッドを定義します (例: **from()**、**intercept()**、および **exception()**)。

RouteBuilder の実装

例1.1「RouteBuilder クラスの実装」 は、最小限の **RouteBuilder** 実装を示しています。**configure()** メソッド本文にはルーティングルールが含まれ、各ルールは単一の Java ステートメントです。

例1.1 RouteBuilder クラスの実装

```
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.
    // ...
}
}
```

from(URL1).to(URL2) ルールの形式は、**src/data** ディレクトリーからファイルを読み込み、それを **target/messages** ディレクトリーに送信するようルーターに指示します。**?noop=true** オプションは、**src/data** ディレクトリー内のソースファイルを (削除ではなく) 保持するようにルーターに指示します。

注記

Spring または Blueprint で **contextScan** を使用して **RouteBuilder** クラスをフィルターする場合、Apache Camel はデフォルトでシングルトン Bean を検索します。ただし、以前の動作をオンにして、新しいオプション **includeNonSingletons** でスコープされたプロトタイプを含めることができます。

1.2. 基本的な JAVA DSL 構文

DSL とは

ドメイン固有言語 (DSL) は、特別な目的のために設計されたミニ言語です。DSL は論理的に完全である必要はありませんが、選択したドメインで適切に問題を記述するのに十分な表現力が必要になります。通常、DSL には専用のパーサー、インタープリター、またはコンパイラーは **必要ありません**。DSL コンストラクトがホスト言語 API のコンストラクトに適切にマッピングされていれば、DSL は既存のオブジェクト指向ホスト言語の上に便乗できます。

架空の DSL で以下の一連のコマンドについて考えてみましょう。

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

これらのコマンドは、以下のような Java メソッド呼び出しにマップできます。

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

ブロックを Java メソッド呼び出しにマップすることもできます。以下に例を示します。

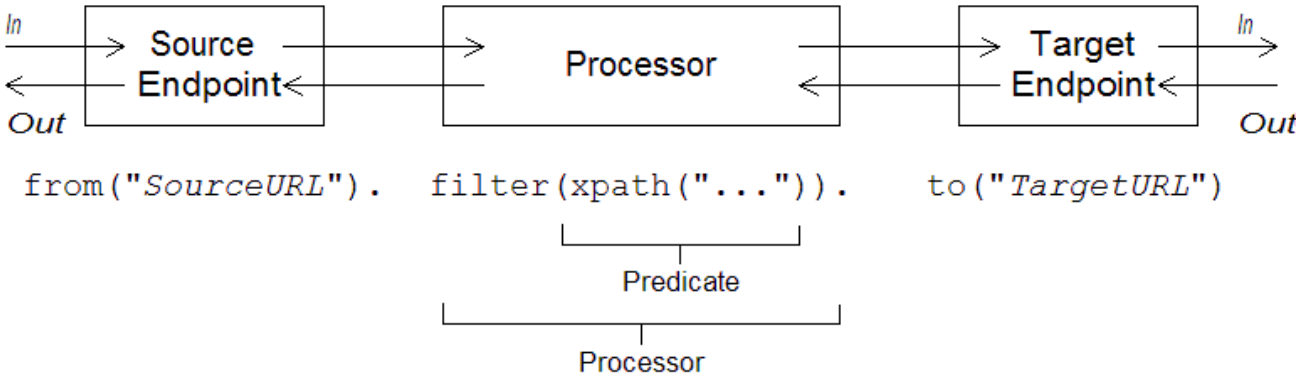
```
command01().startBlock().command02().command03().endBlock()
```

DSL 構文は、ホスト言語 API のデータ型によって暗黙的に定義されます。たとえば、Java メソッドの戻り値の型によって、次 (DSL の次のコマンドと同等) をアクティブに呼び出すことのできるメソッドが決定します。

ルータールール構文

Apache Camel は、ルーティングルールを定義する **ルーター DSL** を定義します。この DSL を使用して、**RouteBuilder.configure()** 実装のボディーにルールを定義できます。図1.1「ローカルルーティングルール」 は、ローカルルーティングルールを定義する基本的な構文の概要を示しています。

図1.1 ローカルルーティングルール



ローカルルールは、常に **from("EndpointURL")** メソッドで開始します。このメソッドは、ルーティングルールのメッセージのソース (**コンシューマーエンドポイント**) を指定します。その後、ルールに任意の長いプロセッサーチェーンを追加できます (例: **filter()**)。通常、**to("EndpointURL")** メソッドでルールを終了します。このメソッドは、ルールを通過するメッセージのターゲット (**producer endpoint**) を指定します。ただし、常に **to()** でルールを終了する必要はありません。メッセージのターゲットをルールに指定する別の方法もあります。

注記

特別なプロセッサータイプ (**intercept()**、**exception()**、または **errorHandler()** など) でルールを開始することで、グローバルルーティングルールを定義することもできます。グローバルルールは本ガイドの対象範囲外です。

コンシューマーおよびプロデューサー

ローカルルールは、常に **from("EndpointURL")** を使用してコンシューマーエンドポイントを定義して開始します。そして (常にではありませんが) 通常は、**to("EndpointURL")** を使用してプロデューサーエンドポイントを定義して終了します。エンドポイント URL である **EndpointURL** は、デプロイ時に設定された任意のコンポーネントを使用できます。たとえば、ファイルエンドポイント **file:MyMessageDirectory**、Apache CXF エンドポイント **cxf:MyServiceName**、または Apache ActiveMQ エンドポイント **activemq:queue:MyQName** を使用できます。全てのコンポーネント型を網羅するリストは、Apache Camel Component Reference を参照してください。

エクスチェンジ

エクスチェンジ オブジェクトは、メタデータで拡張されたメッセージで設定されます。エクスチェンジはメッセージがルーティングルールを介して伝搬される際に使われる標準形式であるため、 Apache Camel において最も重要です。エクスチェンジの主な設定要素は次のとおりです。

- **In**メッセージ - エクスチェンジによってカプセル化された現在のメッセージです。エクスチェンジがルートを通過するにつれて、このメッセージは変更される場合があります。そのため、ルート開始時の **In** メッセージは、通常ルート終了時の **In**メッセージとは **異なります**。**org.apache.camel.Message** タイプは、以下の部分を含むメッセージの汎用モデルを提供します。
  - ボディー
  - ヘッダー
  - アタッチメント

  これはメッセージの **汎用** モデルであることを理解することが重要です。Apache Camel は、さ

まざまなプロトコルおよびエンドポイントをサポートします。したがって、メッセージのボディーまたはヘッダーの形式を標準化することは **できません**。たとえば、JMS メッセージのボディー形式は、HTTP メッセージまたは Web サービスメッセージのボディーとは全く異なります。このため、ボディーとヘッダーは **Object** タイプと宣言されます。ボディーとヘッダーの元のコンテンツは、エクスチェンジインスタンスを作成するエンドポイントによって決定されます (つまり、**from()** コマンドに指定されたエンドポイント)。

- **out** メッセージ - 返信メッセージまたは変換されたメッセージの一時的な保持領域です。特定の処理ノード (特に **to()** コマンド) は、**In** メッセージをリクエストとして処理し、これをプロデューサーエンドポイントに送信し、そのエンドポイントからのリプライを受信することで、現在のメッセージを変更できます。リプライメッセージは、エクスチェンジの **Out** メッセージスロットに挿入されます。
  通常、現在のノードで **Out** メッセージが設定された場合、Apache Camel はエクスチェンジを次のように変更してからルートの次のノードに渡します: **In** メッセージを破棄し、**Out** メッセージを **In** メッセージのスロットに移動します。そのため、リプライは最新のメッセージになります。Apache Camel がノード同士をルート内で接続する方法の詳細は、「パイプライン処理」を参照してください。

  ただし、**Out** メッセージが異なる方法で扱われる特殊なケースが 1 つあります。ルート開始時のコンシューマーエンドポイントがリプライメッセージを期待している場合、ルートの最終端にある **Out** メッセージはコンシューマーエンドポイントのリプライメッセージとみなされます (さらに、この場合、最終ノードが **Out** メッセージを作成 **しなければならず**、さもなくばコンシューマーエンドポイントはハングしてしまいます)。

- メッセージ交換パターン (MEP) - 以下のように、ルート内でのエクスチェンジとエンドポイント間のやり取りに影響を与えます。

  - **コンシューマーエンドポイント** - 元となるエクスチェンジを作成するコンシューマーエンドポイントは、MEP の初期値を設定します。初期値は、コンシューマーエンドポイントがリプライするか (例: **InOut** MEP)、リプライしないか (例: **InOnly** MEP) を示します。

  - **プロデューサーエンドポイント** - MEP は、エクスチェンジがルートで通過するプロデューサーエンドポイントに影響します (例: エクスチェンジが **to()** ノードを通過する場合)。たとえば、現在の MEP が **InOnly** の場合、**to()** ノードはエンドポイントからのリプライを受け取ることを期待しません。場合によっては、エクスチェンジのプロデューサーエンドポイントとのやり取りをカスタマイズするために、現在の MEP を変更する必要があります。詳細は、「エンドポイント」 を参照してください。

- エクスチェンジプロパティー - 現在のメッセージのメタデータが含まれる名前付きプロパティーのリスト。

メッセージ交換パターン

Exchange オブジェクトを使用すると、メッセージ処理を異なる **メッセージ交換パターン** に簡単に一般化することができます。たとえば、非同期プロトコルは、コンシューマーエンドポイントからプロデューサーエンドポイントに流れる単一のメッセージで設定される MEP を定義する場合があります (**InOnly** MEP)。一方、RPC プロトコルは、リクエストメッセージとリプライメッセージで設定される MEP を定義する場合があります (**InOut** MEP)。現在、Apache Camel は以下の MEP をサポートします。

- **InOnly**
- **RobustInOnly**
- **InOut**
- **InOptionalOut**

- **OutOnly**
- **RobustOutOnly**
- **OutIn**
- **OutOptionalIn**

これらのメッセージ交換パターンは、列挙型の定数 **org.apache.camel.ExchangePattern** によって表されます。

グループ化されたエクスチェンジ

複数のエクスチェンジインスタンスをカプセル化した 1 つのエクスチェンジがあると便利な場合もあります。これに対応する為、**グループ化されたエクスチェンジ** を使用できます。グループ化されたエクスチェンジは、基本的 **Exchange.GROUPED_EXCHANGE** エクスチェンジプロパティーに格納されている **Exchange** オブジェクトの **java.util.List** が含まれるエクスチェンジインスタンスです。グループ化されたエクスチェンジの使用例は、「Aggregator」 を参照してください。

プロセッサー

プロセッサー は、ルートを通過するエクスチェンジのストリームにアクセスして変更することができるルート内のノードです。プロセッサーは、動作を変更する **式** や **述語** の引数を取ることができます。たとえば、図1.1「ローカルルーティングルール」 で示されたルールには、**xpath()** 述語を引数とする **filter()** プロセッサーが含まれます。

式および述語

式 (文字列またはその他のデータ型への評価) および述語 (true または false への評価) は、組み込みプロセッサー型の引数として頻繁に発生します。たとえば、以下のフィルタールールは、ヘッダー **foo** の値が **bar** と等しい場合にのみ **In** メッセージを伝播します。

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

header("foo").isEqualTo("bar") ではフィルターが述語によって修飾されます。メッセージの内容に基づいてより高度な述語や式を作成するために、式言語および述語言語のいずれかを使用できます (パートII「ルーティング式と述語言語」 を参照)。

1.3. SPRING XML ファイルのルータースキーマ

名前空間

ルータースキーマは XML DSL を定義し、以下の XML スキーマ名前空間に属します。

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

スキーマの場所の指定

ルータースキーマの場所は通常、Apache Web サイトにある最新バージョンのスキーマを参照する **http://camel.apache.org/schema/spring/camel-spring.xsd** に指定されます。たとえば、Apache Camel Spring ファイルのルート **beans** 要素は、通常 例1.2「ルータースキーマの場所の指定」 のように設定されます。

例1.2 ルータースキーマの場所の指定

```
<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>
```

ランタイムスキーマの場所

実行時に、Apache Camel は Spring ファイルで指定されたスキーマの場所からルータースキーマをダウンロード**しません**。代わりに、Apache Camel は **camel-spring** JAR ファイルの root ディレクトリーからスキーマのコピーを自動的に取得します。これにより、Spring ファイルの解析に使用されるスキーマのバージョンが、常に現在のランタイムバージョンと一致するようになります。これは、Apache Web サイトに公開されているスキーマの最新バージョンが、現在使用しているランタイムのバージョンと一致しない場合があるため、重要になります。

XML エディターの使用

通常、フル機能の XML エディターを使用して Spring ファイルを編集することが推奨されます。XML エディターの自動補完機能により、ルータースキーマに準拠する XML の作成がはるかに容易になり、XML の形式が不適切な場合はエディターが即座に警告します。

通常、XML エディターは、**xsi:schemaLocation** 属性で指定した場所からスキーマをダウンロードすることに **依存しています**。編集中に正しいスキーマバージョンを使用するようにするため、通常は **camel-spring.xsd** ファイルの特定のバージョンを選択することが推奨されます。たとえば、Apache Camel バージョン 2.3 用の Spring ファイルを編集するには、以下のように beans 要素を修正します。

```
<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">
...
```

編集が完了したら、デフォルトの **camel-spring.xsd** に戻します。現在ダウンロードできるスキーマのバージョンを確認するには、Web ページ http://camel.apache.org/schema/spring に移動します。

1.4. エンドポイント

概要

Apache Camel エンドポイントは、ルートにおけるメッセージの水源 (蛇口) とシンク (流し台) です。エンドポイントは一般的なビルディングブロックと言えます。満たす必要がある唯一の要件は、メッセージの水源 (プロデューサーエンドポイント) またはメッセージのシンク (コンシューマーエンドポイント) のいずれかとして機能することです。そのため、Apache Camel でサポートされるエンドポイント型には、HTTP などのプロトコルをサポートするエンドポイントから、定期的な間隔でダミーメッセージを生成する Quartz などの単純なタイマーエンドポイントまで、非常に多様な種類があります。Apache Camel の大きな強みの 1 つは、新しいエンドポイント型を実装するカスタムコンポーネントを、比較的簡単に追加できることです。

エンドポイント URI

エンドポイントは、以下の一般的な形式を持つ **エンドポイント URI** で識別されます。

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

URI **スキームは** は、**http** などのプロトコルを識別し、**contextPath** はプロトコルによって解釈される URI の詳細を提供します。さらに、ほとんどのスキームでは、以下の形式で指定される **queryOptions** (クエリーオプション) を定義できます。

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

たとえば、以下の HTTP URI を使用して Google 検索エンジンページに接続できます。

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

以下のファイル URI を使用して、**C:\temp\src\data** ディレクトリー配下に表示されるすべてのファイルを読み取ることができます。

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

すべての **スキーム** がプロトコルを表しているわけではありません。**スキーム** は、タイマーなどの有用なユーティリティーへのアクセスのみを提供することもあります。たとえば、以下の Timer エンドポイント URI は、1 秒毎 (=1000 ミリ秒) にエクスチェンジを生成します。これを使用して、ルートでアクティビティーをスケジュールできます。

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

長いエンドポイント URI の使用

エンドポイント URI は、提供される設定情報がすべて付随することで、かなり長くなることがあります。Red Hat Fuse 6.2 以降では、長い URI での作業をより管理しやすくする方法が 2 つあります。

エンドポイントの個別設定

エンドポイントは個別に設定でき、ルートから短縮 ID を使用してエンドポイントを参照できます。

```
<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>
```

URI 内でオプションをいくつか設定し、**property** 属性を使用して追加オプションを指定することもできます (または URI からオプションを上書きすることもできます)。

```
<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>
```

複数行にまたがるエンドポイント設定の分割

改行を使用して URI 属性を分割できます。

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

注記

& で区切って、各行に 1 つ以上のオプションを指定できます。

URI での期間指定

Apache Camel コンポーネントの多くは、期間を指定するオプション (タイムアウト値など) を持ちます。デフォルトでは、このような期間を指定するオプションは通常、数値で指定され、ミリ秒単位として解釈されます。ただし、Apache Camel は期間のより読みやすい構文もサポートしており、時、分、秒で期間を表現できます。以下の構文に準拠する文字列を使用すると、人間が読みやすい期間を指定できます。

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

角括弧 **[]** に囲まれた各項は任意であり、表記法 **(A|B)** は、**A** および **B** のどちらかであることを示します。

たとえば、以下のように **timer** エンドポイントを 45 分間隔に設定できます。

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

また、以下のように、時、分、秒の単位を任意に組み合わせて使用することもできます。

```
from("timer:foo?period=1h15m")
```

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

URI オプションへの raw 値の指定

デフォルトでは、URI に指定するオプションの値は自動的に URI エンコードされます。場合によっては、これは望ましくない動作である場合があります。たとえば、password オプションを設定する場合は、URI エンコーディング **なし** で raw 文字列を送信することが推奨されます。

構文 **RAW(RawValue)** でオプションの値を指定して、URI エンコーディングをオフにすることができます。以下に例を示します。

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

この例では、パスワードの値はリテラル値 **se+re?t&23** として送信されます。

大文字と小文字を区別しない列挙オプション

一部のエンドポイント URI オプションは、Java **enum** 定数にマッピングされます。たとえば、Log コンポーネントの **level** オプションは、**INFO**、**WARN**、**ERROR** などの **enum** 値を取ることができます。この型変換は大文字と小文字を区別しないため、以下ののいずれかのオプションを使用して Log プロデューサーエンドポイントのログレベルを設定できます。

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

URI リソースの指定

Camel 2.17 以降、XSLT や Velocity などのリソースベースのコンポーネントは、**ref:** を接頭辞として使用して、レジストリーからリソースファイルをロードできます。

たとえば、レジストリーに格納された 2 つの Bean それぞれの ID である **ifmyvelocityscriptbean** および **mysimplescriptbean** は、以下のように参照することで、Bean の内容を使うことができます。

```
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 コンポーネント

各 URI **スキーム** は本質的にエンドポイントファクトリーとなる **Apache Camel コンポーネント** にマップされます。つまり、特定のタイプのエンドポイントを使用するには、対応する Apache Camel コン

ポーネントをランタイムコンテナーにデプロイする必要があります。たとえば、JMS エンドポイントを使用するには、コンテナーに JMS コンポーネントをデプロイします。

Apache Camel は、アプリケーションをさまざまなトランスポートプロトコルおよびサードパーティー製品と統合できる、多種多様なコンポーネントを提供します。たとえば、より一般的に使用されるコンポーネントには、File、JMS、CXF (Web サービス) 、HTTP、Jetty、Direct、および Mock などがあります。サポートされるコンポーネントの完全なリストは、Apache Camel コンポーネントのドキュメント を参照してください。

Apache Camel コンポーネントの大半は、Camel コアとは別にパッケージ化されています。Maven を使用してアプリケーションをビルドする場合、関連するコンポーネントアーティファクトの依存関係を追加するだけで、コンポーネント (およびサードパーティーの依存関係) を簡単にアプリケーションへ追加できます。たとえば、HTTP コンポーネントを含めるには、以下の Maven 依存関係をプロジェクトの POM ファイルに追加します。

```
<!-- 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>
```

以下のコンポーネントは Camel コア (**camel-core** アーティファクト内) に組み込まれているので、いつでも利用できます。

- Bean
- Browse
- Dataset
- Direct
- File
- Log
- Mock
- Properties
- Ref
- SEDA
- Timer
- VM

コンシューマーエンドポイント

コンシューマーエンドポイント は、ルートの **先頭** に (つまり **from()** DSL コマンドで) 表示されるエンドポイントです。言い換えると、コンシューマーエンドポイントはルート内の処理を開始するロールがあります。新しいエクスチェンジインスタンス (通常は受信または取得したメッセージを基に) を作成し、ルートの残りの部分でエクスチェンジを処理するためのスレッドを提供します。

たとえば、以下の JMS コンシューマーエンドポイントは **payments** キューからメッセージをプルし、ルートでメッセージを処理します。

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

または、Spring XML で同等のものを記述するとこのようになります。

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

コンポーネントには、 **コンシューマー専用** のものもあります。つまり、それらはコンシューマーエンドポイントの定義のみに使用できます。たとえば、Quartz コンポーネントは、コンシューマーエンドポイントを定義するためにのみ使用されます。以下の Quartz エンドポイントは、1秒ごと (1000 ミリ秒) にイベントを生成します。

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

必要に応じて、**fromF()** Java DSL コマンドを使用して、エンドポイント URI をフォーマットされた文字列として指定できます。たとえば、ユーザー名とパスワードを FTP エンドポイントの URI に指定するには、以下のように Java でルートを記述します。

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

最初の **%s** は文字列 **username** の値に置き換えられ、2 番目の **%s** は文字列 **password** の値に置き換えられます。この文字列のフォーマットメカニズムは **String.format()** で実装されており、C 言語の **printf()** 関数によって提供されるフォーマットに似ています。詳細は java.util.Formatter を参照してください。

プロデューサーエンドポイント

プロデューサーエンドポイント は、ルートの **途中** またはルートの **末尾** で表示されるエンドポイントです (例: **to()** DSL コマンド)。つまり、プロデューサーエンドポイントは既存のエクスチェンジオブジェクトを受け取り、エクスチェンジの内容を指定されたエンドポイントに送信します。

たとえば、以下の JMS プロデューサーエンドポイントは、現在のエクスチェンジの内容を指定の JMS キューにプッシュします。

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

または、Spring XML で同等のものを記述すると、このようになります。

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

一部のコンポーネントは **プロデューサー専用** で、プロデューサーエンドポイントを定義するためにのみ使用できます。たとえば、HTTP エンドポイントはプロデューサーエンドポイントを定義するためにのみ使用されます。

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

必要に応じて、**toF()** Java DSL コマンドを使用して、エンドポイント URI をフォーマットされた文字列として指定できます。たとえば、カスタム Google クエリーを HTTP URI に置き換えるには、以下のように Java でルートを記述します。

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

%s と書かれた箇所が、カスタムクエリー文字列 **myGoogleQuery** に置き換えられます。詳細は java.util.Formatter を参照してください。

1.5. プロセッサー

概要

ルーターが単にコンシューマーエンドポイントをプロデューサーエンドポイントに接続するだけでなく、より複雑なことを実行できるようにするため、**プロセッサー** をルートに追加することができます。プロセッサーは、ルーティングルールに挿入して、ルールを通過するメッセージの任意処理を実行するコマンドです。Apache Camel は、表1.1「Apache Camel プロセッサー」 に示されるように、さまざまなプロセッサーを提供します。

表1.1 Apache Camel プロセッサー

Java DSL	XML DSL	説明

Java DSL	XML DSL	説明
aggregate()	**aggregate**	「Aggregator」: 複数の受信エクスチェンジを単一のエクスチェンジに組み合わせるアグリゲーターを作成します。
aop()	**aop**	アスペクト指向プログラミング (AOP) を使用して、指定されたサブルートの前後で作業を行います。
bean(), beanRef()	**bean**	Java オブジェクト (または Bean) でメソッドを呼び出して、現在のエクスチェンジを処理します。「Bean インテグレーション」を参照してください。
choice()	**choice**	「Content-Based Router」: **when** および **otherwise** 句を使い、エクスチェンジの内容に基づいて特定のサブルートを選択します。
convertBodyTo()	**convertBodyTo**	In メッセージボディーを、指定された型に変換します。
delay()	**delay**	「Delayer」: ルートの後続へエクスチェンジを伝搬するのを遅延します。
doTry()	**doTry**	**doCatch**、**doFinally**、および **end** 句を使い、例外を処理するための try/catch ブロックを作成します。
end()	**該当なし**	現在のコマンドブロックを終了します。
enrich(),enrichRef()	**enrich**	「Content Enricher」: 現在のエクスチェンジと、指定された **プロデューサー** エンドポイント URI からリクエストされたデータを統合します。
filter()	**filter**	「Message Filter」: 述語式を使用して受信エクスチェンジをフィルタリングします。

Java DSL	XML DSL	説明
idempotentConsumer()	**idempotentConsumer**	「Idempotent Consumer」: 重複メッセージを抑制するストラテジーを実装します。
inheritErrorHandler()	**@inheritErrorHandler**	特定のルートノードで継承されたエラーハンドラーを無効にするために使用できるブール値オプション (Java DSL でサブ句として定義され、XML DSL の属性として定義) 。
inOnly()	**inOnly**	現在のエクスチェンジの MEP を **InOnly** (引数がない場合) に設定するか、指定されたエンドポイントへエクスチェンジを **InOnly** として送信します。
inOut()	**inOut**	現在のエクスチェンジの MEP を **InOut** (引数がない場合) に設定するか、指定されたエンドポイントへエクスチェンジを **InOut** として送信します。
loadBalance()	**loadBalance**	「Load Balancer」: エンドポイントのコレクションに対する負荷分散を実装します。
log()	**log**	コンソールにメッセージを記録します。
loop()	**loop**	「Loop」: 各エクスチェンジをルートの後続に繰り返し再送信します。
markRollbackOnly()	**@markRollbackOnly**	**(トランザクション)** 現在のトランザクションをロールバックオンリーにマークします (例外は発生しません)。XML DSL では、このオプションは **rollback** 要素にブール値属性として設定されます。Apache Karaf トランザクションガイド を参照してください。

Java DSL	XML DSL	説明
markRollbackOnlyLast()	**@markRollbackOnlyLast**	**(トランザクション)** 1つ以上のトランザクションがこのスレッドに関連付けられてから一時停止されている場合、このコマンドは最新のトランザクションをロールバックオンリーにマークします (例外は発生しません)。XML DSL では、このオプションは **rollback** 要素にブール値属性として設定されます。Apache Karaf トランザクションガイド を参照してください。
marshal()	**marshal**	指定されたデータフォーマットを使用して低レベルまたはバイナリーフォーマットに変換し、特定のトランスポートプロトコルで送信できるようにします。
multicast()	**multicast**	「Multicast」: 現在のエクスチェンジを複数の宛先にマルチキャストし、各宛先がエクスチェンジの独自のコピーを取得します。
onCompletion()	**onCompletion**	メインルートの完了後に実行されるサブルート (Java DSL **end()** で終了) を定義します。「OnCompletion」も併せて参照してください。
onException()	**onException**	指定された例外が発生するたびに実行されるサブルート (Java DSL **end()** で終了) を定義します。通常、ルート内ではなく、専用の行で定義されます。
pipeline()	**パイプライン**	「パイプとフィルター」: あるエンドポイントの出力が次のエンドポイントの入力となるように、一連のエンドポイントにエクスチェンジを送ります。「パイプライン処理」も併せて参照してください。
policy()	**policy**	現在のルートにポリシーを適用します (現時点ではトランザクションポリシーにのみ使用されます)。Apache Karaf Transaction Guide を参照してください。

Java DSL	XML DSL	説明
pollEnrich(),pollEnrichRef()	**pollEnrich**	「Content Enricher」: 現在のエクスチェンジと、指定された **コンシューマー** エンドポイント URI からポーリングされたデータを統合します。
process(),processRef	**process**	現在のエクスチェンジでカスタムプロセッサーを実行します。「カスタムプロセッサー」 および パートIII「高度な Camel プログラミング」 を参照してください。
recipientList()	**recipientList**	「受信者リスト」: エクスチェンジを、実行時に算出される (たとえば、ヘッダーの内容に基づいて) 受信者のリストに送信します。
removeHeader()	**removeHeader**	指定したヘッダーをエクスチェンジの In メッセージから削除します。
removeHeaders()	**removeHeaders**	指定したパターンに一致するヘッダーをエクスチェンジの In メッセージから削除します。パターンにはフォームを持たせることができます。**prefix*** を使用した場合、接頭辞で始まるすべての名前と一致します。それ以外の場合、正規表現として解釈されます。
removeProperty()	**removeProperty**	エクスチェンジから、指定したエクスチェンジプロパティーを削除します。

Java DSL	XML DSL	説明
removeProperties()	**removeProperties**	指定したパターンに一致するプロパティーをエクスチェンジから削除します。コンマで区切られた1つ以上の文字列のリストを引数として取ります。最初の文字列はパターンです (上記の **removeHeaders()** を参照)。後続の文字列は例外を指定します。これらのプロパティーは削除されません。
resequence()	**resequence**	「Resequencer」: 指定されたコンパレータの動作に基づき、受信エクスチェンジの順序を変更します。**バッチ** モードと **ストリーム** モードをサポートします。
rollback()	**rollback**	**(トランザクション)** 現在のトランザクションをロールバックオンリーにマークします (デフォルトでは例外も発生)。Apache Karaf トランザクションガイド を参照してください。
routingSlip()	**routingSlip**	「Routing Slip」: 任意のヘッダーから抽出したエンドポイント URI のリストに基づいて動的に構築されたパイプラインで、エクスチェンジをルーティングします。
sample()	**sample**	サンプリングスロットラーを作成し、ルート上のトラフィックからエクスチェンジのサンプルを抽出できるようにします。
setBody()	**setBody**	エクスチェンジの In メッセージのメッセージボディーを設定します。
setExchangePattern()	**setExchangePattern**	現在のエクスチェンジの MEP を指定された値に設定します。「メッセージ交換パターン」を参照してください。
setHeader()	**setHeader**	エクスチェンジの In メッセージに指定したヘッダーを設定します。

Java DSL	XML DSL	説明
setOutHeader()	**setOutHeader**	エクスチェンジの Out メッセージに指定したヘッダーを設定します。
setProperty()	**setProperty()**	指定したエクスチェンジプロパティーを設定します。
sort()	**sort**	In メッセージボディーの内容を並べ替えます (カスタムコンパレーターをオプションで指定できます) 。
split()	**split**	「Splitter」: 現在のエクスチェンジを一連のエクスチェンジに分割します。分割された各エクスチェンジには元のメッセージボディーの断片が含まれます。
stop()	**stop**	現在のエクスチェンジのルーティングを停止し、完了したとマークします。
threads()	**threads**	ルートの後続部分を並列に処理するためにスレッドプールを作成します。
throttle()	**throttle**	「Throttler」: フローレートを指定レベルに制限します (1 秒あたりのエクスチェンジ数) 。
throwException()	**throwException**	指定された Java 例外を出力します。
to()	**上記を以下のように変更します。**	エクスチェンジを 1 つ以上のエンドポイントに送信します。「パイプライン処理」を参照してください。
toF()	**該当なし**	文字列フォーマットを使用して、エクスチェンジをエンドポイントに送信します。つまり、エンドポイント URI 文字列は C 言語の **printf()** 関数の形式に置換し、埋め込むことができます。

Java DSL	XML DSL	説明
transacted()	**transacted**	ルートの後続部分を囲む Spring トランザクションスコープを作成します。Apache Karaf トランザクションガイド を参照してください。
transform()	**transform**	「メッセージトランスレーター」: **In** メッセージヘッダーを **Out** メッセージヘッダーにコピーし、**Out** メッセージボディーを指定された値に設定します。
unmarshal()	**unmarshal**	指定したデータフォーマットを使用して、**In** メッセージボディーを低レベルまたはバイナリー形式から高レベル形式に変換します。
validate()	**validate**	述語式を取り、現在のメッセージが有効かどうかを検証します。述語が **false** を返す場合は、**PredicateValidationException** 例外を出力します。
wireTap()	**wireTap**	「Wire Tap」: **ExchangePattern.InOnly** MEP を使用して、現在のエクスチェンジのコピーを指定された Wire tap URI に送信します。

サンプルプロセッサー

ルートでプロセッサーを使用する方法をある程度理解するには、以下の例を参照してください。

- Choice
- Filter
- Throttler
- Custom

Choice

choice() プロセッサーは、受信メッセージを別のプロデューサーエンドポイントにルーティングするために使用される条件文です。代替となる各プロデューサーエンドポイントの前には、述語の引数を取る **when()** メソッドがあります。述語が true の場合、後続のターゲットが選択されます。そうでない場

合、ルール内の次の **when()** メソッドに処理が進みます。たとえば、以下の **choice()** プロセッサーは **Predicate1** および **Predicate2** の値に応じて、受信メッセージを **Target1**、**Target2**、または **Target3** のいずれかに転送します。

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

または、Spring XML で同等のものを記述すると、このようになります。

```
<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>
```

Java DSL には、**endChoice()** コマンドを使用する必要がある可能性がある特殊なケースがあります。標準の Apache Camel プロセッサーの中には、特殊なサブ句を使用して追加のパラメーターを指定でき、通常は **end()** コマンドで終了する追加レベルのネストを効果的に展開します。たとえば、ロードバランサー句を **loadBalance().roundRobin().to("mock:foo").to("mock:bar").end()** として指定できます。これにより、**mock:foo** と **mock:bar** エンドポイント間でメッセージの負荷分散が行われます。ただし、ロードバランサー句が choice の条件に組み込まれている場合は、以下のように **endChoice()** コマンドを使用して句を終了する必要があります。

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

フィルター

filter() プロセッサーを使用すると、必要のないメッセージがプロデューサーエンドポイントに到達しないようにすることができます。述語の引数を 1 つ取ります。述語が true の場合、メッセージエクスチェンジはプロデューサーに対して許可されます。述語が false の場合、メッセージエクスチェンジはブ

ロックされます。たとえば、以下のフィルターは、受信メッセージに **bar** の値を持つヘッダー **foo** が含まれない限り、メッセージエクスチェンジをブロックします。

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

または、Spring XML で同等のものを記述すると、このようになります。

```
<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() プロセッサーは、プロデューサーエンドポイントがオーバーロードしないようにします。スロットラーは、1秒間に通過できるメッセージの数を制限することで機能します。受信メッセージが指定されたレートを超える場合、スロットラーは超過したメッセージをバッファーに蓄積し、ゆっくりとプロデューサーエンドポイントに送信します。たとえば、スループットのレートを毎秒 100 メッセージに制限するには、以下のルールを定義します。

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

または、Spring XML で同等のものを記述すると、このようになります。

```
<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>
```

カスタムプロセッサー

ここに記載されている標準プロセッサーがいずれも必要な機能を提供しない場合は、いつでも独自のカスタムプロセッサーを定義できます。カスタムプロセッサーを作成するには、**org.apache.camel.Processor** インターフェイスを実装し、**process()** メソッドを上書きするクラスを定義します。以下のカスタムプロセッサー **MyProcessor** は、受信メッセージから、ヘッダー **foo** を削除します。

例1.3 カスタムプロセッサークラスの実装

```
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");
```

```
    }
  }
};
```

カスタムプロセッサーをルータールールに挿入するには、**process()** メソッドを呼び出します。このメソッドは、ルールにプロセッサーを挿入するための一般的なメカニズムを提供します。たとえば、以下のルールは、例1.3「カスタムプロセッサークラスの実装」 で定義されたプロセッサーを呼び出します。

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

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

第2章 ルート構築の基本原則

概要

Apache Camel は、ルートの中で繋ぎ合わせられる複数のプロセッサーおよびコンポーネントを提供します。本章では、提供されるビルディングブロックを使用してルートを構築する原則を説明します。

2.1. パイプライン処理

概要

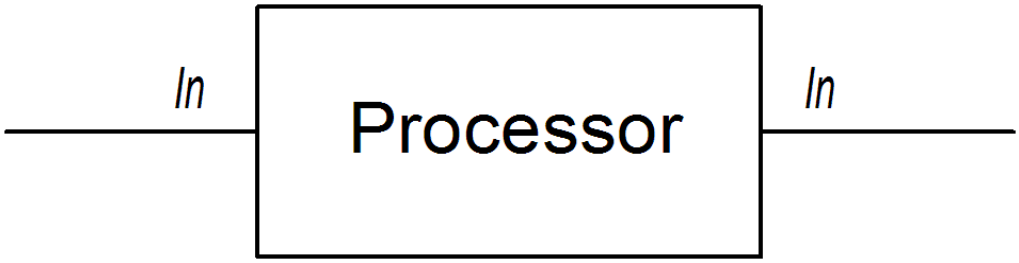
Apache Camel では、パイプライン処理はルート定義でノードを接続するための主要なパラダイムです。パイプラインの概念は、おそらく UNIX オペレーティングシステムのユーザーには最も馴染のあるものでしょう。オペレーティングシステムのコマンドを結合するために使用しているからです。たとえば、**ls | more** はディレクトリー一覧 **ls** をページスクロールのユーティリティー **more** にパイプするコマンドの例です。パイプラインの基本的な概念は、あるコマンドの **出力** が次の **入力** に読み込まれることです。ルートにおいては、あるプロセッサーからの **Out** メッセージが次のプロセッサーの **In** メッセージにコピーされることに相当します。

プロセッサーノード

最初のエンドポイントを除き、ルートのすべてのノードは **プロセッサー** で、**org.apache.camel.Processor** インターフェイスを継承します。つまり、プロセッサーが DSL ルートの基本ビルディングブロックを設定します。たとえば、**filter()**、**delayer()**、**setBody()**、**setHeader()**、および **to()** などの DSL コマンドはすべてプロセッサーを表します。プロセッサーがどのように接続しあってルートを設定するかを考えるときに、2 つの異なる処理アプローチを区別することが重要になります。

最初のアプローチは、図2.1「In メッセージを変更するプロセッサー」 にあるように、プロセッサーが単純にエクスチェンジの **In** メッセージを変更する方法です 。この場合、エクスチェンジの **Out** メッセージは **null** のままになります。

図2.1 In メッセージを変更するプロセッサー

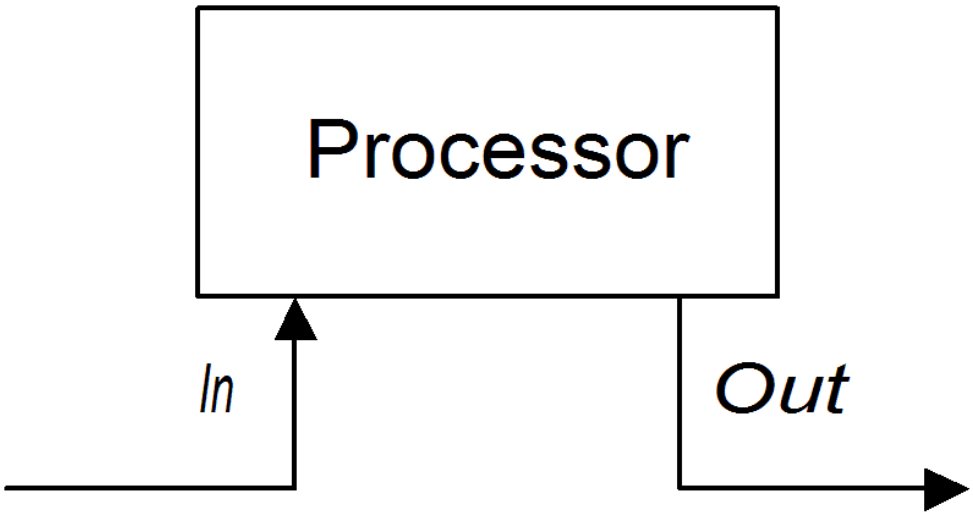


以下のルートは、**BillingSystem** ヘッダーを追加 (または変更) して現在の **In** メッセージを修正する **setHeader()** コマンドを示しています。

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

2 つ目のアプローチは、図2.2「Out メッセージを作成するプロセッサー」 にあるように、プロセッサーが処理の結果として **Out** メッセージを作成する方法です。

図2.2 Out メッセージを作成するプロセッサー



以下のルートは、文字列 **DummyBody** が含まれるメッセージボディーを持った **Out** メッセージを作成する **transform()** コマンドを示しています。

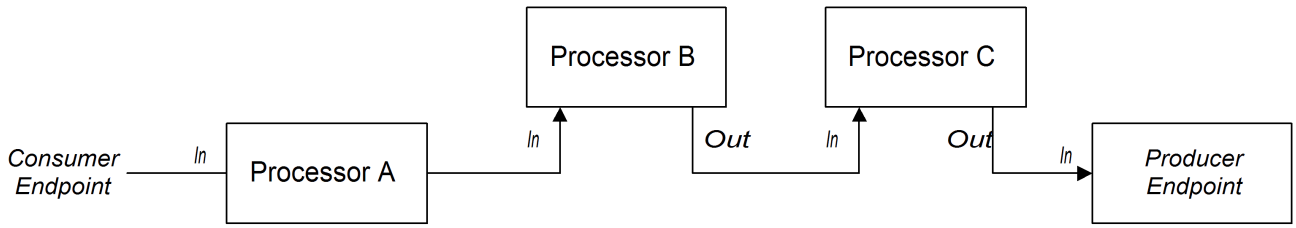
```
from("activemq:orderQueue")
    .transform(constant("DummyBody"))
    .to("activemq:billingQueue");
```

constant("DummyBody") は定数式を表します。**transform()** の引数の型は式である必要があるため、文字列 **DummyBody** を直接渡すことはできません。

InOnly エクスチェンジのパイプライン

図2.3「InOnly エクスチェンジのサンプルパイプライン」 は、 **InOnly** エクスチェンジのプロセッサーパイプラインの例を示しています。プロセッサー A は **In** メッセージを変更し、プロセッサー B および C は **Out** メッセージを作成します。ルートビルダーが、図に示されているようにプロセッサー同士を繋ぎ合わせます。特にプロセッサー B と C は **パイプライン** の形式で繋がれています。つまり、エクスチェンジをプロセッサー C に読み込ませる前にプロセッサー B の **Out** メッセージが **In** メッセージに移動され、エクスチェンジをプロデューサーエンドポイントに読み込ませる前にプロセッサー C の **Out** メッセージが **In** メッセージに移動されています。したがって、図2.3「InOnly エクスチェンジのサンプルパイプライン」 に示されるようにプロセッサーの出力と入力は連続したパイプラインに結合されています。

図2.3 InOnly エクスチェンジのサンプルパイプライン



Apache Camel はデフォルトでパイプラインパターンを使用するため、特別な構文を使用してルートにパイプラインを作成する必要はありません。たとえば、以下のルートは **userdataQueue** キューからメッセージを取り出し、(テキスト形式で顧客アドレスを生成するために) Velocity テンプレートを通してメッセージをパイプ処理し、生成されたテキストアドレスをキュー **envelopeAddresses** に送信します。

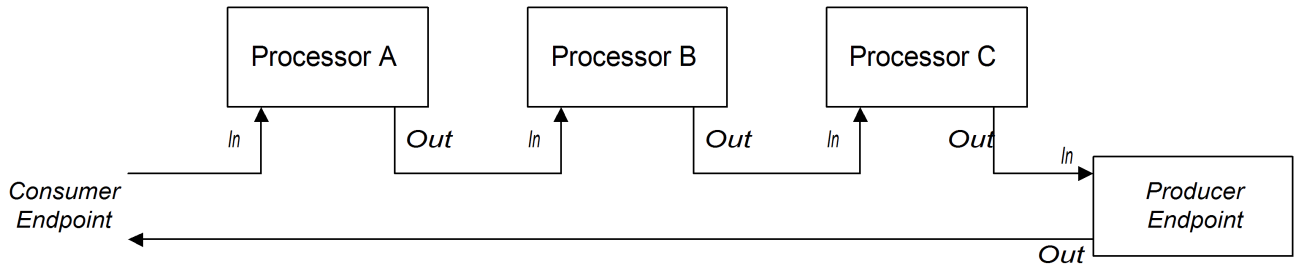
```
from("activemq:userdataQueue")
    .to(ExchangePattern.InOut, "velocity:file:AdressTemplate.vm")
    .to("activemq:envelopeAddresses");
```

Velocity エンドポイントである **velocity:file:AddressTemplate.vm** は、ファイルシステム内の Velocity テンプレートファイルの **file:AddressTemplate.vm** の場所を指定します。この **to()** コマンドは、エクスチェンジを Velocity エンドポイントに送信する前に、エクスチェンジパターンを**InOut** に変更し、その後 **InOnly** に戻します。Velocity エンドポイントの詳細は、**Apache Camel Component Reference Guide** の **Velocity** を参照してください。

InOut エクスチェンジのパイプライン

図2.4「InOut エクスチェンジのサンプルパイプライン」 は、通常リモートプロシージャーコール (RPC) のセマンティクスをサポートするときに使用する、 **InOut** エクスチェンジ用のプロセッサーパイプラインの例を示しています。プロセッサー A、B、および C はパイプライン形式で結合され、各プロセッサーの出力が次の入力に読み込まれます。プロデューサーエンドポイントによって生成された最後の **Out** メッセージは、コンシューマーエンドポイントまで遡って、元のリクエストへの返信として提供されます。

図2.4 InOut エクスチェンジのサンプルパイプライン



InOut 交換パターンをサポートするには、ルートの最後のノード (プロデューサーエンドポイントかその他の種類のプロセッサーかに限らず) が **Out** メッセージを作成することが **必須** です。そうでない場合は、コンシューマーエンドポイントに接続するクライアントがハングし、リプライメッセージを無期限に待ち続けることになります。すべてのプロデューサーエンドポイントが **Out** メッセージを作成するわけではないことに注意してください。

受信 HTTP リクエストを処理し、支払い要求を処理する以下のルートを見てみましょう。

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

受信支払い要求は、Web サービスのパイプライン、**cxf:bean:addAccountDetails**、**cxf:bean:getCreditRating**、および **cxf:bean:processTransaction** を介して渡すことで処理されます。最後の Web サービス **processTransaction** が生成する応答 (**Out** メッセージ) は、Jetty エンドポイント経由で返信されます。

パイプラインがエンドポイントのシーケンスで設定される場合は、以下の代替構文を使用することもできます。

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

InOptionalOut エクスチェンジのパイプライン

InOptionalOut エクスチェンジのパイプラインは、図2.4「InOut エクスチェンジのサンプルパイプライン」 のパイプラインと基本的に同じです。**InOut** と **InOptionalOut** の相違点は、 **InOptionalOut** 交換パターンのエクスチェンジが応答として null **Out** メッセージを使用することが可能であることです。つまり、**InOptionalOut** エクスチェンジの場合、**nullOut** メッセージが、パイプラインの次のノードの **In** メッセージにコピーされます。一方、**InOut** エクスチェンジの場合、**nullOut** メッセージは破棄され、代わりに現在のノードの元の **In** メッセージが、次のノードの **In** メッセージにコピーされます。

2.2. 複数の入力

概要

標準的なルートは、Java DSL の **from(EndpointURL)** 構文を使用して、1 つのエンドポイントから入力を受け取ります。しかし、ルートに複数の入力を定義する必要がある場合はどうすればよいでしょうか。Apache Camel では、ルートに複数の入力を指定する複数の方法があります。選択肢は、エクスチェンジを独立して処理するか、または異なる入力からのエクスチェンジを何らかの方法で組み合わせるか (この場合は、「Content Enricher パターン」 を使います) によって異なります。

複数の独立した入力

複数の入力を指定する最も簡単な方法は、**from()** DSL コマンドのマルチ引数形式を使用することです。以下に例を示します。

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

または、以下の同等の構文を使用できます。

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

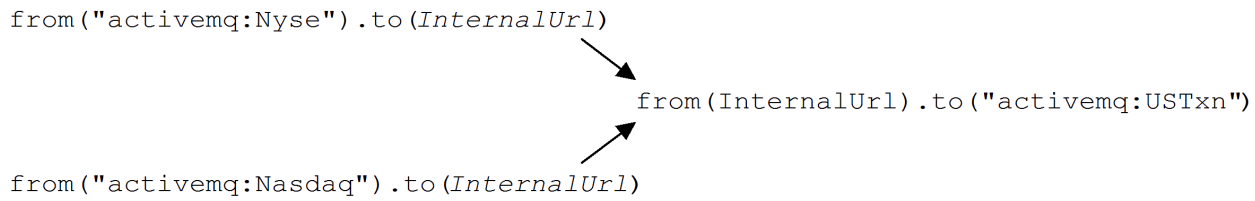
これらの両方の例で、各入力エンドポイント **URI1**、**URI2** 、および **URI3** からのエクスチェンジは、相互に独立に、別個のスレッドで処理されます。実際、上記のルートは以下の 3 つに分かれたルートと同等であると考えることができます。

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

セグメント化されたルート

たとえば、2 つの異なるメッセージングシステムからの受信メッセージをマージし、同じルートを使用して処理する場合があります。ほとんどの場合、図2.5「セグメント化されたルートによる複数入力の処理」 に示されるように、ルートをセグメントに分割して複数の入力に対応できます。

図2.5 セグメント化されたルートによる複数入力の処理



ルートの最初のセグメントは、たとえば **activemq:Nyse** や **activemq:Nasdaq** といったいくつかの外部キューから入力を取得し、その受信エクスチェンジを内部エンドポイント **InternalUrl** に送信します。2 つ目のルートセグメントは、受信エクスチェンジを内部エンドポイントから取得し、宛先キュー **activemq:USTxn** に送信することで、受信エクスチェンジをマージします。**InternalUrl** は、ルーターのアプリケーション **内** でのみ使用することが意図されたエンドポイントの URL です。以下のタイプのエンドポイントが内部使用に適しています。

- Direct エンドポイント
- SEDA エンドポイント
- VM エンドポイント

これらのエンドポイントの主な目的は、ルートの異なるセグメントをまとめることにあります。これらはすべて、複数の入力を単一のルートにマージする効果的な方法を提供します。

Direct エンドポイント

direct コンポーネントは、複数のルートを繋ぎ合わせる最も簡単なメカニズムを提供します。direct コンポーネントのイベントモデルは **同期型** であり、ルートの後続のセグメントは最初のセグメントと同じスレッドで実行されます。direct URL の一般的な形式は **direct:EndpointID** です。エンドポイント ID である **EndpointID** は、エンドポイントのインスタンスを識別する一意の英数字の文字列です。

たとえば、2 つのメッセージキュー **activemq:Nyse** と **activemq:Nasdaq** から入力を受け取り、それらを単一のメッセージキュー **activemq:USTxn** にマージする場合、以下のルートセットを定義することで実行できます。

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

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

最初の 2 つのルートはメッセージキュー **Nyse** と **Nasdaq** から入力を受け取り、それらをエンドポイント **direct:mergeTxns** に送信します。最後のキューは、前の 2 つのキューからの入力を組み合わせ、組み合わせたメッセージストリームを **activemq:USTxn** キューに送信します。

direct エンドポイントの実装は、以下のように動作します。エクスチェンジがプロデューサーエンドポイント (例: **to("direct:mergeTxns")**) に到達するたびに、direct エンドポイントは、同じエンドポイント ID (例: **from("direct:mergeTxns")**) を持つすべてのコンシューマーエンドポイントに直接エクスチェンジを渡します。direct エンドポイントは、同じ Java 仮想マシン (JVM) インスタンス内の同じ **CamelContext** に属するルート間の通信にのみ使用できます。

SEDA エンドポイント

SEDA コンポーネントは、複数のルートを繋ぎ合わせるもう 1 つのメカニズムを提供します。これは direct コンポーネントと同様の使い方ができますが、以下のように基盤となるイベントとスレッドのモデルが異なります。

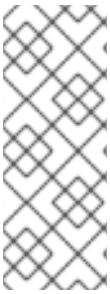
- SEDA エンドポイントの処理は同期され**ません**。つまり、エクスチェンジを SEDA プロデューサーエンドポイントに送信すると、ルート内の前のプロセッサーに制御がすぐに戻されます。
- SEDA エンドポイントはキューバッファー (**java.util.concurrent.BlockingQueue** 型) を持ち、次のルートセグメントによって処理される前の受信エクスチェンジをすべて格納しています。
- 各 SEDA コンシューマーエンドポイントは、ブロッキングキューからのエクスチェンジオブジェクトを処理するためにスレッドプール (デフォルトのサイズは 5) を作成します。

- SEDA コンポーネントは、**競合コンシューマー (competing consumers)** パターンをサポートします。これは、特定のエンドポイントに複数のコンシューマーが接続している場合でも、各受信エクスチェンジが 1 度だけ処理されることを保証するものです。

SEDA エンドポイントを使用する主な利点の 1 つは、組み込みのコンシューマースレッドプールにより、ルートの応答性が向上することです。株式取引の例は、以下のように、direct エンドポイントの代わりに SEDA エンドポイントを使用するように書き換えられます。

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

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

この例と direct の例の主な相違点は、SEDA を使用する場合、2 番目のルートセグメント (**seda:mergeTxns** から **activemq:USTxn**) が 5 つのスレッドのプールで処理される点です。



注記

SEDA は単にルートセグメントを繋ぎ合わせるだけではありません。段階的イベント駆動型アーキテクチャー (staged event-driven architecture、SEDA) は、より管理しやすいマルチスレッドアプリケーションを構築するための設計哲学を含んでいます。Apache Camel の SEDA コンポーネントの目的は、この設計哲学をアプリケーションに適用できるようにすることです。SEDA の詳細は、http://www.eecs.harvard.edu/~mdw/proj/seda/ を参照してください。

VM エンドポイント

VM コンポーネントは SEDA エンドポイントと非常に似ています。唯一の違いは、SEDA コンポーネントが同じ **CamelContext** 内のルートセグメントの繋ぎ合わせに限定されるのに対し、VM コンポーネントでは、同じ Java 仮想マシン内で実行されている限り、異なる Apache Camel アプリケーションからのルートを繋ぎ合わせられることです。

株式取引の例は、以下のように、 SEDA エンドポイントの代わりに VM エンドポイントを使用するように書き換えられます。

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

そして、別のルーターアプリケーション (同じ Java 仮想マシンで実行されている) において、以下のようにルートの 2 つ目のセグメントを定義できます。

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

Content Enricher パターン

Content Enricher パターンは、これまでと根本的に異なる方法でルートへの複数入力の処理を定義します。エクスチェンジが Enricher プロセッサーに入ると、Enricher は外部リソースにアクセスして情報を取得し、その情報を元のメッセージに追加します。このパターンでは、外部リソースが実質的にメッセージへの 2 つ目の入力を表しています。

たとえば、信用リクエストを処理するアプリケーションを作成している場合に、信用リクエストを処理する前に、それを顧客に対して信用格付けを割り当てるデータ (格付けデータはディレクトリー **src/data/ratings** のファイルに格納されている) で拡張する必要があります。以下のよう

に、**pollEnrich()** パターンと **GroupedExchangeAggregationStrategy** 集約ストラテジーを使用して、受信信用リクエストと格付けファイルのデータを組み合わせることができます。

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

GroupedExchangeAggregationStrategy クラスは、**org.apache.camel.processor.aggregate** パッケージの標準集約ストラテジーで、各新しいエクスチェンジを **java.util.List** インスタンスに追加し、生成されるリストを **Exchange.GROUPED_EXCHANGE** エクスチェンジプロパティーに保存します。この場合、リストには、(**creditRequests** JMS キューからの) 元のエクスチェンジと (file エンドポイントからの) Enricher エクスチェンジの 2 つの要素が含まれます。

グループ化されたエクスチェンジにアクセスするには、以下のようなコードを使用します。

```
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
        ...
    }
}
```

このアプリケーションへの別のアプローチとしては、データをマージするコードをカスタム集約ストラテジークラスに直接実装することが考えられます。

Content Enricher パターンの詳細は、「Content Enricher」 を参照してください。

2.3. 例外処理

概要

Apache Camel はいくつかの異なるメカニズムを提供しており、異なるレベルの粒度で例外を処理することができます。まず、**doTry**、**doCatch**、および **doFinally** を使用してルート内で例外を処理できます。また、**onException** を使用して、各例外型に対して実行するアクションを指定し、**RouteBuilder** 内のすべてのルートにそのルールを適用することもできます。または、**errorHandler** を使用して、**すべての** 例外型に対して実行するアクションを指定し、そのルールを **RouteBuilder** 内のすべてのルートに適用することもできます。

例外処理の詳細は、「Dead Letter Channel」 を参照してください。

2.3.1. onException 句

概要

onException 句は、1 つ以上のルートで発生する例外をトラップするための強力なメカニズムです。これは型固有のもので、異なる例外型を処理するための個別のアクションを定義することができます。基

本的にルートと同じ (実際には、若干拡張された) 構文でアクションを定義できるため、例外を処理する方法にかなりの柔軟性が得られます。また、トラップモデルをベースにしていることにより、1 つの **onException** 句で任意のルート内の任意のノードで発生した例外を処理できます。

onException を使用した例外のトラップ

onException 句は、例外をキャッチするのではなく、**トラップ** するメカニズムです。つまり、一度 **onException** 句を定義すると、ルート内の任意の地点で発生する例外がトラップされます。これは、特定のコードフラグメントが try ブロックで **明示的** に囲まれている場合にのみ例外がキャッチされる、Java の try/catch メカニズムとは対照的です。

onException 句を定義すると、Apache Camel ランタイムが各ルートノードを暗黙的に try ブロックで囲んでしまいます。このため、**onException** 句はルートの任意の地点で例外をトラップすることができます。ただし、このラッピングは自動的に行われ、ルート定義には表示されません。

Java DSL の例

以下の Java DSL の例では、**onException** 句は **RouteBuilder** クラスで定義されているすべてのルートに適用されます。いずれかのルート (**from("seda:inputA")** または **from("seda:inputB")**) の処理中に **ValidationException** 例外が発生すると、**onException** 句はその例外をトラップし、現在のエクスチェンジを **validationFailed** JMS キュー (デッドレターキューとして機能する) にリダイレクトします。

```
// 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 の例

上記の例は、exception 句を定義する **onException** 要素を使用して、以下のように XML DSL で表すこともできます。

```
<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>
```

複数の例外のトラップ

複数の **onException** 句を定義して、**RouteBuilder** スコープ内で例外をトラップすることができます。これにより、例外に応じて異なるアクションを実行できます。たとえば、以下の Java DSL で定義された一連の **onException** 句は、**ValidationException**、**IOException**、および **Exception** の異なるデッドレター宛先を定義します。

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

以下のように、XML DSL で同じ一連の **onException** 句を定義することができます。

```
<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>
```

また、複数の例外をグループ化して、同じ **onException** 句でトラップすることもできます。Java DSL では、以下のように複数の例外をグループ化できます。

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

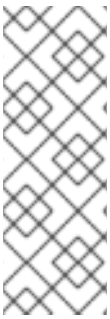
XML DSL では、以下のように **onException** 要素内に複数の **exception** 要素を定義することで、複数の例外をグループ化できます。

```
<onException>
    <exception>com.mycompany.ValidationException</exception>
    <exception>com.mycompany.BuesinessException</exception>
```

```
    <to uri="activemq:validationFailed"/>
</onException>
```

複数の例外をトラップする場合、**onException** 句の順序は重要です。Apache Camel はまず、発生した例外を**最初**の句に対して一致しようと試みます。最初の句が一致しない場合、次の **onException** 句が試行され、一致するものが見つかるまで続きます。各々の一致するかどうかの試行は、以下のアルゴリズムで制御されます。

1. 発生する例外が チェーン例外 (例外がキャッチされて別の例外として出力されたもの) である場合、最もネストされた例外型が最初に一致の基準となります。この例外は、以下のようにテストされます。
   a. テスト対象例外が正確に **onException** 句で指定された型を持っている場合 (**instanceof** によってテストされる) は、一致が起こります。
   b. テスト対象例外が **onException** 句で指定された型のサブタイプである場合、一致が起こります。
2. 最もネストされた例外が一致しなかった場合、チェーンの次の例外 (ラップしている例外) がテストされます。このテストは、一致が起こるかチェーンの最後に到達するまで継続します。



注記

throwException EIP を使用すると、Simple 言語の式から新しい例外インスタンスを生成できます。現在のエクスチェンジから利用可能な情報に基づいて、動的に生成することができます。以下に例を示します。

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

デッドレターチャネル

これまでの基本的な **onException** の使用例は、すべて **デッドレターチャネル** パターンを利用していました。つまり、**onException** 句が例外をトラップすると、現在のエクスチェンジは特別な宛先 (デッドレターチャネル) にルーティングされます。デッドレターチャネルは、処理されて **いない** 失敗したメッセージの保持領域として機能します。管理者は後でメッセージを検査し、どのようなアクションを取る必要があるかを決定できます。

チャネルパターンの詳細は、「Dead Letter Channel」 を参照してください。

元のメッセージの使用

ルートの途中で例外が発生した時点では、エクスチェンジ内のメッセージが大幅に変更されている可能性があります (人間には判読できなくなっている場合もあります)。多くの場合、管理者にとっては、デッドレターキューに表示されるメッセージがルートの開始時に受信したままの **元** のメッセージであれば、どのような修正アクションをとるべきか決定するのが簡単になります。**useOriginalMessage** オプションはデフォルトでは **false** に設定されますが、エラーハンドラーに設定されている場合には自動的に有効になります。

注記

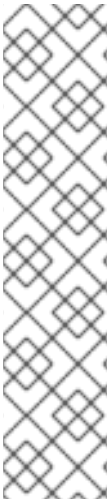
useOriginalMessage オプションは、メッセージを複数のエンドポイントに送信する Camel ルートに適用したり、メッセージを分割したりすると、予期せぬ動作をすることがあります。中間処理ステップが元のメッセージを変更する Multicast、Splitter、または RecipientList のルートでは、元のメッセージは保持されない場合があります。

Java DSL では、エクスチェンジのメッセージを元のメッセージで置き換えることができます。**setAllowUseOriginalMessage()** を **true** に設定し、以下のように **useOriginalMessage()** DSL コマンドを使用します。

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

XML DSL では、以下のように **onException** 要素の **useOriginalMessage** 属性を設定することで、元のメッセージを取得できます。

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



注記

setAllowUseOriginalMessage() オプションが **true** に設定されている場合、Camel はルートの開始時に元のメッセージのコピーを作成します。これにより、**useOriginalMessage()** の呼び出し時に元のメッセージが利用できることを保証します。しかし、**setAllowUseOriginalMessage()** オプションが Camel コンテキストで **false** (デフォルト) に設定されている場合、元のメッセージにはアクセス **できず**、**useOriginalMessage()** を呼び出すことができません。

デフォルトの動作がこうなっている理由は、大きなメッセージを処理する際にパフォーマンスを最適化するためです。

2.18 より前の Camel バージョンでは、**allowUseOriginalMessage** のデフォルト設定は true です。

再配信ポリシー

例外が発生したらすぐにメッセージの処理を中断して諦める代わりに、Apache Camel では例外が発生した時点でメッセージを **再送** するオプションを利用できます。タイムアウトが発生したり、一時的な障害が発生したりするネットワークシステムでは、元の例外が発生してからすぐに再送することで、失敗したメッセージが正常に処理されることがよくあります。

Apache Camel の再配信は、例外の発生後にメッセージを再送するさまざまなストラテジーをサポートします。再配信を設定する際に最も重要なオプションには、以下のものがあります。

maximumRedeliveries()

再配信を試行できる最大回数を指定します (デフォルトは **0**)。負の値は、再配信がいつまでも試行されることを意味します (無限の値と同等です) 。

retryWhile()

Apache Camel が再配信を続行すべきかどうかを決定する述語 (**Predicate** 型) を指定します。述語が現在のエクスチェンジ上で **true** と評価されると、再配信が試行されます。そうでない場合は再配信が停止され、それ以上の再配信の試みは行われません。
このオプションは **maximumRedeliveries()** オプションよりも優先されます。

Java DSL では、再配信ポリシーのオプションは、**onException** 句内の DSL コマンドを使用して指定します。たとえば、以下のように、エクスチェンジが **validationFailed** デッドレターキューに送信される前に、最大 6 回の再配信を指定できます。

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

XML DSL では、**redeliveryPolicy** 要素に属性を設定することで再配信ポリシーオプションを指定します。たとえば、上記のルートは以下のように XML DSL で表現できます。

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

再配信オプションが設定された後のルートの後半部分は、最後の再配信の試みが失敗するまで処理されません。すべての再配信オプションの詳細については、「Dead Letter Channel」 を参照してください。

もう 1 つの方法として、**redeliveryPolicyProfile** インスタンスで再配信ポリシーオプションを指定することもできます。その後、**onException** 要素の **redeliverPolicyRef** 属性を使用して、**redeliveryPolicyProfile** インスタンスを参照できます。たとえば、上記のルートは以下のように表現できます。

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

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

注記

複数の **onException** 句で同じ再配信ポリシーを再利用する場合は、**redeliveryPolicyProfile** を使用するアプローチが便利です。

条件付きトラップ

onWhen オプションを指定することで、**onException** による例外のトラップを条件付きにすることができます。**onException** 句で **onWhen** オプションを指定すると、発生した例外が句と一致し、**かつ**、**onWhen** 述語が現在のエクスチェンジで **true** に評価された場合にのみ一致が起こります。

たとえば、以下の Java DSL フラグメントでは、発生する例外が **MyUserException** に一致し、**user** ヘッダーが現在のエクスチェンジで null でない場合にのみ、最初の **onException** 句が実行されます。

```
// 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);
```

上記の **onException** 句は、以下のように XML DSL で表現できます。

```
<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>
```

例外の処理

デフォルトでは、ルートの途中で例外が発生すると、現在のエクスチェンジの処理が中断され、発生した例外がルート先頭のコンシューマーエンドポイントに伝播されます。**onException** 句がトリガーされても、発生した例外が伝播される前に **onException** 句がいくつかの処理を実行することを除き、この動作は基本的に同じです。

しかし、このデフォルトの動作が例外を処理する唯一の方法では**ありません**。以下のように、**onException** には例外処理の動作を変更するさまざまなオプションが用意されています。

- 例外再出力の抑制 - **onException** 句が完了した後に、再出力された例外を抑制するオプションがあります。つまり、この場合、例外はルート先頭のコンシューマーエンドポイントまで伝播**しません**。
- 継続的な処理 - 例外が発生した時点からエクスチェンジの通常の処理を再開するオプションがあります。このアプローチでは、暗黙的に例外の再出力も抑制されます。
- レスポンスの送信 - ルート先頭にあるコンシューマーエンドポイントがリプライを期待する (つまり **InOut** MEP を持つ) 特別なケースでは、例外をコンシューマーエンドポイントに伝播するのではなく、カスタムのフォールトリプライメッセージを作成する場合があります。

例外再出力の抑制

現在の例外が再出力され、コンシューマーエンドポイントに伝播されないようにするには、以下のように Java DSL で **handled()** オプションを **true** に設定します。

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

Java DSL では、**handled()** オプションの引数はブール型、**Predicate** 型、または **Expression** 型のいずれかを取ります (非ブール型の式は、それが非 null 値として評価された場合には **true** と解釈されます)。

以下のように **handled** 要素を使用して、XML DSL で同じルートを設定して再出力した例外を抑制できます。

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

処理の継続

例外が最初に発生したルート内のポイントから現在のメッセージの処理を続行するには、以下のように Java DSL で **continued** オプションを **true** に設定します。

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

Java DSL では、**continued()** オプションの引数はブール型、**Predicate** 型、または **Expression** 型のいずれかを取ります (非ブール型の式は、それが非 null 値として評価された場合には **true** と解釈されます)。

以下のように **continued** 要素を使用して、XML DSL で同じルートを設定できます。

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

レスポンスの送信

ルートを開始するコンシューマーエンドポイントがリプライを期待している場合、単に発生した例外をコンシューマーに伝播するのではなく、カスタムのフォールトリプライメッセージを作成する場合があります。この場合、2 つのステップが必要になります。まず、**handled** オプションを使用して再出力例外を抑制し、次にエクスチェンジの **Out** メッセージスロットにカスタムのフォールトメッセージを設定します。

たとえば、以下の Java DSL フラグメントは、**MyFunctionalException** 例外が発生するたびに、テキスト文字列 **Sorry** を含むリプライメッセージを送信する方法を示しています。

```
// 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");
```

クライアントにフォールトレスポンスを送信する場合、例外メッセージのテキストをレスポンスに組み

込みたいことがよくあります。**exceptionMessage()** ビルダーメソッドを使用して、現在の例外メッセージのテキストにアクセスできます。たとえば、以下のように **MyFunctionalException** 例外が発生するたびに、例外メッセージのテキストのみを含むリプライを送信できます。

```
// 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());
```

例外メッセージのテキストは、Simple 言語からも **exception.message** 変数を介してアクセスできます。たとえば、以下のように現在の例外のテキストをリプライメッセージに埋め込むことができます。

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

上記の **onException** 句は、以下のように XML DSL で表現できます。

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

例外処理中に発生した例外

既存の例外の処理中に発生した例外 (つまり、**onException** 句の処理中に発生した例外) は、特別な方法で処理されます。このような例外は、特別なフォールバック例外ハンドラーによって処理されます。例外は以下のように処理されます。

- 既存の例外ハンドラーはすべて無視され、処理は直ちに失敗します。
- 新しい例外がログに記録されます。
- 新しい例外がエクスチェンジオブジェクトに設定されます。

このシンプルな戦略は、**onException** 句が無限ループに閉じ込められるような複雑な障害のシナリオを回避します。

スコープ

OnException 句は、以下のスコープのいずれかで有効になります。

- RouteBuilder scope: **RouteBuilder.configure()** メソッド内で独立した文として定義された **onException** 句は、その **RouteBuilder** インスタンスで定義されたすべてのルートに影響しま

す。一方、これらの **onException** 句は他の **RouteBuilder** インスタンス内で定義されたルートに対する **影響はありません**。**onException** 句は、ルート定義の前に表示する **必要があります**。
この時点までのすべての例は、**RouteBuilder** スコープを使用して定義されます。

- **Route スコープ** - **onException** 句をルート内に直接埋め込むこともできます。onException 句は、それらが定義されているルートに **のみ** 影響します。

Route スコープ

ルート定義内のどこにでも **onException** 句を埋め込むことができますが、**end()** DSL コマンドを使用して埋め込んだ **onException** 句を終了する必要があります。

たとえば、以下のように Java DSL で埋め込み **onException** 句を定義できます。

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

XML DSL では、埋め込み **onException** 句を以下のように定義できます。

```
<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. エラーハンドラー

概要

errorHandler() 句は、このメカニズムが異なる例外型を識別 **できない** 点を除いて、**onException** 句と同様の機能を提供します。**errorHandler()** 句は、Apache Camel が提供する元々の例外処理メカニズムで、**onException** 句が実装される前から利用可能でした。

Java DSL の例

errorHandler() 句は **RouteBuilder** クラスで定義され、その **RouteBuilder** クラスのすべてのルートに

適用されます。これは、該当するルートのいずれかで例外が **その種類に関わらず** 発生するたびに実行されます。たとえば、失敗したすべてのエクスチェンジを ActiveMQ の **deadLetter** キューにルーティングするエラーハンドラーを定義するには、以下のように **RouteBuilder** を定義します。

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

ただし、デッドレターチャネルへのリダイレクトは、再配信の試行が終了するまで発生しません。

XML DSL の例

XML DSL では、**errorHandler** 要素を使用して、**camelContext** スコープ内にエラーハンドラーを定義します。たとえば、失敗したすべてのエクスチェンジを ActiveMQ **deadLetter** キューにルーティングするエラーハンドラーを定義するには、以下のように **errorHandler** 要素を定義します。

```
<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>
```

エラーハンドラーの種類

表2.1「エラーハンドラーの種類」 では、定義可能なさまざまな種類のエラーハンドラーの概要を説明します。

表2.1 エラーハンドラーの種類

Java DSL ビルダー	XML DSL Type 属性	説明
defaultErrorHandler()	**DefaultErrorHandler**	例外を呼び出し元に戻し、再配信ポリシーをサポートしますが、デッドレターキューはサポートされません。
deadLetterChannel()	**DeadLetterChannel**	デフォルトのエラーハンドラーと同じ機能をサポートし、さらにデッドレターキューもサポートします。
loggingErrorChannel()	**LoggingErrorChannel**	例外が発生するたびに例外テキストをログに記録します。
noErrorHandler()	**NoErrorHandler**	エラーハンドラーを無効にするために使用できるダミーのハンドラー実装。
	TransactionErrorHandler	トランザクションが有効化されたルートのエラーハンドラー。トランザクションが有効化されたルートでは、デフォルトのトランザクションエラーハンドラーインスタンスが自動的に使用されます。

2.3.3. doTry、doCatch、および doFinally

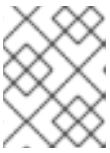
概要

ルート内で例外を処理するには、Java の **try**、**catch**、および **finally** ブロックと同様の方法で例外を処理する、**doTry**、**doCatch**、および **doFinally** 句の組み合わせを使用できます。

doCatch と Java における catch の類似点

通常、ルート定義内の **doCatch()** 句は、Java コードの **catch()** 文と同様の動作をします。具体的には、以下の機能が **doCatch()** 句でサポートされています。

- **複数の** doCatch **句** - 1 つの **doTry** ブロック内に複数の **doCatch** 句を持たせることができます。この **doCatch** 句は、Java の **catch()** 文と同様に、表示される順序でテストされます。Apache Camel は、出力された例外に一致する最初の **doCatch** 句を実行します。



注記

このアルゴリズムは、**onException** 句で使用される例外一致アルゴリズムとは異なります。詳細は 「onException 句」 を参照してください。

- **例外** の再出力 – コンストラクトを使用して **doCatch** 句内から現在の例外を再出力できます (「doCatch での例外の再出力」)。

doCatch の特別機能

ただし、**doCatch()** 句には Java の **catch()** 文に類似するものがない特別な機能がいくつかあります。以下の機能は、**doCatch()** に固有のものです。

- **条件キャッチ** - **doCatch** 句に **onWhen** サブ句を追加することで、例外を条件付きでキャッチできます (「onWhen による条件付き例外キャッチ」 を参照)。

例

以下の例は、Java DSL で **doTry** ブロックを書く方法を示しています。**doCatch()** 句は、**IOException** 例外または **IllegalStateException** 例外のいずれかが発生した場合に実行され、**doFinally()** 句は例外が発生したかどうかに関係なく、**常に** 実行されます。

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

または、Spring XML で同等のものを記述するとこのようになります。

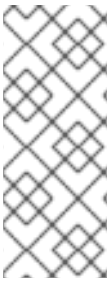
```
<route>
    <from uri="direct:start"/>
    <!-- here the try starts. its a try .. catch .. finally just as regular java code -->
    <doTry>
        <process ref="processorFail"/>
        <to uri="mock:result"/>
        <doCatch>
            <!-- catch multiple exceptions -->
            <exception>java.io.IOException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

doCatch での例外の再出力

次のように、コンストラクトを使用して、**doCatch ()** 句で例外を再出力することができます。

```
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();
```



注記

doTry/**doCatch** 句で非推奨となった **handled (false)** の代わりにプロセッサーを使用して例外を再出力することもできます。

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

上記の例では、**IOException** が **doCatch()** にキャッチされると、現在のエクスチェンジが **mock:io** エンドポイントに送信され、その後に **IOException** が再出力されます。これにより、ルートの先頭 (**from()** コマンド) にあるコンシューマーエンドポイントにも例外を処理する機会が与えられます。

以下の例では、Spring XML で同じルートを定義する方法を示しています。

```
<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>
```

onWhen による条件付き例外キャッチ

Apache Camel の **doCatch()** 句の特別な機能として、実行時に評価される式に基づいて例外のキャッチを条件付けすることができます。つまり、**doCatch(ExceptionList).doWhen(Expression)** の形式の句を使用して例外をキャッチした場合、述語の式 **Expression** が実行時に **true** に評価された場合にのみ例外がキャッチされます。

たとえば、以下の **doTry** ブロックは、例外メッセージが単語 **Severe** を含む場合にのみ、例外 **IOException** と **IllegalStateException** をキャッチします。

```
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();
```

または、Spring XML で同等のものを記述するとこのようになります。

```
<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 のネストされた条件

Camel の例外処理を JavaDSL ルートに追加するためのオプションは複数あります。**dotry()** は例外を処理するための try または catch ブロックを作成します。これはルート固有のエラー処理に役立ちます。

ChoiceDefinition 内部で例外をキャッチする場合は、以下のように **doTry** ブロックを使用できます。

```
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. SOAP 例外の伝播

概要

Camel CXF コンポーネントは Apache CXF とのインテグレーションを提供し、Apache Camel のエンドポイントから SOAP メッセージを送受信できます。XML で Apache Camel のエンドポイントを簡単に定義でき、それをエンドポイントの Bean ID を使用してルート内で参照できます。詳細は、**Apache Camel Component Reference Guide** の CXF を参照してください。

スタックトレース情報を伝播させる方法

Java の例外がサーバー側で発生したときに、例外のスタックトレースがフォールトメッセージにマーシャリングされてクライアントに返されるように、CXF エンドポイントを設定することができます。この機能を有効にするには、以下のように **cxfEndpoint** 要素で、**dataFormat** を **PAYLOAD** に設定し、**faultStackTraceEnabled** プロパティーを **true** に設定します。

```
<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>
```

セキュリティー上の理由から、スタックトレースには原因となる例外 (つまりスタックトレースの **Caused by** 以降の部分) は含まれません。スタックトレースに原因となる例外を含めたい場合は、以下のように **cxfEndpoint** 要素の **exceptionMessageCauseEnabled** プロパティーを **true** に設定します。

```
<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

exceptionMessageCauseEnabled フラグは、テストおよび診断目的でのみ有効にしてください。サーバーにおいて例外の元の原因を隠すことで、敵対的なユーザーがサーバーを調査しにくくするのが、通常の実践的なやり方です。

2.4. BEAN インテグレーション

概要

Bean インテグレーションは、任意の Java オブジェクトを使用してメッセージを処理するための汎用のメカニズムを提供します。Bean の参照をルートに挿入すると、Java オブジェクトの任意のメソッドを呼び出して、受信エクスチェンジにアクセスしたり変更したりすることができます。エクスチェンジの内容を Bean メソッドのパラメーターと戻り値にマッピングするメカニズムは、**パラメーターバインディング** と呼ばれます。パラメーターバインディングは、メソッドのパラメーターを初期化するために、以下のアプローチの任意の組み合わせを使用することができます。

- **規約に従ったメソッドシグネチャー** - メソッドシグネチャーが特定の規約に準拠している場合、パラメーターバインディングは Java リフレクションを使用して、どのパラメーターを渡すかを決定できます。
- **アノテーションと依存性注入** - より柔軟なバインディングメカニズムが必要な場合は、Java アノテーションを使用してメソッドの引数に何を注入するかを指定します。この依存性注入メカニズムは、Spring 2.5 のコンポーネントスキャンに基づきます。通常、Apache Camel アプリケーションを Spring コンテナーにデプロイする場合、依存性注入メカニズムは自動的に機能します。
- **明示的に指定したパラメーター** - Bean が呼び出される段階で、パラメーターを明示的に (定数として、または Simple 言語を使用して) 指定できます。

Bean レジストリー

Bean は **Bean レジストリー** を介してアクセスできます。Bean レジストリーは、クラス名または Bean ID のいずれかをキーとして Bean を検索できるサービスです。Bean レジストリーにエントリーを作成する方法は、基盤となるフレームワーク (たとえばプレーンな Java、Spring、Guice、または Blueprint など) によって異なります。レジストリーのエントリーは通常暗黙的に作成されます (例: Spring XML ファイルで Spring Bean をインスタンス化するときなど)。

レジストリープラグインストラテジー

Apache Camel は Bean レジストリーのプラグインストラテジーを実装しており、基盤となるレジストリー実装から透過的に Bean にアクセスするためのインテグレーション層を定義しています。そのため、表2.2「レジストリープラグイン」 に示されるように、Apache Camel アプリケーションをさまざまな Bean レジストリーと統合させることが可能です。

表2.2 レジストリープラグイン

レジストリー実装	レジストリープラグインのある Camel コンポーネント
Spring Bean レジストリー	**camel-spring**
Guice Bean レジストリー	**camel-guice**
Blueprint Bean レジストリー	**camel-blueprint**
OSGi サービスレジストリー	**OSGi コンテナー**にデプロイされている
JNDI レジストリー	

通常、関連する Bean レジストリーが自動的にインストールされるため、Bean レジストリーの設定を自ら行なう必要はありません。たとえば、Spring フレームワークを使用してルートを定義する場合、Spring **ApplicationContextRegistry** プラグインは現在の **CamelContext** インスタンスに自動的にインストールされます。

OSGi コンテナーへのデプロイは特別なケースになります。Apache Camel ルートが OSGi コンテナーにデプロイされると、**CamelContext** が Bean インスタンスの解決のためにレジストリーチェーンを自動的に設定します。レジストリーチェーンは OSGi レジストリーと、それに続く Blueprint (または Spring) レジストリーで設定されます。

Java で作成された Bean へのアクセス

Java Bean (Plain Old Java Object または POJO) を使用してエクスチェンジオブジェクトを処理するには、インバウンドエクスチェンジを Java オブジェクトのメソッドにバインドする **bean()** プロセッサーを使用します。たとえば、**MyBeanProcessor** クラスを使用してインバウンドエクスチェンジを処理するには、以下のようにルートを定義します。

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

bean() プロセッサーは **MyBeanProcessor** 型のインスタンスを作成し、**processBody()** メソッドを呼び出してインバウンドエクスチェンジを処理します。単一のルートからのみ **MyBeanProcessor** インスタンスにアクセスする場合には、この方法が適切です。しかし、複数のルートから同じ **MyBeanProcessor** インスタンスにアクセスする場合は、**Object** 型を最初の引数として取る **bean()** のバリアントを使用します。以下に例を示します。

```
MyBeanProcessor myBean = new MyBeanProcessor();

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

オーバーロードされた Bean メソッドへのアクセス

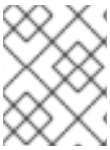
Bean がオーバーロードされた複数のメソッドを定義する場合、メソッド名とそのパラメーター型を指定して、どのオーバーロードされたメソッドを呼び出すかを選択できます。たとえば、**MyBeanBrocessor** クラスに 2 つのオーバーロードされたメソッド **processBody(String)** および **processBody(String,String)** がある場合、後者のオーバーロードされたメソッドを以下のように呼び出すことができます。

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

または、各パラメーターの型を明示的に指定するのではなく、受け取るパラメーターの数でメソッドを特定する場合は、ワイルドカード文字 * を使用できます。たとえば、パラメーターの正確な型に関係なく、2 つのパラメーターを取る名前が **processBody** のメソッドを呼び出すには、以下のように **bean()** プロセッサーを呼び出します。

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

メソッドを指定する場合、単純な修飾なしの型名 (例: **processBody(Exchange)**) または完全修飾型名 (例: **processBody(org.apache.camel.Exchange)**) のいずれかを使用できます。



注記

現在の実装では、指定された型名はパラメーター型に完全に一致する必要があります。型の継承は考慮されません。

パラメーターの明示的な指定

Bean メソッドを呼び出す際に、パラメーター値を明示的に指定できます。以下の単純な型の値を渡すことができます。

- ブール値: **true** または **false**
- 数値: **123**、**7** など
- 文字列: **'In single quotes'** または **"In double quotes"**
- Null オブジェクト: **null**

以下の例は、同じメソッド呼び出しの中で明示的なパラメーター値と型指定子を混在させる方法を示しています。

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

上記の例では、最初のパラメーターの値はパラメーターバインディングアノテーションによって決定されます (「基本アノテーション」 を参照)。

単純な型の値の他に、Simple 言語 (30章*Simple 言語*) を使用してパラメーター値を指定することもできます。これは、パラメーター値を指定する際に **Simple 言語の完全な機能が利用可能である** ことを意味します。たとえば、メッセージボディーと **title** ヘッダーの値を Bean メソッドに渡すには、以下のようにします。

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

ヘッダーのハッシュマップ全体をパラメーターとして渡すこともできます。たとえば、以下の例では、2 つ目のメソッドパラメーターは **java.util.Map** 型として宣言する必要があります。

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

注記

Apache Camel 2.19 のリリースから、Bean メソッド呼び出しから null を返すことで、常にメッセージボディーが null 値として設定されるようになりました。

基本的なメソッドシグネチャー

エクスチェンジを Bean メソッドにバインドするには、特定の規約に準拠するメソッドシグネチャーを定義します。特に、メソッドシグネチャーには 2 つの基本的な規約があります。

- メッセージボディーを処理するメソッドシグネチャー
- エクスチェンジを処理するメソッドシグネチャー

メッセージボディーを処理するメソッドシグネチャー

受信メッセージボディーにアクセスしたり、これを変更したりする Bean メソッドを実装する場合は、単一の **String** 引数を取り、**String** 値を返すメソッドシグネチャーを定義する必要があります。以下に例を示します。

```
// Java
package com.acme;

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

エクスチェンジを処理するメソッドシグネチャー

より柔軟性を高めるために、受信エクスチェンジにアクセスする Bean メソッドを実装できます。これにより、すべてのヘッダー、ボディー、エクスチェンジプロパティーにアクセスしたり、変更したりすることができます。エクスチェンジの処理には、メソッドシグネチャーは単一の **org.apache.camel.Exchange** パラメーターを取り、**void** を返します。以下に例を示します。

```
// 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!");
    }
}
```

Spring XML から Spring Bean へのアクセス

Java で Bean インスタンスを作成する代わりに、Spring XML を使用してインスタンスを作成できます。実際、ルートを XML で定義している場合には、これが唯一の実行可能な方法です。XML で Bean を定義するには、標準の Spring **bean** 要素を使用します。以下の例は、**MyBeanProcessor** のインスタンスを作成する方法を示しています。

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

Spring 構文を使用して、データを Bean のコンストラクター引数に渡すこともできます。Spring **bean** 要素の使用方法に関する詳細は、Spring リファレンスガイドの The IoC Container を参照してください。

bean 要素を使用してオブジェクトインスタンスを作成する場合、Bean の ID (**bean** 要素の **id** 属性の値) を使用すると後でオブジェクトインスタンスを参照できます。たとえば、ID が **myBeanId** と同じ **bean** 要素がある場合は、以下のように **beanRef()** プロセッサーを使用して Java DSL ルート内で Bean を参照できます。

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

beanRef() プロセッサーは、指定された Bean インスタンスで **MyBeanProcessor.processBody()** メソッドを呼び出します。

Spring XML ルート内から、Camel スキーマの **bean** 要素を使用して Bean を呼び出すこともできます。以下に例を示します。

```
<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>
```

効率を若干向上させるために、**cache** オプションを **true** に設定して、Bean が使用されるたびにレジストリーを検索しないようにすることもできます。たとえば、キャッシュを有効にするには、以下のように **bean** 要素の **cache** 属性を設定します。

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

Java からの Spring Bean へのアクセス

Spring **bean** 要素を使用してオブジェクトインスタンスを作成する場合、Bean の ID (**bean** 要素の **id** 属性の値) を使用して Java からオブジェクトインスタンスを参照できます。たとえば、ID が **myBeanId** と同じ **bean** 要素がある場合は、以下のように **beanRef()** プロセッサーを使用して Java DSL ルート内

で Bean を参照できます。

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

または、以下のように **@BeanInject** アノテーションを使用して、依存性注入によって Spring Bean を参照することもできます。

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

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

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

@BeanInject アノテーションから Bean ID を省略した場合、Camel は型別にレジストリーを検索しますが、これは指定された型の Bean が 1 つだけの場合にのみ機能します。たとえば、**com.acme.MyBeanProcessor** 型の Bean を検索して依存性注入するには、以下を実行します。

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

Spring XML における Bean のシャットダウン順序

Camel コンテキストで使用される Bean の場合、通常、正しいシャットダウンの順序は次のようになります。

1. **camelContext** インスタンスをシャットダウンします。
2. 使用された Bean をシャットダウンします。

このシャットダウン順序が逆の場合、Camel コンテキストがすでに破棄された Bean にアクセスしようとすることがあります (直接エラーになるか、または Camel コンテキストが破棄されている間に見つからなかった Bean を作成しようして、結局エラーになるかのどちらかです)。Spring XML のデフォルトのシャットダウン順序は、Bean と **camelContext** が Spring XML ファイルの中で出現する順序によって異なります。誤ったシャットダウン順序によるランダムなエラーを回避するため、**camelContext** は Spring XML ファイルの他の Bean よりも **前に** シャットダウンするように設定されています。これは Apache Camel 2.13.0 以降のデフォルトの動作です。

この動作を変更 (Camel コンテキストが他の Bean の前に強制的にシャットダウン **されない** ように) する必要がある場合は、**camelContext** 要素の **shutdownEager** 属性を **false** に設定します。この場合、Spring の **depends-on** 属性を使用して、シャットダウンの順序をより詳細に制御することもできます。

パラメーターバインディングアノテーション

「基本的なメソッドシグネチャー」 で説明されている基本的なパラメーターバインディングは、必ずしも便利に使えるとは限りません。たとえば、何らかのデータ操作を行うレガシーな Java クラスがある場合、インバウンドエクスチェンジからデータを抽出し、既存のメソッドシグネチャーの引数にマップ

する必要があるかもしれません。このようなパラメーターバインディングには、Apache Camel は以下のような Java アノテーションを提供します。

- 基本アノテーション
- 言語アノテーション
- 継承されたアノテーション

基本アノテーション

表2.3「基本の Bean アノテーション」 は、Bean メソッドの引数にメッセージデータを依存性注入するために使用できる **org.apache.camel** Java パッケージのアノテーションを示しています。

表2.3 基本の Bean アノテーション

アノテーション	意味	パラメーター
@Attachments	アタッチメントのリストにバインドします。	
@Body	インバウンドメッセージのボディーにバインドします。	
@Header	インバウンドメッセージのヘッダーにバインドします。	ヘッダーの文字列名。
@Headers	インバウンドメッセージヘッダーの **java.util.Map** にバインドします。	
@OutHeaders	アウトバウンドメッセージヘッダーの **java.util.Map** にバインドします。	
@Property	名前のあるエクスチェンジプロパティーにバインドします。	プロパティーの文字列名。
@Properties	エクスチェンジプロパティーの **java.util.Map** にバインドします。	

たとえば、以下のクラスは基本アノテーションを使用してメッセージデータを **processExchange()** メソッド引数に依存性注入する方法を示しています。

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

アノテーションがどのようにデフォルトの規約と混在できるかに注目してください。パラメーターバインディングは、アノテーションが付けられた引数を依存性注入するだけでなく、エクスチェンジオブジェクトも **org.apache.camel.Exchange** 引数に自動的に依存性注入します。

式言語アノテーション

式言語アノテーションは、メッセージデータを Bean メソッドの引数に依存性注入する強力なメカニズムを提供します。これらのアノテーションを使用すると、任意のスクリプト言語で書かれた任意のスクリプトを呼び出して、インバウンドエクスチェンジからデータを抽出し、メソッド引数に注入することができます。表2.4「式言語アノテーション」 は、Bean メソッドの引数にメッセージデータを依存性注入するために使用できる **org.apache.camel.language** パッケージ (およびコア以外のアノテーションのサブパッケージ) のアノテーションを示しています。

表2.4 式言語アノテーション

アノテーション	説明
@Bean	Bean 式を注入します。
@Constant	Constant 式を注入します。
@EL	EL 式を注入します。
@Groovy	Groovy 式を注入します。
@Header	ヘッダー式を注入します。
@JavaScript	JavaScript 式を注入します。
@OGNL	OGNL 式を注入します。
@PHP	PHP 式を注入します。
@Python	Python 式を注入します。
@Ruby	Ruby 式を注入します。
@Simple	Simple 式を注入します。
@XPath	XPath 式を注入します。
@XQuery	XQuery 式を注入します。

たとえば、以下のクラスは、XML 形式の受信メッセージのボディーからユーザー名とパスワードを抽出するために **@XPath** アノテーションを使用する方法を示しています。

```
// 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 アノテーションは、特殊なケースになります。登録された Bean の呼び出し結果を依存性注入できるためです。たとえば、相関 ID をメソッド引数に依存性注入するには、以下のように **@Bean** アノテーションを使用して ID 生成クラスを呼び出します。

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

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

文字列 **myCorrIdGenerator** は ID 生成インスタンスの Bean ID です。ID 生成クラスは、以下のように Spring の **bean** 要素を使用してインスタンス化できます。

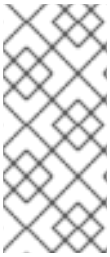
```
<beans ...>
    ...
    <bean id="myCorrIdGenerator" class="com.acme.MyIdGenerator"/>
</beans>
```

MyIdGenerator クラスは以下のように定義することができます。

```
// 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;
    }
}
```

参照された Bean クラス **MyIdGenerator** でアノテーションを使用することもできます。**generate()** メソッドシグネチャーに対する唯一の制限は、**@Bean** アノテーションが付けられた引数に依存性注入するために正しい型を返す必要があることです。**@Bean** アノテーションではメソッド名を指定できないため、依存性注入メカニズムは単純に参照された Bean の戻り値型が一致する最初のメソッドを呼び出します。



注記

言語アノテーションのいくつかはコアコンポーネントで利用できます (**@Bean**、**@Constant**、**@Simple**、および **@XPath**)。しかし、コア以外のコンポーネントの場合、該当するコンポーネントをロードしておく必要があります。たとえば、OGNL スクリプトを使用するには、**camel-ognl** コンポーネントをロードする必要があります。

継承されたアノテーション

パラメーターバインディングアノテーションは、インターフェイスまたはスーパークラスから継承できます。たとえば、以下のように **Header** アノテーションと **Body** アノテーションの付いた Java インターフェイスを定義したとします。

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

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

実装クラス **MyBeanProcessor** で定義されたオーバーロードされたメソッドは、以下のように基本インターフェイスに定義されたアノテーションを継承します。

```
// 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
    ) {
        ...
    }
}
```

インターフェイスの実装

Java インターフェイスを実装するクラスは、多くの場合、**protected**、**private**、または **package-only** の範囲となります。このように制限された実装クラスのメソッドを呼び出す場合、Bean バインディングはフォールバックして、公開アクセス可能な対応するインターフェイスメソッドを呼び出します。

たとえば、以下のパブリック **BeanIntf** インターフェイスについて考えてみましょう。

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

BeanIntf インターフェイスは、以下の protected な **BeanIntfImpl** クラスによって実装されます。

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

以下の Bean 呼び出しは、フォールバックして public な **BeanIntf.processBodyAndHeader** メソッドを呼び出します。

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

static メソッドの呼び出し

Bean インテグレーションには、関連付けられたクラスのインスタンスを作成 **せずに** static メソッドを呼び出す機能があります。たとえば、static メソッド **changeSomething()** を定義した以下の Java クラスについて考えてみましょう。

```
// 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
    }
}
```

以下のように、Bean インテグレーションを使用して static **changeSomething** メソッドを呼び出すことができます。

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

この構文は、通常の関数の呼び出しと同じように見えますが、Bean インテグレーションは Java のリフレクションを利用してこのメソッドを static と識別し、**MyStaticClass** をインスタンス化 **せずに** メソッドの呼び出しに進むことに留意してください。

OSGi サービスの呼び出し

ルートが Red Hat Fuse コンテナーにデプロイされた特別なケースでは、Bean インテグレーションを使用して OSGi サービスを直接呼び出すことができます。たとえば、OSGi コンテナーのバンドルのいずれかがサービス **org.fusesource.example.HelloWorldOsgiService** をエクスポートしているとすると、以下のような Bean インテグレーションのコードを使用して **sayHello** メソッドを呼び出すことができます。

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

以下のように Bean コンポーネントを使用して、Spring または Blueprint XML ファイル内から OSGi サービスを呼び出すこともできます。

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

これが動作する仕組みは、Apache Camel が OSGi コンテナーにデプロイされる際にレジストリーのチェーンを設定することによります。まず、OSGi サービスレジストリーで指定のクラス名を検索します。検索に失敗した場合、ローカルの Spring DM または Blueprint レジストリーにフォールバックします。

2.5. エクスチェンジインスタンスの作成

概要

Java コード (たとえば Bean クラスやプロセッサークラス) でメッセージを処理している際に、新しいエクスチェンジインスタンスの生成が必要になることがあります。**Exchange** オブジェクトを作成する必要がある場合は、ここで説明するように **ExchangeBuilder** クラスのメソッドを呼び出すことが最も簡単な方法になります。

ExchangeBuilder クラス

ExchangeBuilder クラスの完全修飾名は以下の通りです。

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

ExchangeBuilder は、エクスチェンジオブジェクトの構築を開始するために使用できる static メソッド **anExchange** を公開しています。

例

たとえば、以下のコードは、メッセージボディーに文字列 **Hello World!** を持ち、ヘッダーにユーザー名とパスワードのクレデンシャルを含んだ新しいエクスチェンジオブジェクトを作成します。

```
// 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 のメソッド

ExchangeBuilder クラスは以下のメソッドをサポートします。

ExchangeBuilder anExchange(CamelContext context)

(static メソッド) エクスチェンジオブジェクトの構築を開始します。

Exchange build()

エクスチェンジを構築します。

ExchangeBuilder withBody(Object body)

エクスチェンジにメッセージボディーを設定します (つまり、エクスチェンジの **In** メッセージボディーを設定します)。

ExchangeBuilder withHeader(String key, Object value)

エクスチェンジにヘッダーを設定します (つまり、エクスチェンジの **In** メッセージにヘッダーを設定します)。

ExchangeBuilder withPattern(ExchangePattern pattern)

エクスチェンジに交換パターンを設定します。

ExchangeBuilder withProperty(String key, Object value)

エクスチェンジにプロパティーを設定します。

2.6. メッセージコンテンツの変換

概要

Apache Camel は、メッセージコンテンツを変換するためのさまざまなアプローチをサポートしています。Apache Camel は、メッセージコンテンツを変更するためのシンプルなネイティブ API に加えて、いくつかの異なるサードパーティーライブラリーや変換のための標準とのインテグレーションをサポートしています。

2.6.1. シンプルなメッセージ変換

概要

Java DSL にはビルトインの API があり、送受信メッセージのシンプルな変換を実行できます。たとえば、例2.1「受信メッセージのシンプルな変換」 に示すルールは、受信メッセージのボディー部の末尾にテキスト **World!** を追加します。

例2.1 受信メッセージのシンプルな変換

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

ここで、**setBody()** コマンドは受信メッセージボディーのコンテンツを置き換えます。

シンプルな変換の API

以下の API クラスを使用して、ルーターのルールによってメッセージコンテンツのシンプルな変換を実行できます。

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

ProcessorDefinition クラス

org.apache.camel.model.ProcessorDefinition クラスは、ルーターのルールに直接挿入できる DSL コマンドを定義しています (例: 例2.1「受信メッセージのシンプルな変換」 の **setBody()** コマンド)。表2.5「ProcessorDefinition クラスの変換メソッド」 は、メッセージコンテンツの変換に関係のある **ProcessorDefinition** のメソッドを示しています。

表2.5 ProcessorDefinition クラスの変換メソッド

メソッド	説明
Type convertBodyTo(Class type)	IN メッセージのボディーを指定の型に変換します。
Type removeFaultHeader(String name)	FAULT メッセージのヘッダーを削除するプロセッサーを追加します。
Type removeHeader(String name)	IN メッセージ上のヘッダーを削除するプロセッサーを追加します。
Type removeProperty(String name)	エクスチェンジプロパティーを削除するプロセッサーを追加します。
ExpressionClause<ProcessorDefinition<Type>> setBody()	IN メッセージのボディーをセットするプロセッサーを追加します。
Type setFaultBody(Expression expression)	FAULT メッセージのボディーをセットするプロセッサーを追加します。
Type setFaultHeader(String name, Expression expression)	FAULT メッセージにヘッダーをセットするプロセッサーを追加します。
ExpressionClause<ProcessorDefinition<Type>> setHeader(String name)	IN メッセージにヘッダーをセットするプロセッサーを追加します。
Type setHeader(String name, Expression expression)	IN メッセージにヘッダーをセットするプロセッサーを追加します。

メソッド	説明
ExpressionClause<ProcessorDefinition<Type>> setOutHeader(String name)	OUT メッセージにヘッダーをセットするプロセッサーを追加します。
Type setOutHeader(String name, Expression expression)	OUT メッセージにヘッダーをセットするプロセッサーを追加します。
ExpressionClause<ProcessorDefinition<Type>> setProperty(String name)	エクスチェンジプロパティーをセットするプロセッサーを追加します。
Type setProperty(String name, Expression expression)	エクスチェンジプロパティーをセットするプロセッサーを追加します。
ExpressionClause<ProcessorDefinition<Type>> transform()	OUT メッセージのボディーをセットするプロセッサーを追加します。
Type transform(Expression expression)	OUT メッセージのボディーをセットするプロセッサーを追加します。

Builder クラス

org.apache.camel.builder.Builder クラスは、式または述語が想定される文脈でのメッセージコンテンツへのアクセスを提供します。つまり、**Builder** のメソッドは通常 DSL コマンドの **引数** の中で呼び出されます (例: 例2.1「受信メッセージのシンプルな変換」 の **body()** コマンド)。表2.6「Builder クラスのメソッド」 では、**Builder** クラスで利用可能な static メソッドをまとめています。

表2.6 Builder クラスのメソッド

メソッド	説明
static <E extends Exchange> ValueBuilder<E> body()	エクスチェンジのインバウンドボディーに対する述語および値ビルダーを返します。
static <E extends Exchange,T> ValueBuilder<E> bodyAs(Class<T> type)	インバウンドメッセージのボディーを特定の型として、それに対する述語および値ビルダーを返します。
static <E extends Exchange> ValueBuilder<E> constant(Object value)	定数式を返します。
static <E extends Exchange> ValueBuilder<E> faultBody()	エクスチェンジのフォールトボディーに対する述語および値ビルダーを返します。
static <E extends Exchange,T> ValueBuilder<E> faultBodyAs(Class<T> type)	フォールトメッセージのボディーを特定の型として、それに対する述語および値ビルダーを返します。

メソッド	説明
static <E extends Exchange> ValueBuilder<E> header(String name)	エクスチェンジのヘッダーに対する述語および値ビルダーを返します。
static <E extends Exchange> ValueBuilder<E> outBody()	エクスチェンジのアウトバウンドボディーに対する述語および値ビルダーを返します。
static <E extends Exchange> ValueBuilder<E> outBodyAs(Class<T> type)	アウトバウンドメッセージのボディーを特定の型として、それに対する述語および値ビルダーを返します。
static ValueBuilder property(String name)	エクスチェンジのプロパティーに対する述語および値ビルダーを返します。
static ValueBuilder regexReplaceAll(Expression content, String regex, Expression replacement)	正規表現のすべての出現箇所を、指定した置換文字列で置き換える式を返します。
static ValueBuilder regexReplaceAll(Expression content, String regex, String replacement)	正規表現のすべての出現箇所を、指定した置換文字列で置き換える式を返します。
static ValueBuilder sendTo(String uri)	指定したエンドポイント URI にエクスチェンジを送信する式を返します。
static <E extends Exchange> ValueBuilder<E> systemProperty(String name)	指定のシステムプロパティーの式を返します。
static <E extends Exchange> ValueBuilder<E> systemProperty(String name, String defaultValue)	指定のシステムプロパティーの式を返します。

ValueBuilder クラス

org.apache.camel.builder.ValueBuilder クラスを使用すると、**Builder** メソッドによって返される値を変更できます。つまり、**ValueBuilder** のメソッドは、メッセージコンテンツを変更するシンプルな方法を提供します。表2.7「ValueBuilder クラスの変更メソッド」 では、**ValueBuilder** クラスで利用可能なメソッドをまとめています。この表では、呼び出された値を変更するために使用されるメソッドのみが示されています (詳細は **API Reference** ドキュメントを参照してください)。

表2.7 ValueBuilder クラスの変更メソッド

メソッド	説明
ValueBuilder<E> append(Object value)	指定された値をこの式の文字列評価に追加します。
Predicate contains(Object value)	左辺の式に右辺の式の値が含まれた述語を作成します。

メソッド	説明
ValueBuilder<E> convertTo(Class type)	登録された型コンバーターを使用して、現在の値を指定の型に変換します。
ValueBuilder<E> convertToString()	登録された型コンバーターを使用して、現在の値を String に変換します。
Predicate endsWith(Object value)	
<T> T evaluate(Exchange exchange, Class<T> type)	
Predicate in(Object… values)	
Predicate in(Predicate… predicates)	
Predicate isEqualTo(Object value)	現在の値が指定の **value** 引数と等しい場合、true を返します。
Predicate isGreaterThan(Object value)	現在の値が指定の **value** 引数よりも大きい場合、true を返します。
Predicate isGreaterThanOrEqualTo(Object value)	現在の値が指定の **value** 引数より大きい場合、true を返します。
Predicate isInstanceOf(Class type)	現在の値が指定の型のインスタンスである場合、true を返します。
Predicate isLessThan(Object value)	現在の値が指定の **value** 引数未満の場合、true を返します。
Predicate isLessThanOrEqualTo(Object value)	現在の値が指定の **value** 引数以下である場合、true を返します。
Predicate isNotEqualTo(Object value)	現在の値が指定の **value** 引数と等しくない場合、true を返します。
Predicate isNotNull()	現在の値が **null** でない場合、true を返します。
Predicate isNull()	現在の値が **null** の場合、true を返します。
Predicate matches(Expression expression)	
Predicate not(Predicate predicate)	述語の引数を否定にします。
ValueBuilder prepend(Object value)	この式の文字列評価を指定された値に追加します。

メソッド	説明
Predicate regex(String regex)	
ValueBuilder<E> regexReplaceAll(String regex, Expression<E> replacement)	正規表現のすべての出現箇所を、指定した置換文字列で置き換えます。
ValueBuilder<E> regexReplaceAll(String regex, String replacement)	正規表現のすべての出現箇所を、指定した置換文字列で置き換えます。
ValueBuilder<E> regexTokenize(String regex)	指定の正規表現を使用してこの式の文字列変換をトークン化します。
ValueBuilder sort(Comparator comparator)	指定されたコンパレーターを使用して現在の値をソートします。
Predicate startsWith(Object value)	現在の値が **value** 引数の文字列値と一致する場合、true を返します。
ValueBuilder<E> tokenize()	コンマのトークン区切り文字を使用してこの式の文字列変換をトークン化します。
ValueBuilder<E> tokenize(String token)	指定のトークン区切り文字を使用してこの式の文字列変換をトークン化します。

2.6.2. マーシャリングとアンマーシャリング

Java DSL コマンド

以下のコマンドを使用して、低レベルのメッセージ形式と高レベルのメッセージ形式の間で変換を行うことができます。

- **marshal()** - 高レベルのデータフォーマットを低レベルのデータフォーマットに変換します。
- **unmarshal()** - 低レベルのデータフォーマットを高レベルのデータフォーマットに変換します。

データ形式

Apache Camel は、以下のデータフォーマットのマーシャリングおよびアンマーシャリングをサポートします。

- Java シリアライゼーション
- JAXB
- XMLBeans
- XStream

Java シリアライゼーション

Java オブジェクトをバイナリーデータの Blob に変換できるようにします。このデータフォーマットでは、アンマーシャリングはバイナリー Blob を Java オブジェクトに変換し、マーシャリングは Java オブジェクトをバイナリー Blob に変換します。たとえば、エンドポイント **SourceURL** からシリアライズされた Java オブジェクトを読み込み、これを Java オブジェクトに変換するには、以下のようなルールを使用します。

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

または、Spring XML では以下の通りです。

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

JAXB

XML スキーマ型と Java 型間のマッピングを提供します (https://jaxb.dev.java.net/ を参照してください)。JAXB では、アンマーシャリングは XML データ型を Java オブジェクトに変換し、マーシャリングは Java オブジェクトを XML データ型に変換します。JAXB データフォーマットを使用する前に、JAXB コンパイラーを使用して XML スキーマをコンパイルし、スキーマの XML データ型を表す Java クラスを生成する必要があります。これはスキーマの **バインディング** と呼ばれます。スキーマをバインドした後に、以下のようなコードを使用して XML データを Java オブジェクトにアンマーシャリングするルールを定義します。

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

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

GeneratedPackagename は JAXB コンパイラーによって生成された Java パッケージの名前で、XML スキーマを表す Java クラスが含まれます。

または、Spring XML では以下の通りです。

```
<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

XML スキーマ型と Java 型間の代替的なマッピングを提供します (http://xmlbeans.apache.org/ を参照してください) 。XMLBean では、アンマーシャリングは XML データ型を Java オブジェクトに変換し、マーシャリングは Java オブジェクトを XML データ型に変換します。たとえば、XMLBean を使用して XML データを Java オブジェクトにアンマーシャリングするには、以下のようなコードを使用します。

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

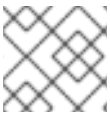
または、Spring XML では以下の通りです。

```
<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

XML 型と Java 型間のもう 1 つのマッピングを提供します (http://www.xml.com/pub/a/2004/08/18/xstream.html を参照してください) 。XStream はシリアライゼーションライブラリー (Java シリアライゼーションなど) で、Java オブジェクトを XML に変換できるものです。XStream では、アンマーシャリングは XML データ型を Java オブジェクトに変換し、マーシャリングは Java オブジェクトを XML データ型に変換します。

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



注記

XStream データフォーマットは現在 Spring XML ではサポート **されません。**

2.6.3. エンドポイントバインディング

バインディングの概要

Apache Camel において、**バインディング** とは、エンドポイントにコントラクトを結び付ける方法です。たとえば Data Format、Content Enricher、または検証ステップを適用することでコントラクトを結び付けます。入力メッセージには条件または変換が適用され、出力メッセージには補完的な条件または変換が適用されます。

DataFormatBinding

DataFormatBinding クラスは、特定のデータフォーマットをマーシャリングしたりアンマーシャリングしたりするバインディングを定義する場合に有効です (「マーシャリングとアンマーシャリング」 を参照)。この場合、バインディングの作成に必要なのは、コンストラクターに必要なデータフォーマットへの参照を渡して **DataFormatBinding** インスタンスを作成することだけです。

たとえば、例2.2「JAXB バインディング」 の XML DSL スニペットは、Apache Camel エンドポイントに関連付けられたときに、JAXB データフォーマットをマーシャリングおよびアンマーシャリングできるバインディング (ID は **jaxb**) を示しています。

例2.2 JAXB バインディング

```
<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>
```

バインディングとエンドポイントの関連付け

エンドポイントとバインディングを関連付けるには、以下の方法を使用できます。

- Binding URI
- コンポーネント

Binding URI

バインディングをエンドポイントに関連付けるには、エンドポイント URI に **binding:NameOfBinding** を接頭辞として付けます。ここで **NameOfBinding** は、バインディングの Bean ID になります (例: Spring XML で作成されたバインディング Bean の ID)。

たとえば、以下の例では、ActiveMQ エンドポイントを 例2.2「JAXB バインディング」 で定義された JAXB バインディングに関連付ける方法を示しています。

```
<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

接頭辞を使用してバインディングをエンドポイントに関連付ける代わりに、関連付けを暗黙的に行い、バインディングが URI に表示されないようにすることもできます。暗黙的なバインディングを持たない既存のエンドポイントの場合、最も簡単な方法は **BindingComponent** クラスを使用してエンドポイントをラップすることです。

たとえば、**jaxb** バインディングを **activemq** エンドポイントに関連付けるには、以下のように新しい **BindingComponent** インスタンスを定義します。

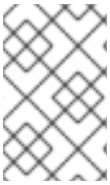
```
<beans ... >
    ...
    <bean id="jaxbmq" class="org.apache.camel.component.binding.BindingComponent">
        <constructor-arg ref="jaxb"/>
        <constructor-arg value="activemq:foo."/>
    </bean>

    <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>
```

jaxbmq の 2 つ目のコンストラクター引数 (オプション) で URI 接頭辞を定義します。これで、この **jaxbmq** ID をエンドポイント URI のスキームとして使用できるようになりました。たとえば、このバインディングコンポーネントを使用して以下のルートを定義できます。

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

上記のルートは、Binding URI のアプローチを使用する以下のルートと同じです。

```
<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>
```



注記

カスタムの Apache Camel コンポーネントを実装する開発者は、**org.apache.camel.spi.HasBinding** インターフェイスを継承するエンドポイントクラスを実装することで、これを実現できます。

BindingComponent コンストラクター

BindingComponent クラスは以下のコンストラクターをサポートします。

public BindingComponent()

無引数の形式です。プロパティー注入を使用してバインディングコンポーネントインスタンスを設定します。

public BindingComponent(Binding binding)

このバインディングコンポーネントを指定された **Binding** オブジェクトの **binding** に関連付けます。

public BindingComponent(Binding binding, String uriPrefix)

このバインディングコンポーネントを指定された **Binding** オブジェクトの **binding**、および URI 接頭辞 **uriPrefix** に関連付けます。これが、最も一般的に使用されるコンストラクターです。

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

このコンストラクターは、追加の URI ポストフィックス **uriPostfix** 引数をサポートします。これは、このバインディングコンポーネントを使用して定義された URI に自動的に追加されます。

カスタムバインディングの実装

マーシャリングおよびアンマーシャリングのデータフォーマットに使用される **DataFormatBinding** に加えて、独自のカスタムバインディングを実装することができます。カスタムバインディングを以下のように定義します。

1. **org.apache.camel.Processor** クラスを実装して、(**from** 要素に登場) コンシューマーエンドポイントで受信するメッセージに対して変換を行います。

2. 補完関係となる **org.apache.camel.Processor** クラスを実装して、プロデューサーエンドポイント (**to** 要素に登場) から送信されるメッセージに対して逆変換を行います。

3. **org.apache.camel.spi.Binding** インターフェイスを実装します。これは上記のプロセッサーインスタンスのファクトリーとして機能します。

Binding インターフェイス

例2.3「org.apache.camel.spi.Binding インターフェイス」 は **org.apache.camel.spi.Binding** インターフェイスの定義を示しています。このインタフェースは、カスタムバインディングを定義するために実装する必要があります。

例2.3 org.apache.camel.spi.Binding インターフェイス

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

バインディングを使うタイミング

バインディングは、多くの異なるエンドポイントに同じ種類の変換を適用する必要がある場合に有効です。

2.7. プロパティープレースホルダー

概要

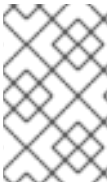
プロパティープレースホルダー機能は、さまざまなコンテキスト (エンドポイント URI や XML DSL 要素の属性など) で文字列を置き換えるために使用できます。プレースホルダーの設定は Java プロパティーファイルに格納されます。この機能は、異なる Apache Camel アプリケーション間で設定を共有する場合や、特定の設定を一元管理する場合に役立ちます。

たとえば、以下のルートはリクエストを Web サーバーに送信します。この Web サーバーのホストとポートは、プレースホルダーの **{{remote.host}}** と **{{remote.port}}** に置き換えられます。

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

プレースホルダーの値は、以下のように Java プロパティーファイルに定義されています。

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



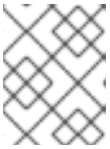
注記

プロパティープレースホルダーはエンコーディングオプションをサポートし、UTF-8 などの特定の文字セットを使用して、**.properties** ファイルの読み取りを可能にします。ただし、デフォルトでは ISO-8859-1 文字セットによる実装になります。

PropertyPlaceholders を使用した Apache Camel では、以下がサポートされます。

- 検索するキーと共にデフォルト値を指定する。
- すべてのプレースホルダーキーがデフォルト値で設定されていて、それらが使用される場合、**PropertiesComponent** を定義する必要がない。

- サードパーティー関数を使用してプロパティー値を検索する。これにより、独自のロジックを実装できます。



注記

プロパティー値を検索する関数として、標準で OS 環境変数、JVM システムプロパティー、サービス名イディオムの 3 つを提供します。

プロパティーファイル

プロパティー設定は 1 つ以上の Java プロパティーファイルに格納され、標準の Java プロパティーファイル形式に準拠する必要があります。各プロパティー設定は、それぞれ独立した行に **Key=Value** の形式で表示されます。空白以外の最初の文字が **#** または **!** から始まる行は、コメントとして扱われます。

たとえば、プロパティーファイルは 例2.4「プロパティーファイルの例」 のような内容になります。

例2.4 プロパティーファイルの例

