



JBoss Enterprise SOA Platform 5

Smooks ユーザーガイド

Smooks を JBoss Enterprise SOA Platform と統合する開発者を対象
エディション 5.2.0

JBoss Enterprise SOA Platform 5 Smooks ユーザーガイド

Smooks を JBoss Enterprise SOA Platform と統合する開発者を対象
エディション 5.2.0

Red Hat Documentation Team

法律上の通知

Copyright © 2011 Red Hat, Inc..

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

本書は Smooks フレームワークのガイドおよび参考書です。Smooks は、CSV、EDI、Java オブジェクトなどの XML および非 XML データを処理する Java フレームワークです。ESB パイプライン内における Smooks のトランスフォメーションアクションに対して、JBoss Enterprise SOA Platform のサポートは限定されます。

目次

| | |
|--|-----|
| 第1章 概要 | 3 |
| 第2章 基本 | 5 |
| 第3章 入力データの消費 | 22 |
| 第4章 検証 | 42 |
| 第5章 出力データの生成 | 50 |
| 第6章 テンプレート | 69 |
| 第7章 出力データのリッチ化 | 77 |
| 第8章 GROOVY スクリプト | 83 |
| 第9章 ルーティング出力データ | 86 |
| 第10章 パフォーマンスの調整 | 88 |
| 第11章 テスト | 89 |
| 第12章 一般的なユースケース | 90 |
| 第13章 SMOOKS の拡張 | 101 |
| 第14章 APACHE CAMEL の統合 | 121 |
| 付録A JBOSS ENTERPRISE SOA PLATFORM に SMOOKS を統合する | 123 |
| 付録B GNU LESSER GENERAL PUBLIC LICENSE 2.1 | 127 |
| 付録C 改訂履歴 | 137 |

第1章 概要

本書では、**Smooks** のデータトランスフォメーションエンジンを **JBoss Enterprise SOA Platform** のアクションパイプラインに統合する方法について説明します。

Smooks は XML データと非 XML データを処理する Java フレームワークです (非 XML データ形式には CSV、EDI、Java ファイルなどが含まれます)。

重要

本書では Smooks プロジェクトのコード例を複数参考しています。例によって表されるユースケースはサポートされますが、コードサンプルは JBoss Enterprise SOA Platform には含まれておらず、テストやサポートもされていません。

Smooks プロジェクトの全コード例は http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.3_Examples を参照してください。

JBoss Enterprise SOA Platform には、Smooks のトランスフォーメーションやルーティングを実証する複数のクイックスタートサンプルプログラムが含まれています。詳細情報は各クイックスタートに含まれる **readme** ファイルを参照してください。

本項では、Smooks の主な機能について説明します。

トランスフォーメーション

XML、**CSV**、**EDI**、**Java** など、多くの形式の間でデータを変換する機能です。

Java バインディング

CSV、EDI、Java ファイルなどのデータソースより *Java* オブジェクトモデルを投入するために使用される機能です。投入されたオブジェクトモデルをそのままオブジェクトモデルとして使用する (トランスフォーメーションの結果として)、XML (または他の文字ベースの結果) が生成されるテンプレートとして使用することができます。

この機能は *仮想オブジェクトモデル* (型付けされたデータのマップやリスト) をサポートします。仮想オブジェクトモデルは、*ETL* (抽出、変換、ロード) とテンプレート機能の両方と共に使用できます。

巨大サイズのメッセージ処理

数キガバイトにおよぶメッセージを処理するため使用される機能です。Java メッセージサービス、ファイル、データベースなどのさまざまな宛先にメッセージの断片を分割、変換、ルーティングします。

メッセージのリッチ化

名前の通り、データベースやその他の外部ソースから提供された情報を用いてメッセージを「リッチ化」するために使用する機能です。

複合体メッセージの検証

ルールベース断片検証機能になります。

オブジェクトリレーショナルマッピングベースのメッセージ永続化

データベースにアクセスするため、*Java Persistence API* 対応の エンティティ永続化フレームワーク (**ibatis** や **Hibernate** など) を使用する機能です。データベース独自のクエリ言語か CRUD (作成、読み取り、更新、削除) メソッドのいずれかを使用して読み書きを行います。

カスタムデータアクセスオブジェクト (DAO) の CRUD メソッドを使用してデータベースにアクセスすることもできます。

組み合わせ

ETL (抽出、変換、ロード) 操作を実行するために使用される機能です。Smooks のトランスフォーメーション、ルーティング、永続化機能を使用してこの操作を行います。

第2章 基本

本章では、Smooks の基本的な仕組みについて説明します。

smooks-core は構造化データイベントストリームプロセッサです。そのコンポーネントは、カスタムのビジター論理をデータソースによって作成されたイベントストリームへ「フック」するよう設計されています。

Smooks の仕組みを理解するには、ビジター論理の概念を理解する必要があります。ビジターは簡単な Java ソースコードで、ターゲットのメッセージ断片上で特定のタスクを実行するためのものです (例えば、XSLT スタイルシートの適用を目的とします)。ビジター論理をサポートするには、**org.milyn.delivery.sax.SAXElementVisitor** インターフェースと **org.milyn.delivery.dom.DOMELEMENTVisitor** インターフェースのどちらかまたは両方を介して、SAX または DOM フィルターのいずれかを実装します。

一般的に、この機能はトランスフォーメーションソリューションを作成するために使用されます。トランスフォーメーションソリューションを作成するには、ソースメッセージによって作成されたイベントストリームを使用してビジター論理を実装し、結果を他の形式で作成します。しかし、他の用途で使用することも可能です。以下はその例になります。

- *Java バインディング*: ソースメッセージから Java オブジェクトモデルに投入する機能です。
- *メッセージの分割とルーティング*: ソースメッセージ上で複雑な分割やルーティングの操作を行う機能です。複数の形式のデータを複数の宛先に同時にルーティングできる機能も含まれます。
- *巨大サイズのメッセージ処理*: コードを大量に記述する必要なく、宣言的に巨大なメッセージを消費 (変換または分割とルーティング) する機能です。

2.1. 基本的な処理モデル

前述の通り、Smooks の基本的な目的はデータソースを取得し、ビジター論理が適用されるイベントストリームを取得したデータソースから生成することです (結果を EDI などの他の形式で作成するために実行します)。

多くのデータソースや結果形式がサポートされるため、複数のトランスフォーメーションタイプを使用できます。一般的な例は次の通りです。

- XML から XML
- XNL から Java
- Java から XML
- Java から Java
- EDI から XML
- EDI から Java
- Java から EDI
- CSV から XML

Smooks は ドキュメントオブジェクトモデル (DOM) と *Simple API for XML* (SAX) イベントモデルの両方をサポートします。ソースと結果をマッピングするためにこれらのイベントモデルを使用します。本書では、SAX イベントモデルを最も詳しく説明します。

名前の通り、SAX イベントモデルは XML ソースから生成できる階層的な SAX イベントをベースにしています。これには、**startElement** や **endElement** などが含まれます。

このイベントモデルの利点は、EDI や CSV、Java ファイルなど、構造化され階層的な別のデータソースへ比較的簡単に適用できることです。

通常、**visitBefore** や **visitAfter** が付けられたイベントが最も重要になります。下図はこれらの階層的本質を表しています。

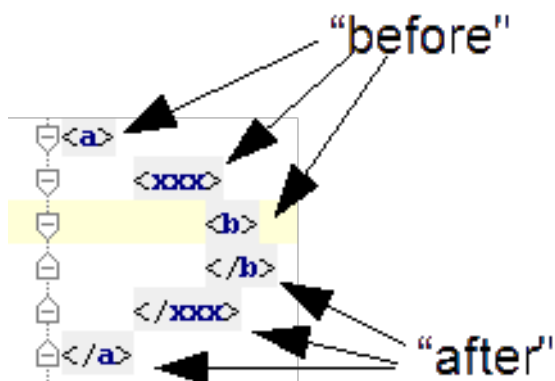


図2.1 visitBefore および visitAfter イベントの階層的本質

2.2. 簡単な例

ソースメッセージから作成された SAX イベントストリームを使用できるようにするには、最初に *SAXVisitor* インターフェースを 1 つ以上実装する必要があります。選択するインターフェースは、特定のケースで消費されるイベントによって異なります。



注記

この例では **ExecutionContext** 名を使用します。これは **BoundAttributeStore** クラスを拡張する公開インターフェースです。

この例は、全体のイベントストリームの特定要素で **visitBefore** イベントと **visitAfter** イベントへ論理を示す方法を表しています。このケースでは、ビジター論理が **<xxx>** 要素のイベントに示されています。



図2.2 ビジター論理の実装

ビジター実装は、イベントごとに1つのメソッド実装で構成されるため、大変シンプルです。この実装を <xxx> 要素の **visitBefore** イベントと **visitAfter** イベントへ示すには、次のように Smooks の設定を作成します。

このチュートリアルは、Smooks を使用して (**FreeMarker** を併用) 巨大なメッセージで XML から XML への変換を行う方法を表しています (このチュートリアルは文字ベースの変換の基本として使用することもできます)。



注記

FreeMarker はテンプレティングエンジンです。 **FreeMarker** を使用して、 **NodeModel** をテンプレート操作のドメインモデルとして作成し、使用することができます。 Smooks は断片ベースのテンプレートトランスフォーメーションを行う機能と、モデルを巨大メッセージに適用する機能を追加します。

ソース形式は次の通りです。

```
<order id='332'>
  <header>
    <customer number="123">Joe</customer>
  </header>
  <order-items>
    <order-item id='1'>
      <product>1</product>
      <quantity>2</quantity>
      <price>8.80</price>
    </order-item>

    <!-- etc etc -->

  </order-items>
</order>
```

ターゲット形式は次のようになります。

```
<salesorder>
```

```

<details>
  <orderid>332</orderid>
  <customer>
    <id>123</id>
    <name>Joe</name>
  </customer>
</details>
<itemList>
  <item>
    <id>1</id>
    <productId>1</productId>
    <quantity>2</quantity>
    <price>8.80</price>
  </item>

  <!-- etc etc -->

</itemList>
</salesorder>

```

使用する Smooks の設定は次の通りです。

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

  <!--
  Filter the message using the SAX Filter (i.e. not DOM, so no
  intermediate DOM for the "complete" message - there are "mini" DOMs
  for the NodeModels below)....
  -->
  <params>
    <param name="stream.filter.type">SAX</param>
    <param name="default.serialization.on">>false</param>
  </params>

  <!--
  Create 2 NodeModels. One high level model for the "order"
  (header etc) and then one per "order-item".

  These models are used in the FreeMarker templating resources
  defined below. You need to make sure you set the selector such
  that the total memory footprint is as low as possible. In this
  example, the "order" model will contain everything except the
  <order-item> data (the main bulk of data in the message). The
  "order-item" model only contains the current <order-item> data
  (i.e. there's max 1 order-item in memory at any one time).
  -->
  <resource-config selector="order,order-item">
    <resource>org.milyn.delivery.DomModelCreator</resource>
  </resource-config>

  <!--
  Apply the first part of the template when we reach the start
  of the <order-items> element. Apply the second part when we

```

reach the end.

Note the `<?TEMPLATE-SPLIT-PI?>` Processing Instruction in the template. This tells Smooks where to split the template, resulting in the `order-items` being inserted at this point.

-->

```
<ftl:freemarker applyOnElement="order-items">
```

```
  <ftl:template><!--<salesorder>
```

```
<details>
```

```
  <orderid>${order.@id}</orderid>
```

```
  <customer>
```

```
    <id>${order.header.customer.@number}</id>
```

```
    <name>${order.header.customer}</name>
```

```
  </customer>
```

```
</details>
```

```
<itemList>
```

```
  <?TEMPLATE-SPLIT-PI?>
```

```
</itemList>
```

```
</salesorder>--></ftl:template>
```

```
</ftl:freemarker>
```

```
<!--
```

Output the `<order-items>` elements. This will appear in the output message where the `<?TEMPLATE-SPLIT-PI?>` token appears in the `order-items` template.

-->

```
<ftl:freemarker applyOnElement="order-item">
```

```
  <ftl:template><!--          <item>
```

```
    <id>${.vars["order-item"].@id}</id>
```

```
    <productId>${.vars["order-item"].product}</productId>
```

```
    <quantity>${.vars["order-item"].quantity}</quantity>
```

```
    <price>${.vars["order-item"].price}</price>
```

```
  </item>
```

```
  --></ftl:template>
```

```
</ftl:freemarker>
```

```
</smooks-resource-list>
```

ここで次を実行する必要があります。

```
Smooks smooks = new Smooks("smooks-config.xml");
try {
    smooks.filterSource(new StreamSource(new FileInputStream("input-
message.xml")), new StreamResult(System.out));
} finally {
    smooks.close();
}
```

結果として、XML から XML へ変換が行われます。

この例では、「低レベル」の Smooks プログラミングモデルの仕組みが実証されます。Smooks には機能的なモジュールの一部が同梱されているため、Smooks に対して大量の Java コードを書く必要はありません。これらのモジュールは、カートリッジと呼ばれる一般的なユースケースの多くに対応します。

2.3. SMOOKS のリソース

Smooks は大変一般的な処理モジュールを使用します。このモジュールは、すべてのビジター論理を イベントセクタへ適用される *Smooks* リソース (**SmooksResourceConfiguration**) として見なします。これは、Smooks Core やそのアーキテクチャーを維持する観点から見ると道理にかなっていません。しかし、設定ではすべてが同じように見えるため、有用性の観点から見ると一般的すぎる場合があります。そのため、Smooks v1.1 には *拡張可能設定モデル* が導入されました。このモデルにより、独自の専用 XSD 名前空間を使用するよう特定のリソースタイプ (Java Bean バインディングの設定や FreeMarker テンプレートの設定など) を指定することができます。

例2.1 Java バインディングリソース

```
<jb:bean beanId="lineOrder" class="example.trgmodel.LineOrder"
  createOnElement="example.srcmodel.Order">
  <jb:wiring property="lineItems" beanIdRef="lineItems" />
  <jb:value property="customerId" data="header/customerNumber" />
  <jb:value property="customerName" data="header/customerName" />
</jb:bean>
```

例2.2 FreeMarker テンプレートリソース

```
<ftl:freemarker applyOnElement="order-item">
  <ftl:template><!-- <item>
    <id>${.vars["order-item"].@id}</id>
    <productId>${.vars["order-item"].product}</productId>
    <quantity>${.vars["order-item"].quantity}</quantity>
    <price>${.vars["order-item"].price}</price>
  </item>-->
</ftl:template>
</ftl:freemarker>
```

これらの例での重要な点は次の通りです。

- 設定は「強くタイプ化」され、ドメインに固有します。これにより、読み取りが楽になります。
- 設定は XSD ベースです。これにより、統合開発環境を使用している場合に自動補間サポートが提供されます。
- リソースタイプの実際のハンドラを定義する必要はありません (Java バインディングの **BeanPopulator** クラスなど)。

2.3.1. セレクター

Smooks リソースセレクタはツールの大変重要な一部分です。セレクタは、どのメッセージ断片にビジター論理を適用するか Smooks に指示します。また、非ビジター論理の簡単なルックアップ値としても機能します。

リソースがビジター実装である場合 (**<jb:bean>** や **<ftl:freemarker>** など)、Smooks はリソースセレクタを XPath セレクタのように扱います。

次の点に注意してください。

1. XPath 式が適用される順番は通常の順番の逆になります。Smooks はメッセージの root 要素からではなく、ターゲットの fragment 要素から逆順に適用します。
2. XPath の仕様は完全サポートされていません。リソースセクタは次の XPath 構文をサポートします。

- リテラル値および数値両方の `text()` および属性 (`@x` など) の値セクタ。

例: `"a/b[text() = 'abc']"`、`"a/b[text() = 123]"`、`"a/b[@id = 'abc']"`、`"a/b[@id = 123]"`。

- `text()` は式の最後のセクタ手順でのみサポートされます。たとえば、`"a/b[text() = 'abc']"` は許可されますが `"a/b[text() = 'abc']/c"` は許可されません。
- `text()` は **SAXVisitAfter** インターフェースのみを使用する **SAXVisitor** 実装でのみサポートされます。**SAXVisitor** が **SAXVisitBefore** インターフェースまたは **SAXVisitChildren** インターフェースを実装すると、エラーが発生します。
- **And** および **Or** 論理演算。例: `"a/b[text() = 'abc' and @id = 123]"`、`"a/b[text() = 'abc' or @id = 123]"`
- 要素と属性両方の名前空間。例: `"a:order/b:address[@b:city = 'NY']"`
- 名前空間の **prefix-to-URI** マッピングを定義しなければなりません。定義しないと、設定エラーが発生します。
- 次の演算もサポートされます。
 - `=` (等号)
 - `!=` (不等号)
 - `<` (小なり記号)
 - `>` (大なり記号)
- 索引セクター。 `"a/b[3]"` など。

2.3.2. 名前空間の宣言

コア設定名前空間を介して名前空間のプレフィックスから URI へのマッピングを設定します。名前空間セクターを解決する時にこれらの設定が使用できるようになります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

  <core:namespaces>
    <core:namespace prefix="a" uri="http://a"/>
    <core:namespace prefix="b" uri="http://b"/>
    <core:namespace prefix="c" uri="http://c"/>
    <core:namespace prefix="d" uri="http://d"/>
  </core:namespaces>
</smooks-resource-list>
```

```
<resource-config selector="c:item[@c:code = '8655']/d:units[text() =  
1]">  
  <resource>com.acme.visitors.MyCustomVisitorImpl</resource>  
</resource-config>  
  
</smooks-resource-list>
```

2.4. カートリッジ

ソリューションを迅速に実装できるようにするため、Smooks にはすぐ使用できるプレビルドのビジター論理が含まれています。ビジター論理はカートリッジと呼ばれるモジュールに組み合わされます。

カートリッジは再使用できるコンテンツハンドラーが含まれる *Java* アーカイブ(JAR) ファイルです。独自の新しいカートリッジを作成して **smooks-core** の基本機能を拡張することもできます。Smooks の各カートリッジは、トランスフォーマーメーションプロセスまたは XML 分析の特定形式のいずれかにすぐ対応できるサポートを提供します。

Smooks に含まれる全カートリッジの一覧は次の通りです。

- Calc: "milyn-smooks-calc"
- CSV: "milyn-smooks-csv"
- 固定長リーダー: "milyn-smooks-fixed-length"
- EDI: "milyn-smooks-edl"
- Javabean: "milyn-smooks-javabean"
- JSON: "milyn-smooks-json"
- ルーティング: "milyn-smooks-routing"
- テンプレート: "milyn-smooks-templating"
- CSS: "milyn-smooks-css"
- Servlet: "milyn-smooks-servlet"
- 永続化: "milyn-smooks-persistence"
- バリデーション: "milyn-smooks-validation"

2.5. フィルタリング処理の選択

本項では、Smooks がフィルタリング処理を選択する方法について説明します。

- すべてのビジターリソースが DOM ビジターインターフェース (**DOMElementVisitor** または **SerializationUnit**) のみを実装する場合、自動的に DOM 処理モデルが選択されます。
- すべてのビジターリソースが SAX ビジターインターフェース (**SAXElementVisitor**) のみを実装する場合、自動的に SAX 処理モデルが選択されます。
- ビジターリソースが DOM インターフェースと SAX インターフェースの両方を実装する場合、Smooks のリソース設定ファイルに SAX が指定されない限り、デフォルトでは DOM 処理モデルが選択されます。



注記

Smooks 1.3 では、`<core:filterSettings type="SAX" />` を使用して行われます。



注記

このコンテキストでは、**readers** などの *非要素* ビジタリソース はビジターリソースに含まれません。

例2.3 Smooks 1.3 でフィルタータイプを SAX に設定する

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

  <core:filterSettings type="SAX" />

</smooks-resource-list>
```

2.5.1. DOM モデルと SAX モデルの混合

ドキュメントオブジェクトモデルでは、ノードのトラバーサルやその他機能が使用できるため、コーディングのレベルでは SAX よりも使用が簡単です。また、ドキュメントオブジェクトを使用すると、DOM 構造のビルトインサポートを持つ **FreeMarker** や **Groovy** など既存のスクリプトエンジンやテンプレートエンジンを使用することもできます。

ただし、DOM にはメモリーによって制約される難点もあります。これにより、巨大なメッセージの処理能力が大幅に制限されます。

Smooks の **DomModelCreator** ビジタークラスを使用して 2 つのモデルを混合することが可能です。このビジタークラスを SAX フィルターと共に使用すると、このビジターは visited 要素から DOM 断片を構築します。そのため、ストリーミング環境内で DOM ユーティリティを使用することができます。

複数のモデルがネストされている場合、外部モデルに内部モデルのデータが含まれることはありません。よって、2 つのモデル内に同じ断片が共存することはありません。次のサンプルメッセージにこの原理が示されています。

```
<order id="332">
  <header>
    <customer number="123">Joe</customer>
  </header>
  <order-items>
    <order-item id='1'>
      <product>1</product>
      <quantity>2</quantity>
      <price>8.80</price>
    </order-item>
    <order-item id='2'>
      <product>2</product>
      <quantity>2</quantity>
      <price>8.80</price>
    </order-item>
  </order-items>
</order>
```

```

        <order-item id='3'>
            <product>3</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
    </order-items>
</order>

```

Smooks 内で **DomModelCreator** を設定して、次のコード例のように **order** メッセージ断片と **order-item** メッセージ断片の両方にモデルを作成することができます。

```

<resource-config selector="order,order-item">
    <resource>org.milyn.delivery.DomModelCreator</resource>
</resource-config>

```

このケースでは、**order** には **order-item** のモデルデータが含まれることはありません (**order-item** 要素は **order** 要素内にネストされるからです)。**order** のインメモリモデルは次のようになります。

```

<order id='332'>
    <header>
        <customer number="123">Joe</customer>
    </header>
    <order-items />
</order>

```

新しいモデルは以前のモデルを上書きするため、メモリ内に複数の **order-item** モデルが存在することはありません。使用されるメモリの容量が最低限になるようソフトウェアはこのように設計されています。

これまでの説明をまとめると、Smooks 処理モデルはイベント主導となります。これは、フィルタリングやストリーミング処理の異なる点に適用するビジター論理を「フック」することができるという意味です。ビジター論理はメッセージ独自の内容を使用して適用されます。そのため、DOM と SAX の混合処理モデルを使用することもできます。

DOM と SAX の混合処理モデルについての詳細は、次の Web サイトを参照してください。

- Groovy スクリプト: <http://www.smooks.org/mediawiki/index.php?title=V1.3:groovy>
- FreeMarker テンプレート: <http://www.smooks.org/mediawiki/index.php?title=V1.3:xml-to-xml>

2.6. BEAN コンテキスト

Bean コンテキストは、フィルタリング処理中に Smooks がアクセスできるオブジェクトのコンテナです。実行コンテキストごとに 1 つの Bean コンテキストが作成されます (**Smooks.filterSource** 操作より)。カートリッジによって作成される各 Bean は、この *BeanId* 下のコンテキストにファイルされます。

Smooks.filterSource プロセスの最後に Bean コンテキストの内容を返したい場合は、**Smooks.filterSource** メソッドへの呼び出しに **org.milyn.delivery.java.JavaResult** オブジェクトが含まれるようにします。次の例はこの方法を示しています。

```

//Get the data to filter
StreamSource source = new

```

```

StreamSource(getClass().getResourceAsStream("data.xml"));

//Create a Smooks instance (cachable)
Smooks smooks = new Smooks("smooks-config.xml");

//Create the JavaResult, which will contain the filter result after
filtering
JavaResult result = new JavaResult();

//Filter the data from the source, putting the result into the JavaResult
smooks.filterSource(source, result);

//Getting the Order bean which was created by the JavaBean cartridge
Order order = (Order)result.getBean("order");

```

ランタイム時に Bean コンテキスト Bean にアクセスする必要がある場合 (カスタムビジター実装より)、**BeanContext** オブジェクトを介して接続します。このオブジェクトは、**getBeanContext()** メソッドを介して **ExecutionContext** より読み出すことができます。

BeanContext よりオブジェクトを追加または読み出しする時、最初に **BeanIdStore** より **BeanId** オブジェクトを読み出すようにしてください (**BeanId** オブジェクトは文字列キーよりも優れたパフォーマンスを実現できる特別なキーです。文字列キーもサポートされます)。

getBeanIdStore() メソッドを介して **ApplicationContext** より **BeanIdStore** を読み出さなければなりません。**BeanId** オブジェクトを作成するには、**register("BeanId name")** メソッドを呼び出します (**BeanId** がすでに登録されている場合は、**getBeanId("BeanId name")** メソッドを呼び出して読み出すことが可能です)。

BeanId オブジェクトは **ApplicationContext** がスコープ指定されたオブジェクトです。これらのオブジェクトをカスタムビジター実装の初期化メソッドに登録し、プロパティとしてビジターオブジェクトに置きます。その後、これらのオブジェクトは **visitBefore** および **visitAfter** メソッドで使用できます (**BeanId** オブジェクトおよび **BeanIdStore** はスレッドセーフです)。

Bean コンテキストの事前インストールされた Bean でランタイム時に使用可能なものは次の通りです。

- **PUUID**: UniqueId Bean です。この Bean は、**ExecutionContext** のフィルタリング対して一意な識別子を提供します。
- **PTIME**: Time Bean です。**ExecutionContext** のフィルタリングに対して、時間ベースのデータを提供します。

以下の例は、**FreeMarker** テンプレートで上記の各 Bean を使用方法を示しています。

- **ExecutionContext** の一意な ID (メッセージをフィルタリング): **\$PUUID.execContext**
- ランダムで一意な ID: **\$PUUID.random**
- メッセージのフィルタリングを開始する時間 (ミリ秒単位): **\$PTIME.startMillis**
- メッセージのフィルタリングを開始する時間 (ナノ秒単位): **\$PTIME.startNanos**
- メッセージのフィルタリングを開始する時間 (日付): **\$PTIME.startDate**
- 現在の時間 (ミリ秒単位): **\$PTIME.nowMillis**

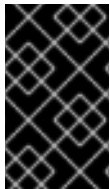
- 現在の時間 (ナノ秒単位): `$PTIME.nowNanos`
- 現在の時間 (日付): `$PTIME.nowDate`

2.6.1. 複数の出力や結果

本項では、Smooks でフィルタリング処理より出力を生成する方法について説明します。

Smooks は次の方法で出力を表示できます。

- 「結果内」のインスタンスより: `Smooks.filterSource` メソッドへ渡された 結果インスタンス内に返されます。
- フィルタリング処理中: フィルタリング処理中に外部エンドポイント (ESB サービス、ファイル、JMS の宛先、データベースなど) へ生成され送信 された出力を介して行われます。メッセージ断片のイベントによって外部エンドポイントへの自動ルーティングが引き起こされます。



重要

Smooks はメッセージストリームの単一のフィルタリングパスより、上記 2 つの方法の両方またはいずれかを用いて出力を生成することができます。複数の出力を生成するためにメッセージストリームを複数回フィルター処理する必要はありません。

2.6.2. 「結果内」のインスタンス

API が公開する通り、複数の結果インスタンスを Smooks に提供できます。

```
public void filterSource(Source source, Result... results) throws
SmooksException
```

Smooks は 標準的な **JDK StreamResult** や **DOMResult** 結果タイプ、および次のような特殊なタイプと動作します。

- **JavaResult**: この結果タイプを使用して Smooks Java Bean コンテキストの内容をキャプチャします。
- **ValidationResult**: この結果タイプを使用して出力をキャプチャします。
- Simple Result タイプ: テストを書き込む時に使用します。 **StringWriter** をラッピングする **StreamResult** 拡張です。

これが Smooks のフィルタリング処理より出力をキャプチャする最も一般的な方法となります。

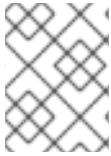


注記

Smooks は同じタイプである複数の結果インスタンスから結果データをキャプチャすることをサポートしていません。例えば、`Smooks.filterSource` メソッド呼び出しで複数の **StreamResult** インスタンスを指定できますが、Smooks はこれらの **StreamResult** インスタンスを 1 つだけ (最初のインスタンス) 出力します。

2.6.3. StreamResults と DOMResults

これらの結果タイプは特別に処理されます。Smooks がメッセージソースを処理すると、イベントのストリームを作成します。**StreamResult** または **DOMResult** が **Smooks.filterSource** 呼び出しへ提供されると、デフォルトでは、イベントストリーム (ソースによって作成された) を提供された **StreamResult** または **DOMResult** へ XML としてシリアルライズします (シリアルライズする前にビクター論理をイベントストリームへ適用できます)。



注記

これが 1 つの入力で 1 つの xml を出力する、標準の文字ベーストランスフォーメーションを実行するために使用されるメカニズムとなります。

2.6.4. フィルタリング処理中

Smooks は **Smooks.filterSource** プロセス中に異なるタイプの出力を生成することもできます (メッセージイベントストリームをフィルタリング処理し、メッセージの最後に到達する前)。他プロセスによる実行のため、エンドポイントの異なるタイプへメッセージ断片を分割およびルーティングするために使用される場合が、この例となります。

Smooks は次のような理由で、メッセージデータを単に一括処理せず、メッセージ全体をフィルタリングした後にすべての出力を生成します。

- パフォーマンスへの影響
- これにより、メッセージイベントストリームを活用し、断片のトランスフォーメーションやルーティング操作をトリガーできます。

たとえば、決定した基準を基に、異なる形式で異なる部署に分割およびルーティングする必要がある注文商品が数十万個注文メッセージに記載されているとしましょう。このような大きさのメッセージを処理する唯一の方法がプロセスのストリーミングになります。

2.7. SMOOKS 実行処理の確認

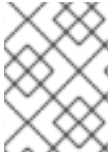
Smooks がメッセージをフィルタリングすると、キャプチャ可能でプログラムによる分析も可能なイベントを **パブリッシュ** します。Smooks より実行レポートを入手する最も簡単な方法は、**ExecutionContext** クラスを設定してレポートを生成する方法です (Smooks を設定して **HtmlReportGenerator** クラスより HTML レポートを生成することも可能です)。

次のコード例は、Smooks を設定して HTML レポートを生成する方法を示しています。

