# Red Hat Fuse 7.13

## Apache CXF 开发指南

使用 Apache CXF Web 服务开发应用程序

Last Updated: 2024-06-12

# Red Hat Fuse 7.13 Apache CXF 开发指南

使用 Apache CXF Web 服务开发应用程序

## 法律通告

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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, the Red Hat 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 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.

## 摘要

使用 Apache CXF 开发 Web 服务指南。

# 目录

# 使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看我们的 CTO Chris Wright 信息。

# 部分 I. 编写 WSDL 合同

这部分描述了如何使用 WSDL 定义 Web 服务接口。

# 第 1 章 WSDL 合同简介

**摘要**

WSDL 文档使用 Web 服务描述语言和多个可能扩展来定义服务。这些文档具有逻辑部分和集合部分。合同摘要部分定义了实施中立数据类型和消息方面的服务。文档的具体部分定义了端点如何实现服务与外部世界交互。

推荐的设计服务的方法是在 WSDL 和 XML Schema 中定义您的服务，然后再编写任何代码。在手动编辑 WSDL 文档时，您必须确保文档有效，以及正确的。为此，您必须对 WSDL 有一定的了解。您可以在 W3C 网站 www.w3.org 中找到标准。

## 1.1. WSDL 文档的结构

### 概述

WSDL 文档最简单的是根 **定义** 元素中包含的元素集合。这些元素描述了一个服务，以及如何访问实施该服务的端点。

WSDL 文档有两个不同的部分：

- 在实施中定义服务的逻辑 部分
- 定义 如何实现该服务的端点在网络上公开的部分

### 逻辑部分

WSDL 文档的逻辑部分包含 **类型**、**消息** 和 **portType** 元素。它描述了服务的接口，以及由服务交换的消息。在 **type** 元素中，XML Schema 用于定义组成消息的数据结构。很多 **消息** 元素用于定义服务使用的消息结构。**portType** 元素包含一个或多个 **operation** 元素，用于定义由服务公开的操作发送的消息。

### concrete 部分

WSDL 文档的具体部分包含 **绑定** 和**服务** 元素。它描述了实现该服务的端点如何连接到外部世界。**绑定** 元素描述 **消息** 元素描述的数据单元如何映射到 concrete, on-the-wire 数据格式，如 SOAP。**服务** 元素包含一个或多个 **port** 元素，用于定义实施该服务的端点。

## 1.2. WSDL 元素

WSDL 文档由以下元素组成：

- **定义** - WSDL 文档的根元素。此元素的属性指定 WSDL 文档的名称、文档的目标命名空间以及 WSDL 文档中的命名空间的简短定义。
- **Type** - 组成服务使用的消息的构建块的数据单元的 XML 架构定义。有关定义数据类型的详情，请参考 第 2 章 *定义逻辑数据单元*。
- Message - 在调用服务操作期间交换的消息的描述。这些元素定义组成服务的操作的参数。有关定义信息的详情，请参考 **第 3 章 *定义服务使用的逻辑消息*。**
- **portType - 描述服务逻辑接口的 操作 元素集合。有关定义端口类型的详情，请参考 第 4 章**

*定义逻辑接口*。

- operation - 服务执行的操作的描述。操作由调用操作时在两个端点之间传递的消息定义。有关定义操作的信息，请参考 “操作”一节。

- binding - 端点的数据格式规格。binding 元素定义抽象信息如何映射到端点使用的数据格式。此元素是指定参数顺序和返回值等特定元素。

- service - 相关端口 元素 的集合。这些元素是用于组织端点定义的存储库。

- port - 绑定和物理地址定义的端点。这些元素将所有抽象定义结合在一起，以及传输详情的定义，它们定义了公开服务的物理端点。

## 1.3. 设计合同

要为您的服务设计 WSDL 合同，您必须执行以下步骤：

1. 定义服务使用的数据类型。
2. 定义服务使用的消息。
3. 为您的服务定义接口。
4. 定义各个接口使用的消息之间的绑定，以及线上数据的指示。
5. 定义每个服务的传输详情。

# 第 2 章 定义逻辑数据单元

**摘要**

当描述 WSDL 合同复杂数据类型中的服务时，将使用 XML 架构定义为逻辑单元。

## 2.1. 逻辑单元简介

在定义服务时，您必须考虑的第一个事项是如何表示用作公开操作参数的数据。与使用固定数据结构的编程语言编写的应用程序不同，服务必须以逻辑单元定义其数据，以供任意数量的应用程序使用。这涉及两个步骤：

1. 将数据分割成可映射到服务物理实现的数据类型中的逻辑单元
2. 将逻辑单元合并到端点间传递的消息中，以便执行操作

本章讨论了第一步。*第 3 章 定义服务使用的逻辑消息* 讨论第二步。

## 2.2. 将数据映射到逻辑数据单元

**概述**

用于实现服务的接口将代表操作参数的数据定义为 XML 文档。如果您要为已实现的服务定义接口，您必须将实施操作的数据类型转换为可汇编为消息的 XML 元素。如果您要从头开始，您必须确定从中构建消息的构建块，以便它们对实施角度有意义。

**用于定义服务数据单元的可用类型系统**

根据 WSDL 规范，您可以使用您选择的任何类型系统来定义 WSDL 合同中的数据类型。但是，W3C 规范指出 XML Schema 是 WSDL 文档的首选规范类型系统。因此，XML Schema 是 Apache CXF 中的内部类型系统。

**XML Schema 作为类型系统**

XML 架构用于定义 XML 文档的结构。这可以通过定义组成文档的元素来完成。这些元素可以使用原生 XML 架构类型，如 xsd:int，或者可以使用用户定义的类型。用户定义的类型是使用 XML 元素的组合

构建的，或通过限制现有类型来定义。通过组合类型定义和元素定义，您可以创建可能包含复杂数据的 XML 文档。

在 WSDL XML Schema 中使用时，定义 XML 文档的结构，其中包含用于与服务交互的数据。在定义服务使用的数据单元时，您可以将其定义为指定消息部分结构的类型。您还可以将数据单元定义为组成消息部分的元素。

### 创建数据单元的注意事项

您可以考虑创建在实施服务时直接映射到您要使用的类型的逻辑数据单元。虽然这种方法可以正常工作，但遵循构建 RPC 风格的应用程序模型，但不一定是构建服务导向型架构的理想选择。

Web 服务互操作性机构的 WS-I 基本配置集提供了多个定义数据单元的指南，并可通过 http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES 访问。此外，W3C 还提供了以下准则，以使用 XML 架构来代表 WSDL 文档中的数据类型：

- 使用元素而不是属性。
- 不要使用特定于协议的类型作为基本类型。

## 2.3. 在合同中添加数据单元

### 概述

根据您选择如何创建 WSDL 合同的方式，创建新数据定义需要不同的知识。Apache CXF GUI 工具提供了很多用于描述使用 XML 架构数据类型的辅助工具。其他 XML 编辑器提供不同的帮助级别。无论您选择哪种编辑器，最好了解结果合同应是什么样子。

### 流程

定义 WSDL 合同中使用的数据涉及以下步骤：

1. 确定合同描述的接口中使用的所有数据单元。
2. 在您的合同中创建 type 元素。

3. 创建一个 schema 元素，如 例 2.1 “WSDL 合同的 schema 条目” 所示，作为 type 元素的子元素。

   targetNamespace 属性指定定义新数据类型的命名空间。最佳实践是定义提供目标命名空间访问权限的命名空间。不应更改剩余的条目。

   **例 2.1. WSDL 合同的 schema 条目**

   ```
   <schema targetNamespace="http://schemas.iona.com/bank.idl"
           xmlns="http://www.w3.org/2001/XMLSchema"
           xmlns:xsd1="http://schemas.iona.com/bank.idl"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   ```

4. 对于作为元素集合的每个复杂类型，请使用 complexType 元素定义数据类型。请参阅 第 2.5.1 节 “定义数据结构”。

5. 对于每个数组，使用 complexType 元素定义数据类型。请参阅 第 2.5.2 节 “定义数组”。

6. 对于从简单类型派生的每个复杂类型，请使用 simpleType 元素定义数据类型。请参阅 第 2.5.4 节 “根据限制定义类型”。

7. 对于每个枚举的类型，使用 simpleType 元素定义数据类型。请参阅 第 2.5.5 节 “定义枚举类型”。

8. 对于每个元素，使用 element 元素 定义它。请参阅 第 2.6 节 “定义元素”。

## 2.4. XML 架构简单类型

### 概述

如果消息部分将是一个简单的类型，则不需要为其创建类型定义。但是，合同中定义的接口使用的复杂类型使用简单类型定义。

### 输入简单类型

XML 架构简单类型主要放在合同的 type 部分中使用的元素中。它们也用于限制元素和扩展 元素的 基

本属性 。

简单的类型总是使用 xsd 前缀输入。例如，要指定一个元素是 int 类型，您将在其 type 属性中输入 xsd:int，如 例 2.2 “使用简单类型定义元素” 所示。

**例 2.2. 使用简单类型定义元素**

```
<element name="simpleInt" type="xsd:int" />
```

### 支持的 XSD 简单类型

Apache CXF 支持以下 XML 架构简单类型：

- xsd:string
- xsd:normalizedString
- xsd:int
- xsd:unsignedInt
- xsd:long
- xsd:unsignedLong
- xsd:short
- xsd:unsignedShort
- xsd:float

- **xsd:double**
- **xsd:boolean**
- **xsd:byte**
- **xsd:unsignedByte**
- **xsd:integer**
- **xsd:positiveInteger**
- **xsd:negativeInteger**
- **xsd:nonPositiveInteger**
- **xsd:nonNegativeInteger**
- **xsd:decimal**
- **xsd:dateTime**
- **xsd:time**
- **xsd:date**
- **xsd:QName**

- **xsd:base64Binary**
- **xsd:hexBinary**
- **xsd:ID**
- **xsd:token**
- **xsd:language**
- **xsd:Name**
- **XSd:NCName**
- **xsd:NMTOKEN**
- **xsd:anySimpleType**
- **xsd:anyURI**
- **xsd:gYear**
- **xsd:gMonth**
- **xsd:gDay**
- **xsd:gYearMonth**

- xsd:gMonthDay

### 2.5. 定义复杂数据类型

**摘要**

XML Schema 提供灵活、强大的机制，用于从其简单数据类型构建复杂数据结构。您可以通过创建一系列元素和属性来创建数据结构。您还可以扩展您定义的类型，以创建更复杂的类型。

除了构建复杂数据结构外，您还可以描述专用类型，如枚举的类型、具有特定值的数据类型，或者通过扩展或限制原语类型来遵循特定模式的数据类型。

#### 2.5.1. 定义数据结构

**概述**

在 XML Schema 中，作为数据字段集合的数据单元使用 complexType 元素定义。指定复杂类型需要三段信息：

1. 定义类型的 name 属性在 complexType 元素的 name 属性中指定。
2. complexType 的第一个子元素描述了在线上放置结构字段的行为。请参阅 “复杂类型差异”一节。
3. 定义结构的每个字段都在 element 元素中定义，它们是 complexType 元素的 grandchildren。请参阅 “定义结构的部分”一节。

例如： 例 2.3 “简单结构” 中显示的结构在 XML Schema 中定义为具有两个元素的复杂类型。

**例 2.3. 简单结构**

```
struct personalInfo
{
  string name;
```

```
  int age;
};
```

例 2.4 “复杂类型” 显示 例 2.3 “简单结构” 中显示的结构的一个可能的 XML 架构映射 例 2.4 “复杂类型” 会生成含有两个元素的消息： name 和 age。

.

**例 2.4. 复杂类型**

```
<complexType name="personalInfo">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
  </sequence>
</complexType>
```

**复杂类型差异**

XML Schema 有三种方法来描述复杂类型字段在以 XML 文档表示，并在有线上传递时如何组织复杂类型字段。complexType 元素的第一个子元素决定使用哪些复杂类型。表 2.1 “复杂的类型描述符元素” 显示用于定义复杂类型行为的元素。

**表 2.1. 复杂的类型描述符元素**

| 元素 | 复杂类型行为 |
| --- | --- |
| **sequence** | 所有复杂的类型字段都可以存在，它们必须按照类型定义中指定的顺序。 |
| **all** | 所有复杂的类型字段都可能存在，但它们可以是任何顺序。 |
| **choice** | 只有结构中的一个元素可以放在消息中。 |

如果使用 choice 元素定义结构，如 例 2.5 “简单的复杂选择类型” 所示，它会生成一条带有 name 元素或 age 元素的消息。

**例 2.5. 简单的复杂选择类型**

```
<complexType name="personalInfo">
  <choice>
```

```
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </choice>
</complexType>
```

定义结构的部分

您可以使用 element 元素 定义组成结构的数据字段。每个 complexType 元素应该至少包含一个 element 元素。complexType 元素中的每个元素代表定义的数据结构中的一个字段。

要完全描述数据结构中的字段，element 元素有两个所需的属性：

- name 属性指定 data 字段的名称，它必须在定义的复杂类型内唯一。
- type 属性指定存储在字段中的数据类型。类型可以是 XML 架构简单类型之一，也可以是合同中定义的命名复杂类型之一。

除了 name 和 type 外，element 元素还有两个常用的可选属性： minOcurrs 和 maxOccurs。这些属性将绑定到字段在结构中发生的次数。默认情况下，每个字段仅在复杂类型中发生一次。使用这些属性，您可以更改字段必须或可以出现在结构中的次数。例如，您可以定义一个字段 previousJobs，它必须至少发生三次，且不超过 7 次，如 例 2.6 “发生限制的简单复杂类型” 所示。

**例 2.6. 发生限制的简单复杂类型**

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
    <element name="previousJobs" type="xsd:string:
          minOccurs="3" maxOccurs="7"/>
  </all>
</complexType>
```

您还可以通过将 minOccurs 设置为 0 来使 age 字段是可选的，如 例 2.7 “将 minOccurs 设置为 0 的简单复杂类型” 所示。在这种情况下，可以省略 年龄，数据仍会有效。

**例 2.7. 将 minOccurs 设置为 0 的简单复杂类型**

```
<complexType name="personalInfo">
  <choice>
```

```
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int" minOccurs="0"/>
  </choice>
</complexType>
```

**定义属性**

**在 XML 文档中，属性包含在元素的标签中。例如，在以下代码中的 complexType 元素中，name 是属性。要为复杂类型指定属性，您可以在 complexType 元素定义中定义 attribute 元素。attribute 元素只能在 所有、序列 或 choice 元素后显示。为每个复杂类型的属性指定一个 属性 元素。任何 属性 元素都必须是 complexType 元素的直接子项。**

**例 2.8. 具有属性的复杂类型**

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="previousJobs" type="xsd:string"
          minOccurs="3" maxOccurs="7"/>
  </all>
  <attribute name="age" type="xsd:int" use="required" />
</complexType>
```

**在前面的代码中，attribute 元素指定 personalInfo 复杂类型具有 age 属性。attribute 元素具有这些属性：**

- **name - 指定标识属性的字符串所需的属性。**
- **type - 指定存储在字段中的数据类型。类型可以是 XML 架构简单类型之一。**
- **use - 指定是否需要复杂的类型具有此属性的可选属性。有效值为 required 或 optional。默认为属性是可选的。**

**在 attribute 元素中，您可以指定可选的 default 属性，它可让您为属性指定默认值。**

### 2.5.2. 定义数组

**概述**

Apache CXF 支持两种方法在合同中定义数组。第一种定义具有单个元素的复杂类型，其 maxOccurs 属性的值大于一。第二个方法是使用 SOAP 阵列。SOAP 阵列提供添加的功能，如轻松定义多维阵列和传输稀疏填充阵列等功能。

### 复杂的类型数组

复杂的类型数组是序列复杂类型的一种特殊情况。只需使用单个元素定义复杂类型，并为 maxOccurs 属性指定一个值。例如，要定义一组二个浮点数，您可以使用与 例 2.9 “复杂的类型数组” 中显示的复杂类型。

**例 2.9. 复杂的类型数组**

```
<complexType name="personalInfo">
  <element name="averages" type="xsd:float" maxOccurs="20"/>
</complexType>
```

您还可以为 minOccurs 属性指定值。

### SOAP 数组

SOAP 数组通过利用 wsdl:arrayType 元素从 SOAP-ENC:Array 基本类型中定义。的语法显示在 例 2.10 “使用 wsdl:arrayType 派生的 SOAP 数组的语法” 中。确保 definitions 元素声明 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"。

**例 2.10. 使用 wsdl:arrayType 派生的 SOAP 数组的语法**

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

使用这个语法，*TypeName* 指定新定义的数组类型的名称。*elementType* 指定阵列中元素的类型。*ArrayBounds* 指定阵列中的维度数。要指定单个维度数组使用 []; 指定双维数组，请使用 [][] 或 [,]。

例如，SOA SOAP Array, SOAPStrings 在 例 2.11 “SOAP 数组的定义” 中显示，定义一个字符串的一个维度数组。wsdl:arrayType 属性指定数组元素的类型 xsd:string，以及尺寸的数量，[] 代表一个维

度。

**例 2.11. SOAP 数组的定义**

```
<complexType name="SOAPStrings">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="xsd:string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

您还可以使用一个简单元素来描述 SOAP 数组，如 SOAP 1.1 规范中所述。的语法显示在 例 2.12 “使用元素派生的 SOAP 数组的语法” 中。

**例 2.12. 使用元素派生的 SOAP 数组的语法**

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <sequence>
        <element name="ElementName" type="ElementType"
                 maxOccurs="unbounded"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

在使用此语法时，元素的 maxOccurs 属性必须始终设置为 未绑定。

### 2.5.3. 通过扩展定义类型

与大多数主要编码语言一样，XML 架构允许您创建从其他数据类型继承其部分元素的数据类型。这称为按扩展定义类型。例如，您可以创建一个名为 alienInfo 的新类型，它通过添加名为 planet 的新元素来扩展 例 2.4 “复杂类型” 中定义的 personalInfo 结构。

由扩展定义的类型有四个部分：

1. 类型的名称由 complexType 元素的 name 属性定义。

2. complexContent 元素指定新类型将有多个元素。

注意

如果您只向复杂类型添加新属性，您可以使用 simpleContent 元素。

3. 新类型派生到的类型（称为 基本 类型）在 extension 元素的基本属性中指定。

4. 新类型的元素和属性在 extension 元素中定义，这与常规复杂类型相同。

例如，ali enInfo 定义，如 例 2.13 “由扩展定义的类型” 所示。

例 2.13. 由扩展定义的类型

```
<complexType name="alienInfo">
  <complexContent>
    <extension base="xsd1:personalInfo">
      <sequence>
        <element name="planet" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

### 2.5.4. 根据限制定义类型

概述

XML Schema 允许您通过限制 XML 架构简单类型的可能值来创建新类型。例如，您可以定义一个简单的类型 SSN，它是一个正好 9 个字符的字符串。通过限制简单类型定义的新类型是使用 simpleType 元素定义的。

根据限制划分类型的定义需要三个内容：

1. 新类型的名称由 simpleType 元素的 name 属性指定。

2. 新类型派生到的简单类型（称为 基本类型 ）在 limitations 元素中指定。请参阅 “指定基本类型”一节。

3. 规则称为 facets，定义对基础类型实施的限制将定义为 limit 元素的子项。请参阅 “定义限制”一节。

### 指定基本类型

基本类型是正在限制定义新类型的类型。它使用一个 limit 元素 来指定。限制 元素是 simpleType 元素的唯一子级，它具有一个属性 base，用于指定基本类型。基本类型可以是任何 XML 架构简单类型。

例如，通过限制 xsd:int 的值来定义一个新类型，您可以使用类似 例 2.14 “使用 int 作为基本类型” 所示的定义。

**例 2.14. 使用 int 作为基本类型**

```
<simpleType name="restrictedInt">
  <restriction base="xsd:int">
    ...
  </restriction>
</simpleType>
```

### 定义限制

定义对基本类型的限制的规则称为 *facets*。facet 是指一个属性( 值为 )的元素，用于定义如何强制实施 facet。可用的 facet 及其 有效值 设置取决于基本类型。例如，xsd:string 支持六个 facets，包括：

- length
- minLength
- maxLength
- pattern

- 空格
- Enumeration

每个 facet 元素都是 Limit 元素 的子级。

**Example**

例 2.15 “SSN 简单类型描述” 显示了一个简单的类型 SSN 的示例，它代表社交安全号码。生成的类型是一个字符串，格式为 xxx-xx-xxxx。<SSN>032-43-9876<SSN> 是此类型的元素的有效值，但 <SSN>032439876</SSN> 不是。

**例 2.15. SSN 简单类型描述**

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}"/>
  </restriction>
</simpleType>
```

### 2.5.5. 定义枚举类型

**概述**

XML 架构中的枚举类型是按限制划分的特殊定义情况。它们通过使用所有 XML 架构原语类型支持的枚举 情况进行描述。与大多数现代编程语言中枚举的类型一样，此类型的变量只能具有指定的值之一。

**在 XML Schema 中定义枚举**

例 2.16 “Enumeration 的语法” 中显示定义枚举的语法。

**例 2.16. Enumeration 的语法**

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value"/>
    <enumeration value="Case2Value"/>
    ...
```

```
    <enumeration value="CaseNValue"/>
  </restriction>
</simpleType>
```

*EnumName* 指定枚举类型的名称。*EnumType* 指定问题单值的类型。*CaseNValue*，其中 *N* 是一个或多个，指定枚举的每个特定情况的值。Enumerated 类型可以具有任意数量的问题单值，但它是从一个简单的类型派生的，因此一次只有一个 case 值有效。

**Example**

例如，一个由 enumeration widgetSize 定义的一个元素的 XML 文档（在 例 2.17 “widgetSize enumeration” 中显示），如果它包含 < widgetSize >big</widgetSize>，则它将无效，但如果它包含 <widgetSize>big,mungo</widgetSize>，它将无效。

**例 2.17. widgetSize enumeration**

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
  </restriction>
</simpleType>
```

## 2.6. 定义元素

XML 架构中的元素代表从架构生成的 XML 文档中的一个元素实例。最基本的元素由单个 元素 组成。与用于定义复杂类型成员的 element 元素类似，它们有三个属性：

- name - 在 XML 文档中显示指定元素名称的必要属性。
- type - 指定元素的类型。类型可以是任何 XML 架构原语类型，也可以是合同中定义的任何指定复杂类型。如果类型有 in-line 定义，则可以省略此属性。
- nillable - 指定是否在文档中完全忽略一个元素。如果 nillable 设为 true，则元素可以在使用 schema 生成的任何文档中省略。

元素也可以具有 in-line 类型定义。使用 complexType 元素或 simpleType 元素指定 in-line 类型。指定数据类型是否复杂或简单后，您可以使用每种类型数据的工具定义所需的任何类型的数据。不鼓励使用

**in-line 类型定义，因为它们无法重复使用。**

# 第 3 章 定义服务使用的逻辑消息

## 摘要

服务由调用其操作时交换的消息定义。在 WSDL 合同中，这些消息使用 message 元素来定义。消息由一个或多个使用 part 元素定义的部分组成。

## 概述

服务的操作是通过指定调用操作时交换的逻辑消息来定义的。这些逻辑消息定义通过网络作为 XML 文档传递的数据。它们包含作为方法调用的一部分的所有参数。 逻辑消息使用合同中的 message 元素定义。每个逻辑消息由一个或多个部分组成，这些部分在部分元素 中定义。

虽然您的消息可以将每个参数列为单独的部分，但建议的做法是使用单一部分来封装操作所需的数据。

## 消息和参数列表

服务公开的每个操作只能有一个输入消息和一个输出消息。输入消息定义服务在调用操作时收到的所有信息。输出消息定义服务在操作完成后返回的所有数据。故障消息定义服务在发生错误时返回的数据。

此外，每个操作可以具有任意数量的故障信息。故障消息定义服务遇到错误时返回的数据。这些消息通常只有一个部分，它为消费者提供足够信息以了解错误。

## 用于与旧系统集成的消息设计

如果您要将现有应用程序定义为服务，您必须确保实现该操作的方法所使用的每个参数都在消息中表示。您还必须确保返回值包含在操作的输出消息中。

定义您的消息的一种方法是 RPC 样式。使用 RPC 样式时，您可以在方法参数列表中为每个参数定义一个部分消息。每个消息部分都基于合同的 type 元素中定义的 类型。您的输入消息包含方法中的每个输入参数的一个部分。您的输出消息包含每个输出参数的一个部分，以及表示返回值的部分（如果需要）。如果参数同时是输入和输出参数，它将被列为输入消息和输出消息的一部分。

当启用使用传输的传统系统（如 Tibco 或 CORBA ）时，RPC 样式消息定义非常有用。这些系统围绕程序和方法设计。因此，它们最容易使用类似于被调用操作的参数列表的消息建模。RPC 样式还在服务及其公开的应用程序之间进行更干净的映射。

**SOAP 服务的消息设计**

RPC 样式可用于对现有系统建模，但服务的社区强烈优先选择嵌套式文档样式。在嵌套文档样式中，每条消息都有一个部分。消息的部分引用合同的 type 元素中定义的 wrapper 元素。wrapper 元素具有以下特征：

- 它是包含一系列元素的复杂类型。如需更多信息，请参阅 第 2.5 节 “定义复杂数据类型”。
- 如果它是输入消息的打包程序：
  - 它为每个方法的输入参数都有一个元素。
  - 其名称与与其关联的操作的名称相同。
- 如果它是输出消息的打包程序：
  - 它为每个方法的输出参数有一个元素，以及每个方法的 inout 参数的一个元素。
  - 其第一个元素代表方法的返回参数。
  - 其名称将通过将 Response 附加到与打包程序关联的操作名称来生成。

**消息命名**

合同中的每个消息都必须在其命名空间内具有唯一的名称。建议您使用以下命名约定：

- 消息仅应由单个操作使用。
- 输入消息名称通过将 Request 附加到操作名称来形成。

- 通过向操作名称附加 Response 来形成输出消息名称。
- 错误消息名称应该代表错误的原因。

### 消息部分

消息部分是逻辑消息的正式数据单元。每个部分都使用 part 元素定义，并由 name 属性标识，type 属性或指定其数据类型的元素属性。数据类型属性列在 表 3.1 “部分数据类型属性” 中。

**表 3.1. 部分数据类型属性**

| 属性 | 描述 |
| --- | --- |
| **元素**="*elem_name*" | 部分的数据类型由名为 *elem_name* 的元素定义。 |
| **type=**"*type_name*" | 部分的数据类型由名为 *type_name* 的类型定义。 |

消息被允许重复使用部分名称。例如，如果方法有一个参数 foo，它通过引用或是 in/out 传递，则它可以是请求消息和响应消息的一部分，如 例 3.1 “重复使用部分” 所示。

**例 3.1. 重复使用部分**

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
<message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
<message>
```

### EXAMPLE

例如，假设您有一个存储个人信息的服务器，并提供了一种方法，它根据员工的 ID 号返回员工的数据。查找数据的方法签名与 例 3.2 “personalInfo 查找方法” 类似。

**例 3.2. personalInfo 查找方法**

```
personalInfo lookup(long empId)
```

这个方法签名可以映射到 例 3.3 “RPC WSDL 消息定义” 中显示的 RPC 风格 WSDL 片段。

**例 3.3. RPC WSDL 消息定义**

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int"/>
<message/>
<message name="personalLookupResponse>
  <part name="return" element="xsd1:personalInfo"/>
<message/>
```

它还可以映射到 例 3.4 “嵌套文档 WSDL 消息定义” 中显示的嵌套文档风格的 WSDL 片段。

**例 3.4. 嵌套文档 WSDL 消息定义**

```
<wsdl:types>
  <xsd:schema ... >
  ...
  <element name="personalLookup">
    <complexType>
      <sequence>
        <element name="empID" type="xsd:int" />
      </sequence>
    </complexType>
  </element>
  <element name="personalLookupResponse">
    <complexType>
      <sequence>
        <element name="return" type="personalInfo" />
      </sequence>
    </complexType>
  </element>
  </schema>
</types>
<wsdl:message name="personalLookupRequest">
  <wsdl:part name="empId" element="xsd1:personalLookup"/>
<message/>
<wsdl:message name="personalLookupResponse">
  <wsdl:part name="return" element="xsd1:personalLookupResponse"/>
<message/>
```

# 第 4 章 定义逻辑接口

## 摘要

逻辑服务接口使用 portType 元素来定义。

## 概述

逻辑服务接口使用 WSDL portType 元素来定义。portType 元素是抽象操作定义的集合。每个操作都由用于完成操作所代表的事务的输入、输出和错误消息定义。当生成代码来实现由 portType 元素定义的服务接口时，每个操作都会转换为包含合同中指定的输入、输出和故障消息定义的参数的方法。

## PROCESS

要在 WSDL 合同中定义逻辑接口，您必须执行以下操作：

1. 创建一个 portType 元素，使其包含接口定义，并为它赋予一个唯一的名称。请参阅 “端口类型”一节。
2. 为接口中定义的每个操作创建一个 operation 元素。请参阅 “操作”一节。
3. 对于每个操作，指定用于表示操作的参数列表、返回类型和例外的消息。请参阅 “操作消息”一节。

## 端口类型

WSDL portType 元素是逻辑接口定义中的 root 元素。虽然许多 Web 服务实现将 portType 元素直接映射到生成的实现对象，但逻辑接口定义不指定实施服务提供的确切功能。例如，名为 ticketSystem 的逻辑接口可能会导致一个实施，该实施销售证书票据或问题解析票据。

portType 元素是 WSDL 文档的单元，它映射到绑定中，以定义端点公开定义的服务所使用的物理数据。

WSDL 文档中的每个 portType 元素都必须具有唯一的名称，该名称使用 name 属性指定，并且由一组操作组成，这些操作元素在操作元素中描述。WSDL 文档可描述任意数量的端口类型。

### 操作

使用 WSDL 操作 元素定义的逻辑操作定义两个端点之间的交互。例如，对检查帐户平衡的请求和小部件的顺序都可以定义为操作。

在 portType 元素中定义的每个操作都必须具有唯一的名称，使用 name 属性指定。定义操作需要 name 属性。

### 操作消息

逻辑操作由一组元素组成，代表端点之间进行通信的逻辑消息来执行操作。可以描述操作的元素列在 表 4.1 “操作消息元素” 中。

**表 4.1. 操作消息元素**

| 元素 | 描述 |
| --- | --- |
| 输入 | 指定在发出请求时客户端端点发送到服务提供商的消息。此消息的部分对应于操作的输入参数。 |
| output | 指定服务提供商发送到客户端端点以响应请求的消息。此消息的部分内容与服务提供商可以更改的任何操作参数对应，如通过引用传递的值。这包括操作的返回值。 |
| Fulting | 指定用于在端点之间传达错误条件的消息。 |

至少需要一个 输入 或一个 output 元素操作。一个操作可以同时具有 输入和输出 元素，但它只能有一个。不需要操作具有任何 fault 元素，但可以根据需要具有任意数量的 fault 元素。

这些元素在 表 4.2 “输入和输出元素的属性” 中列出的两个属性。

**表 4.2. 输入和输出元素的属性**

| 属性 | 描述 |
| --- | --- |
| name | 标识消息，以便在将操作映射到 concrete 数据格式时引用它。名称在括起端口类型中必须是唯一的。 |
| message | 指定描述要发送或接收数据的抽象信息。message 属性的值必须与 WSDL 文档中定义的其中一个抽象消息的 name 属性对应。 |

不需要为所有 输入和输出 元素指定 name 属性；WSDL 根据操作的名称提供默认的命名方案。如果操作中只使用一个元素，则元素名称默认为操作的名称。如果使用了 输入和输出 元素，则元素名称默认为操作的名称，并分别附加 Request 或 Response。

返回值

由于 operation 元素是操作期间传递的数据的一个抽象定义，因此 WSDL 不提供为操作指定的返回值。如果方法返回值，它将映射到 output 元素中作为该消息的最后一部分。

**EXAMPLE**

例如：您可能有一个类似于 例 4.1 “personalInfo 查找接口” 中显示的接口。

**例 4.1. personalInfo 查找接口**

```
interface personalInfoLookup
{
  personalInfo lookup(in int empID)
  raises(idNotFound);
}
```

这个接口可以映射到 例 4.2 “personalInfo 查找端口类型” 中的端口类型。

**例 4.2. personalInfo 查找端口类型**

```
<message name="personalLookupRequest">
  <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalLookupResponse"/>
<message/>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound"/>
<message/>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="tns:personalLookupRequest"/>
    <output name="return" message="tns:personalLookupResponse"/>
    <fault name="exception" message="tns:idNotFoundException"/>
  </operation>
</portType>
```

# 部分 II. WEB 服务绑定

这部分描述了如何将 Apache CXF 绑定添加到 WSDL 文档。

# 第 5 章 了解 WSDL 中的绑定

## 摘要

绑定将用于定义服务的逻辑消息映射到可由端点传输和接收的相对有效负载格式。

## 概述

绑定在服务使用的逻辑消息之间提供桥接，以免除端点在物理世界中使用的数据格式。它们描述逻辑消息如何映射到端点在有线中使用的有效负载格式。它位于详情（如参数顺序、聚合数据类型和返回值）的绑定中。例如，可以通过绑定对消息的部分重新排序，以反映 RPC 调用所需的顺序。根据绑定类型，您还可以识别哪个消息部分（若有）代表方法的返回类型。

## 端口类型和绑定

端口类型和绑定直接相关。端口类型是两个逻辑服务之间的一组交互抽象定义。绑定是关于如何在物理世界中实例化用于实施逻辑服务的消息。然后，每个绑定都与一组网络详情关联，完成一个端点的定义，该端点公开由端口类型定义的逻辑服务。

为确保端点仅定义单个服务，WSDL 要求绑定只能代表单个端口类型。例如，如果您有一个与两种端口类型的合同，您无法编写一个绑定，将其这两个绑定映射到聚合数据格式。您需要两个绑定。

但是，WSDL 允许将端口类型映射到多个绑定。例如，如果您的合同具有单一端口类型，您可以将其映射到两个或多个绑定。每个绑定可能会改变消息的部分映射方式，或者可以为消息指定完全不同的有效负载格式。

## WSDL 元素

绑定使用 WSDL 绑定 元素在合同中定义。binding 元素包含诸如 name 的属性，它指定了对 PortType 的引用的绑定和类型的唯一名称。此属性的值用于将绑定与端点关联，如 第 4 章 *定义逻辑接口* 所述。

实际映射在 binding 元素的子项中定义。这些元素因您决定使用的有效负载格式类型而有所不同。以下章节将讨论不同的有效负载格式和用于指定映射的元素。

## 添加到合同

Apache CXF 提供命令行工具，可为预定义的服务接口生成绑定。

这些工具会将正确的元素添加到您的合同中。但是，我们建议您了解不同类型的绑定的工作方式。

您还可以使用任何文本编辑器将绑定添加到合同。在手动编辑合同时，您需要确保合同有效。

### 支持的绑定

Apache CXF 支持以下绑定：

- SOAP 1.1
- SOAP 1.2
- CORBA
- 纯 XML

# 第 6 章 使用 SOAP 1.1 消息

**摘要**

Apache CXF 提供了一个工具来生成 SOAP 1.1 绑定，该绑定不使用任何 SOAP 标头。但是，您可以使用任何文本或 XML 编辑器将 SOAP 标头添加到绑定中。

## 6.1. 添加 SOAP 1.1 绑定

### 使用 wsdl2soap

要使用 wsdl2soap 生成一个 SOAP 1.1 绑定，请使用以下命令： **wsdl2soap-i*port-type-name*-b*binding-name*-d*output-directory*-o*output-file*-n*soap-body-namespace*-style (document/rpc)-use (literal/encoded)-v-verbose-quiet*wsdlurl***

**注意**

要使用 wsdl2soap，您需要下载 Apache CXF 发行版。

该命令有以下选项：

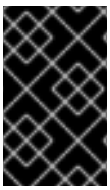
| 选项 | 解释 |
| --- | --- |
| **-i** *port-type-name* | 指定生成绑定的 **portType** 元素。 |
| *wsdlurl* | 包含 **portType** 元素定义的 WSDL 文件的路径和名称。 |

该工具有以下可选参数：

| 选项 | 解释 |
| --- | --- |
| **-b** *binding-name* | 指定生成的 SOAP 绑定的名称。 |
| **-d** *output-directory* | 指定要放置生成的 WSDL 文件的目录。 |
| **-o** *output-file* | 指定生成的 WSDL 文件的名称。 |

| 选项 | 解释 |
| --- | --- |
| **-n** *soap-body-namespace* | 指定 RPC 样式时的 SOAP 正文命名空间。 |
| **-style** (document/rpc) | 指定要在 SOAP 绑定中使用的编码样式（文档或 RPC）。默认为 document。 |
| **-use** （字面/编码） | 指定在 SOAP 绑定中使用的绑定使用（编码或字面）。默认值为 literal。 |
| **-v** | 显示工具的版本号。 |
| **-verbose** | 在代码生成过程中显示注释。 |
| **-quiet** | 在代码生成过程中抑制注释。 |

**需要 -i*port-type-name* 和 *wsdlurl* 参数。如果指定了 -style rpc 参数，则还需要 -n*soap-body-namspace* 参数。所有其他参数都是可选的，可按任何顺序列出。**



**重要**

**wsdl2soap 不支持生成 文档/编码的 SOAP 绑定。**

**Example**

**如果您的系统有一个需要顺序的接口，并提供单个操作来处理其在 WSDL 片段中定义的顺序，类似于 例 6.1 “排序系统接口” 中显示的顺序。**

**例 6.1. 排序系统接口**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
```

```
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

为 **orderWidget** 生成的 **SOAP** 绑定显示在 **例 6.2 “用于 orderWidget 的SOAP 1.1 Binding” 中。**

**例 6.2. 用于 orderWidget 的SOAP 1.1 Binding**

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
```

**此绑定指定消息使用 文档/字面 消息样式发送。**

## 6.2. 在 SOAP 1.1 BINDING 中添加 SOAP 标头

**概述**

**SOAP 标头通过将 soap:header 元素添加到默认的 SOAP 1.1 绑定中定义。soap:header 元素是绑定的 输入、输出和 fault 元素的可选子级。SOAP 标头成为父消息的一部分。SOAP 标头通过指定消息和消息部分来定义。每个 SOAP 标头只能包含一个消息部分，但您可以根据需要插入任意数量的 SOAP 标头。**

语法

例 6.3 “SOAP 标头语法” 中显示定义 SOAP 标头的语法。soap:header 的 message 属性是生成了部分插入到标头中的消息的合格名称。part 属性是插入到 SOAP 标头中的消息部分的名称。由于 SOAP 标头始终是文档样式的，因此必须使用元素定义插入到 SOAP 标头中的 WSDL 消息部分。message 和 part 属性一起完全描述要插入到 SOAP 标头中的数据。

**例 6.3. SOAP 标头语法**

```
<binding name="headwig">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap:operation soapAction="" style="document"/>
      <input name="grain">
        <soap:body ... />
        <soap:header message="QName" part="partName"/>
      </input>
...
</binding>
```

以及强制 消息和 部分 属性，soap:header 也支持 命名空间、use，以及 encodingStyle 属性。这些属性的功能与 soap:body 的 soap:header 相同。

在正文和标头之间分割消息

插入到 SOAP 标头中的消息部分可以是合同中的任何有效消息部分。它甚至也可以是来自用作 SOAP 正文的父消息的一部分。由于不太可能在同一消息中发送信息两次，因此 SOAP 绑定提供了一种指定插入到 SOAP 正文中的消息部分的方法。

soap:body 元素有一个可选属性 parts，它采用以空格分隔的部分名称列表。当定义 部分 时，只有列出的消息部分会插入到 SOAP 正文中。然后，您可以将剩余的部分插入到 SOAP 标头中。

注意

当您使用父消息的部分定义 SOAP 标头时，Apache CXF 会自动填写 SOAP 标头。

Example

例 6.4 “使用 SOAP 标头的 SOAP 1.1 Binding” 显示 例 6.1 “排序系统接口” 中显示的 orderWidgets 服务的修改版本。此版本已被修改，每个顺序都有一个 xsd:base64binary 值，放置在请求和响应的

**SOAP 标头中。SOAP 标头定义为 widgetKey 消息中的 keyVal 部分。在这种情况下，您负责将 SOAP 标头添加到应用程序逻辑中，因为它不是输入或输出消息的一部分。**

**例 6.4. 使用 SOAP 标头的 SOAP 1.1 Binding**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
```

```
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
    </operation>
  </binding>
  ...
</definitions>
```

**您还可以修改 例 6.4 “使用 SOAP 标头的 SOAP 1.1 Binding”，以便标头值是输入和输出信息的一部分。**

# 第 7 章 使用 SOAP 1.2 消息

**摘要**

**Apache CXF 提供生成 SOAP 1.2 绑定的工具，不使用任何 SOAP 标头。您可以使用任何文本或 XML 编辑器将 SOAP 标头添加到绑定中。**

## 7.1. 将 SOAP 1.2 绑定添加到 WSDL 文档

### 使用 wsdl2soap

**注意**

**要使用 wsdl2soap，您需要下载 Apache CXF 发行版。**

**要使用 wsdl2soap 生成一个 SOAP 1.2 绑定，请使用以下命令： wsdl2soap-i*port-type-name*-b*binding-name*-soap12-d*output-directory*-o*output-file*-n*soap-body-namespace*-style (document/rpc)-use (literal/encoded)-v-verbose-*wsdlurl* 工具有以下参数：**

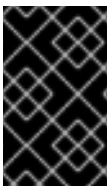
| 选项 | 解释 |
| --- | --- |
| **-i** *port-type-name* | 指定生成绑定的 **portType** 元素。 |
| **-soap12** | 指定生成的绑定使用 SOAP 1.2。 |
| *wsdlurl* | 包含 **portType** 元素定义的 WSDL 文件的路径和名称。 |

**该工具有以下可选参数：**

| 选项 | 解释 |
| --- | --- |
| **-b** *binding-name* | 指定生成的 SOAP 绑定的名称。 |
| **-soap12** | 指定生成的绑定将使用 SOAP 1.2。 |
| **-d** *output-directory* | 指定要放置生成的 WSDL 文件的目录。 |
| **-o** *output-file* | 指定生成的 WSDL 文件的名称。 |

| 选项 | 解释 |
| --- | --- |
| **-n** *soap-body-namespace* | 指定 RPC 样式时的 SOAP 正文命名空间。 |
| **-style** (document/rpc) | 指定要在 SOAP 绑定中使用的编码样式（文档或 RPC）。默认为 document。 |
| **-use** （字面/编码） | 指定在 SOAP 绑定中使用的绑定使用（编码或字面）。默认值为 literal。 |
| **-v** | 显示工具的版本号。 |
| **-verbose** | 在代码生成过程中显示注释。 |
| **-quiet** | 在代码生成过程中抑制注释。 |

**需要 -i *port-type-name* 和 *wsdlurl* 参数。如果指定了 -style rpc 参数，则还需要 -n *soap-body-namspace* 参数。所有其他参数都是可选的，可以按任何顺序列出。**



**重要**

**wsdl2soap 不支持生成 文档/编码的 SOAP 1.2 绑定。**

**Example**

**如果您的系统有一个需要顺序的接口，并提供单个操作来处理其在 WSDL 片段中定义的顺序，类似于 例 7.1 “排序系统接口” 中显示的顺序。**

**例 7.1. 排序系统接口**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
```

```
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

为 **orderWidget** 生成的 **SOAP** 绑定显示在 **例 7.2 “SOAP 1.2 Binding for orderWidgets” 中。**

**例 7.2. SOAP 1.2 Binding for orderWidgets**

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
      </input>
      <output name="bill">
        <wsoap12:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
```

**此绑定指定消息使用 文档/字面 消息样式发送。**

## 7.2. 将标头添加到 SOAP 1.2 消息

### 概述

**SOAP 消息标头通过在 SOAP 1.2 消息中添加 soap12:header 元素来定义。soap12:header 元素是绑定的 输入、输出和 fault 元素的可选子级。SOAP 标头成为父消息的一部分。SOAP 标头通过指定消息和消息部分来定义。每个 SOAP 标头只能包含一个消息部分，但您可以根据需要插入多个标头。**

语法

例 7.3 “SOAP 标头语法” 中显示定义 SOAP 标头的语法。

**例 7.3. SOAP 标头语法**

```
<binding name="headwig">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap12:operation soapAction="" style="documment"/>
      <input name="grain">
        <soap12:body ... />
        <soap12:header message="QName" part="partName"
                       use="literal|encoded"
                       encodingStyle="encodingURI"
                       namespace="namespaceURI" />
      </input>
...
</binding>
```

soap12:header 元素的属性在 表 7.1 “soap12:header 属性” 中进行了描述。

**表 7.1. soap12:header 属性**

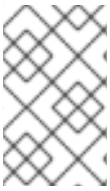
| 属性 | 描述 |
| --- | --- |
| **message** | 一个必需属性，用于指定在标头中插入部分的合格消息名称。 |
| **part** | 指定插入到 SOAP 标头中的消息部分名称的必要属性。 |
| **使用** | 指定消息部分是否使用编码规则编码。如果设置为 **编码** 消息部分，则使用由 **encodingStyle** 属性的值指定的编码规则编码。如果设置为 **literal**，则消息部分由引用的 schema 类型定义。 |
| **encodingStyle** | 指定用于构建消息的编码规则。 |
| **namespace** | 定义要分配给 header 元素的命名空间，使用 **use="encoded"** 序列化。 |

在正文和标头之间分割消息

插入到 SOAP 标头中的消息部分可以是合同中的任何有效消息部分。它甚至也可以是来自用作 SOAP 正文的父消息的一部分。由于您不太可能在同一消息中发送信息两次，因此 SOAP 1.2 绑定提供了一种指

定插入到 SOAP 正文中的消息部分的方法。

soap12:body 元素有一个可选属性 parts，它采用以空格分隔的部分名称列表。当定义 部分 时，只有列出的消息部分会插入到 SOAP 1.2 消息的正文中。然后，您可以将剩余的部分插入到消息的标头中。



**注意**

当您使用父消息的部分定义 SOAP 标头时，Apache CXF 会自动填写 SOAP 标头。

**Example**

例 7.4 “使用 SOAP 标头的 SOAP 1.2 Binding” 显示 例 7.1 “排序系统接口” 中显示的 orderWidgets 服务的修改版本。此版本会被修改，以便每个顺序都有一个 xsd:base64binary 值，放置在请求和响应标头中。标头被定义为 widgetKey 消息中的 keyVal 部分。在这种情况下，您负责添加应用程序逻辑来创建标头，因为它不是输入或输出消息的一部分。

**例 7.4. 使用 SOAP 标头的 SOAP 1.2 Binding**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
```

```
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

**您可以修改 例 7.4 “使用 SOAP 标头的 SOAP 1.2 Binding”，以便标头值是输入和输出信息的一部分，如 例 7.5 “SOAP 1.2 Binding for orderWidgets with a SOAP Headers” 所示。在这种情况下，keyVal 是输入和输出消息的一部分。在 soap12:body 元素中，part 属性指定 keyVal 不应插入到正文中。但是，它将插入到标头中。**

**例 7.5. SOAP 1.2 Binding for orderWidgets with a SOAP Headers**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
```

```
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal" parts="numOrdered"/>
        <soap12:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal" parts="bill"/>
        <soap12:header message="tns:widgetOrderBill" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

# 第 8 章 使用带有附件的 SOAP 发送二进制数据

## 摘要

SOAP attachments 提供发送二进制数据的机制，作为 SOAP 消息的一部分。将 SOAP 与附件一起使用需要将 SOAP 消息定义为 MIME 多部分消息。

## 概述

SOAP 消息通常不会传输二进制数据。但是，W3C SOAP 1.1 规范允许使用 MIME 多部分/相关消息来发送 SOAP 消息中的二进制数据。使用带有附件的 SOAP 称为这种技术。SOAP attachments 在 W3C 的 SOAP 消息中定义，并用附件注释。

## NAMESPACE

用于定义 MIME 多部分/相关消息的 WSDL 扩展在命名空间 http://schemas.xmlsoap.org/wsdl/mime/ 中定义。

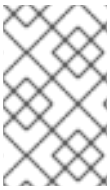
在随后的讨论中，假设此命名空间的前缀为 mime。设置此值的 WSDL 定义 元素中的条目显示在 例 8.1 “合同中的 MIME 命名空间规格” 中。

**例 8.1. 合同中的 MIME 命名空间规格**

```
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

## 更改消息绑定

在默认的 SOAP 绑定中，输入、输出和 fault 元素的第一个子元素是描述用于代表数据的 SOAP 消息正文的 soap:body 元素。将 SOAP 与附件搭配使用时，soap:body 元素将替换为 mime:multipartRelated 元素。



**注意**

WSDL 不支持使用 mime:multipartRelated 用于 故障 消息。

mime:multipartRelated 元素告知 Apache CXF 消息正文是一个多部分消息，可能包含二进制数据。元素的内容定义消息及其内容的部分。MIME：多部分 元素包含一个或多个 mime:part 元素，用于描述消

息的各个部分。

第一个 mime:part 元素必须包含通常出现在默认 SOAP 绑定中的 soap:body 元素。其余的 mime:part 元素定义消息中发送的附件。

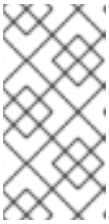
### 描述 MIME 多部分消息

MIME 多部分消息使用 mime:multipartRelated 元素来描述，其中包含多个 mime:part 元素。要完全描述 MIME 多部分信息，您必须执行以下操作：

1. 在您要作为 MIME 多部分消息发送的 输入或输出 消息中，添加一个 mime:mulipartRelated 元素作为括起消息的第一个子元素。

2. 将 mime:part 子元素添加到 mime:multipartRelated 元素，并将其 name 属性设置为唯一字符串。

3. 添加 soap:body 元素作为 mime:part 元素的子项，并相应地设置其属性。



**注意**

如果合同具有默认的 SOAP 绑定，您可以将 soap:body 元素从默认绑定中的对应消息复制到 MIME 多部分消息。

4. 将另一个 mime:part 子元素添加到 mime:multipartReleated 元素，并将其 name 属性设置为唯一字符串。

5. 将 mime:content 子元素添加到 mime:part 元素，以描述消息此部分的内容。

要完全描述 MIME 消息部分的内容，mime:content 元素具有以下属性：

**表 8.1. MIME:content 属性**

| 属性 | 描述 + |
| --- | --- |

| 属性 | 描述 + |
| --- | --- |
| **part** | 指定来自父消息定义的 WSDL 消息部分的名称，该部分用作要在有线上的 MIME 多部分消息的内容。<br><br>+ |
| **type** | 此消息部分中的数据的 MIME 类型部分。MIME 类型定义为类型和子类型，使用语法类型/*子类型*。<br><br>+<br><br>*有很多预定义的 MIME 类型，如* ***image/jpeg*** *和* ***text/plain****。MIME 类型由互联网编号分配机构 (IANA) 维护，并在 多用途互联网邮件扩展(MIME) 中详细介绍：互联网消息 Bodies 和 Multipurpose Internet 邮件扩展(MIME) 第 2 部分。*<br><br>+ |

6. **对于每个额外的 MIME 部分，重复步骤 [i303819] 和 [i303821]。**

**EXAMPLE**

**例 8.2 “使用带有附件的 SOAP 的合同” 显示一个 WSDL 片段，用于定义以 JPEG 格式存储 X-rays 的服务。镜像数据 xRay 存储为 xsd:base64binary，并打包成 MIME 多部分 imageData。输入消息的剩余两个部分是 patientName 和 patientNumber，作为 SOAP 正文的一部分在 MIME 多部分镜像的第一个部分中发送。**

**例 8.2. 使用带有附件的 SOAP 的合同**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="storRequest">
    <part name="patientName" type="xsd:string"/>
    <part name="patientNumber" type="xsd:int"/>
    <part name="xRay" type="xsd:base64Binary"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>

  <binding name="xRayStorageBinding" type="tns:xRayStorage">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="store">
      <soap:operation soapAction="" style="document"/>
      <input name="storRequest">
        <mime:multipartRelated>
          <mime:part name="bodyPart">
            <soap:body use="literal"/>
          </mime:part>
          <mime:part name="imageData">
            <mime:content part="xRay" type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </input>
      <output name="storResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="xRayStorageService">
    <port binding="tns:xRayStorageBinding" name="xRayStoragePort">
      <soap:address location="http://localhost:9000"/>
    </port>
  </service>
</definitions>
```

# 第 9 章 使用 SOAP MTOM 发送二进制数据

**摘要**

SOAP 消息传输优化机制(MTOM)将 SOAP 替换为作为 XML 消息一部分发送二进制数据的机制。使用带有 Apache CXF 的 MTOM 需要将正确的模式类型添加到服务的合同中，并启用 MTOM 优化。

## 9.1. MTOM 概述

SOAP 消息传输优化机制(MTOM)指定作为 SOAP 消息一部分发送二进制数据的最佳方法。与带有附件的 SOAP 不同，MTOM 需要使用 XML 二进制优化打包(XOP)软件包来传输二进制数据。使用 MTOM 发送二进制数据不需要您完全定义 MIME 多部分/Related 消息，作为 SOAP 绑定的一部分。但是，它会要求您执行以下操作：

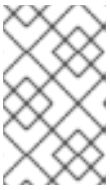
1. 注解 您要作为附件发送的数据。

   您可以注解 WSDL 或实施数据的 Java 类。

2. 启用 运行时的 MTOM 支持。

   这可以以编程方式或通过配置完成。

3. 为 作为附件 传递的数据开发数据处理程序。



**注意**

开发 数据处理程序已超出本书的讨论范围。

## 9.2. 注解数据类型以使用 MTOM

**概述**

在 WSDL 中，定义用于传递二进制数据（如镜像文件或声音文件）的数据类型时，您可以将数据的元素定义为 xsd:base64Binary。默认情况下，类型为 xsd:base64Binary 的任何元素都会生成 byte[]，该字节[] 可以使用 MTOM 序列化。但是，代码生成器的默认行为不会完全利用序列化功能。

为了充分利用 MTOM，您必须将注释添加到服务的 WSDL 文档或实施二进制数据结构的 JAXB 类。向 WSDL 文档添加注释会强制代码生成器为二进制数据生成流数据处理程序。注解 JAXB 类涉及指定正确的内容类型，并且可能还涉及更改包含二进制数据的字段的类型规格。

### WSDL 首先

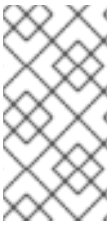
例 9.1 “MTOM 信息” 显示 Web 服务的 WSDL 文档，它使用一条消息，其中包含一个字符串字段、一个整数字段和二进制字段。二进制字段旨在处理大型镜像文件，因此不适合将其作为普通 SOAP 消息的一部分进行发送。

**例 9.1. MTOM 信息**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:xsd1="http://mediStor.org/types/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>
    <schema targetNamespace="http://mediStor.org/types/"
          xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary" />
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
    </schema>
  </types>

  <message name="storRequest">
    <part name="record" element="xsd1:xRay"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>

  <binding name="xRayStorageSOAPBinding" type="tns:xRayStorage">
    <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="store">
```

```
      <soap12:operation soapAction="" style="document"/>
      <input name="storRequest">
        <soap12:body use="literal"/>
      </input>
      <output name="storResponse">
        <soap12:body use="literal"/>
      </output>
    </operation>
  </binding>
  ...
</definitions>
```

如果要使用 MTOM 将消息的二进制部分作为优化的附加发送，您必须将 xmime:expectedContentTypes 属性添加到包含二进制数据的元素中。此属性在 http://www.w3.org/2005/05/xmlmime 命名空间中定义，并指定元素应包含的 MIME 类型。您可以指定以逗号分隔的 MIME 类型列表。此属性的设置会改变代码生成器如何为数据创建 JAXB 类。对于大多数 MIME 类型，代码生成器会创建一个 DataHandler。有些 MIME 类型（如镜像）定义了映射。



**注意**

MIME 类型由互联网编号分配机构(IANA)维护，在 多用途互联网邮件扩展(MIME)第 1 部分所述：互联网消息 Bodies 和 多用途互联网邮件扩展(MIME)第 2 部分。

对于大多数使用，您可以指定 application/octet-stream。

例 9.2 “MTOM 的二进制数据” 演示了如何从 例 9.1 “MTOM 信息” 修改 xRayType 以使用 MTOM。

**例 9.2. MTOM 的二进制数据**

```
...
  <types>
    <schema targetNamespace="http://mediStor.org/types/"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary"
                xmime:expectedContentTypes="application/octet-stream"/>
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
```

```
    </schema>
  </types>
...
```

为 xRayType 生成的 JAXB 类不再包含 byte[]。相反，代码生成器会看到 xmime:expectedContentTypes 属性，并为 imageData 字段生成 DataHandler。

**注意**

您不需要将 binding 元素更改为使用 MTOM。运行时在发送数据时进行适当的更改。

**Java 首先**

如果您正在进行 Java 首次开发，您可以通过执行以下操作使 JAXB 类 MTOM 就绪：

1. 确保包含二进制数据的字段是 DataHandler。
2. 将 @XmlMimeType （） 注释添加到包含您要作为 MTOM 附加的数据的字段。

例 9.3 “JAXB Class for MTOM” 显示使用 MTOM 标注的 JAXB 类。

**例 9.3. JAXB Class for MTOM**

```
@XmlType
public class XRayType {
    protected String patientName;
    protected int patientNumber;
    @XmlMimeType("application/octet-stream")
    protected DataHandler imageData;
  ...
}
```

## 9.3. 启用 MTOM

默认情况下，Apache CXF 运行时不支持 MTOM。它将所有二进制数据作为普通 SOAP 消息的一部分或作为未优化的附件发送。您可以以编程方式激活 MTOM 支持或使用配置。

### 9.3.1. 使用 JAX-WS API

**概述**

服务提供商和消费者都必须启用 MTOM 优化。JAX-WS API 为每种端点提供不同的机制。

**服务供应商**

如果您使用 JAX-WS API 发布您的服务提供商，您可以启用运行时的 MTOM 支持，如下所示：

1. 访问您发布服务的 Endpoint 对象。

   访问 Endpoint 对象的最简单方法是发布端点。如需更多信息，请参阅 第 31 章 *发布服务*。

2. 使用其 getBinding （） 方法从 Endpoint 获取 SOAP 绑定，如 例 9.4 “从端点获取 SOAP 绑定” 所示。

   **例 9.4. 从端点获取 SOAP 绑定**

   ```
   // Endpoint ep is declared previously
   SOAPBinding binding = (SOAPBinding)ep.getBinding();
   ```

   您必须将返回的绑定对象转换为 SOAPBinding 对象来访问 MTOM 属性。

3. 使用绑定的 setMTOMEnabled （） 方法将绑定的 MTOM enabled 属性设置为 true，如 例 9.5 “设置服务提供商的 MTOM Enabled Property” 所示。

   **例 9.5. 设置服务提供商的 MTOM Enabled Property**

   ```
   binding.setMTOMEnabled(true);
   ```

**消费者**

要 MTOM 启用 JAX-WS 使用者，您必须执行以下操作：

1. 将消费者的代理转换为 BindingProvider 对象。

有关获取消费者代理的详情，请参考 第 25 章 *开发没有 WSDL 合同的消费者* 或 第 28 章 *从 WSDL 合同开发一个消费者*。

2. 使用其 getBinding （） 方法从 BindingProvider 获取 SOAP 绑定，如 例 9.6 “从 BindingProvider获取 SOAP 绑定” 所示。

**例 9.6. 从 BindingProvider获取 SOAP 绑定**

```
// BindingProvider bp declared previously
SOAPBinding binding = (SOAPBinding)bp.getBinding();
```

3. 使用绑定的 setMTOMEnabled （） 方法将 bindings MTOM enabled 属性设置为 true，如 例 9.7 “设置 Consumer 的 MTOM Enabled Property” 所示。

**例 9.7. 设置 Consumer 的 MTOM Enabled Property**

```
binding.setMTOMEnabled(true);
```

### 9.3.2. 使用配置

#### 概述

如果您使用 XML 发布服务，如部署到容器时，您可以在端点的 配置文件中启用端点的 MTOM 支持。有关配置端点的更多信息，请参阅 第 IV 部分 “配置 Web 服务端点”。

#### 流程

MTOM 属性在端点的 jaxws:endpoint 元素内设置。要启用 MTOM，请执行以下操作：

1. 将 jaxws:property 子元素添加到端点的 jaxws:endpoint 元素。

2. 向 jaxws:property 元素中 添加一个条目 子元素。

3. **将 entry 元素的 key 属性设置为 mtom-enabled。**

4. **将 entry 元素的 value 属性设置为 true。**

**Example**

例 9.8 “启用 MTOM 的配置” 显示启用 **MTOM** 的端点。

**例 9.8. 启用 MTOM 的配置**

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                          http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">

  <jaxws:endpoint id="xRayStorage"
                  implementor="demo.spring.xRayStorImpl"
                  address="http://localhost/xRayStorage">
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:endpoint>
</beans>
```

# 第 10 章 使用 XML 文档

## 摘要

纯 XML 有效负载格式提供了 SOAP 绑定的替代方法，方法是允许服务使用直接 XML 文档来交换数据，而无需使用 SOAP 信封的开销。

## XML 绑定命名空间

用于描述 XML 格式的绑定的扩展在命名空间 http://cxf.apache.org/bindings/xformat 中定义。Apache CXF 工具使用前缀 xformat 来代表 XML 绑定扩展。在您的合同中添加以下行：

```
xmlns:xformat="http://cxf.apache.org/bindings/xformat"
```

## 手动编辑

要将接口映射到纯 XML 有效负载格式，请执行以下操作：

1. 添加命名空间声明，使其包含定义 XML 绑定的扩展。请参阅 “XML 绑定命名空间”一节。
2. 向您的合同添加标准 WSDL 绑定 元素来保存 XML 绑定，为绑定指定一个唯一的名称，并指定代表正在绑定接口的 WSDL portType 元素 的名称。
3. 将 xformat:binding 子元素添加到 binding 元素，以标识消息正在作为纯 XML 文档处理，而无需 SOAP envelopes。
4. 另外，还可将 xformat:binding 元素的 rootNode 属性设置为有效的 QName。有关 rootNode 属性的影响的更多信息，请参阅 “wire 上的 XML 消息”一节。
5. 对于绑定接口中定义的每个操作，请添加标准 WSDL 操作 元素来存放操作消息的绑定信息。
6. 对于添加到绑定的每个操作，添加 输入、输出和错误 子元素，以表示操作使用的消息。

   这些元素与逻辑操作接口定义中定义的消息对应。

7. （可选）添加带有有效 rootNode 属性的 xformat:body 元素到添加的 输入、output 和 fault 元素，以覆盖绑定级别上的 rootNode 设置的值。

**注意**

如果您的任何消息没有部分，例如返回 void 的操作的输出消息，您必须为消息设置 rootNode 属性，以确保在 wire 上写入的消息有效，但空 XML 文档。

**WIRE 上的 XML 消息**

当您指定接口的消息将作为 XML 文档传递时，如果没有 SOAP 信封，则必须小心谨慎，以确保您的消息在线上写入有效的 XML 文档时。您还需要确保接收 XML 文档的非 Apache CXF 参与者了解 Apache CXF 生成的消息。

解决这两个问题的简单方法是在全局 xformat:binding 元素或单独的消息的 xformat:body 元素上使用可选的 rootNode 属性。rootNode 属性指定作为 Apache CXF 生成的 XML 文档的根节点元素的 QName。如果没有设置 rootNode 属性，当使用 rpc 风格消息时，Apache CXF 使用消息部分的 root 元素作为 root 元素，或使用消息部分名称作为 root 元素的一个元素。

例如，如果 rootNode 属性没有设置 例 10.1 “有效的 XML 绑定消息” 中定义的消息，则会生成带有 root 元素 lineNumber 的 XML 文档。

**例 10.1. 有效的 XML 绑定消息**

```
<type ... >
  ...
  <element name="operatorID" type="xsd:int"/>
  ...
</types>
<message name="operator">
  <part name="lineNumber" element="ns1:operatorID"/>
</message>
```

对于带有一个部分的消息，即使未设置 rootNode 属性，Apache CXF 始终也会生成有效的 XML 文档。但是，例 10.2 “无效的 XML 绑定消息” 的消息会生成无效的 XML 文档。

**例 10.2. 无效的 XML 绑定消息**

```
<types>
  ...
  <element name="pairName" type="xsd:string"/>
```

```
  <element name="entryNum" type="xsd:int"/>
  ...
</types>

<message name="matildas">
  <part name="dancing" element="ns1:pairName"/>
  <part name="number" element="ns1:entryNum"/>
</message>
```

如果没有在 XML 绑定中指定的 rootNode 属性，Apache CXF 将为 例 10.3 “无效的 XML 文档” 中定义的消息生成与 例 10.2 “无效的 XML 绑定消息” 类似的 XML 文档。生成的 XML 文档无效，因为它有两个根元素： pairName 和 entryNum。

**例 10.3. 无效的 XML 文档**

```
<pairName>
  Fred&Linda
</pairName>
<entryNum>
  123
</entryNum>
```

如果您设置了 rootNode 属性，如 例 10.4 “使用 rootNode 设置的 XML Binding” Apache CXF 所示，将嵌套指定根元素中的元素。在本例中，整个绑定定义了 rootNode 属性，并指定 root 元素将命名为 entrants。

**例 10.4. 使用 rootNode 设置的 XML Binding**

```
<portType name="danceParty">
  <operation name="register">
    <input message="tns:matildas" name="contestant"/>
  </operation>
</portType>

<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered"/>
</binding>
```

从输入信息生成的 XML 文档与 例 10.5 “使用 rootNode 属性生成的 XML 文档” 类似。请注意，XML 文档现在只有一个 root 元素。

**例 10.5. 使用 rootNode 属性生成的 XML 文档**

```
<entrants>
  <pairName>
    Fred&Linda
  <entryNum>
    123
  </entryNum>
</entrants>
```

### 覆盖绑定的 ROOTNODE 属性设置

您还可以使用消息绑定中的 **xformat:body** 元素来为各个消息设置 **rootNode** 属性，或覆盖特定消息的全局设置。例如：如果您希望 **例 10.4 “使用 rootNode 设置的 XML Binding”** 中定义的输出信息具有与输入消息不同的 **root** 元素，您可以覆盖 **例 10.6 “使用 xformat:body”** 所示的绑定根元素。

**例 10.6. 使用 xformat:body**

```
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered">
      <xformat:body rootNode="entryStatus" />
    </output>
  </operation>
</binding>
```

# 部分 III. WEB 服务传输

这部分描述了如何将 Apache CXF 传输添加到 WSDL 文档中。

# 第 11 章 了解 WSDL 中如何定义端点

**摘要**

端点代表一个实例化服务。它们通过组合绑定和用于公开端点的网络详情来定义。

**概述**

端点可以被认为是服务的物理清单。它组合了绑定，指定服务使用的逻辑数据的物理表示，以及一组定义供其他端点联系的物理连接详情的联网详情。

**注意**

CXF 供应商是 CXF 用户的服务器，对应于客户端。如果您使用 CXF (camel-cxf)组件作为路由中的起始端点，则端点是 Camel 使用者和 CXF 供应商。如果您使用 Camel CXF 组件，作为路由中的结束端点，则端点是 Camel 生成者和 CXF 使用者。

**端点和服务**

与绑定只能映射单个接口的方式相同，端点只能映射到单个服务。但是，服务可以被任意数量的端点清单。例如，您可以定义一个由四个不同端点组成的票据销售服务。但是，您无法有一个端点来清单一个端点，该端点同时托管一个 ticket 销售服务和一个小部件销售服务。

**WSDL 元素**

端点通过 WSDL 服务 元素和 WSDL 端口 元素的组合在合同中定义。service 元素是相关端口元素 的集合。port 元素定义实际端点。

WSDL 服务 元素具有单个属性，名为，它指定了一个唯一的名称。service 元素用作相关端口元素集合的父 元素。WSDL 不会说明如何关联 端口 元素。您可以使用您看到适合的任何方式关联 端口 元素。

WSDL 端口 元素具有 绑定 属性，用于指定端点使用的绑定，是对 wsdl:binding 元素的引用。它还包括 name 属性，这是必需属性，用于在所有端口中提供唯一名称。port 元素是元素的父元素，用于指定端点使用的实际传输详情。以下部分讨论用于指定传输详情的元素。

**在合同中添加端点**

Apache CXF 提供命令行工具，可为预定义的服务接口和绑定组合生成端点。

这些工具会将正确的元素添加到您的合同中。但是，我们建议您了解定义端点工作时使用的不同传输。

您还可以使用任何文本编辑器将端点添加到合同。当您手动编辑合同时，您需要确保合同有效。

### 支持的传输

端点定义是使用为 Apache CXF 支持的每个传输定义的扩展构建。这包括以下传输：

- HTTP
- CORBA
- Java 消息传递服务

# 第 12 章 使用 HTTP

**摘要**

HTTP 是 Web 的底层传输。它为端点之间的通信提供标准化、强大且灵活的平台。由于这些因素，它是大多数 WSjpeg 规范假定的传输，是 RESTful 架构不可或缺的。

## 12.1. 添加基本 HTTP 端点

备用 HTTP 运行时

Apache CXF 支持以下替代 HTTP 运行时实现：

- Undertow，在 第 12.4 节 “配置 Undertow 运行时” 中进行了详细介绍。
- Netty，在 第 12.5 节 “配置 Netty 运行时” 中进行了详细介绍。

**Netty HTTP URL**

通常，HTTP 端点使用哪些 HTTP 运行时包含在类路径上(Undertow 或 Netty)。但是，如果 classpath 上包含 Undertow 运行时和 Netty 运行时，则需要在想要使用 Netty 运行时时显式指定 Undertow 运行时，因为默认情况下将使用 Undertow 运行时。

如果 classpath 上有多个 HTTP 运行时，您可以通过指定端点 URL 来选择 Undertow 运行时，使其具有以下格式：

```
netty://http://RestOfURL
```

有效负载类型

根据您使用的有效负载格式，可以通过三种方式指定 HTTP 端点地址。

- SOAP 1.1 使用标准化 soap:address 元素。

- SOAP 1.2 使用 soap12:address 元素。
- 所有其他有效负载格式都使用 http:address 元素。

注意

在 Camel 2.16.0 发行版本中，Apache Camel CXF Payload 支持开箱即用的流缓存。

### SOAP 1.1

当您通过 HTTP 发送 SOAP 1.1 消息时，必须使用 SOAP 1.1 address 元素来指定端点的地址。它有一个属性 location，它将端点的地址指定为 URL。SOAP 1.1 地址 元素在命名空间 http://schemas.xmlsoap.org/wsdl/soap/ 中定义。

例 12.1 “SOAP 1.1 端口元素” 显示用于通过 HTTP 发送 SOAP 1.1 消息 的端口 元素。

例 12.1. SOAP 1.1 端口元素

```
<definitions ...
            xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
  ...
  <service name="SOAP11Service">
    <port binding="SOAP11Binding" name="SOAP11Port">
      <soap:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

### SOAP 1.2

当您通过 HTTP 发送 SOAP 1.2 消息时，必须使用 SOAP 1.2 address 元素来指定端点的地址。它有一个属性 location，它将端点的地址指定为 URL。SOAP 1.2 地址 元素在命名空间 http://schemas.xmlsoap.org/wsdl/soap12/ 中定义。

例 12.2 “SOAP 1.2 端口元素” 显示用于通过 HTTP 发送 SOAP 1.2 消息 的端口 元素。

例 12.2. SOAP 1.2 端口元素

```
<definitions ...
        xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" ... >
  <service name="SOAP12Service">
    <port binding="SOAP12Binding" name="SOAP12Port">
      <soap12:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

**其他消息类型**

**当您的消息映射到 SOAP 以外的任何有效负载格式时，您必须使用 HTTP address 元素来指定端点的地址。它有一个属性 location，它将端点的地址指定为 URL。HTTP address 元素在命名空间 http://schemas.xmlsoap.org/wsdl/http/ 中定义。**

**例 12.3 “HTTP 端口元素” 显示用于发送 XML 消息 的端口 元素。**

**例 12.3. HTTP 端口元素**

```
<definitions ...
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" ... >
  <service name="HTTPService">
    <port binding="HTTPBinding" name="HTTPPort">
      <http:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

## 12.2. 配置一个 CONSUMER

### 12.2.1. HTTP 消费者端点的机制

**HTTP 消费者端点可以指定多个 HTTP 连接属性，包括端点是否自动接受重定向响应，端点是否可以使用块，端点是否请求 keep-alive，以及端点如何与代理交互。除了 HTTP 连接属性外，HTTP 使用者端点还可以指定它如何进行保护。**

**可以使用两种机制配置消费者端点：**

- 配置
- WSDL

### 12.2.2. 使用配置

#### Namespace

用于配置 HTTP 消费者端点的元素在命名空间 http://cxf.apache.org/transports/http/configuration 中定义。它通常被引用使用前缀 http-conf。要使用 HTTP 配置元素，您必须将 例 12.4 “HTTP Consumer 配置命名空间” 中显示的行添加到端点配置文件的 beans 元素中。另外，您必须将配置元素的命名空间添加到 xsi:schemaLocation 属性中。

**例 12.4. HTTP Consumer 配置命名空间**

```
<beans ...
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
      ...
      xsi:schemaLocation="...
                          http://cxf.apache.org/transports/http/configuration
                            http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          ...">
```

#### Undertow 运行时或 Netty 运行时

您可以使用 http-conf 命名空间中的元素来配置 Undertow 运行时或 Netty 运行时。

#### conduit 元素

您可以使用 http-conf:conduit 元素及其子项配置 HTTP 使用者端点。http-conf:conduit 元素采用单个属性，名称，用于指定与端点对应的 WSDL 端口 元素。name 属性的值采用 *portQName*'.http-conduit' 形式。例 12.5 “http-conf:conduit Element” 显示 http-conf:conduit 元素，用于为由 WSDL 片段 < port binding="widgetSOAPBinding" name="widgetSOAPPort > 指定的端点添加配置（当端点目标命名空间为 http://widgets.widgetvendor.net 时）。

**例 12.5. http-conf:conduit Element**

```
...
```

```
<http-conf:conduit name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-conduit">
  ...
</http-conf:conduit>
...
```

**http-conf:conduit 元素具有指定配置信息的子元素。它们在 表 12.1 “用于配置 HTTP 消费者端点的元素” 中进行了描述。**

**表 12.1. 用于配置 HTTP 消费者端点的元素**

| 元素 | 描述 |
| --- | --- |
| **http-conf:client** | 指定 HTTP 连接属性，如超时、保留请求、内容类型等。请参阅 “client 元素”一节。 |
| **http-conf:authorization** | 指定用于配置端点使用的基本验证方法的参数。首选的方法是提供 **http-conf:basicAuthSupplier** 对象。 |
| **http-conf:proxyAuthorization** | 指定用于针对传出 HTTP 代理服务器配置基本身份验证的参数。 |
| **http-conf:tlsClientParameters** | 指定用来配置 SSL/TLS 的参数。 |
| **http-conf:basicAuthSupplier** | 指定提供端点使用的基本身份验证信息的 bean 引用或类名称，可以是抢占，或响应 401 HTTP 质询。 |
| **http-conf:trustDecider** | 指定在传输任何信息前，检查 HTTP (S) **URLConnection** 对象的 bean 引用或类名称，以建立与 HTTPS 服务提供商的连接。 |

**client 元素**

**http-conf:client 元素用于配置消费者端点的 HTTP 连接的非安全属性。其属性（如 表 12.2 “HTTP Consumer 配置属性” 所述）指定连接的属性。**

**表 12.2. HTTP Consumer 配置属性**

| 属性 | 描述 |
| --- | --- |
| **connectionTimeout** | 指定消费者在超时前尝试建立连接的时间（以毫秒为单位）。默认值为 **30000**。<br><br>**0** 指定消费者将无限期地发送请求。 |

| 属性 | 描述 |
| --- | --- |
| **receiveTimeout** | 指定消费者在超时前等待响应的时间（以毫秒为单位）。默认值为 **30000**。<br><br>**0** 指定消费者将无限期等待。 |
| **AutoRedirect** | 指定消费者是否自动遵循服务器发出的重定向。默认值为 **false**。 |
| **MaxRetransmits** | 指定消费者重新传输请求以满足重定向的次数。默认值为 **-1，** 它指定允许无限重新传输。 |
| **allowChunking** | 指定消费者是否使用块发送请求。默认值为 **true**，指定消费者在发送请求时使用块。<br><br>如果以下之一为 true，则无法使用块：<br><br>● **http-conf：basicAuth 供应商** 被配置为以抢占方式提供凭证。<br><br>● **AutoRedirect** 设置为 **true**。<br><br>在这两种情况下，都会忽略 **AllowChunking** 的值，并禁止块。 |
| **accept** | 指定消费者为处理准备的介质类型。该值用作 HTTP **Accept** 属性的值。该属性值使用多用途互联网邮件扩展(MIME)类型来指定。 |
| **AcceptLanguage** | 指定消费者选择接收响应的语言（例如，美国英语）。该值用作 HTTP **AcceptLanguage** 属性的值。<br><br>语言标签由 Standards (ISO)的国际组织规范，通常通过组合语言代码（由 ISO-639 标准和国家代码决定）决定，由 ISO-3166 标准，由连字符分开。例如，en-US 代表美国英语。 |
| **AcceptEncoding** | 指定消费者准备处理的内容编码。内容编码标签由互联网分配号授权机构(IANA)规范。该值用作 HTTP **AcceptEncoding** 属性的值。 |

| 属性 | 描述 |
| --- | --- |
| **ContentType** | 指定消息正文中发送的数据的介质类型。介质类型使用多用途互联网邮件扩展(MIME)类型来指定。该值用作 HTTP **ContentType** 属性的值。默认为 **text/xml**。<br><br>对于 Web 服务，这应设置为 **text/xml**。如果客户端将 HTML 表单数据发送到 CGI 脚本，则应将其设置为 **application/x-www-form-urlencoded**。如果 HTTP POST 请求绑定到固定有效负载格式（与 SOAP 不同），则内容类型通常设置为 **application/octet-stream**。 |
| **主机** | 指定正在调用请求的资源的互联网主机和端口号。该值用作 HTTP **Host** 属性的值。<br><br>通常不需要此属性。只有某些 DNS 场景或应用程序设计才需要它。例如，它指示客户端为集群优先选择的主机（即，虚拟服务器映射到同一互联网协议(IP)地址）。 |
| **连接** | 指定在每次请求/响应对话框后是否要保持特定的连接打开或关闭。有效值有两个：<br><br>● **keep-Alive**（默认）- 指定消费者在初始请求/响应序列后需要保持连接打开。如果服务器遵循它，则连接会被保持打开，直到消费者关闭为止。<br><br>● **close** - 指定在每个请求/响应序列后与服务器的连接关闭。 |
| **CacheControl** | 指定关于必须遵循链中涉及的缓存的行为指令，该指令由消费者到服务提供商的请求组成。请参阅 第 12.2.4 节 “消费者缓存控制指令”。 |
| **cookie** | 指定要随所有请求发送的静态 Cookie。 |
| **BrowserType** | 指定请求源自的浏览器的信息。在 World Wide Web consortium (W3C)的 HTTP 规范中，这也称为 **user-agent**。有些服务器根据发送请求的客户端进行优化。 |

| 属性 | 描述 |
| --- | --- |
| **Referer** | 指定指示消费者在特定服务上发出请求的资源 URL。该值用作 HTTP **Referer** 属性的值。<br><br>当请求是浏览器用户点击超链接而不是键入 URL 时，会使用此 HTTP 属性。这允许服务器根据前面的任务流优化处理，并为日志、优化缓存、跟踪过时或输入的链接等目的生成到资源的后端列表。但是，它通常不在 Web 服务应用程序中使用。<br><br>如果 **AutoRedirect** 属性设为 **true**，并且请求被重定向，则在 **Referer** 属性中指定的任何值都会被覆盖。HTTP Referer 属性的值设置为重定向消费者的原始请求的服务的 URL。 |
| **DecoupledEndpoint** | 指定通过单独的 provider→consumer 连接接收响应的分离端点的 URL。有关使用分离端点的更多信息，请参阅 第 12.6 节 “以 Decoupled 模式使用 HTTP 传输”。<br><br>您必须配置使用者端点和服务提供商端点，以使用 WS-Addressing 来使分离的端点正常工作。 |
| **ProxyServer** | 指定路由请求的代理服务器的 URL。 |
| **ProxyServerPort** | 指定路由请求的代理服务器的端口号。 |
| **ProxyServerType** | 指定用于路由请求的代理服务器的类型。有效值为：<br><br>• **HTTP**（默认）<br>• **SOCKS** |

**Example**

**例 12.6 “HTTP 消费者端点配置” 显示 HTTP 消费者端点的配置，该端点希望在请求之间保持与提供程序的连接，该端点将在每次调用后重新传输请求，且无法使用块流。**

**例 12.6. HTTP 消费者端点配置**

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

  <http-conf:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
    <http-conf:client Connection="Keep-Alive"
```

```
                MaxRetransmits="1"
                AllowChunking="false" />
  </http-conf:conduit>
</beans>
```

**更多信息**

有关 HTTP conduits 的详情，请参考 第 16 章 *conduits*。

### 12.2.3. 使用 WSDL

**Namespace**

用于配置 HTTP 消费者端点的 WSDL 扩展元素在命名空间 http://cxf.apache.org/transports/http/configuration 中定义。它通常被引用使用前缀 http-conf。要使用 HTTP 配置元素，您必须将 例 12.7 “HTTP Consumer WSDL 元素的命名空间” 中显示的行添加到端点的 WSDL 文档中的 定义 元素中。

**例 12.7. HTTP Consumer WSDL 元素的命名空间**