```
# 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
```

プロパティーの解決

Properties コンポーネントは、ルート定義の中で使用を開始する前に、1 つ以上のプロパティーファイルのロケーションを指定して設定しておく必要があります。以下のいずれかのリゾルバーを使用して、プロパティー値を提供する必要があります。

classpath:PathName,PathName,...

(デフォルト) クラスパス上のロケーションを指定します。**PathName** は、フォワードスラッシュを使用して区切られたファイルパス名です。

file:PathName,PathName,...

ファイルシステムのロケーションを指定します。**PathName** はフォワードスラッシュを使用して区切られたファイルパス名です。

ref:BeanID

レジストリーの **java.util.Properties** オブジェクトの ID を指定します。

blueprint:BeanID

cm:property-placeholder Bean の ID を指定します。この Bean は、OSGi Blueprint ファイルのコンテキスト内で、**OSGi Configuration Admin** サービスで定義されたプロパティーにアクセスするために使用されます。詳細は、 「OSGi Blueprint プロパティープレースホルダーとの統合」 を参照し

てください。

たとえば、クラスパス上にある **com/fusesource/cheese.properties** プロパティーファイルと **com/fusesource/bar.properties** プロパティーファイルを指定するには、以下のようなロケーション文字列を使用します。

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

注記

クラスパスリゾルバーはデフォルトで使用されるため、この例では **classpath:** 接頭辞は省略できます。

システムプロパティーと環境変数を使用したロケーションの指定

ロケーション **PathName** に Java システムプロパティーおよび O/S 環境変数を埋め込むことができます。

Java システムプロパティーは、**${PropertyName}** 構文を使用してロケーションリゾルバーに埋め込むことができます。たとえば、Red Hat Fuse のルートディレクトリーが Java システムプロパティー **karaf.home** に保存されている場合、以下のようにそのディレクトリーの値をファイルロケーションに埋め込むことができます。

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

O/S 環境変数は、**${env:VarName}** 構文を使用してロケーションリゾルバーに埋め込むことができます。たとえば、Fuse のルートディレクトリーが環境変数 **SMX_HOME** に保存されている場合、以下のようにそのディレクトリーの値をファイルロケーションに埋め込むことができます。

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

Properties コンポーネントの設定

プロパティープレースホルダーの使用を開始する前に、1 つ以上のプロパティーファイルのロケーションを指定して、Properties コンポーネントを設定する必要があります。

Java DSL では、以下のように Properties コンポーネントにプロパティーファイルのロケーションを設定できます。

```
// 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);
```

addComponent() の呼び出しに示されているように、Properties コンポーネントの名前を **properties** に設定する **必要があります**。

XML DSL では、以下のように専用の **propertyPlacholder** 要素を使用して Properties コンポーネントを設定できます。

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

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

Properties コンポーネントの初期化時に見つからない **.properties** ファイルを、Properties コンポーネントに無視させる場合は、**ignoreMissingLocation** オプションを **true** に設定します (通常、**.properties** ファイルが見つからない場合はエラーが発生します)。

また、Java システムプロパティーまたは O/S 環境変数を使用して、指定されたロケーションが見つからないときに Properties コンポーネントが無視するようにする場合も、**ignoreMissingLocation** オプションを **true** に設定することができます。

プレースホルダー構文

Properties コンポーネントは、設定後は (適切なコンテキストで) プレースホルダーを自動的に置き換えます。プレースホルダーの構文は、以下のようにコンテキストによって異なります。

- **エンドポイント URI および Spring XML ファイル**: プレースホルダーは **{{Key}}** のように指定します。
- **XML DSL の属性設定時**: **xs:string** 属性は以下の構文で設定します。

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

  その他の属性タイプ (**xs:int** または **xs:boolean** など) は、以下の構文を使用して設定する必要があります。

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

  prop は、**http://camel.apache.org/schema/placeholder** 名前空間に関連付けられています。
- **Java DSL の EIP オプション設定時**: Java DSL でエンタープライズ統合パターン (EIP) コマンドにオプションを設定するには、流れるような DSL に以下の **placeholder()** 句を追加します。

  ```
  .placeholder("OptionName", "Key")
  ```
- **Simple 言語式**: プレースホルダーは **${properties:Key}**. のように指定します。

エンドポイント URI 内での置換

ルートの中でエンドポイント URI 文字列が現れると、そのエンドポイント URI を構文解析する最初のステップは、常にプロパティープレースホルダーパーサーを適用することです。プレースホルダーパーサーは、二重かっこ **{{Key}}** の間に表示されるプロパティー名を自動的に置換します。たとえば、例 2.4「プロパティーファイルの例」 にあるプロパティー設定では、以下のようにルートを定義できます。

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

デフォルトでは、プレースホルダーパーサーはレジストリーから **properties** Bean ID を検索し、Properties コンポーネントを検索します。必要であれば、エンドポイント URI でスキーマを明示的に指

定できます。たとえば、各エンドポイント URI に接頭辞 **properties:** を付けて、以下のように同等のルートを定義できます。

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

スキーマを明示的に指定する場合、Properties コンポーネントにオプションを指定することもできます。たとえば、プロパティーファイルのロケーションを上書きするために、以下のように **location** オプションを設定できます。

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

Spring XML ファイル内での置換

XML DSL で DSL 要素のさまざまな属性を設定するために、プロパティープレースホルダーを使用することもできます。このコンテキストにおいても、プレースホルダー構文には二重かっこ **{{Key}}** を使用します。たとえば、以下のようにプロパティープレースホルダーを使用して **jmxAgent** 要素を定義できます。

```
<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>
```

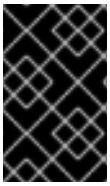
XML DSL 属性値の置換

xs:string 型の属性値を指定するには、通常のプレースホルダー構文を使用できます (例: **<jmxAgent registryPort="{{myjmx.port}}" ...>**)。しかし、他の型の属性 (例: **xs:int** や **xs:boolean**) については、特別な構文 **prop:AttributeName="Key"** を使用する必要があります。

たとえば、プロパティーファイルで **stop.flag** プロパティーの値が **true** に定義されている場合、このプロパティーを使用して以下のように **stopOnException** ブール値属性を設定できます。

```
<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 接頭辞は、前述の例の **beans** 要素に示されるように、Spring ファイルの **http://camel.apache.org/schema/placeholder** 名前空間に明示的に割り当てられている必要があります。

Java DSL における EIP オプションの置換

Java DSL で EIP コマンドを呼び出す場合は、**placeholder("OptionName", "Key")** の形式のサブ句を追加することで、プロパティープレースホルダーの値を使用した EIP オプションの設定ができます。

たとえば、プロパティーファイルで **stop.flag** プロパティーの値が **true** に定義されている場合、このプロパティーを使用して以下のように Multicast EIP の **stopOnException** オプションを設定できます。

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

Simple 言語式内での置換

Simple 言語の式の中でプロパティープレースホルダーを置換することもできますが、この場合プレースホルダーの構文は **${properties:Key}** になります。たとえば、以下のようにして Simple 式内の **cheese.quote** プレースホルダーを置換できます。

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

構文 **${properties:Key:DefaultVal}** を使用すると、プロパティーのデフォルト値を指定できます。以下に例を示します。

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

構文 **${properties-location:Location:Key}** を使用して、プロパティーファイルのロケーションをオーバーライドすることもできます。たとえば、**com/mycompany/bar.properties** プロパティーファイルの設定を使用して、**bar.quote** プレースホルダーを置き換えるには、以下のように Simple 式を定義します。

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

XML DSL 内でのプロパティープレースホルダーの使用

以前のリリースでは、 XML DSL のプレースホルダーをサポートするために **xs:string** 型の属性が使用されていました。たとえば、timeout 属性は **xs:int** 型になります。したがって、文字列の値をプレースホルダーキーとして設定することはできませんでした。

Apache Camel 2.7 以降、特別なプレースホルダーの名前空間を使用することでそれが可能になりました。以下の例は、その名前空間を使うための prop 接頭辞を示しています。これにより、XML DSL の属性に prop 接頭辞を付けて使用できます。

注記

Multicast において、キー **stop** を使用してプレースホルダーの値をオプション **stopOnException** に設定しています。また、プロパティーファイルの中で以下の値を定義しています。

```
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>
```

OSGi Blueprint プロパティープレースホルダーとの統合

Red Hat Fuse の OSGi コンテナーにルートをデプロイする場合、Apache Camel プロパティープレースホルダーのメカニズムを Fuse が持つ Blueprint プロパティープレースホルダーのメカニズムと統合できます (実際には、この統合はデフォルトで有効になっています) 。統合のセットアップには、基本的に以下の 2 つの方法があります。

- 暗黙的な Blueprint の統合
- 明示的な Blueprint の統合

暗黙的な Blueprint の統合

OSGi Blueprint ファイル内で **camelContext** 要素を定義すると、Apache Camel プロパティープレースホルダーのメカニズムは自動的に Blueprint プロパティープレースホルダーのメカニズムと統合します。つまり、**camelContext** のスコープ内に現れる Apache Camel 構文に従ったプレースホルダー (例: **{{cool.end}}**) は、暗黙的に **blueprint property placeholder** のメカニズムを検索することで解決されます。

たとえば、OSGi Blueprint ファイルで定義された以下のようなルートがあるとします。ルートの最後のエンドポイントは、プロパティープレースホルダー **{{result}}** で定義されています。

```
<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>
```

Blueprint プロパティープレースホルダーのメカニズムは、**cm:property-placeholder** Bean を作成することで初期化されます。上記の例では、**cm:property-placeholder** Bean は **camel.blueprint** 永続化 ID に関連付けられています。永続化 ID は、**OSGi Configuration Admin** サービスから関連するプロパティーのグループを参照する標準的な方法です。つまり、**cm:property-placeholder** Bean は、**camel.blueprint** 永続化 ID の下で定義されたすべてのプロパティーへのアクセスを提供します。一部のプロパティーにデフォルト値を指定することもできます (ネストされた **cm:property** 要素を使用します)。

Blueprint のコンテキストでは、Apache Camel プレースホルダーのメカニズムは Bean レジストリー内の **cm:property-placeholder** インスタンスを検索します。このインスタンスが見つかると、Apache Camel プレースホルダーのメカニズムと自動的に統合され、**{{result}}** のようなプレースホルダーは Blueprint プロパティープレースホルダーのメカニズムに対して (この例では **myblueprint.placeholder** Bean を通して) キーを検索することで解決されます。

注記

デフォルトの Blueprint プレースホルダー構文 (Blueprint プロパティーに直接アクセスする) は **${Key}** です。そのため、**camelContext** 要素の **範囲外** では、使用しなければならないプレースホルダー構文は **${Key}** になります。しかし、**camelContext** 要素の **範囲内** では、使用しなければならないプレースホルダー構文は **{{Key}}** になります。

明示的な Blueprint の統合

Apache Camel プロパティープレースホルダーのメカニズムがプロパティーを探す場所をさらに制御する場合は、**propertyPlaceholder** 要素を定義してリゾルバーのロケーションを明示的に指定できます。

たとえば、以下の Blueprint の設定について考えるとします。この例は、明示的に **propertyPlaceholder** インスタンスを作成している点が前述の例とは異なっています。

```
<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>
```

前述の例では、**propertyPlaceholder** 要素は、場所を **blueprint:myblueprint.placeholder** に設定することにより、使用する **cm:property-placeholder** Bean を明示的に指定します。つまり、**blueprint:** リゾルバーは、**cm:property-placeholder** Bean の ID **myblueprint.placeholder** を明示的に参照します。

このスタイルによる設定は、Blueprint ファイルに複数の **cm:property-placeholder** Bean が定義されていて、どれを使用すべきかを指定する必要がある場合に有効です。また、ロケーションをコンマ区切りのリストで指定することで、複数のロケーションからプロパティーを取得することも可能になります。たとえば、**cm:property-placeholder** Bean とクラスパス上のプロパティーファイル **myproperties.properties** の両方からプロパティーを検索する場合、以下のように **propertyPlaceholder** 要素を定義します。

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

Spring プロパティープレースホルダーとの統合

Spring XML ファイルの XML DSL を使用して Apache Camel アプリケーションを定義している場合、**org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer** 型の Spring Bean を宣言することで、Apache Camel プロパティープレースホルダーのメカニズムを Spring プロパティープレースホルダーのメカニズムと統合できます。

BridgePropertyPlaceholderConfigurer を定義します。これは、Spring XML ファイルの Apache Camel の **propertyPlaceholder** 要素と Spring の **ctx:property-placeholder** 要素の両方を置き換えます。その後、Spring の **${PropName}** 構文または Apache Camel の **{{PropName}}** 構文のいずれかを使用して、設定したプロパティーを参照できます。

たとえば、**cheese.properties** ファイルからプロパティー設定を読み取るブリッジプロパティープレースホルダーを定義するとします。

```
<?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>
```

注記

または、Spring プロパティーファイルを指すように、**BridgePropertyPlaceholderConfigurer** の **location** 属性を設定することもできます。Spring プロパティーファイルの構文は完全にサポートされます。

2.8. スレッドモデル

Java スレッドプール API

Apache Camel のスレッドモデルは、強力な Java 並行処理 API パッケージ java.util.concurrent に基づいています。この API は、Sun の JDK 1.5 で初めて利用可能になったものです。この API の主要なインターフェイスは、スレッドプールを表す **ExecutorService** インターフェイスです。並行処理 API を使用すると、幅広いシナリオに対応する、さまざまな種類のスレッドプールを作成できます。

Apache Camel スレッドプール API

Apache Camel スレッドプール API は Java 並行処理 API 上で構築されており、Apache Camel アプリケーションのすべてのスレッドプールに対して中心的なファクトリー (**org.apache.camel.spi.ExecutorServiceManager** 型) が提供されます。このような方法でスレッドプールの作成を一元化することには、以下のようにいくつかの利点があります。

- ユーティリティークラスを使用することで、スレッドプールの作成を簡素化できる。
- スレッドプールを正常なシャットダウンに統合できる。
- スレッドに有益な名前を自動的に付与できる。これは、ロギングと管理に役立ちます。

コンポーネントのスレッドモデル

SEDA、JMS、Jetty など、Apache Camel コンポーネントには本質的にマルチスレッドなものがあります。こうしたコンポーネントはすべて Apache Camel のスレッドモデルとスレッドプール API を使用して実装されています。

独自の Apache Camel コンポーネントを実装しようとしている場合、マルチスレッドなコードは

Apache Camel のスレッドモデルと統合することが推奨されます。たとえば、コンポーネントにスレッドプールが必要な場合は、CamelContext の **ExecutorServiceManager** オブジェクトを使用して作成することが推奨されます。

プロセッサーのスレッドモデル

Apache Camel の標準プロセッサーの中には、デフォルトで独自のスレッドプールを作成するものがあります。これらのスレッド対応プロセッサーも Apache Camel のスレッドモデルと統合されており、使用するスレッドプールのカスタマイズを可能にするさまざまなオプションを提供しています。

表2.8「プロセッサーのマルチスレッドオプション」 では、Apache Camel に組み込まれているスレッド対応のプロセッサーでスレッドプールを制御および設定するためのさまざまなオプションをまとめています。

表2.8 プロセッサーのマルチスレッドオプション

プロセッサー	Java DSL	XML DSL
aggregate	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
multicast	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
recipientList	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
split	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
threads	executorService() executorServiceRef() poolSize() maxPoolSize() keepAliveTime() timeUnit() maxQueueSize() rejectedPolicy()	@executorServiceRef @poolSize @maxPoolSize @keepAliveTime @timeUnit @maxQueueSize @rejectedPolicy

プロセッサー	Java DSL	XML DSL
wireTap	wireTap(String uri, ExecutorService executorService) wireTap(String uri, String executorServiceRef)	@executorServiceRef

threads DSL オプション

threads プロセッサーは汎用の DSL コマンドであり、ルートにスレッドプールを導入するために使用できます。スレッドプールをカスタマイズするために、以下のオプションをサポートしています。

poolSize()

プールの最小スレッド数 (および初期プールサイズ)。

maxPoolSize()

プールの最大スレッド数。

keepAliveTime()

スレッドがこの期間 (秒単位で指定) よりも長い間アイドル状態になっている場合、スレッドを終了させる。

timeUnit()

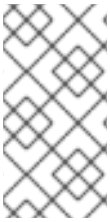
keep alive の時間単位。**java.util.concurrent.TimeUnit** タイプを使用して指定します。

maxQueueSize()

このスレッドプールが受信タスクキューに保持できる保留中の最大タスク数。

rejectedPolicy()

受信タスクキューが満杯の場合に実行すべきアクションを指定する。表2.10「スレッドプールビルダーのオプション」 を参照してください。



注記

前述のスレッドプールのオプションは、**executorServiceRef** オプションと互換性が **ありません** (たとえば、これらのオプションを使用して、**executorServiceRef** オプションで参照されるスレッドプールの設定を上書きすることはできません)。この DSL の制約は Apache Camel が検証することで強制されます。

デフォルトのスレッドプールの作成

スレッド対応プロセッサーに対してデフォルトのスレッドプールを作成するには、**parallelProcessing** オプションを有効にします。Java DSL では **parallelProcessing()** サブ句を、XML DSL では **parallelProcessing** 属性を使用します。

たとえば、Java DSL では、以下のようにデフォルトのスレッドプールを使用して multicast プロセッサーを呼び出すことができます (スレッドプールはマルチキャストの宛先を同時に処理するために使用されます)。

■

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

以下のように XML DSL で同じルートを定義できます。

```
<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>
```

デフォルトスレッドプールプロファイルの設定

デフォルトのスレッドプールは、スレッドファクトリーが**デフォルトスレッドプールプロファイル** から設定を取得することによって自動的に作成されます。デフォルトスレッドプールプロファイルが持つ設定は、 表2.9「デフォルトスレッドプールプロファイルの設定」 に示す通りです (これらの設定はアプリケーションコードによって変更されていないことを前提とします)。

表2.9 デフォルトスレッドプールプロファイルの設定

スレッドオプション	デフォルト値
maxQueueSize	**1000**
poolSize	**10**
maxPoolSize	**20**
keepAliveTime	**60 (秒)**
rejectedPolicy	**CallerRuns**

デフォルトスレッドプールプロファイルの変更

デフォルトスレッドプールプロファイルの設定を変更することで、後続のすべてのデフォルトスレッドプールをカスタムの設定で作成することができます。プロファイルは Java または Spring XML のどちらでも変更できます。

たとえば、Java DSL では、以下のようにデフォルトスレッドプールプロファイルの **poolSize** オプションと **maxQueueSize** オプションをカスタマイズできます。

```
// 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);
...
```

XML DSL では、以下のようにデフォルトスレッドプールプロファイルをカスタマイズできます。

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

前述の XML DSL の例では **defaultProfile** 属性を **true** に設定することが不可欠です。そうしないと、そのスレッドプールプロファイルはデフォルトスレッドプールプロファイルを置き換えるのではなく、カスタムのスレッドプールプロファイルとして扱われてしまいます (「カスタムスレッドプールプロファイルの作成」 を参照)。

プロセッサーのスレッドプールのカスタマイズ

executorService または **executorServiceRef** オプションのいずれかを使用 (**parallelProcessing** オプションの代わりにこれらのオプションを使用) することで、スレッド対応のプロセッサーにスレッドプールを直接指定することもできます。以下のように、プロセッサーのスレッドプールをカスタマイズするには 2 つの方法があります。

- **カスタムスレッドプールの値の指定** - **ExecutorService** (スレッドプール) インスタンスを明示的に作成し、これを **executorService** オプションに渡します。
- **カスタムスレッドプールプロファイルの指定** - カスタムスレッドプールファクトリーを作成して登録します。**executorServiceRef** オプションを使用してこのファクトリーを参照すると、プロセッサーは自動的にそのファクトリーを使用して、カスタムスレッドプールインスタンスを作成します。

Bean ID を **executorServiceRef** オプションに渡すと、スレッド対応のプロセッサーはまずその ID を持つカスタムスレッドプールをレジストリーの中から検索します。その ID でスレッドプールが登録されていない場合、プロセッサーはレジストリーの中からカスタムスレッドプールプロファイルを検索し、そのカスタムスレッドプールプロファイルを使用してカスタムスレッドプールをインスタンス化します。

カスタムスレッドプールの作成

カスタムスレッドプールは、java.util.concurrent.ExecutorService 型の任意のスレッドプールです。Apache Camel では、スレッドプールインスタンスを作成する上で以下の方法が推奨されています。

- **org.apache.camel.builder.ThreadPoolBuilder** ユーティリティーを使用して、スレッドプールクラスをビルドします。

- 現在の **CamelContext** から **org.apache.camel.spi.ExecutorServiceManager** インスタンスを使用して、スレッドプールクラスを作成します。

ThreadPoolBuilder は実際には **ExecutorServiceManager** インスタンスを使用して定義されているため、究極的には 2 つのアプローチには大きな違いはありません。通常、**ThreadPoolBuilder** が推奨されます。これは、より単純なアプローチを提供するためです。ただし、少なくとも 1 種類のスレッド (**ScheduledExecutorService**) は、**ExecutorServiceManager** インスタンスに直接アクセスする方法でしか作成できません。

表2.10「スレッドプールビルダーのオプション」 は、**ThreadPoolBuilder** クラスがサポートするオプションを示します。これらのオプションは、新しいカスタムスレッドプールを定義する際に設定できます。

表2.10 スレッドプールビルダーのオプション

Builder オプション	説明
maxQueueSize()	このスレッドプールが受信タスクキューに保持できる保留中の最大タスク数を設定します。**-1** の値は上限なしキューを指定します。デフォルト値はデフォルトスレッドプールプロファイルから取得されます。
poolSize()	プールの最小スレッド数を設定します (これは初期プールサイズにもなります)。デフォルト値はデフォルトスレッドプールプロファイルから取得されます。
maxPoolSize()	プールで使用できる最大スレッド数を設定します。デフォルト値はデフォルトスレッドプールプロファイルから取得されます。
keepAliveTime()	スレッドがこの期間 (秒単位で指定) よりも長い間アイドル状態になっている場合、スレッドを終了させる。これにより、負荷が軽くなるとスレッドプールが縮小されます。デフォルト値はデフォルトスレッドプールプロファイルから取得されます。

Builder オプション	説明
rejectedPolicy()	受信タスクキューが満杯の場合に実行すべきアクションを指定する。以下の 4 つの値から指定できます。 **CallerRuns** **(デフォルト値)** 呼び出し元のスレッドを使用して、最後に受信したタスクを実行します。しかし、このオプションでは最後に受信したタスクの処理が完了するまで、呼び出し元のスレッドがそれ以上のタスク受信をブロックします。 **Abort** 例外を発生させて、最後に受信したタスクを中断します。 **Discard** 例外を発生させずに、最後に受信したタスクを破棄します。 **DiscardOldest** 最も古い未処理のタスクを破棄して、最後に受信したタスクをタスクキューに入れようと試みます。
build()	カスタムスレッドプールの構築を終了し、**build()** に引数として指定された ID の下に新しいスレッドプールを登録します。

Java DSL では、以下のように **ThreadPoolBuilder** を使用してカスタムスレッドプールを定義できます。

```
// 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");
```

オブジェクト参照 **customPool** を直接 **executorService()** オプションに渡す代わりに、以下のように Bean ID を **executorServiceRef()** オプションに渡すことで、レジストリーの中からスレッドプールを検索できます。

```
// Java
from("direct:start")
  .multicast().executorServiceRef("customPool")
```

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

XML DSL では、**threadPool** 要素を使用して **ThreadPoolBuilder** にアクセスします。その後、以下のように **executorServiceRef** 属性を使用して Spring レジストリーでスレッドプールを ID で検索することで、カスタムスレッドプールを参照できます。

```
<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>
```

カスタムスレッドプールプロファイルの作成

多くのカスタムスレッドプールインスタンスを作成する場合は、スレッドプールのファクトリーとして機能するカスタムスレッドプールプロファイルを定義しておくと便利です。スレッド対応プロセッサーからスレッドプールプロファイルを参照するだけで、プロセッサーは自動的にそのプロファイルを使用して新しいスレッドプールインスタンスを作成します。カスタムスレッドプールプロファイルは、Java DSL または XML DSL のどちらでも定義できます。

たとえば、Java DSL では、以下のようにして Bean ID **customProfile** を持つカスタムスレッドプールプロファイルを作成し、ルート内でそのプロファイルを参照できます。

```
// 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");
```

XML DSL では、**threadPoolProfile** 要素を使用してカスタムプールプロファイルを作成します (これはデフォルトスレッドプールプロファイル **ではない** ため、ここでは **defaultProfile** オプションをデフォ

ルトで **false** に設定します)。以下のようにして、Bean ID **customProfile** を持つカスタムスレッドプールプロファイルを作成し、ルート内でそのプロファイルを参照できます。

```
<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>
```

コンポーネント間でのスレッドプールの共有

File や FTP など、標準のポーリングベースのコンポーネントの中には、使用するスレッドプールを指定できるものがあります。これにより、異なるコンポーネントが同じスレッドプールを共有することが可能となり、JVM 内のスレッド総数を削減することができます。

たとえば、Apache Camel Component Reference Guide の File2 と Apache Camel Component Reference Guide の Ftp2 は、どちらも **scheduledExecutorService** プロパティーを公開しており、コンポーネントの **ExecutorService** オブジェクトを指定するために使用できます。

スレッド名のカスタマイズ

アプリケーションのログをより読みやすくするために、スレッド名 (ログ内でスレッドを識別するために使用されるもの) をカスタマイズすることが推奨されます。スレッド名をカスタマイズするには、**ExecutorServiceStrategy** クラスまたは **ExecutorServiceManager** クラスの **setThreadNamePattern** メソッドを呼び出すことで、**スレッド名パターン** を設定します。または、スレッド名パターンを設定するより簡単な方法として、**CamelContext** オブジェクトに **threadNamePattern** プロパティーを設定する方法もあります。

スレッド名パターンでは、以下のプレースホルダーが使用できます。

#camelId#

現在の **CamelContext** の名前。

#counter#

インクリメントカウンターとして実装された一意のスレッド ID。

#name#

通常の Camel スレッド名。

#longName#

長いスレッド名。エンドポイントパラメーターなどを含めることができる。

以下は、スレッド名パターンの典型的な例です。

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

以下の例は、XML DSL を使用して Camel コンテキストに **threadNamePattern** 属性を設定する方法を示しています。

```
<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. ルートの起動およびシャットダウンの制御

概要

デフォルトでは、Apache Camel アプリケーション (**CamelContext** インスタンスで表される) の起動時にルートが自動的に起動し、Apache Camel アプリケーションのシャットダウン時にルートは自動的にシャットダウンします。クリティカルでないデプロイメントについては、シャットダウン順序の詳細は通常それほど重要ではありません。しかし、本番環境では、データ損失を回避するために、シャットダウン時に残っているタスクを完了させることが重要になります。また、通常、依存関係の順序違反 (実行中のタスクが完了できなくなる) を防止するために、ルートがシャットダウンする順番を制御したくなることもあります。

このため、Apache Camel はアプリケーションの **正常なシャットダウン** をサポートする機能を提供しています。正常なシャットダウンにより、ルートの停止と起動を完全に制御し、ルートのシャットダウン順序を制御し、現在実行中のタスクを完了まで実行できるようになります。

ルート ID の設定

ルート ID を各ルートに割り当てるのが良いプラクティスです。ルート ID を設定すると、ロギングメッセージや管理機能がよりわかりやすくなるだけでなく、ルートの停止と起動の制御がより行いやすくなります。

たとえば、Java DSL では、以下のように **routeId()** コマンドを実行して、ルート ID **myCustomerRouteId** をルートに割り当てることができます。

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

XML DSL では、以下のように **route** 要素の **id** 属性を設定します。

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

ルートの自動起動の無効化

デフォルトでは、起動時に CamelContext が認識しているすべてのルートは自動的に起動されます。しかし、特定のルートの起動を手動で制御する場合は、そのルートの自動起動を無効にできます。

Java DSL ルートが自動的に起動するかどうかを制御するには、**boolean** 引数 (**true** または **false**) か、**String** 引数 (**true** または **false**) のいずれかで、**autoStartup** コマンドを呼び出します。たとえば、以下のように Java DSL でルートの自動起動を無効にできます。

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

XML DSL では、以下のように **route** 要素の **autoStartup** 属性を **false** に設定して、ルートの自動起動を無効にできます。

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

手動によるルートの起動および停止

Java において、**CamelContext** インスタンスの **startRoute()** および **stopRoute()** メソッドを呼び出すことにより、ルートを手動でいつでも起動または停止できます。たとえば、ルート ID **nonAuto** を持つルートを開始するには、以下のように **CamelContext** インスタンス **context** で **startRoute()** メソッドを呼び出します。

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

ルート ID **nonAuto** を持つルートを停止するには、以下のように **CamelContext** インスタンス **context** で **stopRoute()** メソッドを呼び出します。

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

ルートの起動順序

デフォルトでは、Apache Camel はルートを非決定論的な順序で起動します。しかし、アプリケーションによっては、起動順序を制御することが重要になる場合があります。Java DSL で起動順序を制御するには、**startupOrder()** コマンドを使用します。このコマンドは、正の整数値を引数として取ります。整数値が最も小さいルートが最初に起動し、それ以降、起動順序の値が小さいものから順番にルートが起動します。

たとえば、以下の例では最初の 2 つのルートが **seda:buffer** エンドポイントを通して繋ぎ合わされています。以下のように起動順序 (それぞれ 2 と 1) を割り当てることで、最初のルートセグメントを 2 つ目のルートセグメントの **後に** 起動させることができます。

例2.5 Java DSL の起動順序

```
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");
```

または、Spring XML では、以下のように **route** 要素の **startupOrder** 属性を設定することで、同様の効果を得ることができます。

例2.6 XML DSL の起動順序

```
<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>
```

各ルートには、**一意の** 起動順序の値を割り当てる必要があります。値は 1000 未満の正の整数から選択できます。1000 以上の値は Apache Camel 用に予約されており、明示的な起動値を持たないルートにこれらの値が自動的に割り当てられます。たとえば、前述の例の最後のルートは自動的に起動値 1000 が割り当てられます (これにより最初の 2 つのルートの後に起動することになります)。

シャットダウンシーケンス

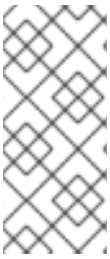
CamelContext インスタンスがシャットダウンしているとき、Apache Camel はプラグ可能な **シャットダウンストラテジー** を使用してシャットダウンシーケンスを制御します。デフォルトのシャットダウンストラテジーは、以下のシャットダウンシーケンスを実装します。

1. ルートが起動順序の **逆順** でシャットダウンされる。
2. 通常、シャットダウンストラテジーは、現在アクティブなエクスチェンジの処理が終了するまで待機する。ただし、実行中タスクの取り扱いは設定可能。
3. 全体的なシャットダウンシーケンスには、タイムアウトの上限がある (デフォルトは 300 秒)。シャットダウンシーケンスがこのタイムアウトを超えると、シャットダウンストラテジーは、一部のタスクが実行中であっても強制的にシャットダウンを実行する。

ルートのシャットダウン順序

ルートは起動順序の逆順でシャットダウンされます。つまり、起動順序が **startupOrder()** コマンド (Java DSL) または **startupOrder** 属性 (XML DSL) を使用して定義されている場合、最初にシャットダウンするルートは、起動順として割り当てられた **最大の** 整数値を持つルートであり、最後にシャットダウンするルートは、起動順として割り当てられた **最小の** 整数値を持つルートになります。

たとえば、例2.5「Java DSL の起動順序」 では、最初にシャットダウンするルートセグメントは ID **first** のルートで、2 番目にシャットダウンするルートセグメントは ID **second** のルートになります。この例は、ルートをシャットダウンする際に守るべき一般的なルールを示しています。つまり、**外部からアクセス可能なコンシューマーエンドポイントを公開するルートは最初にシャットダウンする必要がある**、ということです。これは、残りのルートグラフを流通するメッセージのフローを調整するスロットルとなるためです。



注記

Apache Camel は、オプション **shutdownRoute(Defer)** も提供しています。これにより、ルートを (起動順の値を上書きして) 最後にシャットダウンするように指定できます。ただし、このオプションが必要になることはほとんどありません。このオプションは主に、ルートが起動順序と **同じ** 順序でシャットダウンしていた Apache Camel の以前のバージョン (2.3 以前) への回避策として必要だったものです。

ルート内で実行中のタスクのシャットダウン

シャットダウンの開始時にルートが依然としてメッセージを処理している場合、シャットダウンストラテジーは通常、現在アクティブなエクスチェンジの処理が終了するまで待機してからルートをシャットダウンします。この動作は、ルート毎に **shutdownRunningTask** オプションを使用することで設定できます。このオプションは以下のいずれかの値を取ります。

ShutdownRunningTask.CompleteCurrentTaskOnly

(デフォルト) 通常、ルートは一度に 1 つのメッセージのみを処理するため、現在のタスクが完了した後にルートを安全にシャットダウンできます。

ShutdownRunningTask.CompleteAllTasks

バッチコンシューマー を正常にシャットダウンするには、このオプションを指定します。一部のコンシューマーエンドポイント (File、FTP、Mail、iBATIS、JPA など) は一度に複数のメッセージを一括して処理します。これらのエンドポイントでは、現在のバッチのすべてのメッセージが完了するまで待機することが推奨されます。

たとえば、File コンシューマーエンドポイントを正常にシャットダウンするには、以下の Java DSL フラグメントのように **CompleteAllTasks** オプションを指定する必要があります。

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

同じルートを XML DSL では以下のように定義できます。

```
<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>
```

シャットダウンのタイムアウト

シャットダウンタイムアウトのデフォルト値は 300 秒です。シャットダウンストラテジーの **setTimeout()** メソッドを呼び出すことで、タイムアウトの値を変更できます。たとえば、以下のようにタイムアウト値を 600 秒に変更できます。

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

カスタムコンポーネントとの統合

カスタムの Apache Camel コンポーネント (同じく **org.apache.camel.Service** インターフェイスから継承する) を実装している場合、**org.apache.camel.spi.ShutdownPrepared** インターフェイスを実装することで、カスタムコードがシャットダウン通知を受け取るようにすることができます。これにより、コンポーネントはシャットダウンに備えてカスタムコードを実行できるようになります。

2.9.1. RouteIdFactory

コンシューマーエンドポイントに基づいて、ルート ID に論理名を割り当てられる **RouteIdFactory** を追加することができます。

たとえば、seda または direct コンポーネントをルートの入力として持つルートを使用するとき、以下のようにそれらの名前をルート ID として使用する場合があります。

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

自動的に割り当てられた名前を使用する代わりに、**NodeIdFactory** を用いてルートに論理名を割り当てることができます。また、ルート URL の context-path を名前として使用することもできます。たとえば、以下を実行して **RouteIDFactory** を使用します。

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

注記

REST エンドポイントからカスタムルート ID を取得することもできます。

2.10. 定期実行ルートポリシー

2.10.1. 定期実行ルートポリシーの概要

概要

定期実行ルートポリシーは、実行時にルートに影響するイベントをトリガーすることができます。特に、現在利用可能な実装では、ポリシーで指定された任意の時刻 (または複数時刻) にルートを開始、停止、中断、または再開することができます。

タスクのスケジューリング

定時実行ルートポリシーは、次のようなイベントを発生させることができます。

- **ルートの開始**: 指定した時刻 (または複数時刻) でルートを開始します。このイベントは、ルートが現在停止状態にあり、起動を待っている場合にのみ有効になります。
- **ルートの停止**: 指定した時刻 (または複数時刻) でルートを停止します。このイベントは、ルートが現在アクティブの場合にのみ有効になります。
- **ルートの一時停止**: ルートの先頭にあるコンシューマーエンドポイントを (**from()** で指定したように) 一時的に非アクティブにします。ルートの残りの部分はアクティブですが、クライアントはルートに新しいメッセージを送信することはできません。
- **ルートの再開** : ルートの先頭にあるコンシューマーエンドポイントを再アクティブにし、ルートを完全にアクティブな状態に戻します。

Quartz コンポーネント

Quartz コンポーネントは、ジョブスケジューラーのオープンソース実装である Terratania の Quartz をベースにした Timer コンポーネントです。Quartz コンポーネントは、単純な定期実行ルートポリシーと cron 定期実行ルートポリシーの両方の基礎となる実装を提供しています。

2.10.2. 単純な定期実行ルートポリシー

概要

定期実行ルートポリシーは、ルートの開始、停止、一時停止、および再開を可能にするルートポリシーで、これらのイベントのタイミングは、初回イベントの発生時刻で指定し、(オプションで) その後の繰り返し回数も指定できます。単純な定期実行ルートポリシーを定義するには、以下のクラスのインスタンスを作成します。

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

依存関係

単純な定期実行ルートポリシーは、Quartz コンポーネント **camel-quartz** に依存しています。たとえば、Maven をビルドシステムとして使用する場合は、アーティファクト **camel-quartz** の依存関係を追加してください。

Java DSL の例

例2.7「単純な定期実行ルートの Java DSL の例」 は、Java DSL を使用して起動するルートをスケジュールする方法を示しています。初回の起動時刻 **startTime** は、現在時刻から 3 秒後に設定されています。また、このポリシーは、初期起動時刻の 3 秒後に、2 回目のルートを開始するように設定されています。これは、**routeStartRepeatCount** を 1 に設定し、**routeStartRepeatInterval** を 3000 ミリ秒に設定することで設定されます。

Java DSL では、ルート内で **routePolicy()** DSL コマンドを呼び出して、ルートポリシーをルートにアタッチします。

例2.7 単純な定期実行ルートの Java DSL の例

```
// 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");
```

注記

複数の引数を指定して **routePolicy()** を呼び出すことで、ルート上に複数のポリシーを指定することができます。

XML DSL の例

例2.8「単純な定期実行ルートの XML DSL の例」 は、XML DSL を使用して起動するルートをスケジュールする方法を示しています。

XML DSL では、**route** 要素に **routePolicyRef** 属性を設定して、ルートポリシーをルートにアタッチします。

例2.8 単純な定期実行ルートの XML DSL の例

```
<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>
```

注記

routePolicyRef の値をコンマ区切りの Bean ID リストとして設定することで、ルート上に複数のポリシーを指定できます。

日付と時刻の定義

単純な定期実行ルートポリシーで使用されるトリガーの初動時刻は、**java.util.Date** タイプを使用して指定します。**Date** インスタンスを定義する最も柔軟な方法は、java.util.GregorianCalendar クラスを使用することです。**GregorianCalendar** クラスの便利なコンストラクターとメソッドを使用して日付を定義し、**GregorianCalendar.getTime()** を呼び出して **Date** インスタンスを取得します。

たとえば、2011 年 1 月 1 日正午の日時を定義するには、以下のように **GregorianCalendar** コンストラクターを呼び出します。

```
// 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 クラスは、異なるタイムゾーンの時間の定義もサポートします。デフォルトでは、コンピューターのローカルタイムゾーンを使用します。

正常シャットダウン

単純な定期実行ルートポリシーを設定してルートを停止すると、ルートの停止アルゴリズムが自動的に正常シャットダウンの手順に統合されます (「ルートの起動およびシャットダウンの制御」 を参照)。よって、タスクは現在のエクスチェンジが処理を完了するまで待機してから、ルートをシャットダウンします。ただし、タイムアウトを設定することで、ルートがエクスチェンジの処理を終了したかどうかにかかわらず、指定した時間後にルートを強制的に停止することができます。

タイムアウト時の処理中エクスチェンジのロギング

指定のタイムアウト期間内に正常シャットダウンが適切に行われなかった場合、Apache Camel はより強行なシャットダウンを実行します。ルートやスレッドプールなどを強制的にシャットダウンします。

タイムアウト後、Apache Camel は現在処理中のエクスチェンジの情報をログに記録します。エクスチェンジの元および現在のルートをログに記録します。

たとえば、以下のログは、1つの処理中のエクスチェンジがあり、その元のルートは route1 で、現在 delay1 ノードの同じ route1 にあることを表しています。

正常なシャットダウン中に、**org.apache.camel.impl.DefaultShutdownStrategy** で DEBUG ロギングレベルを有効にすると、同じインフライトエクスチェンジ情報がログに記録されます。

```
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]
```

これらのログを表示したくない場合は、**logInflightExchangesOnTimeout** オプションを false に設定してこれをオフにできます。

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

タスクのスケジューリング

単純な定期実行ルートポリシーを使用して、以下のスケジュールタスクのいずれかを定義することができます。

- ルートの開始
- ルートの停止
- ルートの一時停止
- ルートの再開

ルートの開始

次の表は、ルートの開始を1回以上スケジューリングするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明
routeStartDate	**java.util.Date**	**なし**	ルートの初回起動日時を指定します。
routeStartRepeatCount	**int**	**0**	0 以外の値に設定すると、ルートの開始回数が指定されます。
routeStartRepeatInterval	**long**	**0**	開始の間隔 (ミリ秒単位) を指定します。

ルートの停止

次の表は、ルートの停止を1回以上スケジューリングするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明
routeStopDate	**java.util.Date**	**なし**	ルートの初回停止日時を指定します。
routeStopRepeatCount	**int**	**0**	0 以外の値に設定すると、ルートの停止回数が指定されます。
routeStopRepeatInterval	**long**	**0**	停止の間隔 (ミリ秒単位) を指定します。
routeStopGracePeriod	**int**	**10000**	ルートを強制停止する前に、現在のエクスチェンジの処理が終了するまで待つ時間 (猶予期間) を指定します。猶予期間が無限の場合は 0 に設定します。
routeStopTimeUnit	**long**	**TimeUnit.MILLISECONDS**	猶予期間の時間単位を指定します。

ルートの一時停止

次の表は、ルートの一時停止を 1 回以上スケジュールするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明
routeSuspendDate	**java.util.Date**	**なし**	ルートが初めて一時停止される日時を指定します。
routeSuspendRepeatCount	**int**	0	0 以外の値に設定すると、ルートが一時停止される回数が指定されます。
routeSuspendRepeatInterval	**long**	0	一時停止の間隔 (ミリ秒単位) を指定します。

ルートの再開

次の表は、ルートの再開を 1 回以上スケジュールするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明

パラメーター	型	デフォルト	説明
routeResumeDate	**java.util.Date**	**なし**	ルートの初回再開日時を指定します。
routeResumeRepeatCount	**int**	0	0 以外の値に設定すると、ルートの再開回数が指定されます。
routeResumeRepeatInterval	**long**	0	再開の間隔をミリ秒単位で指定します。

2.10.3. cron 定期実行ルートポリシー

概要

cron 定期実行ルートポリシーは、cron 式で指定することで、ルートの開始、停止、一時停止、および再開を可能にするルートポリシーです。cron 定期実行ルートポリシーを定義するには、以下のクラスのインスタンスを作成します。

> org.apache.camel.routepolicy.quartz.CronScheduledRoutePolicy

依存関係

単純な定期実行ルートポリシーは、Quartz コンポーネント **camel-quartz** に依存しています。たとえば、Maven をビルドシステムとして使用する場合は、アーティファクト **camel-quartz** の依存関係を追加してください。

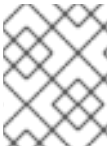
Java DSL の例

例2.9「cron 定期実行ルートの Java DSL の例」 は、Java DSL を使用して起動するルートをスケジュールする方法を示しています。このポリシーは、3 秒ごとに開始イベントをトリガーする cron 式 ***/3 * * * * ?** で設定されています。

Java DSL では、ルート内で **routePolicy()** DSL コマンドを呼び出して、ルートポリシーをルートにアタッチします。

例2.9 cron 定期実行ルートの Java DSL の例

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

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



注記

複数の引数を指定して **routePolicy()** を呼び出すことで、ルート上に複数のポリシーを指定することができます。

XML DSL の例

例2.10「cron 定期実行ルートの XML DSL の例」 は、XML DSL を使用して起動するルートをスケジュールする方法を示しています。

XML DSL では、**route** 要素に **routePolicyRef** 属性を設定して、ルートポリシーをルートにアタッチします。

例2.10 cron 定期実行ルートの XML DSL の例

```
<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>
```

注記

routePolicyRef の値をコンマ区切りの Bean ID リストとして設定することで、ルート上に複数のポリシーを指定できます。

cron 式の定義

cron 式 の構文は、UNIX システム上でバックグラウンドで実行するジョブをスケジュールする UNIX **cron** ユーティリティーに由来しています。cron 式は、日付と時刻にワイルドカードを使用することで、単一のイベントまたは定期的に繰り返される複数のイベントを効果的に指定することができる構文です。

cron 式は、以下の順序の 6 または 7 個のフィールドで設定されます。

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

Year フィールドは任意のフィールドで、一度だけ発生するイベントを定義する場合を除き、通常は省略できます。各フィールドは、リテラルと特殊文字の組み合わせで設定されます。たとえば、以下の cron 式は、毎日 1 回夜 12 時に発生するイベントを指定します。

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

***** 文字は、フィールドのすべての値にマッチするワイルドカードです。したがって、上記の式は毎月の毎日を意味します。**?** の文字は、フィールドの無視を意味するダミーのプレースホルダーです。**DayOfMonth** フィールド **または DayOfWeek** フィールドの両方を同時に指定するのは論理的では

ないので、常にどちらかのフィールドにこの文字を指定します。たとえば、1日1回発生するイベントを、月曜日から金曜日までのみスケジュールする場合は、以下の cron 式を使用します。

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

MON-FRI は、ハイフン文字で範囲を指定しています。/ (スラッシュ) を使用してインクリメントを指定することもできます。たとえば、5 分ごとにイベントが発生するように指定するには、以下の cron 式を使用します。

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

cron 式構文の完全な説明は、Wikipedia の CRON 式 に関する記事を参照してください。

タスクのスケジューリング

cron 定期実行ルートポリシーを使用して、以下のスケジューリングタスクのいずれかを定義することができます。

- ルートの開始
- ルートの停止
- ルートの一時停止
- ルートの再開

ルートの開始

次の表は、ルートの開始を 1 回以上スケジューリングするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明
routeStartString	**String**	**なし**	1 つ以上のルート開始イベントをトリガーする cron 式を指定します。

ルートの停止

次の表は、ルートの停止を 1 回以上スケジューリングするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明
routeStopTime	**String**	**なし**	1 つ以上のルート停止イベントをトリガーする cron 式を指定します。

パラメーター	型	デフォルト	説明
routeStopGracePeriod	**int**	**10000**	ルートを強制停止する前に、現在のエクスチェンジの処理が終了するまで待つ時間 (猶予期間) を指定します。猶予期間が無限の場合は 0 に設定します。
routeStopTimeUnit	**long**	**TimeUnit.MILLISECONDS**	猶予期間の時間単位を指定します。

ルートの一時停止

次の表は、ルートの一時停止を 1 回以上スケジュールするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明
routeSuspendTime	**String**	**なし**	1 つ以上のルート一時停止イベントをトリガーする cron 式を指定します。

ルートの再開

次の表は、ルートの再開を 1 回以上スケジュールするためのパラメーターを示しています。

パラメーター	型	デフォルト	説明
routeResumeTime	**String**	**なし**	1 つ以上のルート再開イベントをトリガーする cron 式を指定します。

2.10.4. ルートポリシーファクトリー

ルートポリシーファクトリーの使用

Camel 2.14 から利用可能

すべてのルートにルートポリシーを使用する場合は、**org.apache.camel.spi.RoutePolicyFactory** をファクトリーとして使用して、個々のルートに **RoutePolicy** インスタンスを作成することができます。これは、すべてのルートに同じ種類のルートポリシーを使用する場合に使用することができます。そうすれば、ファクトリーの設定は 1 度だけで済み、作成されたすべてのルートにポリシーが割り当てられます。

CamelContext には、以下のようなファクトリーを追加するための API が用意されています。

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

XML DSL では、ファクトリーで **<bean>** を定義するだけです。

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

ファクトリーには、ルートポリシーを作成するための createRoutePolicy メソッドが含まれます。

```
/**
 * 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);
```

ルートポリシーファクトリーはいくつでも持つことができます。**addRoutePolicyFactory** を再度呼び出すか、他のファクトリーを XML で **<bean>** と宣言します。

2.11. CAMEL ルートのリロード

Apache Camel 2.19 以降では、エディターから XML ファイルを保存したときに、Camel の XML ルートのライブリロードを有効にすることができます。この機能は、以下の実行方法で使用できます。

- Camel スタンドアロン (Camel Main クラスで実行)
- Camel Spring Boot (Spring Boot で実行)
- camel:run (maven プラグインで実行)

これ以外に、**CamelContext** に **ReloadStrategy** を設定したり、独自のカスタム戦略を設定したりすることで、手動で有効にすることもできます。

2.12. CAMEL MAVEN プラグイン

Camel Maven プラグインは以下のゴールをサポートします。

- camel:run - Camel アプリケーションを実行します。
- camel:validate - ソースコードを検証し、無効な Camel エンドポイント URI を検査します。
- camel:route-coverage - ユニットテストの実行後、Camel ルートのカバレッジを報告します。

2.12.1. camel:run

Camel Maven プラグインのゴール **camel:run** は、Maven からフォークされた JVM で Camel Spring 設定を実行するために使用されます。初めて使用する場合、アプリケーションサンプルとして Spring サンプルを使用するとよいでしょう。

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

これにより、main(...) メソッドを書かなくても、ルーティングルールを起動してテストすることが非常に容易になります。また、複数の jar を作成して、さまざまなルーティングルールのセットをホストし、それらを簡単に個別にテストすることもできます。Camel Maven プラグインは maven プロジェクトのソースコードをコンパイルし、**META-INF/spring/*.xml** のクラスパスの XML 設定ファイルを使用して Spring ApplicationContext を起動します。Camel のルートをもう少し速く起動する場合は、代わりに **camel:embedded** を試してみてください。

2.12.1.1. オプション

Camel Maven プラグインの **run** ゴールは、以下のオプションをサポートします。これらのオプションは、コマンドラインから設定するか (**-D** 構文を使用)、**<configuration>** タグの **pom.xml** ファイルで定義します。

パラメーター	デフォルト値	説明
duration	-1	アプリケーションが終了する前に実行される期間 (秒単位) を設定します。0 以下の値を指定すると、永久に実行されます。
durationIdle	-1	アプリケーションが終了する前にアイドル状態でいられる期間 (秒単位) を設定します。0 以下の値を指定すると、永久に実行されます。
durationMaxMessages	-1	アプリケーションが終了する前にアプリケーションが処理するメッセージ最大数の期間を設定します。
logClasspath	false	起動時にクラスパスをログに記録するかどうか。

2.12.1.2. OSGi Blueprint の実行

camel:run プラグインは、Blueprint アプリケーションの実行もサポートします。デフォルトでは、**OSGI-INF/blueprint/*.xml** の OSGi Blueprint ファイルがスキャンされます。以下のように useBlueprint を true に設定して、camel:run プラグインが Blueprint を使用するように設定する必要があります。

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

これにより、Camel 関連だけでなく、他の Blueprint サービスも起動することができます。**camel:run** ゴールは、camel-blueeprint がクラスパス上にあるか、またはプロジェクト内に blueeprint XML ファ

イルがある場合は、自動検出することができるので、**useBlueprint** オプションを設定する必要がありません。

2.12.1.3. 制限された Blueprint コンテナーの使用

Blueprint のコンテナーとして Felix Connector プロジェクトを使用しています。Felix は完全な Blueprint コンテナーではありません。完全な Blueprint コンテナーで実行する場合は、Apache Karaf または Apache ServiceMix を使用できます。**applicationContextUri** 設定を使用して、明示的な Blueprint XML ファイルを指定できます。例を以下に示します。

```
<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 はクラスパスからファイルをロードするので、上の例では **myBlueprint.xml** ファイルはクラスパスのルートになければなりません。**configAdminPid** は pid 名で、persistence プロパティーファイルを読み込む際に、設定管理サービスの pid 名として使用されます。**configAdminFileName** は、設定管理サービスのプロパティーファイルを読み込むために使用されるファイル名です。

2.12.1.4. CDI の実行

camel:run プラグインは、CDI アプリケーションの実行もサポートします。これにより、Camel 関連だけでなく、すべての CDI 対応サービスを起動できます。下記の例のように、CDI コンテナー (Weld や OpenWebBeans など) を camel-maven-plugin の依存関係に追加する必要があります。Camel のソースからは、以下のように CDI のサンプルを実行できます。

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

2.12.1.5. クラスパスのロギング

camel:run の実行時に、クラスパスをログに記録するかどうかを設定できます。以下のコマンドを使用して、この設定を有効できます。

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

2.12.1.6. XML ファイルのライブリロードの使用

XML ファイルの変更をスキャンし、それらの XML ファイルに含まれる Camel ルートのリロードをトリガーするように、プラグインを設定できます。

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

設定後、プラグインはこのディレクトリーの監視を開始します。エディターからソースコードを編集して保存すると、変更後の内容が実行中の Camel アプリケーションに適用されます。**<routes>** や **<route>** などの Camel ルートへの変更のみがサポートされることに注意してください。Spring や OSGi Blueprint の **<bean>** 要素を変更することはできません。

2.12.2. camel:validate

以下の Camel 機能のソースコード検証の場合

- エンドポイント URI
- Simple 式または述語
- ルート ID の重複

次に、コマンドラインまたは、IDEA や Eclipse などの Java エディターから、**camel:validate** ゴールを実行できます。

```
mvn camel:validate
```

また、プラグインを有効にしてビルドの一部として自動的に実行し、エラーを検出することも可能です。

```
<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>
```

フェーズは、プラグインがいつ実行されるかを決定します。上記の例では、メインのソースコードのコンパイル後に実行される **process-classes** がフェーズになります。この maven プラグインは、テストソースコードを検証するように設定することもできます。以下に示すように、フェーズを **process-test-classes** に合わせて変更してください。

```
<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. 任意の Maven プロジェクトでのゴール実行

プラグインを **pom.xml** ファイルに追加せずに Maven プロジェクトで validate ゴールを実行することもできます。この場合、完全修飾名を使用してプラグインを指定する必要があります。たとえば、Apache Camel から **camel-example-cdi** でゴールを実行するには、次のように実行します。

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

このコマンドを実行すると以下が出力されます。

```
[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] ------------------------------------------------------------------------
```

validate は成功し、4 つのエンドポイントが検証されます。ここで、ソースコードの Camel エンドポイント URI の 1 つに、以下のようなタイプミスがあったとします。

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

period オプションを以下のように変更し、タイプミスが含まれるようにします。

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

validate ゴールを再度実行すると、以下が報告されます。

```
[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. オプション

Camel Maven プラグインの **validate** ゴールは、以下のオプションをサポートします。これらのオプションは、コマンドラインから設定するか (**-D** 構文を使用)、**<configuration>** タグの **pom.xml** ファイルで定義します。

パラメーター	デフォルト値	説明
downloadVersion	true	インターネットからの Camel カタログバージョンのダウンロードを許可するかどうか。プロジェクトが使用する Camel バージョンと、このプラグインがデフォルトで使用する Camel バージョンが異なる場合にダウンロードが必要です。
failOnError	false	無効な Camel エンドポイントが見つかった場合に失敗するかどうか。デフォルトでは、WARN レベルでエラーがプラグインログに記録されます。
logUnparseable	false	解析不可のため検証できないエンドポイント URI をログに記録するかどうか。
includeJava	true	無効な Camel エンドポイントの検証対象となる Java ファイルを含めるかどうか。
includeXml	true	無効な Camel エンドポイントの検証対象となる XML ファイルを含めるかどうか。
includeTest	false	テストソースコードを含めるかどうか。

includes		Java および xml ファイルの名前を絞り込み、指定されたパターンのリスト (ワイルドカードおよび正規表現) と一致するファイルのみが含まれるようにします。複数の値はコンマで区切ることができます。
excludes		Java および xml ファイルの名前を絞り込み、指定されたパターンのリスト (ワイルドカードおよび正規表現) と一致するファイルが除外されるようにします。複数の値はコンマで区切ることができます。
ignoreUnknownComponent	true	不明なコンポーネントを無視するかどうか。
ignoreIncapable	true	解析不可なエンドポイント URI や、Simple 式を無視するかどうか。
ignoreLenientProperties	true	lenient プロパティーを使用するコンポーネントを無視するかどうか。true の場合、URI の検証はより厳密になりますが、lenient プロパティーを使用するため、URI にあってもコンポーネントの一部でないプロパティーでは検証に失敗することがあります。たとえば、HTTP コンポーネントを使用して、エンドポイント URI でクエリーパラメーターを提供する場合がこれに該当します。
ignoreDeprecated	true	**Camel 2.23** の場合: エンドポイント URI で使用される非推奨のオプションを無視するかどうか。
duplicateRouteId	true	**Camel 2.20** の場合: ルート ID の重複を検証するかどうか。ルート ID は一意である必要があります。重複がある場合、Camel は起動に失敗します。
directOrSedaPairCheck	true	**Camel 2.23** の場合: direct/seda エンドポイントが未定義コンシューマーに送信しているかを検証するかどうか。

showAll	false	エンドポイントと Simple 式 (無効と有効の両方) をすべて表示するかどうか。

たとえば、コマンドラインから ignoreDeprecated オプションを無効するには、以下を実行します。

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

オプション名として、**-D** コマンド引数の前に **camel.** (例: **camel.ignoreDeprecated**) を付ける必要があります。

2.12.2.3. include テストを使用したエンドポイントの検証

Maven プロジェクトの場合、プラグインを実行してユニットテストのソースコードで使用されるエンドポイントを検証することもできます。以下のように **-D** スタイルを使用してオプションを渡すことができます。

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

2.12.3. camel:route-coverage

ユニットテストから Camel ルートのカバレッジのレポートを生成するために使用します。これを使用することによって、Camel ルートのどの部分が使用されたかを把握することができます。

2.12.3.1. route-coverage の有効化

以下のいずれかの方法で、ユニットテスト実行時に route-coverage を有効化できます。

- グローバル JVM システムプロパティーを設定してすべてのテストクラスで有効。
- **camel-test-spring** モジュールを使用する場合、テストクラスごとの **@EnableRouteCoverage** アノテーションの使用
- **camel-test** モジュールを使用する場合、テストクラスごとの **isDumpRouteCoverage** メソッドの上書き

2.12.3.2. JVM システムプロパティーを使用した route-coverage の有効化

JVM システムプロパティー **CamelTestRouteCoverage** をオンにして、すべてのテストケースの route-coverage を有効にできます。これは、**maven-surefire-plugin** 設定のいずれかで実行できます。

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

テストの実行中にコマンドラインから設定する場合は次のとおりです。

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

2.12.3.3. @EnableRouteCoverage アノテーションでの route-coverage の有効化

camel-test-spring を使用してテストする場合は、**@EnableRouteCoverage** アノテーションをテストクラスに追加することで、ユニットテストクラスで route-coverage を有効にすることができます。

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

2.12.3.4. isDumpRouteCoverage メソッドでの route-coverage の有効化

camel-test を使っていて、ユニットテストが **CamelTestSupport** を拡張している場合は、以下に示すように route-coverage を有効にすることができます。

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

RouteCoverage メソッドで対象指定できるルートには、固有の ID が割り当てられている必要があります。つまり、匿名ルートは使用できません。Java DSL で **routeId** を使用して行います。

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

また、XML DSL で id 属性を介してルート ID を付与します。

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

2.12.3.5. route-coverage レポートの生成

route-coverage レポートを生成するには、以下のようにユニットテストを実行します。

```
mvn test
```

そして、Maven ゴールを実行して、以下のように route-coverage レポートを生成できます。

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

生成されるレポートでは、ソースコードの行番号でどのルートのルートのカバレッジがないかを確認できます。

```
[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%)
```

この例では、**to** のある最後から 2 番目の行のカウント列が **0** であるため、カバレッジがないことが分かります。また、これはソースコードファイル (XML ファイル **my-camel.xml**) の 34 行目であることも分かります。

2.12.3.6. オプション

Camel Maven プラグインの **coverage** ゴールは、以下のオプションをサポートします。これらのオプションは、コマンドラインから設定するか (**-D** 構文を使用)、**<configuration>** タグの **pom.xml** ファイルで定義します。

パラメーター	デフォルト値	説明
failOnError	false	いずれかのルートのカバレッジが 100% でない場合に失敗するかどうか。
includeTest	false	テストソースコードを含めるかどうか。
includes		Java および xml ファイルの名前を絞り込み、指定されたパターンのリスト (ワイルドカードおよび正規表現) と一致するファイルのみが含まれるようにします。複数の値はコンマで区切ることができます。
excludes		Java および xml ファイルの名前を絞り込み、指定されたパターンのリスト (ワイルドカードおよび正規表現) と一致するファイルが除外されるようにします。複数の値はコンマで区切ることができます。

anonymousRoutes	false	匿名ルート (ルート ID が割り当てられていないルート) を許可するかどうか。ルート ID を使用することで、正確にルートカバレッジのデータをルートのソースコードとマッチングさせることができます。匿名ルートは、テストされたルートがソースコードのどのルートに対応しているかを正確に知ることが難しくなるため、ルートカバレッジの結果の精度が低くなります。

2.13. APACHE CAMEL スタンドアロンの実行

スタンドアロンアプリケーションとして camel を実行する場合、アプリケーションを実行して JVM が終了するまで実行を継続するために使用できる Main クラスを提供します。**MainListener** クラスは、**org.apache.camel.main** Java パッケージ内にあります。

以下は、Main クラスのコンポーネントです。

- **org.apache.camel.Main** クラスの **camel-core** JAR
- **org.apache.camel.spring.Main** クラスの **camel-spring** JAR

以下の例は、Camel から Main クラスを作成して使用する方法を示しています。

```
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

概要

OnCompletion DSL 名は、 **Unit of Work** の完了時に行うアクションを定義するために使用できます。**Unit of Work** は、エクスチェンジ全体に対応する Camel の概念です。 「エクスチェンジ」 を参照してください。**onCompletion** コマンドの機能は以下のとおりです。

- **OnCompletion** コマンドのスコープは、グローバルまたはルートごとに指定できます。ルートスコープはグローバルスコープよりも優先されます。
- **OnCompletion** は、失敗時または成功時にトリガーされるように設定することができます。
- **onWhen** 述語は、特定の状況で **onCompletion** のみをトリガーするために使用できます。
- スレッドプールを使用するかどうかを設定できますが、デフォルトではスレッドプールは使用しません。

onCompletion のルート専用スコープ

エクスチェンジにて **onCompletion** DSL が指定されると、Camel は新しいスレッドを生成します。これにより、**onCompletion** タスクに干渉されることなく、元のスレッドを継続することができます。1

つのルートは1つの **onCompletion** のみをサポートします。以下の例では、エクスチェンジが成功または失敗で完了したかに関係なく、**onCompletion** がトリガーされます。これがデフォルト動作です。

```
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");
```

XML の場合、以下のようになります。

```
<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>
```

失敗時に **onCompletion** をトリガーするには、**onFailureOnly** パラメーターを使用することができます。同様に、成功時に **onCompletion** をトリガーするには、**onCompleteOnly** パラメーターを使用します。

```
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");
```

XML の場合、**onFailureOnly** および **onCompleteOnly** は、**onCompletion** タグのブール値として表現されます。

```
<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>
```

グローバルスコープの onCompletion

onCompletion を複数のルートに対して定義するには、以下を実行します。

```
// 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");
```

onWhen の使用

特定条件で **onCompletion** を起動するには、**onWhen** 述語を使用します。次の例では、メッセージのボディーに **Hello** という単語が含まれている場合に、**onCompletion** がトリガーされます。

```
/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");
```

onCompletion でのスレッドプール

Camel 2.14 以降、**onCompletion** はデフォルトでスレッドプールを使用しません。スレッドプールを強制的に使用するには、**executorService** を設定するか、**parallelProcessing** を true に設定します。たとえば、Java DSL では以下の形式を使用します。

```
onCompletion().parallelProcessing()
     .to("mock:before")
     .delay(1000)
     .setBody(simple("OnComplete:${body}"));
```

XML の場合は以下のようになります。

```
<onCompletion parallelProcessing="true">
   <to uri="before"/>
   <delay><constant>1000</constant></delay>
   <setBody><simple>OnComplete:${body}<simple></setBody>
 </onCompletion>
```

特定のスレッドプールを参照するには **executorServiceRef** オプションを使用します。

```
<onCompletion executorServiceRef="myThreadPool"
  <to uri="before"/>
  <delay><constant>1000</constant></delay>
  <setBody><simple>OnComplete:${body}</simple></setBody>
 </onCompletion>>
```

コンシューマーの応答送信前に onCompletion を実行

onCompletion は 2 つのモードで実行できます。

- AfterConsumer: コンシューマーが終了した後に実行されるデフォルトのモードです。
- BeforeConsumer: コンシューマーが呼び出し元に応答を返信する前に実行されます。これにより、**onCompletion** は、特別なヘッダーの追加などエクスチェンジを変更したり、応答ロガーとしてエクスチェンジをログに記録したりすることができます。

たとえば、**created by** ヘッダーを応答に追加するには、次に示すように **modeBeforeConsumer()** を使用します。

```
.onCompletion().modeBeforeConsumer()
    .setHeader("createdBy", constant("Someone"))
 .end()
```

XML の場合、mode 属性を **BeforeConsumer** に設定します。

```
<onCompletion mode="BeforeConsumer">
  <setHeader headerName="createdBy">
    <constant>Someone</constant>
  </setHeader>
 </onCompletion>
```

2.15. メトリクス

概要

Camel 2.14 から利用可能

Camel は既に多くのメトリクスを提供し、Codahale メトリクスとの統合が Camel ルートに追加されています。これにより、エンドユーザーは、Codahale メトリクスを使用して収集されたメトリクスデータに、Camel のルーティング情報を追加できます。

Codahale メトリクスを使用するには、以下が必要です。

1. camel-metrics コンポーネントの追加
2. XML または Java コードでのルートメトリクスの有効化

パフォーマンスメトリクスは、それらを表示する方法がある場合にのみ使用可能であることに注意してください。メトリクスは JMX 上で利用できるため、JMX と統合できる監視ツールをすべて使用できます。さらに、実際のデータは 100% Codehale JSON です。

メトリクスルートポリシー

単一ルートの Codahale メトリクスを取得するには、ルートごとに **MetricsRoutePolicy** を定義します。

Java DSL の場合、ルートのポリシーとして割り当てるための **MetricsRoutePolicy** のインスタンスを作成します。以下に例を示します。

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

XML DSL の場合、ルートのポリシーとして指定された **<bean>** を定義します。以下に例を示します。

```
<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"/>
[...]
```

メトリクスルートポリシーファクトリー

このファクトリーでは、Codahale メトリクスを使用してルート使用状況の統計を公開するルートごとに、**RoutePolicy** を追加することができます。このファクトリーは、以下の例のように Java および XML で使用できます。

Java DSL の場合は、以下のようにファクトリーを **CamelContext** に追加します。

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

XML DSL の場合は、**<bean>** を以下のように定義します。

```
<!-- use camel-metrics route policy to gather metrics for all routes -->
<bean id="metricsRoutePolicyFactory"
class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

以下に示すように、Java コードから は、**org.apache.camel.component.metrics.routepolicy.MetricsRegistryService** から **com.codahale.metrics.MetricRegistry** を取得できます。

```
MetricRegistryService registryService = context.hasService(MetricsRegistryService.class);
if (registryService != null) {
  MetricsRegistry registry = registryService.getMetricsRegistry();
  ...
}
```

オプション

MetricsRoutePolicyFactory および **MetricsRoutePolicy** は、以下のオプションをサポートします。

名前	デフォルト	説明

durationUnit	**TimeUnit.MILLISECONDS**	メトリクスレポーターまたは統計を json 出力するときの期間に使用する単位。
jmxDomain	**org.apache.camel.metrics**	JXM ドメイン名。
metricsRegistry		共有 **com.codahale.metrics.MetricRegistry** の使用を許可します。指定しない場合は、Camel はこの CamelContext によって使用される共有インスタンスを作成します。
prettyPrint	**false**	統計情報を json 形式で出力する際に pretty print を使用するかどうか。
rateUnit	**TimeUnit.SECONDS**	メトリクスレポーターまたは統計を json 出力するときのレートに使用する単位。
useJmx	**false**	**com.codahale.metrics.JmxReporter** を使って、詳細な統計情報を JMX に報告するかどうか。 CamelContext で JMX が有効になっている場合、JMX ツリーのサービスタイプの下に **MetricsRegistryService** mbean が登録されていることに注意してください。この mbean には、統計を JSON 出力する 1 つのオペレーションがあります。useJmx を true に設定する必要があるのは、統計タイプごとに細かい mbeans を生成する場合のみです。

2.16. JMX の命名

概要

Apache Camel では、**management name pattern** を定義することで、JMX で表示される **CamelContext** Bean の名前をカスタマイズすることができます。たとえば、以下のように、XML **CamelContext** インスタンスの名前パターンをカスタマイズすることができます。

```
<camelContext id="myCamel" managementNamePattern="#name#">
    ...
</camelContext>
```

CamelContext Bean の名前パターンを明示的に設定しないと、Apache Camel はデフォルトの命名ストラテジーに戻ります。

デフォルトの命名ストラテジー

デフォルトでは、OSGi バンドルにデプロイされた **CamelContext** Bean の JMX 名は、バンドルの **OSGi シンボリック名** と同じです。たとえば、OSGi のシンボリック名が **MyCamelBundle** の場合、JMX の名前は **MyCamelBundle** となります。バンドル内に複数の **CamelContext** が存在する場合、接尾辞としてカウンター値を追加することにより、JMX 名が明確になります。たとえば、**MyCamelBundle** バンドルに複数の Camel コンテキストがある場合、対応する JMX MBeans の名前は以下のようになります。

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

JMX 命名ストラテジーのカスタマイズ

デフォルトの命名ストラテジーの欠点の 1 つは、特定の **CamelContext** Bean が実行間で同じ JMX 名を持つことを保証できないことです。実行間の一貫性を高める場合は、**CamelContext** インスタンスの **JMX 名パターン** を定義することで、JMX 名をより正確に制御することができます。

Java DSL での名前パターンの指定

Java の **CamelContext** で名前パターンを指定するには、以下のように **setNamePattern** メソッドを呼び出します。

```
// Java
context.getManagementNameStrategy().setNamePattern("#name#");
```

XML での名前パターンの指定

XML の **CamelContext** で名前パターンを指定するには、以下のように **camelContext** 要素の **managementNamePattern** 属性を設定します。

```
<camelContext id="myCamel" managementNamePattern="#name#">
```

名前パターントークン

以下のいずれかのトークンにリテラルテキストを追加することで、JMX 名前パターンを設定することができます。

表2.11 JMX 名のパターントークン

トークン	説明
#camelId#	**CamelContext** Bean の **id** 属性の値
#name#	**#camelId#** と同じです。

トークン	説明
#counter#	インクリメントカウンター (**1** で開始)
#bundleId#	デプロイされたバンドルの OSGi バンドル ID **OSGi のみ)**
#symbolicName#	OSGi シンボリック名 **(OSGi のみ)**
#version#	OSGi バンドルバージョン **(OSGi のみ)**

例

以下は、サポートされるトークンを使用して定義できる JMX 名前パターンの例です。

```
<camelContext id="fooContext" managementNamePattern="FooApplication-#name#">
    ...
</camelContext>
<camelContext id="myCamel" managementNamePattern="#bundleID#-#symbolicName#-#name#">
    ...
</camelContext>
```

あいまいな名前

カスタマイズされた命名パターンはデフォルトの命名ストラテジーを上書きするため、このアプローチを使用してあいまいな JMX MBean 名を定義することができます。以下に例を示します。

```
<camelContext id="foo" managementNamePattern="SameOldSameOld"> ... </camelContext>
...
<camelContext id="bar" managementNamePattern="SameOldSameOld"> ... </camelContext>
```

この場合、Apache Camel は起動に失敗し、**MBean が既に存在する**ことを示す例外が出力されます。そのため、あいまいな名前のパターンを定義しないように細心の注意を払う必要があります。

2.17. パフォーマンスと最適化

メッセージのコピー

allowUseOriginalMessage オプションのデフォルト設定は **false** です。これは、必要がない場合に、元のメッセージのコピー作成を削減します。**allowUseOriginalMessage** オプションを有効にするには、次のコマンドを使用します。

- エラーハンドラーまたは **onException** 要素に **useOriginalMessage=true** を設定します。
- Java アプリケーションコードで **AllowUseOriginalMessage=true** を設定してから、**getOriginalMessage** メソッドを使用します。

注記

2.18 より前の Camel バージョンでは、**allowUseOriginalMessage** のデフォルト設定は true です。

第3章 エンタープライズ統合パターンの導入

概要

Apache Camel の **エンタープライズ統合パターン** は、Gregor Hohpe および Bobby Woolf 両氏の著書である同名の書籍 Enterprise Integration Patterns の影響を受けています。両著者が説明するパターンは、エンタープライズ統合プロジェクトを開発するための優れたツールボックスを提供します。統合アーキテクチャーを説明するための共通の言語を提供する他に、Apache Camel のプログラミングインターフェイスと XML 設定を使用して、多くのパターンを直接実装することができます。

3.1. パターンの概要

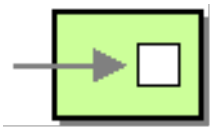
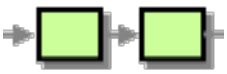
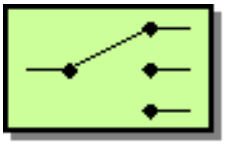
書籍 Enterprise Integration Patterns

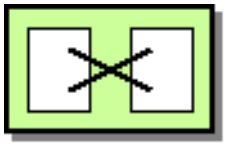
Apache Camel は、Gregor Hohpe および Bobby Woolf 両氏の著書である Enterprise Integration Patterns に記載されているほとんどのパータンをサポートします。

メッセージングシステム

表3.1「メッセージングシステム」 に記載されているメッセージングシステムパターンは、メッセージングシステムを構成する基本的な概念やコンポーネントを紹介します。

表3.1 メッセージングシステム

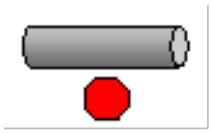
アイコン	名前	ユースケース
	図5.1「Message パターン」	メッセージチャネルによって接続された 2 つのアプリケーションはどのように情報を交換するか。
	図5.2「Message Channel パターン」	メッセージングを使用して単一のアプリケーションが別のアプリケーションと通信する方法。
	図5.3「Message Endpoint パターン」	アプリケーションがメッセージングチャネルに接続してメッセージを送受信する方法。
	図5.4「Pipes and Filters パターン」	独立性と柔軟性を維持しながら、メッセージで複雑な処理を行う方法。
	図5.7「Message Router パターン」	定義された条件のセットに応じてメッセージを異なるフィルターに渡すために、個々の処理ステップを切り離す方法。

アイコン	名前	ユースケース
	図5.8「Message Translator パターン」	メッセージングを使用して、異なるデータフォーマットを使用するシステムの間で通信を行う方法。

メッセージングチャネル

メッセージングチャネルは、メッセージングシステムで参加者の接続に使用される基本的なコンポーネントです。表3.2「メッセージングチャネル」 のパターンは、使用できる異なる種類のメッセージングチャネルを説明しています。

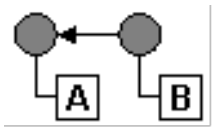
表3.2 メッセージングチャネル

アイコン	名前	ユースケース
	図6.1「Point to Point Channel パターン」	1つの受信側のみがドキュメントの受信や呼び出しを実行するように、呼び出し側が確認する方法。
	図6.2「Publish Subscribe Channel パターン」	送信側が対象のすべての受信側にブロードキャストする方法。
	図6.3「Dead Letter Channel パターン」	メッセージングシステムが配信できないメッセージの処理方法。
	図6.4「Guaranteed Delivery パターン」	メッセージングシステムに障害が発生しても、送信側がメッセージを確実に配信する方法。
	図6.5「Message Bus パターン」	独立し、分離したアプリケーションを連携でき、他のアプリケーションに影響を与えることなく1つ以上のアプリケーションを追加または削除できるアーキテクチャーとは。

メッセージの構築

表3.3「メッセージの構築」 のメッセージ構築パターンは、システムを通過するメッセージのさまざまな形式と関数を表しています。

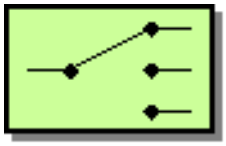
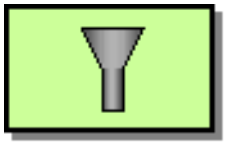
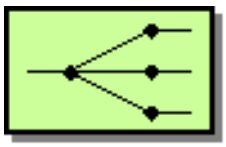
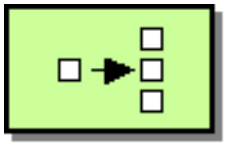
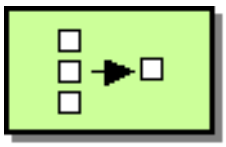
表3.3 メッセージの構築

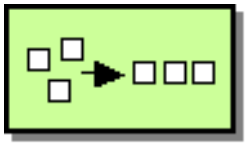
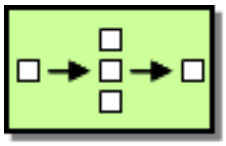
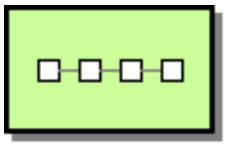
アイコン	名前	ユースケース
	「概要」	受信した応答を生成したリクエストを、要求側が識別する方法。
	「返信先アドレス」	応答側が応答の送信先を認識する方法。

メッセージルーティング

表3.4「メッセージのルーティング」 のメッセージルーティングパターンは、メッセージチャネルをリンクするさまざまな方法を表しています。これには、メッセージのボディーを変更せずにメッセージストリームに適用できるさまざまなアルゴリズムが含まれます。

表3.4 メッセージのルーティング

アイコン	名前	ユースケース
	「Content-Based Router」	単一の論理関数 (在庫確認など) の実装が複数の物理システムに分散されている場合の処理方法。
	「Message Filter」	コンポーネントが不必要なメッセージを受信しないようにする方法。
	「受信者リスト」	動的に指定された受信者のリストにメッセージをルーティングする方法。
	「Splitter」	各要素を異なる方法で処理しなければならない可能性がある、複数の要素が含まれるメッセージの処理方法。
	「Aggregator」	個別かつ関連するメッセージの結果を組み合わせ、全体として処理できるようにする方法。

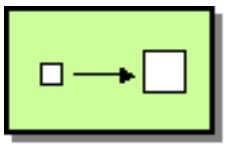
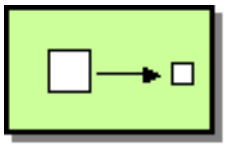
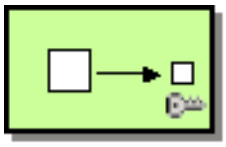
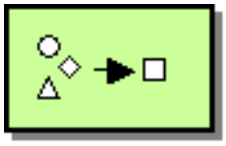
アイコン	名前	ユースケース
	「Resequencer」	順序どおりでない関連するメッセージのストリームを正しい順序に戻す方法。
	「Composed Message Processor」	要素ごとに異なる処理が必要となる可能性がある複数の要素で設定されるメッセージを処理する場合に、メッセージフロー全体を維持する方法。
	「Scatter-Gather」	複数の受信者にメッセージを送信する必要があり、その各受信者が応答を送信する可能性がある場合に、メッセージフロー全体を維持する方法。
	「Routing Slip」	設計時にステップの順序が分からず、ステップの順序がメッセージごとに異なる可能性がある場合に、一連の処理ステップを通じてメッセージを継続的にルーティングする方法。
	「Throttler」	メッセージのスロットリングによって、特定のエンドポイントがオーバーロードされないようにする方法、または外部サービスと合意した SLA を越えないようにする方法。
	「Delayer」	メッセージの送信を遅らせる方法。
	「Load Balancer」	複数のエンドポイント間で負荷を分散する方法。
	「Hystrix」	外部サービスの呼び出し時に、Hystrix サーキットブレーカーを使用する方法。Camel 2.18 の新機能。
	「Service Call」	レジストリーでサービスを検索して、分散システムでリモートサービスを呼び出す方法。Camel 2.18 の新機能。

アイコン	名前	ユースケース
	「Multicast」	メッセージを同時に複数のエンドポイントにルーティングする方法。
	「Loop」	メッセージをループで繰り返し処理する方法。
	「Sampling」	ダウンストリームルートをオーバーロードを防ぐために一定の期間で複数のメッセージから1つのメッセージをサンプリングする方法。

メッセージの変換

表3.5「Message Transformation」 のメッセージ変換パターンは、さまざまな目的のためにメッセージの内容を変更する方法を表しています。

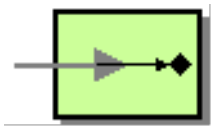
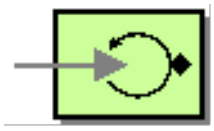
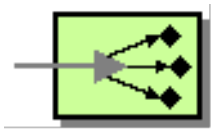
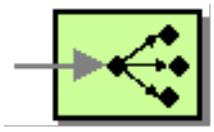
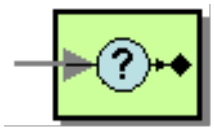
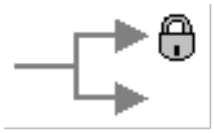
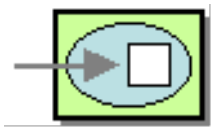
表3.5 Message Transformation

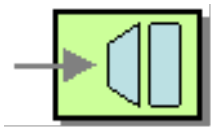
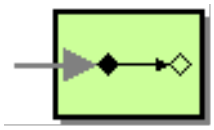
アイコン	名前	ユースケース
	「Content Enricher」	メッセージの送信元に必要なデータ項目がすべてない場合に他のシステムと通信する方法。
	「Content Filter」	数個のデータ項目のみが必要な場合に大きなメッセージの処理を簡単にする方法。
	「Claim Check EIP」	情報の内容を減らさずにシステム全体で送信されるメッセージのデータ量を減らす方法。
	「ノーマライザー」	意味的には同等で、受け取った形式が異なるメッセージの処理方法。
	「並び替え」	メッセージのボディーのソート方法。

メッセージングエンドポイント

メッセージングエンドポイントは、メッセージングチャネルとアプリケーション間の接点を示します。表3.6「Messaging Endpoint」 のメッセージングエンドポイントパターンは、エンドポイントに設定できるサービスのさまざまな機能と特性を表しています。

表3.6 Messaging Endpoint

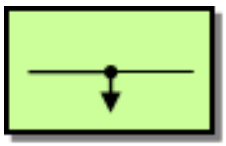
アイコン	名前	ユースケース
	「Messaging Mapper」	ドメインオブジェクトとメッセージングインフラストラクチャーの間でデータを移動し、お互いに独立した状態を維持する方法。
	「Event Driven Consumer」	メッセージが利用できるようになったときにアプリケーションが自動的にメッセージを消費する方法。
	「Polling Consumer」	アプリケーションの準備ができたときに、アプリケーションがメッセージを消費する方法。
	「Competing Consumers」	メッセージングクライアントが複数のメッセージを同時に処理する方法。
	「Message Dispatcher」	1 つのチャネルで複数のコンシューマーがメッセージ処理を調整する方法。
	「Selective Consumer」	メッセージコンシューマーが受信するメッセージを選択する方法
	「Durable Subscriber」	サブスクライバーがメッセージをリッスンしていないときにメッセージの欠落を防ぐ方法。
	「Idempotent Consumer」	メッセージの受信側が重複メッセージを処理する方法。
	「Transactional Client」	クライアントがメッセージングシステムでトランザクションを制御する方法。

アイコン	名前	ユースケース
	「Messaging Gateway」	残りのアプリケーションからメッセージングシステムへのアクセスをカプセル化する方法。
	「Service Activator」	サービスがさまざまなメッセージング技術や、メッセージング以外の技術によって呼び出されるように、アプリケーションで設計する方法。

システム管理

表3.7「システム管理」 のシステム管理パターンは、メッセージングシステムを監視、テスト、および管理する方法を表しています。

表3.7 システム管理

アイコン	名前	ユースケース
	*12章 システム管理*	ポイントツーポイントチャネルで送信されるメッセージを検査する方法。

第4章 REST サービスの定義

概要

Apache Camel は、REST サービスを定義するために複数のアプローチをサポートします。特に、Apache Camel は REST DSL (Domain Specific Language) を提供します。これは、REST コンポーネントを抽象化でき、OpenAPI とも統合できるシンプルながらも強力な Fluent API です。

4.1. CAMEL における REST サービスの概要

概要

Apache Camel は、Camel アプリケーションで REST サービスを定義するためのさまざまなアプローチやコンポーネントを提供します。本セクションでは、これらのアプローチとコンポーネントの概要を紹介し、要件に最適な実装と API を判断できるようにします。

REST とは

Representational State Transfer (REST) は、4 つの基本的な HTTP 動詞 (**GET**、**POST**、**PUT**、および **DELETE**) のみを使用して、HTTP 通信でデータ送信を中心とする分散アプリケーションのアーキテクチャーです。

REST アーキテクチャーは HTTP を直接活用します。これは、SOAP のような HTTP を単なるトランスポートプロトコルとして扱うプロトコルとは対象的です。重要なポイントは、HTTP プロトコル **自体** が、既にいくつかの規約によって拡張され、分散アプリケーションのフレームワークとして機能するのに適していることです。

REST 呼び出しのサンプル

REST アーキテクチャーは標準の HTTP メソッドを中心に設定されているため、多くの場合、通常のブラウザーを REST クライアントとして使用することができます。たとえば、ホストとポート **localhost:9091** で実行されている単純な **Hello World** の REST サービスを呼び出すには、ブラウザーで以下の URL にアクセスします。

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

仮に、**Hello World** REST サービスが、以下のようなレスポンスを返すとします。

```
Hello Garp
```

このレスポンスは、ブラウザーのウィンドウに表示されます。通常のブラウザー (または **curl** コマンドラインユーティリティー) だけを使用して、REST サービスを呼び出せる気軽さは、REST プロトコルが急速に人気を集めている多くの理由の 1 つです。

REST ラッパーレイヤー

REST ラッパーレイヤーは、REST サービスを定義するための簡潔な構文を提供し、異なる REST 実装の上に重ねることができます。

REST DSL

REST DSL (**camel-core** の一部) は、REST サービスを定義するためのシンプルなビルダー API を提供するファサードまたはラッパーレイヤーです。REST DSL 自体は REST 実装を提供しているわけ

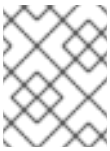
では**ありません**。REST DSL は、ベースとなる REST 実装と組み合わせる必要があります。たとえば、以下の Java コードは、REST DSL を使用してシンプルな **Hello World** の REST サービスを定義する方法を示しています。

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

詳細は、「REST DSL を使用した REST サービスの定義」 を参照してください。

REST コンポーネント

REST コンポーネント (**camel-core** の一部) は、URI 構文を使用して REST サービスの定義を可能にするラッパーレイヤーです。REST DSL と同様に、REST コンポーネント自体は REST 実装を提供しているわけでは**ありません**。ベースとなる REST 実装と組み合わせる必要があります。
HTTP コンポーネントを明示的に指定しない場合、REST DSL はクラスパス上の利用可能なコンポーネントをチェックすることで、どの HTTP コンポーネントを使用するかを自動検出します。REST DSL は、HTTP コンポーネントのデフォルト名を探し、最初に見つかったものを使用します。クラスパス上に HTTP コンポーネントがなく、かつ HTTP トランスポートが明示的に設定されていない場合は、デフォルトの HTTP コンポーネントは **camel-http** になります。



注記

HTTP コンポーネントの自動検出機能が Camel 2.18 で追加されました。Camel 2.17 では利用できません。

以下の Java DSL は、**camel-rest** コンポーネントを使用して **Hello World** のサービスを定義する方法を示しています。

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

REST 実装

Apache Camel は、以下のコンポーネントを通じて、複数の REST 実装を提供します。

Spark-Rest コンポーネント

Spark-Rest コンポーネント (**camel-spark-rest**) は、URI 構文を使用して REST サービスの定義を可能にする REST 実装です。Spark フレームワーク自体は Java の API で、Sinatra フレームワーク (Python の API) を大まかにベースにしています。たとえば、以下の Java コードでは、Spark-Rest コンポーネントを使用して **Hello World** のサービスを定義する方法を示しています。

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

Rest コンポーネントとは対照的に、URI の変数の構文は **{name}** ではなく、**:name** であることに注意してください。

注記

Spark-Rest コンポーネントには Java 8 が必要です。

Restlet コンポーネント

Restlet コンポーネント (**camel-restlet** の一部) は、原則として、異なるトランスポートプロトコル

の上に重ねることができる REST 実装です (ただし、このコンポーネントは HTTP プロトコルに対してのみテストされます)。このコンポーネントは、Java で REST サービスを開発する商用フレームワークである Restlet Framework との統合も提供します。たとえば、以下の Java コードは Restlet コンポーネントを使用して **Hello World** のサービスを定義する方法を示しています。

```
from("restlet:http://0.0.0.0:9091/say/hello/{name}?restletMethod=get")
    .transform().simple("Hello ${header.name}");
```

詳細は、**Apache Camel Component Reference Guide** の **Restlet** を参照してください。

Servlet コンポーネント

Servlet コンポーネント (**camel-servlet** 内) は、Java サーブレットを Camel ルートにバインドするコンポーネントです。言い換えれば、Servlet コンポーネントを使用すると、標準の Java サーブレットのように Camel ルートをパッケージ化してデプロイすることができます。したがって、Servlet コンポーネントは、**サーブレットコンテナー**内部に Camel ルートをデプロイする必要がある場合 (たとえば、Apache Tomcat HTTP サーバーや JBoss Enterprise Appication Platform コンテナーなど) に Camel ルートをデプロイする場合に特に便利です。
ただし、Servlet コンポーネントだけは、REST サービスの定義に便利な REST API を提供しません。そのため、Servlet コンポーネントを使用する最も簡単な方法は、REST DSL と組み合わせることです。これにより、ユーザーフレンドリーな API で REST サービスを定義できます。

詳細は、**Apache Camel Component Reference Guide** の **Servlet** を参照してください。

JAX-RS REST 実装

JAX-RS (Java API for RESTful Web Services) は、REST リクエストを Java オブジェクトにバインドするためのフレームワークです。バインドを定義するには、この Java クラスに JAX-RS のアノテーションを記述する必要があります。JAX-RS フレームワークは比較的成熟しており、REST サービスを開発するための洗練されたフレームワークも提供します。しかし、プログラムするのもやや複雑です。

Apache Camel と JAX-RS の統合は、Apache CXF の上に重ねた CXFRS コンポーネントによって実装されます。簡単にいうと、JAX-RS は以下のアノテーションを使用して REST リクエストを Java クラスにバインドします (以下は多くのアノテーションの一部のみとなります)。

@Path

コンテキストパスを Java クラスにマッピングしたり、サブパスを特定の Java メソッドにマッピングしたりできるアノテーション。

@GET、@POST、@PUT、@DELETE

HTTP メソッドを Java メソッドにマッピングするアノテーション。

@PathParam

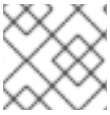
URI パラメーターを Java メソッド引数にマッピングするか、URI パラメーターをフィールドに注入するアノテーション。

@QueryParam

クエリーパラメーターを Java メソッド引数にマッピングするか、クエリーパラメーターをフィールドに注入するアノテーション。

REST リクエストまたは REST レスポンスのボディーは、通常、JAXB (XML) データフォーマットであることが期待されます。しかし、Apache CXF は JSON 形式から JAXB 形式への変換もサポートしているため、JSON メッセージも解析することができます。

詳細は、**Apache Camel Component Reference Guide** の **CXFRS** および Apache CXF Development Guide を参照してください。



注記

CXFRS コンポーネントは REST DSL と統合されて**いません**。

4.2. REST DSL を使用した REST サービスの定義

REST DSL はファサード

REST DSL は、Java DSL または XML DSL (Domain Specific Language) で REST サービスを定義するための簡略化された構文を提供する**ファサード**です。REST DSL は実際の REST 実装を提供しているわけではなく、**既存** の REST 実装 (Apache Camel に複数ある) のラッパーにすぎません。

REST DSL の利点

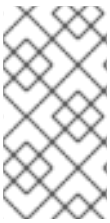
REST DSL ラッパーレイヤーには、以下の利点があります。

- REST サービスを定義するためのモダンで使いやすい構文である。
- 複数の Apache Camel コンポーネントと互換性がある。
- OpenAPI インテグレーション (**camel-openapi-java** コンポーネント経由)。

REST DSL と統合可能なコンポーネント

REST DSL は実際の REST 実装ではないため、最初に、ベースとなる実装を提供する Camel コンポーネントを選択する必要があります。現在、以下の Camel コンポーネントが REST DSL に統合されています。

- Servlet コンポーネント (**camel-servlet**)
- Spark REST コンポーネント (**camel-spark-rest**)
- Netty4 HTTP コンポーネント (**camel-netty4-http**)
- Jetty コンポーネント (**camel-jetty**)
- Restlet コンポーネント (**camel-restlet**)
- Undertow コンポーネント (**camel-undertow**)



注記

REST コンポーネント (**camel-core** の一部) は REST 実装ではありません。REST DSL と同様に、REST コンポーネントはファサードであり、URI 構文を使用して REST サービスを定義するための簡潔な構文を提供します。REST コンポーネントには、ベースとなる REST 実装も必要です。

REST 実装を使用するように REST DSL を設定

REST 実装を指定するには、**restConfiguration()** ビルダー (Java DSL の場合) または **restConfiguration** 要素 (XML DSL の場合) を使用します。たとえば、Spark-Rest コンポーネントを使用するように REST DSL を設定するには、Java DSL で以下のようなビルダー式を使用します。

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

そして、XML DSL では、以下のような要素 (**camelContext** の子として) を使用します。

```
<restConfiguration component="spark-rest" port="9091"/>
```

構文

REST サービスを定義するための Java DS L 構文は以下のとおりです。

```
rest("BasePath").Option().
    .Verb("Path").Option().[to() | route().CamelRoute.endRest()]
    .Verb("Path").Option().[to() | route().CamelRoute.endRest()]
    ...
    .Verb("Path").Option().[to() | route().CamelRoute];
```

ここの **CamelRoute** は、オプションの組み込み Camel ルートです (標準の Java DSL 構文を使用して定義されています)。

REST サービスの定義は **rest()** キーワードで始まり、その後に特定の URL パスセグメントを処理する 1 つ以上の Verb 句が続きます。HTTP 動詞は、**get()**、**head()**、**put()**、**post()**、**delete()**、**patch()** または **verb()** のいずれかになります。Verb 句は以下の構文のいずれかを使用できます。

- **to()** キーワードで終わる Verb 句。以下に例を示します。

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

- キーワード **route()** で終わる Verb 句 (Camel ルートの埋め込みの場合)。以下に例を示します。

  ```
  get("...").Option()+.route("...").CamelRoute.endRest()
  ```

Java による REST DSL

Java で REST DSL でサービスを定義するには、通常の Apache Camel ルート定義と同様に、REST 定義を **RouteBuilder.configure()** メソッドのボディーに配置します。たとえば、REST DSL と Spark-Rest コンポーネントの組み合わせを使用して **Hello World** のサービスを定義するには、以下の Java コードを定義します。

```
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");
```

前述の例では、3 種類のビルダーがあります。

restConfiguration()

特定の REST 実装 (Spark-Rest) を使用するように REST DSL を設定します。

rest()

REST DSL を使用してサービスを定義します。各 Verb 句はキーワード **to()** で終了となります。このキーワードは、受信メッセージをエンドポイント **direct** に転送します (**direct** コンポーネントは、同じアプリケーション内でルートをつなぎ合わせます)。

from()

通常の Camel ルートを定義します。

XML を使用した REST DSL

XML で XML DSL でサービスを定義するには、**rest** 要素を **camelContext** の子要素として定義します。たとえば、Spark-Rest コンポーネントで REST DSL を使用して単純な **Hello World** サービスを定義するには、以下の XML コード (Blueprint) を定義します。

```
<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>
```

ベースパスの指定

rest() キーワード (Java DSL) または **rest** 要素の **path** 属性 (XML DSL) を使用してベースパスを定義できます。このパスはすべての Verb 句のパスに接頭辞として付けられます。たとえば、以下は Java DSL のスニペットを示します。

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

または、以下は XML DSL のスニペットを示します。

```
<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 ビルダーは、以下の URL マッピングで公開します。

```
/say/hello
/say/bye
```

ベースパスはオプションです。必要であれば、各 Verb 句にフルパスを指定することもできます。

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

Dynamic To の使用

REST DSL では、**toD** 動的な to パラメーターをサポートしています。動的な to パラメーターを使用して動的な転送先 URI を指定できます。

たとえば、動的エンドポイント URI を使って動的な JMS キューへ送信するには以下のように定義できます。

```
public void configure() throws Exception {
   rest("/say")
     .get("/hello/{language}").toD("jms:queue:hello-${header.language}");
}
```

XML DSL では、以下のようになります。

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

toD パラメーターの詳細は、「Dynamic To」 を参照してください。

URI テンプレート

Verb 句で利用する引数では、URI テンプレートで指定できます。これにより、特定のパスセグメントを名前付きプロパティーとして取り込むことができます (これらは Camel メッセージヘッダーにマッピングされます)。たとえば、**Hello World** アプリケーションをパーソナライズして、発信者の名前で挨拶するようにする場合は、以下のような REST サービスを定義することができます。

```
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 テンプレートは **{name}** パスセグメントのテキストを取得し、このキャプチャーされたテキストを **name** メッセージヘッダーにコピーします。URL が **/say/hello/Joe** で終わる GET HTTP リクエストを送信してサービスを呼び出す場合、HTTP レスポンスは **Hello Joe** になります。

組み込みルートの構文

to() キーワード (Java DSL) または **to** 要素 (XML DSL) で Verb 句を終わらせる代わりに、 **route()** キーワード (Java DSL) または **route** 要素 (XML DSL) を使用して Apache Camel ルートを直接 REST DSL に埋め込むことも可能です。**route()** キーワードを使用すると、以下の構文でルートを Verb 句に埋め込みできます。

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

endRest() キーワード (Java DSL のみ) は、(**rest()** ビルダーに複数の Verb 句がある場合に) Verb 句を区切ることができる必須の句読点です。

たとえば、**Hello World** の例を Java DSL のように組み込み Camel ルートを使用するようにリファクタリングできます。

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

XML DSL では以下のようになります。

```
<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>
```

注記

現在の **CamelContext** 内で、例外句 (**onException()**) やインターセプター (**intercept()**) を定義した場合、これらの例外句とインターセプターは組み込みルートでもアクティブになります。

REST DSL と HTTP トランスポートコンポーネント

HTTP コンポーネントを明示的に指定しない場合、REST DSL はクラスパス上の利用可能なコンポーネントをチェックすることで、どの HTTP コンポーネントを使用するかを自動検出します。REST DSL は、HTTP コンポーネントのデフォルト名を探し、最初に見つかったものを使用します。クラスパス上に HTTP コンポーネントがなく、かつ HTTP トランスポートが明示的に設定されていない場合は、デフォルトの HTTP コンポーネントは **camel-http** になります。

リクエストとレスポンスのコンテンツタイプの指定

Java の **consumes()** と **produces()** オプション、または XML の **consumes** と **produces** 属性を使用して、HTTP リクエストとレスポンスの**コンテンツタイプ**をフィルターリングすることができます。たとえば、いくつかの一般的なコンテンツタイプ (正式には**インターネットメディアタイプ**と呼ばれます) は以下のとおりです。

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

コンテンツタイプは REST DSL の Verb 句のオプションとして指定されます。たとえば、Verb 句を制限して **text/plain** の HTTP リクエストのみを受け付け、 **text/html** の HTTP レスポンスのみを送信するようにするには、以下のような Java コードを使用します。

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

XML では、以下のように **consumes** および **produces** 属性を設定できます。

```
<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>
```

また、**consumes()** または **produces()** の引数をコンマ区切りのリストとして指定することもできます。たとえば、**consumes("text/plain, application/json")** などです。

追加の HTTP メソッド

HTTP サーバーの実装によっては、REST DSL の標準動詞セット (**get()**、**head()**、**put()**、**post()**、**delete()**、**patch()**) では提供されない追加の HTTP メソッドをサポートしているものもあります。追加の HTTP メソッドにアクセスするには、Java DSL の場合は汎用キーワード **verb()**、XML DSL の場合は汎用要素 **verb** を使用できます。

たとえば、Java DSL の場合、HTTP メソッド TRACE は以下のように実装します。

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

ここで、**transform()** は **IN** メッセージのボディーを **OUT** メッセージのボディーにコピーし、HTTP リクエストに返答させています。

XML DSL の場合、HTTP メソッド TRACE は以下のように実装します。

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  ...
  <rest path="/say">
    <verb uri="/hello" method="TRACE">
      <route>
        <transform/>
      </route>
    </get>
</camelContext>
```

カスタム HTTP エラーメッセージの定義

REST サービスがエラーメッセージを返答する必要がある場合、以下のようにカスタム HTTP エラーメッセージを定義できます。

1. **Exchange.HTTP_RESPONSE_CODE** ヘッダーにエラーコードの値を設定して、HTTP エラーコードを指定します (例: **400**、**404** など)。この設定は、正常時のレスポンスではなく、エラーメッセージをレスポンスする REST DSL を示します。

2. メッセージのボディーにカスタムエラーメッセージを設定します。

3. 必要に応じて **Content-Type** ヘッダーを設定します。

4. REST サービスが Java オブジェクトとの間でマーシャリングするように設定されている場合 (**bindingMode** が有効になっている場合)、**skipBindingOnErrorCode** オプションが有効になっていることを確認する必要があります (デフォルトは有効)。これは、REST DSL がレスポンスを送信する際にメッセージボディーをアンマーシャリングしないようにするためです。
   オブジェクトバインディングの詳細については、「Java オブジェクトとの間のマーシャリング」 を参照してください。

以下の Java DSL の例は、カスタムエラーメッセージを定義する方法を示しています。

```
// 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");
```

この例では、入力 ID が 100 未満の数値の場合、以下のように実装された **UserErrorService** Bean を使用してカスタムエラーメッセージを返します。

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

UserErrorService Bean では、カスタムエラーメッセージを定義し、HTTP エラーコードを **400** に設定します。

パラメーターのデフォルト値

受信する Camel メッセージのヘッダーにデフォルト値を指定することができます。

たとえば、クエリーパラメーターの **verbose** などのキーワードを使用して、デフォルト値を指定することができます。以下のコードではデフォルト値は **false** となります。これは、**verbose** キーを持つヘッダーに他の値が提供されていない場合、デフォルトが **false** となります。

```
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");
```

カスタム HTTP エラーメッセージでの JsonParserException のラッピング

カスタムのエラーメッセージを返する場合によくあるのは、**JsonParserException** 例外をラッピングすることです。例として、以下のように、Camel の例外処理メカニズムを利用して、HTTP エラーコード 400 のカスタム HTTP エラーメッセージを作成することができます。

```
// 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 の他のオプション

一般的に、REST DSL のオプションは、以下のようにサービス定義のベース部分 (**rest()** の直後) に直接適用することができます。

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

この場合、指定したオプションは下位のすべての Verb 句に適用されます。または、以下のように、個々の Verb 句にオプションを適用することもできます。

```
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");
```

この場合、指定したオプションは関連する Verb 句にのみ適用され、ベース部分の設定は上書きされます。

表4.1「REST DSL のオプション」 は、REST DSL でサポートされているオプションをまとめたものです。

表4.1 REST DSL のオプション

Java DSL	XML DSL	説明
bindingMode()	**@bindingMode**	バインディングモードを指定します。これを使用して、受信メッセージを Java オブジェクトにマーシャリングすることができます (オプションで、Java オブジェクトを送信メッセージにアンマーシャリングすることもできます)。値は **off** (デフォルト)、**auto**、**json**、**xml**、**json_xml** です。
consumes()	**@consumes**	HTTP リクエストで指定されたインターネットメディアタイプ (MIME タイプ) のみを受け入れるように Verb 句を制限します。代表的な値は、**text/plain**、**text/http**、**text/xml**、**application/json**、**application/xml** です。
customId()	**@customId**	JMX Management のカスタム ID を指定します。
description()	**description**	REST サービスまたは Verb 句の説明文を記載します。JMX Management やツールを使う場合に便利です。
enableCORS()	**@enableCORS**	**true** の場合、HTTP レスポンスで CORS (オリジン間リソース共有) ヘッダーを有効にします。デフォルトは **false** です。

Java DSL	XML DSL	説明
id()	**@id**	REST サービスの一意の ID を指定します。これは、JMX Management や他のツールを使用する際に便利です。
method()	**@method**	この Verb 句で処理する HTTP メソッドを指定します。通常は一般的な **verb()** キーワードと組み合わせて使用します。
outType()	**@outType**	オブジェクトバインディングが有効な場合 (**bindingMode** オプションが有効な場合)、このオプションは **HTTP レスポンス** メッセージを表す Java 型を指定します。
produces()	**produces**	HTTP レスポンスで指定されたインターネットメディアタイプ (MIME タイプ) のみを生成するように Verb 句を制限します。代表的な値は、**text/plain**、**text/http**、**text/xml**、**application/json**、**application/xml** です。
type()	**@type**	オブジェクトバインディングが有効な場合 (**bindingMode** オプションが有効な場合)、このオプションは **HTTP リクエスト** メッセージを表す Java 型を指定します。
VerbURIArgument	**@uri**	Verb 句の引数としてパスセグメントまたは URI テンプレートを指定します。たとえば、**get(VerbURIArgument)** です。
BasePathArgument	**@path**	**rest()** キーワード (Java DSL) または **rest** 要素 (XML DSL) でベースパスを指定します。

4.3. JAVA オブジェクトとの間のマーシャリング

HTTP で送信するための Java オブジェクトのマーシャリング

REST プロトコルを使用する最も一般的な方法の 1 つは、Java Bean の内容をメッセージボディーで送信することです。これを実現させるには、Java オブジェクトを適切なデータフォーマットとの間で

マーシャリングするメカニズムが必要です。Java オブジェクトのエンコードに適した以下のデータフォーマットが REST DSL でサポートされています。

JSON

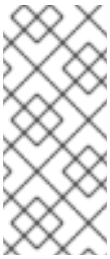
JSON (JavaScript Object Notation) は、Java オブジェクトとの間で簡単にマッピングできる軽量なデータフォーマットです。JSON 構文はコンパクトで、緩く型指定され、人間が読み書きがしやすい構文です。これらの理由から、JSON は REST サービスのメッセージ形式として人気があります。
たとえば、以下の JSON コードは、**id** と **name** の 2 つのプロパティーフィールドを持つ **User** Bean を表現できます。

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

JAXB

JAXB(Java Architecture for XML Binding) は、Java オブジェクトと XML の間で簡単にマッピングできるデータフォーマットです。XML を Java オブジェクトにマーシャリングするには、使用する Java クラスにアノテーションを付ける必要があります。
たとえば、以下の JAXB コードは、**id** と **name** の 2 つのプロパティーフィールドを持つ **User** Bean を表現できます。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<User>
  <Id>1234</Id>
  <Name>Jane Doe</Name>
</User>
```



注記

Camel 2.17.0 以降、JAXB データフォーマットと型コンバーターは、**XmlRootElement** の代わりに **ObjectFactory** を使用するクラスの XML から POJO への変換をサポートします。また、Camel コンテキストでは、値が true の **CamelJaxbObjectFactory** プロパティーを含める必要があります。ただし、後方互換のためにデフォルトは false になっています。

REST DSL による JSON と JAXB の統合

メッセージボディーを Java オブジェクトへ変換するために必要なコードを自分で書くこともできます。しかし、REST DSL は、この変換を自動的に実行する利便性を提供します。特に、JSON と JAXB を REST DSL と統合すると、以下のようなメリットがあります。

- Java オブジェクトとの間のマーシャリングは自動的に実行されます (適切な設定がある場合)。
- REST DSL は、データフォーマット (JSON または JAXB のいずれか) を自動的に検出し、適切な変換を行うことができます。
- REST DSL は **抽象化レイヤー** を提供するので、開発するコードは JSON または JAXB 実装に固有のものではありません。そのため、アプリケーションコードへの影響を最小限に抑えながら、後から実装を切り替えることができます。

サポートされるデータフォーマットコンポーネント

Apache Camel は JSON と JAXB データフォーマットの多くの実装を提供しています。現在、REST DSL では以下のデータフォーマットがサポートされています。

- JSON
  - Jackson データフォーマット (**camel-jackson**) **(デフォルト)**
  - gson データフォーマット (**camel-gson**)
  - XStream データフォーマット (**camel-xstream**)
- JAXB
  - JAXB データフォーマット (**camel-jaxb**)

オブジェクトマーシャリングを有効にする方法

REST DSL でオブジェクトのマーシャリングを有効にする場合は、以下の点に注意してください。

1. **bindingMode** オプションを設定してバインディングモードを有効にします (バインディングモードを設定できるレベルはいくつかあり、詳細は 「バインディングモードの設定」 を参照してください)。
2. 受信メッセージでは **type** オプション (必須)、送信メッセージでは **outType** オプション (任意) を使用して、変換先 (または変換元) の Java 型を指定します。
3. Java オブジェクトを JAXB データフォーマットとの間で変換する場合、Java クラスに適切な JAXB アノテーションを付ける必要があります。
4. **jsonDataFormat** オプションや **xmlDataFormat** オプション (**restConfiguration** ビルダーで指定可能) を使用して、ベースとなるデータフォーマットの実装を指定します。
5. ルートが JAXB 形式の戻り値を提供する場合、通常、エクスチェンジボディーの **Out** メッセージを JAXB アノテーションを持つクラスのインスタンス (JAXB 要素) に設定することが期待されます。ただし、JAXB の戻り値を XML 形式で直接提供する場合は、キー **xml.out.mustBeJAXBElement** で **dataFormatProperty** を **false** に設定します (**restConfiguration** ビルダーで指定可能)。たとえば、XML DSL の構文では以下になります。

```
<restConfiguration ...>
  <dataFormatProperty key="xml.out.mustBeJAXBElement"
                      value="false"/>
  ...
</restConfiguration>
```

6. 必要な依存関係をプロジェクトのビルドファイルに追加します。たとえば、Maven ビルドシステムを使用し、Jackson データフォーマットを使用している場合、次の依存関係を Maven POM ファイルに追加します。

```
<?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. アプリケーションを OSGi コンテナーにデプロイする際には、必ず選択したデータフォーマットに必要な機能をインストールしてください。たとえば、Jackson データフォーマット (デフォルト) を使用している場合、以下の Karaf コンソールコマンドを入力して **camel-jackson** 機能をインストールします。

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

また、Fabric 環境にデプロイする場合は、Fabric プロファイルにその機能を追加します。たとえば、プロファイル **MyRestProfile** を使用している場合は、次のコンソールコマンドを入力して機能を追加できます。

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

バインディングモードの設定

bindingMode オプションはデフォルトで **off** になっているため、Java オブジェクトのマーシャリングを有効にするには、明示的に設定する必要があります。表は、サポートされているバインディングモードの一覧を示しています。

注記

Camel 2.16.3 以降では、POJO から JSon/JAXB へのバインディングは、content-type ヘッダーに **json** または **xml** が含まれている場合にのみ発生します。これにより、カスタムのコンテンツタイプを指定することで、メッセージボディーがバインディングを使用してマーシャリングしなくなります。これは、メッセージボディーがカスタムバイナリーのペイロードである場合などに便利です。

表4.2 REST DSL のバインディングモード

バインディングモード	説明
off	バインディングはオフになります **(デフォルト)**。
auto	バインディングは JSON または XML に対して有効になります。このモードでは、Camel は受信メッセージのフォーマットに基づいて JSON または XML (JAXB) を自動選択します。必ずしも両方のデータフォーマットの実装を有効にする必要は**ありません**。JSON の実装と XML の実装のどちらかまたは両方をクラスパスで提供します。
json	バインディングは JSON でのみ有効になります。クラスパスに JSON の実装を用意 **しなければなりません** (デフォルトでは、Camel は **camel-jackson** の実装を有効にしようとします)。

バインディングモード	説明
xml	バインディングは XML でのみ有効です。クラスパスに XML の実装を用意 **しなければなりません** (デフォルトでは、Camel は **camel-jaxb** の実装を有効にしようとします)。
json_xml	バインディングは JSON と XML の両方に対して有効になります。このモードでは、Camel は受信メッセージのフォーマットに基づいて JSON または XML (JAXB) を自動選択します。クラスパスに **両方** のデータフォーマットの実装を用意する必要があります。

Java では、これらのバインディングモード値は、以下の **enum** 型のインスタンスとして表されます。

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

bindingMode が設定できるレベルは、以下のように複数あります。

REST DSL の設定

以下のように、**restConfiguration** ビルダーから **bindingMode** オプションを設定できます。

```
restConfiguration().component("servlet").port(8181).bindingMode(RestBindingMode.json);
```

サービス定義のベースパート

rest() キーワードの直後 (Verb 句の前) に、以下のように **bindingMode** オプションを設定することができます。

```
rest("/user").bindingMode(RestBindingMode.json).get("/{id}").VerbClause
```

Verb 句

Verb 句で **bindingMode** オプションを設定する場合は、以下のようになります。

```
rest("/user")
    .get("/{id}").bindingMode(RestBindingMode.json).to("...");
```

例

Servlet コンポーネントを REST 実装として使用して、REST DSL を使用する方法を示す完全なコード例は、Apache Camel の **camel-example-servlet-rest-blueprint** の例を参照してください。この例は、スタンドアロンの Apache Camel ディストリビューション **apache-camel-2.23.2.fuse-790054-redhat-00001.zip** をインストールして見ることができます。これは、Fuse インストールの **extras/** サブディレクトリーにあります。

スタンドアロン Apache Camel ディストリビューションのインストール後に、以下のディレクトリーでサンプルコードを確認できます。

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

REST 実装として Servlet コンポーネントを設定

camel-example-servlet-rest-blueprint の例では、REST DSL のベースとなる実装は Servlet コンポーネントによって提供されます。Servlet コンポーネントは、例4.1「REST DSL の Servlet コンポーネントの設定」 に示されているように、Blueprint XML ファイルで設定されます。

例4.1 REST DSL の Servlet コンポーネントの設定