```
Smooks smooks = new Smooks("/smooks/smooks-transform-x.xml");
ExecutionContext execContext = smooks.createExecutionContext();

execContext.setEventListener(new HtmlReportGenerator("/tmp/smooks-report.html"));
smooks.filterSource(execContext, new StreamSource(inputStream), new StreamResult(outputStream));
```

HtmlReportGenerator は、開発作業に従事しているユーザーに便利なツールです。現在、IDE ベースのデバッガーに最も近い Smooks のツールです (「正式」なデバッガーは今後のリリースに同梱される予定です)。

**注記**

レポートの例は、Web ページ <http://www.milyn.org/docs/smooks-report/report.html> を参照してください。

**注記**

代わりに、カスタムの **ExecutionEventListener** 実装を作成することも可能です。

2.8. フィルタリング処理の終了

メッセージの最後に到達する前にフィルタリング処理を終了する必要がある場合があります。途中でフィルタリング処理を終了するには、**<core:terminate>** 設定を Smooks 設定に追加します。この設定は SAX フィルタでのみ動作するため、DOM に追加しても意味がありません。

メッセージの顧客断片の最後にフィルタリングを終了する設定の例は次の通りです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

  <!-- Visitors... -->
  <core:terminate onElement="customer" />

</smooks-resource-list>
```

デフォルトでは、ターゲットの断片の最後に終了します (**visitAfter** イベント)。最初に終了するには (**visitBefore** イベント)、次のコードを使用します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

  <!-- Visitors... -->

  <core:terminate onElement="customer" terminateBefore="true" />

</smooks-resource-list>
```

2.9. グローバル設定

グローバル設定とは、1 回のみで設定内のすべてのリソースがその設定を使用できるオプションのことです。

Smooks は、**default properties** と **global parameters** の 2 種類のグローバル設定をサポートします。

デフォルトプロパティ

デフォルトプロパティは **<resource-config>** 属性のデフォルト値を指定します。対応する **<resource-config>** が属性の値を指定しない場合、これらのプロパティが **SmooksResourceConfiguration** クラスへ自動的に適用されます。

グローバルパラメータ

<param> 要素は **<resource-config>** ごとに指定することができます。これらのパラメータ値は、**SmooksResourceConfiguration** よりランタイム時に取得できますが、できない場合は **@ConfigParam** アノテーションより挿入されます。

グローバル設定パラメータは一ヶ所で定義されます。各ランタイムコンポーネントは **ExecutionContext** を使用してこれらのパラメータにアクセスできます。

2.9.1. グローバル設定パラメータ

グローバルパラメータは root 要素に指定されず、リソースへ自動的に適用されない点がデフォルトのプロパティと異なります。

グローバルパラメータは **<params>** 要素に指定されます。

```
<params>
  <param name="xyz.param1">param1-val</param>
</params>
```

グローバル設定パラメータは **ExecutionContext** よりアクセス可能です。

```
public void visitAfter(
    final Element element, final ExecutionContext executionContext)
    throws SmooksException
{
    String param1 = executionContext.getConfigParameter(
        "xyz.param1", "defaultValueABC");

    ....
}
```

2.9.2. デフォルトプロパティ

デフォルトプロパティとは、Smooks 設定の root 要素に設定でき、その後 **smooks-conf.xml** ファイルにあるすべてのリソース設定へ適用できるプロパティのことです。

たとえば、すべてのリソース設定のセレクト値が同じである場合、すべてのリソース設定にセレクトを指定する代わりに **default-selector=order** を指定することができます。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd"
  default-selector="order">

  <resource-config>
    <resource>com.acme.VisitorA</resource>
    ...
  </resource-config>

  <resource-config>
    <resource>com.acme.VisitorB</resource>
    ...
  </resource-config>
```

<smooks-resource-list>

使用できるオプションは次の通りです。

default-selector

セレクトが定義されていない場合、このセレクトが Smooks 設定ファイルにあるすべての resource-config 要素に適用されます。

default-selector-namespace

デフォルトのセレクト名前空間です。他の名前空間が定義されていない場合に使用されます。

default-target-profile

ターゲットプロファイルが定義されていない場合、Smooks 設定ファイルにあるすべてのリソースに適用されるデフォルトのターゲットプロファイルになります。

default-condition-ref

条件識別子によってグローバル条件を参照します。この条件は、グローバルに定義された条件を参照しない空の条件要素 (<condition/>) を定義するリソースに適用されます。

2.10. フィルター設定

フィルターオプションを設定するには、smooks-core 設定の名前空間を使用します。

例2.4 設定例

```

<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">
  <core:filterSettings type="SAX" defaultSerialization="true"
    terminateOnException="true" readerPoolSize="3" closeSource="true"
    closeResult="true" rewriteEntities="true" />

    .. Other visitor configs etc...

</smooks-resource-list>

```

type

SAX と DOM のどちらか (デフォルトは DOM) から使用される処理モデルのタイプを決定します。

defaultSerialization

デフォルトのシリアル化を有効にするかを決定します。デフォルトは **true** です。有効にすると、**Smooks.filterSource** メソッドに提供される結果オブジェクトにある **StreamResult** (または **DOMResult**) を探そう Smooks に伝え、デフォルトではその結果へのイベントをすべてシリアル化します。

この挙動はグローバル設定パラメータを介して無効にできます。また、結果ライターを所有する断片 (SAX フィルタリングを使用する場合) または DOM を変更する断片 (DOM フィルタリングを使用する場合) ヘビジター実装をターゲットして断片ごとに上書きすることも可能です。

terminateOnException

例外によって処理が停止されるべきであるかどうかを決定します。デフォルトの設定は **true** になります。

closeSource

Smooks.filterSource メソッドに渡されたソースインスタンスストリームを閉じます。デフォルトは **true** です。この場合の例外は閉じられることがない **System.in** になります。

closeResult

Smooks.filterSource メソッドに渡された結果ストリームを閉じます。デフォルトは **true** です。この場合の例外は閉じられることがない **System.out** と **System.err** になります。

rewriteEntities

これを使用して XML を読み書きする時に (デフォルトのシリアライゼーション) XML エンティティを再書き込みします。

readerPoolSize

リーダーのプールサイズを設定します。リーダー実装によっては作成が高価になることがあります。特に小さいメッセージを大量に処理する場合、リーダーインスタンスをプールすると (再使用) パフォーマンスを大幅に改善することができます。この設定のデフォルト値は **0** (プールされず、各メッセージに対して新しいリーダーインスタンスが作成されます) になります。

アプリケーションのスレッディングモデルに従ってこれを設定します。

第3章 入力データの消費

Smooks は ストリームリーダー を使用してソースメッセージのデータストリームより SAX イベントのストリームを生成します。ストリームリーダーは **XMLReader** インターフェース (または **SmooksXMLReader** インターフェース) を実装するクラスです。

デフォルトでは、Smooks は XMLReader (**XMLReaderFactory.createXMLReader()**) を使用しますが、XML 以外のデータソースを読み取るよう設定するには、専用の XML リーダーを設定します。

リーダーを設定してハンドラ、機能、 およびパラメータのセットを使用することも可能です。完全な設定例は次の通りです。

```
<reader class="com.acme.ZZZZReader">
  <handlers>
    <handler class="com.X" />
    <handler class="com.Y" />
  </handlers>
  <features>
    <setOn feature="http://a" />
    <setOn feature="http://b" />
    <setOff feature="http://c" />
    <setOff feature="http://d" />
  </features>
  <params>
    <param name="param1">val1</param>
    <param name="param2">val2</param>
  </params>
</reader>
```

デフォルトでは、Smooks は XML データを読み取ります。デフォルトの XML リーダーに機能を設定するには、設定のクラス名を省略します。

```
<reader>
  <features>
    <setOn feature="http://a" />
    <setOn feature="http://b" />
    <setOff feature="http://c" />
    <setOff feature="http://d" />
  </features>
</reader>
```

CSV リーダーを設定するには、<http://www.milyn.org/xsd/smooks/csv-1.2.xsd> 設定名前空間を使用します。

基本的な設定は次の通りです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

  <!--
  Configure the CSV to parse the message into a stream of SAX events.
  -->
  <csv:reader fields="firstname,lastname,gender,age,country"
```

```
separator="|" quote="'" skipLines="1" />
</smooks-resource-list>
```

上記の設定は次のような形式のイベントストリームを生成します。

```
<csv-set>
  <csv-record>
    <firstname>Tom</firstname>
    <lastname>Fennelly</lastname>
    <gender>Male</gender>
    <age>21</age>
    <country>Ireland</country>
  </csv-record>
  <csv-record>
    <firstname>Tom</firstname>
    <lastname>Fennelly</lastname>
    <gender>Male</gender>
    <age>21</age>
    <country>Ireland</country>
  </csv-record>
</csv-set>
```

名前のコンマ区切りリストを `fields` 属性に使用してフィールドを定義します。

フィールド名は、以下の XML 要素名と同じ命名ルールに従う必要があります。

- アルファベット、数字、およびその他の文字を使用できます。
- 数字または句読文字で始まる名前は使用できません。
- 「xml」 (または XML や Xml など) で始まる名前は使用できません。
- 空白文字は使用できません。

`rootElementName` および `recordElementName` 属性を設定すると、`csv-set` および `csv-record` 要素名を編集することが可能です。これらの名前にも同じルールが適用されます。

フィールドごとに文字列操作の関数を定義することが可能です。これらの関数は、データが SAX イベントに変換される前に実行されます。フィールド名の後に定義し、2 つの関数を疑問符で区切ります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

  <csv:reader fields="lastname?trim.capitalize,country?upper_case" />

</smooks-resource-list>
```

CSV レコードのフィールドを無視するよう Smooks を設定するには、**\$ignore\$** トークンをフィールドの設定値として指定します。無視するフィールド数を指定するには、**\$ignore\$** トークンの後ろに値を追加します(**\$ignore\$3** の場合、後続の 3 フィールドが無視されます)。 **\$ignore\$+** を指定すると、CSV レコードの最後まですべてのフィールドが無視されます。

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

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,$ignore$2,age,$ignore$+" />

</smooks-resource-list>
```

CSV レコードを Java オブジェクトにバインドするのは比較的簡単です。

次のような CSV レコードがあるとします。

```
Tom,Fennelly,Male,4,Ireland
Mike,Fennelly,Male,2,Ireland
```

このコードを使用して、ある個人へレコードをバインドします。

```
public class Person {
    private String firstname;
    private String lastname;
    private String country;
    private Gender gender;
    private int age;
}

public enum Gender {
    Male,
    Female;
}
```

この設定を使用します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <!-- Note how the field names match the property names on the
        Person class. -->
        <csv:listBinding BeanId="people" class="org.milyn.csv.Person" />
    </csv:reader>

</smooks-resource-list>
```

この設定を実行するには、次のコードを使用します。

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

List<Person> people = (List<Person>) result.getBean("people");

Smooks also supports creation of Maps from the CSV record set:

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

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <csv:mapBinding BeanId="people" class="org.milyn.csv.Person"
keyField="firstname" />
    </csv:reader>

</smooks-resource-list>
```

上記の設定は person インスタンスのマップを生成します。各個人の firstname 値へ入力されます。これは次のように実行されます。

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

Map<String, Person> people = (Map<String, Person>)
result.getBean("people");

Person tom = people.get("Tom");
Person mike = people.get("Mike");
```

仮想モデルもサポートされるため、class 属性を **java.util.Map** として定義し、CSV フィールドの値をマップインスタンスにバインドできます。その後、マップインスタンスはリストまたはマップに追加されます。

プログラムを用いて CSV リーダーを設定することも可能です。これには複数の方法を使用できます。

次のコードは、個人のレコードセットを読み取るよう設定された CSV リーダーを用いて Smooks を設定し、レコードを person インスタンスのリストへバインドします。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
CSVReaderConfigurator("firstname,lastname,gender,age,country")
    .setBinding(new CSVBinding("people", Person.class,
CSVBindingType.LIST)));

JavaResult result = new JavaResult();
smooks.filterSource(new StreamSource(csvReader), result);

List<Person> people = (List<Person>) result.getBean("people");
```

Java バインディングの設定は任意です。代わりに、プログラムを用いて Smooks インスタンスを設定し、他のビジター実装を使用して CSV レコードセットを処理することが可能です。

CSV レコードのデータを反映する Java タイプのリストまたはマップへ CSV レコードをバインディングしたい場合、**CSVListBinder** または **CSVMapBinder** クラスを使用できます。

```
// Note: The binder instance should be cached and reused...
CSVListBinder binder = new
CSVListBinder("firstname,lastname,gender,age,country", Person.class);
```

```
List<Person> people = binder.bind(csvStream);

CSVMapBinder:

// Note: The binder instance should be cached and reused...
CSVMapBinder binder = new
CSVMapBinder("firstname,lastname,gender,age,country", Person.class,
"firstname");

Map<String, Person> people = binder.bind(csvStream);
```

バインディング処理を更に制御する必要がある場合、低レベル API の使用に戻ります。

<http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd> 設定名前空間より、固定長のリーダーを設定します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

    <!--
        Configure the Fixed length to parse the message into a stream of SAX
        events.
    -->
    <fl:reader
fields="firstname[10],lastname[10],gender[1],age[2],country[2]"
skipLines="1" />

</smooks-resource-list>
```

入力ファイルの例は次の通りです。

```
#HEADER
Tom      Fennelly  M 21 IE
Maurice  Zeijen   M 27 NL
```

生成されるイベントストリームは次の通りです。

```
<set>
  <record>
    <firstname>Tom      </firstname>
    <lastname>Fennelly  </lastname>
    <gender>M</gender>
    <age> 21</age>
    <country>IE</country>
  </record>
  <record>
    <firstname>Maurice  </firstname>
    <lastname>Zeijen    </lastname>
    <gender>M</gender>
    <age>27</age>
    <country>NL</country>
  </record>
</set>
```

前の例と同様に、文字列操作関数を定義できます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

  <!--
    Configure the fixed length reader to parse the message into a stream
    of SAX events.
  -->
  <fl:reader fields="firstname[10]?
trim,lastname[10]trim.capitalize,gender[1],age[2],country[2]"
skipLines="1" />

</smooks-resource-list>
```

また、前の例と同様にフィールドを無視することも可能です。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

  <fl:reader fields="firstname,$ignore$[2],age,$ignore$[10]" />

</smooks-resource-list>
```

固定長レコードの一部は次のようになります。

```
Tom      Fennelly  M 21 IE
Maurice  Zeijen   M 27 NL
```

これらを次のように個人へバインドします。

```
public class Person {
    private String firstname;
    private String lastname;
    private String country;
    private String gender;
    private int age;
}
```

この設定を使用します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

  <fl:reader fields="firstname[10]?trim,lastname[10]?
trim,gender[1],age[3]?trim,country[2]">
    <!-- Note how the field names match the property names on the
    Person class. -->
    <fl:listBinding BeanId="people"
class="org.milyn.fixedlength.Person" />

  </fl:reader>
</smooks-resource-list>
```

```

        </fl:reader>
    </smooks-resource-list>

```

次のように実行します。

```

Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(fixedLengthStream), result);

List<Person> people = (List<Person>) result.getBean("people");

```

固定長レコードセットよりマップを作成することも可能です。

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

    <fl:reader fields="firstname[10]?trim,lastname[10]?
trim,gender[1],age[3]?trim,country[2]">
        <fl:mapBinding BeanId="people"
class="org.milyn.fixedlength.Person" keyField="firstname" />
    </fl:reader>

</smooks-resource-list>

```

次のように作成された person インスタンスのマップを実行します。

```

Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(fixedLengthStream), result);

Map<String, Person> people = (Map<String, Person>)
result.getBean("people");

Person tom = people.get("Tom");
Person mike = people.get("Maurice");

```

仮想モデルもサポートされるため、class 属性を java.util.Map として定義し、固定長フィールドの値をマップインスタンスにバインドできます。その後、マップインスタンスはリストまたはマップに追加されます。

プログラムを用いて固定長のリーダーを設定することも可能です。

このコードは、個人のレコードセットを読み取り、そのレコードセットを person インスタンスのリストへバインドする固定長リーダーを設定します。

```

Smooks smooks = new Smooks();

smooks.setReaderConfig(new FixedLengthReaderConfigurator("firstname[10]?
trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]")
    .setBinding(new FixedLengthBinding("people",
Person.class, FixedLengthBindingType.LIST)));

```



```

JavaResult result = new JavaResult();
smooks.filterSource(new StreamSource(fixedLengthStream), result);

List<Person> people = (List<Person>) result.getBean("people");

```

Java バインディングの設定は強制ではありません。代わりに、プログラムを用いて Smooks インスタンスを設定し、他のビジター実装を使用して固定長のレコードセット上でさまざまな処理を実行することができます。

固定長レコードのデータを反映する Java タイプのリストまたはマップへ固定長レコードを直接バインディングしたい場合、FixedLengthListBinder または the FixedLengthMapBinder クラスを使用できます。

```

// Note: The binder instance should be cached and reused...
FixedLengthListBinder binder = new FixedLengthListBinder("firstname[10]?trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]", Person.class);

List<Person> people = binder.bind(fixedLengthStream);

FixedLengthMapBinder:

// Note: The binder instance should be cached and reused...
FixedLengthMapBinder binder = new FixedLengthMapBinder("firstname[10]?trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]", Person.class, "firstname");

Map<String, Person> people = binder.bind(fixedLengthStream);

```

バインディング処理を更に制御する必要がある場合、低レベル API の使用に戻ります。

Smooks の EDI 処理は、<http://www.milyn.org/xsd/smooks/edi-1.2.xsd> 設定名前空間よりサポートされます。

設定例は次の通りです。

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:edi="http://www.milyn.org/xsd/smooks/edi-1.2.xsd">
  <!--
    Configure the EDI Reader to parse the message stream into a stream of
    SAX events.
  -->
  <edi:reader mappingModel="edi-to-xml-order-mapping.xml"
validate="false"/>
</smooks-resource-list>

```

- mappingModel: EDI メッセージを Smooks によって処理できる SAX イベントのストリームへ変換するための EDI マッピングモデルを定義します。
- validate: この属性は、EDI パーサーのデータタイプ検証を有効または無効にします (検証はデフォルトでは **on** になっています)。EDI データが Java オブジェクトモデルにバインドされている場合、EDI リーダー上のデータタイプ検証を off にするとよいでしょう (jb:bean のように Java バインディングを使用)。これは、バインディングレベルで検証が行われるからです。

EDI から SAX へのイベントマッピング処理は、EDI リーダーへ提供されるマッピングモデルが基とな

ります (このモデルは常に <http://www.milyn.org/xsd/smooks/edi-1.2.xsd> のスキーマを使用する必要があります。このスキーマより、グループ内のグループ、繰り返しの断片、繰り返しの断片グループなど、断片グループがサポートされることを確認できます。必ずスキーマを再確認するようにしてください)。

medi:segment 要素は、minOccurs と maxOccurs の 2 つの任意属性をサポートします (両方ともデフォルト値は 1 になります)。これらの属性を使用して、断片の特徴を制御します。maxOccurs の値を -1 にすると、(バインドされていない) EDI メッセージがある場所で断片を何回でも繰り返すことができます。

segmentGroup 要素を使用して断片グループを追加できます。断片グループはグループの最初の断片と照合されます。断片グループにはネストされた segmentGroup 要素が含まれることがありますが、segmentGroups の最初の要素は断片でなければなりません。segmentGroup 要素は minOccurs および maxOccurs の濃度をサポートします。任意の xmlTag 属性もサポートされます。この属性が存在する場合、xmlTag 属性値の名前を持つ要素に挿入される、一致した断片グループによって XML が生成されます。

断片は次の方法の 1 つによって合致されます。

- 断片コード (segcode) 上の完全一致。
- 完全な断片上の正規表現パターンの合致。segcode="1A*a.*" のように、segcode 属性が正規表現パターンを定義します。
- required: フィールド、コンポーネント、およびサブコンポーネントの設定は、required 属性をサポートします。この属性は、フィールド、コンポーネント、またはサブコンポーネントに値が必要であることを知らせます。
- デフォルトでは、値は必要ありません (フィールド、コンポーネント、およびサブコンポーネント)。
- truncatable: 断片、フィールド、およびコンポーネントの設定は truncatable 属性をサポートします。断片では、後続のフィールドが指定されず「required」でない場合は (前述の「required」属性を参照) パーサーエラーは生成されません。フィールド/コンポーネントおよびコンポーネント/サブコンポーネントの場合でも同様です。
- デフォルトでは、断片、フィールドおよびコンポーネントは省略できません。

よって、メッセージに存在するフィールド、コンポーネント、サブコンポーネントは次の状態のいずれかとなります。

- 値と存在 (required="true")
- 値なしで存在 (required="false")
- 存在しない (required="false" and truncatable="true")

メッセージグループの多くは同じ断片定義を使用します。断片を一度定義し、トップレベルの設定にインポートできると、重複を避けることができるため、時間を節約することができます。以下は、インポート機能を示す簡単な設定になります。

```
<?xml version="1.0" encoding="UTF-8"?>
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-mapping-1.2.xsd">

    <medi:import truncatableSegments="true" truncatableFields="true"
```

```

truncatableComponents="true" resource="example/edi-segment-definition.xml"
namespace="def"/>

    <medi:description name="DVD Order" version="1.0"/>

    <medi:delimiters segment="
" field="*" component="^" sub-component="~" escape="?" />

    <medi:segments xmltag="Order">
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:HDR"
segcode="HDR" xmltag="header"/>
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:CUS"
segcode="CUS" xmltag="customer-details"/>
        <medi:segment minOccurs="0" maxOccurs="-1" segref="def:ORD"
segcode="ORD" xmltag="order-item"/>
    </medi:segments>

</medi:edimap>

```

今後の再利用を容易にするため、断片および子断片が含まれる断片を別ファイルへ分離することができ
ます。

- segref: インポートする断片を参照する namespace:name が含まれます。
- truncatableSegments: インポートされたリソースマッピングファイルに指定された truncatableSegments を上書きします。
- truncatableFields: インポートされたリソースマッピングファイルに指定された truncatableFields を上書きします。
- truncatableComponents: インポートされたリソースマッピングファイルに指定された truncatableComponents を上書きします。

field、component、および sub-component 要素は type 属性をサポートします。この属性によってデー
タタイプの指定が可能になります。この属性は2つのサブ属性によって構成されます。

- type: type 属性は基本的なデータタイプを指定します。
- typeParameters: typeParameters 属性は、指定されたタイプの DataDecoder に対するデータデ
コードパラメータを指定します。

この例はタイプのサポートを表しています。

```

<?xml version="1.0" encoding="UTF-8"?>
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-mapping-
1.2.xsd">

    <medi:description name="Segment Definition DVD Order" version="1.0"/>

    <medi:delimiters segment="
" field="*" component="^" sub-component="~" escape="?" />

    <medi:segments xmltag="Order">

        <medi:segment segcode="HDR" xmltag="header">
            <medi:field xmltag="order-id"/>
            <medi:field xmltag="status-code" type="Integer"/>

```

```

        <medi:field xmltag="net-amount" type="BigDecimal"/>
        <medi:field xmltag="total-amount" type="BigDecimal"/>
        <medi:field xmltag="tax" type="BigDecimal"/>
        <medi:field xmltag="date" type="Date"
typeParameters="format=yyyyMMddHHmm"/>
    </medi:segment>

</medi:segments>

</medi:edimap>

```

タイプシステムは次のような用途で使用できます。

- フィールド検証。
- **Edifact Java Compilation**

EDIReaderConfigurator を用いて EDIReader を使用するよう、プログラムを用いて Smooks インスタンスを設定することが可能です。

```

Smooks smooks = new Smooks();

// Create and initialise the Smooks config for the parser...
smooks.setReaderConfig(new
EDIReaderConfigurator("/edi/models/invoice.xml"));

// Use the smooks as normal
smooks.filterSource(...);

```

Edifact Java Compiler は、EDI から Java への変換プロセスを大幅に簡易化できます。Edifact Java Compiler は以下を生成します。

- EDI マッピングモデルの Java オブジェクトモデル。
- EDI マッピングモデルによって記述される EDI メッセージのインスタンスより Java Object モデルへ投入する Smooks Java バインディング設定。
- EDI データを Java オブジェクトモデルへバインドする **Edifact Java Compiler** の使用を大変容易にするファクトリクラス。

Edifact Java Compiler を使用すると、次のような簡単な Java コードの書き込みが可能になります。

```

// Create an instance of the EJC generated Factory class. This should
normally be cached and reused...
OrderFactory orderFactory = OrderFactory.getInstance();

// Bind the EDI message stream data into the EJC generated Order model...
Order order = orderFactory.fromEDI(ediStream);

// Process the order data...
Header header = order.getHeader();
Name name = header.getCustomerDetails().getName();
List<OrderItem> orderItems = order.getOrderItems();

```

Maven または **Ant** より **Edifact Java Compiler** を実行できます。

Maven 下で実行するには、POM ファイルにプラグインを追加します。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.milyn</groupId>
      <artifactId>maven-ejc-plugin</artifactId>
      <version>1.2</version>
      <configuration>
        <ediMappingFile>edi-model.xml</ediMappingFile>
        <packageName>com.acme.order.model</packageName>
      </configuration>
      <executions>
        <execution><goals><goal>generate</goal></goals>
      </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

プラグインには 3 つの設定パラメータがあります。

- **ediMappingFile**: **Maven** プロジェクト内にある EDI マッピングモデルファイルへのパスになります (これは任意です。デフォルトのパスは **src/main/resources/edi-model.xml** になります)。
- **packageName**: 生成された Java アーティファクトが格納される Java パッケージです (Java オブジェクトモデルとファクトリクラス)。
- **destDir**: 生成されたアーティファクトが作成されコンパイルされるディレクトリになります (これは任意です。デフォルトは **target/ejc** になります)。

Ant 下で実行するには、EJC タスクを作成し、実行します。次のコードはその方法を表しています。

```
<target name="ejc">

  <taskdef resource="org/milyn/ejc/ant/anttasks.properties">
    <classpath><fileset dir="/smooks-1.2/lib" includes="*.jar"/>
  </classpath>
  </taskdef>

  <ejc edimappingmodel="src/main/resources/edi-model.xml"
        destdir="src/main/java"
        packagename="com.acme.order.model"/>

  <!-- Ant as usual from here on... compile and jar the source... -->

</target>
```

Edifact Java Compiler の詳細について知りたい場合、EJC の例である UN/EDIFACT を確認するのが最も簡単な方法です。

Smooks は以下の方法で UN/EDIFACT メッセージインターチェンジに対するサポートを提供します。

- 正式な UN/EDIFACT メッセージ定義の ZIP ディレクトリから生成された、事前生成された EDI マッピングモデル。これにより、UN/EDIFACT メッセージインターチェンジを消費しやすい XML 形式へ変換できます。
- 純粋な Java を使用した UN/EDIFACT インターチェンジの読み書きを容易にする、事前生成された Java バインディング。

UN/EDIFACT インターチェンジは、ベース `edi:reader` の特殊な亜種を介してサポートされます。このリーダーを設定するには、次のように `http://www.milyn.org/xsd/smooks/unedifact-1.4.xsd` 名前空間を設定します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:unediFact="http://www.milyn.org/xsd/smooks/unedifact-1.4.xsd">

  <unediFact:reader mappingModel="urn:org.milyn.edi.unedifact:d03b-
mapping:v1.4" ignoreNewLines="true" />

</smooks-resource-list>
```

`mappingModel` 属性は、マッピングモデルの ZIP セットの **Maven** アーティファクトを参照する *URN* を定義します。**Maven** アーティファクトはリーダーによって使用されます。

UN/EDIFACT インターチェンジを消費するよう (`UNEdifactReaderConfigurator` を介してなど)、プログラムを用いて Smooks を設定することも可能です。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
UNEdifactReaderConfigurator("urn:org.milyn.edi.unedifact:d03b-
mapping:v1.4"));
```

格納するアプリケーションのクラスパスに以下が含まれるようにする必要があります。

- 必要な EDI マッピングモデル。
- Smooks EDI カートリッジ

次の例は、この設定が必要とする可能性がある **Maven** 依存関係の一部を表しています。

```
<dependency>
  <groupId>org.milyn</groupId>
  <artifactId>milyn-smooks-edi</artifactId>
  <version>1.4</version>
</dependency>

<!-- Required Mapping Models -->
<dependency>
  <groupId>org.milyn.edi.unedifact</groupId>
  <artifactId>d93a-mapping</artifactId>
  <version>v1.4</version>
</dependency>
<dependency>
  <groupId>org.milyn.edi.unedifact</groupId>
```

```

    <artifactId>d03b-mapping</artifactId>
    <version>v1.4</version>
  </dependency>

```

アプリケーションが EDI マッピングモデルの ZIP セットをクラスパスに追加した後、このモデルを使用するよう **Smooks** を設定できます。設定を行うには、URN を unedifact:reader 設定の mappingModel 属性値として使用し、**Maven** アーティファクトを参照します。

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:unedifact="http://www.milyn.org/xsd/smooks/unedifact-1.4.xsd"

  <unedifact:reader mappingModel="urn:org.milyn.edi.unedifact:d03b-
mapping:v1.4" ignoreNewLines="true" />

</smooks-resource-list>

```



注記

mappingModel 属性は、コンマ区切りの EDI マッピングモデルの URN を複数定義できます。複数定義すると、異なるディレクトリに定義された複数の UN/EDIFACT メッセージに対応するインターチェンジを、UN/EDIFACT リーダーが処理しやすくなります。

マッピングモデルの ZIP セットはすべての UN/EDIFACT ディレクトリが使用可能です。これらの ZIP セットは **Maven SNAPSHOT** および **Central** レポジトリより取得し、標準の **Maven** 依存関係管理を使用してアプリケーションに追加します。

たとえば、D93A マッピングモデルの ZIP セットをアプリケーションのクラスパスに追加するには、次の依存関係をアプリケーションの POM ファイルに設定します。

```

<!-- The mapping model sip set for the D93A directory... -->
<dependency>
  <groupId>org.milyn.edi.unedifact</groupId>
  <artifactId>d93a-mapping</artifactId>
  <version>v1.4</version>
</dependency>