```
<definitions ...
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
```

**Undertow 运行时或 Netty 运行时**

您可以使用 http-conf 命名空间中的元素来配置 Undertow 运行时或 Netty 运行时。

**client 元素**

http-conf:client 元素用于在 WSDL 文档中指定 HTTP 使用者的连接属性。http-conf:client 元素是 WSDL 端口 元素的子级。它的属性与配置文件中使用的 client 元素相同。这些属性在 表 12.2 “HTTP Consumer 配置属性” 中进行了描述。

**Example**

例 12.8 “WSDL 配置 HTTP 消费者端点” 显示一个 WSDL 片段，它将配置 HTTP 使用者端点来指定它不与缓存交互。

**例 12.8. WSDL 配置 HTTP 消费者端点**

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:client CacheControl="no-cache" />
  </port>
</service>
```

### 12.2.4. 消费者缓存控制指令

表 12.3 “http-conf:client Cache Control Directives” 列出 HTTP 使用者支持的缓存控制指令。

**表 12.3. http-conf:client Cache Control Directives**

| 指令 | 行为 |
| --- | --- |
| no-cache | 缓存不能在不首先重新评估该服务器的情况下，使用特定的响应来满足后续请求。如果使用这个值指定特定的响应标头字段，则限制仅适用于响应中的标头字段。如果没有指定响应标头字段，则限制适用于整个响应。 |
| no-store | 缓存不得存储响应的任何部分或调用它的请求的任何部分。 |
| max-age | 消费者可以接受其年龄不超过指定时间（以秒为单位）的响应。 |
| max-stale | 消费者可以接受超过其过期时间的响应。如果一个值被分配给 max-stale，它代表响应的过期时间超过秒数，消费者仍然可以接受该响应。如果没有分配值，则消费者可以接受任何年龄的过时的响应。 |
| min-fresh | 消费者希望获得至少指定秒数的最新响应。 |
| no-transform | 缓存不得修改内容在提供程序和消费者之间响应中的介质类型或位置。 |
| only-if-cached | 缓存应仅返回当前存储在缓存中的响应，而不返回需要重新加载或重新验证的响应。 |
| cache-extension | 指定其他缓存指令的额外扩展。扩展可以是信息或行为。在标准指令的上下文中指定扩展指令，以便不了解扩展指令的应用程序可以遵循 standard 指令强制的行为。 |

## 12.3. 配置服务提供商

### 12.3.1. HTTP 服务提供商的机制

HTTP 服务提供商端点可以指定多个 HTTP 连接属性，包括它将遵守实时请求、与缓存交互的方式，以及与消费者通信错误的方式。

服务提供商端点可使用两种机制进行配置：

- 配置
- WSDL

### 12.3.2. 使用配置

**Namespace**

用于配置 HTTP 供应商端点的元素在命名空间 http://cxf.apache.org/transports/http/configuration 中定义。它通常被引用使用前缀 http-conf。要使用 HTTP 配置元素，您必须将 例 12.9 “HTTP 供应商配置命名空间” 中显示的行添加到端点配置文件的 beans 元素中。另外，您必须将配置元素的命名空间添加到 xsi:schemaLocation 属性中。

**例 12.9. HTTP 供应商配置命名空间**

```
<beans ...
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
      ...
      xsi:schemaLocation="...
                          http://cxf.apache.org/transports/http/configuration
                            http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          ...">
```

**Undertow 运行时或 Netty 运行时**

您可以使用 http-conf 命名空间中的元素来配置 Undertow 运行时或 Netty 运行时。

### destination 元素

您可以使用 http-conf:destination 元素及其子项配置 HTTP 服务提供商端点。http-conf:destination 元素使用单个属性，名称，用于指定与端点对应的 WSDL 端口 元素。name 属性的值采用 *portQName*'.http-destination' 形式。例 12.10 “http-conf:destination Element” 显示 http-conf:destination 元素，用于在端点目标命名空间时为 WSDL 片段 < port binding="widgetSOAPBinding" name="widgetSOAPPort > 指定的端点添加配置。

**例 12.10. http-conf:destination Element**

```
...
  <http-conf:destination name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-
destination">
    ...
  </http-conf:destination>
...
```

http-conf:destination 元素具有多个指定配置信息的子元素。它们在 表 12.4 “用于配置 HTTP 服务提供商端点的元素” 中进行了描述。

**表 12.4. 用于配置 HTTP 服务提供商端点的元素**

| 元素 | 描述 |
| --- | --- |
| **http-conf:server** | 指定 HTTP 连接属性。请参阅 “server 元素”一节。 |
| **http-conf:contextMatchStrategy** | 指定配置上下文匹配策略以处理 HTTP 请求的参数。 |
| **http-conf:fixedParameterOrder** | 指定此目的地处理的 HTTP 请求的参数顺序是否已修复。 |

### server 元素

http-conf:server 元素用于配置服务提供商端点的 HTTP 连接的属性。其属性（如 表 12.5 “HTTP 服务提供商配置属性” 所述）指定连接的属性。

**表 12.5. HTTP 服务提供商配置属性**

| 属性 | 描述 |
| --- | --- |
| **receiveTimeout** | 设置服务提供商在连接超时前尝试接收请求的时间长度（以毫秒为单位）。默认值为 **30000**。<br><br>**0** 指定供应商不会超时。 |

| 属性 | 描述 |
| --- | --- |
| **SuppressClientSendErrors** | 指定在收到请求时出现错误时是否抛出异常。默认值为 **false** ；在遇到错误时会抛出异常。 |
| **SuppressClientReceiveErrors** | 指定当遇到错误向消费者发送响应时是否抛出异常。默认值为 **false** ；在遇到错误时会抛出异常。 |
| **HonorKeepAlive** | 指定服务提供商是否遵循在发送响应后保持打开的连接请求。默认值为 **false**; keep-alive 请求会被忽略。 |
| **RedirectURL** | 指定在客户端请求中不再适合请求的资源时，应重定向客户端请求的 URL。在这种情况下，如果服务器响应的第一行中没有自动设置状态代码，状态代码设置为 302，状态描述设置为 Object Moved。该值用作 HTTP RedirectURL 属性的值。 |
| **CacheControl** | 指定由链中涉及的缓存遵循的行为指令，这些指令由从服务提供商到消费者的响应组成。请参阅 第 12.3.4 节 “Service Provider Cache Control Directives”。 |
| **ContentLocation** | 设置响应中发送资源的 URL。 |
| **ContentType** | 指定响应中发送信息的介质类型。介质类型使用多用途互联网邮件扩展(MIME)类型来指定。该值用作 HTTP ContentType 位置的值。 |
| **ContentEncoding** | 指定应用于服务提供商发送的信息的额外内容编码。内容编码标签由互联网分配号授权机构(IANA)规范。可能的内容编码值包括 zip、gzip、compress、deflate 和 identity。这个值被用作 HTTP ContentEncoding 属性的值。<br><br>内容编码的主要用途是允许使用某些编码机制（如 zip 或 gzip）压缩文档。Apache CXF 在内容编码时不执行验证。用户负责确保应用程序级别支持指定内容编码。 |
| **ServerType** | 指定发送响应的服务器类型。值采用 ***program-name/version*** ；例如 **Apache/1.2.5**。 |

**Example**

**例 12.11 “HTTP 服务提供商端点配置” 显示符合 keep-alive 请求并阻止所有通信错误的 HTTP 服务提供商端点配置。**

**例 12.11. HTTP 服务提供商端点配置**

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

  <http-conf:destination name="{http://apache.org/hello_world_soap_http}SoapPort.http-
destination">
    <http-conf:server SuppressClientSendErrors="true"
                      SuppressClientReceiveErrors="true"
                      HonorKeepAlive="true" />
  </http-conf:destination>
</beans>
```

### 12.3.3. 使用 WSDL

#### Namespace

用于配置 HTTP 提供程序端点的 WSDL 扩展元素在命名空间 http://cxf.apache.org/transports/http/configuration 中定义。它通常被引用使用前缀 http-conf。要使用 HTTP 配置元素，您必须将 例 12.12 “HTTP Provider WSDL 元素的命名空间” 中显示的行添加到端点的 WSDL 文档中的 定义 元素中。

**例 12.12. HTTP Provider WSDL 元素的命名空间**

```
<definitions ...
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
```

#### Undertow 运行时或 Netty 运行时

您可以使用 http-conf 命名空间中的元素来配置 Undertow 运行时或 Netty 运行时。

#### server 元素

http-conf:server 元素用于在 WSDL 文档中指定 HTTP 服务提供商的连接属性。http-conf:server 元素是 WSDL 端口 元素的子级。它的属性与配置文件中使用的 server 元素相同。这些属性在 表 12.5 “HTTP 服务提供商配置属性” 中进行了描述。

#### Example

例 12.13 “WSDL 配置 HTTP 服务提供商端点” 显示 WSDL 片段，该片段配置 HTTP 服务提供商端

点，指定它不会与缓存交互。

**例 12.13. WSDL 配置 HTTP 服务提供商端点**

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:server CacheControl="no-cache" />
  </port>
</service>
```

### 12.3.4. Service Provider Cache Control Directives

表 12.6 “http-conf:server Cache Control Directives” 列出 HTTP 服务提供商支持的缓存控制指令。

**表 12.6. http-conf:server Cache Control Directives**

| 指令 | 行为 |
| --- | --- |
| no-cache | 缓存不能在不首先重新评估该服务器的情况下，使用特定的响应来满足后续请求。如果使用这个值指定特定的响应标头字段，则限制仅适用于响应中的标头字段。如果没有指定响应标头字段，则限制适用于整个响应。 |
| public | 任何缓存都可以存储响应。 |
| private | 公共(**共享**)缓存无法存储响应，因为响应适用于单个用户。如果使用这个值指定特定的响应标头字段，则限制仅适用于响应中的标头字段。如果没有指定响应标头字段，则限制适用于整个响应。 |
| no-store | 缓存不得存储响应的任何部分或调用它的请求的任何部分。 |
| no-transform | 缓存不得修改服务器与客户端之间的响应中的介质类型或内容位置。 |
| must-revalidate | 缓存必须重新验证与响应相关的过期条目，然后才能在后续响应中使用该条目。 |
| proxy-revalidate | 执行与 must-revalidate 相同，但只能对共享缓存强制，并由私有非共享缓存忽略。使用这个指令时，还必须使用 public cache 指令。 |
| max-age | 客户端可以接受其年龄不超过指定秒数的响应。 |

| 指令 | 行为 |
| --- | --- |
| s-max-age | 执行与 max-age 相同，但只能对共享缓存强制，并由私有非共享缓存忽略。s-max-age 指定的年龄会覆盖由 max-age 指定的年龄。使用这个指令时，还必须使用 proxy-revalidate 指令。 |
| cache-extension | 指定其他缓存指令的额外扩展。扩展可以是信息或行为。在标准指令的上下文中指定扩展指令，以便不了解扩展指令的应用程序可以遵循 standard 指令强制的行为。 |

## 12.4. 配置 UNDERTOW 运行时

### 概述

**Undertow 运行时由 HTTP 服务提供商和 HTTP 用户使用，使用分离的端点。可以配置运行时的线程池，您也可以通过 Undertow 运行时为 HTTP 服务提供商设置多个安全设置。**

### Maven 依赖项

**如果使用 Apache Maven 作为构建系统，您可以通过在项目的 pom.xml 文件中包含以下依赖项，将 Undertow 运行时添加到项目中：**

```
<dependency>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-rt-transports-http-undertow</artifactId>
   <version>${cxf-version}</version>
</dependency>
```

### Namespace

**用于配置 Undertow 运行时的元素在命名空间 http://cxf.apache.org/transports/http-undertow/configuration 中定义。要使用 Undertow 配置元素，您必须将 例 12.14 “Undertow 运行时配置命名空间” 中显示的行添加到端点配置文件的 beans 元素中。在本例中，命名空间被分配了前缀 httpu。另外，您必须将配置元素的命名空间添加到 xsi:schemaLocation 属性中。**

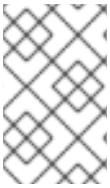
**例 12.14. Undertow 运行时配置命名空间**

```
<beans ...
      xmlns:httpu="http://cxf.apache.org/transports/http-undertow/configuration"
      ...
      xsi:schemaLocation="...
```

```
                http://cxf.apache.org/transports/http-undertow/configuration
                  http://cxf.apache.org/schemas/configuration/http-undertow.xsd
...">
```

**engine-factory 元素**

**httpu:engine-factory 元素是用于配置应用使用的 Undertow 运行时的 root 元素。它有一个必需属性 bus，其值是管理所配置的 Undertow 实例的 总线 名称。**



**注意**

该值通常是 **cxf**，这是默认 总线实例的名称。

**http:engine-factory 元素有三个子对象，其中包含用于配置由 Undertow 运行时工厂实例化的 HTTP 端口的信息。子项在 表 12.7 “配置 Undertow 运行时工厂的元素” 中进行了描述。**

**表 12.7. 配置 Undertow 运行时工厂的元素**

| 元素 | 描述 |
| --- | --- |
| **httpu:engine** | 指定特定 Undertow 运行时实例的配置。请参阅 “engine 元素”一节。 |
| **httpu:identifiedTLSServerParameters** | 指定用于保护 HTTP 服务提供商的可重复使用的属性集合。它有一个属性 **id**，用于指定可以引用属性集的唯一标识符。 |
| **httpu:identifiedThreadingParameters** | 指定用于控制 Undertow 实例的线程池的可重复使用的属性集合。它有一个属性 **id**，用于指定可以引用属性集的唯一标识符。<br><br>请参阅 “配置线程池”一节。 |

**engine 元素**

**httpu:engine 元素用于配置 Undertow 运行时的特定实例。它有两个属性 host，它指定了带有嵌入式 undertow 和 port 的全局 IP 地址，用于指定由 Undertow 实例管理的端口数。**

**注意**

**您可以为 port 属性指定 0 值。httpu:engine 元素中指定的任何线程属性，其 port 属性设置为 0，作为未明确配置的所有 Undertow 侦听程序的配置。**

**每个 httpu:engine 元素可以有两个子级：一个用于配置安全属性，另一个用于配置 Undertow 实例的线程池。对于每种配置，您可以直接提供配置信息，或者您可以提供对父 httpu:engine-factory 元素中定义的一组配置属性的引用。**

**表 12.8 “用于配置 Undertow 运行时实例的元素” 中描述了用来提供配置属性的子元素。**

**表 12.8. 用于配置 Undertow 运行时实例的元素**

| 元素 | 描述 |
| --- | --- |
| **httpu:tlsServerParameters** | 指定一组属性，用于配置用于特定 Undertow 实例的安全性。 |
| **httpu:tlsServerParametersRef** | 指的是由 **identifiedTLSServerParameters** 元素定义的一组安全属性。**id** 属性提供引用的 **TLSServerParameters 元素的** id。 |
| **httpu:threadingParameters** | 指定特定 Undertow 实例使用的线程池的大小。请参阅 “配置线程池”一节。 |
| **httpu:threadingParametersRef** | 指的是由 识别**ThreadingParameters** 元素定义的一组属性。**id** 属性提供引用的 **ThreadingParameters 元素的** id。 |

**配置线程池**

**您可以通过以下任一方式配置 Undertow 实例线程池的大小：**

- **使用 engine-factory 元素 中标识的ThreadingParameters 元素来指定线程池的大小。然后，您可以使用 threadingParametersRef 元素引用元素。**
- **使用 threadingParameters 元素直接指定线程池的大小。**

**threadingParameters 有两个属性，用于指定线程池的大小。这些属性在 表 12.9 “用于配置 Undertow 线程池的属性” 中进行了描述。**

**注意**

**httpu:identifiedThreadingParameters 元素有一个子 threadingParameters 元素。**

**表 12.9. 用于配置 Undertow 线程池的属性**

| 属性 | 描述 |
| --- | --- |
| **workerIOThreads** | 指定为 worker 创建的 I/O 线程数量。如果没有指定，则会选择 dafult 值。默认值为每个 CPU 内核一个 I/O 线程。 |
| **minThreads** | 指定 Undertow 实例可用于处理请求的最少线程数量。 |
| **maxThreads** | 指定 Undertow 实例可用于处理请求的最大线程数。 |

**Example**

**例 12.15 “配置 Undertow 实例” 显示在端口号 9001 上配置 Undertow 实例的配置片段。**

**例 12.15. 配置 Undertow 实例**

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpu="http://cxf.apache.org/transports/http-undertow/configuration"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
  xsi:schemaLocation="http://cxf.apache.org/configuration/security
        http://cxf.apache.org/schemas/configuration/security.xsd
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://cxf.apache.org/transports/http-undertow/configuration
          http://cxf.apache.org/schemas/configuration/http-undertow.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  ...

  <httpu:engine-factory bus="cxf">
    <httpu:identifiedTLSServerParameters id="secure">
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="JKS" password="password"
                file="certs/cherry.jks"/>
      </sec:keyManagers>
    </httpu:identifiedTLSServerParameters>
```

```
    <httpu:engine port="9001">
      <httpu:tlsServerParametersRef id="secure" />
      <httpu:threadingParameters minThreads="5"
                                 maxThreads="15" />
    </httpu:engine>
  </httpu:engine-factory>
</beans>
```

### 限制并发请求和队列大小

您可以配置 Request Limiting Handler，该处理程序将为并发连接请求的最大数量和 Undertow 服务器实例可以处理的队列大小设置限制。此配置示例如下所示 例 12.16 “限制连接请求和队列大小”

**表 12.10. 用于配置 Request Limiting Handler 的属性**

| 属性 | 描述 |
| --- | --- |
| **maximumConcurrentRequests** | 指定 Undertow 实例可以处理的并发请求数。如果请求数超过这个限制，请求将排队。 |
| **queueSize** | 指定可由 Undertow 实例排队进行处理的请求总数。如果请求数超过这个限制，请求将被拒绝。 |

**例 12.16. 限制连接请求和队列大小**

```
<httpu:engine-factory>
    <httpu:engine port="8282">
        <httpu:handlers>
            <bean class="org.jboss.fuse.quickstarts.cxf.soap.CxfRequestLimitingHandler">
                <property name="maximumConcurrentRequests" value="1" />
                <property name="queueSize" value="1"/>
            </bean>
        </httpu:handlers>
    </httpu:engine>
</httpu:engine-factory>
```

## 12.5. 配置 NETTY 运行时

### 概述

Netty 运行时由 HTTP 服务提供商和 HTTP 用户使用分离的端点。可以配置运行时的线程池，您还可以通过 Netty 运行时为 HTTP 服务提供商设置多个安全设置。

**Maven 依赖项**

如果使用 Apache Maven 作为构建系统，您可以通过在项目的 pom.xml 文件中包含以下依赖项，将 Netty 运行时的服务器端（用于定义 Web 服务端点）添加到项目中：

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-netty-server</artifactId>
  <version>${cxf-version}</version>
</dependency>
```

您可以通过在项目的 pom.xml 文件中包含以下依赖项，将 Netty 运行时的客户端实现（用于定义 Web 服务客户端）添加到项目中：

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-netty-client</artifactId>
  <version>${cxf-version}</version>
</dependency>
```

**Namespace**

用于配置 Netty 运行时的元素在命名空间 http://cxf.apache.org/transports/http-netty-server/configuration 中定义。通常使用前缀 httpn。要使用 Netty 配置元素，您必须将 例 12.17 “Netty Runtime 配置命名空间” 中显示的行添加到端点配置文件的 beans 元素中。另外，您必须将配置元素的命名空间添加到 xsi:schemaLocation 属性中。

**例 12.17. Netty Runtime 配置命名空间**

```
<beans ...
      xmlns:httpn="http://cxf.apache.org/transports/http-netty-server/configuration"
      ...
      xsi:schemaLocation="...
            http://cxf.apache.org/transports/http-netty-server/configuration
          http://cxf.apache.org/schemas/configuration/http-netty-server.xsd
            ...">
```

**engine-factory 元素**

httpn:engine-factory 元素是用于配置供应用使用的 Netty 运行时的根元素。它有一个必需属性 bus，其值是管理所配置的 Netty 实例的 总线 名称。

**注意**

该值通常是 **cxf**，这是默认 总线实例的名称。

**httpn:engine-factory 元素有三个子项，其中包含用于配置由 Netty 运行时工厂实例化的 HTTP 端口的信息。子项在 表 12.11 “配置 Netty Runtime Factory 的元素” 中进行了描述。**

**表 12.11. 配置 Netty Runtime Factory 的元素**

| 元素 | 描述 |
| --- | --- |
| **httpn:engine** | 指定特定 Netty 运行时实例的配置。请参阅 “engine 元素”一节。 |
| **httpn:identifiedTLSServerParameters** | 指定用于保护 HTTP 服务提供商的可重复使用的属性集合。它有一个属性 **id**，用于指定可以引用属性集的唯一标识符。 |
| **httpn:identifiedThreadingParameters** | 指定控制 Netty 实例的线程池的可重复使用的属性集合。它有一个属性 **id**，用于指定可以引用属性集的唯一标识符。<br><br>请参阅 “配置线程池”一节。 |

**engine 元素**

**httpn:engine 元素用于配置 Netty 运行时的特定实例。表 12.12 “配置 Netty Runtime 实例的属性” 显示 httpn:engine 元素支持的属性。**

**表 12.12. 配置 Netty Runtime 实例的属性**

| 属性 | 描述 |
| --- | --- |
| **port** | 指定 Netty HTTP 服务器实例使用的端口。您可以为 port 属性指定 **0** 值。在 engine 元素中指定的线程属性，其 port 属性设置为 0，作为未明确配置的 Netty 侦听的配置。 |
| **主机** | 指定 Netty HTTP 服务器实例使用的侦听地址。该值可以是主机名或 IP 地址。如果没有指定，Netty HTTP 服务器将侦听所有本地地址。 |

| 属性 | 描述 |
| --- | --- |
| **readIdleTime** | 指定 Netty 连接的最大读取空闲时间。每当底层流上有任何读取操作时，计时器都会被重置。 |
| **writeIdleTime** | 指定 Netty 连接的最大写入闲置时间。每当底层流上有任何写入操作时，计时器都会被重置。 |
| **maxChunkContentSize** | 指定 Netty 连接的最大聚合内容大小。默认值为 10MB。 |

**httpn:engine 元素具有一个子元素，用于配置安全属性和一个子元素，用于配置 Netty 实例的线程池。对于每种配置，您可以直接提供配置信息，或者您可以提供对父 httpn:engine-factory 元素中定义的一组配置属性的引用。**

**httpn:engine 支持的子元素显示在 表 12.13 “配置 Netty 运行时实例的元素” 中。**

**表 12.13. 配置 Netty 运行时实例的元素**

| 元素 | 描述 |
| --- | --- |
| **httpn:tlsServerParameters** | 指定一组属性，用于配置用于特定 Netty 实例的安全性。 |
| **httpn:tlsServerParametersRef** | 指的是由 **identifiedTLSServerParameters** 元素定义的一组安全属性。**id** 属性提供引用的 **TLSServerParameters 元素的** id。 |
| **httpn:threadingParameters** | 指定特定 Netty 实例使用的线程池的大小。请参阅 “配置线程池”一节。 |
| **httpn:threadingParametersRef** | 指的是由 识别**ThreadingParameters** 元素定义的一组属性。**id** 属性提供引用的 **ThreadingParameters 元素的** id。 |
| **httpn:sessionSupport** | 为 **true** 时，启用对 HTTP 会话的支持。默认为 **false**。 |
| **httpn:reuseAddress** | 指定一个布尔值来设置 **ReuseAddress** TCP socket 选项。默认为 **false**。 |

**配置线程池**

您可以通过以下方法配置 Netty 实例线程池的大小：

- 使用 engine-factory 元素 中标识的ThreadingParameters 元素来指定线程池的大小。然后，您可以使用 threadingParametersRef 元素引用元素。
- 使用 threadingParameters 元素直接指定线程池的大小。

threadingParameters 元素具有一个属性来指定线程池的大小，如 表 12.14 “配置网络线程池的属性” 所述。

注意

httpn:identifiedThreadingParameters 元素有一个子 threadingParameters 元素。

表 12.14. 配置网络线程池的属性

| 属性 | 描述 |
| --- | --- |
| threadPoolSize | 指定 Netty 实例可用于处理请求的线程数量。 |

Example

例 12.18 “配置 Netty 实例” 显示配置各种 Netty 端口的配置片段。

例 12.18. 配置 Netty 实例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:beans="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:h="http://cxf.apache.org/transports/http/configuration"
      xmlns:httpn="http://cxf.apache.org/transports/http-netty-server/configuration"
      xmlns:sec="http://cxf.apache.org/configuration/security"
      xsi:schemaLocation="
       http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
       http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
       http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
       http://cxf.apache.org/transports/http-netty-server/configuration
          http://cxf.apache.org/schemas/configuration/http-netty-server.xsd"
```

```
>
  ...
  <httpn:engine-factory bus="cxf">
    <httpn:identifiedTLSServerParameters id="sample1">
      <httpn:tlsServerParameters jsseProvider="SUN" secureSocketProtocol="TLS">
        <sec:clientAuthentication want="false" required="false"/>
      </httpn:tlsServerParameters>
    </httpn:identifiedTLSServerParameters>

    <httpn:identifiedThreadingParameters id="sampleThreading1">
      <httpn:threadingParameters threadPoolSize="120"/>
    </httpn:identifiedThreadingParameters>

    <httpn:engine port="9000" readIdleTime="30000" writeIdleTime="90000">
      <httpn:threadingParametersRef id="sampleThreading1"/>
    </httpn:engine>

    <httpn:engine port="0">
      <httpn:threadingParameters threadPoolSize="400"/>
    </httpn:engine>

    <httpn:engine port="9001" readIdleTime="40000" maxChunkContentSize="10000">
      <httpn:threadingParameters threadPoolSize="99" />
      <httpn:sessionSupport>true</httpn:sessionSupport>
    </httpn:engine>

    <httpn:engine port="9002">
      <httpn:tlsServerParameters>
        <sec:clientAuthentication want="true" required="true"/>
      </httpn:tlsServerParameters>
    </httpn:engine>

    <httpn:engine port="9003">
      <httpn:tlsServerParametersRef id="sample1"/>
    </httpn:engine>

  </httpn:engine-factory>
</beans>
```

### 12.6. 以 DECOUPLED 模式使用 HTTP 传输

**概述**

在正常的 HTTP 请求/响应场景中，请求和响应使用相同的 HTTP 连接发送。服务提供程序处理请求，并响应包含相应 HTTP 状态代码和响应的响应。如果请求成功，HTTP 状态代码被设置为 200。

在某些情况下，比如使用 WS-RM 或请求执行延长的时间时，最好分离请求和响应消息。在这种情况下，服务供应商通过收到请求的 HTTP 连接的后端通道向消费者发送 202 Accepted 响应。然后，它会处

理请求，并使用新的分离的服务器→client HTTP 连接将响应发回给消费者。消费者运行时接收传入的响应，并在返回应用程序代码前将其与适当的请求相关联。

### 配置分离的交互

在分离的模式中使用 HTTP 传输需要您执行以下操作：

1. 将消费者配置为使用 WS-Addressing。

   请参阅 “配置端点以使用 WS-Addressing”一节。

2. 将消费者配置为使用分离的端点。

   请参阅 “配置消费者”一节。

3. 配置使用者与 交互以使用 WS-Addressing 的任何服务提供商。

   请参阅 “配置端点以使用 WS-Addressing”一节。

### 配置端点以使用 WS-Addressing

指定消费者与之交互的消费者和服务供应商使用 WS-Addressing。

您可以使用两种方式之一指定端点使用 WS 寻址：

- 将 wswa:UsingAddressing 元素添加到端点的 WSDL 端口 元素中，如 例 12.19 “使用 WSDL 激活 WS-Addressing” 所示。

  **例 12.19. 使用 WSDL 激活 WS-Addressing**

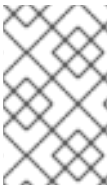
  ```
  ...
  <service name="WidgetSOAPService">
    <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
      <soap:address="http://widgetvendor.net/widgetSeller" />
      <wswa:UsingAddressing xmlns:wswa="http://www.w3.org/2005/02/addressing/wsdl"/>
    </port>
  </service>
  ...
  ```

-

- 将 WS-Addressing 策略添加到端点的 WSDL 端口 元素中，如 例 12.20 “使用策略激活 WS-Addressing” 所示。

**例 12.20. 使用策略激活 WS-Addressing**

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wsp:Policy xmlns:wsp="http://www.w3.org/2006/07/ws-policy"> <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata"> <wsp:Policy/>
</wsam:Addressing> </wsp:Policy>
  </port>
</service>
...
```



**注意**

WS-地址策略取代 wswa:UsingAddressing WSDL 元素。

**配置消费者**

使用 http-conf:conduit 元素的 DecoupledEndpoint 属性将消费者端点配置为使用分离的端点。

例 12.21 “将 Consumer 配置为使用 Decoupled HTTP 端点” 显示设置 例 12.19 “使用 WSDL 激活 WS-Addressing” 中定义的端点以使用分离端点的配置。消费者现在接收 http://widgetvendor.net/widgetSellerInbox 的所有响应。

**例 12.21. 将 Consumer 配置为使用 Decoupled HTTP 端点**

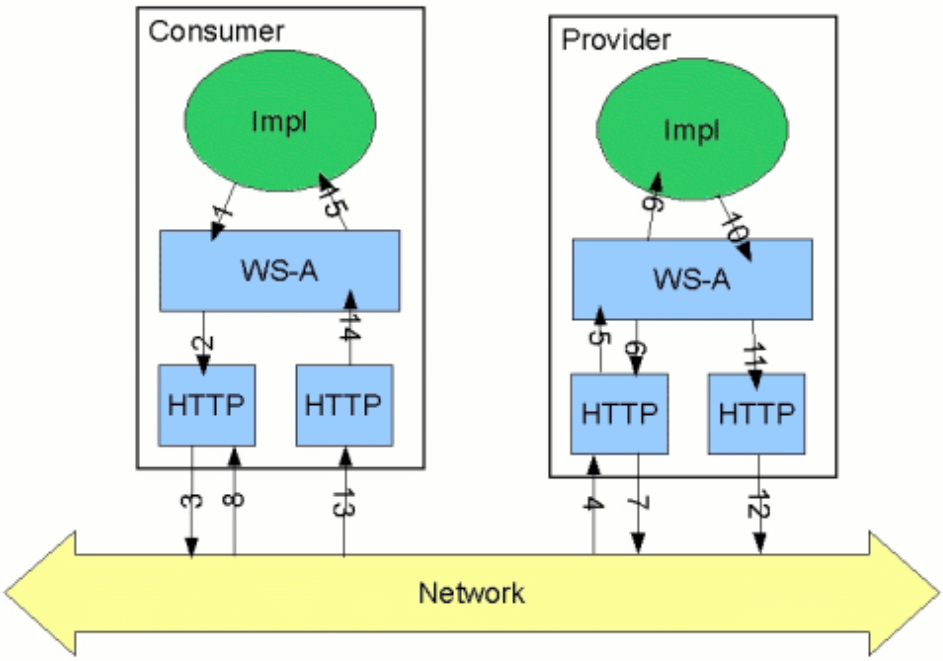
```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                          http://cxf.apache.org/schemas/configuration/http-conf.xsd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

  <http:conduit name="{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">
    <http:client DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
  </http:conduit>
</beans>
```

### 如何处理消息

在分离模式下使用 HTTP 传输可为处理 HTTP 消息增加额外复杂性。虽然增加的复杂性对应用中的实施级别代码是透明的，但了解调试的原因可能很重要。

图 12.1 “用于 Decoupled HTTP 传输的消息流” 在分离模式下使用 HTTP 时显示消息流。

图 12.1. 用于 Decoupled HTTP 传输的消息流




请求启动以下流程：

1. 消费者实施调用一个操作，并且生成了请求消息。

2. WS-地址层将 WS-A 标头添加到消息。

   当在消费者配置中指定分离端点时，分离的端点的地址会被放在 WS-A ReplyTo 标头中。

3. 消息发送到服务提供商。

4. 服务提供程序接收消息。

5. 来自消费者的请求消息被分配给供应商的 WS-A 层。

6. 由于 WS-A ReplyTo 标头没有设置为匿名，因此提供程序会发回一个 HTTP 状态代码设为 202 的信息，并确认了收到了该请求的信息。

7. HTTP 层使用原始连接的后备通道将 202 Accepted 消息发回到消费者。

8. 消费者在用于发送原始消息的 HTTP 连接的 back-channel 上接收 202 Accepted 回复。

   当消费者收到 202 Accepted 回复时，HTTP 连接将关闭。

9. 请求传递到处理请求的服务提供商的实现。

10. 当响应就绪时，它将被分配给 WS-A 层。

11. WS-A 层将 WS-Addressing 标头添加到响应消息。

12. HTTP 传输将响应发送到消费者的分离端点。

13. 消费者分离的端点从服务提供商接收响应。

14. 响应被分配给消费者的 WS-A 层，其中使用 WS-A RelatesTo 标头将其与正确的请求关联。

15. 关联的响应返回给客户端实施，调用的调用将被取消阻塞。

# 第 13 章 使用 SOAP OVER JMS

**摘要**

Apache CXF 实施 W3C 标准 SOAP/JMS 传输。此标准旨在为 SOAP/HTTP 服务提供更强大的替代方案。使用这个传输的 Apache CXF 应用程序应该能够与同时实施 SOAP/JMS 标准的应用程序进行交互。传输直接在端点的 WSDL 中配置。

注意 ： CXF 3.0 中删除了对 JMS 1.0.2 API 的支持。如果您使用红帽JBoss Fuse 6.2 或更高版本（包括 CXF 3.0），您的 JMS 提供程序必须支持 JMS 1.1 API。

## 13.1. 基本配置

**概述**

JMS 协议上的 SOAP 由全球 Web Consortium (W3C)定义，作为向大多数服务使用的定制 SOAP/HTTP 协议提供更可靠的传输层的方法。Apache CXF 实现与规格完全兼容，应该与任何兼容的框架兼容。

此传输使用 JNDI 来查找 JMS 目的地。调用操作时，请求将打包为 SOAP 消息，并在 JMS 消息的正文中发送到指定的目的地。

使用 SOAP/JMS 传输：

1. 指定传输类型是 SOAP/JMS。
2. 使用 JMS URI 指定目标目的地。
3. （可选）配置 JNDI 连接。
4. （可选）添加额外的 JMS 配置。

**指定 JMS 传输类型**

您可以在指定 WSDL 绑定时，将 SOAP 绑定配置为使用 JMS 传输。您可以将 soap:binding 元素的

transport 属性设置为 http://www.w3.org/2010/soapjms/。例 13.1 “SOAP over JMS 绑定规格” 显示使用 SOAP/JMS 的 WSDL 绑定。

**例 13.1. SOAP over JMS 绑定规格**

```
<wsdl:binding ... >
  <soap:binding style="document"
                transport="http://www.w3.org/2010/soapjms/" />
  ...
</wsdl:binding>
```

**指定目标目的地**

在为端点指定 WSDL 端口时，您可以指定 JMS 目标目的地的地址。SOAP/JMS 端点的地址规格使用与 SOAP/HTTP 端点相同的 soap:address 元素和属性。区别是地址规格。JMS 端点使用 JMS 1.0 的 URI Scheme 中定义的 JMS URI。例 13.2 “JMS URI 语法” 显示 JMS URI 的语法。

**例 13.2. JMS URI 语法**

```
jms:variant:destination?options
```

表 13.1 “JMS URI 变体” 描述 JMS URI 的可用变体。

**表 13.1. JMS URI 变体**

| Variant | 描述 |
| --- | --- |
| jndi | 指定目的地名称是 JNDI 队列名称。在使用这种变体时，您必须提供配置来访问 JNDI 提供程序。 |
| jndi-topic | 指定目的地名称是 JNDI 主题名称。在使用这种变体时，您必须提供配置来访问 JNDI 提供程序。 |
| queue | 指定目标是使用 JMS 解析的队列名称。提供的字符串传递到 **Session.createQueue （）**，以创建目的地的表示。 |
| topic | 指定目标是使用 JMS 解析的主题名称。提供的字符串被传递给 **Session.createTopic （）**，以创建目的地的表示。 |

JMS URI 的 *options* 部分用于配置传输，并在 第 13.2 节 “JMS URI” 中讨论。

例 13.3 “SOAP/JMS 端点地址” 显示 SOAP/JMS 端点的 WSDL 端口条目，该端点使用 JNDI 查找其目标目的地。

**例 13.3. SOAP/JMS 端点地址**

```
<wsdl:port ... >
  ...
  <soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
</wsdl:port>
```

#### 配置 JNDI 和 JMS 传输

SOAP/JMS 提供了多种方法来配置 JNDI 连接和 JMS 传输：

- 第 13.2 节 “JMS URI”
- 第 13.3 节 “WSDL 扩展”

### 13.2. JMS URI

#### 概述

使用 SOAP/JMS 时，使用 JMS URI 指定端点的目标目的地。JMS URI 也可以通过将一个或多个选项附加到 URI 来配置 JMS 连接。这些选项在 IETF 标准、Java 消息服务 1.0 的 URI Scheme 中详细介绍。它们可用于配置 JNDI 系统、回复目的地、要使用的交付模式和其他 JMS 属性。

#### 语法

如 例 13.4 “JMS URI 选项的语法” 所示，您可以将一个或多个选项附加到 JMS URI 的末尾，方法是将它们与带有问号(?)的目标地址分开来。多个选项由一个 符号( 和 )分隔。例 13.4 “JMS URI 选项的语法” 显示在 JMS URI 中使用多个选项的语法。

**例 13.4. JMS URI 选项的语法**

```
jms:variant:jmsAddress?option1=value1&option2=value2&_optionN_=valueN
```

#### JMS 属性

表 13.2 “JMS 属性 settable 为 URI 选项” 显示影响 JMS 传输层的 URI 选项。

**表 13.2. JMS 属性 settable 为 URI 选项**

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **conduitIdSelectorPrefix** | | [可选] 前缀为 conduit 创建的所有关联 ID 的字符串值。选择器可以使用它来侦听回复。 |
| **deliveryMode** | **持久性** | 指定是否使用 JMS **PERSISTENT** 或 **NON_PERSISTENT** 消息语义。对于 **PERSISTENT** 传输模式，JMS 代理会在确认它们之前将消息存储在持久存储中；而 **NON_PERSISTENT** 消息仅保存在内存中。 |
| **durableSubscriptionClientID** | | [可选] 指定连接的客户端标识符。此属性用于将连接与供应商代表客户端维护的状态关联。这可让后续具有相同身份的订阅者恢复之前订阅者保留的状态。 |
| **durableSubscriptionName** | | [可选] 指定订阅的名称。 |
| **messageType** | **byte** | 指定 CXF 使用的 JMS 消息类型。有效值为：<br>• byte<br>• text<br>• 二进制 |
| **password** | | [可选] 指定创建连接的密码。不建议将此属性附加到 URI 中。 |
| **priority** | **4** | 指定 JMS 消息优先级，范围从 0（最低）到 9（最高）。 |
| **receiveTimout** | **60000** | 指定使用请求/回复交换时客户端将等待回复的时间，以毫秒为单位。 |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **reconnectOnException** | **true** | [在 CXF 3.0 中弃用] 指定在发生异常时传输是否应重新连接。<br><br>从 3.0 开始，当发生异常时，传输始终重新连接。 |
| **replyToName** | | [可选] 指定队列消息的回复目的地。回复目的地会出现在 **JMSReplyTo** 标头中。建议为具有 request-reply 语义的应用程序设置此属性，因为如果未指定，则 JMS 提供程序将分配一个临时回复队列。<br><br>此属性的值根据 JMS URI 中指定的变体进行解释：<br><br>● **JNDI** 变体 - 由 JNDI 解析的目标队列的名称<br><br>● **队列** 变体 - 使用 JMS 解析的目标队列的名称 |
| **sessionTransacted** | **false** | 指定事务类型。有效值为：<br><br>● **true**-resource 本地事务<br><br>● **false**-JTA 事务 |
| **timeToLive** | **0** | 指定 JMS 提供程序将丢弃邮件的时间（以毫秒为单位）。值 **0** 表示无限生命周期。 |
| **topicReplyToName** | | [可选] 指定主题消息的回复目的地。此属性的值根据 JMS URI 中指定的变体进行解释：<br><br>● **JNDI-topic**- 被 JNDI 解析的目标主题的名称<br><br>● **topic**- JMS 解析的目标主题的名称 |
| **useConduitIdSelector** | **true** | 指定哪个 UUID 是否用作所有关联 ID 的前缀。<br><br>由于所有行为都被分配了唯一的 UUID，请将此属性设置为 **true** 可启用多个端点来共享 JMS 队列或主题。 |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **username** | | [可选] 指定用于创建连接的用户名。 |

**JNDI 属性**

表 13.3 “JNDI properties settable as URI options” 显示可用于为此端点配置 JNDI 的 URI 选项。

**表 13.3. JNDI properties settable as URI options**

| 属性 | 描述 |
| --- | --- |
| **jndiConnectionFactoryName** | 指定 JMS 连接工厂的 JNDI 名称。 |
| **jndiInitialContextFactory** | 指定 JNDI 提供程序的完全限定 Java 类名称（它必须是 **javax.jms.InitialContextFactory** 类型）。等同于设置 **java.naming.factory.initial** Java 系统属性。 |
| **jndiTransactionManagerName** | 指定在 Spring、Blueprint 或 JNDI 中搜索的 JTA 事务管理器的名称。如果找到了事务管理器，则会启用 JTA 事务。请参阅 **sessionTransacted** JMS 属性。 |
| **jndiURL** | 指定初始化 JNDI 供应商的 URL。等同于设置 **java.naming.provider.url** Java 系统属性。 |

**额外的 JNDI 属性**

**属性 java.naming.factory.initial 和 java.naming.provider.url 是标准属性，这是初始化任何 JNDI 提供程序所需的标准属性。但是，除了标准属性外，JNDI 提供程序还可能支持自定义属性。在这种情况下，您可以通过设置 jndi-*PropertyName* 格式的 URI 选项来设置任意 JNDI 属性。**

**例如，如果您使用 SUN 的 LDAP 实现的 JNDI，您可以在 JMS URI 中设置 JNDI 属性 java.naming.factory.control，如 例 13.5 “在 JMS URI 中设置 JNDI 属性” 所示。**

**例 13.5. 在 JMS URI 中设置 JNDI 属性**

```
jms:queue:FOO.BAR?jndi-
java.naming.factory.control=com.sun.jndi.ldap.ResponseControlFactory
```

**Example**

如果 JMS 提供程序尚未配置，则可以在 URI 中使用选项提供必要的 JNDI 配置详情（请参阅 表 13.3 “JNDI properties settable as URI options”）。例如，若要配置端点以使用 Apache ActiveMQ JMS 提供程序并连接到名为 test.cxf.jmstransport.queue 的队列，请使用 例 13.6 “配置 JNDI 连接的 JMS URI” 中显示的 URI。

**例 13.6. 配置 JNDI 连接的 JMS URI**

```
jms:jndi:dynamicQueues/test.cxf.jmstransport.queue
?jndiInitialContextFactory=org.apache.activemq.jndi.ActiveMQInitialContextFactory
&jndiConnectionFactoryName=ConnectionFactory
&jndiURL=tcp://localhost:61616
```

发布服务

JAX-WS 标准 publish （） 方法无法用于发布 SOAP/JMS 服务。反之，您必须使用 Apache CXF 的 JaxWsServerFactoryBean 类，如 例 13.7 “发布 SOAP/JMS 服务” 所示。

**例 13.7. 发布 SOAP/JMS 服务**

```
String address = "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
    + "?jndiInitialContextFactory"
    + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
    + "&jndiConnectionFactoryName=ConnectionFactory"
    + "&jndiURL=tcp://localhost:61500";
Hello implementor = new HelloImpl();
JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();
svrFactory.setServiceClass(Hello.class);
svrFactory.setAddress(address);
svrFactory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANSPORTID);
svrFactory.setServiceBean(implementor);
svrFactory.create();
```

例 13.7 “发布 SOAP/JMS 服务” 中的代码执行以下操作：

创建代表 t he 端点地址的 JMS URI。

实例化 JaxWsServerFactoryBean 来发布该服务。

使用服务的 JMS URI 设置 factory bean 的 address 字段。

指定工厂创建的服务将使用 SOAP/JMS 传输。

使用服务

标准 JAX-WS API 无法用于使用 SOAP/JMS 服务。反之，您必须使用 Apache CXF 的 JaxWsProxyFactoryFactoryBean 类，如 例 13.8 “使用 SOAP/JMS 服务” 所示。

例 13.8. 使用 SOAP/JMS 服务

```
// Java
public void invoke() throws Exception {
    String address = "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
        + "?jndiInitialContextFactory"
        + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
        + "&jndiConnectionFactoryName=ConnectionFactory&jndiURL=tcp://localhost:61500";
    JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
    factory.setAddress(address);
    factory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANSPORTID);
    factory.setServiceClass(Hello.class);
    Hello client = (Hello)factory.create();
    String reply = client.sayHi(" HI");
    System.out.println(reply);
}
```

例 13.8 “使用 SOAP/JMS 服务” 中的代码执行以下操作：

创建代表 t he 端点地址的 JMS URI。

实例化 JaxWsProxyFactoryBean 来创建代理。

使用服务的 JMS URI 设置 factory bean 的 address 字段。

指定工厂创建的代理将使用 SOAP/JMS 传输。

## 13.3. WSDL 扩展

概述

您可以通过将 WSDL 扩展元素插入到合同（在绑定范围、服务范围或端口范围内）来指定 JMS 传输的基本配置。WSDL 扩展允许您指定引导 JNDI InitialContext 的属性，然后可用于查找 JMS 目的地。您还可以设置影响 JMS 传输层行为的一些属性。

### SOAP/JMS 命名空间

SOAP/JMS WSDL 扩展在 http://www.w3.org/2010/soapjms/ 命名空间中定义。要在 WSDL 合同中使用它们，请在 wsdl:definitions 元素中添加以下设置：

```
<wsdl:definitions ...
    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
```

### WSDL 扩展元素

表 13.4 “SOAP/JMS WSDL 扩展元素” 显示可用于配置 JMS 传输的所有 WSDL 扩展元素。

**表 13.4. SOAP/JMS WSDL 扩展元素**

| 元素 | default | 描述 |
| --- | --- | --- |
| **soapjms:jndiInitialContextFactory** | | 指定 JNDI 提供程序的完全限定 Java 类名称。等同于设置 **java.naming.factory.initial** Java 系统属性。 |
| **soapjms:jndiURL** | | 指定初始化 JNDI 供应商的 URL。等同于设置 **java.naming.provider.url** Java 系统属性。 |
| **soapjms:jndiContextParameter** | | 指定用于创建 JNDI **InitialContext** 的额外属性。使用 **name** 和 **value** 属性来指定属性。 |
| **soapjms:jndiConnectionFactoryName** | | 指定 JMS 连接工厂的 JNDI 名称。 |

| 元素 | default | 描述 |
| --- | --- | --- |
| **soapjms:deliveryMode** | **持久性** | 指定是否使用 JMS **PERSISTENT** 或 **NON_PERSISTENT** 消息语义。对于 **PERSISTENT** 传输模式，JMS 代理会在确认它们之前将消息存储在持久存储中；而 **NON_PERSISTENT** 消息仅保存在内存中。 |
| **soapjms:replyToName** | | [可选] 指定队列消息的回复目的地。回复目的地会出现在 **JMSReplyTo** 标头中。建议为具有 request-reply 语义的应用程序设置此属性，因为如果未指定，则 JMS 提供程序将分配一个临时回复队列。<br><br>此属性的值根据 JMS URI 中指定的变体进行解释：<br><br>● **JNDI** 变体 - 由 JNDI 解析的目标队列的名称<br><br>● **队列** 变体 - 使用 JMS 解析的目标队列的名称 |
| **soapjms:priority** | **4** | 指定 JMS 消息优先级，范围从 0（最低）到 9（最高）。 |
| **soapjms:timeToLive** | **0** | JMS 提供程序将丢弃邮件的时间（以毫秒为单位）。值 **0** 代表一个无限生命周期。 |

**配置范围**

**WSDL 合同中的 WSDL 元素将影响到合同中定义的端点配置更改的范围。SOAP/JMS WSDL 元素可以置于 wsdl:binding 元素、wsdl:service 元素或 wsdl:port 元素的子项。SOAP/JMS 元素的父级决定将配置放入以下哪些范围：**

绑定范围

**您可以通过将扩展元素放在 wsdl: *binding* 元素内，在绑定范围内 配置 JMS 传输。此范围内的元素定义使用此绑定的所有端点的默认配置。绑定范围内的任何设置都可以在服务范围或端口范围内覆盖。**

服务范围

**您可以通过将扩展元素放在 wsdl: *service* 元素内，在服务范围内 配置 JMS 传输。此范围内的**

元素定义此服务中所有端点的默认配置。服务范围内的任何设置都可以在端口范围内覆盖。

端口范围

您可以通过将扩展元素放在 **wsdl:*port*** 元素内，在端口范围内 配置 JMS 传输。端口范围内的元素定义此端口的配置。它们覆盖在服务范围或绑定范围中定义的相同扩展元素的默认值。

**Example**

例 13.9 “使用 SOAP/JMS 配置的 WSDL 合同” 显示 SOAP/JMS 服务的 WSDL 合同。它在绑定范围内配置 JNDI 层、服务范围内的消息交付详细信息，以及端口范围内的回复目的地。

**例 13.9. 使用 SOAP/JMS 配置的 WSDL 合同**

```
<wsdl:definitions ...
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
 ... >
 ...
 <wsdl:binding name="JMSGreeterPortBinding" type="tns:JMSGreeterPortType">
  ...
  <soapjms:jndiInitialContextFactory>
   org.apache.activemq.jndi.ActiveMQInitialContextFactory
  </soapjms:jndiInitialContextFactory>
  <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
  <soapjms:jndiConnectionFactoryName>
   ConnectionFactory
  </soapjms:jndiConnectionFactoryName>
  ...
 </wsdl:binding>
 ...
 <wsdl:service name="JMSGreeterService">
  ...
  <soapjms:deliveryMode>NON_PERSISTENT</soapjms:deliveryMode>
  <soapjms:timeToLive>60000</soapjms:timeToLive>
  ...
  <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
   <soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
   <soapjms:replyToName>
    dynamicQueues/greeterReply.queue
   </soapjms:replyToName>
   ...
  </wsdl:port>
  ...
 </wsdl:service>
 ...
</wsdl:definitions>
```

例 13.9 “使用 SOAP/JMS 配置的 WSDL 合同” 中的 WSDL 执行以下操作：

为 SOAP/JMS 扩展声明命名空间。

在绑定范围内配置 JNDI 连接。

将 JMS 交付样式设置为非持久性，每个消息都持续一分钟。

指定目标目的地。

配置 JMS 传输，以便在 greeterReply.queue 队列中发送回复消息。

# 第 14 章 使用通用 JMS

**摘要**

Apache CXF 提供 JMS 传输的通用实现。通用 JMS 传输不仅限于使用 SOAP 消息，并允许连接到使用 JMS 的任何应用。

注意 ： CXF 3.0 中删除了对 JMS 1.0.2 API 的支持。如果您使用红帽JBoss Fuse 6.2 或更高版本（包括 CXF 3.0），您的 JMS 提供程序必须支持 JMS 1.1 API。

## 14.1. 配置 JMS 的方法

Apache CXF 通用 JMS 传输可以连接到任何 JMS 提供程序，并与 TextMessage 或 ByteMessage 的 bodies 交换 JMS 消息的应用程序进行交换。

启用和配置 JMS 传输的方法有两种：

- 第 14.2 节 “使用 JMS 配置 bean”
- 第 14.5 节 “使用 WSDL 配置 JMS”

## 14.2. 使用 JMS 配置 BEAN

**概述**

为简化 JMS 配置并使其更加强大，Apache CXF 使用单个 JMS 配置 bean 来配置 JMS 端点。bean 由 org.apache.cxf.transport.jms.JMSConfiguration 类实施。它可用于直接配置端点，或配置 JMS conduits 和目的地。

**配置命名空间**

JMS 配置 bean 使用 Spring p-namespace 使配置尽可能简单。要使用这个命名空间，您需要在配置的根元素中声明它，如 例 14.1 “声明 Spring p-namespace” 所示。

**例 14.1. 声明 Spring p-namespace**

```
<beans ...
  xmlns:p="http://www.springframework.org/schema/p"
  ... >
  ...
</beans>
```

**指定配置**

**您可以通过定义类 org.apache.cxf.transport.jms.JMSConfiguration 的 bean 来指定 JMS 配置。bean 的属性提供传输的配置设置。**

**重要**

**在 CXF 3.0 中，JMS 传输不再依赖于 Spring JMS，因此删除了一些与 Spring JMS 相关的选项。**

**表 14.1 “常规 JMS 配置属性” 列出提供程序和消费者通用的属性。**

**表 14.1. 常规 JMS 配置属性**

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **connectionFactory** | | [required] 指定定义 JMS ConnectionFactory 的 bean 的引用。 |
| **wrapInSingleConnectionFactory** | **true** [pre v3.0] | **在 CXF 3.0 中删除**<br><br>pre CXF 3.0 指定是否使用 Spring **SingleConnectionFactory** 包装 ConnectionFactory。<br><br>在使用不池连接的 ConnectionFactory 时启用此属性，因为它将提高 JMS 传输的性能。这是因为 JMS 传输为每个消息创建新连接，并且需要 **SingleConnectionFactory** 来缓存连接，因此可以重复使用它。 |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **reconnectOnException** | **false** | **在 CXF 3.0** CXF 中弃用，当出现异常时始终重新连接。<br><br>pre CXF 3.0 指定在发生异常时是否创建新连接。<br><br>当使用 Spring **SingleConnectionFactory** 嵌套 ConnectionFactory 时：<br><br>● **true** criu-wagonon 一个例外，创建一个新连接在使用 PooledConnectionFactory 时不要启用此选项，因为此选项仅返回池连接，但不会重新连接。<br><br>● 异常错误，不要尝试重新连接 |
| **targetDestination** | | 指定目的地的 JNDI 名称或特定于提供程序的名称。 |
| **replyDestination** | | 指定发送回复的 JMS 目的地的名称。此属性允许使用用户定义的目的地进行回复。详情请查看 第 14.6 节 “使用命名的 Reply Destination”。 |
| **destinationResolver** | DynamicDestinationResolver | 指定对 Spring **DestinationResolver** 的引用。<br><br>此属性允许您定义目的地名称如何解析到 JMS 目的地。有效值为：<br><br>● 使用 JMS 提供程序的功能，**DynamicDestinationResolver** 可以解析目的地名称。<br><br>● 使用 JNDI 的 **JndiDestinationResolver** criu- iwlresolve 目的地名称。 |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **transactionManager** | | 指定对 Spring 事务管理器的引用。这可让服务参与 JTA 事务。 |
| **taskExecutor** | **SimpleAsyncTaskExecutor** | **在 CXF 3.0 中删除**<br><br>**pre CXF 3.0** 指定对 Spring TaskExecutor 的引用。这用于监听程序来决定如何处理传入的信息。 |
| **useJms11** | **false** | **在 CXF 3.0 CXF** 3.0 中删除，仅支持 JMS 1.1 功能。<br><br>**pre CXF 3.0** 指定是否使用 JMS 1.1 功能。有效值为：<br><br>● **true** criu-wagonJMS 1.1 功能<br><br>● **false** criu- iwlJMS 1.0.2 功能 |
| **messageIdEnabled** | **true** | **在 CXF 3.0 中删除**<br><br>**pre CXF 3.0** 指定 JMS 传输是否希望 JMS 代理提供消息 ID。有效值为：<br><br>● **true** criu-wagonbroker 需要提供消息 ID<br><br>● **false** criu-wagonbroker 不需要提供消息 ID 在本例中，端点调用其消息制作者的 **setDisableMessageID（）** 方法，值设为 **true**。然后，代理会给出一个提示，它不需要生成消息 ID，或将它们添加到端点的消息中。代理可以接受 hint 或忽略它。 |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **messageTimestampEnabled** | **true** | **在 CXF 3.0 中删除**<br><br>pre CXF 3.0 指定 JMS 传输是否希望 JMS 代理提供消息时间戳。有效值为：<br><br>● **true** criu-criubroker 需要提供消息时间戳<br>● **false** criu-wagonbroker 不需要提供消息时间戳 在本例中，端点调用其消息制作者的 **setDisableMessageTimestamp （）** 方法，值设为 **true**。然后，代理会给出一个提示，它不需要生成时间戳，或将它们添加到端点的消息中。代理可以接受 hint 或忽略它。 |
| **cacheLevel** | **-1** （禁用功能） | **在 CXF 3.0 中删除**<br><br>pre CXF 3.0 指定 JMS 侦听程序容器可能应用的缓存级别。有效值为：<br><br>● **0** – CACHE_NONE<br>● **1** CRIU-WAGONCACHE_CONNECTION<br>● **2** CRIU-WAGONCACHE_SESSION<br>● **3** CRIU-WAGONCACHE_CONSUMER<br>● **4** CRIU-WAGONCACHE_AUTO<br><br>详情请参阅 类 DefaultMessageListenerContainer |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **pubSubNoLocal** | **false** | 指定在使用主题时是否接收您自己的消息。<br>● **true** criu-wagondo 没有接收您自己的消息<br>● **false** criu-wagonreceive your own messages |
| **receiveTimeout** | **60000** | 指定响应消息的时间（以毫秒为单位）。 |
| **explicitQosEnabled** | **false** | 指定 QoS 设置（如优先级、持久性、生存时间）是否明确为每个消息设置(**true**)或者使用默认值(**false**)。 |
| **deliveryMode** | **2** | 指定消息是否持久。有效值为：<br>● **1** (NON_PERSISTENT)-messages 仅保留内存<br>● **2** (PERSISTENT)-messages 持久保留到磁盘 |
| **priority** | **4** | 指定消息优先级。JMS 优先级值范围从 **0 (** 最低)到 **9** （最高）。详情请查看您的 JMS 供应商文档。 |
| **timeToLive** | **0 (** 通常) | 指定发送的消息前的时间（以毫秒为单位）。 |
| **sessionTransacted** | **false** | 指定是否使用 JMS 事务。 |
| **concurrentConsumers** | **1** | **在 CXF 3.0 中删除**<br>pre CXF 3.0 指定监听器的最小并发用户数。 |
| **maxConcurrentConsumers** | **1** | **在 CXF 3.0 中删除**<br>pre CXF 3.0 指定监听器的最大并发用户数。 |
| **messageSelector** | | 指定用于过滤传入消息的选择器的字符串值。此属性使多个连接能够共享一个队列。有关用于指定消息选择器的语法的更多信息，请参阅 JMS 1.1 规范。 |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **subscriptionDurable** | **false** | 指定服务器是否使用持久订阅。 |
| **durableSubscriptionName** | | 指定用于注册持久订阅的名称（字符串）。 |
| **messageType** | **text** | 指定消息数据如何打包为 JMS 消息。有效值为：<br>• **文本** ProductShortName-wagonspec 表示数据将打包为 **TextMessage**<br>• **byte** criu-wagonspec 表示数据将打包为字节数组（**byte[]**）<br>• **二进制** criu-wagonspec 表示数据将打包为 **ByteMessage** |
| **pubSubDomain** | **false** | 指定目标目的地是主题还是队列。有效值为：<br>• **true** criu-categoriestopic<br>• **false** wagon-PROFILEqueue |
| **jmsProviderTibcoEms** | **false** | 指定 JMS 提供程序是否为 Tibco EMS。<br>当设置为 **true** 时，安全上下文中的主体会从 **JMS_TIBCO_SENDER** 标头填充。 |
| **useMessageIDAsCorrelationID** | **false** | **在 CXF 3.0 中删除**<br>指定 JMS 是否将使用消息 ID 来关联消息。<br>当设置为 **true** 时，客户端会设置生成的关联 ID。 |
| **maxSuspendedContinuations** | **-1** （禁用功能） | CXF 3.0 指定 JMS 目的地可能具有的最大暂停暂停次数。当当前数量超过指定的最大值时，JMSListenerContainer 会停止。 |

| 属性 | 默认 | 描述 |
| --- | --- | --- |
| **reconnectPercentOfMax** | **70** | CXF 3.0 指定何时重启 JMSListenerContainer 停止以超过 **maxSuspendedContinuations**。<br><br>当当前暂停的延续的数量低于 **(maxSuspendedContinuations * reconnectPercentOfMax/100)** 时，侦听器容器会被重启。 |

如 例 14.2 “JMS 配置 bean” 所示，bean 的属性被指定为 bean 元素的属性。它们都在 Spring p 命名空间中声明。

**例 14.2. JMS 配置 bean**

```
<bean id="jmsConfig"
      class="org.apache.cxf.transport.jms.JMSConfiguration"
      p:connectionFactory="jmsConnectionFactory"
      p:targetDestination="dynamicQueues/greeter.request.queue"
      p:pubSubDomain="false" />
```

**将配置应用到端点**

JMSConfiguration bean 可以使用 Apache CXF 功能机制直接应用到服务器和客户端端点。要做到这一点：

1. 将端点的 address 属性设置为 jms://。
2. 在端点配置中添加 jaxws:feature 元素。
3. 向该功能中添加类型为 org.apache.cxf.transport.jms.JMSConfigFeature 的 bean。
4. 将 bean 元素的 p:jmsConfig-ref 属性设置为 JMSConfiguration bean 的 ID。

例 14.3 “将 JMS 配置添加到 JAX-WS 客户端” 显示使用 例 14.2 “JMS 配置 bean” 中的 JMS 配置的

JAX-WS 客户端。

**例 14.3. 将 JMS 配置添加到 JAX-WS 客户端**

```
<jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              serviceName="customer:CustomerServiceService"
              endpointName="customer:CustomerServiceEndpoint"
              address="jms://"
              serviceClass="com.example.customerservice.CustomerService">
  <jaxws:features>
    <bean xmlns="http://www.springframework.org/schema/beans"
          class="org.apache.cxf.transport.jms.JMSConfigFeature"
          p:jmsConfig-ref="jmsConfig"/>
  </jaxws:features>
</jaxws:client>
```

### 将配置应用到传输中

JMSConfiguration bean 可以利用 jms:jmsConfig-ref 元素应用到 JMS conduits 和 JMS 目的地。jms:jmsConfig-ref element's 的值是 JMSConfiguration bean 的 ID。

例 14.4 “将 JMS 配置添加到 JMS conduit” 显示使用 例 14.2 “JMS 配置 bean” 中的 JMS 配置的 JMS。

**例 14.4. 将 JMS 配置添加到 JMS conduit**

```
<jms:conduit name="{http://cxf.apache.org/jms_conf_test}HelloWorldQueueBinMsgPort.jms-
conduit">
  ...
  <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

## 14.3. 优化客户端 JMS 性能

### 概述

两个主要设置会影响客户端的 JMS 性能：池和同步接收。

### 池

在客户端上，Fcxf 会为每个消息创建一个新的 JMS 会话和 JMS producer。因此，会话和制作者对

象都不是线程安全。创建制作者特别密集型，因为它需要与服务器通信。

池连接工厂通过缓存连接、会话和制作者来提高性能。

对于 ActiveMQ，配置池很简单，例如：

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.pool.PooledConnectionFactory;

ConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
PooledConnectionFactory pcf = new PooledConnectionFactory();

//Set expiry timeout because the default (0) prevents reconnection on failure
pcf.setExpiryTimeout(5000);
pcf.setConnectionFactory(cf);

JMSConfiguration jmsConfig = new JMSConfiguration();

jmsConfig.setConnectionFactory(pdf);
```

有关池的更多信息，请参阅 Red Hat JBoss Fuse Transaction Guide中的 "Appendix A Optimizing Performance of JMS Single-Resource Transactions"

**避免同步接收**

对于请求/回复交换，JMS 传输会发送请求，然后等待回复。在可能的情况下，使用 JMS MessageListener 异步实施请求/回复消息传递。

但是，当 CXF 需要在端点之间共享队列时，使用同步的 Consumer.receive （） 方法。此场景需要 MessageListener 来使用消息选择器来过滤消息。消息选择器必须提前知道，因此 MessageListener 只打开一次。

应避免提前知道消息选择器的两个情况：

- 当 JMSMessageID 用作 JMSCorrelationID时

  如果 JMS 属性 useConduitIdSelector 和 conduitSelectorPrefix 没有在 JMS 传输上设置，则客户端不会设置 JMSCorrelationId。这会导致服务器将请求消息的 JMSMessageId 用作

JMSCorrelationId。因为无法提前知道 JMSMessageID，客户端必须使用同步的 Consumer.receive （） 方法。

请注意，您必须将 Consumer.receive （） 方法与 IBM JMS 端点（默认）一起使用。

- 用户在请求消息中设置 JMStype，然后设置自定义 JMSCorrelationID。

同样，因为不能提前知道自定义 JMSCorrelationID，客户端必须使用同步的 Consumer.receive （） 方法。

因此，常规规则是避免使用需要使用同步接收的设置。

## 14.4. 配置 JMS 事务

### 概述

CXF 3.0 在使用单向消息传递时支持 CXF 端点上的本地 JMS 事务和 JTA 事务。

### 本地事务

只有在发生异常时，使用本地资源进行事务才会回滚 JMS 消息。它们不直接协调其他资源，如数据库事务。

要设置本地事务，请像通常一样配置端点，并将属性 sessionTrasnsacted 设置为 true。

注意

有关事务和池的更多信息，请参阅 Red Hat JBoss Fuse Transaction Guide。

### JTA 事务

使用 JTA 事务，您可以协调任意数量的 XA 资源。如果为 JTA 事务配置了 CXF 端点，它会在调用服务实现前启动事务。如果没有发生异常，则会提交事务。否则，它将回滚。

在 JTA 事务中，会消耗一个 JMS 消息以及写入数据库的数据。发生异常时，两个资源都会回滚，因

此消耗消息以及数据写入数据库，或者消息将被回滚，数据不会写入数据库。

配置 JTA 事务需要两个步骤：

1. 定义事务管理器

- Bean 方法
  - 定义事务管理器

```
<bean id="transactionManager"
class="org.apache.geronimo.transaction.manager.GeronimoTransactionManager"/>
```

  - 在 JMS URI 中设置事务管理器的名称

```
jms:queue:myqueue?jndiTransactionManager=TransactionManager
```

    本例找到 ID TransactionManager 的 bean。

- OSGi 参考方法
  - 使用 Blueprint 将事务管理器查找为 OSGi 服务

```
<reference id="TransactionManager"
interface="javax.transaction.TransactionManager"/>
```

  - 在 JMS URI 中设置事务管理器的名称

```
jms:jndi:myqueue?jndiTransactionManager=java:comp/env/TransactionManager
```

    本例在 JNDI 中查找事务管理器。

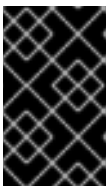
2. **配置 JCA 池的连接工厂**

**使用 Spring 定义 JCA 池的连接工厂：**

```
<bean id="xacf" class="org.apache.activemq.ActiveMQXAConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="ConnectionFactory"
class="org.apache.activemq.jms.pool.JcaPooledConnectionFactory">
  <property name="transactionManager" ref="transactionManager" />
  <property name="connectionFactory" ref="xacf" />
</bean>
```

**在本例中，第一个 bean 定义了一个 ActiveMQ XA 连接工厂，它提供给 JcaPooledConnectionFactory。然后，JcaPooledConnectionFactory 作为默认的 bean 提供，其 id 为 ConnectionFactory。**

**请注意，JcaPooledConnectionFactory 类似于普通的 ConnectionFactory。但是，当打开新连接和会话时，它会检查 XA 事务，如果找到，则自动将 JMS 会话注册为 XA 资源。这使得 JMS 会话能够参与 JMS 事务。**



**重要**

**直接在 JMS 传输上设置 XA ConnectionFactory 将无法正常工作！**

## 14.5. 使用 WSDL 配置 JMS

### 14.5.1. JMS WSDL 扩展名称spance

**用于定义 JMS 端点的 WSDL 扩展在命名空间 http://cxf.apache.org/transports/jms 中定义。要使用 JMS 扩展，您需要将 例 14.5 “JMS WSDL 扩展命名空间” 中显示的行添加到您的合同的 definitions 元素中。**

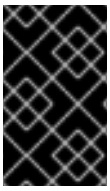
**例 14.5. JMS WSDL 扩展命名空间**

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

### 14.5.2. 基本 JMS 配置

**概述**

**JMS 地址信息使用 jms:address 元素及其子级提供，jms:JMSNamingProperties 元素。jms:address 元素的属性指定识别 JMS 代理和目的地所需的信息。jms:JMSNamingProperties 元素指定用于连接 JNDI 服务的 Java 属性。**



**重要**

**使用 JMS 功能指定的信息将覆盖端点的 WSDL 文件中的信息。**

**指定 JMS 地址**

**JMS 端点的基本配置通过将 jms:address 元素用作 服务端口 元素的子级来完成。WSDL 中使用的 jms:address 元素与配置文件中使用的 jms:address 元素相同。其属性列在 表 14.2 “JMS 端点属性” 中。**

**表 14.2. JMS 端点属性**

| 属性 | 描述 |
| --- | --- |
| **destinationStyle** | 指定 JMS 目标是否为 JMS 队列或 JMS 主题。 |
| **jndiConnectionFactoryName** | 指定连接到 JMS 目的地时要使用的 JMS 连接工厂的 JNDI 名称。 |
| **jmsDestinationName** | 指定发送请求的 JMS 目的地的 JMS 名称。 |
| **jmsReplyDestinationName** | 指定发送回复的 JMS 目的地的名称。此属性允许您使用定义的用户进行回复的目的地。详情请查看 第 14.6 节 “使用命名的 Reply Destination”。 |
| **jndiDestinationName** | 指定绑定到将请求发送到的 JMS 目的地的 JNDI 名称。 |
| **jndiReplyDestinationName** | 指定发送到回复的 JMS 目的地的 JNDI 名称。此属性允许您使用定义的用户进行回复的目的地。详情请查看 第 14.6 节 “使用命名的 Reply Destination”。 |
| **connectionUserName** | 指定连接到 JMS 代理时使用的用户名。 |
| **connectionPassword** | 指定连接到 JMS 代理时要使用的密码。 |

**jms:address WSDL 元素使用 jms:JMSNamingProperties 子元素来指定连接到 JNDI 提供程序所需的其他信息。**

**指定 JNDI 属性**

为提高与 JMS 和 JNDI 提供程序的互操作性，jms:address 元素具有子元素 jms:JMSNamingProperties，允许您指定用于填充连接到 JNDI 提供程序时使用的属性的值。jms:JMSNamingProperties 元素有两个属性： name 和 value。name 指定要设置的属性的名称。value 属性指定指定属性的值。jms:JMSNamingProperties 元素也可用于指定提供程序特定属性。

以下是可设置的常用 JNDI 属性列表：

1. **java.naming.factory.initial**
2. **java.naming.provider.url**
3. **java.naming.factory.object**
4. **java.naming.factory.state**
5. **java.naming.factory.url.pkgs**
6. **java.naming.dns.url**
7. **java.naming.authoritative**
8. **java.naming.batchsize**
9. **java.naming.referral**
10. **java.naming.security.protocol**

11. **java.naming.security.authentication**

12. **java.naming.security.principal**

13. **java.naming.security.credentials**

14. **java.naming.language**

15. **java.naming.applet**

**有关这些属性中要使用的信息的更多详细信息，请检查您的 JNDI 提供程序文档，并查阅 Java API 参考材料。**

**Example**

**例 14.6 “JMS WSDL 端口规格”** 显示 **JMS WSDL 端口** 规范的示例。

**例 14.6. JMS WSDL 端口规格**

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

### 14.5.3. JMS 客户端配置

**概述**

**JMS 消费者端点指定它们使用的消息类型。JMS 消费者端点可以使用 JMS ByteMessage 或 JMS TextMessage。**

使用 ByteMessage 时，消费者端点使用 byte[] 作为将数据存储到 JMS 消息正文并检索数据的方法。发送消息后，消息数据（包括任何格式信息）被打包成一个 字节[]，并在其放置在线上之前将其放入消息正文中。收到消息后，消费者端点将尝试 unmarshall 消息正文中存储的数据，就如同将其打包为 byte[] 一样。

使用 TextMessage 时，消费者端点使用字符串作为从消息正文存储和检索数据的方法。发送消息后，消息信息（包括任何格式特定信息）将转换为字符串，并放入 JMS 消息正文中。收到消息时，消费者端点将尝试 unmarshall 存储在 JMS 消息正文中的数据，就如同将其打包成字符串一样。

当原生 JMS 应用程序与 Apache CXF 用户交互时，JMS 应用程序负责解释消息和格式信息。例如，如果 Apache CXF 合同指定用于 JMS 端点的绑定是 SOAP，并且消息被打包为 TextMessage，则接收 JMS 应用会收到包含所有 SOAP envelope 信息的文本消息。

### 指定消息类型

JMS 消费者端点接受的消息类型使用可选的 jms:client 元素进行配置。jms:client 元素是 WSDL 端口 元素的子级，它有一个属性：

表 14.3. JMS Client WSDL 扩展

| messageType |
| --- |
| 指定消息数据如何打包为 JMS 消息。**text** 指定数据将打包为 **TextMessage**。**binary** 指定数据将打包为 **ByteMessage**。 |

### Example

例 14.7 “用于 JMS 使用者端点的 WSDL” 显示用于配置 JMS 使用者端点的 WSDL。

例 14.7. 用于 JMS 使用者端点的 WSDL

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:client messageType="binary" />
  </port>
</service>
```

### 14.5.4. JMS 供应商配置

**概述**

JMS 提供程序端点具有许多可以配置的行为。它们是：

- 消息关联方式
- 使用持久订阅
- 如果服务使用本地 JMS 事务
- 端点使用的消息选择器

**指定配置**

提供程序端点行为使用可选的 jms:server 元素进行配置。jms:server 元素是 WSDL wsdl:port 元素的子级，具有以下属性：

**表 14.4. JMS 供应商端点 WSDL 扩展**

| 属性 | 描述 |
| --- | --- |
| **useMessageIDAsCorrealationID** | 指定 JMS 是否将使用消息 ID 来关联消息。默认值为 **false**。 |
| **durableSubscriberName** | 指定用于注册持久订阅的名称。 |
| **messageSelector** | 指定要使用的消息选择器的字符串值。有关用于指定消息选择器的语法的更多信息，请参阅 JMS 1.1 规格。 |
| **事务** | 指定本地 JMS 代理是否将创建有关消息处理的事务。默认值为 **false**。 [a] |
| [a] 目前，运行时不支持将 **transactional** 属性设置为 **true**。 | |

**Example**

例 14.8 “用于 JMS 提供程序端点的 WSDL” 显示用于配置 JMS 提供程序端点的 WSDL。

**例 14.8. 用于 JMS 提供程序端点的 WSDL**

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:server messageSelector="cxf_message_selector"
                useMessageIDAsCorrelationID="true"
                transactional="true"
                durableSubscriberName="cxf_subscriber" />
  </port>
</service>
```

## 14.6. 使用命名的 REPLY DESTINATION

### 概述

默认情况下，使用 JMS 的 Apache CXF 端点会创建一个临时队列来回发送回复。如果您希望使用命名队列，您可以配置用于发送回复作为端点 JMS 配置一部分的队列。

### 设置回复目的地名称

您可以使用端点的 JMS 配置中的 jmsReplyDestinationName 属性或 jndiReplyDestinationName 属性来指定回复目的地。客户端端点将侦听指定目的地的回复，它将在所有传出请求的 ReplyTo 字段中指定属性值。服务端点将使用 jndiReplyDestinationName 属性的值，作为在请求的 ReplyTo 字段中指定任何目的地时放置回复的位置。

### Example

例 14.9 “使用命名的 Reply Queue 的 JMS 消费者规格” 显示 JMS 客户端端点的配置。

**例 14.9. 使用命名的 Reply Queue 的 JMS 消费者规格**

```
■
```

```
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
   <jms:address destinationStyle="queue"
                jndiConnectionFactoryName="myConnectionFactory"
                jndiDestinationName="myDestination"
                jndiReplyDestinationName="myReplyDestination" >
     <jms:JMSNamingProperty name="java.naming.factory.initial"
                            value="org.apache.cxf.transport.jms.MyInitialContextFactory" />
     <jms:JMSNamingProperty name="java.naming.provider.url"
                            value="tcp://localhost:61616" />
   </jms:address>
 </jms:conduit>
```

# 第 15 章 与 APACHE ACTIVEMQ 集成

## 概述

如果您使用 Apache ActiveMQ 作为 JMS 提供程序，则目的地的 JNDI 名称可以通过特殊格式来指定，为队列或主题动态创建 JNDI 绑定。这意味着，不要 提前使用您的队列或主题的 JNDI 绑定配置 JMS 提供程序。

## 初始上下文工厂

将 Apache ActiveMQ 与 JNDI 集成的关键是 ActiveMQInitialContextFactory 类。此类用于创建 JNDI InitialContext 实例，然后使用它来访问 JMS 代理中的 JMS 目的地。

例 15.1 “用于连接到 Apache ActiveMQ 的 SOAP/JMS WSDL” 显示 SOAP/JMS WSDL 扩展，以创建与 Apache ActiveMQ 集成的 JNDI InitialContext。

**例 15.1. 用于连接到 Apache ActiveMQ 的 SOAP/JMS WSDL**

```
<soapjms:jndiInitialContextFactory>
  org.apache.activemq.jndi.ActiveMQInitialContextFactory
</soapjms:jndiInitialContextFactory>
<soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
```

在 例 15.1 “用于连接到 Apache ActiveMQ 的 SOAP/JMS WSDL” 中，Apache ActiveMQ 客户端连接到位于 tcp://localhost:61616 的代理端口。

## 查找连接工厂

除了创建 JNDI InitialContext 实例外，您必须指定绑定到 javax.jms.ConnectionFactory 实例的 JNDI 名称。对于 Apache ActiveMQ，在 InitialContext 实例中有一个预定义的绑定，它将 JNDI 名称 ConnectionFactory 映射到 ActiveMQConnectionFactory 实例。例 15.2 “用于指定 Apache ActiveMQ 连接工厂的 SOAP/JMS WSDL” 填充用于指定 Apache ActiveMQ 连接工厂的 SOAP/JMS 扩展元素。

**例 15.2. 用于指定 Apache ActiveMQ 连接工厂的 SOAP/JMS WSDL**

```
<soapjms:jndiConnectionFactoryName>
  ConnectionFactory
</soapjms:jndiConnectionFactoryName>
```

动态目的地的语法

要动态访问队列或主题，请使用以下格式将目的地的 JNDI 复合名称指定为 JNDI 复合名称：

```
dynamicQueues/QueueName
dynamicTopics/TopicName
```

*queueName* 和 *TopicName* 是 Apache ActiveMQ 代理使用的名称。它们不是 抽象 JNDI 名称。

例 15.3 “带有动态创建的队列的 WSDL 端口规格” 显示使用动态创建的队列的 WSDL 端口。

**例 15.3. 带有动态创建的队列的 WSDL 端口规格**

```
<service name="JMSService">
  <port binding="tns:GreeterBinding" name="JMSPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/greeter.request.queue" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

当应用尝试打开 JMS 连接时，Apache ActiveMQ 将检查是否存在一个带有 JNDI 名称 greeter.request.queue 的队列。如果不存在，它将创建一个新的队列，并将它绑定到 JNDI 名称 greeter.request.queue。

# 第 16 章 CONDUITS

## 摘要

conduits 是用来实现出站连接的传输架构的低级部分。其行为和生命周期可能会影响系统性能和处理负载。

## 概述

conduits 管理 Apache CXF 运行时中的客户端或出站传输详情。它们负责打开端口、建立出站连接、发送消息并侦听应用和单一外部端点之间的任何响应。如果应用连接到多个端点，则每个端点都有一个 conduit 实例。

每个传输类型使用 Conduit 接口实现自己的行为。这可实现应用程序级别功能和传输之间的标准化接口。

通常，在配置客户端传输详情时，您只需要担心应用程序所使用的行为。运行时如何处理 conduits 的底层语义通常是开发人员需要注意的。

然而，在了解行为时，会很有用：

- 实现自定义传输
- 高级应用程序调整以管理有限资源

## CONDUIT 生命周期

conduits 由客户端实施对象管理。创建后，客户端实施对象的持续时间内有效。其生命周期为：

1. 创建客户端实现对象时，会获得对 ConduitSelector 对象的引用。
2. 当客户端需要发送消息时，请求来自 conduit 选择器的引用。

   如果消息用于新端点，则 conduit 选择器会创建一个新的 conduit，并将其传递给客户端实

现。否则，它会传递对目标端点的引用。

3. conduit 在需要时发送消息。

4. 当客户端实施对象被销毁时，与其关联的所有引用都会被销毁。

### CONDUIT WEIGHT

conduit 对象的权重取决于传输实现。HTTP 双重点是非常轻巧的权重。JMS conduits 非常重，因为它们与 JMS Session 对象和一个或多个 JMSListenerContainer 对象相关联。