```
<?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>
```

Servlet コンポーネントを REST DSL で設定するには、以下の 3 つのレイヤーを設定する必要があります。

REST DSL レイヤー

REST DSL レイヤーは **restConfiguration** 要素によって設定され、**component** 属性を **servlet** という値に設定することで Servlet コンポーネントと統合されます。

Servlet コンポーネントレイヤー

Servlet コンポーネントレイヤーは、クラス **CamelHttpTransportServlet** のインスタンスとして実装され、このサンプルインスタンスには Bean ID **camelServlet** があります。

HTTP コンテナーレイヤー

Servlet コンポーネントは HTTP コンテナーにデプロイする必要があります。Karaf コンテナーには

通常、ポート 8181 の HTTP リクエストをリッスンするデフォルトの HTTP コンテナー (Jetty HTTP コンテナー) が提供されています。Servlet コンポーネントをデフォルトの Jetty HTTP コンテナーにデプロイするには、以下を行います。

a. OSGi サービス (**org.osgi.service.http.HttpService**) への OSGi 参照を取得します。このサービスは、OSGi のデフォルト HTTP サーバーへのアクセスを提供する標準化された OSGi インターフェイスです。

b. ユーティリティークラスのインスタンス (**OsgiServletRegisterer**) を作成して、HTTP コンテナーに Servlet コンポーネントを登録します。**OsgiServletRegisterer** クラスは、Servlet コンポーネントのライフサイクル管理を簡素化するユーティリティーです。このクラスのインスタンスが作成されると、**HttpService** OSGi サービス上の **registerServlet** メソッドが自動的に呼び出され、インスタンスが破棄されると、**unregister** メソッドが自動的に呼び出されます。

必要な依存関係

この例には、REST DSL にとって重要な以下の 2 つの依存関係があります。

Servlet コンポーネント

REST DSL のベースとなる実装を提供します。以下のように Maven POM ファイルで指定します。

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

また、OSGi コンテナーにデプロイする場合、以下のように Servlet コンポーネント機能をインストールする必要があります。

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

Jackson データフォーマット

JSON データフォーマットの実装を提供します。以下のように Maven POM ファイルで指定します。

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

また、OSGi コンテナーにデプロイする場合、以下のように Jackson データフォーマット機能をインストールする必要があります。

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

レスポンス用 の Java 型

サンプルアプリケーションは、HTTP リクエストメッセージとレスポンスメッセージで **User** 型のオブジェクトを渡し合います。Java クラス **User** は、例4.2「JSON レスポンス用ユーザークラス」 で示すように定義されます。

例4.2 JSON レスポンス用ユーザークラス

```
// 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;
    }
}
```

クラス **User** は、JSON データ形式で比較的シンプルに表現されます。たとえば、JSON 形式で表現されたこのクラスのインスタンスは次のようになります。

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

JSON バインディングを使用した REST DSL の例

この例の REST DSL 設定と REST サービス定義を 例4.3「JSON バインディングでの REST DSL の例」に示します。

例4.3 JSON バインディングでの REST DSL の例

```
<?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 オペレーション

例4.3「JSON バインディングでの REST DSL の例」 からの REST サービスは、以下の REST オペレーションを定義します。

GET /camel-example-servlet-rest-blueprint/rest/user/{id}

{id} で識別されたユーザーの詳細を取得します。HTTP レスポンスは JSON 形式で返されます。

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

新しいユーザーを作成します。ユーザーの詳細は PUT メッセージのボディーに含まれ、JSON 形式でエンコードされます (**User** オブジェクトタイプと一致するように)。

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

すべてのユーザーの詳細を取得します。HTTP レスポンスはユーザーの配列で、JSON 形式で返されます。

REST サービスを呼び出すための URL

例4.3「JSON バインディングでの REST DSL の例」 から REST DSL の定義を調べることで、各 REST オペレーションを呼び出すのに必要な URL をまとめることができます。たとえば、指定した ID を持つユーザーの詳細を返す最初の REST オペレーションを呼び出す場合、URL は以下になります。

http://localhost:8181

restConfiguration では、プロトコルのデフォルトは **http** で、ポートは明示的に **8181** に設定されます。

/camel-example-servlet-rest-blueprint/rest

restConfiguration 要素の **contextPath** 属性によって指定されます。

/user

rest 要素の **path** 属性によって指定されます。

/{id}

verb 要素 **get** の **uri** 属性によって指定されます。

したがって、コマンドラインで以下のコマンドを入力することで、**curl** ユーティリティーでこの REST 操作を呼び出すことができます。

```
curl -X GET -H "Accept: application/json" http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/123
```

同様に、残りの REST オペレーションは、以下のサンプルコマンドを入力することにより、**curl** で呼び出すことができます。

```
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. REST DSL の設定

Java DSL を使用した設定

Java では、**restConfiguration()** builder API を使用して REST DSL を設定することができます。たとえば、以下は Servlet コンポーネントをベースの実装として使用するように REST DSL を設定する場合になります。

```
restConfiguration().component("servlet").bindingMode("json").port("8181")
    .contextPath("/camel-example-servlet-rest-blueprint/rest");
```

XML DSL を使用した設定

XML では、**restConfiguration** 要素を使用して REST DSL を設定できます。たとえば、以下は Servlet コンポーネントをベースの実装として使用するように REST DSL を設定する場合になります。

```
<?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>
```

設定オプション

表4.3「REST DSL の設定オプション」 は、**restConfiguration()** ビルダー (Java DSL) または **restConfiguration** 要素 (XML DSL) を使用して、REST DSL を設定するオプションを示しています。

表4.3 REST DSL の設定オプション

Java DSL	XML DSL	説明
component()	**@component**	REST トランスポートとして使用する Camel コンポーネントを指定します (例: **servlet**、**restlet**、**spark-rest** など)。この値は、標準コンポーネント名またはカスタムインスタンスの Bean ID のいずれかになります。このオプションが指定されていない場合、Camel はクラスパス上または Bean レジストリーで **RestConsumerFactory** のインスタンスを探します。
scheme()	**@scheme**	REST サービスの公開に使用するプロトコル。ベースとなる REST の実装に依存しますが、通常は **http** と **https** はサポートされます。デフォルトは **http** です。
host()	**@host**	REST サービスの公開に使用するホスト名。

Java DSL	XML DSL	説明
port()	**@port**	REST サービスの公開に使用するポート番号。 **注意:** この設定は Servlet コンポーネントによって**無視され**、代わりにコンテナーの標準 HTTP ポートが使用されます。Apache Karaf OSGi コンテナーの場合、標準の HTTP ポートは通常 8181 になります。JMX などのツールのためにポート値を明記するとよいでしょう。
contextPath()	**@contextPath**	REST サービスのリーディングコンテキストパスを指定します。これは Servlet などのコンポーネントで使用することができます。これらのコンポーネントは、**curl** の設定を使用してアプリケーションをデプロイします。
hostNameResolver()	**@hostNameResolver**	ホスト名が明示的に設定されていない場合、このリゾルバーによって REST サービスのホストが決定されます。使用できる値は、ホスト名の形式に解決される **RestHostNameResolver.localHostName** (Java DSL) または **localHostName** (XML DSL)、およびドットの 10 進数の IP アドレス形式に解決される **RestHostNameResolver.localIp** (Java DSL) または **localIp** (XML DSL) になります。Camel 2.17 以降では、**RestHostNameResolver.allLocalIp** を使用して、すべてのローカル IP アドレスに解決することができます。 Camel 2.16 までのデフォルトは **localHostName** です。Camel 2.17 以降のデフォルトは **allLocalIp** です。

Java DSL	XML DSL	説明
bindingMode()	**@bindingMode**	JSON または XML 形式のメッセージのバインディングモードを有効にします。使用できる値は **off**、**auto**、**json**、**xml**、または **json_xml** です。デフォルトは **off** です。
skipBindingOnErrorCode()	**@skipBindingOnErrorCode**	HTTP エラーがある場合、出力のバインディングをスキップするかどうかを指定します。これにより、JSON や XML にバインドせず、カスタムエラーメッセージを作成することができます。デフォルトは **true** です。
enableCORS()	**@enableCORS**	**true** の場合、HTTP レスポンスで CORS (オリジン間リソース共有) ヘッダーを有効にします。デフォルトは **false** です。
jsonDataFormat()	**@jsonDataFormat**	Camel が JSON データフォーマットを変換するために使用するコンポーネントを指定します。使用できる値は、**json-jackson**、**json-gson**、**json-xstream** です。デフォルトは **json-jackson** です。
xmlDataFormat()	**@xmlDataFormat**	Camel が XML データフォーマットを変換するために使用するコンポーネントを指定します。使用できる値は **jaxb** です。デフォルトは **jaxb** です。
componentProperty()	**componentProperty**	ベースにある REST 実装に対し、**コンポーネントレベル** の任意のプロパティーを設定できるようにします。
endpointProperty()	**endpointProperty**	ベースにある REST 実装に対し、**エンドポイントレベル** の任意のプロパティーを設定できるようにします。
consumerProperty()	**consumerProperty**	ベースにある REST 実装に対し、**コンシューマーエンドポイント** の任意のプロパティーを設定できます。

Java DSL	XML DSL	説明
dataFormatProperty()	**dataFormatProperty**	ベースにあるデータフォーマットコンポーネント (Jackson や JAXB など) に対し、任意のプロパティーを設定できます。Camel 2.14.1 以降では、以下の接頭辞をプロパティーキーに割り当てることができます。 • **json.in** • **json.out** • **xml.in** • **xml.out** これによって、プロパティー設定を特定の形式 (JSON または XML) および特定のメッセージ方向 (**IN** または **OUT**) だけに適用することができます。
corsHeaderProperty()	**corsHeaders**	カスタム CORS ヘッダをキー/値のペアで指定できるようにします。

デフォルトの CORS ヘッダー

CORS (オリジン間リソース共有) を有効にすると、デフォルトで以下のヘッダーが設定されます。**corsHeaderProperty** DSL コマンド を呼び出すことで、デフォルト設定を任意で上書きできます。

表4.4 デフォルトの CORS ヘッダー

ヘッダーのキー	ヘッダーの値
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**

Jackson JSON 機能の有効化または無効化

dataFormatProperty オプションに以下のキーを設定すると、特定の Jackson JSON 機能を有効または無効にできます。

- **json.in.disableFeatures**
- **json.in.enableFeatures**

たとえば、Jackson の **FAIL_ON_UNKNOWN_PROPERTIES** 機能を無効にする場合は、以下のとおりです (JSON 入力に Java オブジェクトにマップできないプロパティーがある場合に Jackson は失敗します)。

```
restConfiguration().component("jetty")
    .host("localhost").port(getPort())
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("json.in.disableFeatures", "FAIL_ON_UNKNOWN_PROPERTIES");
```

コンマ区切りにすると、**複数** の機能を無効にできます。以下に例を示します。

```
.dataFormatProperty("json.in.disableFeatures",
"FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE");
```

以下の例は、Java DSL で Jackson JSON 機能を有効および無効する方法を示しています。

```
restConfiguration().component("jetty")
    .host("localhost").port(getPort())
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("json.in.disableFeatures",
"FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE")
    .dataFormatProperty("json.in.enableFeatures",
"FAIL_ON_NUMBERS_FOR_ENUMS,USE_BIG_DECIMAL_FOR_FLOATS");
```

以下の例は、XML DSL で Jackson JSON 機能を有効および無効する方法を示しています。

```
<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>
```

有効化または無効化できる Jackson 機能は、以下の Jackson クラスの **enum** ID に対応しています。

- com.fasterxml.jackson.databind.SerializationFeature
- com.fasterxml.jackson.databind.DeserializationFeature
- com.fasterxml.jackson.databind.MapperFeature

4.5. OPENAPI インテグレーション

概要

OpenAPI サービスを使用して、CamelContext 内の REST 定義されたルートやエンドポイントの API ドキュメントを生成することができます。これを行うには、**camel-openapi-java** モジュール (純粋な

Java ベースのモジュール) で Camel REST DSL を使用します。**camel-openapi-java** モジュールは CamelContext と統合したサーブレットを作成し、各 REST エンドポイントから情報を取得して JSON または YAML 形式の API ドキュメントを生成します。

Maven を使用する場合は、**pom.xml** ファイルを編集して **camel-openapi-java** コンポーネントに依存関係を追加します。

```
<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>
```

CamelContext での OpenAPI の有効化

Camel REST DSL で OpenAPI を使用できるようにするには、**apiContextPath()** を呼び出して、OpenAPI で生成された API ドキュメントのコンテキストパスを設定します。以下に例を示します。

```
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 モジュールの設定オプション

下表のオプションを使用して、OpenAPI モジュールを設定することができます。以下のようにオプションを設定します。

- **camel-openapi-java** モジュールをサーブレットとして使用している場合は、**web.xml** ファイルを編集し、設定する設定オプションごとに **init-param** 要素を指定してオプションを設定します。

- Camel REST コンポーネントから **camel-openapi-java** モジュールを使用している場合は、**enableCORS()**、**host()**、または **contextPath()** などの適切な **RestConfigurationDefinition** メソッドを呼び出してオプションを設定します。**api.xxx** オプションを **RestConfigurationDefinition.apiProperty()** メソッドで設定します。

オプション	型	説明
api.contact.email	String	API 関連の連絡先に使用するメールアドレス。
api.contact.name	String	API 関連の連絡先に使用する個人または組織の名前。
api.contact.url	String	API 関連の問い合わせ先の Web サイトへの URL。
apiContextIdListing	ブール値	アプリケーションに複数の **CamelContext** オブジェクトを使用している場合、デフォルトの動作では、現在の **CamelContext** のみに REST エンドポイントが一覧表示されます。REST サービスを実行している JVM で、実行されている個々の **CamelContext** の REST エンドポイントのリストが必要な場合は、このオプションを true に設定します。**apiContextIdListing** が true の場合、OpenAPI はルートパスの **CamelContext** ID (たとえば **/api-docs**) を JSON 形式の名前のリストとして公開します。OpenAPI で生成されたドキュメントにアクセスするには、REST コンテキストパスを **CamelContext** ID に追加します (たとえば **api-docs/myCamel**)。**apiContextIdPattern** オプションを使用して、この出力リストの名前をフィルターリングできます。

オプション	型	説明
apiContextIdPattern	String	コンテキストリストに表示される CamelContext ID のフィルターパターン。正規表現を指定し、ワイルドカードとして * を使用することができます。これは、Camel Intercept 機能で使用されるのと同じパターンマッチング機能です。
api.license.name	String	API に適用されるライセンス名。
api.license.url	String	API に適用されるライセンスの URL。
api.path	String	ドキュメントを生成する REST API が使用可能なパスを設定します (例: **/api-docs**)。相対パスで指定します。たとえば、**http** または **https** は指定しないでください。**camel-openapi-java** モジュールは、ランタイム時に **protocol://host:port/context-path/api-path** の形式で絶対パスを計算します。
api.termsOfService	String	API の利用規約への URL。
api.title	String	アプリケーションの名前。
api.version	String	API のバージョン。デフォルトは 0.0.0 です。
base.path	String	必須。REST サービスが利用できるパスを設定します。相対パスで指定します。たとえば、**http** または **https** は指定しないでください。**camel-openapi-java** モジュールは、ランタイム時に **protocol://host:port/context-path/base.path** の形式で絶対パスを計算します。

オプション	型	説明
cors	ブール値	CORS (オリジン間リソース共有) を有効にするかどうか。これにより、CORS は REST API ドキュメントを閲覧する場合のみ有効になり、REST サービスにアクセスする場合には有効になりません。デフォルトは false です。この表の後で説明するように、代わりに **CorsFilter** オプションを使用することをお勧めします。
host	String	OpenAPI サービスが実行されているホストの名前を指定します。デフォルトでは、**localhost** に基づいてホスト名が計算されます。
schemes	String	使用するプロトコルスキーム。複数の値はコンマで区切ります (例: **"http,https".**)。デフォルトは **http** です。
opeapi.version	String	OpenAPI 仕様のバージョン。デフォルトは 3.0 です。

JSON または YAML 形式の出力

Camel 3.1 以降、**camel-openapi-java** モジュールは JSON と YAML 形式の両方の出力をサポートしています。必要な出力を指定するには、リクエスト URL に **/openapi.json** または **/openapi.yaml** を追加します。リクエスト URL に形式が指定されていない場合、**camel-openapi-java** モジュールは HTTP Accept ヘッダーをチェックして JSON または YAML を許可するかどうかを検出します。両方が受け入れられるか、両方が受け入れられないと検知された場合は、デフォルトの JSON 形式が出力されます。

例

Apache Camel 3.x ディストリビューションで、**camel-example-openapi-cdi** および **camel-example-openapi-java** は **camel-openapi-java** モジュールの使用方法を実証します。

Apache Camel 2.x ディストリビューションで、**camel-example-swagger-cdi** および **camel-example-swagger-java** は **camel-swagger-java** モジュールの使用方法を実証します。

OpenAPI で生成されたドキュメントの充実

Camel 3.1 以降では、名前、説明、データ型、パラメーター型などのパラメーター詳細を定義することで、OpenAPI で生成されたドキュメントを充実することができます。XML DSL を使用している場合は、**param** 要素を指定してこれらの情報を追加します。以下の XML DSL の例は、ID パスパラメーターに関する情報を補足する方法を示しています。

```
<!-- 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>
```

以下は Java DSL の例です。

```
.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})")
```

名前が **body** のパラメーターを定義する場合は、そのパラメーターの型として **body** も指定する必要があります。以下に例を示します。

```
<!-- 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>
```

以下は Java DSL の例です。

```
.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")
```

Apache Camel ディストリビューションの **examples/camel-example-servlet-rest-tomcat** も参照してください。

第5章 メッセージングシステム

概要

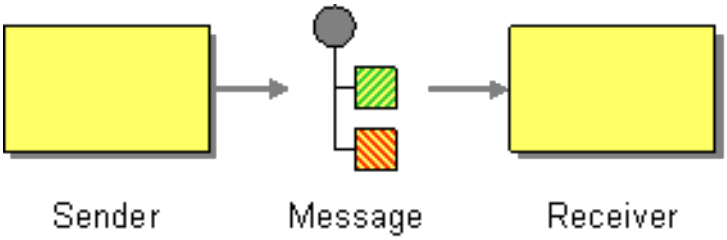
本章では、エンドポイント、メッセージングチャネル、メッセージルーターなどの、メッセージングシステムの基本的な設定要素について紹介します。

5.1. メッセージ

概要

メッセージ とは、メッセージングシステムでデータを送信するための最小単位です (以下の図では灰色の丸で表されています)。複数のパーツが含まれるメッセージなどの場合、メッセージ自体に内部構造がある場合があります。これは、図5.1「Message パターン」 では灰色の丸とつながる図形で表されています。

図5.1 Message パターン



メッセージのタイプ

Apache Camel は以下のメッセージタイプを定義します。

- **In** メッセージ: コンシューマーエンドポイントからプロデューサーエンドポイントへのルートを通過するメッセージ (通常はメッセージのエクスチェンジを開始します)。
- **out** メッセージ: プロデューサーエンドポイントからコンシューマーエンドポイントに戻るルートを通過するメッセージ (通常は **In** メッセージの応答になります)。

これらのメッセージタイプはすべて、**org.apache.camel.Message** インターフェイスによって内部的に表されます。

メッセージの構造

デフォルトでは、Apache Camel は以下の構造をすべてのメッセージタイプに適用します。

- **ヘッダー**: メッセージから抽出されたメタデータまたはヘッダーデータが含まれます。
- **ボディー**: 通常、メッセージ全体が元の形式で含まれます。
- **アタッチメント**: メッセージの添付 (JBI などの特定のメッセージングシステムとの統合に必要です)。

ヘッダー、ボディー、およびアタッチメントに分割することはメッセージの抽象モデルであることを覚えておくことが重要です。Apache Camel は、さまざまなメッセージ形式を生成するさまざまなコンポーネントをサポートします。最終的には、メッセージのヘッダーおよびボディーに何を配置するかを決定する基盤のコンポーネント実装です。

メッセージの関連付け

Apache Camel は内部的に、個別のメッセージの関連付けに使用されるメッセージ ID を記憶します。ただし、Apache Camel がメッセージを照合する最も重要な方法は **エクスチェンジ** オブジェクトを介して行われます。

エクスチェンジオブジェクト

エクスチェンジオブジェクトは、関連するメッセージのコレクションをカプセル化するエンティティーで、関連するメッセージのコレクションは **メッセージエクスチェンジ** と呼ばれ、メッセージのシーケンスを制御するルールは **交換パターン** と呼ばれます。たとえば、一般的な交換パターンには、一方向イベントメッセージ (**In** メッセージ 1 つで設定) と、リクエスト-リプライエクスチェンジ (**In** メッセージ 1 つと、それに続く **Out** メッセージで設定) の 2 つがあります。

メッセージへのアクセス

Java DSL でルーティングルールを定義する場合、以下の DSL ビルダーメソッドを使用してメッセージのヘッダーとボディーにアクセスできます。

- **header(String name)**、**body()** - 現行の **In** メッセージの名前付きヘッダーとボディーを返します。
- **outBody()**: 現在の **Out** メッセージのボディーを返します。

たとえば、**In** メッセージの **username** ヘッダーを設定するには、以下の Java DSL ルートを使用できます。

```
from(SourceURL).setHeader("username", "John.Doe").to(TargetURL);
```

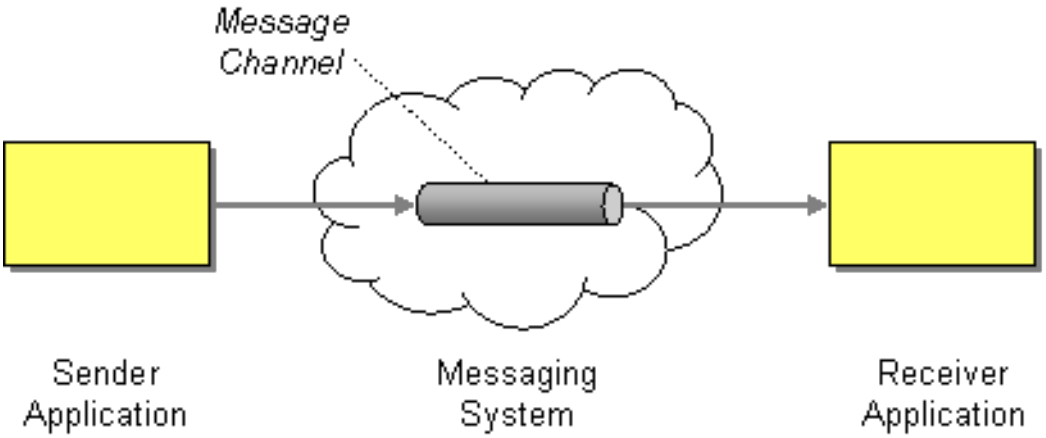
5.2. メッセージチャネル

概要

メッセージチャネル は、メッセージングシステムの論理チャネルです。つまり、異なるメッセージチャネルにメッセージを送信することで、メッセージを異なるメッセージタイプに分類する初歩的な方法を提供します。メッセージチャネルの例として、メッセージキューとメッセージトピックが挙げられます。論理チャネルは物理チャネルと同じでは **ない** ことに注意してください。論理チャネルを物理的に認識する方法はいくつかあります。

Apache Camel では、メッセージチャネルは 図5.2「Message Channel パターン」 のとおり、メッセージ指向コンポーネントのエンドポイント URI によって表されます。

図5.2 Message Channel パターン



メッセージ指向コンポーネント

Apache Camel の以下のメッセージ指向コンポーネントによって、メッセージチャネルの概念がサポートされます。

- ActiveMQ
- JMS
- AMQP

ActiveMQ

ActiveMQ では、メッセージチャネルは **キュー** または **トピック** によって表されます。特定のキューのエンドポイント URI である **QueueName** の形式は次のとおりです。

```
activemq:QueueName
```

特定のトピックのエンドポイント URI である **TopicName** の形式は次のとおりです。

```
activemq:topic:TopicName
```

たとえば、**Foo.Bar** キューにメッセージを送信するには、以下のエンドポイント URI を使用します。

```
activemq:Foo.Bar
```

ActiveMQ コンポーネントの設定に関する詳細や手順については、**Apache Camel Component Reference Guide** の **ActiveMQ** を参照してください。

JMS

Java Messaging Service (JMS) は、さまざまな種類のメッセージシステムにアクセスするために使用される汎用ラッパー層です (たとえば、ActiveMQ、MQSeries、Tibco、BEA、Sonic などをラップするために使用できます)。JMS では、メッセージチャネルはキューまたはトピックによって表されます。特定のキューのエンドポイント URI である **QueueName** の形式は次のとおりです。

```
jms:QueueName
```

特定のトピックのエンドポイント URI である **TopicName** の形式は次のとおりです。

```
jms:topic:TopicName
```

JMS コンポーネントの設定に関する詳細や手順は、**Apache Camel Component Reference Guide** の **Jms** を参照してください。

AMQP

AMQP では、メッセージチャネルはキューまたはトピックで表されます。特定のキューのエンドポイント URI である **QueueName** の形式は次のとおりです。

```
amqp:QueueName
```

特定のトピックのエンドポイント URI である **TopicName** の形式は次のとおりです。

```
amqp:topic:TopicName
```

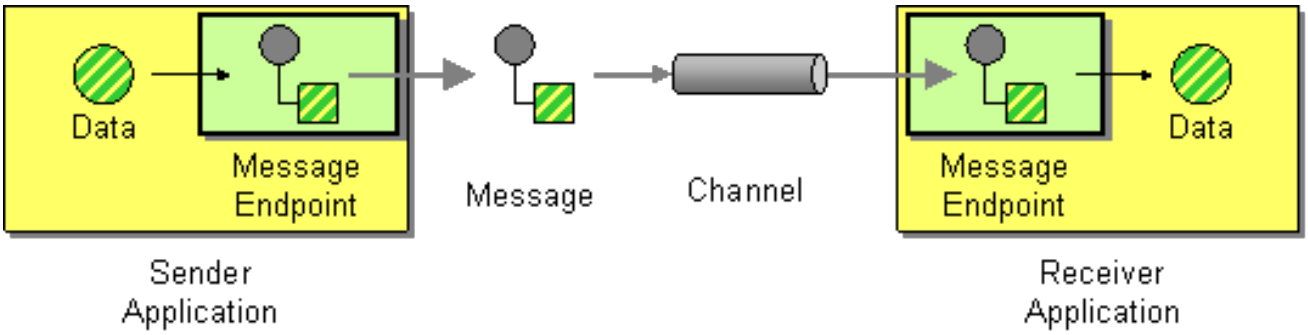
AMQP コンポーネントの設定に関する詳細や手順は、**Apache Camel Component Reference Guide** の **Amqp** を参照してください。

5.3. メッセージエンドポイント

概要

メッセージエンドポイント は、アプリケーションとメッセージングシステム間のインターフェイスです。図5.3「Message Endpoint パターン」 のように、送信者のエンドポイントがあります。これは、プロキシーまたはサービスコンシューマーとも呼ばれ、**In** メッセージの送信を担当します。また、受信者のエンドポイントもあります。これはエンドポイントまたはサービスとも呼ばれ、**In** メッセージの受信を担当します。

図5.3 Message Endpoint パターン



エンドポイントのタイプ

Apache Camel は、2 つの基本タイプのエンドポイントを定義します。

- **コンシューマーエンドポイント**: Apache Camel ルートの最初にあり、受信チャネルから **In** messages を読み取ります (**受信者**エンドポイントと同等です)。
- **プロデューサーエンドポイント**: Apache Camel ルートの最後にあり、 **In** メッセージを送信チャネルに書き込みます (**送信者** エンドポイントと同等です)。複数のプロデューサーエンドポイントでルートを定義できます。

エンドポイント URI

Apache Camel では、エンドポイントはエンドポイント **URI** で表され、通常は以下のようなデータをカプセル化します。

- **コンシューマーエンドポイントのエンドポイント URI**: 特定の場所をアドバタイズします (たとえば、送信者が接続できるサービスを公開する場合など)。または、URI でメッセージキューなどのメッセージソースを指定できます。エンドポイント URI には、エンドポイントの設定を含めることができます。
- **プロデューサーエンドポイントのエンドポイント URI**: メッセージの送信先の詳細と、エンドポイントの設定が含まれます。URI はリモートレシーバーエンドポイントの場所を指定する場合があります。それ以外の場合には、宛先にはキュー名などの抽象的な形式を含むことができます。

Apache Camel のエンドポイント URI の一般的な形式は次のとおりです。

```
ComponentPrefix:ComponentSpecificURI
```

ComponentPrefix は、特定の Apache Camel コンポーネントを識別する URI 接頭辞に置き換えます (サポートされるすべてのコンポーネントの詳細は Apache Camel コンポーネントリファレンス を参照してください)。URI である **ComponentSpecificURI** の残りの部分には特定のコンポーネントによって定義された構文があります。たとえば、JMS キュー **Foo.Bar** に接続するには、以下のようにエンドポイント URI を定義できます。

```
jms:Foo.Bar
```

コンシューマーエンドポイント **file://local/router/messages/foo** を直接プロデューサーエンドポイント **jms:Foo.Bar** に接続するルートを定義するには、以下の Java DSL フラグメントを使用できます。

```
from("file://local/router/messages/foo").to("jms:Foo.Bar");
```

または、以下のように XML で同じルートを定義することもできます。

```
<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>
```

Dynamic To

<toD> パラメーターにより、連結された 1 つ以上の式を使用して、動的に計算されたエンドポイントにメッセージを送信することができます。

デフォルトでは、Simple 言語はエンドポイントの計算に使用されます。以下の例では、ヘッダーによって定義されたエンドポイントにメッセージを送信します。

```
<route>
  <from uri="direct:start"/>
  <toD uri="${header.foo}"/>
</route>
```

Java DSL では、同じコマンドの形式は以下のようになります。

```
from("direct:start")
  .toD("${header.foo}");
```

以下の例のように、URI の前にリテラルを付けることもできます。

```
<route>
  <from uri="direct:start"/>
  <toD uri="mock:${header.foo}"/>
</route>
```

Java DSL では、同じコマンドの形式は以下のようになります。

```
from("direct:start")
  .toD("mock:${header.foo}");
```

上記の例では、header.foo の値が **orange** の場合、URI は **mock:orange** として解決されます。

Simple 以外の言語を使用するには、**language:** パラメーターを定義する必要があります。パート II「ルーティング式と述語言語」を参照してください。

異なる言語を使用する場合の形式では、URI で **language:languagename:** を使用します。たとえば、Xpath を使用する場合は、以下の形式を使用します。

```
<route>
  <from uri="direct:start"/>
  <toD uri="language:xpath:/order/@uri/">
</route>
```

Java DSL では同じ例が以下のようになります。

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

language: を指定しない場合、エンドポイントはコンポーネント名になります。場合によっては、コンポーネントと言語の名前は xquery のように同じになります。

+ 記号を使用して複数の言語を連結できます。以下の例では、URI は Simple 言語と Xpath 言語の組み合わせです。Simple がデフォルトであるため、言語を定義する必要はありません。**+** 記号の後は、**language:xpath** で示される Xpath 命令があります。

```
<route>
  <from uri="direct:start"/>
  <toD uri="jms:${header.base}+language:xpath:/order/@id"/>
</route>
```

Java DSL では形式は以下のようになります。

```
from("direct:start")
  .toD("jms:${header.base}+language:xpath:/order/@id");
```

多くの言語を一度に連結できます。それぞれの言語を **+** で区切り、各言語を **language:languagename** で指定します。

toD で以下のオプションを設定することができます。

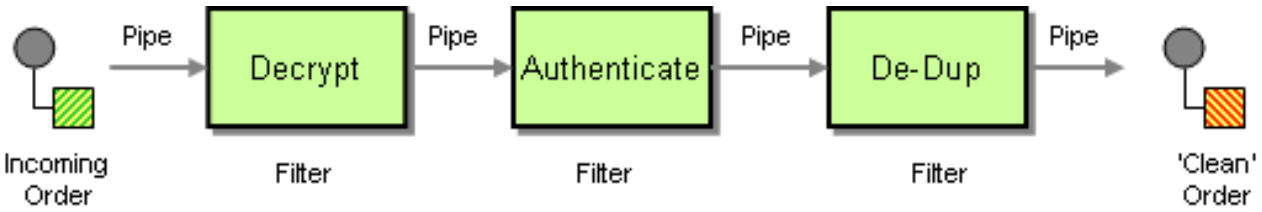
名前	デフォルト値	説明
uri		必須のオプション: 使用する URI。
pattern		エンドポイントに送信する際に使用する特定の交換パターンを設定します。元の MEP は後で復元されます。
cacheSize		再利用のためにプロデューサーをキャッシュする、**ProducerCache** のキャッシュサイズを設定します。デフォルトのキャッシュサイズは 1000 で、他の値が指定されていない場合に使用されます。値を -1 に設定すると、キャッシュを完全にオフにします。
ignoreInvalidEndpoint	**false**	解決できないエンドポイント URI を無視するかどうかを指定します。無効にすると、Camel は無効なエンドポイント URI を特定する例外を出力します。

5.4. パイプとフィルター

概要

図5.4「Pipes and Filters パターン」 に記載されている **Pipes and Filters** パターンは、フィルターチェーンを作成してルートを構築する方法を表しています。フィルターの出力は、パイプラインの次のフィルターの入力に取り込まれます (UNIX の **pipe** コマンドに似ています)。パイプラインアプローチの利点は、サービス (Apache Camel アプリケーションの外部にあるものもあります) を作成して、より複雑な形式のメッセージ処理を作成できることです。

図5.4 Pipes and Filters パターン

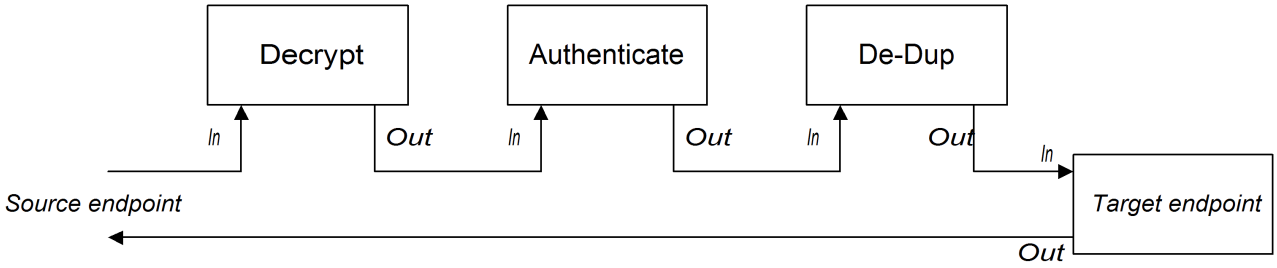


InOut 交換パターンのパイプライン

通常、パイプラインのすべてのエンドポイントには、入力 (**In** メッセージ) および出力 (**Out** メッセー

ジ) があります。これは、**InOut** メッセージ交換パターンと互換性があることを意味しています。**InOut** パイプラインを経由する通常のメッセージフローを 図5.5「InOut エクスチェンジのパイプライン」 に示します。

図5.5 InOut エクスチェンジのパイプライン



パイプラインは、各エンドポイントの出力を次のエンドポイントの入力に接続します。最終的なエンドポイントからの **Out** メッセージは、元の呼び出し元に返されます。以下のように、このパイプラインのルートを定義できます。

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

以下のように XML で同じルートを設定できます。

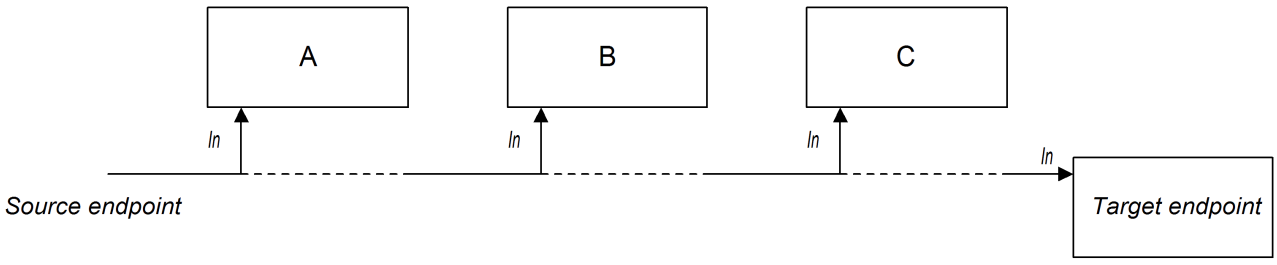
```
<camelContext id="buildPipeline" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:RawOrders"/>
    <to uri="cxf:bean:decrypt"/>
    <to uri="cxf:bean:authenticate"/>
    <to uri="cxf:bean:dedup"/>
    <to uri="jms:CleanOrders"/>
  </route>
</camelContext>
```

XML には専用の pipeline 要素がありません。前述の **from** と **to** 要素の組み合わせは、意味的にはパイプラインと同等です。 「pipeline() および to() DSL コマンドの比較」 を参照してください。

InOnly および RobustInOnly 交換パターンのパイプライン

パイプラインのエンドポイントから利用可能な **Out** メッセージがない場合 (**InOnly** および **RobustInOnly** 交換パターンの場合)、パイプラインは通常の方法で接続できません。この場合、パイプラインは 図5.6「InOnly エクスチェンジのパイプライン」 に示すように、元の **In** メッセージのコピーをパイプラインの各エンドポイントに渡して構築されます。このタイプのパイプラインは、固定の宛先を持つ受信者リストと同等です (「受信者リスト」 を参照)。

図5.6 InOnly エクスチェンジのパイプライン



このパイプラインのルートは、**InOut** パイプラインと同じ構文を使用して定義されます (Java DSL または XML)。

pipeline() および to() DSL コマンドの比較

Java DSL では、以下の構文のいずれかを使用してパイプラインルートを定義できます。

- **pipeline () プロセッサーコマンドの使用**: 以下のように、パイプラインプロセッサーを使用してパイプラインルートを構築します。

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

- **Using the to() command** 以下のように、**to()** コマンドを使用してパイプラインルートを構築します。

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

または、同等の構文を使用することもできます。

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

to() コマンド構文を使用する場合は、パイプラインプロセッサーと常に同等では **ない** ため注意が必要です。Java DSL では、ルートの直前のコマンドで **to()** の意味を変更できます。たとえば、**to()** コマンドの前に **multicast()** コマンドがある場合は、上記のエンドポイントをパイプラインパターンではなく Multicast パターンにバインドします (「Multicast」 を参照)。

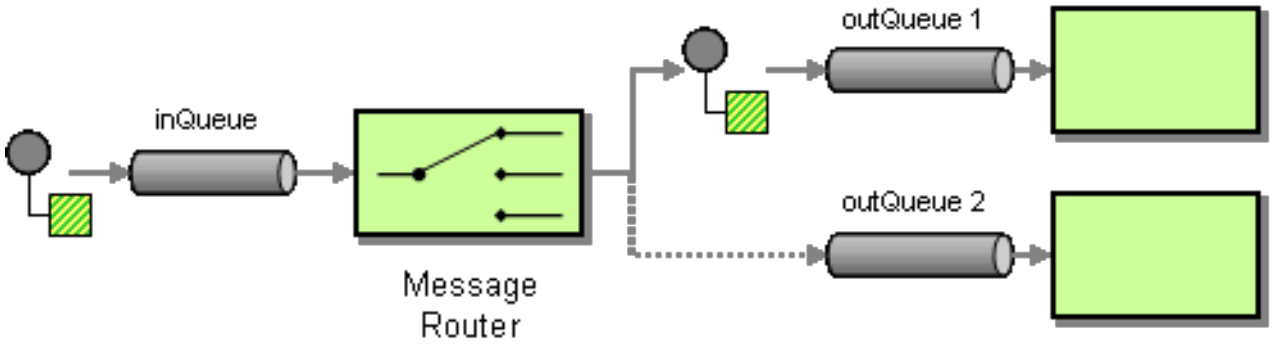
5.5. メッセージルーター

概要

図5.7「Message Router パターン」 に示されている **メッセージルーター** は、単一のコンシューマーエンドポイントからメッセージを消費し、特定の決定基準に基づいて適切なターゲットエンドポイントにリダイレクトするフィルターのタイプです。メッセージルーターはメッセージのリダイレクトのみに関与し、メッセージの内容は変更しません。

しかし、デフォルトでは、Camel がメッセージエクスチェンジを受信者のエンドポイントにルーティングするたびに、元のエクスチェンジオブジェクトのシャローコピーを送信します。シャローコピーでは、メッセージボディー、ヘッダー、アタッチメントなどの元のエクスチェンジの要素は参照のみでコピーされます。リソースを再利用するシャローコピーを送信することで、Camel はパフォーマンスを最適化します。ただし、これらのシャローコピーはすべてリンクされるため、Camel が複数のエンドポイントにメッセージをルーティングする場合は、異なる受信者にルーティングされるコピーにカスタムロジックを適用できないことがトレードオフになります。Camel を有効にして一意なバージョンのメッセージを異なるエンドポイントにルーティングする方法は、送信メッセージへのカスタム処理の適用 を参照してください。

図5.7 Message Router パターン



メッセージルーターは **choice()** プロセッサーを使用して Apache Camel に簡単に実装できます。このプロセッサーでは、**when()** を使用して、代替のターゲットエンドポイントをそれぞれ選択することができます (choice プロセッサーの詳細は、「プロセッサー」 を参照してください)。

Java DSL の例

以下の Java DSL の例は、**foo** ヘッダーの内容に応じて、3 つの代替の宛先 (**seda:a**、**seda:b**、または **seda:c**) にメッセージをルーティングする方法を示しています。

```
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 設定の例

以下の例は、XML で同じルートを設定する方法を示しています。

```
<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>
```

otherwise を使用しない choice

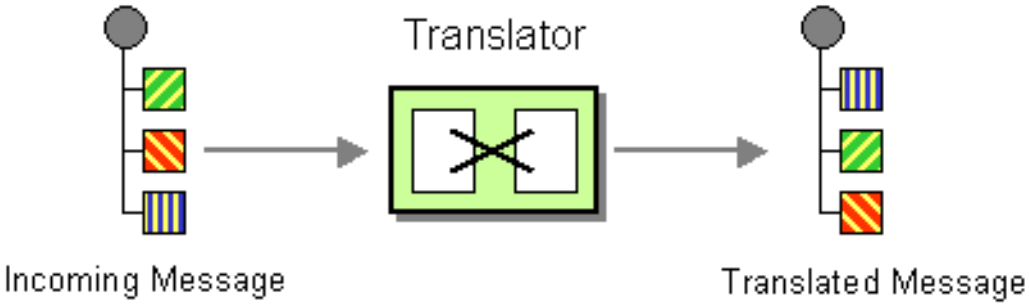
choice() を **otherwise()** 句なしで使用すると、一致しないエクスチェンジはすべてデフォルトでドロップされます。

5.6. メッセージトランスレーター

概要

図5.8「Message Translator パターン」 で示されている **Message Translator**パターンは、メッセージの内容を変更し、異なる形式に変換するコンポーネントを記述します。Apache Camel の Bean インテグレーション機能を使用して、メッセージの変換を実行できます。

図5.8 Message Translator パターン



Bean インテグレーション

登録された Bean でメソッドを呼び出し可能にする Bean インテグレーションを使用して、メッセージを変換できます。たとえば、ID が **myTransformerBean** の Bean でメソッド **myMethodName()** を呼び出すには、以下を実行します。

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

myTransformerBean Bean は Spring XML ファイルまたは JNDI で定義されます。 **beanRef()** で method name パラメーターを省略すると、Bean インテグレーションはメッセージエクスチェンジを確認して、呼び出すメソッド名を推測しようとします。

また、独自の明示的なプロセッサー **Processor** を追加して、以下のように変換を実行することもできます。

```
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");
```

または、DSL を使用して以下のように変換を明示的に設定できます。

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

また、**テンプレート** を使用して、ある宛先からのメッセージを消費し、Velocity や XQuery などのメッセージに変換してから、別の宛先に送信することもできます。**InOnly** 交換パターン (一方向メッセージング) を使用する例は次のとおりです。

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

InOut (request-reply) セマンティクスを使用して、テンプレート生成の応答で ActiveMQ の **My.Queue** キューでリクエストを処理する場合、以下のようなルートを使用して応答を **JMSReplyTo** 宛先に送り返すことができます。

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

5.7. メッセージ履歴

概要

Message History パターンは、粗結合されたシステムで、メッセージのフローの分析およびデバッグを可能にします。メッセージ履歴をメッセージに添付すると、メッセージが送信時以降に通過したすべてのアプリケーションの一覧が表示されます。

Apache Camel では、**getTracedRouteNodes** メソッドを使用すると、Tracer を使用してメッセージフローを追跡するか、UnitOfWork からの Java API を使用して情報にアクセスできます。

ログでの文字長の制限

ロギングメカニズムを使用して Apache Camel を実行すると、メッセージとその内容を随時ログに記録できます。

メッセージによっては、非常に大きなペイロードが含まれる場合があります。デフォルトでは、Apache Camel はログメッセージの最初の 1000 文字のみを表示します。たとえば、以下のログが表示されます。

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

Apache Camel がログのボディーを切り取る際の制限をカスタマイズできます。また、ゼロや -1 などの負の値を設定すると、メッセージボディーはログに記録されません。

Java DSL を使用した制限のカスタマイズ

Java DSL を使用して、Camel プロパティーに制限を設定できます。以下に例を示します。

```
context.getProperties().put(Exchange.LOG_DEBUG_BODY_MAX_CHARS, "500");
```

Spring DSL を使用した制限のカスタマイズ

Spring DSL を使用して、Camel プロパティーに制限を設定できます。以下に例を示します。

```
<camelContext>
   <properties>
      <property key="CamelLogDebugBodyMaxChars" value="500"/>
  </properties>
</camelContext>
```

第6章 メッセージングチャネル

概要

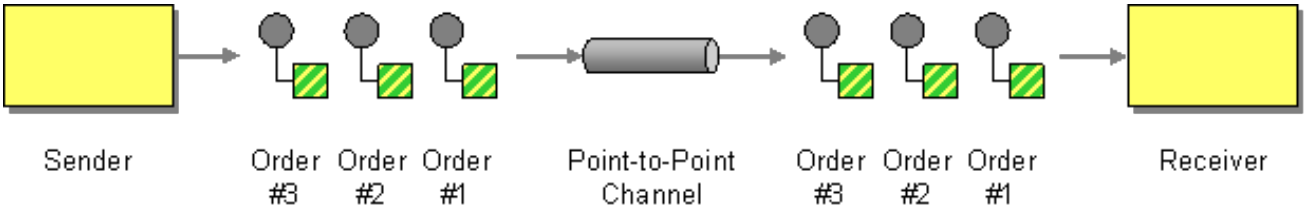
メッセージングチャネルは、メッセージングアプリケーションの組み込みを提供します。本章では、メッセージングシステムで利用可能なメッセージングチャネルの種類と、それらのチャネルのロールについて説明します。

6.1. POINT-TO-POINT CHANNEL

概要

図6.1「Point to Point Channel パターン」 に示されている **Point-to-Point Channel** は、1つの受信側のみが指定のメッセージを消費することを保証するメッセージチャネル です。これは、複数の受信側が同じメッセージを消費できる Publish-Subscribe Channel とは対照的です。特に、Publish-Subscribe Channel では、複数の受信側が同じチャネルにサブスクライブすることが可能です。複数の受信側がメッセージの消費で競合する場合、1つの受信側のみがメッセージを消費するようにするのはメッセージチャネルのロールです。

図6.1 Point to Point Channel パターン



Point to Point Channel をサポートするコンポーネント

以下の Apache Camel コンポーネントは、Point to Point Channel パターンをサポートします。

- JMS
- ActiveMQ
- SEDA
- JPA
- XMPP

JMS

JMS では、Point to Point Channel は **キュー** で表されます。たとえば、**Foo.Bar** という JMS キューのエンドポイント URI を指定できます。

```
jms:queue:Foo.Bar
```

JMS コンポーネントはデフォルトでキューエンドポイントを作成するため、修飾子 **queue:** は任意です。そのため、以下の同等のエンドポイント URI を指定することもできます。

```
jms:Foo.Bar
```

詳細は、**Apache Camel Component Reference Guide** の Jms を参照してください。

ActiveMQ

ActiveMQ では、Point to Point Channel はキューで表されます。たとえば、以下のように **Foo.Bar** という ActiveMQ キューのエンドポイント URI を指定できます。

```
activemq:queue:Foo.Bar
```

詳細は、**Apache Camel Component Reference Guide** の ActiveMQ を参照してください。

SEDA

Apache Camel Staged Event-Driven Architecture (SEDA) コンポーネントは、ブロッキングキューを使用して実装されます。Apache Camel アプリケーションの **内部** にある軽量のポイントツーポイントチャネルを作成する場合は、SEDA コンポーネントを使用します。たとえば、以下のように **SedaQueue** という SEDA キューのエンドポイント URI を指定できます。

```
seda:SedaQueue
```

JPA

Java Persistence API (JPA) コンポーネントは、エンティティー Bean をデータベースに書き出すために使用される EJB 3 永続化の規格です。詳細は、**Apache Camel Component Reference Guide** の JPA を参照してください。

XMPP

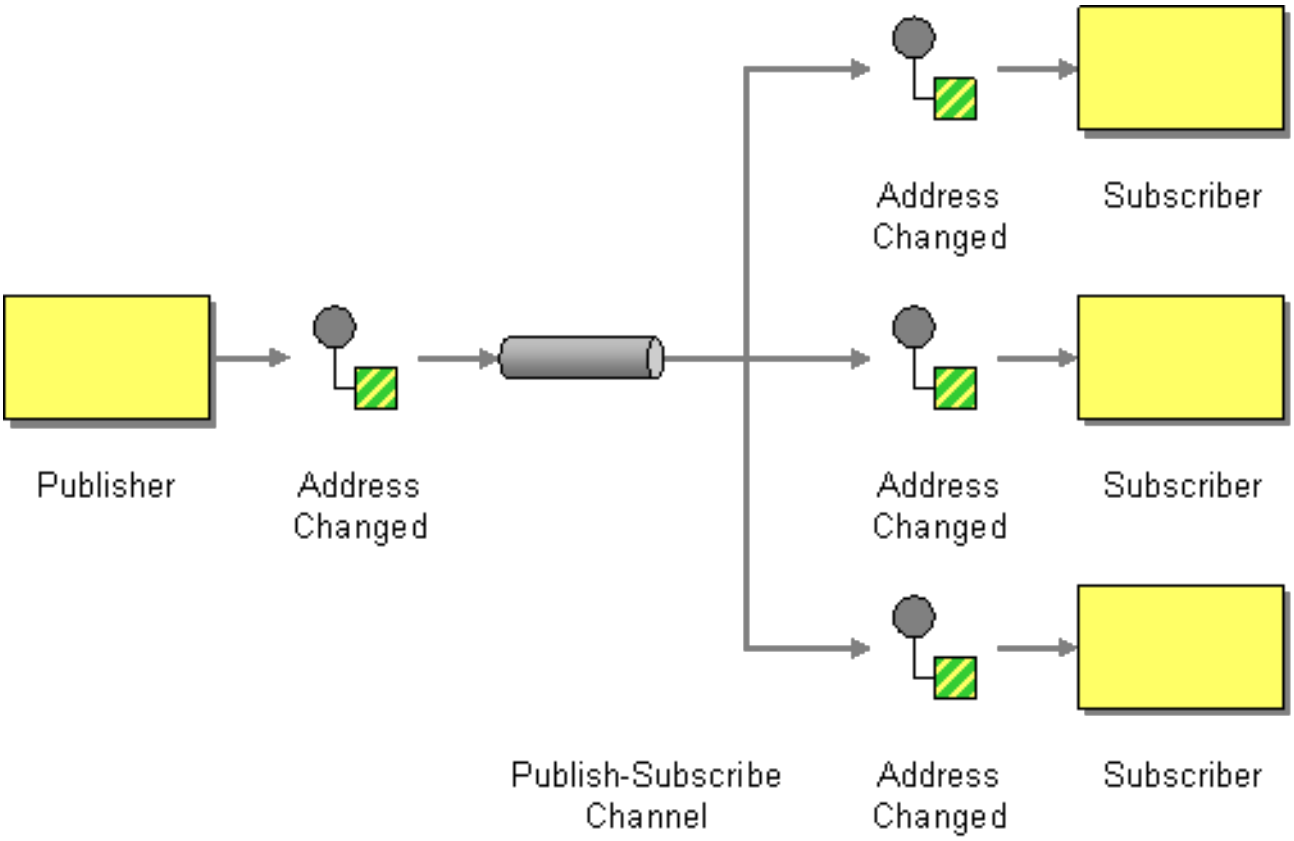
XMPP (Jabber) コンポーネントは、通信でパーソンツーパーソン (Person-to-Person) モードが使用される場合に、Point to Point Channel パターンをサポートします。詳細は、**Apache Camel Component Reference Guide** の XMPP を参照してください。

6.2. PUBLISH-SUBSCRIBE CHANNEL

概要

図6.2「Publish Subscribe Channel パターン」 に示されている**Publish-Subscribe Channel** は、複数のサブスクライバーが任意のメッセージを消費できるようにする 「メッセージチャネル」 です。これは、 「Point-to-Point Channel」 とは対照的です。Publish-Subscribe Channel は、複数のサブスクライバーにイベントや通知をブロードキャストする方法として頻繁に使用されます。

図6.2 Publish Subscribe Channel パターン



Publish-Subscribe Channel をサポートするコンポーネント

以下の Apache Camel コンポーネントは、Publish Subscribe Channel パターンをサポートします。

- JMS
- ActiveMQ
- XMPP
- SEDA (pub-sub で機能する同じ CamelContext で SEDA を使用し、複数のコンシューマーを許可する場合)
- **Apache Camel Component Reference Guide** の **VM** を SEDA とし、同じ JVM 内で使用します。

JMS

JMS では、パブリッシュサブスクライブチャネルは **トピック** で表されます。たとえば、**StockQuotes** という JMS トピックのエンドポイント URI を指定できます。

```
jms:topic:StockQuotes
```

詳細は、**Apache Camel Component Reference Guide** の **Jms** を参照してください。

ActiveMQ

ActiveMQ では、Publish-Subscribe Channel はトピックで表されます。たとえば、以下のように **StockQuotes** という ActiveMQ トピックのエンドポイント URI を指定できます。

```
activemq:topic:StockQuotes
```

詳細は、**Apache Camel Component Reference Guide** の ActiveMQ を参照してください。

XMPP

XMPP (Jabber) コンポーネントは、グループ通信モードで使用される場合に Publish Subscribe Channel パターンをサポートします。詳細は、**Apache Camel Component Reference Guide** の Xmpp を参照してください。

静的サブスクリプションリスト

必要に応じて、Apache Camel アプリケーション内にパブリッシュサブスクライブロジックを実装することもできます。簡単な方法として、ルートの最後にターゲットのエンドポイントがすべて明示的にリストされる **静的サブスクリプションリスト** を定義する方法があります。ただし、この方法は JMS または ActiveMQ トピックほど柔軟ではありません。

Java DSL の例

以下の Java DSL 例は、Publish-Subscribe Channel を単一のパブリッシャー **seda:a** と 3 つのサブスクライバー **seda:b**、**seda:c**、および **seda:d** でシミュレートする方法を示しています。

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

注記

これは **InOnly** メッセージ交換パターンでのみ機能します。

XML 設定の例

以下の例は、XML で同じルートを設定する方法を示しています。

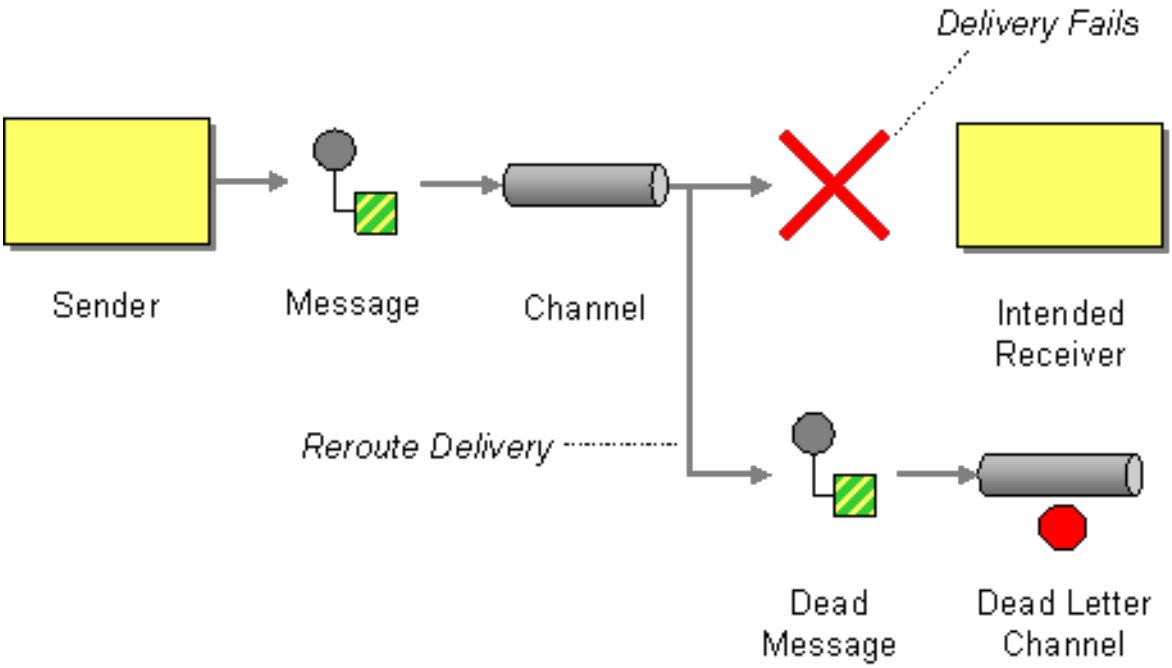
```
<camelContext id="buildStaticRecipientList" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

6.3. DEAD LETTER CHANNEL

概要

図6.3「Dead Letter Channel パターン」 で示されている **Dead Letter Channel** パターンは、メッセージングシステムが目的の受信者にメッセージを配信できない場合に実行するアクションを記述します。これには、配信を再試行する機能などが含まれ、最終的に配信に失敗した場合には、メッセージが Dead Letter Channel に送信され、未達のメッセージをアーカイブします。

図6.3 Dead Letter Channel パターン



Java DSL でのデッドレターチャネルの作成

以下の例は、Java DSL を使用してデッドレターチャネルを作成する方法を示しています。

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

errorHandler() メソッドは Java DSL インターセプターで、現在のルートビルダーで定義された **すべて** のルートがこの設定の影響を受けることを意味します。**deadLetterChannel()** メソッドは、指定の宛先エンドポイント **seda:errors** で新しいデッドレターチャネルを作成する Java DSL コマンドです。

errorHandler() インターセプターは、**すべて** のエラータイプを処理するためのキャッチオールメカニズムを提供します。例外処理により粒度の細かい方法を適用する場合は、代わりに **onException** 句を使用できます (「onException 句」 を参照)。

XML DSL の例

以下のように、XML DSL でデッドレターチャネルを定義できます。

```
<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>
```

再配信ポリシー

通常、配信に失敗した場合、デッドレターチャネルに直接メッセージを送信することはありません。代わりに、最大限度まで再送信を試み、再配信の試行がすべて失敗した場合は、メッセージをデッドレターチャネルに送信します。メッセージの再配信をカスタマイズするには、デッドレターチャネルを設定して **再配信ポリシー** を取得します。たとえば、再配信の最大試行回数を 2 回に指定し、配信試行間の遅延に指数バックオフアルゴリズムを適用するには、以下のようにデッドレターチャネルを設定できます。

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

ここでは、チェーンの関連メソッドを呼び出して、デッドレターチャネルに再配信オプションを設定します (チェーンの各メソッドは現在の **RedeliveryPolicy** オブジェクトの参照を返します)。表6.1「再配信ポリシーの設定」 には、再配信ポリシーの設定に使用できるメソッドがまとめられています。

表6.1 再配信ポリシーの設定

メソッドの署名	デフォルト	説明
allowRedeliveryWhileStopping()	**true**	正常なシャットダウン中またはルートが停止している間、再配信を試行するかどうかを制御します。すでに進行中の配信は停止時に中断され **ません**。
backOffMultiplier(double multiplier)	**2**	指数バックオフが有効な場合は、**m** をバックオフ定数とし、**d** を最初の遅延とします。その後、再配信試行シーケンスは以下のようになります。 d, m*d, m*m*d, m*m*m*d, ...
collisionAvoidancePercent(double collisionAvoidancePercent)	**15**	競合の回避が有効になっている場合は、**p** を競合回避の割合 (パーセント) にします。競合回避ポリシーは、現在の値にその **p%** を足し引きした値を最大値および最小値とするランダムな値で、次の遅延を調整します。

メソッドの署名	デフォルト	説明
deadLetterHandleNewException	**true**	Camel 2.15: デッドレターチャネルでメッセージの処理中に発生する例外を処理するかどうかを指定します。**true** の場合は、例外が処理され、WARN レベルでログに記録されます (そのため、デッドレターチャネルの完了が保証されます)。**false** の場合は例外が処理されないため、デッドレターチャネルは失敗し、新しい例外が伝播されます。
delayPattern(String delayPattern)	**なし**	Apache Camel 2.0: 「Redeliver Delay パターン」 を参照してください。
disableRedelivery()	**true**	Apache Camel 2.0: 再配信機能を無効にします。再配信を有効にするには、**maximumRedeliveries()** を正の整数値に設定します。
handled(boolean handled)	**true**	Apache Camel 2.0: **true** の場合は、メッセージがデッドレターチャネルに移動されたときに現在の例外が消去されます。**false** の場合は、例外はクライアントに伝播されます。
initialRedeliveryDelay(long initialRedeliveryDelay)	**1000**	最初の再配信を試みるまでの遅延 (ミリ秒単位) を指定します。
logNewException	**true**	デッドレターチャネルで例外が発生した場合に WARN レベルでログに記録するかどうかを指定します。
logStackTrace(boolean logStackTrace)	**false**	Apache Camel 2.0: **true** の場合は、JVM スタックトレースがエラーログに含まれます。
maximumRedeliveries(int maximumRedeliveries)	**0**	Apache Camel 2.0: 配信の最大試行回数。

メソッドの署名	デフォルト	説明
maximumRedeliveryDelay(long maxDelay)	**60000**	Apache Camel 2.0: 指数バックオフストラテジーを使用する場合 (**useExponentialBackOff()** を参照)、理論的に再配信の遅延が制限なく増加する可能性があります。このプロパティーは再配信の遅延の上限を指定します (ミリ秒単位)。
onRedelivery(Processor processor)	なし	Apache Camel 2.0: 再配信を試みる前に呼び出されるプロセッサーを設定します。
redeliveryDelay(long int)	**0**	Apache Camel 2.0: 再配信の試行間の遅延 (ミリ秒単位) を指定します。Apache Camel 2.16.0: デフォルトの再配信遅延は 1 秒です。
retriesExhaustedLogLevel(LoggingLevel logLevel)	**LoggingLevel.ERROR**	Apache Camel 2.0: 配信の失敗をログに記録するログレベルを指定します (**org.apache.camel.LoggingLevel** 定数として指定されます)。
retryAttemptedLogLevel(LoggingLevel logLevel)	**LoggingLevel.DEBUG**	Apache Camel 2.0: 再配信の試行に対するログレベルを指定します (**org.apache.camel.LoggingLevel** 定数として指定されます)。
useCollisionAvoidance()	**false**	競合の回避を有効にします。これにより、一定のランダム化をバックオフのタイミングに追加して競合の可能性を低減します。
useOriginalMessage()	**false**	Apache Camel 2.0: この機能が有効な場合、デッドレターチャネルに送信されたメッセージは、ルートの開始時に存在した (**from()** ノードで) **元**のメッセージエクスチェンジのコピーになります。
useExponentialBackOff()	**false**	指数バックオフを有効にします。

再配信ヘッダー

Apache Camel がメッセージの再配信を試みると、表6.2「デッドレター再配信ヘッダー」 に記載されているヘッダーを In メッセージに自動設定します。

表6.2 デッドレター再配信ヘッダー

ヘッダー名	型	説明
CamelRedeliveryCounter	**Integer**	Apache Camel 2.0: 配信に失敗した回数を返します。この値は、**Exchange.REDELIVERY_COUNTER** でも設定されます。
CamelRedelivered	**ブール値**	Apache Camel 2.0: 再配信が 1 回以上試行された場合は true です。この値は **Exchange.REDELIVERED** でも設定されます。
CamelRedeliveryMaxCounter	**Integer**	Apache Camel 2.6: 再配信の最大設定を保持します (**Exchange.REDELIVERY_MAX_COUNTER** エクスチェンジ プロパティーにも設定されます)。**retryWhile** を使用する場合や、再配信の最大回数が無制限に設定されている場合は、このヘッダーは設定されません。

再配信エクスチェンジプロパティー

Apache Camel がメッセージの再配信を試みると、表6.3「再配信エクスチェンジプロパティー」 に記載されているエクスチェンジプロパティーを自動設定します。

表6.3 再配信エクスチェンジプロパティー

エクスチェンジプロパティー名	型	説明
Exchange.FAILURE_ROUTE_ID	**String**	失敗したルートのルート ID を提供します。このプロパティーのリテラル名は **CamelFailureRouteId** です。

元のメッセージの使用

Apache Camel 2.0 で利用可能: エクスチェンジオブジェクトはルートを通過する際に変更される可能性があります。そのため、例外が発生したときに現行であるエクスチェンジがデッドレターチャネルの保存に適したコピーであるとは限りません。多くの場合、ルートによる変換の対象となる前に、ルート開始時に到達したメッセージをログに記録することが推奨されます。たとえば、以下のルートを見てみましょう。

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

上記のルートは受信 JMS メッセージをリッスンした後、**validateOrder**、**transformOrder**、および **handleOrder** の Bean のシーケンスを使用してメッセージを処理します。ただし、エラーが発生した場合にメッセージがどの状態であるかは分かりません。**transformOrder** Bean の前または後にエラーが発生しましたか ?以下のように **useOriginalMessage** オプションを有効にすると、**jms:queue:order:input** からの元のメッセージのログを確実に Dead Letter Chanel に記録することができます。

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

Redeliver Delay パターン

Apache Camel 2.0 で利用可能: delayPattern オプションは、再配信回数の特定範囲に遅延を指定するために使用されます。遅延パターンの構文: **limit1:delay1;limit2:delay2;limit3:delay3;...**。各 delayN は範囲 **limitN ⇐ redeliveryCount < limitN+1** で再配信するように適用されます

たとえば、パターン **5:1000;10:5000;20:20000** について考えてみましょう。このパターンでは、3 つのグループが定義され、以下の再配信の遅延が発生します。

- 1 から 4 の試行 = 0 ミリ秒 (最初のグループは 5 で始まるため) 。
- 5 から 9 の試行 = 1000 ミリ秒 (最初のグループ)。
- 10 から 19 の試行 = 5000 ミリ秒 (2 番目のグループ)。
- 20 以上の試行 = 20000 ミリ秒 (最後のグループ)。

制限 1 を加えてグループを開始し、開始遅延を定義できます。たとえば、**1:1000;5:5000** では以下の再配信の遅延が発生します。

- 1 から 4 の試行 = 1000 ミリ (最初のグループ)。
- 5 以上の試行 = 5000 ミリ (最後のグループ)。

次の遅延を前の遅延よりも長くする必要はありません。あらゆる遅延値を使用できます。たとえば、Delay パターン **1:5000;3:1000** は 5 秒の遅延で始まり、遅延を 1 秒に減らします。

失敗したエンドポイント

Apache Camel ルートメッセージ時に、エクスチェンジが送信された **最後** のエンドポイントが含まれるエクスチェンジプロパティーを更新します。したがって、以下のコードを使用して、現在のエクスチェンジが最後に送信された宛先の URI を取得できます。

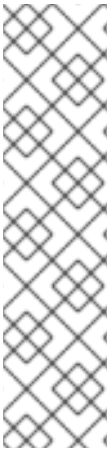
```
// Java
String lastEndpointUri = exchange.getProperty(Exchange.TO_ENDPOINT, String.class);
```

Exchange.TO_ENDPOINT は **CamelToEndpoint** と同等の文字列の定数になります。このプロパティーは、Camel がメッセージを **任意** のエンドポイントに送信するたびに更新されます。

ルーティング中にエラーが発生し、エクスチェンジがデッドレターキューに移動された場合、Apache Camel は **CamelFailureEndpoint** という名前のプロパティーを追加で設定します。これは、エラーが発生する前にエクスチェンジが最後に送信された宛先を特定します。したがって、以下のコードを使用すると、デッドレターキュー内から失敗したエンドポイントにアクセスできます。

```
// Java
String failedEndpointUri = exchange.getProperty(Exchange.FAILURE_ENDPOINT, String.class);
```

Exchange.FAILURE_ENDPOINT は、**CamelFailureEndpoint** と同等の文字列定数です。



注記

これらのプロパティーは、指定の宛先エンドポイントの処理が完了した**後**に障害が発生した場合でも、現在のエクスチェンジで設定された状態を維持します。たとえば、以下のルートを見てみましょう。

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

foo Bean で障害が発生したと仮定します。この場合、**Exchange.TO_ENDPOINT** プロパティーと **Exchange.FAILURE_ENDPOINT** プロパティーに値が含まれ続けます。

onRedelivery プロセッサー

Dead Letter Channel が再配信を実行する場合、再配信を試みる **直前** に実行される **Processor** を設定できます。これは、メッセージを再配信する前に変更する必要がある場合に使用できます。

たとえば、以下の Dead Letter Channel は、エクスチェンジの再配信前に **MyRedeliverProcessor** を呼び出すように設定されます。

```
// 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));
```

ここで **MyRedeliveryProcessor** プロセスは以下のように実装されます。

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

シャットダウンまたは停止中の再配信の制御

ルートを停止したり、正常なシャットダウンを開始する場合、再配信の試行を継続するのがエラー処理のデフォルトの挙動になります。通常、これは望ましい動作ではないため、以下の例のように、**allowRedeliveryWhileStopping** オプションを **false** に設定すると、シャットダウンまたは停止中に再配信を無効にすることができます。

```
errorHandler(deadLetterChannel("jms:queue:dead")
    .allowRedeliveryWhileStopping(false)
    .maximumRedeliveries(20)
    .redeliveryDelay(1000)
    .retryAttemptedLogLevel(LoggingLevel.INFO));
```

注記

後方互換性の理由から、**allowRedelivery whileStopping** オプションはデフォルトで **true** になります。ただし、強行なシャットダウン中は、このオプションの設定に関係なく、再配信が常に抑制されます (たとえば、正常なシャットダウンがタイムアウトした場合など)。

onExceptionOccurred プロセッサーの使用

Dead Letter Channel は、例外発生後にメッセージのカスタム処理を可能にする onExceptionOccurred プロセッサーをサポートします。これは、カスタムロギングにも使用できます。onExceptionOccurred プロセッサーから出力される新しい例外は WARN としてログに記録され、無視されます。既存の例外を上書きすることはありません。

onRedelivery プロセッサーと onExceptionOccurred プロセッサーの違いは、onRedelivery プロセッサーは再配信の試行直前に処理できることです。ただし、例外の発生直後には処理できません。たとえば、再配信を試行する間隔で 5 秒の遅延が発生するようにエラーハンドラーを設定すると、再配信プロセスは例外発生から 5 秒後に呼び出されます。

以下の例は、例外発生時にカスタムロギングを実行する方法を示しています。onExceptionOccurred がカスタムプロセッサーを使用するように設定する必要があります。

```
errorHandler(defaultErrorHandler().maximumRedeliveries(3).redeliveryDelay(5000).onExceptionOccurred(myProcessor));
```

onException 句

ルートビルダーで **errorHandler()** インターセプターを使用する代わりに、さまざまな例外タイプに異なる再配信ポリシーとデッドレターチャネルを定義する、一連の **onException()** 句を定義できます。たとえば、**NullPointerException**、**IOException**、**Exception** タイプごとに異なる動作を定義するには、Java DSL を使用してルートビルダーに以下のルールを定義できます。

```
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");
```

再配信オプションは、再配信ポリシーメソッドをチェーンして指定されます (表6.1「再配信ポリシーの設定」 のように)。また、**to()** DSL コマンドを使用して Dead Letter Channel のエンドポイントを指定します。**onException()** 句で他の Java DSL コマンドを呼び出すこともできます。たとえば、前述の例は **setHeader()** を呼び出して、**messageInfo** という名前のメッセージヘッダーにエラーの情報を記録します。

この例では、**NullPointerException** および **IOException** 例外タイプが特別に設定されています。その他のすべての例外タイプは、汎用 **Exception** 例外インターセプターによって処理されます。デフォルトでは、Apache Camel は出力された例外に最も一致する例外インターセプターを適用します。完全に一致するものが見つからない場合は、最も近いベースタイプなどとの一致を試みます。最後に、他のインターセプターと一致しない場合、その **Exception** タイプのインターセプターは残りの例外すべてと一致します。

OnPrepareFailure

デッドレターキューにエクスチェンジを渡す前に、**onPrepare** オプションを使用してカスタムプロセッサーがエクスチェンジを準備できるようにすることができます。これにより、エクスチェンジ失敗の原因など、エクスチェンジに関する情報を追加できます。たとえば、以下のプロセッサーは例外メッセージが含まれるヘッダーを追加します。

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

以下のように、プロセッサーを使用するようにエラーハンドラーを設定できます。

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

ただし、**onPrepare** オプションは、デフォルトのエラーハンドラーを使用して使用することもできます。

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

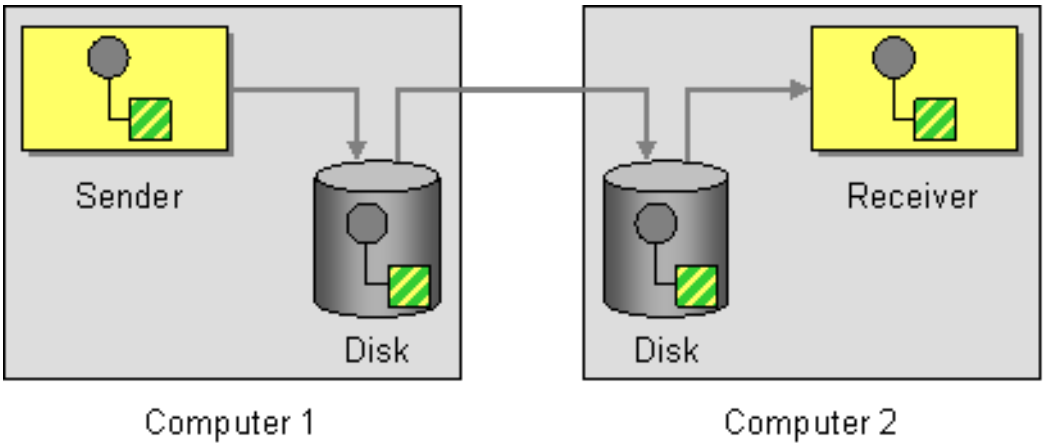
```
<errorHandler id="dlc" type="DeadLetterChannel" deadLetterUri="jms:dead"
onPrepareFailureRef="myPrepare"/>
```

6.4. GUARANTEED DELIVERY

概要

Guaranteed Delivery (保証付き配信) とは、メッセージがメッセージチャネルに配置されると、アプリケーションの一部が失敗してもメッセージが宛先に到達することを保証することです。通常は 図 6.4「Guaranteed Delivery パターン」 のように、宛先への配信を試行する前にメッセージを永続ストレージに書き込むことで、メッセージングシステムは Guaranteed Delivery パターンを実装します。

図6.4 Guaranteed Delivery パターン



Guaranteed Delivery をサポートするコンポーネント

以下の Apache Camel コンポーネントは Guranteed Delivery パターンをサポートします。

- JMS
- ActiveMQ
- ActiveMQ ジャーナル
- **Apache Camel Component Reference Guide**の**File Component**

JMS

JMS では、**deliveryPersistent** クエリーオプションはメッセージの永続ストレージが有効であるかどうかを示します。永続的な配信を有効にするのがデフォルトの動作であるため、通常はこのオプションを設定する必要はありません。Guaranteed Delivery の詳細をすべて設定するには、JMS プロバイダーで設定オプションを設定する必要があります。これらの情報は、使用している JMS プロバイダーによって異なります。たとえば、MQSeries、TibCo、BEA、Sonic などがありますが、いずれも Guaranteed Delivery をサポートするためにさまざまサービスを提供しています。

詳細は、**Apache Camel Component Reference Guide** の Jms を参照してください。

ActiveMQ

ActiveMQ では、メッセージの永続性はデフォルトで有効になっています。ActiveMQ はバージョン 5 以降、AMQ メッセージストアをデフォルトの永続メカニズムとして使用します。ActiveMQ でメッセージの永続化を有効にする方法は複数あります。

最も簡単なオプション (図6.4「Guaranteed Delivery パターン」 とは異なる) は、中央のブローカーで永続性を有効にし、信頼できるプロトコルを使用してそのブローカーに接続することです。メッセージが中央のブローカーに送信された後、コンシューマーへの配信が保証されます。たとえば、Apache Camel 設定ファイル **META-INF/spring/camel-context.xml** では、以下のように OpenWire/TCP プロトコルを使用して中央ブローカーに接続するように ActiveMQ コンポーネントを設定できます。

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://somehost:61616"/>
  </bean>
  ...
</beans>
```

リモートエンドポイントに送信される前にメッセージがローカルに保存されるアーキテクチャーを実装する場合 (図6.4「Guaranteed Delivery パターン」 のように)、Apache Camel アプリケーションで組み込みブローカーをインスタンス化してこれを行います。これは、ActiveMQ Peer-to-Peer プロトコルを使用すると簡単に実現できます。これにより、暗黙的に埋め込みブローカーが作成され、他のピアエンドポイントと通信します。たとえば、ActiveMQ コンポーネントを **camel-context.xml** で以下のように設定し、**GroupA** 内のすべてのピアに接続するようにします。

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="peer://GroupA/broker1"/>
  </bean>
  ...
</beans>
```

broker1 は、埋め込みブローカーのブローカー名に置き換えます (グループの他のピアは異なるブローカー名を使用する必要があります)。Peer-to-Peer プロトコルの 1 つの制限は、IP マルチキャストに依存してグループ内の他のピアを見つけることです。これにより、ワイドエリアネットワークでの使用には適していません (また、IP マルチキャストが有効になっていない一部のローカルエリアネットワークにも適していません)。

組み込みブローカーインスタンスに接続する ActiveMQ の VM プロトコルを利用すると、ActiveMQ コンポーネントでより柔軟に組み込みブローカーを作成できます。必要な名前のブローカーが存在しない場合は、VM プロトコルによって自動的に作成されます。このメカニズムを使用すると、カスタム設定で組み込みブローカーを作成できます。以下に例を示します。

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="vm://broker1?brokerConfig=xbean:activemq.xml"/>
  </bean>
  ...
</beans>
```

activemq.xml は、組み込みブローカーインスタンスを設定する ActiveMQ ファイルに置き換えます。ActiveMQ 設定ファイル内で、以下の永続メカニズムのいずれかを有効にすることができます。

- **AMQ 永続化 (デフォルト)**: ActiveMQ にネイティブな高速で信頼できるメッセージストア。詳細は amqPersistenceAdapter および AMQ Message Store を参照してください。
- **JDBC 永続化**: JDBC を使用して、JDBC 互換データベースにメッセージを格納します。詳細は、jdbcPersistenceAdapter および ActiveMQ Persistence を参照してください。
- **ジャーナル永続化**: メッセージをローリングログファイルに格納する高速の永続化メカニズム。詳細は journalPersistenceAdapter および ActiveMQ Persistence を参照してください。
- **Kaha 永続性**: ActiveMQ に特化して開発された永続メカニズム。詳細は kahaPersistenceAdapter および ActiveMQ Persistence を参照してください。

詳細は、**Apache Camel Component Reference Guide** の **ActiveMQ** を参照してください。

ActiveMQ ジャーナル

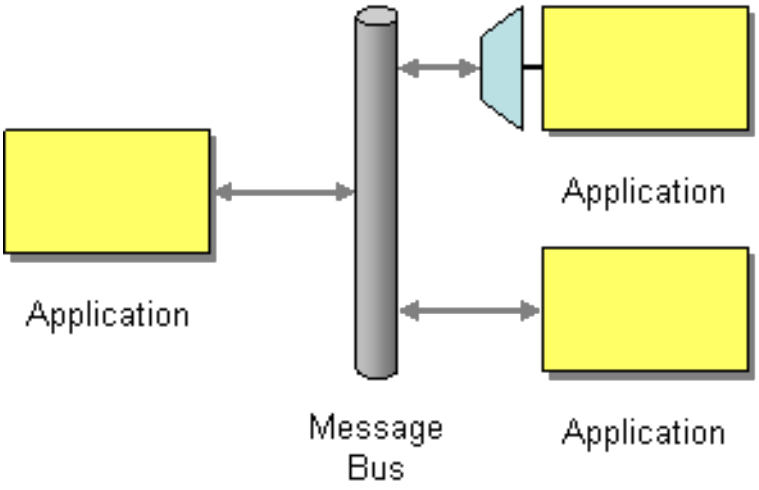
ActiveMQ Journal コンポーネントは、複数の同時実行プロデューサーがメッセージをキューに書き込み、アクティブなコンシューマーが 1 つのみである特殊なユースケースに対して最適化されています。メッセージはローリングログファイルに格納され、効率を向上するために同時書き込みは集約されます。

6.5. MESSAGE BUS

概要

Message Bus は、図6.5「Message Bus パターン」 に示されているメッセージングアーキテクチャーを意味します。これにより、多様なコンピューティングプラットフォーム上で実行されている多様なアプリケーションに接続できます。実質的に、Message Bus は Apache Camel とそのコンポーネントによって設定されます。

図6.5 Message Bus パターン



Message Bus パターンの以下の機能は Apache Camel に反映されます。

- **共通通信インフラストラクチャー**: ルーター自体が Apache Camel の共通通信インフラストラクチャーの中核を提供します。しかし、一部のメッセージバスアーキテクチャーとは対照的に、Apache Camel は異種インフラストラクチャーを提供します。このインフラストラクチャーでは、多種多様なトランスポートとメッセージ形式を使用してメッセージをバスに送信できます。
- **アダプター**: 必要に応じて、Apache Camel は異なるトランスポートを使用してメッセージ形式

を変換し、メッセージを伝播できます。実質的に、Apache Camel はアダプターのように動作できるため、外部アプリケーションはメッセージングプロトコルをリファクタリングせずに Message Bus に接続できます。
場合によっては、アダプターを直接外部アプリケーションに統合することもできます。たとえば、サービスが JAX-WS および JAXB マッピングを使用して実装される Apache CXF を使用してアプリケーションを開発する場合は、多種多様のトランスポートをサービスにバインドできます。これらのトランスポートバインディングはアダプターとして機能します。

第7章 メッセージの構築

概要

メッセージ構築パターンでは、システムを通過するメッセージのさまざまな形式と関数が記述されます。

7.1. 相関識別子

概要

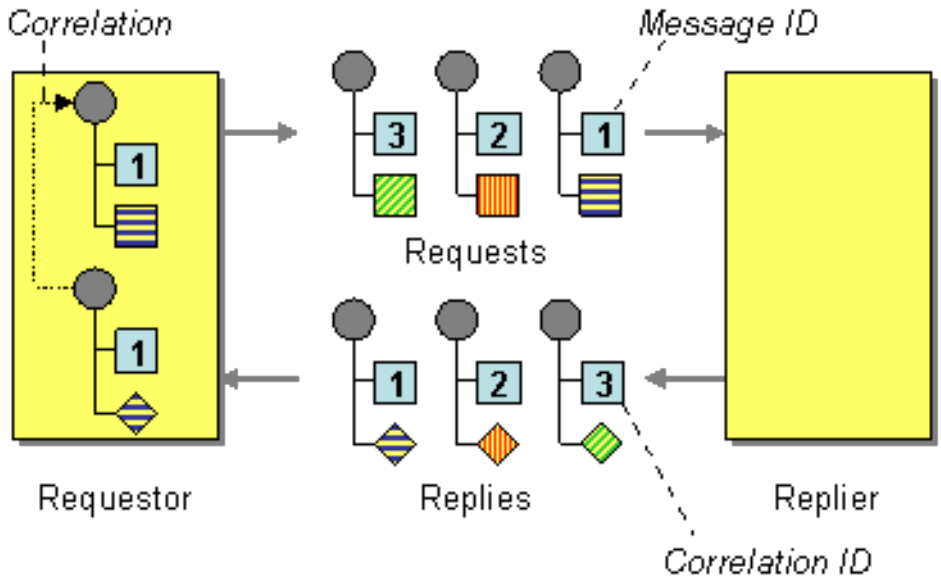
図7.1「相関識別子パターン」 の **相関識別子** (Correlation Identifier) パターンは 、非同期メッセージングシステムを使用してリクエスト-リプライ型プロトコルを実装する場合にリクエストメッセージとリプライメッセージの照合方法を記述します。リクエストメッセージはリクエストメッセージを識別する一意のトークンである **リクエスト ID** によって生成されます。リプライメッセージには、一致するリクエスト ID が含まれるトークンである **相関 ID** が含まれる必要があります。

Apache Camel は、メッセージでヘッダーを取得または設定することにより、EIP パターンからの相関識別子をサポートします。

ActiveMQ または JMS コンポーネントを使用する場合、相関識別子ヘッダーは **JMSCorrelationID** と呼ばれます。独自の相関識別子をメッセージ交換に追加すると、1 つの会話 (またはビジネスプロセス) で複数のメッセージを一緒に関連付けることができます。通常、相関識別子は Apache Camel メッセージヘッダーに格納されます。

一部の EIP パターンでは、サブメッセージがスピンオフされます。この場合、Apache Camel は、ソースエクスチェンジにリンクする **Exchange.CORRELATION_ID** キーのあるプロパティーとして相関 ID をエクスチェンジに追加します。たとえば、Splitter、Multicast、Recipient List、および Wire Tap EIP がこれを行います。

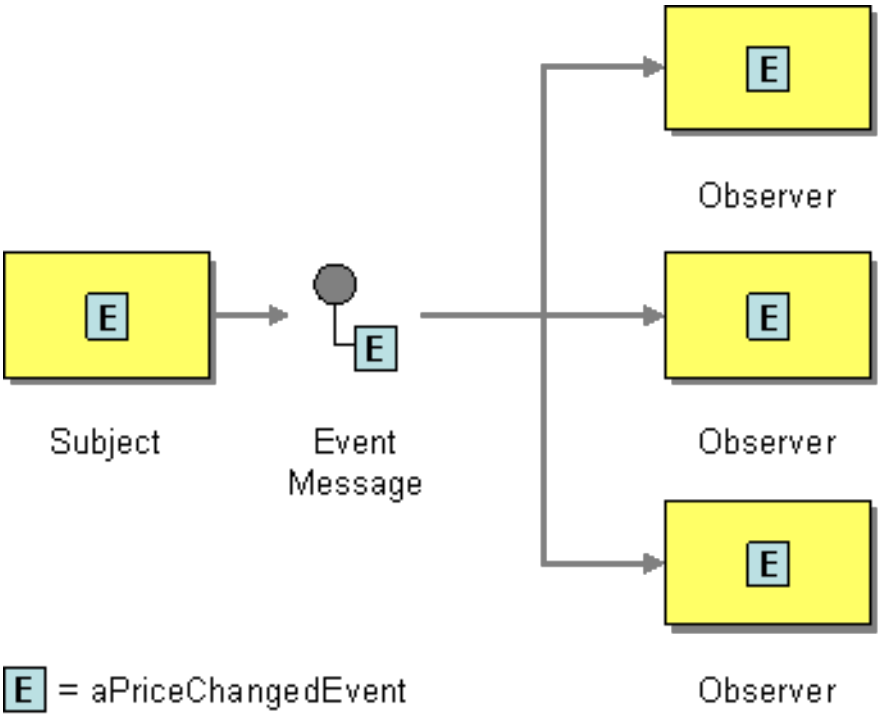
図7.1 相関識別子パターン



7.2. イベントメッセージ

イベントメッセージ

Camel は、一方向のイベントメッセージを示すために **InOnly** に設定できる メッセージ で Exchange パターンをサポートすることにより、エンタープライズ統合パターン からの イベントメッセージ をサポートします。その後、Camel の Apache Camel コンポーネントリファレンス は基盤のトランスポートまたはプロトコルを使用して、このパターンを実装します。



多くの Apache Camel コンポーネントリファレンス のデフォルト動作は、JMS、File、または SEDA などに対して InOnly です。

明示的な **InOnly** の指定

デフォルトが InOut であるコンポーネントを使用している場合、pattern プロパティーを使用して、エンドポイントの メッセージ交換パターン を上書きできます。

```
foo:bar?exchangePattern=InOnly
```

Camel 2.0 以降では、DSL を使用して メッセージ交換パターン を指定できます。

Fluent Builder (流れるようなビルダー) の使用

```
from("mq:someQueue").
  inOnly().
  bean(Foo.class);
```

または、明示的なパターンでエンドポイントを呼び出すこともできます。

```
from("mq:someQueue").
  inOnly("mq:anotherQueue");
```

Spring XML エクステンション の使用

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

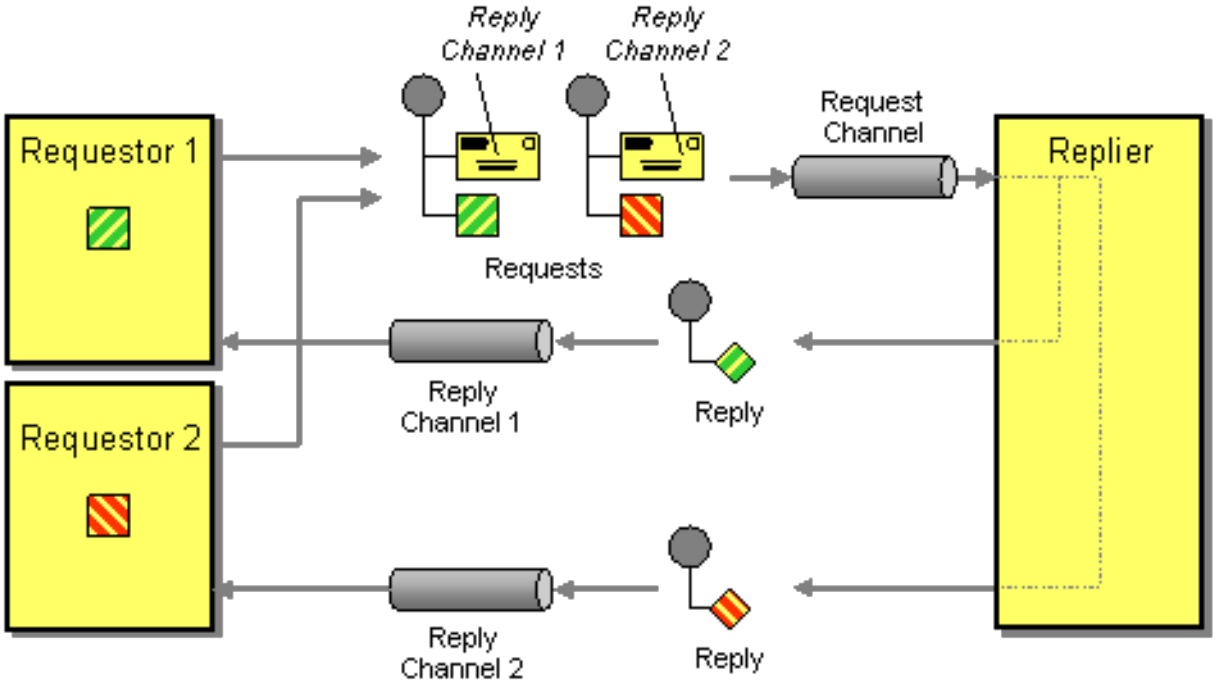
```
</route>
```

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

7.3. 返信先アドレス

返信先アドレス

Apache Camel は、**JMSReplyTo** ヘッダーを使用して、エンタープライズ統合パターン からの 返信先アドレス (Return Address) をサポートします。



たとえば、**InOut** で JMS を使用する場合、コンポーネントはデフォルトで **JMSReplyTo** で指定されたアドレスに返されます。

例

リクエスト側コード

```
getMockEndpoint("mock:bar").expectedBodiesReceived("Bye World");
template.sendBodyAndHeader("direct:start", "World", "JMSReplyTo", "queue:bar");
```

Fluent Builder (流れるようなビルダー) を使用したルート

```
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");
```

Spring XML エクステンション を使用したルート

```
<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>
```

このパターンの完全な例については、JUnit のテストケース を参照してください。

第8章 メッセージのルーティング

概要

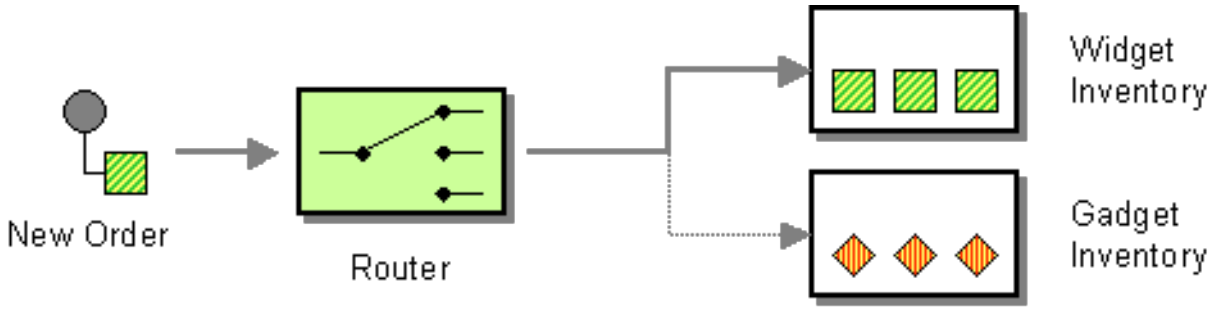
メッセージルーティングパターンでは、メッセージチャネルを互いにリンクするさまざまな方法について説明します。これには、メッセージボディーは変更せずに、メッセージストリームに適用できる、さまざまなアルゴリズムが含まれています。

8.1. CONTENT-BASED ROUTER

概要

図8.1「Content-Based Router パターン」 に示されている **Content-Based Router** を使用すると、メッセージの内容に基づいて適切な宛先にメッセージをルーティングすることができます。

図8.1 Content-Based Router パターン



Java DSL の例

以下の例は、入力エンドポイント **seda:a** からのリクエストを、さまざまな述語式の評価に応じて **seda:b**、**queue:c**、または **seda:d** のいずれかにルーティングする方法を示しています。

```
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 設定の例

以下の例は、XML で同じルートを設定する方法を示しています。

```
<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>
```

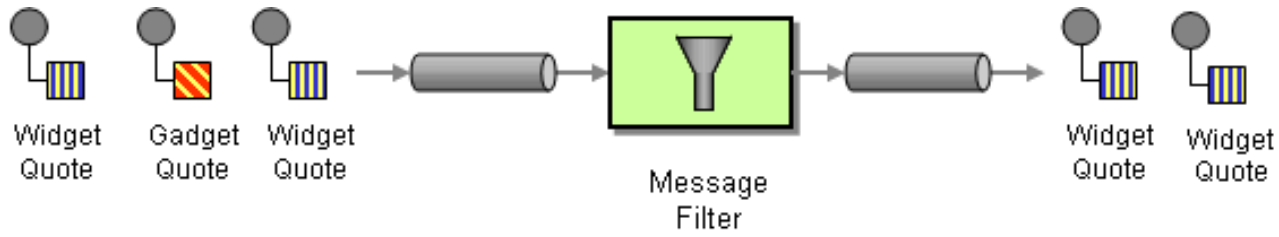
```
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

8.2. MESSAGE FILTER

概要

Massage Filter は、特定の基準に基づいて不要なメッセージを除外するプロセッサーです。Apache Camel では、図8.2「Message Filter パターン」 に示される Message Filter パターンは、Java DSL コマンド **filter()** によって実装されています。この **filter()** コマンドは単一の述語引数を取り、フィルターを制御します。述語が **true** の場合、受信メッセージは続行でき、述語が **false** の場合、受信メッセージはブロックされます。

図8.2 Message Filter パターン



Java DSL の例

以下の例は、エンドポイント **seda:a** からエンドポイント **seda:b** へのルートを作成し、**foo** ヘッダーに **bar**: という値を持つメッセージ以外のすべてのメッセージをブロックする方法を示しています。

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").filter(header("foo").isEqualTo("bar")).to("seda:b");
    }
};
```

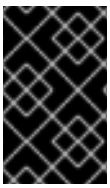
より複雑なフィルター述語を評価するため、XPath、XQuery、SQL (パートII「ルーティング式と述語言語」 を参照) などのサポートされているスクリプト言語のいずれかを呼び出すことができます。以下の例は、**name** 属性が **James** と等しい **person** 要素を含むメッセージ以外のすべてのメッセージをブロックするルートを定義しています。

```
from("direct:start").
        filter().xpath("/person[@name='James']").
        to("mock:result");
```

XML 設定の例

以下の例は、XML で XPath 述語を使用してルートを定義する方法を示しています (パートII「ルーティング式と述語言語」 を参照)。

```
<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>
```



フィルターリングされたエンドポイントは </FILTER> タグの中に定義

フィルタリングするエンドポイント (例: **<to uri="seda:b"/>**) は、クロージングタグ **</filter>** の前に定義する必要があります。そうでない場合は、フィルターは反映されません (2.8 以降で省略するとエラーになります)。

Bean を使用したフィルターリング

フィルターの動作を定義するために Bean を使用した例を以下に示します。

```
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");
    }
}
```

stop() の使用

Camel 2.0 で利用可能

stop は、**すべて** のメッセージをフィルターリングする特別なタイプのフィルターです。stop は、Content-Based Router 内のいずれかの述語で停止する必要がある場合に使用すると便利です。

以下の例では、メッセージボディーに **Bye** という単語を含むメッセージが、ルート内でこれ以上伝播しないようにします。これは、**when()** 述語で **.stop()** を使用しないようにします。

```
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");
```

エクスチェンジがフィルターされたかどうかの確認

Camel 2.5 で利用可能

メッセージフィルター EIP は、フィルターが適用されたかどうかを示すプロパティーを、エクスチェンジに追加します。

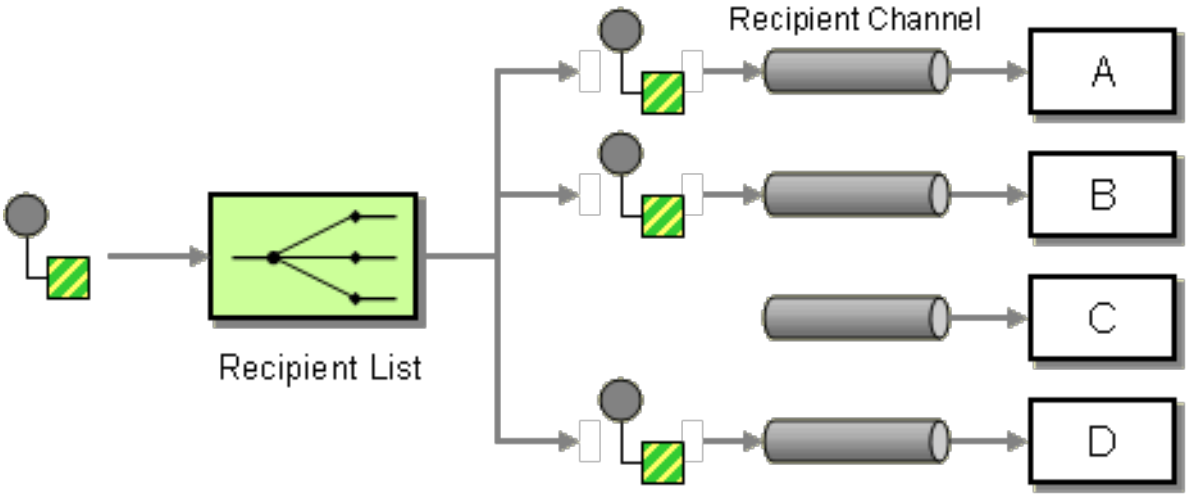
プロパティーにはキー **Exchange.FILTER_MATCHED** があり、これには **CamelFilterMatched** の文字列値が含まれます。この値は、**true** または **false** を示すブール値です。値が **true** の場合、エクスチェンジはフィルターブロック経由でルーティングされました。

8.3. 受信者リスト

概要

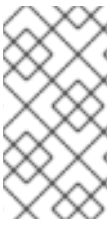
図8.3「Recipient List パターン」 に示されている **Recipient List** は、各受信メッセージを複数の異なる宛先に送信するルーターの一種です。また、Recipient List は通常、実行時に受信者リスト (Recipient List) を演算する必要があります。

図8.3 Recipient List パターン



宛先が固定された Recipient List

最もシンプルな Recipient List は、宛先リストが固定され、事前に認識され、交換パターンは **InOnly** であるものです。この場合、宛先リストを **to()** Java DSL コマンドへハードワイヤーすることができます。



注記

ここで示す例は、宛先が固定された Recepient List のため、**InOnly** 交換パターン (Pipes and Filters パターン と似ています) で **のみ** 動作します。**Out** メッセージを使った交換パターンの Recipient List を作成する場合は、代わりに Multicast パターン を使用してください。

Java DSL の例

以下の例は、コンシューマーエンドポイント **queue:a** から、**InOnly** エクスチェンジを固定された宛先リストにルーティングする方法を示しています。

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

XML 設定の例

以下の例は、XML で同じルートを設定する方法を示しています。

```
<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>
```

実行時に演算された Recipient List

Recipient List パターンを使用するほとんどの場合で、宛先リストを実行時に演算する必要があります。これを行うには、**recipientList()** プロセッサーを使用します。このプロセッサーは、唯一の引数として宛先リストを取ります。Apache Camel はリスト引数に型コンバーターを適用するため、ほとんどの標準的な Java リスト型 (例: コレクション、リスト、配列など) を使用できます。型コンバーターの詳細については 「組み込み型コンバーター」 を参照してください。

受信者は **同じ** エクスチェンジインスタンスのコピーを受け取り、Apache Camel はそれらを順次実行します。

Java DSL の例

以下の例は、**recipientListHeader** という名前のメッセージヘッダーから宛先リストを抽出する方法を示しています。ヘッダーの値は、コンマ区切りのエンドポイント URI のリストになります。

```
from("direct:a").recipientList(header("recipientListHeader").tokenize(","));
```

header の値がリスト型である場合、値を直接 **recipientList()** の引数に使うことができます。以下に例を示します。

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

ただし、この例では、基礎となるコンポーネントがこの特定のヘッダーをどのように解析するかに完全に依存しています。コンポーネントがヘッダーを単純な文字列として解析する場合、この例は動作し **ません**。ヘッダーは、何らかの型の Java リストに変換する必要があります。

XML 設定の例

以下の例では、XML で前述のルートを定義する方法を示しています。ヘッダー値は、コンマ区切りのエンドポイント URI リストです。

```
<camelContext id="buildDynamicRecipientList" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <recipientList delimiter=",">
      <header>recipientListHeader</header>
    </recipientList>
  </route>
</camelContext>
```

複数の受信者に並列で送信

Camel 2.2 で利用可能

Recipient List パターン は、**parallelProcessing** をサポートしています。これは、Splitter パターン の機能に似ています。並列処理機能を使用して、複数の受信者に並列でエクスチェンジを送信します。以下に例を示します:

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

Spring XML では、並列処理機能は **recipientList** タグの属性として実装されています。以下に例を示します。

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

例外時に停止

Camel 2.2 で利用可能

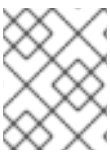
Recipient List は、**stopOnException** 機能をサポートします。これを使用すると、受信者が失敗した場合にそれ以降の受信者への送信を停止することができます。

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

Spring XML では、Recipient List タグの属性です。

Spring XML では、例外時の停止機能は **recipientList** タグの属性として実装されます。以下に例を示します。

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



注記

同じルートで **parallelProcessing** と **stopOnException** を組み合わせることができます。

無効なエンドポイントの無視

Camel 2.3 の時点で利用可能

Recipient List パターン は **ignoreInvalidEndpoints** オプションをサポートします。これにより、Recipient List が無効なエンドポイントをスキップできます (Routing Slips パターン も、このオプションをサポートしています)。以下に例を示します。

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

Spring XML では、以下のように **recipientList** タグに **ignoreInvalidEndpoints** 属性を設定することで、このオプションを有効にすることができます。

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

myHeader に **direct:foo,xxx:bar** の 2 つのエンドポイントが含まれるケースについて考えてみましょう。最初のエンドポイントは有効であり、動作します。2 つ目は無効であるため、無視されます。無効なエンドポイントに遭遇するたびに、Apache Camel ログが **INFO** レベルで記録されます。

カスタム AggregationStrategy の使用

Camel 2.2 で利用可能

Recipient List パターン でカスタム **AggregationStrategy** を使用できます。これは、リスト内の受信者からのリプライを集計する場合に便利です。Apache Camel はデフォルトで **UseLatestAggregationStrategy** 集約ストラテジーを使用して、最後に受信したリプライのみを保持します。より高度な集約ストラテジーについては、**AggregationStrategy** インターフェイスを独自に実装し定義できます。詳細については 「Aggregator」 を参照してください。たとえば、集約ストラテジー **MyOwnAggregationStrategy** をリプライメッセージに適用するには、以下のように Java DSL ルートを定義します。

```
from("direct:a")
    .recipientList(header("myHeader")).aggregationStrategy(new MyOwnAggregationStrategy())
    .to("direct:b");
```

Spring XML では、以下のようにカスタムの集約ストラテジーを **recipientList** タグの属性として指定できます。

```
<route>
  <from uri="direct:a"/>
  <recipientList strategyRef="myStrategy">
    <header>myHeader</header>
  </recipientList>
  <to uri="direct:b"/>
</route>

<bean id="myStrategy" class="com.mycompany.MyOwnAggregationStrategy"/>
```

カスタムスレッドプールの使用

Camel 2.2 で利用可能

これは、**parallelProcessing** を使用する場合にのみ必要です。デフォルトでは、Camel は 10 個のスレッドを持つスレッドプールを使用します。今後スレッドプールの管理と設定内容をメンテナーンスする際に、変更される可能性がありますのでご注意ください (予定どおりであれば Camel 2.2) 。

カスタム集約ストラテジーを使用するのと同じように設定します。

メソッド呼び出しの Recipient List としての使用

受信者を生成するために、Bean を使用できます。以下に例を示します。

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

MessageRouter Bean は以下のように定義されます。

```
public class MessageRouter {

    public String routeTo() {
        String queueName = "activemq:queue:test2";
        return queueName;
    }
}
```

Recipient List としての Bean

Bean を Recipient List として動作させるには、**@RecipientList** アノテーションを Recipient List を返すメソッドに付与します。以下に例を示します。

```
public class MessageRouter {

    @RecipientList
    public String routeTo() {
        String queueList = "activemq:queue:test1,activemq:queue:test2";
        return queueList;
    }
}
```

この場合、ルートに **recipientList** DSL コマンドを含め **ない** でください。以下のようにルートを定義します。

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

タイムアウトの使用

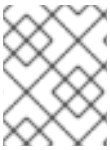
Camel 2.5 で利用可能

parallelProcessing を使用する場合は、合計 **timeout** 値をミリ秒単位で設定できます。Camel はタイムアウトに達するまでメッセージを並行して処理します。これにより、1 つのメッセージが遅い場合でも処理を継続できます。

以下の例では、**recipientlist** ヘッダーの値が **direct:a,direct:b,direct:c** であるため、メッセージは 3 人の受信者に送信されます。250 ミリ秒のタイムアウトがあるので、最後の 2 つのメッセージだけが、タイムフレーム内で完了することができます。そのため、集約すると **BC** という文字列の結果が得られます。

```
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"));
```



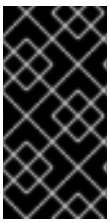
注記

この **timeout** 機能は **splitter** のほか、**multicast** および **recipientList** の両方でもサポートされています。

デフォルトでは、タイムアウトが発生した場合、**AggregationStrategy** は呼び出されません。ただし、特殊なバージョンを実装することができます。

```
// 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);
```

これにより、必要であれば、**AggregationStrategy** でタイムアウトに対応することができます。



TIMEOUT は合計値

タイムアウトは合計です。つまり、X 時間後に Camel は期限内に完了したメッセージのみを集約します。残りの分はキャンセルされます。また Camel は、タイムアウトの原因となった最初のインデックスに対して、**TimeoutAwareAggregationStrategy** の **timeout** メソッドを 1 度だけ呼び出します。

送信メッセージへのカスタム処理の適用

recipientList がメッセージを Recipient List のいずれかに送信する前に、元のメッセージのシャローコピーであるメッセージレプリカを作成します。シャローコピーでは、元のメッセージのヘッダーおよび

ペイロードは参照によってのみコピーされます。各新規コピーには、それらの要素独自のインスタンスが含まれていません。その結果、メッセージのシャローコピーがリンクされ、異なるエンドポイントにルーティングする際にカスタム処理を適用することはできません。

レプリカがエンドポイントに送信される前に、各メッセージレプリカに対して何らかのカスタム処理を行う場合は、**recipientList** 句で **onPrepare** DSL コマンドを呼び出すことができます。この **onPrepare** コマンドは、メッセージがシャローコピーされた **直後**、かつメッセージがエンドポイントにディスパッチされる **直前** に、カスタムプロセッサーを挿入します。たとえば、以下のルートでは、**CustomProc** プロセッサーが **各 recipient エンドポイント** 用のメッセージレプリカで呼び出されます。

```
from("direct:start")
  .recipientList().onPrepare(new CustomProc());
```

onPrepare DSL コマンドの一般的なユースケースとして、メッセージの一部またはすべての要素のディープコピーを実行します。これにより、各メッセージのレプリカは他のレプリカとは独立して変更することができます。たとえば、以下の **CustomProc** プロセッサークラスは、メッセージボディーのディープコピーを実行します。メッセージボディーの型は **BodyType** であると想定され、ディープコピーはメソッド **BodyType.deepCopy()** によって実行されます。

```
// 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.
    }
}
```

オプション

recipientList DSL コマンドは以下のオプションをサポートします。

名前	デフォルト値	説明
delimiter	**,**	式が複数のエンドポイントを返した場合に使用される区切り文字。

strategyRef		AggregationStrategy を参照し、各受信者からのリプライを集約して「受信者リスト」からの唯一となる送信メッセージを生成します。デフォルトでは、Camel は最後のリプライを送信メッセージとして使用します。
strategyMethodName		POJO を **AggregationStrategy** として使用している場合に、使用するメソッド名を明示的に指定するために使用できます。
strategyMethodAllowNull	**false**	POJO を **AggregationStrategy** として使用している場合に、このオプションを使用することができます。**false** に設定すると、エンリッチするデータがない場合に **aggregate** メソッドは使用されません。**true** に設定すると、エンリッチするデータがない場合には、**oldExchange** に **null** 値が使用されます。
parallelProcessing	**false**	Camel 2.2: 有効にすると、受信者へのメッセージ送信が並列処理されます。呼び出し元スレッドは、すべてのメッセージが完全に処理されるまで待機してから続行することに注意してください。受信者への送信と、受信者からのリプライ処理のみが並列で処理されます。
parallelAggregate	**false**	有効にすると、**AggregationStrategy** の **aggregate** メソッドを同時に呼び出すことができます。これには、**AggregationStrategy** の実装がスレッドセーフである必要があることに注意してください。デフォルトでは、このオプションは **false** となっており、Camel が自動的に **aggregate** メソッドへの呼び出しを同期することを意味します。ただし、場合によっては、**AggregationStrategy** をスレッドセーフとして実装し、このオプションを **true** に設定することで、パフォーマンスを向上させることができます。

executorServiceRef		Camel 2.2: 並列処理に使用するカスタムスレッドプールを参照します。このオプションを設定すると、並列処理は自動的に適用されるため、 並列処理用オプションも有効にする必要はありません。
stopOnException	**false**	Camel 2.2: 例外発生時、すぐに継続処理を停止するかどうか。無効にすると、いずれかの失敗の有無に関わらず、Camel はメッセージをすべての受信者に送信します。AggregationStragegy クラス内で、例外処理を完全に制御することができます。
ignoreInvalidEndpoints	**false**	Camel 2.3: エンドポイント URI が解決できない場合、無視されます。false の場合は、Camel はエンドポイント URI が有効ではないことを示す例外を出力します。
streaming	**false**	Camel 2.5: 有効な場合、Camel は返信を順不同で処理します (例: 戻って来た順)。無効な場合、Camel は指定された式と同じ順序で返信を処理します。
timeout		camel 2.5: 合計タイムアウト値をミリ秒単位で設定します。「受信者リスト」 が指定された時間枠内のすべての応答を送信および処理できない場合、タイムアウトが発生し、「受信者リスト」 から抜け出し続行します。AggregationStrategy を提供した場合、**timeout** メソッドが抜け出す前に呼び出されるため、注意してください。
onPrepareRef		camel 2.8: カスタムプロセッサーを参照して、各受信者が受け取るエクスチェンジのコピーを準備します。これにより、必要に応じてメッセージのペイロードをディープクローンするなど、カスタムロジックを実行できます。

shareUnitOfWork	**false**	**Camel 2.8:** Unit of Work を共有すべきかどうか。詳細は、「Splitter」 で同じオプションを参照してください。
cacheSize	**0**	**Camel 2.13.1/2.12.4:** Routing Slip で再使用されるプロデューサーをキャッシュする **ProducerCache** のキャッシュサイズを設定できます。デフォルトのキャッシュサイズ 0 を使用します。値を -1 に設定すると、キャッシュをすべて無効にすることができます。

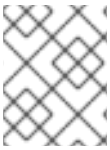
Recipient List で交換パターンを使用

デフォルトでは、Recipient List は既存の交換パターンを使用します。稀ではありますが、別の交換パターンを使用して受信者にメッセージを送信するケースがある可能性があります。

たとえば、**InOnly** ルートとして開始するルートがあるとします。Recipient List で **InOut** 交換パターンを使用する場合、受信者用エンドポイントで直接交換パターンを設定する必要があります。

以下の例は、新規ファイルが InOnly として開始され、Recipient List へルーティングされるルートを示しています。ActiveMQ(JMS) エンドポイントで InOut を使用する場合、exchangePattern=InOut オプションを指定する必要があります。ただし、JMS リクエストやリプライをルーティングし続けるため、レスポンスは outbox ディレクトリー内にファイルとして保存されます。

```
from("file:inbox")
  // the exchange pattern is InOnly initially when using a file route
  .recipientList().constant("activemq:queue:inbox?exchangePattern=InOut")
  .to("file:outbox");
```