```

次の手順では、この ZIP セットを使用するよう Smooks を設定します。次のコードのように、unedifact: リーダーの設定を Smooks の設定に追加します。

```

<unedifact:reader mappingModel="urn:org.milyn.edi.unedifact:d93a-
mapping:v1.4" />

```

Maven アーティファクトの依存関係情報を基にした URN を使用して、どのようにリーダーを設定するか注目してください。

また、複数のマッピングモデルの ZIP セットをアプリケーションのクラスパスに追加することができます。これには、コンマ区切りの URN にてすべての ZIP セットを unedifact:reader 設定に追加します。

事前生成された Java バインディングモデルセットにはツールが提供されています (マッピングモデル ZIP セットごとに 1 つ)。このツールを使用し、大変簡単な生成されたファクトリクラスを用いて UN/EDIFACT インターチェンジを処理します。

次のコード例は、D03B UN/EDIFACT メッセージインターチェンジの処理方法を表しています。

Reading:

```
// Create an instance of the EJC generated factory class... cache this and
// reuse !!!
D03BInterchangeFactory factory = D03BInterchangeFactory.getInstance();

// Deserialize the UN/EDIFACT interchange stream to Java...
UNEdifactInterchange interchange = factory.fromUNEdifact(ediInStream);

// Need to test which interchange syntax version. Supports v4.1 at the
// moment...
if(interchange instanceof UNEdifactInterchange41) {
    UNEdifactInterchange41 interchange41 = (UNEdifactInterchange41)
    interchange;

    for(UNEdifactMessage41 message : interchange41.getMessages()) {
        // Process the messages...

        Object messageObj = message.getMessage();

        if(messageObj instanceof Invoic) {
            // It's an INVOIC message....
            Invoic invoic = (Invoic) messageObj;
            ItemDescription itemDescription = invoic.getItemDescription();
            // etc etc....
        } else if(messageObj instanceof Cuscar) {
            // It's a CUSCAR message...
        } else if(etc etc etc...) {
            // etc etc etc...
        }
    }
}
```

Writing:

```
factory.toUNEdifact(interchange, ediOutputStream);
```

マッピングモデル ZIP セットと同様、Smooks には **Maven SNAPSHOT** および **Central** レポジトリを介して配布される、事前生成の UN/EDIFACT Java オブジェクトモデルが含まれています。

Maven を使用する場合、D03B メッセージインターチェンジを処理するにはそのディレクトリのバインディング依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.milyn.edi.unedifact</groupId>
  <artifactId>d03b-binding</artifactId>
  <version>v1.4</version>
</dependency>
```

JSON データを処理するには、JSON リーダーを設定する必要があります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">
```



```
<json:reader/>

</smooks-resource-list>
```

次の設定オプションを使用すると、root、document、および array 要素の XML 名を設定できます。

- rootName: root 要素の名前です。デフォルトは yam1 です。
- elementName: sequence 要素の名前です (デフォルトは element です)。

JSON では、XML 要素名では使用できない文字をキー名に使用できます。リーダーはこの問題に対して複数の解決法を提供します。リーダーは、空白文字、不正な文字、数字で始まるキー名の数字を検索し、置き換えることが可能です。リーダーを使用して、キー名を完全に異なる名前に置き換えることも可能です。次のコード例は、このような機能をすべて表しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

  <json:reader keyWhitespaceReplacement="_" keyPrefixOnNumeric="n"
    illegalElementNameCharReplacement=".">
    <json:keyMap>
      <json:key from="some key">someKey</json:key>
      <json:key from="some&key" to="someAndKey" />
    </json:keyMap>
  </json:reader>

</smooks-resource-list>
```

- keyWhitespaceReplacement: JSON マップキーの空白文字を置換する文字になります。デフォルトでは定義されていないため、リーダーは自動的に空白文字を検索しません。
- keyPrefixOnNumeric: JSON のノード名が数字で始まる場合、このプレフィックス文字を追加します。デフォルトでは定義されていないため、リーダーは数字で始まる要素名を検索しません。
- illegalElementNameCharReplacement: JSON の要素名で不正な文字が見つかった場合、この値に置き換えられます。

希望する場合、次の任意の設定を設定することも可能です。

- nullValueReplacement: JSON の null 値を置き換える文字列です。デフォルトは空の文字列です。
- encoding: リーダーによって処理される JSON メッセージ InputStream のデフォルトのエンコードです。



注記

この機能は廃止されました。代わりに、**java.io.Reader** を **Smooks.filterSource()** メソッドに提供して JSON ストリームのソース文字のエンコードを管理する必要があります。

JSON 設定を読み取るよう、プログラムを用いて Smooks を設定するには、**JSONReaderConfigurator** クラスを使用します。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new JSONReaderConfigurator()
    .setRootName("root")
    .setArrayElementName("e"));

// Use Smooks as normal...
```

YAML ファイルを処理したい場合は、処理できるリーダーを設定する必要があります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:yaml="http://www.milyn.org/xsd/smooks/yaml-1.4.xsd">

  <yaml:reader/>

</smooks-resource-list>
```

YAML ストリームには複数のドキュメントが含まれることがあります。リーダーは document 要素を root 要素の子として追加し、これに対応します。空の YAML ドキュメントが 1 つ含まれ、XML でシリアライズされた YAML ストリームは次のようになります。

```
<yaml>
  <document>
</document>
</yaml>
```

次のオプションを設定すると root、document、および array 要素の XML 要素名を設定できます。

- rootName: root 要素の名前です。デフォルトは yaml です。
- documentName: document 要素の名前です。デフォルトは document です。
- elementName: sequence 要素の名前です。デフォルトは element です。

YAML では、XML 要素名では使用できない文字をキー名に使用できます。リーダーはこの問題に対して複数の解決法を提供します。リーダーは、空白文字、不正な文字、数字で始まるキー名の数字を検索し、置き換えることが可能です。リーダーを使用して、キー名を完全に異なる名前に置き換えることも可能です。次のコード例は、このような機能をすべて表しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:yaml="http://www.milyn.org/xsd/smooks/yaml-1.4.xsd">

  <yaml:reader keyWhitespaceReplacement="_" keyPrefixOnNumeric="n"
    illegalElementNameCharReplacement=".">
    <yaml:keyMap>
      <yaml:key from="some key">someKey</yaml:key>
      <yaml:key from="some&key" to="someAndKey" />
    </yaml:keyMap>
```

```
</yaml:reader>

</smooks-resource-list>
```

- **keyWhitespaceReplacement**: YAML マップキーの空白文字を置換する文字になります。デフォルトでは定義されません。
- **keyPrefixOnNumeric**: YAML のノード名が数字で始まる場合、このプレフィックス文字を追加します。デフォルトでは定義されません。
- **illegalElementNameCharReplacement**: YAML の要素名で不正な文字が見つかった場合、この値に置き換えられます。デフォルトでは定義されません。

YAML は アンカー と エイリアス の概念を使用します。YAML リーダーは、3 つのストラテジを介してアンカーとエイリアスに対応できます。aliasStrategy 設定オプションより希望のストラテジを定義します。このオプションには以下の値の 1 つを使用できます。

- **REFER**: リーダーは、アンカーまたはエイリアスを持つ要素上で参照属性を作成します。アンカーを持つ要素は、アンカーからの名前が属性値として含まれる id 属性を取得します。エイリアスを持つ要素は、アンカーからの名前が属性値として含まれる ref 属性を取得します。anchorAttributeName および aliasAttributeName プロパティを設定して、アンカーおよびエイリアスの属性名を定義できます。
- **RESOLVE**: リーダーは、エイリアスを見つけた時にアンカーの値またはデータ構造を解決します。そのため、アンカーの SAX イベントは alias 要素の子イベントとして繰り返されます。大量のアンカーまたはアンカーと大量のデータ構造が YAML ドキュメントに含まれている場合、メモリーの問題が発生する可能性があります。
- **REFER_RESOLVE**: REFER と RESOLVE の組み合わせになります。アンカーおよびエイリアス属性が設定され、アンカーの値またはデータ構造も解決されます。アンカーの名前にビジネス的な意味合いがある場合にこのオプションは便利です。

YAML リーダーはデフォルトで REFER ストラテジを使用します。

YamlReaderConfigurator クラスを利用して YAML 設定を読み取るよう、プログラムを用いて Smooks を設定することができます。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new YamlReaderConfigurator()
    .setRootName("root")
    .setDocumentName("doc")
    .setArrayElementName("e"))
    .setAliasStrategy(AliasStrategy.REFER_RESOLVE)
    .setAnchorAttributeName("anchor")
    .setAliasAttributeName("alias");

// Use Smooks as normal...
```

Smooks は Java オブジェクトグラフを別の Java オブジェクトグラフに変換することができます。Smooks は SAX 処理モデルを使用してこれを実行します。そのため、中間のオブジェクトモデルは構築されず、ソース Java オブジェクトグラフは直接 SAX イベントのストリームに変換されます。これらの SAX イベントはターゲットの Java オブジェクトグラフへ投入するために使用されます。

HTML Smooks Report Generator ツールを使用する場合、ソースオブジェクトモデルが作成したイベントストリームは次のようになります。

```

<example.srcmodel.Order>
  <header>
    <customerNumber>
      </customerNumber>
    <customerName>
      </customerName>
    </header>
  <orderItems>
    <example.srcmodel.OrderItem>
      <productId>
        </productId>
      <quantity>
        </quantity>
      <price>
        </price>
      </example.srcmodel.OrderItem>
    </orderItems>
  </example.srcmodel.Order>

```

ここで、Smooks の Java Bean リソースをこのイベントストリームへ向けます。

このトランスフォーメーションを実行するための Smooks 設定 (*smooks-config.xml*) は次の通りです。

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

  <jb:bean BeanId="lineOrder" class="example.trgmodel.LineOrder"
    createOnElement="example.srcmodel.Order">
    <jb:wiring property="lineItems" BeanIdRef="lineItems" />
    <jb:value property="customerId" data="header/customerNumber" />
    <jb:value property="customerName" data="header/customerName" />
  </jb:bean>

  <jb:bean BeanId="lineItems" class="example.trgmodel.LineItem[]"
    createOnElement="orderItems">
    <jb:wiring BeanIdRef="lineItem" />
  </jb:bean>

  <jb:bean BeanId="lineItem" class="example.trgmodel.LineItem"
    createOnElement="example.srcmodel.OrderItem">
    <jb:value property="productCode"
    data="example.srcmodel.OrderItem/productId" />
    <jb:value property="unitQuantity"
    data="example.srcmodel.OrderItem/quantity" />
    <jb:value property="unitPrice"
    data="example.srcmodel.OrderItem/price" />
  </jb:bean>

</smooks-resource-list>

```

ソースオブジェクトモデルは、 **org.milyn.delivery.JavaSource** オブジェクトより Smooks へ提供されます。コンストラクタをソースモデルのルートオブジェクトへ渡してこのオブジェクトを作成します。結果として生じる Java ソースオブジェクトは **Smooks#filter** メソッドで使用されます。そ

の結果となるコードは次の通りです。

```
protected LineOrder runSmooksTransform(Order srcOrder) throws IOException,
SAXException {
    Smooks smooks = new Smooks("smooks-config.xml");
    ExecutionContext executionContext = smooks.createExecutionContext();

    // Transform the source Order to the target LineOrder via a
    // JavaSource and JavaResult instance...
    JavaSource source = new JavaSource(srcOrder);
    JavaResult result = new JavaResult();

    // Configure the execution context to generate a report...
    executionContext.setEventListener(new
    HtmlReportGenerator("target/report/report.html"));

    smooks.filterSource(executionContext, source, result);

    return (LineOrder) result.getBean("lineOrder");
}
```

CSV および固定長リーダーを使用すると、データが SAX イベントへ変換される前に入力データ上で文字列操作関数を実行できます。次の関数を使用することができます。

- `upper_case`: 文字列の大文字バージョン返します。
- `lower_case`: 文字列の小文字バージョンを返します。
- `cap_first`: 最初の言葉が大文字になっている文字列を返します。
- `uncap_first`: 最初の文字が大文字でない文字列を返します。`cap_first` とは逆になります。
- `capitalize`: すべての言葉が大文字の文字列を返します。
- `trim`: 最初と最後に空白文字のない文字列を返します。
- `left_trim`: 最初に空白文字のない文字列を返します。
- `right_trim`: 最後に空白文字のない文字列を返します。

`trim.upper_case` のように、ピリオドで区切って関数をつなげることが可能です。

フィールドごとに関数を定義する方法は、使用するリーダーによって異なります。

第4章 検証

Smooks では、*ルール*という言葉は一般的な概念を意味します。特定のカートリッジに固有なものではありません。

他のコンポーネントより *RuleProvider* を設定および参照できます。



注記

検証カートリッジは、ルール機能を使用する唯一のカートリッジです。

ルールは *ruleBases* より中央的に定義されます。単一の Smooks 設定は複数の *ruleBase* 定義を参照できます。*ruleBase* 設定には名前、ルール **src**、およびルールプロバイダーがあります。

ルールソースの形式はプロバイダー実装に全体的に依存します。参照できるようにするため、各ルールに一意的な名前 (単一ソースのコンテキスト内) を付けることのみが必要になります。

ruleBase 設定の例は次の通りです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

  <rules:ruleBases>
    <rules:ruleBase name="regexAddressing"
      src="/org/milyn/validation/address.properties"
      provider="org.milyn.rules.regex.RegexProvider" />
    <rules:ruleBase name="order"
      src="/org/milyn/validation/order/rules/order-rules.csv"
      provider="org.milyn.rules.mvel.MVELProvider"/>
  </rules:ruleBases>

</smooks-resource-list>
```

rules:ruleBase 設定要素の設定オプションは次の通りです。

- **name**: このルールを参照するために他のコンポーネントが使用します。必須のオプションです。
- **src**: ファイルまたは *RuleProvider* にとって意味のあるものになります。必須のオプションです。
- **provider**: 使用したい実際のプロバイダー実装になります。ここで異なる技術が実行されます。上記の設定では、1 つの *RuleProvider* が正規表現を使用しますが、複数の *ruleBase* 要素を指定でき、これらの *ruleBase* は必要な数の *RuleProviders* を持つことができます。必須のオプションです。

ルールプロバイダは **org.milyn.rules.RuleProvider** インターフェースを実装します。

Smooks には 2 つの *RuleProvider* 実装が事前設定されています。

- *RegexProvider*
- *MVELProvider*

独自の RuleProvider 実装を作成することも可能です。

名前の通り、RegexProvider は正規表現を使用できるようにします。フィルタされたメッセージの選択されたデータフィールドの形式に固有する低レベルなルールを定義できるようにします。たとえば、電子メールアドレスの正しい構文が含まれていることを検証するため、特定のフィールドに適用されることがあります。

次のコードは Regex ruleBase の設定方法を表しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

  <rules:ruleBases>
    <rules:ruleBase name="customer"
      src="/org/milyn/validation/order/rules/customer.properties"
      provider="org.milyn.rules.regex.RegexProvider"/>
  </rules:ruleBases>

</smooks-resource-list>
```

正規表現は標準の **.properties** ファイル形式で定義されます。以下は customer.properties の正規表現ルール定義ファイル (上記の例より) の一例になります。

```
# Customer data rules...
customerId=[A-Z][0-9]{5}
customerName=[A-Z][a-z]*, [A-Z][a-z]
```

MVEL プロバイダは、ルールを MVEL 表現として定義できるようにします。これらの表現は Smooks Javabeen コンテキストの内容上で実行されます。そのため、Smooks Bean コンテキストの Java オブジェクトにデータ (フィルタリングされたメッセージから) をバインドする必要があります。

これは、メッセージ断片でさらに複雑なルールを定義できるようにします (例: ターゲットされた注文商品断片の製品は、注文ヘッダー詳細に指定された顧客の年齢資格制限内であるか)。

このコードは MVEL ruleBase の設定方法を表しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

  <rules:ruleBases>
    <rules:ruleBase name="order"
      src="/org/milyn/validation/order/rules/order-rules.csv"
      provider="org.milyn.rules.mvel.MVELProvider"/>
  </rules:ruleBases>

</smooks-resource-list>
```

MVEL ルールを CSV ファイルに保存する必要があります。これらのファイルを編集する最も簡単な方法は、**LibreOffice Calc** や **Gnumeric** などのスプレッドシートアプリケーションを使用することです。各ルールレコードには、以下で構成される 2 つのフィールドが含まれます。

- ルール名

- MVEL 表現

コメントヘッダーの行を追加するには、最初のフィールドの前にハッシュマーク(#)を追加します。

Smooks 検証カートリッジはルールカートリッジによって提供される機能によって構築され、ルールベースの断片検証を提供します。

これにより、メッセージ断片上で詳細な検証を実行できるようになります。Smooks の他の機能と同様に、検証機能はサポート対象の全データ形式で使用可能です。そのため、XML データだけでなく EDI や JSON、CSV などにも強固な検証を実行することができます。

検証の設定は <http://www.milyn.org/xsd/smooks/validation-1.0.xsd> 設定名前空間によって定義されます。

Smooks は複数のルールプロバイダタイプをサポートし、検証カートリッジはこれらのタイプを使用できます。各ルールプロバイダタイプは異なるレベルの検証を提供しますが、すべて全く同じように設定されます。Smooks の検証カートリッジはルールプロバイダを抽象リソースと見なし、メッセージ断片を検証するためメッセージ断片を目指します。

検証ルールの設定には以下を指定する必要があります。

- executeOn: ルールが実行される断片です。
- executeOnNS: executeOn が属する断片名前空間です。
- name: 適用されるルールの名前です。ドット区切り形式の ruleBase と ruleName の組み合わせ (ruleBaseName.ruleName) を参照する複合ルール名です。
- onFail: 適合の失敗の深刻度を判断します。

検証ルールの設定例は次の通りです。

```
<validation:rule executeOn="order/header/email"
name="regexAddressing.email" onFail="ERROR" />
```

Smooks フィルター操作ごとの検証最大失敗数を設定できます。最大値を越えると例外がスローされます。OnFail.FATALに設定された検証は常に例外をスローし、処理を停止することに注意してください。

最大検証失敗数を設定するには、次のコードを Smooks の設定に追加します。

```
<params>
  <param name="validation.maxFails">5</param>
</params>
```

検証設定の onFail 属性は実行するアクションを指定しました。これは、検証の失敗をどのように報告するかを決定します。

使用できるオプションは次の通りです。

- OK: これを使用して検証を okay として保存します。ValidationResults.getOks を呼び出すとすべての検証警告が返されます。コンテンツベースのルーティングを使用する場合に便利なオプションです。
- WARN: 検証を warning として保存します。ValidationResults.getWarnings を呼び出すとすべての検証警告が返されます。

- **ERROR**: 検証を **error** として保存します。 `ValidationResults.getErrors` を呼び出すとすべての検証エラーが返されます。
- **FATAL**: 検証の失敗が発生した直後に `ValidationException` をスローします。 `ValidationResults.getFatal` を呼び出すと致命的な検証失敗が表示されます。

ルールベースには次の形式の複合ルール名を使用します。

```
<ruleProviderName>.<ruleName>
```

- `ruleProviderName` はルールプロバイダを識別し、 `ruleBase` 要素の `name` 属性へマッピングします。
- `ruleName` はルールプロバイダの認識する特定ルールを識別します。 `src` ファイルに定義されるルールである場合もあります。

`Smooks.filterSource` は検証の結果をキャプチャします。 `filterSource` メソッドが返される時、 `ValidationResult` インスタンスにはすべての検証データが含まれます。

次のコードは `Smooks` がメッセージ断片の検証をどのように実行するかを表しています。

```
ValidationResult validationResult = new ValidationResult();

smooks.filterSource(new StreamSource(messageInStream), new
StreamResult(messageOutStream), validationResult);

List<OnFailResult> errors = validationResult.getErrors();
List<OnFailResult> warnings = validationResult.getWarnings();
```

上記のコードでは、個別の警告とエラーの検証結果が `OnFailResult` インスタンスの形式で `ValidationResult` オブジェクトより使用できるようになります。各 `OnFailResult` インスタンスは障害の詳細を提供します。

検証カートリッジを使用すると、検証の失敗に関連する現地語化メッセージを指定することもできます。現地語化メッセージは標準の `Java ResourceBundle` ファイルに定義します (**.properties** 形式を使用)。



注記

検証メッセージバンドルのベース名はルールソース (`src`) が基になります。ルールソースのファイル拡張子を省略し、 `i18n` という名前のフォルダーを追加します。たとえば、 `/org/milyn/validation/order/rules/order-rules.csv` の MVEL `ruleBase` ソースの場合、対応する検証メッセージバンドルのベース名は `/org/milyn/validation/order/rules/i18n/order-rules` になります。

検証カートリッジを使用すると、**FreeMarker** テンプレートを現地語化メッセージに適応できるため、 `Bean` コンテキストからのコンテキストデータや、実際に発生したルールの失敗に関するデータがメッセージに含まれるようにすることが可能です。**FreeMarker** ベースのメッセージの前には **ftl:** を付ける必要があり、標準の **FreeMarker** の表記法を使用してコンテキストデータを参照する必要があります。 `Bean` コンテキストからの `Bean` は直接参照することが可能ですが、 `RuleEvalResult` とルールの失敗のパスは `ruleResult` および `path Bean` を介して参照することが可能です。

`RegexProvider` ルールを使用する例は次の通りです。

```
customerId=ftl:Invalid customer number '${ruleResult.text}' at '${path}'.
Customer number must match pattern '${ruleResult.pattern}'.
```

この例は Smooks を使用してメッセージ断片データを検証する方法を表しています。2 種類の検証ルールを使用して 2 種類の検証を実行します。

- **.properties** ファイル RuleBase に定義された正規表現を使用したメッセージフィールド値/形式の検証。たとえば、フィールドを有効な電子メールアドレスとして検証することが可能です。
- **.csv file** RuleBase に定義された MVEL 表現を使用したビジネスルールの検証。たとえば、ある注文の注文商品の合計価格 (価格に数量をかけた値) が事前定義されたビジネスルールに違反しないかどうかを検証することができます。

このルールを実行するには、ルートフォルダーの例に移動し、以下を実行します。

- **mvn clean install**
- **mvn exec:java**

次の例では、注文商品が複数含まれた XML メッセージが存在します (この機能は、Smooks によってサポートされる他のデータ形式すべてに対して同様に動作します)。

```
<Order>
  <header>
    <orderId>A188127</orderId>
    <username>user1</username>
    <name>
      <firstname>Harry</firstname>
      <lastname>Fletcher</lastname>
    </name>
    <email>harry.fletcher@gmail.</email>
    <state>South Dakota</state>
  </header>
  <order-item>
    <quantity>1</quantity>
    <productId>364b</productId>
    <title>The 40-Year-Old Virgin</title>
    <price>29.98</price>
  </order-item>
  <order-item>
    <quantity>2</quantity>
    <productId>299</productId>
    <title>Pulp Fiction</title>
    <price>29.99</price>
  </order-item>
</Order>
```

注文メッセージデータに複数の検証を実行します。

提供されたユーザー名が、大文字の後に5 つ数字が続く (S12345 や G54321 など) 形式に従うことを確認します。この検証を実行するには、正規表現を使用する必要があります。

次に、提供された電子メールアドレスが有効な形式であることを確認する必要があります。ここでも正規表現を使用して確認を行います。

さらに、各注文商品の `productId` フィールドの形式が 3 つの数字 (123 や 321 など) であることを確認する必要があります。ここでも正規表現を使用します。

最後に、各注文商品の合計が 50.00 を越えない (価格と数量を掛けた値が 50.00 を越えない) ことを確認する必要があります。この検証には MVEL 表現を使用します。

この検証を実行するには、正規表現ルールを分割し、個別の 2 つの **.properties** ファイルに置きます。

ここで、これらのファイルを例の **rules** ディレクトリに移動します。

MVEL 表現を **.csv** ファイルと **rules** ディレクトリに置きます。

customer.properties ファイルに格納される顧客関係の正規表現ルールは次のようになります。

```
# Customer data rules...
customerId=[A-Z][0-9]{5}

# Email address...
email=^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$
```

product.properties ファイルに格納される商品関係の正規表現ルールは次のようになります。

```
# Product data rules...
productId=[0-9]{3}
```

注文商品の合計をチェックする MVEL 表現は **order-rules.csv** ファイルに格納されます。



注記

OpenOffice.org Calc や **Gnumeric** などの表計算アプリケーションを使用すると、最も簡単に **.csv** ファイルを編集できます。

ここで、各ルールソースファイルに対してリソースバンドルの **.properties** ファイルを作成します。



注記

これらのファイルの名前は、対応するルールファイルの名前を基にしていることに注意してください。

rules/customer.properties に定義されるルールのメッセージバンドルは、**rules/i18n/customer.properties** ファイルにあります。

```
customerId=ftl:Invalid customer number '${ruleResult.text}' at '${path}'.
<!-- Customer number must begin with an uppercase character, followed by
5 digits. -->
email=ftl:Invalid email address '${ruleResult.text}' at '${path}'.
<!-- Email addresses match pattern '${ruleResult.pattern}'. -->
```

rules/product.properties に定義されるルールのメッセージバンドルは **rules/i18n/product.properties** ファイルにあります。

```
# Product data rule messages...
productId=ftl:Invalid product ID '${ruleResult.text}' at '${path}'.
<!-- Product ID must match pattern '${ruleResult.pattern}'. -->
```

rules/order-rules.csv に定義されるルールのメッセージバンドルは **rules/i18n/order-rules.properties** ファイルにあります。

```
# <!-- Order item rule messages. The "orderDetails" and "orderItem" beans
are populated by Smooks bindings - see config in following section. -->
order_item_total=ftl:Order ${orderDetails.orderId}
<!-- contains an order item for product ${orderItem.productId} with a
quantity of ${orderItem.quantity} and a unit price of ${orderItem.price}.
This exceeds the permitted per order item total. -->
```

これらの検証ルールを適用するために使用する必要がある設定は次の通りです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:rules="http://www.milyn.org/xsd/smooks/rules-
1.0.xsd"

xmlns:validation="http://www.milyn.org/xsd/smooks/validation-1.0.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.2.xsd">

    <params>
        <!-- Generate a ValidationException if we get more than 5
validation failures... -->
        <param name="validation.maxFails">5</param>
    </params>

    <!-- Define the ruleBases that are used by the validation rules... -->
    <rules:ruleBases>
        <!-- Field value rules using regex... -->
        <rules:ruleBase name="customer" src="rules/customer.properties"
provider="org.milyn.rules.regex.RegexProvider"/>
        <rules:ruleBase name="product" src="rules/product.properties"
provider="org.milyn.rules.regex.RegexProvider"/>

        <!-- Order business rules using MVEL expressions... -->
        <rules:ruleBase name="order" src="rules/order-rules.csv"
provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

    <!-- Capture some data into the bean context - required by the
business rule validations... -->
    <jb:bean BeanId="orderDetails" class="java.util.HashMap"
createOnElement="header">
        <jb:value data="header/*"/>
    </jb:bean>
    <jb:bean BeanId="orderItem" class="java.util.HashMap"
createOnElement="order-item">
        <jb:value data="order-item/*"/>
    </jb:bean>
```

```

        <!-- Target validation rules... -->
        <validation:rule executeOn="header/username"
name="customer.customerId" onFail="ERROR"/>
        <validation:rule executeOn="email" name="customer.email"
onFail="WARN"/>
        <validation:rule executeOn="order-item/productId"
name="product.productId" onFail="ERROR"/>

        <validation:rule executeOn="order-item" name="order.order_item_total"
onFail="ERROR"/>

</smooks-resource-list>

```

例の **Main** クラスより実行するためのコードは次の通りです。

```

protected static ValidationResult runSmooks(final String messageIn) throws
IOException, SAXException, SmooksException {
    // Instantiate Smooks with the config...
    final Smooks smooks = new Smooks("smooks-config.xml");

    try {
        // Create an exec context - no profiles....
        final ExecutionContext executionContext =
smooks.createExecutionContext();
        final ValidationResult validationResult = new ValidationResult();

        // Configure the execution context to generate a report...
        executionContext.setEventListener(new
HtmlReportGenerator("target/report/report.html"));

        // Filter the input message...
        smooks.filterSource(executionContext, new StringSource(messageIn),
validationResult);

        return validationResult;
    }
    finally {
        smooks.close();
    }
}

```

第5章 出力データの生成

Smooks の Javabeen カートリッジを使用して、メッセージデータより Java オブジェクトを作成および投入することができます。

Smooks のこの機能を XML、EDI、CSV などの単なる Java バインディングフレームワークとして使用することが可能ですが、Smooks の Java バインディング機能は他の機能の基盤ともなっています。これは、**Smooks** が作成する Java オブジェクト (データをバインドする) を **BeanContext** クラスより使用可能にするためです。このクラスは、Smooks の **ExecutionContext** を介して Smooks ビジター実装が使用できるようにする *Java Bean* コンテキストです。

Javabeen カートリッジに提供される機能によって構築される既存機能の一部には次のようなものがあります。

- テンプレート: 通常、テンプレートを BeanContext のオブジェクトに適用します。
- バリデーション: ビジネスルールのバリデーションでは通常、ルールを BeanContext のオブジェクトに適用します。
- メッセージの分割とルーティング: オブジェクト自体をルーティングするか、テンプレートを適用して操作の結果を新しいファイルヘルレーティングして、BeanContext のオブジェクトから分割メッセージを生成します。
- 永続化 (データベースの読み書き): これらの機能は、データベースにコミットされる Java オブジェクト (エンティティなど) を作成および投入するための Java バインディング機能に依存します。データベースから読み取られるデータは通常 BeanContext へバインドされます。
- メッセージのリッチ化: 上記の説明通り、リッチ化されたデータ (データベースなどから読み取られたデータ) は通常 BeanContext へバインドされます。その BeanContext より、Java バインディング機能自体を含む Smooks の他の機能すべてが使用可能になります (表現ベースバインディングに対して使用可能)。これにより、**Smooks** によって生成されたメッセージをリッチ化することができます。

次の例はすべてこの XML メッセージから派生します。

```
<order>
  <header>
    <date>Wed Nov 15 13:45:28 EST 2006</date>
    <customer number="123123">Joe</customer>
  </header>
  <order-items>
    <order-item>
      <product>111</product>
      <quantity>2</quantity>
      <price>8.90</price>
    </order-item>
    <order-item>
      <product>222</product>
      <quantity>7</quantity>
      <price>5.20</price>
    </order-item>
  </order-items>
</order>
```