# 部分 IV. 配置 WEB 服务端点

本指南论述了如何在 Red Hat Fuse 中创建 Apache CXF 端点。

# 第 17 章 配置 JAX-WS 端点

**摘要**

JAX-WS 端点使用以下三个 Spring 配置元素之一进行配置。正确的元素取决于您配置的端点类型以及您要使用的功能。对于消费者，您使用 jaxws:client 元素。对于服务提供商，您可以使用 jaxws:endpoint 元素或 jaxws:server 元素。

用于定义端点的信息通常在端点的合同中定义。您可以使用 configuration 元素的 覆盖合同信息。您还可以使用配置元素提供合同中未提供的信息。

您必须使用配置元素来激活 WS-RM 等高级功能。这可以通过向端点配置元素提供子元素来完成。请注意，当使用 Java 优先方法处理端点时，可能会因为端点的合同缺少有关要使用的绑定和传输类型的信息。

## 17.1. 配置服务提供商

### 17.1.1. 配置服务提供商的元素

Apache CXF 有两个可用于配置服务提供商的元素：

- 第 17.1.2 节 “使用 jaxws:endpoint Element”
- 第 17.1.3 节 “使用 jaxws:server Element”

两个元素之间的差别主要是运行时内部的。jaxws:endpoint 元素将属性注入为支持服务端点而创建的 org.apache.cxf.jaxws.EndpointImpl 对象。jaxws:server 元素将属性注入 org.apache.cxf.jaxws.support.JaxWsServerFactoryBean 对象，以便支持端点。EndpointImpl 对象将配置数据传递给 JaxWsServerFactoryBean 对象。JaxWsServerFactoryBean 对象用于创建实际的服务对象。由于任一配置元素将配置服务端点，因此您可以根据您喜欢的语法进行选择。

### 17.1.2. 使用 jaxws:endpoint Element

**概述**

jaxws:endpoint 元素是用于配置 JAX-WS 服务提供程序的默认元素。其属性和子项指定实例化服务提供商所需的所有信息。许多属性都映射到服务合同中的信息。子项用于配置拦截器和其他高级功能。

### 识别要配置的端点

要使运行时将配置应用到正确的服务提供商，它必须能够识别它。识别服务提供商的基本方法是指定实现端点的类。这可以通过 jaxws:endpoint 元素的 implementor 属性来完成。

对于不同端点共享一个共同实施的实例，可以为每个端点提供不同的配置。在配置中区分特定端点的方法有两种：

- serviceName 属性和 endpointName 属性的组合

  serviceName 属性指定定义服务端点的 wsdl:service 元素。endpointName 属性指定定义服务端点的特定 wsdl:port 元素。这两个属性都使用格式 *ns* 指定为 QNames：*名称*。*ns* 是元素的命名空间，*name* 是元素的 name 属性的值。

  **注意**

  如果 wsdl:service 元素只有一个 wsdl:port 元素，则可以省略 endpointName 属性。

- name 属性

  name 属性指定定义服务端点的特定 wsdl:port 元素的 QName。QName 以 {*ns*}*localPart*格式提供。*ns* 是 wsdl:port 元素的命名空间，*localPart* 是 wsdl:port 元素的 name 属性的值。

### 属性

jaxws:endpoint 元素的属性配置端点的基本属性。这些属性包括端点的地址、实施端点的类以及托管端点的 总线。

表 17.1 “使用 jaxws:endpoint Element 配置 JAX-WS 服务提供商的属性” 描述 jaxws:endpoint 元素的属性。

**表 17.1. 使用 jaxws:endpoint Element 配置 JAX-WS 服务提供商的属性**

| 属性 | 描述 |
| --- | --- |
| id | 指定其他配置元素可用于引用端点的唯一标识符。 |

| 属性 | 描述 |
| --- | --- |
| **implementationor** | 指定实施该服务的类。您可以使用类名称或对配置实施类的 Spring bean 的 ID 引用来指定实施类。此类必须位于 classpath 上。 |
| **implementorClass** | 指定实施该服务的类。当提供给 **implementor** 属性的值是使用 Spring AOP 打包的 bean 的引用时，此属性很有用。 |
| **address** | 指定 HTTP 端点的地址。这个值会覆盖服务合同中指定的值。 |
| **wsdlLocation** | 指定端点的 WSDL 合同的位置。WSDL 合同的位置相对于部署该服务的文件夹。 |
| **endpointName** | 指定服务的 **wsdl:port** 元素的 **name** 属性的值。它使用格式 ***ns*:*name*** 将它指定为 QName，其中 *ns* 是 **wsdl:port** 元素的命名空间。 |
| **serviceName** | 指定服务的 **wsdl:service** 元素的 **name** 属性的值。它使用格式 ***ns*:*name*** 将它指定为 QName，其中 *ns* 是 **wsdl:service** 元素的命名空间。 |
| **publish** | 指定是否应自动发布该服务。如果将其设置为 **false**，则开发人员必须明确发布 第 31 章 *发布服务* 中描述的端点。 |
| **bus** | 指定配置用于管理服务端点总线的 Spring bean 的 ID。这在将多个端点配置为使用一组常用功能时很有用。 |
| **bindingUri** | 指定服务使用的消息绑定的 ID。第 23 章 *Apache CXF 绑定 ID* 中提供了有效绑定 ID 列表。 |
| **name** | 指定服务的 **wsdl:port** 元素的字符串指定的 QName。它使用 ***{ns}localPart*** 格式将其指定为 QName。*ns* 是 **wsdl:port** 元素的命名空间，*localPart* 是 **wsdl:port** 元素的 **name** 属性的值。 |
| **abstract** | 指定 bean 是否为抽象 Bean。抽象 Bean 充当 concrete bean 定义的父项，且不会实例化。默认值为 **false**。把它设置为 **true** 会指示 bean 工厂不会实例化 bean。 |
| **dependent-on** | 指定在可以实例化前端点要实例化的 Bean 列表。 |

| 属性 | 描述 |
| --- | --- |
| **createdFromAPI** | 指定使用 Apache CXF API 创建的 bean 用户，如 **Endpoint.publish （）** 或 **Service.getPort （）**。<br><br>默认值为 **false**。<br><br>把它设置为 **true** 执行以下操作：<br><br>● 通过将 **.jaxws-endpoint** 附加到其 id 来更改 bean 的内部名称<br><br>● 使 bean 抽象 |
| **publishedEndpointUrl** | 放置在生成的 WSDL 的 **address** 元素中的 URL。如果没有指定这个值，则使用 **address** 属性的值。当"public" URL 与部署该服务的 URL 不同时，此属性很有用。 |

**除了 表 17.1 “使用 jaxws:endpoint Element 配置 JAX-WS 服务提供商的属性” 中列出的属性外，您可能需要使用多个 xmlns:*shortName* 属性来声明 endpointName 和 serviceName 属性使用的命名空间。**

### Example

**例 17.1 “简单的 JAX-WS 端点配置” 显示 JAX-WS 端点的配置，用于指定发布端点的地址。示例假定您要将默认值用于所有其他值，或者实现已在注解中指定了值。**

**例 17.1. 简单的 JAX-WS 端点配置**

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="example"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo" />
</beans>
```

**例 17.2 “使用服务名称进行 JAX-WS 端点配置” 显示 JAX-WS 端点的配置，其合同包含两个服务定义。在这种情况下，您必须使用 serviceName 属性指定要实例化哪个服务定义。**

**例 17.2. 使用服务名称进行 JAX-WS 端点配置**

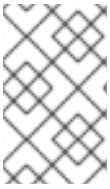
```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">

  <jaxws:endpoint id="example2"
                  implementor="org.apache.cxf.example.DemoImpl"
                  serviceName="samp:demoService2"
                  xmlns:samp="http://org.apache.cxf/wsdl/example" />

</beans>
```

**xmlns:samp 属性指定定义 WSDL 服务 元素的命名空间。**

**例 17.3 “启用 HTTP/2 的 JAX-WS 端点配置” 显示 JAX-WS 端点的配置，该端点指定了启用了 HTTP/2 的地址。**

为 **Apache CXF 配置 HTTP/2**

**在 Apache Karaf 上使用独立 Apache CXF Undertow 传输(http-undertow)时，支持 HTTP/2。要启用 HTTP/2 协议，您必须将 jaxws:endpoint 元素的 address 属性设置为绝对 URL，并将 org.apache.cxf.transports.http_undertow.EnableHttp2 属性设置为 true。**



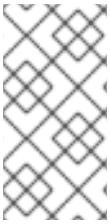
**注意**

**此 HTTP/2 实现只支持使用普通 HTTP 或 HTTPS 的服务器端 HTTP/2 传输。**

**例 17.3. 启用 HTTP/2 的 JAX-WS 端点配置**

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
  http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
  ...">

  <cxf:bus>
    <cxf:properties>
       <entry key="org.apache.cxf.transports.http_undertow.EnableHttp2" value="true"/>
    </cxf:properties>
  </cxf:bus>
```

```
<jaxws:endpoint id="example3"
        implementor="org.apache.cxf.example.DemoImpl"
        address="http://localhost:8080/demo" />
</jaxws:endpoint>

</beans>
```



**注意**

为提高性能，红帽建议在 Apache Karaf (pax-web-undertow)上使用 servlet 传输，它启用了集中配置和调整 Web 容器，但 pax-web-undertow 不支持 HTTP/2 传输协议。

### 17.1.3. 使用 jaxws:server Element

**概述**

**jaxws:server 元素是用于配置 JAX-WS 服务提供程序的元素。它将配置信息注入到 org.apache.cxf.jaxws.support.JaxWsServerFactoryBean。这是 Apache CXF 特定的对象。如果您使用纯 Spring 方法来构建服务，则不会强制使用 Apache CXF 特定 API 与服务交互。**

**jaxws:server 元素的属性和子项指定实例化服务提供商所需的所有信息。属性指定实例化端点所需的信息。子项用于配置拦截器和其他高级功能。**

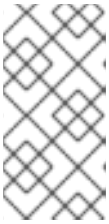
**识别要配置的端点**

**为了让运行时将配置应用到正确的服务提供商，它必须能够识别它。识别服务提供商的基本方法是指定实现端点的类。这使用 jaxws:server 元素的 serviceBean 属性完成。**

**对于不同端点共享一个共同实施的实例，可以为每个端点提供不同的配置。在配置中区分特定端点的方法有两种：**

- **serviceName 属性和 endpointName 属性的组合**

  **serviceName 属性指定定义服务端点的 wsdl:service 元素。endpointName 属性指定定义服务端点的特定 wsdl:port 元素。这两个属性都使用格式 *ns* 指定为 QNames：*名称*。*ns* 是元素的命名空间，*name* 是元素的 name 属性的值。**



**注意**

如果 wsdl:service 元素只有一个 wsdl:port 元素，则可以省略 endpointName 属性。

- name 属性

name 属性指定定义服务端点的特定 wsdl:port 元素的 QName。QName 以 {*ns*}*localPart*格式提供。*ns* 是 wsdl:port 元素的命名空间，*localPart* 是 wsdl:port 元素的 name 属性的值。

**属性**

jaxws:server 元素的属性配置端点的基本属性。这些属性包括端点的地址、实施端点的类以及托管端点的 总线。

表 17.2 “使用 jaxws:server 元素配置 JAX-WS 服务提供商的属性” 描述 jaxws:server 元素的属性。

**表 17.2. 使用 jaxws:server 元素配置 JAX-WS 服务提供商的属性**

| 属性 | 描述 |
| --- | --- |
| **id** | 指定其他配置元素可用于引用端点的唯一标识符。 |
| **serviceBean** | 指定实施该服务的类。您可以使用类名称或对配置实施类的 Spring bean 的 ID 引用来指定实施类。此类必须位于 classpath 上。 |
| **serviceClass** | 指定实施该服务的类。当提供给 **implementor** 属性的值是使用 Spring AOP 打包的 bean 的引用时，此属性很有用。 |
| **address** | 指定 HTTP 端点的地址。这个值将覆盖服务合同中指定的值。 |
| **wsdlLocation** | 指定端点的 WSDL 合同的位置。WSDL 合同的位置相对于部署该服务的文件夹。 |
| **endpointName** | 指定服务的 **wsdl:port** 元素的 **name** 属性的值。它使用格式 ***ns*:*name*** 格式将其指定为 QName，其中 *ns* 是 **wsdl:port** 元素的命名空间。 |

| 属性 | 描述 |
| --- | --- |
| **serviceName** | 指定服务的 **wsdl:service** 元素的 **name** 属性的值。它使用格式 ***ns:name*** 格式将其指定为 QName，其中 *ns* 是 **wsdl:service** 元素的命名空间。 |
| **publish** | 指定是否应自动发布该服务。如果将其设置为 **false**，则开发人员必须明确发布 第 31 章 *发布服务* 中描述的端点。 |
| **bus** | 指定配置用于管理服务端点总线的 Spring bean 的 ID。这在将多个端点配置为使用一组常用功能时很有用。 |
| **bindingId** | 指定服务使用的消息绑定的 ID。第 23 章 *Apache CXF 绑定 ID* 中提供了有效绑定 ID 列表。 |
| **name** | 指定服务的 **wsdl:port** 元素的字符串指定的 QName。它使用 ***{ns}localPart*** 格式将其指定为 QName，其中 *ns* 是 **wsdl:port** 元素的命名空间，*localPart* 是 **wsdl:port** 元素的 **name** 属性的值。 |
| **abstract** | 指定 bean 是否为抽象 Bean。抽象 Bean 充当 concrete bean 定义的父项，且不会实例化。默认值为 **false**。把它设置为 **true** 会指示 bean 工厂不会实例化 bean。 |
| **dependent-on** | 指定在实例化端点前，端点依赖实例化的 Bean 列表。 |
| **createdFromAPI** | 指定使用 Apache CXF API 创建的 bean 用户，如 **Endpoint.publish （）** 或 **Service.getPort （）**。<br><br>默认值为 **false**。<br><br>把它设置为 **true** 执行以下操作：<br><br>● 通过将 **.jaxws-endpoint** 附加到其 id 来更改 bean 的内部名称<br><br>● 使 bean 抽象 |

**除了 表 17.2 “使用 jaxws:server 元素配置 JAX-WS 服务提供商的属性” 中列出的属性外，您可能需要使用多个 xmlns:*shortName* 属性来声明 endpointName 和 serviceName 属性使用的命名空间。**

**Example**

例 17.4 “简单的 JAX-WS 服务器配置” 显示 JAX-WS 端点的配置，用于指定发布端点的地址。

**例 17.4. 简单的 JAX-WS 服务器配置**

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:server id="exampleServer"
                serviceBean="org.apache.cxf.example.DemoImpl"
                address="http://localhost:8080/demo" />
</beans>
```

### 17.1.4. 在服务提供商中添加功能

**概述**

jaxws:endpoint 和 jaxws:server 元素提供了实例化服务提供商所需的基本配置信息。要在服务供应商或执行高级配置中添加功能，您必须将子元素添加到配置中。

子元素允许您执行以下操作：

- 第 19 章 *Apache CXF Logging*
- 第 59 章 *配置端点以使用拦截器*
- 第 20 章 *部署 WS-Addressing*
- 第 21 章 *启用可靠的消息传递*
- 第 17.1.5 节 “在 JAX-WS 端点上启用架构验证”

**元素**

表 17.3 “用于配置 JAX-WS 服务供应商的元素” 描述 jaxws:endpoint 支持的子元素。

**表 17.3. 用于配置 JAX-WS 服务供应商的元素**

| 元素 | 描述 |
| --- | --- |
| **jaxws:handlers** | 指定用于处理消息的 JAX-WS 处理程序列表。有关 JAX-WS 处理程序实现的详情，请参考 第 43 章 *编写处理程序*。 |
| **jaxws:inInterceptors** | 指定处理入站请求的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:inFaultInterceptors** | 指定处理入站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:outInterceptors** | 指定处理出站回复的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:outFaultInterceptors** | 指定处理出站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:binding** | 指定 bean 配置端点使用的消息绑定。消息绑定使用 **org.apache.cxf.binding.BindingFactory** 接口的实现进行配置。[a] |
| **jaxws:dataBinding** [b] | 指定实施端点使用的数据绑定的类。这使用嵌入式 bean 定义来指定。 |
| **jaxws:executor** | 指定用于该服务的 Java executor。这使用嵌入式 bean 定义来指定。 |
| **jaxws:features** | 指定配置 Apache CXF 高级功能的 Bean 列表。您可以提供 bean 引用列表或嵌入式 Bean 列表。 |
| **jaxws:invoker** | 指定服务使用的 org.apache.cxf.service.Invoker 接口的实现。[c] |
| **jaxws:properties** | 指定与端点一起传递的属性的 Spring 映射。这些属性可用于控制诸如启用 MTOM 支持等功能。 |
| **jaxws:serviceFactory** | 指定 bean 配置用于实例化该服务的 **JaxWsServiceFactoryBean** 对象。 |

| 元素 | 描述 |
| --- | --- |

[a] SOAP 绑定使用 **soap:soapBinding** bean 配置。

[b] **jaxws:endpoint** 元素不支持 **jaxws:dataBinding** 元素。

[c] Invoker 实现控制如何调用服务。例如，它控制每个请求是否由服务实施的新实例处理，还是在调用之间保留状态。

### 17.1.5. 在 JAX-WS 端点上启用架构验证

**概述**

**您可以设置 schema-validation-enabled 属性，以便在 jaxws:endpoint 元素或 jaxws:server 元素上启用 schema 验证。启用 schema 验证后，在客户端和服务器间发送的消息会被检查是否符合 schema。默认情况下，架构验证会被关闭，因为它对性能有严重影响。**

**Example**

**要在 JAX-WS 端点上启用架构验证，请在 jaxws:properties 子元素或 jaxws:server 元素的 jaxws:properties 子元素中设置 schema-validation-enabled 属性。例如，要在 jaxws:endpoint 元素中启用模式验证：**

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
    wsdlLocation="wsdl/hello_world.wsdl"
    createdFromAPI="true">
    <jaxws:properties>
        <entry key="schema-validation-enabled" value="BOTH" />
    </jaxws:properties>
</jaxws:endpoint>
```

**有关 schema-validation-enabled 属性的允许值列表，请参阅 第 24.3.4.7 节 “模式验证类型值”。**

## 17.2. 配置消费者端点

**概述**

**JAX-WS 消费者端点使用 jaxws:client 元素进行配置。元素的属性提供创建消费者所需的基本信息。**

**将其他功能（如 WS-RM）添加到您向 jaxws:client 元素添加子项的消费者：子元素也用于配置端点的日志行为，并将其他属性注入端点的实施中。**

### 基本配置属性

表 17.4 “用于配置 JAX-WS Consumer 的属性” 中描述的属性提供了配置 JAX-WS 使用者所需的基本信息。您只需要为您要配置的特定属性提供值。大多数属性都有明显的默认值，或者它们依赖于端点合同提供的信息。

**表 17.4. 用于配置 JAX-WS Consumer 的属性**

| 属性 | 描述 |
|---|---|
| **address** | 指定消费者要发出请求的端点的 HTTP 地址。这个值会覆盖合同中设置的值。 |
| **bindingId** | 指定消费者使用的消息绑定的 ID。第 23 章 *Apache CXF 绑定 ID* 中提供了有效绑定 ID 列表。 |
| **bus** | 指定配置管理端点总线的 Spring bean 的 ID。 |
| **endpointName** | 为消费者发出请求的服务指定 **wsdl:port** 元素的 **name** 属性的值。它使用格式 ***ns*:*name*** 格式将其指定为 QName，其中 *ns* 是 **wsdl:port** 元素的命名空间。 |
| **serviceName** | 为消费者发出请求的服务指定 **wsdl:service** 元素的 **name** 属性的值。它使用格式 ***ns*:*name*** 将它指定为 QName，其中 *ns* 是 **wsdl:service** 元素的命名空间。 |
| **username** | 指定用于简单用户名/密码身份验证的用户名。 |
| **password** | 指定用于简单用户名/密码身份验证的密码。 |
| **serviceClass** | 指定服务端点接口(SEI)的名称。 |
| **wsdlLocation** | 指定端点的 WSDL 合同的位置。WSDL 合同的位置相对于部署客户端的文件夹。 |
| **name** | 为消费者发出请求的服务指定 **wsdl:port** 元素的字符串 QName。它使用 ***{ns}localPart*** 格式将其指定为 QName，其中 *ns* 是 **wsdl:port** 元素的命名空间，*localPart* 是 **wsdl:port** 元素的 **name** 属性的值。 |
| **abstract** | 指定 bean 是否为抽象 Bean。抽象 Bean 充当 concrete bean 定义的父项，且不会实例化。默认值为 **false**。把它设置为 **true** 会指示 bean 工厂不会实例化 bean。 |
| **dependent-on** | 指定在可以实例化前端点要实例化的 Bean 列表。 |

| 属性 | 描述 |
| --- | --- |
| **createdFromAPI** | 指定使用 Apache CXF API （如 **Service.getPort （）** ）创建的 bean 用户。<br><br>默认值为 **false**。<br><br>把它设置为 **true** 执行以下操作：<br><br>● 通过将 **.jaxws-client** 附加到其 id 来更改 bean 的内部名称<br><br>● 使 bean 抽象 |

除了 表 17.4 “用于配置 JAX-WS Consumer 的属性” 中列出的属性外，可能需要使用多个 **xmlns:*shortName*** 属性来声明 **endpointName** 和 **serviceName** 属性使用的命名空间。

### 添加功能

要向消费者添加功能或执行高级配置，您必须在配置中添加子元素。

子元素允许您执行以下操作：

- 第 19 章 *Apache CXF Logging*
- 第 59 章 *配置端点以使用拦截器*
- 第 20 章 *部署 WS-Addressing*
- 第 21 章 *启用可靠的消息传递*
- “在 JAX-WS 使用者上启用架构验证”一节

表 17.5 “用于配置消费者端点的元素” 描述可用于配置 JAX-WS 使用者的子元素。

**表 17.5. 用于配置消费者端点的元素**

| 元素 | 描述 |
| --- | --- |
| **jaxws:binding** | 指定 bean 配置端点使用的消息绑定。消息绑定使用 **org.apache.cxf.binding.BindingFactory** 接口的实现进行配置。[a] |
| **jaxws:dataBinding** | 指定实施端点使用的数据绑定的类。您可以使用嵌入式 bean 定义来指定它。实施 JAXB 数据绑定的类是 **org.apache.cxf.jaxb.JAXBDataBinding**。 |
| **jaxws:features** | 指定配置 Apache CXF 高级功能的 Bean 列表。您可以提供 bean 引用列表或嵌入式 Bean 列表。 |
| **jaxws:handlers** | 指定用于处理消息的 JAX-WS 处理程序列表。有关 JAX-WS 处理程序实现中的更多信息，请参阅 第 43 章 *编写处理程序*。 |
| **jaxws:inInterceptors** | 指定处理入站响应的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:inFaultInterceptors** | 指定处理入站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:outInterceptors** | 指定处理出站请求的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:outFaultInterceptors** | 指定处理出站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxws:properties** | 指定传递给端点的属性映射。 |
| **jaxws:conduitSelector** | 为客户端指定要使用的 org.apache.cxf.endpoint.ConduitSelector 实现。ConduitSelector 实现将覆盖用于处理出站请求的 **Conduit** 对象的默认进程。 |
| [a] SOAP 绑定使用 **soap:soapBinding** bean 配置。 | |

### Example

**例 17.5 “简单的消费者配置” 显示简单的消费者配置。**

**例 17.5. 简单的消费者配置**

```
<beans ...
```

```
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    ...
    schemaLocation="...
      http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
      ...">
    <jaxws:client id="bookClient"
                  serviceClass="org.apache.cxf.demo.BookClientImpl"
                  address="http://localhost:8080/books"/>
    ...
  </beans>
```

**在 JAX-WS 使用者上启用架构验证**

**要在 JAX-WS 消费者上启用架构验证，请在 jaxws:client 元素的 jaxws:properties 子元素中设置 schema-validation-enabled 属性，例如：**

```
<jaxws:client name="{http://apache.org/hello_world_soap_http}SoapPort"
    createdFromAPI="true">
    <jaxws:properties>
        <entry key="schema-validation-enabled" value="BOTH" />
    </jaxws:properties>
</jaxws:client>
```

**有关 schema-validation-enabled 属性的允许值列表，请参阅 第 24.3.4.7 节 “模式验证类型值”。**

# 第 18 章 配置 JAX-RS 端点

## 摘要

本章解释了如何在 Blueprint XML 和 Spring XML 中实例化和配置 JAX-RS 服务器端点，以及如何在 XML 中实例化和配置 JAX-RS 客户端端点（客户端代理 Bean）

## 18.1. 配置 JAX-RS 服务器端点

### 18.1.1. 定义 JAX-RS 服务器端点

#### 基本服务器端点定义

要在 XML 中定义 JAX-RS 服务器端点，您需要至少指定以下内容：

1. jaxrs:server 元素，用于在 XML 中定义端点。请注意，jaxrs: 命名空间前缀分别映射到 Blueprint 和 Spring 中的不同命名空间。

2. JAX-RS 服务的基本 URL，使用 jaxrs:server 元素的 address 属性。请注意，可以通过两种不同的方式指定地址 URL，这会影响端点的部署方式：

   - 作为一个相对 URL- 例如，/customers。在这种情况下，端点被部署到默认的 HTTP 容器中，通过将 CXF servlet 基础 URL 与指定的相对 URL 合并，来隐式获取端点的基本 URL。

     例如，如果您将 JAX-RS 端点部署到 Fuse 容器，指定的 /customers URL 将解析为 URL ，http://Hostname:8181/cxf/customers （假设容器使用默认的 8181 端口）。

   - 例如，作为绝对 URL mvapich- iwl 例如 http://0.0.0.0:8200/cxf/customers。在本例中，为 JAX-RS 端点打开了新的 HTTP 侦听器端口（如果尚未打开）。例如，在 Fuse 的上下文中，将隐式创建新的 Undertow 容器来托管 JAX-RS 端点。特殊的 IP 地址 0.0.0.0 充当通配符，匹配分配给当前主机的任何主机名（这对于多重主机机器很有用）。

3. 一个或多个 JAX-RS 根资源类，提供 JAX-RS 服务的实施。指定资源类的最简单方法是在 jaxrs:serviceBeans 元素中列出它们。

#### 蓝图示例

以下 Blueprint XML 示例演示了如何定义 JAX-RS 端点，该端点指定了相对地址 /customers （因此它部署到默认的 HTTP 容器中），并由 service.CustomerService 资源类实施：

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

    <cxf:bus>
        <cxf:features>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>

     <jaxrs:server id="customerService" address="/customers">
        <jaxrs:serviceBeans>
            <ref component-id="serviceBean" />
        </jaxrs:serviceBeans>
     </jaxrs:server>

     <bean id="serviceBean" class="service.CustomerService"/>
</blueprint>
```

蓝图 XML 命名空间

要在蓝图中定义 JAX-RS 端点，您通常至少需要以下 XML 命名空间：

| prefix | Namespace |
| --- | --- |
| （默认） | http://www.osgi.org/xmlns/blueprint/v1.0.0 |
| cxf | http://cxf.apache.org/blueprint/core |
| jaxrs | http://cxf.apache.org/blueprint/jaxrs |

Spring 示例

以下 Spring XML 示例演示了如何定义 JAX-RS 端点，该端点指定了相对地址 /customers （因此它部署到默认的 HTTP 容器中），并由 service.CustomerService 资源类实施：

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxrs="http://cxf.apache.org/jaxrs"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">

    <jaxrs:server id="customerService" address="/customers">
      <jaxrs:serviceBeans>
        <ref bean="serviceBean"/>
      </jaxrs:serviceBeans>
    </jaxrs:server>

    <bean id="serviceBean" class="service.CustomerService"/>
</beans>
```

**Spring XML 命名空间**

**要在 Spring 中定义 JAX-RS 端点，您通常至少需要以下 XML 命名空间：**

| prefix | Namespace |
| --- | --- |
| **（默认）** | **http://www.springframework.org/schema/beans** |
| **cxf** | **http://cxf.apache.org/core** |
| **jaxrs** | **http://cxf.apache.org/jaxrs** |

**Spring XML 中的自动发现**

**（仅Spring） 指定 JAX-RS 根资源类，Spring XML 允许您配置自动发现，以便搜索特定 Java 软件包以查找资源类（由 @Path注解的类），所有发现的资源类都会自动附加到端点。在这种情况下，您需要只在 jaxrs:server 元素中指定 address 属性和 basePackages 属性。**

**例如，要定义一个 JAX-RS 端点，它使用 a.b.c Java 软件包下的所有 JAX-RS 资源类，您可以在 Spring XML 中定义端点，如下所示：**

```
<jaxrs:server address="/customers" basePackages="a.b.c"/>
```

**自动发现机制还会发现并安装到端点，并安装到其在指定的 Java 软件包下找到的任何 JAX-RS 提供程序类。**

**Spring XML 中的生命周期管理**

（仅限Spring） Spring XML 使您能够通过设置 bean 元素上的 scope 属性来控制 Bean 的生命周期。Spring 支持以下范围值：

单例

（默认） 创建一个 bean 实例，该实例在 Spring 容器的整个生命周期中随处使用和最后一个。

prototype

每次 Bean 注入到另一个 bean 时，或通过调用 bean registry 上的 getBean （） 获取 bean 时，创建一个新的 bean 实例。

Request （请求）

（仅在 Web 感知容器中可用） 为 bean 上调用的每个请求创建一个新的 bean 实例。

会话

（只在 Web 感知容器中可用） 在单个 HTTP 会话的生命周期中创建一个新的 bean。

globalSession

（仅在 Web 感知容器中可用） 为 portlet 之间共享的单一 HTTP 会话的生命周期创建一个新的 bean。

有关 Spring 范围的更多详情，请参阅有关 Bean 范围的 Spring 框架文档。

请注意，如果您通过 jaxrs:serviceBeans 元素指定 JAX-RS 资源 Bean，则 Spring 范围 无法正常工作。如果您在此例中指定资源 Bean 上的 scope 属性，则有效忽略 scope 属性。

要使 bean 范围在 JAX-RS 服务器端点中正常工作，您需要一个由服务工厂提供的间接级别。配置 bean 范围的最简单方法是使用 jaxrs:server 元素上的 beanNames 属性来指定资源 Bean，如下所示：

```
<beans ... >
  <jaxrs:server id="customerService" address="/service1"
    beanNames="customerBean1 customerBean2"/>

  <bean id="customerBean1" class="demo.jaxrs.server.CustomerRootResource1"
scope="prototype"/>
  <bean id="customerBean2" class="demo.jaxrs.server.CustomerRootResource2"
scope="prototype"/>
</beans>
```

上例配置两个资源 Bean，即 customerBean1 和 customerBean2。beanNames 属性指定为以空格分隔的资源 bean ID 列表。

为获得理想的灵活性，您可以选择在配置 JAX-RS 服务器端点时，使用 jaxrs:serviceFactories 元素定义服务工厂对象。这种更加详细的方法具有将默认服务工厂实施替换为您的自定义实现的优势，从而为您提供对 bean 生命周期的最终控制。以下示例演示了如何配置两个资源 Bean，即 customerBean1 和 customerBean2，使用此方法：

```
<beans ... >
  <jaxrs:server id="customerService" address="/service1">
    <jaxrs:serviceFactories>
      <ref bean="sfactory1" />
      <ref bean="sfactory2" />
    </jaxrs:serviceFactories>
  </jaxrs:server>

  <bean id="sfactory1" class="org.apache.cxf.jaxrs.spring.SpringResourceFactory">
     <property name="beanId" value="customerBean1"/>
  </bean>
  <bean id="sfactory2" class="org.apache.cxf.jaxrs.spring.SpringResourceFactory">
     <property name="beanId" value="customerBean2"/>
  </bean>

  <bean id="customerBean1" class="demo.jaxrs.server.CustomerRootResource1"
scope="prototype"/>
  <bean id="customerBean2" class="demo.jaxrs.server.CustomerRootResource2"
scope="prototype"/>
</beans>
```

**注意**

如果您指定了非单例生命周期，通常最好实施和注册 org.apache.cxf.service.Invoker bean （可以通过引用 jaxrs:server/jaxrs:invoker 元素来注册实例）。

**附加 WADL 文档**

您可以选择使用 jaxrs:server 元素上的 docLocation 属性将 WADL 文档与 JAX-RS 服务器端点关联。例如：

```
<jaxrs:server address="/rest" docLocation="wadl/bookStore.wadl">
   <jaxrs:serviceBeans>
      <bean class="org.bar.generated.BookStore"/>
   </jaxrs:serviceBeans>
</jaxrs:server>
```

**模式验证**

如果您有一些外部 XML 架构，用于描述 JAX-B 格式的消息内容，您可以通过 jaxrs:schemaLocations 元素将这些外部模式与 JAX-RS 服务器端点相关联。

例如，如果您与 WADL 文档关联了服务器端点，并且您希望在传入的信息中启用模式验证，您可以指定相关的 XML 模式文件，如下所示：

```
<jaxrs:server address="/rest"
              docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
  <jaxrs:schemaLocations>
    <jaxrs:schemaLocation>classpath:/schemas/a.xsd</jaxrs:schemaLocation>
    <jaxrs:schemaLocation>classpath:/schemas/b.xsd</jaxrs:schemaLocation>
  </jaxrs:schemaLocations>
</jaxrs:server>
```

或者，如果您要在给定目录中包括所有模式文件(.. xsd )，您可以只指定目录名称，如下所示：

```
<jaxrs:server address="/rest"
              docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
  <jaxrs:schemaLocations>
    <jaxrs:schemaLocation>classpath:/schemas/</jaxrs:schemaLocation>
  </jaxrs:schemaLocations>
</jaxrs:server>
```

以这种方式指定架构对于需要访问 JAX-B 模式的任何类型功能通常都很有用。

### 指定数据绑定

您可以使用 jaxrs:dataBinding 元素指定在请求和回复消息中对消息正文进行编码的数据绑定。例如，要指定 JAX-B 数据绑定，您可以配置 JAX-RS 端点，如下所示：

```
<jaxrs:server id="jaxbbook" address="/jaxb">
  <jaxrs:serviceBeans>
    <ref bean="serviceBean" />
  </jaxrs:serviceBeans>
  <jaxrs:dataBinding>
    <bean class="org.apache.cxf.jaxb.JAXBDataBinding"/>
  </jaxrs:dataBinding>
</jaxrs:server>>
```

或者指定 Aegis 数据绑定，您可以配置 JAX-RS 端点，如下所示：

```
<jaxrs:server id="aegisbook" address="/aegis">
  <jaxrs:serviceBeans>
    <ref bean="serviceBean" />
  </jaxrs:serviceBeans>
  <jaxrs:dataBinding>
    <bean class="org.apache.cxf.aegis.databinding.AegisDatabinding">
      <property name="aegisContext">
        <bean class="org.apache.cxf.aegis.AegisContext">
          <property name="writeXsiTypes" value="true"/>
        </bean>
      </property>
    </bean>
  </jaxrs:dataBinding>
</jaxrs:server>
```

### 使用 JMS 传输

可以将 JAX-RS 配置为使用 JMS 消息传递库作为传输协议，而不是 HTTP。由于 JMS 本身 不是 传输协议，因此实际消息传递协议取决于您配置的特定 JMS 实施。

例如，以下 Spring XML 示例演示了如何配置 JAX-RS 服务器端点以使用 JMS 传输协议：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xsi:schemaLocation="
http://cxf.apache.org/transports/jms http://cxf.apache.org/schemas/configuration/jms.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">

    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
    <bean id="ConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL"
value="tcp://localhost:${testutil.ports.EmbeddedJMSBrokerLauncher}" />
    </bean>

    <jaxrs:server xmlns:s="http://books.com"
     serviceName="s:BookService"
     transportId= "http://cxf.apache.org/transports/jms"
     address="jms:queue:test.jmstransport.text?replyToName=test.jmstransport.response">
        <jaxrs:serviceBeans>
            <bean class="org.apache.cxf.systest.jaxrs.JMSBookStore"/>
        </jaxrs:serviceBeans>
```

```
    </jaxrs:server>

</beans>
```

请注意上例的点：

- JMS 实施- JMS 实施由 ConnectionFactory bean 提供，它实例化 Apache ActiveMQ 连接工厂对象。实例化连接工厂后，它会自动作为默认的 JMS 实现层安装。
- JMS conduit 或 destination 对象- Apache CXF 隐式实例化一个 JMS conduit 对象（代表 JMS 消费者）或 JMS 目的地对象（代表 JMS 提供程序）。此对象必须通过 QName 唯一标识，它通过属性 setttings xmlns:s="http://books.com" （定义命名空间前缀）和 serviceName="s:BookService" （定义 QName）定义。
- transport ID- 要选择 JMS 传输，transportId 属性必须设置为 http://cxf.apache.org/transports/jms。
- JMS address- jaxrs:server/@address 属性使用标准化的语法指定要发送到的 JMS 队列或 JMS 主题。有关此语法的详情，请参考 https://tools.ietf.org/id/draft-merrick-jms-uri-06.txt。

### 扩展映射和语言映射

可以配置 JAX-RS 服务器端点，以便它自动将文件后缀（位于 URL 中）映射到 MIME 内容类型标头，并将语言后缀映射到语言类型标头。例如，请考虑以下格式的 HTTP 请求：

```
GET /resource.xml
```

您可以配置 JAX-RS 服务器端点来自动映射 .xml 后缀，如下所示：

```
<jaxrs:server id="customerService" address="/">
  <jaxrs:serviceBeans>
    <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
  </jaxrs:serviceBeans>
  <jaxrs:extensionMappings>
    <entry key="json" value="application/json"/>
    <entry key="xml" value="application/xml"/>
  </jaxrs:extensionMappings>
</jaxrs:server>
```

当前面的服务器端点收到 HTTP 请求时，它会自动创建一个新的类型为 application/xml 的标头，并

从资源 URL 剥离 .xml 后缀。

对于语言映射，请考虑以下格式的 HTTP 请求：

```
GET /resource.en
```

您可以配置 JAX-RS 服务器端点来自动映射 .en 后缀，如下所示：

```
<jaxrs:server id="customerService" address="/">
  <jaxrs:serviceBeans>
    <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
  </jaxrs:serviceBeans>
  <jaxrs:languageMappings>
     <entry key="en" value="en-gb"/>
  </jaxrs:languageMappings>
</jaxrs:server>
```

当前面的服务器端点收到 HTTP 请求时，它会自动创建一个新的 accept 语言标头，其值为 en-gb，并从资源 URL 剥离 .en 后缀。

### 18.1.2. jaxrs:server 属性

#### 属性

表 18.1 “JAX-RS 服务器端点属性” 描述 jaxrs:server 元素上可用的属性。

表 18.1. JAX-RS 服务器端点属性

| 属性 | 描述 |
| --- | --- |
| **id** | 指定其他配置元素可用于引用端点的唯一标识符。 |
| **address** | 指定 HTTP 端点的地址。这个值将覆盖服务合同中指定的值。 |
| **basePackages** | （仅限Spring） 通过指定以逗号分隔的 Java 软件包列表来启用自动发现，这些 Java 软件包列表被搜索来发现 JAX-RS 根资源类和/或 JAX-RS 提供程序类。 |
| **beanNames** | 指定 JAX-RS 根资源 Bean 的 bean ID 列表。在 Spring XML 的上下文中，可以通过在 root 资源 **bean** 元素上设置 **scope** 属性来定义根资源 Bean 的生命周期。 |

| 属性 | 描述 |
| --- | --- |
| **bindingId** | 指定服务使用的消息绑定的 ID。第 23 章 *Apache CXF 绑定 ID* 中提供了有效绑定 ID 列表。 |
| **bus** | 指定配置用于管理服务端点总线的 Spring bean 的 ID。这在将多个端点配置为使用一组常用功能时很有用。 |
| **docLocation** | 指定外部 WADL 文档的位置。 |
| **modelRef** | 将模型模式指定为 classpath 资源（例如，格式为 **classpath:/path/to/model.xml**的 URL）。有关如何定义 JAX-RS 模型模式的详情，请参考 第 18.3 节 “使用 Model Schema 定义 REST 服务”。 |
| **publish** | 指定是否应自动发布该服务。如果设置为 **false**，开发人员必须明确发布端点。 |
| **publishedEndpointUrl** | 指定 URL 基础地址，它插入到自动生成的 WADL 接口的 **wadl:resources/@base** 属性中。 |
| **serviceAnnotation** | **（仅限Spring）** 在 Spring 中指定自动发现的服务注解类名称。当与 **basePackages** 属性结合使用时，此选项限制了自动发现类的集合，**使其仅包含由** 此注解类型标注的类。guess!这是否正确？ |
| **serviceClass** | 指定 JAX-RS root 资源类的名称（实施 JAX-RS 服务）。在这种情况下，该类由 Apache CXF 实例化，而不是按 Blueprint 或 Spring 进行实例化。**如果要实例化 Blueprint 或 Spring 中的类，请使用 jaxrs:serviceBeans 子元素。** |
| **serviceName** | 在特殊情况下，为 JAX-RS 端点指定 service QName（使用格式 ***ns:name***）。详情请查看 “使用 JMS 传输”一节。 |
| **staticSubresourceResolution** | 如果为 **true**，则禁用静态子资源的动态解析。默认为 **false**。 |
| **transportId** | 用于选择非标准传输层（代替 HTTP）。特别是，您可以通过将此属性设置为 **http://cxf.apache.org/transports/jms** 来选择 JMS 传输。详情请查看 “使用 JMS 传输”一节。 |
| **abstract** | **（仅限Spring）** 指定是否 bean 是抽象 Bean。抽象 Bean 充当 concrete bean 定义的父项，且不会实例化。默认值为 **false**。把它设置为 **true** 会指示 bean 工厂不会实例化 bean。 |

| 属性 | 描述 |
| --- | --- |
| **dependent-on** | （仅限Spring） 指定端点在可实例化端点前实例化的 Bean 列表。 |

### 18.1.3. jaxrs:server Child Elements

**子元素**

**表 18.2 “JAX-RS 服务器端点 Child Elements” 描述 jaxrs:server 元素的子元素。**

**表 18.2. JAX-RS 服务器端点 Child Elements**

| 元素 | 描述 |
| --- | --- |
| **jaxrs:executor** | 指定用于该服务的 Java **Executor** （线程池实现）。这使用嵌入式 bean 定义来指定。 |
| **jaxrs:features** | 指定配置 Apache CXF 高级功能的 Bean 列表。您可以提供 bean 引用列表或嵌入式 Bean 列表。 |
| **jaxrs:binding** | **未使用。** |
| **jaxrs:dataBinding** | 指定实施端点使用的数据绑定的类。这使用嵌入式 bean 定义来指定。如需了解更多详细信息，请参阅 “指定数据绑定”一节。 |
| **jaxrs:inInterceptors** | 指定处理入站请求的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxrs:inFaultInterceptors** | 指定处理入站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxrs:outInterceptors** | 指定处理出站回复的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxrs:outFaultInterceptors** | 指定处理出站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。 |
| **jaxrs:invoker** | 指定服务使用的 org.apache.cxf.service.Invoker 接口的实现。 [a] |

| 元素 | 描述 |
| --- | --- |
| **jaxrs:serviceFactories** | 为您提供对与此端点关联的 JAX-RS 根资源生命周期的最大控制程度。此元素的子项（必须是 **org.apache.cxf.jaxrs.lifecycle.ResourceProvider** 类型）的实例来创建 JAX-RS 根资源实例。 |
| **jaxrs:properties** | 指定与端点一起传递的属性的 Spring 映射。这些属性可用于控制诸如启用 MTOM 支持等功能。 |
| **jaxrs:serviceBeans** | 此元素的子项是(**bean** 元素)的实例，或引用(**ref** 元素) JAX-RS root 资源。请注意，在这种情况下，如果 **bean** 元素中存在 **scope** 属性 (Spring only)，则忽略。 |
| **jaxrs:modelBeans** | 包含对一个或多个 **org.apache.cxf.jaxrs.model.UserResource** beans 的引用列表，它们是资源模型的基本元素（与 **jaxrs:resource** 元素相对应）。详情请查看 第 18.3 节 “使用 Model Schema 定义 REST 服务”。 |
| **jaxrs:model** | 直接在此端点中定义资源模型（即，此 **jaxrs:model** 元素可以包含一个或多个 **jaxrs:resource** 元素）。详情请查看 第 18.3 节 “使用 Model Schema 定义 REST 服务”。 |
| **jaxrs:providers** | 允许您将一个或多个自定义 JAX-RS 提供程序注册到此端点。此元素的子项是(**bean** 元素)或对(**ref** 元素) JAX-RS 提供程序的引用。 |
| **jaxrs:extensionMappings** | 当 REST 调用的 URL 以文件扩展名结尾时，您可以使用此元素将它自动与特定的内容类型关联。例如，**.xml** 文件扩展可以与 **application/xml** 内容类型关联。详情请查看 “扩展映射和语言映射”一节。 |
| **jaxrs:languageMappings** | 当 REST 调用的 URL 以语言后缀结尾时，您可以使用此元素映射到特定语言。例如，**.en** 语言后缀可以与 **en-GB** 语言关联。详情请查看 “扩展映射和语言映射”一节。 |
| **jaxrs:schemaLocations** | 指定用于验证 XML 消息内容的一个或多个 XML 模式。此元素可以包含一个或多个 **jaxrs:schemaLocation** 元素，每个元素指定 XML 模式文件的位置（通常是 **类路径** URL）。详情请查看 “模式验证”一节。 |
| **jaxrs:resourceComparator** | 允许您注册自定义资源比较器，它实现了与特定资源类或方法匹配的传入 URL 路径的算法。 |

| 元素 | 描述 |
| --- | --- |
| **jaxrs:resourceClasses** | 如果要从类名称 **创建多个资源，则只能使用 jaxrs:server/@serviceClass** 属性而不是 jaxrs:server/@serviceClass 属性。**jaxrs:resourceClasses** 的子项必须是 **class** 元素，并将 **name** 属性设置为资源类的名称。在这种情况下，类由 Apache CXF 实例化，**而不是由** Blueprint 或 Spring 进行实例化。 |
| [a] Invoker 实现控制如何调用服务。例如，它控制每个请求是否由服务实施的新实例处理，还是在调用之间保留状态。 | |

## 18.2. 配置 JAX-RS 客户端端点

### 18.2.1. 定义 JAX-RS 客户端端点

#### 注入客户端代理

**以 XML 语言(Blueprint XML 或 Spring XML)实例化客户端代理 Bean 的主要点是将其注入另一个 bean，然后使用客户端代理来调用 REST 服务。要在 XML 中创建客户端代理 bean，请使用 jaxrs:client 元素。**

#### 命名空间

**JAX-RS 客户端端点使用与服务器端点 不同的 XML 命名空间来定义。下表显示了哪个 XML 语言使用哪个命名空间：**

| XML 语言 | 客户端端点的命名空间 |
| --- | --- |
| Blueprint | **http://cxf.apache.org/blueprint/jaxrs-client** |
| Spring | **http://cxf.apache.org/jaxrs-client** |

#### 基本客户端端点定义

**以下示例演示了如何在 Blueprint XML 或 Spring XML 中创建客户端代理 bean：**

```
<jaxrs:client id="restClient"
    address="http://localhost:8080/test/services/rest"
    serviceClass="org.apache.cxf.systest.jaxrs.BookStoreJaxrsJaxws"/>
```

您必须设置以下属性来定义基本客户端端点：

**id**

客户端代理的 bean ID 可用于将客户端代理注入 XML 配置中的其他 Bean。

**address**

address 属性指定 REST 调用的基本 URL。

**serviceClass**

serviceClass 属性通过指定根资源类（由 @Path标注）来提供 REST 服务的描述。实际上，这是 服务器 类，但不直接由客户端使用。指定类仅用于其元数据（通过 Java 反映和 JAX-RS 注释），它们用于动态构建客户端代理。

指定标头

您可以使用 jaxrs:headers 子元素将 HTTP 标头添加到客户端代理的调用中，如下所示：

```
<jaxrs:client id="restClient"
      address="http://localhost:8080/test/services/rest"
      serviceClass="org.apache.cxf.systest.jaxrs.BookStoreJaxrsJaxws"
      inheritHeaders="true">
      <jaxrs:headers>
         <entry key="Accept" value="text/xml"/>
      </jaxrs:headers>
</jaxrs:client>
```

### 18.2.2. jaxrs:client 属性

**属性**

表 18.3 “JAX-RS 客户端端点属性” 描述 jaxrs:client 元素上可用的属性。

**表 18.3. JAX-RS 客户端端点属性**

| 属性 | 描述 |
| --- | --- |
| **address** | 指定消费者要发出请求的端点的 HTTP 地址。这个值会覆盖合同中设置的值。 |
| **bindingId** | 指定消费者使用的消息绑定的 ID。第 23 章 *Apache CXF 绑定 ID* 中提供了有效绑定 ID 列表。 |

| 属性 | 描述 |
| --- | --- |
| **bus** | 指定配置管理端点总线的 Spring bean 的 ID。 |
| **inheritHeaders** | 指定如果从这个代理创建了子资源代理，则为这个代理设置的标头是否会被继承。默认为 **false**。 |
| **username** | 指定用于简单用户名/密码身份验证的用户名。 |
| **password** | 指定用于简单用户名/密码身份验证的密码。 |
| **modelRef** | 将模型模式指定为 classpath 资源（例如，格式为 **classpath:/path/to/model.xml**的 URL）。有关如何定义 JAX-RS 模型模式的详情，请参考 第 18.3 节 “使用 Model Schema 定义 REST 服务”。 |
| **serviceClass** | 指定服务接口的名称或资源类（通过 **@PATH**注释），从 JAX-RS 服务器实施重新使用它。在这种情况下，指定的类 **不会** 直接调用（实际上是服务器类）。指定类仅用于其元数据（通过 Java 反映和 JAX-RS 注释），它们用于动态构建客户端代理。 |
| **serviceName** | 在特殊情况下，为 JAX-RS 端点指定 service QName（使用格式 ***ns*:*name***）。详情请查看 “使用 JMS 传输”一节。 |
| **threadSafe** | 指定客户端代理是否为 thread-safe。默认为 **false**。 |
| **transportId** | 用于选择非标准传输层（代替 HTTP）。特别是，您可以通过将此属性设置为 **http://cxf.apache.org/transports/jms** 来选择 JMS 传输。详情请查看 “使用 JMS 传输”一节。 |
| **abstract** | **（仅限Spring）** 指定是否 bean 是抽象 Bean。抽象 Bean 充当 concrete bean 定义的父项，且不会实例化。默认值为 **false**。把它设置为 **true** 会指示 bean 工厂不会实例化 bean。 |
| **dependent-on** | **（仅限Spring）** 指定端点在可实例化前所依赖的 Bean 列表。 |

### 18.2.3. jaxrs:client Child Elements

**子元素**

**表 18.4 “JAX-RS 客户端端点 Child Elements” 描述 jaxrs:client 元素的子元素。**

**表 18.4. JAX-RS 客户端端点 Child Elements**

| 元素 | 描述 |
| --- | --- |
| **jaxrs:executor** | |
| **jaxrs:features** | 指定配置 Apache CXF 高级功能的 Bean 列表。您可以提供 bean 引用列表或嵌入式 Bean 列表。 |
| **jaxrs:binding** | **未使用。** |
| **jaxrs:dataBinding** | 指定实施端点使用的数据绑定的类。这使用嵌入式 bean 定义来指定。如需了解更多详细信息，请参阅 "指定数据绑定"一节。 |
| **jaxrs:inInterceptors** | 指定处理入站响应的拦截器列表。如需更多信息，请参阅 第 VII 部分 "开发 Apache CXF Interceptors"。 |
| **jaxrs:inFaultInterceptors** | 指定处理入站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 "开发 Apache CXF Interceptors"。 |
| **jaxrs:outInterceptors** | 指定处理出站请求的拦截器列表。如需更多信息，请参阅 第 VII 部分 "开发 Apache CXF Interceptors"。 |
| **jaxrs:outFaultInterceptors** | 指定处理出站故障消息的拦截器列表。如需更多信息，请参阅 第 VII 部分 "开发 Apache CXF Interceptors"。 |
| **jaxrs:properties** | 指定传递给端点的属性映射。 |
| **jaxrs:providers** | 允许您将一个或多个自定义 JAX-RS 提供程序注册到此端点。此元素的子项是(**bean** 元素)或对(**ref** 元素) JAX-RS 提供程序的引用。 |
| **jaxrs:modelBeans** | 包含对一个或多个 **org.apache.cxf.jaxrs.model.UserResource** beans 的引用列表，它们是资源模型的基本元素（与 **jaxrs:resource** 元素相对应）。详情请查看 第 18.3 节 "使用 Model Schema 定义 REST 服务"。 |
| **jaxrs:model** | 直接在此端点中定义资源模型（即，**jaxrs:model** 元素包含一个或多个 **jaxrs:resource** 元素）。详情请查看 第 18.3 节 "使用 Model Schema 定义 REST 服务"。 |
| **jaxrs:headers** | 用于设置传出消息上的标头。详情请查看 "指定标头"一节。 |

| 元素 | 描述 |
| --- | --- |
| **jaxrs:schemaLocations** | 指定用于验证 XML 消息内容的一个或多个 XML 模式。此元素可以包含一个或多个 **jaxrs:schemaLocation** 元素，每个元素指定 XML 模式文件的位置（通常是 **类路径** URL）。详情请查看 "模式验证"一节。 |

## 18.3. 使用 MODEL SCHEMA 定义 REST 服务

### 没有注解的 RESTful 服务

**JAX-RS 模型架构使得可以在不注解 Java 类 的情况下 定义 RESTful 服务。也就是说，您不将 @Path、@PathParam、@Consumes、@Produces 等注释直接添加到 Java 类（或接口），您可以使用模型模式在单独的 XML 文件中提供所有相关 REST 元数据。例如，当您无法修改实施该服务的 Java 源时，这非常有用。**

### model 模式示例

**例 18.1 "JAX-RS 模型架构示例" 显示了一个模型模式示例，它为 BookStoreNoAnnotations root 资源类定义服务元数据。**

**例 18.1. JAX-RS 模型架构示例**

```
<model xmlns="http://cxf.apache.org/jaxrs">
  <resource name="org.apache.cxf.systest.jaxrs.BookStoreNoAnnotations" path="bookstore"
    produces="application/json" consumes="application/json">
    <operation name="getBook" verb="GET" path="/books/{id}" produces="application/xml">
      <param name="id" type="PATH"/>
    </operation>
    <operation name="getBookChapter" path="/books/{id}/chapter">
      <param name="id" type="PATH"/>
    </operation>
    <operation name="updateBook" verb="PUT">
      <param name="book" type="REQUEST_BODY"/>
    </operation>
  </resource>
  <resource name="org.apache.cxf.systest.jaxrs.ChapterNoAnnotations">
    <operation name="getItself" verb="GET"/>
    <operation name="updateChapter" verb="PUT" consumes="application/xml">
       <param name="content" type="REQUEST_BODY"/>
    </operation>
  </resource>
</model>
```

命名空间

用于定义模型模式的 XML 命名空间取决于您在 Blueprint XML 或 Spring XML 中定义对应的 JAX-RS 端点。下表显示了哪个 XML 语言使用哪个命名空间：

| XML 语言 | Namespace |
| --- | --- |
| Blueprint | http://cxf.apache.org/blueprint/jaxrs |
| Spring | http://cxf.apache.org/jaxrs |

如何将模型模式附加到端点

要将模型模式定义并附加到端点，请执行以下步骤：

1. 定义模型模式，使用适用于您选择的注入平台的适当 XML 命名空间（打印 XML 或 Spring XML）。

2. 将模型架构文件添加到项目的资源中，以便在最终软件包(JAR、WAR 或 OSGi 捆绑包文件)中的 classpath 上提供 schema 文件。

注意

或者，也可以使用端点的 jaxrs:model 子元素直接将模型架构嵌入到 JAX-RS 端点中。

3. 通过将端点的 modelRef 属性设置为类路径（使用 classpath URL）上模型模式的位置，将端点配置为使用模型模式。

4. 如有必要，使用 jaxrs:serviceBeans 元素显式实例化 root 资源。如果模型模式直接引用 root 资源类（而不是引用基本接口），您可以跳过这一步。

配置引用类的模型模式

如果模型模式直接应用到 root 资源类，则不需要使用 jaxrs:serviceBeans 元素定义任何根资源 Bean，因为模型模式会自动实例化 root 资源 Bean。

例如，如果 customer-resources.xml 是一个模型模式，它将元数据与客户资源类相关联，您可以按照以下方式实例化 customerService 服务端点：

```
<jaxrs:server id="customerService"
              address="/customers"
              modelRef="classpath:/org/example/schemas/customer-resources.xml" />
```

### 配置引用接口的模型模式

如果模型模式适用于 Java 接口（这是 root 资源的基本接口），则必须使用端点中的 jaxrs:serviceBeans 元素实例化 root 资源类。

例如，如果 customer-interfaces.xml 是一个模型模式，它将元数据与客户接口相关联，您可以按照以下方式实例化 customerService 服务端点：

```
<jaxrs:server id="customerService"
              address="/customers"
              modelRef="classpath:/org/example/schemas/customer-interfaces.xml">
   <jaxrs:serviceBeans>
      <ref component-id="serviceBean" />
   </jaxrs:serviceBeans>
</jaxrs:server>

<bean id="serviceBean" class="service.CustomerService"/>
```

### 模型架构参考

模型模式使用以下 XML 元素定义：

**model**

模型模式的根元素。如果您需要引用模型架构（例如，使用 modelRef 属性从 JAX-RS 端点中），您应该在此元素上设置 id 属性。

**model/resource**

resource 元素用于将元数据与特定根资源类（或与对应的接口）关联。您可以在 resource 元素上定义以下属性：

| 属性 | 描述 + |
| --- | --- |

| 属性 | 描述 + |
| --- | --- |
| **name** | 应用此资源模型的资源类（或对应接口）的名称。<br>+ |
| **path** | 映射到此资源的 REST URL 路径的组件。<br>+ |
| **使用** | 指定此资源使用的内容类型(Internet 介质类型)，如 **application/xml** 或 **application/json**。<br>+ |
| **生成** | 指定此资源生成的内容类型(Internet 介质类型)，如 **application/xml** 或 **application/json**。<br>+ |

**model/resource/operation**

**operation 元素用于将元数据与 Java 方法关联。您可以在 operation 元素上定义以下属性：**

| 属性 | 描述 + |
| --- | --- |
| **name** | 应用此元素的 Java 方法的名称。<br>+ |
| **path** | 映射到此方法的 REST URL 路径的组件。此属性值可以包含参数引用，例如：**path="/books/{id}/chapter"**，其中 **{id}** 从路径中提取 **id** 参数的值。<br>+ |
| **verb** | 指定映射到此方法的 HTTP 动词。通常之一：**GET**、**POST**、**PUT** 或 **DELETE**。**如果没有指定** HTTP 动词，则假设 Java 方法是一个 **子资源定位器**，它会返回对子资源对象的引用（子资源类还必须使用 **resource** 元素提供元数据）。<br>+ |
| **使用** | 指定此操作使用的内容类型(Internet 介质类型)，如 **application/xml** 或 **application/json**。<br>+ |

| 属性 | 描述 + |
| --- | --- |
| **生成** | 指定此操作生成的内容类型(Internet 介质类型)，如 **application/xml** 或 **application/json**。<br><br>+ |
| **Oneway** | 如果为 **true**，请将操作配置为 **单向**，即不需要回复消息。默认值为 **false**。<br><br>+ |

**model/resource/operation/param**

**param 元素用于从 REST URL 中提取值，并将它注入到其中一个方法参数中。您可以在 param 元素中定义以下属性：**

| 属性 | 描述 + |
| --- | --- |
| **name** | 应用此元素的 Java method 参数的名称。<br><br>+ |
| **type** | 指定从 REST URL 或消息中提取参数值。它可以设置为以下值之一：**PATH**,**QUERY**,**MATRIX**,**HEADER**,**COOKIE**,**FORM**,**CONTEXT**,**REQUEST_BODY**.<br><br>+ |
| **defaultValue** | 要注入参数的默认值，如果无法从 REST URL 或消息中提取值。<br><br>+ |
| **encoded** | 如果为 **true**，则参数值以 URI 编码的形式注入（即，使用 **%nn** 编码）。默认为 **false**。例如，当从 URL 路径提取参数时，**/name/Joe%20Bloggs** 为 **true**，参数被注入为 **Joe%20Bloggs**; 否则，参数将注入 **Joe Bloggs**。<br><br>+ |

# 第 19 章 APACHE CXF LOGGING

**摘要**

本章论述了如何在 Apache CXF 运行时中配置日志。

## 19.1. APACHE CXF LOGGING 概述

### 概述

Apache CXF 使用 Java 日志记录实用程序 java.util.logging。日志记录以日志配置文件配置，该文件使用标准的 java.util.Properties 格式编写。要在应用程序上运行日志记录，您可以以编程方式指定日志记录，或者在启动应用程序时定义指向日志记录配置文件的命令的属性。

### 默认属性文件

Apache CXF 附带一个默认的 logging.properties 文件，该文件位于 *InstallDir*/etc 目录中。此文件配置日志消息的输出目的地和已发布的消息级别。默认配置将日志记录器设置为将消息标记为 WARNING 级别。您可以在不更改任何配置设置的情况下使用默认文件，也可以更改配置设置以适合您的特定应用程序。

### 日志记录功能

Apache CXF 包括一个日志记录功能，可插入到客户端或您的服务以启用日志记录。例 19.1 “配置启用日志” 显示启用日志记录功能的配置。

**例 19.1. 配置启用日志**

```
<jaxws:endpoint...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

如需更多信息，请参阅 第 19.6 节 “日志记录消息内容”。

### 从何处开始？

要运行简单的日志记录示例，请按照 第 19.2 节 “使用日志记录的简单示例” 中概述的说明进行操作。

有关日志记录如何在 Apache CXF 中工作的更多信息，请阅读此整个章节。

### 有关 java.util.logging 的更多信息

java.util.logging 工具是最广泛的 Java 日志记录框架之一。在线有很多信息描述了如何使用和扩展此框架。作为起点，以下文档提供了 java.util.logging 的良好概述：

- http://download.oracle.com/javase/1.5.0/docs/guide/logging/overview.html
- http://download.oracle.com/javase/1.5.0/docs/api/java/util/logging/package-summary.html

## 19.2. 使用日志记录的简单示例

### 更改日志级别和输出目的地

要更改 wsdl_first 示例应用程序中的日志消息的日志级别和输出目的地，请完成以下步骤：

1. 如使用 *InstallDir*/samples/wsdl_first 目录中的 README.txt 文件的 java 部分运行 示例服务器。请注意，server start 命令指定默认的 logging.properties 文件，如下所示：

| 平台 | 命令 + |
| --- | --- |
| Windows | 启动 **java -Djava.util.logging.config.file=%CXF_HOME%\etc\logging.properties demo.hw.server.Server**<br><br>+ |
| UNIX | **java -Djava.util.logging.config.file=$CXF_HOME/etc/logging.properties demo.hw.server.Server &**<br><br>+ |

默认 logging.properties 文件位于 *InstallDir*/etc 目录中。它将 Apache CXF 日志记录器配置为将 WARNING 级别日志消息输出到控制台。因此，您会看到非常少地打印到控制台。

2. 按照 README.txt 文件中所述停止服务器。

3. 复制默认 logging.properties 文件，将其命名为 mylogging.properties 文件，并将它保存到与默认 logging.properties 文件相同的目录中。

4. 通过编辑以下配置，将 mylogging.properties 文件中的全局日志记录级别和控制台日志记录级别改为 INFO ：

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

5. 使用以下命令重启服务器：

| 平台 | 命令 + |
| --- | --- |
| Windows | 启动 **java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server**<br><br>+ |
| UNIX | **java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &**<br><br>+ |

由于已将全局日志记录和控制台日志记录器配置为记录级别 INFO 的信息，因此您会看到更多输出到控制台的日志消息。

## 19.3. 默认日志记录配置文件

### 19.3.1. 日志配置概述

默认日志记录配置文件 logging.properties 位于 *InstallDir*/etc 目录中。它将 Apache CXF 日志记录

器配置为将 WARNING 级别的信息输出到控制台。如果此级别的日志记录适合您的应用程序，则不必在使用前对该文件进行任何更改。但是，您可以更改日志消息中的详细程度。例如，您可以更改日志消息是否发送到控制台、文件还是两者都发送到控制台。另外，您可以在单个软件包级别指定日志记录。

**注意**

本节讨论出现在默认 logging.properties 文件中的配置属性。但是，您可以设置许多其他 java.util.logging 配置属性。有关 java.util.logging API 的更多信息，请参阅 java.util.logging javadoc：http://download.oracle.com/javase/1.5/docs/api/java/util/logging/package-summary.html.

### 19.3.2. 配置日志记录输出

#### 概述

Java 日志记录实用程序 java.util.logging 使用处理器类输出日志消息。表 19.1 “java.util.logging Handler 类” 显示默认 logging.properties 文件中配置的处理程序。

表 19.1. java.util.logging Handler 类

| 处理程序类 | 输出信息 |
| --- | --- |
| **ConsoleHandler** | 将日志消息输出到控制台 |
| **FileHandler** | 将日志消息输出到文件 |

**重要**

处理程序类必须位于系统类路径上，才能在启动时由 Java 虚拟机安装。这是设置 Apache CXF 环境时完成的。

#### 配置控制台处理器

例 19.2 “配置控制台处理程序” 显示用于配置控制台日志记录器的代码。

**例 19.2. 配置控制台处理程序**

```
handlers= java.util.logging.ConsoleHandler
```

控制台处理程序还支持 例 19.3 “控制台处理程序属性” 中显示的配置属性。

**例 19.3. 控制台处理程序属性**

```
java.util.logging.ConsoleHandler.level = WARNING
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

例 19.3 “控制台处理程序属性” 中显示的配置属性如下：

控制台处理程序支持单独的日志级别配置属性。这可让您在全局日志记录设置不同时限制输出到控制台的日志信息（请参阅 第 19.3.3 节 “配置日志记录级别”）。默认设置为 **WARNING**。

指定控制台处理器类用来格式化日志消息的 **java.util.logging formatter** 类。默认设置为 **java.util.logging.SimpleFormatter**。

**配置文件处理程序**

例 19.4 “配置文件处理程序” 显示配置文件处理程序的代码。

**例 19.4. 配置文件处理程序**

```
handlers= java.util.logging.FileHandler
```

文件处理程序还支持 例 19.5 “文件处理程序配置属性” 中显示的配置属性。

**例 19.5. 文件处理程序配置属性**

```
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

例 19.5 “文件处理程序配置属性” 中显示的配置属性如下：

指定输出文件的位置和模式。默认设置为您的主目录。

指定，以字节为单位，日志记录器写入任何一个文件的最大值。默认设置为 50000。如果将其设置为零，则日志记录器写入任何一个文件的数量没有限制。

指定要循环的输出文件数量。默认设置为 1。

指定文件处理器类用来格式化日志消息的 java.util.logging formatter 类。默认设置为 java.util.logging.XMLFormatter。

配置控制台处理程序和文件处理程序

您可以通过指定控制台处理器和文件处理程序（用逗号分开）将日志记录实用程序设置为控制台和文件发送到文件，如 配置两个控制台日志记录和文件 所示。

配置两个控制台日志记录和文件

```
Logging
```

```
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

### 19.3.3. 配置日志记录级别

日志记录级别

java.util.logging 框架支持以下级别的日志记录，从最低到最详细：

- 严重
- WARNING

- INFO
- CONFIG
- FINE
- FINER
- FINEST

**配置全局日志记录级别**

要配置日志记录在所有日志记录器的事件类型，请配置全局日志记录级别，如 例 19.6 “配置全局日志记录级别” 所示。

**例 19.6. 配置全局日志记录级别**

```
.level= WARNING
```

**在单独的软件包中配置日志记录**

**level**

java.util.logging 框架支持在单个软件包级别上配置日志记录。例如，例 19.7 “在软件包级别配置日志记录” 中显示的代码行在 com.xyz.foo 软件包中的类上配置 SEVERE 级别的日志记录。

**例 19.7. 在软件包级别配置日志记录**

```
com.xyz.foo.level = SEVERE
```

## 19.4. 在命令行中启用日志记录

**概述**

您可以在应用程序上运行日志记录实用程序，方法是在启动应用程序时定义

java.util.logging.config.file 属性。您可以指定默认 logging.properties 文件或该应用程序唯一的 logging.properties 文件。

在应用程序中指定日志配置文件

start-up

要在应用程序启动时指定日志记录，请在启动应用程序时添加 例 19.8 “在命令行中启动日志记录的标记” 中显示的标志。

例 19.8. 在命令行中启动日志记录的标记

```
-Djava.util.logging.config.file=myfile
```

## 19.5. 子系统和服务的日志

### 概述

您可以使用 “在单独的软件包中配置日志记录”一节 中描述的 com.xyz.foo.level 配置属性来为指定的 Apache CXF 日志记录子系统设置精细的日志记录。

### Apache CXF 日志记录子系统

表 19.2 “Apache CXF Logging 子系统” 显示可用 Apache CXF 日志记录子系统的列表。

表 19.2. Apache CXF Logging 子系统

| 子系统 | 描述 |
| --- | --- |
| org.apache.cxf.aegis | Aegis 绑定 |
| org.apache.cxf.binding.coloc | colocated 绑定 |
| org.apache.cxf.binding.http | HTTP 绑定 |
| org.apache.cxf.binding.jbi | JBI 绑定 |
| org.apache.cxf.binding.object | Java 对象绑定 |
| org.apache.cxf.binding.soap | SOAP 绑定 |

| 子系统 | 描述 |
| --- | --- |
| **org.apache.cxf.binding.xml** | XML 绑定 |
| **org.apache.cxf.bus** | Apache CXF 总线 |
| **org.apache.cxf.configuration** | 配置框架 |
| **org.apache.cxf.endpoint** | 服务器和客户端端点 |
| **org.apache.cxf.interceptor** | 拦截器 |
| **org.apache.cxf.jaxws** | JAX-WS 风格的消息交换、JAX-WS 处理程序处理和与 JAX-WS 和配置相关的拦截器的前端 |
| **org.apache.cxf.jbi** | JBI 容器集成类 |
| **org.apache.cxf.jca** | JCA 容器集成类 |
| **org.apache.cxf.js** | JavaScript 前端 |
| **org.apache.cxf.transport.http** | HTTP 传输 |
| **org.apache.cxf.transport.https** | 使用 HTTPS 的 HTTP 传输的安全版本 |
| **org.apache.cxf.transport.jbi** | JBI 传输 |
| **org.apache.cxf.transport.jms** | JMS 传输 |
| **org.apache.cxf.transport.local** | 使用本地文件系统传输实现 |
| **org.apache.cxf.transport.servlet** | HTTP 传输和 servlet 实现，用于将 JAX-WS 端点加载到 servlet 容器中 |
| **org.apache.cxf.ws.addressing** | WS-Addressing 实现 |
| **org.apache.cxf.ws.policy** | WS-Policy 实现 |
| **org.apache.cxf.ws.rm** | WS-ReliableMessaging (WS-RM)实现 |
| **org.apache.cxf.ws.security.wss4j** | WSS4J 安全实施 |

**Example**

**WS-地址示例包含在 *InstallDir*/samples/ws_addressing 目录中。logging 在那个目录中的 logging.properties 文件中配置。例 19.9 “为 WS-Addressing 配置日志记录” 中显示相关的配置行。**

**例 19.9. 为 WS-Addressing 配置日志记录**

```
java.util.logging.ConsoleHandler.formatter = demos.ws_addressing.common.ConciseFormatter
...
org.apache.cxf.ws.addressing.soap.MAPCodec.level = INFO
```

例 19.9 “为 WS-Addressing 配置日志记录” 中的配置启用与 WS-Addressing 标头相关的日志消息，并以简洁的形式将它们显示在控制台中。

有关运行此示例的详情，请查看 *InstallDir*/samples/ws_addressing 目录中的 README.txt 文件。

## 19.6. 日志记录消息内容

### 概述

您可以记录服务与消费者之间发送的消息内容。例如，您可能希望记录在服务和消费者之间发送的 SOAP 消息的内容。

### 配置消息内容日志记录

要记录在服务和消费者之间发送的消息，反之亦然，请完成以下步骤：

1. 将日志记录功能添加到端点配置中。
2. 在您的消费者配置中添加日志记录功能。
3. 配置日志记录系统日志 INFO 级别消息。

### 在端点中添加日志记录功能

添加日志记录功能您的端点配置，如 例 19.10 “在端点配置中添加日志记录” 所示。

**例 19.10. 在端点配置中添加日志记录**

```
<jaxws:endpoint ...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

例 19.10 “在端点配置中添加日志记录” 中显示的 XML 示例允许记录 SOAP 信息。

**将日志记录功能添加到消费者中**

添加您的客户端配置功能，如 例 19.11 “将日志记录添加到客户端配置” 所示。

**例 19.11. 将日志记录添加到客户端配置**

```
<jaxws:client ...>
   <jaxws:features>
     <bean class="org.apache.cxf.feature.LoggingFeature"/>
    </jaxws:features>
</jaxws:client>
```

例 19.11 “将日志记录添加到客户端配置” 中显示的 XML 示例允许记录 SOAP 信息。

**将日志记录设置为日志 INFO 级别信息**

确保与服务关联的 logging.properties 文件被配置为记录 INFO 级别信息，如 例 19.12 “将日志记录级别设置为 INFO” 所示。

**例 19.12. 将日志记录级别设置为 INFO**

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

**日志记录 SOAP 消息**

要查看 SOAP 消息的日志，请修改位于 *InstallDir*/samples/wsdl_first 目录中的 wsdl_first 示例应用程序，如下所示：

1. 将 例 19.13 “Logging SOAP 消息的端点配置” 中显示的 jaxws:features 元素添加到位于

**wsdl_first 示例目录中的 cxf.xml 配置文件：**

**例 19.13. Logging SOAP 消息的端点配置**

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                createdFromAPI="true">
  <jaxws:properties>
    <entry key="schema-validation-enabled" value="true" />
  </jaxws:properties>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

2. **这个示例使用默认 logging.properties 文件，该文件位于 *InstallDir*/etc 目录中。复制此文件并将其命名为 mylogging.properties。**

3. **在 mylogging.properties 文件中，编辑 .level 和 java.util.logging.ConsoleHandler.level 配置属性，将日志记录级别改为 INFO，如下所示：**

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

4. **使用 cxf.xml 文件和 mylogging.properties 文件中的新配置设置启动服务器，如下所示：**

| 平台 | 命令 + |
| --- | --- |
| Windows | 启动 **java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server**<br><br>+ |
| UNIX | **java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &**<br><br>+ |

5. **使用以下命令启动 hello world 客户端：**

| 平台 | 命令 + |
| --- | --- |
| Windows | **java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.client.Client .\wsdl\hello_world.wsdl**<br><br>+ |
| UNIX | **java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.client.Client ./wsdl/hello_world.wsdl**<br><br>+ |

**SOAP 消息记录到控制台。**

# 第 20 章 部署 WS-ADDRESSING

## 摘要

Apache CXF 支持 JAX-WS 应用程序的 WS-地址。本章解释了如何在 Apache CXF 运行时环境中部署 WS-Addressing。

## 20.1. WS-地址简介

### 概述

WS-寻址是一种规范，允许服务以中立的方式通信寻址信息。它由两个部分组成：

- 通信对 Web 服务端点的引用的结构
- 一组将寻址信息与特定消息关联的消息地址属性(MAP)

### 支持的规格

Apache CXF 支持 WS-Addressing 2004/08 规范和 WS-Addressing 2005/03 规范。

### 更多信息

有关 WS-地址的详细信息，请参阅 2004/08 提交，网址为 http://www.w3.org/Submission/ws-addressing/。

## 20.2. WS-ADDRESSING INTERCEPTORS

### 概述

在 Apache CXF 中，WS-Addressing 功能作为拦截器实现。Apache CXF 运行时使用拦截器来截获和处理被发送和接收的原始消息。当传输收到消息时，它会创建一个消息对象，并通过拦截器链发送该消息。如果将 WS-Addressing 拦截器添加到应用程序的拦截器链中，则处理包含消息的任何 WS 寻址信息。

### WS-Addressing Interceptors

WS-寻址实现由两个拦截器组成，如 表 20.1 “WS-Addressing Interceptors” 所述。

表 20.1. WS-Addressing Interceptors

| 拦截器 | 描述 |
| --- | --- |
| org.apache.cxf.ws.addressing.MAPAggregator | 逻辑拦截器负责聚合出站消息的消息地址属性(MAPs)。 |
| org.apache.cxf.ws.addressing.soap.MAPCodec | 负责编码和解码消息地址属性(MAPs)作为 SOAP 标头的特定于协议的拦截器。 |

## 20.3. 启用 WS-ADDRESSING

### 概述

要启用 WS-Addressing WS-Addressing 拦截器，必须添加到入站和出站拦截器链中。这可以通过以下方法之一完成：

- Apache CXF 功能
- RMAssertion 和 WS-Policy Framework
- 在 WS-Addressing 功能中使用策略断言

### 将 WS-Addressing 添加为功能

通过向客户端和服务器配置添加 WS-Addressing 功能可以启用 WS-Addressing，如 例 20.1 “client.xml 并将 WS-Addressing 功能添加到客户端配置” 和 例 20.2 “server.xml 并将 WS-Addressing 功能添加到服务器配置” 所示。

例 20.1. client.xml 并将 WS-Addressing 功能添加到客户端配置

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

```
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/ws/addressing
        http://cxf.apache.org/schemas/ws-addr-conf.xsd">

    <jaxws:client ...>
      <jaxws:features>
        <wsa:addressing/>
      </jaxws:features>
    </jaxws:client>
</beans>
```

**例 20.2. server.xml 并将 WS-Addressing 功能添加到服务器配置**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint ...>
      <jaxws:features>
        <wsa:addressing/>
      </jaxws:features>
    </jaxws:endpoint>
</beans>
```

## 20.4. 配置 WS-ADDRESSING 属性

### 概述

Apache CXF WS-Addressing 功能元素在命名空间 http://cxf.apache.org/ws/addressing 中定义。它支持 表 20.2 “WS-Addressing 属性” 中描述的两个属性。

**表 20.2. WS-Addressing 属性**

| 属性名称 | value |
| --- | --- |
| **allowDuplicates** | 确定是否容许重复的 MessageID 的布尔值。默认设置为 **true**。 |
| **usingAddressingAdvisory** | 指示 WSDL 中存在 **UsingAddressing** 元素是否只是公告的布尔值；也就是说，其 no 不会阻止 WS-Addressing 标头的编码。 |

### 配置 WS-Addressing 属性

通过添加属性以及您要将其设置为服务器或客户端配置文件中的 WS-Addressing 功能来配置 WS-Addressing 属性。例如，以下配置 extract 在服务器端点中将 allowDuplicates 属性设置为 false ：

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing" ...>
   <jaxws:endpoint ...>
      <jaxws:features>
         <wsa:addressing allowDuplicates="false"/>
      </jaxws:features>
   </jaxws:endpoint>
</beans>
```

### 使用嵌入在功能中的 WS-Policy 断言

在 例 20.3 “使用策略配置 WS-Addressing” 中，一个寻址策略断言，启用非匿名响应嵌入在 policies 元素中。

**例 20.3. 使用策略配置 WS-Addressing**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:policy="http://cxf.apache.org/policy-config"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

   <jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort"
                   createdFromAPI="true">
      <jaxws:features>
         <policy:policies>
            <wsp:Policy xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
               <wsam:Addressing>
                  <wsp:Policy>
                     <wsam:NonAnonymousResponses/>
                  </wsp:Policy>
               </wsam:Addressing>
            </wsp:Policy>
         <policy:policies>
```

```
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

# 第 21 章 启用可靠的消息传递

**摘要**

Apache CXF 支持 WS-Reliable Messaging (WS-RM)。本章解释了如何在 Apache CXF 中启用和配置 WS-RM。

## 21.1. WS-RM 简介

### 概述

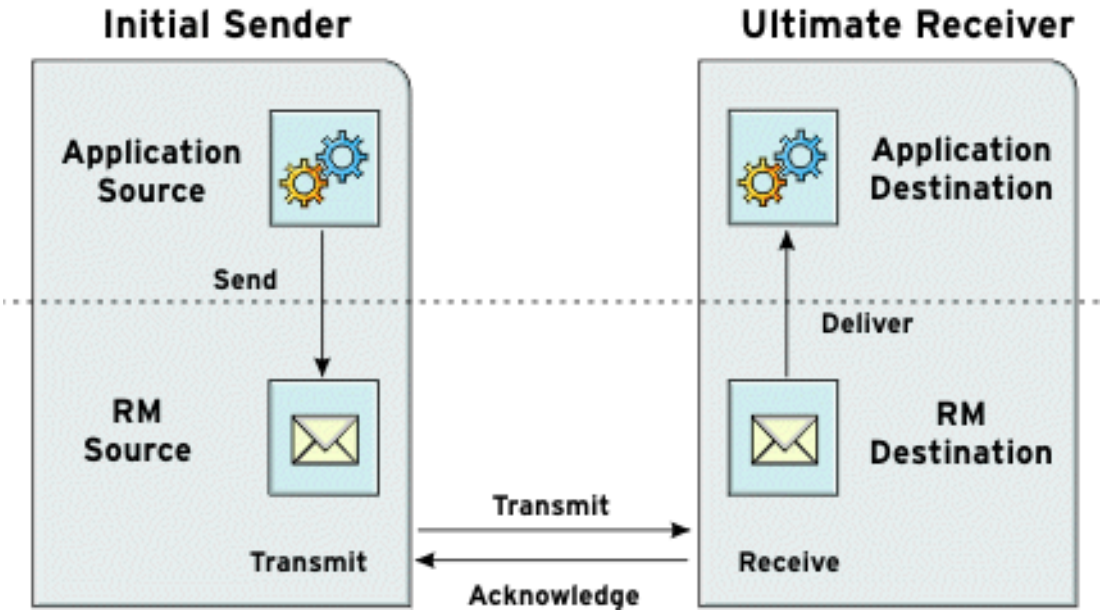
WS-ReliableMessaging (WS-RM)是一种协议，可确保在分布式环境中可靠传递消息。它使消息可以在存在软件、系统或网络故障的分布式应用程序之间可靠地交付。

例如，可以使用 WS-RM 来确保在网络中准确发送正确的消息，并按照正确的顺序发送正确的消息。

### WS-RM 的工作原理

WS-RM 确保在源端点和目标端点之间可靠地传递消息。源是消息的初始发送者，目的地是最终接收器，如 图 21.1 “Web 服务可靠的消息传递” 所示。

图 21.1. Web 服务可靠的消息传递




WS-RM 消息的流程如下所述：

1.