注記

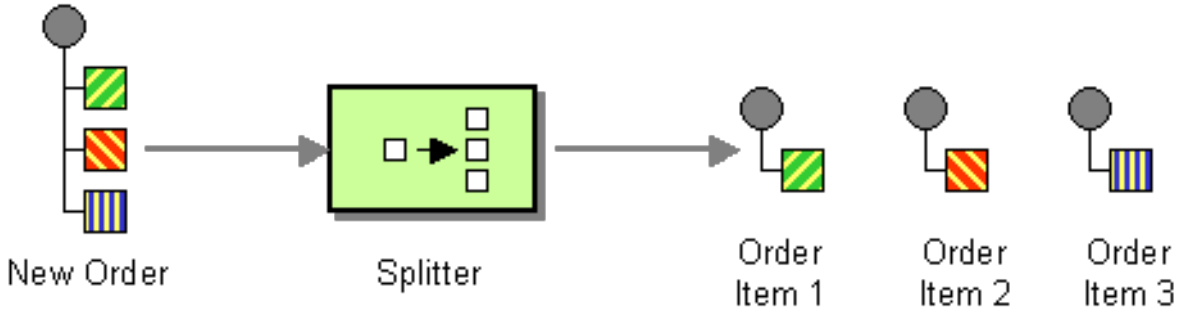
InOut 交換パターンは、タイムアウト時にレスポンスを受け取る必要があります。ただし、レスポンスを受信できない場合は失敗します。

8.4. SPLITTER

概要

Splitter は、受信メッセージを一連の送信メッセージに分割するルーターの種類です。それぞれの送信メッセージには、元のメッセージの一部が含まれています。Apache Camel では、図8.4「Splitter パターン」 に示される Splitter パターンは、**split()** Java DSL コマンドによって実装されます。

図8.4 Splitter パターン



Apache Camel Splitter は、以下の 2 つのパターンをサポートします。

- **Simple Splitter:** Splitter パターンを独自に実装します。
- **Splitter/Aggregator** - Splitter パターンと Aggrigator パターンを組み合わせることで、メッセージの断片が処理された後に再結合されます。

Splitter はオリジナルのメッセージを分割する前に、オリジナルのメッセージのシャローコピーを作成します。シャローコピーでは、元のメッセージのヘッダーおよびペイロードは参照としてのみコピーされます。Splitter 自体は、結果として得られたメッセージの一部を異るエンドポイントにルーティングすることはありませんが、分割されたメッセージの一部は、セカンダリールーティングの影響を受ける可能性があります。

メッセージ部分はシャローコピーであるため、元のメッセージにリンクされたままになります。そのため、それらを単独で修正することはできません。複数のエンドポイントにルーティングする前に、メッセージの異なるコピーへカスタムロジックを適用する場合、**splitter** 句の **onPrepareRef** DSL オプションを使用して、オリジナルのメッセージのディープコピーを作成する必要があります。オプションの使用方法については、「オプション」 を参照してください。

Java DSL の例

以下の例は、**seda:a** から **seda:b** へのルートを定義し、受信メッセージの各行を個別の送信メッセージへ変換することでメッセージを分割しています。

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a")
          .split(bodyAs(String.class).tokenize("\n"))
          .to("seda:b");
    }
};
```

Splitter は式言語を使用できるため、XPath、XQuery、SQL などのサポートされているスクリプト言語を使用して、メッセージを分割することができます (パートII「ルーティング式と述語言語」 を参照)。以下の例は、受信メッセージから **bar** 要素を抽出し、それらを別々の送信メッセージに挿入しています。

```
from("activemq:my.queue")
  .split(xpath("//foo/bar"))
  .to("file://some/directory")
```

XML 設定の例

以下の例は、XPath スクリプト言語を使用して、XML で Splitter ルートを定義する方法を示しています。

```
<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>
```

XML DSL の tokenize 式を使用して、トークンを使い、ボディーまたはヘッダーを分割できます。tokenize 式は、**tokenize** 要素で定義します。以下の例では、メッセージボディーは **\n** 区切り文字を使用してトークン化されています。正規表現パターンを使用するには、**tokenize** 要素に **regex=true** を設定します。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split>
            <tokenize token="\n"/>
            <to uri="mock:result"/>
        </split>
    </route>
    </camelContext>
```

行のグループに分割

大きなファイルを 1000 行のブロックに分割するには、以下のように Splitter ルートを定義します。

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

tokenize への第 2 引数は、1 つのチャンクにグループ化されるべき行数を指定します。**streaming()** 句は、ファイル全体を同時に読み取りしないよう Splitter に指示します (ファイルが大きい場合のパフォーマンスがはるかに改善されます)。

以下のように XML DSL で同じルートを定義できます。

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="\n" group="1000"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

group オプションを使用する際の出力は常に、**java.lang.String** 型になります。

最初の項目のスキップ

メッセージの最初の項目をスキップするには、 **skipFirst** オプションを使用します。

Java DSL では、**tokenize** パラメーターの 3 番目のオプションに **true** を指定します。

```
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");
```

以下のように XML DSL で同じルートを定義できます。

```
<route>
  <from uri="file:inbox"/>
    <split streaming="true">
    <tokenize token="\n" group="1000" skipFirst="true" />
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

Splitter のリプライ

Splitter に入るエクスチェンジが **InOut** メッセージ交換パターンである場合 (つまりリプライが想定される場合)、Splitter は元の入力メッセージのコピーを **Out** メッセージスロットのリプライメッセージとして返します。独自の 集約ストラテジー を実装して、このデフォルト動作をオーバーライドすることができます。

並列実行

生成されたメッセージを並行に実行する場合、並列処理オプションを有効にして、生成されたメッセージを処理するためのスレッドプールをインスタンス化します。以下に例を示します。

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
from("activemq:my.queue").split(xPathBuilder).parallelProcessing().to("activemq:my.parts");
```

並行 Splitter で使用される基盤となる **ThreadPoolExecutor** をカスタマイズすることができます。たとえば、以下のように Java DSL でカスタムエクゼキューターを指定することができます。

```
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");
```

以下のように、XML DSL でカスタムエクゼキューターを指定できます。

```
<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>
```

Bean を使用した分割処理の実行

Splitter は **任意** の式を使用して分割処理ができるので、**method()** 式を呼び出すことで、Bean を使用して分割処理を実行できます。Bean は、**java.util.Collection**、**java.util.Iterator**、または配列などの反復可能な値を返す必要があります。

以下のルートは、**mySplitterBean** Bean インスタンスのメソッドを呼び出す **method()** 式を定義しています。

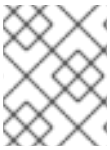
```
from("direct:body")
        // here we use a POJO bean mySplitterBean to do the split of the payload
        .split()
        .method("mySplitterBean", "splitBody")
        .to("mock:result");
from("direct:message")
        // here we use a POJO bean mySplitterBean to do the split of the message
        // with a certain header value
        .split()
        .method("mySplitterBean", "splitMessage")
        .to("mock:result");
```

mySplitterBean は **MySplitterBean** クラスのインスタンスで、以下のように定義されます。

```
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;
    }
}
```

Splitter EIP で **BeanIOSplitter** オブジェクトを使用して、ストリームモードによりコンテンツ全体をメモリーに読み込まないようにしながら、大きなペイロードを分割することができます。以下の例は、クラスパスから読み込まれるマッピングファイルを使用して、**BeanIOSplitter** オブジェクトを設定する方法を示しています。



注記

BeanIOSplitter クラスは Camel 2.18 で新たに追加されました。Camel 2.17 では利用できません。

```
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");
```

以下の例は、エラーハンドラーを追加しています:

```
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");
```

エクスチェンジプロパティー

以下のプロパティーは、分割されたエクスチェンジごとに設定されます。

ヘッダー	型	description
CamelSplitIndex	**int**	Apache Camel 2.0: 各エクスチェンジが分割されるたびに増加するスプリットカウンター。カウンターは 0 から始まります。
CamelSplitSize	**int**	Apache Camel 2.0: 分割されたエクスチェンジの合計数。このヘッダーは、ストリームベースの分割には適用されません。
CamelSplitComplete	**boolean**	Apache Camel 2.4: このエクスチェンジが最後であるかどうか。

Splitter/Aggregator パターン

個々のコンポーネントの処理が完了した後、分割されたメッセージを単一のエクスチェンジに集約する一般的なパターンです。このパターンをサポートするために、 **split()** DSL コマンドでは、第 2 引数として **AggregationStrategy** オブジェクトを指定することができます。

Java DSL の例

以下の例は、カスタム集約ストラテジーを使用して、すべての分割されたメッセージが処理された後に、メッセージを再結合する方法を示しています。

```
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 の実装

上記のルートで使用されるカスタム集約ストラテジー **MyOrderStrategy** は、以下のように実装されています。

```
/**
 * 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;
    }
}
```

ストリームベースの処理

並列処理を有効にすると、後ろの分割されたメッセージが、前の分割されたメッセージよりも先に、集約の準備が整うことが理論的に起こりえます。つまり、分割された個々のメッセージは、異る順序で Aggregator へ到着する可能性があります。デフォルトでは、Splitter 実装は分割されたメッセージを Aggregator へ渡す前に元の順序に再配置するため、これが発生しません。

分割されたメッセージの処理が完了次第集約する場合は、以下のようにストリーミングオプションを有効にできます (メッセージの順序が乱れる可能性があります)。

```
from("direct:streaming")
  .split(body().tokenize(","), new MyOrderStrategy())
    .parallelProcessing()
    .streaming()
    .to("activemq:my.parts")
  .end()
  .to("activemq:all.parts");
```

以下に示すように、ストリーミングで使用するカスタムイテレーターを指定することもできます。

```
// Java
import static org.apache.camel.builder.ExpressionBuilder.beanExpression;
...
from("direct:streaming")
    .split(beanExpression(new MyCustomIteratorFactory(),  "iterator"))
    .streaming().to("activemq:my.parts")
```

ストリーミングおよび XPATH

ストリーミングモードを XPath と併用することはできません。XPath は、メモリー内に完全な DOM XML ドキュメントを必要とします。

XML を使用したストリームベースの処理

受信メッセージが非常に大きな XML ファイルである場合、ストリーミングモードで **tokenizeXML** サブコマンドを使用して最も効率的にメッセージを処理することができます。

たとえば、**order** 要素のシーケンスを含む大きな XML ファイルの場合、以下のようなルートを使用してファイルを **order** 要素に分割することができます。

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

以下のようなルートを定義することで、XML でも同じことができます。

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

トークン要素のエンクロージング (ancestor) 要素で定義される namespace へのアクセスが必要になる場合がよくあります。namespace の定義を ancestor 要素のいずれかから token 要素にコピーするには、namespace 定義を継承する要素を指定する必要があります。

Java DSL で、ancestor 要素を **tokenizeXML** の第 2 引数として指定します。たとえば、enclosing **orders** 要素から namespace 定義を継承するには、以下のようにします。

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

XML DSL では、**inheritNamespaceTagName** 属性を使用して ancestor 要素を指定します。以下に例を示します。

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
```

```
    <tokenize token="order"
              xml="true"
              inheritNamespaceTagName="orders"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

オプション

split DSL コマンドは以下のオプションをサポートします。

名前	デフォルト値	説明
strategyRef		AggregationStrategy を参照し、分割されたメッセージのリプライを集約して、「Splitter」 からの単一となる送信メッセージを生成します。デフォルトで使用されているものについては、**What does the splitter return**というタイトルの項を参照してください。
strategyMethodName		POJO を **AggregationStrategy** として使用している場合に、使用するメソッド名を明示的に指定するために使用できます。
strategyMethodAllowNull	**false**	POJO を **AggregationStrategy** として使用している場合に、このオプションを使用することができます。**false** に設定すると、エンリッチするデータがない場合に **aggregate** メソッドは使用されません。**true** に設定すると、エンリッチするデータがない場合には、**oldExchange** に **null** 値が使用されます。
parallelProcessing	**false**	有効にすると、分割されたメッセージの処理が並列で行われます。呼び出し元スレッドは、すべての分割されたメッセージが完全に処理されるまで待機してから続行することに注意してください。

parallelAggregate	**false**	有効にすると、**AggregationStrategy** の **aggregate** メソッドを同時に呼び出すことができます。これには、**AggregationStrategy** の実装がスレッドセーフである必要があることに注意してください。デフォルトでは、このオプションは **false** となっており、Camel が自動的に **aggregate** メソッドへの呼び出しを同期することを意味します。ただし、場合によっては、**AggregationStrategy** をスレッドセーフとして実装し、このオプションを **true** に設定することで、パフォーマンスを向上させることができます。
executorServiceRef		並列処理に使用するカスタムスレッドプールを参照します。このオプションを設定すると、並列処理は自動的に適用されるため、 並列処理用オプションも有効にする必要はありません。
stopOnException	**false**	Camel 2.2: 例外発生時、すぐに継続処理を停止するかどうか。無効にすると、Camel は分割されたメッセージの 1 つが失敗しても、分割処理を継続します。AggregationStragegy クラス内で、例外処理を完全に制御することができます。
streaming	**false**	有効にすると、Camel はストリーミング方式で input メッセージを分割します。これにより、メモリーのオーバーヘッドが軽減されます。たとえば、大きなメッセージを分割する場合には、streaming オプションを有効にすることが推奨されます。streaming オプションが有効になっていると、分割されたメッセージのリプライは、順不同で集約されます (例: 分割後の処理が終了したメッセージ順)。無効な場合、Camel は分割された順序と同じ順序で、分割されたメッセージを集約します。

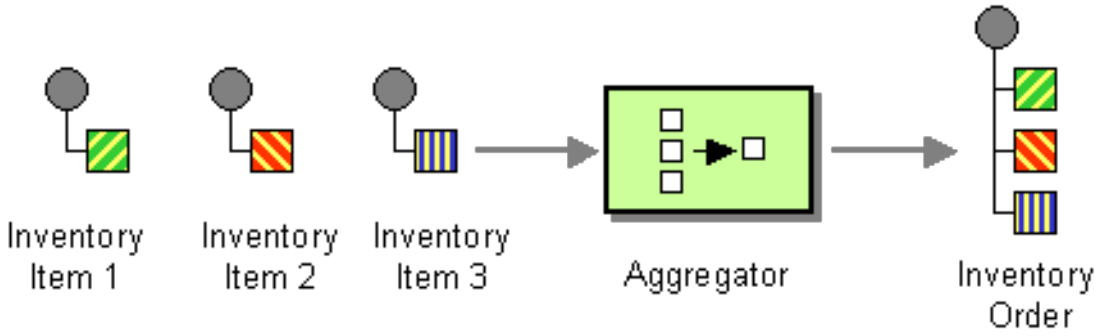
timeout		camel 2.5: 合計タイムアウト値をミリ秒単位で設定します。「受信者リスト」 が指定された時間内に分割してすべてのリプライを処理できない場合、タイムアウト発生して 「Splitter」 から抜け出し続行します。AggregationStrategy を提供した場合、**timeout** メソッドが抜け出す前に呼び出されるため、注意してください。
onPrepareRef		camel 2.8: カスタムプロセッサーを参照することで、エクスチェンジの分割されたメッセージが処理される前に準備をすることができます。これにより、必要に応じてメッセージのペイロードをディープクローンするなど、カスタムロジックを実行できます。
shareUnitOfWork	**false**	Camel 2.8: Unit of Work を共有すべきかどうか。詳細についていは以下をご覧ください

8.5. AGGREGATOR

概要

図8.5「Aggreagator パターン」 に示されている **Aggregator** パターンにより、関連するメッセージのバッチを単一のメッセージにまとめることができます。

図8.5 Aggreagator パターン



Aggregator の動作を制御するため、Apache Camel では以下のように**Enterprise Integration Patterns**で説明されているプロパティーを指定できます。

- **相関式**: 集約するメッセージを決定します。相関式は各受信メッセージに対して評価され、**相関キー** を生成します。同じ相関キーを持つ受信メッセージは、同じバッチにグループ化されます。たとえば、**すべての** 受信メッセージを 1 つのメッセージに集約する場合は、定数式を使用することができあす。

- **完了条件**: メッセージのバッチが完了したかどうかを決定します。これは単純なサイズ制限として指定することもでき、より一般的には、バッチ完了を示すフラグを述語条件として指定することもできます。

- **集約アルゴリズム**: 特定の相関キーを持つメッセージエクスチェンジを単一のメッセージエクスチェンジに統合します。

たとえば、毎秒 30,000 通のメッセージを受信する株式市場のデータシステムについて考えてみましょう。GUI ツールがこのような大規模の更新レートに対応できない場合は、メッセージフローをスロットルダウンした方がよい場合があります。単純に最新の気配値を選択して古い値を破棄することで、入力される株の気配値を集約することができます (一部の履歴をキャプチャーする場合は、delta 処理アルゴリズムを適用することができます)。

注記

Aggregator はより多くの情報を含む **ManagedAggregateProcessorMBean** を使い、JMX へ登録されるようになりました。これにより、集約コントローラーを使い制御できるようになります。

Aggregator の仕組み

図8.6「Aggregator の実装」 は、A、B、C、D などの相関キーを持つエクスチェンジのストリームを使用して、Aggragator がどのように動作するか概要を示しています。

図8.6 Aggregator の実装

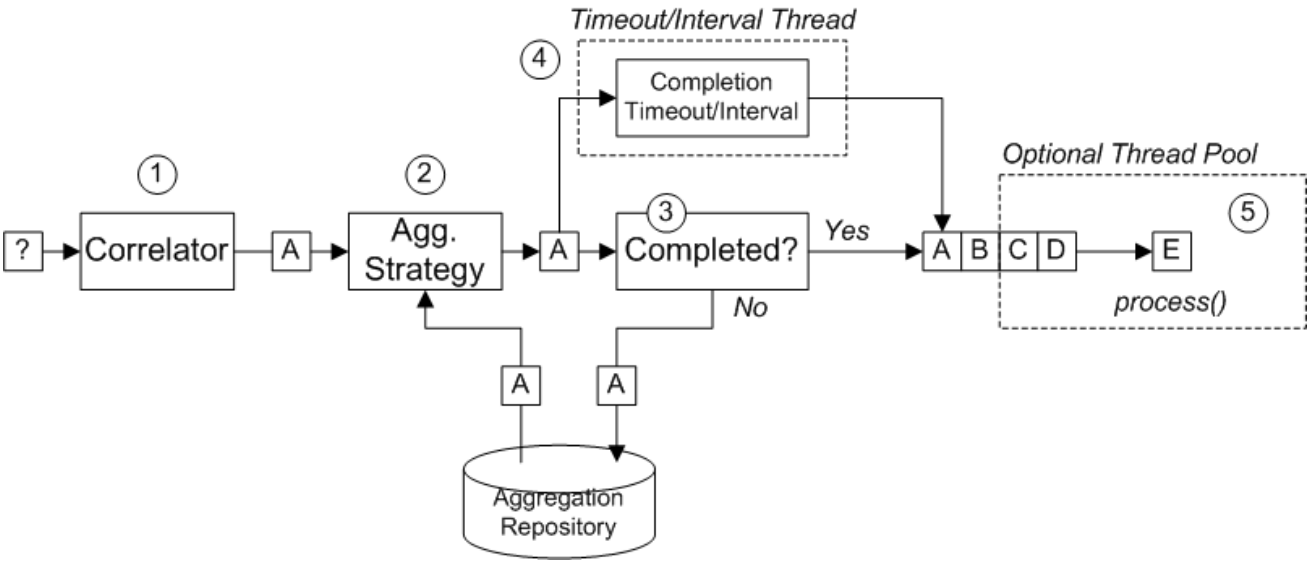


図8.6「Aggregator の実装」 に示されているエクスチェンジの受信ストリームは、以下のように処理されます。

1. **Correlator** は、相関キーに基づいてエクスチェンジをソートします。各受信メッセージごとに相関式が評価され、相関キーを生成します。たとえば、図8.6「Aggregator の実装」 で示されているエクスチェンジでは、相関キーは A と評価されます。

2. **集約ストラテジー** は、同じ相関キーを持つエクスチェンジをマージします。新しいエクスチェンジ A が到達すると、Aggregator は集約リポジトリーで対応する **集約エクスチェンジ** (A') を検索し、新しいエクスチェンジと結合します。
   特定の集約サイクルが完了するまで、受信したエクスチェンジは、対応する集約エクスチェンジへ継続的に集約されます。集約サイクルは、完了メカニズムのいずれかによって終了されるまで継続されます。

注記

Camel 2.16 から、新しい XSLT 集約ストラテジー により、2 つのメッセージを XSLT ファイルでマージできるようになりました。ツールボックスから **AggregationStrategies.xslt()** ファイルにアクセスできます。

3. Aggregator に完了述語が指定された場合、集約エクスチェンジをテストし、ルートの次のプロセッサーに送信する準備ができているかどうかを判断します。以下のように処理を続けます。

   - 完了したら、集約エクスチェンジはルートの後半部分で処理されます。2 つの代替えモデルがあります。1 つは、**同期** (デフォルト) で、呼び出しスレッドがブロックされます。2 つ目は **非同期** (並列処理が有効になっている場合) で、集約エクスチェンジはエクゼキュータースレッドプールに送信されます (図8.6「Aggregator の実装」 を参照)。

   - 完了していない場合、集約エクスチェンジは集約リポジトリーに戻されます。

4. 同期的な完了テストの他、**completionTimeout** オプションまたは **completionInterval** オプションの **いずれか** を有効にすることで、非同期的な完了テストを有効にすることができます。これらの完了テストは別のスレッドで実行され、完了テストを満すたびに、対応するエクスチェンジが完了としてマークされ、ルートの後半部分によって処理されます (並列処理が有効かどうかによって、同期または非同期的に処理されます)。

5. 並列処理が有効な場合、スレッドプールがルートの後半部分でエクスチェンジを処理します。デフォルトでは、このスレッドプールには 10 個のスレッドが含まれますが、プールをカスタマイズすることもできます (「スレッドオプション」)。

Java DSL の例

以下の例は、**UseLatestAggregationStrategy** 集計ストラテジーを使用して、同じ **StockSymbol** ヘッダー値を持つエクスチェンジを集約しています。指定された **StockSymbol** 値について、その相関キーを持つエクスチェンジを最後に受信してから 3 秒以上経過すると、集約されたエクスチェンジは完了とみなされ、**mock** エンドポイントに送信されます。

```
from("direct:start")
    .aggregate(header("id"), new UseLatestAggregationStrategy())
        .completionTimeout(3000)
    .to("mock:aggregated");
```

XML DSL の例

以下の例は、XML で同じルートを設定する方法を示しています。

```
<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"/>
```

相関式の指定

Java DSL では、相関式は常に第 1 引数として **aggregate()** DSL コマンドに渡されます。ここでは、Simple 式言語の使用に制限はありません。XPath、XQuery、SQL などの式言語やスクリプト言語を使用して、相関式を指定することができます。

例えば、XPath 式を使用してエクスチェンジを相関させるには、以下の Java DSL ルートを使用します。

```
from("direct:start")
   .aggregate(xpath("/stockQuote/@symbol"), new UseLatestAggregationStrategy())
      .completionTimeout(3000)
   .to("mock:aggregated");
```

特定の受信エクスチェンジで相関式を評価することができない場合、Aggregator はデフォルトで **CamelExchangeException** を出力します。**ignoreInvalidCorrelationKeys** オプションを設定することで、この例外を抑制できます。たとえば、Java DSL の場合は以下のようになります。

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

XML DSL では、以下のように **ignoreInvalidCorrelationKeys** オプションを属性として設定できます。

```
<aggregate strategyRef="aggregatorStrategy"
          ignoreInvalidCorrelationKeys="true"
          ...>
   ...
</aggregate>
```

集約ストラテジーの指定

Java DSL では、集約ストラテジーを第 2 引数として **aggregate()** DSL コマンドに渡すか、**aggregationStrategy()** 句を使用して指定できます。たとえば、以下のように **aggregationStrategy()** 句を使用できます。

```
from("direct:start")
   .aggregate(header("id"))
      .aggregationStrategy(new UseLatestAggregationStrategy())
      .completionTimeout(3000)
   .to("mock:aggregated");
```

Apache Camel は、以下の基本的な集約ストラテジーを提供しています (各クラスは **org.apache.camel.processor.aggregate** Java パッケージ配下に属します)。

UseLatestAggregationStrategy

指定された相関キーの最後のエクスチェンジを返し、このキーとの以前のエクスチェンジをすべて破棄します。たとえば、このストラテジーは、特定の株式シンボルの最新価格のみを確認する場合、証券取引所からのフィードをスロットリングするのに役立ちます。

UseOriginalAggregationStrategy

指定された相関キーの最初のエクスチェンジを返し、このキーを持つそれ以降のすべてのエクスチェンジを破棄します。このストラテジーを使用する前に、**UseOriginalAggregationStrategy.setOriginal()** を呼び出して、最初のエクスチェンジを設定する必要があります。

GroupedExchangeAggregationStrategy

指定された相関キーの **all** のエクスチェンジをリストに連結し、**Exchange.GROUPED_EXCHANGE** エクスチェンジプロパティーに保存します。「グループ化されたエクスチェンジ」 を参照してください。

カスタム集約ストラテジーの実装

別の集計ストラテジーを適用する場合は、以下の集計ストラテジーのベースとなるインターフェースのいずれかを実装することができます。

org.apache.camel.processor.aggregate.AggregationStrategy

基本的な Aggregation Strategy インターフェイス。

org.apache.camel.processor.aggregate.TimeoutAwareAggregationStrategy

集約サイクルのタイムアウト時にお使いの実装で通知を受け取る場合は、このインターフェイスを実装します。**timeout** 通知メソッドには、以下の署名があります。

```
void timeout(Exchange oldExchange, int index, int total, long timeout)
```

org.apache.camel.processor.aggregate.CompletionAwareAggregationStrategy

集約サイクルが正常に完了したときにお使いの実装で通知を受け取る場合は、このインターフェイスを実装します。通知メソッドには、以下の署名があります。

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

たとえば、以下のコードは、**StringAggregationStrategy** および **ArrayListAggregationStrategy** の 2 つの異なるカスタム集計ストラテジーを示しています。

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

注記

Apache Camel 2.0 以降、**AggregationStrategy.aggregate()** コールバックメソッドも最初のエクスチェンジに対して呼び出されます。**aggregate** メソッドの最初の呼び出しでは、**oldExchange** パラメーターは **null** であり、**newExchange** パラメーターに最初の受信エクスチェンジが含まれます。

カスタムストラテジークラス **ArrayListAggregationStrategy** を使用してメッセージを集約するには、以下のようなルートを定義します。

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

以下のように、XML でカスタム集約ストラテジーを使用してルートを設定することもできます。

```
<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"/>
```

カスタム集約ストラテジーのライフサイクルの管理

カスタム集約ストラテジーを実装し、そのライフサイクルを、それを管理しているエンタープライズインテグレーションパターンのライフサイクルに合わせることができます。これは、集約ストラテジーが正常にシャットダウンできることを保証するのに役立ちます。

ライフサイクルをサポートする集約ストラテジーを実装するには、**org.apache.camel.Service** インターフェイスを実装し (**AggregationStrategy** インターフェイスに加えて)、**start()** および **stop()** ライフサイクルメソッドの実装を提供する必要があります。たとえば、以下のコード例は、ライフサイクル

をサポートする集約ストラテジーの概要を示しています。

```
// 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
      ...
   }
}
```

エクスチェンジプロパティー

以下のプロパティーは、集約された各エクスチェンジ毎に設定されます。

ヘッダー	型	集約されたエクスチェンジ プロパティーに関する説明
Exchange.AGGREGATED_SIZE	**int**	このエクスチェンジに集約されたエクスチェンジの合計数。
Exchange.AGGREGATED_COMPLETED_BY	**String**	エクスチェンジの集約を完了するためのメカニズムを示します。使用できる値は、**predicate**、**size**、**timeout**、**interval**、または **consumer** です。

以下のプロパティーは、SQL コンポーネント集約リポジトリーによって再配信されるエクスチェンジに設定されます (「永続集計リポジトリー」 を参照)。

ヘッダー	型	再配信されたエクスチェンジプロパティー に関する説明
Exchange.REDELIVERY_COUNTER	**int**	現在の再配信試行のシーケンス番号 (**1** から開始)。

完了条件の指定

集約されたエクスチェンジが Aggregator から出て、ルート上の次のノードに遷移するタイミングを判断するので、**少なくとも 1つ** の完了条件を指定する必要があります。以下の完了条件を指定できます。

completionPredicate

各エクスチェンジが集約された後に述語を評価し、完全性を評価します。**true** の値は、集約エクスチェンジが完了したことを示します。または、このオプションを設定する代わりに、**Predicate** インターフェイスを実装するカスタム **AggregationStrategy** を定義することができます。この場合、完了述語として **AggregationStrategy** が使用されます。

completionSize

指定された数の受信エクスチェンジが集約された後、エクスチェンジの集約を完了します。

completionTimeout

(completionInterval とは互換性がありません) 指定されたタイムアウト内に受信エクスチェンジが集約されない場合、エクスチェンジの集約を完了します。
つまり、タイムアウトメカニズムは **各** 相関キー値のタイムアウトを追跡します。特定のキー値を持つ最新のエクスチェンジを受け取ると、クロックがカウントを開始します。指定したタイムアウト値の間に同じキー値を持つ別のエクスチェンジが受信**されない** 場合、対応するエクスチェンジの集約は完了とマークされ、ルート上の次のノードに遷移します。

completionInterval

(completionTimeout とは互換性がありません) 各時間間隔 (指定された長さ) が経過した後、未処理のエクスチェンジの集約を **すべて** 完了します。
時間間隔は、各エクスチェンジの集約毎に調整されて**いません**。このメカニズムは、すべての未処理のエクスチェンジに対し、集約の完了を強制します。したがって、場合によっては、このメカニズムは集約開始直後にエクスチェンジの集約を完了できます。

completionFromBatchConsumer

バッチコンシューマーメカニズムをサポートするコンシューマーエンドポイントと組み合わせて使用すると、この完了オプションは、コンシューマーエンドポイントから受信した情報に基づいて、現在のエクスチェンジのバッチが完了したタイミングを自動的に算出します。「バッチコンシューマー」 を参照してください。

forceCompletionOnStop

このオプションを有効にすると、現在のルートコンテキストが停止したときに、未処理のエクスチェンジの集約をすべて強制的に完了させます。

前述の完了条件は、**completionTimeout** および **completionInterval** のみ同時に有効にできませんが、任意に組み合わせることができます。条件を組み合わせて使用する場合、最初にトリガーされた完了条件が、有効な完了条件になるのが一般的です。

完了述語の指定

エクスチェンジの集約が完了するタイミングを決定する任意の述語式を指定できます。述語式を評価する方法は 2 つあります。

- **最新の集約されたエクスチェンジ** - これがデフォルトの動作です。
- **最新の受信エクスチェンジ** - **eagerCheckCompletion** オプションを有効にすると、この動作が選択されます。

たとえば、**ALERT** メッセージを受信するたびに (最新の受信エクスチェンジの **MsgType** ヘッダーの値で示される)、株式相場のストリームを終了させたい場合は、以下のようなルートを定義できます。

```
from("direct:start")
    .aggregate(
      header("id"),
      new UseLatestAggregationStrategy()
    )
        .completionPredicate(
          header("MsgType").isEqualTo("ALERT")
         )
        .eagerCheckCompletion()
    .to("mock:result");
```

以下の例は、XML を使用して同じルートを設定する方法を示しています。

```
<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"/>
```

動的な完了タイムアウトの指定

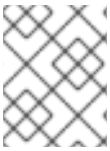
動的な完了タイムアウトを指定できます。この場合、タイムアウト値が受信エクスチェンジごとに再計算されます。たとえば、各受信エクスチェンジで **timeout** ヘッダーからタイムアウト値を設定するには、以下のようにルートを定義します。

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

XML DSL で以下のように同じルートを設定できます。

```
<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"/>
```



注記

固定のタイムアウト値を追加することもでき、動的なタイムアウト値が **null** または **0** であった場合、Apache Camel はこの固定値を使用するようにフォールバックします。

動的な完了サイズの指定

動的な完了サイズ を指定することが可能であり、受信エクスチェンジ毎に完了サイズは再計算されます。たとえば、各受信エクスチェンジの **mySize** ヘッダーから完了サイズを設定するには、以下のようにルートを定義します。

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

Spring XML を使用した同じ例を以下に示します。

```
<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"/>
```

注記

固定のサイズ値を追加することもできます。動的な値が **null** または **0** であった場合、Apache Camel はこの固定値を使用するようにフォールバックします。

集約ストラテジー内から単一グループを強制完了

カスタム **AggregationStrategy** クラスを実装した場合、**AggregationStrategy.aggregate()** メソッド

から返されたエクスチェンジで **Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP** エクスチェンジプロパティーを **true** に設定することで、現在のメッセージグループを強制的に完了させる機能があります。この機能は現在のグループに **のみ** 影響します。他のメッセージグループ (異なる相関 ID を持つ) は強制的に完了**しません**。この機能は、述語、サイズ、タイムアウトなど、その他の完了機能に対して上書きされます。

たとえば、以下のサンプル **AggregationStrategy** クラスは、メッセージのボディーサイズが 5 より大きい場合に現在のグループを完了します。

```
// 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;
    }
}
```

特別なメッセージですべてのグループを強制完了

特別なヘッダーを持つメッセージをルートに送信することで、未処理のメッセージの集約を強制的に完了することができます。強制完了するために使用できる代替ヘッダー設定は 2 つあります。

Exchange.AGGREGATION_COMPLETE_ALL_GROUPS

true に設定して、現在の集約サイクルを強制的に完了させます。このメッセージはシグナルとして純粋に機能し、いかなる集約サイクルにも**含まれません**。このシグナルメッセージを処理した後、メッセージの内容は破棄されます。

Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE

true に設定して、現在の集約サイクルを強制的に完了させます。このメッセージは現在の集計サイクルに **含まれます**。

AggregateController の使用

org.apache.camel.processor.aggregate.AggregateController を使用すると、Java または JMX API を使用してランタイム時に集約を制御することができます。エクスチェンジのグループを強制的に完了したり、現在のランタイム統計情報をクエリーしたりするために使われます。

カスタムが設定されていない場合、Aggregator はデフォルト実装を提供します。これは、**getAggregateController()** メソッドを使用してアクセスできます。ただし、aggregateController を使用して、ルート内でコントローラーを簡単に設定することができます。

```
private AggregateController controller = new DefaultAggregateController();

from("direct:start")
```

```
    .aggregate(header("id"), new MyAggregationStrategy()).completionSize(10).id("myAggregator")
      .aggregateController(controller)
      .to("mock:aggregated");
```

また、**AggregateController** の API を使用して、強制的に完了することもできます。例えば、キー foo を持つグループを完了するには、次のコマンドを実行します。

```
int groups = controller.forceCompletionOfGroup("foo");
```

戻り値は完了したグループの数になります。すべてのグループを完了する API は以下のとおりです。

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

一意な相関キーの強制

集約シナリオによっては、エクスチェンジのバッチごとに相関キーが一意であるという条件を強制する場合があります。つまり、特定の相関キーを持つエクスチェンジの集約が完了したら、その相関キーを持つエクスチェンジの集約がこれ以上続行されないようにします。たとえば、ルートの後半部分で一意の相関キーの値を持つエクスチェンジを処理することを想定している場合に、この条件を実施することができます。

完了条件の設定方法によっては、特定の相関キーで複数のエクスチェンジの集約が生成されるリスクがある可能性があります。たとえば、特定の相関キーを持つ **すべての** エクスチェンジを受信するまで待機する補完述語を定義することもできますが、完了タイムアウトも定義しており、そのキーを持つすべてのエクスチェンジが到着する前に発火してしまう可能性もあります。この場合、遅れて到着するエクスチェンジは、同じ相関キーの値を持つ **2 つ目** のエクスチェンジの集約になる可能性があります。

このようなシナリオでは、**closeCorrelationKeyOnCompletion** オプションを設定することで、以前の相関キー値と重複した集約エクスチェンジを抑制するように Aggregator を設定することができます。相関キー値の重複を抑制するためには、Aggregator が以前の相関キーの値をキャッシュに記録する必要があります。このキャッシュのサイズ (キャッシュされた相関キーの数) は、**closeCorrelationKeyOnCompletion()** DSL コマンドの引数として指定されます。無制限サイズのキャッシュを指定するには、ゼロまたは負の整数値を渡します。たとえば、**10000** キー値のキャッシュサイズを指定するには、次のコマンドを実行します。

```
from("direct:start")
    .aggregate(header("UniqueBatchID"), new MyConcatenateStrategy())
        .completionSize(header("mySize"))
        .closeCorrelationKeyOnCompletion(10000)
    .to("mock:aggregated");
```

相関キー値が重複している状態でエクスチェンジの集約が完了すると、Aggregator は **ClosedCorrelationKeyException** 例外を出力します。

Simple 式を使用したストリームベースの処理

ストリーミングモードで **tokenizeXML** サブコマンドを使用して、Simple 言語式をトークンとして使うことができます。Simple 言語式を使用することで、動的トークンのサポートが可能になります。

たとえば、Java を使用してタグ **person** で区切られた名前のシーケンスを分割するには、**tokenizeXML** Bean および Simple 言語トークンを使用して、ファイルを **name** 要素に分割することができます。

```
public void testTokenizeXMLPairSimple() throws Exception {
        Expression exp = TokenizeLanguage.tokenizeXML("${header.foo}", null);
```

<person> で区切られた名前の入力文字列を取得し、**<person>** をトークンに設定します。

```
        exchange.getIn().setHeader("foo", "<person>");
        exchange.getIn().setBody("<persons><person>James</person><person>Claus</person>
<person>Jonathan</person><person>Hadrian</person></persons>");
```

入力から分割された名前をリストします。

```
        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));
    }
```

グループ化されたエクスチェンジ

送信バッチ内の集約されたすべてのエクスチェンジを、単一の **org.apache.camel.impl.GroupedExchange** ホルダークラスに統合できます。グループ化されたエクスチェンジを有効にするには、以下の Java DSL ルートに示されるように **groupExchanges()** オプションを指定します。

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

mock:result に送信されるグループ化されたエクスチェンジには、メッセージボディーに集約されたエクスチェンジのリストが含まれます。以下のコードは、後続のプロセッサーがリスト形式でグループ化されたエクスチェンジのコンテンツにアクセスする方法を示しています。

```
// Java
List<Exchange> grouped = ex.getIn().getBody(List.class);
```

注記

エクスチェンジをグループ化する機能を有効にする場合、集約ストラテジーを設定**するべきではありません** (エクスチェンジのグループ化機能は、それ自体が集約ストラテジーになります)。

注記

送信エクスチェンジのプロパティーからグループ化されたエクスチェンジにアクセスする従来の方法は現在非推奨となっており、今後のリリースで削除される予定です。

バッチコンシューマー

Aggregator は **batch consumer** パターンと連携して、バッチコンシューマーによって報告されるメッセージの総数を集約できます (バッチコンシューマーエンドポイントは受信エクスチェンジで **CamelBatchSize**、**CamelBatchIndex**、および **CamelBatchComplete** プロパティーを設定します)。たとえば、File コンシューマーエンドポイントで見つかったすべてのファイルを集約するには、以下のようなルートを使用することができます。

```
from("file://inbox")
    .aggregate(xpath("//order/@customerId"), new AggregateCustomerOrderStrategy())
    .completionFromBatchConsumer()
    .to("bean:processOrder");
```

現在、バッチコンシューマー機能をサポートしているエンドポイントは File、FTP、Mail、iBatis、および JPA です。

永続集計リポジトリー

デフォルトの Aggregator はインメモリーのみの **AggregationRepository** を使用します。保留中の集約されたエクスチェンジを永続的に保存する場合は、**SQL コンポーネント** を永続集計リポジトリーとして使用できます。SQL コンポーネントには **JdbcAggregationRepository** が含まれており、集約されたメッセージをオンザフライで永続化し、メッセージを失うことがないようにします。

エクスチェンジが正常に処理された場合、リポジトリーで **confirm** メソッドが呼び出されると、完了とマークされます。つまり、同じエクスチェンジが再度失敗すると、成功するまで再試行さることを意味します。

camel-sql への依存関係の追加

SQL コンポーネントを使用するには、プロジェクトに **camel-sql** への依存関係を含める必要があります。たとえば、Maven **pom.xml** ファイルを使用している場合は、以下を追記します。

```
<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>
```

集約データベーステーブルの作成

永続化のために、集約テーブルと完成テーブルをそれぞれデータベースに作成する必要があります。たとえば、以下のクエリーは **my_aggregation_repo** という名前のデータベーステーブルを作成します。

```
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)
);
}
```

集約リポジトリーの設定

集約リポジトリーの設定

フレームワーク XML ファイル (Spring または Blueprint など) で集約リポジトリーを設定する必要があります。

```
<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**、および **dataSource** プロパティーが必要です。永続集約リポジトリーの設定オプションの詳細は、**Apache Camel Component Reference Guide** の SQL Component を参照してください。

スレッドオプション

図8.6「Aggregator の実装」 にあるように、Aggregator はルートの後半部分から切り離されており、ルートの後半部分へ送信されたエクスチェンジは、専用のスレッドプールによって処理されます。デフォルトでは、このプールには 1 つのスレッドのみがあります。複数のスレッドを持つプールを指定する場合は、以下のように **parallelProcessing** オプションを有効にします。

```
from("direct:start")
    .aggregate(header("id"), new UseLatestAggregationStrategy())
        .completionTimeout(3000)
        .parallelProcessing()
    .to("mock:aggregated");
```

デフォルトでは、ワーカースレッドが 10 個あるプールが作成されます。

作成したスレッドプールをより詳細に制御する場合は、**executorService** オプションを使用してカスタム **java.util.concurrent.ExecutorService** インスタンスを指定します (この場合は、**parallelProcessing** オプションを有効化する必要はありません)。

List への集約

一般的な集約シナリオでは、一連の受信メッセージボディーを **List** オブジェクトに集約します。このシナリオを容易にするため、Apache Camel は **AbstractListAggregationStrategy** 抽象クラスを提供しています。このクラスを手早く拡張して、こういったシチュエーションに応じた集約ストラテジーを作成できます。**T** 型の受信メッセージボディーは、**List<T>** 型のメッセージボディーを持つ完了済みエクスチェンジへと集約されます。

たとえば、一連の **Integer** メッセージボディーを **List<Integer>** オブジェクトに集約するには、以下のように定義された集約ストラテジーを使用することができます。

```
import org.apache.camel.processor.aggregate.AbstractListAggregationStrategy;
...
/**
 * Strategy to aggregate integers into a List<Integer>.
 */
public final class MyListOfNumbersStrategy extends AbstractListAggregationStrategy<Integer> {

    @Override
```

```
    public Integer getValue(Exchange exchange) {
        // the message body contains a number, so just return that as-is
        return exchange.getIn().getBody(Integer.class);
    }
}
```

Aggregator のオプション

Aggregator は以下のオプションをサポートします。

表8.1 Aggregator のオプション

オプション	デフォルト	説明
correlationExpression		集約に使用する相関キーを評価するために必須な式。同じ相関キーを持つエクスチェンジが集約されます。相関キーを評価できない場合、例外が発生します。**ignoreBadCorrelationKeys** オプションを使用してこれを無効にすることができます。
aggregationStrategy		既存のすでにマージされたエクスチェンジと受信エクスチェンジを **マージ** するために使用される必須の **AggregationStrategy**。最初の呼び出し時、**oldExchange** パラメーターは **null** です。その後の呼び出しでは、**oldExchange** にはマージされたエクスチェンジが含まれ、**newExchange** は当然新しい受信エクスチェンジとなります。Camel 2.9.2 以降では、ストラテジーを任意で、タイムアウトコールバックをサポートする **TimeoutAwareAggregationStrategy** 実装にすることができます。Camel 2.16 以降では、ストラテジーを **PreCompletionAwareAggregationStrategy** 実装にすることもできます。pre-completion モードで、完了チェックを実行します。
strategyRef		レジストリーで **AggregationStrategy** を検索するための参照。

オプション	デフォルト	説明
completionSize		集約の完了前に、既に集約されたメッセージの数。このオプションは、固定値を設定することも、動的にサイズを評価する式を使用することもできます。式は **Integer** が結果として使用されます。両方が設定されている場合、Camel は式の結果が **null** または **0** であれば、固定値を使用するようにフォールバックします。
completionTimeout		集約されたエクスチェンジが完了するまで非アクティブになる時間 (ミリ秒単位) 。このオプションは、固定値として設定することも、動的にタイムアウトを評価する式を使用することもできます。式は **Long** が結果として使用されます。両方が設定されている場合、Camel は式の結果が **null** または **0** であれば、固定値を使用するようにフォールバックします。このオプションを completionInterval と併用することはできません。使用できるのはどちらか 1 つだけです。
completionInterval		Aggregator が現在すべてのエクスチェンジを集約し完了するまでに繰り返される期間 (ミリ秒単位)。Camel には、期間ごとにトリガーされるバックグラウンドタスクがあります。このオプションは completionTimeout と併用できません。使用できるのはどちらか 1 つだけです。
completionPredicate		集約されたエクスチェンジの完了時にシグナルを送る述語 (**org.apache.camel.Predicate** 型) を指定します。または、このオプションを設定する代わりに、**Predicate** インターフェイスを実装するカスタム **AggregationStrategy** を定義することができます。この場合、完了述語として **AggregationStrategy** が使用されます。

オプション	デフォルト	説明
completionFromBatchConsumer	**false**	このオプションは、エクスチェンジがバッチコンシューマーから送信される場合に使用します。有効にすると、「Aggregator」は、**CamelBatchSize** メッセージヘッダーの、バッチコンシューマーによって決定されるバッチサイズを使用します。詳細はバッチコンシューマーを参照してください。これは、ポーリングの際、File エンドポイントからコンシュームされたすべてのファイルを集約するために使用できます。
eagerCheckCompletion	**false**	新しい受信エクスチェンジを受け取ったときに、常に完了を確認するかどうか。このオプションは、エクスチェンジが渡される挙動が変わるため、**completionPredicate** オプションの動作に影響します。**false** の場合、述語に渡されるエクスチェンジは**集約された**エクスチェンジであり、**AggregationStrategy** から集約されたエクスチェンジに保存できる任意の情報を述語で利用することができます。**true** の場合、述語に渡されるエクスチェンジは **受信** エクスチェンジであり、受信エクスチェンジからデータにアクセスできます。
forceCompletionOnStop	**false**	**true** の場合、現在のルートコンテキストが停止したときにすべての集約されたエクスチェンジを完了します。

オプション	デフォルト	説明
groupExchanges	**false**	この機能を有効にすると、Camel は集約されたすべてのエクスチェンジを 1 つの **org.apache.camel.impl.GroupedExchange** ホルダークラスにグループ化し、ホルダークラスはすべての集約されたエクスチェンジを保持します。その結果、Aggregator から送信されるエクスチェンジは 1 つだけです。カスタム **AggregationStrategy** を実装せずに、多くの受信エクスチェンジを 1 つの受信エクスチェンジへ集約するために使用できます。
ignoreInvalidCorrelationKeys	**false**	評価できない相関キーを無視するかどうか。デフォルトでは Camel は例外を出力しますが、このオプションを有効にして状況を無視することもできます。
closeCorrelationKeyOnCompletion		**遅い** エクスチェンジを受け入れるべきかどうか。これを有効にすると、相関キーがすでに完了している場合に、同じ相関キーを持つ新しいエクスチェンジを拒否することができます。その後、Camel は **closedCorrelationKeyException** 例外を出力します。このオプションを使用する場合、**integer** を渡します。これは、LRUCache の数であり、最後の X 個のクローズした相関キーを保持します。0 または負の値を渡すことで、無制限のキャッシュを示すことができます。数値を渡すことで、異なる相関キーを大量に使用しても キャッシュが肥大化しないことが保証されます。
discardOnCompletionTimeout	**false**	Camel 2.5: タイムアウトによって完了したエクスチェンジが破棄されるべきかどうか。有効にした場合、タイムアウトが発生すると集約されたメッセージは送信 **されず**、破棄されます。

オプション	デフォルト	説明
aggregationRepository		現在実行中 (inflight) の集約されたエクスチェンジを追跡する **org.apache.camel.spi.AggregationRepository** の独自実装をプラグインすることができます。Camel はデフォルトでメモリーベースの実装を使用します。
aggregationRepositoryRef		レジストリーで **aggregationRepository** を検索するための参照。
parallelProcessing	**false**	集約が完了すると、Aggregator から送信されます。このオプションは、Camel が並列実行用に複数のスレッドを持つスレッドプールを使用するかどうかを指定します。カスタムスレッドプールが指定されていない場合、Camel は 10 個のスレッドを持つデフォルトプールを作成します。
executorService		**parallelProcessing** を使用する場合は、使用するカスタムスレッドプールを指定することができます。また、**parallelProcessing** を使用しない場合も、集約されたエクスチェンジはこのカスタムスレッドプールを使用して送信されます。
executorServiceRef		レジストリーで **executorService** を検索するための参照。
timeoutCheckerExecutorService		**completionTimeout**、**completionTimeout**、**Expression**、または **completionInterval** のいずれかのオプションを使用している場合、バックグラウンドスレッドが作成され、すべての aggregator の完了を確認します。個々の Aggregator へ新しいスレッドを作成するのではなく、カスタムスレッドプールを提供する場合は、このオプションを設定します。

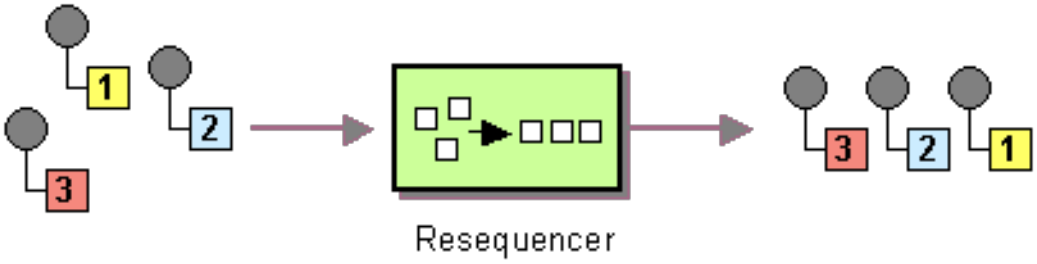
オプション	デフォルト	説明
timeoutCheckerExecutorServiceRef		レジストリーで **timeoutCheckerExecutorService** を検索するための参照。
completeAllOnStop		Aggregator を停止する場合、このオプションを使用すると、集約リポジトリーから保留中のエクスチェンジをすべて完了することができます。
optimisticLocking	**false**	楽観的ロックを有効にします。このロックは、集約リポジトリーと組み合わせて使用できます。
optimisticLockRetryPolicy		楽観的ロックのための Retry ポリシーを設定します。

8.6. RESEQUENCER

概要

図8.7「Resequencer パターン」 に示されている **Resequencer** パターンを使用すると、シーケンス式に応じてメッセージを再配列できます。シーケンス式の値が低いメッセージはバッチの先頭に移動し、値が高いメッセージは後ろに移動します。

図8.7 Resequencer パターン



Apache Camel は、2 つの再配列アルゴリズムをサポートします。

- **Batch resequencing**: メッセージをバッチで収集し、メッセージをソートして出力に送信します。
- **Stream resequencing**: メッセージ間のギャップの検出に基づいて、(継続的な) メッセージ ストリームを再順序付けします。

デフォルトでは、Resequencer は重複メッセージをサポートしておらず、同じメッセージ式のメッセージが到達した場合は、最後のメッセージのみを保持します。ただし、バッチモードでは、Resequencer で重複を許可することができます。

Batch resequencing

Batch resequencing アルゴリズムは、デフォルトで有効になっています。たとえば、**TimeStamp** ヘッダーに含まれるタイムスタンプの値に基づいて受信メッセージのバッチを再配列するには、Java DSL で以下のルートを定義することができます。

```
from("direct:start").resequence(header("TimeStamp")).to("mock:result");
```

デフォルトでは、最大 100 メッセージ (デフォルトの **バッチサイズ**) までとし、1000 ミリ秒 (デフォルトの **バッチタイムアウト**) のインターバルで到着するすべての受信メッセージを収集することによって、バッチを取得します。バッチタイムアウトおよびバッチサイズの値をカスタマイズするには、**BatchResequencerConfig** インスタンスが唯一の引数として使用される **batch()** DSL コマンドを追加します。たとえば、バッチが最大 300 メッセージまでの、4000 ミリ秒インターバルでメッセージを収集するように前述のルートを変更するには、以下のように Java DSL ルートを定義することができます。

```
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");
    }
};
```

XML 設定を使用して Batch resequencer パターンを指定することもできます。以下の例は、バッチサイズが 300 で、バッチタイムアウトが 4000 ミリ秒の Batch resequencer を定義しています。

```
<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>
```

バッチオプション

表8.2「Batch Resequencer オプション」 は、バッチモードでのみ使用できるオプションを示しています。

表8.2 Batch Resequencer オプション

Java DSL	XML DSL	デフォルト	説明

Java DSL	XML DSL	デフォルト	説明
allowDuplicates()	**batch-config/@allowDuplicates**	**false**	**true** の場合、バッチは重複したメッセージを破棄しません (**重複** は、メッセージ式が同じ値に評価されることを意味します)。
reverse()	**batch-config/@reverse**	**false**	**true** の場合、メッセージを逆順で配列します (メッセージ式に適用されるデフォルトの順序は、String.compareTo() で定義されている Java の文字列語彙順に基づいています)。

たとえば、**JMSPriority** に基づいて JMS キューからのメッセージを再配列する場合は、以下のように **allowDuplicates** および **reverse** オプションを組み合わせる必要があります。

```
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");
```

Stream resequencing

Stream resequencing アルゴリズムを有効にするには、**resequence()** DSL コマンドに **stream()** を追加する必要があります。たとえば、**seqnum** ヘッダーのシーケンス番号の値に基づいて受信メッセージを再配列するには、以下のように DSL ルートを定義することができます。

```
from("direct:start").resequence(header("seqnum")).stream().to("mock:result");
```

Stream-prosessing resequencer アルゴリズムは、固定のバッチサイズではなく、メッセージストリーム内のギャップ検出に基づいています。ギャップ検出はタイムアウトと組み合わせることで、シーケンスのメッセージ数 (バッチサイズ) を事前に把握する必要がなくなります。メッセージには、先行および後継がわかる一意のシーケンス番号が含まれている必要があります。たとえば、シーケンス番号 **3** を持つメッセージには、シーケンス番号 **2** が含まれる先行メッセージと、シーケンス番号 **4** を持つ後継メッセージがあります。メッセージのシーケンス **2,3,5** は、**3** の後継がないため、ギャップがあります。したがって、Resequencer は、メッセージ **4** が到着するまで (またはタイムアウトが発生するまで)、メッセージ **5** を保持する必要があります。

デフォルトでは、Stream Resequencer はタイムアウトは 1000 ミリ秒、最大メッセージ容量は 100 で設定されます。ストリームのタイムアウトおよびメッセージ容量をカスタマイズするには、**StreamResequencerConfig** オブジェクトを引数として **stream()** に渡します。たとえば、メッセージ容量が 5000 でタイムアウトが 4000 ミリ秒の Stream resequencer を設定するには、以下のようにルートを定義することができます。

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

メッセージストリーム内の連続するメッセージ (つまり、連続するシーケンス番号を持つメッセージ) 間の最大遅延時間が分かっている場合は、Resequencer の timeout パラメーターに、この値を設定する必要があります。この場合、ストリーム内のすべてのメッセージが正しい順序で次のプロセッサーに送信されることを保証することができます。シーケンス外となる時間差よりもタイムアウト値が小さいほど、Resequencer が未配列のメッセージを配信する可能性が高くなります。大きなタイムアウト値は、充分に高い容量値でサポートされるべきであり、ここでは容量パラメーターを使用して、Resequencer のメモリーが枯渇するのを防いでいます。

long 以外の型でシーケンス番号を使用する場合は、以下のようにカスタム comparator を定義する必要があります。

```
// Java
ExpressionResultComparator<Exchange> comparator = new MyComparator();
StreamResequencerConfig config = new StreamResequencerConfig(5000, 4000L, comparator);
from("direct:start").resequence(header("seqnum")).stream(config).to("mock:result");
```

XML 設定を使用して Stream resequencer パターンを指定することもできます。以下の例は、メッセージ容量が 5000 で、タイムアウトが 4000 ミリ秒の Stream resequencer を定義します。

```
<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>
```

無効なエクスチェンジの無視

Resequencer EIP は、受信エクスチェンジが有効でない場合に **CamelExchangeException** 例外を出力します。これは、シーケンス式が何らかの理由で評価できない場合 (ヘッダーが見つからない場合など) が該当します 。**ignoreInvalidExchanges** オプションを使用して、これらの例外を無視することができます。つまり、Resequencer は無効なエクスチェンジをスキップします。

```
from("direct:start")
  .resequence(header("seqno")).batch().timeout(1000)
    // ignore invalid exchanges (they are discarded)
    .ignoreInvalidExchanges()
  .to("mock:result");
```

古いメッセージを拒否

この **rejectOld** オプションを使用すると、メッセージの再配列に使用されるメカニズムに関係なく、メッセージが未配列のまま送信されるのを防ぐことができます。**rejectOld** オプションを有効にすると、受信メッセージが最後に配信されたメッセージよりも **古い** (現在の comparator によって定義されている) 場合に、Resequencer は受信メッセージを拒否します (**MessageRejectedException** 例外を出力します)。

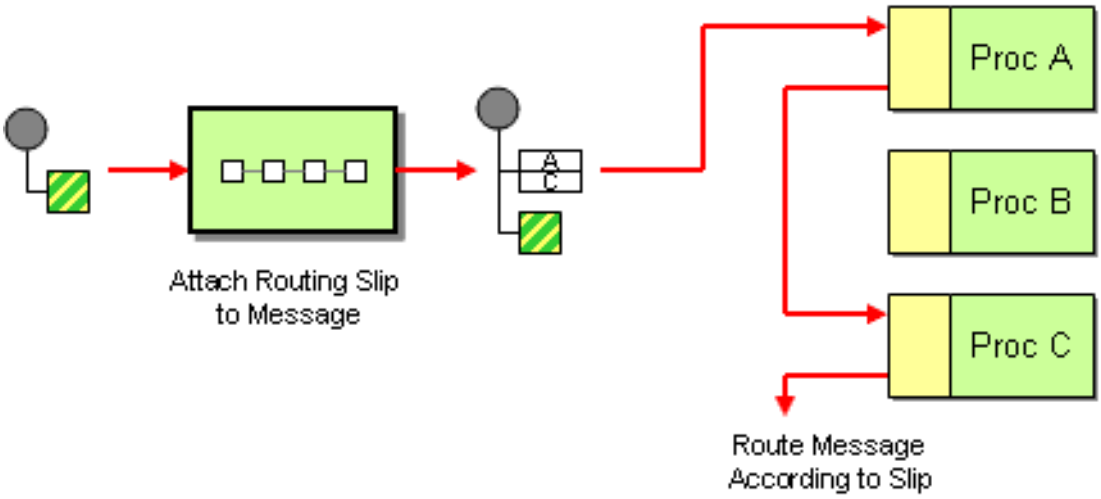
```
from("direct:start")
    .onException(MessageRejectedException.class).handled(true).to("mock:error").end()
    .resequence(header("seqno")).stream().timeout(1000).rejectOld()
    .to("mock:result");
```

8.7. ROUTING SLIP

概要

図8.8「Routing Slip パターン」 に示されている **Routing Slip** パターンでは、一連のプロセスステップを通してメッセージを順番にルーティングできます。設計時にこのステップの順序は分かっておらず、メッセージごとに順序が変わる場合があります。メッセージが通過するエンドポイントのリストは、ヘッダーフィールド (**Slip**) に格納されます。Apache Camel は実行時に読み込み、パイプラインを構築します。

図8.8 Routing Slip パターン



Slip ヘッダー

Routing Slip は、ユーザー定義のヘッダーに表示されます。ヘッダーの値は、エンドポイント URI のコンマ区切りリストになります。たとえば、メッセージの復号、認証、および重複排除などの一連のセキュリティータスクの順序を指定する Routing Slip は、以下のようになります。

```
cxf:bean:decrypt,cxf:bean:authenticate,cxf:bean:dedup
```

現在の エンドポイントプロパティー

Camel 2.5 から Routing Slip は、現在のエンドポイントが slip であった場合、そのエンドポイントが含まれるプロパティー (**Exchange.SLIP_ENDPOINT**) をエクスチェンジに設定します。これにより、エクスチェンジが slip 経由でどこまで進んでいるかを調べることができます。

「Routing Slip」 は、**事前** に Slip を演算します。つまり、Slip は 1 回だけ演算されます。**その場** で Slip を演算する必要がある場合は、代わりに 「Dynamic Router」 パターンを使用してください。

Java DSL の例

以下のルートは、**direct:a** エンドポイントからメッセージを取得し、**aRoutingSlipHeader** ヘッダーから Routing Slip を読み込みます。

```
from("direct:b").routingSlip("aRoutingSlipHeader");
```

ヘッダー名は、文字列リテラルまたは式として指定することができます。

routingSlip() の 2 つの引数形式を使用して、URI の区切り文字をカスタマイズすることもできます。以下の例では、routing slip に **aRoutingSlipHeader** ヘッダーキーを使用し、**#** 文字を URI 区切り文字として使用するルートを定義しています。

```
from("direct:c").routingSlip("aRoutingSlipHeader", "#");
```

XML 設定の例

以下の例は、XML で同じルートを設定する方法を示しています。

```
<camelContext id="buildRoutingSlip" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:c"/>
    <routingSlip uriDelimiter="#">
      <headerName>aRoutingSlipHeader</headerName>
    </routingSlip>
  </route>
</camelContext>
```

無効なエンドポイントの無視

「Routing Slip」 は、「受信者リスト」 パターンもサポートする **ignoreInvalidEndpoints** をサポートするようになりました。これを使用して、無効なエンドポイントをスキップすることができます。以下に例を示します。

```
from("direct:a").routingSlip("myHeader").ignoreInvalidEndpoints();
```

Spring XML では、この機能は **<routingSlip>** タグに **ignoreInvalidEndpoints** 属性を設定して有効にします。

```
<route>
    <from uri="direct:a"/>
    <routingSlip ignoreInvalidEndpoints="true">
      <headerName>myHeader</headerName>
    </routingSlip>
</route>
```

myHeader に **direct:foo,xxx:bar** の 2 つのエンドポイントが含まれるケースについて考えてみましょう。最初のエンドポイントは有効であり、動作します。2 つ目は無効であるため、無視されます。無効なエンドポイントに遭遇するたびに、Apache Camel ログが **INFO** レベルで記録されます。

オプション

routingSlip DSL コマンドは、以下のオプションをサポートします。

名前	デフォルト値	説明
uriDelimiter	,	式が複数のエンドポイントを返した場合に使用される区切り文字。
ignoreInvalidEndpoints	**false**	エンドポイント URI を解決できなかった場合は、無視されます。false の場合は、Camel はエンドポイント URI が有効ではないことを示す例外を出力します。
cacheSize	**0**	**Camel 2.13.1/2.12.4:** Routing Slip で再使用されるプロデューサーをキャッシュする **ProducerCache** のキャッシュサイズを設定できます。デフォルトのキャッシュサイズ 0 を使用します。値を -1 に設定すると、キャッシュをすべて無効にすることができます。

8.8. THROTTLER

概要

Throttler は、受信メッセージのフローレートを制限するプロセッサーです。このパターンを使用して、ターゲットエンドポイントがオーバーロードされないように保護することができます。Apache Camel では、**throttle()** Java DSL コマンドを使用して Throttler パターンを実装できます。

Java DSL の例

フロー速度を毎秒 100 メッセージに制限するには、以下のようにルートを定義します。

```
from("seda:a").throttle(100).to("seda:b");
```

必要な場合は、**timePeriodMillis()** DSL コマンドを使用してフローレートを制御する期間をカスタマイズすることができます。たとえば、30000 ミリ秒あたりのフローレートを 3 つのメッセージに制限するには、以下のようにルートを定義します。

```
from("seda:a").throttle(3).timePeriodMillis(30000).to("mock:result");
```

XML 設定の例

以下の例は、XML で前述のルートを設定する方法を示しています。

```
<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>
```

期間ごとに最大リクエスト数を動的に変更

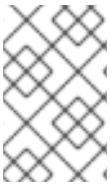
Camel 2.8 で利用可能 式を使用しているため、値を実行時に調整できます。たとえば、ヘッダーで値を指定できます。実行時に Camel は式を評価し、結果を **java.lang.Long** 型に変換します。以下の例では、メッセージのヘッダーを使用して、期間ごとの最大リクエスト数を決定します。ヘッダーがない場合、「Throttler」 は古い値を使用します。そのため、値を変更する場合にのみ、ヘッダーを提供することができます。

```
<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>
```

非同期の遅延

Throttler は、**ノンブロッキングの非同期遅延** を有効にすることができます。これは、Apache Camel がタスクを今後実行するようにスケジュールすることを意味します。このタスクは、ルートの後半部分 (Throttler の後) の処理を担当します。これにより、呼び出し元スレッドはブロックされず、次の受信メッセージに対応することができます。以下に例を示します。

```
from("seda:a").throttle(100).asyncDelayed().to("seda:b");
```



注記

Camel 2.17 から、Throttler はメッセージのフローを改善するための期間に対し、ローリングウィンドウを使用するようになりました。ただし、Throttler のパフォーマンスが向上されます。

オプション

throttle DSL コマンドは、以下のオプションをサポートします。

名前	デフォルト値	説明

maximumRequestsPerPeriod		スロットルする期間毎の最大リクエスト数。このオプションは省略不可で、正の数字を指定する必要があります。XML DSL では、このオプションは Camel 2.8 以降は属性ではなく式を使用して設定されます。
timePeriodMillis	**1000**	Throttler が最大で **maximumRequestsPerPeriod** メッセージ数を許可する期間 (ミリ秒単位)。
asyncDelayed	**false**	Camel 2.4: 有効にすると、遅延しているメッセージはスケジュールされたスレッドプールを使用して非同期的に実行されます。
executorServiceRef		Camel 2.4: **asyncDelay** が有効になっている場合に使用される、カスタムスレッドプールへの参照。
callerRunsWhenRejected	**true**	Camel 2.4: **asyncDelayed** が有効な場合に使用されます。これは、スレッドプールがタスクを拒否した場合に、呼び出し元スレッドがタスクを実行するべきかどうかを制御します。

8.9. DELAYER

概要

Delayer は、受信メッセージに **相対的な** 遅延を適用できるプロセッサーです。

Java DSL の例

delay() コマンドを使用して、受信メッセージに **相対的な** 遅延 (ミリ秒単位) を追加することができます。たとえば、以下のルートは、すべての受信メッセージを 2 秒遅延します。

```
from("seda:a").delay(2000).to("mock:result");
```

あるいは、式を使用して遅延を指定することもできます。

```
from("seda:a").delay(header("MyDelay")).to("mock:result");
```

delay() に続く DSL コマンドは、 **delay()** のサブ句として解釈されます。そのため、場合によっては **end()** コマンドを挿入して、delay() のサブ句を終了する必要があります。たとえば、 **delay()** が **onException()** 句の中にある場合、以下のように終了します。

```
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 設定の例

以下は、XML DSL で delay を使用した例となります。

```
<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>
```

カスタム delay の作成

式と Bean を組み合わせて使用して、以下のように遅延を決定することができます。

```
from("activemq:foo").
  delay().expression().method("someBean", "computeDelay").
  to("activemq:bar");
```

Bean クラスは以下のように定義することができます。

```
public class SomeBean {
  public long computeDelay() {
     long delay = 0;
     // use java code to compute a delay value in millis
     return delay;
 }
}
```

非同期の遅延

Delayer に、**ノンブロッキングの非同期遅延** を使用させることができます。これは、Apache Camel がタスクを今後実行するようにスケジュールすることを意味します。このタスクは、ルートの後半部分 (Delayer の後) の処理を担当します。これにより、呼び出し元スレッドはブロックされず、次の受信メッセージに対応することができます。以下に例を示します。

```
from("activemq:queue:foo")
    .delay(1000)
    .asyncDelayed()
    .to("activemq:aDelayedQueue");
```

以下のように XML DSL で同じルートを作成できます。

```
<route>
  <from uri="activemq:queue:foo"/>
  <delay asyncDelayed="true">
      <constant>1000</constant>
  </delay>
  <to uri="activemq:aDealyedQueue"/>
  </route>
```

オプション

Delayer パターンでは、以下のオプションがサポートされます。

名前	デフォルト値	説明
asyncDelayed	**false**	Camel 2.4: 有効にすると、遅延しているメッセージはスケジュールされたスレッドプールを使用して非同期的に実行されます。
executorServiceRef		Camel 2.4: **asyncDelay** が有効になっている場合に使用される、カスタムスレッドプールへの参照。
callerRunsWhenRejected	**true**	Camel 2.4: **asyncDelayed** が有効な場合に使用されます。これは、スレッドプールがタスクを拒否した場合に、呼び出し元スレッドがタスクを実行するべきかどうかを制御します。

8.10. LOAD BALANCER

概要

Load Balancer パターンにより、さまざまな負荷分散ポリシーを使用して、複数のエンドポイントのうちの 1 つにメッセージ処理を委譲することができます。

Java DSL の例

以下のルートは、ラウンドロビン負荷分散ポリシーを使用して、ターゲットエンドポイント (**mock:x**、**mock:y**、**mock:z**) 間で受信メッセージを分散します。

```
from("direct:start").loadBalance().roundRobin().to("mock:x", "mock:y", "mock:z");
```

XML 設定の例

以下の例は、XML で同じルートを設定する方法を示しています。

```
<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>
```

負荷分散ポリシー

Apache Camel ロードバランサーは、以下の負荷分散ポリシーをサポートしています。

- ラウンドロビン
- ランダム
- スティッキー
- トピック
- フェイルオーバー
- 重み付きラウンドロビンおよび重み付きランダム
- カスタムロードバランサー
- サーキットブレーカー

ラウンドロビン

ラウンドロビン負荷分散ポリシーは、すべてのターゲットエンドポイントを循環し、各受信メッセージをサイクル内の次のエンドポイントに送信します。たとえば、ターゲットエンドポイントのリストが **mock:x**、**mock:y**、**mock:z** である場合、受信メッセージは **mock:x**、**mock:y**、**mock:z**、**mock:x**、**mock:y**、**mock:z** のような順番でエンドポイントに送信されます。

以下のように、Java DSL では ラウンドロビン負荷分散ポリシーを指定することができます。

```
from("direct:start").loadBalance().roundRobin().to("mock:x", "mock:y", "mock:z");
```

または、以下のように XML で同じルートを定義することもできます。

```
<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>
```

ランダム

ランダム負荷分散ポリシーは、指定されたリストからターゲットエンドポイントを無作為に選択します。

以下のように、Java DSL ではランダム負荷分散ポリシーを指定することができます。

```
from("direct:start").loadBalance().random().to("mock:x", "mock:y", "mock:z");
```

または、以下のように XML で同じルートを定義することもできます。

```
<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>
```

スティッキー

スティッキー負荷分散ポリシーは、指定された式からハッシュ値を計算して選択されたエンドポイントに、**In** メッセージを送信します。この負荷分散ポリシーの利点は、同じ値の式であれば、常に同じサーバーに送信されることです。たとえば、ユーザー名が含まれるヘッダーのハッシュ値を計算することで、特定のユーザーのメッセージが同じターゲットエンドポイントへ常に送信されるようになります。もう 1 つの便利な方法は、受信メッセージからセッション ID を抽出する式を指定することです。これにより、同じセッションに属するすべてのメッセージが同じターゲットエンドポイントに送信されるようになります。

以下のように、Java DSL ではスティッキー負荷分散ポリシーを指定することができます。

```
from("direct:start").loadBalance().sticky(header("username")).to("mock:x", "mock:y", "mock:z");
```

または、以下のように XML で同じルートを定義することもできます。

```
<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>
```

注記

スティッキーオプションをフェイルオーバーロードバランサーに追加すると、ロードバランサーは最後に認知した、正常なエンドポイントから開始します。

トピック

トピック負荷分散ポリシーは、各 **In** メッセージのコピーをリストされた **すべて** の宛先エンドポイントに送信します (JMS トピックのように、すべての宛先にメッセージを効果的にブロードキャストします)。

Java DSL を使用して、以下のようにトピック負荷分散ポリシーを指定することができます。

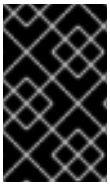
```
from("direct:start").loadBalance().topic().to("mock:x", "mock:y", "mock:z");
```

または、以下のように XML で同じルートを定義することもできます。

```
<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>
```

Failover

Apache Camel 2.0 で利用可能 failover ロードバランサーは、エクスチェンジの処理中に **exception** で失敗した場合に、次のプロセッサーを試すことができます。フェイルオーバーをトリガーする特定の例外のリストを使い、**failover** を設定することができます。例外を指定しない場合、フェイルオーバーはいずれの例外でもトリガーされます。フェイルオーバーロードバランサーは、**onException** 例外句と同じストラテジーを使い、例外のマッチングを行います。



ストリーム使用時にストリームキャッシュを有効にする

ストリーミングを使用する場合は、フェイルオーバーロードバランサーの使用時に、Stream Caching を有効にする必要があります。これは、フェイルオーバー時にストリームを再読み取りできるようにするために必要です。

failover ロードバランサーは以下のオプションをサポートします。

オプション	タイプ	デフォルト	説明
inheritErrorHandler	**boolean**	**true**	Camel 2.3: ルートに設定された **errorHandler** を使用するかどうかを指定します。次のエンドポイントへ即座にフェイルオーバーする場合は、このオプションを無効にする必要があります (**false** の値)。このオプションを有効にすると、Apache Camel は最初に **errorHandler** を使用してメッセージの処理を試みます。 たとえば、**errorHandler** はメッセージを再送し、試行間の遅延を使用するように設定されている可能性があります。Apache Camel は、最初に **オリジナル** のエンドポイントに再配信を試み、**errorHandler** を使い切った場合にのみ次のエンドポイントへフェイルオーバーします。
maximumFailoverAttempts	**int**	**-1**	Camel 2.3: 新しいエンドポイントへのフェイルオーバーの最大試行回数を指定します。値 **0** は、フェイルオーバー**されない**ことを意味し、値 **-1** は、無制限にフェイルオーバーが試行されることを意味します。

roundRobin	**boolean**	**false**	Camel 2.3: **failover** ロードバランサーがラウンドロビンモードで動作するかどうかを指定します。false の場合、新規メッセージが処理される際には **常に** 最初のエンドポイントから開始されます。つまり、すべてのメッセージは、それぞれ先頭にリセットされます。ラウンドロビンが有効になっている場合は、状態を保持し、ラウンドロビン方式で次のエンドポイントへと進みます。ラウンドロビンを使用する場合、最後に認識された適切なエンドポイントに **固定** されず、常に次のエンドポイントを選択して使用します。

以下の例では、**IOException** 例外が出力された場合にのみフェイルオーバーするように設定されています。

```
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");
```

オプションで以下に示すように、フェイルオーバーする例外を複数指定することができます。

```
// enable redelivery so failover can react
errorHandler(defaultErrorHandler().maximumRedeliveries(5));

from("direct:foo")
    .loadBalance()
    .failover(IOException.class, MyOtherException.class)
    .to("direct:a", "direct:b");
```

XML で以下のように同じルートを設定できます。

```
<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>
```

以下の例は、ラウンドロビンモードでフェイルオーバーする方法を示しています。

```
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");
```

XML で以下のように同じルートを設定できます。

```
<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>
```

できるだけ早く次のエンドポイントにフェイルオーバーする場合は、**inheritErrorHandler=false** を設定することで **inheritErrorHandler** を無効にすることができます。エラーハンドラーを無効にすることで、エラーハンドラーが介入しないようにすることができます。これにより、フェイルオーバーロードバランサーはすぐにフェイルオーバー処理をできるようになります。**roundRobin** モードも有効にすると、成功するまでリトライが行われます。**maximumFailoverAttempts** オプションを高い値に設定すると、最終的にしきい値を越え、失敗することができます。

重み付きラウンドロビンおよび重み付きランダム

多くのエンタープライズ環境では、処理能力が不均等なサーバーノードがサービスをホストしており、通常は個々のサーバー処理能力に応じて負荷を分散することが望ましくなります。この問題に対処するために、 **重み付きラウンドロビン** アルゴリズム、または **重み付きランダム** アルゴリズムを使用することができます。

重み付けされた負荷分散ポリシーを使用すると、サーバーごとに負荷 **分散率** を指定できます。この値を、各サーバーごとに正の処理の重みとして指定することができます。数値が大きいほど、サーバーがより大きな負荷を処理できることを示します。処理の重みを使用して、各処理エンドポイントに対するペイロードの分散比率を決定します。

使用可能なパラメーターは、以下の表に記載されています。

表8.3 重み付けオプション

オプション	タイプ	デフォルト	説明
roundRobin	**boolean**	**false**	ラウンドロビンのデフォルト値は **false** です。この設定またはパラメーターがない場合、使用される負荷分散アルゴリズムはランダムです。
distributionRatioDelimiter	**String**	**,**	**distributionRatioDelimiter** は **distributionRatio** を指定するために使用される区切り文字です。この属性が指定されていない場合、コンマ **,** がデフォルトの区切り文字になります。

以下の Java DSL の例では、重み付けされたラウンドロビンのルートと、重み付けされたランダムのルートを定義する方法を示しています。

```
// 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");
```

XML で以下のようにラウンドロビンのルートを設定できます。

```
<!-- 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>
```

カスタムロードバランサー

カスタムロードバランサー (独自の実装など) も使用することができます。

Java DSL を使用した例:

```
from("direct:start")
    // using our custom load balancer
    .loadBalance(new MyLoadBalancer())
    .to("mock:x", "mock:y", "mock:z");
```

XML DSL を使用した場合の同じ例:

```
<!-- 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>
```

上記の XML DSL では、**Camel 2.8** 以降でのみ利用できる <custom> を使用していることに注意してください。以前のリリースでは、代わりに以下のようにする必要がありました。

```
      <loadBalance ref="myBalancer">
        <!-- these are the endpoints to balancer -->
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
      </loadBalance>
```

カスタムロードバランサーを実装するには、**LoadBalancerSupport** および **SimpleLoadBalancerSupport** などの一部のサポートクラスを拡張することができます。前者は非同期ルーティングエンジンに対応し、後者は対応していません。以下に例を示します。

```
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;
    }
}
```

サーキットブレーカー

サーキットブレーカーロードバランサーは、特定の例外に対するすべての呼び出しを監視するために使用されるステートフルなパターンです。初期状態では、サーキットブレーカーはクローズ状態であり、すべてのメッセージを渡します。失敗があり、しきい値に達すると、オープン状態に遷移し、**halfOpenAfter** タイムアウトに達するまですべての呼び出しを拒否します。タイムアウト後に新しい呼び出しがあった場合、サーキットブレーカーはすべてのメッセージを渡します。結果が成功すると、サーキットブレーカーはクローズ状態に戻ります。そうでない場合は、オープン状態に戻ります。

Java DSL の例:

```
from("direct:start").loadBalance()
    .circuitBreaker(2, 1000L, MyCustomException.class)
    .to("mock:result");
```

Spring XML の例:

```
<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

概要

Camel 2.18 から利用可能です。

Hystrix パターンにより、アプリケーションを Netflix Hystrix と統合することができます。これにより、Camel ルートでサーキットブレーカーを提供することができます。hystrix は、レイテンシーとフォールトトレランスのライブラリーで、以下の目的で設計されています

- リモートシステム、サービス、およびサードパーティーライブラリーへのアクセスポイントを分離
- 失敗の連鎖を止める
- 障害を避けられない複雑な分散システムでの耐障害性を実現

Maven を使用する場合は、Hystrix を使用するために以下の依存関係を **pom.xml** ファイルに追加します。

```
<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 の例

以下は、インラインのフォールバックルートにフォールバックすることで、hystrix エンドポイントが遅い処理から保護されることを示すルートの例になります。デフォルトでは、タイムアウトリクエストは **1000ms** であるため、HTTP エンドポイントは素早く応答する必要があります。

```
from("direct:start")
    .hystrix()
        .to("http://fooservice.com/slow")
    .onFallback()
        .transform().constant("Fallback message")
    .end()
    .to("mock:result");
```

XML 設定の例

以下は、XML を使用した場合の同じ例になります。

```
<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>
```

Hystrix フォールバック機能の使用

onFallback() メソッドは、メッセージを変換したり、Bean などをフォールバックとして呼び出すことができるローカル処理用のものです。ネットワーク経由で外部サービスを呼び出す必要がある場合は、独自のスレッドプールを使用する独立した **HystrixCommand** オブジェクトで実行される **onFallbackViaNetwork()** メソッドを使用する必要があります。これにより、最初のコマンドオブジェクトを使い切ることはありません。

Hystrix の設定例

Hystrix には、次のセクションに記載されているように、多くのオプションがあります。以下の例は、実行タイムアウトをデフォルトの 1 秒ではなく 5 秒に設定し、サーキットブレーカーが 5 秒 (デフォルト) ではなく 10 秒待機してから、状態がオープンへ遷移したときにリクエストを再度試行する Java

DSL を示しています。

```
from("direct:start")
    .hystrix()
        .hystrixConfiguration()
            .executionTimeoutInMilliseconds(5000).circuitBreakerSleepWindowInMilliseconds(10000)
        .end()
        .to("http://fooservice.com/slow")
    .onFallback()
        .transform().constant("Fallback message")
    .end()
    .to("mock:result");
```

以下は、XML を使用した場合の同じ例になります。

```
<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>
```

オプション

Ths Hystrix コンポーネントは以下のオプションをサポートします。Hystrix はデフォルト値を提供します。

名前	デフォルト値	型	説明
circuitBreakerEnabled	**true**	ブール値	サーキットブレーカーを使用して健全性を追跡し、トリップした場合にはショートサーキットリクエストを使用するかどうかを決定します。
circuitBreakerErrorThresholdPercentage	**50**	Integer	サーキットがトリップしてオープンになり、フォールバックロジックへのショートサーキットリクエストが開始されるエラーの割合 (パーセント) を設定します。
circuitBreakerForceClosed	**false**	ブール値	true の値を指定すると、強制的にサーキットブレーカーをクローズ状態にします。そのため、エラーの割合に関係なくリクエストは許可されます。
circuitBreakerForceOpen	**false**	ブール値	true の値を設定すると、サーキットブレーカーはすべてのリクエストを拒否するオープン (トリップ) 状態になります。
circuitBreakerRequestVolumeThreshold	**20**	Integer	サーキットをトリップさせるローリングウィンドウの最小リクエスト数を設定します。
circuitBreakerSleepWindownInMilliseconds	**5000**	Integer	サーキットがトリップした後、リクエストを拒否する時間を設定します。この時間が経過すると、サーキットを再度クローズするかどうかを判断するためのリクエストが試行されます。

名前	デフォルト値	型	説明
commandKey	Node ID	String	Hystrix コマンドを識別します。このオプションは設定できません。コマンドを一意にするのは常にノード ID です。
corePoolSize	**10**	Integer	コアスレッドプールのサイズを設定します。これは、同時実行可能な **HystrixCommand** オブジェクトの最大数です。
executionIsolationSemaphoreMaxConcurrentRequests	**10**	Integer	**ExecutionIsolationStrategy.SEMAPHORE** を使用する場合に、**HystrixCommand.run()** メソッドが実行できるリクエストの最大数を設定します。
executionIsolationStrategy	**THREAD**	String	**HystrixCommand.run()** と実行される分離ストラテジーを示します。**THREAD** は別のスレッドで実行され、同時リクエストはスレッドプールのスレッド数によって制限されます。**SEMAPHORE** は呼び出しスレッドで実行され、同時リクエストは semaphore の数によって制限されます。
executionIsolationThreadInterruptOnTimeout	**true**	ブール値	タイムアウトの発生時に **HystrixCommand.run()** の実行を中断するかどうかを示します。
executionTimeoutInMilliseconds	**1000**	Integer	実行完了までのタイムアウトをミリ秒単位で設定します。
executionTimeoutEnabled	**true**	ブール値	**HystrixCommand.run()** の実行のタイミングを調整する必要があるかどうかを示します。

名前	デフォルト値	型	説明
fallbackEnabled	**true**	ブール値	失敗または拒否が発生した場合に **HystrixCommand.getFallback()** への呼び出しを試行するかどうかを決定します。
fallbackIsolationSemaphoreMaxConcurrentRequests	**10**	Integer	**HystrixCommand.getFallback()** メソッドが呼び出しスレッドから実行できるリクエストの最大数を設定します。
groupKey	**CamelHystrix**	String	統計情報とサーキットブレーカー のプロパティーを関連付けるために使用される Hystrix グループを識別します。
keepAliveTime	**1**	Integer	keep-alive 時間 (分単位) を設定します。
maxQueueSize	**-1**	Integer	**BlockingQueue** 実装の最大キューサイズを設定します。
metricsHealthSnapshotIntervalInMilliseconds	**500**	Integer	Health Snapshot を取得できるようにする間隔 (ミリ秒単位) を設定します。Health Snapshot は、成功とエラーの割合を算出し、サーキットブレーカーのステータスに影響を与えます。
metricsRollingPercentileBucketSize	**100**	Integer	バケットごとに保持される実行回数の最大値を設定します。時間内により多くの実行が発生した場合はラップされ、バケットの先頭から上書きを開始します。

名前	デフォルト値	型	説明
metricsRollingPercentileEnabled	**true**	ブール値	実行レイテンシーを追跡するかどうかを示します。レイテンシーはパーセント値として計算されます。false を指定すると、サマリー統計 (平均値、パーセント) が -1 として返されます。
metricsRollingPercentileWindowBuckets	**6**	Integer	**rollingPercentile** ウィンドウを分割するバケット数を設定します。
metricsRollingPercentileWindowInMilliseconds	**60000**	Integer	パーセント計算をするために実行時間が保持されるローリングウィンドウの期間 (ミリ秒単位) を設定します。
metricsRollingStatisticalWindowBuckets	**10**	Integer	ローリング統計ウィンドウを分割するバケット数を設定します。
metricsRollingStatisticalWindowInMilliseconds	**10000**	Integer	このオプションと以下のオプションは、**HystrixCommand** および **HystrixObservableCommand** の実行からメトリックをキャプチャーする場合に適用されます。
queueSizeRejectionThreshold	**5**	Integer	拒否するためのしきい値をキューへ設定します。**maxQueueSize** に到達していない場合でも拒否できる人為的な最大キューサイズを設定します。
requestLogEnabled	**true**	ブール値	**HystrixCommand** の実行およびイベントを **HystrixRequestLog** に記録されるべきかどうかを示します。

名前	デフォルト値	型	説明
threadPoolKey	**null**	String	このコマンドを実行するスレッドプールを定義します。デフォルトでは、グループキーと同じキーを使用します。
threadPoolMetricsRollingStatisticalWindowBucket	**10**	Integer	ローリング統計ウィンドウを分割するバケット数を設定します。
threadPoolMetricsRollingStatisticalWindowInMilliseconds	**10000**	Integer	統計ローリングウィンドウの期間をミリ秒単位で設定します。スレッドプール用にメトリクスが保持される期間です。

8.12. SERVICE CALL

概要

Camel 2.18 から利用可能です。

Service Call パターンにより、分散システムでリモートサービスを呼び出すことができます。呼び出すサービスは、Kubernetes、Consul、etcd、Zookeeper などのサービスレジストリーで検索されます。このパターンは、サービスレジストリーの設定とサービスの呼び出しを分離します。

Maven ユーザーは、使用するサービスレジストリーの依存関係を追加する必要があります。可能性としては、以下のようなものがあります。

- **camel-consul**
- **camel-etcd**
- **camel-kubenetes**
- **camel-ribbon**

サービスを呼び出すための構文

サービスを呼び出すには、以下のようにサービス名を参照してください。

```
from("direct:start")
    .serviceCall("foo")
    .to("mock:result");
```

以下の例は、XML DSL でサービスを呼び出す例を示しています。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
```

```
    <serviceCall name="foo"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

これらの例では、Camel はサービスレジストリーと統合されたコンポーネントを使用して、**foo** という名前でサービスを検索します。ルックアップは、リモートサービスをホストするアクティブなサーバーの一覧を参照する **IP:PORT** ペアのセットを返します。Camel はリストから使用するサーバーをランダムに選択し、選択した **IP** および **PORT** 番号で Camel URI をビルドします。

デフォルトでは、Camel は HTTP コンポーネントを使用します。上記の例では、以下のように動的 **toD** エンドポイントから呼び出される Camel URI で呼び出しが解決されています。

```
toD("http://IP:PORT")
```

```
<toD uri="http:IP:port"/>
```

beer=yes のような URI パラメーターを使用してサービスを呼び出すことができます。

```
serviceCall("foo?beer=yes")
```

```
<serviceCall name="foo?beer=yes"/>
```

コンテキストパスを指定することもできます。例を以下に示します。

```
serviceCall("foo/beverage?beer=yes")
```

```
<serviceCall name="foo/beverage?beer=yes"/>
```

サービス名の URI への変換

説明のとおり、サービス名は Camel エンドポイント URI に解決されます。以下にいくつかの例を示します。→ は、Camel URI の解決策を示します。

```
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
```

解決された URI を完全に制御するには、希望の Camel URI を指定する追加の URI パラメーターを指定します。指定した URI で、**IP:PORT** に解決するサービス名を使用できます。以下に例を示します。

```
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
```

上記の例では、**myService** という名前のサービスを呼び出しています。2 番目のパラメーターは、解決した URI の値を制御します。最初の例では **serviceName.host** と **serviceName.port** を使い、IP または PORT のいずれかを参照している点に注意してください。**serviceName** のみを指定する場合、**IP:PORT** に解決されます。

サービスを呼び出すコンポーネントの設定

デフォルトでは、Camel は HTTP コンポーネントを使用してサービスを呼び出します。HTTP4 や Netty4 HTTP などの異なるコンポーネントを使用する場合、以下のように設定することができます。

```
KubernetesConfigurationDefinition config = new KubernetesConfigurationDefinition();
config.setComponent("netty4-http");

// Register the service call configuration:
context.setServiceCallConfiguration(config);

from("direct:start")
    .serviceCall("foo")
    .to("mock:result");
```

以下は、XML DSL の例になります。

```
&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>
```

すべての実装で共有されるオプション

各実装で以下のオプションを使用することができます。

オプション	デフォルト値	説明
clientProperty		使用するサービスコールの実装に固有のプロパティーを指定します。たとえば、ribbon 実装を使用している場合は、クライアントプロパティーが **com.netflix.client.config.CommonClientConfigKey** に定義されます。

component	**http**	リモートサービスを呼び出すために使用するデフォルトの Camel コンポーネントを設定します。netty4-http、jetty、restlet などのコンポーネントを設定できます。サービスが HTTP プロトコルを使用しない場合は、mqtt、jms、amqp などの別コンポーネントを使用する必要があります。サービスコールで URI パラメーターを指定すると、デフォルトではなく、このパラメーターで指定されたコンポーネントが使用されます。
loadBalancerRef		使用するカスタム **org.apache.camel.spi.ServiceCallLoadBalancer** への参照を設定します。
serverListStrategyRef		使用するカスタム **org.apache.camel.spi.ServiceCallServerListStrategy** への参照を設定します。

Kubernetes を使用する場合のサービスコールオプション

Kubernetes 実装では、以下のオプションがサポートされます。

オプション	デフォルト値	説明
apiVersion		クライアントルックアップを使用する場合の Kubernetes API バージョン。
caCertData		クライアントルックアップをする際に使用する認証局データを設定します。
caCertFile		クライアントルックアップをする際にファイルからロードされる認証局データを設定します。
clientCertData		クライアントルックアップをする際に使用するクライアント証明書データを設定します。
clientCertFile		クライアントルックアップをする際にファイルからロードされるクライアント証明書データを設定します。

clientKeyAlgo		クライアントルックアップをする際に使用する、RSA などのクライアントキーストアアルゴリズムを設定します。
clientKeyData		クライアントルックアップをする際に使用する、クライアントキーストアデータを設定します。
clientKeyFile		クライアントルックアップをする際にファイルからロードされるクライアントキーストアデータを設定します。
clientKeyPassphrase		クライアントルックアップをする際に使用する、クライアントキーストアパスフレーズを設定します。
dnsDomain		**dns** ルックアップに使用する DNS ドメインを設定します。
lookup	**environment**	サービスを検索するために使用されるストラテジーの選択。検索ストラテジーには以下が含まれます。 ● **environment**: 環境変数を使用します。 ● **dns**: DNS ドメイン名を使用します。 ● **client**: Java クライアントを使用して Kubernetes マスター API を呼び出し、サービスをアクティブにホストしているサーバーを照会します。
masterUrl		クライアントルックアップを使用する際の Kubernetes マスターの URL。
namespace		使用する Kubernetes の名前空間。デフォルトでは、namespace の名前は環境変数 **KUBERNETES_MASTER** から取得されます。

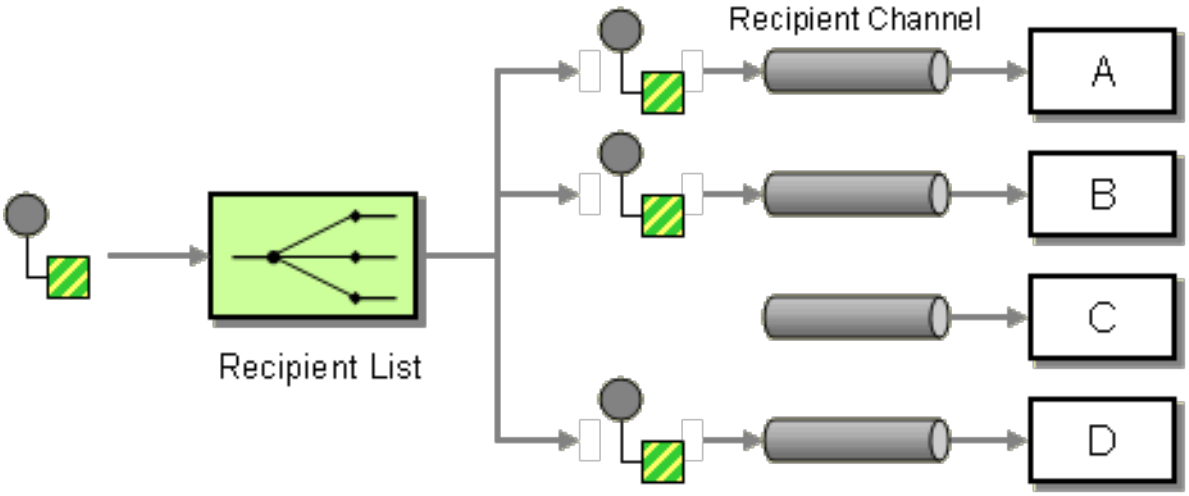
oauthToken		クライアントルックアップを使用する際に (ユーザー名/パスワードの代わりに) 認証用の OAUTH トークンを設定します。
password		クライアントルックアップを使用する際に使用する、認証パスワードを設定します。
trustCerts	false	クライアントルックアップを使用する際に使用する、トラスト証明書のチェックを有効にするかどうかを設定します。
username		クライアントルックアップを使用する際に使用する、認証ユーザー名を設定します。

8.13. MULTICAST

概要

図8.9「Multicast パターン」 に記載されている**Multicast** パターンは、**InOut** メッセージ交換パターンと互換性のある、宛先パターンが固定された Recipient list のバリエーションです。これは Recipient list とは対照的に、 **InOnly** 交換パターンとのみ互換性があります。

図8.9 Multicast パターン



カスタム集約ストラテジーを使用した Multicast

Multicast プロセッサーは、元のリクエストに対して複数の **Out** メッセージを受信しますが (各受信者から 1 つずつ)、呼び出し元は**1つ** のリプライを受け取るだけです。したがって、メッセージエクスチェンジのリプライの行程に固有のミスマッチがあり、この不一致を解消するためには、Multicast プロセッサーにカスタム **集約ストラテジー** を提供する必要があります。集約ストラテジークラスは、すべての **Out** メッセージを単一のリプライメッセージに集約します。

出品者が複数の入札者に販売商品を提供する電子オークションサービスの例について考えてみましょ

う。各入札者は商品に対して入札し、出品者が自動的に最高額の入札を選択します。以下のように、**multicast()** DSL コマンドを使用して、オファーを固定の入札者リストに配布するロジックを実装できます。

```
from("cxf:bean:offer").multicast(new HighestBidAggregationStrategy()).
    to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

seller はエンドポイント **cxf:bean:offer** で表され、利用者はエンドポイント **cxf:bean:Buyer1**、**cxf:bean:Buyer2**、**cxf:bean:Buyer3** によって表されます。様々な入札者からの入札を集約するために、Multicast プロセッサーは集約ストラテジー **HighestBidAggregationStrategy** を使用します以下のように、Java に **HighestBidAggregationStrategy** を実装することができます。

```
// 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;
    }
}
```

入札者は、**Bid** という名前のヘッダーに入札価格を設定することが前提となります。カスタム集約ストラテジーの詳細は、「Aggregator」 を参照してください。

並列処理

デフォルトでは、Multicast プロセッサーは、受信者のエンドポイントを逐次呼び出します (**to()** コマンドに記載されている順序で)。場合によっては、許容されないほどのレイテンシーが発生することがあります。このような待ち時間を回避するために、**parallelProcessing()** 句を追加して並行処理を有効にするオプションがあります。たとえば、電子オークションの例で並行処理を有効にするには、以下のようにルートを定義します。

```
from("cxf:bean:offer")
    .multicast(new HighestBidAggregationStrategy())
        .parallelProcessing()
        .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

Multicast プロセッサーは、各エンドポイントに 1 つのスレッドを割り当てるスレッドプールを使用して、入札者のエンドポイントを呼び出すようになりました。

入札者のエンドポイントを呼び出すスレッドプールのサイズをカスタマイズする場合は、**executorService()** メソッドを呼び出して独自のカスタム executor service を指定することができます。以下に例を示します。

```
from("cxf:bean:offer")
    .multicast(new HighestBidAggregationStrategy())
        .executorService(MyExecutor)
        .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

MyExecutor は、java.util.concurrent.ExecutorService 型のインスタンスです。

エクスチェンジに **InOUt** パターンがある場合、リプライメッセージを集約するために集約ストラテジーが使用されます。デフォルトの集約ストラテジーは、最新のリプライメッセージを取り、それ以前のリプライメッセージを破棄します。たとえば、以下のルートでは、カスタムストラテジー **MyAggregationStrategy** を使用してエンドポイント **direct:a**、**direct:b**、**direct:c** からのリプライを集約します。

```
from("direct:start")
  .multicast(new MyAggregationStrategy())
      .parallelProcessing()
      .timeout(500)
      .to("direct:a", "direct:b", "direct:c")
  .end()
  .to("mock:result");
```

XML 設定の例

以下の例は、XML で同様のルートを設定する方法を示しています。ルートは、カスタム集約ストラテジーとカスタム thread executor を使用しています。

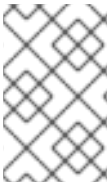
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       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>
```

parallelProcessing 属性と **threadPoolRef** 属性は任意です。Multicast プロセッサーのスレッド動作をカスタマイズする場合にのみ設定する必要があります。

送信メッセージへのカスタム処理の適用

Multicast パターン は、ソース Exchange をコピーして、そのコピーをマルチキャストします。デフォルトでは、ルーターはソースメッセージのシャローコピーを作成します。シャローコピーでは、元の

メッセージのヘッダーとペイロードは参照によってのみコピーされます。つまり、元のメッセージのコピーはそれらへリンクされます。マルチキャストメッセージのシャローコピーはリンクされているため、メッセージボディーが変更可能な場合は、カスタム処理を適用することができません。あるエンドポイントに送信されたコピーに適用するカスタム処理は、他のすべてのエンドポイントに送信されたコピーにも適用されます。



注記

multicast 構文では、**multicast** 句で **process** DSL コマンドを呼び出すことはできますが論理的に意味をなさず、**onPrepare** と同じ効果を**持ちません** (実際、このコンテキストでは **process** DSL コマンドは何の影響も与えません)。

メッセージの準備時にカスタムロジックを実行するための onPrepare の使用

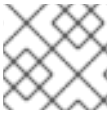
エンドポイントに送信する前に、カスタム処理を各メッセージレプリカに適用する場合は、**multicast** 句で **onPrepare** DSL コマンドを呼び出すことができます。この **onPrepare** コマンドは、メッセージがシャローコピーされた **直後**、かつメッセージがエンドポイントにディスパッチされる **直前** に、カスタムプロセッサーを挿入します。たとえば、以下のルートでは、**direct:a** に送信されたメッセージに対して **CustomProc** プロセッサーが呼び出され、**direct:b** に送信されたメッセージに対しても **CustomProc** プロセッサーが呼び出されます。

```
from("direct:start")
  .multicast().onPrepare(new CustomProc())
  .to("direct:a").to("direct:b");
```

onPrepare DSL コマンドの一般的なユースケースとして、メッセージの一部またはすべての要素のディープコピーを実行します。たとえば、以下の **CustomProc** プロセッサークラスは、メッセージボディーのディープコピーを実行します。メッセージボディーの型は **BodyType** であると想定され、ディープコピーはメソッド **BodyType.deepCopy()** によって実行されます。

```
// 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.
    }
}
```

onPrepare を使い、**Exchange** がマルチキャストされる前に実行するカスタムロジックを任意に実装することができます。



注記

イミュータブルなオブジェクトを設計することが推奨されます。

たとえば、この Animal クラスのようにミュータブルなメッセージボディーがあるとします。

```
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;
    }
}
```

次に、メッセージボディーをクローンするディープクローンプロセッサーを作成します。

```
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);
    }
}
```

次に、以下のように **onPrepare** オプションを使用して Multicast ルートで AnimalDeepClonePrepare クラスを使用します。

```
from("direct:start")
    .multicast().onPrepare(new AnimalDeepClonePrepare()).to("direct:a").to("direct:b");
```

XML DSL を使用した同じ例

```
<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"/>
```

オプション

multicast DSL コマンドは、以下のオプションをサポートします。

名前	デフォルト値	説明

strategyRef		AggregationStrategy の参照は、受信者からの複数のリプライを Multicast からの単一の送信メッセージへ集約するために使用されます。デフォルトでは、Camel は最後のリプライを送信メッセージとして使用します。
strategyMethodName		POJO を **AggregationStrategy** として使用している場合に、使用するメソッド名を明示的に指定するために使用できます。
strategyMethodAllowNull	**false**	POJO を **AggregationStrategy** として使用している場合に、このオプションを使用することができます。**false** に設定すると、エンリッチするデータがない場合に **aggregate** メソッドは使用されません。**true** に設定すると、エンリッチするデータがない場合には、**oldExchange** に **null** 値が使用されます。
parallelProcessing	**false**	有効にすると、マルチキャストへのメッセージの送信が並列処理されます。呼び出し元スレッドは、すべてのメッセージが完全に処理されるまで待機してから続行することに注意してください。マルチキャストからの送信およびリプライ処理のみが並列に処理されます。
parallelAggregate	**false**	有効にすると、**AggregationStrategy** の **aggregate** メソッドを同時に呼び出すことができます。これには、**AggregationStrategy** の実装がスレッドセーフである必要があることに注意してください。デフォルトでは、このオプションは **false** となっており、Camel が自動的に **aggregate** メソッドへの呼び出しを同期することを意味します。ただし、場合によっては、**AggregationStrategy** をスレッドセーフとして実装し、このオプションを **true** に設定することで、パフォーマンスを向上させることができます。

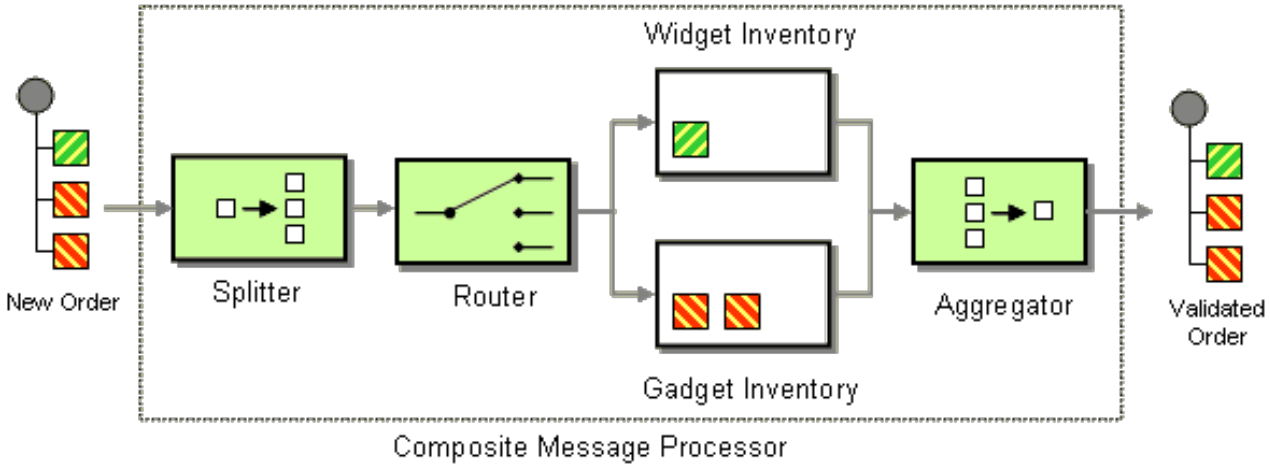
executorServiceRef		並列処理に使用するカスタムスレッドプールを参照します。このオプションを設定すると、並列処理は自動的に適用されるため、 並列処理用オプションも有効にする必要はありません。
stopOnException	**false**	**Camel 2.2:** 例外発生時、すぐに継続処理を停止するかどうか。無効にすると、いずれかの失敗の有無に関わらず、Camel はメッセージをすべてのマルチキャストに送信します。AggregationStragegy クラス内で、例外処理を完全に制御することができます。
streaming	**false**	有効な場合、Camel はリプライを順不同で処理します。たとえば、返信順に処理します。無効な場合、Camel はマルチキャストと同じ順序でリプライを処理します。
timeout		**camel 2.5:** 合計タイムアウト値をミリ秒単位で設定します。Multicast が指定の時間枠内のすべての応答を送信および処理できない場合、タイムアウトが発生し、 Multicast から抜け出し続行します。TimeoutAwareAggregationStrategy を提供する場合、**timeout** メソッドが抜け出す前に呼び出されるため、注意してください。
onPrepareRef		**camel 2.8:** カスタムプロセッサーを参照して、各マルチキャストが受信する Exchange のコピーを準備します。これにより、必要に応じてメッセージのペイロードをディープクローンするなど、カスタムロジックを実行できます。
shareUnitOfWork	**false**	**Camel 2.8:** Unit of Work を共有すべきかどうか。詳細は、「Splitter」 で同じオプションを参照してください。

8.14. COMPOSED MESSAGE PROCESSOR

Composed Message Processor

図8.10「Composed Message Processor パターン」 に記載されているように、 **Composed Message Processor** パターンでは、メッセージを分割し、そのサブメッセージをそれぞれ適切な宛先にルーティングし、そしてレスポンスを再度単一のメッセージに集約し直すといったやり方で、合成メッセージを処理できます。

図8.10 Composed Message Processor パターン



Java DSL の例

以下の例では、複数パーツからなる注文に応じることができるかをチェックしています。ここでは、注文の各パーツでそれぞれ異なる在庫に対してチェックを行う必要があります。

```
// 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 の例

上記のルートは、以下のように XML DSL で記述することもできます。

```
<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>
```

処理のステップ

処理は、「Splitter」 を使用して注文を分割することから始まります。「Splitter」 は続いて、個別の **OrderItems** を 「Content-Based Router」 に送信し、項目種別に基づいてメッセージをルーティングします。**ウィジェット** 項目はチェックのために **widgetInventory** Bean に送信され、**ガジェット** 項目は **gadgetInventory** Bean に送信されます。**OrderItems** はそれぞれ適切な Bean によって検証されると、「Aggregator」 に送信され、検証済みの **OrderItems** を収集して 1 つの注文に再び構築し直します。

受信した注文はそれぞれ、**注文 ID** を含んだヘッダーを持ちます。注文 ID は集約のステップで利用されます。**aggregate()** DSL コマンドの **.header("orderId")** 修飾子は、アグリゲーターがキー **orderId** のヘッダーを相関式として使用するように指示します。

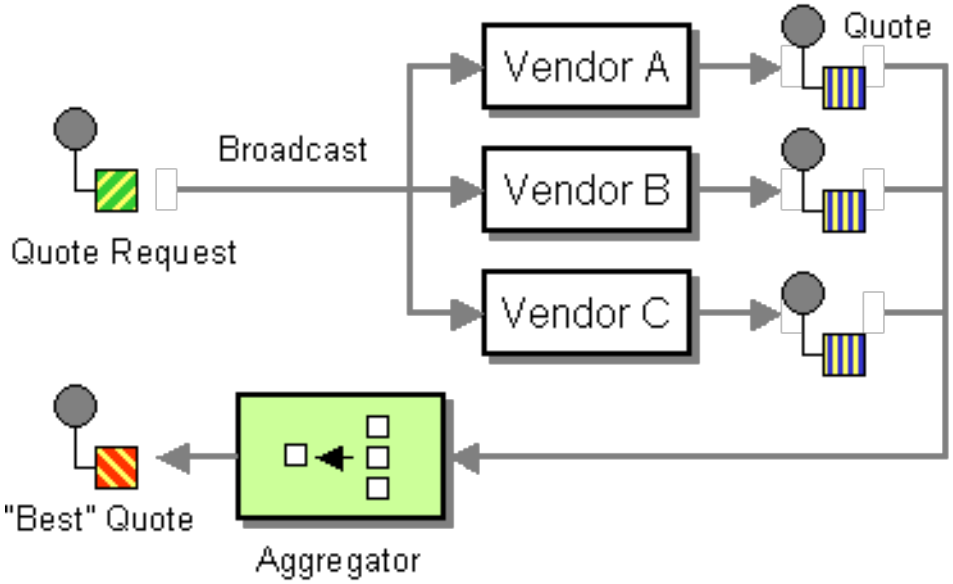
詳細は、camel-core/src/test/java/org/apache/camel/processor にある **ComposedMessageProcessorTest.java** サンプルソースを確認してください。

8.15. SCATTER-GATHER

Scatter-Gather

図8.11「Scatter-Gather パターン」 に記載されているように、**Scatter-Gather パターン** を使用すると、メッセージを動的に指定された複数の受信者にルーティングし、そのレスポンスを単一のメッセージに再集約できます。

図8.11 Scatter-Gather パターン



動的なスキャッター/ギャザーの例

以下の例は、複数の異なるベンダーから最も良いビールの見積もりを取得するアプリケーションの概要を説明しています。この例では、動的な 「受信者リスト」 を使用してすべてのベンダーに見積もりを要求し、 「Aggregator」 を使用してすべてのレスポンスの中から最良の見積もりを選別します。このアプリケーションのルートは、以下のように定義されます。

```
<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>
```

最初のルートでは、 「受信者リスト」 は **listOfVendors** ヘッダーを確認して受信者リストを取得します。したがって、このアプリケーションにメッセージを送信するクライアントは **listOfVendors** ヘッダーをメッセージに追加する必要があります。例8.1「メッセージングクライアントの例」 は、該当のヘッダーデータを送信メッセージに追加するメッセージングクライアントのコードの一例を示しています。

例8.1 メッセージングクライアントの例

```
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);
```

メッセージは **bean:vendor1**、**bean:vendor2**、および **bean:vendor3** のエンドポイントに分散されます。これらの Bean はすべて以下のクラスによって実装されます。

```
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 インスタンス、**vendor1**、**vendor2**、および **vendor3** は、以下のように Spring XML 構文を使用してインスタンス化されます。

```
<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>
```

各 Bean は、それぞれ異なるビール価格で初期化されます (コンストラクター引数に渡されます)。メッセージが各 Bean エンドポイントに送信されると、**MyVendor.getQuote** メソッドに到達します。このメソッドは、この見積もり要求がビールに対してであるかどうかを確認する簡単なチェックを実行し、

それから後のステップで取得できるようにエクスチェンジにビールの価格を設定します。メッセージは POJO 生成 を使用して次のステップに転送されます (@Produce アノテーションを参照)。

次のステップでは、すべてのベンダーからのビールの見積もりを受け取り、どのベンダーの見積もりが最良か (つまり最も低いか) を調べます。そのため、「Aggregator」 をカスタムの集約ストラテジーと共に使用します。「Aggregator」 は、どのメッセージが現在の見積もりに関連するものかを識別する必要があります。これは、**quoteRequestId** ヘッダー (**correlationExpression** に渡された) の値に基づいてメッセージを関連付けることによって行われます。例8.1「メッセージングクライアントの例」 にあるように、相関 ID は **quoteRequest-1** に設定されています (相関 ID は一意である必要があります)。見積もりの集合の中から最も低いものを選別するには、以下のようなカスタム集約ストラテジーを使用します。

```
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;
        }
    }
}
```

静的なスキャッター/ギャザーの例

静的な 「受信者リスト」 を使用することで、スキャッター/ギャザーアプリケーションの中で受信者を明示的に指定することができます。以下の例は、静的なスキャッター/ギャザーのシナリオを実装するために使用するルートを示しています。

```
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 パターンにより、メッセージを複数回処理できます。これは主にテストで使用されます。

デフォルトでは、ループ全体で同じエクスチェンジが使用されます。上記は、次の反復に使用されます (「パイプとフィルター」 を参照)。**Camel 2.8** から、代わりにコピーモードを有効にすることができます。詳細は、オプションの表を参照してください。

エクスチェンジプロパティー

ループ反復ごとに 2 つのエクスチェンジプロパティーが設定され、任意でループに含まれるプロセッサーで読み込むことができます。

プロパティー	説明
CamelLoopSize	Apache Camel 2.0: ループの総数
CamelLoopIndex	Apache Camel 2.0: 現在のイテレーションのインデックス (0 ベース)

Java DSL の例

以下の例は、**direct:x** エンドポイントからリクエストを取得して、そのメッセージを **mock:result** に繰り返し送信する方法を示しています。ループの反復回数は、**loop()** への引数として指定するか、実行時に式を評価することで指定します。評価する場合は、式が **int** として評価される**必要があります** (そうでない場合は **RuntimeCamelException** が出力されます)。

次の例では、ループ回数を定数として渡しています。

```
from("direct:a").loop(8).to("mock:result");
```

次の例では、ループ数を決定するために単純な式を評価しています。

```
from("direct:b").loop(header("loop")).to("mock:result");
```

次の例では、XPath 式を評価してループ数を決定します。

```
from("direct:c").loop().xpath("/hello/@times").to("mock:result");
```

XML 設定の例

Spring XML でも同じルートを設定することができます。

次の例では、ループ回数を定数として渡しています。

```
<route>
  <from uri="direct:a"/>
  <loop>
    <constant>8</constant>
    <to uri="mock:result"/>
  </loop>
</route>
```

次の例では、ループ数を決定するために単純な式を評価しています。

```
<route>
  <from uri="direct:b"/>
  <loop>
    <header>loop</header>
```

```
    <to uri="mock:result"/>
  </loop>
</route>
```

コピーモードの使用

ここで、**direct:start** エンドポイントに A の文字を含むメッセージを送信したとします。このルートの処理の出力は、各 **mock:loop** エンドポイントがメッセージとして AB を受信することになります。

```
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");
```