<http://www.milyn.org/xsd/smooks/javabeen-1.4.xsd> 設定名前空間を介して JavaBean カートリッジを使用します (スキーマを IDE にインストールして、自動補完機能が使用できるようにします)。

設定例は次の通りです。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean BeanId="order" class="example.model.Order"
createOnElement="#document" />

</smooks-resource-list>
```

この設定は **example.model.Order** クラスのインスタンスを作成し、「order」という BeanId 下の Bean コンテキストへバインドします。インスタンスは #document 要素のメッセージの最初に作成されます (root order 要素の最初)。

- BeanId: Bean 識別子です。
- class: Bean の完全修飾クラス名です。
- createOnElement: この属性を使用して Bean インスタンスが作成されるタイミングを制御します。バインディング設定 (jb:bean 要素の子要素) より Bean プロパティの投入を制御します。
- createOnElementNS: この属性より createOnElement の名前空間を指定できます。

Javabean カートリッジは Java Bean の次の状態を設定します。

- 公開された引数のないコンストラクタがあります。
- 公開プロパティセッターメソッドがあります。特定の名前形式に従う必要はありませんが、標準のプロパティセッターメソッド名の形式に従う方がよいでしょう。
- 直接 Java Bean プロパティを設定できません。

前項の設定では、**example.model.Order** Bean インスタンスを作成し、Bean コンテキストにバインドしました。次項ではデータを Bean インスタンスにバインドする方法を説明します。

Javabean カートリッジでは 3 種類のデータバインディングを使用できます (使用するには、データバインディングを jb:bean 要素の子要素として追加します)。

- jb:value: ソースメッセージのイベントストリームからのデータ値をターゲット Bean へバインドするために使用されます。
- jb:wiring: Bean コンテキストからの他の Bean インスタンスをターゲット Bean の Bean プロパティへ「プラグ」するために使用されます。

この設定を使用して オブジェクトグラフ (Java オブジェクトインスタンスの疎コレクションとは逆) を構築できます。

BeanId、Java クラスタイプ、またはアノテーションを基に Bean をプラグできます。

- jb:expression: 名前の通り、この設定を使用して式から算出された値をバインドできます。注文商品の合計値を OrderItem Bean へバインドする場合が簡単な例になります (値段と数量を掛けた値を算出する式の結果を基にします)。

execOnElement 属性式を使用して、式が評価され結果がバインドされる要素を定義します (定義しない場合、親である jb:bean createOnElement の値を基に式が実行されます)。

targeted 要素の値は、「_VALUE」(アンダーラインに注意) 下の文字列変数として式で使用可能です。

ここで、Order XML メッセージを使用して完全な XML から Java バインディングへの設定について学びましょう。XML メッセージから投入する Java オブジェクトは次の通りです (「getter」と「setter」は省略されています)。

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Date date;
    private Long customerNumber;
    private String customerName;
    private double total;
}

public class OrderItem {
    private long productId;
    private Integer quantity;
    private double price;
}
```

データを order XML からオブジェクトモデルにバインドする時に使用しなければならない **Smooks** 設定は次の通りです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

(1)  <jb:bean BeanId="order" class="com.acme.Order"
createOnElement="order">
(1.a)    <jb:wiring property="header" BeanIdRef="header" />
(1.b)    <jb:wiring property="orderItems" BeanIdRef="orderItems" />
        </jb:bean>

(2)  <jb:bean BeanId="header" class="com.acme.Header"
createOnElement="order">
(2.a)    <jb:value property="date" decoder="Date" data="header/date">
          <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
        </jb:value>
(2.b)    <jb:value property="customerNumber"
data="header/customer/@number" />
(2.c)    <jb:value property="customerName" data="header/customer" />
(2.d)    <jb:expression property="total" execOnElement="order-item" >
          += (orderItem.price * orderItem.quantity);
        </jb:expression>
        </jb:bean>

(3)  <jb:bean BeanId="orderItems" class="java.util.ArrayList"
createOnElement="order">
(3.a)    <jb:wiring beanType="com.acme.OrderItem" /> <!-- Could also wire
using BeanIdRef="orderItem" -->
```



```

        </jb:bean>

(4)    <jb:bean BeanId="orderItem" class="com.acme.OrderItem"
createOnElement="order-item">
(4.a)    <jb:value property="productId" data="order-item/product" />
(4.b)    <jb:value property="quantity" data="order-item/quantity" />
(4.c)    <jb:value property="price" data="order-item/price" />
        </jb:bean>

</smooks-resource-list>

```

- 設定 (1) は com.acme.Order Bean インスタンス (トップレベル Bean) の作成ルールを定義します。この Bean インスタンスをメッセージの一番始め (order 要素) に作成します。

各 Bean インスタンスを作成します。(1)、(2)、(3) すべてはメッセージの一番始めの (4) を許可します (order 要素上)。これを行うのは、投入されたモデルではこれら Bean の単一インスタンスのみが存在するためです。

- 設定 (1.a) と (1.b) は、Header (2) および ListOrderItem Bean インスタンス (3) を order Bean インスタンスにワイヤリングするワイヤリング設定を定義します (beanIdRef 属性の値と、(2) と (3) に定義された beanId 値の参照方法を確認してください)。(1.a) と (1.b) のプロパティ属性はワイヤリングが行われる Order Bean プロパティを定義します。

Java クラスタイプ (beanType) を基に Bean をオブジェクトへワイヤリングしたり、特定のアノテーションを付けて (beanAnnotation) Bean をオブジェクトへワイヤリングすることも可能です。

設定 (2) は com.acme.Header Bean インスタンスを作成します。

- 設定 (2.a) は、Header.date プロパティにバインドする値を定義します。どこでバインディング値がソースメッセージから選択されるかはデータ属性が定義することに注意してください。この例では header/date 要素からになります。また decodeParam サブ要素を定義する方法にも注目してください。これは DateDecoder を設定します。
- 設定 (2.b) は、Header.customerNumber プロパティへの値バインディング設定を定義します。ソースメッセージにある要素属性よりバインディング値を選択するため、データ属性を設定する方法に注目してください。

設定 (2.b) は、注文合計が算出され Header.total プロパティに設定される式バインディングを定義します。execOnElement 属性は、order-item 要素上で式を評価 (およびバインド/再バインド) する必要があることを Smooks に伝えます。そのため、複数の order-item 要素がソースメッセージにある場合、この式は各 order-item に実行され、新しい合計値が Header.total プロパティに再バインドされます。式が現在の orderItem の合計を現在の注文合計 (header.total) に追加する方法に注目してください。

- 設定 (2.d) は、各注文商品の合計 (数量に価格を掛けた値) を現在の合計に足してこれまでの合計が算出される式バインディングを定義します。設定 (3) は、OrderItem インスタンスを保持するための ListOrderItem Bean インスタンスを作成します。
- 設定 (3.a) は、com.acme.OrderItem タイプのすべての Bean ((4) など) をリストへワイヤリングします。このワイヤリングはプロパティ属性を定義しないことに注意してください。これは、コレクションへワイヤリングされるためです (アレイへワイヤリングされる場合も同様です)。また、beanType 属性の代わりに BeanIdRef 属性を使用してもこのワイヤリングを実行できたことに注目してください。
- 設定 (4) は OrderItem Bean インスタンスを作成します。createOnElement プロパティがどのように order-item 要素に設定されるか注目してください。これは、各 order-item 要素に対して

この Bean の新しいインスタンスが作成されるようにするためです (そして、ListOrderItem (3.a) ヘワイヤリングするためです)。

この設定の createOnElement 属性が order-item 要素に設定されないと (たとえば order、header、order-item 要素の 1 つに設定された場合)、単一の OrderItem Bean インスタンスのみが作成され、バインディング設定 (4.a など) がソースメッセージの各 order-item 要素に対する Bean インスタンスのプロパティバインディングを上書きします。たとえば、ソースメッセージにある最後の order-item からの order-item データが含まれる単一の OrderItem インスタンスのみがある ListOrderitem となります。

バインディングの秘訣の一部は次の通りです。

- モデルに単一の Bean インスタンスのみが存在する場合は、jb:bean createOnElement を root 要素 (または #document) に設定します。

これを コレクション Bean インスタンス の recurring 要素に設定します。



警告

この場合、正しい要素を指定しないとデータが失われる可能性があります。

- jb:value デコーダ:ほとんどの場合、Smooks は jb:value バインディングで使用するデータタイプデコーダを自動的に検出しますが、設定が必要になるデコーダもあります (DateDecoder [decoder="Date"] がその 1 つです)。このような場合、バインディング上でデコーダー属性を定義する必要があります (そのデコーダのデコードパラメータを指定するための jb:decodeParam 子要素も定義する必要があります)。
- コレクションヘバインドする時、jb:wiring プロパティは必要ありません。
- 必要なコレクションタイプを設定するには、jb:bean クラスを定義し、コレクションエントリでワイヤリングします。アレイの場合、jb:bean クラス属性値に角括弧を用いて postfix に対応します (例: `class="com.acme.OrderItem[]"`)。

DataDecoder 実装は **DataEncoder** インターフェースを実装することもできます。名前の通り、DataEncoder はオブジェクト値を文字列へエンコードしフォーマットするメソッドを実装します。

使用可能な DataDecoder/DataEncoder 実装は次の通りです。

- Date: 文字列を java.util.Date インスタンスへデコードまたはエンコードします。
- Calendar: 文字列を java.util.Calendar インスタンスへデコードまたはエンコードします。
- SqlDate: 文字列を java.sql.Date インスタンスへデコードまたはエンコードします。
- SqlTime: 文字列を java.sql.Time インスタンスへデコードまたはエンコードします。
- SqlTimestamp: 文字列を java.sql.Timestamp インスタンスへデコードまたはエンコードします。

これらの実装はすべて同じように設定します。

Date の例は次の通りです。

```
<jb:value property="date" decoder="Date" data="order/@date">
  <jb:decodeParam name="format">EEE MMM dd HH:mm:ss Z
  yyyy</jb:decodeParam>
  <jb:decodeParam name="locale">sv_SE</jb:decodeParam>
</jb:value>
```

SqlTimestamp の例は次の通りです。

```
<jb:value property="date" decoder="SqlTimestamp" data="order/@date">
  <jb:decodeParam name="format">EEE MMM dd HH:mm:ss Z
  yyyy</jb:decodeParam>
  <jb:decodeParam name="locale">sv</jb:decodeParam>
</jb:value>
```

注記

decodeParam iformat は、日付形式の ISO 8601 を基にしています。

ロケールの decodeParam 値はアンダースコアで区切られた文字列で、最初のトークンがロケールの ISO 言語コード、2 つ目のトークンが ISO 国コードになります。

decodeParam は言語と国の 2 つのパラメータとして指定することも可能です。

```
<jb:value property="date" decoder="Date" data="order/@date">
  <jb:decodeParam name="format">EEE MMM dd HH:mm:ss Z
  yyyy</jb:decodeParam>
  <jb:decodeParam name="locale-language">sv</jb:decodeParam>
  <jb:decodeParam name="locale-country">SE</jb:decodeParam>
</jb:value>
```

使用可能な数字ベースの DataDecoder/DataEncoder 実装は次の通りです。

- BigDecimalDecoder: 文字列を java.math.BigDecimal インスタンスへデコードまたはエンコードするのに使用します。
- BigIntegerDecoder: 文字列を java.math.BigInteger インスタンスへデコードまたはエンコードするのに使用します。
- DoubleDecoder: 文字列を java.lang.Double インスタンス (プリミティブを含む) ヘデコードまたはエンコードするのに使用します。
- FloatDecoder: 文字列を java.lang.Float インスタンス (プリミティブを含む) ヘデコードまたはエンコードするのに使用します。
- IntegerDecoder: 文字列を java.lang.Integer インスタンス (プリミティブを含む) ヘデコードまたはエンコードするのに使用します。
- LongDecoder: 文字列を java.lang.Long インスタンス (プリミティブを含む) ヘデコードまたはエンコードするのに使用します。
- ShortDecoder: 文字列を java.lang.Short インスタンス (プリミティブを含む) ヘデコードまたはエンコードするのに使用します。

これらの実装はすべて同じように設定します。

以下は BigDecimal の例になります。

```
<jb:value property="price" decoder="BigDecimal" data="orderItem/price">
  <jb:decodeParam name="format">#,###.##</jb:decodeParam>
  <jb:decodeParam name="locale">en_IE</jb:decodeParam>
</jb:value>
```

以下は Integer の例になります。

```
<jb:value property="percentage" decoder="Integer" data="vote/percentage">
  <jb:decodeParam name="format">#%</jb:decodeParam>
</jb:value>
```

注記

decodeParam の形式は NumberFormat パターンの構文を基にしています。

ロケールの decodeParam 値はアンダースコアで区切られた文字列で、最初のトークンがロケールの ISO 言語コード、2 目のトークンが ISO 国コードになります。

decodeParam を言語と国の 2 つのパラメータとして指定することも可能です。

```
<jb:value property="price" decoder="Double"
data="orderItem/price">
  <jb:decodeParam name="format">#,###.##</jb:decodeParam>
  <jb:decodeParam name="locale-language">sv</jb:decodeParam>
  <jb:decodeParam name="locale-country">SE</jb:decodeParam>
</jb:value>
```

入力メッセージのデータを基に、異なる値をオブジェクトモデルにバインドしたい場合があります。この場合、式ベースのバインディングを使用することができますが、次のようにマッピングデコーダを使用することも可能です。

```
<jb:value property="name" decoder="Mapping" data="history/@warehouse">
  <jb:decodeParam name="1">Dublin</jb:decodeParam>
  <jb:decodeParam name="2">Belfast</jb:decodeParam>
  <jb:decodeParam name="3">Cork</jb:decodeParam>
</jb:value>
```

上記の例では、入力データ値である「1」が「Dublin」という値として name プロパティへマッピングされます。値「2」と「3」も同様です。

列挙デコーダは特殊なマッピングデコーダです。データ入力値が列挙値/名へ正確にマッピングする場合、列挙は通常自動的にデコードされます (特別な設定は必要ありません)。これ以外の場合、入力データ値から列挙値/名へのマッピングを定義する必要があります。

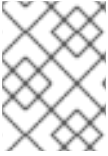
次の例では、入力メッセージの header/priority フィールドには "LOW"、"MEDIUM"、"HIGH" という値が含まれています。これらの値を LineOrderPriority の列挙値である "NOT_IMPORTANT"、"IMPORTANT"、"VERY_IMPORTANT" へそれぞれマッピングする必要があります。

```
<jb:value property="priority" data="header/priority" decoder="Enum">
  <jb:decodeParam
name="enumType">example.trgmodel.LineOrderPriority</jb:decodeParam>
```

```

<jb:decodeParam name="LOW">NOT_IMPORTANT</jb:decodeParam>
<jb:decodeParam name="MEDIUM">IMPORTANT</jb:decodeParam>
<jb:decodeParam name="HIGH">VERY_IMPORTANT</jb:decodeParam>
</jb:value>

```



注記

マッピングが必要な場合、enumType decodeParam を使用して列挙型を指定する必要もあります。

デフォルトでは、Bean を作成した断片 (createOnElement) が処理された後、Smooks 設定の最初にある Bean 以外の Bean はすべて BeanContext から削除されます (createOnElement 断片の start/visitBefore で BeanContext に Bean が追加され、end/visitAfter で BeanContext より削除されます)。

デフォルトでは、Smooks 設定の最初に設定された Bean を除くすべての Bean にこのルールが適用されます (デフォルトでは、最初の Bean のみが BeanContext に保持されるため、メッセージが処理された後もこの Bean にアクセスできます)。

この挙動を変更するには、jb:bean 要素の retain configuration 属性を使用します。この属性を使用すると、Smooks BeanContext 内で Bean の保持を手動制御できます。

Java Bean カートリッジ:

- ソース/入力メッセージストリームより文字列値を抽出します。
- decoder および decodeParam の設定を基に文字列値をデコードします (これらの設定が定義されていないと、デコーダをリフレクションで解決しようとします)。
- ターゲット Bean にデコードされた値を設定します。

デコードの手順を実行する前に、文字列データ値を *事前処理* する必要がある場合があります。

事前処理が必要となる場合の例には、ソースデータに数値デコードのロケール設定によってサポートされない文字がある場合などがあります (たとえば、数値 876592.00 がなんらかの理由で "876_592!00" jb:bean 要素として表される場合など)。この値を二重の値としてデコードするには、アンダースコアと感嘆符を削除し、感嘆符をピリオドに置き換えます。

カスタム DataDecoder を書いて対処するのも 1 つの手段ですが (この操作を繰り返し行う必要がある場合に推奨されます)、迅速な対応が必要な場合は valuePreprocess decodeParam を指定できます。これは、デコードを行う前に文字列値に適用できる簡単な式です。

以下は、前述の数値デコードの問題を解決する一例となります。

```

<!-- A bean property binding example: -->
<jb:bean BeanId="orderItem" class="org.milyn.javabean.OrderItem"
createOnElement="price">
  <jb:value property="price" data="price" decoder="Double">
    <jb:decodeParam name="valuePreprocess">value.replace("_",
    "").replace("!", ".")</jb:decodeParam>
  </jb:value>
</jb:bean>

```

```

<!-- A direct value binding example: -->
<jb:value BeanId="price" data="price" decoder="BigDecimal">

```

```
<jb:decodeParam name="valuePreprocess">value.replace("_",
""").replace("!", ".")</jb:decodeParam>
</jb:value>
```



注記

上記の例では、文字列データ値は値変数名を介して式で参照されます (式は、値文字列上で操作し文字列を返す有効な MVEL 式になります)。

Java Bean カートリッジでは ファクトリを使用して Bean を作成できます。このような場合、パブリックパラメータを持たないコンストラクタを使用する必要はありません。クラス属性で実際のクラス名を定義する必要もありません。オブジェクトのインターフェースを使用すれば十分ですが、そのインターフェースのメソッドへバインドすることのみ可能です (ファクトリを定義しても、常に Bean 定義にクラス属性を設定する必要があります)。

ファクトリ定義は Bean 要素のファクトリ属性で設定されます。デフォルトのファクトリ定義言語は次のようになります。

```
some.package.FactoryClass#staticMethod{.instanceMethod}
```

この基本的な定義言語を使用して、Bean を作成するために Smooks が呼び出す静的なパブリックパラメータを持たないメソッドを定義します。instanceMethod の部分は任意です。これを設定すると、静的メソッドより返されるオブジェクトを呼び出すメソッドを定義し、Bean が作成されるはずです。{ } 文字は任意の部分を示す目的でのみ使用されているため、実際の定義では使用しないようにしてください。

次の例は、静的ファクトリメソッドを使用して ArrayList オブジェクトをインスタンス化する方法を表しています。

```
<jb:bean
  BeanId="orders"
  class="java.util.List"
  factory="some.package.ListFactory#newList"
  createOnElement="orders"
>
  <!-- ... bindings -->
</jb:bean>
```

"some.package.ListFactory#newList" ファクトリ定義は、Bean を作成するには **some.package.ListFactory** クラス上で **newList** メソッドを呼び出す必要があることを定義します。クラス属性は Bean を List オブジェクトとして定義します。特定タイプの List オブジェクト (ArrayList や LinkedList など) は ListFactory 自体によって決定されます。

別の例を見てみましょう。

```
<jb:bean
  BeanId="orders"
  class="java.util.List"
  factory="some.package.ListFactory#getInstance.newList"
  createOnElement="orders"
>
  <!-- ... bindings -->
</jb:bean>
```

これは、静的メソッド **getInstance** を使用して ListFactory のインスタンスを読み出す必要があり、ListFactory オブジェクト上で **newList** メソッドを呼び出して List オブジェクトを作成する必要があることを定義します。このコンストラクトでは シングルトンファクトリを使用できます。

異なる定義言語を使用した後にデフォルトの基本言語を使用することができます。たとえば、MVEL をファクトリ定義言語として使用できます。

使用したい定義言語を宣言する方法は 3 つあります。

定義言語はエイリアスを持つことが可能です。たとえば、MVEL のエイリアスは「mvel」です。特定のファクトリ定義に MVEL を使用したい場合、

「mvel:some.package.ListFactory.getInstance().newList()」のように定義の前に「mvel:」を付けます。デフォルトの基本言語のエイリアスは「basic」です。

言語をグローバルなデフォルトとして設定するには、「factory.definition.parser.class」グローバルパラメータを、使用したい言語の FactoryDefinitionParser インターフェースを実装するクラスの完全クラスパスに設定する必要があります。

注記:「:」が含まれるデフォルト言語を持つ定義がある場合、その定義の前に「default:」を付けないと例外が発生します。

エイリアスを使用する代わりに、

「org.milyn.javabean.factory.MVELFactoryDefinitionParser:some.package.ListFactory.getInstance().newList」のように、使用したい言語の FactoryDefinitionParser インターフェースを実装するクラスの完全クラスパスに設定することも可能ですが、これはテストの目的でのみ使用することが推奨されます。言語にエイリアスを定義した方がよいでしょう。

独自の言語を定義したい場合は、「org.milyn.javabean.factory.FactoryDefinitionParser」インターフェースを実装する必要があります。例として、

「org.milyn.javabean.factory.MVELFactoryDefinitionParser」や
「org.milyn.javabean.factory.BasicFactoryDefinitionParser」を参照してみてください。

定義言語のエイリアスを定義するには、「org.milyn.javabean.factory.Alias」アノテーションとエイリアス名を FactoryDefinitionParser クラスに追加します。

Smooks がエイリアスを検索できるようにするには、クラスパスのルート上に「META-INF/smooks-javabean-factory-definition-parsers.inf」というファイルを作成する必要があります。このファイルには、Alias アノテーションを持つ FactoryDefinitionParser インターフェースを実装するすべてのファイルの完全クラスパスが含まれている必要があります (新しい行で区切ります)。

MVEL には基本のデフォルト定義言語より優れた利点がいくつかあります。たとえば、ファクトリオブジェクトとして Bean コンテキストよりオブジェクトを使用したり、パラメータを用いてファクトリメソッドを呼び出すことが可能です。これらのパラメータは定義内に定義することが可能で、Bean コンテキストからのオブジェクトであることも可能です。MVEL を使用できるようにするには、「mvel」をエイリアスとして使用するか、「factory.definition.parser.class」グローバルパラメータを「org.milyn.javabean.factory.MVELFactoryDefinitionParser」に設定します。

前述と同じユースケースで MVEL を用いた例は次の通りです。

```
<smooks-resource-list
  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

  <jb:bean
    BeanId="orders"
    class="java.util.List"
    factory="mvel:some.package.ListFactory.getInstance().newList()"
```

```

        createOnElement="orders"
    >
        <!-- ... bindings -->
    </jb:bean>

</smooks-resource-list>

```

次の例では、MVEL を使用して Bean コンテキストの既存 Bean より List オブジェクトを抽出します。この例の Order オブジェクトには、order 行の追加に使用しなければならないリストを返すメソッドが含まれています。

```

<smooks-resource-list
  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

  <jb:bean
    BeanId="order"
    class="some.package.Order"
    createOnElement="order"
  >
    <!-- ... bindings -->
  </jb:bean>

  <!--
    The factory attribute uses MVEL to access the order
    object in the bean context and calls its getOrderLines()
    method to get the List. This list is then added to the bean
    context under the BeanId 'orderLines'
  -->
  <jb:bean
    BeanId="orderLines"
    class="java.util.List"
    factory="mvel:order.getOrderLines()"
    createOnElement="order"
  >
    <jb:wiring BeanIdRef="orderLine" />
  </jb:bean>

  <jb:bean
    BeanId="orderLine"
    class="java.util.List"
    createOnElement="order-line"
  >
    <!-- ... bindings -->
  </jb:bean>

</smooks-resource-list>

```

MVEL をデフォルトのファクトリ定義言語として使用しないのはなぜか不思議に思っている方がいるかもしれません。現在、基本の定義言語と MVEL のパフォーマンスは同等です。基本の定義言語の方が高速でない理由は、現在リフレクションを使用してファクトリメソッドを呼び出すためです。しかし、リフレクションの代わりにバイトコード生成を使用する計画があります。これにより、パフォーマンスが大幅に改善されるはずです。MVEL がデフォルトの言語となっていると、基本の定義言語が提供する基本機能のみが必要な場合にパフォーマンスを改善することができなくなります。

アレイオブジェクトはサポートされません。ファクトリがアレイを返す場合、Smooks はある時点で例外をスローします。

キーと値ペアをマップにバインドする

バインディングの `jb:value` プロパティ属性が定義されていない場合 (または空の場合)、選択されたノード名がマップエントリのキーとして使用されます (`beanClass` がマップである場合)。

マップキーを定義する方法はもう 1 つあります。 `jb:value` プロパティ属性の値を `@` 文字で始めることです。 `@` 以降の値は、マップキーが選択される選択したノードの属性名を定義します。例は次の通りです。

```
<root>
  <property name="key1">value1</property>
  <property name="key2">value2</property>
  <property name="key3">value3</property>
</root>
```

設定は次のようになります。

```
<jb:bean BeanId="keyValuePairs" class="java.util.HashMap"
createOnElement="root">
  <jb:value property="@name" data="root/property" />
</jb:bean>
```

これにより、設定されたキー [key1, key2, key3] の 3 つのエントリを持つハッシュマップが作成されます。

`@` 文字表記は Bean ワイヤリングでは機能しません。カートリッジは `@` 文字を含むプロパティ属性の値をマップエントリキーとして使用します。

独自の Bean クラスを記述せずに完全なオブジェクトモデルを作成することは可能です。この仮想モデルは、マップとリストのみを使用して作成されます。たとえば、`xml->java->xml` や `xml->java->edi` などのモデル駆動型のトランスフォメーションの一部としてなど、2 つのプロセス手順の間で Javabeen カートリッジを使用する場合にとっても便利です。

この原理について次の例を参照してください。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

  <!--
    Bind data from the message into a Virtual Object model in the bean
    context....
  -->
  <jb:bean BeanId="order" class="java.util.HashMap"
createOnElement="order">
    <jb:wiring property="header" BeanIdRef="header" />
    <jb:wiring property="orderItems" BeanIdRef="orderItems" />
  </jb:bean>
  <jb:bean BeanId="header" class="java.util.HashMap"
```

```

createOnElement="order">
    <jb:value property="date" decoder="Date" data="header/date">
        <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
    </jb:value>
    <jb:value property="customerNumber" decoder="Long"
data="header/customer/@number" />
    <jb:value property="customerName" data="header/customer" />
    <jb:expression property="total" execOnElement="order-item" >
        header.total + (orderItem.price * orderItem.quantity);
    </jb:expression>
</jb:bean>
<jb:bean BeanId="orderItems" class="java.util.ArrayList"
createOnElement="order">
    <jb:wiring BeanIdRef="orderItem" />
</jb:bean>
<jb:bean BeanId="orderItem" class="java.util.HashMap"
createOnElement="order-item">
    <jb:value property="productId" decoder="Long" data="order-
item/product" />
    <jb:value property="quantity" decoder="Integer" data="order-
item/quantity" />
    <jb:value property="price" decoder="Double" data="order-
item/price" />
</jb:bean>

<!--
    Use a FreeMarker template to perform the model driven transformation
on the Virtual Object Model...
-->
<ftl:freemarker applyOnElement="order">
    <ftl:template>/templates/orderA-to-orderB.ftl</ftl:template>
</ftl:freemarker>

</smooks-resource-list>

```

上記の例では、どのように仮想モデル (マップ) のデコード属性を常に定義するか注目してください。Smooks は、データをマップへバインドするためのデコードタイプを自動検出できないため、型付けされた値を仮想モデルにバインドする必要がある場合は適切なデコードを指定する必要があります。このような場合にデコードが指定されないと、Smooks はデータを文字列として仮想モデルにバインドします。

model-driven-basic と model-driven-basic-virtual の例を見てみましょう。

複数のデータエントリを単一のバインディングに統合する

複数のデータエントリを単一のバインディングに統合するには、表現ベースバインディング (jb:expression) を使用します。

Smooks 1.3 より、Javabeen カートリッジに直接値バインディング (Direct value binding) と呼ばれる新しい機能が導入されました。直接値バインディングは Smooks DataDecoder を使用して選択された data 要素/属性よりオブジェクトを作成し、直接 Bean コンテキストへ追加します。

ValueBinder クラスは値バインディングを行うビジターです。

値バインディングの XML 設定は、Smooks 1.3 の JavaBean スキーマの一部です (<http://www.milyn.org/xsd/smooks/javabean-1.4.xsd>)。値バインディングの要素は value です。

値には次のような属性があります。

- BeanId: 作成されたオブジェクトが Bean コンテキストでバインドされる ID。
- data: バインドされるデータ値のデータセクタ。例: "order/orderid"、"order/header/@date" など。
- dataNS: "data" セクタの名前空間。
- decoder: 値を文字列から異なるタイプへ変換するための DataDecoder 名。DataDecoder は decodeParam 要素を用いて設定できます。
- default: 選択されたデータが null または空の文字列であった場合のデフォルト値です。

例

「典型的な」注文メッセージを例とし、注文番号、名前、および日付を整数 (Integer) と文字列 (String) の形式で値オブジェクトとして取得します。

メッセージ

```
<order xmlns="http://x">
  <header>
    <y:date xmlns:y="http://y">Wed Nov 15 13:45:28 EST 2006</y:date>
    <customer number="123123">Joe</customer>
    <privatePerson></privatePerson>
  </header>
  <order-items>
    <!-- .... -->
  </order-items>
</order>
```

設定

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

  <jb:value
    BeanId="customerName"
    data="customer"
    default="unknown"
  />

  <jb:value
    BeanId="customerNumber"
    data="customer/@number"
    decoder="Integer"
  />

  <jb:value
    BeanId="orderDate"
    data="date"
    dateNS="http://y"
    decoder="Date"
  >
```

```

        <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
        <jb:decodeParam name="locale-language">en</jb:decodeParam>
        <jb:decodeParam name="locale-country">IE</jb:decodeParam>
    </jb:value>

</smooks-resource-list>

```

値バインダーは `org.milyn.javabean`. 値オブジェクトを使用してプログラムを用いて設定することができます。

xml 設定の例と同じメッセージの例を使用します。