RM 源向 RM 目的地发送 CreateSequence 协议消息。这包括接收确认的端点的引用( wsrm:AcksTo 端点)。

2. RM 目的地将 CreateSequenceResponse 协议消息发回到 RM 源。此消息包含 RM 序列会话的序列 ID。

3. RM 源向应用程序源发送的每个消息添加一个 RM Sequence 标头。此标头包含序列 ID 和唯一消息 ID。

4. RM 源将每个消息传输到 RM 目的地。

5. RM 目的地通过发送包含 RM SequenceAcknowledgement 标头的消息来确认来自 RM 源的消息。

6. RM 目的地以正好一次顺序向应用程序目的地提供消息。

7. RM 源重新传输了尚未收到确认的消息。

在基础重新传输间隔后进行第一次重新传输尝试。默认情况下，会进行连续重新传输尝试，以指数避退间隔，或者以固定间隔为固定间隔。如需了解更多详细信息，请参阅 第 21.5 节 “配置 WS-RM”。

此整个进程对请求和响应消息都是对称的；也就是说，如果响应消息，服务器将充当 RM 源，客户端则充当 RM 目的地。

### WS-RM 交付保证

WS-RM 保证在分布式环境中提供可靠的消息，无论使用的传输协议是什么。如果无法保证提供可靠，则源或目标端点会记录错误。

### 支持的规格

Apache CXF 支持以下 WS-RM 规范版本：

#### WS-ReliableMessaging 1.0

（默认） 从 2011 年 2 月开始提交版本 （现在已过时）。但是，出于向后兼容的原因，这个版本被用作默认版本。

WS-RM 的版本 1.0 使用以下命名空间：

```
http://schemas.xmlsoap.org/ws/2005/02/rm/
```

这个 WS-RM 版本可用于以下 WS-Addressing 版本之一：

```
http://schemas.xmlsoap.org/ws/2004/08/addressing (default)
http://www.w3.org/2005/08/addressing
```

严格说，为了遵守 WS-RM 的 2 月 5 日提交版本，您完全不建议使用这些 WS 寻址版本（这是 Apache CXF 中的默认设置）。但是大多数其他 Web 服务实现已切换到最新的 WS 寻址规范，因此 Apache CXF 允许您选择 WS-A 版本以促进互操作性（请参阅 第 21.4 节 “运行时控制”）。

**WS-ReliableMessaging 1.1/1.2**

对应于官方 1.1/1.2 Web 服务可靠的消息传递规格。

WS-RM 的版本 1.1 和 1.2 使用以下命名空间：

```
http://docs.oasis-open.org/ws-rx/wsrm/200702
```

WS-RM 的 1.1 和 1.2 版本使用以下 WS-Addressing 版本：

```
http://www.w3.org/2005/08/addressing
```

选择 WS-RM 版本

您可以选择要使用的 WS-RM 规格版本，如下所示：

**服务器端**

在提供商一侧，Apache CXF 适应客户端使用 WS-ReliableMessaging 的任何版本并适当响应。

**客户端**

在客户端上，WS-RM 版本由您在客户端配置中使用的命名空间决定（请参阅 第 21.5 节 “配置 WS-RM”），或使用运行时控制选项覆盖 WS-RM 版本（请参阅 第 21.4 节 “运行时控制”）。

## 21.2. WS-RM INTERCEPTORS

### 概述

在 Apache CXF 中，WS-RM 功能实施为拦截器。Apache CXF 运行时使用拦截器来截获和处理被发送和接收的原始消息。当传输收到消息时，它会创建一个消息对象，并通过拦截器链发送该消息。如果应用的拦截器链包含 WS-RM 拦截器，则应用程序可以参与可靠的消息传递会话。WS-RM 拦截器处理消息块的集合和聚合。它们还处理所有确认和重新传输逻辑。

### Apache CXF WS-RM Interceptors

Apache CXF WS-RM 实现由四个拦截器组成，这些拦截器在 表 21.1 “Apache CXF WS-ReliableMessaging Interceptors” 中进行了描述。

表 21.1. Apache CXF WS-ReliableMessaging Interceptors

| 拦截器 | 描述 |
| --- | --- |
| **org.apache.cxf.ws.rm.RMOutInterceptor** | 处理为出站消息提供可靠性保证的逻辑方面。<br><br>负责发送 **CreateSequence** 请求并等待其 **CreateSequenceResponse** 响应。<br><br>还负责聚合应用程序消息的序列属性-ID 和消息编号。 |
| **org.apache.cxf.ws.rm.RMInInterceptor** | 负责拦截和处理 RM 协议消息和 **序列** 消息，这些消息在应用消息上得到缓解。 |
| **org.apache.cxf.ws.rm.RMCaptureInInterceptor** | 为持久性存储缓存传入的消息。 |
| **org.apache.cxf.ws.rm.RMDeliveryInterceptor** | 确保向应用程序发送信息。 |
| **org.apache.cxf.ws.rm.soap.RMSoapInterceptor** | 负责编码和解码可靠性属性作为 SOAP 标头。 |
| **org.apache.cxf.ws.rm.RetransmissionInterceptor** | 负责为将来的重新发送创建应用消息副本。 |

### 启用 WS-RM

在拦截器链中存在 WS-RM 拦截器可确保在需要时交换 WS-RM 协议消息。例如，当截获出站拦截器

链上的第一个应用程序消息时，RMOutInterceptor 会发送 CreateSequence 请求并等待处理原始应用程序消息，直到它收到 CreateSequenceResponse 响应。此外，WS-RM 拦截器将序列标头添加到应用程序消息中，并在目标端从消息中提取它们。不需要对应用程序代码进行任何更改，从而使消息交换可靠。

有关如何启用 WS-RM 的更多信息，请参阅 第 21.3 节 “启用 WS-RM”。

### 配置 WS-RM 属性

通过配置，您可以控制序列解译和其他可靠的交换方面。例如，默认情况下，Apache CXF 会尝试最大化序列的生命周期，从而减少带 WS-RM 协议消息产生的开销。要强制在每个应用消息中使用单独的序列，请配置 WS-RM 源的序列终止策略（将最大序列号设置为 1）。

有关配置 WS-RM 行为的详情，请参考 第 21.5 节 “配置 WS-RM”。

## 21.3. 启用 WS-RM

### 概述

要启用可靠的消息传递，必须将 WS-RM 拦截器添加到用于入站和出站消息和故障的拦截器链中。由于 WS-RM 拦截器使用 WS-Addressing，因此 WS-地址拦截器还必须出现在拦截器链中。

您可以通过以下两种方式之一来确保存在这些拦截器：

- 显式 使用 Spring Bean 将它们添加到分配链中
- 使用 WS-Policy 断言隐式，这会导致 Apache CXF 运行时代表您透明地添加拦截器。???

### Spring beans：显式添加拦截器

要启用 WS-RM 将 WS-RM 和 WS-Addressing 拦截器添加到 Apache CXF 总线，或使用 Spring bean 配置添加到消费者或服务端点。这是在 WS-RM 示例中使用的方法，可在 *InstallDir*/samples/ws_rm 目录中找到。配置文件 ws-rm.cxf 显示 WS-RM 和 WS-Addressing 拦截器被添加一次为 Spring Bean （请参阅 例 21.1 “使用 Spring Beans 启用 WS-RM”）。

**例 21.1. 使用 Spring Beans 启用 WS-RM**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
  beans http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="mapAggregator" class="org.apache.cxf.ws.addressing.MAPAggregator"/>
  <bean id="mapCodec" class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
  <bean id="rmLogicalOut" class="org.apache.cxf.ws.rm.RMOutInterceptor">
      <property name="bus" ref="cxf"/>
  </bean>
  <bean id="rmLogicalIn" class="org.apache.cxf.ws.rm.RMInInterceptor">
      <property name="bus" ref="cxf"/>
  </bean>
  <bean id="rmCodec" class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>
  <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
      <property name="inInterceptors">
         <list>
            <ref bean="mapAggregator"/>
            <ref bean="mapCodec"/>
            <ref bean="rmLogicalIn"/>
            <ref bean="rmCodec"/>
         </list>
      </property>
      <property name="inFaultInterceptors">
         <list>
            <ref bean="mapAggregator"/>
            <ref bean="mapCodec"/>
            <ref bean="rmLogicalIn"/>
            <ref bean="rmCodec"/>
         </list>
      </property>
      <property name="outInterceptors">
         <list>
            <ref bean="mapAggregator"/>
            <ref bean="mapCodec"/>
            <ref bean="rmLogicalOut"/>
            <ref bean="rmCodec"/>
         </list>
      </property>
      <property name="outFaultInterceptors">
         <list>
            <ref bean="mapAggregator">
            <ref bean="mapCodec"/>
            <ref bean="rmLogicalOut"/>
            <ref bean="rmCodec"/>
         </list>
      </property>
   </bean>
</beans>
```

例 21.1 “使用 Spring Beans 启用 WS-RM” 中显示的代码如下：

Apache CXF 配置文件是一个 Spring XML 文件。您必须包含一个 open Spring beans 元素，用于声明由 beans 元素封装的子元素的命名空间和 schema 文件。

配置每个 WS-Addressing interceptors-MAPAggregator 和 MAPCodec。有关 WS-地址的更多信息，请参阅 第 20 章 *部署 WS-Addressing*。

配置每个 WS-RM 拦截器-RMOutInterceptor、RMInInterceptor 和 RMSoapInterceptor。

将 WS-Addressing 和 WS-RM 拦截器添加到入站消息的拦截器链中。

将 WS-Addressing 和 WS-RM 拦截器添加到用于入站故障的拦截器链中。

将 WS-Addressing 和 WS-RM 拦截器添加到用于出站消息的拦截器链中。

将 WS-Addressing 和 WS-RM 拦截器添加到拦截器链中以进行出站故障。

**WS-Policy 框架：隐式添加拦截器**

WS-Policy 框架提供基础架构和 API，允许您使用 WS-Policy。它符合 2006 年 11 月发布 Web 服务政策 1.5-Framework 和 Web Services Policy 1.5-Attachment 规范的发布。

要使用 Apache CXF WS-Policy 框架启用 WS-RM，请执行以下操作：

1. 将策略功能添加到您的客户端和服务器端点。例 21.2 “使用 WS-Policy 配置 WS-RM” 显示嵌套在 jaxws:feature 元素内的参考 bean。reference bean 指定 AddressingPolicy，它定义为同一配置文件中的单独元素。

**例 21.2. 使用 WS-Policy 配置 WS-RM**

```
<jaxws:client>
  <jaxws:features>
    <ref bean="AddressingPolicy"/>
  </jaxws:features>
```

```
</jaxws:client>
<wsp:Policy wsu:Id="AddressingPolicy"
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
   <wsam:Addressing>
    <wsp:Policy>
      <wsam:NonAnonymousResponses/>
    </wsp:Policy>
   </wsam:Addressing>
</wsp:Policy>
```

2. **向 wsdl:service 元素或任何其他 WSDL 元素添加可靠的消息传递策略，这些元素可用作策略或策略参考元素的附件点 - 到 WSDL 文件，如 例 21.3 “将 RM 策略添加到您的 WSDL 文件中” 所示。**

**例 21.3. 将 RM 策略添加到您的 WSDL 文件中**

```
<wsp:Policy wsu:Id="RM"
  xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd">
   <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
      <wsp:Policy/>
   </wsam:Addressing>
   <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
      <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
   </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
   <wsdl:port binding="tns:GreeterSOAPBinding" name="GreeterPort">
      <soap:address location="http://localhost:9020/SoapContext/GreeterPort"/>
      <wsp:PolicyReference URI="#RM" xmlns:wsp="http://www.w3.org/2006/07/ws-
policy"/>
   </wsdl:port>
</wsdl:service>
```

## 21.4. 运行时控制

### 概述

**可以在客户端代码中设置几个消息上下文属性值，以控制运行时的 WS-RM，在 org.apache.cxf.ws.rm.RMManager 类中由公共常量定义键值。**

### 运行时控制选项

下表列出了 org.apache.cxf.ws.rm.RMManager 类定义的键。

| 键 | 描述 |
| --- | --- |
| WSRM_VERSION_PROPERTY | 字符串 WS-RM 版本命名空间 (http://schemas.xmlsoap.org/ws/2005/02/rm/ 或 http://docs.oasis-open.org/ws-rx/wsrm/200702)。 |
| WSRM_WSA_VERSION_PROPERTY | 字符串 WS-Addressing 版本命名空间 (http://schemas.xmlsoap.org/ws/2004/08/addressing 或 http://www.w3.org/2005/08/addressing)- 除非使用 http://schemas.xmlsoap.org/ws/2005/02/rm/ RM 命名空间，否则会忽略此属性。 |
| WSRM_LAST_MESSAGE_PROPERTY | 布尔值 true 告知 WS-RM 代码正在发送的最后一个消息，允许代码关闭 WS-RM 序列和发布资源（从 3.0.0 版本为 CXF ），WS-RM 默认关闭 RM 序列。 |
| WSRM_INACTIVITY_TIMEOUT_PROPERTY | 以毫秒为单位的长时间不活跃超时。 |
| WSRM_RETRANSMISSION_INTERVAL_PROPERTY | 长时间的基本重新传输间隔（以毫秒为单位）。 |
| WSRM_EXPONENTIAL_BACKOFF_PROPERTY | 布尔值 exponential back-off 标志。 |
| WSRM_ACKNOWLEDGEMENT_INTERVAL_PROPERTY | 较长的确认间隔（以毫秒为单位）。 |

### 通过 JMX 控制 WS-RM

您还可以使用 Apache CXF 的 JMX 管理功能监控和控制 WS-RM 的很多方面。JMX 操作的完整列表由 org.apache.cxf.ws.rm.ManagedRMManager 和 org.apache.cxf.ws.rm.ManagedRMEndpoint 定义，但这些操作包括查看单个消息级别的当前 RM 状态。您还可以使用 JXM 关闭或终止 WS-RM 序列，并接收之前由远程 RM 端点确认的消息的通知。

### JMX 控制示例

例如，如果您在客户端配置中启用了 JMX 服务器，您可以使用以下代码来跟踪收到的最后确认号：

```
// Java
private static class AcknowledgementListener implements NotificationListener {
```

```
    private volatile long lastAcknowledgement;

    @Override
    public void handleNotification(Notification notification, Object handback) {
        if (notification instanceof AcknowledgementNotification) {
            AcknowledgementNotification ack = (AcknowledgementNotification)notification;
            lastAcknowledgement = ack.getMessageNumber();
        }
    }

    // initialize client
...
    // attach to JMX bean for notifications
    //  NOTE: you must have sent at least one message to initialize RM before executing this code
    Endpoint ep = ClientProxy.getClient(client).getEndpoint();
    InstrumentationManager im = bus.getExtension(InstrumentationManager.class);
    MBeanServer mbs = im.getMBeanServer();
    RMManager clientManager = bus.getExtension(RMManager.class);
    ObjectName name = RMUtils.getManagedObjectName(clientManager, ep);
    System.out.println("Looking for endpoint name " + name);
    AcknowledgementListener listener = new AcknowledgementListener();
    mbs.addNotificationListener(name, listener, null, null);

    // send messages using RM with acknowledgement status reported to listener
...
```

## 21.5. 配置 WS-RM

### 21.5.1. 配置 Apache CXF-Specific WS-RM 属性

#### 概述

要配置特定于 Apache CXF 的属性，请使用 rmManager Spring bean。在配置文件中添加以下内容：

- 到命名空间列表的 http://cxf.apache.org/ws/rm/manager 命名空间。
- 您要配置的特定属性的 rmManager Spring bean。

例 21.4 “配置 Apache CXF-Specific WS-RM 属性” 显示一个简单的示例。

**例 21.4. 配置 Apache CXF-Specific WS-RM 属性**

```
&lt;beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager http://cxf.apache.org/schemas/configuration/wsrm-
manager.xsd"&gt;
...
&lt;wsrm-mgr:rmManager&gt;
&lt;!--
  ...Your configuration goes here
--&gt;
&lt;/wsrm-mgr:rmManager&gt;
```

**rmManager Spring bean 的子项**

**表 21.2 “rmManager Spring Bean 的子项” 显示 http://cxf.apache.org/ws/rm/manager 命名空间中定义的 rmManager Spring bean 的子元素。**

**表 21.2. rmManager Spring Bean 的子项**

| 元素 | 描述 |
| --- | --- |
| **RMAssertion** | 一个类型 **RMAssertion**的元素 |
| **deliveryAssurance** | 描述应应用的交付保证的 **DeliveryAssuranceType** 的一个元素 |
| **sourcePolicy** | 一个 **SourcePolicyType** 类型的元素，允许您配置 RM 源的详情 |
| **destinationPolicy** | 一个类型为 **DestinationPolicyType** 的元素，允许您配置 RM 目的地的详情 |

**Example**

**例如，请参阅 “最大未确认的消息阈值”一节。**

### 21.5.2. 配置标准 WS-RM 策略属性

**概述**

**您可以使用以下方法之一配置标准 WS-RM 策略属性：**

- “rmManager Spring bean 中的 RMAssertion”一节
- “功能中的策略”一节
- “WSDL 文件”一节
- “外部附加”一节

### WS-Policy RMAssertion Children

表 21.3 “WS-Policy RMAsertion 元素的子项” 显示 http://schemas.xmlsoap.org/ws/2005/02/rm/policy 命名空间中定义的元素：

**表 21.3. WS-Policy RMAsertion 元素的子项**

| Name | 描述 |
| --- | --- |
| **InactivityTimeout** | 指定在端点可以认为 RM 序列因为不活跃而终止前必须传递的时间长度。 |
| **BaseRetransmissionInterval** | 为给定消息设置 RM Source 必须接收确认的时间间隔。如果在 **BaseRetransmissionInterval** 设置的时间内未收到确认，RM 源将重新传输消息。 |
| **ExponentialBackoff** | 指示将使用通常已知的 exponential backoff 算法 (Tanenbaum)调整重新传输间隔。<br><br>如需更多信息，请参阅 计算机网络、Andrew S. Tanenbaum、Prentice Hall PTR、2003。 |
| **AcknowledgementInterval** | 在 WS-RM 中，在返回消息或独立发送时发送确认。如果返回消息无法发送确认，则 RM Destination 可以在发送独立确认前最多等待确认间隔。如果没有未确认的信息，则 RM Destination 可以选择不发送确认信息。 |

### 更详细的参考信息

有关更详细的参考信息，包括每个元素的子元素和属性的描述，请参阅 http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd。

**rmManager Spring bean 中的 RMAssertion**

您可以通过在 Apache CXF rmManager Spring bean 中添加 RMAssertion 来配置标准 WS-RM 策略属性。如果您要在同一配置文件中保留所有 WS-RM 配置，这是最佳方法；也就是说，如果您想要在同一文件中配置特定于 Apache CXF 的属性和标准 WS-RM 策略属性。

例如，例 21.5 “在 rmManager Spring Bean 中使用 RMAssertion 配置 WS-RM 属性” 中的配置显示：

- 标准 WS-RM 策略属性 BaseRetransmissionInterval，使用 rmManager Spring bean 中的 RMAsertion 配置。
- 特定于 Apache CXF 的 RM 属性，在同一配置文件中配置。

**例 21.5. 在 rmManager Spring Bean 中使用 RMAssertion 配置 WS-RM 属性**

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
      xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
   <wsrm-policy:RMAssertion>
      <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
   </wsrm-policy:RMAssertion>
   <wsrm-mgr:destinationPolicy>
      <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
   </wsrm-mgr:destinationPolicy>
</wsrm-mgr:rmManager>
</beans>
```

**功能中的策略**

您可以在功能中配置标准 WS-RM 策略属性，如 例 21.6 “将 WS-RM 属性配置为功能内的策略” 所示。

**例 21.6. 将 WS-RM 属性配置为功能内的策略**

```
<xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
```

```
        xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort"
createdFromAPI="true">
        <jaxws:features>
            <wsp:Policy>
                <wsrm:RMAssertion
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                  <wsrm:AcknowledgementInterval Milliseconds="200" />
                </wsrm:RMAssertion>
                <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                    <wsp:Policy>
                        <wsam:NonAnonymousResponses/>
                    </wsp:Policy>
                </wsam:Addressing>
            </wsp:Policy>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

**WSDL 文件**

**如果您使用 WS-Policy 框架启用 WS-RM，您可以在 WSDL 文件中配置标准 WS-RM 策略属性。如果您希望服务与部署到其他策略感知型 Web 服务堆栈的用户无缝地交互和使用 WS-RM，这是良好的方法。**

**例如，请参阅 “WS-Policy 框架：隐式添加拦截器”一节，在 WSDL 文件中配置基本重新传输间隔。**

**外部附加**

**您可以在外部附加文件中配置标准 WS-RM 策略属性。如果您无法或不想更改 WSDL 文件，这是很好的方法。**

**例 21.7 “在外部附加中配置 WS-RM” 显示一个外部附件，为特定 EPR 启用 WS-A 和 WS-RM （基本重新传输间隔 30 秒）。**

**例 21.7. 在外部附加中配置 WS-RM**

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsp:PolicyAttachment>
        <wsp:AppliesTo>
```

```
            <wsa:EndpointReference>
                <wsa:Address>http://localhost:9020/SoapContext/GreeterPort</wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:Policy>
            <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                <wsp:Policy/>
            </wsam:Addressing>
            <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                <wsrmp:BaseRetransmissionInterval Milliseconds="30000"/>
            </wsrmp:RMAssertion>
        </wsp:Policy>
    </wsp:PolicyAttachment>
</attachments>/
```

### 21.5.3. WS-RM 配置用例

**概述**

**本小节重点介绍从用例角度配置 WS-RM 属性。如果属性是 http://schemas.xmlsoap.org/ws/2005/02/rm/policy/ 命名空间中定义的标准 WS-RM 策略属性，则仅显示在 rmManager Spring bean 中的 RMAssertion 中设置它的示例。有关如何在功能中设置此类属性作为策略的详情；在 WSDL 文件或外部附加中，请参阅 第 21.5.2 节 “配置标准 WS-RM 策略属性”。**

**涵盖了以下用例：**

- **“基本重新传输间隔”一节**
- **“重新传输的指数 backoff”一节**
- **“确认间隔”一节**
- **“最大未确认的消息阈值”一节**
- **“RM 序列的最大长度”一节**
- **“消息交付保证策略”一节**

基本重新传输间隔

BaseRetransmissionInterval 元素指定 RM 源重新传输了尚未确认的消息的时间间隔。它在 http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd 模式文件中定义。默认值为 3000 毫秒。

例 21.8 “设置 WS-RM Base Retransmission Interval” 显示如何设置 WS-RM 基础重新传输间隔。

**例 21.8. 设置 WS-RM Base Retransmission Interval**

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

重新传输的指数 backoff

ExponentialBackoff 元素决定了是否以指数级间隔执行非确认消息尝试连续重新传输。

存在 ExponentialBackoff 元素可启用此功能。默认使用 exponential backoff 比率 2。ExponentialBackoff 是一个标志。当元素存在时，启用了 exponential backoff。当元素不存在时，会禁用 exponential backoff。不需要值。

例 21.9 “设置 WS-RM Exponential backoff 属性” 显示如何为重新传输设置 WS-RM 指数 backoff。

**例 21.9. 设置 WS-RM Exponential backoff 属性**

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:ExponentialBackoff/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

确认间隔

AcknowledgementInterval 元素指定 WS-RM 目标发送异步确认的时间间隔。除了在接收传入消息时发送的同步确认之外。默认异步确认间隔为 0 毫秒。这意味着，如果 AcknowledgementInterval 没有配置为特定值，则确认将立即发送（即，第一个可用机会）。

只有满足以下条件时，RM 目的地才会发送异步确认：

- RM 目的地使用非匿名 wsrm:acksTo 端点。
- 在确认间隔到期之前，不会发生对响应消息的确认机会。

例 21.10 “设置 WS-RM 确认间隔” 显示如何设置 WS-RM 确认间隔。

**例 21.10. 设置 WS-RM 确认间隔**

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

**最大未确认的消息阈值**

maxUnacknowledged 属性设置在序列终止前每个序列可以意外处理的最大未确认消息数。

例 21.11 “设置 WS-RM Maximum Unacknowledged Message Threshold” 显示如何设置 WS-RM 最大值未确认的消息阈值。

**例 21.11. 设置 WS-RM Maximum Unacknowledged Message Threshold**

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

**RM 序列的最大长度**

**maxLength 属性设置 WS-RM 序列的最大长度。默认值为 0，这意味着 WS-RM 序列的长度未绑定。**

**当设置此属性时，RM 端点会在达到限制时创建一个新的 RM 序列，并在收到之前发送消息的所有确认后。新消息使用新的sequence 发送。**

**例 21.12 “设置 WS-RM 消息序列的最大长度”** 显示如何设置 RM 序列的最大长度。

**例 21.12. 设置 WS-RM 消息序列的最大长度**

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

**消息交付保证策略**

**您可以将 RM 目的地配置为使用以下交付保证策略：**

- **AtMostOnce - RM 目的地仅向应用程序目的地发送一次消息。如果消息被发送多次，则会引发错误。序列中的一些消息可能无法发送。**
- **AtLeastOnce - RM 目的地至少向应用程序目的地发送一次。发送的每个消息都会被发送，否则将引发错误。有些消息可能会多次发送。**
- **InOrder - RM 目的地按发送的顺序将消息发送到应用程序目的地。这种交付保证可以与 AtMostOnce 或 AtLeastOnce 保证结合使用。**

**例 21.13 “设置 WS-RM 消息交付保证策略”** 显示如何设置 WS-RM 消息交付保证。

**例 21.13. 设置 WS-RM 消息交付保证策略**

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
   <wsrm-mgr:deliveryAssurance>
      <wsrm-mgr:AtLeastOnce />
   </wsrm-mgr:deliveryAssurance>
</wsrm-mgr:reliableMessaging>
</beans>
```

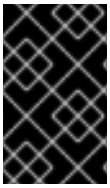
## 21.6. 配置 WS-RM PERSISTENCE

**概述**

**本章中已描述的 Apache CXF WS-RM 功能为网络故障等情况提供了可靠性。WS-RM 持久性在其他类型的故障之间提供可靠性，如 RM 源或 RM 目的地崩溃。**

**WS-RM 持久性涉及将各种 RM 端点的状态存储在持久性存储中。这可让端点在重新发送时继续发送和接收信息。**

**Apache CXF 在配置文件中启用 WS-RM 持久性。默认的 WS-RM 持久性存储是基于 JDBC。为方便起见，Apache CXF 包括用于开箱即用部署的 Derby。此外，持久存储也使用 Java API 公开。要实施自己的持久性机制，您可以将这个 API 与首选 DB 搭配使用。**



**重要**

**WS-RM 持久性仅支持单向调用，它默认是禁用的。**

**它如何工作**

**Apache CXF WS-RM 持久性可以正常工作：**

- **在 RM 源端点上，传出消息会在传输前保留。在收到确认后，它会从持久性存储中驱除。**
- **恢复崩溃后，它会恢复保留的消息并重新传输，直到所有消息都已确认。此时，RM 序列关闭。**

- 在 RM 目的地端点上，会保留传入的消息，并在成功存储后发送确认。成功分配消息时，它会从持久性存储中驱除。

- 从崩溃中恢复后，它会恢复保留的消息并发送它们。它还会将 RM 序列进入接受、确认和发送新消息的状态。

### 启用 WS-persistence

要启用 WS-RM 持久性，您必须为 WS-RM 指定实施持久性存储的对象。您可以自行开发，也可以使用 Apache CXF 附带的基于 JDBC 的存储。

例 21.14 “配置默认 WS-RM Persistence 存储” 中显示的配置启用了 Apache CXF 附带的基于 JDBC 的存储。

**例 21.14. 配置默认 WS-RM Persistence 存储**

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore"/>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <property name="store" ref="RMTxStore"/>
</wsrm-mgr:rmManager>
```

### 配置 WS-persistence

Apache CXF 附带的基于 JDBC 的存储支持 表 21.4 “JDBC 存储属性” 中显示的属性。

**表 21.4. JDBC 存储属性**

| 属性名称 | 类型 | 默认设置 |
| --- | --- | --- |
| driverClassName | 字符串 | org.apache.derby.jdbc.EmbeddedDriver |
| userName | 字符串 | null |
| passWord | 字符串 | null |
| url | 字符串 | jdbc:derby:rmdb;create=true |

例 21.15 “为 WS-RM 持久性配置 JDBC 存储” 中显示的配置启用了 Apache CXF 附带的基于 JDBC 的存储，同时将 driverClassName 和 url 设置为非默认值。

**例 21.15. 为 WS-RM 持久性配置 JDBC 存储**

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore">
    <property name="driverClassName" value="com.acme.jdbc.Driver"/>
    <property name="url" value="jdbc:acme:rmdb;create=true"/>
</bean>
```

# 第 22 章 启用高可用性

## 摘要

本章解释了如何在 Apache CXF 运行时中启用和配置高可用性。

## 22.1. 高可用性简介

### 概述

可扩展且可靠的应用程序需要高可用性，以避免分布式系统中出现单点故障。您可以使用以下方法保护您的系统免受单点故障 *复制的服务*。

复制的服务由同一服务的多个实例或 *副本* 组成。它们整合为一个逻辑服务。客户端在复制服务上调用请求，Apache CXF 将请求发送到其中一个成员副本。到副本的路由对客户端是透明的。

### 带有静态故障切换的 HA

Apache CXF 支持带有静态故障切换的高可用性(HA)，其中副本详情会在服务 WSDL 文件中编码。WSDL 文件包含多个端口，并且可以为同一服务包含多个主机。只要 WSDL 文件保持不变，集群中的副本数量仍保持静态。更改集群大小涉及编辑 WSDL 文件。

## 22.2. 使用静态故障切换启用 HA

### 概述

要使用静态故障切换启用 HA，您必须执行以下操作：

1. “在服务 WSDL 文件中对副本的详细信息进行编码”一节
2. “在客户端配置中添加集群功能”一节

### 在服务 WSDL 文件中对副本的详细信息进行编码

您必须在服务 WSDL 文件中对集群中的副本的详细信息进行编码。例 22.1 “使用静态故障切换启用 HA：WSDL 文件” 显示定义三个副本的服务集群的 WSDL 文件提取。

**例 22.1. 使用静态故障切换启用 HA：WSDL 文件**

```
<wsdl:service name="ClusteredService">
   <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica1">
      <soap:address location="http://localhost:9001/SoapContext/Replica1"/>
   </wsdl:port>

   <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica2">
      <soap:address location="http://localhost:9002/SoapContext/Replica2"/>
   </wsdl:port>

   <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica3">
      <soap:address location="http://localhost:9003/SoapContext/Replica3"/>
   </wsdl:port>

</wsdl:service>
```

例 22.1 “使用静态故障切换启用 HA：WSDL 文件” 中显示的 WSDL 提取如下所示：

定义在三个端口上公开的服务 **ClusterService** ：

1. **Replica1**
2. **Replica2**
3. **Replica3**

定义 **Replica1**，以通过端口 **9001** 上的 **HTTP** 端点公开 **ClusterService**。

定义 **Replica2**，以通过端口 **9002** 上的 **HTTP** 端点公开 **ClusterService**。

定义 **Replica3**，以通过端口 **9003** 上的 **HTTP** 端点公开 **ClusterService**。

在客户端配置中添加集群功能

在客户端配置文件中，添加集群功能，如 例 22.2 “使用静态故障切换启用 HA：客户端配置” 所示。

**例 22.2. 使用静态故障切换启用 HA：客户端配置**

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

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica1"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica2"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica3"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

</beans>
```

## 22.3. 使用静态故障切换配置 HA

### 概述

默认情况下，如果客户端通信的原始服务不可用，或者失败，则具有静态故障转移的 HA 使用一个顺序策略。sequential 策略在每次使用时都以相同的顺序选择副本服务。选择由 Apache CXF 的内部服务模型决定，并产生确定的故障转移模式。

### 配置随机策略

您可以使用静态故障转移配置 HA，以便在选择副本时使用随机策略而不是后续策略。每次服务不可用时，随机策略都会选择一个随机副本服务，或者失败。从集群中的存活成员中选择故障转移目标完全是随机的。

要配置随机策略，请将 例 22.3 “为静态故障切换配置随机策略” 中显示的配置添加到您的客户端配置文件中。

**例 22.3. 为静态故障切换配置随机策略**

```
<beans ...>
    <bean id="Random" class="org.apache.cxf.clustering.RandomStrategy"/>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica3"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover>
                <clustering:strategy>
                    <ref bean="Random"/>
                </clustering:strategy>
            </clustering:failover>
        </jaxws:features>
    </jaxws:client>
</beans>
```

例 22.3 “为静态故障切换配置随机策略” 中显示的配置如下：

定义实施随机策略的 Random bean 和实施类。

指定在选择副本时使用随机策略。

# 第 23 章 APACHE CXF 绑定 ID

## 绑定 ID 表

**表 23.1. 消息绑定的绑定 ID**

| 绑定 | ID |
| --- | --- |
| CORBA | http://cxf.apache.org/bindings/corba |
| HTTP/REST | http://apache.org/cxf/binding/http |
| SOAP 1.1 | http://schemas.xmlsoap.org/wsdl/soap/http |
| SOAP 1.1 w/ MTOM | http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true |
| SOAP 1.2 | http://www.w3.org/2003/05/soap/bindings/HTTP/ |
| SOAP 1.2 w/ MTOM | http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true |
| XML | http://cxf.apache.org/bindings/xformat |

# 附录 A. 使用 MAVEN OSGI 工具

**摘要**

为大型项目手动创建捆绑包或捆绑包集合可能很繁琐。Maven 捆绑包插件通过自动化进程并提供多个指定捆绑包清单内容的快捷方式，使作业变得更加容易。

## A.1. MAVEN 捆绑包插件

红帽 Fuse OSGi 工具使用 Apache Felix 中的 Maven 捆绑包插件。捆绑插件基于 Peter Kriens 中的 bnd 工具。它通过内省捆绑包中打包的类内容来自动化 OSGi 捆绑包清单的结构。利用捆绑包中包含的类知识，插件可以计算正确的值，以填充捆绑包清单中的 Import-Packages 和 Export-Package 属性。该插件也具有默认值，用于捆绑包清单中的其他必要属性。

要使用 bundle 插件，请执行以下操作：

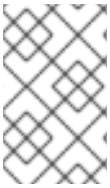
1. 第 A.2 节 “设置红帽 Fuse OSGi 项目” 项目的 POM 文件的捆绑包插件。
2. 第 A.3 节 “配置捆绑包插件” 用于正确填充捆绑包清单的插件。

## A.2. 设置红帽 FUSE OSGI 项目

**概述**

用于构建 OSGi 捆绑包的 Maven 项目可以是一个简单的单级别项目。它不需要任何子项目。但是，它要求您执行以下操作：

1. 将 捆绑包插件添加到 POM 中。
2. 指示 Maven 将结果打包为 OSGi 捆绑包。



**注意**

您可以使用几个 Maven archetypes 设置项目，使用适当的设置。

目录结构

构建 OSGi 捆绑包的项目可以是单个级别项目。它仅要求您有一个顶级 POM 文件和 src 文件夹。与所有 Maven 项目中一样，您可以将所有 Java 源代码放在 src/java 文件夹中，并将任何非 Java 资源放在 src/resources 文件夹中。

非 Java 资源包括 Spring 配置文件、JBI 端点配置文件和 WSDL 合同。

注意

使用 Apache CXF、Apache Camel 或其他 Spring 配置的 bean 的 Red Hat Fuse OSGi 项目还包括一个 beans.xml 文件，位于 src/resources/META-INF/spring 文件夹中。

添加捆绑包插件

在使用捆绑包插件前，您必须添加对 Apache Felix 的依赖。添加依赖项后，您可以将捆绑包插件添加到 POM 的插件部分中。

例 A.1 “将 OSGi 捆绑插件添加到 POM 中” 显示将捆绑包插件添加到项目所需的 POM 条目。

**例 A.1. 将 OSGi 捆绑插件添加到 POM 中**

```
...
<dependencies>
  <dependency>
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
          <Import-Package>*,org.apache.camel.osgi</Import-Package>
          <Private-Package>org.apache.servicemix.examples.camel</Private-Package>
        </instructions>
      </configuration>
```

```
    </plugin>
  </plugins>
</build>
...
```

例 A.1 “将 OSGi 捆绑插件添加到 POM 中” 中的条目执行以下操作：

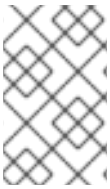
添加对 Apache Felix 的依赖性

将捆绑包插件添加到项目中

配置插件，以使用项目的工件 ID 作为捆绑包的符号名称

配置插件，使其包含捆绑类导入的所有 Java 软件包；也导入 org.apache.camel.osgi 软件包

配置插件以捆绑列出的类，但不将它们包含在导出的软件包列表中



注意

编辑配置，以满足您的项目要求。

有关配置捆绑包插件的详情，请参考 第 A.3 节 “配置捆绑包插件”。

激活捆绑包插件

要让 Maven 使用捆绑包插件，请指示它将项目的结果打包为捆绑包。为此，可将 POM 文件的 packaging 元素设置为 捆绑包。

有用的 Maven archetypes

有几个 Maven archetypes 可用来生成预先配置为使用捆绑包插件的项目：

- “Spring OSGi archetype”一节
- “Apache CXF code-first archetype”一节
- “Apache CXF wsdl-first archetype”一节
- “Apache Camel archetype”一节

### Spring OSGi archetype

**Spring OSGi archetype 创建一个通用项目，用于使用 Spring DM 构建 OSGi 项目，如下所示：**

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

**您可以使用以下命令调用 archetype ：**

```
mvn archetype:generate -DarchetypeGroupId=org.springframework.osgi -
DarchetypeArtifactId=spring-osgi-bundle-archetype -DarchetypeVersion=1.1.2 -DgroupId=groupId -
DartifactId=artifactId -Dversion=version
```

### Apache CXF code-first archetype

**Apache CXF code-first archetype 创建一个项目，用于从 Java 构建服务，如下所示：**

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2010.02.0-fuse-02-00
```

**您可以使用以下命令调用 archetype ：**

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-cxf-code-first-archetype -DarchetypeVersion=2010.02.0-fuse-
02-00 -DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

### Apache CXF wsdl-first archetype

**Apache CXFdl-first archetype 创建一个项目，用于从 WSDL 创建服务，如下所示：**

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2010.02.0-fuse-02-00
```

**您可以使用以下命令调用 archetype ：**

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype -DarchetypeVersion=2010.02.0-fuse-
02-00 -DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

**Apache Camel archetype**

**Apache Camel archetype 创建一个项目来构建部署到红帽 Fuse 中的路由，如下所示：**

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2010.02.0-fuse-02-00
```

**您可以使用以下命令调用 archetype ：**

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-camel-archetype -DarchetypeVersion=2010.02.0-fuse-02-00 -
DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

### A.3. 配置捆绑包插件

**概述**

**捆绑包插件需要很少的信息才能正常工作。所有必要属性都使用默认设置来生成有效的 OSGi 捆绑包。**

**虽然您只能使用默认值创建有效的捆绑包，但您可能需要修改一些值。您可以在插件的 instructions 元素中指定大多数属性。**

**配置属性**

**一些常用的配置属性是：**

- **Bundle-SymbolicName**

- bundle-Name
- bundle-Version
- export-Package
- private-Package
- import-Package

设置捆绑包的符号名称

默认情况下，捆绑包插件将 Bundle-SymbolicName 属性的值设置为 *groupId* + "." + *artifactId*，但有以下例外：

- 如果 *groupId* 只有一个部分（没有点），则返回第一个带有类的软件包名称。

  例如，如果组 Id 是 commons-logging:commons-logging，则捆绑包的符号名称为 org.apache.commons.logging。

- 如果 *artifactId* 等于 *groupId* 的最后部分，则使用 *groupId*。

  例如，如果 POM 将组 ID 和工件 ID 指定为 org.apache.maven:maven，则捆绑包的符号名称为 org.apache.maven。

- 如果 *artifactId* 以 *groupId* 的最后部分开头，则会删除该部分。

  例如，如果 POM 将组 ID 和工件 ID 指定为 org.apache.maven:maven-core，则捆绑包的符号名称为 org.apache.maven.core。

要为捆绑包的符号名称指定您自己的值，请在插件的 instructions 元素中添加一个 Bundle-SymbolicName 子级，如 例 A.2 “设置捆绑包的符号名称” 所示。

**例 A.2.** 设置捆绑包的符号名称

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

设置捆绑包名称

默认情况下，捆绑包的名称设置为 **${project.name}**。

要为捆绑包的名称指定您自己的值，请在插件的 **instructions** 元素中添加一个 **Bundle-Name** 子对象，如 例 A.3 “设置捆绑包名称” 所示。

**例 A.3.** 设置捆绑包名称

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Name>JoeFred</Bundle-Name>
     ...
    </instructions>
  </configuration>
</plugin>
```

设置捆绑包的版本

默认情况下，捆绑包的版本设置为 **${project.version}**。任何短划线(-)替换为点(.)，数字会加到四位。例如，**4.2-SNAPSHOT** 变为 **4.2.0.SNAPSHOT**。

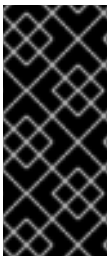
要为捆绑包的版本指定您自己的值，请在插件的 **instructions** 元素中添加一个 **Bundle-Version** 子对象，如 例 A.4 “设置捆绑包的版本” 所示。

**例 A.4.** 设置捆绑包的版本

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

**指定导出的软件包**

**默认情况下，OSGi 清单的 Export-Package 列表由您的本地 Java 源代码中的所有软件包填充（在 src/main/java 下），除了默认软件包.. 以及包含 .impl 或 .internal 的任何软件包。**



**重要**

**如果您在插件配置中使用 Private-Package 元素，且您没有指定要导出的软件包列表，则默认行为仅包含捆绑包中 Private-Package 元素中列出的软件包。没有导出软件包。**

**默认行为可能会导致软件包非常大，并导出应保持私有的软件包。要更改导出的软件包列表，您可以在插件的 instructions 元素中添加 Export-Package 子子。**

**Export-Package 元素指定要包含在捆绑包中的软件包列表，以及要导出的软件包列表。可以使用 * 通配符符号指定软件包名称。例如，条目 com.fuse.demo 需要包括以 com.fuse.demo 开头的项目的 classpath 中的所有软件包。**

**您可以使用 ! 指定要排除的软件包作为前缀。例如，条目 !com.fuse.demo.private 排除软件包 com.fuse.demo.private。**

**在排除软件包时，列表中条目的顺序非常重要。从开始顺序处理列表，并忽略后续字典条目。**

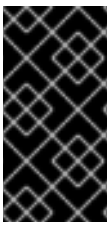
**例如，要包含以 com.fuse.demo 开头的所有软件包，软件包 com.fuse.demo.private 除外，请使用以下方法列出软件包：**

```
!com.fuse.demo.private,com.fuse.demo.*
```

但是，如果您使用 com.fuse.demofuse,!com.fuse.demo.private 列出软件包，那么 com.fuse.demo.private 将包含在捆绑包中，因为它与第一个模式匹配。

### 指定私有软件包

如果要在 不 导出捆绑包中指定要包含的软件包列表，您可以在捆绑插件配置中添加 Private-Package 指令。默认情况下，如果您没有指定 Private-Package 指令，则捆绑包中包含本地 Java 源中的所有软件包。



**重要**

如果软件包与 Private-Package 元素和 Export-Package 元素中的条目匹配，则 Export-Package 元素将具有优先权。软件包被添加到捆绑包中并导出。

Private -Package 元素的工作方式与您要包含在捆绑包中的软件包列表类似。bundle 插件使用列表来查找要包含在捆绑包中的项目的 classpath 中的所有类。这些软件包打包在捆绑包中，但不会导出（除非它们也由 Export-Package 指令选择）。

例 A.5 “在捆绑包中包含私有软件包” 显示在捆绑包中包含私有软件包的配置

**例 A.5. 在捆绑包中包含私有软件包**

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

### 指定导入的软件包

默认情况下，捆绑包插件使用由捆绑包内容引用的所有软件包列表填充 OSGi 清单的 Import-Package 属性。

虽然默认行为通常足以满足大多数项目，但您可能会找到您想导入没有自动添加到列表中的软件包的

实例。默认行为还会导致导入不需要的软件包。

要指定捆绑包要导入的软件包列表，请在插件的 instructions 元素中添加 Import-Package 子级。软件包列表的语法与 Export-Package 元素和 Private-Package 元素的语法相同。

重要

使用 Import-Package 元素时，插件不会自动扫描捆绑包的内容，以确定是否有所需的导入。要确保扫描捆绑包的内容，您必须在软件包列表中放置一个 * 作为最后一个条目。

例 A.6 “指定捆绑包导入的软件包” 显示指定捆绑包导入的软件包的配置

**例 A.6. 指定捆绑包导入的软件包**

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Import-Package>javax.jws, javax.wsdl, org.apache.cxf.bus, org.apache.cxf.bus.spring,
org.apache.cxf.bus.resource, org.apache.cxf.configuration.spring, org.apache.cxf.resource,
org.springframework.beans.factory.config, * </Import-Package>
      ...
   </instructions>
  </configuration>
</plugin>
```

更多信息

有关配置捆绑包插件的更多信息，请参阅：

- olink:OsgiDependencies/OsgiDependencies
- Apache Felix 文档
- LL Kriens 的 AQute Software Consultancy 网站

# 部分 V. 使用 JAX-WS 开发应用程序

本指南论述了如何使用标准 JAX-WS API 开发 Web 服务。

# 第 24 章 向上更新服务开发

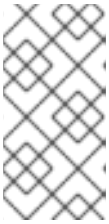
**摘要**

有许多实例已经实现了一组您要作为面向服务应用程序一部分公开的功能。您可能还想避免使用 WSDL 来定义您的接口。使用 JAX-WS 注释，您可以添加启用 Java 类所需的信息。您还可以创建可用于代替 WSDL 合同的服务端点 *接口* (SEI)。如果您需要 WSDL 合同，Apache CXF 提供了工具，以便从注释的 Java 代码生成合同。

## 24.1. JAX-WS 服务开发简介

要创建从 Java 开始的服务，您必须执行以下操作：

1. 第 24.2 节 “创建 SEI” 一个服务端点接口(SEI)，用于定义您要作为服务公开的方法。

   **注意**

   您可以直接从 Java 类工作，但建议使用接口。接口更适合与负责使用您的服务的应用程序的开发人员共享。接口更小，不提供任何服务的实施详情。

2. 第 24.3 节 “注解代码” 代码所需的注解。

3. 第 24.4 节 “生成 WSDL” 服务的 WSDL 合同。

   **注意**

   如果要使用 SEI 作为服务的合同，则不需要生成 WSDL 合同。

4. 第 31 章 *发布服务* 该服务作为服务提供商。

## 24.2. 创建 SEI

**概述**

*服务端点接口* (SEI)是服务实施和在该服务上发出请求的用户之间共享的 Java 代码片段。SEI 定义服务实施的方法，并提供了有关如何公开该服务作为端点的详细信息。从 WSDL 合同开始，SEI 由代码生成器生成。但是，从 Java 开始时，开发人员负责创建 SEI。 创建 SEI 有两个基本模式：

- 绿色字段开发 - 在这个模式中，您要在不现有的 Java 代码或 WSDL 的情况下开发新服务。最好先创建 SEI。然后，您可以将 SEI 分发到负责实施使用 SEI 的服务供应商和消费者的任何开发人员。

**注意**

进行绿色字段服务开发的建议方法是，创建定义服务及其接口的 WSDL 合同。请参阅 第 26 章 *开始点 WSDL 合同*。

- 服务启用 - 在这个模式中，您通常具有一组现有功能，它们作为 Java 类实施，您希望服务启用它。这意味着您必须执行两个操作：

a. 创建一个 SEI，它只包含将要作为服务一部分公开的操作。

b. 修改现有 Java 类，使其实施 SEI。

**注意**

虽然您可以将 JAX-WS 注释添加到 Java 类，但不推荐这样做。

**编写接口**

SEI 是一个标准 Java 接口。它定义类实施的一组方法。它还可以定义多个成员字段，并持续实施类有权访问。

如果是 SEI，定义的方法旨在映射到服务公开的操作。SEI 对应于 wsdl:portType 元素。SEI 定义的方法对应于 wsdl:portType 元素中的 wsdl:operation 元素。

**注意**

**JAX-WS 定义了一个注释，允许您指定没有作为服务的一部分公开的方法。但是，最佳实践是将这些方法离开 SEI。**

**例 24.1 “简单 SEI” 显示了用于库存更新服务的简单 SEI。**

**例 24.1. 简单 SEI**

```
package com.fusesource.demo;

public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

**实施接口**

**由于 SEI 是标准 Java 接口，因此实施它的类是标准 Java 类。如果从 Java 类开始，您必须修改它才能实现接口。如果您从 SEI 开始，实施类会实施 SEI。**

**例 24.2 “简单实施类” 显示了在 例 24.1 “简单 SEI” 中实现接口的类。**

**例 24.2. 简单实施类**

```
package com.fusesource.demo;

import java.util.*;

public class stockQuoteReporter implements quoteReporter
{
  ...
public Quote getQuote(String ticker)
  {
    Quote retVal = new Quote();
    retVal.setID(ticker);
    retVal.setVal(Board.check(ticker));[1]
    Date retDate = new Date();
    retVal.setTime(retDate.toString());
    return(retVal);
  }
}
```

## 24.3. 注解代码

### 24.3.1. JAX-WS 注释概述

JAX-WS 注释指定用于将 SEI 映射到完全指定的服务定义的元数据。注解中提供的信息包括：

- 服务的目标命名空间。
- 用于保存请求消息的类名称
- 用于保存响应消息的类名称
- 如果操作是一个方法
- 服务使用的绑定风格
- 用于任何自定义例外的类名称
- 定义服务使用的类型的命名空间

**注意**

大多数注解都有明显的默认值，不需要为其提供值。但是，您在注解中提供的更多信息，您的服务定义越好。指定良好的服务定义会增加分布式应用程序的所有部分将一起工作的可能性。

### 24.3.2. 所需的注解

**概述**

要从 Java 代码创建服务，您只需要为您的代码添加一个注解。您必须在 SEI 和实施类上添加 @WebService 注释。

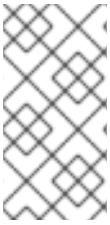
**@WebService 注释**

@WebService 注释由 javax.jws.WebService 接口定义，它放置在旨在用作服务的接口或类上。@WebService 具有中描述的属性 表 24.1 “@WebService Properties”

表 24.1. @WebService Properties

| 属性 | 描述 |
| --- | --- |
| **name** | 指定服务接口的名称。此属性映射到 **wsdl:portType** 元素的 **name** 属性，该元素定义 WSDL 合同中的服务接口。默认为将 **PortType** 附加到实现类的名称中。[a] |
| **targetNamespace** | 指定定义该服务的目标命名空间。如果没有指定此属性，则目标命名空间派生自软件包名称。 |
| **serviceName** | 指定发布的服务的名称。此属性映射到定义已发布服务的 **wsdl:service** 元素的 **name** 属性。默认为使用服务的实施类的名称。 |
| **wsdlLocation** | 指定存储服务的 WSDL 合同的 URL。这必须使用相对 URL 指定。默认为部署该服务的 URL。 |
| **endpointInterface** | 指定实现类实现的 SEI 的全名。只有在服务实施类中使用属性时，才会指定此属性。 |
| **portName** | 指定发布该服务的端点的名称。此属性映射到 **wsdl:port** 元素的 **name** 属性，用于指定发布服务的端点详情。默认为将 **Port** 附加到服务的实现类的名称中。 |

[a] 当您从 SEI 生成 WSDL 时，将使用接口名称来代替实施类的名称。



**注意**

**不需要为任何 @WebService 注释的属性提供值。但是，我们建议您提供尽可能多的信息。**

### 注解 SEI

SEI 要求您添加 @WebService 注释。由于 SEI 是定义该服务的合同，因此您应该在 @WebService 注释的属性中指定服务可能的详细信息。

例 24.3 “带有 @WebService Annotation 的接口” 显示 例 24.1 “简单 SEI” 中定义的接口，它带有 @WebService 注释。

**例 24.3. 带有 @WebService Annotation 的接口**

```
package com.fusesource.demo;

import javax.jws.*;

@WebService(name="quoteUpdater",
            targetNamespace="http://demos.redhat.com",
          serviceName="updateQuoteService",
            wsdlLocation="http://demos.redhat.com/quoteExampleService?wsdl",
            portName="updateQuotePort")
public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

例 24.3 “带有 @WebService Annotation 的接口” 中的 @WebService 注释执行以下操作：

指定定义服务接口的 wsdl:portType 元素的 name 属性的值是 quoteUpdater。

指定服务的目标命名空间是 http://demos.redhat.com。

指定定义 published 服务的 wsdl:service 元素 的名称 值是 updateQuoteService。

指定该服务将在 http://demos.redhat.com/quoteExampleService?wsdl 发布其 WSDL 合同。

指定定义公开服务端点的 wsdl:port 元素的 name 属性的值是 updateQuotePort。

**注解服务实施**

除了使用 @WebService 注释标注 SEI 外，您还必须使用 @WebService 注释给服务实施类标注。将注解添加到服务实现类时，您只需要指定 endpointInterface 属性。如 例 24.4 “注解的服务实现类” 所示，属性必须设置为 SEI 的全名。

**例 24.4. 注解的服务实现类**

```
package org.eric.demo;
```

```
import javax.jws.*;

@WebService(endpointInterface="com.fusesource.demo.quoteReporter")
public class stockQuoteReporter implements quoteReporter
{
public Quote getQuote(String ticker)
  {
  ...
  }
}
```

### 24.3.3. 可选注解

#### 摘要

虽然 @WebService 注释足以用于启用 Java 接口或 Java 类，但它并不完全描述该服务如何公开为服务提供商。JAX-WS 编程模型使用多个可选注释来为您的服务（如其使用的绑定）添加到 Java 代码的详细信息。您可以将这些注解添加到服务的 SEI 中。

您在 SEI 中提供的更多详细信息，开发人员更易于实施可以使用它所定义功能的应用程序。它还使由工具生成的 WSDL 文档更具体。

#### 概述

使用注解定义绑定属性

如果您正在为您的服务使用 SOAP 绑定，您可以使用 JAX-WS 注释来指定多个绑定属性。这些属性与您可以在服务的 WSDL 合同中指定的属性直接对应。某些设置（如参数风格）可以限制您如何实施方法。这些设置还可以在注解方法参数时使用哪些注解。

#### @SOAPBinding 注释

@SOAPBinding 注释由 javax.jws.soap.SOAPBinding 接口定义。它提供了关于服务在部署时使用的 SOAP 绑定的详细信息。如果没有指定 @SOAPBinding 注释，则使用嵌套的 doc/literal SOAP 绑定发布服务。

您可以将 @SOAPBinding 注释放在 SEI 上，以及任何 SEI 的方法。在方法上使用时，将优先设置方法的 @SOAPBinding 注释。

表 24.2 “@SOAPBinding Properties” 显示 @SOAPBinding 注释的属性。

**表 24.2. @SOAPBinding Properties**

| 属性 | 值 | 描述 |
| --- | --- | --- |
| **style** | 样式.DOCUMENT （默认）<br><br>Style.RPC | 指定 SOAP 消息的样式。如果指定了 RPC 样式，则 SOAP 正文中的每个消息部分都是参数或返回值，并显示在 **soap:body** 元素中的 wrapper 元素中。wrapper 元素中的 message 部分对应于操作参数，且必须按照与操作中的参数相同的顺序。如果指定了 DOCUMENT 样式，则 SOAP 正文的内容必须是有效的 XML 文档，但它的格式不如严格约束。 |
| **使用** | use.LITERAL （默认）<br><br>use.ENCODED[a] | 指定 SOAP 消息的数据如何流化。 |
| **parameterStyle**[b] | ParameterStyle.BARE<br><br>ParameterStyle.WRAPPED （默认） | 指定方法参数（对应于 WSDL 合同中的消息部分）如何放入 SOAP 消息正文中。如果指定了 BARE，则每个参数将放置在消息正文中，作为消息根的子元素。如果指定了 WRAPPED，则所有输入参数都会嵌套在请求消息上的单个元素中，所有输出参数都会嵌套到响应消息中的单个元素中。 |

[a] use.ENCODED 目前不受支持。

[b] 如果将 **样式** 设置为 RPC，则必须使用 WRAPPED 参数样式。

记录**裸机风格的参数**

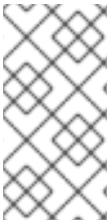
**文档 风格是 Java 代码和服务的 XML 表示法之间的最直接映射。使用此样式时，模式类型直接从操作参数列表中定义的输入和输出参数生成。**

**您要使用 @SOAPBinding 注释，并将其 style 属性设置为 Style.DOCUMENT，其 parameterStyle 属性设为 ParameterStyle.BARE 来使用裸机文档\literal 样式。**

**为确保在使用裸机参数时操作不会违反使用文档风格的限制，您的操作必须遵循以下条件：**

- **操作不能有多个输入或输入/输出参数。**

- 如果操作具有 void 以外的返回类型，则它不能具有任何输出或输入/输出参数。
- 如果操作具有返回类型 void，则它不能有多个输出或输入/输出参数。



注意

使用 @WebParam 注释或 @WebResult 注释放置在 SOAP 标头中的任何参数都不会根据允许的参数数量计算。

### 文档嵌套参数

文档嵌套式允许更多的 RPC，如 Java 代码之间的映射以及服务的 XML 表示。使用此样式时，方法参数列表中的参数由绑定嵌套到单个元素中。这样做的缺点是，它在 Java 实现之间引入了一个额外的间接层，以及消息在线上放置的方式。

要指定您要使用嵌套文档\literal 样式，请使用 @SOAPBinding 注释，其 样式 属性设为 Style.DOCUMENT，并将其 parameterStyle 属性设置为 ParameterStyle.WRAPPED。

您有一些控制如何使用 “@RequestWrapper 注释”一节 注解和 “@ResponseWrapper 注释”一节 注解生成打包程序。

### Example

例 24.5 “使用 SOAP Binding Annotation 指定 Document Bare SOAP Binding” 显示一个使用文档 bare SOAP 消息的 SEI。

**例 24.5. 使用 SOAP Binding Annotation 指定 Document Bare SOAP Binding**

```
package org.eric.demo;

import javax.jws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;

@WebService(name="quoteReporter")
@SOAPBinding(parameterStyle=ParameterStyle.BARE)
public interface quoteReporter
{
  ...
}
```

■

概述

使用注解定义操作属性

当运行时将 Java 方法定义映射到 XML 操作定义中时，它提供了如下详细信息：

- 在 XML 中交换的消息是什么
- 如果消息可以以一种方式优化
- 定义消息的命名空间

@WebMethod 注释

@WebMethod 注释由 javax.jws.WebMethod 接口定义。它放置在方法的 SEI 中。@WebMethod 注释提供了通常在 wsdl:operation 元素中表示的信息，描述该方法所关联的操作。

表 24.3 “@WebMethod Properties” 描述 @WebMethod 注释的属性。

表 24.3. @WebMethod Properties

| 属性 | 描述 |
| --- | --- |
| operationName | 指定关联的 wsdl:operation 元素的名称。默认值为方法的名称。 |
| action | 指定为方法生成的 soap:operation 元素的 soapAction 属性的值。默认值为空字符串。 |
| exclude | 指定是否应从服务接口中排除该方法。默认值为 false。 |

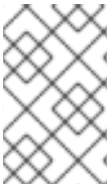
@RequestWrapper 注释

@RequestWrapper 注释由 javax.xml.ws.RequestWrapper 接口定义。它放置在方法的 SEI 中。@RequestWrapper 注释指定为请求消息启动消息的方法参数实施 wrapper bean 的 Java 类。它还指定元素名称和命名空间，供运行时在 marshalling 和 unmarshalling 请求消息时使用。

表 24.4 “@RequestWrapper Properties” 描述 @RequestWrapper 注释的属性。

表 24.4. @RequestWrapper Properties

| 属性 | 描述 |
| --- | --- |
| localName | 指定请求消息的 XML 表示中的 wrapper 元素的本地名称。默认值为方法的名称，也可以是 “@WebMethod 注释”一节 注解的 operationName 属性的值。 |
| targetNamespace | 指定定义 XML wrapper 元素的命名空间。默认值为 SEI 的目标命名空间。 |
| className | 指定实施 wrapper 元素的 Java 类的全名。 |



注意

仅需要 className 属性。

重要

如果方法也标上 @SOAPBinding 注释，其 parameterStyle 属性设为 ParameterStyle.BARE，则此注释将被忽略。

@ResponseWrapper 注释

@ResponseWrapper 注释由 javax.xml.ws.ResponseWrapper 接口定义。它放置在方法的 SEI 中。@ResponseWrapper 指定在消息交换中响应消息中实施方法参数的 wrapper bean 的 Java 类。它还指定元素名称和命名空间，供运行时在 marshaling 和 unmarshalling 处理响应消息时使用。

表 24.5 “@ResponseWrapper Properties” 描述 @ResponseWrapper 注释的属性。

表 24.5. @ResponseWrapper Properties

| 属性 | 描述 |
| --- | --- |
| localName | 指定响应消息的 XML 表示中的 wrapper 元素的本地名称。默认值是附加 Response 的方法的名称，或者附加 Response 的 “@WebMethod 注释”一节 注解的 operationName 属性的值。 |

| 属性 | 描述 |
| --- | --- |
| **targetNamespace** | 指定定义 XML wrapper 元素的命名空间。默认值为 SEI 的目标命名空间。 |
| **className** | 指定实施 wrapper 元素的 Java 类的全名。 |

**注意**

仅需要 **className** 属性。

**重要**

**如果方法也标上 @SOAPBinding 注释，其 parameterStyle 属性设为 ParameterStyle.BARE，则此注释将被忽略。**

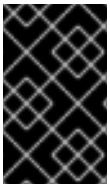
**@WebFault 注释**

**@WebFault 注释由 javax.xml.ws.WebFault 接口定义。它放置在您的 SEI 引发的异常上。@WebFault 注释用于将 Java 异常映射到 wsdl:fault 元素。此信息用于将异常分解为可由服务及其使用者处理的表示形式。**

**表 24.6 “@WebFault Properties” 描述 @WebFault 注释的属性。**

**表 24.6. @WebFault Properties**

| 属性 | 描述 |
| --- | --- |
| **name** | 指定 fault 元素的本地名称。 |
| **targetNamespace** | 指定定义 fault 元素的命名空间。默认值为 SEI 的目标命名空间。 |
| **faultName** | 指定实现例外的 Java 类的全名。 |



**重要**

**name 属性是必需的。**

### @Oneway 注释

@Oneway 注释由 javax.jws.Oneway 接口定义。它放置在不需要来自服务的 SEI 的方法上。@Oneway 注释告知运行时间，它可以通过不等待响应来优化方法的执行，并且不会保留任何资源来处理响应。

此注解只能用于满足以下条件的方法：

- 它们返回 void
- 它们没有实现 Holder 接口的参数
- 它们不会抛出任何可传递给消费者的异常

### Example

例 24.6 “带有注解方法的 SEI” 显示带有其方法标注的 SEI。

**例 24.6. 带有注解方法的 SEI**

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;

@WebService(name="quoteReporter")
public interface quoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                   className="org.eric.demo.Quote")
  public Quote getQuote(String ticker);
}
```

### 概述

### 使用注解定义参数属性

SEI 中的 method 参数对应于 wsdl:message 元素及其 wsdl:part 元素。JAX-WS 提供了注释，允

许您描述为方法参数生成的 wsdl:part 元素。

### @WebParam 注释

@WebParam 注释由 javax.jws.WebParam 接口定义。它放置在 SEI 中定义的方法的参数上。@WebParam 注释允许您指定参数的方向，如果参数将放在 SOAP 标头中，以及生成的 wsdl:part 的其他属性。

表 24.7 “@WebParam Properties” 描述 @WebParam 注释的属性。

**表 24.7. @WebParam Properties**

| 属性 | 值 | 描述 |
| --- | --- | --- |
| **name** | | 指定在生成的 WSDL 文档中显示的参数名称。对于 RPC 绑定，这是代表参数的 **wsdl:part** 的名称。对于文档绑定，这是代表 参数的 XML 元素的本地名称。根据 JAX-WS 规范，默认值为 **arg*N***，其中 *N* 被替换为基于零的参数索引（如 arg0、arg1 等）。 |
| **targetNamespace** | | 指定参数的命名空间。它仅用于参数映射到 XML 元素的文档绑定。默认为使用服务的命名空间。 |
| **模式** | mode.IN （默认）[a]<br>mode.OUT<br>mode.INOUT | 指定参数的方向。 |
| **header** | false （默认）<br>true | 指定参数是否作为 SOAP 标头的一部分传递。 |
| **partName** | | 指定参数的 **wsdl:part** 元素的 **name** 属性的值。此属性用于文档风格的 SOAP 绑定。 |

[a] 实现 Holder 接口的任何参数都会被映射到 Mode.INOUT。

### @WebResult 注释

@WebResult 注释由 javax.jws.WebResult 接口定义。它放置在 SEI 中定义的方法

上。@WebResult 注释允许您指定为方法的返回值生成的 wsdl:part 的属性。

表 24.8 “@WebResult Properties” 描述 @WebResult 注释的属性。

表 24.8. @WebResult Properties

| 属性 | 描述 |
| --- | --- |
| **name** | 指定在生成的 WSDL 文档中出现的返回值的名称。对于 RPC 绑定，这是代表返回值的 **wsdl:part** 的名称。对于文档绑定，这是代表返回值的 XML 元素的本地名称。默认值为 return。 |
| **targetNamespace** | 指定返回值的命名空间。它仅用于返回值映射到 XML 元素的文档绑定。默认为使用服务的命名空间。 |
| **header** | 指定返回值是否作为 SOAP 标头的一部分传递。 |
| **partName** | 指定返回值的 **wsdl:part** 元素的 **name** 属性的值。此属性用于文档风格的 SOAP 绑定。 |

**Example**

例 24.7 “完全解析的 SEI” 显示被完全注解的 SEI。

**例 24.7. 完全解析的 SEI**

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;
import javax.jws.WebParam.*;

@WebService(targetNamespace="http://demo.redhat.com",
            name="quoteReporter")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface quoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                   className="org.eric.demo.Quote")
  @WebResult(targetNamespace="http://demo.redhat.com/types",
             name="updatedQuote")
  public Quote getQuote(
```

```
                @WebParam(targetNamespace="http://demo.redhat.com/types",
                        name="stockTicker",
                        mode=Mode.IN)
                String ticker
    );
}
```

### 24.3.4. Apache CXF 注解

#### 24.3.4.1. WSDL 文档

**@WSDL 文档注释**

**@WSDL 文档 注释由 org.apache.cxf.annotations.WSDL 文档接口定义。它可以放在 SEI 或 SEI 方法上。**

**此注释允许您添加文档，然后在 SEI 转换为 WSDL 后出现在 wsdl:documentation 元素中。默认情况下，文档元素出现在端口类型内，但您可以指定 placement 属性，使文档出现在 WSDL 文件中的其他位置上。第 24.3.4.2 节 “@WSDL 文档属性” 显示 @WSDL 文档注释支持的属性。**

#### 24.3.4.2. @WSDL 文档属性

| 属性 | 描述 |
| --- | --- |
| **value** | **（必需）** 包含文档文本的字符串。 |
| **placement** | **（可选）** 指定此文档在 WSDL 文件中显示的位置。有关可能的放置值列表，请参阅 “放置在 WSDL 合同中”一节。 |
| **faultClass** | **（可选）** 如果将放置设置为 **FAULT_MESSAGE**、**PORT_TYPE_OPERATION_FAULT** 或 **BINDING_OPERATION_FAULT**，您还必须将此属性设置为代表故障的 Java 类。 |

**@WSDLDocumentationCollection 注释**

**@WSDLDocumentationCollection 注释由 org.apache.cxf.annotations.WSDLDocumentationCollection 接口定义。它可以放在 SEI 或 SEI 方法上。**

此注解用于在单个放置位置或不同放置位置插入多个文档元素。

放置在 WSDL 合同中

要指定在 WSDL 合同中应显示文档的位置，您可以指定 放置 属性，该属性类型为 WSDLDocumentation.Placement。放置可以具有以下值之一：

- WSDLDocumentation.Placement.BINDING
- WSDLDocumentation.Placement.BINDING_OPERATION
- WSDLDocumentation.Placement.BINDING_OPERATION_FAULT
- WSDLDocumentation.Placement.BINDING_OPERATION_INPUT
- WSDLDocumentation.Placement.BINDING_OPERATION_OUTPUT
- WSDLDocumentation.Placement.DEFAULT
- WSDLDocumentation.Placement.FAULT_MESSAGE
- WSDLDocumentation.Placement.INPUT_MESSAGE
- WSDLDocumentation.Placement.OUTPUT_MESSAGE
- WSDLDocumentation.Placement.PORT_TYPE
- WSDLDocumentation.Placement.PORT_TYPE_OPERATION

- **WSDLDocumentation.Placement.PORT_TYPE_OPERATION_FAULT**
- **WSDLDocumentation.Placement.PORT_TYPE_OPERATION_INPUT**
- **WSDLDocumentation.Placement.PORT_TYPE_OPERATION_OUTPUT**
- **WSDLDocumentation.Placement.SERVICE**
- **WSDLDocumentation.Placement.SERVICE_PORT**
- **WSDLDocumentation.Placement.TOP**

**@WSDL 文档示例**

**第 24.3.4.3 节 “使用 @WSDL 文档” 演示了如何向 SEI 和其中一个方法添加 @WSDL 文档 注释。**

#### 24.3.4.3. 使用 @WSDL 文档

```
@WebService
@WSDLDocumentation("A very simple example of an SEI")
public interface HelloWorld {
    @WSDLDocumentation("A traditional form of greeting")
    String sayHi(@WebParam(name = "text") String text);
}
```

**当 WSDL 在 第 24.3.4.4 节 “使用文档生成的 WSDL” 中显示时，从 第 24.3.4.3 节 “使用 @WSDL 文档” 中的 SEI 生成，文档 元素的默认位置分别是 PORT_TYPE 和 PORT_TYPE_OPERATION。**

#### 24.3.4.4. 使用文档生成的 WSDL

```
<wsdl:definitions ... >
  ...
  <wsdl:portType name="HelloWorld">
    <wsdl:documentation>A very simple example of an SEI</wsdl:documentation>
    <wsdl:operation name="sayHi">
      <wsdl:documentation>A traditional form of greeting</wsdl:documentation>
      <wsdl:input name="sayHi" message="tns:sayHi">
    </wsdl:input>
      <wsdl:output name="sayHiResponse" message="tns:sayHiResponse">
```

```
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

**@WSDLDocumentationCollection 示例**

第 24.3.4.5 节 “使用 @WSDLDocumentationCollection” 演示了如何将 @WSDLDocumentationCollection 注释添加到 SEI。

#### 24.3.4.5. 使用 @WSDLDocumentationCollection

```
@WebService
@WSDLDocumentationCollection(
    {
        @WSDLDocumentation("A very simple example of an SEI"),
        @WSDLDocumentation(value = "My top level documentation",
                           placement = WSDLDocumentation.Placement.TOP),
        @WSDLDocumentation(value = "Binding documentation",
                           placement = WSDLDocumentation.Placement.BINDING)
    }
)
public interface HelloWorld {
    @WSDLDocumentation("A traditional form of Geeky greeting")
    String sayHi(@WebParam(name = "text") String text);
}
```

#### 24.3.4.6. 消息架构验证

**@SchemaValidation 注释**

@SchemaValidation 注释由 org.apache.cxf.annotations.SchemaValidation 接口定义。它可以放置在 SEI 和单独的 SEI 方法上。

此注解打开发送到此端点的 XML 消息的架构验证。当您怀疑传入的 XML 消息格式存在问题时，这可用于测试目的。默认情况下禁用验证，因为它对性能有严重影响。

**模式验证类型**

模式验证行为由 type 参数控制，其值是 org.apache.cxf.annotations.SchemaValidation.SchemaValidationType 类型的枚举。第 24.3.4.7 节

“模式验证类型值” 显示可用验证类型的列表。

#### 24.3.4.7. 模式验证类型值

| 类型 | 描述 |
| --- | --- |
| **IN** | 将架构验证应用到客户端和服务器上的传入消息。 |
| **OUT** | 将架构验证应用到客户端和服务器上的传出消息。 |
| **两者都** | 对客户端和服务器上的传入和传出消息应用架构验证。 |
| **NONE** | 禁用所有模式验证。 |
| **REQUEST （请求）** | 将架构验证应用到 Request 消息-即，导致验证应用到传出客户端消息和传入的服务器消息。 |
| **响应** | 将架构验证应用到响应消息（即，验证应用到传入客户端消息）和传出服务器消息。 |

**Example**

以下示例演示了如何根据 MyService SEI 为端点启用架构消息验证。请注意，注解如何作为一个整体应用到 SEI，以及 SEI 中的单个方法。

```
@WebService
@SchemaValidation(type = SchemaValidationType.BOTH)
public interface MyService {
    Foo validateBoth(Bar data);

    @SchemaValidation(type = SchemaValidationType.NONE)
    Foo validateNone(Bar data);

    @SchemaValidation(type = SchemaValidationType.IN)
    Foo validateIn(Bar data);

    @SchemaValidation(type = SchemaValidationType.OUT)
    Foo validateOut(Bar data);

    @SchemaValidation(type = SchemaValidationType.REQUEST)
    Foo validateRequest(Bar data);

    @SchemaValidation(type = SchemaValidationType.RESPONSE)
    Foo validateResponse(Bar data);
}
```

#### 24.3.4.8. 指定数据绑定

### @dataBinding 注释

@DataBinding 注释由 org.apache.cxf.annotations.DataBinding 接口定义。它放置在 SEI 上。

此注释用于将数据绑定与 SEI 关联，以取代默认的 JAXB 服务绑定。@DataBinding 注释的值必须是提供数据绑定 *ClassName*.class 的类。

### 支持的数据绑定

Apache CXF 目前支持以下数据绑定：

- org.apache.cxf.jaxb.JAXBDataBinding

  （默认） 标准 JAXB 数据集。

- org.apache.cxf.sdo.SDODataBinding

  Service Data Objects (SDO)数据绑定基于 Apache Tuscany SDO 实现。如果要在 Maven 构建上下文中使用此数据绑定，则需要对 cxf-rt-databinding-sdo 工件添加依赖项。

- org.apache.cxf.aegis.databinding.AegisDatabinding

  如果要在 Maven 构建上下文中使用此数据绑定，则需要对 cxf-rt-databinding-aegis 工件添加依赖项。

- org.apache.cxf.xmlbeans.XmlBeansDataBinding

  如果要在 Maven 构建上下文中使用此数据绑定，则需要对 cxf-rt-databinding-xmlbeans 工件添加依赖项。

- org.apache.cxf.databinding.source.SourceDataBinding

此数据绑定属于 Apache CXF 内核。

- org.apache.cxf.databinding.stax.StaxDataBinding

  此数据绑定属于 Apache CXF 内核。

**Example**

第 24.3.4.9 节 “设置数据绑定” 演示了如何将 SDO 绑定与 HelloWorld SEI 关联

#### 24.3.4.9. 设置数据绑定

```
@WebService
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

#### 24.3.4.10. 压缩消息

**@GZIP 注释**

@GZIP 注释由 org.apache.cxf.annotations.GZIP 接口定义。它放置在 SEI 上。

启用 GZIP 压缩消息。GZIP 是一个协商的增强。也就是说，不会对客户端的初始请求进行 gzip 压缩，而是添加 Accept 标头，如果服务器支持 GZIP 压缩，则响应将被 gzip 压缩，并且后续请求也会被压缩。

第 24.3.4.11 节 “@GZIP Properties” 显示 @GZIP 注释支持的可选属性。

#### 24.3.4.11. @GZIP Properties

| 属性 | 描述 |
| --- | --- |
| threshold | 小于此属性指定大小的消息不会被 gzipped。默认为 -1（无限制）。 |

**@FastInfoset**

@FastInfoset 注释由 org.apache.cxf.annotations.FastInfoset 接口定义。它放置在 SEI 上。

为消息启用 FastInfoset 格式。fastinfoset 是 XML 的二进制编码格式，旨在优化消息大小和 XML 消息处理性能。详情请查看 Fast Infoset 中的以下 Sun 文章。

fastinfoset 是一个协商的增强。也就是说，来自客户端的初始请求不会采用 FastInfoset 格式，而是添加 Accept 标头，如果服务器支持 FastInfoset，则响应将位于 FastInfoset 中，并且任何后续请求也会被添加。

第 24.3.4.12 节 “@FastInfoset Properties” 显示 @FastInfoset 注释支持的可选属性。

#### 24.3.4.12. @FastInfoset Properties

| 属性 | 描述 |
| --- | --- |
| **force** | 强制使用 FastInfoset 格式的布尔值属性，而不是 negotiating。为 **true** 时，强制使用 FastInfoset 格式；否则，协商。默认为 **false**。 |

**@GZIP 示例**

第 24.3.4.13 节 “启用 GZIP” 演示了如何为 HelloWorld SEI 启用 GZIP 压缩。

#### 24.3.4.13. 启用 GZIP

```
@WebService
@GZIP
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

**exampe @FastInfoset**

第 24.3.4.14 节 “启用 FastInfoset” 演示了如何为 HelloWorld SEI 启用 FastInfoset 格式。

#### 24.3.4.14. 启用 FastInfoset

```
@WebService
@FastInfoset
public interface HelloWorld {
```

```
    String sayHi(@WebParam(name = "text") String text);
}
```

#### 24.3.4.15. 在端点上启用日志记录

**@logging 注解**

**@Logging 注释由 org.apache.cxf.annotations.Logging 接口定义。它放置在 SEI 上。**

**此注解为与 SEI 关联的所有端点启用日志记录。第 24.3.4.16 节 “@logging Properties” 显示您可以在此注解中设置的可选属性。**

#### 24.3.4.16. @logging Properties

| 属性 | 描述 |
| --- | --- |
| **limit** | 指定大小限制，超过日志中消息被截断。默认为 64K。 |
| **inLocation** | 指定记录传入的消息的位置。可以是 < **stderr> ,** &lt **;stdout>**, **<logger** > , 或一个文件名。默认为 &lt **;logger>**。 |
| **outLocation** | 指定记录传出消息的位置。可以是 < **stderr> ,** &lt **;stdout>**, **<logger** > , 或一个文件名。默认为 &lt **;logger>**。 |

**Example**

**第 24.3.4.17 节 “使用注解进行日志记录配置” 演示了如何为 HelloWorld SEI 启用日志记录，其中传入的消息发送到 < stdout >，传出消息将发送到 < logger>。**

#### 24.3.4.17. 使用注解进行日志记录配置

```
@WebService
@Logging(limit=16000, inLocation="<stdout>")
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

#### 24.3.4.18. 在端点中添加属性和策略

**摘要**

**属性和策略都可用于将配置数据与端点关联。它们之间的基本区别在于属性是 Apache CXF 特定的配置机制，而 策略 则是标准的 WSDL 配置机制。策略通常源自 WS 规范和标准，通常通过定义 WSDL 合同中显示的 wsdl:policy 元素来设置。相反，属性是特定于 Apache CXF 的，它们通常在 Apache CXF Spring 配置文件中定义 jaxws:properties 元素来设置。**

**但是，也可以使用注释在 Java 中定义属性设置和 WSDL 策略设置，如下所述。**

#### 24.3.4.19. 添加属性

**@EndpointProperty 注释**

**@EndpointProperty 注释由 org.apache.cxf.annotations.EndpointProperty 接口定义。它放置在 SEI 上。**

**此注解在端点中添加特定于 Apache CXF 的配置设置。端点属性也可以在 Spring 配置文件中指定。例如，要在端点上配置 WS-Security，您可以使用 Spring 配置文件中的 jaxws:properties 元素添加端点属性，如下所示：**

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

  <jaxws:endpoint
    id="MyService"
    address="https://localhost:9001/MyService"
    serviceName="interop:MyService"
    endpointName="interop:MyServiceEndpoint"
    implementor="com.foo.MyService">

    <jaxws:properties>
      <entry key="ws-security.callback-handler" value="interop.client.UTPasswordCallback"/>
      <entry key="ws-security.signature.properties" value="etc/keystore.properties"/>
      <entry key="ws-security.encryption.properties" value="etc/truststore.properties"/>
      <entry key="ws-security.encryption.username" value="useReqSigCert"/>
    </jaxws:properties>

  </jaxws:endpoint>
</beans>
```