しかし、コピーモードを有効に**しない**と、**mock:loop** は AB、ABB、および ABBB のメッセージを受信します。

```
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");
```

コピーモードの XL DSL での同様な例は以下のとおり

```
<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>
```

オプション

loop DSL コマンドは、以下のオプションをサポートします。

名前	デフォルト値	説明

copy	**false**	Camel 2.8: コピーモードを使用するかどうか。**false** の場合は、ループ中に同じエクスチェンジが使用されます。そのため、前のイテレーションの結果は次のイテレーションでも**表示されます**。その代わりに、コピーモードを有効にすると、入力「エクスチェンジ」の新規コピーで各反復が**再開されます**。

Do While ループ

do while ループを使って、条件が満たされるまでループを実行することができます。条件は true または false のいずれかになります。

DSL の場合、コマンドは **LoopDoWhile** になります。以下の例では、メッセージ本文の長さが 5 文字以下になるまでループを実行します。

```
from("direct:start")
    .loopDoWhile(simple("${body.length} <= 5"))
        .to("mock:loop")
        .transform(body().append("A"))
    .end()
    .to("mock:result");
```

XML の場合、コマンドは **loop doWhile** になります。以下の例では、メッセージ本文の長さが 5 文字以下になるまでループを実行します。

```
<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

サンプリングスロットラー

サンプリングスロットラーを使用すると、ルート上のトラフィックからエクスチェンジのサンプルを抽出することができます。**単一の**エクスチェンジのみを通過させるサンプリング期間で設定されています。それ以外のエクスチェンジはすべて停止します。

デフォルトでは、サンプル期間は 1 秒です。

Java DSL の例

sample() DSL コマンドを使用して、以下のようにサンプラーを起動します。

```
// 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 の例

Spring XML では、sample 要素を使用してサンプラーを呼び出します。ここには **samplePeriod** および **units** 属性を使用してサンプリング期間を指定するオプションがあります。

```
<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>
```

オプション

sample DSL コマンドは、以下のオプションをサポートします。

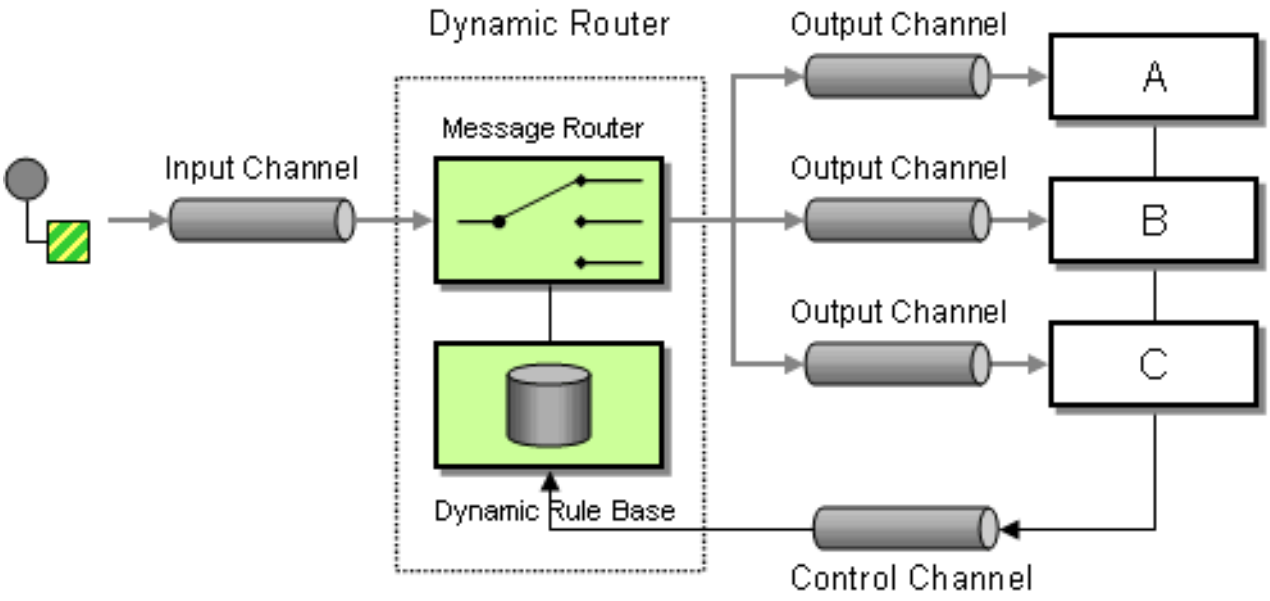
名前	デフォルト値	説明
messageFrequency		N 番目のメッセージごとにメッセージをサンプリングします。頻度または周期のいずれかを使用できます。
samplePeriod	**1**	N 周期ごとにメッセージをサンプリングします。頻度または周期のいずれかを使用できます。
units	**SECOND**	JDK の **java.util.concurrent.TimeUnit** の列挙としての時間単位。

8.18. DYNAMIC ROUTER

Dynamic Router

図8.12「Dynamic Router パターン」 に示すように、Dynamic Router パターンを使用すると、設計時にはステップの順序がわからない一連の処理ステップを介して、メッセージを連続してルーティングすることができます。メッセージが通過するエンドポイントのリストは、**実行時に**動的に計算されます。メッセージがエンドポイントから戻るたびに、動的ルーターはルート内の次のエンドポイントを発見するために Bean にコールバックします。

図8.12 Dynamic Router パターン



Camel 2.5では、DSL に **dynamicRouter** が導入されました。これは、**その場で** slip を評価する動的な「Routing Slip」 のようなものです。

注意事項

dynamicRouter (Bean など) に使用される式が **null** を返して終了を示すようにしなければなりません。そうしないと、**dynamicRouter** は無限ループで続行されます。

Camel 2.5 以降の Dynamic Router

Camel 2.5 以降では、「Dynamic Router」 は、slip を通過する際にエクスチェンジプロパティー **Exchange.SLIP_ENDPOINT** を現在のエンドポイントで更新します。これにより、エクスチェンジが slip 経由でどこまで進んでいるかを調べることができます。(slip としているのは、「Dynamic Router」の実装が 「Routing Slip」 をベースとしているためです)。

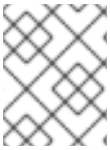
Java DSL

Java DSL では、以下のように **dynamicRouter** を使用できます。

```
from("direct:start")
    // use a bean as the dynamic router
    .dynamicRouter(bean(DynamicRouterTest.class, "slip"));
```

Bean の統合を利用して**その場**で slip を計算していますが、これは以下のように実装することができます。

```
// 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;
    }
```



注記

上記の例はスレッドセーフでは**ありません**。スレッドの安全性を確保するために、**Exchange** に状態を格納する必要があります。

Spring XML

Spring XML での同じ例は次のとおりです。

```
<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>
```

オプション

dynamicRouter DSL コマンドは以下のオプションをサポートします。

名前	デフォルト値	説明
uriDelimiter	,	パートII「ルーティング式と述語言語」 によって返された複数のエンドポイントに使用される区切り文字。
ignoreInvalidEndpoints	**false**	エンドポイント URI を解決できなかった場合は、無視されます。false の場合は、Camel はエンドポイント URI が有効ではないことを示す例外を出力します。

@DYNAMICROUTER アノテーション

@DynamicRouter アノテーションを使用することもできます。以下に例を示します。

```
// 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.
    }
}
```

route メソッドは、メッセージが処理される際に繰り返し呼び出されます。次の宛先のエンドポイント URI を返すために行われます。**null** を返して終了を示します。「Routing Slip」 のように複数のエンドポイントを返すことができ、各エンドポイントは区切り文字で区切られます。

第9章 SAGA EIP

9.1. 概要

Saga EIP は、Camel ルートの中で互いに関連のある一連のアクションに対して、それらが正常に完了したか、実行されないか、補正されるかのいずれかとなるように定義する方法を提供します。Saga の実装は、あらゆるトランスポートを使用して通信する分散サービスが、グローバルに一貫性のある結果を得られるように調整します。Saga EIP は、従来の ACID 分散 (XA) トランザクションとは異なります。さまざまな参加サービスのステータスが Saga の最後にのみ一貫性を持つことが保証され、中間ステップにおいては保証されないためです。

Saga EIP は、分散トランザクションの使用が推奨されないユースケースに適しています。たとえば、Saga に参加するサービスは、古典的なデータベースから NoSQL 非トランザクションデータストアまで、あらゆる種類のデータストアを使用できます。また、ステートレスなクラウドサービスではサービスに伴ってトランザクションログを保存する必要がないため、Saga に参加するサービスはこうしたサービスの中での使用にも適しています。Saga EIP はさらに、トランザクションとは異なりデータベースレベルのロックを使用しないため、短い時間で処理を完了する必要がありません。そのため、数秒から数日間まで、長い期間にわたって存続することができます。

Saga EIP はデータに対するロックを使用しません。代わりに、標準フローでエラーが発生した場合に、フロー実行前のステータスを復元する目的で実行されるべきアクションである、補正アクション (Compensating Action) の概念を定義しています。補正アクションは Camel ルートの中で Java または XML DSL を使用して宣言でき、必要な場合 (エラーによって Saga がキャンセルされた場合) にのみ Camel から呼び出されます。

9.2. SAGA EIP のオプション

Saga EIP は、以下の 6 つのオプションをサポートします。

名前	説明	デフォルト	型
propagation	Saga の伝搬モード (REQUIRED、REQUIRES_NEW、MANDATORY、SUPPORTS、NOT_SUPPORTED、NEVER) を設定します。	必須	SagaPropagation
completionMode	Saga が完了したことをどうやって判断するかを決定します。**AUTO** に設定すると、Saga を開始したエクスチェンジが正常に処理されると Saga は完了し、異常終了すると補正が行われます。**MANUAL** に設定すると、**saga:complete** または **saga:compensate** エンドポイントを使用してユーザーが Saga を完了または補正する必要があります。	AUTO	SagaCompletionMode
timeoutInMilliseconds	Saga の最大時間上限を設定します。タイムアウトの期限が切れると、Saga は自動的に補正されます (その間に異なる決定が行われていない限り)。		Long

名前	説明	デフォルト	型
compensation	ルート内で実行されたすべての変更を補正するために呼び出される必要のある、補正エンドポイント URI。補正 URI に対応するルートは、補正を実行し、エラーを起こさずに完了する必要があります。補正中にエラーが発生した場合、Saga サービスは再度補正 URI を呼び出して再試行します。		SagaActionUriDefinition
completion	Saga が正常に完了したときに呼び出される完了エンドポイント URI。完了 URI に対応するルートは、完了タスクを実行し、エラーを起こさずに終了する必要があります。完了中にエラーが発生した場合、Saga サービスは完了 URI を再度呼び出して再試行します。		SagaActionUriDefinition
option	現在のエクスチェンジのプロパティーを保存して、補正または完了のコールバックルートで再利用できるようにします。オプションは通常、補正アクションで削除されたオブジェクトの識別子を保存しておいて取得する場合などに役立ちます。オプションの値は、補正/完了エクスチェンジの入力ヘッダーに変換されます。		List

9.3. SAGA サービスの設定

Saga EIP では、インターフェイス **org.apache.camel.saga.CamelSagaService** を実装したサービスを Camel コンテキストに追加する必要があります。Camel は、現在以下の Saga サービスをサポートしています。

- **InMemorySagaService**: これは Saga EIP の **基本** 実装で、高度な機能をサポートしません (リモートコンテキストの伝搬に非対応、アプリケーション障害時の一貫性は非保証)。

9.3.1. インメモリー Saga サービスの使用

インメモリー Saga サービスは本番環境では推奨されません。Saga ステータスの永続性をサポートせず (インメモリーにのみ保持される)、アプリケーション障害が発生した場合 (JVM クラッシュなど) に Saga EIP の一貫性を保証できないためです。また、インメモリー Saga サービスを使用する場合は、トランスポートレベルのヘッダーを使用して Saga コンテキストをリモートサービスに伝播することができません (他の実装では可能です)。インメモリーの Saga サービスを使用するには、以下のコードを追加して Camel コンテキストをカスタマイズします。このサービスは **camel-core** モジュールに属しています。

```
context.addService(new org.apache.camel.impl.saga.InMemorySagaService());
```

9.4. 例

たとえば、新規注文を行うとします。システムには異なる 2 つのサービスである、注文を管理するサー

ビスとクレジット (残高) を管理するサービスがあるとします。ロジックとしては、十分なクレジット (残高) がある場合には注文を行うことができます。Saga EIP を使用すると、**direct:buy** ルートを 2 つの異なるアクションで設定される Saga としてモデル化できます。1 つ目は注文の作成、もう 1 つ目はクレジット (残高) の確保です。**どちらのアクションも実行されるか、またはどちらも実行されないかのいずれかである必要があります。**クレジット (残高) がないのに注文が行われるのは、不整合な結果と見なされるからです (注文がないのに支払いが行われるのも同様です)。

```
from("direct:buy")
  .saga()
    .to("direct:newOrder")
    .to("direct:reserveCredit");
```

この Buy アクションについては、これ以降の例の中で変更はありません。New Order および Reserve Credit アクションをモデリングする上で使用されるオプションは以下のとおりです。

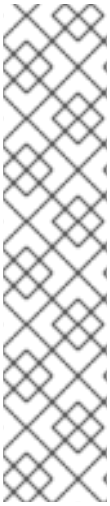
```
from("direct:newOrder")
  .saga()
  .propagation(SagaPropagation.MANDATORY)
  .compensation("direct:cancelOrder")
    .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
    .bean(orderManagerService, "newOrder")
    .log("Order ${body} created");
```

ここでは、伝搬モードは **MANDATORY** に設定されています。これは、このルートを流れるエクスチェンジはすでに Saga の一部でなければならないことを意味します (この例では、**direct:buy** ルートで Saga が作成されているため条件を満たします)。**direct:newOrder** ルートは **direct:cancelOrder** と呼ばれる補正アクションを宣言しており、Saga がキャンセルされた場合に注文の取り消しを行います。

各エクスチェンジには常に **Exchange.SAGA_LONG_RUNNING_ACTION** ヘッダーが含まれ、ここでは注文の ID として使用されます。この ID は対応する補正アクション内で削除すべき注文を特定するのに使われますが、必須ではありません (オプションで代替ソリューションを使用できます)。**direct:newOrder** の補正アクションは **direct:cancelOrder** で、以下のようになります。

```
from("direct:cancelOrder")
  .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
  .bean(orderManagerService, "cancelOrder")
  .log("Order ${body} cancelled");
```

このアクションは、注文を取り消す必要があるときに Saga EIP 実装によって自動的に呼び出されます。このアクションはエラーで終了することはありません。**direct:cancelOrder** ルートでエラーが出力された場合、EIP 実装は一定の上限回数まで補正アクションの実行を定期的に再試行します。つまり、**補正アクションはべき等でなければならない**ことを意味します。アクションが複数回トリガーされる可能性を考慮に入れる必要があり、またどのような場合でも失敗しないようにする必要があります。再試行回数の上限に達しても補正アクションが終わらなかった場合には、手動による介入プロセスを Saga 実装からトリガーする必要があります。



注記

direct:newOrder ルートの実行に遅延が生じたために、その間に他の参加者によって Saga がキャンセルされることがあります (並列実行中のルートでのエラーや、Saga レベルでのタイムアウトなど)。そのため、補正アクション **direct:cancelOrder** が呼び出されたときには、キャンセルされた Order レコードが見つからないことがあります。完全にグローバルな一貫性を保証するためには、**主となるアクションとそれに対応した補正アクションが可換である**ことが重要です。たとえば、もし補正が主となるアクションより前に実行されたとしても、同じ結果となる必要があります。

もう 1 つの取りうるアプローチは、振る舞いを可換にするのが不可能な場合に、主となるアクションで生成されるデータが見つかるまで (または最大再試行回数に到達するまで) 補正アクションの中で失敗し続けるようにすることです。このアプローチは多くの状況で機能するかもしれませんが、**ヒューリスティック**になります。

クレジット (残高) サービスは、注文サービスとほぼ同様に実装されます。

```
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}");
```

補正アクションは以下のとおりです。

```
from("direct:refundCredit")
  .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
  .bean(creditService, "refundCredit")
  .log("Credit for action ${body} refunded");
```

ここでは、クレジットの予約に対する補正アクションは予約解除 (refund) です。

9.4.1. 完了イベントの取り扱い

Saga の完了時には、何らかの処理が必要となります。何らかの問題が発生して Saga がキャンセルされた場合には、補正エンドポイントが呼び出されます。Saga が正常に完了した場合は、**完了エンドポイント** を呼び出して追加の処理を行うことができます。たとえば、上記の注文サービスでは、実際に注文の準備を開始するために、注文がいつ完了したか (そしてクレジット (残高) が予約されたか) を知る必要があることがあります。支払いが完了していないのに、注文の準備を開始したくはありません (最新の CPU のように、読み込み権限を確認する前に予約メモリーへのアクセス権を与えてしまうのとは異なります)。これは、**direct:newOrder** エンドポイントを変更すると簡単に実現できます。

1. 完了エンドポイントを呼び出します。

```
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** は直前の例と同じです。正常に完了した場合は以下のとおりです。

```
from("direct:completeOrder")
  .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
  .bean(orderManagerService, "findExternalId")
  .to("jms:prepareOrder")
  .log("Order ${body} sent for preparation");
```

Saga が完了すると、注文は準備のために JMS キューに送信されます。補正アクションと同様、完了アクションも Saga のコーディネーターによって複数回呼び出される可能性があります (特にネットワークエラーなどのエラーが発生した場合) 。この例では、 **prepareOrder** JMS キューをリッスンするサービスが重複を受け取る可能性について備えています (重複をどのように扱うかについての例は、Idempotent Consumer EIP を参照してください)。

9.4.2. カスタム識別子とオプションの使用

Saga のいくつかのオプションを使用してカスタム識別子を登録することができます。たとえば、クレジット (残高) サービスは以下のようにリファクタリングされます。

1. 以下のように、カスタム ID を生成してボディーに設定します。

```
from("direct:reserveCredit")
  .bean(idService, "generateCustomId")
  .to("direct:creditReservation")
```

1. アクションを委譲し、現在のボディーを補正アクションに必要なものとしてマークします。

```
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. Saga がキャンセルされた場合のみ、ヘッダーから CreditId オプションを取得します。

```
from("direct:creditRefund")
  .transform(header("CreditId")) // retrieve the CreditId option from headers
  .bean(creditService, "refundCredit")
  .log("Credit for Custom Id ${body} refunded");
```

direct:creditReservation エンドポイントは、伝搬モードを **SUPPORTS** に設定することで Saga の外から呼び出すことができます。このようにして、複数のオプションを 1 つの Saga ルートの中で宣言できます。

9.4.3. タイムアウトの設定

Saga EIP でタイムアウトを設定することで、マシン障害の発生時に Saga が永久に停止したままにならないことが保証されます。Saga EIP の実装では、明示的にタイムアウトを指定していないすべての Saga EIP に対してデフォルトのタイムアウトが設定されます。タイムアウトの期限が切れると、Saga EIP はそれ以前に異なる決定がなされていない限り、**Saga のキャンセル** (およびすべての参加者への補正) を決定します。

タイムアウトは、Saga の参加者に対して以下のように設定できます。

```
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");
```

すべての参加者 (クレジット (残高) サービス、注文サービスなど) は、それぞれ独自のタイムアウトを設定できます。これらの参加者が 1 つに設定されたときに、それらのタイムアウトの中の最小値が Saga のタイムアウトとなります。タイムアウトは、以下のように Saga レベルで指定することもできます。

```
from("direct:buy")
  .saga()
  .timeout(5, TimeUnit.MINUTES) // timeout at saga level
    .to("direct:newOrder")
    .to("direct:reserveCredit");
```

9.4.4. 伝播方法の選択

上記の例では、**MANDATORY** および **SUPPORTS** の伝搬モードを使用していますが、他に何も指定されなかった場合に使用されるデフォルトの伝播モードである **REQUIRED** もあります。これらの伝搬モードは、トランザクションの文脈で使用される同等のモードと 1 対 1 に対応します。

伝搬方法	説明
REQUIRED	既存の Saga に参加するか、または存在しない場合は新しい Saga を作成します。
REQUIRES_NEW	常に新しい Saga を作成します。古い Saga は一時停止し、新しい Saga が終了したときに再開します。
MANDATORY	すでに Saga が存在している必要があります。既存の Saga に参加します。
SUPPORTS	Saga がすでに存在している場合は参加します。
NOT_SUPPORTED	Saga がすでに存在している場合は一時停止し、現在のブロックが完了したときに再開します。
NEVER	現在のブロックを Saga 内で呼び出すことはできません。

9.4.5. 手動完了の使用 (高度な設定)

Saga をすべて同期的に実行できず、たとえば非同期通信チャネルを使用した外部サービスとの通信などが必要となる場合には、完了モードを **AUTO** (デフォルト) には設定できません。これは、Saga を作成したエクスチェンジが完了した時点ではその Saga は完了していないためです。実行期間が長い (数時間、数日) Saga EIP では、このようなことがよくあります。このような場合には、**MANUAL** 完了モードを使用する必要があります。

```
from("direct:mysaga")
  .saga()
  .completionMode(SagaCompletionMode.MANUAL)
  .completion("direct:finalize")
  .timeout(2, TimeUnit.HOURS)
    .to("seda:newOrder")
    .to("seda:reserveCredit");
```

seda:newOrder および seda:reserveCredit に非同期の処理を追加します。これらは seda:operationCompleted に非同期のコールバックを送信します。

```
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()
```

direct:finalize エンドポイントを追加することで、最終のアクションを実行できます。

完了モードを **MANUAL** に設定すると、ルート **direct:mysaga** でエクスチェンジが処理されても Saga が完了せずに持続します (最大持続時間は 2 時間に設定されます)。非同期アクションが両方とも完了することで、Saga が完了します。完了の呼び出しは、Camel Saga コンポーネントの saga **:complete** エンドポイントを使用して行われます。手動で Saga を補正するための同様のエンドポイントがあります (**saga:compensate**) 。

9.5. XML の設定

Saga の機能は、XML の設定を利用するユーザーにも使用できます。以下のスニペットに例を示します。

```
<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

概要

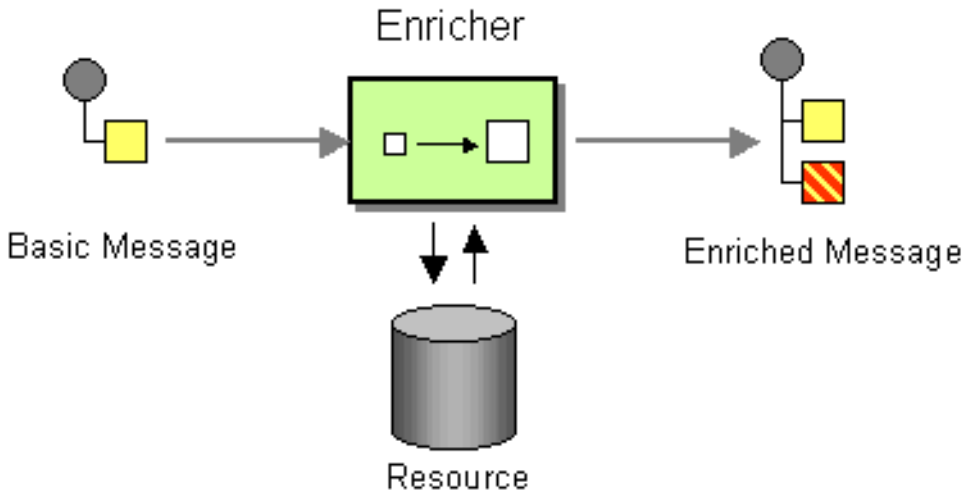
Message Transformation パターンは、さまざまな目的のためにメッセージの内容を変更する方法を表しています。

10.1. CONTENT ENRICHER

概要

Content Enricher パターンは、メッセージの宛先が元のメッセージに存在する以上のデータを必要とするシナリオを表しています。この場合、メッセージ変換、ルーティングロジック内の任意のプロセッサー、または Content Enricher メソッドを使用して外部リソースから追加データをプルします。

図10.1 Content Enricher パターン



コンテンツ補完の代替方法

Apache Camel は、コンテンツを補完するための複数の方法をサポートします。

- ルーティングロジックの任意のプロセッサーを使用したメッセージ変換
- **enrich()** メソッドは、現在のエクスチェンジのコピーを **プロデューサー** エンドポイントに送信し、生成された応答のデータを使用して、リソースから追加のデータを取得します。Enricher によって作成されるエクスチェンジは、常に **InOut** エクスチェンジです。
- **pollEnrich()** メソッドは、データの **コンシューマー** エンドポイントをポーリングして追加のデータを取得します。実質的に、メインルートからのコンシューマーエンドポイントと **pollEnrich()** 操作中のコンシューマーエンドポイントは結合されます。つまり、ルートの初期コンシューマーの受信メッセージが、ポーリングするコンシューマーの **pollEnrich()** をトリガーします。

注記

enrich() および **pollEnrich()** メソッドは動的エンドポイント URI をサポートします。URI を取得するには、現在のエクスチェンジから値を取得できる式を指定します。たとえば、データエクスチェンジから計算される名前でファイルをポーリングできます。この動作は Camel 2.16 で導入されました。この変更により XML DSL を使わず、容易に移行ができます。Java DSL は後方互換性を維持します。

メッセージ変換およびプロセッサーを使用したコンテンツの補完

Camel は、IDE 上でタイプセーフなコード補完をしながら、ルーティングおよび仲介ルールを作成できる Fluent Builder (流れるようなビルダー) を提供します。これにより、スマートな補完を提供し、リファクタリングを安全に行うことができます。分散システムをテストする場合、特定システムが利用可能または書き込みされるまで、システムの他の部分をテストするために、システムの特定部分のスタブを作成しなければならないような要件が一般的です。これを実行する 1 つの方法として、何らかの テンプレート システムを使用して、ほとんどの静的なボディーを持つ動的メッセージを生成することで、要求への応答を生成します。テンプレートを使用する別の方法として、ある宛先からメッセージを受信し、Velocity、XQuery などを使って変換してから、別の宛先に送信する方法もあります。以下の例は、**InOnly** (一方向) メッセージに対する例になります。

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

InOut (request-reply) メッセージングを使用して ActiveMQ の **My.Queue** キューでリクエストを処理するとします。**JMSReplyTo** の宛先に送信されるテンプレートが生成した応答が必要です。以下の例は、これらを行う方法を示しています。

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

以下の例は、DSL を使用してメッセージのボディーを変換する方法を示しています。

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

以下の例は、明示的な Java コードを使用してプロセッサーを追加します。

```
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");
```

次の例では、変換機能として機能する Bean を有効化するために Bean 統合を使用しています。

```
from("activemq:My.Queue").
  beanRef("myBeanName", "myMethodName").
  to("activemq:Another.Queue");
```

以下の例は Spring XML 実装を示しています。

```
<route>
```

```
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>/>
```

Enrich() メソッドを使用したコンテンツの補完

```
AggregationStrategy aggregationStrategy = ...

from("direct:start")
  .enrich("direct:resource", aggregationStrategy)
  .to("direct:result");

from("direct:resource")
...
```

Content Enricher (**enrich**) は、(**元のエクスチェンジ** に含まれる) 受信メッセージを補完するために、**リソースエンドポイント** から追加のデータを取得します。集約ストラテジーは、元のエクスチェンジと **リソースエクスチェンジ** を組み合わせたものです。**AggregationStrategy.aggregate(Exchange, Exchange)** メソッドの最初のパラメーターは元のエクスチェンジに対応し、2 番目のパラメーターはリソースエクスチェンジに対応します。リソースエンドポイントの結果は、リソースエクスチェンジの **Out** メッセージに保存されます。以下は、独自の集約ストラテジークラスを実装するためのテンプレートの例です。

```
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;
    }

}
```

このテンプレートを使用すると、元のエクスチェンジがあらゆる交換パターンを持つことができます。Enricher によって作成されるリソースエクスチェンジは、常に **InOut** エクスチェンジです。

Spring XML 補完の例

上記の例は Spring XML にも実装できます。

```
<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="..." />
```

コンテンツの補完時のデフォルト集約ストラテジー

集約ストラテジーは任意です。提供しない場合、Apache Camel はデフォルトでリソースから取得したボディーを使用します。以下に例を示します。

```
from("direct:start")
  .enrich("direct:resource")
  .to("direct:result");
```

上記のルートでは、**direct:result** エンドポイントに送信されたメッセージには **direct:resource** からの出力が含まれます。これは、この例ではカスタムの集約を使用していないためです。

XML DSL では、以下のように **strategyRef** 属性を省略します。

```
<route>
    <from uri="direct:start"/>
    <enrich uri="direct:resource"/>
    <to uri="direct:result"/>
</route>
```

Enrich() メソッドでサポートされるオプション

enrich DSL コマンドは、以下のオプションをサポートします。

名前	デフォルト値	説明
expression	なし	Camel 2.16 以降では、このオプションは必須です。Enrich from を使用するよう外部サービスの URI を設定する式を指定します。Simple 式言語、Constant 式言語、または現在のエクスチェンジの値から動的に URI を計算できるその他の言語を使用できます。
uri		これらのオプションは削除されています。代わりに **expression** オプションを指定してください。Camel 2.15 以前では、**uri** オプションまたは **ref** オプションを指定する必要がありました。各オプションは、外部サービスの Enrich からのエンドポイント URI を指定します。

ref		補完のための外部サービスのエンドポイントを参照します。**uri** または **ref** のいずれかを使用する必要があります。
strategyRef		外部サービスからの応答を 1 つの送信メッセージにマージするために使用される AggregationStrategy を参照します。デフォルトでは、Camel は外部からの応答を送信メッセージとして使用します。POJO を **AggregationStrategy** として使用することができます。詳細は、Aggregate パターンのドキュメントを参照してください。
strategyMethodName		POJO を **AggregationStrategy** として使用している場合は、このオプションを指定して、集約メソッドの名前を明示的に宣言します。詳細は、Aggregate パターンを参照してください。
strategyMethodAllowNull	false	デフォルトの動作では、補完するデータがない場合、集約メソッドは使用されません。このオプションが true の場合、補完のデータがなく、POJO を **AggregationStrategy** に使用していると、null 値は **oldExchange** として使用されます。詳細は、Aggregate パターンを参照してください。
aggregateOnException	false	デフォルトの動作では、リソースから補完するデータの取得中に例外が発生すると、集約メソッドは **使用されません**。このオプションを **true** に設定すると、**aggregate** メソッドに例外がある場合に、エンドユーザーがアクションを制御できるようになります。たとえば、例外を非表示にしたり、カスタムメッセージのボディーを設定したりすることができます。

shareUntOfWork	false	Camel 2.16 以降、補完操作は親エクスチェンジとリソースエクスチェンジの間で Unit of Work を共有しないのがデフォルトの動作になります。これは、リソースエクスチェンジに個別の Unit of Work があることを意味します。詳細は、Splitter パターンについてのドキュメントを参照してください。
cacheSize	**1000**	Camel 2.16 以降では、このオプションを指定し、補完の操作でプロデューサーを再利用するためにプロデューサーをキャッシュすることができる **ProducerCache** のキャッシュサイズを指定します。このキャッシュを無効にするには、**cacheSize** オプションを **-1** に設定します。
ignoreInvalidEndpoint	false	Camel 2.16 以降、このオプションは解決できないエンドポイント URI を無視するかどうかを示します。デフォルトの動作では、無効なエンドポイント URI を特定する例外が Camel によって出力されます。

Enrich() メソッド使用時の集約ストラテジーの指定

enrich() メソッドは、リソースエンドポイントから追加のデータを取得し、元のエクスチェンジに含まれる受信メッセージを強化します。集約ストラテジーを使用して、元のエクスチェンジとリソースエクスチェンジを組み合わせることができます。**AggregationStrategy.aggregate(Exchange, Exchange)** メソッドの最初のパラメーターは、元のエクスチェンジに対応します。2 番目のパラメーターは、リソースのエクスチェンジに対応します。リソースエンドポイントの結果は、リソースエクスチェンジの **Out** メッセージに保存されます。以下に例を示します。

```
AggregationStrategy aggregationStrategy = ...

  from("direct:start")
  .enrich("direct:resource", aggregationStrategy)
  .to("direct:result");

  from("direct:resource")
...
```

以下のコードは、集約ストラテジーを実装するためのテンプレートです。このテンプレートを使用する実装では、元のエクスチェンジはメッセージエクスチェンジパターンになります。Enricher によって作成されるリソースエクスチェンジは、常に InOut メッセージエクスチェンジパターンです。

```
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;
    }

}
```

以下の例は、Spring XML DSL を使用して集約ストラテジーを実装する方法を示しています。

```
<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="..." />
```

Enrich() での動的 URI の使用

Camel 2.16 以降、**enrich()** および **pollEnrich()** メソッドは、現在のエクスチェンジからの情報に基づいて計算される動的 URI の使用をサポートします。たとえば、**orderId** キーのあるヘッダーが HTTP URL のコンテンツパスの一部として使用される HTTP エンドポイントから補完するには、以下のような操作を行うことができます。

```
from("direct:start")
  .enrich().simple("http:myserver/${header.orderId}/order")
  .to("direct:result");
```

以下は、XML DSL と同じ例です。

```
<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>
```

pollEnrich() メソッドを使用したコンテンツの補完

この **pollEnrich** コマンドは、リソースのエンドポイントを **コンシューマー** として扱います。エクスチェンジをリソースエンドポイントに送信する代わりに、エンドポイントを **ポーリング** します。デフォルトでは、リソースエンドポイントからエクスチェンジがない場合は、ポーリングはすぐに返します。たとえば、以下のルートは、受信 JMS メッセージのヘッダーから抽出される名前のファイルを読み取ります。

```
from("activemq:queue:order")
  .pollEnrich("file://order/data/additional?fileName=orderId")
  .to("bean:processOrder");
```

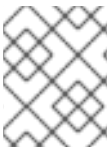
ファイルが準備できるまで待機する時間を制限できます。以下の例は、20 秒の最大待機時間を示しています。

```
from("activemq:queue:order")
  .pollEnrich("file://order/data/additional?fileName=orderId", 20000) // timeout is in milliseconds
  .to("bean:processOrder");
```

pollEnrich() の集計ストラテジーを指定することもできます。以下に例を示します。

```
  .pollEnrich("file://order/data/additional?fileName=orderId", 20000, aggregationStrategy)
```

pollEnrich() メソッドは、**consumer.bridgeErrorHandler=true** で設定されたコンシューマーをサポートします。これにより、ポーリングからの例外がルートエラーハンドラーに伝播され、たとえばポーリングを再試行できます。



注記

consumer.bridgeErrorHandler=true のサポートが Camel 2.18 で新たに追加されました。この動作は Camel 2.17 ではサポートされません。

集計ストラテジーの **aggregate()** メソッドに渡されるリソースエクスチェンジは、エクスチェンジを受け取る前にポーリングがタイムアウトした場合は **null** の可能性があります。

pollEnrich() によって使用されるポーリングメソッド

pollEnrich() メソッドは、以下のポーリングメソッドのいずれかを呼び出して、コンシューマーエンドポイントをポーリングします。

- **receiveNoWait()(これはデフォルトです。)**
- **receive()**
- **receive(long timeout)**

pollEnrich() コマンドのタイムアウト引数 (ミリ秒単位) は、以下のように呼び出すメソッドを決定します。

- タイムアウトが **0** または指定されていない場合、**pollEnrich()** は **receiveNoWait** を呼び出します。
- タイムアウトが負の値の場合、**pollEnrich()** は **receive** を呼び出します。

- それ以外の場合は、**pollEnrich()** は **receive(timeout)** を呼び出します。

データがない場合、集約ストラテジーの **newExchange** は null になります。

pollEnrich() メソッドの使用例

以下の例は、**inbox/data.txt** ファイルからコンテンツを読み込むことによるメッセージの補完を示しています。

```
 from("direct:start")
   .pollEnrich("file:inbox?fileName=data.txt")
   .to("direct:result");
```

以下は、XML DSL と同じ例です。

```
<route>
   <from uri="direct:start"/>
   <pollEnrich>
      <constant>file:inbox?fileName=data.txt"</constant>
   </pollEnrich>
   <to uri="direct:result"/>
</route>
```

指定したファイルが存在しない場合は、メッセージは空になります。ファイルが存在するまで待機するタイムアウトを指定するか、特定の時間まで待機できます。以下の例では、コマンドは 5 秒未満待機します。

```
<route>
   <from uri="direct:start"/>
   <pollEnrich timeout="5000">
      <constant>file:inbox?fileName=data.txt"</constant>
   </pollEnrich>
   <to uri="direct:result"/>
</route>
```

pollEnrich() での動的 URI の使用

Camel 2.16 以降、**enrich()** および **pollEnrich()** メソッドは、現在のエクスチェンジからの情報に基づいて計算される動的 URI の使用をサポートします。たとえば、ヘッダーを使用して SEDA キュー名を示すエンドポイントから Enrich をポーリングするには、以下を行います。

```
from("direct:start")
  .pollEnrich().simple("seda:${header.name}")
  .to("direct:result");
```

以下は、XML DSL と同じ例です。

```
<route>
   <from uri="direct:start"/>
   <pollEnrich>
      <simple>seda${header.name}</simple>
```

```
    </pollEnrich>
    <to uri="direct:result"/>
</route>
```

pollEnrich() メソッドでサポートされるオプション

pollEnrich DSL コマンドは、以下のオプションをサポートします。

名前	デフォルト値	説明
expression	なし	Camel 2.16 以降では、このオプションは必須です。Enrich from を使用するよう外部サービスの URI を設定する式を指定します。Simple 式言語、Constant 式言語、または現在のエクスチェンジの値から動的に URI を計算できるその他の言語を使用できます。
uri		これらのオプションは削除されています。代わりに **expression** オプションを指定してください。Camel 2.15 以前では、**uri** オプションまたは **ref** オプションを指定する必要がありました。各オプションは、外部サービスの Enrich からのエンドポイント URI を指定します。
ref		補完のための外部サービスのエンドポイントを参照します。**uri** または **ref** のいずれかを使用する必要があります。
strategyRef		外部サービスからの応答を 1 つの送信メッセージにマージするために使用される AggregationStrategy を参照します。デフォルトでは、Camel は外部からの応答を送信メッセージとして使用します。POJO を **AggregationStrategy** として使用することができます。詳細は、Aggregate パターンのドキュメントを参照してください。
strategyMethodName		POJO を **AggregationStrategy** として使用している場合は、このオプションを指定して、集約メソッドの名前を明示的に宣言します。詳細は、Aggregate パターンを参照してください。

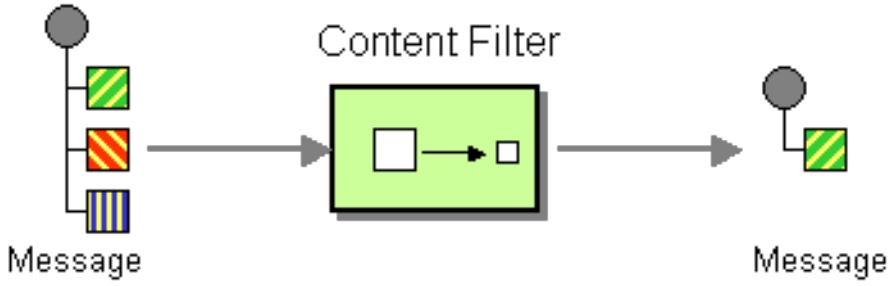
strategyMethodAllowNull	false	デフォルトの動作では、補完するデータがない場合、集約メソッドは使用されません。このオプションが true の場合、補完のデータがなく、POJO を **AggregationStrategy** に使用していると、null 値は **oldExchange** として使用されます。詳細は、Aggregate パターンを参照してください。
timeout	**-1**	外部サービスからのポーリング時に応答を待つ最大時間 (ミリ秒単位)。デフォルトの動作では、**pollEnrich()** メソッドが **receive()** メソッドを呼び出します。メッセージが利用可能になるまで **receive()** はブロックできるので、常にタイムアウトを指定することが推奨されます。
aggregateOnException	false	デフォルトの動作では、リソースから補完するデータの取得中に例外が発生すると、集約メソッドは**使用されません**。このオプションを **true** に設定すると、**aggregate** メソッドに例外がある場合に、エンドユーザーがアクションを制御できるようになります。たとえば、例外を非表示にしたり、カスタムメッセージのボディーを設定したりすることができます。
cacheSize	**1000**	このオプションを指定し、**pollEnrich()** 操作で再利用するためにコンシューマーをキャッシュする **ConsumerCache** のキャッシュサイズを設定します。このキャッシュを無効にするには、**cacheSize** オプションを **-1** に設定します。
ignoreInvalidEndpoint	false	解決できないエンドポイント URI を無視するかどうかを示します。デフォルトの動作では、無効なエンドポイント URI を特定する例外が Camel によって出力されます。

10.2. CONTENT FILTER

概要

Content Filter パターンは、メッセージから余分なコンテンツをフィルターリングしてから目的の受信者に配信する必要があるシナリオを説明します。たとえば、コンテンツフィルターを使用して、メッセージから機密情報を削除します。

図10.2 Content Filter パターン



メッセージをフィルターする一般的な方法は、サポートされているスクリプト言語 (たとえば、XQuery、JQuery、JSQL など) で記述された DSL で式を使用することです。

コンテンツフィルターの実装

コンテンツフィルターは、基本的に特定の目的のためのメッセージ処理手法のアプリケーションです。コンテンツフィルターを実装するには、以下のメッセージ処理技術を使用できます。

- **メッセージトランスレーター**: 「メッセージトランスレーター」 を参照してください 。
- **プロセッサー**: 35章 *プロセッサーの実装* を参照してください。
- Bean インテグレーション

XML 設定の例

以下の例は、XML で同じルートを設定する方法を示しています。

```
<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>
```

XPath フィルターの使用

XPath を用いて対象のメッセージの一部をフィルターリングすることもできます。

```
<route>
  <from uri="activemq:Input"/>
  <setBody><xpath resultType="org.w3c.dom.Document">//foo:bar</xpath></setBody>
  <to uri="activemq:Output"/>
</route>
```

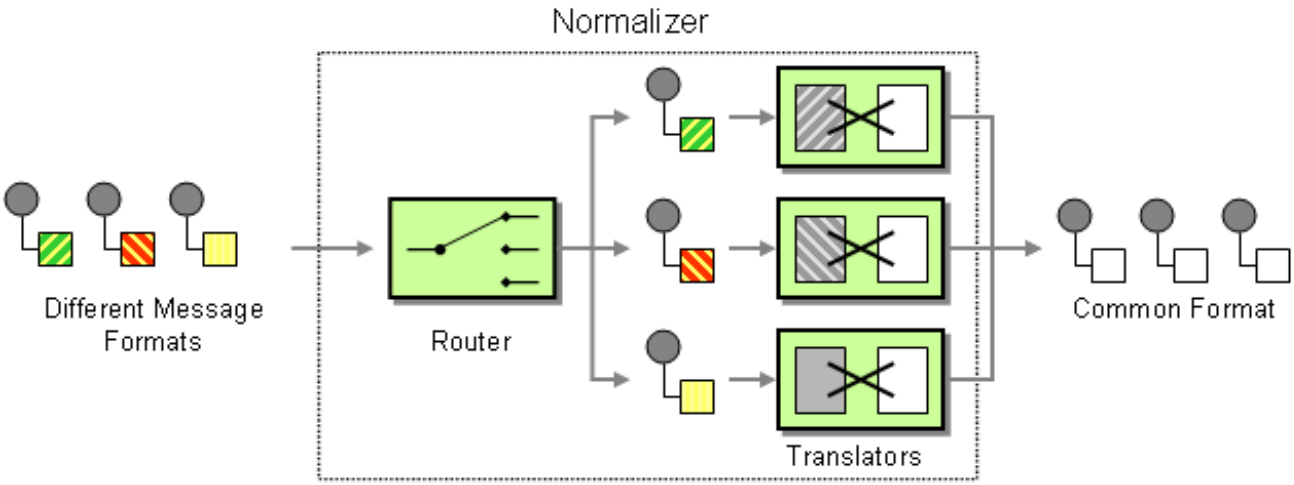
10.3. ノーマライザー

概要

Normalizer パターンはセマンティック的に同等のメッセージを処理するために使用されますが、異なる形式で受信します。ノーマライザーは受信メッセージを共通の形式に変換します。

Apache Camel では、「Content-Based Router」 で受信メッセージのフォーマットを検出し、「メッセージトランスレーター」 で異なる受信形式を共通の形式に変換して組み合わせることにより、Normalizer パターンを実装することができます。

図10.3 Normalizer パターン



Java DSL の例

この例では、2 種類の XML メッセージを共通の形式に変換する Message Normalizer を示しています。この一般的な形式のメッセージはフィルターされます。

Fluent Builder (流れるようなビルダー) の使用

```
// 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");
```

この場合、Java Bean をノーマライザーとして使用します。クラスは次のようになります。

```
// 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 設定の例

XML DSL と同じ例

```
<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. CLAIM CHECK EIP

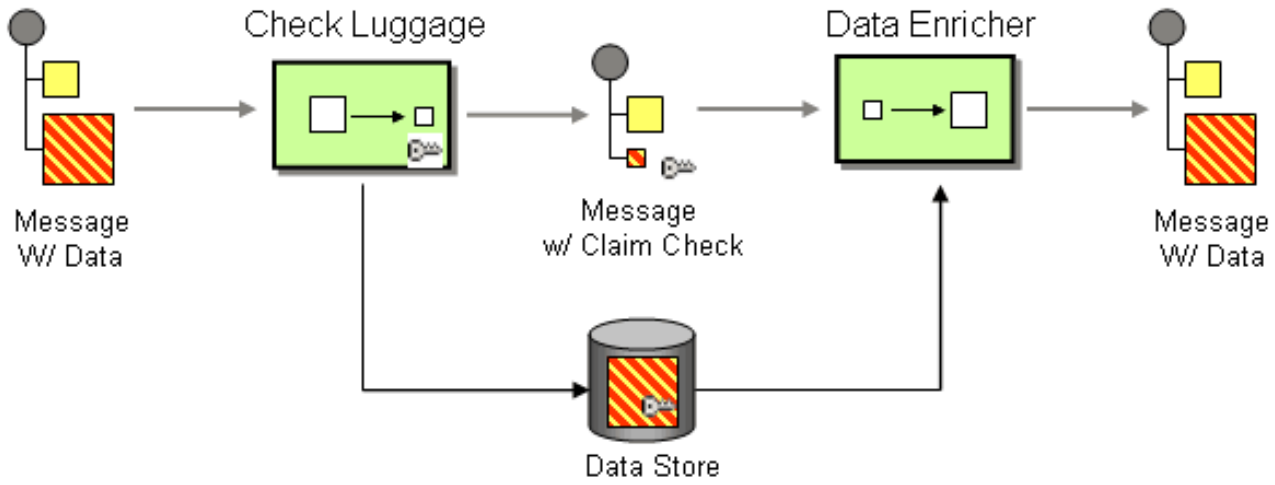
Claim Check EIP

図10.4「Claim Check パターン」 に記載されている **Claim Check** EIP パターンでは 、メッセージコンテンツをクレームチェック (一意の鍵) に置き換えることができます。**Claim Check** EIP パターンを使用して、後でメッセージコンテンツを取得します。メッセージコンテンツは、一時的にデータベースやファイルシステムなどの永続ストアに格納できます。このパターンは、メッセージコンテンツが非常に大きく (送信するに高価)、すべてのコンポーネントにすべての情報が必要でない場合に役立ちます。

また、外部の情報で信頼できない場合にも便利です。この場合、**Claim Check** を使用してデータの機密部分を非表示にします。

EIP パターンの Camel 実装は、メッセージコンテンツを内部メモリーストアに一時的に格納します。

図10.4 Claim Check パターン



10.4.1. Claim Check EIP オプション

Claim Check EIP は、以下の表に記載されているオプションをサポートしています。

名前	説明	デフォルト	型

operation	Claim Check 操作を使用する必要があります。以下の操作をサポートします。 * **Get** - 指定のキーによる Claim Check を取得します (削除しません)。 * **GetAndRemove** - 指定のキーによる Claim Check を取得して削除します。 * **Set** - 指定のキーで新規の Claim Check を設定します。キーがすでに存在する場合は上書きされます。 * **Push** - スタックに新しい Claim Check を設定します (キーは使用しません)。 * **Pop** - スタックから最新の Claim Check を取得します (キーは使用しません)。 **Get**、**GetAndRemove**、または **Set** 操作を使用する場合は、キーを指定する必要があります。これらの操作はキーを使用してデータを保存し、取得します。これらの操作を使用して、複数のデータを異なるキーに保存します。ただし、**push** および **pop** 操作はキーを使用しませんが、データをスタック構造体に保存します。		ClaimCheckOperation
key	Claim Check ID に特定のキーを使用します。		String
filter	Claim Check リポジトリーからマージするデータを制御するフィルターを指定します。		String

strategyRef	デフォルトの実装の代わりにカスタム **AggregationStrategy** を使用します。カスタム集約ストラテジーと同時にデータを設定することはできません。		String

フィルターオプション

Filter オプションを使用して、**Get** または **Pop** の操作を使用する場合にマージして戻すデータを定義します。**AggregationStrategy** を使用してデータをマージして戻します。デフォルトのストラテジーでは、filter オプションを使用して、マージするデータを簡単に指定します。

filter オプションは、以下の構文を持つ String 値を取ります。

- **body**: メッセージボディーを集約します。
- **attachments**: すべてのメッセージ添付を集約します。
- **headers**: すべてのメッセージヘッダーを集約します。
- **header:pattern**: パターンに一致するすべてのメッセージヘッダーを集約します。

パターンルールはワイルドカードおよび正規表現をサポートします。

- ワイルドカードの一致 (パターンが * で終わり、名前がパターンで始まります)
- 正規表現の一致

複数のルールを指定するには、**commas** (,) で区切ります。

以下は、メッセージボディーおよび **foo** で始まるすべてのヘッダーを含む基本的なフィルターの例です。

```
body, header:foo*
```

- メッセージのボディーのみをマージする場合: **body**
- メッセージの添付のみをマージする場合: **attachments**
- ヘッダーのみをマージする場合: **headers**
- ヘッダー名 **foo** のみをマージする場合: **header:foo**

フィルタールールを空またはワイルドカードとして指定すると、すべてをマージできます。詳細は、Filter what data to merge back を参照してください。

注記

データをマージすると、システムは既存のデータを上書きします。また、既存のデータを格納します。

10.4.2. Include および Exclude パターンを使用したフィルターオプション

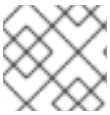
以下は、オプションを包含、除外、または削除オプションを指定するために使用できる接頭辞をサポートする構文です。

- +: 包含します (デフォルトのモード)。
- -: 除外します (除外は包含よりも優先されます)
- --: 削除します (削除が優先されます)

以下に例を示します。

- メッセージのボディーをスキップしてその他をすべてマージするには、**-body** を使用します。
- メッセージヘッダー **foo** を省略してその他をすべてマージするには、**-header:foo** を使用します。

また、データをマージする際にヘッダーを削除するようにシステムに指示することもできます。たとえば、bar で始まるすべてのヘッダーを削除するには、**--headers:bar*** を使用します。



注記

header:pattern で包含 (+) と除外 (-) の両方を同時に使用しないでください。

10.4.3. Java の例

以下の例は、実際の **Push** および **Pop** 操作を示しています。

```
from("direct:start")
    .to("mock:a")
    .claimCheck(ClaimCheckOperation.Push)
    .transform().constant("Bye World")
    .to("mock:b")
    .claimCheck(ClaimCheckOperation.Pop)
    .to("mock:c");
```

以下は、**Get** および **Set** 操作の使用例です。この例では、**foo** キーを使用しています。

```
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");
```

注記

データを削除しないため、**Get** 操作を使用して同じデータを 2 度取得できます。ただし、データを 1 度だけ取得する場合は、**GetAndRemove** 操作を使用します。

以下の例は、**foo** または **bar** としてヘッダーのみを取得する場合に **filter** オプションを使用する方法を示しています。

```
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 の例

以下の例は、実際の **Push** および **Pop** 操作を示しています。

```
<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>
```

以下は、**Get** および **Set** 操作の使用例です。この例では、**foo** キーを使用しています。

```
<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>
```

注記

データを削除しないため、**Get** 操作を使用して同じデータを 2 度取得できます。ただし、データを 1 度だけ取得する場合は、**GetAndRemove** 操作を使用します。

以下の例は、**filter** オプションを使用して、ヘッダーを **foo** または **bar** として取得する方法を示しています。

```
<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. 並び替え

Sort

Sort パターンは、メッセージのボディーにソートできる項目の一覧が含まれていることを前提として、メッセージのボディーの内容をソートするために使用します。

デフォルトでは、メッセージのコンテンツは、数値または文字列を処理するデフォルトのコンパレーターを使用してソートされます。独自のコンパレーターを提供でき、ソートするリストを返す式を指定できます (式は **java.util.List** に変換可能である必要があります)。

Java DSL の例

以下の例では、改行文字でトークン化してソートする項目のリストを生成します。

```
from("file://inbox").sort(body().tokenize("\n")).to("bean:MyServiceBean.processLine");
```

sort() に 2 番目の引数として独自のコンパレーターを渡すことができます。

```
from("file://inbox").sort(body().tokenize("\n"), new
MyReverseComparator()).to("bean:MyServiceBean.processLine");
```

XML 設定の例

Spring XML でも同じルートを設定することができます。

以下の例では、改行文字でトークン化してソートする項目のリストを生成します。

```
<route>
  <from uri="file://inbox"/>
  <sort>
    <simple>body</simple>
```

```
  </sort>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>
```

また、カスタムコンパレーターを使用するには、これを Spring Bean として参照できます。

```
<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"/>
```

<simple> の他にも、任意の言語を使用してリストを返す式を指定することもできます。

オプション

sort DSL コマンドは以下のオプションをサポートします。

名前	デフォルト値	説明
comparatorRef		メッセージボディーのソートに使用するカスタムの **java.util.Comparator** を参照します。Camel はデフォルトで、A から Z のソートを行うコンパレーターを使用します。

10.6. トランスフォーマー

トランスフォーマーは、ルート定義で宣言された **Input Type** や **Output Type** に従って、メッセージの宣言的な変換を実行します。デフォルトの Camel メッセージは **DataTypeAware** を実装します。これは **DataType** で表されるメッセージタイプを保持します。

10.6.1. トランスフォーマーの仕組み

ルート定義は **Input Type** や **Output Type** を宣言します。**Input Type** や **Output Type** がランタイムのメッセージタイプと異なる場合、Camel 内部プロセッサーはトランスフォーマーを検索します。トランスフォーマーは現在のメッセージタイプを予想されるメッセージタイプに変換します。メッセージが正常に変換されたり、メッセージが想定されるタイプである場合は、メッセージデータタイプが更新されます。

10.6.1.1. データタイプフォーマット

データタイプのフォーマットは **scheme:name** です。scheme は、**java**、**xml** または **json** といったデータモデルのタイプで、name はデータタイプ名です。

注記

scheme のみを指定すると、そのスキームですべてのデータタイプと一致します。

10.6.1.2. サポート対象のトランスフォーマー

トランスフォーマー	説明
データフォーマットトランスフォーマー	データフォーマットを使用した変換
エンドポイントトランスフォーマー	エンドポイントを使用した変換
カスタムトランスフォーマー	カスタムトランスフォーマークラスを使用した変換。

10.6.1.3. 共通オプション

すべてのトランスフォーマーには、トランスフォーマーでサポートされるデータタイプを指定するための以下の共通のオプションがあります。

重要

scheme または **fromType** および **toType** の両方を指定する必要があります。

名前	説明
scheme	**xml** または **json** などのデータモデルのタイプ。たとえば、**xml** が指定されている場合は、トランスフォーマーがすべての java -> xml および xml -> java 変換に適用されます。
fromType	変換元の Data type。
toType	変換する Data type。

10.6.1.4. DataFormat トランスフォーマーオプション

名前	説明
type	データフォーマットの種類
ref	データフォーマット ID への参照

bindy DataFormat タイプを指定する例は次のとおりです。

Java DSL の場合

```
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 の場合

```
<dataFormatTransformer fromType="java:com.example.Order" toType="csv:CSVOrder">
    <bindy id="csvdf" type="Csv" classType="com.example.Order"/>
</dataFormatTransformer>
```

10.6.2. エンドポイントトランスフォーマーオプション

名前	説明
ref	エンドポイント ID への参照
uri	エンドポイント URI

Java DSL でエンドポイント URI を指定する例:

```
transformer()
    .fromType("xml")
    .toType("json")
    .withUri("dozer:myDozer?mappingFile=myMapping.xml...");
```

XML DSL でエンドポイント ref を指定する例:

```
<transformers>
<endpointTransformer ref="myDozerEndpoint" fromType="xml" toType="json"/>
</transformers>
```

10.6.3. カスタムトランスフォーマーオプション

注記

トランスフォーマーは **org.apache.camel.spi.Transformer** のサブクラスである必要があります。

名前	説明
ref	カスタムトランスフォーマー Bean ID への参照
className	カスタムトランスフォーマークラスの完全修飾クラス名

カスタムトランスフォーマークラスを指定する例:

Java DSL の場合

```
transformer()
    .fromType("xml")
    .toType("json")
    .withJava(com.example.MyCustomTransformer.class);
```

XML DSL の場合

```
<transformers>
<customTransformer className="com.example.MyCustomTransformer" fromType="xml"
toType="json"/>
</transformers>
```

10.6.4. トランスフォーマーの例

この例は 2 つの部分があり、最初の部分はメッセージを変換するエンドポイントトランスフォーマーを宣言します。2 番目の部分は、トランスフォーマーをルートに適用する方法を示します。

10.6.4.1. 最初の部分

xslt コンポーネントを使用して **xml:ABCOrder** から **xml:XYZOrder** に変換する、エンドポイントトランスフォーマーを宣言します。

Java DSL の場合

```
transformer()
    .fromType("xml:ABCOrder")
    .toType("xml:XYZOrder")
    .withUri("xslt:transform.xsl");
```

XML DSL の場合

```
<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. 2 番目の部分

direct:abc エンドポイントが **direct:xyz** にメッセージを送信するときに、上記のトランスフォーマーは以下のルート定義に適用されます。

Java DSL の場合

```
from("direct:abc")
    .inputType("xml:ABCOrder")
    .to("direct:xyz");
```

```
from("direct:xyz")
    .inputType("xml:XYZOrder")
    .to("somewhere:else");
```

XML DSL の場合

```
<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. バリデーター

バリデーターは、メッセージタイプを宣言するルート定義で、宣言された **Input Type** や **Output Type** に従って、メッセージの宣言的検証を実行します。

注記

検証は、type 宣言の **validate** 属性が true の場合にのみ実行されます。

Input Type や **Output Type** の宣言で **validate** 属性が true の場合、Camel 内部プロセッサーはレジストリーから対応するバリデーターを検索します。

10.7.1. データタイプフォーマット

データタイプのフォーマットは **scheme:name** です。**scheme** は、**java**、**xml**、または **json** といったデータモデルのタイプで、**name** はデータタイプ名です。

10.7.2. サポート対象のバリデーター

バリデーター	説明
述語バリデーター	式または述語を使用して検証します。
エンドポイントバリデーター	Validation Component や Bean Validation Component などの検証コンポーネントと使用する Endpoint へ転送して検証します。
カスタムバリデーター	カスタムバリデータークラスを使用して検証します。Validator は **org.apache.camel.spi.Validator** のサブクラスである必要があります。

10.7.3. 共通オプション

すべてのバリデーターには検証する データタイプ を指定する type オプションが含まれている必要があります。

10.7.4. 述語バリデーターのオプション

名前	説明
expression	検証に使用する式または述語。

検証の述語を指定する例:

Java DSL の場合

```
validator()
    .type("csv:CSVOrder")
    .withExpression(bodyAs(String.class).contains("{name:XOrder}"));
```

XML DSL の場合

```
<predicateValidator Type="csv:CSVOrder">
    <simple>${body} contains 'name:XOrder'</simple>
</predicateValidator>
```

10.7.5. エンドポイントバリデーターのオプション

名前	説明
ref	エンドポイント ID への参照。
uri	エンドポイント URI

Java DSL でエンドポイント URI を指定する例:

```
validator()
    .type("xml")
    .withUri("validator:xsd/schema.xsd");
```

XML DSL でエンドポイント ref を指定する例:

```
<validators>
<endpointValidator uri="validator:xsd/schema.xsd" type="xml"/>
</validators>
```

注記

エンドポイントバリデーターはメッセージを指定されたエンドポイントに転送します。上記の例では、Camel はメッセージを Validation Component である **validator:** エンドポイントに転送します。Bean Validation Component などの異なる検証コンポーネントを使用することもできます。

10.7.6. カスタムバリデーターのオプション

注記

Validator は **org.apache.camel.spi.Validator** のサブクラスである必要があります。

名前	説明
ref	カスタムバリデーター Bean ID への参照。
className	カスタムバリデータークラスの完全修飾クラス名。

カスタムバリデータークラスを指定する例:

Java DSL の場合

```
validator()
    .type("json")
    .withJava(com.example.MyCustomValidator.class);
```

XML DSL の場合

```
<validators>
<customValidator className="com.example.MyCustomValidator" type="json"/>
</validators>
```

10.7.7. バリデーターの例

この例は 2 つの部分があり、最初の部分はメッセージを検証するエンドポイントバリデーターを宣言します。2 番目の部分は、バリデーターをルートに適用する方法を示します。

10.7.7.1. 最初の部分

バリデーターコンポーネントを使用して、**xml:ABCOrder** から検証するエンドポイントバリデーターを宣言します。

Java DSL の場合

```
validator()
    .type("xml:ABCOrder")
    .withUri("validator:xsd/schema.xsd");
```

XML DSL の場合

```
<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. 2 番目の部分

direct:abc エンドポイントがメッセージを受信すると、上記のバリデーターが以下のルート定義に適用されます。

注記

Java DSL では、**inputType** ではなく、**inputTypeWithValidate** が使用され、XML DSL では inputType 宣言の **validate** 属性は **true** に設定されます。

Java DSL の場合

```
from("direct:abc")
    .inputTypeWithValidate("xml:ABCOrder")
    .log("${body}");
```

XML DSL の場合

```
<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

概要

Validate パターンは、メッセージの内容が有効かどうかをチェックする便利な構文を提供します。validate DSL コマンドは、唯一の引数として述語式を取ります。述語が **true** と評価されると、ルートは正常に処理を継続します。述語が **false** と評価されると、**PredicateValidationException** が出力されます。

Java DSL の例

以下のルートは、正規表現を使用して現在のメッセージのボディーを検証します。

```
from("jms:queue:incoming")
  .validate(body(String.class).regex("^\\w{10}\\,\\d{2}\\,\\w{24}$"))
  .to("bean:MyServiceBean.processLine");
```

メッセージヘッダーを検証することもできます。以下に例を示します。

```
from("jms:queue:incoming")
  .validate(header("bar").isGreaterThan(100))
  .to("bean:MyServiceBean.processLine");
```

また、Simple 式言語で検証を使用できます。

```
from("jms:queue:incoming")
  .validate(simple("${in.header.bar} == 100"))
  .to("bean:MyServiceBean.processLine");
```

XML DSL の例

XML DSL で検証を使用するには、Simple 式言語を使用することが推奨されます。

```
<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"/>
```

メッセージヘッダーを検証することもできます。以下に例を示します。

```
<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 MESSAGING ENDPOINT

概要

Messaging Endpoint パターンは、エンドポイントに設定できるさまざまな機能とサービス品質を表しています。

11.1. MESSAGING MAPPER

概要

Messaging Mapper パターンは、ドメインオブジェクトと正規のメッセージ形式とを双方向にマップする方法を説明しています。ここでは、メッセージ形式は可能な限りプラットフォーム非依存なものが選ばれます。選ばれたメッセージ形式は、「Message Bus」 上での送信に適したものである必要があります。メッセージバスは、さまざまなシステムを統合するための基盤であり、統合するシステムの一部はオブジェクト指向ではない場合があります。

多くの異なるアプローチが可能ですが、そのすべてがメッセージングマッパーの要件を満たしている訳ではありません。たとえば、オブジェクトを送信する明らかな方法の 1 つは、**オブジェクトのシリアライズ** を使用することです。この方法を使えば、明確なエンコーディング (Java でネイティブにサポートされている) を使用してオブジェクトをデータストリームに書き込むことができます。しかし、これはメッセージングマッパーパターンに使用するのに適した方法では **ありません**。シリアライズ形式は Java アプリケーションでしか認識できないためです。Java オブジェクトのシリアライズは、元のアプリケーションとメッセージングシステム内の他のアプリケーションとの間でインピーダンスミスマッチを発生させます。

メッセージングマッパーの要件は、以下のように要約できます。

- ドメインオブジェクトの送信に使用される正規のメッセージ形式は、オブジェクト指向でないアプリケーションでの使用に適したものである必要があります。
- マッパーのコードは、ドメインオブジェクトのコードからもメッセージングインフラストラクチャーからも分離して実装する必要があります。Apache Camel は、マッパーのコードをルートに挿入するためのフックを提供することで、この要件に対応しています。
- マッパーは継承、オブジェクト参照、オブジェクトツリーなどのオブジェクト指向の概念を効果的に扱う方法を見つけなければならない場合があります。こうした問題の複雑さはアプリケーションによって異なりますが、マッパー実装の目的は、常に、オブジェクト指向でないアプリケーションが効果的に処理できるメッセージを作成することにあります。

マップすべきオブジェクトの検索

以下のメカニズムのいずれかを使用して、マップするオブジェクトを見つけることができます。

- **登録された Bean を検索する。** - シングルトンオブジェクトや数の少ないオブジェクトの場合は、**CamelContext** レジストリーを使用して Bean への参照を格納することができます。たとえば、Bean インスタンスが Spring XML を使用してインスタンス化されている場合、Bean はレジストリーに自動的に登録され、**id** 属性の値で識別されます。
- **JoSQL 言語を使用してオブジェクトを選択する。** - アクセスするオブジェクトがすべて実行時にすでにインスタンス化されている場合は、JoSQL 言語を使用して特定のオブジェクト (または複数のオブジェクト) を見つけることができます。たとえば、**name** Bean プロパティーを持

つ **org.apache.camel.builder.sql.Person** クラスがあり、受信メッセージが **UserName** ヘッダーを持つ場合は、以下のコードを使用して **name** プロパティーの値が **UserName** ヘッダーの値に等しいオブジェクトを選択できます。

```
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);
```

構文 **:HeaderName** は、JSQL 式のヘッダーの値を置き換えるために使用されます。

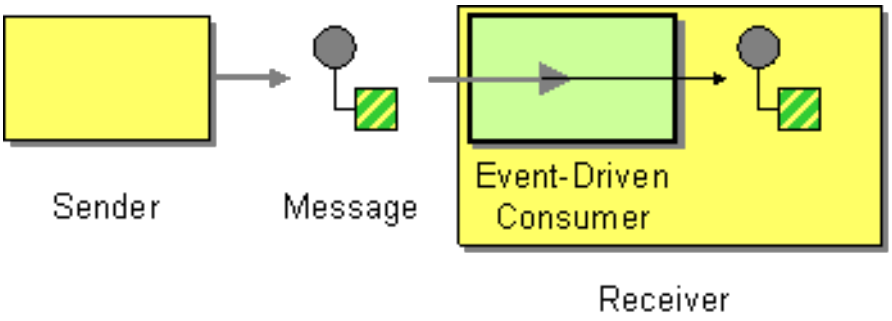
- **動的** - よりスケーラブルなソリューションを必要とする場合には、データベースからオブジェクトデータを読み取る必要がある可能性があります。場合によっては、既存のオブジェクト指向アプリケーションがすでにデータベースからオブジェクトをロードできるファインダーオブジェクトを提供していることがあります。そうでない場合、データベースからオブジェクトを抽出するカスタムコードを書く必要がある可能性があります。その場合には、JDBC コンポーネントと SQL コンポーネントが役に立つことがあります。

11.2. EVENT DRIVEN CONSUMER

概要

図11.1「Event Driven Consumer パターン」 に示される **Event Driven Consumer パターン** は、Apache Camel コンポーネントでコンシューマーエンドポイントを実装するためのパターンの 1 つであり、Apache Camel でカスタムコンポーネントを開発する必要があるプログラマーにのみ関係があります。既存のコンポーネントには、すでに何らかのコンシューマー実装パターンが組み込まれています。

図11.1 Event Driven Consumer パターン



このパターンに準拠するコンシューマーは、受信メッセージを受け取るたびに、メッセージングチャネルまたはトランスポート層から自動的に呼び出されるイベントメソッドを提供します。Event Driven Consumer パターンの特徴の 1 つは、コンシューマーエンドポイントそれ自体は受信メッセージを処理するためのスレッドを提供しないことです。代わりに、基礎となるトランスポートまたはメッセージングチャネルが、公開されたイベントメソッドを呼び出す際に暗黙的に処理用のスレッドを提供します (イベントメソッドはメッセージ処理の期間中そのスレッドをブロックします)。

この実装パターンの詳細については、「コンシューマーパターンおよびスレッド」 および 41 章*Consumer インターフェイス* を参照してください。

11.3. POLLING CONSUMER

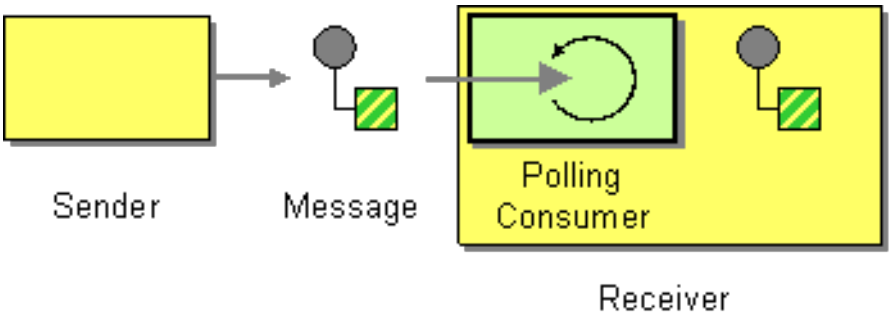
概要

図11.2「Polling Consumer パターン」 に示される **Polling Consumer** パターンは、Apache Camel コンポーネントでコンシューマーエンドポイントを実装するためのパターンの 1 つであり、そのため Apache Camel でカスタムコンポーネントを開発する必要があるプログラマーにのみ関係があります。既存のコンポーネントには、すでに何らかのコンシューマー実装パターンが組み込まれています。

このパターンに準拠するコンシューマーは、監視対象のリソースから利用可能な場合にのみ新規のエクスチェンジオブジェクトを返すポーリングメソッド **receive()**、**receive(long timeout)**、および **receiveNoWait()** を公開します。ポーリングコンシューマーの実装は、ポーリングを実行するために独自にスレッドプールを提供する必要があります。

この実装パターンの詳細については、「コンシューマーパターンおよびスレッド」、41章*Consumer インターフェイス* および 「コンシューマーテンプレートの使用」 を参照してください。

図11.2 Polling Consumer パターン



スケジュールされたポーリングコンシューマー

Apache Camel コンシューマーエンドポイントの多くは、スケジュールされたポーリングのパターンを使用してルートの開始時にメッセージを受信します。つまり、そのエンドポイントはイベント駆動型コンシューマーのインターフェイスを実装しているように見えますが、内部的にはスケジュールされたポーリングを使用して、エンドポイントに受信メッセージを提供するリソースを監視しています。

このパターンの実装方法についての詳細は、「Consumer インターフェイスの実装」 を参照してください。

Quartz コンポーネント

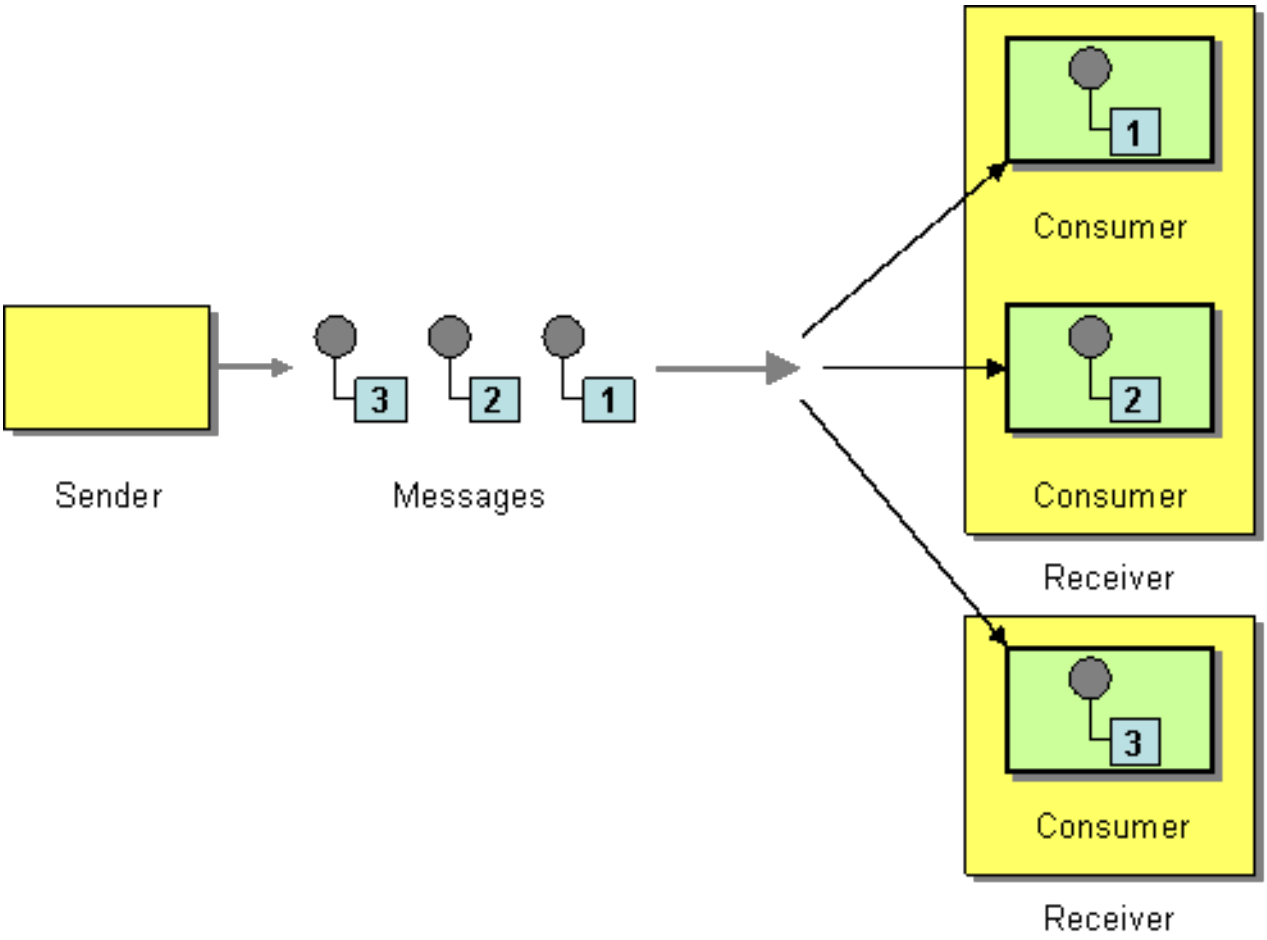
Quartz コンポーネントを使用することで、**Quartz** エンタープライズスケジューラーを使用してスケジュールされたメッセージの配信を提供できます。詳細は、**Apache Camel Component Reference Guide** の **Quartz** および Quartz Component を参照してください。

11.4. COMPETING CONSUMERS

概要

図11.3「Competing Consumers パターン」 に示される **Competing Consumers** パターンは、複数のコンシューマーが同じキューからメッセージを引き出すことを可能にしつつ、**各メッセージはたった一度だけ消費される** ことを保証します。このパターンを使用すると、逐次的なメッセージ処理を並行的なメッセージ処理に置き換えることができます (結果として応答レイテンシーの減少をもたらします)。

図11.3 Competing Consumers パターン



以下のコンポーネントは、Competing Consumers パターンの例になります。

- JMS ベースの競合コンシューマー
- SEDA ベースの競合コンシューマー

JMS ベースの競合コンシューマー

通常の JMS キューは、各メッセージがたった一度だけ消費されることを暗黙的に保証しています。そのため、JMS キューは自動的に Competing Consumers パターンをサポートします。たとえば、以下のように、JMS キュー **HighVolumeQ** からメッセージを引き出す 3 つの Competing Consumers を定義できます。

```
from("jms:HighVolumeQ").to("cxf:bean:replica01");
from("jms:HighVolumeQ").to("cxf:bean:replica02");
from("jms:HighVolumeQ").to("cxf:bean:replica03");
```

ここでは、CXF (Web サービス) エンドポイント **replica01**、**replica02**、および **replica03** は、**HighVolumeQ** キューからのメッセージを並行して処理します。

もう 1 つの方法として、JMS クエリーオプション **concurrentConsumers** を設定して、Competing Consumers のスレッドプールを作成することもできます。たとえば、以下のルートは、指定されたキューからメッセージを取得する 3 つの競合スレッドのプールを作成します。

```
from("jms:HighVolumeQ?concurrentConsumers=3").to("cxf:bean:replica01");
```

concurrentConsumers オプションは、以下のように XML DSL においても指定することができます。

```
<route>
  <from uri="jms:HighVolumeQ?concurrentConsumers=3"/>
  <to uri="cxf:bean:replica01"/>
</route>
```

注記

JMS トピックは Competing Consumers パターンをサポート **しません**。定義上、JMS トピックは、同じメッセージの複数のコピーを異なるコンシューマーに送信することを目的としています。したがって、Competing Consumers パターンとは互換性がありません。

SEDA ベースの競合コンシューマー

SEDA コンポーネントの目的は、計算を複数のステージに分割することで並行処理を単純化することです。SEDA エンドポイントは本質的に、インメモリーのブロッキングキュー (**java.util.concurrent.BlockingQueue** で実装された) をカプセル化します。そのため、SEDA エンドポイントを使用してルートを複数のステージに分割し、各ステージでは複数のスレッドを使用することもできます。たとえば、以下のように 2 つのステージで設定される SEDA ルートを定義できます。

```
// 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");
```

最初のステージには、ファイルエンドポイント **file://var/messages** からのメッセージを消費し、それらを SEDA エンドポイント **seda:fanout** にルーティングする単一のスレッドが含まれています。2 番目のステージには、3 つのスレッドが含まれます。エクスチェンジを **cxf:bean:replica01** にルーティングするスレッド、エクスチェンジを **cxf:bean:replica02** にルーティングするスレッド、そしてエクスチェンジを **cxf:bean:replica03** にルーティングするスレッドです。これら 3 つのスレッドは、ブロッキングキューを使用して実装された SEDA エンドポイントから、エクスチェンジインスタンスを取得するために競合します。ブロッキングキューはロックを使用して一度に複数のスレッドがキューにアクセスするのを防ぐため、エクスチェンジインスタンスは一度だけ消費されることが保証されます。

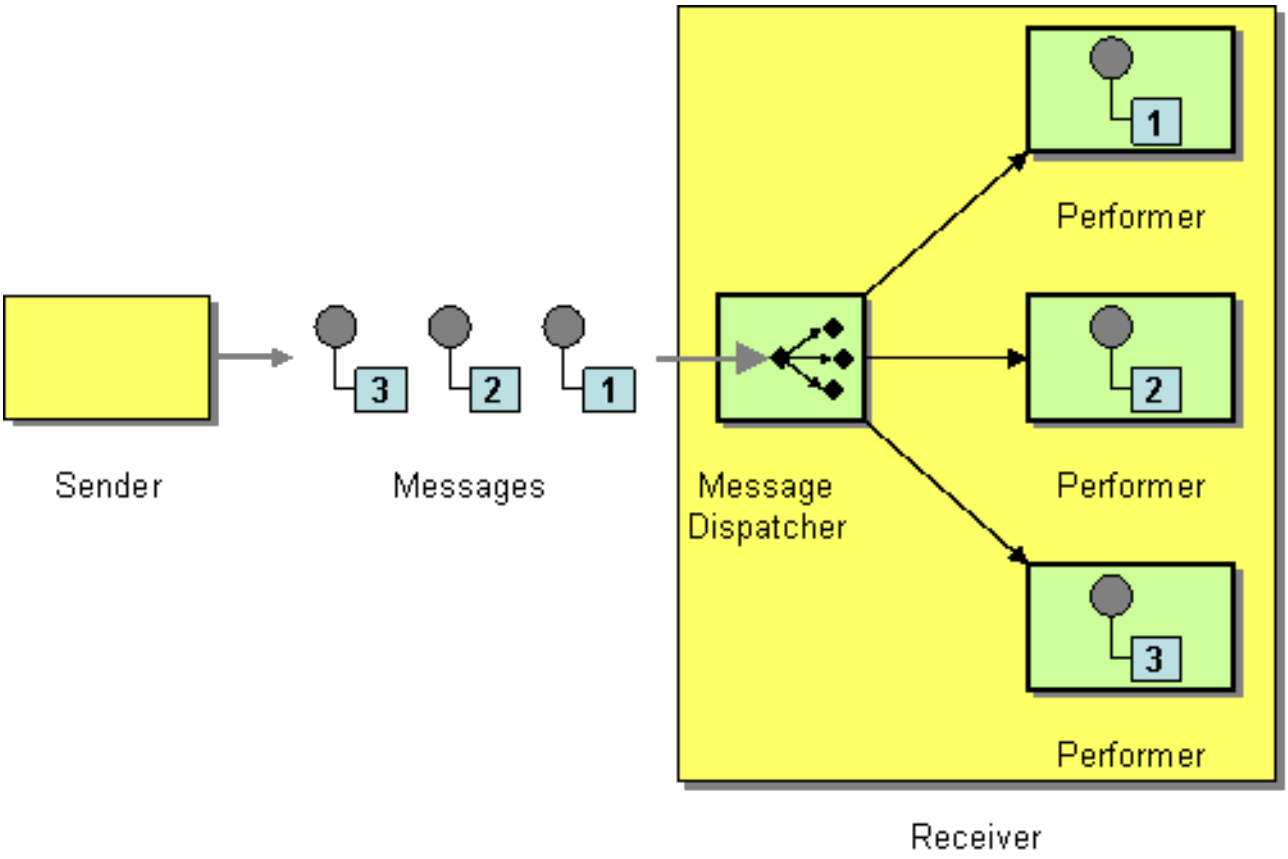
SEDA エンドポイントと **thread()** によって作成されたスレッドプールとの違いについては、**Apache Camel Component Reference Guide** の SEDA component を参照してください。

11.5. MESSAGE DISPATCHER

概要

図11.4「Message Dispatcher パターン」 に示される **Message Dispatcher** パターンは、チャネルからメッセージを消費し、それらをメッセージ処理を担当する **パフォーマー** にローカルに配信するために使用されます。Apache Camel アプリケーションでは通常、パフォーマーはプロセス内エンドポイントで表されます。これらのエンドポイントは、メッセージをルートの別のセクションに転送するために使用されます。

図11.4 Message Dispatcher パターン



Apache Camel で Message Dispatcher パターンを実装するには、以下の方法があります。

- JMS セレクター
- ActiveMQ の JMS セレクター
- コンテンツベースルーター

JMS セレクター

アプリケーションが JMS キューからメッセージを消費する場合は、**JMS セレクター** を使用して Message Dispatcher パターンを実装できます。JMS セレクターは、JMS ヘッダーと JMS プロパティーに対する述語の式です。セレクターが **true** に評価されると JMS メッセージはコンシューマーに到達でき、セレクターが **false** に評価されると JMS メッセージはブロックされます。多くの点で JMS セレクターは 「Message Filter」 に似ていますが、そのフィルタリングが JMS プロバイダーの中で実装されているという追加の利点があります。これは、JMS セレクターがメッセージを Apache Camel アプリケーションに送信される前にブロックできることを意味します。これにより、効率を大幅に向上させることができます。

Apache Camel では、JMS エンドポイント URI で **selector** クエリーオプションを設定することで、コンシューマーエンドポイントに JMS セレクターを定義できます。以下に例を示します。

```
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");
```

セレクター文字列の述語は、SQL92 条件式構文のサブセットに基づきます (完全な詳細は JMS 仕様 を参照してください)。セレクター文字列の識別子は、JMS ヘッダーまたは JMS プロパティーのいずれかを参照できます。たとえば、前述のルートでは、送信側は **CountryCode** という JMS プロパティー

を設定しています。

Apache Camel アプリケーション内からメッセージに JMS プロパティーを追加する場合は、メッセージヘッダー (**In** メッセージまたは **Out** メッセージのいずれかに対して) を設定することで実現できます。JMS エンドポイントへの読み取りまたは書き込み時に、Apache Camel は JMS ヘッダーおよび JMS プロパティーとネイティブメッセージのヘッダーとの間のマッピングを行います。

技術的には、セレクター文字列は **application/x-www-form-urlencoded** MIME 形式に従い、URL エンコードされている必要があります (HTML 仕様 を参照)。実際には、**&** (アンパサンド) 文字は URI の各クエリーオプションを区切るために使用されるので、問題が発生する可能性があります。**&** 文字を埋め込む必要があるような複雑なセレクター文字列については、**java.net.URLEncoder** ユーティリティークラスを使用して文字列をエンコードできます。以下に例を示します。

```
from("jms:dispatcher?selector=" + java.net.URLEncoder.encode("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

ここでは UTF-8 エンコーディングを使用する必要があります。

ActiveMQ の JMS セレクター

ActiveMQ エンドポイントで JMS セレクターを定義することもできます。以下に例を示します。

```
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");
```

詳細は、ActiveMQ: JMS Selectors および ActiveMQ Message Properties を参照してください。

コンテンツベースルーター

Content-Based Router パターンと Message Dispatcher パターンの基本的な違いは、コンテンツベースルーターがメッセージを物理的に別々の宛先 (リモートのエンドポイント) にディスパッチするのに対し、メッセージディスパッチャーは同じプロセス空間内でローカルにメッセージをディスパッチする点です。Apache Camel では、これらの 2 つのパターンの違いは、ターゲットエンドポイントによって決定されます。コンテンツベースルーターとメッセージディスパッチャーのどちらも、同じルーターロジックを使用して実装されます。ターゲットエンドポイントがリモートの場合、そのルートはコンテンツベースルーターを定義します。ターゲットエンドポイントがプロセス内にある場合、そのルートはメッセージディスパッチャーを定義します。

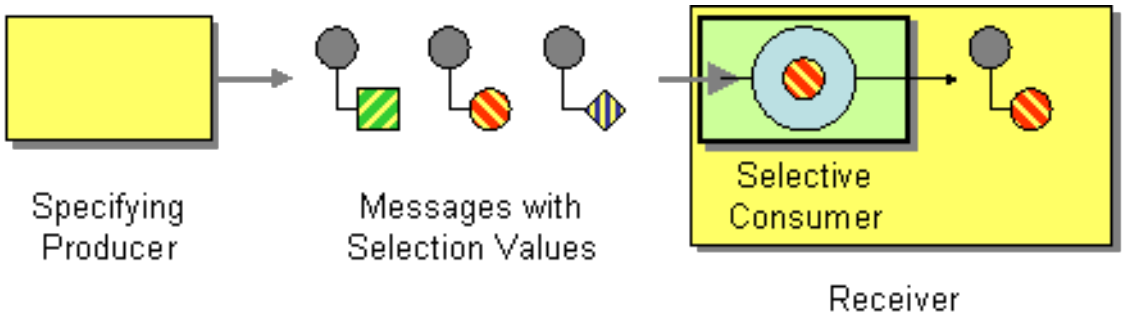
Content-Based Router パターンの使用方法の詳細および例については、「Content-Based Router」 を参照してください。

11.6. SELECTIVE CONSUMER

概要

図11.5「Selective Consumer パターン」 に記載されている **Selective Consumer** パターンは、受信メッセージにフィルターを適用することで、特定の選択基準を満たしたメッセージのみが処理されるようにするコンシューマーを示します。

図11.5 Selective Consumer パターン



Apache Camel に Selective Consumer パターンを実装するには、以下の方法があります。

- JMS セレクター
- ActiveMQ の JMS セレクター
- メッセージフィルター

JMS セレクター

JMS セレクターは、JMS ヘッダーと JMS プロパティーに対する述語の式です。セレクターが **true** に評価されると JMS メッセージはコンシューマーに到達でき、セレクターが **false** に評価されると JMS メッセージはブロックされます。たとえば、キュー **selective** からメッセージを消費し、国コードプロパティーが **US** に等しいメッセージのみを選択するには、以下の Java DSL ルートを使用できます。

```
from("jms:selective?selector=" + java.net.URLEncoder.encode("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

セレクター文字列 **CountryCode='US'** は、クエリーオプションの解析で問題が発生しないように、URL エンコード (UTF-8 文字を使用) されている必要があります。この例では、JMS プロパティー **CountryCode** が送信者によって設定されていることを前提としています。JMS セレクターの詳細は、「JMS セレクター」 を参照してください。

注記

セレクターが JMS キューに適用されると、選択されなかったメッセージはキュー上に残り、同じキューに割り当てられた他のコンシューマーがある場合はそちらから利用できることになります。

ActiveMQ の JMS セレクター

ActiveMQ エンドポイントで JMS セレクターを定義することもできます。以下に例を示します。

```
from("acivemq:selective?selector=" + java.net.URLEncoder.encode("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

詳細は、ActiveMQ: JMS Selectors および ActiveMQ Message Properties を参照してください。

メッセージフィルター

コンシューマーエンドポイントがセレクターをサポートしていない場合は、代わりにフィルタープロセッサーをルートに挿入することで対応できます。たとえば、以下のように、Java DSL を使用して、US 国コードが設定されたメッセージのみを処理する選択的コンシューマーを定義できます。

```
from("seda:a").filter(header("CountryCode").isEqualTo("US")).process(myProcessor);
```

以下のように、XML の設定を使用して同じルートを定義できます。

```
<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>
```

Apache Camel のフィルタープロセッサーについての詳細は、「Message Filter」 を参照してください。

警告

メッセージフィルターを使用して JMS **キュー** からメッセージを選択する場合には、注意が必要です。フィルタープロセッサーを使用する場合、ブロックされたメッセージは単に破棄されます。したがって、メッセージがキューから消費される場合 (キューでは各メッセージは 1 回のみ消費可能、「Competing Consumers」 を参照)、ブロックされたメッセージはまったく処理されないことになります。これは望ましい動作ではない可能性があります。

11.7. DURABLE SUBSCRIBER

概要

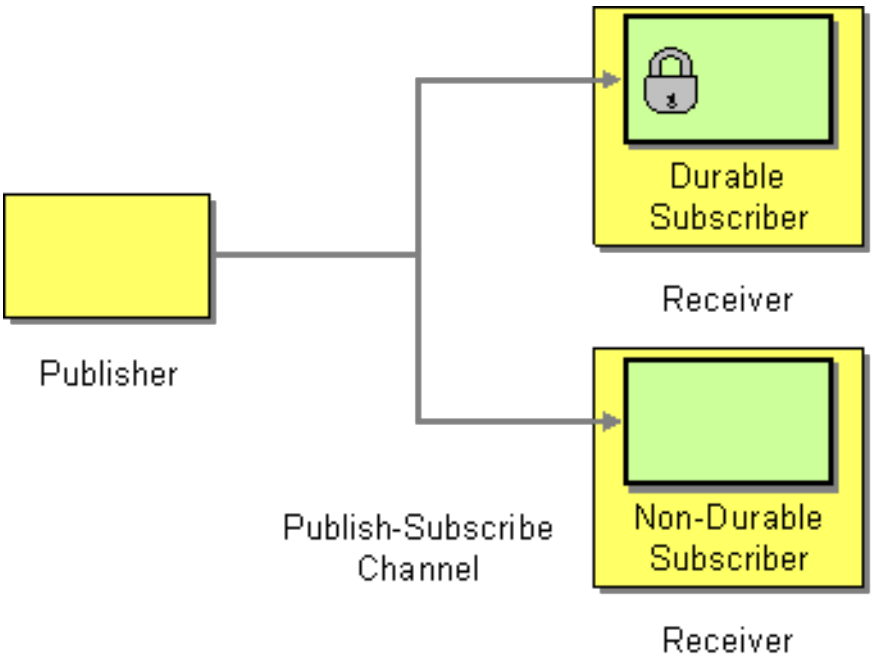
図11.6「Durable Subscriber パターン」 に記載されている **永続サブスクライバー** (Durable Subscriber) は、特定の 「Publish-Subscribe Channel」 チャネルに送信されたすべてのメッセージを受信することを意図したコンシューマーです。これには、コンシューマーがメッセージングシステムから切断されている間に送信されたメッセージも含まれます。そのためには、メッセージングシステムが切断されたコンシューマーに対して後でリプレイするためにメッセージを保存しておく必要があります。また、コンシューマー側には永続サブスクリプションの確立が必要であることを示すメカニズムも必要です。通常、パブリッシュサブスクライブチャネル (またはトピック) には、永続サブスクライバーと非永続サブスクライバーの両方を設定できます。それぞれ以下のように動作します。

- **非永続サブスクライバー** - **接続** と **切断** の 2 つの状態を持つことができます。非永続サブスクライバーがトピックに接続されている間は、トピックのすべてのメッセージをリアルタイムで受信します。しかし、非永続サブスクライバーが切断されている間は、トピックに送信されたメッセージを一切受信しません。

- **永続的サブスクライバー** - **接続** と **非アクティブ** の 2 つの状態を持つことができます。非アク

ティブ状態とは、永続サブスクライバーはトピックから切断されているものの、その間に到達したメッセージを受信するつもりであることを意味します。永続サブスクライバーがトピックに再接続すると、非アクティブ時に送信されたすべてのメッセージがリプレイされます。

図11.6 Durable Subscriber パターン



JMS 永続サブスクライバー

JMS コンポーネントは Durable Subscriber パターンを実装しています。JMS エンドポイントに永続サブスクリプションを設定するには、この接続を識別するための **クライアント ID** と永続サブスクライバーを識別するための **永続サブスクリプション名** を指定する必要があります。たとえば、以下のルートは、クライアント ID が **conn01** で、永続サブスクリプション名が **John.Doe** の JMS トピック **news** に永続サブスクリプションを設定します。

```
from("jms:topic:news?clientId=conn01&durableSubscriptionName=John.Doe").
    to("cxf:bean:newsprocessor");
```

ActiveMQ エンドポイントを使用して永続サブスクリプションを設定することもできます。

```
from("activemq:topic:news?clientId=conn01&durableSubscriptionName=John.Doe").
    to("cxf:bean:newsprocessor");
```

受信メッセージを同時に処理する場合は、SEDA エンドポイントを使用して、以下のように複数の並列セグメントにルートを分散させることができます。

```
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");
```

SEDA コンポーネントは Competing Consumers パターンをサポートするため、各メッセージは一度だけ処理されます。

代替例

もう 1 つの代替方法として、「Message Dispatcher」 または 「Content-Based Router」 を、永続サブスクライバーの File または JPA コンポーネントと組み合わせた後、非永続サブスクライバーの SEDA のようなコンポーネントと組み合わせる方法もあります。

以下は、JMS トピックへの永続サブスクライバーを作成する簡単な例です。

Fluent Builder (流れるようなビルダー) の使用

```
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");
```

Spring XML エクステンション の使用

```
<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>
```

以下は、JMS 永続サブスクライバーのもう 1 つの例ですが、今回は 仮想トピック を使用しています (AMQ で永続サブスクリプションを使用する場合に推奨されます)。

Fluent Builder (流れるようなビルダー) の使用

```
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");
```

Spring XML エクステンション の使用

```
<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

概要

Idempotent Consumer パターンは、重複したメッセージをフィルターするために使用され ます。たとえば、システム障害により、メッセージングシステムとコンシューマーエンドポイント間の接続が突然失われるシナリオについて考えてみましょう。メッセージングシステムがメッセージの送信中だった場合は、コンシューマーが最後のメッセージを受け取ったかどうかが不明な可能性があります。配信の信頼性を向上させるために、メッセージングシステムは、接続が再確立され次第、メッセージを再配信することを決定する場合があります。ただし、これにより、コンシューマーが重複したメッセージを受信する可能性があり、場合によっては、メッセージの重複により、望ましくない結果 (口座から 2 回お金が引き落とされるなど) となる可能性があります。このシナリオでは、メッセージストリームから不必要な重複を取り除くために、べき等コンシューマーを使用します。

Camel は以下のべき等コンシューマー実装を提供します。

- **MemoryIdempotentRepository**
- KafkaIdempotentRepository
- File
- Hazelcast
- SQL
- JPA

インメモリーキャッシュを持つべき等コンシューマー

Apache Camel では、Idempotent Consumer パターンが **idempotentConsumer()** プロセッサーによって実装されます。これには、以下の 2 つの引数を持ちます。

- **messageIdExpression**: 現在のメッセージのメッセージ ID 文字列を返す式。
- **messageIdRepository**: 受信したすべてのメッセージの ID を格納するメッセージ ID リポジトリーへの参照。

各メッセージを受信すると、べき等コンシューマープロセッサーは、リポジトリーの現在のメッセージ ID を検索し、このメッセージが以前にあったものかを確認します。yes の場合、メッセージは破棄されます。no の場合、メッセージは渡され、その ID がリポジトリーに追加されます。

例11.1「インメモリーキャッシュを使用した重複メッセージのフィルターリング」 に表示されているコードは、**TransactionID** ヘッダーを使用して重複をフィルターリングします。

例11.1 インメモリーキャッシュを使用した重複メッセージのフィルターリング

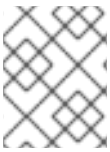
```
import static
org.apache.camel.processor.idempotent.MemoryMessageIdRepository.memoryMessageIdRepositor
```

```
y;
...
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a")
          .idempotentConsumer(
            header("TransactionID"),
            memoryMessageIdRepository(200)
          ).to("seda:b");
    }
};
```

memoryMessageIdRepository(200) が呼び出されると、最大 200 個のメッセージ ID を保持できるインメモリーキャッシュが作成されます。

XML 設定を使用してべき等コンシューマーを定義することもできます。たとえば、以下のように XML で前述のルートを定義できます。

```
<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>
```



注記

Camel 2.17 より、べき等リポジトリーはオプションのシリアライズされたヘッダーをサポートします。

JPA リポジトリーを使用したべき等コンシューマー

インメモリーキャッシュは、メモリー不足になりやすく、クラスター環境で機能しないという欠点があります。これらの欠点に対処するために、代わりに Java Persistent API (JPA) ベースのリポジトリーを使用できます。JPA メッセージ ID リポジトリーは、オブジェクト指向のデータベースを使用してメッセージ ID を保存します。たとえば、以下のように、べき等コンシューマーの JPA リポジトリーを使用するルートを定義できます。

```
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 メッセージ ID リポジトリーは 2 つの引数で初期化されます。

- **JpaTemplate** インスタンス: JPA データベースのハンドルを指定します。
- プロセッサー名: 現在のべき等コンシューマープロセッサーを特定します。

SpringRouteBuilder.bean() メソッドは、Spring XML ファイルで定義された Bean を参照するショートカットです。**JpaTemplate** Bean は、基礎となる JPA データベースに対するハンドルを提供します。この Bean の設定方法に関する詳細は、JPA のドキュメントを参照してください。

JPA リポジトリーの設定に関する詳細は、JPA Component ドキュメント、Spring JPA ドキュメント、および Camel JPA unit test を参照してください。

Spring XML の例

以下の例では、**myMessageId** ヘッダーを使用して重複をフィルターします。

```
<!-- 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>
```

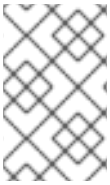
JDBC リポジトリーを使用したべき等コンシューマー

JDBC リポジトリーは、Idemopotent Consumer パターンのメッセージ ID の格納でもサポートされます。JDBC リポジトリーの実装は SQL コンポーネントによって提供されるので、Maven ビルドシステムを使用している場合は、**camel-sql** アーティファクトに依存関係を追加します。

SQL データベースへの接続をインスタンス化するために、Spring persistence API から **SingleConnectionDataSource** JDBC ラッパークラスを使用できます。たとえば、HyperSQL データベースインスタンスへの JDBC 接続をインスタンス化するには、以下の JDBC データソースを定義できます。

```
<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>
```



注記

前述の JDBC データソースは HyperSQL **mem** プロトコルを使用し、メモリーのみのデータベースインスタンスを作成します。これは、HyperSQL データベースの簡易実装で、永続的では **ありません**。

前述のデータソースを使用して、JDBC メッセージ ID リポジトリーを使用する Idempotent Consumer パターンを以下のように定義できます。

```
<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>
```

ルートで重複メッセージを処理する方法

Camel 2.8 から利用可能

skipDuplicate オプションを **false** に設定して、べき等コンシューマーに重複したメッセージもルーティングするように指示できるようになりました。ただし、「エクスチェンジ」 プロパティーを true に設定することにより、重複メッセージが duplicate とマークされます。重複メッセージを検出し、処理するために 「Content-Based Router」 または 「Message Filter」 が使用されます。

以下の例では、メッセージを重複エンドポイントに送信するために 「Message Filter」 を使用し、メッセージのルーティングを停止します。

```
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 の例を以下に示します。

```
 <!-- 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>
```

データグリッドを使用したクラスター環境で重複メッセージを処理する方法

クラスター環境で Camel を実行している場合、メモリーのべき等リポジトリーでは機能しません (上記を参照)。中央のデータベースを設定するか、Hazelcast データ グリッドを基にしたべき等コンシューマー実装を使用できます。Hazelcast は、マルチキャスト経由のノード (デフォルトは tcp-ip の Hazelcast を設定) を見つけ、マップベースのリポジトリーを自動的に作成します。

```
HazelcastIdempotentRepository idempotentRepo = new HazelcastIdempotentRepository("myrepo");

from("direct:in").idempotentConsumer(header("messageId"), idempotentRepo).to("mock:out");
```

各メッセージ ID を保持するリポジトリーの期間を定義する必要があります (デフォルトは削除しません)。メモリーが不足しないようにするには、Hazelcast 設定 に基づいてエビクションストラテジーを作成する必要が あります。詳細は、Hazelcast を参照してください。

http://camel.apache.org/hazelcast-idempotent-repository-tutorial.html[Idempotent Repository tutorial] を参照してください。

Apache Karaf を使用して、2 つのクラスターノードにべき等リポジトリーを設定する方法について詳しく説明します。

オプション

Idempotent Consumer には以下のオプションがあります。

オプション	デフォルト	説明
eager	**true**	**Camel 2.0:** Eager は、エクスチェンジの処理前後に Camel がメッセージをリポジトリーに追加するかどうかを制御します。これを事前に有効にしておくと、メッセージが現在進行中であっても、Camel は重複メッセージを検出できます。無効にすると、Camel はメッセージが正常に処理されたときにのみ重複を検出します。
messageIdRepositoryRef	**null**	レジストリーで検索するための **IdempotentRepository** への参照。XML DSL を使用する場合は、このオプションは必須です。
skipDuplicate	**true**	**camel 2.8:** 重複メッセージをスキップするかどうかを設定します。**false** に設定すると、メッセージは継続されます。ただし、**Exchange.DUPLICATE_MESSAG** エクスチェンジプロパティーが **Boolean.TRUE** 値に設定されている場合、「エクスチェンジ」 は重複としてマークされます。

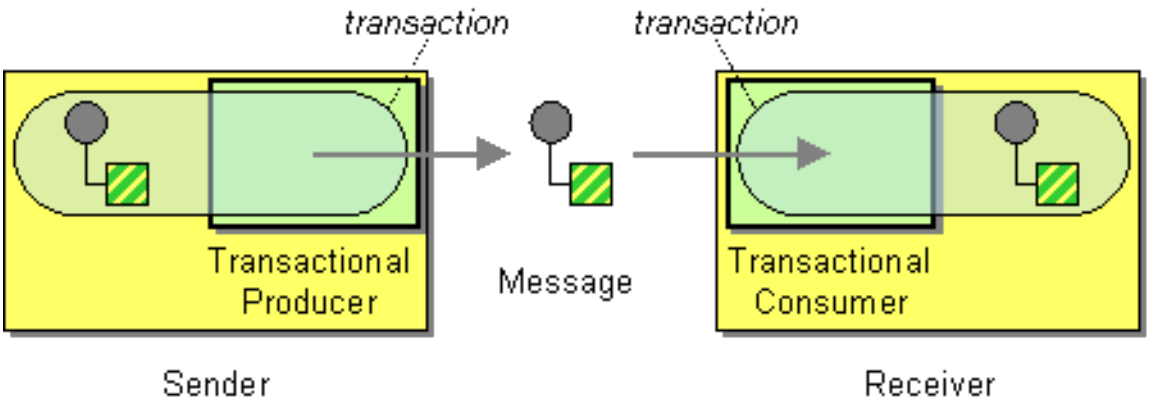
completionEager	**false**	camel 2.16: エクスチェンジが完了すると、Idempotent コンシューマーの Eager を完了するかどうかを設定します。 **completeEager** オプションを true に設定すると、エクスチェンジが Idempotent Consumer パターンブロックの末尾に到達すると、べき等コンシューマーによって完了がトリガーされます。ただし、エンドブロック後もエクスチェンジがルーティングを継続しても、べき等コンシューマーの状態には影響しません。 **completeEager** オプションを false に設定すると、エクスチェンジが完了してルーティングされた後に、べき等コンシューマーによって完了がトリガーされます。ただし、ブロックの終了後もエクスチェンジがルーティングを継続する場合、べき等コンシューマーにも影響します。たとえば、エクスチェンジが失敗した場合の例外により、べき等コンシューマーの状態がロールバックされます。

11.9. TRANSACTIONAL CLIENT

概要

図11.7「Transactional Client パターン」 に示される **Transactional Client**パターンは 、トランザクションに参加できるメッセージングエンドポイントを参照します。Apache Camel は Spring transaction management を使用してトランザクションをサポートします。

図11.7 Transactional Client パターン



トランザクション指向のエンドポイント

すべての Apache Camel エンドポイントがトランザクションをサポートする訳ではありません。サポートするものは、**トランザクション指向のエンドポイント** (または TOE) と呼ばれます。たとえば、JMS コンポーネントと ActiveMQ コンポーネントの両方がトランザクションをサポートします。

コンポーネントでトランザクションを有効にするには、コンポーネントを **CamelContext** に追加する前に、適切な初期化を実行する必要があります。すなわち、トランザクションコンポーネントを明示的に初期化するためにコードを記述する必要があります。

その他の参考資料

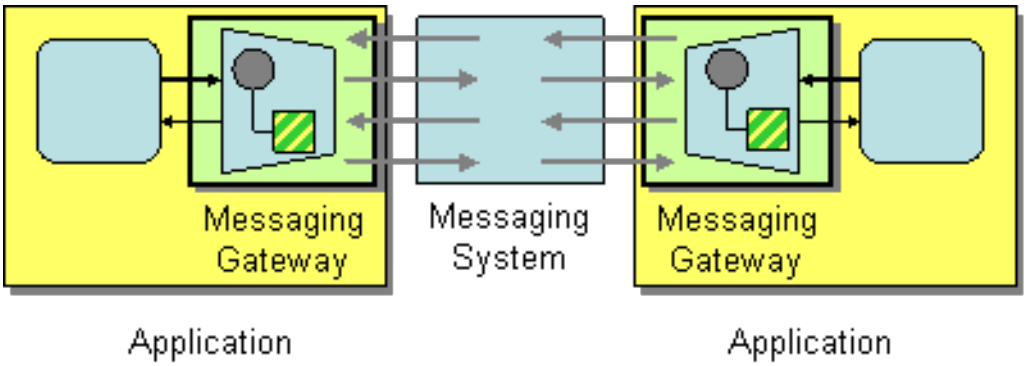
Apache Camel でのトランザクションの設定に関する詳細は、本ガイドの対象外となります。トランザクションの使用方法に関する詳細は、Apache Camel の**Transaction Guide**を参照してください。

11.10. MESSAGING GATEWAY

概要

図11.8「Messaging Gateway パターン」 に記載されている **Messaging Gateway** パターンは、メッセージングシステムの API がアプリケーションレベルでプログラマーから隠されているメッセージングシステムとの統合方法を示しています。より一般的な例として、プログラマーが認識せずに同期メソッド呼び出しをリクエスト/リプライメッセージ交換に変換する場合が挙げられます。

図11.8 Messaging Gateway パターン



以下の Apache Camel コンポーネントは、このようなメッセージングシステムとの統合を提供します。

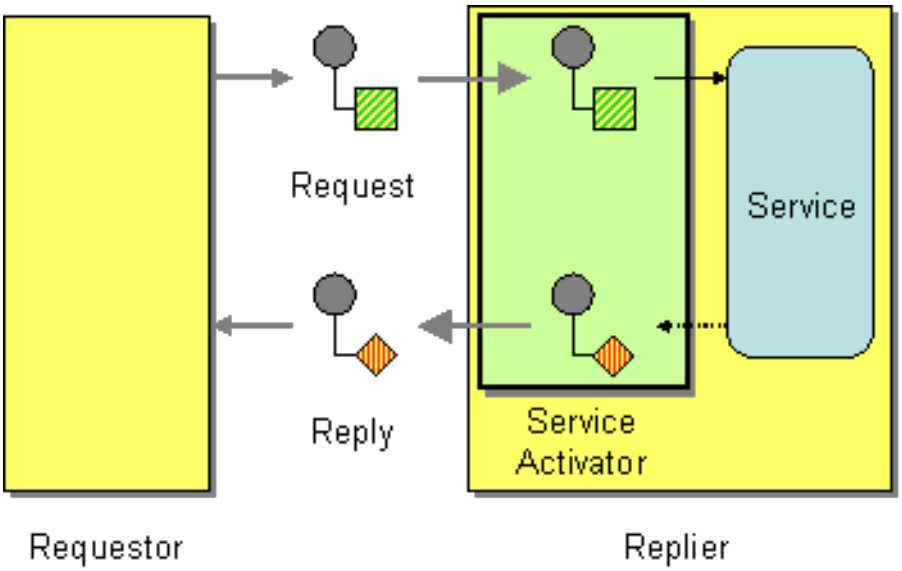
- CXF
- Bean コンポーネント

11.11. SERVICE ACTIVATOR

概要

図11.9「Service Activator パターン」 に示されている **Service Activator** パターンは、受信したリクエストメッセージへの応答でサービスの操作が呼び出されるシナリオを説明します。Service Activator は、呼び出す操作を特定し、操作のパラメーターとして使用するデータを抽出します。最後に、Service Activator は、メッセージから抽出したデータを使用して操作を呼び出します。操作呼び出しは一方向 (リクエストのみ) または双方向 (リクエスト/リプライ) のいずれかになります。

図11.9 Service Activator パターン



多くの点で、Service Activator は従来のリモートプロシージャーコール (RPC) に類似しています。ここでは、操作呼び出しはメッセージとしてエンコードされます。主な違いは、Service Activator がより柔軟である必要があることです。RPC フレームワークは、リクエストおよびリプライメッセージエンコーディングを標準化します (たとえば、Web サービスの操作は SOAP メッセージとしてエンコードされます)。一方、Service Activator は通常、メッセージングシステムとサービスの操作間のマッピングを即応する必要があります。

Bean インテグレーション

Apache Camel が Service Activator パターンのサポートに提供する主なメカニズムが **Bean インテグレーション** です。Bean インテグレーション は、受信メッセージを Java オブジェクトのメソッド呼び出しにマッピングするための一般的なフレームワークを提供します。たとえば、Java fluent DSL は、**bean()** および **beanRef()** プロセッサーを提供し、ルートに挿入して、登録された Java Bean のメソッドを呼び出すことができます。メッセージデータの Java メソッドパラメーターへの詳細なマッピングは、 **Bean バインディング**によって決定され、Bean クラスにアノテーションを追加することで実装できます。

たとえば、JMS/ActiveMQ キューで受信されるサービスリクエストに対して、**BankBean.getUserAccBalance()** の Java メソッドを呼び出す以下のルートについて考えてみましょう。

```
from("activemq:BalanceQueries")
  .setProperty("userid", xpath("/Account/BalanceQuery/UserID").stringResult())
  .beanRef("bankBean", "getUserAccBalance")
  .to("velocity:file:src/scripts/acc_balance.vm")
  .to("activemq:BalanceResults");
```

ActiveMQ エンドポイント **activemq:BalanceQueries** から引き出されたメッセージは、銀行口座のユーザー ID を提供する単純な XML 形式です。以下に例を示します。

```
<?xml version='1.0' encoding='UTF-8'?>
<Account>
  <BalanceQuery>
    <UserID>James.Strachan</UserID>
  </BalanceQuery>
</Account>
```

ルートの最初のプロセッサー **setProperty()** は、**In** メッセージからユーザー ID を抽出し、**userid** エクスチェンジプロパティーに保存します。**In** ヘッダーは Bean の呼び出し後に利用できないため、プロパティーに保存することが推奨されます。

サービスのアクティベーションの手順は **beanRef()** プロセッサーによって実行されます。このプロセッサーは、受信メッセージを **bankBean** の Bean ID で識別される Java オブジェクトの **getUserAccBalance()** メソッドにバインドします。以下のコードは、**BankBean** クラスの実装例を示しています。

```
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;
        }
      }
}
```

message data to method パラメーターのバインディングは、**UserID** XML 要素の内容を **user** method パラメーターに注入する **@XPath** アノテーションによって有効にされます。呼び出しの完了時に、戻り値は **Out** メッセージのボディーに挿入され、ルートの次のステップのために **In** メッセージにコピーされます。Bean が **beanRef()** プロセッサーにアクセスできるようにするには、Spring XML でインスタンスをインスタンス化する必要があります。たとえば、以下の行を **META-INF/spring/camel-context.xml** 設定ファイルに追加して Bean をインスタンス化できます。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <bean id="bankBean" class="tutorial.BankBean"/>
</beans>
```

Bean ID **bankBean** は、レジストリーでこの Bean インスタンスを特定します。

Bean 呼び出しの出力は、適切にフォーマットされた結果メッセージを生成するために Velocity テンプレートに注入されます。Velocity エンドポイント **velocity:file:src/scripts/acc_balance.vm** は、以下の内容を含む velocity スクリプトの場所を指定します。

```
<?xml version='1.0' encoding='UTF-8'?>
<Account>
  <BalanceResult>
    <UserID>${exchange.getProperty("userid")}</UserID>
    <Balance>${body}</Balance>
  </BalanceResult>
</Account>
```

エクスチェンジインスタンスは Velocity 変数 **exchange** として利用できます。これにより、**${exchange.getProperty("userid")}** を使用して **userid** エクスチェンジプロパティーを取得できます。現在の **In** メッセージのボディー **${body}** には、**getUserAccBalance()** メソッド呼び出しの結果が含まれます。