```

//Create Smooks. normally done globally!
Smooks smooks = new Smooks();

//Create the Value visitors
Value customerNumberValue = new Value( "customerNumber",
"customer/@number").setDecoder("Integer");
Value customerNameValue = new Value( "customerName",
"customer").setDefault("Unknown");

//Add the Value visitors
smooks.addVisitors(customerNumberValue);
smooks.addVisitors(customerNameValue);

//And the execution code:
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(orderMessageStream), result);
Integer customerNumber = (Integer) result.getBean("customerNumber");
String customerName = (String) result.getBean("customerName");

```

Java バインディングの設定は、Bean 設定クラスを使用するとプログラムを用いて Smooks に追加できます。

このクラスを使用すると、特定クラス上で Java バインディングを実行するための Smooks インスタンスをプログラムを用いて設定することができます。グラフに投入するには、Bean 同士をバインディングして Bean インスタンスのグラフを作成します。Bean クラスは Fluent API を使用します (すべてのメソッドが Bean インスタンスを返します)。そのため、設定同士を「文字列化」して Bean 設定グラフを構築することが容易になります。

「典型的な注文メッセージを例とし、対応する Java オブジェクトモデルへバインドします。

メッセージ

```

<order xmlns="http://x">
  <header>
    <y:date xmlns:y="http://y">Wed Nov 15 13:45:28 EST 2006</y:date>
    <customer number="123123">Joe</customer>
    <privatePerson></privatePerson>
  </header>
  <order-items>
    <order-item>
      <product>111</product>
      <quantity>2</quantity>
    </order-item>
  </order-items>
</order>

```

```

        <price>8.90</price>
    </order-item>
    <order-item>
        <product>222</product>
        <quantity>7</quantity>
        <price>5.20</price>
    </order-item>
</order-items>
</order>

```

Java モデル (getter/setter は省略)

```

public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Long customerNumber;
    private String customerName;
}

public class OrderItem {
    private long productId;
    private Integer quantity;
    private double price;
}

```

設定コード

```

Smooks smooks = new Smooks();

Bean orderBean = new Bean(Order.class, "order", "/order");

orderBean.bindTo("header",
    orderBean.newBean(Header.class, "/order")
        .bindTo("customerNumber", "header/customer/@number")
        .bindTo("customerName", "header/customer")
    ).bindTo("orderItems",
    orderBean.newBean(ArrayList.class, "/order")
        .bindTo(orderBean.newBean(OrderItem.class, "order-item")
            .bindTo("productId", "order-item/product")
            .bindTo("quantity", "order-item/quantity")
            .bindTo("price", "order-item/price"))
    );

smooks.addVisitors(orderBean);

```

実行コード

```

JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(orderMessageStream), result);
Order order = (Order) result.getBean("order");

```

The API supports factories. You can provide a factory object of the type `org.milyn.javabean.factory.Factory`, that will be called when a new bean instance needs to be created.

Here is an example where an anonymous Factory class is defined and used:

```
Bean orderBean = new Bean(Order.class, "order", "/order", new
Factory<Order>() {

    public Order create(ExecutionContext executionContext) {
        return new Order();
    }

});
```

Java Bean カートリッジには、バインディング設定テンプレートを生成するため使用できる `org.milyn.javabean.gen.ConfigGenerator` ユーティリティクラスが含まれています。このテンプレートをバインディング定義の基準として使用することができます。

コマンドラインからの実行

```
$JAVA_HOME/bin/java -classpath <classpath>
org.milyn.javabean.gen.ConfigGenerator -c <rootBeanClass> -o
<outputFilePath> [-p <propertiesFilePath>]
```

- `-c` コマンドライン引数は、バインディング設定が生成されるモデルのルートクラスを指定します。
- `-o` コマンドライン引数は生成された設定出力のパスとファイル名を指定します。
- `-p` コマンドライン引数は追加のバインディングパラメータを指定する、パスとファイル名の任意バインディング設定ファイルを指定します。
- 任意の `-p` プロパティファイルパラメータを使用すると追加の設定パラメータを指定できます。
- `packages.included`: セミコロン区切りのパッケージリストです。生成されたバインディング設定にこれらのパッケージと一致するクラスのフィールドが含まれます。
- `packages.excluded`: セミコロン区切りのパッケージリストです。生成されたバインディング設定にはこれらのパッケージと一致するクラスのフィールドは含まれません。

バインディング設定がソースデータモデルで機能するようにするため、ターゲットクラスに対してこのユーティリティを実行した後、通常次のフォローアップタスクを実行する必要があります。

各 `jb:bean` 要素に対し、Bean インスタンスの作成に使用する必要がある `event` 要素へ `createOnElement` 属性を設定します。

`jb:value` データ属性を更新し、Bean プロパティのバインディングデータを提供する `event` 要素/属性を選択します。

`jb:value` デコーダ属性をチェックします。実際のプロパティタイプに応じて設定されるため属性はすべて設定されません。これらの属性は手作業で設定する必要があります。たとえば、データフィールドなど、バインディングの一部にデコーダの `jb:decodeParam` サブ要素を設定する必要がある場合があります。

バインディング設定要素 (jb:value および jb:wiring) を再度チェックします。必ず、生成された設定にすべての Java プロパティが含まれるようにします。

セクタ値の判定は特に Java のような非 XML ソース (Java など) では難しいことがあります。このような場合、HTML Reporting ツールを使用すると便利です。このツールは Smooks で表示されるように入力メッセージモデル (セクタが適用されます) を想定することができます。最初に、空のトランスフォーメーション設定を用いてソースデータを使用してレポートを生成します。このレポートには、設定を追加する必要があるモデルが記載されます。設定は 1 つずつ追加し、適用されたことを確認するためレポートを再実行します。

生成された設定の例は次の通りです。"\$TODO\$" トークンに注目してください。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean BeanId="order" class="org.milyn.javabean.Order"
createOnElement="$TODO$">
        <jb:wiring property="header" BeanIdRef="header" />
        <jb:wiring property="orderItems" BeanIdRef="orderItems" />
        <jb:wiring property="orderItemsArray" BeanIdRef="orderItemsArray"
/>
    </jb:bean>

    <jb:bean BeanId="header" class="org.milyn.javabean.Header"
createOnElement="$TODO$">
        <jb:value property="date" decoder="$TODO$" data="$TODO$" />
        <jb:value property="customerNumber" decoder="Long" data="$TODO$"
/>
        <jb:value property="customerName" decoder="String" data="$TODO$"
/>
        <jb:value property="privatePerson" decoder="Boolean" data="$TODO$"
/>
        <jb:wiring property="order" BeanIdRef="order" />
    </jb:bean>

    <jb:bean BeanId="orderItems" class="java.util.ArrayList"
createOnElement="$TODO$">
        <jb:wiring BeanIdRef="orderItems_entry" />
    </jb:bean>

    <jb:bean BeanId="orderItems_entry"
class="org.milyn.javabean.OrderItem" createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" BeanIdRef="order" />
    </jb:bean>

    <jb:bean BeanId="orderItemsArray"
class="org.milyn.javabean.OrderItem[]" createOnElement="$TODO$">
        <jb:wiring BeanIdRef="orderItemsArray_entry" />
    </jb:bean>

    <jb:bean BeanId="orderItemsArray_entry"
class="org.milyn.javabean.OrderItem" createOnElement="$TODO$">
```

```

        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" BeanIdRef="order" />
    </jb:bean>

</smooks-resource-list>

```

Smooks.filterSource メソッドが呼び出された後、JavaResult インスタンスの正確な内容は保証されません。このメソッドの呼び出し後、JavaResult インスタンスにはビジター実装によって追加できる Bean コンテキストの最終内容が含まれます。

jb:result 設定を Smooks の設定で使用すると JavaResult に返される Bean セットを制限できます。次の設定例では、ResultSet に order Bean のみを保持するよう Smooks に伝えます。

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd">

    <!-- Capture some data from the message into the bean context... -->
    <jb:bean BeanId="order" class="com.acme.Order"
createOnElement="order">
        <jb:value property="orderId" data="order/@id"/>
        <jb:value property="customerNumber"
data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItems" BeanIdRef="orderItems"/>
    </jb:bean>
    <jb:bean BeanId="orderItems" class="java.util.ArrayList"
createOnElement="order">
        <jb:wiring BeanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean BeanId="orderItem" class="com.acme.OrderItem"
createOnElement="order-item">
        <jb:value property="itemId" data="order-item/@id"/>
        <jb:value property="productId" data="order-item/product"/>
        <jb:value property="quantity" data="order-item/quantity"/>
        <jb:value property="price" data="order-item/price"/>
    </jb:bean>

    <!-- Only retain the "order" bean in the root of any final JavaResult.
-->
    <jb:result retainBeans="order"/>

</smooks-resource-list>

```

この設定が適用された後、order Bean 以外の JavaResult.getBean(String) メソッドへの呼び出しは null を返すようになります。他の Bean インスタンスは「order」グラフヘワイヤリングされるため、上記のような例では適切に挙動します。

Smooks v1.2 より、JavaSource インスタンスが Smooks.filterSource メソッドへ提供されると (フィルターソースインスタンスとして)、Smooks は JavaSource を使用して、Smooks.filterSource 呼び出しに対し ExecutionContext に関連する Bean コンテキストを構築するようになりました。そのため、JavaSource Bean インスタンスの一部は JavaResult で可視できるようになりました。

第6章 テンプレート

本章では、Smooks で使用できる FreeMarker (<http://freemarker.org>) と XSL テンプレート (<http://www.w3.org/Style/XSL/>) の 2 種類のテンプレートについて説明します。テンプレートの適用方法を完全に理解するには、Java バインディングの知識が必要となります。

これらの技術は Smooks フィルタリング処理のコンテキスト内で使用することができます。よって次のことが言えます。

- 「断片ごと」にソースメッセージへ適用することができます。これはメッセージ全体に適用される断片ベースのトランスフォーメーションプロセスとは対照的です。SOAP メッセージにヘッダーを追加する時など、特定時にメッセージヘッダーを挿入する必要がある場合に断片ごとの適用は便利です。この場合、他の断片を混乱させることなくプロセスを使用して希望の断片を「狙う」ことができます。
- メッセージデータをデコードし、Bean コンテキストへバインドできる Java Bean カートリッジなど、Smooks のその他の技術を利用することができます。**FreeMarker** テンプレート内よりデコードされたデータへの参照を作成します (Smooks は **FreeMarker** がデータを利用できるようにします)。
- 比較的単純な処理モデルと小型のメモリフットプリントを維持しながら、同時に「巨大」メッセージストリーム (数ギガバイト以上) を処理するために使用できます。
- 分割メッセージ断片の生成に使用できます。分割メッセージ断片は、エンタープライズサービスバスにある物理または論理エンドポイントヘルパーティングすることができます。

6.1. FREEMARKER テンプレート

FreeMarker は大変強力なテンプレートエンジンです。Smooks はテキストベースのコンテンツを生成するため **FreeMarker** を使用します。その後、このコンテンツをメッセージストリームへ挿入することができます (この処理は断片ベーストランスフォーメーションと呼ばれます)。この処理は、別プロセスへの後続のルーティングに対する分割メッセージの断片にも使用できます。

設定名前空間 <http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd> より Smooks の **FreeMarker** テンプレートを設定します。その後、使用を開始するため統合開発環境で XSD を設定します。

例6.1 例 - インラインテンプレート

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
  <ftl:freemarker applyOnElement="order">
    <ftl:template><!--<orderId>${order.id}</orderId>-->
  </ftl:template>
</ftl:freemarker>
</smooks-resource-list>
```

例6.2 例 - 外部テンプレートの参照

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
  <ftl:freemarker applyOnElement="order">
```

```

        <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    </ftl:freemarker>
</smooks-resource-list>

```

結果出力時に Smooks が複数の操作を実行できるようにするため、`<use>` 要素を `<ftl:freemarker>` 設定に追加します。

例6.3 例 - テンプレート結果の「インライン化」

```

<ftl:freemarker applyOnElement="order">
    <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    <ftl:use>
        <ftl:inline directive="insertbefore" />
    </ftl:use>
</ftl:freemarker>

```

名前の通り、インラインはテンプレートの結果を **Smooks.filterSource** の結果へ「インライン化」できるようにします。複数の「指示文」がサポートされます。

- `addto`: ターゲット要素にテンプレートの結果を追加します。
- `replace` (デフォルト): テンプレートの結果を使用してターゲット要素を置き換えます。 `<use>` 要素が設定されていない場合、 `<ftl:freemarker>` 設定のデフォルト動作になります。
- `insertbefore`: ターゲット要素の前にテンプレートの結果を追加します。
- `insertafter`: ターゲット要素の後にテンプレートの結果を追加します。

`<ftl:bindTo>` を使用すると、テンプレートの結果を Smooks Bean コンテキストへバインドすることができます。ルーティングに使用したコンポーネントなど、Smooks のその他のコンポーネントを使用して結果にアクセスできます。これは、「巨大」メッセージを小さなメッセージに分割する時に特に便利です。

例6.4 例 - テンプレートの結果を Smooks Bean コンテキストへバインドする

```

<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jms="http://www.milyn.org/xsd/smooks/jms-
routing-1.2.xsd"

    xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <jms:router routeOnElement="order-item" beanId="orderItem_xml"
        destination="queue.orderItems" />

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template>/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Bind the templating result into the bean context, from
where
                                it can be accessed by the JMSRouter (configured
above). -->
            <ftl:bindTo id="orderItem_xml"/>
        </ftl:use>
    </ftl:freemarker>

```

```

</ftl:freemarker>

</smooks-resource-list>

```

`<ftl:outputTo>` を使用して結果を直接 **OutputStreamResource** クラスに書き込みます。これは、巨大なメッセージを小型で「消費可能」なメッセージに分割する便利なメカニズムの1つです。

例6.5 例 - テンプレートの結果を **OutputStreamSource** に書き込む

```

<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.3.xsd"
  xmlns:file="http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd"
  xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

  <!-- Create/open a file output stream. This is written to by the
  freemarker template (below).. -->
  <file:outputStream openOnElement="order-item"
    resourceName="orderItemSplitStream">
    <file:fileNamePattern>order- $\{order.orderId\}$ -
 $\{order.orderItem.itemId\}$ .xml</file:fileNamePattern>

    <file:destinationDirectoryPattern>target/orders</file:destinationDirectoryPattern>
    <file:listFileNamePattern>order-
 $\{order.orderId\}$ .lst</file:listFileNamePattern>

    <file:highWaterMark mark="3"/>
  </file:outputStream>

  <!--
  Every time we hit the end of an <order-item> element, apply this
  freemarker template,
  outputting the result to the "orderItemSplitStream" OutputStream,
  which is the file
  output stream configured above.
  -->
  <ftl:freemarker applyOnElement="order-item">
    <ftl:template>target/classes/orderitem-split.ftl</ftl:template>
    <ftl:use>
      <!-- Output the templating result to the
      "orderItemSplitStream" file output stream... -->
      <ftl:outputTo outputStreamResource="orderItemSplitStream"/>
    </ftl:use>
  </ftl:freemarker>
</smooks-resource-list>

```



注記

包括的なチュートリアルは http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.3_Examples を参照してください。

6.2. NODEMODELS を使用して FREEMARKER 変換を実行する

FreeMarker でメッセージ変換を構築する最も簡単な方法は、**FreeMarker** の *NodeModel* 機能を使用することです (http://freemarker.org/docs/xgui_expose_dom.html を参照)。**FreeMarker** はテンプレートモデルに W3C DOM を使用し、テンプレート内より直接 DOM ノードを参照できるようにします。

Smooks は 2 つの機能を追加します。

- 「断片毎」に実行する機能。DOM モデルの基盤としてメッセージ全体ではなく、ターゲットの断片のみを使用します。
- ストリーミングフィルター処理で **NodeModel** を使用する機能。
- 形式が XML 以外のメッセージで使用する機能。

この機能を Smooks で使用するには、**NodeModels** が「キャプチャ」(SAX ストリーミングの場合は「作成」)されたことを宣言する追加のリソースを定義します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.3.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

  <!--
  Create 2 NodeModels. One high level model for the "order"
  (header etc) and then one per "order-item".

  These models are used in the FreeMarker templating resources
  defined below. You need to make sure you set the selector such
  that the total memory footprint is as low as possible. In this
  example, the "order" model will contain everything except the
  <order-item> data (the main bulk of data in the message). The
  "order-item" model only contains the current <order-item> data
  (i.e. there's max 1 order-item in memory at any one time).
  -->
  <resource-config selector="order,order-item">
    <resource>org.milyn.delivery.DomModelCreator</resource>
  </resource-config>

  <!--
  Apply the first part of the template when we reach the start
  of the <order-items> element. Apply the second part when we
  reach the end.

  Note the <?TEMPLATE-SPLIT-PI?> Processing Instruction in the
  template. This tells Smooks where to split the template,
  resulting in the order-items being inserted at this point.
  -->
  <ftl:freemarker applyOnElement="order-items">
```

```

        <ftl:template><!--<salesorder>
<details>
    <orderid>${order.@id}</orderid>
    <customer>
        <id>${order.header.customer.@number}</id>
        <name>${order.header.customer}</name>
    </customer>
</details>
<itemList>
<?TEMPLATE-SPLIT-PI?>
</itemList>
</salesorder>--></ftl:template>
</ftl:freemarker>

<!--
Output the <order-items> elements. This will appear in the
output message where the <?TEMPLATE-SPLIT-PI?> token appears in the
order-items template.
-->
    <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!-- <item>
<id>${.vars["order-item"].@id}</id>
<productId>${.vars["order-item"].product}</productId>
<quantity>${.vars["order-item"].quantity}</quantity>
<price>${.vars["order-item"].price}</price>
<item>--></ftl:template>
    </ftl:freemarker>

</smooks-resource-list>

```



注記

拡張された例として、<http://www.smooks.org/mediawiki/index.php?title=V1.3:xml-to-xml> のチュートリアルを参照してください。

6.2.1. FreeMarker と Java Bean カートリッジ

FreeMarker NodeMode1 は高性能で簡単に使用することができますが、パフォーマンスに影響します。W3C DOM の構築は「低負荷」ではありません。また、データをオブジェクトセットとして Java メッセージサービスのエンドポイントにルーティングする必要がある場合など、必要なデータが既に抽出され Java オブジェクトモデルへ投入されていることもあります。

NodeMode1 の使用が実用的でない場合、Java Bean カートリッジを使用して適切な Java オブジェクトや仮想モデルへ投入してください。その後、このモデルを **FreeMarker** テンプレート処理で 사용할 수 있습니다。

例6.6 例 (仮想モデルを使用)

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"

xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.3.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

```

```

    <!-- Extract and decode data from the message. Used in the
    freemarker template (below). -->
    <jb:bean beanId="order" class="java.util.Hashtable"
    createOnElement="order">
        <jb:value property="orderId" decoder="Integer"
    data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long"
    data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItem" beanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.Hashtable"
    createOnElement="order-item">
        <jb:value property="itemId" decoder="Integer" data="order-
    item/@id"/>
        <jb:value property="productId" decoder="Long" data="order-
    item/product"/>
        <jb:value property="quantity" decoder="Integer" data="order-
    item/quantity"/>
        <jb:value property="price" decoder="Double" data="order-
    item/price"/>
    </jb:bean>

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!--<orderitem id="{order.orderItem.itemId}"
    order="{order.orderId}">
        <customer>
            <name>${order.customerName}</name>
            <number>${order.customerNumber?c}</number>
        </customer>
        <details>
            <productId>${order.orderItem.productId}</productId>
            <quantity>${order.orderItem.quantity}</quantity>
            <price>${order.orderItem.price}</price>
        </details>
    </orderitem>-->
        </ftl:template>
    </ftl:freemarker>

</smooks-resource-list>

```



注記

拡張された例は http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.3_Examples を参照してください。

6.2.2. プログラムを用いた設定

プログラムを使用して **FreeMarker** のテンプレート設定を Smooks インスタンスに追加することができます。これには、**FreeMarkerTemplateProcessor** インスタンスを追加し、設定します。次の例は、Java バインディングの設定と **FreeMarker** テンプレートの設定を Smooks に追加します。

```
Smooks smooks = new Smooks();

smooks.addVisitor(new Bean(OrderItem.class, "orderItem", "order-
item").bindTo("productId", "order-item/product/@id"));
smooks.addVisitor(new FreeMarkerTemplateProcessor(new
TemplatingConfiguration("/templates/order-tem.ftl"), "order-item");

// And then just use Smooks as normal... filter a Source to a Result etc...
```

6.2.3. XSL テンプレート

重要

JBoss Enterprise SOA Platform 5.0.0 およびそれ以降のバージョンでは、ルート断片に適用される単一の XSLT のみが Smooks 設定に含まれる場合に断片フィルターを迂回します。この場合、XSLT は直接適用されます。これはパフォーマンス上の理由で実行されます。この挙動を無効にするには、**enableFilterBypass** というパラメータを追加し、**false** に設定します。

```
<param name="enableFilterBypass">false</param>
```

XSL テンプレートを Smooks に設定する処理は、**FreeMarkerTemplateProcessor** の設定とほぼ同じです。即座に使用するには、<http://www.milyn.org/xsd/smooks/xsl-1.1.xsd> を統合開発環境に設定します。

例6.7 例

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd">

    <xsl:xsl applyOnElement="#document">
        <xsl:template><!--<xxxxxx/>--></xsl:template>
    </xsl:xsl>

</smooks-resource-list>
```

FreeMarker の場合と同様に、別タイプの外部テンプレートは `<xsl:template>` 要素の URI 参照を使用して設定可能です。

6.2.3.1. XSL サポートに関する注意

以下の場合のみ、Smooks を使用して XSL テンプレートの実行する意味があります。

- メッセージ全体の変換ではなく、断片の変換を実行する必要がある場合。
- Smooks の他の機能を使用して、メッセージ上で分割や永続化などの追加操作を実行する必要がある場合。

- XSL テンプレーティングは DOM フィルタからのみサポートされます。SAX フィルタからはサポートされません。XSL の SAX ベースのアプリケーションと比較すると、パフォーマンスが劣化する場合があります (適用された XSL によります)。
- Smooks はメッセージ断片ごとに XSL テンプレートを適用します。これは、XSL の断片化には大変便利ですが、スタンドアロンコンテキスト向けに書かれたテンプレートが変更なしに Smooks で自動的に動作することを想定しません。そのため、Smooks はドキュメントルートノードへターゲットされた XSL には異なった処理を行い、テンプレートをルート DOM 要素ではなく DOM ドキュメントノードへ適用します。
- ほとんどの XSL には root 要素と一致するテンプレートが含まれています。これは、Smooks は断片毎に XSL を適用しますが、これが有効ではないためです。スタイルシートにコンテキストノードに対して一致するテンプレートが含まれるようにしてください (ターゲットされた断片)。

6.2.3.2. 潜在的な問題: XSLT は外部で動作するが、Smooks 内では動作しない

この問題は、通常次のようなケースの 1 つに当てはまる場合に発生します。

- ドキュメントルートノードへの絶対パス参照を使用するテンプレートがスタイルシートに含まれている場合、この問題が Smooks の断片ベース処理モデルで発生します。これは、正しくない要素が Smooks によってターゲットされるためです。この問題を修正するには、Smooks によってターゲットされるコンテキストノードに一致するテンプレートが XSLT に含まれるようにしてください。
- *SAX と DOM の処理*: 現在、Smooks は XSL の処理に対して DOM ベース処理モデルのみをサポートしています。正確な比較を行うには、Smooks の外部で XSL テンプレートを実行する時に名前空間を認識する *DOMSource* を使用します (SAX または DOM を使用して XSL テンプレートを適用しようとする場合、指定の XSL プロセッサが常に同じ出力を生成するとは限りません)。

第7章 出力データのリッチ化

出力データをリッチ化するためにすぐ使用できる方法は次の3つになります。

JDBC データソースを使用してデータベースにアクセスし、SQL ステートメントを使用してデータベースへ読み書きします。この機能は、Smooks のルーティングカートリッジより提供されます。「SQL を用いたデータベースへのルーティング」の項を参照してください。

エンティティ永続化フレームワーク (Ibatis、Hibernate、JPA 互換フレームワークなど) を使用してデータベースへアクセスし、そのクエリ言語か CRUD メソッドを使用してデータベースへ読み書きします。「エンティティ永続化フレームワーク」の項を参照してください。

カスタムのデータアクセスオブジェクト (DAO) を使用してデータベースにアクセスし、その CRUD メソッドを使用してデータベースへ読み書きします。「DAO サポート」の項を参照してください。

Smooks 1.2 の新しい Smooks 永続カートリッジでは、Smooks 内より複数のエンティティ永続フレームワークを直接使用することができます (Hibernate、JPA など)。

Hibernate の例を見てみましょう。別のJPA 対応フレームワークでもプリンシパルは同じです。

この例で処理されるデータは XML 注文メッセージですが、入力データは CSV、JSON、EDI、Java、これ以外の構造化/階層化されたデータ形式でも可能です。データの形式に関係なく、同じプリンシパルが適用されます。

```
<order>
  <ordernumber>1</ordernumber>
  <customer>123456</customer>
  <order-items>
    <order-item>
      <product>11</product>
      <quantity>2</quantity>
    </order-item>
    <order-item>
      <product>22</product>
      <quantity>7</quantity>
    </order-item>
  </order-items>
</order>
```

Hibernate のエンティティは次の通りです。

```
@Entity
@Table(name="orders")
public class Order {

    @Id
    private Integer ordernumber;

    @Basic
    private String customerId;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private List orderItems = new ArrayList();

    public void addOrderLine(OrderLine orderLine) {
        orderItems.add(orderLine);
    }
}
```

```

    }

    // Getters and Setters....
}

@Entity
@Table(name="orderlines")
public class OrderLine {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    @JoinColumn(name="orderid")
    private Order order;

    @Basic
    private Integer quantity;

    @ManyToOne
    @JoinColumn(name="productid")
    private Product product;

    // Getters and Setters....
}

@Entity
@Table(name = "products")
@NamedQuery(name="product.byId", query="from Product p where p.id = :id")
public class Product {

    @Id
    private Integer id;

    @Basic
    private String name;

    // Getters and Setters....
}

```

この例では、注文の処理と永続化を行います。最初に、注文データを注文エンティティ (Order、OrderLine および Product) へバインドする必要があります。バインドするには以下が必要となります。

- Java Binding フレームワークを使用して Order エンティティと OrderLine エンティティを作成し投入します。
- 各 OrderLine インスタンスを Order インスタンスへワイヤリングします。
- 各 OrderLine インスタンス内で、関連する注文行の Product エンティティをルックアップしワイヤリングする必要があります。
- 最後に、Order インスタンスを挿入 (永続化) する必要があります。

これには、次の Smooks 設定が必要になります。

```

<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd"

xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean BeanId="order" class="example.entity.Order"
createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber" />
        <jb:value property="customerId" data="customer" />
        <jb:wiring setterMethod="addOrderLine" BeanIdRef="orderLine" />
    </jb:bean>

    <jb:bean BeanId="orderLine" class="example.entity.OrderLine"
createOnElement="order-item">
        <jb:value property="quantity" data="quantity" />
        <jb:wiring property="order" BeanIdRef="order" />
        <jb:wiring property="product" BeanIdRef="product" />
    </jb:bean>

    <dao:locator BeanId="product" lookupOnElement="order-item"
onNoResult="EXCEPTION" uniqueResult="true">
        <dao:query>from Product p where p.id = :id</dao:query>
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer" />
        </dao:params>
    </dao:locator>

    <dao:inserter BeanId="order" insertOnElement="order" />

</smooks-resource-list>

```

クエリ文字列の代わりに productById という名前のクエリを使用したい場合は、DAO ロケーターを次のように設定します。

```

<dao:locator BeanId="product" lookupOnElement="order-item"
lookup="product.byId" onNoResult="EXCEPTION" uniqueResult="true">
    <dao:params>
        <dao:value name="id" data="product" decoder="Integer"/>
    </dao:params>
</dao:locator>

```

次のコードは Smooks を実行します。SessionRegister オブジェクトを使用するため、Smooks 内より Hibernate セッションへアクセスできることに注意してください。

```

Smooks smooks = new Smooks("smooks-config.xml");

ExecutionContext executionContext = smooks.createExecutionContext();

// The SessionRegister provides the bridge between Hibernate and the
// Persistence Cartridge. We provide it with the Hibernate session.
// The Hibernate Session is set as default Session.
DaoRegister register = new SessionRegister(session);

// This sets the DAO Register in the executionContext for Smooks

```

```
// to access it.
PersistenceUtil.setDAORegister(executionContext, register);

Transaction transaction = session.beginTransaction();

smooks.filterSource(executionContext, source);

transaction.commit();
```

ここで DAO ベースの例を見てみましょう。この例は、注文情報が含まれている XML ファイルを読み取ります (EDI や CSV などでも同じように動作します)。javabeen カートリッジを使用して、XML データをエンティティ Bean のセットへバインドします。注文商品内の商品 ID を使用して (product 要素)、商品エンティティを見つけ、order エンティティ Bean へバインドします。最終的に注文 Bean が永続化されます。

注文の XML メッセージは次のようになります。

```
<order>
  <ordernumber>1</ordernumber>
  <customer>123456</customer>
  <order-items>
    <order-item>
      <product>11</product>
      <quantity>2</quantity>
    </order-item>
    <order-item>
      <product>22</product>
      <quantity>7</quantity>
    </order-item>
  </order-items>
</order>
```

次のカスタム DAO は Order エンティティを永続化するために使用されます。

```
@Dao
public class OrderDao {

    private final EntityManager em;

    public OrderDao(EntityManager em) {
        this.em = em;
    }

    @Insert
    public void insertOrder(Order order) {
        em.persist(order);
    }
}
```

このクラスを検証すると、**@Dao** アノテーションと **@Insert** アノテーションがあることが分かります。**@Dao** アノテーションは **OrderDao** が DAO オブジェクトであることを宣言します。**@Insert** アノテーションは **insertOrder** メソッドを使用して **Order** エンティティを挿入すべきであることを宣言します。