**或者，您可以通过在 SEI 中添加 @EndpointProperty 注解来指定 Java 前面的配置设置，如 第 24.3.4.20 节 “使用 @EndpointProperty Annotations 配置 WS-Security” 所示。**

#### 24.3.4.20. 使用 @EndpointProperty Annotations 配置 WS-Security

```
@WebService
@EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback")
@EndpointProperty(name="ws-security.signature.properties" value="etc/keystore.properties")
@EndpointProperty(name="ws-security.encryption.properties" value="etc/truststore.properties")
@EndpointProperty(name="ws-security.encryption.username" value="useReqSigCert")
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

**@EndpointProperties 注释**

**@EndpointProperties 注释由 org.apache.cxf.annotations.EndpointProperties 接口定义。它放置在 SEI 上。**

**此注释提供了将多个 @EndpointProperty 注解分组到列表中的方法。使用 @EndpointProperties 时，可以重新写入 第 24.3.4.20 节 “使用 @EndpointProperty Annotations 配置 WS-Security”，如 第 24.3.4.21 节 “使用 @EndpointProperties Annotation 配置 WS-Security” 所示。**

#### 24.3.4.21. 使用 @EndpointProperties Annotation 配置 WS-Security

```
@WebService
@EndpointProperties(
  {
  @EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback"),
  @EndpointProperty(name="ws-security.signature.properties" value="etc/keystore.properties"),
  @EndpointProperty(name="ws-security.encryption.properties" value="etc/truststore.properties"),
  @EndpointProperty(name="ws-security.encryption.username" value="useReqSigCert")
})
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

#### 24.3.4.22. 添加策略

**@policy 注释**

**@Policy 注释由 org.apache.cxf.annotations.Policy 接口定义。它可以放在 SEI 或 SEI 方法上。**

**此注释用于将 WSDL 策略与 SEI 或 SEI 方法关联。该策略通过提供 URI 来指定，该 URI 引用包含标准 wsdl:policy 元素的 XML 文件。如果要从 SEI 生成 WSDL 合同（例如，使用 java2ws 命令行工具），您可以指定您是否要在 WSDL 中包含此策略。**

第 24.3.4.23 节 “@policy Properties” 显示 @Policy 注释支持的属性。

#### 24.3.4.23. @policy Properties

| 属性 | 描述 |
| --- | --- |
| **uri** | **（必需）** 包含策略定义的文件的位置。 |
| **includeInWSDL** | **（可选）** 在生成 WSDL 时，是否在生成的合同中包含策略。默认为 **true**。 |
| **placement** | **（可选）** 指定此文档在 WSDL 文件中显示的位置。有关可能的放置值列表，请参阅 “放置在 WSDL 合同中”一节。 |
| **faultClass** | **（可选）** 如果将放置设置为 **BINDING_OPERATION_FAULT** 或 **PORT_TYPE_OPERATION_FAULT**，则必须设置此属性来指定此策略应用到哪些错误。值是代表 fault 的 Java 类。 |

**@policies 注释**

**@Policies 注释由 org.apache.cxf.annotations.Policies 接口定义。它可以放置在 SEI 或第 SEI 方法上。**

**此注释提供了将多个 @Policy 注解分组到列表中的方法。**

**放置在 WSDL 合同中**

**要指定在 WSDL 合同中应显示策略的位置，您可以指定 放置 属性，即 Policy.Placement 类型。放置可以具有以下值之一：**

```
Policy.Placement.BINDING
Policy.Placement.BINDING_OPERATION
Policy.Placement.BINDING_OPERATION_FAULT
Policy.Placement.BINDING_OPERATION_INPUT
Policy.Placement.BINDING_OPERATION_OUTPUT
Policy.Placement.DEFAULT
Policy.Placement.PORT_TYPE
Policy.Placement.PORT_TYPE_OPERATION
Policy.Placement.PORT_TYPE_OPERATION_FAULT
Policy.Placement.PORT_TYPE_OPERATION_INPUT
```

```
Policy.Placement.PORT_TYPE_OPERATION_OUTPUT
Policy.Placement.SERVICE
Policy.Placement.SERVICE_PORT
```

**@Policy 示例**

以下示例演示了如何将 WSDL 策略与 HelloWorld SEI 关联，以及如何将策略与 sayHi 方法关联。策略本身存储在文件系统的 XML 文件中，在 annotationpolicies 目录下。

```
@WebService
@Policy(uri = "annotationpolicies/TestImplPolicy.xml",
        placement = Policy.Placement.SERVICE_PORT),
@Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
        placement = Policy.Placement.PORT_TYPE)
public interface HelloWorld {
    @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
            placement = Policy.Placement.PORT_TYPE_OPERATION),
    String sayHi(@WebParam(name = "text") String text);
}
```

**@Policies 示例**

您可以使用 @Policies 注释将多个 @Policy 注释分组到列表中，如下例所示：

```
@WebService
@Policies({
    @Policy(uri = "annotationpolicies/TestImplPolicy.xml",
            placement = Policy.Placement.SERVICE_PORT),
    @Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
            placement = Policy.Placement.PORT_TYPE)
})
public interface HelloWorld {
    @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
            placement = Policy.Placement.PORT_TYPE_OPERATION),
    String sayHi(@WebParam(name = "text") String text);
}
```

## 24.4. 生成 WSDL

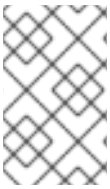
**使用 Maven**

注解代码后，您可以使用 java2ws Maven 插件的 -wsdl 选项为您的服务生成 WSDL 合同。有关 java2ws Maven 插件选项的详细列表，请参阅 第 44.3 节 “java2ws”。

例 24.8 “从 Java 生成 WSDL” 显示如何设置 java2ws Maven 插件来生成 WSDL。

**例 24.8. 从 Java 生成 WSDL**

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <genWsdl>true</genWsdl>
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



**注意**

**将 className 的值替换为合格的 className。**

**Example**

例 24.9 “从 SEI生成 WSDL” 显示为 例 24.7 “完全解析的 SEI” 中显示的 SEI 生成的 WSDL 合同。

**例 24.9. 从 SEI生成 WSDL**

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://demo.eric.org/"
        xmlns:tns="http://demo.eric.org/"
    xmlns:ns1=""
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns2="http://demo.eric.org/types"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema>
      <xs:complexType name="quote">
        <xs:sequence>
          <xs:element name="ID" type="xs:string" minOccurs="0"/>
          <xs:element name="time" type="xs:string" minOccurs="0"/>
```

```
          <xs:element name="val" type="xs:float"/>
        </xs:sequence>
      </xs:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="getStockQuote">
    <wsdl:part name="stockTicker" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getStockQuoteResponse">
    <wsdl:part name="updatedQuote" type="tns:quote">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="quoteReporter">
    <wsdl:operation name="getStockQuote">
      <wsdl:input name="getQuote" message="tns:getStockQuote">
    </wsdl:input>
      <wsdl:output name="getQuoteResponse" message="tns:getStockQuoteResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="quoteReporterBinding" type="tns:quoteReporter">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="getStockQuote">
      <soap:operation style="rpc" />
      <wsdl:input name="getQuote">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="getQuoteResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="quoteReporterService">
    <wsdl:port name="quoteReporterPort" binding="tns:quoteReporterBinding">
      <soap:address location="http://localhost:9000/quoteReporterService" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

**[1]** **Board is an assumed class whose implementation is left to the reader.**

# 第 25 章 开发没有 WSDL 合同的消费者

## 摘要

您不需要 WSDL 合同来开发服务消费者。您可以从注解的 SEI 创建服务消费者。除了 SEI 外，您需要知道发布该服务的端点的地址、定义公开该服务的端点的 QName，以及定义消费者发出请求的端点的 QName。此信息可以在 SEI 的注解中指定，也可以单独提供。

## 25.1. JAVA-FIRST CONSUMER DEVELOPMENT

要创建没有 WSDL 合同的消费者，您必须执行以下操作：

1. 为服务创建一个 Service 对象，使用者将在其上调用操作。
2. 向 Service 对象 添加一个端口。
3. 使用 Service 对象的 getPort （） 方法获取服务的代理。
4. 实施消费者的业务逻辑。

## 25.2. 创建服务对象

### 概述

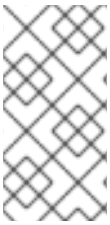
javax.xml.ws.Service 类代表 wsdl:service 元素，其中包含公开服务的所有端点的定义。因此，它提供允许您获取由 wsdl:port 元素定义的端点的方法，它们是在服务上执行远程调用的代理。

**注意**

Service 类提供抽象，允许客户端代码与 Java 类型一起使用，而不是使用 XML 文档。

create （）方法

Service 类有两个静态 create （） 方法，可用于创建新 Service 对象。如 例 25.1 “service create （） 方法” 所示，两个 create （） 方法都使用 wsdl:service 元素的 QName，Service 对象将代表，另一个采用指定 WSDL 合同位置的 URI。



注意

所有服务都会发布其 WSDL 合同。对于 SOAP/HTTP 服务，URI 通常是附加 ?wsdl 的服务的 URI。

例 25.1. service create （） 方法

公共静态服务创建URLwsdlLocationQNameserviceNameWebServiceExceptionpublic 静态ServicecreateQNameserviceNameWebServiceException

serviceName 参数的值是一个 QName。其命名空间部分的值是服务的目标命名空间。服务的目标命名空间在 @WebService 注解的 targetNamespace 属性中指定。QName 的本地部分的值是 wsdl:service 元素的 name 属性的值。您可以使用以下方法之一确定这个值： .它在 @WebService 注释的 serviceName 属性中指定。

1. 您可以将 Service 附加到 @WebService 注释的 name 属性的值。

2. 您可以将 Service 附加到 SEI 的名称。

**重要**

**在 OSGi 环境中以编程方式创建的 CXF 消费者需要特殊处理，以避免发生发生 类 NotFoundException的可能性。对于包含以编程方式创建的 CXF 用户的每个捆绑包，您需要创建一个单例 CXF 默认总线，并确保所有捆绑包的 CXF 用户都使用它。如果没有这种保护，可以分配在另一个捆绑包中创建的 CXF 默认总线，这可能会导致继承捆绑包失败。**

**例如，假设捆绑包 A 没有明确设置 CXF 默认总线，并在捆绑包 B 中分配 CXF 默认总线。如果捆绑包 A 中的 CXF 总线需要配置额外的功能（如 SSL 或 WS-Security），或者需要从捆绑包 A 中加载某些类或资源，它将会失败。这是因为 CXF 总线实例将线程上下文类加载程序(TCCL)设置为创建它的捆绑包的捆绑类加载程序（本例中为 bundle B）。此外，某些框架，如 wss4j ( CXF 中的 WS-Security)使用 TCCL 加载资源，如 calback 处理程序类或其他属性文件（捆绑包内部）。因为捆绑包 A 被分配 B 的默认 CXF 总线，并且是 TCCL，wss4j 层无法从捆绑包 A 加载所需资源，这会导致 ClassNotFoundException 错误。**

**要创建单例 CXF 默认总线，请将此代码插入到创建服务对象的 主 方法的开头，如 “Example”一节 所示：**

```
BusFactory.setThreadDefaultBus(BusFactory.newInstance().createBus());
```

**Example**

**例 25.2 “创建服务 对象” 显示为 例 24.7 “完全解析的 SEI” 中显示的 SEI 创建 Service 对象的代码。**

**例 25.2. 创建服务 对象**

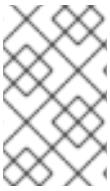
```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    BusFactory.setThreadDefaultBus(BusFactory.newInstance().createBus());
    QName serviceName = new QName("http://demo.redhat.com", "stockQuoteReporter");
    Service s = Service.create(serviceName);
   ...
  }
}
```

例 25.2 “创建服务 对象” 中的代码执行以下操作：

创建一个单例 CXF 默认总线，供服务的所有 CXF 用户使用。

使用 targetNamespace 属性和 @WebService 注释的 name 属性为服务构建 QName。

调用单个参数 create （） 方法来创建新 Service 对象。



注意

使用单个参数 create （） 可为您取消对访问 WSDL 合同的任何依赖项。

## 25.3. 在服务中添加端口

### 概述

服务的端点信息在 wsdl:port 元素中定义，并且 Service 对象为 WSDL 合同中定义的每个端点创建一个代理实例（如果指定了）。如果您在创建 Service 对象时没有指定 WSDL 合同，Service 对象没有有关实现您的服务的端点的信息，因此无法创建任何代理实例。在这种情况下，您必须为 Service 对象提供使用 addPort （） 方法代表 wsdl:port 元素所需的信息。

### addPort （）方法

Service 类定义了一个 addPort （） 方法，如 例 25.3 “addPort （） 方法” 所示，该方法在没有适用于消费者实施的 WSDL 合同时使用。addPort （） 方法允许您为 Service 对象提供信息，该对象通常存储在 wsdl:port 元素中，这是为服务实施创建代理所必需的。

例 25.3. addPort （） 方法

addPortQNameportNameStringbindingIdStringendpointAddressWebServiceException

portName 的值是一个 QName。其命名空间部分的值是服务的目标命名空间。服务的目标命名空间

在 @WebService 注解的 targetNamespace 属性中指定。QName 的本地部分的值是 wsdl:port 元素的 name 属性的值。您可以使用以下方法之一确定这个值：

1. 在 @WebService 注释的 portName 属性中指定它。
2. 将 Port 附加到 @WebService 注释的 name 属性的值中。
3. 将 Port 附加到 SEI 的名称。

bindingId 参数的值是一个字符串，用于标识端点使用的绑定类型。对于 SOAP 绑定，您可以使用标准 SOAP 命名空间： http://schemas.xmlsoap.org/soap/。如果端点没有使用 SOAP 绑定，则 bindingId 参数的值由绑定开发人员决定。 endpointAddress 参数的值是发布端点的地址。对于 SOAP/HTTP 端点，地址是 HTTP 地址。HTTP 以外的传输使用不同的地址方案。

**Example**

例 25.4 “在服务对象中 添加端口” 显示向 例 25.2 “创建服务 对象” 中创建的 Service 对象添加端口的代码。

**例 25.4. 在服务对象中 添加端口**

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    ...
    QName portName = new QName("http://demo.redhat.com", "stockQuoteReporterPort");
    s.addPort(portName,
            "http://schemas.xmlsoap.org/soap/",
            "http://localhost:9000/StockQuote");
    ...
  }
}
```

例 25.4 “在服务对象中 添加端口” 中的代码执行以下操作：

为 portName 参数创建 QName。

调用 addPort （） 方法。

指定端点使用 SOAP 绑定。

指定发布端点的地址。

## 25.4. 为端点获取代理

### 概述

服务代理是一个提供远程服务公开的所有方法的对象，并处理进行远程调用所需的所有详细信息。Service 对象为它通过 getPort （） 方法了解的所有端点提供服务代理。获得服务代理后，您可以调用其方法。代理使用服务合同中指定的连接详情，将调用转发到远程服务端点。

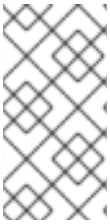
### getPort （）方法

getPort （） 方法（如 例 25.5 “getPort （） 方法” 所示）返回指定端点的服务代理。返回的代理与 SEI 相同的类。

**例 25.5. getPort （） 方法**

```
公共 &lt;T&gt;
TgetPortQNameportNameClass<T&gt;serviceEndpointInterfaceWebServiceException
```

portName 参数的值是一个 QName，用于标识 wsdl:port 元素来定义创建代理的端点。serviceEndpointInterface 参数的值是 SEI 的完全限定名称。



**注意**

当您没有 WSDL 合同时，portName 参数的值通常与调用 addPort （） 时用于 portName 参数的值相同。

**Example**

例 25.6 “获取服务代理” 显示获取添加到 例 25.4 “在服务对象中 添加端口” 中的端点的服务代理的代码。

**例 25.6. 获取服务代理**

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    ...
    quoteReporter proxy = s.getPort(portName, quoteReporter.class);
    ...
  }
}
```

## 25.5. 实施 CONSUMER 的 BUSINESS LOGIC

**概述**

为远程端点实例化服务代理后，您可以调用其方法，就如同它是本地对象一样。调用块，直到远程方法完成。

**注意**

如果方法标上 @OneWay 注释，则调用会立即返回。

**Example**

例 25.7 “没有 WSDL 合同的消费者” 显示 例 24.7 “完全解析的 SEI” 中定义的服务的使用者。

**例 25.7. 没有 WSDL 合同的消费者**

```
package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    QName serviceName = new QName("http://demo.eric.org", "stockQuoteReporter");
  Service s = Service.create(serviceName);

    QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
  s.addPort(portName, "http://schemas.xmlsoap.org/soap/",
"http://localhost:9000/EricStockQuote");

  quoteReporter proxy = s.getPort(portName, quoteReporter.class);

  Quote quote = proxy.getQuote("ALPHA");
    System.out.println("Stock "+quote.getID()+" is worth "+quote.getVal()+" as of
"+quote.getTime());
  }
}
```

例 25.7 “没有 WSDL 合同的消费者” 中的代码执行以下操作：

创建 Service 对象。

在 Service 对象中添加端点定义。

从 Service 对象获取服务代理。

在服务代理上调用操作。

# 第 26 章 开始点 WSDL 合同

## 26.1. WSDL 合同示例

例 26.1 “helloworld WSDL Contract” 显示 HelloWorld WSDL 合同。这个合同在 wsdl:portType 元素中定义了一个接口 Greeter。该合同还定义了将在 wsdl:port 元素中实施该服务的端点。

**例 26.1. helloworld WSDL Contract**

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"
                  xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
          xmlns="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified">
      <element name="sayHiResponse">
        <complexType>
          <sequence>
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMe">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeResponse">
        <complexType>
          <sequence>
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeOneWay">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="pingMe">
        <complexType/>
```

```
    </element>
    <element name="pingMeResponse">
      <complexType/>
    </element>
    <element name="faultDetail">
      <complexType>
        <sequence>
          <element name="minor" type="short"/>
          <element name="major" type="short"/>
        </sequence>
      </complexType>
    </element>
  </schema>
</wsdl:types>

<wsdl:message name="sayHiRequest">
  <wsdl:part element="x1:sayHi" name="in"/>
</wsdl:message>
<wsdl:message name="sayHiResponse">
  <wsdl:part element="x1:sayHiResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeRequest">
  <wsdl:part element="x1:greetMe" name="in"/>
</wsdl:message>
<wsdl:message name="greetMeResponse">
  <wsdl:part element="x1:greetMeResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeOneWayRequest">
  <wsdl:part element="x1:greetMeOneWay" name="in"/>
</wsdl:message>
<wsdl:message name="pingMeRequest">
  <wsdl:part name="in" element="x1:pingMe"/>
</wsdl:message>
<wsdl:message name="pingMeResponse">
  <wsdl:part name="out" element="x1:pingMeResponse"/>
</wsdl:message>
<wsdl:message name="pingMeFault">
  <wsdl:part name="faultDetail" element="x1:faultDetail"/>
</wsdl:message>

<wsdl:portType name="Greeter">
  <wsdl:operation name="sayHi">
    <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
    <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
  </wsdl:operation>

  <wsdl:operation name="greetMe">
    <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
    <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
  </wsdl:operation>

  <wsdl:operation name="greetMeOneWay">
    <wsdl:input message="tns:greetMeOneWayRequest" name="greetMeOneWayRequest"/>
  </wsdl:operation>

  <wsdl:operation name="pingMe">
```

```
    <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
    <wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
    <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
  ...
</wsdl:binding>

<wsdl:service name="SOAPService">
  <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location="http://localhost:9000/SoapContext/SoapPort"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

**例 26.1 “helloworld WSDL Contract” 中定义的 Greeter 接口定义了以下操作：**

**sayhi - Hass a output parameter of xsd:string.**

**greetMe - 具有 xsd:string 的输入参数，以及 xsd:string 的 output 参数。**

**greetMeOneWay - Has a single input 参数，xsd:string.由于此操作没有输出参数，因此它被优化为单向调用（即，消费者不会等待服务器的响应）。**

**pingMe - Has no 输入参数，没有输出参数，但可能会引发错误异常。**

# 第 27 章 顶级服务开发

**摘要**

在开发服务提供商的顶端方法中，您从 WSDL 文档开始，该文档定义服务提供程序将实施的操作和方法。使用 WSDL 文档，您可以为服务提供商生成起点代码。将业务逻辑添加到生成的代码中是使用正常的 Java 编程 API 完成的。

## 27.1. JAX-WS 服务提供商开发概述

获得 WSDL 文档后，开发 JAX-WS 服务提供商的流程如下：

1. 第 27.2 节 “生成开始点代码” 起点代码。
2. 实施 服务提供商的操作。
3. 第 31 章 *发布服务* 实施的服务。

## 27.2. 生成开始点代码

**概述**

JAX-WS 指定从 WSDL 中定义的服务到将实施该服务作为服务提供商的 Java 类的详细映射。由 wsdl:portType 元素定义的逻辑接口映射到服务端点接口(SEI)。WSDL 中定义的任何复杂类型都映射到 Java 类，遵循 Java 架构为 XML 绑定(JAXB)规范定义的映射。wsdl:service 元素定义的端点也会生成在 Java 类中，供使用者用于访问实施该服务的服务提供商。

cxf-codegen-plugin Maven 插件会生成此代码。它还为您提供了为实施生成起点代码的选项。代码生成器提供了多个控制生成的代码的选项。

运行代码生成器

例 27.1 “Service Code Generation” 演示了如何使用代码生成器为服务生成起点代码。

**例 27.1. Service Code Generation**

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>wsdl</wsdl>
            <extraargs>
              <extraarg>-server</extraarg>
              <extraarg>-impl</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

这执行以下操作：

- **-impl** 选项为 **WSDL** 合同中的每个 **wsdl:portType** 元素生成一个 **shell** 实施类。
- **-server** 选项生成一个简单的 **main** （），将您的服务提供商作为独立应用程序运行。
- **sourceRoot** 指定生成的代码被写入一个名为 ***outputDir*** 的目录。
- **WSDL** 元素指定生成代码的 **WSDL** 合同。

有关代码生成器选项的完整列表，请参阅 第 44.2 节 “cxf-codegen-plugin”。

**生成的代码**

表 27.1 “为服务提供商生成的类” 描述为创建服务提供商生成的文件。

**表 27.1. 为服务提供商生成的类**

| File | 描述 |
| --- | --- |
| ***portTypeName*.java** | SEI。此文件包含您的服务提供商实施的接口。您不应该编辑此文件。 |
| ***serviceName*.java** | 端点。此文件包含 Java 类使用者，用于对服务发出请求。 |
| ***portTypeName*Impl.java** | 框架实施类。修改此文件以构建您的服务提供商。 |
| ***portTypeName*Server.java** | 一个基本的服务器主线，允许您将服务提供商部署为独立进程。如需更多信息，请参阅 第 31 章 *发布服务*。 |

此外，代码生成器将为 WSDL 合同中定义的所有类型生成 Java 类。

### 生成的软件包

生成的代码根据 WSDL 合同中使用的命名空间放入软件包中。为支持服务生成的类（基于 wsdl:portType 元素、wsdl:service 元素和 wsdl:port 元素）将放置到基于 WSDL 合同目标命名空间中的软件包中。为实施合同的 type 元素中定义的类，根据 type 元素的 targetNamespace 属性将放置在软件包中。

映射算法如下：

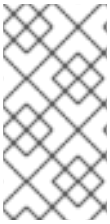
1. 前面的 http:// 或 urn:// 被剥离命名空间。
2. 如果命名空间中的第一个字符串是有效的互联网域，例如以 .com 或 .gov 结束，则前导 www. 将去除该字符串，其余两个组件将被解译。
3. 如果命名空间中的最终字符串以模式 .xxx 或 .xx 的文件扩展名结尾，则扩展会被剥离。
4. 命名空间中的其余字符串附加到生成的字符串中，并以点分开。

5. 所有字母都是小写。

## 27.3. 实施服务提供商

### 生成实施代码

您可以使用代码生成器的 -impl 标志生成用于构建服务提供商的实施类。



**注意**

如果您的服务合同包含 XML Schema 中定义的任何自定义类型，您必须确保该类型的类已生成并可用。

有关使用代码生成器的更多信息，请参阅 第 44.2 节 “cxf-codegen-plugin”。

### 生成的代码

实现代码由两个文件组成：

- *portTypeName*.java - 服务的服务接口(SEI)。
- *portTypeName*Impl.java - 用于实现服务定义的操作的类。

### 实施操作的逻辑

要为您的服务操作提供业务逻辑，请在 *portTypeName*Impl.java 中完成 stub 方法。您通常使用标准 Java 来实施业务逻辑。如果您的服务使用自定义 XML Schema 类型，则必须对每种类型使用生成的类来操作它们。还有一些 Apache CXF 特定的 API，可用于访问一些高级功能。

### Example

例如： 例 26.1 “helloworld WSDL Contract” 中定义的服务的实现类可能类似于 例 27.2 “Greeter Service 的实现”。只有以粗体突出显示的代码部分才能被编程者插入。

**例 27.2. Greeter Service 的实现**

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")

public class GreeterImpl implements Greeter {

    public String greetMe(String me) {
        System.out.println("Executing operation greetMe"); System.out.println("Message received: " +
me + "\n"); return "Hello " + me;
    }

    public void greetMeOneWay(String me) {
        System.out.println("Executing operation greetMeOneWay\n"); System.out.println("Hello there
" + me);
    }

    public String sayHi() {
        System.out.println("Executing operation sayHi\n"); return "Bonjour";
    }

    public void pingMe() throws PingMeFault {
        FaultDetail faultDetail = new FaultDetail(); faultDetail.setMajor((short)2);
faultDetail.setMinor((short)1); System.out.println("Executing operation pingMe, throwing
PingMeFault exception\n"); throw new PingMeFault("PingMeFault raised by server", faultDetail);
    }
}
```

# 第 28 章 从 WSDL 合同开发一个消费者

**摘要**

创建消费者的一种方法是从 WSDL 合同开始。该合同定义了消费者发出请求的服务的操作、消息和传输详情。消费者的起点代码由 WSDL 合同生成。消费者所需的功能添加到生成的代码中。

## 28.1. 生成 STUB 代码

### 概述

cxf-codegen-plugin Maven 插件生成来自 WSDL 合同的 stub 代码。stub 代码提供了在远程服务上调用操作所需的支持代码。

对于消费者，cxf-codegen-plugin Maven 插件会生成以下类型的代码：

- Stub 代码 - 支持用于实施消费者的文件。
- 起点代码 - 连接到远程服务的示例代码，并在远程服务上调用每个操作。

### 生成消费者代码

要生成消费者代码，请使用 cxf-codegen-plugin Maven 插件。例 28.1 “消费者代码生成” 演示了如何使用代码生成器生成消费者代码。

**例 28.1. 消费者代码生成**

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
```

```
        <wsdlOptions>
          <wsdlOption>
            <wsdl>wsdl</wsdl>
            <extraargs>
              <extraarg>-client</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

其中 *outputDir* 是放置生成的文件的目录位置， *wsdl* 指定 WSDL 合同的位置。-client 选项为消费者的 main （） 方法生成起点代码。

有关 cxf-codegen-plugin Maven 插件的参数的完整列表，请参阅 第 44.2 节 “cxf-codegen-plugin”。

生成的代码

代码生成插件为 例 26.1 “helloworld WSDL Contract” 中显示的合同生成以下 Java 软件包：

- org.apache.hello_world_soap_http - 这个软件包是从 http://apache.org/hello_world_soap_http 目标命名空间生成的。此命名空间中定义的所有 WSDL 实体（例如， Greeter 端口类型和 SOAPService 服务）映射到 Java 类。
- org.apache.hello_world_soap_http.types - 这个软件包是从 http://apache.org/hello_world_soap_http/types 目标命名空间生成的。此命名空间中定义的所有 XML 类型（即 HelloWorld 合同的 wsdl:types 元素中定义的一切）都会映射到此 Java 软件包中的 Java 类。

由 cxf-codegen-plugin Maven 插件生成的 stub 文件属于以下类别：

- 在 org.apache.hello_world_soap_http 软件包中代表 WSDL 实体的类。生成以下类来代表 WSDL 实体：
  -

greeter - 代表 Greeter wsdl:portType 元素的 Java 接口。在 JAX-WS 术语中，此 Java 接口是服务端点接口(SEI)。

- SOAPService - Java 服务类(extending javax.xml.ws.Service)，它代表 SOAPService wsdl:service 元素。
- PingMeFault - Java 异常类(extending java.lang.Exception)代表 pingMeFault wsdl:fault 元素。

- 在 org.objectweb.hello_world_soap_http.types 软件包中代表 XML 类型的类。在 HelloWorld 示例中，唯一生成的类型是请求和回复消息的各种打包程序。其中一些数据类型可用于异步调用模型。

## 28.2. 实施一个 CONSUMER

### 概述

要在从 WSDL 合同开始时实施消费者，您必须使用以下 stubs：

- 服务类
- SEI

使用这些存根时，使用者代码会实例化服务代理来在远程服务上发出请求。它还实施使用者的业务逻辑。

### 生成的服务类

例 28.2 “生成的 Service Class 的概述” 显示生成的服务类 *ServiceName*_Service的典型概述[2]，它扩展了 javax.xml.ws.Service 基本类。

**例 28.2. 生成的 Service Class 的概述**

```
@WebServiceClient(name="..." targetNamespace="..."
                  wsdlLocation="...")
public class ServiceName extends javax.xml.ws.Service
{
```

```
  ...
  public ServiceName(URL wsdlLocation, QName serviceName) { }

  public ServiceName() { }

  // Available only if you specify '-fe cxf' option in wsdl2java
  public ServiceName(Bus bus) { }

  @WebEndpoint(name="...")
  public SEI getPortName() { }
  .
  .
  .
}
```

例 28.2 “生成的 Service Class 的概述” 中的 ***ServiceName*** 类定义以下方法：

- **service Name (URL wsdlLocation, QName serviceName) - 根据 wsdl:service 元素中的带有 QName *ServiceName* 服务的 wsdl:service 元素中的数据构建服务对象，该对象可从 *wsdlLocation* 获取的 WSDL 合同中。**
- ***servicename*（） - 默认构造器。它根据服务名称和在生成 stub 代码时提供的 WSDL 合同构建服务对象（例如，在运行 wsdl2java 工具时）。使用此构造器假定 WSDL 合同在指定位置仍然可用。**
- ***serviceName*(Bus bus) - (CXF specific) 一个额外的构造器，允许您指定用于配置该服务的 Bus 实例。这在多线程应用程序上下文中很有用，其中多个总线实例可以与不同的线程关联。此构造器提供了一种简单的方法，可确保您指定的总线是此服务一起使用的总线。仅在调用 wsdl2java 工具时指定 -fe cxf 选项时才可用。**
- **get*PortName*（） - 为 wsdl:port 元素定义的端点返回代理，其 name 属性等于 *PortName*。为 *ServiceName* 服务定义的每个 wsdl:port 元素生成 getter 方法。wsdl:service 元素包含多个端点定义，生成带有多个 get*PortName*（）方法的 服务类。**

服务端点接口

对于原始 WSDL 合同中定义的每个接口，您可以生成对应的 SEI。服务端点接口是 wsdl:portType 元素的 Java 映射。原始 wsdl:portType 元素中定义的每个操作都映射到 SEI 中的对应方法。操作的参数映射如下： .输入参数映射到 method 参数。

1. 第一个输出参数映射到一个返回值。

2. 如果有多个输出参数，第二个和后续输出参数映射到方法参数（更多，必须使用 Holder 类型传递这些参数的值）。

例如，例 28.3 “Greeter 服务端点接口” 显示 Greeter SEI，它从 例 26.1 “helloworld WSDL Contract” 中定义的 wsdl:portType 元素生成。为简单起见，例 28.3 “Greeter 服务端点接口” 省略标准 JAXB 和 JAX-WS 注释。

**例 28.3. Greeter 服务端点接口**

```
package org.apache.hello_world_soap_http;
  ...
public interface Greeter
{
  public String sayHi();
  public String greetMe(String requestType);
  public void greetMeOneWay(String requestType);
  public void pingMe() throws PingMeFault;
}
```

### 消费者主要功能

例 28.4 “消费者实施代码” 显示实施 HelloWorld 使用者的代码。消费者连接到 SOAPService 服务上的 SoapPort 端口，然后继续调用 Greeter 端口类型支持的每个操作。

**例 28.4. 消费者实施代码**

```
package demo.hw.client;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.SOAPService;

public final class Client {

  private static final QName SERVICE_NAME =
  new QName("http://apache.org/hello_world_soap_http",
            "SOAPService");

  private Client()
  {
```

```
 }

 public static void main(String args[]) throws Exception
 {
if (args.length == 0)
  {
    System.out.println("please specify wsdl");
    System.exit(1);
  }

URL wsdlURL;
  File wsdlFile = new File(args[0]);
  if (wsdlFile.exists())
  {
    wsdlURL = wsdlFile.toURL();
  }
  else
  {
    wsdlURL = new URL(args[0]);
  }

  System.out.println(wsdlURL);
SOAPService ss = new SOAPService(wsdlURL,SERVICE_NAME);
Greeter port = ss.getSoapPort();
  String resp;

System.out.println("Invoking sayHi...");
  resp = port.sayHi();
  System.out.println("Server responded with: " + resp);
  System.out.println();

  System.out.println("Invoking greetMe...");
  resp = port.greetMe(System.getProperty("user.name"));
  System.out.println("Server responded with: " + resp);
  System.out.println();

  System.out.println("Invoking greetMeOneWay...");
  port.greetMeOneWay(System.getProperty("user.name"));
  System.out.println("No response from server as method is OneWay");
  System.out.println();

 try {
    System.out.println("Invoking pingMe, expecting exception...");
    port.pingMe();
  } catch (PingMeFault ex) {
    System.out.println("Expected exception: PingMeFault has occurred.");
    System.out.println(ex.toString());
  }
  System.exit(0);
 }
}
```

**例 28.4 “消费者实施代码” 中的 Client.main （） 方法按如下方式进行：**

只要 Apache CXF 运行时类位于您的 classpath 上，则会隐式初始化运行时。不需要调用特殊功能来初始化 Apache CXF。

消费者需要一个字符串参数，它为 HelloWorld 提供 WSDL 合同的位置。WSDL 合同的位置存储在 wsdlURL 中。

您可以使用构造器创建服务对象，它需要 WSDL 合同的位置和服务名称。 调用适当的 get*PortName*（） 方法以获取所需端口的实例。在这种情况下，SOAPService 服务只支持 SoapPort 端口，它实现了 Greeter 服务端点接口。

消费者调用 Greeter 服务端点接口支持的每个方法。

对于 pingMe （） 方法，示例代码演示了如何捕获 PingMeFault 错误异常。

### 使用 -fe cxf 选项生成的客户端代理

如果您在 wsdl2java （选择 cxf frontend）中指定 -fe cxf 选项来生成客户端代理，则生成的客户端代理代码最好与 Java 7 集成。在这种情况下，当调用 get*ServiceName*Port （） 方法时，您可以返回一个类型，它是 SEI 的子接口，并实现以下附加接口：

- java.lang.AutoCloseable
- javax.xml.ws.BindingProvider (JAX-WS 2.0)
- org.apache.cxf.endpoint.Client

要查看这如何简化使用客户端代理，请考虑以下 Java 代码示例，使用标准 JAX-WS 代理对象编写：

```
// Programming with standard JAX-WS proxy object
//
(ServiceNamePortType port = service.getServiceNamePort();
((BindingProvider)port).getRequestContext()
        .put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, address);
port.serviceMethod(...);
((Closeable)port).close();
```

**将前面的代码与以下等效代码示例进行比较，使用 cxf frontend 生成的代码编写：**

```
// Programming with proxy generated using '-fe cxf' option
//
try (ServiceNamePortTypeProxy port = service.getServiceNamePort()) {
    port.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, address);
    port.serviceMethod(...);
}
```

---

**[2]**
**如果 wsdl:service 元素的 name 属性在 Service 中结束，则不会使用 _Service。**

# 第 29 章 在运行时查找 WSDL

**摘要**

**将 WSDL 文档的位置硬编码到应用程序中不可扩展。在实际部署环境中，您希望允许 WSDL 文档的位置在运行时解决。Apache CXF 提供了很多工具来执行此操作。**

## 29.1. LOCATING WSDL 文档的机制

**在使用 JAX-WS API 开发消费者时，您必须提供定义服务的 WSDL 文档的硬编码路径。虽然这在小环境中正常，但使用硬编码路径在企业部署中无法正常工作。**

**要解决这个问题，Apache CXF 提供了三种机制来删除使用硬编码路径的要求：**

- **第 29.2 节 “通过注入实例化代理”**
- **第 29.3 节 “使用 JAX-WS 目录”**
- **第 29.4 节 “使用合同解析器”**

**注意**

**将代理注入您的实施代码通常是最佳选择，因为它是最容易实施的。它只需要客户端端点和配置文件来注入和实例化服务代理。**

## 29.2. 通过注入实例化代理

**概述**

**Apache CXF 使用 Spring Framework 可让您避免使用 JAX-WS API 创建服务代理。它允许您在配置文件中定义客户端端点，然后将代理直接注入实施代码。当运行时实例化 implementation 对象时，它还会根据配置实例化外部服务的代理。该实施通过引用实例化代理来分发。**

**由于代理使用配置文件中的信息实例化，因此 WSDL 位置不需要硬编码。它在部署时可以更改。您还可以指定运行时应搜索 WSDL 的类路径。**

流程

要将外部服务的代理注入服务提供商的实现中，请执行以下操作：

1. 在已知位置（应用程序的所有部分都可以访问）中部署所需的 WSDL 文档。

注意

如果您要将应用程序部署为 WAR 文件，建议您将所有 WSDL 文档和 XML Schema 文档放在 WAR 的 WEB-INF/wsdl 文件夹中。

注意

如果您要将应用部署为 JAR 文件，建议您将所有 WSDL 文档和 XML Schema 文档放在 JAR 的 META-INF/wsdl 文件夹中。

2. 为 要注入的代理配置 JAX-WS 客户端端点。

3. 使用 @Resource 注释，将代理 注入到 您的服务中。

配置代理

您可以使用应用配置文件中的 jaxws:client 元素配置 JAX-WS 客户端端点。这将告知运行时使用指定属性实例化 org.apache.cxf.jaxws.JaxWsClientProxy 对象。此对象是要注入到服务提供商的代理。

您至少需要为以下属性提供值：

- id- 指定用于识别要注入的客户端的 ID。
- serviceClass-Spec 标识代理发出请求的服务的 SEI。

例 29.1 “配置要注入服务实施中的代理” 显示 JAX-WS 客户端端点的配置。

■

**例 29.1. 配置要注入服务实施中的代理**

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:client id="bookClient"
                serviceClass="org.apache.cxf.demo.BookService"
                wsdlLocation="classpath:books.wsdl"/>
  ...
</beans>
```

**注意**

在 例 29.1 “配置要注入服务实施中的代理” 中，wsdlLocation 属性指示运行时从类路径加载 WSDL。如果 books.wsdl 在 classpath 上，则运行时将能够找到它。

有关配置 JAX-WS 客户端的详情，请参考 第 17.2 节 “配置消费者端点”。

对供应商实施进行编码

您可以使用 @Resource 将配置的代理注入服务实施中，如 例 29.2 “将代理注入服务实现” 所示。

**例 29.2. 将代理注入服务实现**

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")
public class StoreImpl implements Store {

@Resource(name="bookClient") private BookService proxy;

}
```

该注释的 name 属性对应于 JAX-WS 客户端的 id 属性的值。配置的代理会在注释后立即注入到 BookService 对象中。您可以使用此对象在代理的外部服务上进行调用。

## 29.3. 使用 JAX-WS 目录

### 概述

JAX-WS 规范要求所有实现都支持：

在解析属于 Web 服务描述的任何 Web 服务文档时，要使用的标准目录功能，特别是 WSDL 和 XML Schema 文档。

此目录工具使用 OASIS 指定的 XML 目录工具。所有采用 WSDL URI 的 JAX-WS API 和注释都使用目录解析 WSDL 文档的位置。

这意味着，您可以提供一个 XML 目录文件，将 WSDL 文档的位置重写为套件特定的部署环境。

### 编写目录

JAX-WS 目录是 OASIS XML Catalogs 1.1 规范定义的标准 XML 目录。它们允许您指定映射：

- 文档的公共标识符和/或系统标识符到 URI。
- 到另一个 URI 的资源的 URI。

表 29.1 “常见 JAX-WS 目录元素” 列出用于 WSDL 位置解析的一些常见元素。

**表 29.1. 常见 JAX-WS 目录元素**

| 元素 | 描述 |
| --- | --- |
| **uri** | 将 URI 映射到备用 URI。 |
| **rewriteURI** | 重写 URI 的开头。例如，这个元素允许您将以 http://cxf.apache.org 开头的所有 URI 映射到以 classpath: 开头的 URI。 |
| **uriSuffix** | 根据原始 URI 的后缀将 URI 映射到备用 URI。例如，您可以将以 foo.xsd 结尾的所有 URI 映射到 classpath:foo.xsd。 |

打包目录

JAX-WS 规范要求用于解析 WSDL 和 XML 架构文档的目录使用名为 META-INF/jax-ws-catalog.xml 的所有可用资源来组装。如果您的应用程序被打包成单个 JAR 或 WAR，您可以将目录放在单个文件中。

如果您的应用打包为多个 JAR，您可以将目录分成多个文件。每个目录文件可以模块化，以仅处理特定 JAR 中的代码访问的 WSDL。

## 29.4. 使用合同解析器

概述

在运行时解析 WSDL 文档位置的最涉及的机制是实施您自己的自定义合同解析器。这要求您提供 Apache CXF 特定 ServiceContractResolver 接口的实现。您还需要将自定义解析器注册到总线。

正确注册后，将使用自定义合同解析器来解决任何所需的 WSDL 和架构文档的位置。

实施合同解析器

合同解析器是 org.apache.cxf.endpoint.ServiceContractResolver 接口的实现。如 例 29.3 “ServiceContractResolver Interface” 所示，这个接口有一个方法 getContractLocation （），需要实现它。getContractLocation （） 采用服务的 QName，并返回服务的 WSDL 合同的 URI。

**例 29.3. ServiceContractResolver Interface**

```
public interface ServiceContractResolver
{
  URI getContractLocation(QName qname);
}
```

用于解析 WSDL 合同位置的逻辑特定于应用。您可以添加可解析来自 UDDI registry、数据库、文件系统上的自定义位置或您选择的任何其他机制的逻辑。

以编程方式注册合同解析器

在 Apache CXF 运行时将使用您的合同解析器前，您必须将其注册到合同解析器 registry。合同解析器 registry 实施 org.apache.cxf.endpoint.ServiceContractResolverRegistry 接口。但是，您不需要

实施自己的 registry。Apache CXF 在 org.apache.cxf.endpoint.ServiceContractResolverRegistryImpl 类中提供默认实现。

要将合同解析器注册到默认 registry，请执行以下操作：

1. 获取对默认总线对象的引用。
2. 使用总线的 getExtension （） 方法从总线获取服务合同 registry。
3. 创建合同解析器的实例。
4. 使用 registry 的 register （） 方法将合同解析器注册到 registry。

例 29.4 “注册合同解析器” 显示将合同解析器注册到默认 registry 的代码。

例 29.4. 注册合同解析器

```
BusFactory bf=BusFactory.newInstance();
Bus bus=bf.createBus();

ServiceContractResolverRegistry registry = bus.getExtension(ServiceContractResolverRegistry);

JarServiceContractResolver resolver = new JarServiceContractResolver();

registry.register(resolver);
```

例 29.4 “注册合同解析器” 中的代码执行以下操作：

获取总线实例。

获取总线的合同解析器 registry。

创建合同解析器实例。

将合同解析器注册到 registry。

### 使用配置注册合同解析器

您还可以实施合同解析器，以便通过配置将其添加到客户端。合同解析器采用这样一种方式，即当运行时读取配置并实例化解析器（解析器）时，解析器会自行注册。由于运行时处理初始化，因此您可以在运行时决定客户端是否需要使用合同解析器。

要实现合同解析器，以便可以通过配置将其添加到客户端中：

1. 在您的合同解析器实施中添加 init （） 方法。
2. 在您的 init （） 方法中添加逻辑，以将合同解析器注册到合同解析器 registry，如 例 29.4 “注册合同解析器” 所示。
3. 使用 @PostConstruct 注释拒绝 init （） 方法。

例 29.5 “可使用配置注册的服务合同解析器” 显示了可以使用 配置添加到客户端中的合同解析器实施。

**例 29.5. 可使用配置注册的服务合同解析器**

```
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.xml.namespace.QName;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;

public class UddiResolver implements ServiceContractResolver
{
  private Bus bus;
  ...

  @PostConstruct
  public void init()
  {
    BusFactory bf=BusFactory.newInstance();
    Bus bus=bf.createBus();
    if (null != bus)
    {
      ServiceContractResolverRegistry resolverRegistry =
```

```
    bus.getExtension(ServiceContractResolverRegistry.class);
        if (resolverRegistry != null)
        {
          resolverRegistry.register(this);
        }
      }
    }

    public URI getContractLocation(QName serviceName)
    {
      ...
    }
  }
```

要将合同解析器注册到客户端，您需要向客户端的配置中添加 bean 元素。bean 元素的 class 属性是实施合同解析器的类的名称。

例 29.6 “Bean 配置合同解析器” 显示一个 bean，用于添加由 org.apache.cxf.demos.myContractResolver 类实现的配置解析器。

**例 29.6. Bean 配置合同解析器**

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  ...
  <bean id="myResolver" class="org.apache.cxf.demos.myContractResolver" />
  ...
</beans>
```

**合同解析顺序**

创建新代理时，运行时使用合同注册表解析器来查找远程服务的 WSDL 合同。合同解析器 registry 按照注册解析器的顺序调用每个合同解析器的 getContractLocation （） 方法。它返回从其中一个注册的合同解析器返回的第一个 URI。

如果您注册了一个在已知已知共享文件系统上解析 WSDL 合同的合同解析器，则它是唯一使用的合同解析器。但是，如果您随后注册了一个使用 UDDI 注册表解析 WSDL 位置的合同解析器，registry 就可以使用两个解析器来定位服务的 WSDL 合同。该注册表首先会尝试使用共享文件系统合同解析器查找合同。如果该合同解析器失败，registry 将尝试使用 UDDI 合同解析器查找它。

# 第 30 章 通用故障处理

## 摘要

JAX-WS 规范定义了两种类型的错误。一个是通用的 JAX-WS 运行时异常。另一个是消息处理过程中抛出的协议特定的异常。

## 30.1. 运行时故障

### 概述

大多数 JAX-WS API 会抛出通用 javax.xml.ws.WebServiceException 异常。

### 抛出 WebServiceException 的 API

表 30.1 “Throw WebServiceException” 列出可以抛出通用 WebServiceException 异常的一些 JAX-WS API。

表 30.1. Throw WebServiceException

| API | 原因 |
| --- | --- |
| **Binding.setHandlerChain()** | 处理程序链配置中出现错误。 |
| **BindingProvider.getEndpointReference()** | 指定的类不是从 **W3CEndpointReference** 分配。 |
| **assign.invoke （）** | Dispatch 实例的配置中有一个错误，或者在与服务通信时出现错误。 |
| **Dispatch.invokeAsync()** | Dispatch 实例的配置中出现错误。 |
| **Dispatch.invokeOneWay()** | Dispatch 实例的配置中有一个错误，或者在与服务通信时出现错误。 |
| **LogicalMessage.getPayload()** | 当使用提供的 **JAXBContext** 到 unmarshall 有效负载时会出现一个错误。WebServiceException 的 **cause** 字段包含原始 JAXBException。 |

| API | 原因 |
| --- | --- |
| **LogicalMessage.setPayload()** | 设置消息的有效负载时会出现错误。如果使用 **JAXBContext** 时抛出异常，WebServiceException 的 **cause** 字段包含原始 JAXBException。 |
| **WebServiceContext.getEndpointReference()** | 指定的类不是从 **W3CEndpointReference** 分配。 |

## 30.2. 协议故障

### 概述

当处理请求过程中发生错误时，会抛出协议异常。所有同步远程调用可能会抛出协议异常。根本原因是在消费者的消息处理链或服务提供商中发生。

JAX-WS 规范定义了通用协议异常。它还指定特定于 SOAP 的协议异常和特定于 HTTP 的协议异常。

### 协议例外的类型

JAX-WS 规范定义了三种类型的协议异常。哪个例外情况取决于应用程序使用的传输和绑定。

表 30.2 “通用协议例外的类型” 描述三种类型的协议异常，以及何时抛出。

**表 30.2. 通用协议例外的类型**

| 例外类 | 星期三 |
| --- | --- |
| javax.xml.ws.ProtocolException | 这个例外是通用协议例外。无论使用的协议是什么，都可能会发现它。如果您使用 SOAP 绑定或 HTTP 绑定，则可以将其转换为特定的故障类型。将 XML 绑定与 HTTP 或 JMS 传输结合使用时，无法将通用协议异常转换为更具体的故障类型。 |
| javax.xml.ws.soap.SOAPFaultException | 在使用 SOAP 绑定时，远程调用会抛出此异常。如需更多信息，请参阅 “使用 SOAP 协议异常”一节。 |
| javax.xml.ws.http.HTTPException | 当使用 Apache CXF HTTP 绑定开发 RESTful Web 服务时，会抛出这个异常。如需更多信息，请参阅 第 VI 部分 “开发 RESTful Web 服务”。 |

### 使用 SOAP 协议异常

SOAPFaultException 异常包装 SOAP 错误。底层 SOAP 故障作为 javax.xml.soap.SOAPFault 对象存储在 fault 字段中。

如果服务实施需要抛出异常，不适合于为应用程序创建的任何自定义异常，则它可以使用例外创建者嵌套在 SOAPFaultException 中出现错误，并将其返回给消费者。例 30.1 “抛出 SOAP 协议例外” 显示在方法传递无效参数时用于创建和丢弃 SOAPFaultException 的代码。

**例 30.1. 抛出 SOAP 协议例外**

```
public Quote getQuote(String ticker)
{
  ...
  if(tickers.length()<3)
  {
    SOAPFault fault = SOAPFactory.newInstance().createFault();
    fault.setFaultString("Ticker too short");
    throw new SOAPFaultException(fault);
  }
  ...
}
```

当消费者捕获 SOAPFaultException 异常时，它们可以通过检查嵌套的 SOAPFault 异常来检索异常的底层原因。如 例 30.2 “从 SOAP 协议例外获取故障” 所示，使用 SOAPFaultException 异常的 getFault （） 方法检索 SOAPFault 异常。

**例 30.2. 从 SOAP 协议例外获取故障**

```
...
try
{
  proxy.getQuote(ticker);
}
catch (SOAPFaultException sfe)
{
  SOAPFault fault = sfe.getFault();
  ...
}
```

# 第 31 章 发布服务

**摘要**

当您要将 JAX-WS 服务部署为独立 Java 应用时，您必须明确实施发布服务提供程序的代码。

## 31.1. 发布服务的时间

Apache CXF 提供了多种方法将服务发布为服务提供商。如何发布服务取决于您所使用的部署环境。Apache CXF 支持的许多容器都不需要为发布端点写入逻辑。有两个例外：

- 将服务器部署为独立 Java 应用程序
- 在没有蓝图的情况下将服务器部署到 OSGi 容器中

有关将应用程序部署到支持的容器中的详细信息，请参阅 第 IV 部分 “配置 Web 服务端点”。

## 31.2. 用于发布服务的 API

**概述**

javax.xml.ws.Enddpoint 类执行发布 JAX-WS 服务提供商的工作。要发布端点，请执行以下操作：

1. 为您的服务提供商创建 Endpoint 对象。
2. 发布端点。
3. 在应用程序关闭时停止端点。

Endpoint 类提供用于创建和发布服务提供商的方法。它还提供了一种方法，可以在单一方法调用中创建和发布服务提供商。

**实例化服务提供商**

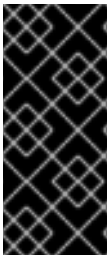
服务供应商使用 Endpoint 对象实例化。您可以使用以下方法之一为您的服务提供商实例化 Endpoint 对象：

- 静态端点创建对象实施或 此 create（） 方法返回指定服务实施的端点。Endpoint 对象是使用实现类的 javax.xml.ws.BindingType 注解（如果存在）提供的信息创建的。如果注解不存在，则 Endpoint 使用默认的 SOAP 1.1/HTTP 绑定。

- 静态端点创建URI绑定ID对象实施或 这个 create （） 方法使用指定的绑定返回指定实现对象的 Endpoint 对象。如果存在，此方法会覆盖 javax.xml.ws.BindingType 注解提供的绑定信息。如果无法解析 bindingID，或者是 null，则 javax.xml.ws.BindingType 中指定的绑定用于创建 Endpoint。如果无法使用 bindingID 或 javax.xml.ws.BindingType，则 端点 是使用默认的 SOAP 1.1/HTTP 绑定创建的。

- 静态端点发布字符串地址对象实施或 publish （） 方法为指定的实现创建一个 Endpoint 对象，并发布它。用于 Endpoint 对象的绑定由提供的地址的 URL 方案 决定。对于支持 URL 方案的绑定，会扫描可用于实现的绑定列表。如果找到了 Endpoint 对象，则已创建并发布。如果没有找到，则方法会失败。

  使用 publish （） 方法与调用其中一个 create （） 方法相同，然后调用 ???TITLE??? 中使用的 publish （） 方法。



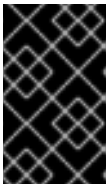
重要

传递给任何 Endpoint 创建方法的实现对象必须是带有 javax.jws.WebService 注解的类实例，并满足 SEI 实现的要求，或者它必须是使用 javax.xml.ws.WebServiceProvider 注解的类的实例，并实施 Provider 接口。

发布服务提供商

您可以使用以下 Endpoint 方法之一发布服务提供商：

- 发布String地址 此 publish （） 方法，在指定的地址发布服务提供商。



重要

地址的 URL 方案必须与其中一个服务提供商绑定兼容。

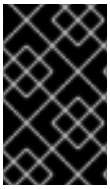
- 发布ObjectserverContext 此 publish （） 方法，根据指定服务器上下文中提供的信息发布服务提供商。服务器上下文必须为端点定义一个地址，上下文也必须与其中一个服务提供商的可用绑定兼容。

停止公布的服务提供商

当不再需要服务提供商时，您应该使用其 stop （） 方法来停止该服务。在 例 31.1 “停止发布端点的方法” 中显示的 stop （） 方法会关闭端点并清理它使用的任何资源。

例 31.1. 停止发布端点的方法

```
stop
```



重要

端点停止后，无法重新发布。

## 31.3. 在 PLAIN JAVA APPLICATION 中发布服务

概述

当您要将应用程序部署为普通 java 应用时，您需要实施在应用的 main （） 方法中发布端点的逻辑。Apache CXF 提供两个编写应用程序的 main （） 方法的选项。

- 使用 wsdl2java 工具生成的 main （） 方法
- 编写发布端点的自定义 main （） 方法

生成服务器主线

代码生成器 -server 标志使工具生成一个简单的服务器主线。生成的服务器主线（如 例 31.2 “生成的服务器主线” 所示），为指定 WSDL 合同中的每个 端口 元素发布一个服务提供商。

如需更多信息，请参阅 第 44.2 节 “cxf-codegen-plugin”。

例 31.2 “生成的服务器主线” 显示生成的服务器主线。

**例 31.2. 生成的服务器主线**

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer {

    protected GreeterServer() throws Exception {
        System.out.println("Starting Server");
        Object implementor = new GreeterImpl();
        String address = "http://localhost:9000/SoapContext/SoapPort";
        Endpoint.publish(address, implementor);
    }

    public static void main(String args[]) throws Exception {
        new GreeterServer();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

例 31.2 “生成的服务器主线” 中的代码执行以下操作：

实例化服务实施对象的副本。

根据端点合同中 wsdl:port 元素的 address 子级的内容为端点创建 地址。

发布端点。

编写服务器主线

如果您使用 Java 第一开发模型，或者您不想使用生成的服务器主线，您可以自行编写。要编写服务器主行，您必须执行以下操作：

1. “实例化服务提供商”一节 服务提供商的 javax.xml.ws.Endpoint 对象。

2. 创建在发布服务提供商时使用的可选服务器上下文。

3. “发布服务提供商”一节 服务供应商使用其中一个 publish （） 方法。

4. 当应用准备好退出时，停止服务提供商。

例 31.3 “自定义服务器主线” 显示发布服务提供商的代码。

**例 31.3. 自定义服务器主线**

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer
{
  protected GreeterServer() throws Exception
  {
  }

  public static void main(String args[]) throws Exception
  {
    GreeterImpl impl = new GreeterImpl();
    Endpoint endpt.create(impl);
    endpt.publish("http://localhost:9000/SoapContext/SoapPort");

    boolean done = false;
   while(!done)
    {
      ...
    }

   endpt.stop();
    System.exit(0);
  }
}
```

例 31.3 “自定义服务器主线” 中的代码执行以下操作：

实例化服务实施对象的副本。

为服务实施创建未发布 的端点。

在 http://localhost:9000/SoapContext/SoapPort 发布服务提供商。

循环，直到服务器关闭为止。

停止发布的端点。

## 31.4. 在 OSGI 容器中发布服务

### 概述

当您开发要部署到 OSGi 容器中的应用程序时，您需要使用打包的捆绑包的生命周期协调您的端点发布和停止。您希望在捆绑包启动时发布端点，并且您希望在捆绑包停止时停止端点。

您可以通过实施 OSGi 捆绑器将端点生命周期绑定到捆绑包的生命周期。OSGi 容器使用捆绑包激活器，在捆绑包启动时为其创建资源。容器也使用捆绑包激活器在捆绑包停止时清除捆绑包资源。

### 捆绑包激活器接口

您可以通过实施 org.osgi.framework.BundleActivator 接口来为应用程序创建一个捆绑包激活器。在 例 31.4 “捆绑包激活器接口” 中显示的 BundleActivator 接口有两个需要实施的方法。

**例 31.4. 捆绑包激活器接口**

```
interface BundleActivator
{
  public void start(BundleContext context)
  throws java.lang.Exception;

  public void stop(BundleContext context)
  throws java.lang.Exception;
}
```

当容器启动捆绑包时，容器会调用 **start** （） 方法。这是您实例化并发布端点的位置。

当容器停止捆绑包时，容器会调用 **stop** （） 方法。这是您将停止端点的位置。

### 实施 start 方法

捆绑包激活器的 **start** 方法是您发布端点的位置。要发布端点，**start** 方法必须执行以下操作：

1. “实例化服务提供商”一节 服务提供商的 **javax.xml.ws.Endpoint** 对象。
2. 创建在发布服务提供商时使用的可选服务器上下文。
3. “发布服务提供商”一节 服务供应商使用其中一个 **publish** （） 方法。

例 31.5 “捆绑包激活器开始方法发布端点” 显示发布服务提供商的代码。

**例 31.5. 捆绑包激活器开始方法发布端点**

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void start(BundleContext context)
  {
    WidgetOrderImpl impl = new WidgetOrderImpl();
    endpt = Endpoint.create(impl);
    endpt.publish("http://localhost:9000/SoapContext/SoapPort");
  }

  ...

}
```

例 31.5 “捆绑包激活器开始方法发布端点” 中的代码执行以下操作：

实例化服务实施对象的副本。

为服务实施创建未发布 的端点。

在 http://localhost:9000/SoapContext/SoapPort 发布服务提供商。

### 实施 stop 方法

捆绑包激活器的 stop 方法是您清理应用程序使用的资源的位置。其实施应包含用于停止应用发布的所有端点的逻辑。

例 31.6 “Bundle Activator Stop Method for Stopping a Endpoint” 显示停止发布端点的 stop 方法。

**例 31.6. Bundle Activator Stop Method for Stopping a Endpoint**

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void stop(BundleContext context)
  {
    endpt.stop();
  }

  ...

}
```

### 通知容器

您必须添加告知容器，应用程序的捆绑包包含捆绑包。您可以通过在捆绑包的清单中添加 Bundle-Activator 属性来完成此操作。此属性告知容器在激活捆绑包时要使用的捆绑包中的类。其值是实施捆绑包器的类的完全限定名称。

例 31.7 “捆绑包激活器清单条目” 显示捆绑包的清单条目，其 activator 由类 com.widgetvendor.osgi.widgetActivator 实施。

**例 31.7. 捆绑包激活器清单条目**

```
Bundle-Activator: com.widgetvendor.osgi.widgetActivator
```

# 第 32 章 基本数据绑定概念

**摘要**

有很多常规主题适用于 Apache CXF 处理类型映射的方式。

## 32.1. 包含和导入架构定义

**概述**

Apache CXF 支持包括和导入 schema 定义，使用 include 和 import schema 标签。这些标签允许您将外部文件或资源的定义插入到 schema 元素的范围。包括和导入之间的基本区别是：

- 包括在属于与封闭 schema 元素相同的目标命名空间的定义中。
- 导入会导致定义属于来自分隔 schema 元素的不同目标命名空间。

**xsd:include 语法**

include 指令使用以下语法：

```
<include schemaLocation="anyURI" />
```

由 *anyURI* 提供的引用模式必须属于与外围模式相同的目标命名空间，或者根本不属于任何目标命名空间。如果引用的模式不属于任何目标命名空间，则在包含该模式时会自动考虑它的命名空间。

例 32.1 “包含 Another Schema 的 Schema 示例” 显示了包含另一个 XML 架构文档的 XML 架构文档示例。

**例 32.1. 包含 Another Schema 的 Schema 示例**

```
<definitions targetNamespace="http://schemas.redhat.com/tests/schema_parser"
             xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <include schemaLocation="included.xsd"/>
```

```
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="tns:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  ...
</definitions>
```

例 32.2 “包含的 Schema 示例” 显示包含的架构文件的内容。

**例 32.2. 包含的 Schema 示例**

```
<schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

**xsd:import 语法**

**import 指令的语法如下：**

```
<import namespace="namespaceAnyURI"
        schemaLocation="schemaAnyURI" />
```

导入的定义必须属于 ***namespaceAnyURI*** *目标命名空间*。如果 ***namespaceAnyURI*** 为空或未指定，则导入的模式定义是不限定的。

例 32.3 “Imports Another Schema 的 Schema 示例” 显示了导入另一个 XML 架构的 XML 架构示例。

**例 32.3. Imports Another Schema 的 Schema 示例**

```
<definitions targetNamespace="http://schemas.redhat.com/tests/schema_parser"
             xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
             xmlns:imp="http://schemas.redhat.com/tests/imported_types"
```

```
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
          xmlns="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://schemas.redhat.com/tests/imported_types"
           schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="imp:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  ...
</definitions>
```

例 32.4 “导入的 Schema 示例” 显示导入的模式文件的内容。

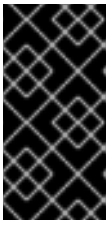
**例 32.4. 导入的 Schema 示例**

```
<schema targetNamespace="http://schemas.redhat.com/tests/imported_types"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

### 使用非引用模式文档

使用架构文档中定义的类型，在服务的 WSDL 文档中没有引用这个类型包含三个步骤：

1. 使用 xsd2wsdl 工具将架构文档转换为 WSDL 文档。
2. 在生成的 WSDL 文档中使用 wsdl2java 工具为类型生成 Java。



**重要**

您将从 wsdl2java 工具中收到一条警告，表示 WSDL 文档没有定义任何服务。您可以忽略这个警告。

3. 将生成的类添加到您的 classpath 中。

## 32.2. XML 命名空间映射

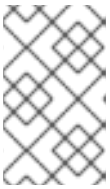
### 概述

XML 架构类型、组和元素定义使用命名空间进行范围。命名空间可防止使用相同名称的实体之间可能存在命名。Java 软件包提供了类似目的。因此，Apache CXF 将 schema 文档的目标命名空间映射到包含实现架构文档中定义的结构所必需的软件包中。

### 软件包命名

生成的软件包的名称通过以下算法从模式的目标命名空间派生：

1. URI 方案（如果存在）被剥离。



**注意**

Apache CXF 将仅剥离 http:、https: 和 urn: 方案。

例如，命名空间 http:\\www.widgetvendor.com\types\widgetTypes.xsd 变为 \\widgetvendor.com\types\widgetTypes.xsd。

2. 尾随文件类型标识符（如果存在）。

例如，\\www.widgetvendor.com\types\widgetTypes.xsd 变为 \\widgetvendor.com\types\widgetTypes。

3. 生成的字符串使用 / 和 : 作为分隔符分成字符串列表。

因此，\\www.widgetvendor.com\types\widgetTypes 变为列表 {"www.widegetvendor.com", "types", "widgetTypes"}。

4. 如果列表中的第一个字符串是一个互联网域名，它被处理如下：

   a. 领导的 www. 被剥离。

   b. 剩余的字符串使用 . 作为分隔符，被分成多个部分。

   c. 列表的顺序被撤销。

   因此, {"www.widegetvendor.com", "types", "widgetTypes"} 变为 {"com", "widegetvendor", "types", "widgetTypes"}

**注意**

Internet 域名以以下之一结尾： .com、.net、.edu、.org、.gov，或位于两个字母国家代码之一。

5. 字符串将转换为所有小写。

   因此, {"com", "widegetvendor", "types", "widgetTypes"} 变为 {"com", "widegetvendor", "types", "widgettypes"}。

6. 字符串被规范化为有效的 Java 软件包名称组件，如下所示：

   a. 如果字符串包含任何特殊字符，则特殊字符将转换为下划线(_)。

   b. 如果有任何字符串是 Java 关键字，则关键字以下划线(__)作为前缀。

   c. 如果有任何字符串以 numeral 开头，则字符串以下划线(_)作为前缀。

7. 字符串使用 . 作为分隔符进行串联。

   因此, {"com", "widegetvendor", "types", "widgettypes"} 变为软件包名称 com.widgetvendor.types.widgettypes。

命名空间 http:\\www.widgetvendor.com\types\widgetTypes.xsd 中定义的 XML Schema 结构映射到 Java 软件包 com.widgetvendor.types.widgettypes。

软件包内容

生成的 JAXB 软件包包含以下内容：

- 一个类实现架构中定义的每个复杂类型

  有关复杂类型映射的详情，请参考 第 35 章 *使用复杂类型*。

- 使用 enumeration facet 定义的任何简单类型的 enum 类型

  有关如何映射枚举的更多信息，请参阅 第 34.3 节 “Enumerations”。

- 一个公共 ObjectFactory 类，其中包含从 schema 中实例化对象的方法

  如需有关 ObjectFactory 类的更多信息，请参阅 第 32.3 节 “Object Factory”。

- 一个 package-info.java 文件，提供有关软件包中类的元数据

## 32.3. OBJECT FACTORY

概述

JAXB 使用对象工厂提供用于实例化 JAXB 生成的构造实例的机制。对象工厂包含实例化软件包范围中定义的所有 XML 模式构造的方法。唯一的例外是枚举不会获取对象工厂中的创建方法。

复杂类型工厂方法

对于为实现 XML 模式复杂类型生成的每个 Java 类，对象工厂包含创建类实例的方法。这个方法的格式如下：

```
typeName createtypeName();
```

例如，如果您的模式包含了一个名为 widgetType 的复杂类型，Apache CXF 会生成名为 WidgetType 的类来实现它。例 32.5 “复杂类型对象工厂条目” 在对象工厂中显示生成的创建方法。

**例 32.5. 复杂类型对象工厂条目**

```
public class ObjectFactory
{
  ...
  WidgetType createWidgetType()
  {
    return new WidgetType();
  }
  ...
}
```

**元素工厂方法**

对于模式全局范围内声明的元素，Apache CXF 将工厂方法插入到对象工厂中。如 第 33 章 *使用 XML 元素* 所述，XML Schema 元素映射到 JAXBElement<T> 对象。创建方法的格式如下：

```
public JAXBElement<elementType> createelementName(elementType value);
```

例如，如果您有一个类型为 xsd:string 的项，则 Apache CXF 会生成对象工厂方法。 例 32.6 “元素工厂条目”

**例 32.6. 元素工厂条目**

```
public class ObjectFactory
{
    ...
    @XmlElementDecl(namespace = "...", name = "comment")
    public JAXBElement<String> createComment(String value) {
        return new JAXBElement<String>(_Comment_QNAME, String.class, null, value);
    }
    ...
}
```

## 32.4. 将类添加到 RUNTIME MARSHALLER

### 概述

当 Apache CXF 运行时读取和写入 XML 数据时，它使用映射将 XML 架构类型与其代表 Java 类型相关联。默认情况下，映射包含 WSDL 合同 schema 元素的目标命名空间中定义的所有类型。它还包含从导入 WSDL 合同的任何模式的命名空间生成的任何类型。

应用 schema 元素使用的 namespace 以外的类型是使用 @XmlSeeAlso 注释来完成的。如果您的应用需要处理应用 WSDL 文档范围之外的类型，您可以编辑 @XmlSeeAlso 注释，将它们添加到 JAXB 映射中。

### 使用 @XmlSeeAlso 注释

@XmlSeeAlso 注释可以添加到您的服务的 SEI 中。它包含要包含在 JAXB 上下文中的以逗号分隔的类列表。例 32.7 “将类添加到 JAXB 上下文的语法” 显示使用 @XmlSeeAlso 注释的语法。

**例 32.7. 将类添加到 JAXB 上下文的语法**

```
import javax.xml.bind.annotation.XmlSeeAlso;
    @WebService()
    @XmlSeeAlso({Class1.class, Class2.class, ..., ClassN.class})
    public class GeneratedSEI {
        ...
    }
```

如果您有权访问 JAXB 生成的类，使用为支持所需类型生成的 ObjectFactory 类更高效。包含 ObjectFactory 类，包括对象工厂已知的所有类。

### Example

例 32.8 “将类添加到 JAXB 上下文” 显示带有 @XmlSeeAlso 标注的 SEI。

**例 32.8. 将类添加到 JAXB 上下文**

```
...
import javax.xml.bind.annotation.XmlSeeAlso;
...
    @WebService()
    @XmlSeeAlso({org.apache.schemas.types.test.ObjectFactory.class,
org.apache.schemas.tests.group_test.ObjectFactory.class})
```

```
public interface Foo {
    ...
}
```

# 第 33 章 使用 XML 元素

**摘要**

XML 架构元素用于定义 XML 文档中的元素实例。元素可以在 XML 架构文档的全局范围内定义，或者定义为复杂类型的成员。当它们在全局范围内定义时，Apache CXF 将它们映射到一个 JAXB 元素类，以便更轻松地操作它们。

## 概述

XML 文档中的一个元素实例由 XML 架构文档全局范围内的 XML Schema 元素定义，以便 Java 开发人员更轻松地处理元素，Apache CXF 将全局范围的元素映射到特殊的 JAXB 元素类或生成的 Java 类，以匹配其内容类型。

该元素是如何映射的，这取决于元素是否使用 type 属性引用的命名类型定义，或者该元素是使用 in-line 类型定义定义的。使用 in-line 类型定义定义的元素映射到 Java 类。

建议使用命名类型来定义元素，因为 in-line 类型无法被架构中的其他元素重复使用。

## XML 架构映射

在 XML Schema 元素中，使用 element 项定义。元素 具有一个必需属性。name 指定在 XML 文档中出现的元素名称。

除了 name 属性 元素 外，还在 表 33.1 “用于定义元素的属性” 中列出的可选属性。

**表 33.1. 用于定义元素的属性**

| 属性 | 描述 |
| --- | --- |
| **type** | 指定元素的类型。类型可以是任何 XML 架构原语类型，也可以是合同中定义的任何指定复杂类型。如果没有指定此属性，则需要包含 in-line 类型定义。 |
| **nillable** | 指定元素是否完全保留在文档中。如果 **nillable** 设为 **true**，则元素可以在使用 schema 生成的任何文档中省略。 |

| 属性 | 描述 |
| --- | --- |
| **abstract** | 指定是否在实例文档中使用元素。**true** 表示元素不能出现在实例文档中。相反，其 replace **Group** 属性包含此元素的 QName 的另一个元素必须出现在此元素的位置。有关此属性如何生成代码的信息，请参考 "抽象元素的 Java 映射"一节。 |
| **substitutionGroup** | 指定可替换为此元素的元素名称。有关使用类型替换的详情，请参考 第 37 章 *元素替换*。 |
| **default** | 为元素指定默认值。有关此属性如何生成代码的信息，请参考 "使用默认值的元素映射 Java"一节。 |
| **已修复** | 为元素指定一个固定值。 |

**例 33.1 "简单的 XML 架构元素定义" 显示一个简单的元素定义。**

**例 33.1. 简单的 XML 架构元素定义**

```
<element name="joeFred" type="xsd:string" />
```

**元素也可以使用在线类型定义定义自己的类型。使用 complexType 元素或 simpleType 元素指定 in-line 类型。指定数据类型是否复杂或简单后，您可以使用每种类型数据的工具定义所需的任何类型的数据。**

**例 33.2 "使用 In-Line Type 的 XML 架构元素定义" 显示带有 in-line 类型定义的元素定义。**

**例 33.2. 使用 In-Line Type 的 XML 架构元素定义**

```
<element name="skate">
  <complexType>
    <sequence>
      <element name="numWheels" type="xsd:int" />
      <element name="brand" type="xsd:string" />
    </sequence>
  </complexType>
</element>
```

**带有指定类型的元素的 JAVA 映射**

默认情况下，全局定义元素映射到 **JAXBElement<T>** 对象，其中模板类由 element 元素的 type 属性的值决定。对于原语类型，模板类使用 **"wrapper 类"一节** 中描述的打包程序类映射衍生而来。对于复杂类型，生成的 Java 类支持复杂类型用作模板类。

为了支持映射和减轻开发人员对元素的 QName 的不必要的担心，会为每个全局定义的元素生成一个对象工厂方法，如 **例 33.3 "全局范围元素的对象工厂方法"** 所示。

**例 33.3. 全局范围元素的对象工厂方法**

```
public class ObjectFactory {

    private final static QName _name_QNAME = new QName("targetNamespace", "localName");

    ...

    @XmlElementDecl(namespace = "targetNamespace", name = "localName")
    public JAXBElement<type> createname(type value);

}
```

例如，**例 33.1 "简单的 XML 架构元素定义"** 中定义的元素会导致对象工厂方法在 **例 33.4 "简单元素的对象工厂"** 中显示。

**例 33.4. 简单元素的对象工厂**

```
public class ObjectFactory {

    private final static QName _JoeFred_QNAME = new QName("...", "joeFred");

    ...