第12章 システム管理

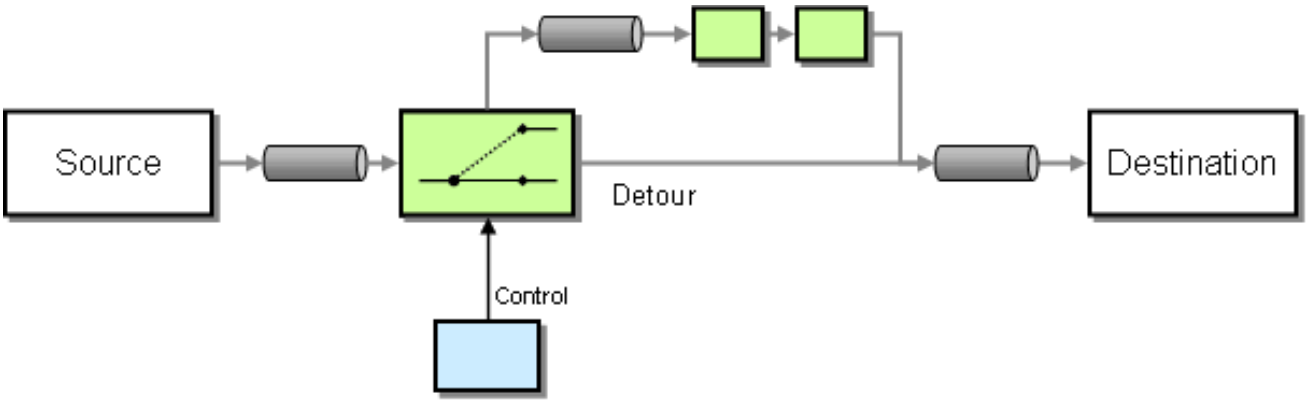
概要

システム管理パターンは、メッセージングシステムを監視、テスト、および管理する方法を表しています。

12.1. DETOUR

Detour

3章エンタープライズ統合パターンの導入 からの Detour では、制御条件が満たされる場合に追加のステップによりメッセージを送信できます。これは、追加の検証、テスト、および必要時のコードのデバッグを行う場合に役立ちます。



例

この例では、ルートの途中で **mock:detour** のエンドポイントまで条件付きで迂回する **from("direct:start").to("mock:result")** のようなルートを基本的に持っています。

```
from("direct:start").choice()
    .when().method("controlBean", "isDetour").to("mock:detour").end()
    .to("mock:result");
```

Spring XML エクステンション の使用

```
<route>
  <from uri="direct:start"/>
    <choice>
      <when>
        <method bean="controlBean" method="isDetour"/>
 <to uri="mock:detour"/>
      </when>
    </choice>
    <to uri="mock:result"/>
  </split>
</route>
```

detour がオンまたはオフであるかどうかは **ControlBean** によって決定されます。したがって、detour がオンのとき、メッセージは **mock:detour** にルーティングされ、続いて **mock:result** にルーティングされます。detour がオフになると、メッセージは **mock:result** にルーティングされます。

詳細は、以下のサンプルソースを確認してください。

camel-core/src/test/java/org/apache/camel/processor/DetourTest.java

12.2. LOGEIP

概要

Apache Camel では、ルートでログを出力する方法は複数あります。

- **log** DSL コマンドの使用。
- メッセージコンテンツをログに記録できる Log コンポーネントの使用。
- メッセージフローを追跡するトレーサーの使用。
- **Processor** または Bean エンドポイントを使用した Java でのロギング。

LOG DSL コマンドと LOG コンポーネントの違い

log DSL は非常に軽量で、**Starting to do ...** のように人の作業をロギングします。これは、Simple 言語に基づいたメッセージのみにログを記録できます。対照的に、Log コンポーネントは、フル機能のロギングコンポーネントです。Log コンポーネントはメッセージ自体をログに記録でき、ロギングを制御する多くの URI オプションがあります。

Java DSL の例

Apache Camel 2.2 以降、**log** DSL コマンドを使用して、Simple 式言語を使ってランタイム時にログメッセージを作成できます。たとえば、以下のように、ルート内にログメッセージを作成できます。

```
from("direct:start").log("Processing ${id}").to("bean:foo");
```

このルートは、ランタイムに **String** フォーマットメッセージを作成します。ルート ID をログ名として使用し、ログメッセージは **INFO** レベルでログに記録されます。デフォルトでは、ルートには **route-1**、**route-2** などの連続した名前が付けられます。ただし、DSL コマンド **routeId("myCoolRoute")** を使用して、カスタムルート ID を指定することもできます。

ログ DSL は、ロギングレベルとログ名を明示的に設定できるようにするバリアントも提供します。たとえば、ロギングレベルを明示的に **LoggingLevel.DEBUG** に設定するには、以下のようにログ DSL を呼び出すことができます。

ログ DSL には、ロギングレベルや名前を設定するためのオーバーロードされたメソッドもあります。

```
from("direct:start").log(LoggingLevel.DEBUG, "Processing ${id}").to("bean:foo");
```

ログ名を **fileRoute** に設定するには、以下のようにログ DSL を呼び出すことができます。

```
from("file://target/files").log(LoggingLevel.DEBUG, "fileRoute", "Processing file
${file:name}").to("bean:foo");
```

XML DSL の例

XML DSL では、ログ DSL は **log** 要素によって表され、ログメッセージは以下のように **message** 属性を Simple 式に設定することで指定されます。

```
<route id="foo">
    <from uri="direct:foo"/>
    <log message="Got ${body}"/>
    <to uri="mock:foo"/>
</route>
```

log 要素は、**message**、**loggingLevel**、および **logName** 属性をサポートします。以下に例を示します。

```
<route id="baz">
    <from uri="direct:baz"/>
    <log message="Me Got ${body}" loggingLevel="FATAL" logName="cool"/>
    <to uri="mock:baz"/>
</route>
```

グローバルログ名

ルート ID はデフォルトのログ名として使用されます。**Apache Camel 2.17** 以降、logname パラメーターを設定してログ名を変更できます。

Java DSL では、以下の例のようにログ名を設定します。

```
CamelContext context = ...
context.getProperties().put(Exchange.LOG_EIP_NAME, "com.foo.myapp");
```

XML では、以下のようにログ名を設定します。

```
<camelContext ...>
  <properties>
    <property key="CamelLogEipName" value="com.foo.myapp"/>
  </properties>
```

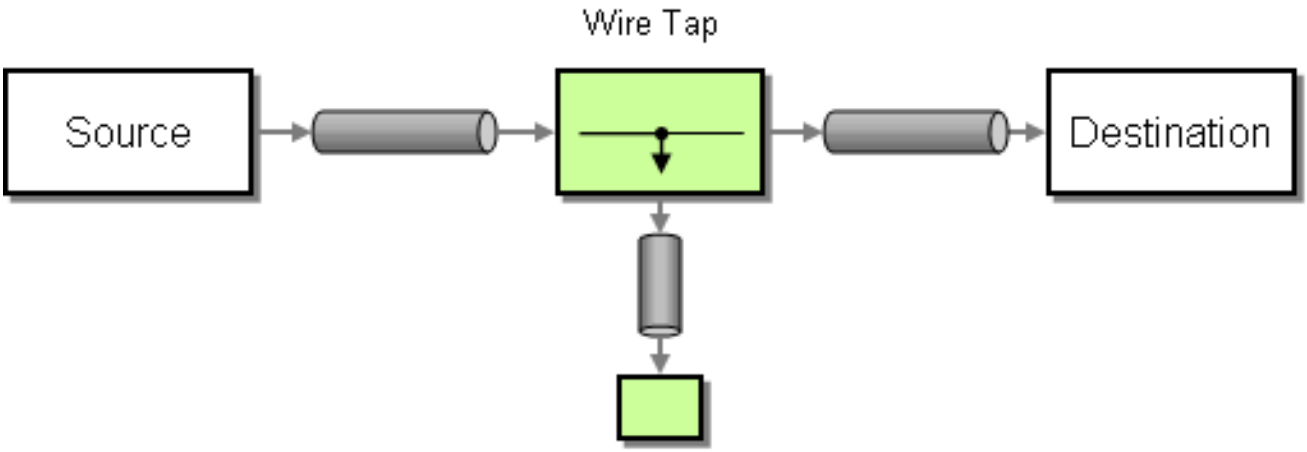
複数のログがあり、すべてのログに同じログ名を設定する場合は、各ログに設定を追加する必要があります。

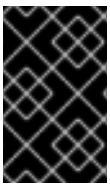
12.3. WIRE TAP

Wire Tap

Wire Tapパターンは 図12.1「Wire Tap パターン」 に記載されているように、元のメッセージが宛先に転送される一方で、メッセージのコピーを別の Tap の場所にルーティングできます。

図12.1 Wire Tap パターン





ストリーム

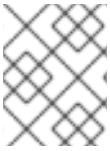
ストリームメッセージボディーに **WireTap** を実行する場合、ストリームキャッシング を有効にして、メッセージボディーを再読み込みできるようにすることを検討してください。詳細は、Stream Caching を参照してください。

wireTap ノード

Apache Camel 2.0 では、Wire Tap を実行する **wireTap** ノードが導入されました。**wireTap** ノードは、交換パターンが **InOnly** に設定されたタップされたエクスチェンジに元のエクスチェンジをコピーします。これは、タップされたエクスチェンジを **一方向** で伝播する必要があるためです。タップされたエクスチェンジは個別のスレッドで処理されるため、メインルートと同時に実行できます。

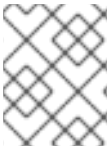
wireTap は、エクスチェンジのタップに関して、以下の 2 つの異なるアプローチをサポートします。

- 元のエクスチェンジのコピーをタップします
- タップされたエクスチェンジをカスタマイズできるように、新しいエクスチェンジインスタンスをタップします



注記

Camel 2.16 以降、Wire Tap 宛先にエクスチェンジを送信するときに Wire Tap EIP がイベント通知を送信します。



注記

Camel 2.20 の時点では、Wire Tap EIP はシャットダウン中にすべての実行中の Wire Tap エクスチェンジを終了します。

元のエクスチェンジのコピーをタップします。

Java DSL を使用

```
from("direct:start")
    .to("log:foo")
    .wireTap("direct:tap")
    .to("mock:result");
```

Spring XML エクステンションを使用

```
<route>
    <from uri="direct:start"/>
    <to uri="log:foo"/>
    <wireTap uri="direct:tap"/>
    <to uri="mock:result"/>
</route>
```

元のエクスチェンジのコピーをタップおよび変更

Java DSL を使用すると、Apache Camel はプロセッサーまたは式を使用して元のエクスチェンジのコピーを変更することができます。プロセッサーを使用すると、プロパティーやヘッダーなどを設定することができるため、エクスチェンジの配置を完全に制御することができます。式は、**In** のメッセージボディーの変更にのみ使用できます。

たとえば、**プロセッサー** を使用して元のエクスチェンジのコピーを変更するには、以下を実行します。

```
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");
```

また、**式** を使用して元のエクスチェンジのコピーを変更するには、以下を実行します。

```
from("direct:start")
    .wireTap("direct:foo", constant("Bye World"))
    .to("mock:result");

from("direct:foo").to("mock:foo");
```

Spring XML エクステンションを使用すると、**プロセッサー** アプローチを使用して元のエクスチェンジのコピーを変更できます。ここで、**processorRef** 属性は **myProcessor** ID を持つ Spring Bean を参照します。

```
<route>
    <from uri="direct:start2"/>
    <wireTap uri="direct:foo" processorRef="myProcessor"/>
    <to uri="mock:result"/>
</route>
```

また、**式** を使用して元のエクスチェンジのコピーを変更するには、以下を実行します。

```
<route>
    <from uri="direct:start"/>
    <wireTap uri="direct:foo">
        <body><constant>Bye World</constant></body>
    </wireTap>
    <to uri="mock:result"/>
</route>
```

- 新しいエクスチェンジインスタンスをタップします

コピーフラグを **false** (デフォルトは **true**) に設定すると、新規のエクスチェンジインスタンスで Wire Tap を定義できます。この場合、Wire Tap 用に最初の空のエクスチェンジが作成されます。

たとえば、**プロセッサー** を使用して新しいエクスチェンジインスタンスを作成するには、以下のコマンドを実行します。

```
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");
```

2 番目の **wireTap** 引数は、コピーフラグを **false** に設定し、元のエクスチェンジがコピー **されず**、代わりに空のエクスチェンジが作成されることを示します。

式 を使用して新しいエクスチェンジインスタンスを作成するには、以下を実行します。

```
from("direct:start")
    .wireTap("direct:foo", false, constant("Bye World"))
    .to("mock:result");

from("direct:foo").to("mock:foo");
```

Spring XML エクステンションを使用すると、**wireTap** 要素の **copy** 属性を **false** に設定することで、新しいエクスチェンジが作成されることを示すことができます。

プロセッサー アプローチを使用して、新しいエクスチェンジインスタンスを作成するには、次のように、**processorRef** 属性が **myProcessor** ID を持つ Spring Bean を参照します。

```
<route>
    <from uri="direct:start2"/>
    <wireTap uri="direct:foo" processorRef="myProcessor" copy="false"/>
    <to uri="mock:result"/>
</route>
```

式 を使用して新しいエクスチェンジインスタンスを作成するには、以下を実行します。

```
<route>
    <from uri="direct:start"/>
    <wireTap uri="direct:foo" copy="false">
        <body><constant>Bye World</constant></body>
    </wireTap>
    <to uri="mock:result"/>
</route>
```

DSL での新規エクスチェンジの送信およびヘッダーの設定

Camel 2.8 から利用可能

「Wire Tap」 を使用して新しいメッセージを送信する場合、DSLから パートII「ルーティング式と述語言語」 を使用してメッセージボディーのみを設定できます。新しいヘッダーも設定する必要がある場合には、「プロセッサー」 を使用する必要があります。Camel 2.8 以降では、DSL にもヘッダーを設定できるようになりました。

以下は、次の条件で新しいメッセージを送信する例になります。

- メッセージボディーは Bye World
- キー id のあるヘッダー (値 は 123)
- キー date のあるヘッダー (値は現在の日付)

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 は、メッセージボディーとヘッダーの設定方法が Java DSL とは若干異なります。XML では、以下のように <body> および <setHeader> を使用します。

```
<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>
```

URI の使用

Wire Tap は、静的および動的エンドポイント URI をサポートします。静的エンドポイント URI は **Camel 2.20** で利用できます。

以下の例は、ヘッダー ID がキュー名の一部である JMS キューに Wire Tap を実行する方法を示しています。

```
from("direct:start")
  .wireTap("jms:queue:backup-${header.id}")
  .to("bean:doSomething");
```

動的エンドポイント URI の詳細は、「Dynamic To」 を参照してください。

メッセージの準備時にカスタムロジックを実行するための onPrepare の使用

Camel 2.8 から利用可能

詳細は、「Multicast」 を参照してください。

オプション

wireTap DSL コマンドは、以下のオプションをサポートします。

名前	デフォルト値	説明
uri		Wire Tap メッセージの送信先のエンドポイント URI。**uri** または **ref** のいずれかを使用する必要があります。
ref		Wire Tap メッセージの送信先のエンドポイントを参照します。**uri** または **ref** のいずれかを使用する必要があります。
executorServiceRef		Wire Tap メッセージを処理する際に使用するカスタム「スレッドモデル」 を参照します。設定されていない場合、Camel はデフォルトのスレッドプールを使用します。
processorRef		新しいメッセージを作成するために使用するカスタム「プロセッサー」 を参照します (送信モードなど)。以下を参照してください。
copy	**true**	**camel 2.3**: メッセージに Wire Tap を実行する際に使用する「エクスチェンジ」 をコピーします。

onPrepareRef		**Camel 2.8:** Wire Tap のために「エクスチェンジ」 のコピーを用意するカスタム「プロセッサー」を参照します。これにより、必要に応じてメッセージのペイロードをディープクローンするなど、カスタムロジックを実行できます。

パート II. ルーティング式と述語言語

本ガイドでは、Apache Camel でサポートされる評価言語で使われている基本的な構文を説明します。

第13章 はじめに

概要

本章では、Apache Camel でサポートされているすべての式言語の概要を説明します。

13.1. 言語の概要

式および述語言語の表

表13.1「式および述語」 は、式言語と述語言語を呼び出すための異なる構文の概要をまとめています。

表13.1 式および述語

言語	Static メソッド	Fluent DSL メソッド	XML 要素	アノテーション	アーティファクト
カスタマポータルの Apache Camel 開発ガイド の Bean インテグレーション を参照してください。	**bean()**	**EIP().method()**	**メソッド**	**@Bean**	Camel core
14章*定数*	**constant()**	**EIP().constant()**	**constant**	**@Constant**	Camel core
15章*EL*	**el()**	**EIP().el()**	**el**	**@EL**	**camel-juel**
17章*Groovy*	**groovy()**	**EIP().groovy()**	**groovy**	**@Groovy**	**camel-groovy**
18章ヘッダー	**header()**	**EIP().header()**	**header**	**@Header**	Camel core
19章*JavaScript*	**javaScript()**	**EIP().javaScript()**	**javaScript**	**@JavaScript**	**camel-script**
20章*JoSQL*	**sql()**	**EIP().sql()**	**sql**	**@SQL**	**camel-josql**
21章*JsonPath*	なし	**EIP().jsonpath()**	**jsonpath**	**@JsonPath**	**camel-jsonpath**
22章*JXPath*	なし	**EIP().jxpath()**	**jxpath**	**@JXPath**	**camel-jxpath**
23章*MVEL*	**mvel()**	**EIP().mvel()**	**mvel**	**@MVEL**	**camel-mvel**

言語	Static メソッド	Fluent DSL メソッド	XML 要素	アノテーション	アーティファクト
24章*Object-Graph Navigation Language (OGNL)*	**ognl()**	**EIP().ognl()**	**ognl**	**@OGNL**	**camel-ognl**
25章*PHP (非推奨)*	**php()**	**EIP().php()**	**php**	**@PHP**	**camel-script**
26章 *エクスチェンジプロパティー*	**property()**	**EIP().property()**	**プロパティー**	**@Property**	Camel core
27章*Python (非推奨)*	**python()**	**EIP().python()**	**python**	**@Python**	**camel-script**
28章*Ref*	**ref()**	**EIP().ref()**	**ref**	**該当なし**	Camel core
29章*Ruby (非推奨)*	**ruby()**	**EIP().ruby()**	**ruby**	**@Ruby**	**camel-script**
30章*Simple 言語*/16章*File 言語*	**simple()**	**EIP().simple()**	**simple**	**@Simple**	Camel core
31章*SpEL*	**spel()**	**EIP().spel()**	**spel**	**@SpEL**	**camel-spring**
32章*XPath 言語*	**xpath()**	**EIP().xpath()**	**xpath**	**@XPath**	Camel core
33章*XQuery*	**xquery()**	**EIP().xquery()**	**xquery**	**@XQuery**	**camel-saxon**

13.2. 式言語の呼び出し方法

前提条件

特定の式言語を使用する前に、必要な JAR ファイルがクラスパス上にあることを確認してください。使用する言語が Apache Camel Core に含まれていない場合、関連する JAR をクラスパスに追加する必要があります。

Maven ビルドシステムを使用している場合は、関連する依存関係を POM ファイルに追加するだけで、JAR をビルド時のクラスパスに含むことができます。たとえば、Ruby 言語を使いたい場合は、以下の依存関係を POM ファイルに追加します。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-groovy</artifactId>
  <!-- Use the same version as your Camel core version -->
  <version>${camel.version}</version>
</dependency>
```

アプリケーションを Red Hat Fuse OSGi コンテナーにデプロイする場合は、関連する言語機能がインストールされていることを確認する必要があります (機能の名前は該当する Maven アーティファクトの名前になります)。たとえば、OSGi コンテナーで Groovy 言語を使用するには、以下の OSGi コンソールコマンドを入力して **camel-groovy** 機能をインストールしておく必要があります。

```
karaf@root> features:install camel-groovy
```

注記

ルートで式や述語を使用している場合は、**resource:classpath:path** または **resource:file:path** を使用して、外部リソースとして値を参照します。たとえば、**resource:classpath:com/foo/myscript.groovy** です。

Camel on EAP デプロイメント

camel-groovy コンポーネントは、Camel on EAP (Wildfly Camel) フレームワークによってサポートされており、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上でシンプルなデプロイモデルを提供します。

呼び出す方法

表13.1「式および述語」 で示されているように、式言語を呼び出すための構文は、それが使用されるコンテキストによって異なります。以下の方法で式言語を呼び出すことができます。

- Static メソッド
- Fluent DSL メソッド
- XML 要素
- アノテーション

Static メソッド

ほとんどの言語は、**org.apache.camel.Expression** 型や **org.apache.camel.Predicate** 型が想定される **あらゆる** コンテキストで使用できる Static メソッドを定義します。Static メソッドは、文字列式 (または述語) を引数とし、**Expression** オブジェクト (**Predicate** オブジェクトでもある) を返します。

たとえば、XML 形式のメッセージを処理するコンテンツベースルーターを実装するには、以下のように **/order/address/countryCode** 要素の値に基づいてメッセージをルーティングできます。

```
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");
```

Fluent DSL メソッド

Java Fluent DSL は、式言語を呼び出す別のスタイルとなります。式をエンタープライズ統合パターン (EIP) の引数として指定する代わりに、DSL コマンドのサブ句で式を指定できます。たとえば、XPath 式を **filter(xpath("Expression"))** として呼び出す代わりに、式を **filter().xpath("Expression")** として呼び出すことができます。

たとえば、前述のコンテンツベースのルーターは、以下のように、この形式の呼び出しで再実装できます。

```
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");
```

XML 要素

関連する XML 要素の中に文字列の式を入れることで、XML 内で式言語を呼び出すこともできます。

たとえば、XML 内で XPath を呼び出すための XML 要素は **xpath** (標準の Apache Camel 名前空間に属している) です。XML DSL でコンテンツベース ルーターの実装では、以下のように XPath 式を使用することができます。

```
<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>
```

あるいは、**language** 要素を使用して言語式を指定することもできます。この場合、**language** 属性で言語名を指定します。たとえば、以下のように **language** 要素を使用して、XPath 式を定義できます。

```
<language language="xpath">/order/address/countryCode = 'us'</language>
```

アノテーション

アノテーションは、Bean インテグレーションのコンテキストで使用されます。アノテーションは、メッセージまたはヘッダーから情報を抽出して、抽出したデータを Bean のメソッドパラメーターに注入するという便利な方法でもあります。

たとえば、**filter()** EIP の述語として呼び出される Bean **myBeanProc** について考えてみます。Bean の **checkCredentials** メソッドが **true** を返す場合、メッセージの処理が続行されますが、メソッドが **false** を返す場合は、メッセージがフィルターによってブロックされます。この Filter パターンは以下のように実装されます。

```
// Java
MyBeanProcessor myBeanProc = new MyBeanProcessor();

from("SourceURL")
  .filter().method(myBeanProc, "checkCredentials")
  .to("TargetURL");
```

MyBeanProcessor クラスの実装は **@XPath** アノテーションを利用して、以下のように、元の XML メッセージから **username** と **password** を抽出します。

```
// 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 アノテーションは、注入先となるパラメーターの直前に置かれます。パスに **/text()** を追加し、終了タグではなく要素の内容のみが選択されるようにすることで、XPath 式がテキストノードを **明示的に** 選択することに注意してください。

Camel エンドポイント URI

Camel Language コンポーネントを使用すると、エンドポイント URI でサポートされる言語を呼び出すことができます。使用可能な構文は 2 つあります。

ファイル (または **Scheme** で定義される他のリソースタイプ) に保存されている言語スクリプトを呼び出すには、以下の URI 構文を使用します。

```
language://LanguageName:resource:Scheme:Location[?Options]
```

スキームは、**file:**、**classpath:**、または **http:** にすることができます。

たとえば、以下のルートでは、クラスパスから **mysimplescript.txt** を実行します。

```
from("direct:start")
  .to("language:simple:classpath:org/apache/camel/component/language/mysimplescript.txt")
  .to("mock:result");
```

2. 埋め込み言語スクリプトを実行するには、以下の URI 構文を使用します。

```
language://LanguageName[:Script][?Options]
```

たとえば、**script** 文字列に格納されている Simple 言語のスクリプトを実行するには、次のようにします。

```
String script = URLEncoder.encode("Hello ${body}", "UTF-8");
from("direct:start")
  .to("language:simple:" + script)
  .to("mock:result");
```

Language コンポーネントの詳細は、**Apache Camel Component Reference Guide** の Language を参照してください。

第14章 定数

概要

定数言語は、プレーンテキストの文字列を指定するために使用される簡単な組み込み言語です。これにより、式タイプが想定されるコンテキストでプレーンテキストの文字列を指定できるようになります。

XML の例

XML では、以下のように **username** ヘッダーの値を **Jane Doe** に設定できます。

```
<camelContext>
  <route>
    <from uri="SourceURL"/>
    <setHeader headerName="username">
      <constant>Jane Doe</constant>
    </setHeader>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

JAVA の例

Java では、以下のように **username** ヘッダーの値を **Jane Doe** に設定できます。

```
from("SourceURL")
  .setHeader("username", constant("Jane Doe"))
  .to("TargetURL");
```

第15章 EL

概要

Unified Expression Language (EL) は当初 JSP 2.1 標準 (JSR-245) の一部として策定されました。現在は、スタンドアロン言語としても利用できるようになりました。Apache Camel は EL 言語のオープンソース実装である JUEL (http://juel.sourceforge.net/) と統合しています。

JUEL パッケージの追加

ルートで EL を使用するには、例15.1「camel-juel 依存関係の追加」 で示すように、**camel-juel** の依存関係をプロジェクトに追加する必要があります。

例15.1 camel-juel 依存関係の追加

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-790054-redhat-00001</camel-version>
  ...
</properties>

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

静的インポート

アプリケーションコードで **el()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.language.juel.JuelExpression.el;
```

変数

表15.1「EL 変数」 は、EL を使用する際にアクセス可能な変数の一覧を示しています。

表15.1 EL 変数

変数	型	値
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ
in	**org.apache.camel.Message**	IN メッセージ

変数	型	値
out	**org.apache.camel.Message**	OUT メッセージ

例

例15.2「EL を使用したルート」 は、EL を使用する 2 つのルートを示しています。

例15.2 EL を使用したルート

```
<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 FILE 言語

概要

File 言語 は Simple 言語 の拡張であり、File 言語自体は独立した言語ではありません。しかし、File 言語は、File または FTP エンドポイントと組み合わせた場合のみ使用できます。

16.1. FILE 言語を使用する場合

概要

File 言語は、Simple 言語の拡張であり、どこでも利用できるではありません。File 言語は以下の状況で使用できます。

- File または FTP コンシューマーエンドポイント 。
- File または FTP コンシューマーによって作成されたエクスチェンジ 。

注記

エスケープ文字 \ は、File 言語では使用できません。

File または FTP コンシューマーエンドポイントでの使用

File または FTP コンシューマーエンドポイントで設定できる URI オプションがいくつかあり、それらに File 言語の式を適用できます。たとえば、File コンシューマーエンドポイント URI で、File 式を使用して **fileName**、**move**、**preMove**、**moveFailed**、および **sortBy** オプションを設定することができます。

File コンシューマーエンドポイントでは、**fileName** オプションはフィルターとして機能し、開始ディレクトリーから実際に読み取られるファイルを決定します。プレーンテキスト文字列が指定されている場合 (例: **fileName=report.txt**)、File コンシューマーは更新されるたびに同じファイルを読み込みます。しかし、File 言語の式を指定することで、このオプションをより動的なものにすることができます。たとえば、以下のように、File コンシューマーがディレクトリーをポーリングするたびに、Counter Bean を使用して異なるファイルを読み込めます。

```
file://target/filelanguage/bean/?fileName=${bean:counter.next}.txt&delete=true
```

${bean:counter.next} 式が、ID **counter** で登録された Bean 上で **next()** のメソッドを呼び出します。

move オプションは、File コンシューマーエンドポイントによってファイルが読み込まれた後、ファイルをバックアップ場所に移動するために使用されます。たとえば、以下のエンドポイント URI は、ファイルが処理された後にファイルをバックアップディレクトリーに移動します。

```
file://target/filelanguage/?
move=backup/${date:now:yyyyMMdd}/${file:name.noext}.bak&recursive=false
```

${file:name.noext}.bak 式は、ファイルの拡張子を **.bak** に置き換えて、元のファイル名を変更します。

sortBy オプションを使用すると、ファイルを処理する順序を指定できます。たとえば、ファイル名のアルファベット順でファイルを処理するには、次のように File コンシューマーエンドポイントを指定します。

```
file://target/filelanguage/?sortBy=file:name
```

たとえば、ファイルの最終更新日時の順でファイルを処理するには、以下のように File コンシューマーエンドポイントを指定します。

```
file://target/filelanguage/?sortBy=file:modified
```

また、以下のように **reverse:** 接頭辞を追加すると順序を逆にすることもできます。

```
file://target/filelanguage/?sortBy=reverse:file:modified
```

File または FTP コンシューマーによって作成されたエクスチェンジ

エクスチェンジ が File または FTP コンシューマーエンドポイントによって作成されている場合、ルート全体を通して、File 言語をエクスチェンジに適用することができます (元のメッセージヘッダーが残っている場合のみ)。たとえば、以下のように、ファイルの拡張子に応じてメッセージをルーティングするコンテンツベースルーターを定義することができます。

```
<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. FILE 変数

概要

File 変数は、ルートが File または FTP のコンシューマーエンドポイントで始まる場合は常に使用できます。これは、基になるメッセージのボディーが **java.io.File** 型であることを前提にしています。File 変数を使用すると、**java.io.File** クラスのメソッドを呼び出している場合と同様に、ファイルパス名の様々な部分にアクセスすることができます (実際は、File 言語は File や FTP エンドポイントで設定されたメッセージヘッダーから必要な情報を抽出します)。

起動ディレクトリー

起動ディレクトリー は、File または FTP エンドポイントに指定されるディレクトリーで、一部の File 変数は、このディレクトリーを起点とした相対パスを返します。たとえば、以下の File コンシューマーエンドポイントは、開始ディレクトリー **./filetransfer** (相対パス) が指定されています。

```
file:filetransfer
```

以下の FTP コンシューマーエンドポイントには、開始ディレクトリー **./ftptransfer** (相対パス) が指定されています。

```
ftp://myhost:2100/ftptransfer
```

File 変数の命名規則

通常、File 変数は **java.io.File** クラスの対応するメソッドの後に名前が付けられます。たとえば、**file:absolute** 変数は、**java.io.File.getAbsolute()** メソッドによって返される値を提供します。

注記

ただし、この命名規則は厳密ではありません。たとえば、**java.io.File.getSize()** というメソッドは **ありません**。

変数の一覧表

表16.1「File 言語の変数」 は、File 言語でサポートされるすべての変数の一覧表になります。

表16.1 File 言語の変数

変数	型	説明
file:name	**String**	起動ディレクトリーを起点とした相対パス名。
file:name.ext	**String**	ファイルの拡張子 (パス名の最後の **.** に続く文字)。複数ドットのあるファイル拡張子をサポートします (例: .tar.gz)。
file:name.ext.single	**String**	ファイルの拡張子 (パス名の最後の **.** に続く文字)。ファイル拡張子に複数のドットがある場合、この変数は最後の部分のみを返します。
file:name.noext	**String**	開始ディレクトリーを起点とし、ファイル拡張子を除外した相対パス名。
file:name.noext.single	**String**	開始ディレクトリーを起点とし、ファイル拡張子を除外した相対パス名。ファイル拡張子に複数のドットがある場合、この変数は最後の部分のみを取り除き、他の部分を返します。
file:onlyname	**String**	パス名の最後のセグメント。つまり、親ディレクトリーのパスがないファイル名です。

変数	型	説明
file:onlyname.noext	**String**	パス名の最後のセグメントで、ファイルの拡張子を省略します。つまり、親ディレクトリーのパスがなく、拡張子もないファイル名です。
file:onlyname.noext.single	**String**	パス名の最後のセグメントで、ファイルの拡張子を省略します。つまり、親ディレクトリーのパスがなく、拡張子もないファイル名です。ファイル拡張子に複数のドットがある場合、この変数は最後の部分のみを取り除き、他の部分を返します。
file:ext	**String**	ファイル拡張子 (**file:name.ext** と同じ)。
file:parent	**String**	開始ディレクトリーを含む、親ディレクトリーのパス名。
file:path	**String**	開始ディレクトリーを含む、ファイルのパス名。
file:absolute	**ブール値**	開始ディレクトリーが絶対パスとして指定された場合は **true**、そうでない場合は **false**。
file:absolute.path	**String**	ファイルの絶対パス名。
file:length	**Long**	ファイルのサイズ。
file:size	**Long**	**file:length と同じです。**
file:modified	**java.util.Date**	最終更新日。

16.3. 例

相対パス名

File コンシューマーエンドポイントに、開始ディレクトリーが**相対パス名**で指定されているシナリオを考えてみます。たとえば、以下の File エンドポイントの開始ディレクトリーは **./filelanguage** であるとします。

```
file://filelanguage
```

そして、**filelanguage** ディレクトリーをスキャンしている間、エンドポイントが以下のファイルを読み込んだとします。

```
./filelanguage/test/hello.txt
```

そして最後に、**filelanguage** ディレクトリー自体が以下のような絶対位置にあるとします。

```
/workspace/camel/camel-core/target/filelanguage
```

前述のシナリオで、File 言語の File 変数を、現在のエクスチェンジに適用した場合、以下の値が返されます。

表現	結果
file:name	**test/hello.txt**
file:name.ext	**txt**
file:name.noext	**test/hello**
file:onlyname	**hello.txt**
file:onlyname.noext	**hello**
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**

絶対パス名

File コンシューマーエンドポイントに、開始ディレクトリーが **絶対パス名** で指定されているシナリオを考えてみます。たとえば、以下の File エンドポイントには、開始ディレクトリー **/workspace/camel/camel-core/target/filelanguage** があります。

```
file:///workspace/camel/camel-core/target/filelanguage
```

そして、**filelanguage** ディレクトリーをスキャンしている間、エンドポイントが以下のファイルを読み込んだとします。

```
./filelanguage/test/hello.txt
```

前述のシナリオで、File 言語の File 変数を、現在のエクスチェンジに適用した場合、以下の値が返されます。

表現	結果
file:name	**test/hello.txt**
file:name.ext	**txt**
file:name.noext	**test/hello**
file:onlyname	**hello.txt**
file:onlyname.noext	**hello**
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

概要

Groovy は、オブジェクトを素早く解析できる Java ベースのスクリプト言語です。Groovy のサポートは **camel-groovy** モジュールに含まれます。

スクリプトモジュールの追加

ルートで Groovy を使用するには、例17.1「camel-groovy 依存関係の追加」 で示したたように、**camel-groovy** の依存関係をプロジェクトに追加する必要があります。

例17.1 camel-groovy 依存関係の追加

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-790054-redhat-00001</camel-version>
  ...
</properties>

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

静的インポート

アプリケーションコードで **groovy()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

組み込み属性

表17.1「Groovy 属性」 に、Groovy を使用する際にアクセス可能な組み込み属性の一覧を示します。

表17.1 Groovy 属性

属性	型	値
context	**org.apache.camel.CamelContext**	Camel コンテキスト
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ

属性	型	値
request	**org.apache.camel.Message**	IN メッセージ
response	**org.apache.camel.Message**	OUT メッセージ
properties	**org.apache.camel.builder.script.PropertiesFunction**	スクリプト内でプロパティーコンポーネントを簡単に使用できるようにする **resolve** メソッドを使用した関数。

属性はすべて **ENGINE_SCOPE** に設定されます。

例

例17.2「Groovy を使用したルート」 は、Groovy スクリプトを使用する 2 つのルートを示しています。

例17.2 Groovy を使用したルート

```
<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>
```

プロパティーコンポーネントの使用

プロパティーコンポーネントからプロパティー値にアクセスするには、以下のように組み込み **properties** 属性で **resolve** メソッドを呼び出します。

```
.setHeader("myHeader").groovy("properties.resolve(PropKey)")
```

PropKey は、解決するプロパティーのキーで、キーの値は **String** タイプになります。

プロパティーコンポーネントの詳細は、**Apache Camel Component Reference Guide** の Properties を参照してください。

GROOVY SHELL のカスタマイズ

カスタムの **GroovyShell** インスタンスを Groovy 式で使用する必要がある場合があります。カスタム **GroovyShell** を指定するには、**org.apache.camel.language.Groovy.GroovyShellFactory** SPI インターフェイスの実装を Camel レジストリーに追加します。

たとえば、以下の Bean を Spring コンテキストに追加することで、Apache Camel はデフォルトの GroovyShell インスタンスではなく、カスタムの静的インポートを含む **GroovyShell** インスタンスを使用します。

```
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 ヘッダー

概要

Header 言語は、メッセージのヘッダー値にアクセスするための便利な方法を提供します。Header 言語は指定したヘッダー名の大文字と小文字を区別せずに検索を行い、対象のヘッダー値を返します。

Header 言語は **camel-core** の一部です。

XML の例

たとえば、**SequenceNumber** ヘッダー値 (シーケンス番号は正の整数でなければなりません) に応じて受信エクスチェンジを再配列するには、以下のようにルートを定義します。

```
<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 の例

以下のように、Java DSL で同じルートを定義できます。

```
from("SourceURL")
  .resequence(header("SequenceNumber"))
  .to("TargetURL");
```

第19章 JAVASCRIPT

概要

ECMAScript としても知られる JavaScript は、オブジェクトを素早く解析できる Java ベースのスクリプト言語です。JavaScript サポートは **camel-script** モジュールに含まれます。

スクリプトモジュールの追加

ルートで JavaScript を使用するには、例19.1「camel-script 依存関係の追加」 で示したたように、**camel-script** の依存関係をプロジェクトに追加する必要があります。

例19.1 camel-script 依存関係の追加

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-790054-redhat-00001</camel-version>
  ...
</properties>

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

静的インポート

アプリケーションコードで **javaScript()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

組み込み属性

表19.1「JavaScript 属性」 に、JavaScript を使用する際にアクセス可能な組み込み属性の一覧を示します。

表19.1 JavaScript 属性

属性	型	値
context	**org.apache.camel.CamelContext**	Camel コンテキスト

属性	型	値
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ
request	**org.apache.camel.Message**	IN メッセージ
response	**org.apache.camel.Message**	OUT メッセージ
properties	**org.apache.camel.builder.script.PropertiesFunction**	スクリプト内でプロパティーコンポーネントを簡単に使用できるようにする **resolve** メソッドを使用した関数。

属性はすべて **ENGINE_SCOPE** に設定されます。

例

例19.2「JavaScript を使用したルート」 は、JavaScript を使用したルートを示しています。

例19.2 JavaScript を使用したルート

```
<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>
```

プロパティーコンポーネントの使用

プロパティーコンポーネントからプロパティー値にアクセスするには、以下のように組み込み **properties** 属性で **resolve** メソッドを呼び出します。

```
.setHeader("myHeader").javaScript("properties.resolve(PropKey)")
```

PropKey は、解決するプロパティーのキーで、キーの値は **String** タイプになります。

プロパティーコンポーネントの詳細は、**Apache Camel Component Reference Guide** の Properties を参照してください。

第20章 JOSQL

概要

JoSQL (SQL for Java オブジェクト) 言語を使用すると、Apache Camel で判定式および表現式を評価できます。JoSQL は SQL のようなクエリー構文を使用して、インメモリーの Java オブジェクトに対してデータ抽出やソートを実行できます。しかし、JoSQL はデータベースでは **ありません**。JoSQL 構文では、各 Java オブジェクトのインスタンスをテーブルの行とみなし、各オブジェクトのメソッドを列とみなして操作します。この構文を使用すると、Java オブジェクトのコレクションからデータを抽出および再設定する強力なステートメントを作成できます。詳細は http://josql.sourceforge.net/ を参照してください。

JOSQL モジュールの追加

ルートで JoSQL を使用するには、例20.1「camel-josql 依存関係の追加」 で示したように、**camel-josql** の依存関係をプロジェクトに追加する必要があります。

例20.1 camel-josql 依存関係の追加

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-josql</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静的インポート

アプリケーションコードで **sql()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.builder.sql.SqlBuilder.sql;
```

変数

表20.1「JoSQL 変数」 に、JoSQL を使用する際にアクセス可能な組み込み変数の一覧を示します。

表20.1 JoSQL 変数

名前	型	説明
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ
in	**org.apache.camel.Message**	IN メッセージ

名前	型	説明
out	**org.apache.camel.Message**	OUT メッセージ
プロパティー	**オブジェクト**	キーが **property**であるエクスチェンジプロパティー
header	**オブジェクト**	キーが **header** である IN メッセージヘッダー
variable	**オブジェクト**	キーが **variable** である変数

例

例20.2「JoSQL を使用したルート」 は、JoSQL を使用したルートを示しています。

例20.2 JoSQL を使用したルート

```
<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

概要

JsonPath は、JSON メッセージの一部を抽出する際に便利な構文を提供します。JsonPath の構文は XPath に似ていますが、XML ドキュメントではなく、JSON メッセージから JSON オブジェクトを抽出するために使用されます。**jsonpath** は式または述語 (空の結果がブール値として解釈される **false**) として使用できます。

JSONPATH パッケージの追加

Camel ルートで JsonPath を使用するには、以下のように **camel-jsonpath** の依存関係をプロジェクトに追加する必要があります。

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

JAVA の例

以下の Java の例は、**jsonpath()** DSL コマンドを使用して、特定の価格範囲内の商品を選択する方法を示しています。

```
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")
```

JsonPath のクエリー結果が空のセットの場合、結果は **false** と解釈されます。このため、JsonPath のクエリーを述語として使用することができます。

XML の例

以下の XML の例は、**jsonpath** DSL の要素を使用してルート内で述語を定義する方法を示しています。

```
<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>
```

簡略化構文

jsonpath の構文を使って基本的な述語を定義する場合、構文を覚えるのが少し難しいかもしれません。たとえば、安価な本 (20 未満) をすべて検索する場合は、以下のような構文を書く必要があります。

```
$.store.book[?(@.price < 20)]
```

しかし、以下のように簡略化した記述もできます。

```
store.book.price < 20
```

さらに、価格キーを持つノードを確認する場合は、パスを省略することもできます。

```
price < 20
```

この構文をサポートするために、**EasyPredicateParser** を使用して基本的な記法で述語を定義します。つまり基本的な記法とは、述語を **$** 記号で始めてはならず、演算子が 1 つだけ含まれるようにすることです。簡略化構文は以下になります。

```
left OP right
```

下記の例のように、right 部分には、Camel Simple 言語を使用することもできます。

```
store.book.price < ${header.limit}
```

サポートされるメッセージボディーのタイプ

Camel JSonPath は、以下のタイプのメッセージボディーをサポートしています。

型	説明
File	ファイルからの読み込み
String	プレーンテキスト
マップ	メッセージボディーは **java.util.Map** 型
リスト	メッセージボディーは java.util.List 型

型	説明
POJO	任意。Jackson がクラスパス上にある場合、**camel-jsonpath** は Jackson を使用してメッセージボディーを **POJO** から **java.util.Map** (JSonPath でサポートされる) に変換の上、JSonPath で処理することができます。たとえば、依存関係として **camel-jackson** を追加して、Jackson を含めることができます。
InputStream	上記のタイプがどれも一致しない場合、Camel はメッセージボディーを **java.io.InputStream** 型で読み込みます。

メッセージボディーがサポートされないタイプの場合は、デフォルトでは例外が出力されますが、JSonPath の設定で例外を抑止できます。

例外の抑制

jsonpath 式によって設定されたパスが見つからない場合、JSONPath は例外を出力します。**SuppressExceptions** オプションを true に設定すると、例外を無視できます。たとえば、以下のコードで、**jsonpath** パラメーターの一部として true オプションを追加します。

```
from("direct:start")
    .choice()
        // use true to suppress exceptions
        .when().jsonpath("person.middlename", true)
            .to("mock:middle")
        .otherwise()
            .to("mock:other");
```

XML DSL では、以下の構文を使用します。

```
<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 の注入

Bean インテグレーションを使用して Bean メソッドを呼び出す場合、JsonPath を使用してメッセージから値を抽出し、それをメソッドパラメーターにバインドできます。以下に例を示します。

```
// 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
    }
}
```

インライン SIMPLE 式

Camel 2.18 の新機能。

Camel は **JsonPath** 式でインライン **Simple** 式をサポートします。**Simple** 言語の挿入は、以下のように **Simple** 構文で記述する必要があります。

```
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");
```

Simple 式のサポートを無効にするには、以下のようにオプション **allowSimple=false** を設定します。

Java の場合

```
// Java DSL
.when().jsonpath("$.store.book[?(@.price < 10)]", false, false)
```

XML DSL の場合

```
// XML DSL
<jsonpath allowSimple="false">$.store.book[?(@.price &lt; 10)]</jsonpath>
```

リファレンス

JsonPath の詳細については、JSonPath プロジェクト のページを参照してください。

第22章 JXPATH

概要

JXPath を使用すると、Apache Commons JXPath で Java Bean のメソッドを呼び出すことができます。JXPath は XPath に似た構文です。対照的に、XML ドキュメントの要素や属性のノードにアクセスのではなく、Java Bean のオブジェクトのメソッドや属性にアクセスします。さらに、ある Bean の属性が XML ドキュメント (DOM/JDOM インスタンス) となっている場合、パスの残り部分は XPath 式として解釈され、XML ドキュメントから XML ノードの抽出に使用されます。言い換えれば、JXPath 言語はオブジェクトグラフのナビゲーションと XML ノード選択のハイブリッドを提供します。

JXPATH パッケージの追加

ルートで JXPath を使用するには、例22.1「camel-jxpath 依存関係の追加」 で示したように、**camel-jxpath** の依存関係をプロジェクトに追加する必要があります。

例22.1 camel-jxpath 依存関係の追加

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-790054-redhat-00001</camel-version>
  ...
</properties>

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

変数

表22.1「JXPath 変数」 に、JXPath を使用する際にアクセス可能な組み込み変数の一覧を示します。

表22.1 JXPath 変数

変数	型	値
this	**org.apache.camel.Exchange**	現在のエクスチェンジ
in	**org.apache.camel.Message**	IN メッセージ
out	**org.apache.camel.Message**	OUT メッセージ

オプション

表22.2「JXPath オプション」 では、JXPath のオプションを説明します。

表22.2 JXPath オプション

オプション	型	説明
lenient	**boolean**	Camel 2.11/2.10.5: JXPathContext で lenient を有効化します。このオプションを有効すると、JXPath 式は、無効なデータや欠落したデータである可能性のあるメッセージボディや式に対して評価することができます。詳細は JXPath ドキュメント を参照してください。このオプションはデフォルトで false です。

例

以下は JXPath を使用したルールを示しています。

```
<camelContext>
  <route>
    <from uri="activemq:MyQueue"/>
    <filter>
      <jxpath>in/body/name = 'James'</xpath>
      <to uri="mqseries:SomeOtherQueue"/>
    </filter>
  </route>
</camelContext>
```

以下の例では、メッセージフィルターの判定式として JXPath 式を使用しています。

```
from("direct:start").
    filter().jxpath("in/body/name='James'").
    to("mock:result");
```

JXPATH の注入

Bean インテグレーションを使用して Bean のメソッドを呼び出す場合、JXPath (他の 言語も使用可) を使用してメッセージから値を抽出し、メソッドパラメーターにバインドすることができます。

以下に例を示します。

```
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 }
  }
```

外部リソースからの読み込み

Camel 2.11 から利用可能

スクリプトを外部化して、**"classpath:"**、**"file:"**、または **"http:"** などのリソースから Camel に読み込むことができます。以下の構文を使用してください。

```
"resource:scheme:location"
```

たとえば、クラスパスのファイルを読み込むには、次のように指定します。

```
.setHeader("myHeader").jxpath("resource:classpath:myjxpath.txt")
```

第23章 MVEL

概要

MVEL は Java ベースの動的な言語で、OGNL に似ていますが、OGNL より高速だと言われています。MVEL のサポートは **camel-mvel** モジュールにあります。

構文

MVEL のドット構文を使用して Java メソッドを呼び出すことができます。以下に例を示します。

```
getRequest().getBody().getFamilyName()
```

MVEL は動的に型付けされているため、**getFamilyName()** メソッドを呼び出す前にメッセージボディーのインスタンス (**Object** 型) をキャストする必要はありません。Bean の属性を取得する場合は、以下のような簡単な構文を使用することもできます。

```
request.body.familyName
```

MVEL モジュールの追加

ルートで MVEL を使用するには、例23.1「camel-mvel 依存関係の追加」 で示すように、**camel-mvel** の依存関係をプロジェクトに追加する必要があります。

例23.1 camel-mvel 依存関係の追加

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-790054-redhat-00001</camel-version>
  ...
</properties>

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

組み込み変数

表23.1「MVEL 変数」 に、MVEL を使用する際にアクセス可能な組み込み変数の一覧を示します。

表23.1 MVEL 変数

名前	型	説明
this	**org.apache.camel.Exchange**	現在のエクスチェンジ
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ
exception	**Throwable**	エクスチェンジの例外 (ある場合)
exchangeID	**String**	エクスチェンジの ID
fault	**org.apache.camel.Message**	Fault メッセージ (ある場合)
request	**org.apache.camel.Message**	IN メッセージ
response	**org.apache.camel.Message**	OUT メッセージ
properties	**マップ**	エクスチェンジプロパティー
property(name)	**オブジェクト**	指定されたエクスチェンジプロパティーの値
property(name, type)	**Type**	指定されたエクスチェンジプロパティーのタイプの値。

例

例23.2「MVEL を使用したルート」 は、MVEL を使用するルートを示しています。

例23.2 MVEL を使用したルート

```
<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 NAVIGATION LANGUAGE (OGNL)

概要

OGNL は、Java オブジェクトのプロパティーを取得および設定するための式言語です。プロパティーの値の取得および設定に同じ式を使用します。OGNL サポートは **camel-ognl** モジュールに含まれます。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。

OGNL モジュールの追加

ルートで OGNL を使用するには、例24.1「camel-ognl 依存関係の追加」 に示したように、**camel-ognl** への依存関係をプロジェクトに追加する必要があります。

例24.1 camel-ognl 依存関係の追加

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ognl</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静的インポート

アプリケーションコードで **ognl()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.language.ognl.OgnlExpression.ognl;
```

組み込み変数

表24.1「OGNL 変数」 に、OGNL を使用する際にアクセス可能な組み込み変数の一覧を示します。

表24.1 OGNL 変数

名前	型	説明
this	**org.apache.camel.Exchange**	現在のエクスチェンジ

名前	型	説明
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ
exception	**Throwable**	エクスチェンジの例外 (ある場合)
exchangeID	**String**	エクスチェンジの ID
fault	**org.apache.camel.Message**	Fault メッセージ (ある場合)
request	**org.apache.camel.Message**	IN メッセージ
response	**org.apache.camel.Message**	OUT メッセージ
properties	**マップ**	エクスチェンジプロパティー
property(name)	**オブジェクト**	指定されたエクスチェンジプロパティーの値
property(name, type)	**Type**	指定されたエクスチェンジプロパティーのタイプの値。

例

例24.2「OGNL を使用したルート」 は、OGNL を使用するルートを示しています。

例24.2 OGNL を使用したルート

```
<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 (非推奨)

概要

PHP は、広く使用されている汎用スクリプト言語で、特に Web 開発に適しています。PHP サポートは **camel-script** モジュールに含まれます。

重要

Apache Camel の PHP は非推奨となり、今後のリリースで削除される予定です。

スクリプトモジュールの追加

ルートで PHP を使用するには、例25.1「camel-script 依存関係の追加」 で示したたように、**camel-script** の依存関係をプロジェクトに追加する必要があります。

例25.1 camel-script 依存関係の追加

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静的インポート

アプリケーションコードで **php()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

組み込み属性

表25.1「PHP 属性」 に、PHP を使用する際にアクセス可能な組み込み属性の一覧を示します。

表25.1 PHP 属性

属性	型	値
context	**org.apache.camel.CamelContext**	Camel コンテキスト
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ

属性	型	値
request	**org.apache.camel.Message**	IN メッセージ
response	**org.apache.camel.Message**	OUT メッセージ
properties	**org.apache.camel.builder.script.PropertiesFunction**	スクリプト内でプロパティーコンポーネントを簡単に使用できるようにする **resolve** メソッドを使用した関数。

属性はすべて **ENGINE_SCOPE** に設定されます。

例

例25.2「PHP を使用したルート」 は、PHP を使用するルートを示しています。

例25.2 PHP を使用したルート

```
<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>
```

プロパティーコンポーネントの使用

プロパティーコンポーネントからプロパティー値にアクセスするには、以下のように組み込み **properties** 属性で **resolve** メソッドを呼び出します。

```
.setHeader("myHeader").php("properties.resolve(PropKey)")
```

PropKey は、解決するプロパティーのキーで、キーの値は **String** タイプになります。

プロパティーコンポーネントの詳細は、**Apache Camel Component Reference Guide** の Properties を参照してください。

第26章 エクスチェンジプロパティー

概要

Exchange Property 言語は、**エクスチェンジプロパティー**にアクセスする便利な方法を提供します。エクスチェンジプロパティー名の 1 つに一致するキーを指定すると、Exchange Property 言語は対応する値を返します。

Exchange Property 言語は **camel-core** の一部です。

XML の例

たとえば、**listOfEndpoints** エクスチェンジプロパティーに Recipient List が含まれる場合に Recipient List パターンを実装するには、以下のようにルートを定義できます。

```
<camelContext>
  <route>
    <from uri="direct:a"/>
    <recipientList>
      <exchangeProperty>listOfEndpoints</exchangeProperty>
    </recipientList>
  </route>
</camelContext>
```

JAVA の例

以下のように、同じ Recipient List を Java で実装できます。

```
from("direct:a").recipientList(exchangeProperty("listOfEndpoints"));
```

第27章 PYTHON (非推奨)

概要

Python は、さまざまなアプリケーション領域で使用される非常に強力な動的プログラミング言語です。Python は Tcl、Perl、Ruby、Scheme、または Java と比較されることがよくあります。Python サポートは **camel-script** モジュールに含まれます。

重要

Apache Camel の Python は非推奨となり、今後のリリースで削除される予定です。

スクリプトモジュールの追加

ルートで Python を使用するには、例27.1「camel-script 依存関係の追加」 で示したたように、**camel-script** の依存関係をプロジェクトに追加する必要があります。

例27.1 camel-script 依存関係の追加

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静的インポート

アプリケーションコードで **python()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

組み込み属性

表27.1「Python 属性」 に、Python を使用する際にアクセス可能な組み込み属性の一覧を示します。

表27.1 Python 属性

属性	型	値
context	**org.apache.camel.CamelContext**	Camel コンテキスト

属性	型	値
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ
request	**org.apache.camel.Message**	IN メッセージ
response	**org.apache.camel.Message**	OUT メッセージ
properties	**org.apache.camel.builder.script.PropertiesFunction**	スクリプト内でプロパティーコンポーネントを簡単に使用できるようにする **resolve** メソッドを使用した関数。

属性はすべて **ENGINE_SCOPE** に設定されます。

例

例27.2「Python を使用したルート」 は、Python を使用するルートを示しています。

例27.2 Python を使用したルート

```
<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>
```

プロパティーコンポーネントの使用

プロパティーコンポーネントからプロパティー値にアクセスするには、以下のように組み込み **properties** 属性で **resolve** メソッドを呼び出します。

```
.setHeader("myHeader").python("properties.resolve(PropKey)")
```

PropKey は、解決するプロパティーのキーで、キーの値は **String** タイプになります。

プロパティーコンポーネントの詳細は、**Apache Camel Component Reference Guide** の Properties を参照してください。

第28章 REF

概要

Ref 式言語は、レジストリー からカスタム 式 を検索する方法です。これは、XML DSL で使用すると特に便利です。

Ref 言語は **camel-core** の一部です。

静的インポート

Java のアプリケーションコードで Ref 言語を使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.language.ref.RefLanguage.ref;
```

XML の例

たとえば、Splitter パターンは、以下のように Ref 言語を使用してカスタム式を参照できます。

```
<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 DSL の例

前述のルートは、以下のように Java DSL でも実装できます。

```
from("seda:a")
  .split().ref("myExpression")
  .to("seda:b");
```

第29章 RUBY (非推奨)

概要

Ruby は動的なオープンソースプログラミング言語で、シンプルさと生産性に重点を置いています。自然な読みやすさと書きやすさを兼ね備えたエレガントな構文を持ち合わせます。Ruby サポートは **camel-script** モジュールの一部です。

重要

Apache Camel の Ruby は非推奨となり、今後のリリースで削除される予定です。

スクリプトモジュールの追加

ルートで Ruby を使用するには、例29.1「camel-script 依存関係の追加」 で示したたように、**camel-script** の依存関係をプロジェクトに追加する必要があります。

例29.1 camel-script 依存関係の追加

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

静的インポート

アプリケーションコードで **ruby()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

組み込み属性

表29.1「Ruby 属性」 に、Ruby を使用する際にアクセス可能な組み込み属性の一覧を示します。

表29.1 Ruby 属性

属性	型	値
context	**org.apache.camel.CamelContext**	Camel コンテキスト

属性	型	値
exchange	**org.apache.camel.Exchange**	現在のエクスチェンジ
request	**org.apache.camel.Message**	IN メッセージ
response	**org.apache.camel.Message**	OUT メッセージ
properties	**org.apache.camel.builder.script.PropertiesFunction**	スクリプト内でプロパティーコンポーネントを簡単に使用できるようにする **resolve** メソッドを使用した関数。

属性はすべて **ENGINE_SCOPE** に設定されます。

例

例29.2「Ruby を使用したルート」 は、Ruby を使用するルートを示しています。

例29.2 Ruby を使用したルート

```
<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>
```

プロパティーコンポーネントの使用

プロパティーコンポーネントからプロパティー値にアクセスするには、以下のように組み込み **properties** 属性で **resolve** メソッドを呼び出します。

```
.setHeader("myHeader").ruby("properties.resolve(PropKey)")
```

PropKey は、解決するプロパティーのキーで、キーの値は **String** タイプになります。

プロパティーコンポーネントの詳細は、**Apache Camel Component Reference Guide** の Properties を参照してください。

第30章 SIMPLE 言語

概要

Simple 言語は、特にエクルチェンジオブジェクトのさまざまな部分にアクセスして操作することを目的とし、Apache Camel で開発された言語です。この言語は、初期に開発された時ほどシンプルではなく、現在では論理演算子と接続詞の包括的なセットが備えられています。

30.1. JAVA DSL

Java DSL での Simple 式

Java DSL には、ルートで **simple()** コマンドを使用するために、2 つのスタイルがあります。以下のように、**simple()** コマンドを引数としてプロセッサーに渡すこともできます。

```
from("seda:order")
  .filter(simple("${in.header.foo}"))
  .to("mock:fooOrders");
```

または、以下のように、プロセッサーのサブ句として **simple()** コマンドを実行することもできます。

```
from("seda:order")
  .filter()
  .simple("${in.header.foo}")
  .to("mock:fooOrders");
```

文字列への埋め込み

プレーンテキスト文字列内に Simple 式を埋め込む場合は、プレースホルダー構文 **${Expression}** を使用する必要があります。たとえば、以下のように式 **in.header.name** を文字列に埋め込みます。

```
simple("Hello ${in.header.name}, how are you?")
```

開始トークンおよび終了トークンのカスタマイズ

Java から、**SimpleLanguage** オブジェクトの **changeFunctionStartToken** static メソッドと **changeFunctionEndToken** static メソッドを呼び出すことで、開始トークンおよび終了トークン (デフォルトでは、**{** と **}**) をカスタマイズできます。

たとえば、Java では以下のように、開始トークンおよび終了トークンを **[** および **]** に変更することができます。

```
// Java
import org.apache.camel.language.simple.SimpleLanguage;
...
SimpleLanguage.changeFunctionStartToken("[");
SimpleLanguage.changeFunctionEndToken("]");
```

注記

開始トークンおよび終了トークンをカスタマイズすると、クラスパス上で同じ **camel-core** ライブラリーを共有するすべての Apache Camel アプリケーションに影響します。たとえば、OSGi サーバーでは多くのアプリケーションに影響する可能性がありますが、Web アプリケーション (WAR ファイル) では、Web アプリケーション自体にしか影響を与えません。

30.2. XML DSL

XML DSL での Simple 式

XML DSL では、**simple** 要素内に式を配置することで、Simple 式を使用できます。たとえば、ヘッダー **foo** の内容に基づいてフィルターリングを実行するルートを定義します。

```
<route id="simpleExample">
  <from uri="seda:orders"/>
  <filter>
    <simple>${in.header.foo}</simple>
    <to uri="mock:fooOrders"/>
  </filter>
</route>
```

代替プレースホルダー構文

たとえば、Spring プロパティープレースホルダーや OSGi Blueprint プロパティープレースホルダーを有効にしている場合、**${Expression}** の構文が別のプロパティープレースホルダーの構文と競合することがあります。この場合、Simple 式に使用する代替構文 **$simple{Expression}** を使用して、プレースホルダーの曖昧さを解決できます。以下に例を示します。

```
<simple>Hello $simple{in.header.name}, how are you?</simple>
```

開始トークンおよび終了トークンのカスタマイズ

XML 設定から、**SimpleLanguage** インスタンスをオーバーライドすることで、開始トークンおよび終了トークン (デフォルトでは **{** および **}**) をカスタマイズできます。たとえば、開始トークンおよび終了トークンを **[** および **]** に変更するには、以下のように XML 設定ファイルで新しい **SimpleLanguage** Bean を定義します。

```
<bean id="simple" class="org.apache.camel.language.simple.SimpleLanguage">
  <constructor-arg name="functionStartToken" value="["/>
  <constructor-arg name="functionEndToken" value="]"/>
</bean>
```

注記

開始トークンおよび終了トークンをカスタマイズすると、クラスパス上で同じ **camel-core** ライブラリーを共有するすべての Apache Camel アプリケーションに影響します。たとえば、OSGi サーバーでは多くのアプリケーションに影響する可能性がありますが、Web アプリケーション (WAR ファイル) では、Web アプリケーション自体にしか影響を与えません。

XML DSL の空白と自動トリミング

デフォルトでは、XML DSL における Simple 式の前後にある空白文字は、自動的にトリミングされます。以下の式は空白で囲まれています。

```
<transform>
  <simple>
    data=${body}
  </simple>
</transform>
```

よって、自動でトリミングされ、空白のない以下の式と同等になります。

```
<transform>
  <simple>data=${body}</simple>
</transform>
```

式の前後に改行を入れたい場合は、以下のように改行文字を明示的に追加できます。

```
<transform>
  <simple>data=${body}\n</simple>
</transform>
```

また、以下のように **trim** 属性に **false** を設定すると、自動トリミング機能をオフにすることができます。

```
<transform trim="false">
  <simple>data=${body}
</simple>
</transform>
```

30.3. 外部スクリプトの呼び出し

概要

以下で説明されているように、外部リソースに保存されている Simple スクリプトを実行できます。

スクリプトリソースの構文

以下の構文を使用して、外部リソースとして保存されている Simple スクリプトにアクセスします。

```
resource:Scheme:Location
```

Scheme: は、**classpath:**、**file:**、または **http:** のいずれかにできます。

たとえば、以下はクラスパスから **mysimple.txt** スクリプトを読み取る式になります。

```
simple("resource:classpath:mysimple.txt")
```

30.4. 式

概要

Simple 言語は、メッセージエクスチェンジの各種パーツを返すさまざまな式を提供します。たとえば、式 **simple("${header.timeOfDay}")** は、受信メッセージからの **timeOfDay** というヘッダーの内容を返します。

注記

Apache Camel 2.9 以降、変数の値を返すには、**常に** プレースホルダー構文の **${Expression}** を使用する必要があります。エンクロージングトークン **${** および **}**) を省略することは許容されません。

単一変数の内容

用意された変数に基づいて、Simple 言語を使い、文字列式を定義できます。たとえば、**in.header.HeaderName** という形式の変数を使用して、以下のように **HeaderName** ヘッダーの値を取得できます。

```
simple("${in.header.foo}")
```

文字列に組み込まれた変数

simple 変数を文字列式に埋め込むことができます。以下に例を示します。

```
simple("Received a message from ${in.header.user} on ${date:in.header.date:yyyyMMdd}.")
```

date および bean 変数

Simple 言語は、エクスチェンジのさまざまな部分にアクセスする変数を提供するだけでなく (表30.1「Simple 言語の変数」 を参照)、日付をフォーマットするための特別な変数である **date:command:pattern** を提供し、Bean メソッドを呼び出すための特別な変数である **bean:beanRef** も提供します。たとえば、以下のように date および bean 変数を使用できます。

```
simple("Todays date is ${date:now:yyyyMMdd}")
simple("The order type is ${bean:orderService?method=getOrderType}")
```

結果の型指定

式の結果の型を明示的に指定できます。これは主に、結果の型をブール値や数値型に変換するのに便利です。

Java DSL では、**simple()** への追加引数として結果の型を指定します。たとえば、整数の結果を返すには、以下のように Simple 式を評価できます。

```
...
.setHeader("five", simple("5", Integer.class))
```

XML DSL では、**resultType** 属性を使用して結果の型を指定します。以下に例を示します。

```
<setHeader headerName="five">
  <!-- use resultType to indicate that the type should be a java.lang.Integer -->
  <simple resultType="java.lang.Integer">5</simple>
```

```
</setHeader>
```

動的ヘッダーキー

Camel 2.17 から、**setHeader** および **setExchange** プロパティーで、キーの名前に Simple 言語式を使用している場合に Simple 言語による動的ヘッダーキーを使用できるようになりました。

```
<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>
```

ネストされた式

Simple 式は入れ子にすることができます。以下に例を示します。

```
simple("${header.${bean:headerChooser?method=whichHeader}}")
```

定数または列挙型へのアクセス

以下の構文を使用し、Bean の定数または列挙フィールドにアクセスすることができます。

```
type:ClassName.Field
```

たとえば、以下の Java **enum** 型を見てみましょう。

```
package org.apache.camel.processor;
...
public enum Customer {
    GOLD, SILVER, BRONZE
}
```

以下のように **Customer** enum フィールドにアクセスできます。

```
 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 式

Object Graph Navigation Language (OGNL) は、チェーン状に Bean メソッドを呼び出す表記法です。メッセージボディーに Java Bean が含まれる場合、OGNL 表記を使用して Bean プロパティーに簡単にアクセスすることができます。たとえば、メッセージボディーが **getAddress()** アクセサーを持つ Java オブジェクトである場合、以下のように **Address** オブジェクトと **Address** オブジェクトのプロパティーにアクセスできます。

```
simple("${body.address}")
simple("${body.address.street}")
simple("${body.address.zip}")
simple("${body.address.city}")
```

ここで、表記 **${body.address.street}** は、**${body.getAddress.getStreet}** の省略形になります。

OGNL null-safe 演算子

null-safe 演算子 **?.** を使用して、ボディーに address が **ない** 場合に null-pointer 例外が発生しないようにすることができます。以下に例を示します。

```
simple("${body?.address?.street}")
```

ボディーが **java.util.Map** 型である場合、以下の表記法を使用して、 **foo** キーでマップの値を検索することができます。

```
simple("${body[foo]?.name}")
```

OGNL リスト要素へのアクセス

リスト要素にアクセスするには、角括弧 **[k]** を使用することもできます。以下に例を示します。

```
simple("${body.address.lines[0]}")
simple("${body.address.lines[1]}")
simple("${body.address.lines[2]}")
```

last キーワードは、リストの最後の要素のインデックスを返します。たとえば、以下のようにリストの**最後から 2 番目** の要素にアクセスできます。

```
simple("${body.address.lines[last-1]}")
```

size メソッドを使用して、以下のようにリストのサイズを問い合わせることができます。

```
simple("${body.address.lines.size}")
```

OGNL 配列の長さへのアクセス

以下のように、**length** メソッドを使用して Java 配列の長さにアクセスできます。

```
String[] lines = new String[]{"foo", "bar", "cat"};
exchange.getIn().setBody(lines);

simple("There are ${body.length} lines")
```

30.5. 述語

概要

等号の式をテストする述語を作成できます。たとえば、述語 **simple("${header.timeOfDay} == '14:30'")** は、受信メッセージの **timeOfDay** ヘッダーが **14:30** と等しいかどうかをテストします。

また、**resultType** がブール値として指定される場合は常に、式は式ではなく述語として評価されます。これにより、これらの式に述語構文を使用することができます。

構文

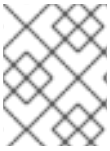
また、Simple 述語を使用して、エクスチェンジのさまざまなパーツ (ヘッダー、メッセージボディーなど) をテストすることもできます。Simple 述語には、以下の一般的な構文があります。

```
${LHSVariable} Op RHSValue
```

ここで、左側の変数 **LHSVariable** は、表30.1「Simple 言語の変数」 で示される変数のいずれかであり、右側の値 **RHSValue** は、以下のいずれかになります。

- 別の変数 **${RHSVariable}**。
- 単一引用符 **' '** で囲まれた文字列リテラル。
- 単一引用符 **' '** で囲まれた数値定数。
- Null オブジェクト **null**。

Simple 言語は、常に RHS 値を LHS 値の型に変換しようとします。



注記

Simple 言語は、RHS の変換を試みますが、演算子によっては、比較を行う前に LHS を適切な型にキャストする必要があることがあります。

例

たとえば、以下のように simple 文字列比較と数値比較を実行できます。

```
simple("${in.header.user} == 'john'")

simple("${in.header.number} > '100'")  // String literal can be converted to integer
```

以下のように、左辺 (Left Hand Side) がコンマ区切りのリストメンバーであるかどうかをテストします。

```
simple("${in.header.type} in 'gold,silver'")
```

以下のように、左辺 (Left Hand Side) が正規表現に一致するかどうかをテストします。

```
simple("${in.header.number} regex '\d{4}'")
```

以下のように、**is** 演算子を使用して、左辺 (Left Hand Side) の型をテストすることができます。

```
simple("${in.header.type} is 'java.lang.String'")
simple("${in.header.type} is 'String'") // You can abbreviate java.lang. types
```

以下のように、指定した数値の範囲 (範囲が範囲に含まれる) に左辺 (Left Hand Side) のものがあるかどうかをテストすることができます。

```
simple("${in.header.number} range '100..199'")
```

接続詞

また、論理接続 **&&** および || を使用して述語を組み合わせることもできます。

たとえば、以下は、**&&** 接続詞 (論理積) を使用する式になります。

```
simple("${in.header.title} contains 'Camel' && ${in.header.type} == 'gold'")
```

そして、|| 接続詞 (論理和) を使用した式がこちらです。

```
simple("${in.header.title} contains 'Camel' || ${in.header.type} == 'gold'")
```

30.6. 変数の参照

変数の一覧表

表30.1「Simple 言語の変数」 は、Simple 言語でサポートされているすべての変数を表示します。

表30.1 Simple 言語の変数

変数	型	説明
camelContext	**オブジェクト**	Camel コンテキスト**OGNL 式をサポートします。**
camelId	**String**	Camel コンテキストの ID 値。
exchangeId	**String**	エクスチェンジの ID 値。
id	**String**	In メッセージ ID の値。
body	**オブジェクト**	In メッセージ ボディー**OGNL 式をサポートします。**
in.body	**オブジェクト**	In メッセージ ボディー**OGNL 式をサポートします。**
out.body	**オブジェクト**	Out メッセージ ボディー

変数	型	説明
bodyAs(Type)	型	In メッセージボディーを指定された型に変換します。すべての型 (Type) は、**byte[]**、**String**、**Integer**、および **Long** を除き、完全修飾 Java 名を使用して指定する必要があります 。変換された本文は null にすることができます。
mandatoryBodyAs(Type)	型	In メッセージボディーを指定された型に変換します。すべての型 (Type) は、**byte[]**、**String**、**Integer**、および **Long** を除き、完全修飾 Java 名を使用して指定する必要があります 。変換されたボディーは null 以外であることが想定されています。
header. HeaderName	オブジェクト	In メッセージの HeaderName ヘッダー。**OGNL 式をサポートします。**
header[HeaderName**]**	オブジェクト	In メッセージの HeaderName ヘッダー (代替構文)。
headers. HeaderName	オブジェクト	In メッセージの HeaderName ヘッダー。
headers[HeaderName**]**	オブジェクト	In メッセージの HeaderName ヘッダー (代替構文)。
in.header. HeaderName	オブジェクト	In メッセージの HeaderName ヘッダー。**OGNL 式をサポートします。**
in.header[HeaderName**]**	オブジェクト	In メッセージの HeaderName ヘッダー (代替構文)。
in.headers. HeaderName	オブジェクト	In メッセージの HeaderName ヘッダー。**OGNL 式をサポートします。**
in.headers[HeaderName**]**	オブジェクト	In メッセージの HeaderName ヘッダー (代替構文)。
out.header. HeaderName	オブジェクト	Out メッセージの HeaderName ヘッダー。

変数	型	説明
out.header[HeaderName**]**	**オブジェクト**	Out メッセージの HeaderName ヘッダー (代替構文)。
out.headers. HeaderName	**オブジェクト**	Out メッセージの HeaderName ヘッダー。
out.headers[HeaderName**]**	**オブジェクト**	Out メッセージの HeaderName ヘッダー (代替構文)。
headerAs(Key,Type)	**型**	指定された型に変換された Key ヘッダー。すべての型 (Type) は、**byte[]**、**String**、**Integer**、および **Long** を除き、完全修飾 Java 名を使用して指定する必要があります 。変換された値は null にすることができます。
ヘッダー	**マップ**	In ヘッダーのすべて (**java.util.Map** 型)。
in.headers	**マップ**	In ヘッダーのすべて (**java.util.Map** 型)。
exchangeProperty. PropertyName	**オブジェクト**	エクスチェンジの PropertyName プロパティー。
exchangeProperty[PropertyName**]**	**オブジェクト**	エクスチェンジの PropertyName プロパティー (代替構文)。
exchangeProperty. PropertyName.OGNL	**オブジェクト**	エクスチェンジの PropertyName プロパティーを指定し、Camel OGNL 式を使用してその値を呼び出します。
sys. SysPropertyName	**String**	SysPropertyName Java システムプロパティー。
sysenv. SysEnvVar	**String**	SysEnvVar システム環境変数。
exception	**String**	**Exchange.getException()** からの例外オブジェクトか、この値が null の場合は、**Exchange.EXCEPTION_CAUGHT** プロパティーでキャッチされた例外。それ以外の場合は null になります。**OGNL 式をサポートします。**

変数	型	説明
exception.message	**String**	例外がエクスチェンジに設定されている場合は、**Exception.getMessage()** の値を返します。そうでない場合は **null** を返します。
exception.stacktrace	**String**	例外がエクスチェンジに設定されている場合は、**Exception.getStackTrace()** の値を返します。そうでない場合は **null** を返します。**注記:** Simple 言語は、まず **Exchange.getException()** から例外を取得しようとします。このプロパティーが設定されていない場合、**Exchange.getProperty(Exchange.CAUGHT_EXCEPTION)** の呼び出しによって、キャッチされた例外をチェックします。
date:command**:**pattern	**String**	java.text.SimpleDateFormat パターンを使用してフォーマットされた日付。以下のコマンドがサポートされます: 現在の日時の場合は **now**。**header.**HeaderName または **in.header.**HeaderName を使用し、In メッセージの HeaderName ヘッダーで、java.util.Date オブジェクトを使います。Out メッセージの HeaderName ヘッダーで、java.util.Date オブジェクトを使うための **out.header.**HeaderName。
bean:beanID.Method	**オブジェクト**	参照される Bean のメソッドを呼び出し、**メソッド呼び出しの結果を返します。**メソッド名を指定するには、**beanID.Method** 構文を使用するか、**beanID?method=**methodName 構文を使用できます。

変数	型	説明
ref:beanID	**オブジェクト**	レジストリーで ID (**beanID**) を使い Bean を検索し、**bean 自体への参照を返します。**たとえば Splitter EIP を使用している場合は、この変数を使用して、分割アルゴリズムを実装する Bean を参照できます。
properties:Key	**String**	Key プロパティープレースホルダーの値。
properties:Location:Key	**String**	プロパティーファイルの場所が **Location** によって提供される **Key** プロパティープレースホルダーの値。
threadName	**String**	現在のスレッドの名前。
routeId	**String**	**Exchange** がルーティングされている、現在のルートの ID を返します。
type:Name[.Field]	**オブジェクト**	型またはフィールドを完全修飾名 (FQN) で参照します。フィールドを参照するには、**.Field** を追加します。たとえば、**type:org.apache.camel.Exchange.FILE_NAME** として、**Exchange** クラスの **FILE_NAME** 定数フィールドを参照できます。
collate(group)	**リスト**	Camel 2.17 から、collate 関数はメッセージボディーをイテレートし、データを特定のサイズのサブリストにグループ化するようになりました。Splitter EIP を使用して、メッセージボディーを分割してグループ化したり、サブメッセージを N 個のサブリストのグループにまとめたりすることができます。
skip(number)	**Iterator**	skip 関数はメッセージのボディーをイテレートし、最初の項目数をスキップします。Splitter EIP と併用することで、メッセージボディーを分割し、最初の N 項目数をスキップすることができます。

30.7. 演算子リファレンス

バイナリー演算子

Simple 言語用バイナリー演算子は、表30.2「Simple 言語用バイナリー演算子」 に表示されています。

表30.2 Simple 言語用バイナリー演算子

演算子	説明
==	等しい
=~	等しい場合に無視されます。文字列値を比較する際には、大文字と小文字の違いを区別しません。
>	より大きい
>=	より大きいか等しいか。
<	より小さい
⇐	より小さいか等しいか。
!=	等しくない。
contains	LHS 文字列に RHS 文字列が含まれるかどうかをテストします。
not contains	LHS 文字列に RHS 文字列が含まれて **いない** かどうかをテストします。
regex	LHS 文字列が RHS 正規表現と一致するかどうかをテストします。
not regex	LHS 文字列が RHS 正規表現と一致 **しない** かどうかをテストします。
in	LHS 文字列が RHS のコンマ区切りリストの中にあるかどうかをテストします。
not in	LHS 文字列が RHS のコンマ区切りリストの中に **ない** かどうかをテストします。
is	LHS が RHS の Java 型のインスタンスであるかどうかをテストします (Java **instanceof** 演算子を使用)。

演算子	説明
not is	LHS が RHS の Java 型のインスタンスでは **ない** かどうかをテストします (Java **instanceof** 演算子を使用)。
range	LHS の数値が RHS の範囲内にあるかどうかをテストします (範囲の形式は、**'min...max'** です)。
not range	tyLHS の数値が RHS の範囲内に **ない**かどうかをテストします (範囲の形式は、**'min...max'** です) 。
は次の値で始まる	Camel 2.18 の新機能。LHS 文字列が RHS 文字列で始まるかどうかをテストします。
は次の値で終了する	Camel 2.18 の新機能。LHS 文字列が RHS 文字列で終わるかどうかをテストします。

単項演算子およびエスケープ文字

Simple 言語用バイナリー演算子は、表30.3「Simple 言語用単項演算子」 に表示されています。

表30.3 Simple 言語用単項演算子

演算子	説明
++	数値を 1 増加します。
--	数値を 1 減少します。
\n	改行文字です。
\r	キャリッジリターン文字です。
\t	タブ文字です。
\	**(廃止)** Camel バージョン 2.11 以降、バックスラッシュエスケープ文字はサポートされません。

述語の組み合わせ

表30.4「Simple 言語の述語用接続詞」 に記載されている接続詞を使用して、2 つ以上の Simple 言語の述語を組み合わせることができます。

表30.4 Simple 言語の述語用接続詞

演算子	説明
&&	2 つの述語を論理 **積** で組み合わせます。
||	2 つの述語を論理 **和** で組み合わせます。
および	**非推奨**代わりに **&&** を使用します。
または	**非推奨**代わりに || を使用します。

第31章 SPEL

概要

Spring Expression Language (SpEL) は、Spring 3 で提供される Object Graph Navigation 言語で、ルートの述語および式を作成するために使用できます。SpEL の注目すべき機能は、レジストリーから Bean に簡単にアクセスできる機能です。

構文

SpEL 式はプレースホルダー構文 **#{SpelExpression}** を使用する必要があります。これにより、プレーンテキストの文字列に組み込むことができます (つまり、SpEL では Expression templating が有効になっています)。

SpEL は **@BeanID** 構文を使用してレジストリー (通常は Spring レジストリー) で Bean を検索することもできます。たとえば、ID の Bean **headerUtils** およびメソッド **count()** (現在のメッセージのヘッダーの数をカウント) の場合、以下のように SpEL 述語で **headerUtils** Bean を使用できます。

```
#{@headerUtils.count > 4}
```

SPEL パッケージの追加

ルートで SpEL を使用するには、例31.1「camel-spring 依存関係の追加」 に示したように、**camel-spring** への依存関係をプロジェクトに追加する必要があります。

例31.1 camel-spring 依存関係の追加

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.23.2.fuse-790054-redhat-00001</camel-version>
  ...
</properties>

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

変数

表31.1「SpEL 変数」 に、SpEL を使用する際にアクセス可能な組み込み変数の一覧を示します。

表31.1 SpEL 変数

変数	型	説明
this	**エクスチェンジ**	現在のエクスチェンジはルートオブジェクトです。
exchange	**エクスチェンジ**	現在のエクスチェンジ
exchangeId	**String**	現在のエクスチェンジ ID
exception	**Throwable**	エクスチェンジの例外 (ある場合)
fault	**メッセージ**	Fault メッセージ (ある場合)
request	**メッセージ**	エクスチェンジの In メッセージ。
response	**メッセージ**	エクスチェンジの Out メッセージ (ある場合)。
properties	**マップ**	エクスチェンジプロパティー
property(Name)	**オブジェクト**	Name で指定されたエクスチェンジプロパティー。
property(Name, Type)	**型**	Type に変換された Name で指定されたエクスチェンジプロパティー。

XML の例

たとえば、**Country** ヘッダーに **USA** があるメッセージのみを選択するには、以下の SpEL 式を使用できます。

```
<route>
  <from uri="SourceURL"/>
  <filter>
    <spel>#{request.headers['Country'] == 'USA'}}</spel>
    <to uri="TargetURL"/>
  </filter>
</route>
```

JAVA の例

以下のように、Java DSL で同じルートを定義できます。

```
from("SourceURL")
  .filter().spel("#{request.headers['Country'] == 'USA'}")
  .to("TargetURL");
```

以下の例は、プレーンテキストの文字列内に SpEL 式を埋め込む方法を示しています。

```
from("SourceURL")
  .setBody(spel("Hello #{request.body}! What a beautiful #{request.headers['dayOrNight']}"))
  .to("TargetURL");
```

第32章 XPATH 言語

概要

XML メッセージを処理する際に、XPath 言語を使用すると、メッセージの Document Object Model (DOM) に作用する XPath 式を指定して、メッセージの一部を選択することができます。また、要素や属性の内容をテストするための XPath 述語を定義することもできます。

32.1. JAVA DSL

基本的な式

xpath("Expression") を使用して、現在のエクスチェンジで XPath 式を評価することができます (XPath 式は現在の **In** メッセージのボディーに適用されます)。**xpath()** 式の結果は、XML ノード (または複数のノードが一致する場合はノードセット) になります。

たとえば、現在の **In** メッセージボディーから **/person/name** 要素の内容を抽出し、それを使用して **user** という名前のヘッダーを設定するには、以下のようなルートを定義することができます。

```
from("queue:foo")
    .setHeader("user", xpath("/person/name/text()"))
    .to("direct:tie");
```

setHeader() の引数に **xpath()** を指定する代わりに、Fluent Builder **xpath()** コマンドを使用することができます。たとえば、

```
from("queue:foo")
    .setHeader("user").xpath("/person/name/text()")
    .to("direct:tie");
```

結果を特定の型に変換する場合は、**xpath()** の第 2 引数に結果の型を指定します。たとえば、結果の型が **String** であることを明示的に指定するには、次のようにします。

```
xpath("/person/name/text()", String.class)
```

Namespaces

通常、XML 要素はスキーマに属し、名前空間 URI によって識別されます。このようなドキュメントを処理する際には、名前空間 URI と接頭辞を関連付けておく必要があり、XPath 式の中で要素名を明確に識別できるようにします。Apache Camel はヘルパークラス **org.apache.camel.builder.xml.Namespaces** を提供しています。これにより、名前空間と接頭辞の関連付けを定義することができます。

たとえば、接頭辞 **cust** を名前空間 **http://acme.com/customer/record** と関連付けて、要素 **/cust:person/cust:name** コンテンツを抽出するには、以下のようなルートを定義します。

```
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");
```

Namespaces オブジェクト **ns** を追加の引数として渡すことで、**xpath()** 式ビルダーで名前空間の定義を利用できるようにします。複数の名前空間を定義する必要がある場合は、以下のように **Namespace.add()** メソッドを使用します。

```
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");
```

結果の型を指定し、**さらに**名前空間も定義する必要がある場合は、以下のように **xpath()** の 3 つの引数形式を使用できます。

```
xpath("/person/name/text()", String.class, ns)
```

名前空間の監査

XPath 式を使用する際に最も頻繁に発生する問題の 1 つは、受信メッセージに表示される名前空間と XPath 式で使用される名前空間の間に不一致があることです。この種の問題のトラブルシューティングを支援するために、XPath 言語では、すべての受信メッセージからすべての名前空間をシステムログにダンプするオプションがサポートされています。

INFO ログレベルでネームスペースロギングを有効にするには、以下のように Java DSL で **logNamespaces** オプションを有効にします。

```
xpath("/foo:person/@id", String.class).logNamespaces()
```

あるいは、**org.apache.camel.builder.xml.XPathBuilder** ロガーで **TRACE** レベルのロギングを有効にするようにロギングシステムを設定することもできます。

名前空間のロギングを有効にすると、処理されたメッセージごとに以下のようなログメッセージが表示されます。

```
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

基本的な式

XML DSL で XPath 式を評価するには、**xpath** 要素の中に XPath 式を入れます。XPath 式は、現在の **In** メッセージのボディーに適用され、XML ノード (またはノードセット) を返します。通常、返された XML ノードは自動的に文字列に変換されます。

たとえば、現在の **In** メッセージボディーから **/person/name** 要素の内容を抽出し、それを使用して **user** という名前のヘッダーを設定するには、以下のようなルートを定義することができます。

```
<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>
```

結果を特定の型に変換する場合は、**resultType** 属性を Java 型名 (ここでは完全修飾型名を指定する必要があります) に設定して、結果の型を指定します。たとえば、結果の型が **java.lang.String** であることを明示的に指定するには、以下を実行します (ここでは **java.lang.** 接頭辞を省略できます)。

```
<xpath resultType="String">/person/name/text()</xpath>
```

Namespaces

要素が 1 つ以上の XML スキーマに属する文書を処理する場合、通常、名前空間 URI を接頭辞で関連付ける必要があります。こうすることで XPath 式の中で要素名を明確に識別できるようにします。接頭辞を名前空間 URI に関連付けるための標準的な XML メカニズムを使用することができます。つまり、**xmlns:Prefix="NamespaceURI"** のような属性を設定することができます。

たとえば、接頭辞 **cust** を名前空間 **http://acme.com/customer/record** と関連付けて、要素 **/cust:person/cust:name** コンテンツを抽出するには、以下のようなルートを定義します。

```
<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>
```

名前空間の監査

XPath 式を使用する際に最も頻繁に発生する問題の 1 つは、受信メッセージに表示される名前空間と XPath 式で使用される名前空間の間に不一致があることです。この種の問題のトラブルシューティングを支援するために、XPath 言語では、すべての受信メッセージからすべての名前空間をシステムログにダンプするオプションがサポートされています。

INFO ログレベルでネームスペースロギングを有効にするには、以下のように XML DSL で **logNamespaces** オプションを有効にします。

```
<xpath logNamespaces="true" resultType="String">/foo:person/@id</xpath>
```

あるいは、**org.apache.camel.builder.xml.XPathBuilder** ロガーで **TRACE** レベルのロギングを有効にするようにロギングシステムを設定することもできます。

名前空間のロギングを有効にすると、処理されたメッセージごとに以下のようなログメッセージが表示されます。

```
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 の注入

パラメーターバインディングアノテーション

Apache Camel Bean インテグレーションを使用して Java Bean 上でメソッドを呼び出す場合、**@XPath** アノテーションを使用して、エクスチェンジから値を抽出してメソッドパラメーターにバインドすることができます。

たとえば、**AccountService** オブジェクトで **credit** メソッドを呼び出す以下のルートのフラグメントについて考えてみましょう。

```
from("queue:payments")
    .beanRef("accountService","credit")
    ...
```

credit メソッドは、パラメーターバインディングアノテーションを使用してメッセージボディーから関連データを抽出し、以下のようにパラメーターに挿入します。

```
public class AccountService {
    ...
    public void credit(
            @XPath("/transaction/transfer/receiver/text()") String name,
            @XPath("/transaction/transfer/amount/text()") String amount
            )
    {
        ...
    }
    ...
}
```

詳細については、カスタマポータルの**Apache Camel 開発者ガイド**の**Bean インテグレーション**を参照してください。

Namespaces

表32.1「@XPath の定義済み名前空間」 は、XPath に対して定義済みの名前空間を示します。これらの名前空間接頭辞は、**@XPath** アノテーションに表示される **XPath** 式で使用できます。

表32.1 @XPath の定義済み名前空間

名前空間 URI	接頭辞
http://www.w3.org/2001/XMLSchema	**xsd**
http://www.w3.org/2003/05/soap-envelope	**soap**

カスタム名前空間

@NamespacePrefix アノテーションを使用して、カスタム XML 名前空間を定義することができます。**@NamespacePrefix** アノテーションを呼び出して、**@XPath** アノテーションの **namespaces** 引数を初期化します。その後、**@NamespacePrefix** で定義された名前空間を **@XPath** アノテーションの式の値で使用することができます。

たとえば、接頭辞 **ex** をカスタム名前空間 **http://fusesource.com/examples** と関連付けるには、以下のように **@XPath** アノテーションを呼び出します。

```
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 ビルダー

概要

org.apache.camel.builder.xml.XPathBuilder クラスを使用すると、エクスチェンジとは独立して XPath 式を評価することができます。つまり、任意のソースからの XML フラグメントがあれば、**XPathBuilder** を使用して XML フラグメントの XPath 式を評価することができます。

式のマッチング

matches() メソッドを使用して、指定された XPath 式に一致する XML ノードが 1 つ以上見つかるかどうかをチェックします。**XPathBuilder** を使用した XPath 式のマッチングの基本的な構文は以下のとおりです。

```
boolean matches = XPathBuilder
                .xpath("Expression")
                .matches(CamelContext, "XMLString");
```

ここで、指定された式である **Expression** が XML フラグメントである **XMLString** に対して評価され、式に一致するノードが少なくとも 1 つ見つかった場合に結果は true になります。たとえば、以下の例では XPath 式が **xyz** 属性に一致するものを見つけたため、**true** が返されます。

```
boolean matches = XPathBuilder
                .xpath("/foo/bar/@xyz")
                .matches(getContext(), "<foo><bar xyz='cheese'/></foo>"));
```

式の評価

evaluate() メソッドを使用して、指定された XPath 式と一致する最初のノードの内容を返します。**XPathBuilder** を使用して XPath 式を評価するための基本的な構文は以下のとおりです。

```
String nodeValue = XPathBuilder
                .xpath("Expression")
                .evaluate(CamelContext, "XMLString");
```

また、**evaluate()** に第 2 引数として必要な型を渡すことで、結果の型を指定することもできます。

```
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. SAXON の有効化

前提条件

Saxon パーサーを使用するための前提条件は、**camel-saxon** アーティファクトに依存関係を追加することです (Maven を使用している場合は、この依存関係を Maven POM に追加するか、**camel-saxon-2.23.2.fuse-790054-redhat-00001.jar** ファイルをクラスパスに追加します)。

Java DSL での Saxon パーサーの使用

Java DSL では、Saxon パーサーを有効にする最も簡単な方法は、**saxon()** Fluent Builder メソッドを呼び出すことです。たとえば、次の例のように Saxon パーサーを呼び出すことができます。

```
// 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>");
```

XML DSL での Saxon パーサーの使用

XML DSL では、Saxon パーサーを有効にする最も簡単な方法は、**xpath** 要素で **saxon** 属性を true に設定することです。たとえば、次の例のように Saxon パーサーを呼び出すことができます。

```
<xpath saxon="true" resultType="java.lang.String">current-dateTime()</xpath>
```

Saxon を使ったプログラミング

アプリケーションコードで Saxon XML パーサーを使用する場合は、以下のコードを使用して明示的に Saxon トランスフォーマーファクトリーのインスタンスを作成することができます。

```
// Java
import javax.xml.transform.TransformerFactory;
import net.sf.saxon.TransformerFactoryImpl;
...
TransformerFactory saxonFactory = new net.sf.saxon.TransformerFactoryImpl();
```

一方、汎用の JAXP API を使用してトランスフォーマーファクトリーのインスタンスを作成する場合は、最初に **ESBInstall/etc/system.properties** ファイルの**javax.xml.transform.TransformerFactory** プロパティーを以下のように設定する**必要があります**。

```
javax.xml.transform.TransformerFactory=net.sf.saxon.TransformerFactoryImpl
```

そして、以下のように汎用の JAXP API を使用して Saxon ファクトリーをインスタンス化することができます。

```
// Java
import javax.xml.transform.TransformerFactory;
...
TransformerFactory factory = TransformerFactory.newInstance();
```

アプリケーションが Saxon を使用するサードパーティーのライブラリーに依存している場合、2 番目の汎用的なアプローチを使用する必要がある可能性があります。

注記

Saxon ライブラリーは、OSGi バンドル **net.sf.saxon/saxon9he** としてコンテナーにインストールする必要があります (通常はデフォルトでインストールされています)。バージョンが 7.1 未満の Fuse ESB では、汎用的な JAXP API を使用して Saxon をロードすることはできません。

32.6. 式

結果の型

デフォルトでは、XPath 式は **org.w3c.dom.NodeList** 型の 1 つ以上の XML ノードのリストを返します。しかし、型コンバーターのメカニズムを使用して、結果を別の型に変換することができます。Java DSL では、**xpath()** コマンドの第 2 引数に結果の型を指定することができます。たとえば、XPath 式の結果を **String** として返すには次のようにします。

```
xpath("/person/name/text()", String.class)
```

XML DSL では、以下のように **resultType** 属性で結果の型を指定することができます。

```
<xpath resultType="java.lang.String">/person/name/text()</xpath>
```

ロケーションパスにおけるパターン

XPath ロケーションパスでは、以下のパターンを使用することができます。

/people/person

基本的なロケーションパスは、特定の要素の入れ子になったロケーションを指定します。つまり、前のロケーションパスは、次の XML フラグメントの中の person 要素と一致することになります。

```
<people>
  <person>...</person>
</people>
```

この基本パターンは、**複数の**ノードにマッチする可能性があることに注意してください。たとえば、**people** 要素内に **person** 要素が 1 つ以上ある場合などがこれに該当します。

/name/text()

要素の内部の**テキスト**にアクセスするだけであれば、ロケーションパスに **/text()** を追加します。それ以外の場合は、ノードには要素の開始タグと終了タグが含まれます (ノードを文字列に変換するときにこれらのタグが含まれます)。

/person/telephone/@isDayTime

AttributeName 属性の値を選択するには、構文 **@AttributeName** を使用します。たとえば、以下の XML フラグメントに適用すると、前述のロケーションパスは **true** を返します。

```
<person>
  <telephone isDayTime="true">1234567890</telephone>
</person>
```

*****

指定したスコープ内のすべての要素に一致するワイルドカード。たとえば、**/people/person/*** は **person** のすべての子要素にマッチします。

@*

一致した要素のすべての属性にマッチするワイルドカード。たとえば、**/person/name/@*** は、すべての一致した **name** 要素のすべての属性にマッチします。

//

すべてのネストされたレベルのロケーションパスにマッチします。たとえば、**//name** パターンは、以下の XML フラグメントで強調表示されている **name** 要素にすべてマッチします。

```
<invoice>
  <person>
    <name .../>
  </person>
</invoice>
<person>
  <name .../>
</person>
<name .../>
```

..

現在のコンテキストノードの親を選択します。現在のコンテキストノードはドキュメントルートであり、親を持たないので、Apache Camel XPath 言語では通常は有用ではありません。

node()

任意の種類のノードにマッチします。

text()

テキストノードにマッチします。

comment()

コメントノードにマッチします。

processing-instruction()

処理命令ノードにマッチします。

述語フィルター

[Predicate] のように、角括弧内に述語を追加すると、ロケーションパスに一致するノードのセットをフィルターリングできます。たとえば、ロケーションパスに **[N]** を追加すると、一致するノードのリストから N^{th} 番目のノードを選択することができます。以下の式は、最初にマッチする **person** 要素を選択します。

```
/people/person[1]
```

以下の式は、最後から 2 番目の **person** 要素を選択します。

```
/people/person[last()-1]
```

特定の属性値を持つ要素を選択するために、属性の値をテストすることができます。以下の式は、**surname** 属性が Strachan または Davies のいずれかである **name** 要素を選択します。

```
/person/name[@surname="Strachan" or @surname="Davies"]
```

述語式は、接続詞 **and**、**or**、**not()** のいずれかを使用して組み合わせることができ、さらに比較式 **=**、**!=**、**>**、**>=**、**<**、⇐ を使用して式を比較することができます (実際には、小なりの記号は **<** エンティティーに置き換える必要があります)。述語フィルターで XPath 関数を使うこともできます。

軸

XML 文書の構造では、ルート要素に一連の子要素が含まれており、それらの子要素の中にはさらに子要素が含まれているものもあります。このように見てみると、**child-of** 関係によってネストされた要素間がリンクされている場合、XML 文書全体が **ツリー**構造になります。この要素ツリーの特定のノード (これを **コンテキストノード** と呼びます) を選択した場合、選択したノードに関連したツリーの異なる部分を参照することがあります。たとえば、コンテキストノードの子、コンテキストノードの親、またはコンテキストノードと同じ親を共有するすべてのノード (**兄弟ノード**) を参照する場合があります。

XPath 軸 を使用して、ノード一致のスコープを指定し、現在のコンテキストノードを起点として相対的にノードツリーの特定部分に検索を制限します。軸は、一致させたいノード名の接頭辞として、**AxisType::MatchingNode** という構文を使用して添付されます。たとえば、以下のように **child::** 軸を使用して、現在のコンテキストノードの子を検索することができます。

```
/invoice/items/child::item
```

child::item のコンテキストノードは、パス **/invoice/items** で選択される **items** 要素です。**child::** 軸は、検索対象をコンテキストノード **items** の子に制限し、**child::item** は **item** という名前の **items** の子にマッチします。**child::** 軸はデフォルトの軸であるため、先ほどの例は以下のように書くこともできます。

```
/invoice/items/item
```

しかし、他の軸も複数あり (合計 13 個)、その一部はすでに省略形で見てきました。**@** は **attribute::** の略であり、// は **descendant-or-self::** の略です。軸の一覧は以下のとおりです (詳細は下記リファレンスを参照)。

- **ancestor**
- **ancestor-or-self**
- **attribute**
- **child**
- **descendant**
- **descendant-or-self**
- **following**
- **following-sibling**
- **namespace**
- **parent**
- **preceding**
- **preceding-sibling**
- **self**

関数

XPath は、述語を評価する際に便利な標準関数の小さなセットを提供します。たとえば、ノードセットから最後にマッチするノードを選択するには、以下のようにノードセット内の最後のノードのインデックスを返す last() 関数を使用します。

```
/people/person[last()]
```

前述の例では、シーケンスの最後の **person** 要素が選択されています (文書順)。

XPath が提供するすべての関数の詳細については、以下のリファレンスを参照してください。

リファレンス

XPath 文法の詳細については、XML Path Language, Version 1.0 仕様を参照してください。

32.7. 述語

基本的な述語

Java DSL や XML DSL では、**xpath** を述語が期待されるコンテキストで使用できます。たとえば、**when()** プロセッサーや、**filter()** 句の引数として使用できます。

たとえば、以下のルートは、**/person/city** 要素に **London** という値が含まれている場合にのみ受信メッセージをフィルターリングし、メッセージの通過を許可します。

```
from("direct:tie")
    .filter().xpath("/person/city = 'London'").to("file:target/messages/uk");
```

以下のルートは、**when()** 句の XPath 述語を評価します。

```
from("direct:tie")
    .choice()
        .when(xpath("/person/city = 'London'")).to("file:target/messages/uk")
        .otherwise().to("file:target/messages/others");
```

XPath 述語演算子

XPath 言語は、表32.2「XPath 言語の演算子」 に示されているように、標準の XPath 述語演算子をサポートしています。

表32.2 XPath 言語の演算子

演算子	説明
=	等しい
!=	等しくない。
>	より大きい
>=	より大きいか等しいか。
<	より小さい
⇐	より小さいか等しいか。
および	2 つの述語を論理 **積** で組み合わせます。
または	2 つの述語を論理 **和** で組み合わせます。
not()	述語の引数を否定にします。

32.8. 変数と関数の使用

ルート内の変数の評価

ルート内で XPath 式を評価する場合、XPath 変数を使用して、現在のエクスチェンジの内容、O/S 環

境変数、および Java システムプロパティーにアクセスすることができます。XML 名前空間を介して変数にアクセスする場合、変数の値にアクセスするための構文は **$VarName** または **$Prefix:VarName** です。

たとえば、In メッセージのボディーには **$in:body**、In メッセージのヘッダー値には **$in:HeaderName** としてアクセスできます。O/S 環境変数は **$env:EnvVar** として、Java システムプロパティーは **$system:SysVar** としてアクセスできます。

以下の例では、最初のルートは **/person/city** 要素の値を抽出し、**city** ヘッダーに挿入します。2 番目のルートは、XPath 式 (**$in:city = 'London'**) を使用してエクスチェンジをフィルターリングします。**$in:city** 変数は **city** ヘッダーの値に置き換えられます。

```
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");
```

ルート内の関数の評価

標準の XPath 関数に加えて、XPath 言語では追加の関数が定義されています。これらの追加関数 (表 32.4「XPath カスタム関数」 に記載) は、基礎となるエクスチェンジへのアクセス、単純な式の評価、Apache Camel のプロパティープレースホルダコンポーネントのプロパティー検索などに使用できます。

たとえば、以下の例は、**in:header()** 関数と **in:body()** 関数を使用して、基礎となるエクスチェンジのヘッダーとボディーにアクセスします。

```
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");
```

in:HeaderName または **in:body** 変数で、これらの関数と対応する機能との類似性に注目してください。ただし、これらの関数は構文が若干異なり、**in:HeaderName** の代わりに **in:header('HeaderName')**、**in:body** の代わりに **in:body()** となっています。

XPathBuilder での変数の評価

XPathBuilder クラスを使用して評価される式で変数を使用することもできます。この場合、評価対象となる Exchange オブジェクトがないため、**$in:body** や **$in:HeaderName** などの変数を使用することはできません。しかし、**variable(Name, Value)** Fluent Builder メソッドを使ってインラインで定義された変数を使うことができます。

たとえば、以下の XPathBuilder 設定は、**$test** 変数を評価します。変数の値は、**London** となるように定義されています。

```
String var = XPathBuilder.xpath("$test")
            .variable("test", "London")
            .evaluate(getContext(), "<name>foo</name>");
```

この方法で定義された変数は、自動的にグローバル名前空間に入力されることに注意してください (たとえば、変数 **$test** は接頭辞を使用しません)。

32.9. 変数の名前空間

名前空間の表

表32.3「XPath 変数の名前空間」 は、様々な名前空間プレフィックスに関連付けられた名前空間 URI を示しています。

表32.3 XPath 変数の名前空間

名前空間 URI	接頭辞	説明
http://camel.apache.org/schema/spring	**なし**	デフォルトの名前空間 (名前空間接頭辞を持たない変数に関連付けられます)。
http://camel.apache.org/xml/in/	**in**	現在のエクスチェンジの In メッセージのヘッダーまたはボディーを参照するために使用されます。
http://camel.apache.org/xml/out/	**out**	現在のエクスチェンジの Out メッセージのヘッダーまたはボディーを参照するために使用されます。
http://camel.apache.org/xml/functions/	**関数**	一部のカスタム関数を参照するために使用されます。
http://camel.apache.org/xml/variables/environment-variables	**env**	O/S 環境変数の参照に使用します。
http://camel.apache.org/xml/variables/system-properties	**system**	Java システムのプロパティーを参照するために使用します。
http://camel.apache.org/xml/variables/exchange-property	**未定義**	エクスチェンジプロパティーの参照に使用します。この名前空間には、独自の接頭辞を定義する必要があります。

32.10. 関数の参考情報

カスタム関数の表

表32.4「XPath カスタム関数」 は、Apache Camel の XPath 式で使用できるカスタム関数を示しています。これらの関数は、標準の XPath 関数に加えて使用することができます。

表32.4 XPath カスタム関数

関数	説明
in:body()	In メッセージのボディーを返します。

関数	説明
in:header(HeaderName)	HeaderName という名前の In メッセージヘッダーを返します。
out:body()	Out メッセージのボディーを返します。
out:header(HeaderName)	HeaderNameという名前の Out メッセージヘッダーを返します。
function:properties(PropKey)	PropKey キーを持つプロパティーを検索します。
function:simple(SimpleExp)	指定された Simple 式である SimpleExp を評価します。

第33章 XQUERY

概要

XQuery は当初、データベース内の XML フォームに保存されているデータのクエリー言語として開発されました。XQuery 言語を使用すると、メッセージが XML 形式の場合に、現在のメッセージの一部を選択できます。XQuery は、XPath 言語のスーパーセットです。したがって、有効なすべての XPath 式は有効な XQuery 式でもあります。

JAVA 構文

XQuery 式を **xquery()** に渡すには、いくつかの方法があります。単純な式の場合は、XQuery の式を文字列として渡すことができます (**java.lang.String**)。XQuery 式が長い場合は、ファイルに式を保存することが推奨されます。このファイルは、オーバーロードされた **xquery()** メソッドに **java.io.File** 引数または **java.net.URL** 引数を渡すことで参照できます。XQuery 式は、メッセージの内容に対して暗黙的に作用し、結果としてノードセットを返します。コンテキストに応じて、戻り値は述語 (空のノードセットは false として解釈されます) または式として解釈されます。

SAXON モジュールの追加

XQuery をルートで使用するには、例33.1「camel-saxon 依存関係の追加」 で示されたように、**camel-saxon** の依存関係をプロジェクトに追加する必要があります。

例33.1 camel-saxon 依存関係の追加

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-saxon</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

CAMEL ON EAP デプロイメント

camel-saxon コンポーネントは、Camel on EAP (Wildfly Camel) フレームワークによってサポートされており、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上でシンプルなデプロイモデルを提供します。

静的インポート

アプリケーションコードで **xquery()** static メソッドを使用するには、以下の import ステートメントを Java ソースファイルに追加します。

```
import static org.apache.camel.component.xquery.XQueryBuilder.xquery;
```

変数

表33.1「XQuery 変数」 に、XQuery を使用する際にアクセス可能な変数の一覧を示します。

表33.1 XQuery 変数

変数	型	説明
exchange	**エクスチェンジ**	現在のエクスチェンジ
in.body	**オブジェクト**	IN メッセージのボディー
out.body	**オブジェクト**	OUT メッセージのボディー
in.headers.key	**オブジェクト**	キーが **key** である IN メッセージヘッダー
out.headers.key	**オブジェクト**	キーが **key** である OUT メッセージヘッダー
key	**オブジェクト**	キーが **key** であるエクスチェンジプロパティー

例

例33.2「XQuery を使用するルート」 は、XQuery を使用するルートを示しています。

例33.2 XQuery を使用するルート

```
<camelContext>
  <route>
    <from uri="activemq:MyQueue"/>
    <filter>
      <language langauge="xquery">/foo:person[@name='James']</language>
      <to uri="mqseries:SomeOtherQueue"/>
    </filter>
  </route>
</camelContext>
```

パート III. 高度な CAMEL プログラミング

本ガイドでは、Apache Camel API の使用方法について説明します。

第34章 メッセージ形式について

概要

Apache Camel でプログラミングを開始する前に、メッセージとメッセージ交換をモデル化する方法を明確に理解する必要があります。Apache Camel は多くのメッセージ形式を処理できるため、基本メッセージタイプは抽象形式を持つように設計されています。Apache Camel は、メッセージボディーおよびメッセージヘッダーのデータフォーマットにアクセスして変換するために必要な API を提供します。

34.1. エクスチェンジ

概要

エクスチェンジオブジェクト は、受信したメッセージをカプセル化し、関連するメタデータ (**交換プロパティー**を含む) を格納するラッパーです。さらに、現在のメッセージがプロデューサーエンドポイントにディスパッチされると、エクスチェンジは応答 (**Out** メッセージ) を保持する一時的なスロットを提供します。

Apache Camel でのエクスチェンジの重要な機能は、メッセージの Lazy Creation をサポートすることです。これにより、メッセージへの明示的なアクセスを必要としないルートの場合、大幅に最適化される可能性があります。

図34.1 ルート経由のエクスチェンジオブジェクト

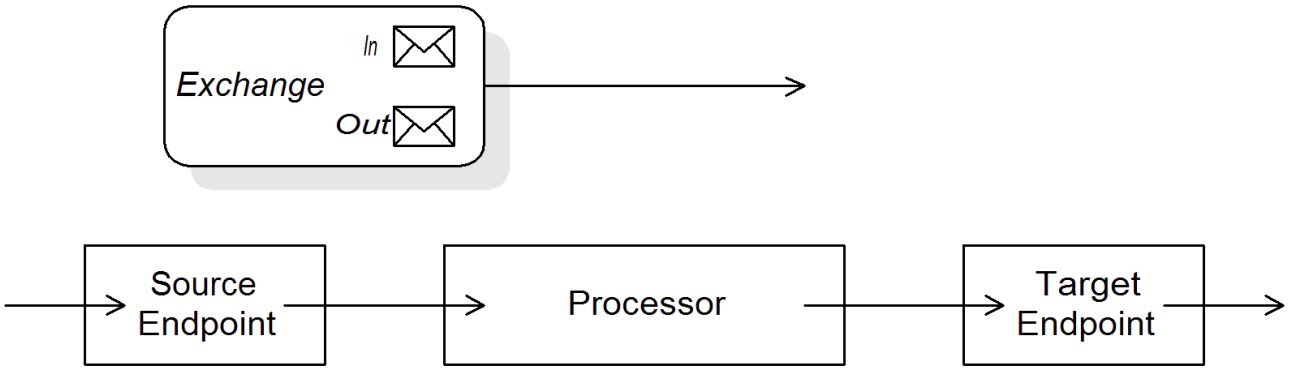


図34.1「ルート経由のエクスチェンジオブジェクト」 は、ルートを通過するエクスチェンジオブジェクトを表示します。ルートのコンテキストでは、エクスチェンジオブジェクトは **Processor.process()** メソッドの引数として渡されます。つまり、エクスチェンジオブジェクトはソースエンドポイント、ターゲットエンドポイント、および間のすべてのプロセッサーから直接アクセスすることができます。

Exchange インターフェイス

org.apache.camel.Exchange インターフェースでは、例34.1「エクスチェンジメソッド」 で示されるように、**In** および **Out** メッセージにアクセスするためのメソッドを定義します。

例34.1 エクスチェンジメソッド

```
// 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);
```

エクスチェンジインターフェースのメソッドの詳細は、「Exchange インターフェイス」 を参照してください。

メッセージの Lazy Creation

Apache Camel は、**In**、**Out**、および **Fault** のそれぞれのメッセージで Lazy Creation をサポートします。これは、メッセージインスタンスはアクセスを試みるまで作成されないことを意味します (例: **getIn()** または **getOut()**)。遅延メッセージ作成のセマンティクスは、**org.apache.camel.impl.DefaultExchange** クラスによって実装されています。

無引数のアクセッサー (**getIn()** または **getOut()**) のいずれかを呼び出す場合や、ブール値の引数 **true** を持つアクセッサー (**getIn(true)** または **getOut(true)**) を呼び出す場合は、メッセージインスタンスがまだ存在しない場合にはデフォルトのメソッド実装によって新たに作成されます。

ブール値の引数を持つアクセサーを **false** (**getIn(false)** または **getOut(false)**) と等しい場合、デフォルトのメソッド実装は現在のメッセージ値を返します。[1]

エクスチェンジ ID の Lazy Creation

Apache Camel は、エクスチェンジ ID の Lazy Creation をサポートします。エクスチェンジで **getExchangeId()** を呼び出し、そのエクスチェンジインスタンスの一意の ID を取得することができますが、この ID はメソッドを実際に呼び出す場合にのみ生成されます。このメソッドの **DefaultExchange.getExchangeId()** の実装では、ID 生成を **CamelContext** で登録された UUID ジェネレーターに委任します。

CamelContext で UUID ジェネレーターを登録する方法の詳細は、「ビルトイン UUID ジェネレーター」 を参照してください。

34.2. メッセージ

概要

メッセージオブジェクトは、以下の抽象モデルを使用してメッセージを表します。

- **メッセージボディー**
- **メッセージヘッダー**
- **メッセージの添付**

メッセージボディーとメッセージヘッダーは任意タイプ (**Object** タイプとして宣言) で宣言でき、メッセージの添付は **javax.activation.DataHandler** タイプとして宣言され、任意の MIME タイプを含めることができます。メッセージコンテンツの具体的な内容を取得する必要がある場合は、型コンバーターメカニズムを使用して、またマーシャリングおよびアンマーシャリングメカニズムを使用しても、ボディーとヘッダーを別のタイプに変換できます。

Apache Camel メッセージの重要な機能の 1 つは、メッセージ本文とヘッダーの **Lazy Creation** をサポートしていることです。場合によっては、メッセージを解析しなくてもルートを通過できることがあります。

Message インターフェイス

org.apache.camel.Message インターフェースは、例34.2「Message インターフェイス」 に示されているように、メッセージボディー、メッセージヘッダー、およびメッセージの添付にアクセスするためのメソッドを定義します。

例34.2 Message インターフェイス