次のカスタム DAO は、Product エンティティを検索するために使用されます。

```

@Dao
public class ProductDao {

    private final EntityManager em;

    public ProductDao(EntityManager em) {
        this.em = em;
    }

    @Lookup(name = "id")
    public Product findProductById(@Param("id")int id) {
        return em.find(Product.class, id);
    }
}

```

このクラスを見ると、**@Lookup** アノテーションと **@Param** アノテーションの存在に気付くはずで
す。**@Lookup** アノテーションは、**ProductDao#findByProductId** メソッドを使用して **Product** エ
ンティティをルックアップすることを宣言します。**@Lookup** アノテーションの **name** パラメータは、
メソッドに対するルックアップ名の参照を設定します。この名前が宣言されないと、メソッド名が使
用されます。任意の **@Param** アノテーションはパラメータを命名できるようにします。これにより、
Smooks と DAO 間の抽象化が向上されます。**@Param** アノテーションが宣言されないと、パラメータ
はポジションによって解決されます。

Smooks の設定は次のようになります。

```

<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd"

xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean BeanId="order" class="example.entity.Order"
createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber"/>
        <jb:value property="customerId" data="customer"/>
        <jb:wiring setterMethod="addOrderLine" BeanIdRef="orderLine"/>
    </jb:bean>

    <jb:bean BeanId="orderLine" class="example.entity.OrderLine"
createOnElement="order-item">
        <jb:value property="quantity" data="quantity"/>
        <jb:wiring property="order" BeanIdRef="order"/>
        <jb:wiring property="product" BeanIdRef="product"/>
    </jb:bean>

    <dao:locator BeanId="product" dao="product" lookup="id"
lookupOnElement="order-item" onNoResult="EXCEPTION">
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer"/>
        </dao:params>
    </dao:locator>

    <dao:insertter BeanId="order" dao="order" insertOnElement="order"/>

</smooks-resource-list>

```

次のコードが Smooks を実行します。

```
Smooks smooks=new Smooks("./smooks-configs/smooks-dao-config.xml");
ExecutionContext executionContext=smooks.createExecutionContext();

// The register is used to map the DAO's to a DAO name. The DAO name isbe
// used in
// the configuration.
// The MapRegister is a simple Map like implementation of the DaoRegister.
DaoRegister<object>register = MapRegister.builder()
    .put("product",new ProductDao(em))
    .put("order",new OrderDao(em))
    .build();

PersistenceUtil.setDAORegister(executionContext,mapRegister);

// Transaction management from within Smooks isn't supported yet,
// so we need to do it outside the filter execution
EntityTransaction tx=em.getTransaction();
tx.begin();

smooks.filter(new StreamSource(messageIn),null,executionContext);

tx.commit();
```

第8章 GROOVY スクリプト

Groovy スクリプト (<http://groovy.codehaus.org>) のサポートは *設定名前空間* (<http://www.milyn.org/xsd/smooks/groovy-1.1.xsd>) より利用できます。この名前空間は DOM ベーススクリプトと SAX ベーススクリプトのサポートを提供します。

例8.1 設定例

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">

  <g:groovy executeOnElement="xxx">
    <g:script>
      <!--
        //Rename the target fragment element from "xxx" to "yyy"...
        DomUtils.renameElement(element, "yyy", true, true);
      -->
    </g:script>
  </g:groovy>

</smooks-resource-list>
```

Groovy スクリプトを使用する秘訣は次の通りです。

- 適切に名前が付けられた imports 要素を使用してインポートを追加します。自動的にインポートされるクラスは次の通りです。
 - `org.milyn.xml.DomUtils`
 - `org.milyn.javabean.context.BeanContext`。Smooks 1.3 および以降のバージョンでのみ含まれます。
 - `org.milyn.javabean.repository.BeanRepository`
 - `org.w3c.dom.*`
 - `groovy.xml.dom.DOMCategory`
 - `groovy.xml.dom.DOMUtil`
 - `groovy.xml.DOMBuilder`
- `element` と適切に名前付けされた変数よりスクリプトは *visited* 要素を使用できます (要素名にアルファベットと数値のみが使用されている場合、要素名と同じ変数名でも使用できます)。
- *前に実行 (Execute Before)* / *後で実行 (Execute After)*: デフォルトでは、スクリプトは `visitAfter` イベントで実行されます。 `executeBefore` 属性を `true` に設定し、`visitBefore` 上で実行するよう指示します。
- *コメント / CDATA スクリプトラッピング*: スクリプトに特別な XML 文字が含まれている場合、スクリプトを **XML Comment** または **CDATA** セクションでラッピングできます。

8.1. DOM と SAX の混合を GROOVY で使用する

Groovy は DOM と SAX モデルの混合をサポートします。

そのため、**Groovy** の DOM ユーティリティを使用してターゲットのメッセージ断片を処理することができます。SAX フィルターが使用される場合でも DOM 「要素」は **Groovy** スクリプトによって受け取られます。これにより、SAX フィルターを使用する **Groovy** スクリプトが大変簡単になり、ストリームされた形式で巨大メッセージを処理する機能も維持されます。

注意事項:

- デフォルトモードでのみ使用できます (executeBefore が **false** である場合)。executeBefore が **true** に設定されると、この機能は使用できないため、**Groovy** スクリプトは SAXElement のみにアクセスできます。
- DOM 断片を **Smooks.filterSource StreamResult** に書き込むには writeFragment が呼び出されなければなりません。
- DOM 構築機能を使用するとパフォーマンスのオーバーヘッドが発生します。巨大メッセージの処理は行えますが、処理時間が若干長くなることがあります。「有用性」とパフォーマンスがトレードオフになります。

8.1.1. DOM と SAX の混合例

下例のような XML メッセージを使用します。

```
<shopping>
  <category type="groceries">
    <item>Chocolate</item>
    <item>Coffee</item>
  </category>
  <category type="supplies">
    <item>Paper</item>
    <item quantity="4">Pens</item>
  </category>
  <category type="present">
    <item when="Aug 10">Kathryn's Birthday</item>
  </category>
</shopping>
```

ショッピングリストの「supplies」カテゴリに 2 つのペンを追加して変更したいとします。これには、簡単な **Groovy** スクリプトを書いて、メッセージの <category> 要素へ向けます。

カテゴリタイプが「supplies」で項目が「pens」であるカテゴリの <item> 要素上でスクリプトが反復し、数量の値が 2 増えます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"
  xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">

  <core:filterSettings type="SAX" />

  <g:groovy executeOnElement="category">
    <g:script>
```



```
<!--
  use(DOMCategory) {
    // Modify "supplies": we need an extra 2 pens...
    if (category.'@type' == 'supplies') {
      category.item.each { item ->
        if (item.text() == 'Pens') {
          item['@quantity'] = item.'@quantity'.toInteger() +
2;
        }
      }
    }
  }

  // When using the SAX filter, we need to explicitly write the
fragment
  // to the result stream...
  writeFragment(category);
-->
</g:script>
</g:groovy>

</smooks-resource-list>
```

第9章 ルーティング出力データ

Smooks は、メッセージ断片の分割とルーティングに対して複数のオプションをサポートしています。メッセージを断片に分割し、これらの断片を異なる種類のエンドポイント (ファイル、JMS など) へルーティングする機能は大変重要な機能です。Smooks はこの機能を大変興味深い「ひねり」を加えて提供します。

- 基本の断片分割: メッセージにあるすべての注文商品断片を分割してファイルにルーティングするなど、メッセージ上で単にダム分割(dumb split) を実行する必要がある場合があります。「ダム分割」とは、ルーティングを行う前にメッセージ階層の他の部分からデータをマージするなど、分割メッセージの断片でトランスフォーメーションを実行する必要がないことを意味します (ルーティングを行う前に顧客詳細情報を注文商品断片に追加する場合など)。基本的な分割やルーティングには、分割するメッセージ断片の XPath の定義や、未変更の分割メッセージ断片をルーティングするためのルーティングコンポーネント (JBoss ESB や Camel など) の定義が関係します。「基本の分割とルーティング」を参照してください。
- 複雑な断片分割: 基本の断片分割は多くのユースケースに使用でき、ほとんどの分割とルーティングのソリューションで提供されます。Smooks では、ルーティングが適用される前に分割断片データ上でトランスフォーメーションの実行を可能にすることで、基本の分割機能を拡張します。注文商品の分割断片をルーティングする前に顧客詳細の注文情報を各注文商品情報とマージすることがこの例になります。
- インストリーム分割とルーティング (大型メッセージのサポート): Smooks はルーティングを「インストリーム」(メッセージ全体を処理した後、ルーティング向けに一括処理されない) で実行できるため、巨大なメッセージのストリーム (> GB) の処理に対応できます。
- 複数の分割とルーティング: 入力メッセージストリームの単一のフィルタリングパスにある異なるエンドポイントへ複数のメッセージ断片 (XML、EDI、Java などの異なる形式) を条件付きで分割およびルーティングします。\$1,000 を越える注文商品に対しては HighValueOrdersValidation JMS キューへ OrderItem Java オブジェクトインスタンスをルーティングし、ログに記録するためすべての注文商品 (無条件) を XML/JSON として HTTP エンドポイントへルーティングすることがこの例の 1 つとなります。

<http://www.milyn.org/xsd/smooks/camel-1.4.xsd> の設定名前空間より設定をルーティングすると、camel を使用して断片を Apache Camel へルーティングすることが可能です。

例えば、以下を Smooks の設定に指定すると、Camel エンドポイントへのルーティングが可能になります。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:camel="http://www.milyn.org/xsd/smooks/camel-1.4.xsd">

  <!-- Create some bean instances from the input source... -->
  <jb:bean BeanId="orderItem" ... ">
    <!-- etc... See Smooks Java Binding docs -->
  </jb:bean>

  <!-- Route bean to camel endpoints... -->
  <camel:route BeanId="orderItem">
    <camel:to endpoint="direct:slow" if="orderItem.priority == 'Normal'"
  />
    <camel:to endpoint="direct:express" if="orderItem.priority == 'High'"
  />
  </camel:route>

</smooks-resource-list>
```

上記の例では、Smooks BeanContext からの Java Bean を Camel エンドポイントへルーティングします。テンプレートを (FreeMarker など) を同じ Bean に適用して、Bean の代わりにテンプレートの結果をルーティングすることも可能です (XML や CSV などとして)。

上記の設定は、BeanId 属性を使用したルーティングを表しています。routeOnElement という属性を使用してルーティングすることも可能です。

第10章 パフォーマンスの調整

すべてのソフトウェアで言えることですが、適切に設定を行わないと Smooks のパフォーマンスは劣化します。

10.1. 全般

Smooks オブジェクトをキャッシュし再使用する

Smooks の初期化には時間がかかるため、再使用することが重要となります。

可能な場合はリーダーインスタンスをプールする

リーダーによってはリーダーの作成が高負荷となるため、これによりパフォーマンスが大幅に向上されます。

可能な場合は SAX フィルタリングを使用する

SAX 処理は DOM 処理よりもはるかに高速で、メモリフットプリントも一貫して小さくなります。大型のメッセージを処理するために必須となります。Smooks のカートリッジがすべて SAX 対応であることを確認してください (詳細は「[フィルタリング処理の選択](#)」を参照してください)。

デバッグのロギングを無効にする

Smooks はコードの一部で集中的にデバッグのロギングを行います。これにより、処理のオーバーヘッドが大幅に増加し、スループットが低下します。また、ロギングを全く設定しないと、デバッグログステートメントが実行されることがあるので注意してください。

開発環境では HTMLReportGenerator のみを使用する

HTMLReportGenerator を有効にすると、大型のメッセージによってパフォーマンスのオーバーヘッドが大幅に増加します。**OutOfMemory** 例外が発生することもあります。

コンテキストセクタ

コンテキストセクタはパフォーマンスに悪影響を与えます。例えば、"**a/b/c/d/e**" のようなセクタの一致を判断する場合、"**d/e**" のようなセクタよりも多くの処理が必要となります。場合によってはデータモデルに詳細なセクタが必要となることがありますが、必要ない場合はパフォーマンスに対してセクタを最適化するようにしてください。

10.2. SMOOKS カートリッジ

パフォーマンスの最適化に関しては、各カートリッジのドキュメントを参照するようにしてください。

10.3. JAVA BEAN カートリッジ

可能な場合は仮想 Bean モデルを使用しないようにし、マップの代わりに Bean を作成するようにします。データを作成し、マップに追加することは、*Plain Old Java Object* (POJO) を作成し「セッター」メソッドを呼び出すよりも低速になります。

第11章 テスト

11.1. 単体テスト

Smooks で単体テストを実行するのは比較的簡単です。

```
public class MyMessageTransformTest
{
    @Test
    public void test_transform() throws IOException, SAXException
    {
        Smooks smooks = new Smooks(
            getClass().getResourceAsStream("smooks-config.xml") );

        try {
            Source source = new StreamSource(
                getClass().getResourceAsStream("input-message.xml" ) );
            StringResult result = new StringResult();

            smooks.filterSource(source, result);

            // compare the expected xml with the transformation result.
            XMLUnit.setIgnoreWhitespace( true );
            XMLAssert.assertXMLEqual(
                new InputStreamReader(
                    getClass().getResourceAsStream("expected.xml")),
                new StringReader(result.getResult()));
        } finally {
            smooks.close();
        }
    }
}
```

上記のテストケースでは、**XMLUnit** というソフトウェアが使用されています (詳細は <http://xmlunit.sourceforge.net> を参照してください)。

注記

上記のテストでは次のような **Maven** の依存関係が必要になります。

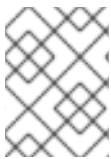
```
<dependency>
  <groupId>xmlunit</groupId>
  <artifactId>xmlunit</artifactId>
  <version>1.1</version>
</dependency>
```

第12章 一般的なユースケース

Smooks v1.0 より導入された主な機能の 1 つが巨大なメッセージ (数ギガバイト) を処理する機能です。Smooks は以下の巨大メッセージの処理をサポートします。

- 1 対 1 のトランスフォーメーション: ソースの形式 (XML など) からターゲットの形式 (EDI、CSV、XML など) へ巨大サイズのメッセージを変換する処理になります。
- 分割とルーティング: 巨大サイズのメッセージを、小さい (消費しやすい) 任意形式 (EDI、XML、Java など) のメッセージに分割し、複数の宛先 (ファイル、JMS、データベースなど) へルーティングします。
- 永続性: 巨大なメッセージのコンポーネントをデータベースへ永続化し、巨大メッセージのコンポーネントのクエリや処理を容易にします。Smooks 内では、分割とルーティング (データベースへのルーティング) の 1 つとして考慮されます。

上記の方法はすべてコードを書かずに (宣言的に対応するなどして) 実現することが可能です。一般的に、上記のいずれかの方法を処理するには、比較的大きく難解/維持不可能なコードを書く必要があります。また、巨大なメッセージが小型のメッセージに分割され (ステージ 1)、小型のメッセージがそれぞれ永続化やルーティングなどのために処理される (ステージ 2) マルチステージ処理として実装されている場合もあります。これは、難解で維持不可能なコードを若干維持しやすく再使用できるようにするために行います。Smooks では、これらのユースケースのほとんどはコードを書かずに対応することが可能です。さらに、ソースメッセージ上の単一のパスで処理し、分割とルーティングを平行して行うことも可能です (異なる形式で異なるタイプの複数の宛先へルーティングも可能)。



注記

パフォーマンス上のヒント: Smooks で巨大なメッセージを処理する場合、SAX フィルターを使用するようにしてください。

別形式の単一メッセージに変換して巨大なメッセージを処理する必要がある場合、複数の FreeMarker テンプレートをソースメッセージのイベントストリームに適用し、**Smooks.filterSource** の結果ストリームへ出力するのが Smooks における最も簡単なメカニズムになります。

適切なモデルのタイプによりますが、FreeMarker のテンプレートを用いてこれを行う方法は 2 つあります。

- モデルに FreeMarker + NodeModels を使用します。
- モデルに対して FreeMarker と Java Object モデルを使用します。Javabean カートリッジを使用して、メッセージのデータよりモデルを構築できます。

ユースケースにおけるトレードオフが妥当である場合、最初のオプションを使用するとよいでしょう。詳細は FreeMarker のテンプレートに関するドキュメントを参照してください。

メッセージに order-item 要素が含まれる場合を想定してみてください。このような場合に Smooks と FreeMarker (NodeModel 使用) を用いて巨大サイズのメッセージを処理するのは複雑ではありません。メッセージが大変大きいため、ランタイム時のメモリフットプリントをできるだけ小さくするよう複数の NodeModel をメッセージ内に指定する必要があります。メモリーに保持するには大きすぎるため、単一のモデルを使用してメッセージを処理することはできません。order メッセージの場合、メインの order データのモデルと order-item データのモデルの 2 つのモデルがあります。

このケースでは、メモリーに存在するほとんどの情報はメインの注文データとなり、注文商品が 1 つだけ存在するようになります。NodeModels はネストされているため、Smooks は注文データの NodeModels に注文商品の NodeModels の情報が含まれないようにします。また、Smooks はメッ

セージをフィルタリングするため、注文商品の NodeModels は注文商品ごとに上書きされます (未収集の場合など)。詳細は「Smooks で DOM モデルと SAX モデルを混合する」の項を参照してください。

FreeMarker テンプレートによって使用される複数の NodeModels をキャプチャするよう Smooks を設定するには、各モデルのルートノードをターゲットにするよう DomModelCreator を設定します。Smooks は SAX フィルタリングもこの機能を使用できるようにします (巨大メッセージを処理する秘訣となります)。次の Smooks 設定は、巨大なメッセージに対して NodeModels を作成します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:core="http://www.milyn.org/xsd/smooks/smooks-
core-1.3.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM for the "complete" message - there are "mini" DOMs
    for the NodeModels below)....
    -->
    <core:filterSettings type="SAX" defaultSerialization="false" />

    <!--
    Create 2 NodeModels. One high level model for the "order"
    (header etc) and then one for the "order-item" elements...
    -->
    <resource-config selector="order,order-item">
        <resource>org.milyn.delivery.DomModelCreator</resource>
    </resource-config>

    <!-- FreeMarker templating configs to be added below... -->
```

ここで FreeMarker のテンプレートを追加する必要があります。合計で 3 つのテンプレートを適用する必要があります。

- 注文商品まで (注文商品は含まない) の注文「ヘッダー」詳細を出力するテンプレート。
- salesorder の item 要素を生成するために必要な各注文商品のテンプレート。
- メッセージを閉じるテンプレート。

Smooks では、2 つの FreeMarker テンプレートを定義してこれを実装します。テンプレートの 1 つは上記の 1 と 3 に対応する (組み合わせる) ようにし、2 つ目のテンプレートは item 要素に対応するようにします。

1 つ目の FreeMarker テンプレートは order-items 要素を目的とし、次のようになるはずです。

```
<ftl:freemarker applyOnElement="order-items">
    <ftl:template><!--<salesorder>
    <details>
        <orderid>${order.@id}</orderid>
        <customer>
            <id>${order.header.customer.@number}</id>
            <name>${order.header.customer}</name>
        </customer>
    </details>
```

```

        <itemList>
        <?TEMPLATE-SPLIT-PI?>
        </itemList>
    </salesorder>-->
        </ftl:template>
    </ftl:freemarker>

```

次の存在に気が付くはずです。

?TEMPLATE-SPLIT-PI?

?TEMPLATE-SPLIT-PI? の処理命令はテンプレートを分割する場所を Smooks に知らせ、order-items の始めにテンプレートの最初の部分を出力し、残りの部分を order-items 要素の最後に出力します。item 要素テンプレート (2 つ目のテンプレート) はこの 2 つの出力の間に出力されます。

2 つ目の FreeMarker テンプレートは大変単純です。ソースメッセージにある各 order-item 要素の最後に item 要素を出力するだけです。

```

<ftl:freemarker applyOnElement="order-item">
    <ftl:template><!-- <item>
    <id>${.vars["order-item"].@id}</id>
    <productId>${.vars["order-item"].product}</productId>
    <quantity>${.vars["order-item"].quantity}</quantity>
    <price>${.vars["order-item"].price}</price>
    </item>-->
    </ftl:template>
</ftl:freemarker>
</smooks-resource-list>

```

2 つ目のテンプレートは order-item 要素の最後にファイアします。

?TEMPLATE-SPLIT-PI?

そのため、最初のテンプレートの上記の処理命令がある場所に出力を効率的に生成します。2 つ目のテンプレートは order NodeModel のデータも参照した可能性があることに注意してください。これは、チュートリアルの実行可能な例として使用可能です。

メモリーにあるオブジェクトは、常に注文ヘッダー詳細と現在の注文商品詳細 (仮想オブジェクトモデルにおける) のみであるため、巨大なメッセージの 1 対 1 のトランスフォーメーションを実行する方法が適用できます。トランスフォーメーションが不明瞭で、ソースメッセージにあるすべてのデータへの完全アクセスが常に必要になるような場合では動作しません (メッセージで注文商品すべてを逆順にしたりソートする必要がある場合など)。このような場合、注文詳細と商品をデータベースヘルペティングした後にデータベースのストレージ、クエリ、および呼び出し機能を使用してトランスフォーメーションを実行することが可能です。

大型/巨大メッセージを処理する方法としては、独立して処理できる小さなメッセージに分割する方法も一般的です。分割とルーティングは、巨大メッセージのみを処理する方法ではなく、小さいメッセージを処理するために必要となることもよくあります (メッセージの大きさは関係ない場合もあります)。たとえば、注文メッセージの注文商品を分割し、異なる部署や関連企業ヘルペティング (内容を基にしたルーティング) して処理する必要がある場合などがあります。このような場合、以下のように宛先に応じて異なるメッセージ形式が必要となる場合があります。

- "destination1" はファイルシステムより XML が必要。
- "destination2" は JMS キューを介して Java オブジェクトが必要。

- "destination3" はデータベースなどのテーブルよりメッセージを取得。
- "destination4" は JMS キューを介して EDI メッセージが必要。

Smooks では上記すべてが可能です。メッセージ上の単一パスで、複数の宛先 (異なるタイプ) へ分割とルーティングの操作を複数実行することができます。

基本概念はシンプルです。Smooks よりメッセージをストリームすると、以下が実行されます。

- ルーティングされる断片のスタンドアロンメッセージ (分割) が繰り返し作成されます。
- 固有の beanId で分割メッセージが Bean コンテキストへ繰り返しバインドされます。
- 必要なエンドポイント (ファイル、DB、JMS、ESB) へ分割メッセージが繰り返しルーティングされます。

注文メッセージの各 orderItem など、ソースメッセージにある分割メッセージの各インスタンスに対してこれらの操作が発生する点を強調するため、上記では「繰り返し」という表現を使用しました。

上記の最初の 2 項目では、Smooks は分割メッセージを作成する方法を提供します。

- 基本的 (非変換/非リッチ化) な断片の分割とバインド。これは大変シンプルな設定で、メッセージ断片を XML 形式へ繰り返しシリアルライズし、文字列として Bean コンテキストに保存します。
- Java バインディングとテンプレートイングカートリッジを使用するより複雑な方法。Smooks を設定してソースメッセージからデータを抽出し、Bean コンテキスト (jb:bean 設定を使用) へ挿入します。また、任意でテンプレートを適用して分割メッセージを作成します。この方法は前述の方法よりも複雑ですが、次の利点があります。
 - 分割断片を変換できます (基本的な方法とは異なり、XML 以外にも対応します)。
 - メッセージをリッチ化できます。
 - 複数のソース断片からのデータを各分割メッセージへ結合できるようにし、複雑な分割が可能です (たとえば、orderItem 断片だけでなく、注文ヘッダー情報も統合できます)。
 - Java オブジェクトを分割メッセージとして分割およびルーティングできます (JMS 上など)。

複雑な方法について上記で説明しましたが、巨大サイズのメッセージを処理する秘訣は (基本的な方法では問題になりませんが)、少量のメモリフットプリントを常に維持することです。これには、常に最も関連性の高いメッセージデータのみをバインドし (Bean コンテキストへ)、Javabean カートリッジを使用するようにします。後続の項の例は、すべて注文メッセージからの注文商品の分割とルーティングが基になっています。Smooks の Javabean カートリッジのバインディング設定は、メインの注文詳細 (注文ヘッダーなど) と「現在の」注文商品詳細のみがデータとしてメモリに保持されるよう実装されるため、巨大メッセージに対するすべての作業が解決法に掲載されています。

前述の通り、メッセージの断片を分割しルーティングする最も簡単な方法は、ルーティングカートリッジより基本的な frag:serialize コンポーネントと *:router コンポーネント (jms:router、file:router など) を使用することです。http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd 名前空間に frag:serialize コンポーネント独自の設定があります。

以下は、SOAP メッセージボディーの内容をシリアルライズし、「soapBody」の BeanId 下にある Bean コンテキストに保存する例になります。

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

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:frag="http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd">

    <frag:serialize fragment="Envelope/Body" bindTo="soapBody"
childContentOnly="true"/>

</smooks-resource-list>
```

これを実行するための Smooks のコードは次の通りです。

```
Smooks smooks = new Smooks(configStream);
JavaResult javaResult = new JavaResult();

smooks.filterSource(new StreamSource(soapMessageStream), javaResult);

String bodyContent = javaResult.getBean("soapBody").toString().trim();
```

プログラムを用いて実行することも可能です (XML 設定は不必要)。

```
Smooks smooks = new Smooks();

smooks.addVisitor(new FragmentSerializer().setBindTo("soapBody"),
"Envelope/Body");

JavaResult javaResult = new JavaResult();
smooks.filterSource(new StreamSource(soapMessageStream), javaResult);

String bodyContent = javaResult.getBean("soapBody").toString().trim();
```

上記のコードスニペットは、分割メッセージを作成し、アクセスできる Bean コンテキストへバインドする方法のみを表しています。処理のためにこれらの分割メッセージを別のエンドポイントヘルレーティングする場合はどうなるでしょうか。この場合でも簡単です。後続の項に概説されているルーティングコンポーネントの 1 つを使用すればよいだけです。

次の例は、処理のために分割メッセージ (ここでは order-item 断片) を JMS の宛先ヘルレーティングする設定を表す簡単な例になります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:frag="http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd"
xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd">

    <!-- Create the split messages for the order items... -->
    <frag:serialize fragment="order-items/order-item" bindTo="orderItem"
/>

    <!-- Route each order items split mesage to the orderItem JMS
processing queue... -->
    <jms:router routeOnElement="order-items/order-item" BeanId="orderItem"
destination="orderItemProcessingQueue" />

</smooks-resource-list>
```

上記の例の JMS ルーティングに関する詳細は、JMS ルーターのドキュメント (下記) を参照してください。jms:router は他のルーターに置き換えることができます。例えば、JBoss ESB を使用している

場合は、esbr:routeBean 設定を使用して ESB エンドポイントへ分割メッセージをルーティングできます。

ファイルベースのルーティングは、<http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd> 設定名前空間から file:outputStream 設定を介して実行されます。

本項では、以下の Smooks の機能を組み合わせ、ファイルシステムでメッセージを小さなメッセージに分割する方法を説明します。

メッセージからデータを抽出し、Bean コンテキスト内の変数に保持するための Javabean カートリッジ。この場合、DOM NodeModel を使用してテンプレティングデータモデルとして使用する注文と注文商品のデータをキャプチャすることもできます。

ファイルシステムストリーム (命名、開閉、スロットルの作成など) を管理するルーティングカートリッジの file:outputStream 設定。

Java Bean カートリッジ (前述の最初の項目参照) によって Bean コンテキストにバインドされるデータより個別の分割メッセージを生成するテンプレティングカートリッジ。テンプレートの結果はファイル出力ストリーム (前述の 2 番目の項目参照) へ書き込まれます。

次の例では、巨大な注文メッセージを処理し、個別の注文商品詳細をファイルへルーティングします。分割メッセージには注文商品断片からのデータが含まれるだけでなく、order-header 要素や root 要素からの情報も含まれます。

これを Smooks で実現するには、次の Smooks 設定をアセンブルします。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:core="http://www.milyn.org/xsd/smooks/smooks-
core-1.3.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd"
                      xmlns:file="http://www.milyn.org/xsd/smooks/file-
routing-1.1.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM, so we can process huge messages...
    -->
    <core:filterSettings type="SAX" />

    <!-- Extract and decode data from the message. Used in the
    freemarker template (below).
        Note that we could also use a NodeModel here... -->
    (1)    <jb:bean BeanId="order" class="java.util.Hashtable"
createOnElement="order">
            <jb:value property="orderId" decoder="Integer"
data="order/@id"/>
            <jb:value property="customerNumber" decoder="Long"
data="header/customer/@number"/>
            <jb:value property="customerName" data="header/customer"/>
            <jb:wiring property="orderItem" BeanIdRef="orderItem"/>
        </jb:bean>
    (2)    <jb:bean BeanId="orderItem" class="java.util.Hashtable"
createOnElement="order-item">
```

```

        <jb:value property="itemId" decoder="Integer" data="order-
item/@id"/>
        <jb:value property="productId" decoder="Long" data="order-
item/product"/>
        <jb:value property="quantity" decoder="Integer" data="order-
item/quantity"/>
        <jb:value property="price" decoder="Double" data="order-
item/price"/>
    </jb:bean>

    <!-- Create/open a file output stream. This is written to by the
freemarker template (below).. -->
(3)    <file:outputStream openOnElement="order-item"
resourceName="orderItemSplitStream">
        <file:fileNamePattern>order-`${order.orderId}-
${order.orderItem.itemId}.xml</file:fileNamePattern>

<file:destinationDirectoryPattern>target/orders</file:destinationDirectory
Pattern>
        <file:listFileNamePattern>order-
${order.orderId}.lst</file:listFileNamePattern>