    @XmlElementDecl(namespace = "...", name = "joeFred")
    public JAXBElement<String> createJoeFred(String value);

}
```

**例 33.5 "使用全局范围元素"** 显示了在 Java 中使用全局范围的元素的示例。

**例 33.5. 使用全局范围元素**

```
JAXBElement<String> element = createJoeFred("Green");
String color = element.getValue();
```

### 在 WSDL 中使用带有命名类型的元素

如果使用全局范围元素来定义消息部分，则生成的 Java 参数不是 JAXBElement<T> 实例。相反，它被映射到常规的 Java 类型或类。

根据 例 33.6 “WSDL 将元素用作消息部分” 中显示的 WSDL 片段，生成的方法具有类型为 String 的参数。

**例 33.6. WSDL 将元素用作消息部分**

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"
                  xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
          xmlns="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"><element name="sayHi">
      <element name="sayHi" type="string"/>
      <element name="sayHiResponse" type="string"/>
    </schema>
  </wsdl:types>

  <wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
  </wsdl:message>
  <wsdl:message name="sayHiResponse">
    <wsdl:part element="x1:sayHiResponse" name="out"/>
  </wsdl:message>

  <wsdl:portType name="Greeter">
    <wsdl:operation name="sayHi">
      <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
      <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

例 33.7 “使用全局元素作为部分的 Java 方法” 显示为 sayHi 操作生成的方法签名。

**例 33.7. 使用全局元素作为部分的 Java 方法**

字符串sayHi字符串

### 使用 IN-LINE 类型进行元素映射的 JAVA 映射

当使用 in-line 类型定义元素时，它会按照与用于将其他类型映射到 Java 相同的规则映射到 Java。简单类型的规则在 第 34 章 *使用简单类型* 中进行了描述。第 35 章 *使用复杂类型* 中描述了复杂类型的规则。

当为带有在线类型定义的元素生成 Java 类时，生成的类使用 @XmlRootElement 注释进行解码。@XmlRootElement 注释有两个有用的属性： name 和 namespace。这些属性在 表 33.2 “@XmlRootElement Annotation 的属性” 中进行了描述。

表 33.2. @XmlRootElement Annotation 的属性

| 属性 | 描述 |
| --- | --- |
| **name** | 指定 XML Schema 元素的 name 属性的值。 |
| **namespace** | 指定定义元素的命名空间。如果在目标命名空间中定义了此元素，则不会指定属性。 |

如果元素满足以下一个或多个条件，则不会使用 @XmlRootElement 注释：

- 元素的 nillable 属性设为 true
- 元素是替换组的 head 元素

  有关替换组的详情，请参考 第 37 章 *元素替换*。

### 抽象元素的 JAVA 映射

当元素的 abstract 属性设置为 true 时，不会生成实例化类型的实例的对象工厂方法。如果使用在线类型定义元素，则生成支持 in-line 类型的 Java 类。

### 使用默认值的元素映射 JAVA

当使用元素 的默认 属性时，defaultValue 属性将添加到生成的 @XmlElementDecl 注释中。例如，例 33.8 “使用默认值的 XML 架构元素” 中定义的元素会导致对象工厂方法在 例 33.9 “具有默认值的元素的对象工厂方法” 中显示。

**例 33.8. 使用默认值的 XML 架构元素**

```
<element name="size" type="xsd:int" default="7"/>
```

**例 33.9. 具有默认值的元素的对象工厂方法**

```
@XmlElementDecl(namespace = "...", name = "size", defaultValue = "7")
public JAXBElement<Integer> createUnionJoe(Integer value) {
    return new JAXBElement<Integer>(_Size_QNAME, Integer.class, null, value);
}
```

# 第 34 章 使用简单类型

**摘要**

XML Schema 简单类型是 XML 架构原语类型，如 xsd:int，或使用 simpleType 元素定义。它们用于指定不包含任何子对象或属性的元素。它们通常映射到原生 Java 构造，不需要生成特殊类来实施它们。Enumerated simple 类型不会导致生成的代码，因为它们被映射到 Java enum 类型。

## 34.1. 原语类型

**概述**

当使用 XML Schema 原语类型定义消息部分时，生成的参数的类型映射到对应的 Java 原生类型。在映射复杂类型范围中定义的元素时，使用相同的模式。生成的字段是对应的 Java 原生类型。

**映射**

表 34.1 “XML Schema 原语类型到 Java 原生类型映射” 列出 XML 架构原语类型和 Java 原生类型之间的映射。

表 34.1. XML Schema 原语类型到 Java 原生类型映射

| XML Schema Type | Java 类型 |
|---|---|
| xsd:string | 字符串 |
| xsd:integer | BigInteger |
| xsd:int | int |
| xsd:long | long |
| xsd:short | short |
| xsd:decimal | BigDecimal |
| xsd:float | 浮点值 |
| xsd:double | double |
| xsd:boolean | 布尔值 |
| xsd:byte | byte |

| XML Schema Type | Java 类型 |
| --- | --- |
| **xsd:QName** | **qname** |
| **xsd:dateTime** | **XMLGregorianCalendar** |
| **xsd:base64Binary** | **byte[]** |
| **xsd:hexBinary** | **byte[]** |
| **xsd:unsignedInt** | **long** |
| **xsd:unsignedShort** | **int** |
| **xsd:unsignedByte** | **short** |
| **xsd:time** | **XMLGregorianCalendar** |
| **xsd:date** | **XMLGregorianCalendar** |
| **xsd:g** | **XMLGregorianCalendar** |
| **xsd:anySimpleType** [a] | **对象** |
| **xsd:anySimpleType** [b] | **字符串** |
| **xsd:duration** | **Duration** |
| **xsd:NOTATION** | **qname** |

[a] 对于这种类型的元素。

[b] 对于此类型的属性。

**wrapper 类**

**将 XML 架构类型映射到 Java 原语类型不适用于所有可能的 XML 架构结构。一些情况要求 XML 架构原语类型映射到 Java 原语类型对应的打包程序类型。这些情况包括：**

- **一个 元素，它的 nillable 属性设置为 true，如下所示：**

```
<element name="finned" type="xsd:boolean"
        nillable="true" />
```

- 一个 元素，其 minOccurs 属性设置为 0， 它的 maxOccurs 属性设置为 1，或者其 maxOccurs 属性未指定，如下所示：

```
<element name="plane" type="xsd:string" minOccurs="0" />
```

- 一个 attribute 元素，其 use 属性设置为 optional，或未指定，且没有其默认属性及其 固定属性，如下所示：

```
<element name="date">
  <complexType>
    <sequence/>
    <attribute name="calType" type="xsd:string"
               use="optional" />
  </complexType>
</element>
```

表 34.2 “Primitive Schema Type to Java Wrapper Class Mapping” 展示了 XML 架构原语类型如何映射到 Java 打包程序类。

**表 34.2. Primitive Schema Type to Java Wrapper Class Mapping**

| 模式类型 | Java 类型 |
|---|---|
| xsd:int | java.lang.Integer |
| xsd:long | java.lang.Long |
| xsd:short | java.lang.Short |
| xsd:float | java.lang.Float |
| xsd:double | java.lang.Double |
| xsd:boolean | java.lang.Boolean |
| xsd:byte | java.lang.Byte |
| xsd:unsignedByte | java.lang.Short |
| xsd:unsignedShort | java.lang.Integer |
| xsd:unsignedInt | java.lang.Long |
| xsd:unsignedLong | java.math.BigInteger |

| 模式类型 | Java 类型 |
| --- | --- |
| xsd:duration | java.lang.String |

## 34.2. 通过 RESTRICTION 定义的简单类型

### 概述

XML Schema 允许您通过从另一个原语类型或简单类型派生新类型来创建简单的类型。使用 simpleType 元素描述简单类型。

使用一个或多个方面限制 基本类型 来描述新类型。这些方面会限制可以在新类型中存储的可能的有效值。例如，您可以定义一个简单的类型 SSN，它是一个正好 9 个字符的字符串。

每个原语 XML 架构类型都有自己的一组可选问题。

### 流程

要自行定义简单类型，请执行以下操作：

1. 确定您的新简单类型的基本类型。
2. 确定哪个限制根据所选基本类型的可用方面定义新类型。
3. 使用本节中介绍的语法，在合同的 type 部分输入适当的 simpleType 元素。

### 在 XML 架构中定义一个简单的类型

例 34.1 “简单类型语法” 显示描述简单类型的语法。

**例 34.1. 简单类型语法**

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value" />
    <facet value="value" />
```

```
    ...
  </restriction>
</simpleType>
```

类型描述包含在 **simpleType** 元素中，并由 **name** 属性的值标识。定义新简单类型的基本类型由 **xsd:restriction** 元素的基本属性指定。每个 **facet** 元素都在 **限制** 元素中指定。可用的 **facet** 及其有效设置取决于基本类型。例如，**xsd:string** 有很多方面，包括：

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **空格**

**例 34.2 “postal Code Simple Type”** 显示一个简单的类型定义，它代表了用于美国状态的双字母邮政代码。它只能包含两个大写字母。**TX** 是有效的值，但 **tx** 或 **tX** 不是有效的值。

**例 34.2. postal Code Simple Type**

```
<xsd:simpleType name="postalCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{2}" />
  </xsd:restriction>
</xsd:simpleType>
```

**映射到 Java**

**Apache CXF** 将用户定义的简单类型映射到简单类型的基本类型的 **Java** 类型。因此，任何使用简单类型 **postalCode** 的消息（如 **例 34.2 “postal Code Simple Type”** 所示）被映射到 **String**，因为 **postalCode** 的基本类型是 **xsd:string**。例如，**例 34.3 “带有简单类型的信用请求”** 中显示的 **WSDL** 片段生成 **Java** 方法 **state** （），它采用 **字符串** 的参数后代码。

**例 34.3. 带有简单类型的信用请求**

```
<message name="stateRequest">
  <part name="postalCode" type="postalCode" />
</message>
...
<portType name="postalSupport">
  <operation name="state">
    <input message="tns:stateRequest" name="stateRec" />
    <output message="tns:stateResponse" name="credResp" />
  </operation>
</portType>
```

#### enforcing facets

默认情况下，Apache CXF 不会强制任何用于限制简单类型的问题。但是，您可以通过启用模式验证，将 Apache CXF 端点配置为强制面临的问题。

要将 Apache CXF 端点配置为使用 schema 验证，请将 schema-validation-enabled 属性设置为 true。例 34.4 “Service Provider Configured to Use Schema Validation” 显示使用 schema 验证的服务提供商的配置

**例 34.4. Service Provider Configured to Use Schema Validation**

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                wsdlLocation="wsdl/hello_world.wsdl"
                createdFromAPI="true">
  <jaxws:properties>
    <entry key="schema-validation-enabled" value="BOTH" />
  </jaxws:properties>
</jaxws:endpoint>
```

有关配置模式验证的更多信息，请参阅 第 24.3.4.7 节 “模式验证类型值”。

## 34.3. ENUMERATIONS

#### 概述

在 XML Schema 中，枚举的类型是使用 xsd:enumeration facet 定义的简单类型。与 atomic 简单类型不同，它们映射到 Java 枚举。

#### 在 XML Schema 中定义枚举类型

Enumerations 是使用 xsd:enumeration facet 的简单类型。每个 xsd:enumeration facet 为枚举类型定义一个可能的值。

例 34.5 “XML 架构定义枚举” 显示枚举的类型的定义。它有以下可能的值：

- big
- 大
- mungo
- gargantuan

**例 34.5. XML 架构定义枚举**

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
    <enumeration value="gargantuan"/>
  </restriction>
```

**映射到 Java**

XML Schema enumerations，其中基本类型为 xsd:string 会自动映射到 Java enum 类型。您可以使用 第 38.4 节 “自定义枚举映射” 中描述的自定义来指示代码生成器与其他基本类型的枚举映射到 Java enum 类型。

enum 类型创建如下：

1. 类型的名称取自简单类型定义的 name 属性并转换为 Java 标识符。

   通常，这意味着将 XML 架构名称的第一个字符转换为大写字母。如果 XML 架构名称的第一个字符是无效字符，则名称前将加上下划线(_)。

2. 对于每个 枚举 的问题，会根据 value 属性的值生成一个枚举常量。

   常数的名称通过将值中的所有小写字母转换为等效的大写来派生。

3. 生成构造器，它将获取从枚举的基本类型映射的 Java 类型。

4. 生成了一个名为 value （） 的公共方法来访问由类型实例代表的 facet 值。

   value （） 方法的返回类型是 XML Schema 类型的基本类型。

5. 生成了一个名为 fromValue （） 的公共方法，以根据 facet 值创建 enum 类型的实例。

   value （） 方法的参数类型是 XML Schema 类型的基本类型。

6. 类使用 @XmlEnum 注释进行解码。

例 34.5 “XML 架构定义枚举” 中定义的枚举类型映射到 例 34.6 “为 String Bases XML Schema Enumeration 生成 Enumerated Type” 中显示的 enum 类型。

**例 34.6. 为 String Bases XML Schema Enumeration 生成 Enumerated Type**

```
@XmlType(name = "widgetSize")
@XmlEnum
public enum WidgetSize {

    @XmlEnumValue("big")
    BIG("big"),
    @XmlEnumValue("large")
    LARGE("large"),
    @XmlEnumValue("mungo")
    MUNGO("mungo"),
    @XmlEnumValue("gargantuan")
    GARGANTUAN("gargantuan");
    private final String value;

    WidgetSize(String v) {
        value = v;
    }
```

```
    public String value() {
        return value;
    }

    public static WidgetSize fromValue(String v) {
        for (WidgetSize c: WidgetSize.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }

}
```

### 34.4. 列表

#### 概述

XML Schema 支持定义数据类型的机制，它们是以空格分隔的简单类型的列表。例 34.7 “列出类型示例” 中显示一个使用列表类型的元素示例 primeList。

**例 34.7. 列出类型示例**

```
<primeList>1 3 5 7 9 11 13<\primeList>
```

XML 架构列表类型通常映射到 Java List<T> 对象。这个模式的唯一变化是消息部分直接映射到 XML Schema 列表类型的实例。

#### 在 XML 架构中定义列表类型

XML Schema 列表类型是一个简单的类型，因此通过使用 simpleType 元素来定义。例 34.8 “XML 架构列表类型的语法” 中显示用来定义列表类型的最常见语法。

**例 34.8. XML 架构列表类型的语法**

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value" />
    <facet value="value" />
    ...
  </list>
</simpleType>
```

为 *atomicType* 给出的值定义列表中元素的类型。它只能是 XML 架构原子类型中的一个，如 xsd:int 或 xsd:string，也可以是用户定义的简单类型，它不是列表。

除了定义列表类型中列出的元素类型外，您还可以使用 facets 来进一步限制列表类型的属性。表 34.3 “列表类型 Facets” 显示列表类型所使用的 facet。

**表 34.3. 列表类型 Facets**

| facet | 效果 |
| --- | --- |
| **length** | 定义列表类型的实例中元素数量。 |
| **minLength** | 定义列表类型实例中允许的最小元素数。 |
| **maxLength** | 定义列表类型实例中允许的最大元素数。 |
| **Enumeration** | 定义列表类型实例中元素的允许值。 |
| **pattern** | 定义列表类型实例中元素的字典形式。使用正则表达式定义模式。 |

例如，例 34.7 “列出类型示例” 中显示的 simpleList 元素的定义。例 34.9 “列表类型的定义”

**例 34.9. 列表类型的定义**

```
<simpleType name="primeListType">
  <list itemType="int"/>
</simpleType>
<element name="primeList" type="primeListType"/>
```

除了 例 34.8 “XML 架构列表类型的语法” 中显示的语法外，您还可以使用 例 34.10 “列出类型的替代语法” 中显示的通用语法定义一个列表类型。

**例 34.10. 列出类型的替代语法**

```
<simpleType name="listType">
  <list>
    <simpleType>
      <restriction base="atomicType">
        <facet value="value"/>
        <facet value="value"/>
        ...
      </restriction>
```

```
  </simpleType>
</list>
</simpleType>
```

**将列表类型元素映射到 Java**

当元素定义列表类型时，list 类型映射到集合属性。collection 属性是一个 Java List<T> 对象。List<T> 使用的模板类是从列表的基本类型映射的 wrapper 类。例如： 例 34.9 “列表类型的定义” 中定义的列表类型映射到 List<Integer>。

有关打包程序类型映射的详情，请参考 “wrapper 类”一节。

**将列表类型参数映射到 Java**

当消息部分定义为列表类型或映射到列表类型的元素时，生成的 method 参数映射到数组而不是 List<T> 对象。数组的基本类型是列表类型的基本类。

例如，例 34.11 “带有 List Type Message Part 的 WSDL” 中的 WSDL 片段生成方法签名，如 例 34.12 “带有 List Type 参数的 Java 方法” 所示。

**例 34.11. 带有 List Type Message Part 的 WSDL**

```
<definitions ...>
  ...
  <types ...>
    <schema ... >
      <simpleType name="primeListType">
        <list itemType="int"/>
      </simpleType>
      <element name="primeList" type="primeListType"/>
    </schemas>
  </types>
  <message name="numRequest"> <part name="inputData" element="xsd1:primeList" />
</message>
  <message name="numResponse">;
    <part name="outputData" type="xsd:int">
  ...
  <portType name="numberService">
    <operation name="primeProcessor">
      <input name="numRequest" message="tns:numRequest" />
      <output name="numResponse" message="tns:numResponse" />
    </operation>
    ...
```

```
  </portType>
  ...
</definitions>
```

**例 34.12. 带有 List Type 参数的 Java 方法**

```
public interface NumberService {

    @XmlList
    @WebResult(name = "outputData", targetNamespace = "", partName = "outputData")
    @WebMethod
    public int primeProcessor(
        @WebParam(partName = "inputData", name = "primeList", targetNamespace = "...")
java.lang.Integer[] inputData
    );
}
```

## 34.5. UNIONS

### 概述

在 XML Schema 中，union 是一种构造，可用于描述其数据可以是多个简单类型之一的类型。例如，您可以定义一个类型，其值为整数 1 或字符串。unions 映射到 Java Strings。

### 在 XML 架构中定义

XML 架构 unions 使用 simpleType 元素来定义。它们至少包含一个用于定义 联合 成员类型的联合元素。联合的成员类型是可存储在联合实例中的有效数据类型。它们使用 union 元素的 memberTypes 属性来定义。memberTypes 属性的值包含一个或多个定义的简单类型名称的列表。例 34.13 “simple Union Type” 显示可存储整数或字符串的联合定义。

**例 34.13. simple Union Type**

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

除了将指定类型指定为联合的成员类型外，您还可以将匿名简单类型定义为联合的成员类型。这可以通过在 union 元素中添加匿名类型定义来完成。例 34.14 “Union with an Anonymous Member Type” 显示包含匿名成员类型的联合示例，该类型将有效整数的可能值限制为范围 1 到 10。

**例 34.14. Union with an Anonymous Member Type**

```
<simpleType name="restrictedOrderNumUnion">
  <union memberTypes="xsd:string">
    <simpleType>
      <restriction base="xsd:int">
        <minInclusive value="1" />
        <maxInclusive value="10" />
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

**映射到 Java**

XML 架构联合类型映射到 Java String 对象。默认情况下，Apache CXF 不会验证生成的对象的内容。要使 Apache CXF 验证内容，您必须将运行时配置为使用模式验证，如 “enforcing facets”一节 所述。

## 34.6. 简单类型替换

**概述**

XML 允许使用 xsi:type 属性在兼容类型之间进行简单的类型替换。但是，简单类型的默认映射到 Java 原语类型，但并不完全支持简单的类型替换。运行时可以处理基本的简单类型替换，但信息会丢失。可以自定义代码生成器来生成 Java 类，以便无损失简单的类型替换。

**默认映射和 marshaling**

由于 Java 原语类型不支持类型替换，因此简单类型的默认映射到 Java 原语类型会显示支持简单类型替换的问题。如果尝试传递 一个简短的 变量，即使定义类型的架构也允许，Java 虚拟机也会忽略。

要解决 Java 类型系统的限制，Apache CXF 允许当元素的 xsi:type 属性的值满足以下条件之一时，Apache CXF 允许简单的类型替换：

- 它指定一个与元素的 schema 类型兼容的原语类型。
- 它指定根据元素的 schema 类型的限制派生的类型。
- 它指定由扩展从元素的 schema 类型派生的复杂类型。

当运行时执行类型替换时，它不会保留元素 xsi:type 属性中指定的类型的任何知识。如果类型替换是从复杂类型到简单类型，则只保留与简单类型相关的值。通过扩展添加的任何其他元素和属性都将丢失。

### 支持无丢失类型替换

您可以自定义简单类型的生成，以便使用以下方法减少对简单类型替换的丢失支持：

- 将 globalBindings 自定义元素的 mapSimpleTypeDef 设置为 true。

  这指示代码生成器为全局范围中定义的所有名为 simple 类型创建 Java 值类。

  如需更多信息，请参阅 第 38.3 节 “为简单类型生成 Java 类”。

- 在 globalBindings 自定义元素中添加 javaType 元素。

  这指示代码生成器将 XML 架构类型的所有实例映射到 s 的特定类对象。

  如需更多信息，请参阅 第 38.2 节 “指定 XML 架构 Primitive 的 Java 类”。

- 将 baseType 自定义元素添加到您要自定义的特定元素中。

  baseType 自定义元素允许您指定生成的 Java 类型来代表属性。为确保简单类型替换的最佳兼容性，请使用 java.lang.Object 作为基本类型。

  如需更多信息，请参阅 第 38.6 节 “指定元素或属性的基本类型”。

# 第 35 章 使用复杂类型

**摘要**

复杂的类型可以包含多个元素，它们可以包含属性。它们映射到可保存由类型定义表示数据的 Java 类。通常，映射是到一个带有代表元素和内容模型属性的一组属性的 bean。

## 35.1. 基本复杂类型映射

**概述**

XML 架构复杂类型定义包含比简单类型更复杂信息的构造。最简单的复杂类型定义一个带有属性的空元素。更复杂的类型由一组元素组成。

默认情况下，XML 架构复杂类型映射到 Java 类，其 member 变量代表 XML Schema 定义中列出的每个元素和属性。类有 setters 和 getters，用于每个成员变量。

**在 XML 架构中定义**

XML 架构复杂类型使用 complexType 元素定义。complexType 元素嵌套用于定义数据结构的其余元素。它可以显示为指定类型定义的父元素，也可以是匿名定义 元素 中存储的信息结构的子元素。当使用 complexType 元素来定义命名类型时，它需要使用 name 属性。name 属性指定引用类型的唯一标识符。

包含一个或多个元素的复杂类型定义具有 表 35.1 “在复杂类型中定义 Elements Appear 的元素” 中描述的子元素之一。这些元素决定了在类型的实例中如何显示指定的元素。

**表 35.1. 在复杂类型中定义 Elements Appear 的元素**

| 元素 | 描述 |
| --- | --- |
| **all** | 作为复杂类型的一部分定义的所有元素都必须显示在类型的实例中。但是，它们可能会以任何顺序出现。 |
| **choice** | 只有定义为复杂类型一部分的元素之一才能显示在类型的实例中。 |
| **sequence** | 作为复杂类型的一部分定义的所有元素都必须显示在类型的实例中，它们还必须按照类型定义中指定的顺序显示。 |

注意

如果复杂类型定义只使用属性，则不需要 表 35.1 “在复杂类型中定义 Elements Appear 的元素” 中描述的元素之一。

在决定如何显示元素后，您可以通过向定义中添加一个或多个 元素元素 来定义元素。

例 35.1 “XML 架构复杂类型” 显示 XML 架构中的复杂类型定义。

例 35.1. XML 架构复杂类型

```
<complexType name="sequence">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="street" type="xsd:short" />
    <element name="city" type="xsd:string" />
    <element name="state" type="xsd:string" />
    <element name="zipCode" type="xsd:string" />
  </sequence>
</complexType>
```

### 映射到 Java

XML 架构复杂类型映射到 Java 类。复杂类型定义中的每个元素都映射到 Java 类中的 member 变量。也会为复杂类型中的每个元素生成 getter 和 setter 方法。

所有生成的 Java 类都使用 @XmlType 注释进行解码。如果映射用于命名复杂类型，则注解 名称 将设置为 complexType 元素的 name 属性的值。如果复杂类型定义为元素定义的一部分，则 @XmlType 注释的 name 属性的值是 元素的 name 属性的值。

注意

如 “使用 in-line 类型进行元素映射的 Java 映射”一节 所述，如果为定义为元素定义的一部分的复杂类型生成了 @XmlRootElement 注释，则生成的类与 @XmlRootElement 注释分离。

为了为运行时提供指示 XML 架构复杂类型元素的指南，代码生成器会更改用于分离类及其成员变量的注解。

所有复杂类型

所有复杂的类型都使用 all 元素定义。它们被标注如下：

- @XmlType 注释的 propOrder 属性为空。
- 每个元素使用 @XmlElement 注释进行解码。
- @XmlElement 注释 的必要 属性设为 true。

例 35.2 “All Complex Type 的映射” 显示具有两个元素的所有复杂类型的映射。

例 35.2. All Complex Type 的映射

```
@XmlType(name = "all", propOrder = {

})
public class All {
    @XmlElement(required = true)
    protected BigDecimal amount;
    @XmlElement(required = true)
    protected String type;

    public BigDecimal getAmount() {
        return amount;
    }

    public void setAmount(BigDecimal value) {
        this.amount = value;
    }

    public String getType() {
        return type;
    }

    public void setType(String value) {
        this.type = value;
    }
}
```

选择复杂类型

使用 choice 元素定义选择复杂类型。它们被标注如下：

-

@XmlType 注释的 propOrder 属性按照它们出现在 XML 架构定义中的顺序列出元素的名称。

- 没有成员变量都会被注解。

例 35.3 “选择复杂类型的映射” 显示具有两个元素的所选复杂类型的映射。

**例 35.3. 选择复杂类型的映射**

```
@XmlType(name = "choice", propOrder = {
    "address",
    "floater"
})
public class Choice {

    protected Sequence address;
    protected Float floater;

    public Sequence getAddress() {
        return address;
    }

    public void setAddress(Sequence value) {
        this.address = value;
    }

    public Float getFloater() {
        return floater;
    }

    public void setFloater(Float value) {
        this.floater = value;
    }

}
```

### 序列复杂类型

使用 sequence 元素定义一系列复杂类型。它被注解如下：

- @XmlType 注释的 propOrder 属性按照它们出现在 XML 架构定义中的顺序列出元素的名称。

- 每个元素使用 @XmlElement 注释进行解码。

- **@XmlElement 注释 的必要 属性设为 true。**

**例 35.4 “序列复杂类型的映射” 显示 例 35.1 “XML 架构复杂类型” 中定义的复杂类型的映射。**

**例 35.4. 序列复杂类型的映射**

```
@XmlType(name = "sequence", propOrder = {
    "name",
    "street",
    "city",
    "state",
    "zipCode"
})
public class Sequence {

    @XmlElement(required = true)
    protected String name;
    protected short street;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String state;
    @XmlElement(required = true)
    protected String zipCode;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public short getStreet() {
        return street;
    }

    public void setStreet(short value) {
        this.street = value;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String value) {
        this.city = value;
    }

    public String getState() {
        return state;
    }
```

```
    public void setState(String value) {
        this.state = value;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String value) {
        this.zipCode = value;
    }
}
```

## 35.2. 属性

### 概述

Apache CXF 支持在 complexType 元素的范围内使用属性元素和 attribute Group 元素。当为 XML 文档属性声明定义结构时，提供添加标签中指定的信息的方法，而不是标签包含的值。例如，当描述 XML 元素 <value currency ="euro">410<\value> in XML Schema the currency 属性时，使用 例 35.5 “XML 架构定义和属性” 所示的 attribute 项描述。

attributeGroup 元素允许您定义一组可重复使用的属性，这些属性可以被 schema 定义的所有复杂类型引用。例如，如果您要定义一系列都使用属性类别和 pubDate 的元素，您可以使用这些属性定义属性组，并在所有使用该属性的元素中引用它们。这在 例 35.7 “属性组定义” 中显示。

当描述在开发应用程序逻辑中使用的数据类型时，其 use 属性设置为 optional 或 required 的属性将被视为结构的元素。对于复杂类型描述中包含的每个属性声明，在类中为 属性生成元素，以及适当的 getter 和 setter 方法。

### 在 XML 架构中定义属性

XML Schema 属性 元素具有一个必需属性，名为，用于标识属性。它还有四个可选属性，它们在 表 35.2 “用于定义 XML 架构中属性的可选属性” 中进行了描述。

表 35.2. 用于定义 XML 架构中属性的可选属性

| 属性 | 描述 |
| --- | --- |
| 使用 | 指定是否需要属性。有效值为、 可选 ， 或 禁止使用。可选 是默认值。 |

| 属性 | 描述 |
| --- | --- |
| **type** | 指定属性可以采用的值类型。如果没有使用属性的 schema 类型，则必须在线定义。 |
| **default** | 指定用于属性的默认值。只有在 **属性** 元素的 **use** 属性设置为 **optional** 时，才会使用它。 |
| **已修复** | 指定用于属性的固定值。只有在 **属性** 元素的 **use** 属性设置为 **optional** 时，才会使用它。 |

**例 35.5 “XML 架构定义和属性” 显示一个属性元素，定义属性 currency，其值为字符串。**

**例 35.5. XML 架构定义和属性**

```
<element name="value">
  <complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="currency" type="xsd:string"
                       use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

**如果在 attribute 元素中省略了 type 属性，则必须使用命令行描述数据格式。例 35.6 “带有 In-Line Data Description 的属性” 显示 属性 ( category )的一个属性元素，它可以取值 autobiography、non-fiction 或 fiction。**

**例 35.6. 带有 In-Line Data Description 的属性**

```
<attribute name="category" use="required">
  <simpleType>
    <restriction base="xsd:string">
      <enumeration value="autobiography"/>
      <enumeration value="non-fiction"/>
      <enumeration value="fiction"/>
    </restriction>
  </simpleType>
</attribute>
```

**在 XML Schema 中使用属性组**

在复杂类型定义中使用属性组分为两个步骤：

1. 定义属性 group。

   属性组使用带有多个属性子元素的 attribute Group 元素来定义。attributeGroup 需要一个 name 属性，用于定义用于引用属性组的字符串。属性 元素定义属性组的成员，并如 “在 XML 架构中定义属性”一节 所示指定。例 35.7 “属性组定义” 显示属性组 catalogIndecies 的描述。属性组有两个成员： 类别，它是可选的，而 pubDate 是必需的。

   **例 35.7. 属性组定义**

   ```
   <attributeGroup name="catalogIndices">
     <attribute name="category" type="catagoryType" />
     <attribute name="pubDate" type="dateTime"
                use="required" />
   </attributeGroup>
   ```

2. 在复杂类型的定义中使用属性组。

   您可以使用 attributeGroup 元素和 ref 属性在复杂类型定义中使用属性组。ref 属性的值是给定您要用作类型定义一部分的属性组的名称。例如，如果要使用复杂类型 dvdType 中的属性组 catalogIndecies，您可以使用 <attributeGroup ref="catalogIndecies" />，如 例 35.8 “具有属性组的复杂类型” 所示。

   **例 35.8. 具有属性组的复杂类型**

   ```
   <complexType name="dvdType">
     <sequence>
       <element name="title" type="xsd:string" />
       <element name="director" type="xsd:string" />
       <element name="numCopies" type="xsd:int" />
     </sequence>
     <attributeGroup ref="catalogIndices" />
   </complexType>
   ```

**将属性映射到 Java**

属性与 member 元素映射到 Java 非常相似。必要属性和可选属性映射到生成的 Java 类中的成员变量。成员变量使用 @XmlAttribute 注释进行解码。如果需要该属性，则 @XmlAttribute 注释的 required 属性设置为 true。

**例 35.9 “techdoc Description” 中定义的复杂类型映射到 例 35.10 “techdoc Java Class” 中显示的 Java 类。**

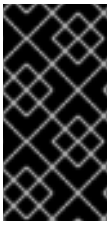
**例 35.9. techdoc Description**

```
<complexType name="techDoc">
  <all>
    <element name="product" type="xsd:string" />
    <element name="version" type="xsd:short" />
  </all>
  <attribute name="usefullness" type="xsd:float"
             use="optional" default="0.01" />
</complexType>
```

**例 35.10. techdoc Java Class**

```
@XmlType(name = "techDoc", propOrder = {

})
public class TechDoc {

    @XmlElement(required = true)
    protected String product;
    protected short version;
    @XmlAttribute protected Float usefullness;

    public String getProduct() {
        return product;
    }

    public void setProduct(String value) {
        this.product = value;
    }

    public short getVersion() {
        return version;
    }

    public void setVersion(short value) {
        this.version = value;
    }

    public float getUsefullness() { if (usefullness == null) { return 0.01F; } else { return usefullness; }
}

    public void setUsefullness(Float value) {
        this.usefullness = value;
    }
}
```

如 例 35.10 “techdoc Java Class” 中所示，default 属性和 fixed 属性指示代码生成器将代码添加到为属性生成的 getter 方法中。此额外代码可确保如果没有设置值，则返回指定的值。



重要

fixed 属性被视为与 default 属性相同。如果您希望 固定 属性被视为 Java 常数，您可以使用 第 38.5 节 “自定义修复的值属性映射” 中描述的自定义。

### 将属性组映射到 Java

属性组映射到 Java，就像类型定义中明确使用组成员一样。如果属性组有三个成员，且在复杂类型中使用，则该类型生成的类将包含 member 变量，以及 attribute 组的每个成员的 getter 和 setter 方法。例如，例 35.8 “具有属性组的复杂类型” 中定义的复杂类型，Apache CXF 会生成一个包含成员变量 类别和 pubDate 的类，以支持属性组的成员，如 例 35.11 “dvdType Java Class” 所示。

**例 35.11. dvdType Java Class**

```
@XmlType(name = "dvdType", propOrder = {
    "title",
    "director",
    "numCopies"
})
public class DvdType {

    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String director;
    protected int numCopies;
    @XmlAttribute protected CatagoryType category; @XmlAttribute(required = true)
@XmlSchemaType(name = "dateTime") protected XMLGregorianCalendar pubDate;

    public String getTitle() {
        return title;
    }

    public void setTitle(String value) {
        this.title = value;
    }

    public String getDirector() {
        return director;
    }

    public void setDirector(String value) {
        this.director = value;
    }

    public int getNumCopies() {
```

```
        return numCopies;
    }

    public void setNumCopies(int value) {
        this.numCopies = value;
    }

    public CatagoryType getCatagory() {
        return catagory;
    }

    public void setCatagory(CatagoryType value) {
        this.catagory = value;
    }

    public XMLGregorianCalendar getPubDate() {
        return pubDate;
    }

    public void setPubDate(XMLGregorianCalendar value) {
        this.pubDate = value;
    }

}
```

### 35.3. 从简单类型派生复杂类型

#### 概述

Apache CXF 支持从简单类型派生复杂类型。根据定义，一个简单的类型没有子元素或属性。因此，从简单类型派生复杂类型的主要原因是将属性添加到简单类型。

从一个简单的类型派生复杂类型的方法有两种：

- 按扩展
- 按限制

#### 按扩展进行派生

例 35.12 “根据扩展从简单类型派生复杂类型” 显示了从 xsd:decimal primitive 类型到包含货币属性的扩展所派生的复杂类型 国际价格 示例。

**例 35.12. 根据扩展从简单类型派生复杂类型**

```
<complexType name="internationalPrice">
  <simpleContent>
    <extension base="xsd:decimal">
      <attribute name="currency" type="xsd:string"/>
    </extension>
  </simpleContent>
  </complexType>
```

simpleContent 元素表示新类型不包含任何子元素。extension 元素指定新类型扩展 xsd:decimal。

### 根据限制派生

例 35.13 “通过 Restriction 从简单类型派生复杂类型” 显示了一个复杂类型 idType 的示例，它由来自 xsd:string 限制派生。定义的类型将 xsd:string的可能值限制为长度为十个字符的值。它还向类型添加一个属性。

**例 35.13. 通过 Restriction 从简单类型派生复杂类型**

```
<complexType name="idType">
  <simpleContent>
    <restriction base="xsd:string">
     <length value="10" />
     <attribute name="expires" type="xsd:dateTime" />
    </restriction>
  </simpleContent>
</complexType>
```

与 例 35.12 “根据扩展从简单类型派生复杂类型” 一样，simpleContent 元素表示新类型不包含任何子项。这个示例使用 Limit 元素 来限制新类型中使用的可能值。attribute 元素将元素添加到新类型。

### 映射到 Java

从简单类型派生的复杂类型映射到使用 @XmlType 注释解码的 Java 类。生成的类包含 member 变量 ( 值 )，这是从中派生复杂类型的简单类型。member 变量使用 @XmlValue 注释进行解码。类也具有 getValue （） 方法和 setValue （） 方法。此外，生成的类具有 member 变量，以及关联的 getter 和 setter 方法，用于扩展简单类型的每个属性。

例 35.14 “idType Java Class” 显示为 例 35.13 “通过 Restriction 从简单类型派生复杂类型” 中定义的 idType 类型生成的 Java 类。

**例 35.14. idType Java Class**

```
@XmlType(name = "idType", propOrder = {
    "value"
})
public class IdType {

    @XmlValue
    protected String value;
    @XmlAttribute
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar expires;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public XMLGregorianCalendar getExpires() {
        return expires;
    }

    public void setExpires(XMLGregorianCalendar value) {
        this.expires = value;
    }

}
```

## 35.4. 从复杂类型派生复杂类型

### 概述

使用 XML Schema，您可以通过使用 complexContent 元素扩展或限制其他复杂类型来生成新的复杂类型。当生成 Java 类来代表派生的复杂类型时，Apache CXF 会扩展基本类型的类。这样，生成的 Java 代码会保留 XML 架构中所预期的继承层次结构。

### 模式语法

您可以使用 complexContent 元素以及 extension 元素或 limitations 元素从其他复杂类型获取复杂类型。complexContent 元素指定包含的数据描述包含多个字段。extension 元素和 limit 元素（即 complexContent 元素的子项）指定要修改的基本类型以创建新类型。基本类型由 base 属性指定。

### 扩展复杂类型

要扩展复杂类型，请使用 extension 元素来定义组成新类型的额外元素和属性。作为新类型定义的一部分，允许复杂类型描述中允许的所有元素。例如，您可以向新类型添加匿名枚举，或者您可以使用 choice 元素来指定每次只有一个新字段可以有效。

例 35.15 “根据扩展派生复杂类型” 显示一个 XML 架构片段，用于定义两个复杂的类型，即 widgetOrderInfo 和 widgetOrderBillInfo。widgetOrderBillInfo 通过扩展 widgetOrderInfo 来包括两个新元素： orderNumber 和 amtDue。

**例 35.15. 根据扩展派生复杂类型**

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element name="order_date" type="xsd:dateTime"/>
    <element name="type" type="xsd1:widgetSize"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:boolean" use="optional" />
</complexType>
<complexType name="widgetOrderBillInfo">
  <complexContent>
    <extension base="xsd1:widgetOrderInfo">
      <sequence>
        <element name="amtDue" type="xsd:decimal"/>
        <element name="orderNumber" type="xsd:string"/>
      </sequence>
      <attribute name="paid" type="xsd:boolean"
                default="false" />
    </extension>
  </complexContent>
</complexType>
```

### 限制复杂类型

要限制复杂类型，请使用 limit 元素来限制基本类型的元素或属性的可能值。在限制复杂类型时，您必须列出基本类型的所有元素和属性。对于每个元素，您可以在定义中添加限制的属性。例如，您可以在元素中添加 maxOccurs 属性来限制它可能发生的次数。您还可以使用 fixed 属性强制一个或多个元素具有预先确定的值。

例 35.16 “通过 Restriction 定义复杂类型” 演示了通过限制另一个复杂类型来定义复杂类型的示例。restricted 类型 wallawallaAddress 只能用于 Walla Walla, Washington 中的地址，因为 city 元素的值、state 元素和 zipCode 元素的值已被修复。

**例 35.16. 通过 Restriction 定义复杂类型**

```
<complexType name="Address">
```

```
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="street" type="xsd:short" maxOccurs="3"/>
    <element name="city" type="xsd:string"/>
    <element name="state" type="xsd:string"/>
    <element name="zipCode" type="xsd:string"/>
  </sequence>
</complexType>
<complexType name="wallawallaAddress">
  <complexContent>
    <restriction base="xsd1:Address">
      <sequence>
        <element name="name" type="xsd:string"/>
        <element name="street" type="xsd:short"
                maxOccurs="3"/>
        <element name="city" type="xsd:string"
                fixed="WallaWalla"/>
        <element name="state" type="xsd:string"
                fixed="WA" />
        <element name="zipCode" type="xsd:string"
                fixed="99362" />
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

**映射到 Java**

**与所有复杂类型一样，Apache CXF 会生成一个类来代表从另一个复杂类型派生的复杂类型。为派生复杂类型生成的 Java 类扩展为支持基本复杂类型生成的 Java 类。基本 Java 类也被修改为包含 @XmlSeeAlso 注释。基本类的 @XmlSeeAlso 注释列出了扩展基本类的所有类。**

**当新的复杂类型由扩展派生时，生成的类将包括所有添加的元素和属性的成员变量。新成员变量将根据与所有其他元素相同的映射生成。**

**当新的复杂类型由限制派生时，生成的类将没有新的成员变量。生成的类将只是不提供任何其他功能的 shell。完全允许您确保强制实施 XML 架构中定义的限制。**

**例如，例 35.15 “根据扩展派生复杂类型” 中的 schema 生成两个 Java 类： WidgetOrderInfo 和 WidgetBillOrderInfo。WidgetOrderBillInfo 扩展 WidgetOrderInfo，因为 widgetOrderBillInfo 派生自 widgetOrderInfo 的扩展。例 35.17 “WidgetOrderBillInfo” 显示为 widgetOrderBillInfo 生成的类。**

**例 35.17. WidgetOrderBillInfo**

```
@XmlType(name = "widgetOrderBillInfo", propOrder = {
    "amtDue",
```

```
    "orderNumber"
})
public class WidgetOrderBillInfo
    extends WidgetOrderInfo
{
    @XmlElement(required = true)
    protected BigDecimal amtDue;
    @XmlElement(required = true)
    protected String orderNumber;
    @XmlAttribute
    protected Boolean paid;

    public BigDecimal getAmtDue() {
        return amtDue;
    }

    public void setAmtDue(BigDecimal value) {
        this.amtDue = value;
    }

    public String getOrderNumber() {
        return orderNumber;
    }

    public void setOrderNumber(String value) {
        this.orderNumber = value;
    }

    public boolean isPaid() {
        if (paid == null) {
            return false;
        } else {
            return paid;
        }
    }

    public void setPaid(Boolean value) {
        this.paid = value;
    }
}
```

## 35.5. 发生约束

### 35.5.1. 模式元素支持 Occurrence 约束

**XML Schema 允许您指定组成复杂类型定义的四个 XML 架构元素上的发生限制：**

- 第 35.5.2 节 “在 All Element 上发生限制”

- 第 35.5.3 节 “选择元素上的发生限制”
- 第 35.5.4 节 “元素上发生约束”
- 第 35.5.5 节 “序列发生的限制”

### 35.5.2. 在 All Element 上发生限制

**XML Schema**

使用 all 元素定义的复杂类型不允许多次出现 all 元素定义的结构。但是，您可以通过将其 minOccurs 属性设置为 0， 使 all 元素定义的结构是可选的。

**映射到 Java**

将 all 元素的 minOccurs 属性设置为 0 对生成的 Java 类没有影响。

### 35.5.3. 选择元素上的发生限制

**概述**

默认情况下，选择 元素的结果只能在复杂类型的实例中显示一次。您可以更改选择代表 选择 使用其 minOccurs 属性及其 mxOccurs 属性定义的结构的次数。您可以使用这些属性指定在复杂类型的实例中将选择类型为零到无限次数。在出现的每个类型时，为选择类型选择的元素不需要相同。

**在 XML 架构中使用**

minOccurs 属性指定必须显示所选类型的最小次数。其值可以是任意正整数。将 minOccurs 属性设置为 0， 指定选择类型不需要显示在复杂类型的实例中。

maxOccurs 属性指定选择类型可以显示的最大次数。其值可以是任何非零、正整数或 未绑定 的值。将 maxOccurs 属性设置为 unbounded 指定选择类型可以显示无限次数。

例 35.18 “选择观察约束” 显示选择类型 ClubEvent 的定义，并有选择发生的限制。选择类型 overall 可以重复 0 到未绑定的时间。

■

**例 35.18. 选择观察约束**

```
<complexType name="ClubEvent">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="MemberName" type="xsd:string"/>
    <element name="GuestName" type="xsd:string"/>
  </choice>
</complexType>
```

**映射到 Java**

与单实例选择结构不同，可以多次实现的 XML 架构选择结构映射到具有单个成员变量的 Java 类。这个单一成员变量是一个 List<T > 对象，其中包含序列多次出现的所有数据。例如：如果 例 35.18 “选择观察约束” 中定义的序列发生两次，则列表将有两个项目。

Java 类的成员变量的名称通过串联 member 元素的名称来派生。元素名称以 Or 分隔，变量名称的第一个字母将转换为小写。例如，从 例 35.18 “选择观察约束” 生成的 member 变量将命名为 memberNameOrGuestName。

列表中存储的对象类型取决于成员元素的类型之间的关系。例如：

- 如果成员元素与生成的列表相同，则生成的列表将包含 JAXBElement<T> 对象。JAXBElement<T> 对象的基本类型由 member 元素类型的正常映射决定。
- 如果成员元素是不同类型的，并且其 Java 表示实现了通用接口，则列表将包含通用接口的对象。
- 如果成员元素是不同类型的，其 Java 表示扩展通用基本类，则列表将包含通用基本类的对象。
- 如果没有满足其他条件，列表将包含对象 对象。

生成的 Java 类将只有 member 变量的 getter 方法。getter 方法返回对 live 列表的引用。对返回列表所做的任何修改都将影响实际对象。

Java 类使用 @XmlType 注释进行解码。注解的 name 属性设为 XML Schema 定义的 parent 元素的 name 属性的值。该注解的 propOrder 属性包含代表序列中元素的单个成员变量。

代表选择结构中的元素的 member 变量使用 @XmlElements 注释进行解码。@XmlElements 注释包含以逗号分隔的 @XmlElement 注释列表。该列表具有一个 @XmlElement 注释，用于类型 XML 架构定义中定义的每个 member 元素。列表中 @XmlElement 注释的 name 属性设为 XML Schema 元素的 name 属性的值，其 type 属性设置为由 XML Schema 元素类型映射生成的 Java 类。

例 35.19 “使用 Occurrence 约束的 Java 代表选择结构” 显示 例 35.18 “选择观察约束” 中定义的 XML 架构选择结构的 Java 映射。

**例 35.19. 使用 Occurrence 约束的 Java 代表选择结构**

```
@XmlType(name = "ClubEvent", propOrder = {
    "memberNameOrGuestName"
})
public class ClubEvent {

    @XmlElementRefs({
        @XmlElementRef(name = "GuestName", type = JAXBElement.class),
        @XmlElementRef(name = "MemberName", type = JAXBElement.class)
    })
    protected List<JAXBElement<String>> memberNameOrGuestName;

    public List<JAXBElement<String>> getMemberNameOrGuestName() {
        if (memberNameOrGuestName == null) {
            memberNameOrGuestName = new ArrayList<JAXBElement<String>>();
        }
        return this.memberNameOrGuestName;
    }

}
```

**minOccurs 设置为 0**

如果只指定 minOccurs 元素，其值为 0，则代码生成器会生成 Java 类，就像未设置 minOccurs 属性一样。

### 35.5.4. 元素上发生约束

**概述**

您可以使用元素元素的 minOccurs 属性和 maxOccurs 属性指定复杂类型中特定元素的次数。这两个属性的默认值为 1。

**minOccurs 设置为 0**

当您将一个复杂类型的 member 元素的 minOccurs 属性设置为 0 时，@XmlElement 注释会减少对应的 Java 成员变量。@XmlElement 注释的 必需 属性设为 false，而不是将其 必需 属性设为 true。

#### minOccurs 设置为大于 1 的值

在 XML Schema 中，您可以通过将元素元素的 minOccurs 属性设置为大于 1 的值来指定一个 元素必须在 type 的实例中发生多次。但是，生成的 Java 类不支持 XML Schema 约束。Apache CXF 生成支持 Java 成员变量，就像未设置 minOccurs 属性一样。

#### 带有 maxOccurs 设置的元素

当您希望 member 元素在复杂类型的实例中出现多次时，您可以将元素的 maxOccurs 属性设置为大于 1 的值。您可以将 maxOccurs 属性的值设置为 unbounded，以指定 member 元素可以显示无限次数。

代码生成器将 maxOccurs 属性设置为大于 1 的 member 元素映射到是一个 List<T > 对象的 Java member 变量。列表的基本类由映射元素的类型到 Java 来确定。对于 XML 架构原语类型，使用 wrapper 类，如 “wrapper 类”一节 所述。例如，如果 member 元素的类型是 xsd:int，则生成的 member 变量是一个 List<Integer> 对象。

### 35.5.5. 序列发生的限制

#### 概述

默认情况下，sequence 元素的内容只能在复杂类型的实例中出现一次。您可以更改序列元素定义的元素序列的次数，允许使用其 minOccurs 属性及其 maxOccurs 属性。通过使用这些属性，您可以指定序列类型在复杂类型的实例中可能会为零到无限次数。

#### 使用 XML 架构

minOccurs 属性指定定义复杂类型的实例中必须执行序列的最小次数。其值可以是任意正整数。将 minOccurs 属性设置为 0， 指定序列不需要显示在复杂类型的实例中。

maxOccurs 属性指定定义复杂类型的实例中可以发生序列的次数上限。其值可以是任何非零、正整数或 未绑定 的值。将 maxOccurs 属性设置为 unbounded 指定序列可能会显示无限次数。

例 35.20 “使用 Occurrence 约束序列” 显示序列类型的定义，Culture Info，带有序列发生的限制。序列可以重复 0 到 2 次。

**例 35.20. 使用 Occurrence 约束序列**

```
<complexType name="CultureInfo">
  <sequence minOccurs="0" maxOccurs="2">
    <element name="Name" type="string"/>
    <element name="Lcid" type="int"/>
  </sequence>
</complexType>
```

**映射到 Java**

与单个实例序列不同，可以多次发生的 XML 架构序列使用单个成员变量映射到 Java 类。这个单一成员变量是一个 List<T > 对象，其中包含序列多次出现的所有数据。例如，如果 例 35.20 “使用 Occurrence 约束序列” 中定义的序列发生两次，则列表将有四个项目。

Java 类的成员变量的名称通过串联 member 元素的名称来派生。元素名称由 And 分隔，变量名称的第一个字母将转换为小写。例如，从 例 35.20 “使用 Occurrence 约束序列” 生成的 member 变量名为 nameAndLcid。

列表中存储的对象类型取决于成员元素的类型之间的关系。例如：

- 如果成员元素与生成的列表相同，则生成的列表将包含 JAXBElement<T> 对象。JAXBElement<T> 对象的基本类型由 member 元素类型的正常映射决定。
- 如果成员元素是不同类型的，并且其 Java 表示实现了通用接口，则列表将包含通用接口的对象。
- 如果成员元素是不同类型的，其 Java 表示扩展通用基本类，则列表将包含通用基本类的对象。
- 如果没有满足其他条件，列表将包含对象 对象。

生成的 Java 类仅具有 member 变量的 getter 方法。getter 方法返回对 live 列表的引用。对返回的列表所做的任何修改都会影响实际对象。

Java 类使用 @XmlType 注释进行解码。注解的 name 属性设为 XML Schema 定义的 parent 元素的 name 属性的值。该注解的 propOrder 属性包含代表序列中元素的单个成员变量。

代表序列中的元素的 member 变量使用 @XmlElements 注释进行解码。@XmlElements 注释包含以逗号分隔的 @XmlElement 注释列表。该列表具有一个 @XmlElement 注释，用于类型 XML 架构定义中定义的每个 member 元素。列表中 @XmlElement 注释的 name 属性设为 XML Schema 元素的 name 属性的值，其 type 属性设置为由 XML Schema 元素类型映射生成的 Java 类。

例 35.21 “带有 Occurrence 约束的 Java 代表序列” 显示 例 35.20 “使用 Occurrence 约束序列” 中定义的 XML 架构序列的 Java 映射。

**例 35.21. 带有 Occurrence 约束的 Java 代表序列**

```
@XmlType(name = "CultureInfo", propOrder = {
    "nameAndLcid"
})
public class CultureInfo {

    @XmlElements({
        @XmlElement(name = "Name", type = String.class),
        @XmlElement(name = "Lcid", type = Integer.class)
    })
    protected List<Serializable> nameAndLcid;

    public List<Serializable> getNameAndLcid() {
        if (nameAndLcid == null) {
            nameAndLcid = new ArrayList<Serializable>();
        }
        return this.nameAndLcid;
    }

}
```

**minOccurs 设置为 0**

如果只指定 minOccurs 元素，其值为 0，则代码生成器会生成 Java 类，就像未设置 minOccurs 属性一样。

## 35.6. 使用模型组

### 概述

XML 架构模型组是方便的快捷方式，允许您从用户定义的复杂类型引用一组元素。例如，您可以定义一组对应用中多种类型通用的元素，然后重复引用组。模型组使用 group 元素定义，类似于复杂的类型定义。模型组到 Java 的映射也类似于复杂类型的映射。

### 在 XML 架构中定义模型组

您可以使用 group 元素和 name 属性在 XML Schema 中定义模型组。name 属性的值是一个字符串，用于引用整个架构中的组。group 元素（如 complexType 元素）可以具有 sequence 元素、all 元素或 choice 元素作为其即时子。

在子元素中，您可以使用 element 元素 定义组的成员。对于组的每个成员，指定一个 element 元素。组成员可以使用 元素 元素的任何标准属性，包括 minOccurs 和 maxOccurs。因此，如果您的组有三个元素，并且其中一个可能发生三次，则您可以使用三个 元素 定义组，其中之一使用 maxOccurs="3"。例 35.22 “XML 架构模型组” 显示具有三个元素的模型组。

**例 35.22. XML 架构模型组**

```
<group name="passenger">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="clubNum" type="xsd:long" />
    <element name="seatPref" type="xsd:string"
             maxOccurs="3" />
  </sequence>
</group>
```

### 在类型定义中使用模型组

定义了模型组后，就可以将其用作复杂类型定义的一部分。要在复杂类型定义中使用模型组，请使用 group 元素和 ref 属性。ref 属性的值是定义时提供给组的名称。例如：要使用 例 35.22 “XML 架构模型组” 中定义的组，请使用 {ref="tns:passenger" >，如 例 35.23 “使用模型组进行复杂类型” 所示。

**例 35.23. 使用模型组进行复杂类型**

```
<complexType name="reservation">
  <sequence>
    <group ref="tns:passenger" />
    <element name="origin" type="xsd:string" />
    <element name="destination" type="xsd:string" />
    <element name="fltNum" type="xsd:long" />
  </sequence>
</complexType>
```

当在类型定义中使用模型组时，组将变为类型的成员。因此，保留 实例有四个成员元素。第一个元素是 passenger 元素，它包含由 例 35.22 “XML 架构模型组” 中显示的组定义的成员元素。例 35.24 “带有模型组的类型实例” 中显示 保留 实例示例。

**例 35.24. 带有模型组的类型实例**

```
<reservation>
  <passenger> <name>A. Smart</name> <clubNum>99</clubNum> <seatPref>isle1</seatPref>
</passenger>
  <origin>LAX</origin>
  <destination>FRA</destination>
  <fltNum>34567</fltNum>
</reservation>
```

### 映射到 Java

默认情况下，模型组仅在包含复杂类型定义中时映射到 Java 工件。为包含模型组的复杂类型生成代码时，Apache CXF 将模型组的成员变量包含在为类型生成的 Java 类中。代表模型组的成员变量根据模型组的定义标注。

例 35.25 “使用组键入” 显示为 例 35.23 “使用模型组进行复杂类型” 中定义的复杂类型生成的 Java 类。

**例 35.25. 使用组键入**

```
@XmlType(name = "reservation", propOrder = {
    "name",
    "clubNum",
    "seatPref",
    "origin",
    "destination",
    "fltNum"
})
public class Reservation {

    @XmlElement(required = true)
    protected String name;
    protected long clubNum;
    @XmlElement(required = true)
    protected List<String> seatPref;
    @XmlElement(required = true)
    protected String origin;
    @XmlElement(required = true)
    protected String destination;
    protected long fltNum;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }
```

```
public long getClubNum() {
    return clubNum;
}

public void setClubNum(long value) {
    this.clubNum = value;
}

public List<String> getSeatPref() {
    if (seatPref == null) {
        seatPref = new ArrayList<String>();
    }
    return this.seatPref;
}

public String getOrigin() {
    return origin;
}

public void setOrigin(String value) {
    this.origin = value;
}

public String getDestination() {
    return destination;
}

public void setDestination(String value) {
    this.destination = value;
}

public long getFltNum() {
    return fltNum;
}

public void setFltNum(long value) {
    this.fltNum = value;
}
```

**多次发生**

**您可以通过将 group 元素的 maxOccurs 属性设置为大于 1 的值来指定模型组显示多次。要允许多次出现 Apache CXF 模型组 Apache CXF 将模型组映射到 List<T> 对象。List<T > 对象按照组第一个子级的规则生成：**

- **如果组是使用 序列 元素定义的，** 请参阅 **第 35.5.5 节 “序列发生的限制”。**

- 如果组是使用 选择 元素定义的，请参阅 第 35.5.3 节 “选择元素上的发生限制”。

# 第 36 章 使用 WILD 卡类型

**摘要**

当模式作者希望将元素或属性延迟到定义的类型时，会有一些实例。对于这些情况，XML Schema 提供三种机制来指定通配符位置持有者。它们都以保留其 XML 架构功能的方式映射到 Java。

## 36.1. 使用任何元素

### 概述

XML 架构 任何 元素用于在复杂类型定义中创建通配符位置拥有者。当 XML 元素实例化 XML 元素时，它可以是任何有效的 XML 元素。任何 元素不会对内容或实例化 XML 元素的名称施加任何限制。

例如，如果 例 36.1 “使用任何元素定义的 XML 架构类型” 中定义的复杂类型，您可以实例化 例 36.2 “带有任何元素的 XML 文档” 中显示的任一 XML 元素。

**例 36.1. 使用任何元素定义的 XML 架构类型**

```
<element name="FlyBoy">
  <complexType>
    <sequence>
      <any />
      <element name="rank" type="xsd:int" />
    </sequence>
  </complexType>
</element>
```

**例 36.2. 带有任何元素的 XML 文档**

```
<FlyBoy>
  <learJet>CL-215</learJet>
  <rank>2</rank>
</element>
<FlyBoy>
  <viper>Mark II</viper>
  <rank>1</rank>
</element>
```

XML 架构 任何 元素映射到 Java 对象对象或 Java org.w3c.dom.Element 对象。

### 在 XML 架构中指定

在定义序列复杂类型和选择复杂类型时，可以使用任何 元素。在大多数情况下，任何 元素都是空元素。但是，它可以取一个 annotation 元素作为子元素。

表 36.1 “XML 架构任何元素的属性” 描述 任何 元素的属性。

**表 36.1. XML 架构任何元素的属性**

| 属性 | 描述 |
| --- | --- |
| **namespace** | 指定可用于实例化 XML 文档中的元素元素的命名空间。有效值为：<br>**RENTT**<br>指定可以使用任何命名空间中的元素。这是默认值。<br>**##other**<br>指定可以使用 parent 元素命名空间以外的任何命名空间中 的元素。<br>**##local**<br>必须使用没有命名空间的元素。<br>**##targetNamespace**<br>指定必须使用 parent 元素命名空间中的元素。<br>**一个以空格分隔的 URI 列表：\#local 和 \#\#targetNamespace**<br>指定可以使用任何列出的命名空间中的元素。 |
| **maxOccurs** | 指定元素实例在 parent 元素中的最大次数。默认值为 **1**。要指定元素的实例可能会显示无限次数，您可以将属性的值设置为 **未绑定**。 |
| **minOccurs** | 指定元素实例在 parent 元素中可以包括的最小次数。默认值为 **1**。 |
| **processContents** | 指定用于实例化任何元素的元素应如何进行验证。有效值为：<br>**strict**<br>指定必须针对正确的模式验证元素。这是默认值。<br>**lax**<br>指定该元素应根据正确的架构进行验证。如果无法验证，则不会抛出错误。<br>**skip**<br>指定不应验证元素。 |

例 36.3 “使用任何元素定义的复杂类型” 显示 使用任意 元素定义的复杂类型

**例 36.3. 使用任何元素定义的复杂类型**

```
<complexType name="surprisePackage">
  <sequence>
    <any processContents="lax" />
    <element name="to" type="xsd:string" />
    <element name="from" type="xsd:string" />
  </sequence>
</complexType>
```

**映射到 Java**

XML 架构 任何 元素会导致创建名为 任何 的 Java 属性。属性具有关联的 getter 和 setter 方法。生成的属性的类型取决于元素的 processContents 属性的值。如果将 任何 元素的 processContents 属性设置为 skip，则元素将映射到 org.w3c.dom.Element 对象。对于任何元素的 processContents 属性的所有其他值，任何 元素映射到 Java 对象对象。

生成的属性使用 @XmlAnyElement 注释进行解码。此注解有一个可选的 lax 属性，它指示在处理数据时运行时要做什么。其默认值为 false，它指示运行时自动将数据放入 org.w3c.dom.Element 对象。将 lax 设置为 true 会指示运行时尝试将数据放入 JAXB 类型。当任何 元素的 processContents 属性设置为 skip 时，lax 属性将设为默认值。对于 processContents 属性的所有其他值，lax 被设置为 true。

例 36.4 “带有任何元素的 Java 类” 显示 例 36.3 “使用任何元素定义的复杂类型” 中定义的复杂类型如何映射到 Java 类。

**例 36.4. 带有任何元素的 Java 类**

```
public class SurprisePackage {

    @XmlAnyElement(lax = true) protected Object any;
    @XmlElement(required = true)
    protected String to;
    @XmlElement(required = true)
    protected String from;

    public Object getAny() { return any; }

    public void setAny(Object value) { this.any = value; }

    public String getTo() {
        return to;
    }
```

```
    public void setTo(String value) {
        this.to = value;
    }

    public String getFrom() {
        return from;
    }

    public void setFrom(String value) {
        this.from = value;
    }

}
```

### Marshalling

如果任何 元素的 Java 属性将其 lax 设为 false，或者未指定属性，则运行时不会尝试将 XML 数据解析到 JAXB 对象中。数据始终存储在 DOM Element 对象中。

如果任何 元素的 Java 属性将其 lax 设为 true，则运行时会尝试将 XML 数据放入适当的 JAXB 对象中。运行时会尝试按照以下流程识别正确的 JAXB 类：

1. 它根据运行时已知的元素列表检查 XML 元素的元素标签。如果找到匹配项，则运行时会将 XML 数据放入该元素的正确 JAXB 类中。

2. 它检查 XML 元素的 xsi:type 属性。如果找到匹配项，则运行时会将 XML 元素放入该类型的正确的 JAXB 类中。

3. 如果找不到匹配，则将 XML 数据放入 DOM Element 对象中。

通常，应用程序的运行时知道从架构在其合同中包含的所有类型。这包括合同的 wsdl:types 元素中定义的类型，通过包含添加到合同的任何数据类型，以及通过导入其他模式添加到合同中的任何类型的。您还可以使用 @XmlSeeAlso 注释，使运行时了解额外的类型，该注释在 第 32.4 节 “将类添加到 Runtime Marshaller” 中进行了描述。

### unmarshalling

如果任何 元素的 Java 属性将其 lax 设为 false，或者未指定属性，则运行时将仅接受 DOM Element 对象。尝试使用任何其他类型的对象将导致 marshalling 错误。

如果任何 元素的 Java 属性将其 lax 设为 true，则运行时会使用其 Java 数据类型和 XML 架构之间的内部映射，以确定要写入有线的 XML 结构。如果运行时知道类并将其映射到 XML 架构结构，它会写出数据并插入 xsi:type 属性来识别元素包含的数据类型。

如果运行时无法将 Java 对象映射到已知的 XML 架构结构，它将抛出 marshaling 异常。您可以使用 @XmlSeeAlso 注释，将类型添加到运行时的映射中，该注释在 第 32.4 节 “将类添加到 Runtime Marshaller” 中描述。

## 36.2. 使用 XML SCHEMA ANYTYPE TYPE

### 概述

XML Schema 类型 xsd:anyType 是所有 XML 架构类型的根类型。所有原语都是此类型的衍生性，与所有用户定义复杂类型一样。因此，定义为 xsd:anyType 的元素可以包含任何 XML 架构原语形式的数据，以及架构文档中定义的任何复杂类型。

在 Java 中，最接近的匹配 类型是 对象类。它是所有其他 Java 类是子输入的类。

### 在 XML 架构中使用

您可以使用 xsd:anyType 类型，作为任何其他 XML 架构复杂类型。它可用作元素的 type 元素 的值。它还可用作定义其他类型的基本类型。

例 36.5 “带有 Wild Card Element 的复杂类型” 显示了一个复杂类型的示例，它包含一个类型为 xsd:anyType 的元素。

**例 36.5. 带有 Wild Card Element 的复杂类型**

```
<complexType name="wildStar">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="ship" type="xsd:anyType" />
  </sequence>
</complexType>
```

### 映射到 Java

xsd:anyType 类型的元素映射到 Object 对象。例 36.6 “Java 代表 Wild Card Element” 显示 例 36.5 “带有 Wild Card Element 的复杂类型” 到 Java 类的映射。

**例 36.6. Java 代表 Wild Card Element**

```
public class WildStar {

    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true) protected Object ship;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public Object getShip() { return ship; }

    public void setShip(Object value) { this.ship = value; }
}
```

此映射允许您将任何数据放在代表通配符元素的属性中。Apache CXF 运行时处理数据的 marshaling 和 unmarshaling，以可用的 Java 表示。

### Marshalling

当 Apache CXF XML 数据放入 Java 类型时，它会尝试 将任何Type 元素放入已知的 JAXB 对象中。要确定是否可以将 anyType 元素放入一个 JAXB 生成的对象中，运行时会检查元素的 xsi:type 属性，以确定在元素中构建数据的实际类型。如果没有 xsi:type 属性，则运行时会尝试通过内省来识别元素的实际数据类型。如果元素的实际数据类型确定是应用的 JAXB 上下文已知的类型之一，则元素将放入正确类型的 JAXB 对象中。

如果运行时无法确定元素的实际数据类型，或者元素的实际数据类型不是已知类型，则运行时会将内容放入 org.w3c.dom.Element 对象中。然后，您需要使用 DOM APIs 处理元素的内容。

应用的运行时通常知道其合同中包含的模式内的所有类型。这包括合同的 wsdl:types 元素中定义的类型，通过包含添加到合同的任何数据类型，以及通过导入其他模式文档添加到合同中的任何类型的。您还可以使用 @XmlSeeAlso 注释，使运行时了解额外的类型，该注释在 第 32.4 节 “将类添加到 Runtime Marshaller” 中进行了描述。

### unmarshalling

当 Apache CXF unmarshals Java 类型进入 XML 数据时，它使用 Java 数据类型和所代表的 XML 架构之间的内部映射，以确定要写入有线的 XML 结构。如果运行时知道类，并且可以将类映射到 XML 架构结构，它会写出数据并插入 xsi:type 属性来识别元素包含的数据类型。如果数据存储在 org.w3c.dom.Element 对象中，则运行时会写入对象代表的 XML 结构，但它不包括 xsi:type 属性。

如果运行时无法将 Java 对象映射到已知的 XML 架构结构，它会抛出 marshaling 异常。您可以使用 @XmlSeeAlso 注释，将类型添加到运行时的映射中，该注释在 第 32.4 节 “将类添加到 Runtime Marshaller” 中描述。

## 36.3. 使用 UNBOUND 属性

### 概述

XML Schema 具有一种机制，允许您在复杂的类型定义中保留任意属性的位置拥有者。使用这个机制，您可以定义可以具有任何属性的复杂类型。例如，您可以创建一个类型来定义元素 <robot name="epsilon" />、<robot age="10000" /> 或 <robot type="weevil" /> 而不指定三个属性。当数据需要灵活性时，这特别有用。

### 在 XML 架构中定义

Undeclared 属性在 XML Schema 中定义，使用 anyAttribute 元素。它可在什么地方使用 attribute 元素。anyAttribute 元素没有属性，如 例 36.7 “带有 Undeclared Attribute 的复杂类型” 所示。

**例 36.7. 带有 Undeclared Attribute 的复杂类型**

```
<complexType name="arbitter">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="rate" type="xsd:float" />
  </sequence>
  <anyAttribute />
</complexType>
```

定义的类型 arbitter 有两个元素，可以具有任何类型的一个属性。例 36.8 “使用 Wild Card 属性定义的元素示例” 中显示的元素可以从复杂类型 arbitter 生成。

**例 36.8. 使用 Wild Card 属性定义的元素示例**

```
<officer rank="12"><name>...</name><rate>...</rate></officer>
<lawyer type="divorce"><name>...</name><rate>...</rate></lawyer>
<judge><name>...</name><rate>...</rate></judge>
```

### 映射到 Java

当包含 anyAttribute 元素的复杂类型映射到 Java 时，代码生成器会将名为 otherAttributes 的成员添加到生成的类。otherAttributes 是类型为 java.util.Map<QName, String >，它具有返回映射实时实例的 getter 方法。由于从 getter 返回的映射是实时的，因此对映射的任何修改都会被自动应用。例 36.9 “class for a Complex Type with a Undeclared Attribute” 显示为 例 36.7 “带有 Undeclared Attribute 的复杂类型” 中定义的复杂类型生成的类。

**例 36.9. class for a Complex Type with a Undeclared Attribute**

```
public class Arbitter {

    @XmlElement(required = true)
    protected String name;
    protected float rate;

    @XmlAnyAttribute private Map<QName, String> otherAttributes = new HashMap<QName,
String>();

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public float getRate() {
        return rate;
    }

    public void setRate(float value) {
        this.rate = value;
    }

    public Map<QName, String> getOtherAttributes() { return otherAttributes; }

}
```

### 使用未决定的属性

生成的类的 otherAttributes 成员预期使用 Map 对象填充。该映射使用 QNames 键。获取映射后，您可以访问对象上设置的任何属性，并在对象上设置新属性。

例 36.10 “使用 Undeclared 属性” 显示使用未拒绝属性的示例代码。

**例 36.10. 使用 Undeclared 属性**

```
Arbitter judge = new Arbitter();
Map<QName, String> otherAtts = judge.getOtherAttributes();

QName at1 = new QName("test.apache.org", "house");
QName at2 = new QName("test.apache.org", "veteran");

otherAtts.put(at1, "Cape");
otherAtts.put(at2, "false");

String vetStatus = otherAtts.get(at2);
```

例 36.10 “使用 Undeclared 属性” 中的代码执行以下操作：

获取包含未拒绝属性的映射。

创建 QNames 以使用属性。

将属性的值设置为映射中。

检索其中一个属性的值。

# 第 37 章 元素替换

## 摘要

XML 架构替换组允许您定义可替换顶级或 head 元素的一组元素。当您有多个共享一个通用基本类型或需要交换的元素时，这非常有用。

## 37.1. 在 XML SCHEMA 中替换组

### 概述

替换组是 XML 模式的一种功能，允许您指定可替换从该模式生成的文档中的另一个元素的元素。replaceable 元素称为 head 元素，必须在模式的全局范围内定义。替换组的元素必须与 head 元素或从 head 元素类型派生的类型相同。

本质上，通过替换组，您可以构建可以使用通用元素指定的元素集合。例如，如果您要为销售三种类型的小部件的公司构建排序系统，您可以定义一个通用小部件元素，其中包含所有三种小部件类型的一组通用数据。然后，您可以定义一个替换组，其中包含每种小部件类型更具体的数据集。然后，您可以将通用小部件元素指定为消息部分，而不是为每种小部件定义特定的排序操作。当构建实际消息时，消息可以包含替换组的任何元素。

### 语法

使用 XML Schema 元素的 replacementGroup 属性来定义替换组。replacement Group 属性的值是要定义的元素替换的元素的名称。例如，如果您的 head 元素是 widget，将属性 replaceGroup="widget" 添加到名为 woodWidget 的元素中，指定使用 widget 元素的任何元素，您可以替换 woodWidget 元素。这在 例 37.1 “使用替换组” 中显示。

**例 37.1. 使用替换组**

```
<element name="widget" type="xsd:string" />
<element name="woodWidget" type="xsd:string"
        substitutionGroup="widget" />
```

### 类型限制

替换组的元素必须与 head 元素或从 head 元素类型派生的类型相同。例如，如果 head 元素类型为 xsd:int，则替换组的所有成员都必须是 xsd:int 类型，或者派生自 xsd:int 的类型。您还可以定义一个与 例 37.2 “使用复杂类型替换组” 中显示的替换组，其中替换组的元素是从 head 元素类型派生的类型。

**例 37.2. 使用复杂类型替换组**

```
<complexType name="widgetType">
  <sequence>
    <element name="shape" type="xsd:string" />
    <element name="color" type="xsd:string" />
  </sequence>
</complexType>
<complexType name="woodWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="woodType" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="plasticWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="moldProcess" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="widget" type="widgetType" />
<element name="woodWidget" type="woodWidgetType"
         substitutionGroup="widget" />
<element name="plasticWidget" type="plasticWidgetType"
         substitutionGroup="widget" />
<complexType name="partType">
  <sequence>
    <element ref="widget" />
  </sequence>
</complexType>
<element name="part" type="partType" />
```

**替换组的 head 元素定义为 widget Type 类型。替换组的每个元素都扩展 widgetType，使其包含特定于排序该小部件类型的数据。**

**根据 例 37.2 “使用复杂类型替换组” 中的 schema，例 37.3 “使用 Substitution Group 的 XML 文档” 中的部分元素有效。**

**例 37.3. 使用 Substitution Group 的 XML 文档**

```
<part>
  <widget>
    <shape>round</shape>
    <color>blue</color>
```

```
  </widget>
</part>
<part>
  <plasticWidget>
    <shape>round</shape>
    <color>blue</color>
    <moldProcess>sandCast</moldProcess>
  </plasticWidget>
</part>
<part>
  <woodWidget>
    <shape>round</shape>
    <color>blue</color>
    <woodType>elm</woodType>
  </woodWidget>
</part>
```

**抽象头元素**

**您可以定义一个抽象头元素，它永远不会出现在使用您的 schema 生成的文档中。抽象头元素与 Java 中的抽象类类似，因为它们被用作定义更具体的通用类实现的基础。抽象头还可防止在最终产品中使用通用元素。**

**您可以通过将元素元素的 abstract 属性设置为 true 来声明抽象头 元素，如 例 37.4 “抽象头定义” 所示。使用这个模式，有效的 review 元素可以包含 正注释 元素 或负注释 元素，但不能包含 comment 元素。**

**例 37.4. 抽象头定义**

```
<element name="comment" type="xsd:string" abstract="true" />
<element name="positiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="negtiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="review">
  <complexContent>
    <all>
      <element name="custName" type="xsd:string" />
      <element name="impression" ref="comment" />
    </all>
  </complexContent>
</element>
```

## 37.2. 在 JAVA 中替换组

**概述**

根据 JAXB 规范中指定的 Apache CXF，支持使用 Java 的原生类层次结构替换组，以及 JAXBElement 类对通配符定义的支持的功能。由于替换组的成员必须全部共享一个通用基本类型，因此为支持元素的类型而生成的类也共享一个通用的基本类型。此外，Apache CXF 将 head 元素的实例映射到 JAXBElement<? 扩展 T> 属性。

### 生成的对象工厂方法

生成对象工厂来支持包含替换组的软件包有替换组中的每个元素的方法。对于替换组的每个成员，但 head 元素除外，@XmlElementDecl 注解会减少对象工厂方法包括两个额外的属性，如 表 37.1 “Declaring a JAXB Element 的属性是 Substitution 组成员” 所述。

**表 37.1. Declaring a JAXB Element 的属性是 Substitution 组成员**

| 属性 | 描述 |
| --- | --- |
| **substitutionHeadNamespace** | 指定定义 head 元素的命名空间。 |
| **substitutionHeadName** | 指定 head 元素的 **name** 属性的值。 |

替换组的 @XmlElementDecl 的 head 元素的对象工厂方法仅包含 default namespace 属性和默认 name 属性。

除了元素实例化方法外，对象工厂还包含一个代表 head 元素的实例化对象的方法。如果替换组的成员都是复杂的类型，对象工厂还包含使用每种复杂类型的实例化实例的方法。

例 37.5 “Substitution 组的对象工厂方法” 显示 例 37.2 “使用复杂类型替换组” 中定义的替换组的对象工厂方法。

**例 37.5. Substitution 组的对象工厂方法**

```
public class ObjectFactory {

    private final static QName _Widget_QNAME = new QName(...);
    private final static QName _PlasticWidget_QNAME = new QName(...);
    private final static QName _WoodWidget_QNAME = new QName(...);

    public ObjectFactory() {
    }

    public WidgetType createWidgetType() {
        return new WidgetType();
    }

    public PlasticWidgetType createPlasticWidgetType() {
```

```
        return new PlasticWidgetType();
    }

    public WoodWidgetType createWoodWidgetType() {
        return new WoodWidgetType();
    }

    @XmlElementDecl(namespace="...", name = "widget")
    public JAXBElement<WidgetType> createWidget(WidgetType value) {
        return new JAXBElement<WidgetType>(_Widget_QNAME, WidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "plasticWidget", substitutionHeadNamespace =
"...", substitutionHeadName = "widget")
    public JAXBElement<PlasticWidgetType> createPlasticWidget(PlasticWidgetType value) {
        return new JAXBElement<PlasticWidgetType>(_PlasticWidget_QNAME,
PlasticWidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "woodWidget", substitutionHeadNamespace =
"...", substitutionHeadName = "widget")
    public JAXBElement<WoodWidgetType> createWoodWidget(WoodWidgetType value) {
        return new JAXBElement<WoodWidgetType>(_WoodWidget_QNAME,
WoodWidgetType.class, null, value);
    }

}
```

### 在接口中替换组

如果将替换组的 head 元素用作操作消息中的一个消息部分，则生成的 method 参数将是为支持该元素生成的类对象。它不一定是 JAXBElement<? extends T& gt; 类的实例。运行时依赖于 Java 的原生类型层次结构来支持类型替换，Java 将捕获任何尝试使用不支持的类型。

为确保运行时知道支持元素替换所需的所有类，而 SEI 被认为是 @XmlSeeAlso 注释。此注解指定运行时为 marshalling 所需的类列表。有关使用 @XmlSeeAlso 注释的更多信息，请参阅 第 32.4 节 “将类添加到 Runtime Marshaller”。

例 37.7 “使用替换组生成接口” 显示为 例 37.6 “使用替换组的 WSDL 接口” 中显示的接口生成的 SEI。接口使用 例 37.2 “使用复杂类型替换组” 中定义的替换组。

**例 37.6. 使用替换组的 WSDL 接口**

```
<message name="widgetMessage">
    <part name="widgetPart" element="xsd1:widget" />
  </message>
  <message name="numWidgets">
    <part name="numInventory" type="xsd:int" />
```

```
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int" />
  </message>
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order" />
      <output message="tns:widgetOrderBill" name="bill" />
      <fault message="tns:badSize" name="sizeFault" />
    </operation>
    <operation name="checkWidgets">
      <input message="tns:widgetMessage" name="request" />
      <output message="tns:numWidgets" name="response" />
    </operation>
  </portType>
```

**例 37.7. 使用替换组生成接口**

```
@WebService(targetNamespace = "...", name = "orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget", targetNamespace = "...")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );
}
```

**例 37.7 “使用替换组生成接口” 中显示的 SEI 列出了 @XmlSeeAlso 注释中的对象工厂。列出命名空间的对象工厂可以访问该命名空间的所有生成的类。**

**在复杂类型中替换组**

**当替换组的 head 元素用作复杂类型中的一个元素时，代码生成器会将元素映射到 JAXBElement<? extends T> 属性。它不会将它映射到包含生成的类实例的属性，以支持替换组。**

**例如： 例 37.8 “使用替换组进行复杂类型” 中定义的复杂类型会导致 Java 类在 例 37.9 “使用 Substitution Group 的复杂类型的 Java 类” 中显示。复杂类型使用 例 37.2 “使用复杂类型替换组” 中定义的替换组。**

**例 37.8. 使用替换组进行复杂类型**

```
<complexType name="widgetOrderInfo">
```

```
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element ref="xsd1:widget"/>
  </sequence>
</complexType>
```

**例 37.9. 使用 Substitution Group 的复杂类型的 Java 类**

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "widgetOrderInfo", propOrder = {"amount","widget",})
public class WidgetOrderInfo {

    protected int amount;
    @XmlElementRef(name = "widget", namespace = "...", type = JAXBElement.class) protected
JAXBElement<? extends WidgetType> widget;
    public int getAmount() {
        return amount;
    }

    public void setAmount(int value) {
        this.amount = value;
    }

    public JAXBElement<? extends WidgetType> getWidget() { return widget; }

    public void setWidget(JAXBElement<? extends WidgetType> value) { this.widget =
((JAXBElement<? extends WidgetType> ) value); }

}
```

设置替换组属性

您如何使用替换组，这取决于将组映射到直接 Java 类还是一个 JAXBElement<? extends T> 类的代码生成器。当元素直接映射到生成的值类的对象时，您可以像使用属于类型层次结构的其他 Java 对象一样使用对象。您可以将任何子类替换为父类。您可以检查对象以确定其确切的类，并相应地广播它。

JAXB 规格建议您使用对象工厂方法来实例化所生成类的对象。

当代码生成器创建 JAXBElement<? 扩展 T > 对象来保存替换组的实例时，您必须将元素的值嵌套在 JAXBElement<? extends T> 对象中。执行此操作的最佳方法是使用对象工厂提供的元素创建方法。它们提供了一种简单的方法，可以基于其值创建元素。

例 37.10 “设置替换组成员” 显示设置替换组实例的代码。

**例 37.10. 设置替换组成员**

```
ObjectFactory of = new ObjectFactory();
PlasticWidgetType pWidget = of.createPlasticWidgetType();
pWidget.setShape = "round';
pWidget.setColor = "green";
pWidget.setMoldProcess = "injection";

JAXBElement<PlasticWidgetType> widget = of.createPlasticWidget(pWidget);

WidgetOrderInfo order = of.createWidgetOrderInfo();
order.setWidget(widget);
```

例 37.10 “设置替换组成员” 中的代码执行以下操作：

实例化对象工厂。

实例化 PlasticWidgetType 对象。

实例化 JAXBElement<PlasticWidgetType& gt; 对象来容纳 plastic widget 元素。

实例化 WidgetOrderInfo 对象。

将 WidgetOrderInfo 对象的小部件设置为含有 plastic widget 元素的 JAXBElement 对象。

获取替换组属性的值

从 JAXBElement<? extends T > 对象提取元素值时，对象工厂方法无法帮助。您必须使用 JAXBElement<? 扩展 T> 对象的 getValue （） 方法。以下选项决定了 getValue （） 方法返回的对象类型：

- 使用所有可能类的 isInstance （） 方法来确定元素的值对象的类。
- 使用 JAXBElement<? extends T > 对象的 getName （） 方法来确定元素的名称。

  getName （） 方法返回一个 QName。使用元素的本地名称，您可以为 value 对象确定正确

的类。

- 使用 JAXBElement<? extends T > 对象的 getDeclaredType （） 方法来确定 value 对象的类。

  getDeclaredType （） 方法返回元素值对象的 Class 对象。

警告

getDeclaredType （） 方法有可能返回 head 元素的基本类，而不考虑 value 对象的实际类。

例 37.11 “获取 Substitution 组成员的值” 显示从替换组检索值的代码。要确定元素值对象的正确类，示例使用元素的 getName （） 方法。

**例 37.11. 获取 Substitution 组成员的值**

```
String elementName = order.getWidget().getName().getLocalPart();
if (elementName.equals("woodWidget")
{
  WoodWidgetType widget=order.getWidget().getValue();
}
else if (elementName.equals("plasticWidget")
{
  PlasticWidgetType widget=order.getWidget().getValue();
}
else
{
  WidgetType widget=order.getWidget().getValue();
}
```

### 37.3. 小部件供应商示例

#### 37.3.1. 小部件排序接口

本节展示了 Apache CXF 中用于解决实际应用程序的替换组示例。服务和消费者使用 例 37.2 “使用复杂类型替换组” 中定义的小部件替换组开发。该服务提供了两个操作： checkWidgets 和

**placeWidgetOrder。例 37.12 “小部件排序接口” 显示排序服务的接口。**

**例 37.12. 小部件排序接口**

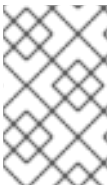
```
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation"
        type="xsd1:widgetOrderBillInfo"/>
</message>
<message name="widgetMessage">
  <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

**例 37.13 “小部件排序 SEI” 显示为接口生成的 Java SEI。**

**例 37.13. 小部件排序 SEI**

```
@WebService(targetNamespace = "http://widgetVendor.com/widgetOrderForm", name =
"orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget", targetNamespace =
"http://widgetVendor.com/types/widgetTypes")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "widgetOrderConformation", targetNamespace = "", partName =
"widgetOrderConformation")
    @WebMethod
    public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo placeWidgetOrder(
```

```
        @WebParam(partName = "widgetOrderForm", name = "widgetOrderForm",
targetNamespace = "")
        com.widgetvendor.types.widgettypes.WidgetOrderInfo widgetOrderForm
    ) throws BadSize;
}
```



**注意**

**因为该示例仅显示使用替换组，因此不会显示一些业务逻辑。**

### 37.3.2. checkWidgets Operation

**概述**

**checkWidgets 是一个简单操作，它有一个参数，它是替换组的头成员。此操作演示了如何处理作为替换组成员的单个参数。消费者必须确保该参数是替换组的有效成员。该服务必须正确确定在请求中发送了替换组的成员。**

**消费者实施**

**生成的方法签名使用 Java 类，支持替换组的 head 元素的类型。由于替换组的 member 元素与 head 元素或从 head 元素类型派生的类型相同，因此生成的 Java 类支持从生成的 Java 类继承的 Java 类，以支持为支持 head 元素而继承的 Java 类。Java 类型层次结构原生支持使用子类代替父类。**

**由于 Apache CXF 如何为替换组和 Java 类型层次结构生成类型，客户端可以在不使用任何特殊代码的情况下调用 checkWidgets （）。在开发调用 checkWidgets （） 的逻辑时，您可以传递一个生成的类的对象，以支持小部件替换组。**

**例 37.14 “consumer Invoking checkWidgets （）”** 显示调用 **checkWidgets （） 的使用者。**

**例 37.14. consumer Invoking checkWidgets （）**