```
// 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);
```

Message インターフェースのメソッドの詳細な説明は、「Message インターフェイス」 を参照してください 。

本文、ヘッダー、および添付の Lazy Creation

Apache Camel は、ボディー、ヘッダー、および添付の Lazy Creation をサポートします。つまり、メッセージボディー、メッセージヘッダー、またはメッセージ添付ファイルを表すオブジェクトは、必要になるまで作成されません。

たとえば、**In** メッセージから **foo** メッセージヘッダーにアクセスする以下のルートについて考えてみましょう。

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

このルートでは、**SourceURL** によって参照されるコンポーネントがレイジーの作成をサポートすることを前提としている場合、**In** メッセージヘッダーは **header("foo")** 呼び出しが実行されるまで実際に解析されません。この時点で、基礎となるメッセージ実装はヘッダーを解析し、ヘッダーマップを設定します。メッセージの **本文** は、ルートの最後の **to("TargetURL")** の呼び出し時に到達するまで解析されません。この時点で、ボディーはターゲットエンドポイント **TargetURL** に書き込むために必要な形式に変換されます。

本文、ヘッダー、および添付を生成する前の最後の瞬間まで待機することで、不要な型変換を回避できます。場合によっては、解析を完全に回避できます。たとえば、ルートにメッセージヘッダーへの明示的な参照がない場合、メッセージはヘッダーを解析せずにルートを通過する可能性があります。

Lazy Creation が実際に実装されるかどうかは、基礎となるコンポーネントの実装によって異なります。通常、Lazy Creation は、メッセージボディー、メッセージヘッダー、またはメッセージの添付の処理の負荷がある場合に役立ちます。Lazy Creation をサポートするメッセージタイプの実装に関する詳細は、「Message インターフェイスの実装」 を参照してください。

メッセージ ID の Lazy Creation

Apache Camel は、メッセージ ID の Lazy Creation をサポートします。つまり、メッセージ ID は、**getMessageId()** メソッドを実際に呼び出す場合にのみ生成されます。このメソッドの **DefaultExchange.getExchangeId()** の実装では、ID 生成を **CamelContext** で登録された UUID ジェネレーターに委任します。

エンドポイント実装では、エンドポイントが一意のメッセージ ID を必要とするプロトコルを実装する場合、**getMessageId()** メソッドを暗黙的に呼び出します。特に、JMS メッセージには、通常一意のメッセージ ID を含むヘッダーが含まれるため、JMS コンポーネントが自動的に **getMessageId()** を呼び出してメッセージ ID を取得します (これは JMS エンドポイントの **messageIdEnabled** オプションによって制御されます)。

CamelContext で UUID ジェネレーターを登録する方法の詳細は、「ビルトイン UUID ジェネレーター」 を参照してください。

初期メッセージの形式

In メッセージの最初のフォーマットはソースエンドポイントによって決定され、**Out** メッセージの初期フォーマットはターゲットエンドポイントによって決定されます。基礎となるコンポーネントで Lazy Creation がサポートされる場合、メッセージはアプリケーションによって明示的にアクセスされるまで解析されません。ほとんどの Apache Camel コンポーネントでは、raw 形式でメッセージボディーを作成します。たとえば、**byte[]**、**ByteBuffer**、**InputStream**、**OutputStream** のようなタイプを使用して表現します。これにより、初期メッセージの作成に必要なオーバーヘッドは最小限に抑えられます。より詳細なメッセージ形式は、通常、**型コンバーター** または **マーシャリングプロセッサー** に依存します。

型コンバーター

メッセージの最初のフォーマットは重要ではありません。組み込み型コンバーターを使用して、メッセージをある形式から別の形式に簡単に変換できるためです (「組み込み型コンバーター」 を参照)。型変換機能を公開する Apache Camel API には、さまざまな方法があります。たとえば、**convertBodyTo(Class type)** メソッドをルートに挿入して、**In** メッセージのボディーを以下のように変換できます。

```
from("SourceURL").convertBodyTo(String.class).to("TargetURL");
```

In メッセージのボディーが、**java.lang.String** に変換されます。以下の例は、**In** メッセージボディーの最後に文字列を追加する方法を示しています。

```
from("SourceURL").setBody(bodyAs(String.class).append("My Special Signature")).to("TargetURL");
```

ここでは、メッセージボディーは文字列を最後に追加する前に文字列形式に変換されます。この例では、メッセージボディーを明示的に変換する必要はありません。以下のように使用することもできます。

```
from("SourceURL").setBody(body().append("My Special Signature")).to("TargetURL");
```

ここでは、**append()** メソッドは、引数を追加する前にメッセージボディーを文字列に自動的に変換します。

メッセージの型変換メソッド

org.apache.camel.Message インターフェイスは、型変換を明示的に実行するメソッドを公開します。

- **getBody(Class<T> type)** - **T** 型としてメッセージボディーを返します。
- **getHeader(String name, Class<T> type)** - 名前付きヘッダー値を **T** 型として返します。

サポートされる変換タイプの完全なリストは、「組み込み型コンバーター」 を参照してください。

XML への変換

単純なタイプ (**byte[]**、**ByteBuffer**、**String** など) 間の変換をサポートする他に、組み込み型コンバーターは XML 形式への変換もサポートします。たとえば、メッセージのボディーを **org.w3c.dom.Document** タイプに変換できます。この変換には、メッセージ全体を解析し、XML ドキュメント構造を表すノードのツリーを作成する必要があるため、単純な変換よりも負荷が高くなります。以下の XML ドキュメントタイプに変換することができます。

- **org.w3c.dom.Document**
- **javax.xml.transform.sax.SAXSource**

XML 型変換は、単純な変換よりも厳密な適用性を持ちます。すべてのメッセージ本文が XML 構造に準拠するわけではないので、このタイプの変換が失敗する可能性があることを念頭に置いてください。一方、ルーターが XML メッセージタイプのみを扱うシナリオが多数あります。

マーシャリングとアンマーシャリング

マーシャリング では、高レベルなフォーマットを低レベルなフォーマットに変換し、**アンマーシャリング** では低レベルなフォーマットを高レベルなフォーマットに変換する必要があります。以下の 2 つのプロセッサーは、ルートでマーシャリングまたはアンマーシャリングを実行するために使用されます。

- **marshal()**
- **unmarshal()**

たとえば、シリアライズされた Java オブジェクトをファイルから読み取り、それを Java オブジェクトにアンマーシャリングするには、例34.3「Java オブジェクトのアンマーシャリング」 で示されるルート定義を使用することができます。

例34.3 Java オブジェクトのアンマーシャリング

```
from("file://tmp/appfiles/serialized")
```

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

最終的なメッセージの形式

In メッセージがルートの最後に到達すると、ターゲットエンドポイントはメッセージボディーを物理エンドポイントに書き込むことのできる形式に変換する必要があります。ソースエンドポイントに到達する **Out** メッセージにも同じルールが適用されます。この変換は通常、Apache Camel の型コンバーターを使用して暗黙的に実行されます。通常、これには、**byte[]** アレイから **InputStream** タイプへの変換など、低レベルのフォーマットから別の低レベルのフォーマットへの変換を行います。

34.3. 組み込み型コンバーター

概要

本セクションでは、マスター型コンバーターがサポートする変換を説明します。これらの変換は Apache Camel コアに組み込まれています。

通常、型コンバーターは、**Message.getBody(Class<T> type)** または **Message.getHeader(String name, Class<T> type)** などの便利な関数を介して呼び出されます。マスター型コンバーターを直接呼び出すこともできます。たとえば、エクスチェンジオブジェクト **exchange** がある場合は、例34.4「値の文字列への変換」 で示すように、指定された値を **String** に変換できます。

例34.4 値の文字列への変換

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
String str_value = tc.convertTo(String.class, value);
```

基本型コンバーター

Apache Camel は、以下の基本タイプとの変換を実行する組み込み型コンバーターを提供します。

- **java.io.File**
- **String**
- **byte[]** および **java.nio.ByteBuffer**
- **java.io.InputStream** and **java.io.OutputStream**
- **java.io.Reader** および **java.io.Writer**
- **java.io.BufferedReader** および **java.io.BufferedWriter**
- **java.io.StringReader**

ただし、これらすべてのタイプが変換可能ではありません。ビルトインコンバーターは、主に **File** および **String** タイプからの変換の提供に焦点を当てています。**File** 型は、**Reader**、**Writer**、および **StringReader**. を除く、前述の型のいずれかに変換することができます。**String** 型

は、**File**、**byte[]**、**ByteBuffer**、**InputStream**、または **StringReader** に変換できます。**String** から **File** への変換は、文字列をファイル名として解釈することで機能します。**String**、**byte[]**、および **ByteBuffer** の 3 つは完全に相互変換可能です。

注記

現在のエクスチェンジで **Exchange.CHARSET_NAME** エクスチェンジプロパティーを設定することで、**byte[]** から **String**、**String** から **byte[]** への変換に使用する文字エンコーディングを明示的に指定できます。たとえば、UTF-8 文字エンコーディングを使用して変換を実行するには、**exchange.setProperty("Exchange.CHARSET_NAME", "UTF-8")** を呼び出します。サポートされる文字セットは **java.nio.charset.Charset** クラスで説明されています。

コレクション型コンバーター

Apache Camel は、以下のコレクションタイプの変換を実行する組み込み型コンバーターを提供します。

- **Object[]**
- **java.util.Set**
- **java.util.List**

前述のコレクションタイプ間の変換の切り替えはすべてサポートされます。

マップ型コンバーター

Apache Camel は、以下のマップ型との変換を実行する組み込み型コンバーターを提供します。

- **java.util.Map**
- **java.util.HashMap**
- **java.util.Hashtable**
- **java.util.Properties**

前述のマップタイプは、set 要素が **MapEntry<K,V>** である **java.util.Set** タイプのセットに変換することもできます。

DOM 型コンバーター

以下のドキュメントオブジェクトモデル (DOM) タイプへの型変換を実行できます。

- **org.w3c.dom.Document** - **byte[]**、**String**、**java.io.File**、および **java.io.InputStream** から変換可能。
- **org.w3c.dom.Node**
- **javax.xml.transform.dom.DOMSource** - **String** から変換可能。
- **javax.xml.transform.Source** - **byte[]** と **String** から変換可能。

前述の DOM 型間の変換のすべての変更がサポートされています。

SAX 型コンバーター

SAX イベント駆動型の XML パーサーをサポートする **javax.xml.transform.sax.SAXSource** 型への変換も実行できます (詳細は SAX Web site を参照してください)。以下の型から **SAXSource** に変換します。

- **String**
- **InputStream**
- **Source**
- **StreamSource**
- **DOMSource**

enum 型コンバーター

Camel では、**String** を **enum** 型に変換する型コンバーターを提供します。ここで、文字列の値は指定のエミュレーションクラスから一致する **enum** 定数に変換されます (一致する値は **大文字と小文字を区別しません**)。この型コンバーターはメッセージ本文の変換にはほとんど必要ありませんが、Apache Camel によって特定のオプションを選択するために頻繁に使用されます。

たとえば、logging level オプションを設定する場合、以下の値 **INFO** は **enum** 定数に変換されます。

```
<to uri="log:foo?level=INFO"/>
```

enum 型コンバーターは大文字と小文字を区別しないため、以下のいずれの代替機能も利用できます。

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

カスタム型コンバーター

Apache Camel を使用すると、独自のカスタム型コンバーターを実装することもできます。カスタム型コンバーターの実装方法は、36章*型コンバーター* を参照してください。

34.4. ビルトイン UUID ジェネレーター

概要

Apache Camel では、**CamelContext** で UUID ジェネレーターを登録できます。この UUID ジェネレーターは、Apache Camel が一意の ID を生成する必要があるたびに使用されます。特に、登録済みの UUID ジェネレーターが呼び出され、**Exchange.getExchangeId()** および **Message.getMessageId()** メソッドによって返される ID を生成します。

たとえば、アプリケーションの一部が 36 文字の ID をサポートしていない場合には、デフォルトの UUID ジェネレーターを置き換えます (Websphere MQ など)。また、テスト目的でシンプルなカウンター (**SimpleUuidGenerator** を参照) を使用して ID を生成すると便利です。

提供される UUID ジェネレーター

Apache Camel を設定して、コアで提供される以下の UUID ジェネレーターのいずれかを使用できます。

- **org.apache.camel.impl.ActiveMQUuidGenerator - (Default)** は、Apache ActiveMQ で使用されるものと同じスタイルの ID を生成します。この実装は、クラウドコンピューティング (Google App Engine など) で禁止されている JDK API を使用するので、すべてのアプリケーションに適しているとは限りません。
- **org.apache.camel.impl.SimpleUuidGenerator - 1** から始まるシンプルなカウンター ID を実装します。基礎となる実装では **java.util.concurrent.atomic.AtomicLong** タイプを使用するため、スレッドセーフになります。
- **org.apache.camel.impl.JavaUuidGenerator - java.util.UUID** タイプに基づいて ID を実装します。**java.util.UUID** は同期されるため、同時システムのパフォーマンスに影響する可能性があります。

カスタム UUID ジェネレーター

カスタム UUID ジェネレーターを実装するには、例34.5「UuidGenerator インターフェイス」 に示されている **org.apache.camel.spi.UuidGenerator** インターフェイスを実装します。一意の ID 文字列を返すには、**generateUuid()** を実装する必要があります。

例34.5 UuidGenerator インターフェイス

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

/**
 * Generator to generate UUID strings.
 */
public interface UuidGenerator {
    String generateUuid();
}
```

Java を使用した UUID ジェネレーターの指定

Java を使用してデフォルトの UUID ジェネレーターを置き換えるには、現在の **CamelContext** オブジェクトの **setUuidGenerator()** メソッドを呼び出します。たとえば、以下のように **SimpleUuidGenerator** インスタンスを現在の **CamelContext** に登録できます。

```
// Java
getContext().setUuidGenerator(new org.apache.camel.impl.SimpleUuidGenerator());
```

注記

ルートがアクティベートされる **前に**、起動時に **setUuidGenerator()** メソッドを呼び出す必要があります。

Spring を使用した UUID ジェネレーターの指定

Spring を使用してデフォルトの UUID ジェネレーターを置き換えるには、Spring **bean** 要素を使用して UUID ジェネレーターのインスタンスを作成することのみが必要です**camelContext** インスタンスが作

成されると、Spring レジストリーが自動的に検索され、**org.apache.camel.spi.UuidGenerator** を実装する Bean を検索します。たとえば、以下のように **CamelContext** で **SimpleUuidGenerator** インスタンスを登録することができます。

```
<beans ...>
  <bean id="simpleUuidGenerator"
        class="org.apache.camel.impl.SimpleUuidGenerator" />

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

[1] アクティブなメソッドがない場合は、戻り値に **null** が使用されます。

第35章 プロセッサーの実装

概要

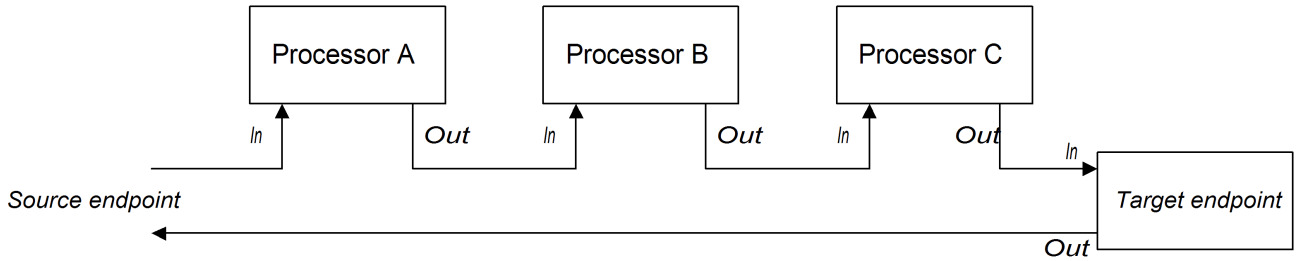
Apache Camel では、カスタムプロセッサーを実装することができます。カスタムプロセッサーをルートへ組み込み、ルートを通過するエクスチェンジオブジェクトに対して操作を実行することができます。

35.1. 処理モデル

パイプラインモデル

パイプラインモデル は、プロセッサーが 「パイプとフィルター」 に配置される方法を記述します。パイプラインは、シーケンス状に並んだエンドポイントを処理する最も一般的な方法です (プロデューサーエンドポイントは、特殊なタイプのプロセッサーに過ぎません)。プロセッサーがこのように配置される場合、エクスチェンジの **In** および **Out** メッセージは、図35.1「パイプラインモデル」 で示すようにに処理されます。

図35.1 パイプラインモデル



パイプライン内のプロセッサーはサービスのように見え、**In** メッセージはリクエストに、**Out** メッセージはリプライに似ています。実際に使用されているパイプラインでは、パイプライン内のノードは、多くの場合で CXF コンポーネントなどの Web サービスエンドポイントによって実装されています。

たとえば、例35.1「Java DSL パイプライン」 は、シークエンス状に並んだ 2 つのプロセッサーである **ProcessorA**、**ProcessorB** と、プロデューサーエンドポイントである **TargetURI** から設定された Java DSL パイプラインを示しています。

例35.1 Java DSL パイプライン

```
from(SourceURI).pipeline(ProcessorA, ProcessorB, TargetURI);
```

35.2. シンプルなプロセッサーの実装

概要

本セクションでは、ルート内の次のプロセッサーにエクスチェンジを委譲する前に、メッセージ処理ロジックを実行する簡単なプロセッサーを実装する方法について説明します。

Processor インターフェイス

シンプルなプロセッサーは、org.apache.camel.Processor インターフェイスを実装して作成されます。例35.2「Processor インターフェイス」 に示されているように、インターフェイスはエクスチェンジオブジェクトを処理する単一のメソッド **process()** を定義しています。

例35.2 Processor インターフェイス

```
package org.apache.camel;

public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

Processor インターフェイスの実装

シンプルなプロセッサーを作成するには、Processor インターフェイスを実装し、**process()** メソッドのロジックを提供する必要があります。例35.3「シンプルなプロセッサーの実装」 は、単純なプロセッサー実装の概要を示しています。

例35.3 シンプルなプロセッサーの実装

```
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() メソッド内のすべてのコードは、エクスチェンジオブジェクトがチェーン内の次のプロセッサーへ委譲される **前** に実行されます。

シンプルなプロセッサー内でメッセージボディーおよびヘッダー値にアクセスする方法については、「メッセージコンテンツへのアクセス」 を参照してください。

シンプルなプロセッサーのルートへの組み込み

process() DSL コマンドを使用して、シンプルなプロセッサーをルートに組み込みます。カスタムプロセッサーのインスタンスを作成し、このインスタンスを引数として **process()** メソッドに渡します。

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

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

35.3. メッセージコンテンツへのアクセス

メッセージヘッダーへのアクセス

メッセージヘッダーは、ルーターサービスで処理されることを意図していることが多いため、ルーターの観点から見て最も有用なメッセージコンテンツを含んでいることが一般的です。ヘッダーデータにアクセスするには、まずエクスチェンジオブジェクトからメッセージを取得してから (例: **Exchange.getIn** を使用)、Message インターフェイスを使用して個別のヘッダーを取得する必要があります (例: **Message.getHeader()** を使用)。

例35.4「認証ヘッダーへのアクセス」 は、**Authorization** という名前のヘッダーの値にアクセスするカスタムプロセッサーの例を示しています。この例では、**ExchangeHelper.getMandatoryHeader()** メソッドを使用しているため、NULL ヘッダー値をテストする必要がなくなります。

例35.4 認証ヘッダーへのアクセス

```
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...
    // ...
  }
}
```

Message インターフェースの詳細については、「メッセージ」 を参照してください。

メッセージボディーへのアクセス

メッセージボディーにもアクセスできます。たとえば、**In** メッセージの末尾に文字列を追加するには、例35.5「メッセージボディーへのアクセス」 に示されているようにプロセッサーを使用することができます。

例35.5 メッセージボディーへのアクセス

```
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!");
    }
}
```

メッセージのアタッチメントへのアクセス

Message.getAttachment() メソッドまたは **Message.getAttachments()** メソッドのいずれかを使用して、メッセージの添付にアクセスできます。詳細は、例34.2「Message インターフェイス」 を参照してください。

35.4. EXCHANGEHELPER クラス

概要

org.apache.camel.util.ExchangeHelper クラスは、プロセッサーの実装に役立つメソッドを提供する Apache Camel のユーティリティークラスです。

エンドポイントの解決

static **resolveEndpoint()** メソッドは、**ExchangeHelper** クラスの最も有用なメソッドの 1 つです。プロセッサー内でこれを使用して、その場で新しい **Endpoint** インスタンスを生成します。

例35.6 resolveEndpoint() メソッド

```
public final class ExchangeHelper {
    ...
    @SuppressWarnings({"unchecked" })
    public static Endpoint
    resolveEndpoint(Exchange exchange, Object value)
        throws NoSuchEndpointException { ... }
    ...
}
```

resolveEndpoint に対する最初の引数はエクスチェンジインスタンスで、2 番目の引数は通常エンドポイント URI 文字列です。例35.7「File エンドポイントの作成」 では、エクスチェンジインスタンス **exchange** から新しい File エンドポイントを作成する方法を示しています。

例35.7 File エンドポイントの作成

```
Endpoint file_endp = ExchangeHelper.resolveEndpoint(exchange, "file://tmp/messages/in.xml");
```

エクスチェンジアクセサーのラップ

ExchangeHelper クラスは、**Exchange** クラスの対応する **getBeanProperty()** メソッドをラップする **getMandatoryBeanProperty()** 形式の static メソッドを複数提供します。これらの違いは、オリジナルの **getBeanProperty()** アクセサーは、対応するプロパティーが利用できない場合に **null** を返し、ラッパーメソッド **getMandatoryBeanProperty()** は、Java の例外を投げることです。以下のラッパーメソッドが **ExchangeHelper** クラスに実装されています。

```
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 { ... }
    ...
}
```

交換パターンのテスト

一部の交換パターンは、**In** メッセージの保持に対応しています。また、一部の交換パターンは、**Out** メッセージの保持にも対応しています。エクスチェンジオブジェクトが **In** メッセージまたは **Out** メッセージを保持することができるかどうかを簡単に確認するために、**ExchangeHelper** クラスは以下のメソッドを提供しています。

```
public final class ExchangeHelper {
    ...
    public static boolean isInCapable(Exchange exchange) { ... }

    public static boolean isOutCapable(Exchange exchange) { ... }
    ...
}
```

In メッセージの MIME コンテンツタイプを取得します。

エクスチェンジの **In** メッセージの MIME コンテンツタイプを確認する場合、**ExchangeHelper.getContentType(exchange)** メソッドを呼び出すことでアクセスすることができます。これを実装するには、**ExchangeHelper** オブジェクトは **In** メッセージのヘッダー **Content-Type** の値を確認します。このメソッドはヘッダー値を設定するため、基礎となるコンポーネントに依存します。

第36章 型コンバーター

概要

Apache Camel には組み込みの型変換メカニズムがあり、メッセージボディーとメッセージヘッダーを異なる型へ変換するために使用されます。本章では、独自の型コンバーターメソッドを追加して、型変換メカニズムを拡張する方法を説明します。

36.1. 型コンバーターアーキテクチャー

概要

本セクションでは、カスタム型コンバーターを作成する際に理解しておく必要がある、型コンバーターメカニズムの全体的なアーキテクチャーについて説明します。組み込みの型コンバーターのみを使用される場合は、34章*メッセージ形式について*を参照してください。

型コンバーターインターフェイス

例36.1「TypeConverter インターフェイス」 は、すべての型コンバーターが実装しなければならない org.apache.camel.TypeConverter インターフェースの定義を示しています。

例36.1 TypeConverter インターフェイス

```
package org.apache.camel;

public interface TypeConverter {
    <T> T convertTo(Class<T> type, Object value);
}
```

controller 型のコンバーター

Apache Camel タイプのコンバーターメカニズムは、コントローラー/ワーカーのパターンに従います。それぞれが限られた数の型変換を実行することができる多数の **worker** 型コンバーターと、スレーブによって実行された型変換を集約する単一の **controller** 型コンバーターがあります。controller 型のコンバーターは、worker 型のコンバーターのフロントエンドとして機能します。コントローラーに型変換の実行を要求すると、コントローラーは適切なワーカーを選択し、変換タスクをそのワーカーに委任します。

変換メカニズムにアクセスするためのエントリーポイントを提供するため、controller 型コンバーターは型変換メカニズムのユーザーにとって最も重要です。起動時に、Apache Camel は Controller 型コンバーターインスタンスを自動的に **CamelContext** オブジェクトに関連付けます。Controller 型コンバーターへの参照を取得するには、**CamelContext.getTypeConverter()** メソッドを呼び出します。たとえば、エクスチェンジオブジェクト (**exchange**) がある場合は、例36.2「controller 型コンバーターの取得」 のように Controller 型コンバーターへの参照を取得できます。

例36.2 controller 型コンバーターの取得

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
```

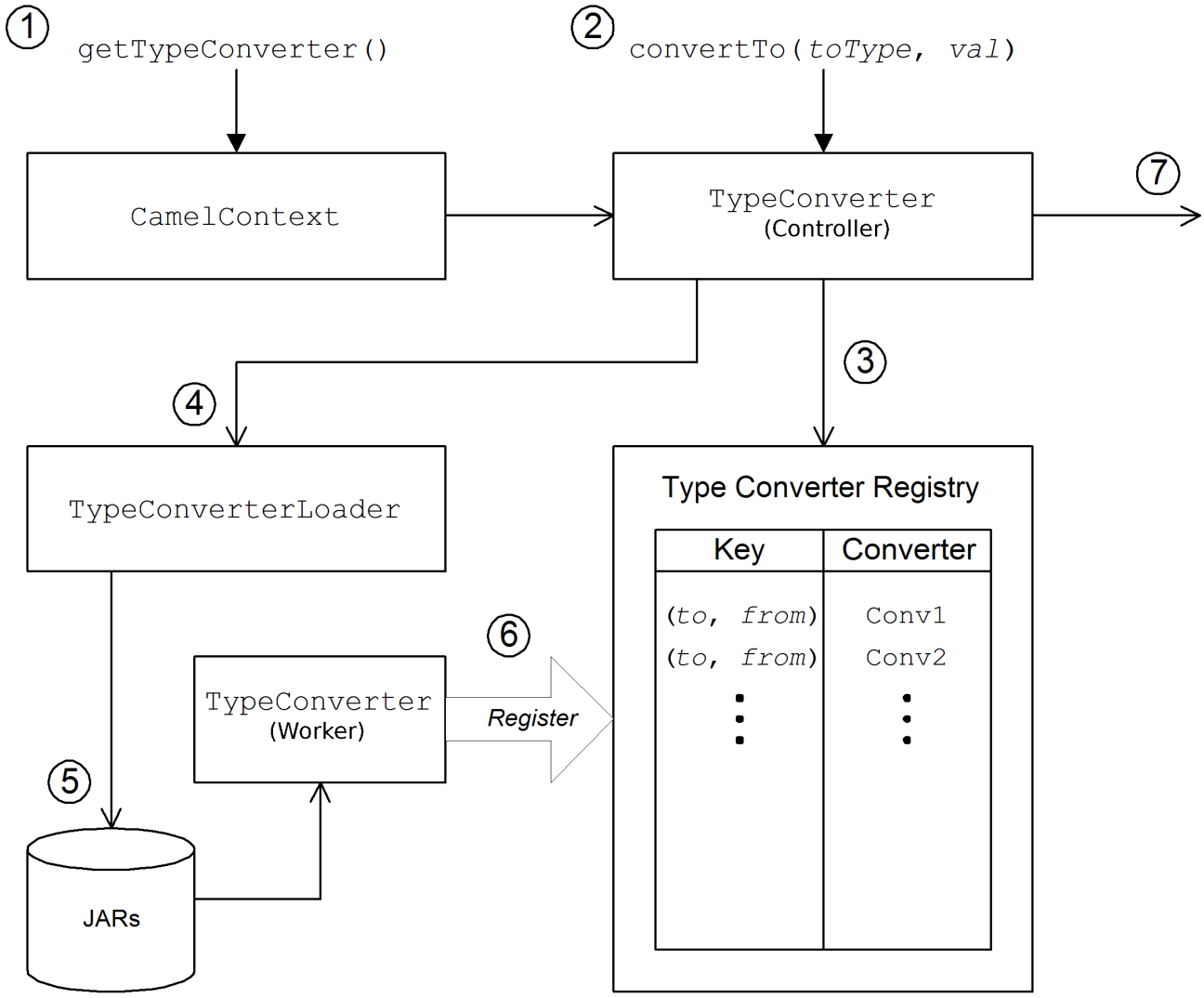
型コンバーターローダー

controller 型コンバーターは、**型コンバータローダー**を使用して、ワーカー型コンバータのレジストリーにデータを入力します。型コンバーターローダーは、TypeConverterLoader インターフェイスを実装するクラスです。Apache Camel は現在、1 種類の型コンバーターローダーのみを使用します。つまり、**アノテーション型コンバーター ローダー** (**AnnotationTypeConverterLoader** 型) のみを使用します。。

型変換プロセス

図36.1「型変換プロセス」 は、型変換プロセスの概要を説明し、与えられたデータ値 (**value**) を指定された型 (**toType**) に変換する際の手順を示しています。

図36.1 型変換プロセス



型変換メカニズムは以下のように行われます。

1. **CamelContext** オブジェクトは、Controller 型コンバーターインスタンスへの参照を保持します。変換プロセスの最初のステップは、**CamelContext.getTypeConverter()** を呼び出し、Controller 型コンバーターを取得することです。

2. Controller 型コンバーターの **convertTo()** メソッドを呼び出すと、型変換が開始されます。このメソッドは、データオブジェクト **value** を元の型から **toType** 引数で指定された型に変換するように、型コンバーターへ指示します。

3. Controller 型コンバーターは多くの異なる Worker 型コンバーターのフロントエンドであるた

め、型マッピングのレジストリーをチェックすることで適切な Worker 型コンバーターを検索します。型コンバーターのレジストリーは、型マッピングのペア **(toType, fromType)** によってキーが設定されています。適切な型コンバーターがレジストリーにある場合、Controller 型コンバーターが Worker の **convertTo()** メソッドを呼び出して、結果を返します。

4. 適切な型コンバーターがレジストリーに **ない** 場合、controller 型コンバーターは型コンバーターローダーを使用して新しい型コンバーターをロードします。

5. 型コンバーターローダーは、クラスパスで利用可能な JAR ライブラリーを検索し、適切な型コンバーターを検索します。現在、使用されるローダーストラテジーは、アノテーション型コンバーターローダーによって実装され、**org.apache.camel.Converter** アノテーションが付けられたクラスをロードしようとします。「TypeConverter ファイルの作成」 を参照してください。

6. 型コンバーターローダーが正常に行われると、新しい worker 型コンバーターがロードされ、型コンバーターレジストリーに登録されます。その後、この型コンバーターは **value** 引数を **toType** 型に変換するために使用されます。

7. データが正常に変換されると、変換されたデータ値が返されます。変換が正常に行われない場合は、**null** が返されます。

36.2. 重複型コンバーターの処理

重複型コンバーターが追加された際の挙動を設定できます。

TypeConverterRegistry (「アノテーションを使用した型コンバーターの実装」 を参照) では、以下のコードを使用してアクションを **Override**、**Ignore**、または **Fail** のいずれかにに設定することができます。

```
typeconverterregistry = camelContext.getTypeConverter()
// Define the behaviour if the TypeConverter already exists
typeconverterregistry.setTypeConverterExists(TypeConverterExists.Override);
```

このコードの **Override** は、要件に応じて **Ignore** または **Fail** に置き換えることができます。

TypeConverterExists クラス

TypeConverterExists クラスは以下のコマンドで設定されます。

```
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. アノテーションを使用した型コンバーターの実装

概要

型変換メカニズムは、新しい worker 型コンバーターを追加することで簡単にカスタマイズできます。ここでは、worker 型コンバーターの実装方法と、アノテーション型コンバーターローダーによって自動的にロードされるように Apache Camel と統合する方法を説明します。

型コンバーターの実装方法

カスタム型コンバーターを実装するには、以下の手順にしたがいます。

1. 「アノテーションが付けられたコンバータークラスの実装」.
2. 「TypeConverter ファイルの作成」.
3. 「型コンバーターのパッケージ化」.

アノテーションが付けられたコンバータークラスの実装

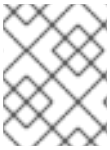
@Converter アノテーションを使用してカスタム型コンバータークラスを実装できます。クラス自体にアノテーションを付け、型変換を実行する各 **static** メソッドにもアノテーションを付ける必要があります。各コンバーターメソッドは **from** 型を定義する引数を取り、オプションで 2 番目の **Exchange** 引数を取り、 **to** 型を定義する void でない戻り値を持ちます。型コンバーターローダーは Java リフレクションを使用してアノテーション付きのメソッドを見つけ、それらを型コンバーターメカニズムに統合します。例36.3「アノテーションが付けられたコンバータークラスの例」 は、**java.io.File** から **java.io.InputStream** へ変換するためのコンバーターメソッドと、**byte[]** から **String** へ変換するための別のコンバーターメソッド (引数 **Exchange** を含む) を定義するアノテーション付きコンバータークラスの例を示しています。

例36.3 アノテーションが付けられたコンバータークラスの例

```
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() メソッドは **File** 型から **InputStream** 型への変換を実行し、**toString()** メソッドは **byte[]** 型から **String** 型への変換を実行します。



注記

メソッド名は重要ではなく、自由に定義できます。重要なのは、引数の型、戻り値の型、および **@Converter** アノテーションです。

TypeConverter ファイルの作成

カスタムコンバーターの検出メカニズム (**アノテーション型コンバーターローダー** で実装されている) を有効にするには、以下の場所に **TypeConverter** ファイルを作成します。

```
META-INF/services/org/apache/camel/TypeConverter
```

この **TypeConverter** ファイルには、型コンバータークラスの FQN (完全修飾名) がコンマ区切りで列挙されたリストが含まれている必要があります。たとえば、型コンバーターローダーが **YourPackageName.YourClassName** パッケージからアノテーション付きコンバータークラスを検索する場合は、**TypeConverter** ファイルの内容は以下のようになります。

```
com.PackageName.FooClass
```

検索メカニズムを有効にする別の方法は、**TypeConverter** ファイルにパッケージ名のみを追加することです。たとえば、**TypeConverter** ファイルの内容は以下のようになります。

```
com.PackageName
```

これにより、パッケージスキャナーが **@Converter** タグのパッケージをスキャンします。FQN メソッドの使用は高速であり、推奨される方法です。

型コンバーターのパッケージ化

型コンバーターは、カスタム型コンバーターのコンパイルされたクラスと **META-INF** ディレクトリーを含む JAR ファイルとしてパッケージ化されます。この JAR ファイルをクラスパスに置き、Apache Camel アプリケーションで利用できるようにします。

フォールバックコンバーターメソッド

@Converter アノテーションを使用して通常のコンバーターメソッドを定義する他に、**@FallbackConverter** アノテーションを使用してフォールバックコンバーターメソッドを定義することもできます。フォールバックコンバーターメソッドは、controller 型コンバーターが型レジストリーで通常のコンバーターメソッドの検索に失敗した場合にのみ試行されます。

通常のコンバーターメソッドとフォールバックコンバーターメソッドの基本的な違いは、通常のコンバーターメソッドが特定の型 (例: **byte[]** から **String** など) のペア間で変換を実行するのに対し、フォールバックコンバーターは、潜在的に**任意** の型のペア間で変換できるところです。どの変換を実行できるかは、フォールバックコンバーターメソッド本体の実装ロジック次第です。実行時に通常のコンバーターで変換できない場合、controller 型コンバーターは変換できるコンバーターを見つけるまで、利用可能なすべてのフォールバックコンバーターを繰り返し実行します。

フォールバックコンバーターのメソッドの署名には、以下のいずれかの形式を使用できます。

```
// 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
)
```

MethodName はフォールバックコンバーターの任意のメソッド名です。

たとえば、以下の抜粋コード (File コンポーネントの実装から取得) は、**GenericFile** オブジェクトのボディーを変換できるフォールバックコンバーターを示し、型コンバーターレジストリーですでに利用可能な型コンバーターを利用しています。

```
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. 型コンバーターの直接実装

概要

通常、型コンバーターを実装する方法として、前述のセクション「アノテーションを使用した型コンバーターの実装」で説明されているように、アノテーション付きクラスを使用することが推奨される方法です。ただし、型コンバーターの登録を完全に制御する場合は、ここで説明するように、カスタムのワーカー型コンバーターを実装し、型コンバーターレジストリーに直接追加することができます。

TypeConverter インターフェイスの実装

独自の型コンバータークラスを実装するには、**TypeConverter** インターフェイスを実装するクラスを定義します。たとえば、以下の **MyOrderTypeConverter** クラスは整数値を **MyOrder** オブジェクトに変換します。整数値は **MyOrder** オブジェクトの order ID を初期化するために使用されます。

```
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);
    }
}
```

型コンバーターのレジストリーへの追加

以下のようなコードを使用して、カスタム型コンバーターを **直接** 型コンバーターレジストリーに追加できます。

```
// Add the custom type converter to the type converter registry
context.getTypeConverterRegistry().addTypeConverter(MyOrder.class, String.class, new
MyOrderTypeConverter());
```

context は、現在の **org.apache.camel.CamelContext** インスタンスです。**addTypeConverter()** メソッドは、**String.class** から **MyOrder.class** への特定の型変換に対して **MyOrderTypeConverter** クラスを登録します。

カスタム型コンバーターは、**META-INF** ファイルを使用せずに Camel アプリケーションに追加することができます。**Spring** または **Blueprint** を使用している場合は、<bean> を宣言すればよいだけです。CamelContext は Bean を自動的に検出し、コンバーターを追加します。

```
<bean id="myOrderTypeConverters" class="..."/>
 <camelContext>
   ...
</camelContext>
```

複数のクラスがある場合は、複数の <bean> を宣言することができます。

第37章 プロデューサーおよびコンシューマーテンプレート

概要

Apache Camel のプロデューサーテンプレートおよびコンシューマーテンプレートは、Spring コンテナー API の機能にちなんでモデル化されています。リソースへのアクセスは、**テンプレート** と呼ばれる単純で使いやすい API を使用して提供されます。Apache Camel の場合、プロデューサーテンプレートおよびコンシューマーテンプレートは、プロデューサーエンドポイントおよびコンシューマーエンドポイントとの間でメッセージを送受信するためのシンプルなインターフェイスを提供します。

37.1. プロデューサーテンプレートの使用

37.1.1. プロデューサーテンプレートの概要

概要

プロデューサーテンプレートは、プロデューサーエンドポイントを呼び出すさまざまな方法をサポートします。リクエストメッセージの異なる形式 (**Exchange** オブジェクト、メッセージボディー、単一のヘッダー設定を持つメッセージボディーなど) をサポートするメソッドや、呼び出しの同期と非同期スタイルの両方をサポートするメソッドがあります。全体的なプロデューサーテンプレートメソッドは、以下のカテゴリーにグループ化できます。

- 同期呼び出し
- プロセッサーを使用した同期呼び出し
- 非同期呼び出し
- コールバックを使用した非同期呼び出し

または、「Fluent Producer テンプレートの使用」 を参照してください。

同期呼び出し

エンドポイントを同期的に呼び出すメソッドには、**sendSuffix()** および **requestSuffix()** という形式の名前があります。たとえば、デフォルトのメッセージ交換パターン (MEP) または明示的に指定された MEP を使用してエンドポイントを呼び出すメソッドには、**send()**、**sendBody()**、および **sendBodyAndHeader()** の名前が付けられます (これらのメソッドがそれぞれ **Exchange** オブジェクト、メッセージボディー、またはメッセージボディーおよびヘッダー値を送信する場合)。MEP を **InOut** (リクエスト/リプライセマンティクス) に強制するには、代わりに **request()**、**requestBody()**、および **requestBodyAndHeader()** メソッドを呼び出します。

以下の例は、**ProducerTemplate** インスタンスを作成し、これを使用してメッセージボディーを **activemq:MyQueue** エンドポイントに送信する方法を示しています。この例は、**sendBodyAndHeader()** を使用してメッセージボディーとヘッダー値を送信する方法も示しています。

```
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" );
```

プロセッサーを使用した同期呼び出し

同期呼び出しの特別なケースとして、**send()** メソッドに引数 **Exchange** ではなく引数 **Processor** を指定することがあります。この場合、プロデューサーテンプレートは暗黙的に指定のエンドポイントに **Exchange** インスタンスを作成するよう指示します (常にではありませんが、通常はデフォルトで **InOnly** MEP を持ちます)。このデフォルトのエクスチェンジはプロセッサーに渡され、エクスチェンジオブジェクトの内容を初期化します。

以下の例は、**MyProcessor** プロセッサーによって初期化されたエクスチェンジを **activemq:MyQueue** エンドポイントに送信する方法を示しています。

```
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 クラスは以下の例のように実装されます。(ここで示したように) **In** メッセージボディーを設定する他に、メッセージヘッダーおよび交換プロパティーを初期化することもできます。

```
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>");
    }
}
```

非同期呼び出し

エンドポイントを **非同期的** に呼び出すメソッドの名前の形式は、**asyncSendSuffix()** および **asyncRequestSuffix()** です。たとえば、デフォルトのメッセージ交換パターン (MEP) または明示的に指定された MEP を使用してエンドポイントを呼び出すメソッドには、**asyncSend()** および **asyncSendBody()** の名前が付けられます (これらのメソッドはそれぞれ **Exchange** オブジェクトまたはメッセージボディーを送信します)。MEP を **InOut** (リクエスト/リプライセマンティクス) に強制するには、代わりに **asyncRequestBody()**、**asyncRequestBodyAndHeader()**、および **asyncRequestBodyAndHeaders()** メソッドを呼び出します。

以下の例は、**direct:start** エンドポイントにエクスチェンジを非同期的に送信する方法を示しています。**asyncSend()** メソッドは、後で呼び出しの結果を取得するために使用される **java.util.concurrent.Future** オブジェクトを返します。

```
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();
```

プロデューサーテンプレートは、メッセージのボディーを非同期的に送信するメソッドも提供します (例: **asyncSendBody()** または **asyncRequestBody()** を使用します)。この場合、以下のヘルパーメソッドのいずれかを使用して、**Future** オブジェクトから返されたメッセージのボディーを抽出できます。

```
<T> T extractFutureBody(Future future, Class<T> type);
<T> T extractFutureBody(Future future, long timeout, TimeUnit unit, Class<T> type) throws
TimeoutException;
```

呼び出しが完了し、リプライメッセージが表示されるまで **extractFutureBody()** メソッドの最初のバージョンはブロックされます。**extractFutureBody()** メソッドの 2 番目のバージョンでは、タイムアウトを指定できます。どちらのメソッドも type 引数 **type** を持ち、組み込み型コンバーターを使用して、返されるメッセージのボディーを指定された型にキャストします。

以下の例は、**asyncRequestBody()** メソッドを使用してメッセージボディーを **direct:start** エンドポイントに送信する方法を示しています。次に、ブロッキングしている **extractFutureBody()** メソッドを使用して **Future** オブジェクトからリプライメッセージのボディーを取得します。

```
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);
```

コールバックを使用した非同期呼び出し

上記の非同期例では、リクエストメッセージはサブスレッドでディスパッチされ、応答はメインスレッドによって取得および処理されます。しかし、プロデューサーテンプレートでも、**asyncCallback()**、**asyncCallbackSendBody()**、または **asyncCallbackRequestBody()** メソッドのいずれかを使用して、サブスレッドで返信を処理するオプションもあります。この場合、コールバックオブジェクト (**org.apache.camel.impl.SynchronizationAdapter** タイプ) を指定します。このオブジェクトは、リプライメッセージが到達するとすぐに自動的にサブスレッドで呼び出されます。

Synchronization コールバックインターフェイスは、以下のように定義されます。

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

import org.apache.camel.Exchange;
```

```
public interface Synchronization {
    void onComplete(Exchange exchange);
    void onFailure(Exchange exchange);
}
```

ここで、通常の応答の受信時に **onComplete()** メソッドが呼び出され、障害メッセージの応答を受け取る際に **onFailure()** メソッドが呼び出されます。これらのメソッドの 1 つのみが呼び出されるため、両方のメソッドを上書きして、すべてのタイプの応答が処理されるようにする必要があります。

以下の例は、リプライメッセージが **SynchronizationAdapter** コールバックオブジェクトによってサブスレッドで処理される、**direct:start** エンドポイントにエクスチェンジを送信する方法を示しています。

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

SynchronizationAdapter クラスは **Synchronization** インターフェイスのデフォルト実装で、**onComplete()** および **onFailure()** コールバックメソッドの独自の定義を上書きできます。

asyncCallback() メソッドによって **Future** object vulkan-conditionals も返すため、メインスレッドから応答にアクセスするオプションは依然としてあります。

```
// Retrieve the reply from the main thread, specifying a timeout
Exchange reply = future.get(10, TimeUnit.SECONDS);
```

37.1.2. 同期送信

概要

同期送信 メソッドは、プロデューサーエンドポイントの呼び出しに使用できるメソッドのコレクションです。メソッド呼び出しが完了し、応答 (ある場合) が受信されるまで現在のスレッドをブロックします。これらのメソッドは、あらゆる種類のメッセージ交換プロトコルと互換性があります。

エクスチェンジの送信

基本的な **send()** メソッドは、エクスチェンジのメッセージ交換パターン (MEP) を使用して、**Exchange** オブジェクトのコンテンツをエンドポイントに送信する汎用メソッドです。戻り値は、プロデューサーエンドポイントによって処理された後に取得するエクスチェンジです (MEP によっては

Out メッセージが含まれる可能性があります)。

デフォルトのエンドポイント、エンドポイント URI、または **send()** オブジェクトとしてターゲットエンドポイントを指定できるエクスチェンジの送信では、3 種類の **Endpoint** メソッドがあります。

```
Exchange send(Exchange exchange);
Exchange send(String endpointUri, Exchange exchange);
Exchange send(Endpoint endpoint, Exchange exchange);
```

プロセッサーによって提供されたエクスチェンジの送信

一般的な **send()** メソッドは、エクスチェンジオブジェクトを明示的に提供するのではなく、プロセッサーを使用してデフォルトのエクスチェンジを設定することです (詳細は 「プロセッサーを使用した同期呼び出し」 を参照してください)。

プロセッサーによって設定されるエクスチェンジを送信する **send()** メソッドにより、デフォルトのエンドポイント、エンドポイント URI、または **Endpoint** オブジェクトとしてターゲットエンドポイントを指定できます。さらに、オプションでエクスチェンジの MEP を指定するには、デフォルトを許可せずに、**pattern** 引数を指定します。

```
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
);
```

メッセージボディーの送信

送信するメッセージボディーの内容のみに関心がある場合は、**sendBody()** メソッドを使用してメッセージボディーを引数として提供し、プロデューサーテンプレートにボディーをデフォルトのエクスチェンジオブジェクトに挿入させることができます。

sendBody() メソッドにより、ターゲットエンドポイントをデフォルトのエンドポイント、エンドポイント URI、または **Endpoint** オブジェクトとして指定できます。さらに、オプションでエクスチェンジの MEP を指定するには、デフォルトを許可せずに、**pattern** 引数を指定します。**pattern** 引数の **ない** メソッドは、**void** を返します (呼び出しによって応答が発生する可能性もあります)。また、**pattern** 引数の **ある** メソッドは、**Out** メッセージのボディー (ある場合) または **In** メッセージのボディー (それ以外の場合) のいずれかを返します。

```
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
);
```

メッセージボディーとヘッダーの送信

テストの目的で、**単一** のヘッダー設定の影響を試すことが望ましいことが多く、**sendBodyAndHeader()** メソッドはこのようなヘッダーテストに役立ちます。メッセージボディーとヘッダー設定を引数として **sendBodyAndHeader()** に提供し、プロデューサーテンプレートがボディーとヘッダー設定をデフォルトのエクスチェンジオブジェクトに挿入できるようにします。

sendBodyAndHeader() メソッドにより、ターゲットエンドポイントをデフォルトのエンドポイント、エンドポイント URI、または **Endpoint** オブジェクトとして指定できます。さらに、オプションでエクスチェンジの MEP を指定するには、デフォルトを許可せずに、**pattern** 引数を指定します。**pattern** 引数の **ない** メソッドは、**void** を返します (呼び出しによって応答が発生する可能性もあります)。また、**pattern** 引数の **ある** メソッドは、**Out** メッセージのボディー (ある場合) または **In** メッセージのボディー (それ以外の場合) のいずれかを返します。

```
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() メソッドは **sendBodyAndHeader()** メソッドと似ていますが、単一のヘッダー設定を指定する代わりに、ヘッダー設定の完全なハッシュマップを指定することができます。

```
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
);
```

メッセージボディーおよびエクスチェンジプロパティーの送信

sendBodyAndProperty() メソッドを使用して、単一のエクスチェンジプロパティーを設定する効果を試すことができます。**sendBodyAndProperty()** にメッセージボディーとプロパティー設定を引数として提供し、プロデューサーテンプレートがボディーを挿入し、エクスチェンジプロパティーをデフォルトのエクスチェンジオブジェクトに挿入できるようにします。

sendBodyAndProperty() メソッドにより、ターゲットエンドポイントをデフォルトのエンドポイント、エンドポイント URI、または **Endpoint** オブジェクトとして指定できます。さらに、オプションでエクスチェンジの MEP を指定するには、デフォルトを許可せずに、**pattern** 引数を指定します。**pattern** 引数の **ない** メソッドは、**void** を返します (呼び出しによって応答が発生する可能性もあります)。また、**pattern** 引数の **ある** メソッドは、**Out** メッセージのボディー (ある場合) または **In** メッセージのボディー (それ以外の場合) のいずれかを返します。

```
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. InOut パターンでの同期リクエスト

概要

同期リクエスト メソッドは同期送信メソッドと似ていますが、リクエストメソッドはメッセージ交換パターンを強制的に **InOut** にします (リクエスト/リプライセマンティクスに準拠)。そのため、プロデューサーエンドポイントから応答を受信することが予想される場合は、一般的に同期リクエストメソッドを使用することが推奨されます。

プロセッサーによって設定されたエクスチェンジの要求

基本的な **request()** メソッドは、プロセッサーを使用してデフォルトのエクスチェンジの設定を行い、メッセージ交換パターンを **InOut** に強制する (よって呼び出しがリクエスト/リプライセマンティクスに従う) 汎用メソッドです。戻り値は、プロデューサーエンドポイントによって処理された後に得られるエクスチェンジです。**Out** メッセージにはリプライメッセージが含まれます。

プロセッサーによって設定されるエクスチェンジを送信するために **request()** メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとしてターゲットエンドポイントを指定できます。

```
Exchange request(String endpointUri, Processor processor);
Exchange request(Endpoint endpoint, Processor processor);
```

メッセージボディーの要求

リクエストと応答のメッセージボディーの内容のみに関心がある場合は、**requestBody()** メソッドを使用してリクエストメッセージのボディーを引数として提供し、プロデューサーテンプレートにボディーをデフォルトのエクスチェンジオブジェクトに挿入させることができます。

requestBody() メソッドにより、ターゲットエンドポイントをデフォルトのエンドポイント、エンドポイント URI、または **Endpoint** オブジェクトとして指定できます。戻り値は、リプライメッセージのボディー (**Out** メッセージボディー) で、プレーン **Object** として返されるか、組み込み型コンバーター (「組み込み型コンバーター」 を参照) を使用して特定のタイプ **T** に変換されます。

```
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
);
```

メッセージのボディーおよびヘッダーの要求

requestBodyAndHeader() メソッドを使用して、単一のヘッダー値を設定する効果を試すことができます。メッセージボディーとヘッダー設定を引数として **requestBodyAndHeader()** に提供し、プロデューサーテンプレートがボディーを挿入し、プロパティーをデフォルトのエクスチェンジオブジェクトに挿入できるようにします。

requestBodyAndHeader() メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。戻り値は、リプライメッセージのボディー (**Out** メッセージボディー) で、プレーン **Object** として返されるか、組み込み型コンバーター (「組み込み型コンバーター」 を参照) を使用して特定のタイプ **T** に変換されます。

```
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
);
```

requiestBodyAndHeaders() メソッドは **requestBodyAndHeader()** メソッドと似ていますが、単一のヘッダー設定を指定する代わりに、ヘッダー設定の完全なハッシュマップを指定することができます。

```
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. 非同期送信

概要

プロデューサーテンプレートは、プロデューサーエンドポイントを非同期的に呼び出すさまざまな方法を提供します。これにより、メインスレッドが呼び出しの完了を待機している間にブロックされず、リプライメッセージが後で取得できます。本項で説明されている非同期送信方法は、あらゆる種類のメッセージ交換プロトコルと互換性があります。

エクスチェンジの送信

基本的な **asyncSend()** メソッドは **Exchange** 引数を取り、指定されたエクスチェンジのメッセージ交換パターン (MEP) を使用してエンドポイントを非同期的に呼び出します。戻り値は **java.util.concurrent.Future** オブジェクトで、後でリプライメッセージを収集するために使用できるチケットです。**Future** オブジェクトから戻り値を取得する方法の詳細は、「非同期呼び出し」 を参照してください。

以下の **asyncSend()** メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。

```
Future<Exchange> asyncSend(String endpointUri, Exchange exchange);
Future<Exchange> asyncSend(Endpoint endpoint, Exchange exchange);
```

プロセッサーによって提供されたエクスチェンジの送信

一般的な **asyncSend()** メソッドは、エクスチェンジオブジェクトを明示的に提供するのではなく、プロセッサーを使用してデフォルトのエクスチェンジを設定することです。

以下の **asyncSend()** メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。

```
Future<Exchange> asyncSend(String endpointUri, Processor processor);
Future<Exchange> asyncSend(Endpoint endpoint, Processor processor);
```

メッセージボディーの送信

送信するメッセージボディーの内容のみに関心がある場合は、**asyncSendBody()** メソッドを使用してメッセージボディーを非同期的に送信し、プロデューサーテンプレートにボディーをデフォルトのエクスチェンジオブジェクトに挿入させることができます。

asyncSendBody() メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。

```
Future<Object> asyncSendBody(String endpointUri, Object body);
Future<Object> asyncSendBody(Endpoint endpoint, Object body);
```

37.1.5. InOut パターンを使用した非同期リクエスト

概要

非同期リクエスト メソッドは非同期送信メソッドと似ていますが、リクエストメソッドはメッセージ交換パターンを強制的に **InOut** にします (リクエスト/リプライセマンティクスに準拠)。そのため、プロデューサーエンドポイントから応答を受信することが予想される場合は、通常は非同期リクエストメソッドを使用することが推奨されます。

メッセージボディーの要求

リクエストと応答のメッセージボディーの内容のみに関心がある場合は、**requestBody()** メソッドを使用してリクエストメッセージのボディーを引数として提供し、プロデューサーテンプレートにボディーをデフォルトのエクスチェンジオブジェクトに挿入させることができます。

asyncRequestBody() メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。**Future** オブジェクトから取得できる戻り値は、リプライメッセージのボディー (**Out** メッセージボディー) です。これは、プレーン **Object** として返されるか、組み込み型コンバーターを使用して特定タイプ **T** に変換されます (「非同期呼び出し」を参照)。

```
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
);
```

メッセージのボディーおよびヘッダーの要求

asyncRequestBodyAndHeader() メソッドを使用して、単一のヘッダー値を設定する効果を試すことができます。メッセージボディーとヘッダー設定を引数として **asyncRequestBodyAndHeader()** に提供し、プロデューサーテンプレートがボディーを挿入し、プロパティーをデフォルトのエクスチェンジオブジェクトに挿入できるようにします。

asyncRequestBodyAndHeader() メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。**Future** オブジェクトから取得できる戻り値は、リプライメッセージのボディー (**Out** メッセージボディー) です。これは、プレーン **Object** として返されるか、組み込み型コンバーターを使用して特定タイプ **T** に変換されます (「非同期呼び出し」 を参照)。

```
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
);
```

asyncRequiestBodyAndHeaders() メソッドは **asyncRequestBodyAndHeader()** メソッドと似ていますが、単一のヘッダー設定を指定する代わりに、ヘッダー設定の完全なハッシュマップを指定することができます。

```
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. コールバックを使用した非同期送信

概要

プロデューサーテンプレートは、プロデューサーエンドポイントを呼び出すために使用される同じサブスレッドでリプライメッセージを処理するオプションも提供します。この場合、コールバックオブジェクトを作成します。このオブジェクトは、リプライメッセージが受信されるとすぐに自動的にサブスレッドで呼び出されます。つまり、**コールバックメソッドを使用した非同期送信** により、メインスレッドで呼び出しを開始でき、(サブスレッドで非同期的に発生する) プロデューサーエンドポイントの呼び出し、応答の待機、応答の処理をすべて実行できます。

エクスチェンジの送信

基本的な **asyncCallback()** メソッドは **Exchange** 引数を取り、指定されたエクスチェンジのメッセージ交換パターン (MEP) を使用してエンドポイントを非同期的に呼び出します。このメソッドはエクスチェンジの **asyncSend()** メソッドと似ていますが、**onComplete()** および **onFailure()** という 2 つのメソッドを持つコールバックインターフェイスである **org.apache.camel.spi.Synchronization** を追加の引数で必要とする点が異なります。**Synchronization** コールバックの使用方法の詳細は、「コールバックを使用した非同期呼び出し」 を参照してください。

以下の **asyncCallback()** メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。

```
Future<Exchange> asyncCallback(
    String endpointUri,
    Exchange exchange,
    Synchronization onCompletion
);
Future<Exchange> asyncCallback(
    Endpoint endpoint,
```

```
    Exchange exchange,
    Synchronization onCompletion
);
```

プロセッサーによって提供されたエクスチェンジの送信

プロセッサーの **asyncCallback()** メソッドは、プロセッサーを呼び出してデフォルトのエクスチェンジの設定を行い、メッセージ交換パターンを強制的に **InOut** にします (よって呼び出しがリクエスト/リプライセマンティクスに従います)。

以下の **asyncCallback()** メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。

```
Future<Exchange> asyncCallback(
    String endpointUri,
    Processor processor,
    Synchronization onCompletion
);
Future<Exchange> asyncCallback(
    Endpoint endpoint,
    Processor processor,
    Synchronization onCompletion
);
```

メッセージボディーの送信

送信するメッセージボディーの内容のみに関心がある場合は、**asyncCallbackSendBody()** メソッドを使用してメッセージボディーを非同期的に送信し、プロデューサーテンプレートにボディーをデフォルトのエクスチェンジオブジェクトに挿入させることができます。

asyncCallbackSendBody() メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。

```
Future<Object> asyncCallbackSendBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackSendBody(
    Endpoint endpoint,
    Object body,
    Synchronization onCompletion
);
```

メッセージボディーの要求

リクエストと応答のメッセージボディーの内容のみに関心がある場合は、**asyncCallbackRequestBody()** メソッドを使用してリクエストメッセージのボディーを引数として提供し、プロデューサーテンプレートにボディーをデフォルトのエクスチェンジオブジェクトに挿入させることができます。

asyncCallbackRequestBody() メソッドを使用すると、エンドポイント URI または **Endpoint** オブジェクトとして、ターゲットエンドポイントを指定することができます。

```
Future<Object> asyncCallbackRequestBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackRequestBody(
    Endpoint endpoint,
    Object body,
    Synchronization onCompletion
);
```

37.2. FLUENT PRODUCER テンプレートの使用

Camel 2.18 から利用可能

FluentProducerTemplate インターフェイスは、プロデューサーを構築するための Fluent 構文を提供します。**DefaultFluentProducerTemplate** クラスは **FluentProducerTemplate** を実装します。

以下の例では、**DefaultFluentProducerTemplate** オブジェクトを使用してヘッダーとボディーを設定します。

```
Integer result = DefaultFluentProducerTemplate.on(context)
    .withHeader("key-1", "value-1")
    .withHeader("key-2", "value-2")
    .withBody("Hello")
    .to("direct:inout")
    .request(Integer.class);
```

以下の例は、**DefaultFluentProducerTemplate** オブジェクトでプロセッサーを指定する方法を示しています。

```
Integer result = DefaultFluentProducerTemplate.on(context)
    .withProcessor(exchange -> exchange.getIn().setBody("Hello World"))
    .to("direct:exception")
    .request(Integer.class);
```

以下の例では、デフォルトの Fluent Producer テンプレートをカスタマイズする方法を表しています。

```
Object result = DefaultFluentProducerTemplate.on(context)
    .withTemplateCustomizer(
        template -> {
            template.setExecutorService(myExecutor);
            template.setMaximumCacheSize(10);
        }
    )
    .withBody("the body")
    .to("direct:start")
    .request();
```

FluentProducerTemplate インスタンスを作成するには、Camel コンテキストで **createFluentProducerTemplate()** メソッドを呼び出します。以下に例を示します。

```
FluentProducerTemplate fluentProducerTemplate = context.createFluentProducerTemplate();
```

37.3. コンシューマーテンプレートの使用

概要

コンシューマーテンプレートは、受信メッセージを受信するためにコンシューマーエンドポイントをポーリングする方法を提供します。受信メッセージをエクスチェンジオブジェクトの形式またはメッセージボディーの形式で受信するかを選択できます (メッセージボディーは組み込み型コンバーターを使用して特定の型にキャストできます)。

エクスチェンジのポーリングの例

コンシューマーテンプレートを使用し、ブロッキングの **receive()**、タイムアウトのある **receive()**、もしくは即座に返される **receiveNoWait()** ポーリングメソッドのいずれかを使用して、エクスチェンジのコンシューマーエンドポイントをポーリングできます。コンシューマーエンドポイントはサービスを表すため、エクスチェンジのポーリングを試みる前に、**start()** 呼び出しを行ってサービススレッドを開始する必要があります。

以下の例は、ブロッキング **receive()** メソッドを使用して **seda:foo** コンシューマーエンドポイントからエクスチェンジをポーリングする方法を示しています。

```
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();
```

コンシューマーテンプレートインスタンス **consumer** は、**CamelContext.createConsumerTemplate()** メソッドを使用してインスタンス化され、コンシューマーサービススレッドは **ConsumerTemplate.start()** 呼び出しによって開始されます。

メッセージボディーをポーリングする例

また、ブロッキング **receiveBody()**、タイムアウトのある **receiveBody()**、または即座に返される **receiveBodyNoWait()** メソッドのいずれかを使用して、受信メッセージボディーのコンシューマーエンドポイントをポーリングすることもできます。上記の例のように、エクスチェンジをポーリングする前に **start()** を呼び出してサービススレッドを起動する必要もあります。

以下の例は、ブロッキング **receiveBody()** メソッドを使用して **seda:foo** コンシューマーエンドポイントから受信メッセージボディーをポーリングする方法を示しています。

```
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();
```

エクスチェンジをポーリングする方法

コンシューマーエンドポイントから **エクスチェンジ** をポーリングするための基本的な方法には、タイムアウトブロックが無期限にない **receive()**、指定されたミリ秒の期間にタイムアウトブロックが設定された **receive()**、およびブロックされない **receiveNoWait()** の 3 つがあります。エンドポイント URI または **Endpoint** インスタンスとして、コンシューマーエンドポイントを指定できます。

```
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);
```

メッセージボディーのポーリング方法

コンシューマーエンドポイントから **メッセージボディ** をポーリングするための基本的な方法には、タイムアウトブロックが無期限にない **receive()**、指定されたミリ秒の期間にタイムアウトブロックが設定された **receive()**、およびブロックされない **receiveNoWait()** の 3 つがあります。エンドポイント URI または **Endpoint** インスタンスとして、コンシューマーエンドポイントを指定できます。さらに、これらのメソッドのテンプレート形式を呼び出すことで、組み込み型コンバーターを使用して、返されるボディーを特定のタイプ **T** に変換できます。

```
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 コンポーネントの実装

概要

本章では、Apache Camel コンポーネントを実装するための概要を示します。

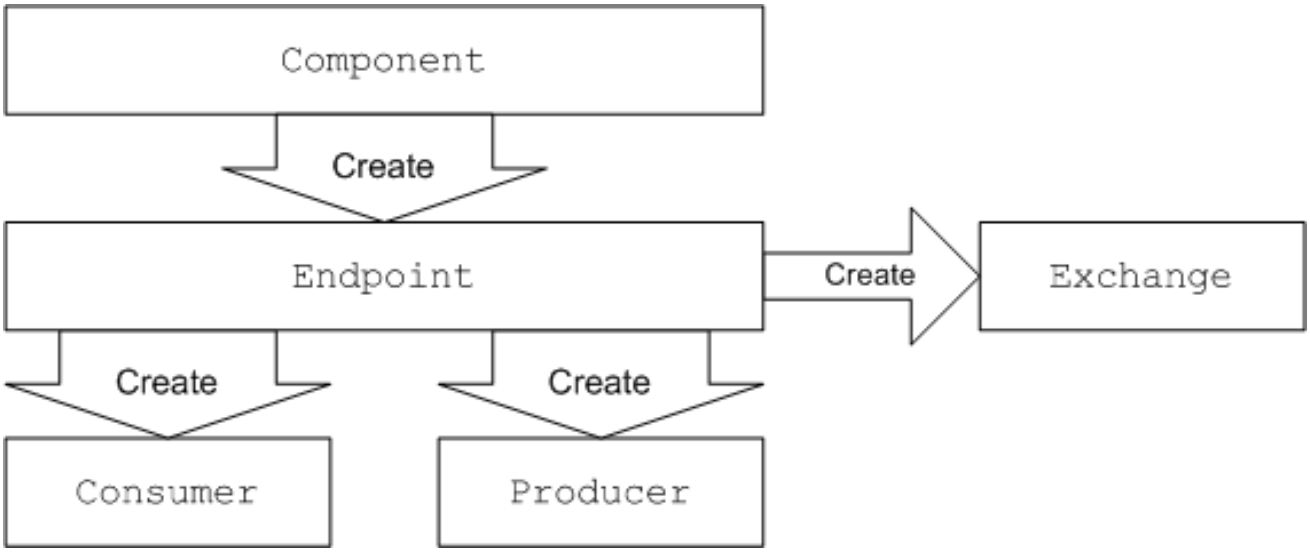
38.1. コンポーネントのアーキテクチャー

38.1.1. コンポーネントのファクトリーパターン

概要

Apache Camel コンポーネントは、ファクトリーパターンを介して相互に関連するクラスセットで設定されます。主なエントリーポイントは **Component** コンポーネントオブジェクト自体です (**org.apache.camel.Component** タイプのインスタンス)。**Coponnet** オブジェクトは、**Endpoint** オブジェクトを作成するためのファクトリーとして使用することができます。エンドポイントオブジェクトは、**Consumer**、**Producer**、および **Exchange** オブジェクトを作成するためのファクトリーとして機能します。これらの関係の概要は、図38.1「コンポーネントファクトリーパターン」 にまとめられています。

図38.1 コンポーネントファクトリーパターン



コンポーネント

コンポーネント実装はエンドポイントファクトリーです。コンポーネント実装の主なタスクは、**Component.createEndpoint()** メソッドを実装することです。このメソッドは、オンデマンドで新しいエンドポイントを作成します。

各種類のコンポーネントは、エンドポイント URI に表示される **コンポーネント接頭辞** に関連付ける必要があります。たとえば、ファイルコンポーネントは、通常 file://tmp/messages/input などのエンドポイント URI で使用できる **file** 接頭辞に関連付けられます。Apache Camel に新しいコンポーネントをインストールする場合、特定のコンポーネント接頭辞とコンポーネントを実装するクラス名の関連付けを定義する必要があります。

エンドポイント

各エンドポイントインスタンスは特定のエンドポイント URI をカプセル化します。Apache Camel が新

しいエンドポイント URI に遭遇するたびに、新しいエンドポイントインスタンスが作成されます。エンドポイントオブジェクトは、コンシューマーエンドポイントおよびプロデューサーエンドポイントを作成するためのファクトリーでもあります。

エンドポイントは org.apache.camel.Endpoint インターフェイスを実装する必要があります。Endpoint インターフェイスは、以下のファクトリーメソッドを定義します。

- **createConsumer()** および **createPollingConsumer()** コンシューマーエンドポイントを作成します。コンシューマーエンドポイントは、ルートの最初のソースエンドポイントを表します。
- **createProducer()**: ルートの最後にターゲットエンドポイントを表すプロデューサーエンドポイントを作成します。
- **createExchange()**: エクスチェンジオブジェクトを作成します。これにより、ルート上のメッセージをカプセル化します。

コンシューマー

コンシューマーエンドポイントはリクエストを **消費** します。これらはルートの先頭に現れ、受信したリクエストおよび応答のディスパッチを行うコードをカプセル化します。サービス指向の考え方から、コンシューマーが **サービス** を表します。

コンシューマーは org.apache.camel.Consumer インターフェイスを実装する必要があります。コンシューマーの実装時には、フォローできるさまざまなパターンがあります。これらのパターンは、「コンシューマーパターンおよびスレッド」 で説明されています。

プロデューサー

プロデューサーエンドポイントはリクエストを **生成** します。これらはルートの最後に常に表示され、リクエスト送信のディスパッチおよび応答を受信するコードをカプセル化します。サービス指向の考え方から、プロデューサーは**サービスコンシューマー**を表します。

プロデューサーは、**org.apache.camel.Producer** インターフェイスを実装する必要があります。オプションでプロデューサーを実装し、非同期処理をサポートすることができます。詳細は、「非同期処理」 を参照してください。

エクスチェンジ

エクスチェンジオブジェクトは関連するメッセージセットをカプセル化します。たとえば、メッセージエクスチェンジの 1 つは、要求メッセージとその関連する応答で設定される同期呼び出しです。

エクスチェンジは org.apache.camel.Exchange インターフェイスを実装する必要があります。デフォルトの実装である **DefaultExchange** は、多くのコンポーネントの実装には十分です。ただし、エクスチェンジと追加のデータを関連付ける場合や、エクスチェンジに追加の処理を行う場合、エクスチェンジの実装をカスタマイズすると便利です。

メッセージ

Exchange オブジェクトには、2 つの異なるメッセージスロットがあります。

- **in** メッセージ: 現在のメッセージを保持します。
- **out** メッセージ: リプライメッセージを一時的に保持します。

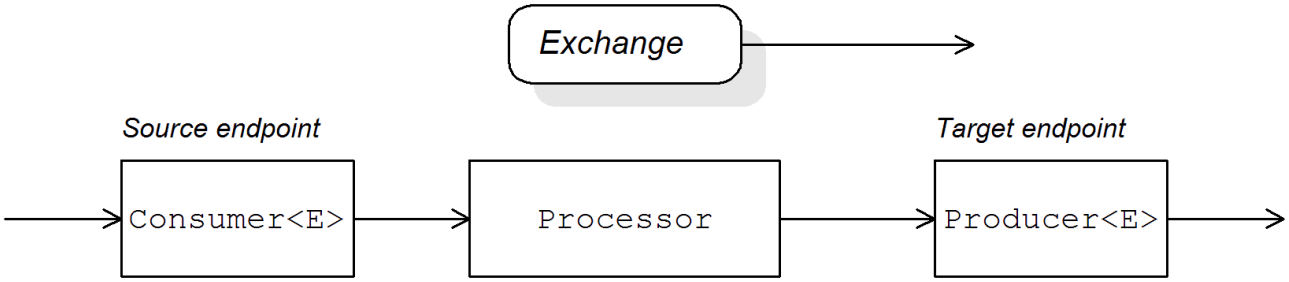
すべてのメッセージタイプは同じ Java オブジェクト **org.apache.camel.Message** によって表されます。デフォルト実装のメッセージ **DefaultMessage** は通常、カスタマイズする必要はありません。

38.1.2. ルートでのコンポーネントの使用

概要

Apache Camel ルートは基本的に org.apache.camel.Processor タイプのプロセッサーのパイプラインです。メッセージは、**process()** メソッドを呼び出してノードからノードに渡されるエクスチェンジオブジェクト **process()** でカプセル化されます。プロセッサーパイプラインのアーキテクチャーは、図 38.2「ルートのコンシューマーおよびプロデューサーインスタンス」 に示されています。

図38.2 ルートのコンシューマーおよびプロデューサーインスタンス



ソースエンドポイント

ルートの最初には、**org.apache.camel.Consumer** オブジェクトで表現されるソースエンドポイントがあります。ソースエンドポイントは、受信したリクエストメッセージを受け入れ、応答をディスパッチします。ルートを構築する際、Apache Camel は、「コンポーネントのファクトリーパターン」 で説明されているように、エンドポイント URI のコンポーネント接頭辞に基づいて、適切な **Consumer** タイプを作成します。

プロセッサー

パイプラインの各中間ノードは、プロセッサーオブジェクトによって表されます (org.apache.camel.Processor インターフェイスの実装)。標準のプロセッサー (たとえば **filter**、**throttler**、または **delayer**) を挿入したり、独自のカスタムプロセッサー実装を挿入したりできます。

ターゲットエンドポイント

ルートの最後には、**org.apache.camel.Producer** オブジェクトで表現されるターゲットエンドポイントがあります。プロセッサーパイプラインの最後にあるため、プロデューサーはプロセッサーオブジェクトでもあります (org.apache.camel.Processor インターフェイスを実装します)。ターゲットエンドポイントは、リクエストメッセージを送信し、応答を受信します。ルートを作成するとき、Apache Camel はエンドポイント URI からコンポーネント接頭辞に基づいて適切な **Producer** タイプを作成します。

38.1.3. コンシューマーパターンおよびスレッド

概要

コンシューマーの実装に使用されるパターンは、受信エクスチェンジの処理に使用されるスレッドモデルを決定します。コンシューマーは、以下のパターンのいずれかを使用して実装できます。

- イベント駆動型のパターン: コンシューマーは外部のスレッドによって実行されます。
- スケジュールされたポーリングパターン: コンシューマーは専用のスレッドプールによって実行されます。

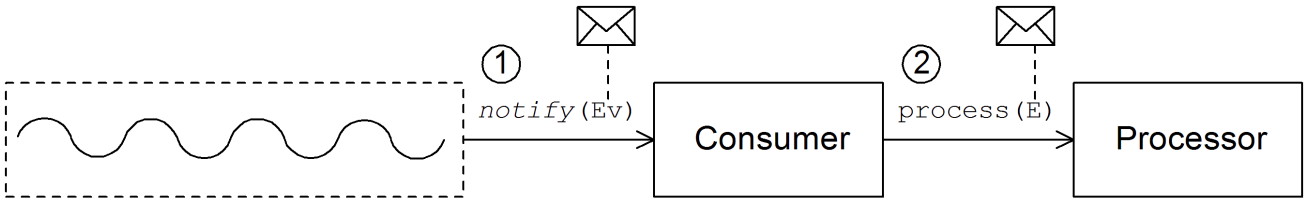
- ポーリングパターン: スレッドモデルは未定義のままです。

イベント駆動型のパターン

イベント駆動型のパターンでは、アプリケーションの別の部分 (通常はサードパーティーライブラリー) がコンシューマーによって実装されたメソッドを呼び出すと、受信したリクエストの処理が開始されます。イベント駆動型のコンシューマーの適切な例として、イベントが JMX ライブラリーによって開始される Apache Camel JMX コンポーネントがあります。JMX ライブラリーは、**handleNotification()** メソッドを呼び出してリクエスト処理を開始します。詳細は 例41.4「JMXConsumer 実装」 を参照してください。

図38.3「イベント駆動型コンシューマー」 イベント駆動型のコンシューマーパターンの概要を示します。この例では、**notify()** メソッドへの呼び出しによって処理がトリガーされることを前提としています。

図38.3 イベント駆動型コンシューマー



イベント駆動型のコンシューマーは、以下のように受信リクエストを処理します。

1. コンシューマーは受信イベントを受信するメソッドを実装する必要があります (図38.3「イベント駆動型コンシューマー」 では **notify()** メソッドで表されます)。通常、**notify()** を呼び出すスレッドはアプリケーションの別の部分であるため、コンシューマーのスレッドポリシーは外部で実行されます。
   たとえば、JMX コンシューマーの実装では、コンシューマーは JMX から通知を受け取る **NotificationListener.handleNotification()** メソッドを実装します。コンシューマー処理を駆動するスレッドは JMX レイヤー内に作成されます。

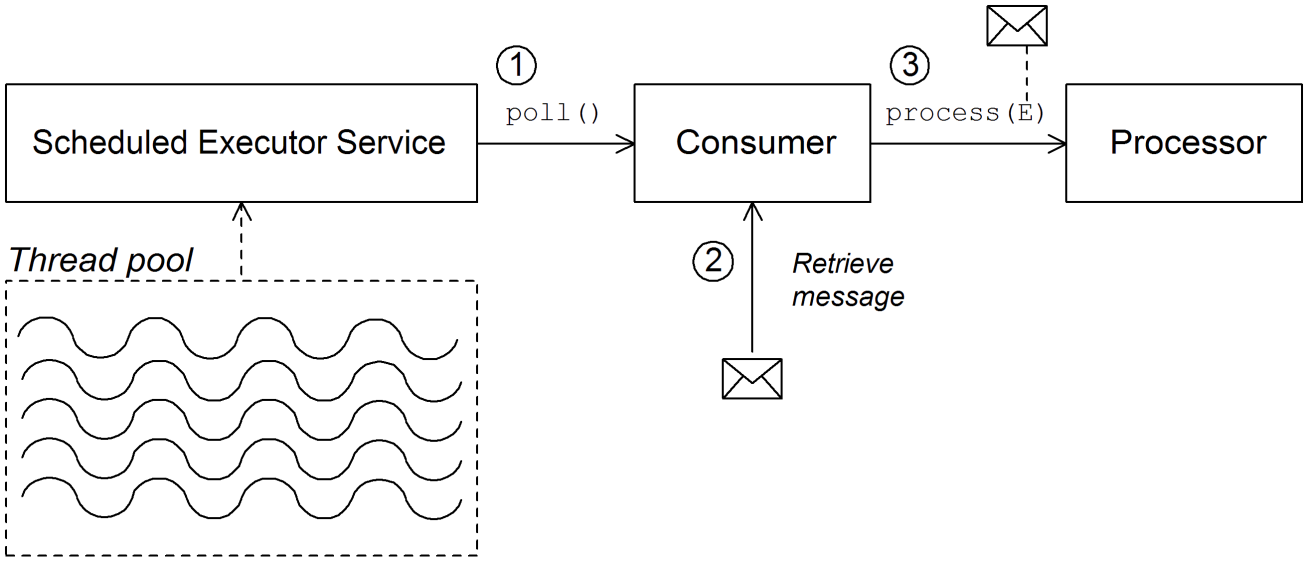
2. **notify()** メソッドの本文では、コンシューマーが最初に受信イベントをエクスチェンジオブジェクト **E** に変換し、ルートの次のプロセッサーを **process()** で呼び出しを行い、エクスチェンジオブジェクトを引数として渡します。

スケジュールされたポーリングパターン

スケジュールされたポーリングパターンでは、リクエストが到達したかどうかを定期的にチェックして、コンシューマーは受信リクエストを取得します。リクエストの確認は、**java.util.concurrent** ライブラリーによって提供される標準パターンで **スケジュールされたエグゼキューターサービス** である組み込みタイマークラスによって自動的にスケジュールされます。スケジュールされたエグゼキューターサービスは、特定のタスクを時間間隔で実行し、タスクインスタンスの実行に使用されるスレッドのプールも管理します。

図38.4「スケジュールされたポーリングコンシューマー」 は、スケジュールされたポーリングコンシューマーパターンの概要を示しています。

図38.4 スケジュールされたポーリングコンシューマー



スケジュールされたポーリングコンシューマーは、以下のようにリクエストを処理します。

1. スケジュールされたエグゼキューターサービスには、利用できるスレッドプールがあり、コンシューマーの処理を開始できます。スケジュール設定した時間間隔が経過すると、スケジュール済みエグゼキューターサービスはプールから空きスレッドの取得を試みます (デフォルトではプールには 5 つのスレッドがあります)。空きスレッドが利用可能な場合、そのスレッドを使用してコンシューマーで **poll()** メソッドを呼び出します。

2. コンシューマーの **poll()** メソッドは、受信したリクエストの処理をトリガーすることを目的としています。**poll()** メソッドの本文では、コンシューマーは受信メッセージの取得を試行します。リクエストがない場合、**poll()** メソッドを即座に返します。

3. 要求メッセージが利用可能な場合、コンシューマーはこれをエクスチェンジオブジェクトに挿入し、ルート内の次のプロセッサーで **process()** の呼び出しを行い、エクスチェンジオブジェクトを引数として渡します。

ポーリングパターン

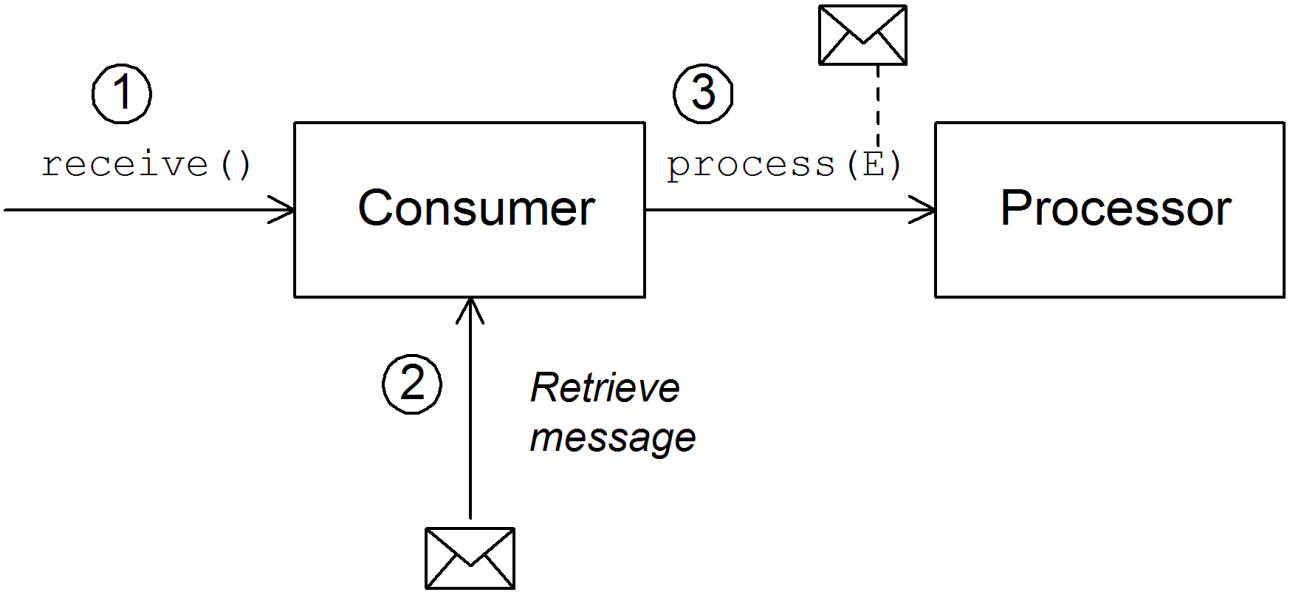
ポーリングパターンでは、サードパーティーがコンシューマーのポーリングメソッドの 1 つを呼び出すと、受信したリクエストの処理が開始されます。

- **receive()**
- **receiveNoWait()**
- **receive(long timeout)**

ポーリングメソッドの呼び出しを開始するための正確なメカニズムを定義することは、コンポーネントの実装の責務です。このメカニズムはポーリングパターンで指定されません。

図38.5「Polling Consumer」 は、ポーリングコンシューマーパターンの概要を示しています。

図38.5 Polling Consumer



ポーリングを行うコンシューマーは、以下のようにリクエストを処理します。

1. 受信リクエストの処理は、コンシューマーのポーリングメソッドの1つが呼び出されるたびに開始されます。これらのポーリングメソッドを呼び出すメカニズムは、コンポーネントの実装によって定義されます。

2. **receive()** メソッドのボディーでは、コンシューマーは受信したリクエストメッセージの取得を試みます。現在利用できるメッセージがない場合、動作は、呼び出された受信メソッドによって異なります。

   - **receiveNoWait()** が即座に返されます。
   - **receive(long timeout)** は、指定されたタイムアウト間隔で待機します。[2] 返す前
   - **receive()** は、メッセージが受信されるまで待機します。

3. 要求メッセージが利用可能な場合、コンシューマーはこれをエクスチェンジオブジェクトに挿入し、ルート内の次のプロセッサーで **process()** の呼び出しを行い、エクスチェンジオブジェクトを引数として渡します。

38.1.4. 非同期処理

概要

プロデューサーエンドポイントは通常、エクスチェンジの処理時に **同期** パターンに従います。上記のプロセッサーがパイプラインでプロデューサーの **process()** を呼び出すと、応答を受け取るまで **process()** メソッドはブロックされます。この場合、プロセッサーのスレッドは、リクエストの送信および応答の受信サイクルが完了するまでブロックされます。

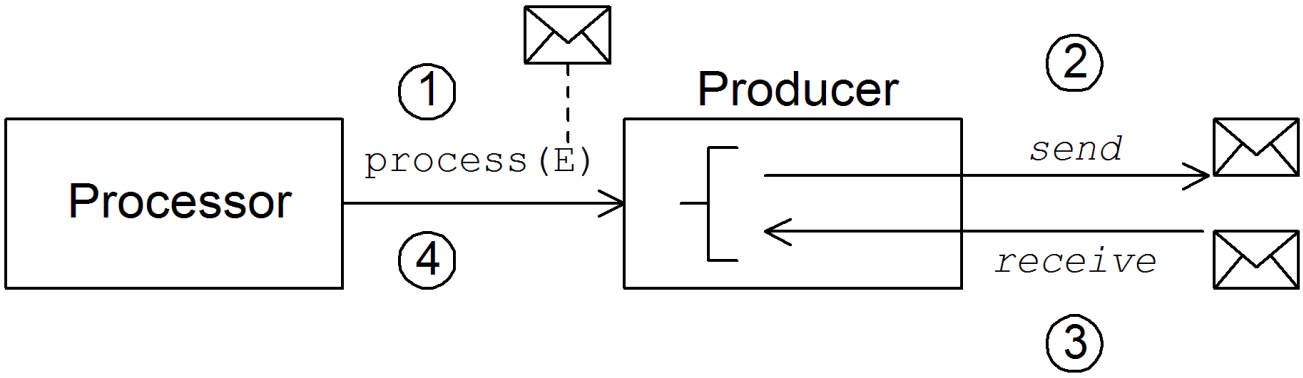
ただし、プロセッサーのスレッドがすぐにリリースされ、**process()** 呼び出しがブロックされ **ない** ようにするため、前のプロセッサーをプロデューサーから切り離す方が望ましい場合があります。この場合、asynchronous パターンを使用してプロデューサーを実装する必要があります。これにより、前のプロセッサーでは、非ブロッキングバージョンの **process()** メソッドを呼び出すオプションが提供されます。

異なる実装オプションの概要を示すために、本セクションでは、プロデューサーエンドポイントを実装する同期パターンと非同期パターンの両方を説明します。

同期プロデューサー

図38.6「同期プロデューサー」 は、プロデューサーがエクスチェンジの処理を完了するまで、前のプロセッサーがブロックする同期プロデューサーの概要を示しています。

図38.6 同期プロデューサー



同期プロデューサーは以下のようにエクスチェンジを処理します。

1. パイプラインの前のプロセッサーは、プロデューサー上の同期 **process()** メソッドを呼び出して、同期処理を開始します。同期 **process()** メソッドは単一のエクスチェンジ引数を取ります。

2. **process()** メソッドのボディー部で、プロデューサーはリクエスト (**In** メッセージ) をエンドポイントに送信します。

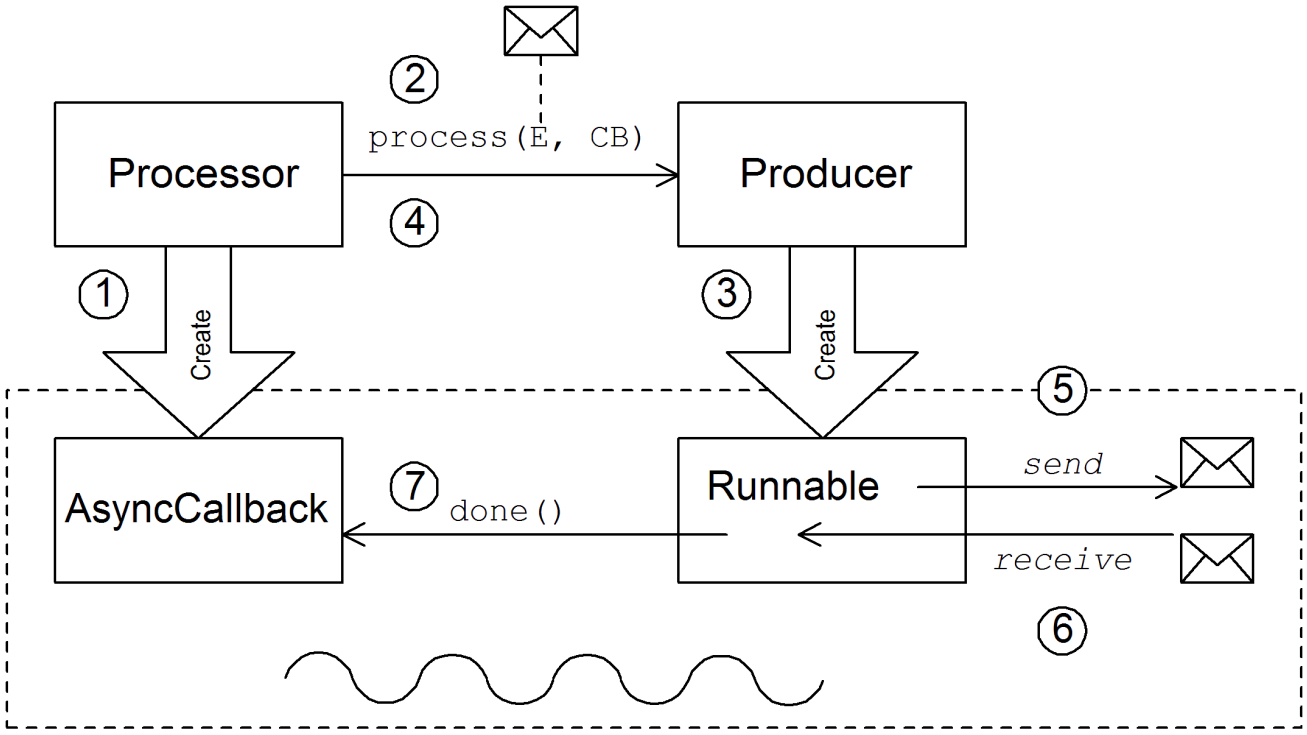
3. 交換パターンで必要な場合、プロデューサーは応答 (**Out** メッセージ) がエンドポイントから到達するまで待機します。このステップにより、**process()** メソッドが無限にブロックされる可能性があります。ただし、交換パターンが応答を強制しない場合、**process()** メソッドはリクエストの送信直後に返すことができます。

4. **process()** メソッドが返されると、エクスチェンジオブジェクトには同期呼び出しからの応答が含まれます (**Out** メッセージメッセージ)。

非同期プロデューサー

図38.7「非同期プロデューサー」 は、プロデューサーがサブスレッドでのエクスチェンジを処理し、前のプロセッサーは長時間ブロックされない非同期プロデューサーの概要を示しています。

図38.7 非同期プロデューサー



非同期プロデューサーは以下のようにエクスチェンジを処理します。

1. プロセッサーが非同期 **process()** メソッドを呼び出す前に、**非同期コールバック** オブジェクトを作成する必要があります。これはルートの戻り部分でエクスチェンジを処理するロールを果たします。非同期コールバックでは、プロセッサーは AsyncCallback インターフェイスから継承されるクラスを実装する必要があります。

2. プロセッサーは、プロデューサー上の非同期 **process()** メソッドを呼び出して非同期処理を開始します。非同期 **process()** メソッドは、2 つの引数を取ります。

   - エクスチェンジオブジェクト
   - 同期コールバックオブジェクト

3. **process()** メソッドのボディー部で、プロデューサーは処理コードをカプセル化する **Runnable** オブジェクトを作成します。その後、プロデューサーはこの **Runnable** オブジェクトの実行をサブスレッドに委任します。

4. 非同期 **process()** メソッドが返されるため、プロセッサーのスレッドが解放されますエクスチェンジの処理は個別のサブスレッドで続行されます。

5. **Runnable** オブジェクトは In メッセージをエンドポイントに送信します。

6. 交換パターンで必要な場合、**Runnable** オブジェクトはエンドポイントから応答 (**Out** または **Fault** メッセージ) が到達するのを待機します。**Runnable** オブジェクトは応答を受け取るまでブロックされます。

7. 応答が到達すると、**Runnable** オブジェクトは応答 (**Out** メッセージ) をエクスチェンジオブジェクトに挿入し、非同期コールバックオブジェクトの **done()** を呼び出します。次に、非同期コールバックはリプライメッセージを処理します (サブスレッドで実行されます)。

38.2. コンポーネントの実装方法

概要

このセクションでは、カスタム Apache Camel コンポーネントを実装するのに必要な手順の概要を説明します。

実装する必要のあるインターフェイス。

コンポーネントを実装する場合は通常、以下の Java インターフェイスを実装する必要があります。

- org.apache.camel.Component
- org.apache.camel.Endpoint
- org.apache.camel.Consumer
- org.apache.camel.Producer

さらに、以下の Java インターフェイスを実装する必要もあります。

- org.apache.camel.Exchange
- org.apache.camel.Message

実装手順

通常、以下のようにカスタムコンポーネントを実装します。

1. **Component インターフェイスを実装する**: コンポーネントオブジェクトはエンドポイントファクトリーとして機能します。**DefaultComponent** クラスを拡張し、**createEndpoint()** メソッドを実装します。
   39章*Component インターフェイス*を参照してください。

2. **Endpoint インターフェイスを実装する**: エンドポイントは、特定の URI で識別されるリソースを表します。エンドポイントを実装する際の方法は、コンシューマーが **イベント駆動型** のパターン、**スケジュールされたポーリング** パターン、または **ポーリング** パターンに従うかによって異なります。 イベント駆動型のパターンでは、**DefaultEndpoint** クラスを拡張し、以下のメソッドを実装してエンドポイントを実装します。

   - **createProducer()**
   - **createConsumer()**
     スケジュールされたポーリングパターンの場合、**ScheduledPollEndpoint** クラスを拡張し、以下のメソッドを実装してエンドポイントを実装します。
   - **createProducer()**
   - **createConsumer()**
     ポーリングパターンの場合、**DefaultPollingEndpoint** クラスを拡張し、以下のメソッドを実装してエンドポイントを実装します。
   - **createProducer()**
   - **createPollConsumer()**
     40章*Endpoint インターフェイス*を参照してください。

3. **Consumer インターフェイスを実装する**: 実装する必要があるパターン (イベント駆動、スケ

ジュールされたポーリング、ポーリング) に応じて、コンシューマーを実装する方法が複数あります。また、メッセージ交換の処理に使用されるスレッドモデルを決定する際には、コンシューマーの実装も重要です。
「Consumer インターフェイスの実装」を参照してください。

4. **Producer インターフェイスを実装**- Producer インターフェイス を実装するには、**DefaultProducer** クラスを拡張し、**process()** メソッドを実装します。
*42章Producer インターフェイス*を参照してください。

5. **任意で Exchange または Message インターフェイスを実装する**: エクスチェンジとメッセージのデフォルト実装を直接使用できますが、場合によってはこれらのタイプをカスタマイズする必要がある場合があります。
43章Exchange インターフェイス および *44章Message インターフェイス* を参照してください。

コンポーネントのインストールおよび設定

カスタムコンポーネントは、以下のいずれかの方法でインストールできます。

- **コンポーネントを CamelContext に直接追加する**: この **CamelContext.addComponent()** メソッドは、プログラム的にコンポーネントを追加します。
- **Spring 設定を使用してコンポーネントを追加する**: 標準の Spring **bean** 要素はコンポーネントインスタンスを作成します。Bean の **id** 属性は、コンポーネント接頭辞を暗黙的に定義します。詳細は、「コンポーネントの設定」 を参照してください。
- **Apache Camel をコンポーネント自動検出するように設定する**: Apache Camel が必要に応じてコンポーネントを自動的に読み込みます。詳細は、「自動検出の設定」 を参照してください。

38.3. 自動検出と設定

38.3.1. 自動検出の設定

概要

自動検出は、コンポーネントを Apache Camel アプリケーションに動的に追加できるようにするメカニズムです。コンポーネントの URI 接頭辞は、必要に応じてコンポーネントをロードするキーとして使用されます。たとえば、Apache Camel がエンドポイント URI **activemq://MyQName** に到達し、ActiveMQ エンドポイントがロードされていない場合、Apache Camel は **activemq** 接頭辞によって識別されるコンポーネントを検索し、コンポーネントを動的にロードします。

コンポーネントクラスの可用性

自動検出を設定する前に、カスタムコンポーネントクラスが現在のクラスパスからアクセスできることを確認する必要があります。通常、カスタムコンポーネントクラスを JAR ファイルにバンドルし、JAR ファイルをクラスパスに追加します。

自動検出の設定

コンポーネントの自動検出を有効にするには、コンポーネント接頭辞 **component-prefix** の名前の Java プロパティーファイルを作成し、そのファイルを以下の場所に保存します。

```
/META-INF/services/org/apache/camel/component/component-prefix
```

component-prefix プロパティーファイルには、以下のプロパティー設定が含まれている必要があります。

```
class=component-class-name
```

ここで、**component-class-name** はカスタムコンポーネントクラスの完全修飾名になります。このファイルに追加のシステムプロパティー設定を定義することもできます。

例

たとえば、以下の Java プロパティーファイルを作成して、Apache Camel FTP コンポーネントの自動検出を有効にできます。

```
/META-INF/services/org/apache/camel/component/ftp
```

以下の Java プロパティー設定が含まれます。

```
class=org.apache.camel.component.file.remote.RemoteFileComponent
```

注記

FTP コンポーネントの Java プロパティーファイルは、JAR ファイル **camel-ftp-Version.jar** にすでに定義されています。

38.3.2. コンポーネントの設定

概要

コンポーネントを追加する場合は、Apache Camel Spring 設定ファイル (**META-INF/spring/camel-context.xml**) でコンポーネントを設定します。コンポーネントを見つけるために、コンポーネントの URI 接頭辞は Spring 設定の **bean** 要素の ID 属性と照合されます。コンポーネント接頭辞が bean 要素 ID と一致する場合、Apache Camel は参照されたクラスをインスタンス化し、Spring 設定に指定されたプロパティーを注入します。

注記

このメカニズムは自動検出よりも優先されます。CamelContext が必須 ID で Spring Bean を見つけた場合は、自動検出を使用したコンポーネントの検索は行われません。

コンポーネントクラスで Bean プロパティーを定義します。

コンポーネントクラスに注入するプロパティーがある場合は、これを Bean プロパティーとして定義します。以下に例を示します。

```
public class CustomComponent extends
  DefaultComponent<CustomExchange> {
    ...
    PropType getProperty() { ... }
    void setProperty(PropType v) { ... }
}
```

getProperty() メソッドと **setProperty()** メソッドは、**プロパティー** の値にアクセスします。

Spring のコンポーネントの設定

Spring でコンポーネントを設定するには、例38.1「Spring でのコンポーネントの設定」 にあるように設定ファイル **META-INF/spring/camel-context.xml** を編集します。

例38.1 Spring でのコンポーネントの設定

```
<?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>
```

component-prefix の ID が付いた **bean** 要素は **component-class-name** コンポーネントを設定します。**property** 要素を使用して、プロパティーをコンポーネントインスタンスに注入することができます。たとえば、前述の例の **property** 要素は、コンポーネントで **setProperty()** を呼び出して、**propertyValue** の値を **property** プロパティーに注入します。

例

例38.2「Spring JMS コンポーネントの設定」 は、ID が **jms** と等しい bean 要素を定義して Apache Camel の JMS コンポーネントを設定する方法の例を示しています。これらの設定は Spring 設定ファイル (**camel-context.xml**) に追加されます。

例38.2 Spring JMS コンポーネントの設定

```
<?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** は、指定された Java パッケージ (**org.apache.camel.example.spring**) にある **RouteBuilder** クラスを自動的にインスタンス化します。

2 ID **jms** の付いた bean 要素は、JMS コンポーネントを設定します。Bean ID は、コンポーネントの URI 接頭辞に対応します。たとえば、ルートが URI でエンドポイント (**jms://MyQName**) を指定する場合、Apache Camel は **jms** bean 要素の設定を使用して JMS コンポーネントを自動的に読み込みます。

3 JMS は、メッセージングサービスのラッパーです。**JmsComponent** クラスに **connectionFactory** プロパティーを設定して、メッセージングシステムの具象実装を指定する必要があります。

4 この例では、JMS メッセージングサービスの具象実装は Apache ActiveMQ です。**brokerURL** プロパティーは、メッセージブローカーがローカルの Java 仮想マシン (JVM) に組み込まれている ActiveMQ ブローカーインスタンスへの接続を初期化します。ブローカーが JVM にない場合、ActiveMQ はオプション **broker.persistent=false** (ブローカーはメッセージを永続化しない) および **broker.useJmx=false** (ブローカーは JMX ポートを開かない) でインスタンス化します。

---

[2] 通常、タイムアウトの間隔はミリ秒単位で指定します。

第39章 COMPONENT インターフェイス

概要

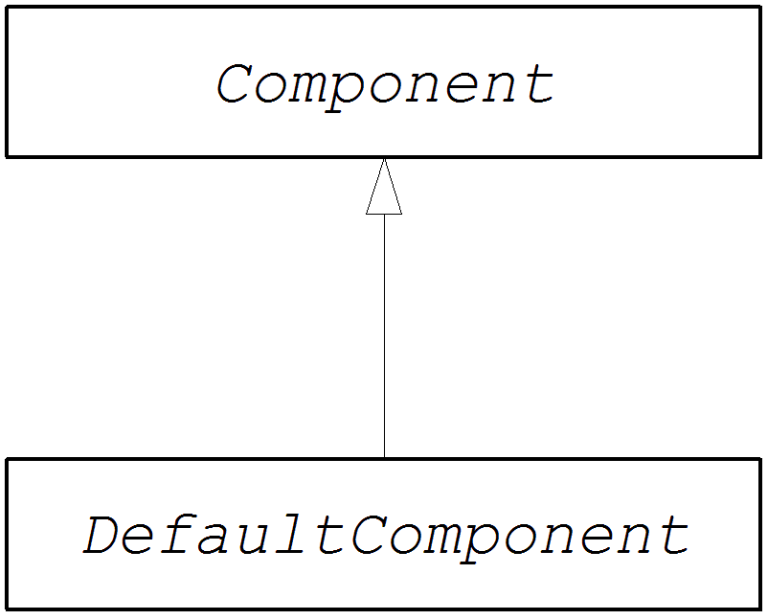
本章では、Component インターフェイスの実装方法を説明します。

39.1. COMPONENT インターフェイス

概要

Apache Camel コンポーネントを実装するには、org.apache.camel.Component インターフェイスを実装する必要があります。**Component** タイプのインスタンスは、カスタムコンポーネントへのエントリーポイントを提供します。つまり、コンポーネントの他のオブジェクトはすべて、**Component** インスタンスから最終的にアクセスできます。図39.1「コンポーネントの継承階層」 は、**Component** 継承階層を設定する、関連のある Java インターフェイスとクラスを示しています。

図39.1 コンポーネントの継承階層



Component インターフェイス

例39.1「Component インターフェイス」 は、org.apache.camel.Exchange インターフェースの定義を示しています。

例39.1 Component インターフェイス

```
package org.apache.camel;

public interface Component {
    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    Endpoint createEndpoint(String uri) throws Exception;
}
```

コンポーネントメソッド

Component インターフェイスは以下のメソッドを定義します。

- **getCamelContext()** および **setCamelContext()** - このコンポーネントが属する **CamelContext** を参照します。**setCamelContext()** メソッドは、**CamelContext** にコンポーネントを追加すると自動的に呼び出されます。
- **createEndpoint()**: このコンポーネントの **Endpoint** インスタンスを作成するために呼び出されるファクトリーメソッド。**uri** パラメーターはエンドポイントの作成に必要な詳細が含まれるエンドポイント URI です。

39.2. COMPONENT インターフェイスの実装

DefaultComponent クラス

org.apache.camel.impl.DefaultComponent クラスを拡張して新しいコンポーネントを実装します。これにより、一部のメソッドに標準機能とデフォルトの実装が提供されます。特に、**DefaultComponent** クラスは URI 解析と **スケジュール済みエグゼキューター** の作成をサポートします (スケジュールされたポーリングパターンに使用されます)。

URI の解析

ベースコンポーネントインターフェイスで定義される **createEndpoint(String uri)** メソッドは、完全かつ解析されていないエンドポイント URI を唯一の引数として取ります。一方、**DefaultComponent** クラスは、以下の署名で **createEndpoint()** メソッドの 3 つの引数バージョンを定義します。

```
protected abstract Endpoint createEndpoint(
    String uri,
    String remaining,
    Map parameters
)
throws Exception;
```

uri は、元の解析されていない URI です。 **remaining** は、開始時にコンポーネント接頭辞を削除し、最後のクエリーオプションを削除した後に残る URI の一部で、**parameters** は解析されたクエリーオプションが含まれます。**createEndpoint()** から継承時に上書きする必要があるのは、**DefaultComponent** メソッドのこのバージョンです。これには、エンドポイント URI がすでに解析されているという利点があります。

次の **file** コンポーネントのサンプルエンドポイント URI は、URI 解析の仕組みを示しています。

```
file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
```

この URI では、引数が 3 つのバージョンの **createEndpoint()** に以下の引数が渡されます。

引数	サンプル値
uri	file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
remaining	**/tmp/messages/foo**

引数	サンプル値
parameters	2 つのエントリーが **java.util.Map** に設定されています。 ● パラメーター **delete** はブール値 **true** です。 ● パラメーター **moveNamePostfix** には、文字列値 **.old** があります。

パラメーターの注入

デフォルトでは、URI クエリーオプションから抽出されたパラメーターはエンドポイントの Bean プロパティーに注入されます。**DefaultComponent** クラスは、ユーザーのパラメーターを自動的に注入します。

たとえば、2 つの URI クエリーオプション (**delete** および **moveNamePostfix**) をサポートするカスタムエンドポイントを定義する場合などです。エンドポイントクラスで対応する Bean メソッド (getter と setter) を定義するだけです。

```
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;
    }
}
```

URI クエリーオプションを **コンシューマーパラメーター** に注入することもできます。詳細は、「コンシューマーパラメーターの注入」 を参照してください。

エンドポイントパラメーター注入の無効化

Endpoint クラスにパラメーターが定義されていない場合は、エンドポイントパラメーターの注入を無効にすることで、エンドポイント作成のプロセスを最適化できます。エンドポイントでパラメーターの注入を無効にするには、以下のように **useIntrospectionOnEndpoint()** メソッドを上書きし、**false** を返すように実装します。

```
protected boolean useIntrospectionOnEndpoint() {
  return false;
}
```

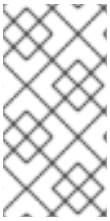
注記

useIntrospectionOnEndpoint() メソッドは、**Consumer** クラスで実行される可能性のあるパラメーターの注入には影響**しません**。このレベルのパラメーターの注入は **Endpoint.configureProperties()** メソッドによって制御されます (「エンドポイントインターフェイスの実装」 を参照)。

スケジュール済みエグゼキューターサービス

スケジュールされたエグゼキューターは、スケジュールされたポーリングパターンで使用されます。ここでは、コンシューマーエンドポイントの定期的なポーリングを行います (スケジュール済みエグゼキューターは、実質的にスレッドプールの実装です)。

スケジュールされたエグゼキューターサービスをインスタンス化するには、**CamelContext.getExecutorServiceStrategy()** メソッドによって返される **ExecutorServiceStrategy** オブジェクトを使用します。Apache Camel スレッドモデルの詳細は、「スレッドモデル」 を参照してください。



注記

Apache Camel 2.3 以前、**DefaultComponent** クラスはスレッドプールインスタンスを作成するための **getExecutorService()** メソッドを提供していました。ただし、2.3 以降、スレッドプールの作成は **ExecutorServiceStrategy** オブジェクトによって集中管理されるようになりました。

URI の検証

エンドポイントインスタンスを作成する前に URI を検証する場合は、以下の署名を持つ **DefaultComponent** クラスから **validateURI()** メソッドを上書きすることができます。

```
protected void validateURI(String uri,
                           String path,
                           Map parameters)
  throws ResolveEndpointFailedException;
```

提供された URI に必要な形式がない場合は、**validateURI()** の実装によって **org.apache.camel.ResolveEndpointFailedException** 例外が発生するはずです。

エンドポイントの作成

例39.2「**createEndpoint()** の実装」 では、オンデマンドでエンドポイントインスタンスを作成する **DefaultComponent.createEndpoint()** メソッドの実装方法を概説します。

例39.2 createEndpoint() の実装

```
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 は、**DefaultComponent** クラスを拡張することで定義されるカスタムコンポーネントクラスの名前です。
2. **DefaultComponent** を拡張する場合は、3 つの引数 (「URI の解析」 を参照) で **createEndpoint()** メソッドを実装する必要があります。
3. コンストラクターを呼び出して、カスタムエンドポイントタイプの **CustomEndpoint** のインスタンスを作成します。少なくとも、このコンストラクターは、元の URI 文字列 **uri** のコピーと、このコンポーネントインスタンスへの参照 **this** を取得します。

例

例39.3「FileComponent 実装」 は、**FileComponent** クラスの実装例を示しています。

例39.3 FileComponent 実装

```
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. クラスの自動インスタンス化を容易にするために、コンポーネントクラスの引数なしコンストラクターを常に定義します。
2. プログラミングでコンポーネントを作成する際に、親の **CamelContext** インスタンスを引数として取るコンストラクターが便利です。
3. **FileComponent.createEndpoint()** メソッドの実装は、例39.2「**createEndpoint()** の実装」 に記載のパターンに従います。実装により **FileEndpoint** オブジェクトが作成されます。

SynchronizationRouteAware インターフェイス

SynchronizationRouteAware インターフェイスを使用すると、エクスチェンジがルーティングされる前および後にコールバックを指定できます。

- **onBeforeRoute**: エクスチェンジが指定のルートによってルーティングされる前に呼び出されます。ただし、ルートの起動後に **SynchronizationRouteAware** 実装を **UnitOfWork** に追加した場合、このコールバックは呼び出されないことがあります。

- **onAfterRoute**: エクスチェンジが指定のルートによってルーティングされた後に呼び出されます。ただし、エクスチェンジが複数のルートでルーティングされる場合は、ルートごとにコールバックを生成します。
  この呼び出しは、以下のコールバックの前に行われます。

  a. ルートのコンシューマーは、すべての応答を呼び出し元に書き込みます (**InOut** モードの場合)。

  b. **UnitOfWork** は、**Synchronization.onComplete(org.apache.camel.Exchange)** または **Synchronization.onFailure(org.apache.camel.Exchange)** を呼び出すことで行われます。

第40章 ENDPOINT インターフェイス

概要

本章では、Apache Camel コンポーネントの実装における必須ステップである Endpoint インターフェイスを実装する方法を説明します。

40.1. ENDPOINT インターフェイス

概要

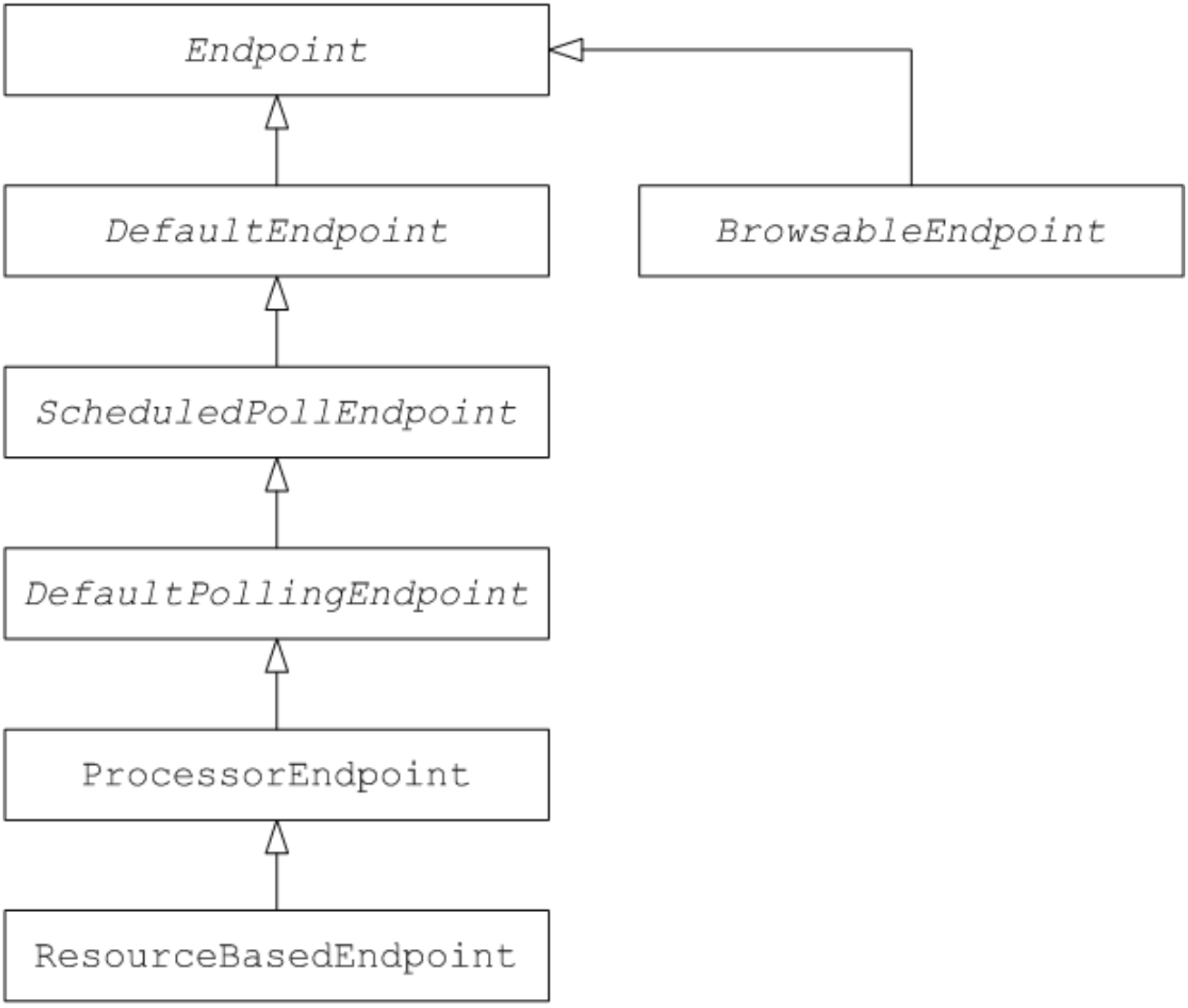
org.apache.camel.Endpoint タイプのインスタンスはエンドポイント URI をカプセル化します。また、**Consumer**、**Producer**、**Exchange** オブジェクトのファクトリーとしても機能します。エンドポイントを実装するには、以下の 3 つの方法があります。

- イベント駆動型
- スケジュールされたポーリング
- ポーリング

これらのエンドポイント実装パターンは、コンシューマーの実装に対応するパターンを補完します。「Consumer インターフェイスの実装」 を参照してください。

図40.1「コンシューマー継承階層」 は、**Endpoint** 継承階層を設定する、関連のある Java インターフェイスとクラスを示しています。

図40.1 コンシューマー継承階層



Endpoint インターフェイス

例40.1「Endpoint インターフェイス」 は、org.apache.camel.Exchange インターフェースの定義を示しています。

例40.1 Endpoint インターフェイス

```
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;
}
```