        <file:highWaterMark mark="10"/>
    </file:outputStream>

    <!--
    Every time we hit the end of an <order-item> element, apply this
freemarker template,
    outputting the result to the "orderItemSplitStream" OutputStream,
which is the file
    output stream configured above.
    -->
(4)    <ftl:freemarker applyOnElement="order-item">
        <ftl:template>target/classes/orderitem-
split.ftl</ftl:template>
        <ftl:use>
            <!-- Output the templating result to the
"orderItemSplitStream" file output stream... -->
            <ftl:outputTo
outputStreamResource="orderItemSplitStream"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>

```

Smooks リソース設定の (1) と (2) は、注文ヘッダー情報 (1) と注文商品情報 (2) を抽出する Java バインディングを定義します。常に現在の注文商品のみがメモリに存在するようにすることが巨大メッセージを処理する秘訣となります。Smooks の Javabeen カートリッジは、これらをすべて管理し、order-item 断片が処理されると orderItem Bean を作成および再作成します。

設定 (3) の file:outputStream 設定は、ファイルシステムでファイルの生成を管理します。設定から見た通り、ファイル名は Bean コンテキストのデータより動的に作成することが可能です。また、highWaterMark 設定パラメータを介してファイルの作成をスロットルすることができます。これにより、ターゲットのファイルシステムが圧倒されないようにファイル作成の管理を行えるようにします。

Smooks リソース設定 (4) は、file:outputStream (3) によって作成された分割メッセージへ書き込みするために使用される FreeMarker テンプレートリソースを定義します。(4) がどのように file:outputStream リソースを参照するか注目してください。FreeMarker のテンプレートは次のようになります。

```
<orderitem id="${.vars["order-item"].@id}" order="${order.@id}">
  <customer>
    <name>${order.header.customer}</name>
    <number>${order.header.customer.@number}</number>
  </customer>
  <details>
    <productId>${.vars["order-item"].product}</productId>
    <quantity>${.vars["order-item"].quantity}</quantity>
    <price>${.vars["order-item"].price}</price>
  </details>
</orderitem>
```

JMS ルーティングは、<http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd> 設定名前空間より `jets:router` 設定を介して実行されます。

`orderItem_xml` bean を `smooks.exampleQueue` というJMS キューへルーティングする (「ファイルへのルーティング」の例も読み取ります) `jets:router` の例は次の通りです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-
core-1.3.xsd"
  xmlns:jets="http://www.milyn.org/xsd/smooks/jets-
routing-1.2.xsd"
  xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

  <!--
  Filter the message using the SAX Filter (i.e. not DOM, so no
  intermediate DOM, so we can process huge messages...
  -->
  <core:filterSettings type="SAX" />

  (1) <resource-config selector="order,order-item">
    <resource>org.milyn.delivery.DomModelCreator</resource>
  </resource-config>

  (2) <jets:router routeOnElement="order-item" BeanId="orderItem_xml"
  destination="smooks.exampleQueue">
    <jets:message>
      <!-- Need to use special FreeMarker variable ".vars" -->
      <jets:correlationIdPattern>${order.@id}-${.vars["order-
item"].@id}</jets:correlationIdPattern>
    </jets:message>
    <jets:highWaterMark mark="3"/>
  </jets:router>

  (3) <ftl:freemarker applyOnElement="order-item">
    <!--
    Note in the template that we need to use the special
    FreeMarker variable ".vars"
    because of the hyphenated variable names ("order-item"). See
```

```

http://freemarker.org/docs/ref_specvar.html.
    -->
    <ftl:template>/orderitem-split.ftl</ftl:template>
    <ftl:use>
        <!-- Bind the templating result into the bean context,
from where
        it can be accessed by the JMSRouter (configured above). -
    ->
        <ftl:bindTo id="orderItem_xml"/>
    </ftl:use>
</ftl:freemarker>

</smooks-resource-list>

```

この例では、FreeMarker 操作の結果が JMS キューヘルレーティングされます (文字列としてなど)。完全オブジェクトモデルをルーティングした可能性もありますが、この場合はシリアルライズされた `ObjectMessage` としてルーティングされます。

データベースへのルーティングは比較的簡単です。「ファイルへのルーティング」の項を読んでからこの項を読むようにしてください。

前述のファイルルーティングの例と同じ状況であることを仮定しますが、ここでは注文および注文商品データをデータベースにルーティングします。

最初に、データストリームより注文および注文商品データを抽出する Java バインディングのセットを定義します。

```

<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jbb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd">

    <!-- Extract the order data... -->
    <jbb:bean BeanId="order" class="java.util.Hashtable"
createOnElement="order">
        <jbb:value property="orderId" decoder="Integer" data="order/@id"/>
        <jbb:value property="customerNumber" decoder="Long"
data="header/customer/@number"/>
        <jbb:value property="customerName" data="header/customer"/>
    </jbb:bean>

    <!-- Extract the order-item data... -->
    <jbb:bean BeanId="orderItem" class="java.util.Hashtable"
createOnElement="order-item">
        <jbb:value property="itemId" decoder="Integer" data="order-
item/@id"/>
        <jbb:value property="productId" decoder="Long" data="order-
item/product"/>
        <jbb:value property="quantity" decoder="Integer" data="order-
item/quantity"/>
        <jbb:value property="price" decoder="Double" data="order-
item/price"/>
    </jbb:bean>

</smooks-resource-list>

```

次に、データソースの設定と、そのデータソースを使用して Java オブジェクトモデルへバインドされたデータをデータベースへ挿入する複数の db:executor 設定を定義します。

直接のデータベース接続を読み出すデータソースの設定 (名前空間 <http://www.milyn.org/xsd/smooks/datasource-1.3.xsd>)

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:ds="http://www.milyn.org/xsd/smooks/datasource-1.3.xsd">

  <ds:direct bindOnElement="#document"
    datasource="DBExtractTransformLoadDS"
    driver="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:hsql://localhost:9201/milyn-hsql-9201"
    username="sa"
    password=""
    autoCommit="false" />

</smooks-resource-list>
```

データベース接続の読み出しに JNDI データソースを使用することも可能です。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:ds="http://www.milyn.org/xsd/smooks/datasource-1.3.xsd">

  <!-- This JNDI datasource can handle JDBC and JTA transactions or
        it can leave the transaction management to an other external
        component.
        An external component could be an other Smooks visitor, the EJB
        transaction manager
        or you can do it your self. -->
  <ds:JNDI
    bindOnElement="#document"
    datasource="DBExtractTransformLoadDS"
    datasourceJndi="java:/someDS"
    transactionManager="JTA"
    transactionJndi="java:/mockTransaction"
    targetProfile="jta"/>

</smooks-resource-list>
```

データソーススキーマはデータソースの設定方法を説明します。

db:executor の設定 (名前空間 <http://www.milyn.org/xsd/smooks/db-routing-1.1.xsd>)

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:db="http://www.milyn.org/xsd/smooks/db-
  routing-1.1.xsd">

  <!-- Assert whether it's an insert or update. Need to do this just
        before we do the insert/update... -->
  <db:executor executeOnElement="order-items"
    datasource="DBExtractTransformLoadDS" executeBefore="true">
```

```
<db:statement>select OrderId from ORDERS where OrderId =
${order.orderId}</db:statement>
  <db:resultSet name="orderExistsRS"/>
</db:executor>

<!-- If it's an insert (orderExistsRS.isEmpty()), insert the order
before we process the order items... -->
  <db:executor executeOnElement="order-items"
datasource="DBExtractTransformLoadDS" executeBefore="true">
    <condition>orderExistsRS.isEmpty()</condition>
    <db:statement>INSERT INTO ORDERS VALUES(${order.orderId},
${order.customerNumber}, ${order.customerName})</db:statement>
  </db:executor>

  <!-- And insert each orderItem... -->
  <db:executor executeOnElement="order-item"
datasource="DBExtractTransformLoadDS" executeBefore="false">
    <condition>orderExistsRS.isEmpty()</condition>
    <db:statement>INSERT INTO ORDERITEMS VALUES (${orderItem.itemId},
${order.orderId}, ${orderItem.productId}, ${orderItem.quantity},
${orderItem.price})</db:statement>
  </db:executor>

  <!-- Ignoring updates for now!! -->

</smooks-resource-list>
```


第13章 SMOOKS の拡張

既存の Smooks の全機能 (Java バインディング、EDI 処理など) は、明確に定義された複数の API の拡張より構築されます。これらの API を今後の項で取り上げます。

Smooks の主な拡張ポイント/API はリーダー API および ビジター API になります。

リーダー API

すべてのメッセージ断片やサブ断片に対する正確に定義された階層的イベント (SAX イベントモデルを基にした) として他の Smooks コンポーネントが消費できるよう、ソース/入力データ (リーダー) を処理するための API です。

ビジター API

ソース/入力リーダーによって作成されるメッセージ断片の SAX イベントを消費するための API です。

これらのコンポーネントがどのように設定されるかは、Smooks の拡張を書く時に大変重要になります。これは、すべての Smooks コンポーネントに共通するため、最初に取り上げます。

13.1. SMOOKS コンポーネントの設定

Smooks のコンポーネントはすべて全く同じように設定されます。Smooks Core では、Smooks コンポーネントはすべて「リソース」で、以前の項で説明した SmooksResourceConfiguration インスタンスを使用して設定されます。

Smooks はコンポーネントに対して名前空間 (XSD) 固有の XML 設定を構築するメカニズムを提供しますが、最も基本的な設定 (および SmooksResourceConfiguration クラスへ直接マッピングする設定) はベース設定の名前空間 (<http://www.milyn.org/xsd/smooks-1.1.xsd>) からの基本的な <resource-config> XML 設定です。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">
  <resource-config selector="">
    <resource></resource>
    <param name=""></param>
  </resource-config>
</smooks-resource-list>
```

- **selector** 属性はリソースが「選択される」メカニズムです (ビジター実装の XPath など)。
- **resource** 要素は実際のリソースです。Java クラス名や、テンプレートなどの他のリソース形式になります。本項では、リソースは Java クラス名であることを前提とします。
- **param** 要素は、resource 要素に定義されたリソースの設定パラメータです。

Smooks は、リソースのランタイム表現の作成 (resource 要素で命名されたクラスの構築など) や設定パラメーターのインジェクションをすべて処理します。また、リソースタイプを判断し、判断したリソースタイプよりセレクタなどを解釈する方法を判断します。たとえば、リソースがビジターインスタンスの場合、セレクターが XPath で、ソースメッセージ断片を選択することを認識します。

13.1.1. 設定アノテーション

コンポーネントが作成された後、**<param>** 要素の詳細を用いて設定する必要があります。この設定には **@ConfigParam** および **@Config** アノテーションを使用します。

13.1.1.1. @ConfigParam

@ConfigParam アノテーションは、リフレクションを用いて、アノテーションが付いたプロパティと同じ名前を持つ **<param>** 要素から名前付きパラメータをインジェクトします。異なる名前でも問題ありませんが、デフォルトの挙動はコンポーネントプロパティの名前に対して照合します。

このアノテーションは以下の理由でコンポーネントの無駄なコードを大幅に削除できます。

- アノテーションの付いたコンポーネントプロパティに設定する前に **<param>** 値のデコードを処理します。Smooks は主なタイプすべて (int、Double、File、Enums など) に **DataDecoder** を提供しますが、同梱されるデコーダーが特定のデコード要件に対応できない場合、カスタムの **DataDecoder** を実装し、使用することが可能です (例: **@ConfigParam(decoder = MyQuirkyDataDecoder.class)**)。カスタムデコーダーが登録されている場合、Smooks は自動的にそのカスタムデコーダーを使用します (アノテーション上にデコーダープロパティを定義する必要はありません)。DetaDecoder 実装の登録に関する詳細は、**DataDecoder** の Javadoc を参照してください。
- **config** プロパティの **choice** 制約をサポートするため、設定された値が定義された choice の値の 1 つでない場合に設定例外が生成されます。たとえば、**ON** と **OFF** を制約される値として持つプロパティがある場合があります。このアノテーションに choice プロパティを使用して、設定を制約し、例外を発生させることが可能です (例: **@ConfigParam(choice = {"ON", "OFF"})**)。
- デフォルトの設定値を指定できます (例: **@ConfigParam(defaultVal = "true")**)。
- プロパティの設定値が必須または任意であるかどうかを指定できます (例: **@ConfigParam(use = Use.OPTIONAL)**)。デフォルトでは、すべてのプロパティは **REQUIRED** ですが、**defaultVal** を暗黙的に設定すると **OPTIONAL** としてプロパティをマーク付けできます。

例13.1 @ConfigParam の使用

次の例は、アノテーション付けされたコンポーネント **DataSeeder** とそれに対応する Smooks 設定を表しています。

```
public class DataSeeder
{
    @ConfigParam
    private File seedDataFile;

    public File getSeedDataFile()
    {
        return seedDataFile;
    }

    // etc...
}
```

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">
  <resource-config selector="dataSeeder">
    <resource>com.acme.DataSeeder</resource>
    <param name="seedDataFile">./seedData.xml</param>
  </resource-config>
</smooks-resource-list>
```

```

        </resource-config>
    </smooks-resource-list>

```

13.1.1.2. @Config

@Config アノテーションは、コンポーネントに関連する完全な **SmooksResourceConfiguration** インスタンスをリフレクションを用いてアノテーション付けされたコンポーネントプロパティにインジェクトします。タイプが **SmooksResourceConfiguration** でないコンポーネントプロパティにアノテーションが追加されると、結果的にエラーが発生します。

例13.2 @Config の使用

```

public class MySmooksComponent
{
    @Config
    private SmooksResourceConfiguration config;

    // etc...
}

```

13.1.1.3. @Initialize および @Uninitialize

@ConfigParam アノテーションは、単純な値でコンポーネントを設定する場合に便利ですが、初期化コードを書く必要があるさらに複雑な設定がコンポーネントに必要なこともあります。このような場合、Smooks は **@Initialize** アノテーションを提供します。

また、初期化中に取得されたリソースの一部を開放する場合など、関連する Smooks インスタンスが破棄された時に (ガベッジコレクション) 初期化中に実行される作業を取り消す必要がある時があります。このような場合、Smooks は **@Uninitialize** アノテーションを提供します。

基本的な初期化/非初期化シーケンスは次のように記述できます。

```

smooks = new Smooks(..);

// Initialize all annotated components
@Initialize

// Use the smooks instance through a series of filterSource invocations...
smooks.filterSource(...);
smooks.filterSource(...);
smooks.filterSource(...);
... etc ...

smooks.close();

// Uninitialize all annotated components
@Uninitialize

```

例13.3 @Initialize と @Uninitialize の使用

この例では、初期化時にデータベースへ複数の接続を開くコンポーネントがあり、Smooks インスタンスを閉じる時にこれらのデータベースリソースをすべて開放する必要があることを仮定します。

```
public class MultiDataSourceAccessor
{
    @ConfigParam
    private File dataSourceConfig;

    Map<String, Datasource> datasources = new HashMap<String,
    Datasource>();

    @Initialize
    public void createDataSources()
    {
        // Add DS creation code here....
        // Read the dataSourceConfig property to read the DS configs...
    }

    @Uninitialize
    public void releaseDataSources()
    {
        // Add DS release code here....
    }

    // etc...
}
```

@Initialize および **@Uninitialize** アノテーションを使用する時は次の点に注意してください。

- **@Initialize** および **@Uninitialize** メソッドは、引数がゼロのパブリックメソッドでなければなりません。
- **@ConfigParam** プロパティはすべて最初の **@Initialize** メソッドが呼び出される前に初期化されます。そのため、**@ConfigParam** コンポーネントプロパティを初期化処理への入力として使用できます。
- **@Uninitialize** メソッドはすべて **Smooks.close** メソッドへの呼び出しへ応答するために呼び出されます。

13.1.1.4. カスタム設定名前空間の定義

Smooks は、コンポーネントのカスタム設定名前空間を定義するメカニズムをサポートします。これにより、**<resource-config>** ベースの設定を使用して、汎用の Smooks リソースとして扱う代わりに、検証可能なコンポーネントの XSD ベースのカスタム設定をサポートすることが可能になります。

基本処理には 2 つの手順が関係します。

1. ベースの <http://www.milyn.org/xsd/smooks-1.1.xsd> 設定名前空間を拡張するコンポーネントの設定 XSD を書きます。この XSD はコンポーネントのクラスパス上に置く必要があります。**/META-INF/** フォルダにあり、名前空間 URI と同じパスを持つ必要があります。たとえば、拡張された名前空間 URI が <http://www.acme.com/schemas/smooks/acme-core-1.0.xsd> である場合、物理的な XSD ファイルは **/META-INF/schemas/smooks/acme-core-1.0.xsd** のクラスパスに置く必要があります。

2. カスタムの名前空間設定を **SmooksResourceConfiguration** インスタンスへマッピングする Smooks 設定の名前空間マッピング設定ファイルを書きます。このファイルはマッピングする名前空間の名前を基に命名する必要があります (慣例により)、XSD と同じフォルダのクラスパス上に物理的に置く必要があります。上記の例の場合、Smooks マッピングファイルは **/META-INF/schemas/smooks/acme-core-1.0.xsd-smooks.xml** になります。-**smooks.xml** が末尾にあることに注目してください。

このメカニズムを理解するには、Smooks のコード内にある既存の拡張された名前空間設定を確認するのが最も簡単です。Smooks コンポーネントはすべて (Java バインディング機能を含む) このメカニズムを使用して設定を定義します。Smooks Core は拡張された設定名前空間を複数定義します。

13.2. ソースリーダーの実装

新しいソースリーダーを Smooks に実装し、設定するのは簡単です。プロセスの Smooks 固有の部分は簡単で、問題にはなりません。リーダーを実装するためのソースデータ形式の複雑な関数に対して努力を必要とします。

カスタムデータ形式のリーダーを実装すると、即座にすべての Smooks 機能がそのデータ形式 (Java バインディング、テンプレート、永続化、検証、分割とルーティングなど) へ開放されます。そのため、比較的小さな投資でも大きな利益を生むことができます。リーダーが標準の **org.xml.sax.XMLReader** インターフェースを Java JDK より実装することのみが Smooks の要件になります。しかし、リーダー実装を設定可能にしたい場合は、**org.milyn.xml.SmooksXMLReader** インターフェースを実装する必要があります。**org.milyn.xml.SmooksXMLReader** は **org.xml.sax.XMLReader** の拡張です。既存の **org.xml.sax.XMLReader** 実装を使用したり、新しいリーダーを実装するのは簡単です。

詳細は <http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/XMLReader.html> を参照してください。

Smooks を使用するリーダーを実装する簡単な例を見てみましょう。この例では、コンマ区切り値 (CSV) レコードのストリームを読み取れるリーダーを実装し、CSV ストリームを Smooks によって処理可能な SAX イベントのストリームへ変換します。Smooks が許可する作業をすべて行うことができます。

最初に基本のリーダークラスを実装します。

```
public class MyCSVReader implements SmooksXMLReader
{
    // Implement all of the XMLReader methods...
}
```

org.xml.sax.XMLReader インターフェースからの 2 つのメソッドに注目してください。

1. **setContentHandler(ContentHandler)** は Smooks Core によって呼び出されます。これは、リーダーの **org.xml.sax.ContentHandler** インスタンスを設定します。**org.xml.sax.ContentHandler** インスタンスのメソッドは **parse(InputSource)** メソッド内より呼び出されます。
2. **parse(InputSource)** : ソースデータ入力ストリームを受け取り、解析を行い (この例では CSV ストリーム)、**setContentHandler(ContentHandler)** メソッドに提供される **org.xml.sax.ContentHandler** インスタンスへの呼び出しより SAX イベントストリームを生成するメソッドです。

詳細は <http://download.oracle.com/javase/6/docs/api/org/xml/sax/ContentHandler.html> を参照してください。

CSV レコードに関連するフィールドの名前を用いて CSV リーダーを設定する必要があります。カスタムリーダー実装の設定は他の Smooks コンポーネントと同じです。

上記のメソッドとフィールドの設定をもう少し詳しく見てみましょう。

```
public class MyCSVReader implements SmooksXMLReader
{
    private ContentHandler contentHandler;

    @ConfigParam
    private String[] fields; // Auto decoded and injected from the
    "fields" <param> on the reader config.

    public void setContentHandler(ContentHandler contentHandler) {
        this.contentHandler = contentHandler;
    }

    public void parse(InputSource csvInputSource) throws IOException,
    SAXException {
        // TODO: Implement parsing of CSV Stream...
    }

    // Other XMLReader methods...
}
```

基本的なリーダー実装スタブがあります。新しいリーダー実装をテストするため、単体テストを書くことができます。

最初に、CSV 入力の例が必要となります。**names.csv** という名前のファイルにある簡単な名前のリストを使用しましょう。

```
Tom, Jones
Mike, Jones
Mark, Jones
```

次に、MyCSVReader を用いて Smooks を設定するテスト Smooks 設定が必要となります。前述の通り、Smooks にあるものすべてはリソースで、基本的な **<resource-config>** 設定を用いて設定することができます。このように設定することもできますが、若干無駄が多いため、Smooks はリーダーの設定に特化した基本的な **<reader>** 設定要素を提供します。テストの設定は **mycsvread-config.xml** にあり、次のようになります。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">
  <reader class="com.acme.MyCSVReader">
    <params>
      <param name="fields">firstname,lastname</param>
    </params>
  </reader>
</smooks-resource-list>
```

そして、JUnit テストクラスが必要になります。

```
public class MyCSVReaderTest extends TestCase
{
```



```

    public void test() {
        Smooks smooks = new
Smooks(getClass().getResourceAsStream("mycsvread-config.xml"));
        StringResult serializedCSVEvents = new StringResult();

        smooks.filterSource(new
StreamSource(getClass().getResourceAsStream("names.csv")),
serializedCSVEvents);

        System.out.println(serializedCSVEvents);

        // TODO: add assertions etc
    }
}

```

これで、カスタムリーダー実装を持つ基本設定が設定され、開発を促進する単体テストが作成されました。リーダーの **parse** メソッドはまだ何も実行せず、テストクラスはアサーションなどを作成していません。次に、**parse** メソッドを実装しましょう。

```

public class MyCSVReader implements SmooksXMLReader
{
    private ContentHandler contentHandler;

    @ConfigParam
    private String[] fields; // Auto decoded and injected from the
"fields" <param> on the reader config.

    public void setContentHandler(ContentHandler contentHandler)
    {
        this.contentHandler = contentHandler;
    }

    public void parse(InputSource csvInputSource) throws IOException,
SAXException
    {
        BufferedReader csvRecordReader = new
BufferedReader(csvInputSource.getCharacterStream());
        String csvRecord;

        // Send the start of message events to the handler...
        contentHandler.startDocument();
        contentHandler.startElement(XMLConstants.NULL_NS_URI, "message-
root", "", new AttributesImpl());

        csvRecord = csvRecordReader.readLine();
        while(csvRecord != null)
        {
            String[] fieldValues = csvRecord.split(",");

            // perform checks...

            // Send the events for this record...
            contentHandler.startElement(XMLConstants.NULL_NS_URI,
"record", "", new AttributesImpl());
            for(int i = 0; i < fieldValues.length; i++)
            {

```

```

        contentHandler.startElement(XMLConstants.NULL_NS_URI,
fields[i], "", new AttributesImpl());
        contentHandler.characters(fieldValues[i].toCharArray(),
0, fieldValues[i].length());
        contentHandler.endElement(XMLConstants.NULL_NS_URI,
fields[i], "");
    }
    contentHandler.endElement(XMLConstants.NULL_NS_URI, "record",
    "");

    csvRecord = csvRecordReader.readLine();
}

// Send the end of message events to the handler...
contentHandler.endElement(XMLConstants.NULL_NS_URI, "message-
root", "");
contentHandler.endDocument();
}

// Other XMLReader methods...
}

```

この時点で単体テストを実行すると、次の出力がコンソールに表示されます (フォーマット済み)。

```

<message-root>
  <record>
    <firstname>Tom</firstname>
    <lastname>Jones</lastname>
  </record>
  <record>
    <firstname>Mike</firstname>
    <lastname>Jones</lastname>
  </record>
  <record>
    <firstname>Mark</firstname>
    <lastname>Jones</lastname>
  </record>
</message-root>

```

この後に、テストを拡大したり、リーダー実装コードを強化したりします。

これで、リーダーを使用して Smooks によってサポートされる操作をすべて実行できるようになりました。たとえば、名前を **PersonName** オブジェクトの **List** へバインドするために次の設定 (**java-binding-config.xml**) を使用することができます。

```

<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">
  <reader class="com.acme.MyCSVReader">
    <params>
      <param name="fields">firstname,lastname</param>
    </params>
  </reader>
  <jb:bean BeanId="peopleNames" class="java.util.ArrayList"
createOnElement="message-root">
    <jb:wiring BeanIdRef="personName" />
  </jb:bean>
</smooks-resource-list>

```



```

    </jb:bean>
    <jb:bean BeanId="personName" class="com.acme.PersonName"
createOnElement="message-root/record">
        <jb:value property="first" data="record/firstname" />
        <jb:value property="last" data="record/lastname" />
    </jb:bean>
</smooks-resource-list>

```

この設定のテストは次のようになるでしょう。

```

public class MyCSVReaderTest extends TestCase
{
    public void test_java_binding()
    {
        Smooks smooks = new Smooks(getClass().getResourceAsStream("java-
binding-config.xml"));
        JavaResult javaResult = new JavaResult();

        smooks.filterSource(new
StreamSource(getClass().getResourceAsStream("names.csv")), javaResult);

        List<PersonName> peopleNames = (List<PersonName>)
javaResult.getBean("peopleNames");

        // TODO: add assertions etc
    }
}

```

- リーダーインスタンスが平行使用されることはありません。Smooks Core は各メッセージに対して新しいインスタンスを作成するか、**readerPoolSize FilterSettings** プロパティに従ってインスタンスをプールし、再使用します。
- 現在のフィルタリングコンテキストに対する Smooks **ExecutionContext** へリーダーがアクセスする必要がある場合、リーダーは **org.milyn.xml.SmooksXMLReader** インターフェースを実装する必要があります。
- ソースデータがバイナリデータストリームである場合、リーダーは **org.milyn.delivery.StreamReader** インターフェースを実装する必要があります。
- **Smookscode>** インスタンス上で設定する **GenericReaderConfigurator** インスタンスを使用して、リーダーをソースコード内 (単体テストなど) で設定することができます。
- 基本の **<reader>** 設定を使用できますが、カスタムの CSV リーダー実装にカスタムの設定名前空間 (XSD) を定義することが可能です。この定義についてはここでは取り上げません。**EDIReader**、**CSVReader**、**JSONReader** など、Smooks に提供されるリーダー実装に対する拡張された設定名前空間を見るため、ソースコードを確認します。これより、カスタムリーダーに対してカスタムの設定名前空間を定義する方法が分かるはずです。

13.2.1. バイナリソースリーダーの実装

バイナリデータソースのソースリーダーを実装することも可能です。この場合、リーダーは **org.milyn.delivery.StreamReader** インターフェースを実装する必要があります。これは、**InputStream** が確実に提供されるよう Smooks ランタイムに伝えるマーカーインターフェースです。

バイナリリーダー実装は非バイナリリーダー実装 (上記を参照) と本質的に同じですが、**parse** メソッドの実装は **InputSource** から **InputStream** を使用し (例: **InputSource.getCharacterStream()** ではなく **InputSource.getByteStream()** を呼び出す)、デコードされたバイナリデータから XML イベントを生成することが異なります。

例13.4 バイナリソースリーダーの実装

簡単な **parse** メソッド実装は次のようになります。

```
public static class BinaryFormatXXReader implements SmooksXMLReader,
StreamReader
{
    @ConfigParam
    private String xProtocolVersion;

    @ConfigParam
    private int someOtherXProtocolConfig;

    // etc...

    public void parse(InputSource inputSource) throws IOException,
SAXException {
        // Use the InputStream (binary) on the InputSource...
        InputStream binStream = inputSource.getByteStream();

        // Create and configure the data decoder...
        BinaryFormatXDecoder xDecoder = new BinaryFormatXDecoder();
        xDecoder.setProtocolVersion(xProtocolVersion);
        xDecoder.setSomeOtherXProtocolConfig(someOtherXProtocolConfig);
        xDecoder.setXSource(binStream);

        // Generate the XML Events on the contentHandler...
        contentHandler.startDocument();

        // Use xDecoder to fire startElement, endElement etc events on
the contentHandler (see previous section)...

        contentHandler.endDocument();
    }

    // etc....
}
```

BinaryFormatXXReader リーダーを Smooks に設定する方法は、他のリーダーの場合と同じです (前項の説明通り)。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <reader class="com.acme.BinaryFormatXXReader">
        <params>
            <param name="xProtocolVersion">2.5.7</param>
            <param name="someOtherXProtocolConfig">1</param>
            ... etc...
        </params>
    </reader>
```

```
... Other Smooks configurations e.g. <jb:bean> configs for binding
the binary data into Java objects...
```

```
</smooks-resource-list>
```

Smooks 実行コードは次のようになります (**StreamSource** に提供される **InputStream** に注意してください)。この場合、XML と Java オブジェクトの 2 つの結果を生成します。

```
StreamResult xmlResult = new StreamResult(xmlOutWriter);
JavaResult javaResult = new JavaResult();

InputStream xBinaryInputStream = getXByteStream();

smooks.filterSource(new StreamSource(xBinaryInputStream), xmlResult,
javaResult);

// etc... Use the beans in the javaResult...
```

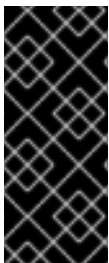
13.3. 断片ビジターの実装

ビジター実装は Smooks の主戦力となる実装です。Smooks のそのまま使用可能な機能 (Java バインディング、テンプレート、永続化など) のほとんどは、ビジター実装を 1 つ以上作成すると作成されます。ビジター実装は **ExecutionContext** および **ApplicationContext** コンテキストオブジェクトを通じて連携することが多く、連携して共通の目的を達成します。

Smooks は 2 つのタイプのビジター実装をサポートします。

1. **org.milyn.delivery.sax.SAXVisitor** サブインターフェースを基にした SAX ベースの実装。
2. **org.milyn.delivery.sax.SAXVisitor** サブインターフェースを基にした DOM ベースの実装。

実装は SAX と DOM の両方をサポートできますが、SAX 専用ビジターを実装することが推奨されます。SAX ベースの実装は通常、作成が容易で、より高速です。そのため、ここでは SAX のみを集中的に説明します。



重要

ビジター実装はすべてステートレスオブジェクトとして扱われます。単一のビジターインスタンスは、複数のメッセージ全体で同時に使用可能でなければなりません (例: **Smooks.filterSource** メソッドへの複数の同時呼び出し全体)。現在の **Smooks.filterSource** の実行に関連するステートはすべて **ExecutionContext** に格納される必要があります。

13.3.1. SAX ビジター API

SAX ビジター API は複数のインターフェースで構成されます。これらのインターフェースは、**SAXVisitor** 実装がキャプチャし処理できる **org.xml.sax.ContentHandler** SAX イベントが基になっています。**SAXVisitor** によって解決されたユースケースによっては、これらのインターフェースの 1 つまたはすべてを実装する必要がある場合があります。

org.milyn.delivery.sax.SAXVisitBefore

ターゲットの fragment 要素の **startElement** SAX イベントをキャプチャします。