```
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
```

```
  case '1':
  {
    WidgetType widget = new WidgetType();
    ...
    break;
  }
  case '2':
  {
    WoodWidgetType widget = new WoodWidgetType();
    ...
    break;
  }
  case '3':
  {
    PlasticWidgetType widget = new PlasticWidgetType();
    ...
    break;
  }
  default :
    System.out.println("Invaid Widget Selection!!");
}

proxy.checkWidgets(widgets);
```

服务实现

服务 checkWidgets （） 的实现获得小部件描述，作为 WidgetType 对象，检查小部件清单，并返回库存中的小部件数量。由于用于实施替换组从同一基础类继承的所有类，因此您可以实施 checkWidgets （），而无需使用任何特定于 JAXB 的 API。

生成的所有类都支持为 widget 的替换组成员扩展 WidgetType 类。由于此事实，您可以使用 instanceof 来确定传递的小部件类型，并只需将 widgetPart 对象转换为更严格的类型（如果适用）。正确类型的对象后，您可以检查右侧小部件类型的清单。

例 37.15 “checkWidgets （）的服务实现” 显示了可能的实现。

**例 37.15. checkWidgets （）的服务实现**

```
public int checkWidgets(WidgetType widgetPart)
{
  if (widgetPart instanceof WidgetType)
  {
    return checkWidgetInventory(widgetType);
  }
  else if (widgetPart instanceof WoodWidgetType)
  {
    WoodWidgetType widget = (WoodWidgetType)widgetPart;
    return checkWoodWidgetInventory(widget);
```

```
  }
  else if (widgetPart instanceof PlasticWidgetType)
  {
    PlasticWidgetType widget = (PlasticWidgetType)widgetPart;
    return checkPlasticWidgetInventory(widget);
  }
}
```

### 37.3.3. placeWidgetOrder Operation

#### 概述

**placeWidgetOrder 使用两个包含替换组的复杂类型。此操作演示了如何在 Java 实施中使用这样的结构。消费者和服务必须获取和设置替换组的成员。**

#### 消费者实施

**要调用 placeWidgetOrder （），消费者必须构建包含 widget 替换组的一个元素的小部件顺序。将小部件添加到顺序时，消费者应使用为替换组的每个元素生成的对象工厂方法。这样可确保运行时和服务可以正确处理顺序。例如，如果为 plastic 小部件放置了一个顺序，则 ObjectFactory.createPlasticWidget （） 方法用于在将其添加到顺序之前创建元素。**

**例 37.16 “设置替换组成员” 显示用于设置 WidgetOrderInfo 对象的 widget 属性的消费者代码。**

**例 37.16. 设置替换组成员**

```
ObjectFactory of = new ObjectFactory();

WidgetOrderInfo order = new of.createWidgetOrderInfo();
...
System.out.println();
System.out.println("What color widgets do you want to order?");
String color = reader.readLine();
System.out.println();
System.out.println("What shape widgets do you want to order?");
String shape = reader.readLine();
System.out.println();
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
```

```
    case '1':
    {
      WidgetType widget = of.createWidgetType();
      widget.setColor(color);
      widget.setShape(shape);
      JAXB<WidgetType> widgetElement = of.createWidget(widget);
  order.setWidget(widgetElement);
      break;
    }
    case '2':
    {
      WoodWidgetType woodWidget = of.createWoodWidgetType();
      woodWidget.setColor(color);
      woodWidget.setShape(shape);
      System.out.println();
      System.out.println("What type of wood are your widgets?");
      String wood = reader.readLine();
      woodWidget.setWoodType(wood);
      JAXB<WoodWidgetType> widgetElement = of.createWoodWidget(woodWidget);
  order.setWoodWidget(widgetElement);
      break;
    }
    case '3':
    {
      PlasticWidgetType plasticWidget = of.createPlasticWidgetType();
      plasticWidget.setColor(color);
      plasticWidget.setShape(shape);
      System.out.println();
      System.out.println("What type of mold to use for your
                        widgets?");
      String mold = reader.readLine();
      plasticWidget.setMoldProcess(mold);
      JAXB<WidgetType> widgetElement = of.createPlasticWidget(plasticWidget);
  order.setPlasticWidget(widgetElement);
      break;
    }
    default :
      System.out.println("Invaid Widget Selection!!");
      }
```

服务实现

**placeWidgetOrder （） 方法以 WidgetOrderInfo 对象的形式收到一个顺序，处理顺序，并以 WidgetOrderBillInfo 对象的形式向消费者返回 bill。顺序可以是普通小部件、plastic widget 或 wooden 小部件。排序的小部件类型由将哪些类型的对象存储在 widget OrderForm 对象的小部件属性中。widget 属性是一个替换组，可以包含 widget 元素、woodWidget 元素或 plasticWidget 元素。**

该实施必须确定按顺序存储哪些元素。这可以通过 **JAXBElement<? extends T>** 对象的 **getName （） 方法来实现，以确定元素的 QName。然后，可以使用 QName 来确定替换组中的哪个元素是按顺序的。当已知包含的元素后，您可以将其值提取到正确的对象类型。**

**例 37.17 “实现 placeWidgetOrder （）”显示了可能的实现。**

**例 37.17. 实现 placeWidgetOrder （）**

```
public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo
placeWidgetOrder(WidgetOrderInfo widgetOrderForm)
{
  ObjectFactory of = new ObjectFactory();

  WidgetOrderBillInfo bill = new WidgetOrderBillInfo()

   // Copy the shipping address and the number of widgets
   // ordered from widgetOrderForm to bill
   ...

  int numOrdered = widgetOrderForm.getAmount();

  String elementName = widgetOrderForm.getWidget().getName().getLocalPart();
  if (elementName.equals("woodWidget")
  {
    WoodWidgetType widget=order.getWidget().getValue();
    buildWoodWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<WoodWidgetType> widgetElement = of.createWoodWidget(widget);
    bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.75;
    bill.setAmountDue(amtDue);
  }
  else if (elementName.equals("plasticWidget")
  {
    PlasticWidgetType widget=order.getWidget().getValue();
    buildPlasticWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<PlasticWidgetType> widgetElement = of.createPlasticWidget(widget);
    bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.90;
    bill.setAmountDue(amtDue);
  }
  else
  {
    WidgetType widget=order.getWidget().getValue();
    buildWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<WidgetType> widgetElement = of.createWidget(widget);
    bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.30;
    bill.setAmountDue(amtDue);
  }
```

```
    return(bill);
  }
```

例 37.17 “实现 placeWidgetOrder （）” 中的代码执行以下操作：

实例化对象工厂以创建元素。

实例化 WidgetOrderBillInfo 对象来容纳 bill。

获取排序的小部件数量。

获取以顺序存储元素的本地名称。

检查该元素是否为 woodWidget 元素。

将元素的值从顺序提取到正确类型的对象。

创建一个位于 bill 中的 JAXBElement<T > 对象。

设置 bill 对象 的小部件 属性。

设置 bill 对象的 amountDue 属性。

# 第 38 章 自定义如何生成类型

**摘要**

默认的 JAXB 映射解决了使用 XML 架构定义 Java 应用的对象时遇到的大部分情况。对于默认映射不足的实例，JAXB 提供了广泛的自定义机制。

## 38.1. 自定义类型映射的基础知识

### 概述

JAXB 规范定义了多个 XML 元素，它们自定义 Java 类型映射到 XML 架构结构的方式。这些元素可以在线使用 XML 架构结构指定。如果无法或不想修改 XML 架构定义，您可以在外部绑定文档中指定自定义。

### Namespace

用于自定义 JAXB 数据绑定的元素在命名空间 http://java.sun.com/xml/ns/jaxb 中定义。您必须添加与 例 38.1 “JAXB 自定义命名空间” 中显示的命名空间声明。这添加到定义 JAXB 自定义的所有 XML 文档的根元素中。

**例 38.1. JAXB 自定义命名空间**

```
xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
```

### 版本声明

在使用 JAXB 自定义时，您必须指明正在使用的 JAXB 版本。这可以通过在外部绑定声明的根元素中添加 jaxb:version 属性来实现。如果使用 in-line 自定义，则必须在包含自定义的 schema 元素中包含 jaxb:version 属性。属性的值始终为 2.0。

例 38.2 “指定 JAXB 自定义版本” 显示 schema 元素中使用的 jaxb:version 属性示例。

**例 38.2. 指定 JAXB 自定义版本**

```
< schema ...
      jaxb:version="2.0">
```

使用在线自定义

自定义代码生成器如何将 XML 架构结构映射到 Java 构造的最直接方法是将自定义元素直接添加到 XML 架构定义中。JAXB 自定义元素放在被修改的 XML 架构结构的 xsd:appinfo 元素中。

例 38.3 “自定义 XML 架构” 显示包含在线 JAXB 定制的架构示例。

**例 38.3. 自定义 XML 架构**

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <complexType name="size">
    <annotation> <appinfo> <jaxb:class name="widgetSize" /> </appinfo> </annotation>
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

使用外部绑定声明

如果您无法或不想更改定义您的类型的 XML 架构文档，您可以使用外部绑定声明指定自定义。外部绑定声明由多个嵌套的 jaxb:bindings 元素组成。例 38.4 “JAXB External Binding Declaration Syntax” 显示外部绑定声明的语法。

**例 38.4. JAXB External Binding Declaration Syntax**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings [schemaLocation="schemaUrl" | wsdlLocation="wsdlUrl">
    <jaxb:bindings node="nodeXPath">
      binding declaration
    </jaxb:bindings>
    ...
  </jaxb:bindings>
<jaxb:bindings>
```

schemaLocation 属性和 wsdlLocation 属性用于识别修改要应用到的 schema 文档。如果您要从 schema 文档生成代码，请使用 schemaLocation 属性。如果您要从 WSDL 文档生成代码，请使用 wsdlLocation 属性。

node 属性用于识别要修改的特定 XML 模式结构。它是解析为 XML Schema 元素的 XPath 语句。

根据 例 38.5 “XML 架构文件” 中显示的架构文档 widgetSchema.xsd，例 38.6 “外部绑定声明” 中显示的外部绑定声明会修改复杂类型 大小的 生成。

**例 38.5. XML 架构文件**

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        version="1.0">
  <complexType name="size">
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

**例 38.6. 外部绑定声明**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="wsdlSchema.xsd">
    <jaxb:bindings node="xsd:complexType[@name='size']">
       <jaxb:class name="widgetSize" />
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

要指示代码生成器使用外部 binging 声明，请使用 wsdl2java 工具的 -b *binding-file* 选项，如下所示：

```
wsdl2java -b widgetBinding.xml widget.wsdl
```

## 38.2. 指定 XML 架构 PRIMITIVE 的 JAVA 类

### 概述

默认情况下，XML 架构类型映射到 Java 原语类型。虽然这是 XML 架构和 Java 之间的最逻辑映射，但它并不总是满足应用程序开发人员的要求。您可能希望将 XML 架构原语类型映射到可保存额外信息的

Java 类，或者您可能希望将 XML 原语类型映射到允许简单类型替换的类。

JAXB javaType 自定义元素允许您自定义 XML 架构原语类型和 Java 原语类型之间的映射。它可用于自定义全局级别和单个实例级别的映射。您可以使用 javaType 元素作为简单类型定义的一部分，或作为复杂类型定义的一部分。

在使用 javaType 自定义元素时，您必须指定将 primitive 类型的 XML 表示转换为目标 Java 类的方法。有些映射有默认的转换方法。对于没有默认映射的实例，Apache CXF 提供了 JAXB 方法来简化所需方法的开发。

语法

javaType 自定义元素采用四个属性，如 表 38.1 “为 XML 架构类型自定义 Java 类的 Generation 的属性” 所述。

表 38.1. 为 XML 架构类型自定义 Java 类的 Generation 的属性

| 属性 | 必填 | 描述 |
| --- | --- | --- |
| **name** | 是 | 指定 XML Schema 原语类型映射到的 Java 类的名称。它必须是有效的 Java 类名称或 Java 原语类型的名称。您必须确保此类存在，并可以被应用程序访问。代码生成器不会检查此类。 |
| **xmlType** | 否 | 指定正在自定义的 XML 架构原语类型。只有在将 **javaType** 元素用作 **globalBindings** 元素的子时，才会使用此属性。 |
| **parseMethod** | 否 | 指定负责将数据解析到 Java 类实例中基于字符串的方法。如需更多信息，请参阅 “指定转换器”一节。 |
| **printMethod** | 否 | 指定负责将 Java 对象转换为基于字符串的 XML 表示数据的方法。如需更多信息，请参阅 “指定转换器”一节。 |

javaType 自定义元素可通过三种方式使用：

- 要修改 XML Schema 原语类型的所有实例 - javaType 元素在用作 globalBindings 自定义

元素的子时修改 schema 文档中的 XML Schema 类型的所有实例。以这种方式使用时，您必须为 xmlType 属性指定一个值，用于标识正在修改的 XML Schema 原语类型。

例 38.7 “全局原语类型自定义” 显示一个 in-line 全局自定义，它指示代码生成器对架构中的所有 xsd:short 的所有实例使用 java.lang.Integer。

**例 38.7. 全局原语类型自定义**

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings ...>
        <jaxb:javaType name="java.lang.Integer"
                       xmlType="xsd:short" />
      </globalBindings
    </appinfo>
  </annotation>
  ...
</schema>
```

- 要修改简单的类型定义 - javaType 元素修改在应用到名为 simple 类型定义时为 XML 简单类型的所有实例生成的类。当使用 javaType 元素修改简单类型定义时，请不要使用 xmlType 属性。

例 38.8 “用于自定义简单类型的绑定文件” 显示一个外部绑定文件，它修改名为 zipCode 的简单类型生成。

**例 38.8. 用于自定义简单类型的绑定文件**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings wsdlLocation="widgets.wsdl">
    <jaxb:bindings node="xsd:simpleType[@name='zipCode']">
      <jaxb:javaType name="com.widgetVendor.widgetTypes.zipCodeType"
                     parseMethod="com.widgetVendor.widgetTypes.support.parseZipCode"
                     printMethod="com.widgetVendor.widgetTypes.support.printZipCode" />
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

- 要修改复杂类型定义的元素或属性 - javaType 可以应用于复杂类型定义的单个部分，方法是

将它作为 JAXB 属性自定义的一部分包含在内。javaType 元素作为子项放在属性的 baseType 元素中。当使用 javaType 元素修改复杂类型定义的特定部分时，请不要使用 xmlType 属性。

例 38.9 “用于在 Complex Type 中自定义元素的绑定文件” 显示修改复杂类型的元素的绑定文件。

**例 38.9. 用于在 Complex Type 中自定义元素的绑定文件**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='cost']">
        <jaxb:property>
          <jaxb:baseType>
            <jaxb:javaType name="com.widgetVendor.widgetTypes.costType"
                           parseMethod="parseCost"
                           printMethod="printCost" >
          </jaxb:baseType>
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

有关使用 baseType 元素的更多信息，请参阅 第 38.6 节 “指定元素或属性的基本类型”。

### 指定转换器

Apache CXF 无法将 XML 架构原语类型转换为随机 Java 类。当您使用 javaType 元素自定义 XML Schema 原语类型的映射时，代码生成器会创建一个适配器类，用于 marshal 和 unmarshal 自定义 XML 架构原语类型。例 38.10 “JAXB Adapter 类” 中显示示例适配器类。

**例 38.10. JAXB Adapter 类**

```
public class Adapter1 extends XmlAdapter<String, javaType>
{
  public javaType unmarshal(String value)
  {
    return(parseMethod(value));
  }

  public String marshal(javaType value)
  {
```

```
    return(printMethod(value));
  }
}
```

*parseMethod* 和 *printMethod* 被对应的 parseMethod 属性和 printMethod 属性的值替代。值必须识别有效的 Java 方法。您可以通过以下两种方式之一指定方法的名称：

- 完全限定的 Java 方法名称，格式为 *packagename.ClassName.methodName*
- *methodName* 格式的简单方法名称

  当您只提供简单方法名称时，代码生成器假定方法存在于由 javaType 元素的 name 属性指定的类中。

重要

代码生成器 不会生成 解析或打印方法。您负责提供它们。有关开发解析和打印方法的详情，请参考 “实施转换器”一节。

如果没有提供 parseMethod 属性的值，则代码生成器假定 name 属性指定的 Java 类具有构造器，其第一个参数是一个 Java String 对象。生成的适配器的 unmarshal （） 方法使用假定的构造器使用 XML 数据填充 Java 对象。

如果没有提供 printMethod 属性的值，则代码生成器假定 name 属性指定的 Java 类具有 toString （） 方法。生成的适配器的 marshal （） 方法使用假定的 toString （） 方法将 Java 对象转换为 XML 数据。

如果 javaType 元素的 name 属性指定 Java 原语类型，或者 Java 原语的打包程序类型之一，则代码生成器使用默认的转换器。有关默认转换器的详情请参考 “默认原语类型转换器”一节。

生成内容

如 “指定转换器”一节 所述，使用 javaType customization 元素为每个自定义 XML 架构原语类型触发一个适配器类的生成。适配器按顺序使用 pattern Adapter*N* 命名。如果您指定了两个原语类型自定义，则代码生成器会创建两个适配器类： Adapter1 和 Adapter2。

为 XML 模式构建生成的代码取决于效果的 XML 架构结构是全局定义的元素，或者被定义为复杂类型的一部分。

当 XML Schema 结构是全局定义的元素时，为类型生成的对象工厂方法将从默认方法修改，如下所示：

- 该方法使用 @XmlJavaTypeAdapter 注释进行解码。

  该注解指示在处理此元素实例时要使用的适配器类的运行时。适配器类指定为 class 对象。

- 默认 type 被 javaType 元素的 name 属性指定的类替代。

例 38.11 “全局元素的自定义对象工厂方法” 显示受 例 38.7 “全局原语类型自定义” 中显示的自定义影响的元素的对象工厂方法。

**例 38.11. 全局元素的自定义对象工厂方法**

```
@XmlElementDecl(namespace = "http://widgetVendor.com/types/widgetTypes", name = "shorty")
@XmlJavaTypeAdapter(org.w3._2001.xmlschema.Adapter1.class)
public JAXBElement<Integer> createShorty(Integer value) {
    return new JAXBElement<Integer>(_Shorty_QNAME, Integer.class, null, value);
}
```

当将 XML Schema 结构定义为复杂类型的一部分时，生成的 Java 属性将修改，如下所示：

- 属性使用 @XmlJavaTypeAdapter 注释进行解码。

  该注解指示在处理此元素实例时要使用的适配器类的运行时。适配器类指定为 class 对象。

- 属性的 @XmlElement 包含 type 属性。

  type 属性的值是代表生成对象的默认基本类型的类对象。对于 XML 架构原语类型，则类是 String。

- 属性使用 @XmlSchemaType 注释进行解码。

  该注解标识了结构的 XML 架构原语类型。

- 默认 type 被 javaType 元素的 name 属性指定的类替代。

例 38.12 “自定义复杂类型” 显示受 例 38.7 “全局原语类型自定义” 中显示的自定义影响的元素的对象工厂方法。

**例 38.12. 自定义复杂类型**

```
public class NumInventory {

    @XmlElement(required = true, type = String.class) @XmlJavaTypeAdapter(Adapter1.class)
@XmlSchemaType(name = "short") protected Integer numLeft;
    @XmlElement(required = true)
    protected String size;

    public Integer getNumLeft() {
        return numLeft;
    }

    public void setNumLeft(Integer value) {
        this.numLeft = value;
    }

    public String getSize() {
        return size;
    }

    public void setSize(String value) {
        this.size = value;
    }

}
```

### 实施转换器

Apache CXF 运行时不知道如何将 XML 原语类型转换为 javaType 元素指定的 Java 类，但它应调用 parseMethod 属性和 printMethod 属性指定的方法。您负责提供方法运行时调用的实现。实施的方法必须能够使用 XML 原语类型的字典结构。

为简化数据转换方法的实现，Apache CXF 提供 javax.xml.bind.DatatypeConverter 类。此类提供解析和打印所有 XML 架构原语类型的方法。parse 方法使用 XML 数据的字符串表示，它们会返回 表 34.1

"XML Schema 原语类型到 Java 原生类型映射" 中定义的默认类型实例。打印方法采用默认类型的实例，它们返回 XML 数据的字符串表示。

DatatypeConverter 类的 Java 文档可在 https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/DatatypeConverter.html 中找到。

默认原语类型转换器

在 javaType 元素的 name 属性中指定 Java 原语类型或 Java 原语类型 Wrapper 类时，不需要为 parseMethod 属性或 printMethod 属性指定值。如果没有提供任何值，Apache CXF 运行时会替换默认的转换器。

默认数据转换器使用 JAXB DatatypeConverter 类来解析 XML 数据。默认转换器还会提供进行转换所需的任何类型广播。

## 38.3. 为简单类型生成 JAVA 类

概述

默认情况下，命名的简单类型不会生成类型，除非它们被枚举。使用简单类型定义的元素映射到 Java 原语类型的属性。

当您需要生成简单类型到 Java 类中时，例如，您需要使用类型替换时。

要指示代码生成器为所有全局定义的简单类型生成类，请将 globalBindings Custom 元素的 mapSimpleTypeDef 设置为 true。

添加自定义

要指示代码生成器为简单类型创建 Java 类，请添加 globalBinding 元素的 mapSimpleTypeDef 属性，并将其值设为 true。

例 38.13 "用于为 SimpleTypes 强制生成 Java 类的 in-Line 自定义" 显示一行自定义，用于强制代码生成器为命名简单类型生成 Java 类。

例 38.13. 用于为 SimpleTypes 强制生成 Java 类的 in-Line 自定义

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings mapSimpleTypeDef="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

例 38.14 “绑定文件以强制生成 Constants” 显示自定义简单类型的外部绑定文件。

**例 38.14. 绑定文件以强制生成 Constants**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings mapSimpleTypeDef="true" />
  <jaxb:bindings>
<jaxb:bindings>
```

**重要**

**此自定义仅影响 全局范围 内定义 的命名 简单类型。**

**生成的类**

为简单类型生成的类具有一个名为 值 的属性。value 属性是 第 34.1 节 “原语类型” 中映射定义的 Java 类型。生成的类具有 getter 和 value 属性的 setter。

例 38.16 “简单类型的自定义映射” 显示为 例 38.15 “自定义映射的简单类型” 中定义的简单类型生成的 Java 类。

**例 38.15. 自定义映射的简单类型**

```
<simpleType name="simpleton">
  <restriction base="xsd:string">
    <maxLength value="10"/>
```

```
  </restriction>
</simpleType>
```

**例 38.16. 简单类型的自定义映射**

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "simpleton", propOrder = {"value"})
public class Simpleton {

    @XmlValue
    protected String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

}
```

## 38.4. 自定义枚举映射

**概述**

**如果您希望基于 xsd:string 以外的模式类型枚举类型，则必须指示代码生成器进行映射。您还可以控制生成的枚举常数的名称。**

**自定义是使用 jaxb:typesafeEnumClass 元素以及一个或多个 jaxb:typesafeEnumMember 元素完成的。**

**有些情况下，代码生成器的默认设置无法为枚举的所有成员创建有效的 Java 标识符。您可以使用 globalBindings 自定义属性自定义代码生成器如何处理它。**

**成员名称自定义器**

**如果代码生成器在生成枚举成员时遇到命名冲突，或者如果无法为枚举的成员创建有效的 Java 标识符，则代码生成器默认生成一个警告，且不会为枚举器生成 Java 枚举 类型。**

您可以通过添加 globalBinding 元素的 typesafeEnumMemberName 属性来更改此行为。typesafeEnumMemberName 属性的值在 表 38.2 “自定义枚举成员名称生成的值” 中描述。

表 38.2. 自定义枚举成员名称生成的值

| value | 描述 |
| --- | --- |
| **skipGeneration**（默认） | 指定没有生成 Java **enum** 类型并生成警告。 |
| **generateName** | 指定按照 **VALUE_*N*** 格式生成成员名称。*n* 从一个开始，并为每个枚举的成员递增。 |
| **generateError** | 指定代码生成器无法将枚举映射到 Java **enum** 类型时生成错误。 |

例 38.17 “自定义以强制类型安全成员名称” 显示一行自定义，用于强制代码生成器生成类型安全成员名称。

例 38.17. 自定义以强制类型安全成员名称

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings typesafeEnumMemberName="generateName" />
    </appinfo>
  </annotation>
  ...
</schema>
```

类自定义器

jaxb:typesafeEnumClass 元素指定 XML Schema enumeration 应该映射到 Java enum 类型。它有两个属性，如 表 38.3 “自定义生成的枚举类的属性” 中所述。当 jaxb:typesafeEnumClass 项被指定时，它必须放在修改简单类型的 xsd:annotation 元素中。

表 38.3. 自定义生成的枚举类的属性

| 属性 | 描述 |
| --- | --- |

| 属性 | 描述 |
| --- | --- |
| **name** | 指定生成的 Java **enum** 类型的名称。这个值必须是有效的 Java 标识符。 |
| **map** | 指定枚举是否应映射到 Java **enum** 类型。默认值为 **true**。 |

成员自定义器

**jaxb:typesafeEnumMember 元素指定 XML Schema enumeration facet 和 Java enum 类型常量之间的映射。您必须在被自定义的枚举中为每个 枚举 的 facet 使用一个 jaxb:typesafeEnumMember 元素。**

**使用在线自定义时，可以通过以下两种方式之一使用此元素：**

- **它可以放在 枚举 的 xsd:annotation 元素中，该元素正在修改。**
- **它们都可以作为 jaxb:typesafeEnumClass 元素的子项放在用于自定义枚举的 jaxb:typesafeEnumClass 元素。**

**jaxb:typesafeEnumMember 元素具有所需的 name 属性。name 属性指定生成的 Java enum 类型常数的名称。它的值必须是有效的 Java 标识符。**

**jaxb:typesafeEnumMember 元素也有一个 value 属性。该值 用于将 枚举 与正确的 jaxb:typesafeEnumMember 元素关联。value 属性的值必须与 枚举 的 facets' 值之一匹配。当使用外部绑定规格来自定义类型时，或将 jaxb:typesafeEnumMember 元素分组为 jaxb:typesafeEnumClass 元素的子项时，需要此属性。**

例子

**例 38.18 “Enumerated Type 的在线自定义” 显示一个枚举类型，它使用在线自定义，并单独自定义枚举的成员。**

**例 38.18. Enumerated Type 的在线自定义**

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```
      xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
      jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass />
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="one" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="2">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="two" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="3">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="three" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="4">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="four" />
          </appinfo>
        </annotation>
      </enumeration>
    </restriction>
  </simpleType>
<schema>
```

例 38.19 “使用组合映射的 Enumerated Type 的直接自定义” 显示使用在线自定义的枚举类型，并在类自定义中合并成员自定义。

**例 38.19. 使用组合映射的 Enumerated Type 的直接自定义**

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
      jaxb:version="2.0">
  <simpleType name="widgetInteger">
```

```
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass>
          <jaxb:typesafeEnumMember value="1" name="one" />
          <jaxb:typesafeEnumMember value="2" name="two" />
          <jaxb:typesafeEnumMember value="3" name="three" />
          <jaxb:typesafeEnumMember value="4" name="four" />
        </jaxb:typesafeEnumClass>
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1" />
      <enumeration value="2" />
      <enumeration value="3" />
      <enumeration value="4" >
    </restriction>
  </simpleType>
<schema>
```

**例 38.20 “用于自定义枚举的绑定文件”** 显示自定义枚举类型的外部绑定文件。

**例 38.20. 用于自定义枚举的绑定文件**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:simpleType[@name='widgetInteger']">
      <jaxb:typesafeEnumClass>
        <jaxb:typesafeEnumMember value="1" name="one" />
        <jaxb:typesafeEnumMember value="2" name="two" />
        <jaxb:typesafeEnumMember value="3" name="three" />
        <jaxb:typesafeEnumMember value="4" name="four" />
      </jaxb:typesafeEnumClass>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

## 38.5. 自定义修复的值属性映射

### 概述

默认情况下，代码生成器映射属性定义为具有固定值到普通属性。使用模式验证时，Apache CXF 可以强制使用模式定义（请参阅 第 24.3.4.7 节 “模式验证类型值”）。但是，使用模式验证会增加消息处理

时间。

将固定值的属性映射到 Java 的另一种方法是将它们映射到 Java 常数。您可以使用 globalBindings 自定义元素指示代码生成器将固定值属性映射到 Java 常数。您还可以使用 property 元素自定义固定值属性到 Java 常量的映射。

全局自定义

您可以通过添加 globalBinding 元素的 fixedAttributeAsConstantProperty 属性来更改此行为。将此属性设置为 true 会指示代码生成器将任何使用 fixed 属性定义的属性映射到 Java 常量。

例 38.21 “用于强制生成 Constants 的 in-Line 自定义” 显示一行自定义，它会强制代码生成器为带有固定值的属性生成常量。

**例 38.21. 用于强制生成 Constants 的 in-Line 自定义**

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

例 38.22 “绑定文件以强制生成 Constants” 显示自定义固定属性生成的外部绑定文件。

**例 38.22. 绑定文件以强制生成 Constants**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
  <jaxb:bindings>
<jaxb:bindings>
```

本地映射

您可以使用 property 元素的 fixedAttributeAsConstantProperty 属性根据每个属性自定义属性映射。将此属性设置为 true 会指示代码生成器将任何使用 fixed 属性定义的属性映射到 Java 常量。

例 38.23 “用于强制生成 Constants 的 in-Line 自定义” 显示一行自定义，它会强制代码生成器为带有固定值的单个属性生成常量。

**例 38.23. 用于强制生成 Constants 的 in-Line 自定义**

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <complexType name="widgetAttr">
    <sequence>
      ...
    </sequence>
    <attribute name="fixer" type="xsd:int" fixed="7">
      <annotation> <appinfo> <jaxb:property fixedAttributeAsConstantProperty="true" /> </appinfo>
</annotation>
    </attribute>
  </complexType>
  ...
</schema>
```

例 38.24 “绑定文件以强制生成 Constants” 显示自定义固定属性生成的外部绑定文件。

**例 38.24. 绑定文件以强制生成 Constants**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:bindings node="xsd:complexType[@name='widgetAttr']">
      <jaxb:bindings node="xsd:attribute[@name='fixer']">
          <jaxb:property fixedAttributeAsConstantProperty="true" />
        </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

**Java 映射**

在默认映射中，所有属性都通过 getter 和 setter 方法映射到标准 Java 属性。当此自定义应用于使用 fixed 属性定义的属性时，属性映射到 Java 常数，如 例 38.25 “将固定值属性映射到 Java Constant” 所

示。

**例 38.25. 将固定值属性映射到 Java Constant**

```
@XmlAttribute
public final static type NAME = value;
```

*类型* 是通过使用 第 34.1 节 “原语类型” 中描述的映射将属性的基本类型映射到 Java 类型来确定。

***NAME*** 通过将 attribute 元素的 name 属性的值转换为所有大写字母来确定。

*值* 由 attribute 元素 的固定 属性的值决定。

例如，例 38.23 “用于强制生成 Constants 的 in-Line 自定义” 中定义的属性映射，如 例 38.26 “修复了映射到 Java Constant 的值属性” 所示。

**例 38.26. 修复了映射到 Java Constant 的值属性**

```
@XmlRootElement(name = "widgetAttr")
public class WidgetAttr {

    ...

  @XmlAttribute
  public final static int FIXER = 7;

  ...

}
```

## 38.6. 指定元素或属性的基本类型

### 概述

有时您需要自定义为元素生成的对象类，或作为 XML 架构复杂类型的一部分定义的属性。例如，您可能希望使用更常规的对象类来允许替换简单的类型。

执行此操作的一种方法是使用 JAXB 基础类型自定义。它允许开发人员根据需要指定生成的对象类来代表元素或属性。基本类型自定义允许您指定 XML 架构结构和生成的 Java 对象之间的备用映射。这个

备用映射可以是简单的分类，也可以是默认基本类的规范化。它还可以是 XML 架构原语类型到 Java 类的映射。

自定义使用

要将 JAXB 基础 type 属性应用到 XML Schema 结构，可使用 JAXB baseType 自定义元素。baseType 自定义元素是 JAXB 属性 元素的子级，因此必须正确嵌套它。

根据您要自定义 XML Schema 结构到 Java 对象的映射的方式，您可以添加 baseType 自定义元素的 name 属性或 javaType 子元素。name 属性用于将生成的对象的默认类映射到同一类层次结构中的另一个类。当您要将 XML Schema 原语类型映射到 Java 类时，会使用 javaType 元素。

重要

您不能在同一 baseType 自定义元素中使用 name 属性和 javaType 子元素。

特殊化或常规默认映射

baseType 自定义元素的 name 属性用于将生成的对象的类重新定义到同一 Java 类层次结构中的类。属性指定 XML 架构结构映射到的 Java 类的完全限定名称。指定的 Java 类 必须是 代码生成器通常为 XML Schema 构造生成的 Java 类的超级类或子类。对于映射到 Java 原语类型的 XML 架构类型，打包程序类用作自定义目的的默认基本类。

例如，定义为 xsd:int 的元素使用 java.lang.Integer 作为其默认基本类。name 属性的值可以指定任意 Integer 的超级类，如 Number 或 Object。

对于简单类型替换，最常见的自定义是将原语类型映射到对象对象。

例 38.27 “基本类型的命令行自定义” 显示一行自定义，它将复杂类型中的一个元素映射到 Java 对象对象。

**例 38.27. 基本类型的命令行自定义**

```
<complexType name="widgetOrderInfo">
  <all>
    <element name="amount" type="xsd:int" />
     <element name="shippingAdress" type="Address">
      <annotation> <appinfo> <jaxb:property> <jaxb:baseType name="java.lang.Object" />
</jaxb:property> </appinfo> </annotation>
    </element>
```

```
    <element name="type" type="xsd:string"/>
  </all>
</complexType>
```

例 38.28 “用于自定义基本类型的外部绑定文件” 显示用于自定义的外部绑定文件，如 例 38.27 “基本类型的命令行自定义” 所示。

**例 38.28. 用于自定义基本类型的外部绑定文件**

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='shippingAddress']">
        <jaxb:property>
          <jaxb:baseType name="java.lang.Object" />
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

生成的 Java 对象的 @XmlElement 注释包含一个 type 属性。type 属性的值是代表生成对象的默认基本类型的类对象。如果是 XML Schema 原语类型，则类是相应 Java 原语类型的打包程序类。

例 38.29 “带有修改的基本类的 Java 类” 显示根据 例 38.28 “用于自定义基本类型的外部绑定文件” 中的模式定义生成的类。

**例 38.29. 带有修改的基本类的 Java 类**

```
public class WidgetOrderInfo {

    protected int amount;
    @XmlElement(required = true)
    protected String type;
    @XmlElement(required = true, type = Address.class) protected Object shippingAddress;

    ...
    public Object getShippingAddress() {
        return shippingAddress;
    }

    public void setShippingAddress(Object value) {
        this.shippingAddress = value;
```

```
    }

}
```

**使用 javaType**

**javaType 元素可用于自定义如何使用 XML 架构原语类型定义的元素和属性映射到 Java 对象。使用 javaType 元素比使用 baseType 元素的 name 属性提供更大的灵活性。javaType 元素允许您将原语类型映射到任何类对象。**

**有关使用 javaType 元素的详细描述，请参阅 第 38.2 节 “指定 XML 架构 Primitive 的 Java 类”。**

# 第 39 章 使用 A JAXBCONTEXT 对象

## 摘要

JAXBContext 对象允许 Apache CXF 运行时在 XML 元素和 Java 对象之间转换数据。应用程序开发人员需要在消息处理程序中实例化 JAXBContext 对象，并在实施使用原始 XML 消息的用户时实例化。

## 概述

JAXBContext 对象是运行时使用的低级别对象。它允许运行时在 XML 元素及其相应的 Java 表示法之间进行转换。应用程序开发人员通常不需要使用 JAXBContext 对象。XML 数据的 marshaling 和 unmarshaling 通常由 JAX-WS 应用的传输和绑定层处理。

但是，有些情况下应用程序需要直接操作 XML 消息内容。在这两个实例中：

- 第 41.1 节 “在一个 Consumer 中使用 XML”
- 第 43 章 *编写处理程序*

您将需要使用两个可用的 JAXBContext.newInstance （） 方法之一来实例化 JAXBContext 对象。

## 最佳实践

JAXBContext 对象是资源密集型，可以实例化。建议应用尽可能创建几个实例。其中一种实现方式是创建一个 JAXBContext 对象，它可以管理应用使用的所有 JAXB 对象，并尽可能在应用的多个部分之间共享。

JAXBContext 对象是线程安全。

## 使用对象工厂获取 JAXBCONTEXT 对象

JAXBContext 类提供了 newInstance （） 方法，如 例 39.1 “使用类获取 JAXB 上下文” 中所示，它

取实施 JAXB 对象的类列表。

例 39.1. 使用类获取 JAXB 上下文

```
staticJAXBContextnewInstanceClass…classesToBeBoundJAXBException
```

返回的 JAXBObject 对象将能够对通过传递给方法的类实施的 JAXB 对象进行 marshal 和 unmarshal 数据。它还能够处理从传递给方法中的任何类静态引用的任何类。

虽然可以将应用使用的每个 JAXB 类的名称传递给 newInstance （） 方法，但效率不高。完成相同目标的更有效的方法是传递为您的应用程序生成的对象工厂或对象工厂。生成的 JAXBContext 对象将能够管理指定对象工厂可以实例化的任何 JAXB 类。

使用软件包名称获取 JAXBCONTEXT 对象

JAXBContext 类 提供了一个新的Instance （） 方法，如 例 39.2 “使用类获取 JAXB 上下文” 中所示，它采用冒号(:)分隔的软件包名称列表。指定的软件包应包含从 XML Schema 派生的 JAXB 对象。

例 39.2. 使用类获取 JAXB 上下文

```
staticJAXBContextnewInstanceStringcontextPathJAXBException
```

返回的 JAXBContext 对象能够对指定软件包中的类实施的所有 JAXB 对象进行 marshal 和 unmarshal 数据。

# 第 40 章 开发同步应用程序

## 摘要

JAX-WS 提供了一种简单的机制，用于异步访问服务。SEI 可以指定可用于异步访问服务的其他方法。Apache CXF 代码生成器为您生成额外的方法。您只需添加业务逻辑。

## 40.1. 同步调用的类型

除了调用的常见同步模式外，Apache CXF 支持两种异步调用形式：

- 轮询方法 - 要使用轮询方法调用远程操作，您需要调用没有输出参数的方法，但会返回 javax.xml.ws.Response 对象。可以轮询 Response 对象（从 javax.util.concurrency.Future 接口继承），以检查响应消息是否已到达。

- 回调方法 - 要使用回调方法调用远程操作，您需要调用一个方法，该方法取对回调对象(javax.xml.ws.AsyncHandler 类型)的引用作为其参数之一。当响应消息到达客户端时，运行时调用会返回 AsyncHandler 对象，并为其提供响应消息的内容。

## 40.2. 用于异步示例的 WSDL

例 40.1 “用于同步示例的 WSDL Contract” 显示用于异步示例的 WSDL 合同。合同定义了单个接口 GreeterAsync，其中包含单个操作，greetMeSometime。

**例 40.1. 用于同步示例的 WSDL Contract**

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions
xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_async_soap_http"
                  xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  targetNamespace="http://apache.org/hello_world_async_soap_http"
                  name="HelloWorld">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_async_soap_http/types"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
          elementFormDefault="qualified">
      <element name="greetMeSometime">
        <complexType>
          <sequence>
```

```
          <element name="requestType" type="xsd:string"/>
        </sequence>
      </complexType>
    </element>
    <element name="greetMeSometimeResponse">
      <complexType>
        <sequence>
          <element name="responseType"
                   type="xsd:string"/>
        </sequence>
      </complexType>
    </element>
  </schema>
</wsdl:types>

<wsdl:message name="greetMeSometimeRequest">
  <wsdl:part name="in" element="x1:greetMeSometime"/>
</wsdl:message>
<wsdl:message name="greetMeSometimeResponse">
  <wsdl:part name="out"
             element="x1:greetMeSometimeResponse"/>
</wsdl:message>

<wsdl:portType name="GreeterAsync">
  <wsdl:operation name="greetMeSometime">
    <wsdl:input name="greetMeSometimeRequest"
                message="tns:greetMeSometimeRequest"/>
    <wsdl:output name="greetMeSometimeResponse"
                 message="tns:greetMeSometimeResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="GreeterAsync_SOAPBinding"
              type="tns:GreeterAsync">
  ...
</wsdl:binding>

<wsdl:service name="SOAPService">
  <wsdl:port name="SoapPort"
             binding="tns:GreeterAsync_SOAPBinding">
    <soap:address location="http://localhost:9000/SoapContext/SoapPort"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

### 40.3. 生成 STUB 代码

**概述**

调用异步样式需要额外的 stub 代码，用于 SEI 中定义的专用异步方法。默认不会生成这个特殊的 stub 代码。要切换异步功能并生成必需的存根代码，您必须使用 WSDL 2.0 规范中的映射自定义功能。

自定义可让您修改 Maven 代码生成插件生成 stub 代码的方式。特别是，它可让您修改 WSDL-to-Java 映射，并切换到某些功能。在这里，自定义用于切换异步调用功能。自定义通过绑定声明来指定，您可以使用 jaxws:bindings 标签（其中 jaxws 前缀与 http://java.sun.com/xml/ns/jaxws 命名空间关联）。指定绑定声明的方法有两种：

### 外部绑定声明

当使用外部绑定声明时，jaxws:bindings 元素在独立于 WSDL 合同的文件中定义。在生成 stub 代码时，您可以指定绑定声明文件的位置到代码生成器。

### 嵌入式绑定声明

当使用嵌入式绑定声明时，您直接将 jaxws:bindings 元素嵌入到 WSDL 合同中，将其视为 WSDL 扩展。在这种情况下，jaxws:bindings 中的设置仅适用于 immediate parent 元素。

### 使用外部绑定声明

例 40.2 “Asynchronous Binding Declaration 模板” 中显示切换异步调用的绑定声明文件的模板。

**例 40.2. Asynchronous Binding Declaration 模板**

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          wsdlLocation="AffectedWSDL"
          xmlns="http://java.sun.com/xml/ns/jaxws">
  <bindings node="AffectedNode">
    <enableAsyncMapping>true</enableAsyncMapping>
  </bindings>
</bindings>
```

其中 *AffectedWSDL* 指定受此绑定声明影响的 WSDL 合同的 URL。*AffectedNode* 是一个 XPath 值，用于指定来自 WSDL 合同的节点（或节点）会受到此绑定声明的影响。如果您希望整个 WSDL 合同会受到影响，您可以将 *AffectedNode* 设置为 wsdl:definitions。jaxws:enableAsyncMapping 元素设为 true，以启用异步调用功能。

例如，如果您只想为 GreeterAsync 接口生成异步方法，您可以在前面的绑定声明中指定 <bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']">。

假设绑定声明存储在文件 **async_binding.xml** 中，您可以设置 POM，如 **例 40.3 “消费者代码生成”** 所示。

**例 40.3. 消费者代码生成**

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>hello_world.wsdl</wsdl>
            <extraargs>
              <extraarg>-client</extraarg>
              <extraarg>-b async_binding.xml</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

**-b** 选项告知代码生成器在哪里可以找到外部绑定文件。

有关代码生成器的更多信息，请参阅 **第 44.2 节 “cxf-codegen-plugin”**。

### 使用嵌入式绑定声明

您还可以通过将 **jaxws:bindings** 元素及其关联的 **jaxws:enableAsynchMapping** 子部分直接嵌入到 WSDL 中，将绑定自定义直接嵌入到定义服务的 WSDL 文档中。您还必须为 **jaxws** 前缀添加命名空间声明。

**例 40.4 “带有嵌入式绑定声明的 WSDL 用于同步映射”** 显示带有嵌入式绑定声明的 WSDL 文件，该文件激活某个操作的异步映射。

**例 40.4. 带有嵌入式绑定声明的 WSDL 用于同步映射**

```
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
                  ...
                  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
                  ...>
  ...
  <wsdl:portType name="GreeterAsync">
    <wsdl:operation name="greetMeSometime">
      <jaxws:bindings> <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
</jaxws:bindings>
      <wsdl:input name="greetMeSometimeRequest"
                  message="tns:greetMeSometimeRequest"/>
      <wsdl:output name="greetMeSometimeResponse"
                   message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

将绑定声明嵌入到 WSDL 文档中时，您可以通过更改放置声明的位置来控制受声明影响的范围。当声明作为 **wsdl:definitions** 元素的子项放置时，代码生成器会为 WSDL 文档中定义的所有操作创建异步方法。如果它作为 **wsdl:portType** 元素的子级放置，则代码生成器会为接口中定义的所有操作创建异步方法。如果它作为 **wsdl:operation** 元素的子项放置，则代码生成器仅为该操作创建异步方法。

在使用嵌入式声明时，不需要将任何特殊选项传递给代码生成器。代码生成器将识别它们并相应地操作。

### 生成的接口

以这种方式生成 stub 代码后，GreeterAsync SEI （在文件 GreeterAsync.java中）定义，如 例 40.5 “带有同步调用的方法的服务端点接口” 所示。

**例 40.5. 带有同步调用的方法的服务端点接口**

```
package org.apache.hello_world_async_soap_http;

import org.apache.hello_world_async_soap_http.types.GreetMeSometimeResponse;
...

public interface GreeterAsync
{
  public Future<?> greetMeSometimeAsync(
        java.lang.String requestType,
        AsyncHandler<GreetMeSometimeResponse> asyncHandler
    );
```

```
    public Response<GreetMeSometimeResponse> greetMeSometimeAsync(
        java.lang.String requestType
    );

    public java.lang.String greetMeSometime(
        java.lang.String requestType
    );
}
```

除了常见的同步方法外，还可为 **greetMeSometime （） 生成两个异步方法：**

- **回调方法 public future<? >greetMeSomtimeAsyncjava.lang.StringrequestTypeAsyncHandler<GreetMeSomtime Response>asyncHandler**
- **轮询方法公共响应 <GreetMeSomeTimeResponse>greetMeSometimeAsyncjava.lang.StringrequestType**

### 40.4. 使用轮询方法实施同步客户端

**概述**

轮询方法是开发异步应用的两种方法更为简单。客户端调用名为 ***OperationName*Async （） 的异步方法，并返回 Response<T >** 对象来轮询响应。客户端在等待响应时执行的操作取决于应用程序的要求。处理轮询有两种基本模式：

- **非阻塞轮询- 通过调用非阻塞 Response<T>.isDone （） 方法，定期检查结果是否已就绪。如果结果就绪，客户端会处理它。如果没有，客户端将继续执行其他操作。**
- **阻塞轮询- 您立即调用 Response<T>.get （），并阻止响应到达（可选指定超时）。**

**使用非阻塞模式**

**例 40.6 “对异步操作调用的非阻塞方法” 演示使用非阻塞轮询来对 例 40.1 “用于同步示例的 WSDL Contract” 中定义的 greetMeSometime 操作进行异步调用。客户端调用异步操作，并定期检查结果是否**

返回。

**例 40.6. 对异步操作调用的非阻塞方法**

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {

    // set up the proxy for the client

    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));

      while (!greetMeSomeTimeResp.isDone()) {
      // client does some work
      }
      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response

      System.exit(0);
  }
}
```

例 40.6 “对异步操作调用的非阻塞方法” 中的代码执行以下操作：

在代理上调用 greetMeSometimeAsync （）。

方法调用会立即将 Response<GreetMeSometimeResponse > 对象返回到客户端。Apache CXF 运行时处理从远程端点接收回复的详细信息，并填充 Response<GreetMeSometimeResponse> 对象。

**注意**

运行时将请求传送到远程端点的 **greetMeSometime** （） 方法，并透明处理调用的异步性质的详细信息。端点（因此服务实现）不会担心客户端要等待响应的详细信息。

通过检查返回的 **Response** 对象的 **isDone** （） 来检查响应是否已到达。

如果响应没有到达，客户端会在再次检查前继续工作。

当响应到达时，客户端使用 **get** （） 方法从 **Response** 对象检索它。

**使用块模式**

在使用块轮询模式时，**Response** 对象的 **isDone** （） 不会被调用。相反，在调用远程操作后，**Response** 对象的 **get** （） 方法会被立即调用。**get** （） 会阻止，直到响应可用。

您还可以将超时限制传递给 **get** （） 方法。

例 40.7 “阻止轮询方法进行异步操作调用” 显示使用阻塞轮询的客户端。

**例 40.7. 阻止轮询方法进行异步操作调用**

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {
```

```
    // set up the proxy for the client

    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));
    GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
      System.exit(0);
  }
}
```

## 40.5. 使用回调方法实施同步客户端

### 概述

进行异步操作调用的另一种方法是实施回调类。然后，您将调用异步远程方法，该方法将回调对象用作参数。运行时将响应返回到回调对象。

要实现使用回调的应用程序，请执行以下操作：

1. 创建 实现 **AsyncHandler** 接口的回调类。

   **注意**

   您的回调对象可以执行应用程序所需的任何类型的响应处理。

2. 使用 ***operationName***Async （） 进行远程调用，该函数将回调对象用作参数并返回 future&lt ;?> 对象。

3. 如果您的客户端需要访问响应数据，您可以轮询返回的 future&lt ;?> 对象的 isDone （） 方法，以查看远程端点是否发送了响应。

   如果回调对象执行所有响应处理，则不需要检查响应是否已到达。

### 实现回调

回调类必须实施 javax.xml.ws.AsyncHandler 接口。接口定义了一个方法：handleResponseResponse<T> 重新调用 handleResponse （） 方法，以通知客户端响应已到达。例 40.8 “javax.xml.ws.AsyncHandler 接口” 显示您必须实现的 AsyncHandler 接口的概述。

**例 40.8. javax.xml.ws.AsyncHandler 接口**

```
public interface javax.xml.ws.AsyncHandler
{
  void handleResponse(Response<T> res)
}
```

例 40.9 “回调实施类” 显示 例 40.1 “用于同步示例的 WSDL Contract” 中定义的 greetMeSometime 操作的回调类。

**例 40.9. 回调实施类**

```
package demo.hw.client;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.types.*;

public class GreeterAsyncHandler implements AsyncHandler<GreetMeSometimeResponse>
{
  private GreetMeSometimeResponse reply;

  public void handleResponse(Response<GreetMeSometimeResponse>
                             response)
  {
    try
    {
      reply = response.get();
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
    }
  }

  public String getResponse()
  {
    return reply.getResponseType();
  }
}
```

例 40.9 “回调实施类” 中显示的回调实现执行以下操作：

定义 member 变量 响应，其中包含从远程端点返回的响应。

实施 handleResponse （）。

这种实施只是提取响应并将其分配给 member 变量 回复。

实施名为 getResponse （） 的添加方法。

此方法是一种方便的方法，可以从 回复 中提取数据并返回数据。

实现消费者

例 40.10 “异步操作调用的回调方法” 演示了使用回调方法对 例 40.1 “用于同步示例的 WSDL Contract” 中定义的 GreetMeSometime 操作进行异步调用的客户端。

**例 40.10. 异步操作调用的回调方法**

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  ...

  public static void main(String args[]) throws Exception
  {
    ...
    // Callback approach
    GreeterAsyncHandler callback = new GreeterAsyncHandler();

    Future<?> response =
      port.greetMeSometimeAsync(System.getProperty("user.name"),
                                callback);
    while (!response.isDone())
```

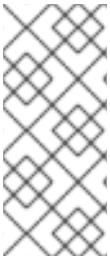
```
    {
      // Do some work
    }
    resp = callback.getResponse();
    ...
    System.exit(0);
  }
}
```

例 40.10 “异步操作调用的回调方法” 中的代码执行以下操作：

实例化回调对象。

调用 greetMeSometimeAsync （），它将使用代理上的回调对象。

这个方法调用会立即将 future <?> 对象返回到客户端。Apache CXF 运行时处理从远程端点接收回复的详情，调用回调对象的 handleResponse （） 方法，并填充 Response< GreetMeSometimeResponse > 对象。



注意

运行时将请求传送到远程端点的 greetMeSometime （） 方法，并处理调用的异步性质的详细信息，而无需远程端点的知识。端点（因此服务实施）不需要担心客户端要等待响应的详细信息。

使用返回的 Future<?> 对象的 isDone （） 方法检查响应是否到达远程端点。

调用回调对象的 getResponse （） 方法，以获取响应数据。

## 40.6. 捕获从远程服务返回的例外

### 概述

发出异步请求的消费者不会收到与同步请求时返回的异常。异步返回到消费者的任何例外都嵌套在 ExecutionException 异常中。服务抛出的实际异常存储在 ExecutionException 异常的 cause 字段中。

### 捕获异常

远程服务生成的异常是通过将响应传递给消费者业务逻辑的方法在本地引发。当消费者发出同步请求时，进行远程调用的方法会抛出异常。当消费者发出异步请求时，Response<T> 对象的 get （） 方法会抛出异常。消费者不会发现在处理请求时遇到错误，直到尝试检索响应消息为止。

与 JAX-WS 框架生成的方法不同，Response<T> 对象的 get （） 方法不会抛出用户建模的异常或通用 JAX-WS 异常。相反，它会抛出 java.util.concurrent.ExecutionException 异常。

### 获取例外详情

框架将来自远程服务返回的异常存储在 ExecutionException 异常的 cause 字段中。通过获取 cause 字段的值并检查存储的异常来提取远程异常的详细信息。存储的异常可以是任何用户定义的异常，也可以是通用 JAX-WS 异常之一。

### Example

例 40.11 “使用轮询方法捕获例外” 演示了使用轮询方法捕获异常的示例。

**例 40.11. 使用轮询方法捕获例外**

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client
{
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception
  {
    ...
    // port is a previously established proxy object.
    Response<GreetMeSometimeResponse> resp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));

    while (!resp.isDone())
```

```
    {
      // client does some work
    }

    try
    {
      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
    }
    catch (ExecutionException ee)
    {
       Throwable cause = ee.getCause();
       System.out.println("Exception "+cause.getClass().getName()+" thrown by the remote
service.");
    }
  }
}
```

**例 40.11 “使用轮询方法捕获例外” 中的代码执行以下操作：**

**在 try/catch 块中嵌套对 Response<T> 对象的 get （） 方法的调用。**

**捕获 ExecutionException 异常。**

**从异常中提取 cause 字段。**

**如果消费者使用回调方法，用于捕获异常的代码将放置在提取服务响应的回调对象中。**

# 第 41 章 使用 RAW XML 消息

## 摘要

高级别 JAX-WS API 通过将数据放入 JAXB 对象来阻止开发人员使用原生 XML 消息。然而，在有些情况下，最好直接访问在线上传递的原始 XML 消息数据。JAX-WS API 提供了两个提供对原始 XML 的访问的接口： Dispatch 接口是客户端的接口，provider 接口是服务器端接口。

## 41.1. 在一个 CONSUMER 中使用 XML

### 摘要

Dispatch 接口是一个低级 JAX-WS API，允许您直接使用原始消息。它接受和返回多种类型的消息或有效负载，包括 DOM 对象、SOAP 消息和 JAXB 对象。由于它是一个低级 API，因此 Dispatch 接口不会执行更高级别的 JAX-WS API 执行的任何消息准备。您必须确保传递给 Dispatch 对象的消息或有效负载已被正确构建，并对被调用的远程操作有意义。

### 41.1.1. 使用模式

#### 概述

分配对象有两种 *使用模式* ：

- 消息 模式
- 消息 Payload 模式(Payload 模式)

为 Dispatch 对象指定的使用模式决定了传递给用户级别代码的详细信息数量。

#### 消息模式

*在消息模式* 中，Dispatch 对象可用于完整的消息。完整的消息包括任何绑定特定的标头和打包程序。例如，与需要 SOAP 消息的服务交互的消费者必须提供 Dispatch 对象的 invoke （） 方法完全指定的 SOAP 消息。invoke （） 方法还会返回完全指定的 SOAP 消息。使用者代码负责完成和读取 SOAP 消息的标头和 SOAP 消息的信封信息。

使用 JAXB 对象时，消息模式不是理想的选择。

要指定 Dispatch 对象使用消息模式在创建 Dispatch 对象时提供 java.xml.ws.Service.Mode.MESSAGE 的值。有关创建 Dispatch 对象的更多信息，请参阅 “创建 Dispatch 对象”一节。

有效负载模式

在 *有效负载模式* 中，也称为消息有效负载模式，Dispatch 对象只能用于消息的有效负载。例如，在有效负载模式下工作的 Dispatch 对象只适用于 SOAP 消息的正文。绑定层处理任何绑定级别打包程序和标头。当从 invoke （） 方法返回结果时，绑定级别的打包程序和标头已经被条带化，只有消息正文会被保留。

当使用不使用特殊打包程序的绑定时，如 Apache CXF XML 绑定、有效负载模式和消息模式会提供相同的结果。

要指定 Dispatch 对象使用有效负载模式，请在创建 Dispatch 对象时提供 java.xml.ws.Service.Mode.PAYLOAD 值。有关创建 Dispatch 对象的更多信息，请参阅 “创建 Dispatch 对象”一节。

### 41.1.2. 数据类型

概述

由于 Dispatch 对象是低级对象，因此它们不会优化，以使用与更高级别的消费者 API 相同的 JAXB 生成的类型。分配对象用于以下类型的对象：

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource
- “使用 JAXB 对象”一节

使用 Source 对象

Dispatch 对象接受并返回从 javax.xml.transform.Source 接口派生的对象。源对象受到任何绑定支

持，在消息模式或有效负载模式中支持。

源对象是保存 XML 文档的低级别对象。每个源实施都提供了访问存储的 XML 文档的方法，然后操作其内容。以下对象实现 Source 接口：

DOMSource

包含 XML 消息作为文档对象模型(DOM)树。XML 消息作为一组使用 get Node （） 方法访问的节点对象保存。节点可以使用 setNode （） 方法更新或添加到 DOM 树中。

SAXSource

包含 XML 消息作为 XML (SAX)对象的简单 API。SAX 对象包含一个 InputSource 对象，其中包含原始数据和 XMLReader 对象来解析原始数据。

StreamSource

包含 XML 消息作为数据流。数据流可以像任何其他数据流一样操作。

如果您创建 Dispatch 对象使其使用通用 Source 对象，则 Apache CXF 会将消息返回为 SAXSource 对象。

可以使用端点的 source-preferred-format 属性来更改此行为。有关配置 Apache CXF 运行时的详情，请参考 第 IV 部分 “配置 Web 服务端点”。

使用 SOAPMessage 对象

如果满足以下条件，则分配对象可以使用 javax.xml.soap.SOAPMessage 对象：

- Dispatch 对象使用 SOAP 绑定
- Dispatch 对象使用消息模式

SOAPMessage 对象包含 SOAP 消息。它们包含一个 SOAPPart 对象和零个或多个 AttachmentPart 对象。SOAPPart 对象包含 SOAP 消息的特定部分，包括 SOAP 信封、任何 SOAP 标头和 SOAP 消息正文。AttachmentPart 对象包含作为附件传递的二进制数据。

使用 DataSource 对象

分配对象可在以下条件满足时使用实现 javax.activation.DataSource 接口的对象：

- Dispatch 对象使用 HTTP 绑定
- Dispatch 对象使用消息模式

数据源对象提供了一种使用来自各种源的 MIME 类型数据的机制，包括 URL、文件和字节数组。

使用 JAXB 对象

虽然 Dispatch 对象旨在低级 API，允许您处理原始消息，它们也允许您使用 JAXB 对象。若要使用 JAXB 对象，必须传递一个 JAXBContext，该对象可以使用 marshal 和 unmarshal。当创建 Dispatch 对象时，会传递 JAXBContext。

您可以将 JAXBContext 对象理解的任何 JAXB 对象作为参数传递给 invoke （） 方法。您还可以将返回的消息定向到 JAXBContext 对象理解的任何 JAXB 对象。

有关创建 JAXBContext 对象的详情，请参考 第 39 章 *使用 A JAXBContext 对象*。

### 41.1.3. 使用 Dispatch 对象

流程

要使用 Dispatch 对象调用远程服务，应遵循以下序列：

1. 创建 Dispatch 对象。
2. 构建 请求消息。
3. 调用正确的 invoke （） 方法。

4. 解析响应消息。

### 创建 Dispatch 对象

要创建 Dispatch 对象，请执行以下操作：

1. 创建一个 Service 对象来代表 wsdl:service 元素，该元素定义 Dispatch 对象将发出调用的服务。请参阅 第 25.2 节 “创建服务对象”。

2. 使用 Service 对象的 createDispatch （） 方法创建 Dispatch 对象，如 例 41.1 “createDispatch （） 方法” 所示。

**例 41.1. createDispatch （） 方法**

```
PublicDispatch<T>createDispatchQNameportNamejava.lang.Class<T&gt;typeService.
ModemodeWebServiceException
```

**注意**

如果您使用 JAXB 对象，则 createDispatch （） 的方法签名为：public Dispatch<T>createDispatchQNameportNamejavax.xml.bind.JAXBContextcontextService.ModemodeWebServiceException

表 41.1 “createDispatch （） 的参数” 描述 createDispatch （） 方法的参数。

**表 41.1. createDispatch （） 的参数**

| 参数 | 描述 |
| --- | --- |
| **portName** | 指定代表服务供应商的 **wsdl:port** 元素的 QName，其中 Dispatch 对象将发出调用。 |
| **type** | 指定 Dispatch 对象使用的对象类型。请参阅 第 41.1.2 节 “数据类型”。使用 JAXB 对象时，此参数指定用于 marshal 和 unmarshal 的 **JAXBContext** 对象。 |

| 参数 | 描述 |
| --- | --- |
| **模式** | 指定 Dispatch 对象的使用模式。请参阅 第 41.1.1 节 “使用模式”。 |

例 41.2 “创建 Dispatch 对象” 显示在有效负载模式下创建与 DOMSource 对象配合使用的 Dispatch 对象的代码。

**例 41.2. 创建 Dispatch 对象**

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    QName serviceName = new QName("http://org.apache.cxf", "stockQuoteReporter");
    Service s = Service.create(serviceName);

    QName portName = new QName("http://org.apache.cxf", "stockQuoteReporterPort");
    Dispatch<DOMSource> dispatch = s.createDispatch(portName,
                                       DOMSource.class,
                                       Service.Mode.PAYLOAD);
    ...
```

**构建请求消息**

使用 Dispatch 对象时，必须从头开始构建请求。开发人员负责确保传递到 Dispatch 对象的消息与目标服务提供商可以处理的请求匹配。这需要精确了解服务提供商使用的消息及其所需标头信息（如果有的话）。

此信息可以通过 WSDL 文档或定义消息的 XML 架构文档提供。虽然服务提供商有很大变化，但需要遵循几个准则：

- 请求的 root 元素基于与正在调用的操作对应的 wsdl:operation 元素的 name 属性的值。

警告

如果被调用的服务使用 doc/literal 裸机消息，则请求的 root 元素将基于 wsdl:part 元素引用的 name 属性的值。

- 请求的根元素是命名空间限定项。
- 如果被调用的服务使用 rpc/literal 信息，则请求中的顶级元素将不合格命名空间。

重要

顶级元素的子项可以是命名空间限定性。要确定，您必须检查其架构定义。

- 如果被调用的服务使用 rpc/literal 信息，则任何顶级元素都不为空。
- 如果被调用的服务使用 doc/literal 消息，消息的 schema 定义将确定是否有任何元素是否合格。

有关服务如何使用 XML 消息的更多信息，请参阅 WS-I 基本配置文件。

同步调用

对于生成响应的同步调用的用户，请使用 例 41.3 “Dispatch.invoke （） 方法” 中显示的 Dispatch 对象的 invoke （） 方法。

例 41.3. Dispatch.invoke （） 方法

```
TinvokeTmsgWebServiceException
```

在创建 Dispatch 对象时，决定传递给 invoke （） 方法的响应类型和请求的类型。例如，如果您使

用 createDispatch (portName、SOAMessage.class、Service.Mode.MESSAGE) 创建 Dispatch 对象，则响应和请求都是 SOAPMessage 对象。

注意

使用 JAXB 对象时，响应和请求可以是提供的 JAXBContext 对象可以 marshal 和 unmarshal 的任何类型。此外，响应和请求也可以是不同的 JAXB 对象。

例 41.4 “使用 Dispatch 对象进行同步调用” 显示使用 DOMSource 对象在远程服务上进行同步调用的代码。

例 41.4. 使用 Dispatch 对象进行同步调用

```
// Creating a DOMSource Object for the request
DocumentBuilder db = DocumentBuilderFactory.newDocumentBuilder();
Document requestDoc = db.newDocument();
Element root = requestDoc.createElementNS("http://org.apache.cxf/stockExample",
                                           "getStockPrice");
root.setNodeValue("DOW");
DOMSource request = new DOMSource(requestDoc);

// Dispatch disp created previously
DOMSource response = disp.invoke(request);
```

异步调用

分配对象也支持异步调用。与 第 40 章 *开发同步应用程序* 中讨论的异步 API 一样，Dispatch 对象可以使用轮询方法和回调方法。

使用轮询方法时，invokeAsync （） 方法会返回 Response<t& gt; 对象，它可以轮询来查看响应是否已到达。例 41.5 “轮询的 Dispatch.invokeAsync （） 方法” 显示使用轮询方法进行异步调用的方法签名。

例 41.5. 轮询的 Dispatch.invokeAsync （） 方法

响应 <T>invokeAsyncTmsgWebServiceException

有关使用轮询方法进行异步调用的详细信息，请参阅 第 40.4 节 “使用轮询方法实施同步客户端”。

在使用回调方法时，invokeAsync （） 方法采用 AsyncHandler 实现，它会在返回时处理响应。例 41.6 “使用回调的 Dispatch.invokeAsync （） 方法” 显示使用回调方法进行异步调用的方法签名。

**例 41.6. 使用回调的 Dispatch.invokeAsync （） 方法**

```
future<?&gt;invokeAsyncTmsgAsyncHandler<T&gt;handlerWebServiceException
```

有关使用回调方法进行异步调用的详情，请参考 第 40.5 节 “使用回调方法实施同步客户端”。

**注意**

与同步 调用 （） 方法一样，在创建 Dispatch 对象时，响应的类型和请求的类型会被决定。

单向调用

当请求没有生成响应时，请使用 Dispatch 对象的 invokeOneWay （） 进行远程调用。例 41.7 “Dispatch.invokeOneWay （） 方法” 显示此方法的签名。

**例 41.7. Dispatch.invokeOneWay （） 方法**

```
invokeOneWayTmsgWebServiceException
```

用于在创建 Dispatch 对象时确定用于打包请求的对象类型。例如，如果使用 createDispatch (portName, DOMSource.class, Service.Mode.PAYLOAD) 创建 Dispatch 对象，则请求将打包成一个 DOMSource 对象。

**注意**

使用 JAXB 对象时，响应和请求可以是提供的 JAXBContext 对象可以 marshal 和 unmarshal 的任何类型的对象。

例 41.8 “使用 Dispatch 对象制作一个途径” 显示使用 JAXB 对象在远程服务上进行单向调用的代码。

**例 41.8. 使用 Dispatch 对象制作一个途径**

```
// Creating a JAXBContext and an Unmarshaller for the request
JAXBContext jbc = JAXBContext.newInstance("org.apache.cxf.StockExample");
Unmarshaller u = jbc.createUnmarshaller();

// Read the request from disk
File rf = new File("request.xml");
GetStockPrice request = (GetStockPrice)u.unmarshal(rf);

// Dispatch disp created previously
disp.invokeOneWay(request);
```

## 41.2. 在服务提供商中使用 XML

**摘要**

**Provider 接口是一个低级 JAX-WS API，允许您实施直接将消息用作原始 XML 的服务提供商。在将消息传递给实施提供程序接口的对象之前，消息不会打包到 JAXB 对象。**

### 41.2.1. 消息传递模式

**概述**

实施提供程序接口的对象有两种 ***消息传递模式*** ：

- **消息 模式**
- **有效负载 模式**

**您指定的消息传递模式决定了传递给您的实施的消息传递详情级别。**

**消息模式**

**使用 *消息模式* 时，Provider 实现可用于完整的消息。完整的消息包括任何绑定特定的标头和打包程序。例如，使用 SOAP 绑定的 Provider 实现会接收请求完全指定的 SOAP 消息。从实施返回的任何响应都必须是完全指定的 SOAP 消息。**

**要指定 Provider 实现使用消息模式，方法是提供值 java.xml.ws.Service.Mode.MESSAGE 作为**

javax.xml.ws.ServiceMode 注解的值，如 例 41.9 “指定提供程序实施使用消息模式” 所示。

**例 41.9. 指定提供程序实施使用消息模式**

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
public class stockQuoteProvider implements Provider<SOAPMessage>
{
  ...
}
```

#### 有效负载模式

*在 有效负载模式中*，提供程序实施仅适用于消息的载荷。例如，在有效负载模式下工作的提供程序实施仅适用于 SOAP 消息的正文。绑定层处理任何绑定级别打包程序和标头。

当使用不使用特殊打包程序的绑定时，如 Apache CXF XML 绑定、有效负载模式和消息模式会提供相同的结果。

要指定 Provider 实现使用有效负载模式，方法是提供 java.xml.ws.Service.Mode.PAYLOAD 值作为 javax.xml.ws.ServiceMode 注解的值，如 例 41.10 “指定供应商实施使用 Payload 模式” 所示。

**例 41.10. 指定供应商实施使用 Payload 模式**

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class stockQuoteProvider implements Provider<DOMSource>
{
  ...
}
```

如果没有为 @ServiceMode 注释提供值，则 Provider 实现将使用有效负载模式。

### 41.2.2. 数据类型

#### 概述

由于它们是低级对象，因此提供程序实施无法使用与更高级别的消费者 API 相同的 JAXB 生成的类型。供应商实现可用于以下类型的对象：

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource

使用 Source 对象

提供程序实施可以接受和返回从 javax.xml.transform.Source 接口派生的对象。源对象是保存 XML 文档的低级别对象。每个源实施都提供了访问存储的 XML 文档和操作其内容的方法。以下对象实现 Source 接口：

DOMSource

包含 XML 消息作为文档对象模型(DOM)树。XML 消息作为一组使用 get Node （） 方法访问的节点对象保存。节点可以使用 setNode （） 方法更新或添加到 DOM 树中。

SAXSource

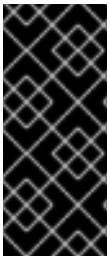
包含 XML 消息作为 XML (SAX)对象的简单 API。SAX 对象包含一个 InputSource 对象，其中包含原始数据和 XMLReader 对象来解析原始数据。

StreamSource

包含 XML 消息作为数据流。数据流可以像任何其他数据流一样操作。

如果您创建 Provider 对象使其使用通用 Source 对象，则 Apache CXF 会将消息返回为 SAXSource 对象。

可以使用端点的 source-preferred-format 属性来更改此行为。有关配置 Apache CXF 运行时的详情，请参考 第 IV 部分 “配置 Web 服务端点”。



重要

在使用 Source 对象时，开发人员负责确保所有需要的绑定特定打包程序都添加到消息中。例如，当与期望 SOAP 消息的服务交互时，开发人员必须确保所需的 SOAP 环境 elope 添加到传出请求中，并且 SOAP 信封内容正确。

使用 SOAPMessage 对象

当以下条件为 true 时，供应商实现可以使用 javax.xml.soap.SOAPMessage 对象：

- Provider 实现使用 SOAP 绑定
- Provider 实现使用消息模式

SOAPMessage 对象包含 SOAP 消息。它们包含一个 SOAPPart 对象和零个或多个 AttachmentPart 对象。SOAPPart 对象包含 SOAP 消息的特定部分，包括 SOAP 信封、任何 SOAP 标头和 SOAP 消息正文。AttachmentPart 对象包含作为附件传递的二进制数据。

使用 DataSource 对象

当满足以下条件时，供应商实现可以使用 javax.activation.DataSource 接口的对象：

- 实现使用 HTTP 绑定
- 实现使用消息模式

数据源对象提供了一种使用来自各种源的 MIME 类型数据的机制，包括 URL、文件和字节数组。

### 41.2.3. 实施提供程序对象

概述

提供商界面相对容易实施。它只有一个必须实施的方法 invoke （）。另外，它有三个简单的要求：

- 实施必须具有 @WebServiceProvider 注释。

- 实施必须具有默认的公共构造器。

- 实施必须实施类型化的 Provider 接口版本。

  换句话说，您无法实施 Provider<T> 接口。您必须实现一个使用 第 41.2.2 节 “数据类型” 中列出的聚合数据类型的接口版本。例如，您可以实施 Provider<SAXSource> 的实例。

实施 Provider 接口的复杂性位于处理请求消息的逻辑中，并构建正确的响应。

使用消息

与基于更高级别的 SEI 的服务实现不同，提供程序实施将请求作为原始 XML 数据接收，且必须发送响应作为原始 XML 数据。这要求开发人员熟悉所实施的服务所使用的消息。这些详细信息通常可在描述该服务的 WSDL 文档中找到。

WS-I Basic Profile 提供有关服务使用的消息的指南，包括：

- 请求的根元素基于与调用的操作对应的 wsdl:operation 元素的 name 属性的值。

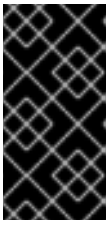
  

  警告

  如果服务使用 doc/literal 裸机消息，则请求的 root 元素将基于 wsdl:part 元素的 name 属性的值，引用由 wsdl:operation 元素引用。

- 所有消息的 root 元素都是命名空间限定性。

- 如果服务使用 rpc/literal 消息，则消息中的顶级元素不是命名空间限定性。



重要

顶级元素的子项可能具有命名空间限定性，但要确保您必须检查其架构定义。

- 如果服务使用 rpc/literal 消息，则顶级元素中的任何一个都是 null。
- 如果服务使用 doc/literal 信息，则消息的 schema 定义确定了任何元素是否合格。

### @WebServiceProvider 注释

若要被 JAX-WS 识别为服务实施，提供程序实施必须使用 @WebServiceProvider 注释进行解码。

表 41.2 “@WebServiceProvider Properties” 描述可以为 @WebServiceProvider 注释设置的属性。

表 41.2. @WebServiceProvider Properties

| 属性 | 描述 |
| --- | --- |
| **portName** | 指定定义服务端点的 **wsdl:port** 元素的 **name** 属性的值。 |
| **serviceName** | 指定包含服务端点的 **wsdl:service** 元素的 **name** 属性的值。 |
| **targetNamespace** | 指定服务的 WSDL 定义的目标名称空间。 |
| **wsdlLocation** | 指定定义该服务的 WSDL 文档的 URI。 |

所有这些属性都是可选的，默认为空。如果您将其留空，Apache CXF 将使用实施类中的信息创建值。

### 实施 invoke （）方法

Provider 接口只有一个方法 invoke （），必须实施它。invoke （） 方法接收被实施的 Provider 接口声明的传入请求，并返回打包到同一类型的对象中的响应消息。例如，一个 Provider<SOAPMessage> 接口的实现会接收作为 SOAPMessage 对象的请求，并将响应返回为 SOAPMessage 对象。

提供程序实施使用的消息传递模式决定了请求包含的绑定特定信息的数量。使用消息模式实施会接收所有绑定特定的打包程序和标头以及请求。它们还必须将所有绑定特定打包程序和标头添加到响应消息。使用有效负载模式的实现仅接收请求的正文。使用有效负载模式返回的 XML 文档被放在请求消息的正文中。

**例子**

例 41.11 “provider<SOAPMessage> 实现” 显示在消息模式中与 SOAPMessage 对象配合使用的 Provider 实现。

**例 41.11. provider<SOAPMessage> 实现**

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
                    serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.MESSAGE")
public class  stockQuoteReporterProvider implements Provider<SOAPMessage>
{
public stockQuoteReporterProvider()
  {
  }

public SOAPMessage invoke(SOAPMessage request)
  {
  SOAPBody requestBody = request.getSOAPBody();
  if(requestBody.getElementName.getLocalName.equals("getStockPrice"))
    {
    MessageFactory mf = MessageFactory.newInstance();
      SOAPFactory sf = SOAPFactory.newInstance();

    SOAPMessage response = mf.createMessage();
      SOAPBody respBody = response.getSOAPBody();
      Name bodyName = sf.createName("getStockPriceResponse");
      respBody.addBodyElement(bodyName);
      SOAPElement respContent = respBody.addChildElement("price");
      respContent.setValue("123.00");
      response.saveChanges();
    return response;
    }
    ...
  }
}
```

例 41.11 “provider<SOAPMessage> 实现” 中的代码执行以下操作：

指定以下类实施实施 wsdl:service 元素的服务的 Provider 对象，其名为 stockQuoteReporter，其 wsdl:port 元素名为 stockQuoteReporterPort。

指定此 Provider 实现使用消息模式。

提供所需的默认公共构造器。

提供使用 SOAPMessage 对象并返回 SOAPMessage 对象的 invoke （） 方法的实现。

从传入 SOAP 消息的正文中提取请求消息。

检查请求消息的 root 元素，以确定如何处理请求。

创建构建响应所需的工厂。

为响应构建 SOAP 消息。

将响应返回为 SOAPMessage 对象。

例 41.12 “Provider<DOMSource> 实现” 显示了在有效负载模式中使用 DOMSource 对象进行提供程序实施的示例。

**例 41.12. Provider<DOMSource> 实现**

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.PAYLOAD")
public class  stockQuoteReporterProvider implements Provider<DOMSource>
public stockQuoteReporterProvider()
  {
  }
```