エンドポイントメソッド

Endpoint インターフェイスは以下のメソッドを定義します。

- **isSingleton()**: 各 URI が CamelContext 内の単一のエンドポイントにマップされるように保証する場合には、**true** を返します。このプロパティーが **true** の場合、ルート内の同一 URI への複数の参照は常に **単一の** エンドポイントインスタンスを参照します。一方、このプロパティーが **false** の場合、ルート内の同じ URI への複数の参照は **個別の** エンドポイントインスタンスを参照します。ルートで URI を参照するたびに、新しいエンドポイントインスタンスが作成されます。

- **getEndpointUri()** : このエンドポイントのエンドポイント URI を返します。

- **getEndpointKey()** – エンドポイントを登録する際に、**org.apache.camel.spi.LifecycleStrategy** によって使用されます。

- **getCamelContext()**: このエンドポイントが属する **CamelContext** インスタンスへの参照を返します。

- **setCamelContext()**: このエンドポイントが属する **CamelContext** インスタンスへの参照を返します。

- **configureProperties()** : 新規に **Consumer** インスタンスを作成する時にパラメーターの挿入に使用されるパラメーターマップのコピーを保存します。

- **isLenientProperties()**: **true** の場合は URI が不明なパラメーターを含むことができることを示します (つまり、エンドポイントまたは **Consumer** クラスに注入できないパラメーターです)。通常、このメソッドは **false** を返すために実装する必要があります。

- **createExchange()**: 以下のバリアントを持つオーバーロードされたメソッド。

  - **Exchange createExchange()**: デフォルトの交換パターン設定を使用して、新規のエクスチェンジインスタンスを作成します。

  - **Exchange createExchange(ExchangePattern pattern)** – 指定されたエクスチェンジパターンで新しいエクスチェンジインスタンスを作成します。

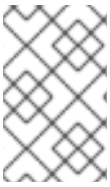
  - **Exchange createExchange(Exchange exchange)** – 指定された **exchange** 引数を、このエンドポイントに必要なエクスチェンジのタイプに変換します。指定のエクスチェンジがまだ正しいタイプでない場合、このメソッドはこれを正しいタイプの新規インスタンスにコピーします。このメソッドのデフォルト実装は **DefaultEndpoint** クラスにあります。

- **createProducer()**: 新しい **Producer** インスタンスを作成するために使用されるファクトリーメソッド。

- **createConsumer()**: イベント駆動型のコンシューマーインスタンスを新たに作成するためのファクトリーメソッド。**processor** 引数は、ルートの最初のプロセッサーへの参照です。
- **createPollingConsumer()**: 新しいポーリングコンシューマーインスタンスを作成するためのファクトリーメソッド。

エンドポイントシングルトン

不要なオーバーヘッドを避けるために、同じ URI (CamelContext 内) を持つすべてのエンドポイントに**単一** のエンドポイントインスタンスを作成することが推奨されます。**isSingleton()** を実装して **true** を返すことにより、この条件を強制することができます。



注記

このコンテキストでは、**同じ URI** は文字列の等価を使用して 2 つの URI が同じであることを意味します。原則では、異なる文字列で表されますが、同等の URI を 2 つ持つことができます。この場合、URI は同じものとして処理されません。

40.2. エンドポイントインターフェイスの実装

エンドポイントを実装する代替方法

以下の代替エンドポイント実装パターンがサポートされます。

- イベント駆動型のエンドポイント実装
- スケジュールされたポーリングエンドポイントの実装
- ポーリングエンドポイントの実装

イベント駆動型のエンドポイント実装

カスタムエンドポイントがイベント駆動型のパターン (「コンシューマーパターンおよびスレッド」 を参照) に準拠する場合、例40.2「DefaultEndpoint の実装」 に示されているように **org.apache.camel.impl.DefaultEndpoint** 抽象クラスを拡張することで実装されます。

例40.2 DefaultEndpoint の実装

```
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. **DefaultEndpoint** クラスを拡張して、イベント駆動型 CustomEndpoint カスタムエンドポイントを実装します。
2. エンドポイント URI の **endpointUri** および親コンポーネントの参照 **component** を引数として取るコンストラクターが少なくとも 1 つ必要です。
3. **createProducer()** ファクトリーメソッドを実装し、プロデューサーエンドポイントを作成します。
4. **createConsumer()**: イベント駆動型のコンシューマーインスタンスを作成するためのファクトリーメソッド。
5. 通常、**createExchange()** メソッドを上書きする必要は **ありません**。**DefaultEndpoint** から継承された実装は、デフォルトで **DefaultExchange** オブジェクトを作成します。これは任意の Apache Camel コンポーネントで使用できます。ただし、**DefaultExchange** オブジェクトで一部のエクスチェンジプロパティーを初期化する必要がある場合は、エクスチェンジプロパティー設定を追加するために、この **createExchange()** メソッドを上書きすることが推奨されます。

重要

createPollingConsumer() メソッドをオーバーライド **しないでください**。

DefaultEndpoint クラスは、以下のメソッドのデフォルト実装を提供します。これは、カスタムエンドポイントコードを書き込む際に役立つ場合があります。

- **getEndpointUri()**: エンドポイント URI を返します。
- **getCamelContext()**: **CamelContext** への参照を返します。
- **getComponent()**: 親コンポーネントへの参照を返します。
- **createPollingConsumer()**: ポーリングコンシューマーを作成します。作成されたポーリングコンシューマーの機能は、イベント駆動型のコンシューマーに基づいています。**createConsumer()** でイベント駆動型のコンシューマーメソッドを上書きする場合は、ポーリングコンシューマー実装を取得できます。
- **createExchange(Exchange e)**: 指定のエクスチェンジオブジェクト **e** を、このエンドポイントに必要な型に変換します。このメソッドは、上書きされた **createExchange()** エンドポイントを使用して、新しいエンドポイントを作成します。これにより、メソッドがカスタムのエクスチェンジタイプでも機能するようになります。

スケジュールされたポーリングエンドポイントの実装

カスタムエンドポイントがスケジュールされたポーリングパターン (「コンシューマーパターンおよびスレッド」 を参照) に準拠する場合、例40.3「ScheduledPollEndpoint 実装」 に示されている **org.apache.camel.impl.ScheduledPollEndpoint** の抽象クラスから継承して実装されます。

例40.3 ScheduledPollEndpoint 実装

```
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 **ScheduledPollEndpoint** クラスを拡張して、スケジュールされたポーリングのカスタムエンドポイント **CustomEndpoint** を実装します。

2 エンドポイント URI の **endpointUri** および親コンポーネントの参照 **component** を引数として取るコンストラクターが少なくとも 1 つ必要です。

3 **createProducer()** ファクトリーメソッドを実装し、プロデューサーエンドポイントを作成します。

4 スケジュールされたポーリングコンシューマーインスタンスを作成するには、**createConsumer()** ファクトリーメソッドを実装します。

5 **ScheduledPollEndpoint** ベースクラスで定義される **configureConsumer()** メソッドは、コンシューマークエリーオプションをコンシューマーに注入します。「コンシューマーパラメーターの注入」を参照してください。

6 通常、**createExchange()** メソッドを上書きする必要は **ありません**。**DefaultEndpoint** から継承された実装は、デフォルトで **DefaultExchange** オブジェクトを作成します。これは任意の Apache Camel コンポーネントで使用できます。ただし、**DefaultExchange** オブジェクトで一部のエクスチェンジプロパティーを初期化する必要がある場合は、エクスチェンジプロパティー設定を追加するために、この **createExchange()** メソッドを上書きすることが推奨されます。

重要

createPollingConsumer() メソッドをオーバーライド **しないでください**。

ポーリングエンドポイントの実装

カスタムエンドポイントがポーリングコンシューマーパターン (「コンシューマーパターンおよびスレッド」 を参照) に準拠する場合、例40.4「DefaultPollingEndpoint 実装」 に示されている **org.apache.camel.impl.DefaultPollingEndpoint** の抽象クラスから継承して実装されます。

例40.4 DefaultPollingEndpoint 実装

```
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.
    ...
}
```

この **CustomEndpoint** クラスはポーリングエンドポイントであるため、**createConsumer()** メソッドの代わりに **createPollingConsumer()** メソッドを実装する必要があります。**createPollingConsumer()** から返されるコンシューマーインスタンスは、PolingConsumer インターフェイスから継承する必要があります。ポーリングコンシューマーの実装方法は、「ポーリングコンシューマーの実装」 を参照してください。

createPollingConsumer() メソッドの実装以外に、**DefaultPollingEndpoint** を実装する手順は、**ScheduledPollEndpoint** を実装する手順と似ています。詳細は、例40.3「ScheduledPollEndpoint 実装」 を参照してください。

BrowsableEndpoint インターフェイスの実装

現在のエンドポイントで保留中のエクスチェンジインスタンスの一覧を公開する場合は、例40.5「BrowsableEndpoint インターフェイス」 に示されているように org.apache.camel.spi.BrowsableEndpoint インターフェースを実装することができます。エンドポイントが受信イベントのバッファー処理を実行する場合は、このインターフェイスを実装することが理にかなっています。たとえば、Apache Camel SEDA エンドポイントは BrowsableEndpoint インターフェースを実装します。例40.6「SedaEndpoint 実装」 を参照してください。

例40.5 BrowsableEndpoint インターフェイス

```
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();
}
```

例

例40.6「SedaEndpoint 実装」 は、**SedaEndpoint** の実装例を示しています。SEDA エンドポイントは、**イベント駆動型のエンドポイント** の例です。受信イベントは FIFO キュー (**java.util.concurrent.BlockingQueue** のインスタンス) に格納され、SEDA コンシューマーは、イベントの読み取りおよび処理のためにスレッドを起動します。イベント自体は **org.apache.camel.Exchange** オブジェクトによって表されます。

例40.6 SedaEndpoint 実装

```
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** クラスは、**DefaultEndpoint** クラスを拡張してイベント駆動型のエンドポイントを実装するパターンに従います。この **SedaEndpoint** クラスは、キュー内のエクスチェンジオブ
2. **SedaEndpoint** はイベント駆動型のコンシューマーの通常のパターンに従い、エンドポイント引数 **endpointUri** およびコンポーネント参照引数 **component** を使用するコンストラクターを定義します。
3. もう 1 つのコンストラクターが提供され、キューの作成を親コンポーネントインスタンスに委譲します。
4. ファクトリーメソッド **createProducer()** は、イベントをキューに追加するプロデューサーの実装である **CollectionProducer** のインスタンスを作成します。
5. ファクトリーメソッド **createConsumer()** は、**SedaConsumer** のインスタンスを作成します。これはキューからイベントをプルし、それらを処理します。
6. **getQueue()** メソッドはキューへの参照を返します。
7. **isSingleton()** メソッドは **true** を返します。これは、一意の URI 文字列ごとに単一のエンドポイントインスタンスを作成する必要があることを示します。
8. この **getExchanges()** メソッドは、対応する BrowsableEndpoint からの抽象メソッドを実装します。

第41章 CONSUMER インターフェイス

概要

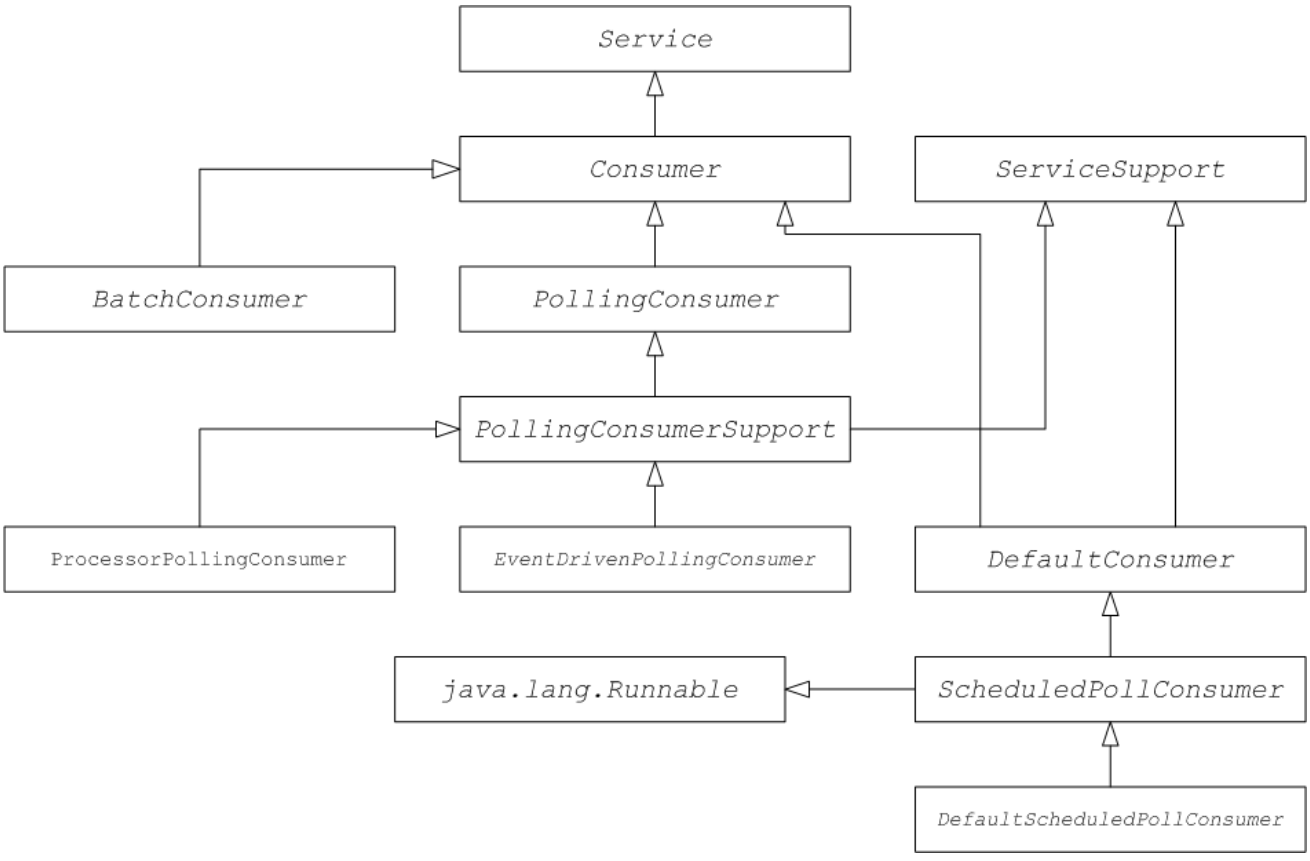
本章では、Apache Camel コンポーネントの実装における必須ステップである Consumer インターフェイスを実装する方法を説明します。

41.1. CONSUMER インターフェイス

概要

org.apache.camel.Consumer タイプのインスタンスは、ルートのソースエンドポイントを表します。コンシューマーの実装方法がいくつかあります (「コンシューマーパターンおよびスレッド」 を参照)。このレベルの柔軟性は継承階層 (図41.1「コンシューマー継承階層」 を参照) に反映されます。これには、コンシューマーを実装するための複数の異なるベースクラスが含まれます。

図41.1 コンシューマー継承階層



コンシューマーパラメーターの注入

Apache Camel は、スケジュールされたポーリングパターン (「スケジュールされたポーリングパターン」 を参照) を使用しているコンシューマーの場合、パラメーターをコンシューマーインスタンスに注入するサポートを提供します。たとえば、**custom** 接頭辞で識別されるコンポーネントの以下のエンドポイント URI について考えてみましょう。

```
custom:destination?consumer.myConsumerParam
```

Apache Camel は、フォームのクエリーオプション **consumer.*** を自動的に注入するためのサポートを提供します。**consumer.myConsumerParam** パラメーターには、以下のように Consumer 実装クラスで対応する setter メソッドと getter メソッドを定義する必要があります。

```
public class CustomConsumer extends ScheduledPollConsumer {
    ...
    String getMyConsumerParam() { ... }
    void setMyConsumerParam(String s) { ... }
    ...
}
```

getter および setter メソッドは、通常の Java Bean の命名規則に従います (プロパティー名の最初の文字を大文字にすることも含む)。

Consumer 実装で Bean メソッドを定義する他に、必ず **Endpoint.createConsumer()** の実装で **configureConsumer()** メソッドを呼び出す必要があります (「スケジュールされたポーリングエンドポイントの実装」 を参照してください)。

例41.1「FileEndpoint createConsumer () 実装」 は、ファイルコンポーネントの **FileEndpoint** クラスから取得した **createConsumer()** メソッド実装の例を示しています。

例41.1 FileEndpoint createConsumer () 実装

```
...
public class FileEndpoint extends ScheduledPollEndpoint {
    ...
    public Consumer createConsumer(Processor processor) throws Exception {
        Consumer result = new FileConsumer(this, processor);
        configureConsumer(result);
        return result;
    }
    ...
    }
```

ランタイム時に、コンシューマーパラメーターの注入は以下のようになります。

1. エンドポイントが作成されると、**DefaultComponent.createEndpoint(String uri)** のデフォルトの実装は URI を解析してコンシューマーパラメーターを抽出し、**ScheduledPollEndpoint.configureProperties()** を呼び出してエンドポイントインスタンスに保存します。

2. **createConsumer()** が呼び出されると、メソッド実装は **configureConsumer()** を呼び出し、コンシューマーパラメーターを注入します (例41.1「FileEndpoint createConsumer () 実装」 を参照)。

3. **configureConsumer()** メソッドは Java のリフレクションを使用して、**consumer.** 接頭辞を削除にした後に関連するオプションと一致する名前を持つ setter メソッドを呼び出します。

スケジュールされたポーリングパラメーター

スケジュールされたポーリングパターンに続くコンシューマーは、表41.1「スケジュールされたポーリングパラメーター」 に記載されているコンシューマーパラメーターを自動的にサポートします (エンドポイント URI のクエリーオプションとして表示されます)。

表41.1 スケジュールされたポーリングパラメーター

名前	デフォルト	説明
initialDelay	**1000**	最初のポーリングの前の遅延 (ミリ秒単位)。
delay	**500**	**useFixedDelay** フラグの値によって異なります (時間単位はミリ秒)。
useFixedDelay	**false**	**false** の場合は **delay** パラメーターはポーリング期間として解釈されます。ポーリングは **initialDelay**、**initialDelay+delay**、**initialDelay+2*delay** などで行われます。 **true** の場合は、**delay** パラメーターは前の実行から次の実行までに経過した時間として解釈されます。ポーリングは **initialDelay**、**initialDelay+[ProcessingTime]+delay** などで行われます。ProcessingTime は、現在のスレッドでエクスチェンジオブジェクトを処理するのにかかった時間です。

イベント駆動型のコンシューマーとポーリングコンシューマー間の変換

Apache Camel は、イベント駆動型のコンシューマーとポーリングコンシューマー間の変換に使用できる 2 つの特殊なコンシューマー実装を提供します。以下の変換クラスが提供されます。

- **org.apache.camel.impl.EventDrivenPollingConsumer**: イベント駆動型コンシューマーをポーリングコンシューマーインスタンスに変換します。
- **org.apache.camel.impl.DefaultScheduledPollConsumer**: ポーリングコンシューマーをイベント駆動型コンシューマーインスタンスに変換します。

実際には、これらのクラスを使用して Endpoint タイプの実装タスクを単純化します。Endpoint インターフェイスは、コンシューマーインスタンスを作成するための以下の 2 つのメソッドを定義します。

```
package org.apache.camel;

public interface Endpoint {
    ...
    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

createConsumer() はイベント駆動型のコンシューマーを返し、**createPollingConsumer()** はポーリングコンシューマーを返します。これらのメソッドは 1 つのみ実装します。たとえば、コンシューマーの

イベント駆動パターンに従っている場合は、単に例外を発生させる **createConsumer()** のメソッド実装を提供するために **createPollingConsumer()** メソッドを実装します。ただし、変換クラスを利用して Apache Camel はより有用なデフォルト実装を提供できます。

たとえば、イベント駆動のパターンに従ってコンシューマーを実装する場合は、**DefaultEndpoint** を拡張し、**createConsumer()** メソッドを実装してエンドポイントを実装します。**createPollingConsumer()** の実装は、以下のように定義される **DefaultEndpoint** から継承されます。

```
public PollingConsumer<E> createPollingConsumer() throws Exception {
    return new EventDrivenPollingConsumer<E>(this);
}
```

EventDrivenPollingConsumer コンストラクターはイベント駆動のコンシューマーへの参照 **this** を取得し、効果的にラップし、ポーリングコンシューマーに変換します。変換を実装するには、**EventDrivenPollingConsumer** インスタンスは受信イベントをバッファーし、**receive()**、**receive(long timeout)**、および **receiveNoWait()** メソッドを介してオンデマンドで利用できるようにします。

同様に、ポーリングパターンに従ってコンシューマーを実装する場合は、**DefaultPollingEndpoint** を拡張し、**createPollingConsumer()** メソッドを実装してエンドポイントを実装します。この場合、**createConsumer()** メソッドの実装は **DefaultPollingEndpoint** から継承され、デフォルトの実装は **DefaultScheduledPollConsumer** インスタンス (ポーリングコンシューマーをイベント駆動型のコンシューマーに変換) を返します。

ShutdownPrepared インターフェイス

コンシューマークラスはオプションで **org.apache.camel.spi.ShutdownPrepared** インターフェイスを実装できます。これにより、カスタムコンシューマーエンドポイントがシャットダウン通知を受け取ることができます。

例41.2「ShutdownPrepared インターフェイス」 は、**ShutdownPrepared** インターフェイスの定義を示しています。

例41.2 ShutdownPrepared インターフェイス

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

public interface ShutdownPrepared {

    void prepareShutdown(boolean forced);

}
```

ShutdownPrepared インターフェイスは以下のメソッドを定義します。

prepareShutdown

以下のように、1 または 2 フェーズでコンシューマーエンドポイントをシャットダウンするための通知を受信します。

a. **正常なシャットダウン**: **forced** 引数に **false** の値がある場合。リソースを正常にクリーンアップしようとします。たとえば、スレッドを正常に停止することによりクリーンアップします。

b. Forced shutdown- **forced** 引数には、値 **true** があります。これは、シャットダウンがタイムアウトしたことを意味するため、リソースをより積極的にクリーンアップする必要があります。これは、プロセスが終了する前にリソースをクリーンアップする最後の契機となります。

ShutdownAware インターフェイス

コンシューマークラスはオプションで **org.apache.camel.spi.ShutdownAware** インターフェイスを実装できます。このインターフェイスは、正常なシャットダウンメカニズムと対話し、コンシューマーがシャットダウンするための追加の時間を要求できるようにします。これは通常、内部キューに保留中のエクスチェンジを保存できる SEDA などのコンポーネントに必要です。通常、SEDA コンシューマーをシャットダウンする前にキューのすべてのエクスチェンジを処理します。

例41.3「ShutdownAware インターフェイス」 は、**ShutdownAware** インターフェイスの定義を示しています。

例41.3 ShutdownAware インターフェイス

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

import org.apache.camel.ShutdownRunningTask;

public interface ShutdownAware extends ShutdownPrepared {

    boolean deferShutdown(ShutdownRunningTask shutdownRunningTask);

    int getPendingExchangesSize();
}
```

ShutdownAware インターフェイスは以下のメソッドを定義します。

deferShutdown

コンシューマーのシャットダウンを遅延させる場合は、このメソッドから **true** を返します。**shutdownRunningTask** 引数は **enum** で、以下のいずれかの値を取ることができます。

- **ShutdownRunningTask.CompleteCurrentTaskOnly**: コンシューマーのスレッドプールによって現在処理されているエクスチェンジの処理を終了しますが、それ以上のエクスチェンジの処理は試みません。
- **ShutdownRunningTask.CompleteAllTasks**- 保留中のエクスチェンジ**すべて**を処理します。たとえば、SEDA コンポーネントの場合、コンシューマーは受信キューからすべてのエクスチェンジを処理します。

getPendingExchangesSize

コンシューマーによって処理されるエクスチェンジの数を示します。値をゼロにすると、処理が完了し、コンシューマーをシャットダウンできます。

ShutdownAware メソッドを定義する方法の例は、例41.7「カスタムスレッド実装」 を参照してください。

41.2. CONSUMER インターフェイスの実装

コンシューマーを実装する代替方法

コンシューマーは以下のいずれかの方法で実装できます。

- イベント駆動型のコンシューマーの実装
- スケジュールされたポーリングコンシューマーの実装
- ポーリングコンシューマーの実装
- カスタムスレッドの実装

イベント駆動型のコンシューマーの実装

イベント駆動型のコンシューマーでは、処理は外部イベントによって明示的に実行されます。イベントは、リスナーインターフェイスが特定のイベントソースに固有である event-listener インターフェイスを介して受信されます。

例41.4「JMXConsumer 実装」 は、Apache Camel JMX コンポーネント実装から取得した **JMXConsumer** クラスの実装を示しています。**JMXConsumer** クラスはイベント駆動型のコンシューマーの例で、**org.apache.camel.impl.DefaultConsumer** クラスから継承されることで実装されます。この **JMXConsumer** 例では、イベントは **NotificationListener.handleNotification()** メソッドの呼び出しによって表されます。これは JMX イベントを受信する標準的な方法です。これらの JMX イベントを受信するには、例41.4「JMXConsumer 実装」 にあるように NotificationListener インターフェイスを実装し、**handleNotification()** メソッドを上書きする必要があります。

例41.4 JMXConsumer 実装

```
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)
        }
    }
}
```

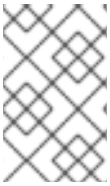
1. **JMXConsumer** パターンは、**DefaultConsumer** クラスを拡張してイベント駆動型のコンシューマーの通常のパターンに従います。また、このコンシューマーは (JMX 通知で表される) JMX から

2. 親エンドポイントへの参照 **endpoint** とチェーン内の次のプロセッサーへの参照 **processor** を引数に取るコンストラクターを少なくとも 1 つ実装する必要があります。

3. JMX 通知が到着すると (**NotificationListener** で定義された)、**handleNotification()** メソッドは JMX によって自動的に呼び出されます。このメソッドの本文には、コンシューマーのイベント処理を実行するコードが含まれている必要があります。**handleNotification()** 呼び出しは JMX レイヤーから発生するため、コンシューマーのスレッドモデルは **JMXConsumer** クラスではなく、JMX レイヤーによって暗黙的に制御されます。

4. このコード行は、2 つの手順を組み合わせたものです。まず、JMX 通知オブジェクトは、Apache Camel でのイベントの汎用表現であるエクスチェンジオブジェクトに変換されます。次に、新たに作成されたエクスチェンジオブジェクトはルートの次のプロセッサーに渡されます (同期的に呼び出します)。

5. **handleException()** メソッドは、**DefaultConsumer** ベースクラスによって実装されます。デフォルトでは、**org.apache.camel.impl.LoggingExceptionHandler** クラスを使用して例外を処理します。



注記

handleNotification() メソッドは JMX の例に固有です。独自のイベント駆動型のコンシューマーを実装する場合、カスタムコンシューマーに実装する同様のイベントリスナーメソッドを特定する必要があります。

スケジュールされたポーリングコンシューマーの実装

スケジュールされたポーリングコンシューマーでは、ポーリングイベントはタイマークラス **java.util.concurrent.ScheduledExecutorService** によって自動的に生成されます。生成されたポーリングイベントを受信するには、 **ScheduledPollConsumer.poll()** メソッドを実装する必要があります (「コンシューマーパターンおよびスレッド」 を参照)。

例41.5「ScheduledPollConsumer 実装」 では、スケジュールされたポーリングパターンに従ってコンシューマーを実装する方法を示します。これは、**ScheduledPollConsumer** クラスを拡張することで実装されます。

例41.5 ScheduledPollConsumer 実装

```
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 **org.apache.camel.impl.ScheduledPollConsumer** クラスを拡張して、スケジュールされたポーリングコンシューマークラス **CustomConsumer** を実装します。

2 親エンドポイントへの参照 **endpoint** とチェーン内の次のプロセッサーへの参照 **processor** を引数に取るコンストラクターを少なくとも 1 つ実装する必要があります。

3 スケジュールされたポーリングイベントを受信するには、**poll()** メソッドを上書きします。これは、受信イベントを取得して処理するコードを配置する場所です (エクスチェンジオブジェクトで表されます)。

4 この例では、イベントは同期的に処理されます。イベントを非同期的に処理する場合は、**getAsyncProcessor()** を代わりに呼び出して非同期プロセッサーへの参照を使用する必要があります。イベントを非同期的に処理する方法は、「非同期処理」 を参照してください。

5 **(オプション)** コンシューマーの開始時にコードを実行する場合は、以下のように **doStart()** メソッドを上書きします。

6 **(オプション)** コンシューマーの停止時にコードを実行する場合は、以下のように **doStop()** メソッドを上書きします。

ポーリングコンシューマーの実装

例41.6「PollingConsumerSupport 実装」 では、ポーリングパターンに従ってコンシューマーを実装する方法を説明します。これは、**PollingConsumerSupport** クラスを拡張することで実装されます。

例41.6 PollingConsumerSupport 実装

```
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) **org.apache.camel.impl.PollingConsumerSupport** クラスを拡張して、**ポーリングコンシューマークラス CustomConsumer** を実装します。

(2) 親エンドポイントへの参照 **endpoint** を引数に取るコンストラクターを少なくとも 1 つ実装する必要があります。ポーリングコンシューマーはプロセッサーインスタンスへの参照を必要としません。

(3) **receiveNoWait()** メソッドは、イベント (エクスチェンジオブジェクト) を取得するための非ブロッキングアルゴリズムを実装する必要があります。利用できるイベントがない場合は、**null** が返されます。

(4) この **receive()** メソッドでは、イベントを取得するためのブロッキングのアルゴリズムを実装する必要があります。このメソッドは、イベントが利用できない状態が続く場合に無期限にブロックできます。

5 **receive(long timeout)** メソッドは、指定したタイムアウト (通常はミリ秒単位で指定) までブロックできるアルゴリズムを実装します。

6 コンシューマーの起動またはシャットダウン中に実行するコードを挿入する場合は、**doStart()** メソッドと **doStop()** メソッドをそれぞれ実装します。

カスタムスレッドの実装

標準のコンシューマーパターンがコンシューマーの実装に適さない場合には、**Consumer** インターフェイスを直接実装してスレッドコードを作成することができます。ただし、スレッドコードを作成する場合は、「スレッドモデル」 に説明されているように、標準の Apache Camel スレッドモデルに従うことが重要です。

たとえば、**camel-core** にある SEDA コンポーネントは Apache Camel スレッドモデルに一貫性のある独自のコンシューマースレッドを実装しています。例41.7「カスタムスレッド実装」 クラスがスレッドを実装する方法の概要を **SedaConsumer** に示しています。

例41.7 カスタムスレッド実装

```
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** クラスは、**org.apache.camel.impl.ServiceSupport** クラスを拡張し、**Consumer**、**Runnable**、および **ShutdownAware** インターフェイスを実装することで実装されます。

(2)

Runnable.run() メソッドを実装し、スレッドで実行中にコンシューマーの動作を定義します。この場合、コンシューマーはループで実行され、新しいエクスチェンジのためにキューをポーリング

3 **doStart()** メソッドは **ServiceSupport** から継承されます。このメソッドは、コンシューマーの起動時に行う動作を定義するために上書きされます。

4 スレッドを直接作成するのではなく、**CamelContext** と共に登録された **ExecutorServiceStrategy** オブジェクトを使用してスレッドプールを作成する必要があり ます。これは、Apache Camel がスレッドの集中管理を実装し、正常なシャットダウンなどの機能をサポートすることができるため、重要になります。詳細は、「スレッドモデル」 を参照してください。

5 **ExecutorService.execute()** メソッド **poolSize** を呼び出し、スレッドを開始します。

6 **doStop()** メソッドは **ServiceSupport** から継承されます。このメソッドは、シャットダウン時にコンシューマーの動作を定義するために上書きされます。

7 **executor** インスタンスが表すスレッドプールをシャットダウンします。

第42章 PRODUCER インターフェイス

概要

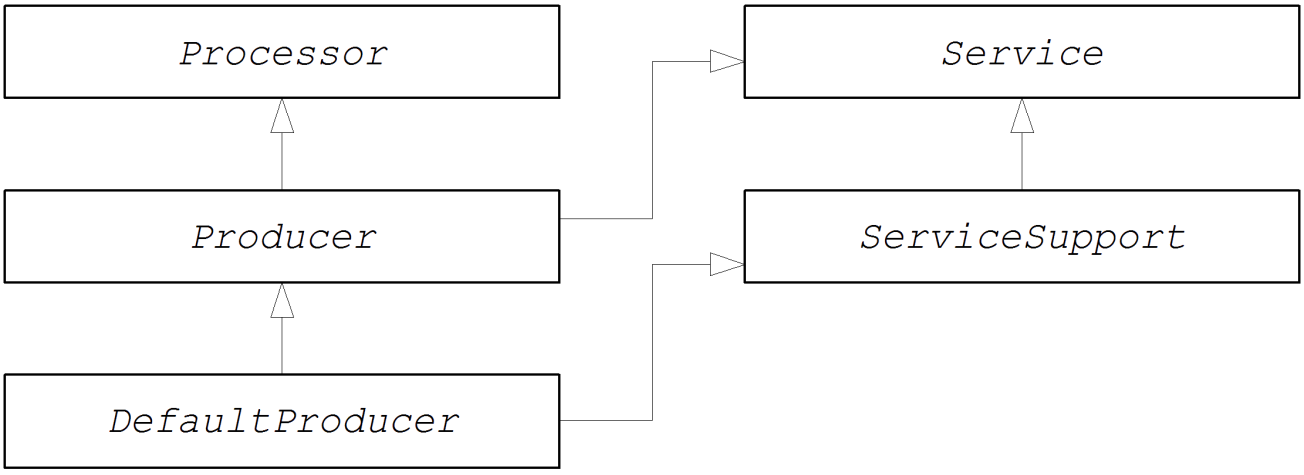
本章では、Apache Camel コンポーネントの実装における必須ステップである Producer インターフェイスを実装する方法を説明します。

42.1. PRODUCER インターフェイス

概要

org.apache.camel.Producer タイプのインスタンスは、ルートのターゲットエンドポイントを表します。プロデューサーのロールは、リクエスト (**In** メッセージ) を特定の物理エンドポイントに送信し、対応する応答 (**Out** または **Fault** メッセージ) を受信することです。**Producer** オブジェクトは、基本的にプロセッサーチェーン (ルートと同等) の最後に出現する特別な種類の **Processor** です。プロデューサーの継承階層を 図42.1「プロデューサーの継承階層」 に示します。

図42.1 プロデューサーの継承階層



Producer インターフェイス

例42.1「Producer インターフェイス」 は、**org.apache.camel.Producer** インターフェイスの定義を示しています。

例42.1 Producer インターフェイス

```
package org.apache.camel;

public interface Producer extends Processor, Service, IsSingleton {

    Endpoint<E> getEndpoint();

    Exchange createExchange();

    Exchange createExchange(ExchangePattern pattern);

    Exchange createExchange(E exchange);
}
```

プロデューサーメソッド

Producer インターフェイスは以下のメソッドを定義します。

- **process() (プロセッサーから継承)**: 最も重要なメソッドです。プロデューサーは、エクスチェンジオブジェクトを別のプロセッサーに転送する代わりに、リクエストをエンドポイントに送信する特別なタイプのプロセッサーです。**process()** メソッドを上書きすることで、プロデューサーが関連するエンドポイントとの間でメッセージを送受信する方法を定義します。
- **getEndpoint()**: 親エンドポイントインスタンスへの参照を返します。
- **createExchange()**: これらのオーバーロードされたメソッドは、Endpoint インターフェイスで定義された対応するメソッドに類似しています。通常、これらのメソッドは親エンドポイントインスタンスで定義された対応するメソッドに委譲されます (これはデフォルトで **DefaultEndpoint** クラスが実行するものです)。時折、これらのメソッドを上書きする必要がある場合があります。

非同期処理

プロデューサーでエクスチェンジオブジェクトを処理します。これは通常、リモートの宛先にメッセージを送信し、応答を待つことを含みます。そのためプロデューサーは、長時間ブロックする可能性があります。現在のスレッドをブロックしないようにするには、プロデューサーを **非同期プロセッサー** として実装できます。非同期処理パターンは、前述のプロセッサーをプロデューサーから切り離し、**process()** メソッドは遅延なく返却されます。「非同期処理」を参照してください。

プロデューサーを実装する場合、org.apache.camel.AsyncProcessor インターフェイスを実装して非同期処理モデルをサポートすることができます。それ自体では、非同期処理モデルが使用される訳ではありません。また、チェーン内で前述のプロセッサーを使用して **process()** メソッドの非同期バージョンを呼び出す必要もあります。AsyncProcessor インターフェースの定義は 例42.2「AsyncProcessor インターフェイス」 に示されています。

例42.2 AsyncProcessor インターフェイス

```
package org.apache.camel;

public interface AsyncProcessor extends Processor {
    boolean process(Exchange exchange, AsyncCallback callback);
}
```

process() メソッドの非同期バージョンは、org.apache.camel.AsyncCallback 型の追加の引数である **callback** を取ります。対応する AsyncCallback インターフェースは、例42.3「AsyncCallback インターフェイス」 に従って定義されます。

例42.3 AsyncCallback インターフェイス

```
package org.apache.camel;

public interface AsyncCallback {
    void done(boolean doneSynchronously);
}
```

AsyncProcessor.process() の呼び出し元は、処理が完了した通知を受信する AsyncCallback の実装を

提供する必要があります。**AsyncCallback.done()** メソッドは、処理が同期的に実行されたかどうかを示すブール値引数を取ります。通常、フラグは **false** で非同期処理であることを示します。ただし、プロデューサーが (要求されたにもかかわらず) 非同期的に処理 **しない** ようにすることにも意味があります。たとえば、プロデューサーがエクスチェンジの処理がすぐに完了することを認識している場合は、同期して処理を行い、最適化できます。この場合、**doneSynchronous flag** は **true** に設定する必要があります。

ExchangeHelper クラス

プロデューサーを実装するとき、**org.apache.camel.util.ExchangeHelper** ユーティリティークラスのメソッドを呼び出すと便利です。**ExchangeHelper** クラスの詳細は、「ExchangeHelper クラス」 を参照してください。

42.2. PRODUCER インターフェイスの実装

プロデューサーを実装する代替方法

プロデューサーは以下のいずれかの方法で実装できます。

- 同期プロデューサーの実装方法
- 非同期プロデューサーの実装方法

同期プロデューサーの実装方法

例42.4「DefaultProducer 実装」 は、同期プロデューサーの実装方法の概要を説明しています。この場合、**Producer.process()** の呼び出しは応答を受け取るまでブロックします。

例42.4 DefaultProducer 実装

```
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 **org.apache.camel.impl.DefaultProducer** クラスを拡張して、カスタム同期プロデューサークラス **CustomProducer** を実装します。

2 親エンドポイントへの参照を取得するコンストラクターを実装します。

3 **process()** メソッド実装は、プロデューサーコードの中心となります。**process()** メソッドの実装は、実装するコンポーネントのタイプに完全に依存します。

概説では、**process()** メソッドは通常以下のように実装されます。

- エクスチェンジに In メッセージが含まれ、指定した交換パターンと一致する場合は、In メッセージを指定のエンドポイントに送信します。
- 交換パターンが Out メッセージの受信を予測する場合は、 Out メッセージが受信されるまで待ちます。これにより、通常、**process()** メソッドは長時間ブロックします。
- 返信が受信されたら、エクスチェンジオブジェクトに返信を添付するために **exchange.setOut()** を呼び出します。応答に障害メッセージが含まれる場合は、Out を使用して **Message.setFault(true)** メッセージに fault フラグを設定します。

非同期プロデューサーの実装方法

例42.5「CollectionProducer 実装」 に非同期プロデューサーを実装する方法を概説します。この場合、同期の **process()** メソッドと非同期の **process()** メソッド (追加の **AsyncCallback** 引数を取る) の両方を実装する必要があります。

例42.5 CollectionProducer 実装

```
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 **org.apache.camel.impl.DefaultProducer** クラスを拡張し、AsyncProcessor インターフェイスを実装してカスタム非同期プロデューサークラス **CustomProducer** を実装します。

2 親エンドポイントへの参照を取得するコンストラクターを実装します。

3 同期 **process()** メソッドを実装します。

4 非同期 **process()** メソッドを実装します。非同期メソッドは複数の方法で実装できます。ここでは、サブスレッドで実行されるコードを表す **java.lang.Runnable** インスタンスである **task** を作成する方法を示します。次に、Java スレッド API を使用してサブスレッドでタスクを実行します。たとえば、新しいスレッドを作成したり、既存のスレッドプールにタスクを割り当てたりして、これを行います。

5 通常、エクスチェンジが非同期的に処理されたことを示すために非同期 **process()** メソッドから **false** が返されます。

6 **CustomProducerTask** クラスは、サブスレッドで実行される処理コードをカプセル化します。このクラスは、**Exchange** オブジェクト (**exchange**) および **AsyncCallback** オブジェクト (**callback**) のコピーをプライベートメンバー変数として保存する必要があります。

7 **run()** メソッドには、In メッセージをプロデューサーエンドポイントに送信し (ある場合)、応答を受信するのを待つコードが含まれます。応答 (Out メッセージまたは Fault メッセージ) を受信し、エクスチェンジオブジェクトに挿入した後、呼び出し元に処理の完了を通知するために **callback.done()** を呼び出す必要があります。

第43章 EXCHANGE インターフェイス

概要

本章では、Exchange インターフェイスについて説明します。Apache Camel 2.0 で実行される camel-core モジュールのリファクタリングにより、カスタムエクスチェンジタイプを定義する必要がなくなりました。**DefaultExchange** 実装をすべてのケースで使用できるようになりました。

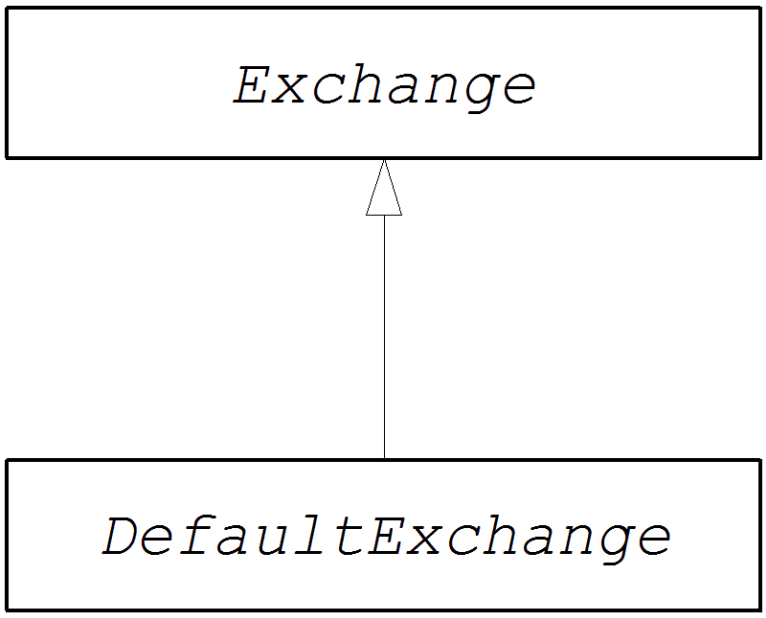
43.1. EXCHANGE インターフェイス

概要

org.apache.camel.Exchange 型のインスタンスは、エクスチェンジプロパティーとしてエンコードされた追加のメタデータと共に、ルートを通過する現在のメッセージをカプセル化します。

図43.1「エクスチェンジ継承階層」 エクスチェンジタイプの継承階層を表示します。デフォルトの実装は常に **DefaultExchange** が使用されます。

図43.1 エクスチェンジ継承階層



Exchange インターフェイス

例43.1「Exchange インターフェイス」 は、org.apache.camel.Exchange インターフェースの定義を示しています。

例43.1 Exchange インターフェイス

```
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);
}
```

エクスチェンジのメソッド

Exchange インターフェイスは以下のメソッドを定義します。

- **getPattern()**、**setPattern()** – エクスチェンジパターンは、**org.apache.camel.ExchangePattern** に列挙される値のいずれかになります。以下の交換パターンの値がサポートされます。
  - **InOnly**
  - **RobustInOnly**
  - **InOut**
  - **InOptionalOut**
  - **OutOnly**
  - **RobustOutOnly**
  - **OutIn**
  - **OutOptionalIn**
- **setProperty()**、**getProperty()**、**getProperties()**、**removeProperty()**、**hasProperties()** – プロパティー setter および getter メソッドを使用して、名前付きプロパティーをエクスチェンジのインスタンスに関連付けます。プロパティーは、コンポーネントの実装に必要なその他のメタデータで設定されます。
- In メッセージの setter および getter メソッド **setIn()**、**getIn()**。**DefaultExchange** クラスが提供する **getIn()** 実装は Lazy Creation セマンティクスを実装します。**In** メッセージが null の場合に **getIn()** が呼ばれると、**DefaultExchange** クラスはデフォルトの **In** メッセージを作成します。
- **setOut()**、**getOut()**、**hasOut()**: **Out** メッセージの setter メソッドおよび getter メソッド。この **getOut()** メソッドは、**Out** メッセージの Lazy Creation を暗黙的にサポートします。つまり、現在の **Out** メッセージが **null** の場合は、新しいメッセージインスタンスが自動的に作成されます。
- **setException()**, **getException()** – (**Throwable** 型の) 例外オブジェクトの getter および setter メソッド。
- **isFailed()**: エクスチェンジが例外または障害におり失敗した場合に **true** を返します。
- **isTransacted()** – エクスチェンジが処理された場合に、**true** を返します。
- **isRollback()**: エクスチェンジがロールバック用にマークされている場合に **true** を返します。
- **getContext()** – 関連付けられた **CamelContext** インスタンスへの参照を返します。
- **copy()**: 現在のカスタムエクスチェンジオブジェクトのコピーを (エクスチェンジ ID は異なります) 新たに作成します。**In** メッセージのボディーおよびヘッダー、**Out** メッセージ (存在する場合)、および **Fault** メッセージ (存在する場合) もこの操作によってコピーされます。
- **setFromEndpoint()**、**getFromEndpoint()**: このメッセージを初期化したコンシューマーエンドポイントの getter メソッドおよび setter メソッド (通常は、ルートの開始時に **from()** DSL コマンドに表示されるエンドポイント)。

- **setFromRouteId()**、**getFromRouteId()**: このエクスチェンジを初期化したルート ID の getter および setter メソッド。**getFromRouteId()** メソッドは内部でのみ呼び出す必要があります。

- **setUnitOfWork()**、**getUnitOfWork()** – **org.apache.camel.spi.UnitOfWork** Bean プロパティーの getter および setter メソッド。このプロパティーは、トランザクションに参加できるエクスチェンジにのみ必要です。

- **setExchangeId()**、**getExchangeId()**: エクスチェンジ ID の getter メソッドおよび setter メソッド。カスタムコンポーネントがエクスチェンジ ID を使用するかどうかは実装の詳細です。

- **addOnCompletion()** – エクスチェンジの処理が完了したときに呼び出される **org.apache.camel.spi.Synchronization** コールバックオブジェクトを追加します。

- **handoverCompletions()**: すべての OnCompletion コールバックオブジェクトを、指定されたエクスチェンジオブジェクトに渡します。

第44章 MESSAGE インターフェイス

概要

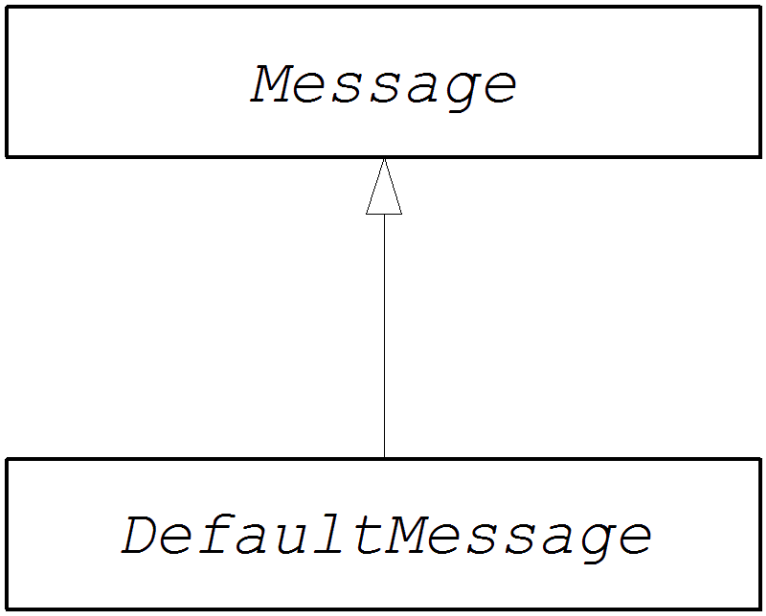
本章では、Apache Camel コンポーネントの実装で任意のステップである Message インターフェイスを実装する方法を説明します。

44.1. MESSAGE インターフェイス

概要

org.apache.camel.Message タイプのインスタンスは、あらゆる種類のメッセージ (**In** または **Out**) を表すことができます。メッセージタイプの継承階層を 図44.1「メッセージの継承階層」 に示します。コンポーネントにカスタムメッセージタイプを常に実装する必要はありません。多くの場合、デフォルトの実装 **DefaultMessage** で十分です。

図44.1 メッセージの継承階層



Message インターフェイス

例44.1「Message インターフェイス」は、**org.apache.camel.Message** インターフェイスの定義を示しています。

例44.1 Message インターフェイス

```
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();
}
```

メッセージメソッド

Message インターフェイスは以下のメソッドを定義します。

- **setMessageId()**、**getMessageId()**: メッセージ ID の getter メソッドおよび setter メソッド。カスタムコンポーネントでメッセージ ID を使用する必要があるかどうかが実装の詳細となります。
- **getExchange()**: 親エクスチェンジオブジェクトへの参照を返します。
- **isFault()**、**setFault()**: fault フラグの getter および setter メソッド。このメッセージが障害メッセージであるかどうかを示します。
- **getHeader()**、**getHeaders()**、**setHeader()**、**setHeaders()**、**removeHeader()**、**hasHeaders()**:

メッセージヘッダーのゲッターメソッドとセッターメソッド。通常、これらのメッセージヘッダーを使用して実際のヘッダーデータを保存するか、その他のメタデータを保存することもできます。

- **getBody()**、**getMandatoryBody()**、**setBody()**: メッセージボディーのゲッターメソッドとセッターメソッド。getMandatoryBody() アクセッサーは返されるボディーが null 以外であることを保証します。それ以外の場合は **InvalidPayloadException** 例外が発生します。
- **getAttachment()**、**getAttachments()**、**getAttachmentNames()**、**removeAttachment()**、**addAttachment()**、**setAttachments()**、**hasAttachments()**: 添付ファイルを取得、設定、追加、および削除するメソッド。
- **copy()**: 現在のカスタムメッセージオブジェクトと同一の (メッセージ ID を含む) 新しいコピーを作成します。
- **copyFrom()**: 指定した汎用メッセージオブジェクト **message** の完全なコンテンツ (メッセージ ID を含む) を現在のメッセージインスタンスにコピーします。このメソッドは **どのような** メッセージタイプからでもコピーできる必要があるため、汎用メッセージプロパティーがコピーされますが、カスタムプロパティーはコピーされません。
- **createExchangeId()**: メッセージ実装が ID を提供できる場合は、このエクスチェンジの一意の ID を返します。それ以外の場合は、**null** の戻り値を返します。

44.2. MESSAGE インターフェイスの実装

カスタムメッセージの実装方法

例44.2「カスタムメッセージの実装」 は、**DefaultMessage** クラスを拡張してメッセージを実装する方法を概説します。

例44.2 カスタムメッセージの実装

```
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 **org.apache.camel.impl.DefaultMessage** クラスを拡張し、カスタムメッセージクラス CustomMessage を実装します。

2 通常、デフォルトのプロパティーでメッセージを作成するデフォルトコンストラクターが必要です。

3 **toString()** メソッドを上書きして、メッセージ文字列をカスタマイズします。

4 **newInstance()** メソッドは、**MessageSupport.copy()** メソッド内から呼び出されます。**newInstance()** メソッドのカスタマイズは、現在のメッセージインスタンスのすべての **カスタム** プロパティーを新しいメッセージインスタンスにコピーすることにフォーカスする必要があります。この **MessageSupport.copy()** メソッドは、**copyFrom()** を呼び出すことにより汎用メッセージプロパティーをコピーします。

5 この **createBody()** メソッドは、**MessageSupport.getBody()** メソッドと連携して機能し、メッセージボディーへの遅延アクセスを実装します。デフォルトでは、メッセージのボディーは **null** です。これは、アプリケーションコードが (**getBody()** を呼び出して) ボディーにアクセスしようとする場合にのみ、ボディーが作成されます。メッセージのボディーに初めてアクセスすると、**MessageSupport.getBody()** は自動的に **createBody()** を呼び出します。

6 この **populateInitialHeaders()** メソッドは、ヘッダー getter および setter メソッドと連携して機能し、メッセージヘッダーへの遅延アクセスを実装します。このメソッドはメッセージを解析して、メッセージヘッダーを抽出し、ハッシュマップ **map** に挿入します。**populateInitialHeaders()** メソッドは、ユーザーが (**getHeader()**、**getHeaders()**、**setHeader()**、または **setHeaders()** を呼び出すことにより) ヘッダーに初めてアクセスしようとすると自動的に呼び出されます。

7 この **populateInitialAttachments()** メソッドは、アタッチメントの getter および setter メソッドと連携して機能し、アタッチメントへの遅延アクセスを実装します。このメソッドは、メッセージのアタッチメントを抽出し、ハッシュマップ **map** に挿入します。**populateInitialAttachments()** メソッドは、**getAttachment()**、**getAttachments()**、**getAttachmentNames()**、または **addAttachment()** を呼び出して、ユーザーが初めて添付ファイルにアクセスしようとすると、自動的に呼び出されます。

パート IV. API コンポーネントフレームワーク

API コンポーネントフレームワークを用いて、どのような Java API もラップする Camel コンポーネントを作成する方法

第45章 API コンポーネントフレームワークの概要

概要

API コンポーネントフレームワークは、大規模な Java API をベースとした複雑な Camel コンポーネントの実装に役立ちます。

45.1. API COMPONENT FRAMEWORK とは

必要になる状況

オプションの数が少ないコンポーネントの場合、コンポーネントの実装に関する標準的なアプローチ (38章 *コンポーネントの実装*) は、非常に効果的です。ただし、ここで問題となるのは、多数のオプションを指定してコンポーネントを実装する必要がある場合です。この問題は、エンタープライズレベルのコンポーネントになると発生します。これには、**数百もの** 操作で設定される API をラップすることが必要になる場合があります。このようなコンポーネントでは、作成と保守に大きな作業が必要です。

API コンポーネントフレームワークは、これらのコンポーネントの実装における課題に対処するために正確に開発されました。

API のコンポーネントへの切り替え

Java API をベースとした Camel コンポーネントを実装する経験から、多くの作業がルーチンかつ機械的なものであることが分かります。これは、特定の Java メソッドを取得して、特定の URI 構文にマッピングし、ユーザーが URI オプションを使用してメソッドパラメーターを設定できるようにすることから設定されます。このタイプの作業は、自動化およびコード生成の明確な候補です。

汎用 URI 形式

Java API の実装を自動化する最初の手順は、API メソッドを URI にマッピングする標準的な方法を設計することです。そのためには、**Java** API をラップするために使用できる汎用 URI 形式を定義する必要があります。そのため、API コンポーネントフレームワークはエンドポイント URI に以下の構文を定義します。

```
scheme://endpoint-prefix/endpoint?Option1=Value1&...&OptionN=ValueN
```

ここで **scheme** は、コンポーネントで定義されるデフォルトの URI スキームです。**endpoint-prefix** は、ラップされた Java API からクラスまたはインターフェイスのいずれかにマップする短い API 名で、**endpoint** はメソッド名にマップします。URI オプションはメソッド引数名にマップします。

単一 API クラスの URI 形式

API が単一の Java クラスで設定される場合は、URI の **endpoint-prefix** の部分が冗長になり、URI を以下の短い形式で指定できます。

```
scheme://endpoint?Option1=Value1&...&OptionN=ValueN
```

注記

この URI 形式を有効にするには、コンポーネント実装者が API コンポーネント Maven プラグインの設定で **apiName** 要素を空白のままにしておく必要もあります。詳細は、「API マッピングの設定」 セクションを参照してください。

リフレクションとメタデータ

Java メソッド呼び出しを URI 構文にマッピングするには、リフレクションメカニズムの形式が必ず必要になります。しかし、標準の Java リフレクション API は、メソッド引数名を保持しないという制限があります。有用な URI オプション名を生成するにはメソッド引数名が必要なため、これは問題です。このソリューションは、Javadoc またはメソッド署名ファイルのいずれかの代替形式でメタデータを提供することです。

Javadoc

javadoc はメソッド引数名を含む完全なメソッド署名を保持するため、API コンポーネントフレームワークのメタデータの理想的な形式です。また、生成が簡単で (特に **maven-javadoc-plugin** を使用して)、多くの場合はサードパーティーライブラリーですでに提供されています。

メソッド署名ファイル

Javadoc が何らかの理由で利用できない、または不適切な場合、API コンポーネントフレームワークは、メタデータの代替ソース (メソッド署名ファイル) もサポートします。署名ファイルは、Java メソッド署名の一覧で設定される単純なテキストファイルです。これらのファイルは、Java コードからコピーおよび貼り付ける (また生成されたファイルを編集する) ことにより、手動で簡単に作成できます。

フレームワークの設定

コンポーネント開発者の観点からは、API コンポーネントフレームワークは以下のように多数の異なる要素で設定されます。

Maven archetype

camel-archetype-api-component Maven archetype は、コンポーネント実装のスケルトンコードを生成するために使用されます。

Maven プラグイン

camel-api-component-maven-plugin Maven プラグインは、Java API とエンドポイント URI 構文との間でマッピングを実装するコードを生成します。

特殊なベースクラス

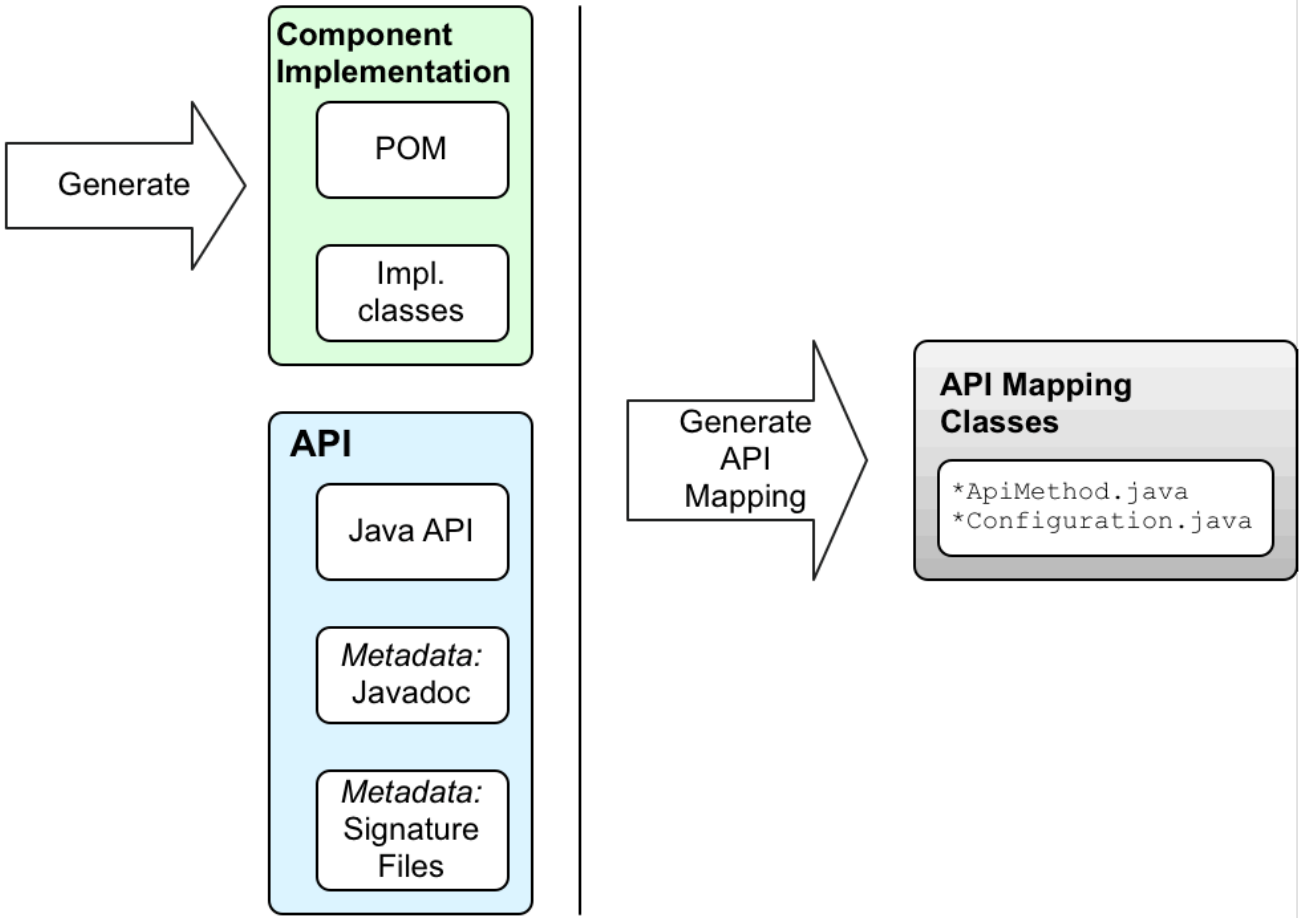
API コンポーネントフレームワークのプログラミングモデルをサポートするために、Apache Camel コアは **org.apache.camel.util.component** パッケージで特殊な API を提供します。この API は、コンポーネント、エンドポイント、コンシューマー、およびプロデューサークラスに特殊なベースクラスを提供します。

45.2. フレームワークの使用方法

概要

API フレームワークを使用してコンポーネントを実装する手順には、Maven POM ファイルを編集して、自動化されたコード生成、Java コードの実装、およびビルドのカスタマイズが関係します。以下の図は、この開発プロセスの概要を示しています。

図45.1 API コンポーネントフレームワークの使用



Java API

API コンポーネントの始点は常に Java API です。通常、Camel のコンテキストでは、リモートサーバーエンドポイントに接続する Java クライアント API を意味します。最初の質問は、Java API のソースが何であるかです。以下である可能性があります。

- Java API を独自に実装します (ただし、これは通常多くの作業を伴い、一般的には推奨されません)。
- サードパーティーの Java API を使用します。たとえば、Apache Camel Box コンポーネントはサードパーティーの Box Java SDK ライブラリーに基づいています。
- 言語に依存しないインターフェイスから Java API を生成します。

Javadoc メタデータ

Javadoc (API コンポーネントフレームワークでコードを生成するのに必要な) 形式で Java API のメタデータを提供するオプションがあります。Maven リポジトリーからサードパーティーの Java API を使用する場合、通常は Javadoc がすでに Maven アーティファクトで提供されています。ただし、Javadoc が提供されて **いない** 場合でも、**maven-javadoc-plugin** Maven プラグインを使用すると簡単に生成できます。

注記

現在、Javadoc メタデータの処理には、一般的なネストには対応していないといった制限があります。たとえば、**java.util.List<String>** がサポートされていますが、**java.util.List<java.util.List<String>>** はサポートされていません。回避策として、ネストされた汎用タイプを署名ファイルで **java.util.List<java.util.List>** として指定します。

署名ファイルメタデータ

何らかの理由で Javadoc の形式で Java API メタデータを提供することが適切ではない場合がありますが、**署名ファイル** の形式でメタデータを提供するオプションがあります。署名ファイルは、メソッド署名のリストで設定されます (行ごとに 1 つのメソッド署名)。これらのファイルは手動で作成でき、ビルド時にのみ必要です。

署名ファイルについては、以下の点に注意してください。

- 各プロキシークラス (Java API クラス) ごとに署名ファイルを 1 つ作成する必要があります。
- メソッドの署名では例外は発生 **しません**。ランタイム時に発生するすべての例外は、**RuntimeCamelException** でラップされ、エンドポイントから返されます。
- 引数のタイプを指定するクラス名は完全修飾クラス名である必要があります (**java.lang.*** タイプを除く)。パッケージ名をインポートするメカニズムはありません。
- 現在、署名パーサーには、一般的なネストがサポートされていないといった制限があります。たとえば、**java.util.List<String>** がサポートされていますが、**java.util.List<java.util.List<String>>** はサポートされていません。回避策として、ネストされた汎用タイプを **java.util.List<java.util.List>** として指定します。

以下は、署名ファイルの内容の簡単な例です。

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

Maven archetype での開始コードの生成

API コンポーネントの開発を開始する最も簡単な方法は、**camel-archetype-api-component** Maven archetype を使用して初期 Maven プロジェクトを生成することです。archetype の実行方法は、「Maven archetype でのコードの生成」 を参照してください。

Maven archetype を実行すると、生成された **ProjectName** ディレクトリーの下に 2 つのサブプロジェクトが表示されます。

ProjectName-api

このプロジェクトには、API コンポーネントの基礎を設定する Java API が含まれます。このプロジェクトをビルドする際に、Maven バンドルで Java API をパッケージ化し、必要な Javadoc も生成します。Java API および Javadoc がサードパーティーによってすでに提供されている場合は、このサブプロジェクトは必要ありません。

ProjectName-component

このプロジェクトには、API コンポーネントのスケルトンコードが含まれます。

コンポーネントクラスの編集

ProjectName-component のスケルトンコードを編集して、独自のコンポーネント実装を開発することができます。以下の生成されたクラスはスケルトン実装の中核を成しています。

> **ComponentName**Component
> **ComponentName**Endpoint
> **ComponentName**Consumer
> **ComponentName**Producer
> **ComponentName**Configuration

POM ファイルのカスタマイズ

また、Maven POM ファイルを編集してビルドをカスタマイズし、**camel-api-component-maven-plugin** Maven プラグインを設定する必要もあります。

camel-api-component-maven-plugin の設定

POM ファイルの設定で最も重要なことは、**camel-api-component-maven-plugin** Maven プラグインの設定です。このプラグインは、API メソッドとエンドポイント URI 間のマッピングを生成し、プラグイン設定を編集してマッピングをカスタマイズできます。

たとえば、**ProjectName-component/pom.xml** ファイルでは、以下の **camel-api-component-maven-plugin** プラグイン設定は、**ExampleJavadocHello** という API クラスの最小限の設定を示しています。

```
<configuration>
  <apis>
    <api>
      <apiName>hello-javadoc</apiName>
      <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
      <fromJavadoc/>
    </api>
  </apis>
</configuration>
```

この例では、**hello-javadoc** API 名は **ExampleJavadocHello** クラスにマップされています。つまり、**scheme://hello-javadoc/endpoint** 形式の URI を使用して、このクラスからメソッドを呼び出すことができます。**fromJavadoc** 要素が存在する場合は、**ExampleJavadocHello** クラスが Javadoc からメタデータを取得することを示しています。

OSGi バンドルの設定

component サブプロジェクトのサンプル POM である **ProjectName-component/pom.xml** は、コンポーネントを OSGi バンドルとしてパッケージ化するように設定されています。コンポーネント POM には、**maven-bundle-plugin** の設定例が含まれています。Maven がコンポーネント用に適切に設定された OSGi バンドルを生成するには、**maven-bundle-plugin** プラグインの設定をカスタマイズする必要があります。

コンポーネントの構築

Maven でコンポーネントをビルドする場合 (例: **mvn clean package** を使用)、**camel-api-component-maven-plugin** プラグインは (Java API とエンドポイント URI 構文間のマッピングを定義する) API マッピングクラスを自動的に生成し、それらを **target/classes** プロジェクトのサブディレクトリーに配置します。大規模で複雑な Java API を扱う場合、この生成されたコードは実際にはコンポーネントのソースコードの大部分を設定します。

Maven ビルドが完了すると、コンパイルされたコードおよびリソースが OSGi バンドルとしてパッケージ化され、ローカル Maven リポジトリーに Maven アーティファクトとして保存されます。

第46章 フレームワークの使用方法

概要

本章では、**camel-archetype-api-component** Maven archetype を使用して生成されたコードに基づいた、API コンポーネントフレームワークを使用して Camel コンポーネントを実装する基本的な原則について説明します。

46.1. MAVEN ARCHETYPE でのコードの生成

Maven archetype

Maven archetype はコードウィザードに類似しています。簡単なパラメーターをいくつか提供するとサンプルコードと共に、完全な作業用の Maven プロジェクトを生成します。その後、このプロジェクトをテンプレートとして使用し、実装をカスタマイズして独自のアプリケーションを作成することができます。

API コンポーネント Maven archetype

API コンポーネントフレームワークは **camel-archetype-api-component** により、独自の API コンポーネント実装の開始するコードを生成できる Maven archetype を提供します。これは、独自の API コンポーネントの作成を開始するための推奨される方法です。

前提条件

camel-archetype-api-component archetype を実行するための前提条件は Apache Maven がインストールされ、Maven **settings.xml** ファイルが標準の Fuse リポジトリーを使用するよう設定されていることだけです。

Maven archetype の呼び出し

example URI スキームを使用する **Example** コンポーネントを作成するには、以下のように **camel-archetype-api-component** archetype を呼び出して新しい Maven プロジェクトを生成します。

```
mvn archetype:generate \
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-api-component \
-DarchetypeVersion=2.23.2.fuse-790054-redhat-00001 \
-DgroupId=org.jboss.fuse.example \
-DartifactId=camel-api-example \
-Dname=Example \
-Dscheme=example \
-Dversion=1.0-SNAPSHOT \
-DinteractiveMode=false
```

注記

各行の最後にあるバックスラッシュ \ は、Linux プラットフォームおよび UNIX プラットフォームでのみ機能する行継続を表します。Windows プラットフォームでは、バックスラッシュを削除し、引数をすべて 1 行に配置します。

オプション

オプションは、**-DName=Value** 構文を使用して archetype 生成コマンドに提供されます。オプションのほとんどは前述の **mvn archetype:generate** コマンドのように設定する必要がありますが、生成されたプロジェクトをカスタマイズするためにいくつかのオプションを変更できます。生成された API コンポーネントプロジェクトをカスタマイズするために使用できるオプションを以下の表に示します。

名前	説明
groupId	**(汎用 Maven オプション)** 生成された Maven プロジェクトのグループ ID を指定します。デフォルトでは、この値は生成されたクラスの Java パッケージ名も定義します。そのため、この値を希望する Java パッケージ名と一致するように選択することが推奨されます。
artifactId	**(汎用 Maven オプション)** 生成された Maven プロジェクトのアーティファクト ID を指定します。
name	コンポーネントの名前この値は、生成されたコードでクラス名を生成するために使用されます (よって、名前が大文字で始まることが推奨されます)。
scheme	このコンポーネントの URI で使用するデフォルトのスキーム。このスキームが既存の Camel コンポーネントのスキームと競合しないようにしてください。
archetypeVersion	**(Maven の汎用オプション)** 正確には、コンポーネントをデプロイする予定であるコンテナーによって使用される Apache Camel バージョンになります。ただし、必要に応じて、プロジェクトの生成**後**に Maven 依存関係のバージョンを変更することもできます。

生成されたプロジェクトの構造

コード生成ステップが正常に完了すると、新しい Maven プロジェクトが含まれる新しいディレクトリー **camel-api-example** を確認できるはずです。**camel-api-example** ディレクトリー内に以下の一般的な構造があることを確認できます。

```
camel-api-example/
    pom.xml
    camel-api-example-api/
    camel-api-example-component/
```

プロジェクトのトップレベルは集約 POM **pom.xml** で、以下のように 2 つのサブプロジェクトをビルドするように設定されます。

camel-api-example-api

API サブプロジェクト (名前は **ArtifactId-api**) は、コンポーネントに変換する予定の Java API を保持します。独自に作成した Java API で API コンポーネントを指定する場合は、Java API コードを直接このプロジェクトに配置できます。
API サブプロジェクトは、以下のいずれかの目的で使用できます。

- Java API コードをパッケージ化する (Maven パッケージとして利用できない場合)。
- Java API の Javadoc を生成する (API コンポーネントフレームワークに必要なメタデータを提供)。
- API の説明から Java API コードを生成する (REST API の WADL 記述など)。

ただし、場合によっては、これらのタスクを実行する必要がない場合があります。たとえば、API コンポーネントのベースとなるサードパーティー API が、Maven パッケージで Java API および Javadoc をすでに提供している場合がこれに該当します。このような場合には、API サブプロジェクトを削除できます。

camel-api-example-component

コンポーネントサブプロジェクト (名前は **ArtifactId-component**) は、新規 API コンポーネントの実装を保持します。これには、コンポーネント実装クラス (Java API から API マッピングクラスを生成する) と **camel-api-component-maven** プラグインの設定が含まれます。

46.2. 生成される API サブプロジェクト

概要

「Maven archetype でのコードの生成」 の説明どおりに新しい Maven プロジェクトを生成した場合、**camel-api-example/camel-api-example-api** プロジェクトディレクトリーに Java API をパッケージ化するための Maven サブプロジェクトがあります。このセクションでは、生成されたサンプルコードの詳細と、その動作を説明します。

Java API のサンプル

生成されたサンプルコードには、API コンポーネントのベースとなるサンプル Java API が含まれています。サンプル Java API は比較的シンプルで、2 つの Hello World クラス (**ExampleJavadocHello** および **ExampleFileHello**) で設定されています。

ExampleJavadocHello クラス

例46.1「ExampleJavadocHello クラス」 は、サンプル Java API からの **ExampleJavadocHello** クラスを示しています。クラスの名前が示すように、このクラスは Javadoc からマッピングメタデータを提供する方法を示すために使用されます。

例46.1 ExampleJavadocHello クラス

```
// 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 クラス

例46.2「ExampleFileHello クラス」 は、サンプル Java API からの **ExampleFileHello** クラスを示しています。クラスの名前が示すように、このクラスは署名ファイルからマッピングメタデータを提供する方法を示すために使用されます。

例46.2 ExampleFileHello クラス

```
// 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;
    }
}
```

ExampleJavadocHello の Javadoc メタデータの生成

ExampleJavadocHello のメタデータは Javadoc として提供されるため、サンプル Java API の Javadoc を生成し、これを **camel-api-example-api** Maven アーティファクトにインストールする必要があります。API POM ファイルである **camel-api-example-api/pom.xml** は、Maven ビルド中にこの手順を自動的に実行するように **maven-javadoc-plugin** を設定します。

46.3. 生成されたコンポーネントサブプロジェクト

概要

新しいコンポーネントのビルド用の Maven サブプロジェクトは **camel-api-example/camel-api-example-component** プロジェクトディレクトリーの下にあります。このセクションでは、生成されたサンプルコードの詳細と、その動作を説明します。

コンポーネント POM での Java API の提供

Java API はコンポーネント POM の依存関係として提供される必要があります。たとえば、サンプル Java API は、以下のようにコンポーネント POM ファイルの依存関係 **camel-api-example-component/pom.xml** として定義されます。

```
<?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>
```

コンポーネント POM での Javadoc メタデータの提供

Java API のすべてのまたは一部に Javadoc メタデータを使用している場合は、コンポーネント POM の依存関係として Javadoc を指定する必要があります。この依存関係に関して注意することが 2 つあります。

- Javadoc の Maven コーディネートは Java API の Maven コーディネートとほぼ同じですが、以下のように **classifier** 要素も指定する必要があります。

  ```
  <classifier>javadoc</classifier>
  ```

- 以下のように、**provided** スコープを持つよう Javadoc を宣言する必要があります。

  ```
  <scope>provided</scope>
  ```

たとえば、コンポーネント POM では、Javadoc 依存関係は以下のように定義されます。

```
<?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>
```

サンプルファイル Hello のファイルメタデータの定義

ExampleFileHello のメタデータは署名ファイルで提供されます。通常、このファイルは手動で作成する必要がありますが、メソッド署名のリスト (各行に 1 つずつ) で設定される非常にシンプルな形式です。このサンプルコードは **camel-api-example-component/signatures** ディレクトリーに以下の内容を含む署名ファイル **file-sig-api.txt** を提供します。

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

署名ファイル形式の詳細は、「署名ファイルメタデータ」 を参照してください。

API マッピングの設定

API コンポーネントフレームワークの主な機能の 1 つは、**API マッピング** を実行するコードを自動的に生成することです。つまり、エンドポイント URI を Java API のメソッド呼び出しにマッピングするスタブコードを生成します。API マッピングへの基本的な入力は、Java API、Javadoc メタデータ、署名ファイルのメタデータなどです。

API マッピングを実行するコンポーネントは、コンポーネント POM で設定された **camel-api-component-maven-plugin** Maven プラグインです。コンポーネント POM からの以下の抜粋は、**camel-api-component-maven-plugin** プラグインの設定方法を示しています。

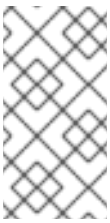
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">

  ...
  <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>
```

プラグインは、Java API のクラスを設定するための単一の **apis** 子要素が含まれる **configuration** 要素によって設定されます。各 API クラスは以下のように **api** 要素によって設定されます。

apiName

API name は、API クラスの短縮名で、エンドポイント URI の **endpoint-prefix** として使用されます。



注記

API が単一の Java クラスのみで設定される場合は、**apiName** 要素を空のままにすることができます。これにより、**endpoint-prefix** は冗長になります。その後、「単一 API クラスの URI 形式」 に示す形式を使用して、エンドポイント URI を指定できます。

proxyClass

このプロキシークラスの要素は、API クラスの完全修飾名を指定します。

fromJavadoc

API クラスに Javadoc メタデータが付随する場合、それを示すために **fromJavadoc** 要素を含める必要があります。**provided** 依存関係として、Javadoc 自体も Maven ファイルで指定する必要があります (「コンポーネント POM での Javadoc メタデータの提供」 を参照)。

fromSignatureFile

API クラスに署名ファイルのメタデータが付随する場合、これを示すために **fromSignatureFile** 要素を含める必要があります。この要素の内容で署名ファイルの場所を指定します。

注記

署名ファイルは、実行時ではなくビルド時にのみ必要であるため、Maven によってビルドされた最終パッケージには**含まれません**。

生成されたコンポーネントの実装

API コンポーネントは、**camel-api-example-component/src/main/java** ディレクトリー下において以下のコアクラス (Camel コンポーネントすべてに実装する必要がある) から設定されます。

ExampleComponent

コンポーネント自体を表します。このクラスは、エンドポイントインスタンスのファクトリーとして機能します (例: **ExampleEndpoint** のインスタンス)。

ExampleEndpoint

エンドポイント URI を表します。このクラスは、コンシューマーエンドポイント (例: **ExampleConsumer**) のファクトリーとして機能し、またプロデューサーエンドポイント (例: **ExampleProducer**) のファクトリーとして機能します。

ExampleConsumer

エンドポイント URI で指定された場所からメッセージを消費できるコンシューマーエンドポイントの具象インスタンスを表します。

ExampleProducer

プロデューサーエンドポイントの具象インスタンスを表します。これは、エンドポイント URI で指定された場所にメッセージを送信できます。

ExampleConfiguration

エンドポイント URI オプションを定義するために使用できます。この設定クラスによって定義される URI オプションは特定の API クラスには**関連付けられません**。つまり、これらの URI オプションを API クラスまたはメソッドのいずれかと組み合わせることができます。これは、たとえば、リモートサービスへの接続にユーザー名およびパスワードの認証情報を宣言する必要がある場合などに役立ちます。**ExampleConfiguration** クラスの主な目的は、API クラスまたは API インターフェイスを実装するクラスに必要なパラメーターの値を提供することです。たとえば、これらはコンストラクターパラメーターや、ファクトリーメソッドまたはクラスのパラメーター値になります。
このクラスで URI オプション **option** を実装するために必要なのは、アクセッサーメソッドのペア **getOption** and **setOption** を実装することだけです。コンポーネントフレームワークはエンドポイント URI を自動的に解析し、実行時にオプション値を挿入します。

ExampleComponent クラス

生成された **ExampleComponent** クラスは、以下のように定義されます。

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

このクラスの重要なメソッドは **createEndpoint** で、これは新しいエンドポイントインスタンスを作成します。通常、コンポーネントクラスのデフォルトコードを変更する必要はありません。ただし、このコンポーネントのライフサイクルと同じオブジェクトが存在する場合は、これらのオブジェクトをコンポーネントクラスから利用できるようにすることができます (例: メソッドを追加してこれらのオブジェクトを作成、これらのオブジェクトをコンポーネントに注入)。

ExampleEndpoint クラス

生成された **ExampleEndpoint** クラスは以下のように定義されます。

```
// 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;
    }
}
```

API コンポーネントフレームワークのコンテキストでは、エンドポイントクラスによって実行される主なステップの 1 つは、**API プロキシー** を作成することです。API プロキシーは、ターゲット Java API からのインスタンスで、メソッドがエンドポイントによって呼び出されます。通常、Java API は多くのクラスで設定されるため、URI に表示される **endpoint-prefix** を基にして、適切な API クラスを選択する必要があります (URI の一般形式は **scheme://endpoint-prefix/endpoint** であることに留意してください)。

ExampleConsumer クラス

生成された **ExampleConsumer** クラスは以下のように定義されます。

```
// 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);
    }

}
```

ExampleProducer クラス

生成された **ExampleProducer** クラスは以下のように定義されます。

```
// 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 クラス

生成された **ExampleConfiguration** クラスは以下のように定義されます。

```
// 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
}
```

このクラスに URI オプション **option** を追加するには、適切なタイプのフィールドを定義し、対応するアクセッサーメソッド (**getOption** および **setOption**) のペアを実装します。コンポーネントフレームワークはエンドポイント URI を自動的に解析し、実行時にオプション値を挿入します。

注記

このクラスは、すべての API メソッドと組み合わせることができる **一般的な** URI オプションを定義するために使用されます。特定の API メソッドに関連する URI オプションを定義するには、API コンポーネント Maven プラグインに **追加のオプション** を設定します。詳細は、「追加オプション」 を参照してください。

URI 形式

API コンポーネント URI の一般的な形式を取ります。

```
scheme://endpoint-prefix/endpoint?Option1=Value1&...&OptionN=ValueN
```

通常、URI は Java API の特定のメソッド呼び出しにマッピングされます。たとえば、API メソッド **ExampleJavadocHello.greetMe("Jane Doe")** を呼び出す場合、以下のように URI が作成されます。

scheme

Maven archetype でコードを生成した時に指定される API コンポーネントスキーム。この場合、スキームは **example** です。

endpoint-prefix

camel-api-component-maven-plugin Maven プラグイン設定で定義される API クラスにマップする API 名。**ExampleJavadocHello** クラスの場合、関連する設定は以下のとおりです。

```
<configuration>
  <apis>
    <api>
      <apiName>hello-javadoc</apiName>
      <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
      <fromJavadoc/>
    </api>
    ...
  </apis>
</configuration>
```

これは、必要な **endpoint-prefix** が **hello-javadoc** であることを示しています。

エンドポイント

endpoint は、メソッド名 **greetMe** にマップします。

Option1=Value1

URI オプションはメソッドパラメーターを指定します。**greetMe(String name)** メソッドは、**name=Jane%20Doe** として指定できる単一のパラメーター **name** を取ります。オプションのデフォルト値を定義する場合は、**interceptProperties** メソッドを上書きしてこれを行うことができます (「プログラミングモデル」 を参照)。

URI の一部をまとめると、以下の URI で **ExampleJavadocHello.greetMe("Jane Doe")** を呼び出しできることがわかります。

```
example://hello-javadoc/greetMe?name=Jane%20Doe
```

デフォルトのコンポーネントインスタンス

example URI スキームをデフォルトのコンポーネントインスタンスにマッピングするために、Maven archetype は **camel-api-example-component** サブプロジェクトに以下のファイルを作成します。

```
src/main/resources/META-INF/services/org/apache/camel/component/example
```

このリソースファイルは、Camel コアが **example** URI スキームに関連付けられたコンポーネントを特定できるようにします。ルートで **example:**// URI を使用するたびに、Camel はクラスパスを検索し、対応する **example** リソースファイルを探します。この **example** ファイルには、以下のコンテンツが含まれます。

```
class=org.jboss.fuse.example.ExampleComponent
```

これにより、Camel コアは **ExampleComponent** コンポーネントのデフォルトインスタンスを作成できます。このファイルを編集する必要があるのは、コンポーネントクラスの名前をリファクタリングする場合のみです。

46.4. プログラミングモデル

概要

API コンポーネントフレームワークのコンテキストでは、主なコンポーネント実装クラスは **org.apache.camel.util.component** パッケージのベースクラスから派生します。これらのベースクラスは、コンポーネントの実装時に (任意で) オーバーライドできるメソッドを定義します。このセクションでは、これらのメソッドと、コンポーネントの実装でこれらを使用する方法についての概要を説明します。

実装するコンポーネントメソッド

生成されるメソッド実装 (通常は変更する必要はありません) のほかに、**Component** クラスの以下のメソッドの一部を任意で上書きすることもできます。

doStart()

(任意) コールド起動時にコンポーネントのリソースを作成するためのコールバック。別の方法としては、**遅延初期化** の設定 (リソースが必要な場合のみ作成) を実行する方法があります。実際、遅延初期化が最適なストラテジーとなることが多いため、**doStart** メソッドは必要ないことがよくあります。

doStop()

(任意) コンポーネントが停止している間にコードを呼び出すコールバック。コンポーネントを停止すると、そのリソースがすべてシャットダウンされ、内部状態が削除され、キャッシュが消去されることを意味します。

注記

Camel は、対応する **doStart** が呼び出されなかった場合でも、現在の **CamelContext** のシャットダウン時に **常に doStop** が呼び出されることを保証します。

doShutdown

(オプション) CamelContext がシャットダウンしている間にコードを呼び出すコールバック。停止したコンポーネントを再起動することはできますが (コールドスタートのセマンティクスを使用)、シャットダウンするコンポーネントが完全に終了します。したがって、このコールバックは、コンポーネントに属するリソースを解放する最後の可能性を表します。

その他に Component クラスに実装するもの

Component クラスは、コンポーネントオブジェクト自体に同じ (または同様の) ライフサイクルを持つオブジェクトへの参照を保持するための最適な場所です。たとえば、コンポーネントが OAuth セキュリティーを使用する場合、**Component** クラスで必要な OAuth オブジェクトへの参照を保持し、OAuth オブジェクトを作成するための **Component** クラスでメソッドを定義するのが一般的です。

実装するエンドポイントメソッド

生成されたメソッドの一部を変更でき、必要に応じて以下のように **Endpoint** クラスで継承されたメソッドの一部を上書きすることができます。

afterConfigureProperties()

この方法で必要なのは、API 名に一致するプロキシークラス (API クラス) を作成することです。継承された **apiName** フィールドまたは **getApiName** アクセッサーを介して、(すでにエンドポイント URI から抽出されている) API 名を利用できます。通常、**apiName** フィールドでスイッチを実行して、対応するプロキシークラスを作成します。以下に例を示します。

```
// 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)

このメソッドを上書きして、**afterConfigureProperties** で作成したプロキシーインスタンスを返します。以下に例を示します。

```
@Override
public Object getApiProxy(ApiMethod method, Map<String, Object> args) {
    return apiProxy;
}
```

特殊なケースでは、API メソッドおよび引数に依存するプロキシーを選択する必要がある場合があります。必要に応じて、**getApiProxy** で、このアプローチを柔軟に行うことができます。

doStart()

(任意) コールド起動時にリソースを作成するためのコールバック。**Component.doStart()** と同じセマンティクスがあります。

doStop()

(任意) コンポーネントが停止している間にコードを呼び出すコールバック。**Component.doStop()** と同じセマンティクスがあります。

doShutdown

(オプション) コンポーネントがシャットダウンしている間にコードを呼び出すコールバック。**Component.doShutdown()** と同じセマンティクスがあります。

interceptPropertyNames(Set<String> propertyNames)

(任意) API コンポーネントフレームワークは、エンドポイント URI と指定されたオプション値を使用して、呼び出す方法を決定します (オーバーロードおよびエイリアスによる曖昧さ)。ただし、コンポーネントが内部的にオプションまたはメソッドパラメーターを追加する場合、このフレームワークでは呼び出す適切な方法を判断するためにヘルプが必要になる場合があります。この場合、**interceptPropertyNames** メソッドをオーバーライドし、追加の (非表示または暗黙的) オプションを **propertyNames** セットに追加する必要があります。メソッドパラメーターの完全なリストが **propertyNames** セットにあると、フレームワークは呼び出す適切なメソッドを特定できます。

注記

このメソッドは **Endpoint**、**Producer**、または **Consumer** クラスのレベルで上書きできます。基本的なルールは、オプションがプロデューサーエンドポイントとコンシューマーエンドポイントの **両方** に影響する場合に、**Endpoint** クラスのメソッドを上書きします。

interceptProperties(Map<String,Object> properties)

(任意) このメソッドを上書きすると、API メソッドが呼び出される前に、オプションの実際の値を変更または設定できます。たとえば、この方法を使用して、必要に応じて、一部のオプションにデフォルト値を設定できます。実際には、**interceptPropertyNames** メソッドおよび **interceptProperty** メソッドの両方を上書きする必要があることが多いです。

注記

このメソッドは **Endpoint**、**Producer**、または **Consumer** クラスのレベルで上書きできます。基本的なルールは、オプションがプロデューサーエンドポイントとコンシューマーエンドポイントの **両方** に影響する場合に、**Endpoint** クラスのメソッドを上書きします。

実装するコンシューマーメソッド

任意で、以下のように **Consumer** クラスで継承されたメソッドの一部を上書きできます。

interceptPropertyNames(Set<String> propertyNames)

(任意) このメソッドのセマンティクスは、**Endpoint.interceptPropertyNames** と似ています。

interceptProperties(Map<String,Object> properties)

(任意) このメソッドのセマンティクスは、**Endpoint.interceptProperties** と似ています。

doInvokeMethod(Map<String, Object> args)

(任意) このメソッドを上書きすると、Java API メソッドの呼び出しを妨害できます。このメソッドを上書きする最も一般的な理由は、メソッド呼び出しに関するエラー処理をカスタマイズすることです。たとえば、以下のコードフラグメントに **doInvokeMethod** を上書きする一般的な方法が示されています。

```
// Java
@Override
protected Object doInvokeMethod(Map<String, Object> args) {
    try {
        return super.doInvokeMethod(args);
    } catch (RuntimeCamelException e) {
        // TODO - Insert custom error handling here!
        ...
    }
}
```

Java API メソッドが呼び出されるように、この実装のある時点で **doInvokeMethod** をスーパークラスで呼び出す必要があります。

interceptResult(Object methodResult, Exchange resultExchange)

(任意) API メソッド呼び出しの結果に追加処理を行います。たとえば、この時点でカスタムヘッダーを Camel エクスチェンジオブジェクト **resultExchange** に追加できます。

Object splitResult(Object result)

(任意) デフォルトでは、メソッド API 呼び出しの結果が **java.util.Collection** オブジェクトまたは Java 配列である場合、API コンポーネントフレームワークは結果を **複数** のエクスチェンジオブジェクトに分割します (したがって、単一の呼び出しの結果が複数のメッセージに変換されます)。デフォルトの動作を変更する場合は、コンシューマーエンドポイントで **splitResult** メソッドを上書きすることができます。**result** 引数には、API メッセージ呼び出しの結果が含まれます。結果を分割する場合は、配列タイプを返す必要があります。

注記

エンドポイント URI で **consumer.splitResult=false** に設定して、デフォルトの分割動作をオフにすることもできます。

実装するプロデューサーメソッド

任意で、以下のように継承されたメソッドの一部を **Producer** クラスで上書きできます。

interceptPropertyNames(Set<String> propertyNames)

(任意) このメソッドのセマンティクスは、**Endpoint.interceptPropertyNames** と似ています。

interceptProperties(Map<String,Object> properties)

(任意) このメソッドのセマンティクスは、**Endpoint.interceptProperties** と似ています。

doInvokeMethod(Map<String, Object> args)

(任意) このメソッドのセマンティクスは、**Consumer.doInvokeMethod** と似ています。

interceptResult(Object methodResult, Exchange resultExchange)

(任意) このメソッドのセマンティクスは、**Consumer.interceptResult** と似ています。

注記

Producer.splitResult() メソッドは呼び出しされないため、API メソッドの結果を、コンシューマーエンドポイントの場合と同じ方法で分割することは**できません**。プロデューサーエンドポイントに対して同様の効果を取得するには、Camel の **split()** DSL コマンド (標準のエンタープライズ統合パターンの 1 つ) を使用して **Collection** または配列の結果を分割できます。

コンシューマーポーリングおよびスレッドモデル

API コンポーネントフレームワークのコンシューマーエンドポイントのデフォルトのスレッドモデルは、**スケジュールされたポーリングコンシューマー**です。これは、コンシューマーエンドポイントの API メソッドが、定期的なスケジュールされた時間間隔で呼び出されることを意味します。詳細は、「スケジュールされたポーリングコンシューマーの実装」 を参照してください。

46.5. コンポーネントの実装例

概要

Apache Camel で配布されるコンポーネントの一部は、API コンポーネントフレームワークを利用して実装されています。フレームワークを使用して Camel コンポーネントを実装するテクニックについて学びたい場合は、これらのコンポーネント実装のソースコードを調べるとよいでしょう。

Box.com

Camel Box コンポーネント は、API コンポーネントフレームワークを使用してサードパーティーの Box.com Java SDK をモデル化し、呼び出す方法を示しています。また、Box.com の長いポーリング API をサポートするために、コンシューマーポーリングのカスタマイズにフレームワークがどのように適応するかも実証します。

GoogleDrive

Camel GoogleDrive コンポーネントは、API コンポーネントフレームワークが Method Object スタイルの Google API をどのように処理できるかを実証します。この場合、URI オプションはメソッドオブジェクトにマッピングされ、コンシューマーおよびプロデューサーの **doInvoke** メソッドを上書きすることで呼び出されます。

Olingo2

Camel Olingo2 コンポーネント は、コールバックベースの Asynchronous API が API コンポーネントフレームワークを使用してラップされる方法を実証します。この例は、HTTP NIO 接続などの基礎となるリソースに非同期処理をプッシュして、Camel エンドポイントをより効率的に実行する方法を示しています。

第47章 API コンポーネント MAVEN プラグインの設定

概要

本章では、API コンポーネント Maven プラグインで利用可能なすべての設定オプションのリファレンスを提供します。

47.1. プラグイン設定の概要

概要

API コンポーネント Maven プラグイン **camel-api-component-maven-plugin** の主な目的は、エンドポイント URI と API メソッド呼び出し間のマッピングを実装する API マッピングクラスを生成することです。API コンポーネント Maven プラグインの設定を編集することにより、API マッピングのさまざまな側面をカスタマイズできます。

生成されたコードの場所

API コンポーネント Maven プラグインによって生成された API マッピングクラスは、デフォルトで以下の場所に配置されます。

> **ProjectName**-component/target/generated-sources/camel-component

前提条件

API コンポーネント Maven プラグインの主な入力は、Java API クラスおよび Javadoc メタデータです。これらは、通常の Maven 依存関係として宣言することで、プラグインで利用できます (Javadoc の Maven 依存関係を **provided** スコープで宣言する必要があります)。

プラグインの設定

API コンポーネント Maven プラグインの設定方法として、API コンポーネントの archetype を使用して開始点とするコードを生成することが推奨されます。これにより、プロジェクト用にカスタマイズできるデフォルトのプラグイン設定が **ProjectName-component/pom.xml** ファイルに生成されます。プラグインのセットアップの主な内容は以下のとおりです。

1. 必要な Java API と Javadoc メタデータに対して、Maven の依存関係を宣言する必要があります。
2. プラグインの基本設定は **pluginManagement** スコープで宣言されます (使用するプラグインのバージョンも定義します) 。
3. プラグインインスタンス自体が宣言され、設定されます。
4. **build-helper-maven** プラグインは、**target/generated-sources/camel-component** ディレクトリーから生成されたソースを取得して、Maven ビルドに含めるように設定されています。

基本設定のサンプル

以下の POM ファイルの抜粋は、API コンポーネント Maven プラグインの基本設定です。API コンポーネント archetype を使用してコードが生成された際に、Maven **pluginManagement** スコープで定義されます。

```
<?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-790054-redhat-00001</version>
          <configuration>
            <scheme>${schemeName}</scheme>
            <componentName>${componentName}</componentName>
            <componentPackage>${componentPackage}</componentPackage>
            <outPackage>${outPackage}</outPackage>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
    ...
  </build>
  ...
</project
```

pluginManagement スコープで指定した設定は、プラグインのデフォルト設定を提供します。実際にはプラグインのインスタンスを作成するわけではありませんが、デフォルト設定は API コンポーネントプラグインインスタンスによって使用されます。

基本設定

前述の基本設定では、(**version** 要素で) プラグインバージョンを指定する他に、以下の設定プロパティーを指定します。

scheme

この API コンポーネントの URI スキーム。

componentName

この API コンポーネントの名前 (生成されるクラス名の接頭辞としても使用されます)。

componentPackage

API コンポーネント Maven archetype によって生成されたクラスが含まれる Java パッケージを指定します。このパッケージは、デフォルトの **maven-bundle-plugin** 設定でもエクスポートされます。したがって、クラスを一般に公開する場合は、この Java パッケージに配置してください。

outPackage

生成された API マッピングクラスが配置される Java パッケージを指定します (API コンポーネント Maven プラグインによって生成された場合)。デフォルトでは、これには **componentName** プロパティーの値があり、**.internal** 接尾辞が追加されています。このパッケージは、デフォルトの **maven-bundle-plugin** 設定では private として宣言されています。したがって、クラスを private にする場合は、この Java パッケージに配置してください。

インスタンスの設定例

以下の POM ファイルの抜粋は、API コンポーネント Maven プラグインのサンプルインスタンスを示しています。このインスタンスは、Maven ビルド中に API マッピングを生成するよう設定されています。

```
<?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>
```

基本的なマッピングの設定

プラグインは、Java API のクラスを設定するための単一の **apis** 子要素が含まれる **configuration** 要素によって設定されます。各 API クラスは以下のように **api** 要素によって設定されます。

apiName

API name は、API クラスの短縮名で、エンドポイント URI の **endpoint-prefix** として使用されます。

注記

API が単一の Java クラスのみで設定される場合は、**apiName** 要素を空のままにすることができます。これにより、**endpoint-prefix** は冗長になります。その後、「単一 API クラスの URI 形式」 に示す形式を使用して、エンドポイント URI を指定できます。

proxyClass

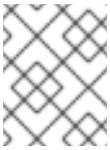
この要素は、API クラスの完全修飾名を指定します。

fromJavadoc

API クラスに Javadoc メタデータが付随する場合、それを示すために **fromJavadoc** 要素を含める必要があります。**provided** 依存関係として、Javadoc 自体も Maven ファイルで指定する必要があります。

fromSignatureFile

API クラスに署名ファイルのメタデータが付随する場合、これを示すために **fromSignatureFile** 要素を含める必要があります。この要素の内容で署名ファイルの場所を指定します。



注記

署名ファイルは、実行時ではなくビルド時にのみ必要であるため、Maven によってビルドされた最終パッケージには**含まれません**。

API マッピングのカスタマイズ

プラグインを設定することで、API マッピングの以下の部分をカスタマイズすることができます。

- **メソッドのエイリアス**: **aliases** 設定要素を使用して API メソッドの名前 (エイリアス) を定義できます。詳細は、「メソッドのエイリアス」 を参照してください。
- **Null 可能なオプション** - **nullableOptions** 設定要素を使用して、デフォルトが **null** のメソッド引数を宣言することができます。詳細は、「Null 可能なオプション」 を参照してください。
- **引数名の置換**: API マッピングの実装方法により、特定の API クラスのすべてのメソッドの引数は **同じ** 名前空間に属します。同じ名前の 2 つの引数が異なる型であると宣言されている場合は競合になります。このような名前の競合を回避するには、**substitutions** 設定要素を使用してメソッド引数の名前を変更することができます (URI 内で表示されます)。詳細は、「引数名の置換」 を参照してください。
- **引数の除外**: Java の引数を URI オプションにマッピングする際、マッピングから特定の引数を除外することがあります。**excludeConfigNames** 要素または **excludeConfigTypes** 要素のどちらかを指定し、不要な引数をフィルターリングすることができます。詳細は、「除外された引数」 を参照してください。
- **追加オプション** : Java API の一部ではない追加のオプションを定義することがあります。これは **extraOptions** 設定要素を使って行うことができます。

Javadoc のメタデータ設定

Javadoc のメタデータをフィルターして、特定のコンテンツを無視または明示的に含めることができます。設定方法の詳細については、「Javadoc オプション」 を参照してください。

署名ファイルのメタデータ設定

Javadoc が利用できない場合、必要なマッピングメタデータを提供するために署名ファイルを用いることができます。**fromSignatureFile** は、対応する署名ファイルの場所を指定するために使用されます。特別なオプションはありません。

47.2. JAVADOC オプション

概要

Java API のメタデータが Javadoc によって提供されている場合、通常はオプションなしで **fromJavadoc** 要素を指定すれば十分です。ただし、API マッピングに Java API 全体を含めたくない場合は、Javadoc メタデータをフィルターリングして内容をカスタマイズできます。言い換えると、API コンポーネント Maven プラグインは Javadoc メタデータを反復処理することで API マッピングを生成するため、Javadoc メタデータの不要な部分をフィルターリングすることで、生成される API マッピングのスコープをカスタマイズすることができます。

構文

fromJavadoc 要素には、以下のようにオプションの子要素を設定できます。

```
<fromJavadoc>
  <excludePackages>PackageNamePattern</excludePackages>
  <excludeClasses>ClassNamePattern</excludeClasses>
  <excludeMethods>MethodNamePattern</excludeMethods>
  <includeMethods>MethodNamePattern</includeMethods>
  <includeStaticMethods>[true|false]<includeStaticMethods>
</fromJavadoc>
```

スコープ

以下の抜粋で示されているように、**fromJavadoc** 要素は、任意で **apis** 要素の子や **api** 要素の子として表示できます。

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <fromJavadoc>...</fromJavadoc>
    </api>
    <fromJavadoc>...</fromJavadoc>
    ...
  </apis>
</configuration>
```

以下のスコープで **fromJavadoc** 要素を定義できます。

- **api 要素の子として**: **fromJavadoc** オプションは、**api** 要素で指定された API クラスにのみ適用されます。
- **apis 要素の子として**: **fromJavadoc** オプションは、デフォルトですべての API クラスに適用されますが、**api** レベルで上書きできます。

オプション

以下のオプションは、**fromJavadoc** の子要素として定義できます。

excludePackages

API マッピングモデルから Java パッケージを除外するための正規表現 (**java.util.regex** 構文) を指定します。正規表現に一致するパッケージ名はすべて除外され、**除外されたクラスから派生したすべてのクラスも無視されます**。デフォルト値は **javax?\.lang.*** です。

excludeClasses

API マッピングから API ベースクラスを除外するための正規表現 (**java.util.regex** 構文) を指定します。正規表現に一致するクラス名はすべて除外され、**除外されたクラスから派生するすべてのクラスも無視されます**。

excludeMethods

API マッピングモデルからメソッドを除外するための正規表現 (**java.util.regex** 構文) を指定します。

includeMethods

API マッピングモデルのメソッドを含むための正規表現 (**java.util.regex** 構文) を指定します。

includeStaticMethods

true の場合、static メソッドも API マッピングモデルに含まれます。デフォルトは **false** です。

47.3. メソッドのエイリアス

概要

Java API に表示される標準のメソッド名に加えて、与えられたメソッドに追加の名前 (エイリアス) を定義すると便利なことがよくあります。特に一般的なケースとして、プロパティー名 (例: **widget**) をアクセサーメソッド (**getWidget** または **setWidget** など) のエイリアスとして使用できるようにします。

構文

aliases 要素は、以下のように 1 つ以上の **alias** 子要素で定義できます。

```
<aliases>
  <alias>
    <methodPattern>MethodPattern</methodPattern>
    <methodAlias>Alias</methodAlias>
  </alias>
  ...
</aliases>
```

MethodPattern は、Java API からのメソッド名の照合に使用する正規表現 (**java.util.regex** 構文) です。パターンには通常、キャプチャーグループが含まれます。**Alias** は、(URI で使用するための) 代替式で、前述のキャプチャーグループのテキストを使用できます。たとえば、1 番目、2 番目、または 3 番目のキャプチャーグループのテキストは、**$1**、**$2**、または **$3** のように指定されます)。

スコープ

以下の抜粋で示されているように、**aliases** 要素は、任意で **apis** 要素の子や **api** 要素の子として表示できます。

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <aliases>...</aliases>
    </api>
    <aliases>...</aliases>
    ...
  </apis>
</configuration>
```

以下のスコープで **aliases** 要素を定義できます。

- **api 要素の子として**: **aliases** マッピングは、**api** 要素で指定された API クラスにのみ適用されます。
- **apis 要素の子として**: **aliases** マッピングはデフォルトですべての API クラスに適用されますが、**api** レベルで上書きできます。

例

以下の例は、一般的な get/set bean メソッドパターンのエイリアスを生成する方法を示しています。

```
<aliases>
  <alias>
    <methodPattern>[gs]et(.+)</methodPattern>
    <methodAlias>$1</methodAlias>
  </alias>
</aliases>
```

前述のエイリアス定義では、**getWidget** または **setWidget** のいずれかのメソッドのエイリアスとして **widget** を使用できます。メソッド名の後半部分をキャプチャーするために 、キャプチャーグループ **(.+)** を使用していることに注意してください (例: **Widget**)。

47.4. NULL 可能なオプション

概要

場合によっては、メソッドの引数をデフォルトで **null** にすることは理にかなっています。ただし、これはデフォルトでは許可されません。Java API メソッド引数の一部に **null** の値を取ることを許可する場合は、**nullableOptions** 要素を使用して明示的に宣言する必要があります。

構文

nullableOptions 要素は、以下のように 1 つ以上の **nullableOption** 子要素で定義できます。

```
<nullableOptions>
```

```
  <nullableOption>ArgumentName</nullableOption>
  ...
</nullableOptions>
```

ArgumentName は、Java API のメソッド引数の名前です。

スコープ

以下の抜粋で示されているように、**nullableOptions** 要素は、任意で **apis** 要素の子や **api** 要素の子として表示できます。

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <nullableOptions>...</nullableOptions>
    </api>
    ...
    <nullableOptions>...</nullableOptions>
  </apis>
</configuration>
```

以下のスコープで **nullableOptions** 要素を定義できます。

- **api 要素の子として**: **nullableOptions** マッピングは、**api** 要素で指定された API クラスにのみ適用されます。
- **apis 要素の子として**: **nullableOptions** マッピングは、デフォルトですべての API クラスに適用されますが、**api** レベルで上書きできます。

47.5. 引数名の置換

概要

API コンポーネントフレームワークでは、URI オプション名が **各プロキシークラス (Java API クラス) 内で一意** である必要があります。しかし、メソッドの引数名の場合は、必ずしもそうとは限りません。たとえば、API クラスの以下の Java メソッドについて考えてみましょう。

```
public void doSomething(int id, String name);
public void doSomethingElse(int id, String name);
```

Maven プロジェクトのビルド時、**camel-api-component-maven-plugin** は設定クラス **ProxyClassEndpointConfiguration** を生成します。これには、**ProxyClass** クラスの**すべての**引数の getter および setter メソッドが含まれます。たとえば、上記のメソッドを指定すると、プラグインは設定クラスに以下の getter メソッドと setter メソッドを生成します。

```
public int  getId();
public void setId(int id);
public String getName();
public void   setName(String name);
```

ただし、以下の例のように、**id** 引数が異なるタイプとして複数回表示された場合はどうなるでしょうか。

```
public void doSomething(int id, String name);
public void doSomethingElse(int id, String name);
public String lookupByID(String id);
```

この場合、コードの生成に失敗します。これは、**int** を返す **getId** メソッドと、**String** を返す **getId** メソッドを同じスコープで定義できないためです。この問題に対する解決策は、**引数名の置換**を使用して、引数名と URI オプション名のマッピングをカスタマイズすることです。

構文

substitutions 要素は、以下のように 1 つ以上の **substitution** 子要素で定義できます。

```
<substitutions>
  <substitution>
    <method>MethodPattern</method>
    <argName>ArgumentNamePattern</argName>
    <argType>TypeNamePattern</argType>
    <replacement>SubstituteArgName</replacement>
    <replaceWithType>[true|false]</replaceWithType>
  </substitution>
  ...
</substitutions>
```

argType 要素と **replaceWithType** 要素は任意であり、省略できます。

スコープ

以下の抜粋で示されているように、**substitutions** 要素は、任意で **apis** 要素の子や **api** 要素の子として表示できます。

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <substitutions>...</substitutions>
    </api>
    <substitutions>...</substitutions>
    ...
  </apis>
</configuration>
```

以下のスコープで **substitutions** 要素を定義できます。

- **api 要素の子として**: **substitutions** は、**api** 要素で指定された API クラスにのみ適用されます。
- **apis 要素の子として**: **substitutions** は、デフォルトですべての API クラスに適用されますが、**api** レベルで上書きできます。

子要素

各 **substitution** 要素は以下の子要素で定義できます。

メソッド

Java API のメソッド名と一致する正規表現 (**java.util.regex** 構文) を指定します。

argName

正規表現 (**java.util.regex** 構文) を指定して、一致したメソッドの引数名とマッチさせます。通常、パターンにはキャプチャーグループが含まれます。

argType

(任意) 引数の型に一致する正規表現 (**java.util.regex** 構文) を指定します。**replaceWithType** オプションを **true** に設定した場合、通常はこの正規表現でキャプチャーグループを使用します。

replacement

method パターン、**argName** パターン、および (任意の) **argType** パターンに一致する場合、**replacement** 要素は代替の引数名を定義します (URI で使用)。代替テキストは、**argName** 正規表現パターンから取得した文字列を使用して作成できます (構文 **$1**、**$2**、**$3** を使用して、それぞれ第 1、第 2、第 3 のキャプチャーグループを挿入します)。または、**replaceWithType** オプションを **true** に設定した場合は、**argType** 正規表現パターンから取得した文字列を使用して代替テキストを作成することもできます。

replaceWithType

true の場合、代替テキストが **argType** 正規表現から取得した文字列を使用して作成されるように指定します。デフォルトは **false** です。

例

以下の置換例は、接尾辞 **Param** を引数名に追加して、**java.lang.String** タイプのすべての引数を変更します。

```
<substitutions>
  <substitution>
    <method>^.+$</method>
    <argName>^.+$</argName>
    <argType>java.lang.String</argType>
    <replacement>$1Param</replacement>
    <replaceWithType>false</replaceWithType>
  </substitution>
</substitutions>
```

たとえば、次のようなメソッドの署名があるとします。

```
public String greetUs(String name1, String name2);
```

このメソッドの引数は、エンドポイント URI でオプション **name1Param** および **name2Param** で指定されます。

47.6. 除外された引数

概要

Java 引数を URI オプションにマッピングする際、特定の引数を除外する必要がある場合があります。**camel-api-component-maven-plugin** プラグイン設定で **excludeConfigNames** 要素または **excludeConfigTypes** 要素を指定して、不要な引数をフィルターリングすることができます。

構文

excludeConfigNames 要素と **excludeConfigTypes** 要素は以下のように指定されます。

```
<excludeConfigNames>ArgumentNamePattern</excludeConfigNames>
<excludeConfigTypes>TypeNamePattern</excludeConfigTypes>
```

ここでの **ArgumentNamePattern** と **TypeNamePattern** は、それぞれ引数名と引数型に一致する正規表現です。

スコープ

以下の抜粋で示されているように、**excludeConfigNames** 要素と **excludeConfigTypes** 要素は、任意で **apis** 要素や **api** 要素の子とすることができます。

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <excludeConfigNames>...</excludeConfigNames>
      <excludeConfigTypes>...</excludeConfigTypes>
    </api>
    <excludeConfigNames>...</excludeConfigNames>
    <excludeConfigTypes>...</excludeConfigTypes>
    ...
  </apis>
</configuration>
```

以下のスコープで **excludeConfigNames** 要素および **excludeConfigTypes** 要素を定義できます。

- **api 要素の子として**: 除外は **api** 要素で指定された API クラスにのみ適用されます。
- **apis 要素の子として**: 除外はデフォルトですべての API クラスに適用されますが、**api** レベルで上書きできます。

要素

以下の要素を使用して、(URI オプションとして利用できないように引数を API マッピングから除外することができます。

excludeConfigNames

引数名の一致に基づいて、引数を除外するための正規表現 (**java.util.regex** 構文) を指定します。

excludeConfigTypes

引数型の一致に基づいて、引数を除外するための正規表現 (**java.util.regex** 構文) を指定します。

47.7. 追加オプション

概要

extraOptions オプションは通常より簡単なオプションを提供することで、複雑な API パラメーターを計算または非表示するために使用されます。たとえば、API メソッドには POJO オプションが使用される場合がありまが、これは URI 内の POJO の一部としてより簡単に提供することができます。コン

ポーネントは、パーツを追加オプションとして付与し、内部で POJO パラメーターを作成することでこれを行うことができます。これらの追加オプションの実装を完了するには、**EndpointConsumer** や **EndpointProducer** クラスで **interceptProperties** メソッドをオーバーライドする必要もあります (「プログラミングモデル」 を参照)。

構文

ExtraOptions 要素は、以下のように 1 つ以上の **extraOption** 子要素で定義できます。

```
<extraOptions>
  <extraOption>
    <type>TypeName</type>
    <name>OptionName</name>
  </extraOption>
</extraOptions>
```

TypeName は追加オプションの完全修飾型名であり、**OptionName** は追加 URI オプションの名前です。

スコープ

以下の抜粋で示されているように、**extraOptions** 要素は、任意で **apis** 要素の子や **api** 要素の子として表示できます。

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
      <extraOptions>...</extraOptions>
    </api>
    <extraOptions>...</extraOptions>
    ...
  </apis>
</configuration>
```

以下のスコープで **extraOptions** 要素を定義できます。

- **api 要素の子として**: **extraOptions** は **api** 要素で指定された API クラスにのみ適用されます。
- **apis 要素の子として**: **extraOptions** はデフォルトですべての API クラスに適用されますが、**api** レベルで上書きできます。

子要素

それぞれの **extraOptions** 要素は、以下の子要素で定義できます。

type
追加オプションの完全修飾型名を指定します。

name
エンドポイント URI に表示されるオプション名を指定します。

例

以下の例は、**java.util.list<String>** タイプの追加の URI オプションである **customOption** を定義します。

```
<extraOptions>
  <extraOption>
    <type>java.util.List<String></type>
    <name>customOption</name>
  </extraOption>
</extraOptions>
```

INDEX

T

U

W