```
public interface SAXVisitBefore extends SAXVisitor
{
    void visitBefore(SAXElement element, ExecutionContext
executionContext)
                                throws SmooksException,
    IOException;
}
```

org.milyn.delivery.sax.SAXVisitChildren

ターゲットの fragment 要素に対する文字ベースの SAX イベントをキャプチャし、子 fragment 要素の **startElement** イベントに対応する Smooks が生成した (擬似) イベントもキャプチャします。

```
public interface SAXVisitChildren extends SAXVisitor
{
    void onChildText(SAXElement element, SAXText childText,
ExecutionContext
                                executionContext) throws SmooksException,
    IOException;

    void onChildElement(SAXElement element, SAXElement childElement,
ExecutionContext executionContext) throws SmooksException,
    IOException;
}
```

org.milyn.delivery.sax.SAXVisitAfter

ターゲット fragment 要素の **endElement** SAX イベントをキャプチャします。

```
public interface SAXVisitAfter extends SAXVisitor
{
    void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                throws SmooksException,
    IOException;
}
```

SAX イベントをすべてキャプチャする必要があるこれらの実装が便利のように、上記の 3 つのインターフェースは **org.milyn.delivery.sax.SAXElementVisitor** インターフェースの単一のインターフェースと一緒にプルされます。

XML の一部を用いてこれらのイベントを表します。

```
<message>
  <target-fragment>      <!-- SAXVisitBefore.visitBefore -->
    Text!!               <!-- SAXVisitChildren.onChildText -->
    <child>              <!-- SAXVisitChildren.onChildElement -->
    </child>
  </target-fragment>    <!-- SAXVisitAfter.visitAfter -->
</message>
```

上記は XML のソースメッセージイベントストリームを表しています。形式は EDI、CSV、JSON やその他の形式でも可能です。理解しやすくするため、XML としてシリアル化されたソースメッセージイベントストリームとして考慮してください。

上記の SAX インターフェースの通り、**org.milyn.delivery.sax.SAXElement** タイプはすべてのメソッド呼び出しへ渡されます。このオブジェクトには、属性やそれらの値が含まれるターゲット断片要素に関する詳細が含まれています。また、テキストの蓄積を管理するためのメソッド **Smooks.filterSource(Source, Result)** メソッド呼び出しへ渡された **StreamResult** インスタンスに関連する **Writer** にアクセスするためのメソッドも含まれます。テキストの蓄積や **StreamResult** 書き込みについては、今後の項で詳細に説明します。

13.3.2. テキストの蓄積

SAX はストリームベースの処理モデルです。ドキュメントオブジェクトモデル (DOM) を作成したり、イベントデータを「蓄積」することはありません。SAX が巨大なメッセージストリームの処理に適切な処理モデルであるのはそのためです。

org.milyn.delivery.sax.SAXElement には常にターゲット要素に関連する属性データが含まれますが、**SAXVisitBefore.visitBefore** イベントと **SAXVisitAfter.visitAfter** イベントの間発生する SAX イベント (**SAXVisitChildren.onChildText**) を持つ断片の子テキストデータは含まれません (上図を参照)。結果的にパフォーマンスが大幅に劣化することがあるため、テキストイベントは **SAXElement** 上では蓄積されません。**SAXVisitor** 実装がある断片のテキストの内容にアクセスする必要がある場合、ターゲットの断片に対してテキストを蓄積するよう、Smooks に明示的に伝える必要があります。これを実行するには、**SAXVisitor** の **SAXVisitBefore.visitBefore** メソッド実装内部より **SAXElement.accumulateText** メソッドを呼び出します。

```
public class MyVisitor implements SAXVisitBefore, SAXVisitAfter
{
    public void visitBefore(SAXElement element, ExecutionContext
executionContext)
                                                                    throws SmooksException,
IOException
    {
        element.accumulateText();
    }

    public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                                    throws SmooksException,
IOException
    {
        String fragmentText = element.getTextContent();

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

SAXVisitBefore.visitBefore を実装する代わりに、**@TextConsumer** アノテーションを **SAXVisitor** 実装に付けて対応することが可能です。

```
@TextConsumer
public class MyVisitor implements SAXVisitAfter
{
    public void visitAfter(SAXElement element, ExecutionContext
executionContext)
```

`throws SmooksException,`

```
IOException
{
    String fragmentText = element.getTextContent();

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

SAXVisitAfter.visitAfter イベントが発生するまで、断片テキストはすべて使用できないことに注意してください。

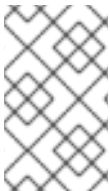
13.3.3. StreamResult 書き込み/シリアルライゼーション

Smooks.filterSource(Source, Result) メソッドに 1 つまたは複数の **Result** タイプ実装を用いることができますが、その 1 つが **javax.xml.transform.stream.StreamResult** クラスです。Smooks は **StreamResult** インスタンスを通じてソースをストリーミングします。

デフォルトでは、常に **Smooks.filterSource(Source, Result)** メソッドへ提供される **StreamResult** インスタンスに対して、完全なソースイベントストリームが XML としてシリアルライズされます。**Smooks.filterSource(Source, Result)** メソッドに提供されるソースが XML ストリームで、**StreamResult** インスタンスが **Result** インスタンスの 1 つとして提供される場合、1 つ以上の断片を変更する **SAXVisitor** 実装を 1 つ以上用いて Smooks インスタンスが設定されている場合を除き、ソース XML は変更されずに **StreamResult** へ書き出されます。

デフォルトのシリアルライゼーションの挙動は、フィルター設定にて有効または無効にすることができます。

あるメッセージ断片のシリアルライズされた形式を変更したい場合、**SAXVisitor** を実装してトランスフォーメーションを実行し、XPath のような式を使用してメッセージ断片へターゲットする必要があります。



注記

提供されるテンプレートコンポーネントの 1 つを使用してメッセージ断片のシリアルライズされた形式を変更することも可能です。これらのコンポーネントは **SAXVisitor** 実装でもあります。

断片のシリアルライズされた形式を変換するよう **SAXVisitor** を実装するには、その **SAXVisitor** 実装が **StreamResult** へ書き込むよう Smooks に伝えることが重要になります。これは、Smooks では単一の断片へ複数の **SAXVisitor** 実装をターゲットできますが、断片ごとに 1 つの **SAXVisitor** のみが **StreamResult** へ書き込みできるためです。2 つ目の **SAXVisitor** が **StreamResult** へ書き込みしようとする、**SAXWriterAccessException** が発生するため、Smooks の設定を変更する必要があります。

SAXVisitor の 1 つが **StreamResult** へ書き込みできるようにするには、**StreamResult** への **Writer** の「所有権」をその **SAXVisitor** が「取得する」必要があります。これには、**SAXVisitBefore.visitBefore** メソッド内部より **SAXElement.getWriter(SAXVisitor)** メソッドへ呼び出しを行い、**this** を **SAXVisitor** パラメータとして渡します。

```
public class MyVisitor implements SAXElementVisitor
{
    public void visitBefore(SAXElement element, ExecutionContext
```



```

executionContext)
                                throws SmooksException,
IOException
{
    Writer writer = element.getWriter(this);

    // ... write the start of the fragment...
}

public void onChildText(SAXElement element, SAXText childText,
                        ExecutionContext executionContext)
                        throws SmooksException,
IOException
{
    Writer writer = element.getWriter(this);

    // ... write the child text...
}

public void onChildElement(SAXElement element, SAXElement childElement,
                           ExecutionContext executionContext)
                           throws SmooksException,
IOException
{
}

public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                throws SmooksException,
IOException
{
    Writer writer = element.getWriter(this);
    // ... close the fragment...
}
}

```

サブ断片のシリアル化を制御する必要がある場合、**Writer** インスタンスをリセットし、サブ断片のシリアル化を迂回しなければなりません。これには、**SAXElement.getWriter** を呼び出します。

シリアル化/変換するターゲット断片にサブ断片が発生しないことが分かっている場合があります。このような場合、**SAXElement.getWriter** メソッドを呼び出して **Writer** の所有権を獲得するため、**SAXVisitBefore.visitBefore** を実装するのでは不十分です。**@StreamResultWriter** アノテーションがあるのはこのためです。このアノテーションを **@TextConsumer** アノテーションと組み合わせて使用すると、**SAXVisitAfter.visitAfter** メソッドの実装のみが必要となります。

```

@TextConsumer
@StreamResultWriter
public class MyVisitor implements SAXVisitAfter
{
    public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                throws SmooksException,
IOException
{

```

```

        Writer writer = element.getWriter(this);

        // ... serialize to the writer ...
    }
}

```

13.3.3.1. SAXToXMLWriter

SAXElement データを XML としてシリアル化することを少しでも容易にするため、Smooks は **SAXToXMLWriter** クラスを提供します。このクラスにより、**SAXVisitor** 実装を書くことが可能になります。

```

@StreamResultWriter
public class MyVisitor implements SAXElementVisitor
{
    private SAXToXMLWriter xmlWriter = new SAXToXMLWriter(this, true);

    public void visitBefore(SAXElement element, ExecutionContext
executionContext)
                                                                    throws SmooksException,
IOException
    {
        xmlWriter.writeStartElement(element);
    }

    public void onChildText(SAXElement element, SAXText childText,
ExecutionContext
                                                                    executionContext) throws SmooksException,
IOException
    {
        xmlWriter.writeText(childText, element);
    }

    public void onChildElement(SAXElement element, SAXElement childElement,
ExecutionContext executionContext) throws SmooksException,
IOException
    {
    }

    public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                                    throws SmooksException,
IOException
    {
        xmlWriter.writeEndElement(element);
    }
}

```

SAXToXMLWriter コンストラクタの 2 つ目の引数がブール値であることに気付かれたかもしれませんが。これは **encodeSpecialChars** 引数であり、**rewriteEntities** フィルター設定を基に設定する必要があります。@**StreamResultWriter** アノテーションをクラスから **SAXToXMLWriter** インスタンス宣言へ移動する場合、Smooks は **SAXToXMLWriter** インスタンスを作成し、関連する Smooks インスタンスの **rewriteEntities** フィルター設定を用いて初期化します。

■


```

@TextConsumer
public class MyVisitor implements SAXVisitAfter
{
    @StreamResultWriter
    private SAXToXMLWriter xmlWriter;

    public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                                    throws SmooksException,
IOException
    {
        xmlWriter.writeStartElement(element);
        xmlWriter.writeText(element);
        xmlWriter.writeEndElement(element);
    }
}

```

13.3.4. ビジターの設定

SAXVisitor 設定は、他の Smooks コンポーネントと同じように挙動します。

Smooks ビジターインスタンスの設定に関し、設定 **selector** が XPath 式と同様に解釈されることが最も重要な点になります。

また、ビジターインスタンスは Smooks インスタンス上のプログラムコード内で設定することが可能です。これは単体テストに使用すると大変便利です。

13.3.4.1. ビジター設定の例

この例では、次のような大変単純な **SAXVisitor** 実装を使用します。

```

@TextConsumer
public class ChangeItemState implements SAXVisitAfter
{
    @StreamResultWriter
    private SAXToXMLWriter xmlWriter;

    @ConfigParam
    private String newState;

    public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                                    throws SmooksException,
IOException
    {
        element.setAttribute("state", newState);

        xmlWriter.writeStartElement(element);
        xmlWriter.writeText(element);
        xmlWriter.writeEndElement(element);
    }
}

```

状態が **OK** の `<order-item>` 断片上でファイアするよう **ChangeItemState** を宣言的に設定するのは、次の通り簡単です。

```
<smooks-resource-list
  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

  <resource-config selector="order-items/order-item[@status = 'OK']">
    <resource>com.acme.ChangeItemState </resource>
    <param name="newState">COMPLETED</param>
  </resource-config>

</smooks-resource-list>
```

カスタム設定名前空間を使用して、**ChangeItemState** コンポーネントの明確でより強く型付けされた設定を定義することが可能です。カスタム設定名前空間を用いると、この例のように簡単にコンポーネントを設定できます。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:order="http://www.acme.com/schemas/smooks/order.xsd">

  <order:changeItemState itemElement="order-items/order-item[@status =
'OK']"
    newState="COMPLETED" />

</smooks-resource-list>
```

このビジターは次のようにソースコードで設定することも可能です。

```
Smooks smooks = new Smooks();

smooks.addVisitor(new ChangeItemState().setNewState("COMPLETED"),
                  "order-items/order-item[@status =
'OK']");

smooks.filterSource(new StreamSource(inReader), new
StreamResult(outWriter));
```

13.3.5. ビジターインスタンスのライフサイクル

Smooks は **ExecutionLifecycleCleanable** および **VisitLifecycleCleanable** インターフェースを使用して、ビジターコンポーネントに固有する 2 つのコンポーネントライフサイクルイベントをサポートします。

13.3.5.1. ExecutionLifecycleCleanable

このライフサイクルインターフェースを実装するビジターコンポーネントは、**Smooks.filterSource** 後のライフサイクル操作を実行できます。

```
public interface ExecutionLifecycleCleanable extends Visitor
{
    public abstract void executeExecutionLifecycleCleanup(
                                ExecutionContext
    executionContext);
}
```

基本の呼び出しシーケンスは次のように記述できます (**executeExecutionLifecycleCleanup** 呼び出しに注目してください)。

```
smooks = new Smooks(...);

smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
... etc ...
```

このライフサイクルメソッドは、**Smooks.filterSource** 実行ライフサイクルの周囲にスコープ指定されたリソースが、関連する **ExecutionContext** に対して確実にクリーンアップされるようにします。

13.3.5.2. VisitLifecycleCleanable

このライフサイクルインターフェースを実装するビジターコンポーネントは、**SAXVisitAfter.visitAfter** 後のライフサイクル操作を実行できます。

```
public interface VisitLifecycleCleanable extends Visitor
{
    public abstract void executeVisitLifecycleCleanup(ExecutionContext
executionContext);
}
```

基本の呼び出しシーケンスは次のように記述できます (**executeVisitLifecycleCleanup** 呼び出しに注目してください)。

```
smooks.filterSource(...);

<message>
  <target-fragment>      < --- VisitorXX.visitBefore
    Text!!                < --- VisitorXX.onChildText
    <child>               < --- VisitorXX.onChildElement
  </child>
</target-fragment>      < --- VisitorXX.visitAfter
** VisitorXX.executeVisitLifecycleCleanup **
<target-fragment>      < --- VisitorXX.visitBefore
  Text!!                < --- VisitorXX.onChildText
  <child>               < --- VisitorXX.onChildElement
  </child>
</target-fragment>      < --- VisitorXX.visitAfter
** VisitorXX.executeVisitLifecycleCleanup **
</message>
VisitorXX.executeExecutionLifecycleCleanup

smooks.filterSource(...);

<message>
  <target-fragment>      < --- VisitorXX.visitBefore
```

```

        Text!!                                < --- VisitorXX.onChildText
        <child>                                < --- VisitorXX.onChildElement
        </child>
    </target-fragment>                        < --- VisitorXX.visitAfter
    ** VisitorXX.executeVisitLifecycleCleanup **
    <target-fragment>                          < --- VisitorXX.visitBefore
        Text!!                                < --- VisitorXX.onChildText
        <child>                                < --- VisitorXX.onChildElement
        </child>
    </target-fragment>                        < --- VisitorXX.visitAfter
    ** VisitorXX.executeVisitLifecycleCleanup **
</message>
VisitorXX.executeExecutionLifecycleCleanup

```

このライフサイクルメソッドは、**SAXVisitor** 実装の単一での断片実行の周囲にスコープ指定されたりソースが、関連する **ExecutionContext** に対して確実にクリーンアップされるようにします。

13.3.6. ExecutionContext と ApplicationContext

Smooks はステート情報を保存 2 つのコンテキストオブジェクトを提供します。

ExecutionContext は、**Smooks.filterSource** メソッドの単一実行の周囲にスコープ指定されます。Smooks ビジター実装はすべて単一の **Smooks.filterSource** 実行のコンテキスト内でステートレスである必要があるため、ビジター実装は **Smooks.filterSource** メソッドの複数の同時実行にまたがって使用することが可能です。**Smooks.filterSource** の実行が完了すると、**ExecutionContext** インスタンスに保存されたすべてのデータを損失します。

ExecutionContext はビジター API のメッセージイベント呼び出しすべてに提供されます。

ApplicationContext は、関連する **Smooks** インスタンスの周囲にスコープ指定されます。たとえば、1 つの **ApplicationContext** インスタンスのみが **Smooks** インスタンスごとに存在します。このコンテキストオブジェクトを使用して、複数の **Smooks.filterSource** 実行にまたがって維持およびアクセスする必要があるデータを保存することが可能です。**ApplicationContext** クラスプロパティを宣言し、**@AppContext** アノテーションを付けると、コンポーネント (**SAXVisitor** コンポーネントを含む) は関連する **ApplicationContext** インスタンスへアクセスできます。

```

public class MySmooksComponent
{
    @AppContext
    private ApplicationContext appContext;

    // etc...
}

```

第14章 APACHE CAMEL の統合

Camel と Smooks の統合を行うと、Apache Camel 内から Smooks のすべての機能にアクセスすることが可能になります。既存の Smooks 設定を Camel のルートで使うことができます。

Smooks を Apache Camel で使用する方法は 3 つあります。

- SmooksComponent
- SmooksDataformat
- SmooksProcessor

SmooksComponent は、Smooks を用いて Camel メッセージボディーを処理したい時に使用できる Camel のモジュールです。使用するには、Camel のルート設定にルートを追加します。次のコードはそのやり方を示しています。

```
from("file://inputDir?noop=true")
.to("smooks://smooks-config.xml")
.to("jms:queue:order")
```

Smooks コンポーネントを設定するには、上記の例にある smooks-config.xml ファイルの値を編集します。上記のルート定義を見るだけでは SmooksComponent がどのようなタイプの出力を生成するか分かりません。出力のタイプは、exports 要素を介して Smooks の設定に表されます。



注記

プログラムを用いて Smooks を設定したい場合は、SmooksProcessor を用いて設定します。

Apache コンポーネントは、Smooks 設定ファイルの後に指定されているオプションを使用できます。現在、SmooksComponent が使用できるオプションは次の 1 つのみです。

- reportPath: 生成される Smooks Execution Report へのパス (ファイル名を含む) です。

SmooksDataFormat は Apache Camel の *DataFormat* です。あるデータ形式を他のデータ形式に変換し、さらに変換したデータ形式を元に戻すことが可能です。1 つの形式を別の形式に変換することのみを目的とし、他の Smooks 機能を使用する必要がない場合のみ使用してください。

このコード例は、SmooksDataFormat を使用してコンマ区切り値の文字列を Customer オブジェクトインスタンスの java.util.List へ変換する方法を表しています。

```
SmooksDataFormat sdf = new SmooksDataFormat("csv-smooks-unmarshal-
config.xml");
from("direct:unmarshal")
.unmarshal(sdf)
.convertBodyTo(List.class)
.to("mock:result");
```

SmooksProcessor を使用すると Smooks を完全に制御できます (たとえば、プログラムを用いて基盤の Smooks インスタンスを作成したい場合にこのコンポーネントを使用します)。

SmooksProcessor を使用する場合、Smooks をプログラムを用いて設定し、Smooks インスタンスをコンストラクターに渡すことができます。

次の例は Apache Camel のルートで SmooksProcessor を使用方法を表しています。

```
Smooks smooks = new Smooks("edi-to-xml-smooks-config.xml");
ExecutionContext context = smooks.createExecutionContext();
...
SmooksProcessor processor = new SmooksProcessor(smooks, context);

from("file://input?noop=true")
    .process(processor)
    .to("mock:result");

SmooksComponent と同様、Smooks が生成する結果タイプを指定していません（ある場合）。結
果タイプは exports 要素を使用して Smooks 設定に表現するか、次のようにプログラムを用い
て同じように表現します。

Smooks smooks = new Smooks();
ExecutionContext context = smooks.createExecutionContext();
smooks.setExports(new Exports(StringResult.class));
SmooksProcessor processor = new SmooksProcessor(smooks, context);
...
from("file://input?noop=true")
    .process(processor)
    .to("mock:result");
```

表14.1 Camel プロパティ

| 名前 | 説明 |
|-------------------|--|
| camel-dataformat | この例は SmooksDataFormat の使用 方法を表しています (DataFormat は Camel の org.apache.camel.spi.DataFormat を 実装するクラスです)。 |
| camel-integration | この例は Camel SmooksComponent. ("smooks://file:./configs/smooks- config.xml") の使用方法を表していま す。 |
| splitting-camel | この例は、Smooks と Apache Camel を使用して UN/EDIFACT メッセージインターチェンジを処理 し、個別のインターチェンジメッ セージを Java および XML 断片に分 割し、Apache Camel を使用して断 片をルーティングする方法を表して います。 |

付録A JBOSS ENTERPRISE SOA PLATFORM に SMOOKS を統合する

JBoss Enterprise SOA Platform のメッセージトランスフォーメーション機能は、SmooksAction コンポーネントによって提供されます。このコンポーネントは、Smooks フレームワークをパイプラインを処理している ESB アクションヘプラグできるようにする ESB アクションモジュールです。Smooks を ESB に適用するのはややこしいと感じる新規ユーザーもいるため、Smooks を公開する ESB アクションのみを設定し、設定ファイルが必要であることを念頭に置いて作業することが重要となります。



注記

JBoss Enterprise SOA Platform 製品にはトランスフォーメーションクイックスタートのサンプルが複数含まれています。特に **transform_** クイックスタートに注目してください。XML2XML の例はトランスフォーメーションについて学びたいユーザーに適しています。

最も基本的な設定では、このアクションは Smooks 設定ファイルを参照する resource-config プロパティを使用します。

resource-config プロパティの値は、**URIResourceLocator** クラスによって定義されたように URI ベースのリソースになります。

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
  <property name="resource-config" value="/smooks-config.xml" >
</property>
</action>
```

A.1. 入出力の設定

アクションは **MessagePayloadProxy** を使用してメッセージロードにアクセスします。デフォルトでは、Message.Body.DEFAULT_LOCATION の場所よりペイロードを取得するように設定されています。また、処理後はこの場所に返されます。



注記

これらのデフォルト設定を上書きするには、get-payload-location および set-payload-location アクションプロパティを変更します。

A.2. JAVA 出力の設定

メッセージを Java オブジェクトに変換するには、有用な例を提供する **Transform_XML2POJO** クイックスタートを最初に確認してください。

構築された Java オブジェクトモデルをモデル駆動型トランスフォーメーションの一部として使用したり、パイプラインの SmooksAction の後に続く他の ESB アクションインスタンスによって使用可能にすることができます。

後続のパイプラインアクションインスタンスはこのような Java オブジェクトグラフを使用できます。これは、このアクションによって Java オブジェクトグラフが ESB メッセージ出力へ添付され、後続のアクションへ入力されるためです。Smooks Java Bean 設定に定義された通り、BeanId オブジェクト

を直接ベースとするキー下で `Message.getBody().add(...)` を介して Java オブジェクトグラフは添付されます。そのため、`Body.get(BeanId)` 呼び出しを実行することで、ESB メッセージボディーよりオブジェクトが使用可能になります。

java-output-location プロパティを追加すると、完全な Java オブジェクトマップと出力メッセージへ添付することもできます。

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
    <property name="resource-config" value="/smooks-config.xml" >
</property>
    <property name="java-output-location" value="order-message-
objects-map" ></property>
</action>
```

マップを Default Message Body Location へバインドする簡単な方法は次の通りです。

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
    <property name="resource-config" value="/smooks-config.xml" >
</property>
    <property name="java-output-location" value="$default" >
</property>
</action>
```

A.3. プロファイルベースのトランスフォーメーション

以下の例では、3つのソースと1つのターゲット宛先の間でメッセージが交換されます。

エンタープライズサービスバスは、宛先サービスへ送信する前に3つのソースから提供されたメッセージ (異なる形式) を Java オブジェクトに変換する必要があります。

ESB の観点では、宛先に対する単一のサービス設定が存在します。

```
<service category="ServiceCat" name="TargetService"
description="Target Service">
    <listeners>
        <!-- Message listeners for getting the message into the action
pipeline... -->
        <jms-listener name="Gateway-Listener"
busidref="quickstartGwChannel" is-gateway="true"></jms-listener>
        <jms-listener name="Service-Listener"
busidref="quickstartEsbChannel"></jms-listener>
    </listeners>
    <actions>

        <action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
            <property name="resource-config" value="/smooks-
config.xml"></property>
        </action>
```



```

        <!-- An action to process the Java Object(s) -->
        <action name="process" class="com.acme.JavaProcessor" >
</action>

    </actions>
</service>

```

SmooksAction の設定で分かるように、単一のトランスフォーメーション設定ファイル (**smooks-config.xml**) のみを定義する必要があります。次に、ソースごとに 1 つ、計 3 つの異なるトランスフォーメーションを定義する必要があります。これには、*Smooks Message Profiling* を使用します。

定義を 3 つの Smooks 設定ファイル

(**from_source1.xml**、**from_source2.xml**、**from_source3.xml**) に保存します。各ファイルに設定セットの **default-target-profile** を指定します。

```

    <smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.0.xsd"
default-target-profile="from:source1">
        <!-- Source1 to Target Java message transformation resource
configurations... -->
    </smooks-resource-list>

```

次に、トップレベルの **smooks-config.xml** に追加します。

```

<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.0.xsd">
    <import file="classpath:/from_source1.xml" ></import>
    <import file="classpath:/from_source2.xml" ></import>
    <import file="classpath:/from_source3.xml" ></import>
</smooks-resource-list>

```

これで、それぞれ一意のプロファイル名で定義された 3 つのトランスフォーメーションを持つ単一の SmooksAction インスタンスをロードするシステムが設定されました。

3 つのトランスフォーメーションのどれを指定のメッセージに適用するか SmooksAction が認識できるようにするには、メッセージが SmooksAction にフローする前にメッセージの **from** プロパティを設定する必要があります。これには、ソース自体または SmooksAction 以前のコンテンツベースのアクションで設定を行います (下記の **transform_XML2POJO2** クイックスタートの詳細を参照してください)。



注記

JBoss Enterprise SOA Platform は **from** 以外に、**from-type**、**to**、**to-type** もサポートします。さらに複雑で交換ベースのトランスフォーメーションを実現するため、これらのプロパティを組み合わせ使用することも可能です。

A.4. TRANSFORM_XML2POJO2

上記の基本的なシナリオは、**/samples/quickstarts/transform_XML2POJO2/** というクイックスタートで実装されています。このクイックスタートではメッセージソースが 2 つあります。

クイックスタートは、受信メッセージの **from** プロファイルを検出および設定するアクションパイプライン上で **Groovy** スクリプトを実行します。このプロセスで使用されるファイルは次の通りです。

- **jboss-esb.xml**: JBoss ESB 設定ファイルです。
- **smooks-config.xml**: トップレベルのトランスフォーメーション設定が含まれているファイルです。
- **from-dvdstore.xml**: トップレベルの **smooks-config.xml** ファイルへインポートされる DVD Store のメッセージトランスフォーメーション設定です (プロファイル設定を確認してください)。これは DVD Store のメッセージを Java オブジェクトに変換するための設定です。
- **from-petstore.xml**: トップレベルの **smooks-config.xml** ファイルへインポートされる Pet Store のメッセージトランスフォーメーション設定です (プロファイル設定を確認してください)。これは Pet Store のメッセージを Java オブジェクトに変換するための設定です。
- **check-origin.groovy**: 内容を基に各メッセージのオリジンを判断する簡単な **Groovy** スクリプトです。

付録B GNU LESSER GENERAL PUBLIC LICENSE 2.1

GNU LESSER GENERAL PUBLIC LICENSE Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts
as the successor of the GNU Library Public License, version 2, hence
the version number 2.1.]

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it. You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use,
not price. Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights. These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you. You must make sure that they, too, receive or can get the source
code. If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that
there is no warranty for the free library. Also, if the library is
modified by someone else and passed on, the recipients should know

that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a

"work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the

Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library

facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME

THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the

library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

付録C 改訂履歴

| | | |
|---|------------------------|------------------------------------|
| 改訂 5.2.0-0.1.400 Rebuild with publican 4.0.0 | 2013-10-31 | Rüdiger Landmann |
| 改訂 5.2.0-0.1 Translation files synchronised with XML sources 5.2.0-0 | Tue Feb 19 2013 | Junko Ito |
| 改訂 5.2.0-0 Smooks Wiki と同期するためのリベースを実行 | Wed Jun 15 2011 | David Le Sage |
| 改訂 5.1.0-0 SOA 5.1.0 向けの Smooks 1.3 を更新 「4.4 プログラミングによる設定」を更新 11.2.4 Routing to a Database with SQL を更新 2.3.1 の追加 2.3.2 Namespace Declaration の追加 2.7 Terminating the Filtering Process の追加 2.9 Filter Settings の追加 3 Extending Smooks の追加 4.3.1 Pre-processing Binding Values と追加 4.3.2 Creating beans using a factory を追加 4.3.3 Extending Life-Cycle Binding を追加 4.5 Direct Value Binding を追加 7.1.1 String manipulation functions を追加 7.2 Processing Fixed Length を追加 7.6 String manipulation functions for readers を追加 SOA-2134 - Smooks XSLT 処理の変更を文書化 | Fri Feb 18 2011 | Darrin Mison |
| 改訂 5.0.2-0 SOA 5.0.2 向けに更新 | Wed May 26 2010 | David Le Sage |
| 改訂 5.0.1-0 SOA 5.0.1 向けに更新 | Tue Apr 20 2010 | David Le Sage |
| 改訂 5.0.0-0 第 1 版 | Mon Oct 19 2009 | David Le Sage, Darrin Mison |