```
public DOMSource invoke(DOMSource request)
  {
    DOMSource response = new DOMSource();
    ...
    return response;
  }
}
```

例 41.12 “Provider<DOMSource> 实现” 中的代码执行以下操作：

**指定该类实施服务（其 wsdl:service 元素名为 stockQuoteReporter ）的 Provider 对象，其 wsdl:port 元素名为 stockQuoteReporterPort。**

**指定此提供程序实施使用有效负载模式。**

**提供所需的默认公共构造器。**

**提供使用 DOMSource 对象并返回 DOMSource 对象的 invoke （） 方法的实现。**

# 第 42 章 使用上下文

## 摘要

JAX-WS 使用上下文在消息传递链中传递元数据。此元数据取决于其范围，可以被实施级别代码访问。也可以被 JAX-WS 处理程序访问，这些处理程序在实施级别下的消息上运行。

## 42.1. 了解上下文

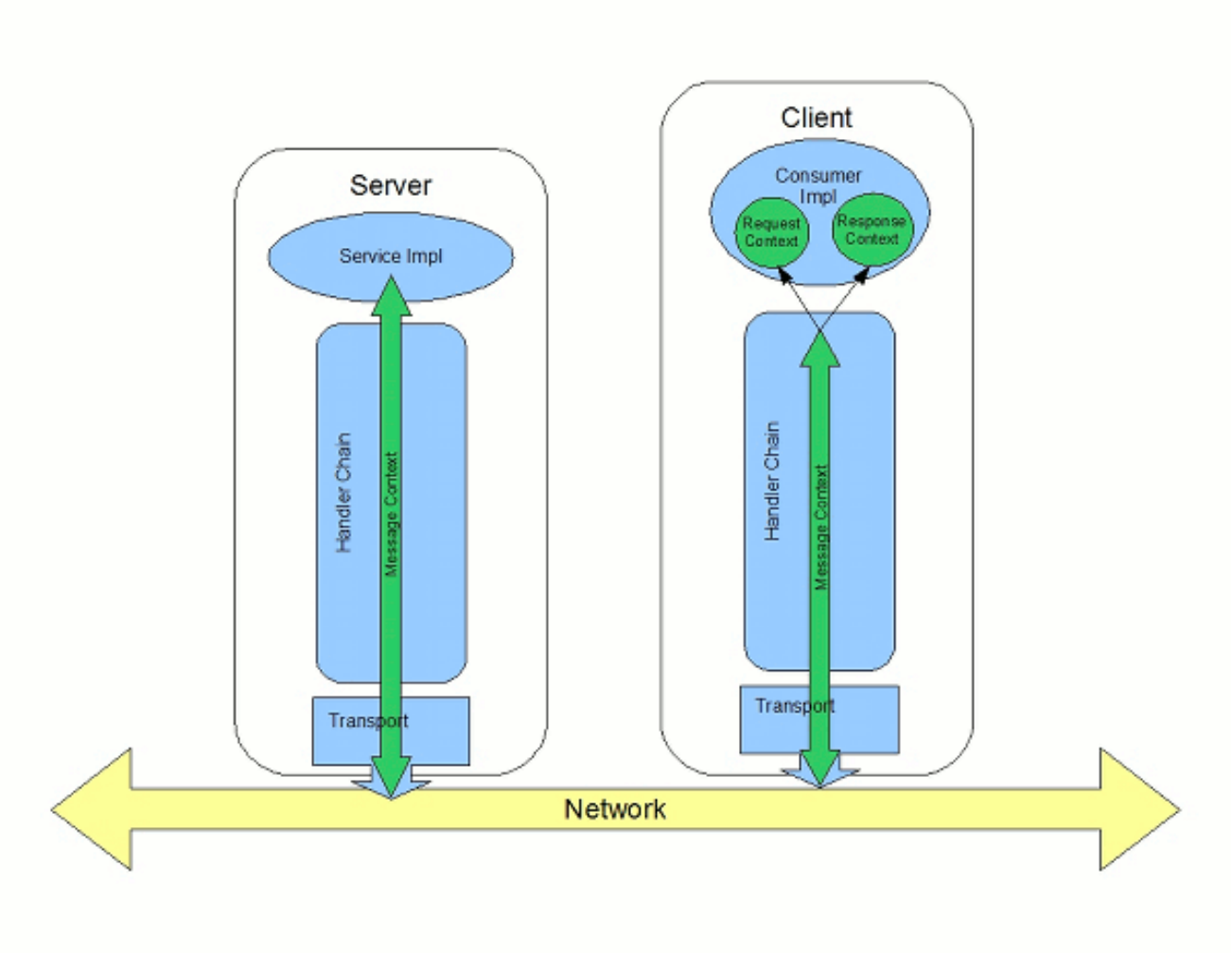
### 概述

在很多实例中，需要将有关消息的信息传递给应用程序的其他部分。Apache CXF 使用上下文机制进行此操作。上下文是包含与传出或传入消息相关的属性的映射。上下文中存储的属性通常是有关消息的元数据，以及用于通信消息的底层传输。例如，用于传输消息的传输特定标头（如 HTTP 响应代码或 JMS 关联 ID）存储在 JAX-WS 上下文中。

该上下文位于 JAX-WS 应用的所有级别。但是，它们根据您在消息处理堆栈中访问上下文的位置而有所不同。JAX-WS Handler 实施可直接访问上下文，并可以访问它们中设置的所有属性。服务实现通过注入来访问上下文，并且只能访问 APPLICATION 范围内设置的属性。消费者实施只能访问应用程序范围内设置的属性。

图 42.1 “消息上下文和消息处理路径” 显示上下文属性如何通过 Apache CXF 传递。当消息通过消息传递链时，其关联的消息上下文会随其一起传递。

图 42.1. 消息上下文和消息处理路径




属性如何在上下文中存储

消息上下文是 javax.xml.ws.handler.MessageContext 接口的所有实现。MessageContext 接口扩展了 java.util.Map<String key, Object value> 接口。将信息映射为键值对。

在消息上下文中，属性作为名称/值对存储。属性的键是用于标识 属性 的字符串。属性的值可以是存储在任何 Java 对象中的任何值。当从消息上下文返回值时，应用程序必须知道类型才会预期并相应地进行 cast。例如，如果属性的值存储在 UserInfo 对象中，它仍然会从消息上下文返回，该对象需要重新转换为 UserInfo 对象。

消息上下文中的属性也具有范围。范围决定了消息处理链中可以访问属性的位置。

属性范围

消息上下文中的属性有范围。属性可以在以下范围之一中：

应用

将属性范围为 APPLICATION 可用于 JAX-WS Handler 实施、消费者实施代码和服务提供商实施代码。如果处理程序需要将属性传递给服务提供商实施，它会将属性的范围设置为 APPLICATION。从消费者实施或服务提供商实施上下文设置的所有属性都自动限定为 APPLICATION。

HANDLER

属性范围为 HANDLER 仅适用于 JAX-WS Handler 实施。默认情况下，存储在处理程序实施的消息上下文中的属性被限定为 HANDLER。

您可以使用消息上下文的 setScope （） 方法更改属性的范围。例 42.1 “MessageContext.setScope （） 方法” 显示方法的签名。

例 42.1. MessageContext.setScope （） 方法

setScopeStringkeyMessageContext.Scopescopejava.lang.IllegalArgumentException

第一个参数指定属性的键。第二个参数指定属性的新范围。范围可以是：

- MessageContext.Scope.APPLICATION
- MessageContext.Scope.HANDLER

处理程序中的上下文概述

实施 JAX-WS Handler 接口的类可以直接访问消息的上下文信息。消息的上下文信息被传递给处理程序实现的 handleMessage （）、handleFault （） 和 close （） 方法。

处理程序实现有权访问消息上下文中存储的所有属性，无论其范围如何。此外，逻辑处理程序使用称为 LogicalMessageContext 的专用消息上下文。LogicalMessageContext 对象具有访问消息正文内容的方法。

服务实现中的上下文概述

服务实现可以访问范围为来自 消息 上下文的属性。服务提供商的实施对象通过 WebServiceContext 对象访问消息上下文。

如需更多信息，请参阅 第 42.2 节 “在服务实施中使用上下文”。

### 消费者实现中的上下文概述

消费者实施具有对消息上下文内容的间接访问权限。消费者实现有两个单独的消息上下文：

- Request context - 包含用于传出请求的属性副本
- 响应上下文 - 包含来自传入响应的属性副本

分配层在消费者实施的消息上下文和由处理程序实施使用的消息上下文之间传输属性。

当从消费者实施传递给分配层时，请求上下文的内容将复制到分配层使用的消息上下文中。从服务返回响应时，分配层将处理消息，并将适当的属性设置为消息上下文。在分配层处理响应后，它会将消息上下文中的所有属性限定为 APPLICATION，到消费者实施的响应上下文。

如需更多信息，请参阅 第 42.3 节 “在 Consumer Implementation 中使用上下文”。

## 42.2. 在服务实施中使用上下文

### 概述

使用 WebServiceContext 接口将上下文信息提供给服务实现。您可以从 WebServiceContext 对象获取 MessageContext 对象，该对象使用应用范围内当前请求的上下文属性填充。您可以操作 属性的值，并通过响应链传播它们。

注意

MessageContext 接口继承自 java.util.Map 接口。它的内容可以使用 Map 接口的方法进行操作。

### 获取上下文

要在服务实现中获取消息上下文，请执行以下操作：

1. 声明类型为 WebServiceContext 的变量。
2. 使用 javax.annotation.Resource 注释减少变量，以指示上下文信息被注入到 变量中。
3. 使用 get MessageContext （） 方法从 WebServiceContext 对象获取 MessageContext 对象。

重要

getMessageContext （） 只能用于使用 @WebMethod 注释。

例 42.2 “在服务实施中获取上下文对象” 显示获取上下文对象的代码。

**例 42.2. 在服务实施中获取上下文对象**

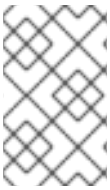
```
import javax.xml.ws.*;
import javax.xml.ws.handler.*;
import javax.annotation.*;

@WebServiceProvider
public class WidgetServiceImpl
{
  @Resource
  WebServiceContext wsc;

  @WebMethod
  public String getColor(String itemNum)
  {
    MessageContext context = wsc.getMessageContext();
  }

  ...
}
```

### 从上下文读取属性

获取适用于您的实现的 MessageContext 对象后，您可以使用 例 42.3 “MessageContext.get （） 方法” 中显示的 get （） 方法访问存储的属性。

例 42.3. MessageContext.get （） 方法

vgetObjectkey



注意

此 get （） 继承自 Map 接口。

key 参数是代表您要从上下文检索的属性的字符串。get （） 返回必须转换为属性的正确类型的对象。表 42.1 “服务实施上下文中提供的属性” 列出服务实施上下文中可用的多个属性。

重要

更改从上下文返回的对象的值也会更改上下文中的属性值。

例 42.4 “从服务的消息上下文获取属性” 显示获取代表调用操作的 WSDL 操作 元素名称的代码。

例 42.4. 从服务的消息上下文获取属性

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

 ...
 // MessageContext context retrieved in a previous example
 QName wsdl_operation = (QName)context.get(Message.WSDL_OPERATION);
```

在上下文中设置属性

获取适用于您的实现的 MessageContext 对象后，您可以使用 例 42.5 “MessageContext.put （） 方法” 中显示的 put （） 方法设置属性并更改现有属性。

例 42.5. MessageContext.put （） 方法

vputKkeyVvalueClassCastExceptionIllegalArgumentExceptionNullPointerException

如果在消息上下文中已存在设置的属性，则 put （） 方法将现有值替换为新值并返回旧值。如果消息上下文中不存在该属性，则 put （） 方法将设置属性并返回 null。

例 42.6 “在服务的消息上下文中设置属性” 显示用于为 HTTP 请求设置响应代码的代码。

**例 42.6. 在服务的消息上下文中设置属性**

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

...
// MessageContext context retrieved in a previous example
context.put(Message.RESPONSE_CODE, new Integer(404));
```

**支持的上下文**

表 42.1 “服务实施上下文中提供的属性” 列出通过服务实施对象中上下文访问的属性。

**表 42.1. 服务实施上下文中提供的属性**

| 属性名称 | 描述 |
| --- | --- |
| **org.apache.cxf.message.Message** | |
| PROTOCOL_HEADERS[a] | 指定特定于传输的标头信息。该值存储为 **java.util.Map<String, List<String>>**。 |
| RESPONSE_CODE | 指定返回到消费者的响应代码。该值存储为一个 **Integer** 对象。 |
| ENDPOINT_ADDRESS | 指定服务提供商的地址。该值存储为 **String**。 |
| HTTP_REQUEST_METHOD | 指定通过请求发送的 HTTP 动词。该值存储为 **String**。 |

| 属性名称 | 描述 |
| --- | --- |
| PATH_INFO | 指定正在请求的资源的路径。该值存储为 **String**。<br><br>该路径是主机名后 URI 的部分，并在任何查询字符串之前。例如，如果端点的 URI 是 http://cxf.apache.org/demo/widgets，则路径为 **/demo/widgets**。 |
| QUERY_STRING | 指定附加到用于调用请求的 URI 的查询（若有）。该值存储为 **String**。<br><br>在 URI 的末尾，查询会出现在 URI 的末尾，**?**。例如：如果向 http://cxf.apache.org/demo/widgets?color 发出了一个请求，则查询为 **颜色**。 |
| MTOM_ENABLED | 指定服务提供商是否可以将 MTOM 用于 SOAP 附加。该值存储为 **布尔值**。 |
| SCHEMA_VALIDATION_ENABLED | 指定服务提供商是否针对 schema 验证信息。该值存储为 **布尔值**。 |
| FAULT_STACKTRACE_ENABLED | 指定运行时是否提供堆栈追踪以及错误消息。该值存储为 **布尔值**。 |
| CONTENT_TYPE | 指定消息的 MIME 类型。该值存储为 **String**。 |
| BASE_PATH | 指定正在请求的资源的路径。该值存储为 **java.net.URL**。<br><br>该路径是主机名后 URI 的部分，并在任何查询字符串之前。例如，如果端点的 URL 是 http://cxf.apache.org/demo/widgets，基本路径为 **/demo/widgets**。 |
| 编码 | 指定消息的编码。该值存储为 **String**。 |
| FIXED_PARAMETER_ORDER | 指定参数是否必须以特定顺序显示在消息中。该值存储为 **布尔值**。 |
| MAINTAIN_SESSION | 指定消费者是否希望为将来的请求维护当前会话。该值存储为 **布尔值**。 |
| WSDL_DESCRIPTION | 指定定义所实施服务的 WSDL 文档。该值存储为 **org.xml.sax.InputSource** 对象。 |
| WSDL_SERVICE | 指定定义所实施服务的 **wsdl:service** 元素的合格名称。该值存储为 **QName**。 |
| WSDL_PORT | 指定 **wsdl:port** 元素的合格名称，用于定义用于访问该服务的端点。该值存储为 **QName**。 |

| 属性名称 | 描述 |
| --- | --- |
| WSDL_INTERFACE | 指定定义所实施服务的 **wsdl:portType** 元素的合格名称。该值存储为 **QName**。 |
| WSDL_OPERATION | 指定与消费者调用的操作对应的 **wsdl:operation** 元素的合格名称。该值存储为 **QName**。 |
| **javax.xml.ws.handler.MessageContext** | |
| MESSAGE_OUTBOUND_PROPERTY | 指定消息是否出站。该值存储为 **布尔值**。**true** 指定消息是出站的。 |
| INBOUND_MESSAGE_ATTACHMENTS | 包含请求消息中包含的任何附件。该值存储为 **java.util.Map<String, DataHandler** >。<br><br>映射的键值是标头的 MIME Content-ID。 |
| OUTBOUND_MESSAGE_ATTACHMENTS | 包含响应消息的任何附件。该值存储为 **java.util.Map<String, DataHandler** >。<br><br>映射的键值是标头的 MIME Content-ID。 |
| WSDL_DESCRIPTION | 指定定义所实施服务的 WSDL 文档。该值存储为 **org.xml.sax.InputSource** 对象。 |
| WSDL_SERVICE | 指定定义所实施服务的 **wsdl:service** 元素的合格名称。该值存储为 **QName**。 |
| WSDL_PORT | 指定 **wsdl:port** 元素的合格名称，用于定义用于访问该服务的端点。该值存储为 **QName**。 |
| WSDL_INTERFACE | 指定定义所实施服务的 **wsdl:portType** 元素的合格名称。该值存储为 **QName**。 |
| WSDL_OPERATION | 指定与消费者调用的操作对应的 **wsdl:operation** 元素的合格名称。该值存储为 **QName**。 |
| HTTP_RESPONSE_CODE | 指定返回到消费者的响应代码。该值存储为一个 **Integer** 对象。 |
| HTTP_REQUEST_HEADERS | 指定请求上的 HTTP 标头。该值存储为 **java.util.Map<String, List<String>>**。 |
| HTTP_RESPONSE_HEADERS | 指定响应的 HTTP 标头。该值存储为 **java.util.Map<String, List<String>>**。 |
| HTTP_REQUEST_METHOD | 指定通过请求发送的 HTTP 动词。该值存储为 **String**。 |

| 属性名称 | 描述 |
| --- | --- |
| SERVLET_REQUEST | 包含 servlet 的请求对象。该值存储为 **javax.servlet.http.HttpServletRequest**。 |
| SERVLET_RESPONSE | 包含 servlet 的响应对象。该值存储为 **javax.servlet.http.HttpResponse**。 |
| SERVLET_CONTEXT | 包含 servlet 的上下文对象。该值存储为 **javax.servlet.ServletContext**。 |
| PATH_INFO | 指定正在请求的资源的路径。该值存储为 **String**。<br><br>该路径是主机名后 URI 的部分，并在任何查询字符串之前。例如，如果端点的 URL 是 http://cxf.apache.org/demo/widgets，其路径为 **/demo/widgets**。 |
| QUERY_STRING | 指定附加到用于调用请求的 URI 的查询（若有）。该值存储为 **String**。<br><br>在 URI 的末尾，查询会出现在 URI 的末尾，**?**。例如：如果向 http://cxf.apache.org/demo/widgets?color 发出了一个请求，查询字符串为**颜色**。 |
| REFERENCE_PARAMETERS | 指定 WS-Addressing 参考参数。这包括其 **wsa:IsReferenceParameter** 属性设置为 **true** 的所有 SOAP 标头。该值存储为 **java.util.List**。 |
| **org.apache.cxf.transport.jms.JMSConstants** | |
| JMS_SERVER_HEADERS | 包含 JMS 消息标头。如需更多信息，请参阅 第 42.4 节 “使用 JMS 消息属性”。 |
| [a] 使用 HTTP 此属性时，与标准 JAX-WS 定义的属性相同。 | |

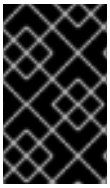
## 42.3. 在 CONSUMER IMPLEMENTATION 中使用上下文

**概述**

**消费者实施可通过 BindingProvider 接口访问上下文信息。BindingProvider 实例在两个独立上下文中包含上下文信息：**

- 请求上下文 *请求上下文* 允许您设置影响出站消息的属性。请求上下文属性应用到特定的端口实例，并且设置后，属性会影响端口上进行的每个操作调用，直到这些时间明确清除。例如，您可以使用 request context 属性来设置连接超时，或者初始化在标头中发送的数据。

- 响应上下文 *响应上下文* 允许您读取响应从当前线程发出的最后一个操作调用所设置的属性值。每次操作调用后都会重置响应上下文属性。例如，您可以访问响应上下文属性，以读取从最后一个入站消息接收的标头信息。



重要

只有放置在消息上下文的应用程序范围内的信息才可以被消费者实施访问。

获取上下文

使用 javax.xml.ws.BindingProvider 接口获取上下文。BindingProvider 接口有两个获取上下文的方法：

- getRequestContext() 例 42.7 “getRequestContext （） 方法” 中显示的 getRequestContext （） 方法将请求上下文返回为 Map 对象。返回的 Map 对象可用于直接操作上下文的内容。

> 例 42.7. getRequestContext （） 方法
>
> Map<String, Object>getRequestContext

- getResponseContext() 例 42.8 “getResponseContext （） 方法” 中显示的 getResponseContext （） 将响应上下文返回为 Map 对象。返回的 Map 对象的内容反映了响应上下文内容的状态，来自当前线程中进行的远程服务的最新成功请求。

> 例 42.8. getResponseContext （） 方法
>
> Map<String, Object>getResponseContext

由于代理对象实现 BindingProvider 接口，因此可以通过广播代理对象来获取 BindingProvider 对象。从 BindingProvider 对象获取的上下文仅对用于创建它的代理对象调用的操作有效。

例 42.9 “获取 Consumer 的 Request 上下文” 显示获取代理请求上下文的代码。

**例 42.9. 获取 Consumer 的 Request 上下文**

```
// Proxy widgetProxy obtained previously
BindingProvider bp = (BindingProvider)widgetProxy;
Map<String, Object> requestContext = bp.getRequestContext();
```

**从上下文读取属性**

消费者上下文存储在 java.util.Map<String, Object> 对象中。映射的键是 String 对象和包含任意对象的值。使用 java.util.Map.get （） 访问响应上下文属性映射中的条目。

要检索特定的上下文属性 *ContextPropertyName*，请使用 例 42.10 “读取响应上下文属性” 中显示的代码。

**例 42.10. 读取响应上下文属性**

```
// Invoke an operation.
port.SomeOperation();

// Read response context property.
java.util.Map<String, Object> responseContext =
  ((javax.xml.ws.BindingProvider)port).getResponseContext();
PropertyType propValue = (PropertyType) responseContext.get(ContextPropertyName);
```

**在上下文中设置属性**

消费者上下文是存储在 java.util.Map<String, Object > 对象中的哈希映射。映射的键是 String 对象和值，它们是任意对象。要在上下文中设置属性，请使用 java.util.Map.put （） 方法。

虽然您可以在请求上下文和响应上下文中设置属性，但只有对请求上下文所做的更改对消息处理产生任何影响。当每个远程调用在当前线程上完成时，响应上下文中的属性将被重置。

例 42.11 “设置请求上下文属性” 中显示的代码通过设置 BindingProvider.ENDPOINT_ADDRESS_PROPERTY 的值来更改目标服务提供商的地址。

**例 42.11. 设置请求上下文属性**

```
// Set request context property.
java.util.Map<String, Object> requestContext =
    ((javax.xml.ws.BindingProvider)port).getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
"http://localhost:8080/widgets");

// Invoke an operation.
port.SomeOperation();
```

**重要**

**在请求上下文中设置属性后，将其值用于后续所有远程调用。您可以更改值，然后使用更改的值。**

**支持的上下文**

**Apache CXF 在消费者实现中支持以下上下文属性：**

**表 42.2. 消费者上下文属性**

| 属性名称 | 描述 |
|---|---|
| **javax.xml.ws.BindingProvider** | |
| ENDPOINT_ADDRESS_PROPERTY | 指定目标服务的地址。该值存储为 **String**。 |
| USERNAME_PROPERTY[a] | 指定用于 HTTP 基本身份验证的用户名。该值存储为 **String**。 |
| PASSWORD_PROPERTY[b] | 指定用于 HTTP 基本身份验证的密码。该值存储为 **String**。 |
| SESSION_MAINTAIN_PROPERTY[c] | 指定客户端是否希望维护会话信息。该值作为 **布尔值对象存储**。 |
| **org.apache.cxf.ws.addressing.JAXWSAConstants** | |
| CLIENT_ADDRESSING_PROPERTIES | 指定消费者用来联系所需服务提供商的 WS-寻址信息。该值存储为 **org.apache.cxf.ws.addressing.AddressingProperties**。 |
| **org.apache.cxf.transports.jms.context.JMSConstants** | |

| 属性名称 | 描述 |
| --- | --- |
| JMS_CLIENT_REQUEST_HEADERS | 包含消息的 JMS 标头。如需更多信息，请参阅 第 42.4 节 “使用 JMS 消息属性”。 |

[a] 此属性被 HTTP 安全设置中定义的用户名覆盖。

[b] 此属性被 HTTP 安全设置中定义的密码覆盖。

[c] Apache CXF 忽略此属性。

### 42.4. 使用 JMS 消息属性

**摘要**

**Apache CXF JMS 传输具有一个上下文机制，可用于检查 JMS 消息的属性。上下文机制也可用于设置 JMS 消息的属性。**

#### 42.4.1. 检查 JMS 消息标头

**摘要**

**使用者和服务使用不同的上下文机制来访问 JMS 邮件标题属性。但是，这两种机制都会将标头属性返回为 org.apache.cxf.transports.jms.context.JMSMessageHeadersType 对象。**

**在服务中获取 JMS Message Headers**

**要从 WebServiceContext 对象获取 JMS 消息标头属性，请执行以下操作：**

1. **获取上下文，如 “获取上下文”一节 所述。**
2. **使用消息上下文的 get （） 方法以及参数 org.apache.cxf.transports.jms.JMSConstants.JMS_SERVER_HEADERS 获取消息标头。**

**例 42.12 “在服务实施中获取 JMS 消息标头” 显示用于从服务的消息上下文获取 JMS 消息标头的代码：**

> **例 42.12. 在服务实施中获取 JMS 消息标头**

```
import org.apache.cxf.transport.jms.JMSConstants;
import org.apache.cxf.transports.jms.context.JMSMessageHeadersType;

@WebService(serviceName = "HelloWorldService",
                    portName = "HelloWorldPort",
                    endpointInterface = "org.apache.cxf.hello_world_jms.HelloWorldPortType",
                    targetNamespace = "http://cxf.apache.org/hello_world_jms")
 public class GreeterImplTwoWayJMS implements HelloWorldPortType
 {
   @Resource
   protected WebServiceContext wsContext;
   ...

   @WebMethod
   public String greetMe(String me)
   {
     MessageContext mc = wsContext.getMessageContext();
     JMSMessageHeadersType headers = (JMSMessageHeadersType)
mc.get(JMSConstants.JMS_SERVER_HEADERS);
      ...
    }
     ...
}
```

**在 Consumer 中获取 JMS 消息标头属性**

**从 JMS 传输成功检索消息后，您可以使用使用者的响应上下文来检查 JMS 标头属性。另外，您可以设置或检查客户端在超时前等待响应的时间长度，如 “client Receive Timeout”一节 所述。 要从消费者的响应上下文中获取 JMS 消息标头，请执行以下操作：**

1. **获取响应上下文，如 “获取上下文”一节 所述。**

2. **使用上下文的 get （） 方法和 org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RESPONSE_HEADERS 作为参数，从响应上下文获取 JMS 消息标头属性。**

**例 42.13 “从 Consumer Response Header 获取 JMS 标头” 显示用于从消费者的响应上下文获取 JMS 消息标头属性的代码。**

**例 42.13. 从 Consumer Response Header 获取 JMS 标头**

```
import org.apache.cxf.transports.jms.context.*;
// Proxy greeter initialized previously
BindingProvider  bp = (BindingProvider)greeter;
Map<String, Object> responseContext = bp.getResponseContext();
JMSMessageHeadersType responseHdr = (JMSMessageHeadersType)
```

```
                responseContext.get(JMSConstants.JMS_CLIENT_RESPONSE_HEADERS);
...
}
```

例 42.13 “从 Consumer Response Header 获取 JMS 标头” 中的代码执行以下操作：

将代理转换为 BindingProvider。

获取响应上下文。

从响应上下文检索 JMS 消息标头。

### 42.4.2. 检查消息标头属性

#### 标准 JMS 标头属性

表 42.3 “JMS Header Properties” 列出您可以在 JMS 标头中检查的标准属性。

**表 42.3. JMS Header Properties**

| 属性名称 | 属性类型 | getter Method |
|---|---|---|
| 关联 ID | **string** | **getJMSCorralationID()** |
| 交付模式 | **int** | **getJMSDeliveryMode()** |
| 消息过期 | **long** | **getJMSExpiration()** |
| 消息 ID | **string** | **getJMSMessageID()** |
| 优先级 | **int** | **getJMSPriority()** |
| redelivered | **布尔值** | **getJMSRedlivered()** |
| 时间戳 | **long** | **getJMSTimeStamp()** |
| 类型 | **string** | **getJMSType()** |
| 生存时间 | **long** | **getTimeToLive()** |

可选标头属性

此外，您可以使用 JMSMessageHeadersType.getProperty （） 检查 JMS 标头中存储的任何可选属性。可选属性返回为 org.apache.cxf.transports.jms.context.JMSPropertyType 的列表。可选属性作为名称/值对存储。

Example

例 42.14 “读取 JMS 标头属性” 显示使用响应上下文检查某些 JMS 属性的代码。

例 42.14. 读取 JMS 标头属性

```
// JMSMessageHeadersType messageHdr retrieved previously
System.out.println("Correlation ID: "+messageHdr.getJMSCorrelationID());
System.out.println("Message Priority: "+messageHdr.getJMSPriority());
System.out.println("Redelivered: "+messageHdr.getRedelivered());

JMSPropertyType prop = null;
List<JMSPropertyType> optProps = messageHdr.getProperty();
Iterator<JMSPropertyType> iter = optProps.iterator();
while (iter.hasNext())
{
  prop = iter.next();
   System.out.println("Property name: "+prop.getName());
   System.out.println("Property value: "+prop.getValue());
}
```

例 42.14 “读取 JMS 标头属性” 中的代码执行以下操作：

显示消息关联 ID 的值。

显示消息优先级属性的值。

显示消息 redelivered 属性的值。

获取消息可选标头属性的列表。

获取 迭代 器以遍历属性列表。

迭代可选属性列表并打印其名称和值。

### 42.4.3. 设置 JMS Properties

**摘要**

在消费者端点中使用请求上下文，您可以设置多个 JMS 消息队列属性和消费者端点的超时值。这些属性对单个调用有效。每次在服务代理上调用操作时，您必须重置它们。

请注意，您无法在服务中设置标头属性。

**JMS Header Properties**

表 42.4 “settable JMS Header Properties” 列出 JMS 标头中可以使用消费者端点的请求上下文设置的属性。

**表 42.4. settable JMS Header Properties**

| 属性名称 | 属性类型 | setter Method |
| --- | --- | --- |
| 关联 ID | **string** | **setJMSCorralationID()** |
| 交付模式 | **int** | **setJMSDeliveryMode()** |
| 优先级 | **int** | **setJMSPriority()** |
| 生存时间 | **long** | **setTimeToLive()** |

要设置这些属性，请执行以下操作：

1. 创建 **org.apache.cxf.transports.jms.context.JMSMessageHeadersType** 对象。
2. 使用 表 42.4 “settable JMS Header Properties” 中描述的适当设置方法填充您要设置的值。
3. 使用 **org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_REQUEST_HEADERS** 作为第

一个参数，将值设置为请求上下文的 put （） 方法。

### 可选的 JMS 标头属性

您还可以将可选属性设置为 JMS 标头。可选的 JMS 标头属性存储在 JMSMessageHeadersType 对象中，用于设置其他 JMS 标头属性。它们存储为包含 org.apache.cxf.transports.jms.context.JMSPropertyType 对象的 List 对象。要在 JMS 标头中添加可选属性，请执行以下操作：

1. 创建 JMSPropertyType 对象。
2. 使用 setName （） 设置属性的 name 字段。
3. 使用 setValue （） 设置属性的 value 字段。
4. 使用 JMSMessageHeadersType.getProperty （）.add (JMSPropertyType) 将属性添加到 JMS 消息标头中。
5. 重复这个过程，直到所有属性都添加到消息标头中。

### client Receive Timeout

除了 JMS 标头属性外，您还可以设置消费者端点在超时前等待响应的时间长度。您可以使用 org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RECEIVE_TIMEOUT 调用请求上下文的 put （） 方法来设置值，时长 （以毫秒为单位），您希望消费者等待第二个参数。

### Example

例 42.15 “使用请求上下文设置 JMS 属性” 显示使用请求上下文设置某些 JMS 属性的代码。

**例 42.15. 使用请求上下文设置 JMS 属性**

```
import org.apache.cxf.transports.jms.context.*;
 // Proxy greeter initialized previously
InvocationHandler handler = Proxy.getInvocationHandler(greeter);

BindingProvider bp= null;
if (handler instanceof BindingProvider)
```

```
{
  bp = (BindingProvider)handler;
  Map<String, Object> requestContext = bp.getRequestContext();

  JMSMessageHeadersType requestHdr = new JMSMessageHeadersType();
  requestHdr.setJMSCorrelationID("WithBob");
  requestHdr.setJMSExpiration(3600000L);


  JMSPropertyType prop = new JMSPropertyType;
  prop.setName("MyProperty");
  prop.setValue("Bluebird");
  requestHdr.getProperty().add(prop);

  requestContext.put(JMSConstants.CLIENT_REQUEST_HEADERS, requestHdr);

  requestContext.put(JMSConstants.CLIENT_RECEIVE_TIMEOUT, new Long(1000));
}
```

**例 42.15 “使用请求上下文设置 JMS 属性” 中的代码执行以下操作：**

**获取您要更改的 JMS 属性的代理的 InvocationHandler。**

**检查 InvocationHandler 是否为 BindingProvider。**

**将返回的 InvocationHandler 对象转换为 BindingProvider 对象，以检索请求上下文。**

**获取请求上下文。**

**创建一个 JMSMessageHeadersType 对象来保存新的消息标头值。**

**设置 Correlation ID。**

**将 Expiration 属性设置为 60 分钟。**

**创建新的 JMSPropertyType 对象。**

设置可选属性的值。

在消息标头中添加 optional 属性。

将 JMS 消息标头值设置为请求上下文。

将 client receive timeout 属性设置为 1 秒。

# 第 43 章 编写处理程序

## 摘要

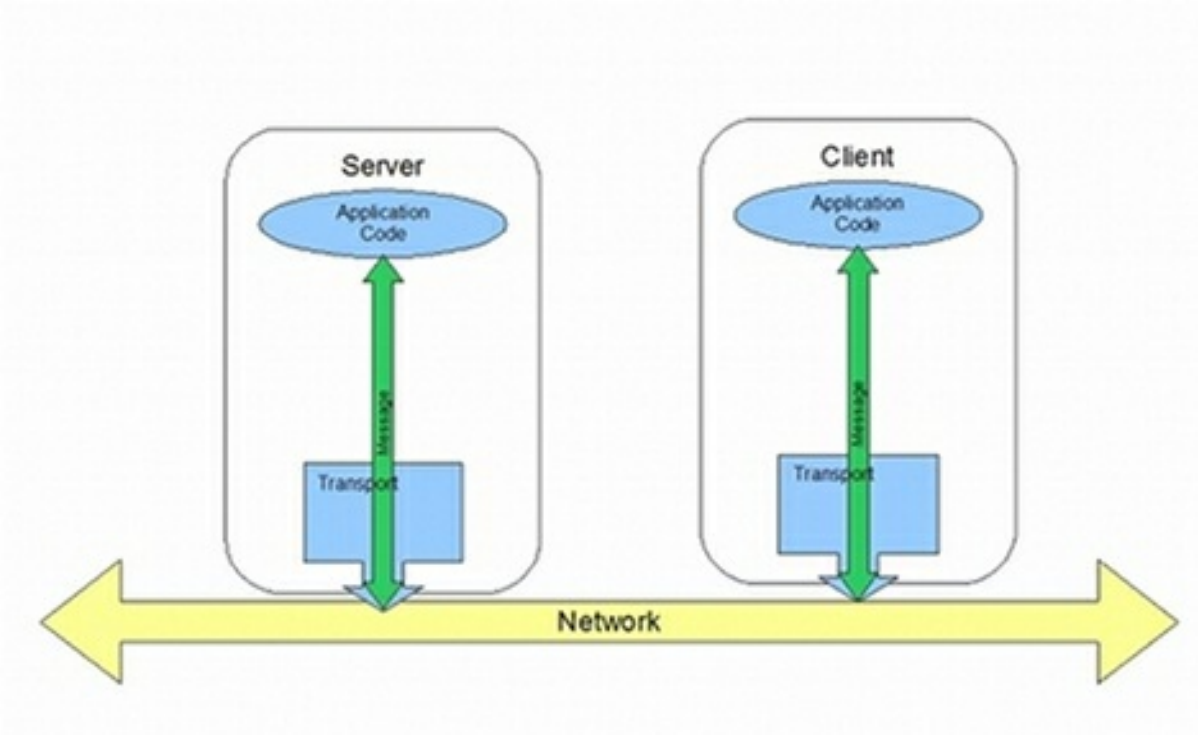
JAX-WS 提供灵活的插件框架，用于向应用添加消息处理模块。这些模块称为处理程序，独立于应用级别代码，可以提供低级消息处理功能。

## 43.1. 处理程序：简介

### 概述

当服务代理调用服务上的操作时，操作的参数将传递给 Apache CXF，其中它们构建到消息中并放置在有线条上。当服务收到消息时，Apache CXF 从有线条读取消息，重新构建消息，然后将操作参数传递给负责实施操作的应用程序代码。当应用程序代码完成处理请求时，回复消息会在其出去到源自请求的服务代理时执行类似事件链。这在 图 43.1 “消息交换路径” 中显示。

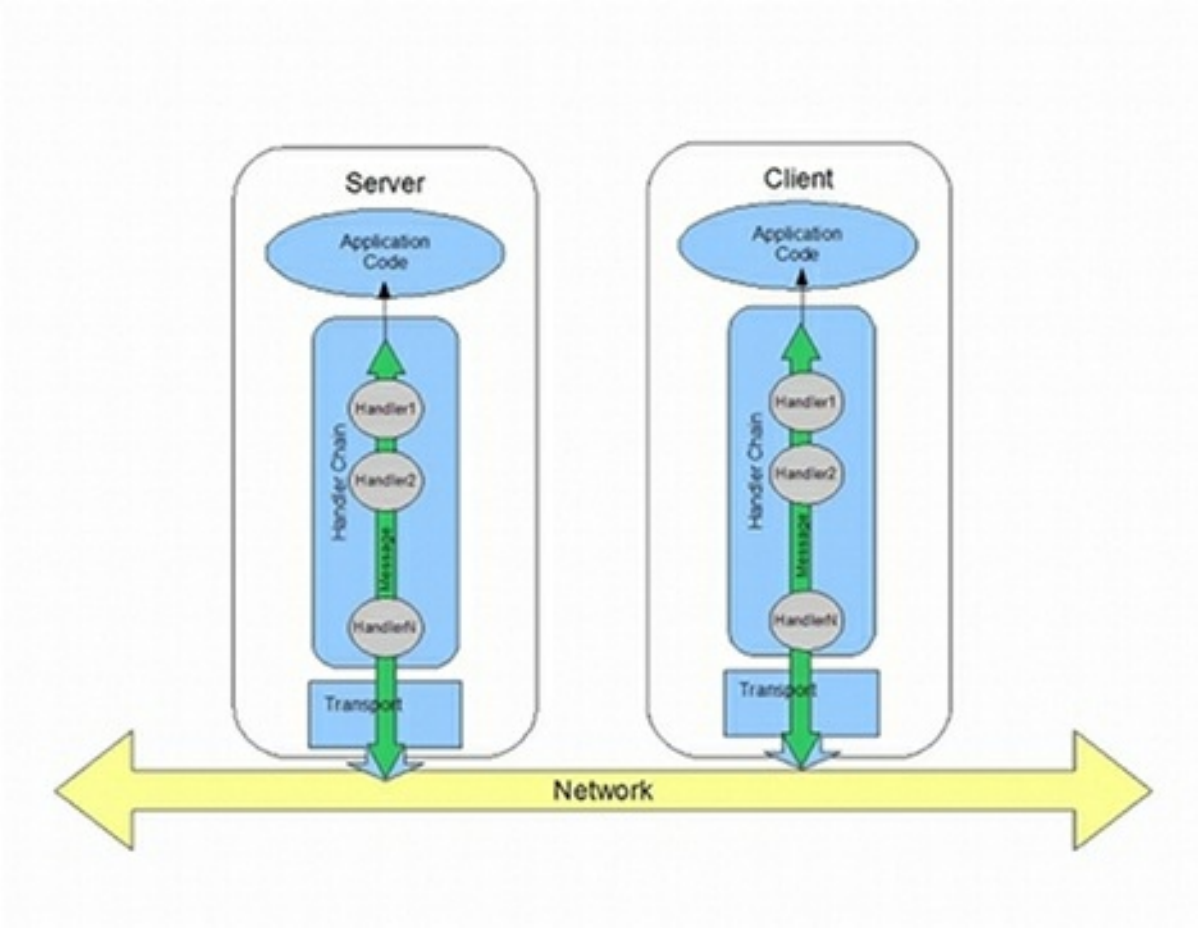
图 43.1. 消息交换路径




JAX-WS 定义用于操作应用级别代码和网络之间的消息数据的机制。例如，您可能希望通过开放网络传递的消息数据使用专有加密机制进行加密。您可以编写加密和解密数据的 JAX-WS 处理程序。然后，您可以将处理程序插入到所有客户端和服务器的消息处理链中。

如 图 43.2 “带有处理程序的消息交换路径” 所示，处理程序放置在链中，它会在应用级别代码和将消息放入网络中的传输代码之间遍历。

图 43.2. 带有处理程序的消息交换路径




处理程序类型

JAX-WS 规范定义了两种基本处理程序类型：

- 逻辑处理程序 逻辑处理程序可以处理消息有效负载和消息上下文中存储的属性。例如，如果应用使用纯 XML 消息，逻辑处理程序可以访问整个消息。如果应用使用 SOAP 消息，则逻辑处理程序可以访问 SOAP 正文的内容。它们无法访问 SOAP 标头或任何附件，除非它们被放入消息上下文中。

  逻辑处理程序放在处理程序链中的应用程序代码最接近的。这意味着，当消息从应用程序代码传递给传输时，首先执行它们。从网络接收消息并传递回应用程序代码时，逻辑处理程序将最后执行。

- 协议处理程序 协议处理程序可以处理从网络接收的整个消息，以及存储在消息上下文中的属性。例如，如果应用使用 SOAP 消息，则协议处理程序将能够访问 SOAP 正文、SOAP 标头和任何附件的内容。

  协议处理程序放在处理程序链中的传输最接近。这意味着，在从网络收到消息时，首先执行它们。当消息从应用程序代码发送到网络时，协议处理程序将最后执行。

**注意**

Apache CXF 支持的唯一协议处理程序特定于 SOAP。

处理程序的实现

两个处理程序类型之间的区别非常小，它们共享一个通用的基本接口。由于其常见的父项，逻辑处理程序和协议处理程序共享必须实施的一些方法，包括：

- handleMessage() handleMessage （） 方法是任何处理程序中的中央方法。它是负责处理正常消息的方法。
- handleFault() handleFault （） 是负责处理错误消息的方法。
- close() 当消息到达链末尾时，在处理程序链中执行的所有处理程序都会调用 close （）。它用于清理消息处理期间消耗的任何资源。

逻辑处理器实现和协议处理器的实现之间的不同会围绕以下内容：

- 实施的特定接口

  所有处理程序实施从处理程序接口派生的接口。逻辑处理程序实施 LogicalHandler 接口。协议处理程序实现处理程序接口的特定协议扩展。例如，SOAP 处理程序实施 SOAPHandler 接口。

- 处理程序可用的信息量

  协议处理程序有权访问消息的内容以及所有与消息内容一起打包的协议特定信息。逻辑处理程序只能访问消息的内容。逻辑处理程序不知道协议详情。

将处理程序添加到应用程序

要在应用程序中添加处理器，您必须执行以下操作：

1. 确定处理程序是否在服务提供商、消费者或两者中使用。

2. 确定最适合该作业的处理程序类型。

3. 实施正确的接口。

   要实现逻辑处理器，请参阅 第 43.2 节 “实施逻辑处理程序”。

   要实现协议处理器，请参阅 第 43.4 节 “实施协议处理程序”。

4. 将您的端点配置为使用处理程序。请参阅 第 43.10 节 “配置端点以使用处理程序”。

## 43.2. 实施逻辑处理程序

### 概述

逻辑处理程序实施 javax.xml.ws.handler.LogicalHandler 接口。例 43.1 “LogicalHandler Synopsis” 中显示的 LogicalHandler 接口将 LogicalMessageContext 对象传递给 handleMessage （） 方法和 handleFault （） 方法。上下文 对象提供对消息 的正文 和在消息交换上下文中设置的任何属性的访问。

**例 43.1. LogicalHandler Synopsis**

```
public interface LogicalHandler extends Handler
{
  boolean handleMessage(LogicalMessageContext context);
  boolean handleFault(LogicalMessageContext context);
  void close(LogicalMessageContext context);
}
```

### 流程

要实现逻辑手，您可以执行以下操作：

1. 实施处理程序所需的任何 第 43.6 节 “初始化处理程序” 逻辑。

2. 实施 第 43.3 节 “处理逻辑处理程序中的消息” 逻辑。

3. 实施 第 43.7 节 “处理故障消息” 逻辑。

4. 在完成时，实施 第 43.8 节 “关闭处理程序” 处理程序的逻辑。

5. 在销毁前，为 第 43.9 节 “释放处理程序” 处理程序的资源实施任何逻辑。

### 43.3. 处理逻辑处理程序中的消息

#### 概述

正常消息处理由 handleMessage （） 方法处理。

handleMessage （） 方法收到一个 LogicalMessageContext 对象，它提供对消息正文的访问以及消息上下文中存储的任何属性。

handleMessage （） 方法返回 true 或 false，具体取决于消息处理如何继续。它还可能会抛出异常。

#### 获取消息数据

传递给逻辑消息处理程序的 LogicalMessageContext 对象允许使用上下文的 getMessage （） 方法访问消息正文。getMessage （） 方法（如 例 43.2 “在逻辑处理程序中获取消息有效负载的方法” 所示）将消息有效负载返回为 LogicalMessage 对象。

**例 43.2. 在逻辑处理程序中获取消息有效负载的方法**

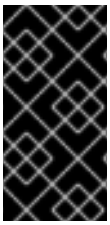
```
LogicalMessagegetMessage
```

具有 LogicalMessage 对象后，您可以使用它来操作消息正文。例 43.3 “逻辑消息冻结器” 中显示的

LogicalMessage 接口具有 getters 和 setters，用于处理实际消息正文。

**例 43.3.** 逻辑消息冻结器

```
LogicalMessageSourcegetPayloadObjectgetPayloadJAXBContextcontextsetPayloadObjectpa
yloadJAXBContextcontextsetPayloadSourcepayload
```



**重要**

消息有效负载的内容由使用的绑定类型决定。SOAP 绑定只允许访问消息的 SOAP 正文。XML 绑定允许访问整个消息正文。

**使用消息正文作为 XML 对象**

逻辑消息的一对 getters 和 setters 使用消息有效负载作为 javax.xml.transform.dom.DOMSource 对象。

没有参数的 getPayload （） 方法将消息有效负载返回为 DOMSource 对象。返回的对象是实际消息有效负载。对返回对象所做的任何更改都会立即更改消息正文。

您可以使用采用单个 Source 对象的 setPayload （） 方法将消息的正文替换为 DOMSource 对象。

**使用消息正文作为 JAXB 对象**

通过另一对 getters 和 setters，您可以将消息有效负载用作 JAXB 对象。它们使用 JAXBContext 对象将消息有效负载转换为 JAXB 对象。

要使用 JAXB 对象，请执行以下操作：

1. 获取 JAXBContext 对象，它可以管理消息正文中的数据类型。

   有关创建 JAXBContext 对象的详情，请参考 第 39 章 *使用 A JAXBContext 对象*。

2. 获取 例 43.4 “将消息正文作为 JAXB 对象获取” 中显示的消息正文。

**例 43.4. 将消息正文作为 JAXB 对象获取**

```
JAXBContext jaxbc = JAXBContext(myObjectFactory.class);
Object body = message.getPayload(jaxbc);
```

3. 将返回的对象转换为正确的类型。

4. 根据需要操作消息正文。

5. 将更新的消息正文重新置于上下文中，如 例 43.5 “使用 JAXB 对象更新消息正文” 所示。

**例 43.5. 使用 JAXB 对象更新消息正文**

```
message.setPayload(body, jaxbc);
```

**使用上下文属性**

传递给逻辑处理程序的逻辑消息上下文是应用的消息上下文的实例，可以访问它中存储的所有属性。处理程序有权访问应用范围和 HANDLER 范围中的属性。

与应用的消息上下文一样，逻辑消息上下文是 Java Map 的子类。要访问存储在上下文中的属性，您可以使用从 Map 接口继承的 get （） 方法和 put （） 方法。

默认情况下，您在逻辑处理程序内部的消息上下文中设置的任何属性都会被分配 HANDLER 范围。如果您希望应用程序代码能够访问所需的属性，您需要使用上下文的 setScope （） 方法将属性明确设置为 APPLICATION。

有关在消息上下文中使用属性的详情，请参考 第 42.1 节 “了解上下文”。

**确定消息的方向**

知道消息通过处理程序链的方向通常非常重要。例如，您想要从传入请求检索安全令牌，并将安全令牌附加到传出响应。

消息的方向存储在消息上下文的出站消息属性中。您可以使用 MessageContext.MESSAGE_OUTBOUND_PROPERTY 密钥从消息上下文检索出站消息属性，如 例 43.6 “从 SOAP 消息上下文获取消息方向” 所示。

**例 43.6. 从 SOAP 消息上下文获取消息方向**

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

属性作为 布尔值对象存储。您可以使用对象的 booleanValue （） 方法来确定属性值。如果属性设为 true，则会出站消息。如果属性设为 false，则消息为 inbound。

### 确定返回值

handleMessage （） 方法如何完成其消息处理对消息处理的方式有直接影响。它可以通过执行以下操作之一完成：

1. 将 true-Returning true 信号返回到消息处理应该继续正常运行的 Apache CXF 运行时。下一个处理程序（若有）调用了其 handleMessage （）。

2. 将 false-Returning false 信号返回到正常消息处理必须停止的 Apache CXF 运行时。运行时的继续取决于用于 当前消息 的消息交换模式。

   对于请求响应消息交换，会出现以下情况：

   a. 消息处理的方向被撤销。

      例如，如果某个请求由服务提供商处理，消息将停止进入服务的实施对象。相反，它会发回到绑定，以返回到源自请求的消费者。

   b. 在新处理方向中驻留在处理程序链的任何消息处理程序都按照它们位于链中的顺序调用其 handleMessage （） 方法。

   c. 当消息到达处理程序链的末尾时，会被分配。

对于单向消息交换会出现以下情况：

d. 消息处理将停止。

e. 所有之前调用的消息处理程序都具有其 close （） 方法调用。

f. 消息被分配。

3. 抛出一个 ProtocolException 异常-Throwing a ProtocolException 异常或这个例外的子类，向 Apache CXF 运行时发送错误消息处理。运行时的继续取决于用于 当前消息 的消息交换模式。

对于请求响应消息交换，会出现以下情况：

a. 如果处理程序尚未创建 fault 消息，则运行时会将消息嵌套在出错消息中。

b. 消息处理的方向被撤销。

例如，如果某个请求由服务提供商处理，消息将停止进入服务的实施对象。相反，它会发回到绑定，以返回到源自请求的消费者。

c. 在新处理方向中驻留在处理程序链的任何消息处理程序都按照它们位于链中的顺序调用其 handleFault （） 方法。

d. 当故障消息到达处理程序链的末尾时，它将被分配。

对于单向消息交换会出现以下情况：

e. 如果处理程序尚未创建 fault 消息，则运行时会将消息嵌套在出错消息中。

f. 消息处理将停止。

g. 所有之前调用的消息处理程序都具有其 close （） 方法调用。

h. 已分配 fault 消息。

4. 抛出任何其他运行时异常-浏览除 ProtocolException 异常之外的运行时异常，信号了消息处理要停止的 Apache CXF 运行时。所有之前调用的消息处理程序都有一个 close （） 方法，并分配异常。如果消息是请求响应消息交换的一部分，则会分配异常，使其返回到源自请求的消费者。

**Example**

例 43.7 “逻辑消息处理程序消息处理” 显示一个由服务消费者使用的逻辑消息处理程序的 handleMessage （） 消息实现。它会在请求发送到服务提供商之前处理请求。

**例 43.7. 逻辑消息处理程序消息处理**

```
public class SmallNumberHandler implements LogicalHandler<LogicalMessageContext>
{
   public final boolean handleMessage(LogicalMessageContext messageContext)
   {
      try
      {
         boolean outbound =
(Boolean)messageContext.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

         if (outbound)
         {
            LogicalMessage msg = messageContext.getMessage();

            JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
            Object payload = msg.getPayload(jaxbContext);
            if (payload instanceof JAXBElement)
            {
               payload = ((JAXBElement)payload).getValue();
            }

            if (payload instanceof AddNumbers)
            {
               AddNumbers req = (AddNumbers)payload;

               int a = req.getArg0();
               int b = req.getArg1();
               int answer = a + b;

               if (answer < 20)
               {
                 AddNumbersResponse resp = new AddNumbersResponse();
```

```
                    resp.setReturn(answer);
                    msg.setPayload(new ObjectFactory().createAddNumbersResponse(resp),
                                   jaxbContext);

                    return false;
                  }
                }
                else
                {
                  throw new WebServiceException("Bad Request");
                }
              }
              return true;
            }
            catch (JAXBException ex)
            {
              throw new ProtocolException(ex);
            }
          }
        ...
        }
```

例 43.7 “逻辑消息处理程序消息处理” 中的代码执行以下操作：

检查消息是否为出站请求。

如果消息是出站请求，处理程序会执行额外的消息处理。

从消息上下文获取消息有效负载的 **LogicalMessage** 表示。

获取实际的消息有效负载作为 **JAXB** 对象。

检查以确保请求是正确的类型。

如果是，处理程序将继续处理消息。

检查 **sum** 的值。

如果它小于 20 阈值，则它构建响应并返回到客户端。

构建响应。

返回 false 以停止消息处理，并将响应返回给客户端。

如果消息不是正确的类型，则抛出运行时异常。

此例外返回给客户端。

如果消息是入站响应或 sum 没有达到阈值，则返回 true。

消息处理通常继续。

如果遇到 JAXB marshalling 错误，则抛出 ProtocolException。

异常通过当前处理程序和客户端之间的处理程序的 handleFault （） 方法处理后传递给客户端。

## 43.4. 实施协议处理程序

### 概述

协议处理程序特定于使用的协议。Apache CXF 提供 JAX-WS 指定的 SOAP 协议处理程序。SOAP 协议处理程序实施 javax.xml.ws.handler.soap.SOAPHandler 接口。

在 例 43.8 “SOAPHandler Synopsis” 中显示的 SOAPHandler 接口使用 SOAP 特定的消息上下文，作为 SOAPMessage 对象提供对消息的访问。它还允许您访问 SOAP 标头。

**例 43.8. SOAPHandler Synopsis**

```
public interface SOAPHandler extends Handler
{
  boolean handleMessage(SOAPMessageContext context);
```

```
    boolean handleFault(SOAPMessageContext context);
    void close(SOAPMessageContext context);
    Set<QName> getHeaders()
  }
```

除了使用 SOAP 特定消息上下文外，SOAP 协议处理程序还需要实施名为 getHeaders （） 的额外方法。这个附加方法返回标头的 QNames 会阻止处理器可以处理。

**流程**

要实现逻辑手，请执行以下操作：

1. 实施处理程序所需的任何 第 43.6 节 “初始化处理程序” 逻辑。
2. 实施 第 43.5 节 “处理 SOAP 处理程序中的消息” 逻辑。
3. 实施 第 43.7 节 “处理故障消息” 逻辑。
4. 实施 getHeaders （） 方法。
5. 在完成时，实施 第 43.8 节 “关闭处理程序” 处理程序的逻辑。
6. 在销毁前，为 第 43.9 节 “释放处理程序” 处理程序的资源实施任何逻辑。

**实现 getHeaders （）方法**

getHeaders （） 显示在 例 43.9 “SOAPHander.getHeaders （） 方法” 所示，方法告知 Apache CXF 运行时处理器负责处理的 SOAP 标头。它返回每个 SOAP 标头的 outer 元素的 QNames，处理程序会理解。

**例 43.9. SOAPHander.getHeaders （） 方法**

```
set<QName&gt;getHeaders
```

对于许多情况下，只需返回 null 就足够了。但是，如果应用程序使用任何 SOAP 标头的 mustUnderstand 属性，则务必要指定应用的 SOAP 处理程序理解的标头。运行时检查一组 SOAP 标头，所有已注册的处理程序都对 mustUnderstand 属性设置为 true 的标头列表理解。如果任何标记的标头不在理解标头列表中，则运行时会拒绝消息，并抛出 SOAP 必须理解异常。

## 43.5. 处理 SOAP 处理程序中的消息

### 概述

正常消息处理由 handleMessage （） 方法处理。

handleMessage （） 方法接收 SOAPMessageContext 对象，以 SOAPMessage 对象和与消息关联的 SOAP 标头提供对消息的访问。此外，上下文提供对消息上下文中存储的任何属性的访问。

handleMessage （） 方法返回 true 或 false，具体取决于消息处理如何继续。它还可能会抛出异常。

### 使用消息正文

您可以使用 SOAP 消息上下文的 getMessage （） 方法获取 SOAP 消息。它将消息返回为 live SOAPMessage 对象。处理程序中消息的任何更改都会自动反映在上下文中存储的消息中。

如果要将现有消息替换为新消息，您可以使用上下文的 setMessage （） 方法。setMessage （） 方法采用 SOAPMessage 对象。

### 获取 SOAP 标头

您可以使用 SOAPMessage 对象的 getHeader （） 方法访问 SOAP 消息的标头。这将返回 SOAP 标头作为 SOAPHeader 对象，您需要检查以查找您要处理的标头元素。

SOAP 消息上下文提供 getHeaders （） 方法，如 例 43.10 “SOAPMessageContext.getHeaders （） 方法” 所示，它将返回一个包含指定 SOAP 标头的 JAXB 对象的数组。

**例 43.10. SOAPMessageContext.getHeaders （） 方法**

> Ojbect[]getHeadersQNameheaderJAXBContext上下文布尔值allRoles

您可以使用其元素的 QName 指定标头。您可以通过将 allRoles 参数设置为 false 来进一步限制返回的标头。这指示运行时仅返回适用于活动 SOAP 角色的 SOAP 标头。

如果没有找到标头，则方法会返回一个空数组。

有关实例化 JAXBContext 对象的更多信息，请参阅 第 39 章 *使用 A JAXBContext 对象*。

使用上下文属性

传递到逻辑处理程序的 SOAP 消息上下文是应用的消息上下文的实例，可以访问其中存储的所有属性。处理程序有权访问应用范围 和处理程序 范围的属性。

与应用程序的消息上下文一样，SOAP 消息上下文是 Java Map 的子类。要访问存储在上下文中的属性，您可以使用从 Map 接口继承的 get （） 方法和 put （） 方法。

默认情况下，您在逻辑处理程序内的上下文中设置的任何属性都会被分配 HANDLER 范围。如果您希望应用程序代码能够访问所需的属性，您需要使用上下文的 setScope （） 方法将属性明确设置为 APPLICATION。

有关在消息上下文中使用属性的详情，请参考 第 42.1 节 “了解上下文”。

确定消息的方向

知道消息通过处理程序链的方向通常非常重要。例如，您要将标头添加到传出消息，并从传入的消息中剥离标头。

消息的方向存储在消息上下文的出站消息属性中。您可以使用 MessageContext.MESSAGE_OUTBOUND_PROPERTY 密钥从消息上下文检索出站消息属性，如 例 43.11 “从 SOAP 消息上下文获取消息方向” 所示。

> 例 43.11. 从 SOAP 消息上下文获取消息方向

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

属性作为 布尔值对象存储。您可以使用对象的 booleanValue （） 方法来确定属性值。如果属性设为 true，则会出站消息。如果属性设为 false，则消息为 inbound。

确定返回值

handleMessage （） 方法如何完成其消息处理对消息处理的方式有直接影响。它可以通过执行以下操作之一完成：

1. 将 true-Returning true 信号返回到消息处理应该继续正常运行的 Apache CXF 运行时。下一个处理程序（若有）调用了其 handleMessage （）。

2. 将 false-Returning false 信号返回到正常消息处理要停止的 Apache CXF 运行时。运行时的继续取决于用于 当前消息 的消息交换模式。

   对于请求响应消息交换，会出现以下情况：

   a. 消息处理的方向被撤销。

      例如，如果某个请求由服务提供商处理，则该消息将停止进入服务的实施对象。相反，它将发回到该绑定，以返回到源自请求的消费者。

   b. 在新处理方向中驻留在处理程序链的任何消息处理程序都按照它们位于链中的顺序调用其 handleMessage （） 方法。

   c. 当消息到达处理程序链的末尾时，会被分配。

      对于单向消息交换会出现以下情况：

   d. 消息处理将停止。

e.
所有之前调用的消息处理程序都具有其 close （） 方法调用。

f.
消息被分配。

3.
抛出一个 ProtocolException 异常，或者这个例外的子类，信号 Apache CXF 运行时会启动错误消息处理。运行时的继续取决于用于 当前消息 的消息交换模式。

对于请求响应消息交换，会出现以下情况：

a.
如果处理程序尚未创建 fault 消息，则运行时会将消息嵌套在出错消息中。

b.
消息处理的方向被撤销。

例如，如果某个请求由服务提供商处理，则该消息将停止进入服务的实施对象。它将发回到该绑定，以返回到源自请求的消费者。

c.
在新处理方向中驻留在处理程序链的任何消息处理程序都按照它们位于链中的顺序调用其 handleFault （） 方法。

d.
当故障消息到达处理程序链的末尾时，它将被分配。

对于单向消息交换会出现以下情况：

e.
如果处理程序尚未创建 fault 消息，则运行时会将消息嵌套在出错消息中。

f.
消息处理将停止。

g.
所有之前调用的消息处理程序都具有其 close （） 方法调用。

h.
已分配 fault 消息。

4.

抛出任何其他运行时异常-浏览除 ProtocolException 异常之外的运行时异常，信号了消息处理要停止的 Apache CXF 运行时。所有之前调用的消息处理程序都有一个 close （） 方法，并分配异常。如果消息是请求响应消息交换的一部分，则会发送异常，使其返回到源自请求的消费者。

**Example**

例 43.12 “在 SOAP 处理程序中处理消息” 显示一个 handleMessage （） 实现，它将 SOAP 消息打印到屏幕。

**例 43.12. 在 SOAP 处理程序中处理消息**

```
public boolean handleMessage(SOAPMessageContext smc)
{
  PrintStream out;

  Boolean outbound =
(Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

  if (outbound.booleanValue())
  {
    out.println("\nOutbound message:");
  }
  else
  {
    out.println("\nInbound message:");
  }

  SOAPMessage message = smc.getMessage();

  message.writeTo(out);
  out.println();

  return true;
}
```

例 43.12 “在 SOAP 处理程序中处理消息” 中的代码执行以下操作：

从消息上下文检索 outbound 属性。

测试消息方向并打印适当的消息。

从上下文检索 SOAP 消息。

将消息输出到控制台。

## 43.6. 初始化处理程序

### 概述

当运行时创建处理程序的实例时，它会创建手器需要处理消息的所有资源。虽然您可以将执行此操作的所有逻辑放在处理程序的构造器中，但可能不是最合适的位置。处理程序框架在实例化处理程序时执行多个可选步骤。您可以添加在可选步骤中执行的资源注入和其他初始化逻辑。

您不必为处理程序提供任何初始化方法。

### 初始化顺序

Apache CXF 运行时以以下方式初始化处理器：

1. 处理程序的构造器称为.
2. 由 @Resource 注释指定的任何资源都会被注入。
3. 存在 @PostConstruct 注释的方法被调用。

**注意**

使用 @PostConstruct 注释解码的方法必须具有 void 返回类型，且没有参数。

4. 处理程序置于 Ready 状态。

## 43.7. 处理故障消息

### 概述

当消息处理过程中抛出 ProtocolException 异常时，处理程序使用 handleFault （） 方法来处理 fault 消息。

handleFault （） 方法根据处理程序的类型接收 LogicalMessageContext 对象或 SOAPMessageContext 对象。收到的上下文授予处理器对消息有效负载的实施访问权限。

handleFault （） 方法返回 true 或 false，具体取决于故障消息处理如何进行。它还可能会抛出异常。

### 获取消息有效负载

handleFault （） 方法接收的上下文对象与 handleMessage （） 方法接收的上下文对象类似。您可以使用上下文的 getMessage （） 方法以同样的方式访问消息有效负载。唯一的区别是上下文中所含的有效负载。

有关使用 LogicalMessageContext 的更多信息，请参阅 第 43.3 节 “处理逻辑处理程序中的消息”。

有关使用 SOAPMessageContext 的更多信息，请参阅 第 43.5 节 “处理 SOAP 处理程序中的消息”。

### 确定返回值

handleFault （） 方法如何完成其消息处理对消息处理的方式有直接影响。它通过执行以下操作之一完成：

#### 返回 true

返回错误处理的真正信号应该继续正常。将调用链中下一个处理程序的 handleFault （） 方法。

#### 返回错误

返回错误处理停止的假信号。在处理当前消息时调用的处理程序的 close （） 方法会被调用，并分配错误消息。

#### 抛出异常

引发异常会停止出错消息处理。在处理当前消息时调用的处理程序的 close （） 方法会被调用，并分配异常。

**Example**

例 43.13 “处理消息处理程序中的故障” 显示 **handleFault （）** 的实现，它将消息正文输出到屏幕。

**例 43.13.** 处理消息处理程序中的故障

```
public final boolean handleFault(LogicalMessageContext messageContext)
{
  System.out.println("handleFault() called with message:");

  LogicalMessage msg=messageContext.getMessage();
  System.out.println(msg.getPayload());

  return true;
}
```

### 43.8. 关闭处理程序

当处理器链完成处理消息时，运行时会调用每个执行的处理程序的 **close （）** 方法。这是在消息处理期间清理处理程序使用的任何资源或将任何属性重置为默认状态的适当位置。

如果资源需要保持在单一消息交换之外，您不应在处理器的 **close （）** 方法中清除它。

### 43.9. 释放处理程序

**概述**

当处理程序绑定到的服务或服务代理时，运行时会释放处理程序。在调用处理程序的structor 之前，运行时将调用一个可选的 **release** 方法。此可选发行版本方法可用于释放处理程序使用的任何资源，或者执行在处理器的structor 中不合适的其他操作。

您不必为处理程序提供任何清理方法。

发行顺序

当处理器被释放时，会出现以下情况：

1. 处理程序完成处理任何活动消息。

2. 运行时调用使用 @PreDestroy 注释划分的方法。

   此方法应清理处理程序使用的任何资源。

3. 处理程序的structor 被调用。

## 43.10. 配置端点以使用处理程序

### 43.10.1. 编程配置

#### 43.10.1.1. 将处理程序链添加到 Consumer

**概述**

向消费者添加处理程序链涉及显式构建处理程序链。然后，您将直接在服务代理的 Binding 对象上设置处理器链。

**重要**

使用 Spring 配置配置的任何处理器链都会覆盖处理器链配置的程序分析。

**流程**

要在消费者中添加处理器链，请执行以下操作：

1. 创建一个 List<Handler& gt; 对象来存放处理程序链。

2. 创建将添加到链的每个处理程序的实例。

3. 按照运行时调用的顺序，将每个实例化处理器对象添加到列表中。

4. 从服务代理获取 Binding 对象。

   Apache CXF 提供名为 org.apache.cxf.jaxws.binding.DefaultBindingImpl 的绑定接口的实现。

5. 使用 Binding 对象的 setHandlerChain （） 方法在代理上设置处理器链。

**Example**

例 43.14 “将处理程序链添加到 Consumer” 显示向消费者添加处理程序链的代码。

**例 43.14. 将处理程序链添加到 Consumer**

```
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.Handler;
import java.util.ArrayList;
import java.util.List;

import org.apache.cxf.jaxws.binding.DefaultBindingImpl;
...
SmallNumberHandler sh = new SmallNumberHandler();
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(sh);

DefaultBindingImpl binding = ((BindingProvider)proxy).getBinding();
binding.getBinding().setHandlerChain(handlerChain);
```

例 43.14 “将处理程序链添加到 Consumer” 中的代码执行以下操作：

实例化处理程序。

创建用于存放链的 List 对象。

将处理程序添加到链。

从代理获取 Binding 对象作为 DefaultBindingImpl 对象。

将 handler 链分配给代理的绑定。

#### 43.10.1.2. 将处理程序链添加到服务提供商

**概述**

您可以通过使用 @HandlerChain 注释拒绝 SEI 或实施类，向服务提供商添加处理程序链。该注释指向定义由服务提供商使用的处理程序链的元数据数据文件。

**流程**

要在服务供应商中添加处理器链，您可以执行以下操作：

1. 使用 @HandlerChain 注释减少提供程序的实施类。
2. 创建定义处理程序链的处理程序配置文件。

**@HandlerChain 注释**

javax.jws.HandlerChain 注解解封服务提供商的实施类。它指示运行时加载由 file 属性指定的处理程序链配置文件。

该注解的 file 属性支持两种方法来识别要加载的处理器配置文件：

- a URL
- 相对路径名称

例 43.15 “加载处理程序链的服务实现” 显示一个服务提供商实施，它将使用名为 handlers.xml 的文件中定义的处理程序链。handlers.xml 必须位于运行服务提供商的目录中。

**例 43.15. 加载处理程序链的服务实现**

```
import javax.jws.HandlerChain;
import javax.jws.WebService;
```

```
...

@WebService(name = "AddNumbers",
            targetNamespace = "http://apache.org/handlers",
            portName = "AddNumbersPort",
            endpointInterface = "org.apache.handlers.AddNumbers",
            serviceName = "AddNumbersService")
@HandlerChain(file = "handlers.xml")
public class AddNumbersImpl implements AddNumbers
{
...
}
```

处理程序配置文件

处理程序配置文件使用 XML grammar 定义处理器链，该链是 JSR 109 （用于 Java EE 版本 1.2 的 Web 服务）。这个 grammar 在 http://java.sun.com/xml/ns/javaee 中定义。

处理程序配置文件的 root 元素是 handler-chains 元素。handler-chains 元素具有一个或多个 handler-chain 元素。

handler-chain 元素定义处理程序链。表 43.1 “用于定义服务器端处理程序链的元素” 描述 handler-chain 元素的子项。

表 43.1. 用于定义服务器端处理程序链的元素

| 元素 | 描述 |
| --- | --- |
| **handler** | 包含描述处理程序的元素。 |
| **service-name-pattern** | 指定 WSDL **服务** 元素的 QName，定义处理程序链绑定到的服务。在定义 QName 时，您可以使用 * 作为通配符。 |
| **port-name-pattern** | 指定 WSDL **端口** 元素的 QName，定义处理程序链绑定到的端点。在定义 QName 时，您可以使用 * 作为通配符。 |
| **protocol-binding** | 指定使用处理器链的消息绑定。该绑定被指定为 URI 或使用以下别名之一：{\#SOAP11_HTTP，**\##SOAP11_HTTP _MTOM ,\## #SOAP12_HTTP**,**\##SOAP12_HTTP_MTOM**, 或 **\#\#XML_HTTP**。<br><br>有关消息绑定 URI 的更多信息，请参阅 第 23 章 *Apache CXF 绑定 ID*。 |

handler-chain 元素只需要将单个 handler 元素作为子元素。但是，它可以根据需要支持定义完整的处理程序 链所需的处理程序元素。链中的处理程序按照处理程序链定义中指定的顺序执行。

重要

执行的最终顺序将通过将指定的处理程序排序到逻辑处理程序和协议处理程序来确定。在分组中，将使用配置中指定的顺序。

其他子级（如 protocol-binding ）用于限制定义的处理器链的范围。例如，如果您使用 service-name-pattern 元素，处理程序链将仅附加到其 WSDL 端口 元素为指定 WSDL 服务 元素的子项。您只能在 handler 元素中使用其中一个限制的子项。

handler 元素在处理程序链中定义一个处理程序。其 handler-class 子元素指定实施处理程序的类的完全限定域名。handler 元素也可以具有可选的 handler-name 元素，用于指定处理程序的唯一名称。

例 43.16 “处理程序配置文件” 显示定义单一处理程序链的处理程序配置文件。链由两个处理程序组成。

**例 43.16. 处理程序配置文件**

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
      <handler-name>LoggingHandler</handler-name>
      <handler-class>demo.handlers.common.LoggingHandler</handler-class>
    </handler>
    <handler>
      <handler-name>AddHeaderHandler</handler-name>
      <handler-class>demo.handlers.common.AddHeaderHandler</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

### 43.10.2. Spring 配置

#### 概述

将端点配置为使用处理程序链的最简单方法是在端点配置中定义链。这可以通过向配置端点的元素中添加 jaxwxs:handlers 子级来实现。

重要

通过配置文件添加的处理程序链优先于以编程方式配置的处理程序链。

流程

要配置端点来加载处理器链，您可以执行以下操作：

1. 如果端点还没有配置元素，请添加一个。

   有关配置 Apache CXF 端点的详情，请参考 第 17 章 *配置 JAX-WS 端点*。

2. 将 jaxws:handlers 子元素添加到端点的配置元素。

3. 对于链中的每一处理程序，添加一个 bean 元素，指定实施该处理程序的类。

   如果您的处理器实施在多个位置中使用，您可以使用 ref 元素引用 bean 元素。

handlers 元素

jaxws:handlers 元素在端点配置中定义处理程序链。它可以显示为所有 JAX-WS 端点配置元素的子级。这些是：

- jaxws:endpoint 配置服务提供商。
- jaxws:server 也配置服务提供商。
- jaxws:client 配置服务消费者。

您可以通过以下两种方式之一为处理器链添加处理程序：

- 添加一个定义实现类的 bean 元素
- 使用 ref 元素引用配置文件中其他位置的命名 bean 元素

处理程序在配置中定义的顺序是执行它们的顺序。如果您混合了逻辑处理程序和协议处理程序，可以修改顺序。运行时时间将它们排序为正确的顺序，同时维护配置中指定的基本顺序。

### Example

例 43.17 “配置端点以在 Spring 中使用处理程序链” 显示加载处理程序链的服务提供商的配置。

**例 43.17. 配置端点以在 Spring 中使用处理程序链**

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="HandlerExample"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo">
    <jaxws:handlers> <bean class="demo.handlers.common.LoggingHandler" /> <bean
class="demo.handlers.common.AddHeaderHandler" /> </jaxws:handlers>
  </jaws:endpoint>
</beans>
```

# 第 44 章 MAVEN 工具参考

## 44.1. 插件设置

**摘要**

在使用 Apache CXF 插件前，您必须首先将正确的依赖项和存储库添加到 POM 中。

**依赖项**

您需要在项目的 POM 中添加以下依赖项：

- JAX-WS 前端

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-frontend-jaxws</artifactId>
  <version>version</version>
</dependency>
```

- HTTP 传输

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http</artifactId>
  <version>version</version>
</dependency>
```

- Undertow 传输

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-undertow</artifactId>
  <version>version</version>
</dependency>
```

## 44.2. CXF-CODEGEN-PLUGIN

**摘要**

从 WSDL 文档生成符合 JAX-WS 的 Java 代码

### 概述

### 基本示例

以下 POM 提取显示如何配置 Maven cxf-codegen-plugin 来处理 myService.wsdl WSDL 文件的简单示例：

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>3.3.6.fuse-7_13_0-00015-redhat-00001</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>target/generated/src/main/java</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>src/main/resources/wsdl/myService.wsdl</wsdl>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

### 基本配置设置

在前面的示例中，您可以自定义以下配置设置

**configuration/sourceRoot**

指定存储生成的 Java 文件的目录。默认为 target/generated-sources/cxf。

**configuration/wsdlOptions/wsdlOption/wsdl**

指定 WSDL 文件的位置。

### 描述

wsdl2java 任务采用 WSDL 文档，并生成从实施服务的完全注释 Java 代码。WSDL 文档必须具有有效的 portType 元素，但不需要包含 绑定 元素或 service 元素。您可以使用可选参数自定义生成的代码。

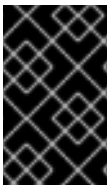
WSDL 选项

至少需要一个 wsdlOptions 元素来配置插件。wsdlOptions 元素的 wsdl 子级是必需的，并指定插件要处理的 WSDL 文档。除了 wsdl 元素外，wsdlOptions 元素还可以提取多个子对象，它可以自定义 WSDL 文档的处理方式。

可以在插件配置中列出多个 wsdlOptions 元素。每个元素配置单个 WSDL 文档进行处理。

默认选项

defaultOptions 元素是一个可选元素。它可用于设置可在所有指定的 WSDL 文档中使用的选项。



重要

如果在 wsdlOptions 元素中重复某个选项，wsdlOptions 元素中的值将采用前缀。

指定代码生成选项

要指定通用代码生成选项（与 Apache CXF wsdl2java 命令行工具支持的交换机相对应），您可以添加 extraargs 元素作为 wsdlOption 元素的子部分。例如，您可以添加 -impl 选项和 -verbose 选项，如下所示：

```
...
<configuration>
  <sourceRoot>target/generated/src/main/java</sourceRoot>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <!-- you can set the options of wsdl2java command by using the <extraargs> -->
      <extraargs>
        <extraarg>-impl</extraarg>
        <extraarg>-verbose</extraarg>
      </extraargs>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

如果交换机使用参数，您可以使用后续 **extraarg** 元素指定这些参数。例如，要指定 **jibx** 数据绑定，您可以配置插件，如下所示：

```
...
<configuration>
  <sourceRoot>target/generated/src/main/java</sourceRoot>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <extraargs>
        <extraarg>-databinding</extraarg>
        <extraarg>jibx</extraarg>
      </extraargs>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

**指定绑定文件**

要指定一个或多个 **JAX-WS** 绑定文件的位置，您可以将 **bindingFiles** 元素添加为 **wsdlOption**的子 - 例如：

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <bindingFiles>
        <bindingFile>${basedir}/src/main/resources/wsdl/async_binding.xml</bindingFile>
      </bindingFiles>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

**为特定 WSDL 服务生成代码**

要指定要为其生成代码的 **WSDL** 服务的名称，您可以将 **serviceName** 元素添加为 **wsdlOption** 的子级（默认为 **WSDL** 文档中的每个服务生成代码）-例如：

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <serviceName>MyWSDLService</serviceName>
    </wsdlOption>
```

```
  </wsdlOptions>
</configuration>
...
```

为多个 WSDL 文件生成代码

要为多个 WSDL 文件生成代码，只需为 WSDL 文件插入额外的 wsdlOption 元素。如果要指定适用于所有 WSDL 文件的一些常用选项，请将常用选项放在 defaultOptions 元素中，如下所示：

```
<configuration>
  <defaultOptions>
    <bindingFiles>
      <bindingFile>${basedir}/src/main/jaxb/bindings.xml</bindingFile>
    </bindingFiles>
    <noAddressBinding>true</noAddressBinding>
  </defaultOptions>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <serviceName>MyWSDLService</serviceName>
    </wsdlOption>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myOtherService.wsdl</wsdl>
      <serviceName>MyOtherWSDLService</serviceName>
    </wsdlOption>
  </wsdlOptions>
</configuration>
```

也可以使用通配符匹配指定多个 WSDL 文件。在这种情况下，使用 wsdlRoot 元素指定包含 WSDL 文件的目录，然后使用 include 元素选择所需的 WSDL 文件，它支持使用 * 字符通配符。例如，要从 src/main/resources/wsdl 根目录中选择以 Service.wsdl 结尾的所有 WSDL 文件，您可以配置插件，如下所示：

```
<configuration>
  <defaultOptions>
    <bindingFiles>
      <bindingFile>${basedir}/src/main/jaxb/bindings.xml</bindingFile>
    </bindingFiles>
    <noAddressBinding>true</noAddressBinding>
  </defaultOptions>
  <wsdlRoot>${basedir}/src/main/resources/wsdl</wsdlRoot>
  <includes>
    <include>*Service.wsdl</include>
  </includes>
</configuration>
```

从 Maven 存储库下载 WSDL

要直接从 Maven 存储库下载 WSDL 文件，请将 wsdlArtifact 元素添加为 wsdlOption 元素的子项，

并指定 Maven 工件的协调，如下所示：

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdlArtifact>
        <groupId>org.apache.pizza</groupId>
        <artifactId>PizzaService</artifactId>
        <version>1.0.0</version>
      </wsdlArtifact>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

编码

（需要 JAXB 2.2） 指定用于生成的 Java 文件的字符编码(Charset)，请添加 encoding 元素作为 配置元素 的子级，如下所示：

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
    </wsdlOption>
  </wsdlOptions>
  <encoding>UTF-8</encoding>
</configuration>
...
```

对独立进程进行分叉

您可以通过将 fork 元素添加为配置元素的子级，将 codegen 插件配置为 分叉 单独的 JVM 进行代码生成。fork 元素可以设置为以下值之一：

**once**

对单个新 JVM 进行分叉，以处理 codegen 插件配置中指定的所有 WSDL 文件。

**always**

分叉一个新的 JVM，以处理 codegen 插件配置中指定的每个 WSDL 文件。

**false**

（默认） 禁用分叉。

如果 codegen 插件配置为 fork 独立的 JVM （即，fork 选项被设置为非错误值），您可以通过 additionalJvmArgs 元素为 forked JVM 指定额外的 JVM 参数。例如，以下片段将 codegen 插件配置为 fork 单个 JVM，该 JVM 仅限于从本地文件系统访问 XML 模式（通过设置 javax.xml.accessExternalSchema 系统属性）：

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
    </wsdlOption>
  </wsdlOptions>
  <fork>once</fork>
  <additionalJvmArgs>-Djavax.xml.accessExternalSchema=jar:file,file</additionalJvmArgs>
</configuration>
...
```

选项参考

下表中列出了用于管理代码生成过程的选项。

| 选项 | 解释 |
| --- | --- |
| **-fe\|-frontend** ***frontend*** | 指定代码生成器使用的前端。可能的值有 **jaxws**、**jaxws21** 和 **cxf**。**jaxws21** 前端用于生成 JAX-WS 2.1 兼容代码。**cxf** frontend （可选）可以选择性地使用，而不是 **jaxws** frontend，为服务类提供额外的构造器。这种构造器可让您方便地指定用来配置该服务的总线实例。默认为 **jaxws**。 |
| **-db\|-databinding** ***databinding*** | 指定代码生成器使用的数据绑定。可能的值有：**jaxb**,**xmlbeans**,**sdo** (**sdo-static** 和 **sdo-dynamic**)和 **jibx**。默认为 **jaxb**。 |
| **-wv** ***wsdlVersion*** | 指定工具预期的 WSDL 版本。默认为 **1.1**。[a] |
| **-p** *wsdlNamespace=PackageName* | 指定用于生成的代码的零个或多个软件包名称。（可选）将 WSDL 命名空间指定到软件包名称映射。 |
| **-b** ***bindingName*** | 指定一个或多个 JAXWS 或 JAXB 绑定文件。每个绑定文件都使用单独的 **-b** 标志。 |
| **-sn** ***serviceName*** | 指定要为其生成代码的 WSDL 服务的名称。默认值是为 WSDL 文档中的每个服务生成代码。 |

| 选项 | 解释 |
| --- | --- |
| **-reserveClass *classname*** | 与 **-autoNameResolution** 一起使用，定义在生成类时不使用 wsdl-to-java 的类名称。**多次将这个选项用于多个类。** |
| **-catalog *catalogUrl*** | 指定用于解析导入的模式和 WSDL 文档的 XML 目录的 URL。 |
| **-d *output-directory*** | 指定生成的代码文件写入的目录。 |
| **-compile** | 编译生成的 Java 文件。 |
| **-classdir *complile-class-dir*** | 指定编译的类文件要写入的目录。 |
| **-clientjar *jar-file-name*** | 生成包含所有客户端类和 WSDL 的 JAR 文件。当指定了这个选项时，指定的 **wsdlLocation** 无法正常工作。 |
| **-client** | 为客户端主行生成起点代码。 |
| **-server** | 为服务器主线生成起点代码。 |
| **-impl** | 为实现对象生成起点代码。 |
| **-all** | 生成所有起点代码：类型、服务代理、服务接口、服务器主线、客户端主线、实施对象和 Ant **build.xml** 文件。 |
| **-ant** | 生成 Ant **build.xml** 文件。 |
| **-autoNameResolution** | 自动解决命名冲突，无需使用绑定自定义。 |
| **-defaultValues**=*DefaultValueProvider* | 指示工具为生成的客户端和生成的实施生成默认值。另外，您还可以提供用于生成默认值的类名称。默认情况下，使用 **RandomValueProvider** 类。 |
| **-nexclude** *schema-namespace=java-packagename* | 在生成代码时忽略指定的 WSDL 模式命名空间。这个选项可多次指定。另外，还可指定排除命名空间中描述的类型所使用的 Java 软件包名称。 |
| **-exsh** (**true**/**false**) | 启用或禁用扩展 soap 标头消息绑定的处理。默认值为 false。 |
| **-noTypes** | 关闭生成类型。 |
| **-DNS** (true/false) | 启用或禁用默认命名空间软件包名称映射的加载。默认为 true。 |

| 选项 | 解释 |
| --- | --- |
| **-Dex** (true/false) | 启用或禁用默认排除命名空间映射的加载。默认为 true。 |
| **-xjc*args*** | 指定使用 JAXB 数据绑定时要直接传递给 XJC 的以逗号分隔的参数列表。要获得所有可能的 XJC 参数的列表，请使用 **-xjc-X**。 |
| **-noAddressBinding** | 指示工具使用 Apache CXF 专有 WS-Addressing 类型，而不是 JAX-WS 2.1 兼容映射。 |
| **-validate [=all\|basic\|none]** | 指示工具在尝试生成任何代码之前验证 WSDL 文档。 |
| **-keep** | 指示工具不会覆盖任何现有的文件。 |
| **-wsdlLocation *wsdlLocation*** | 指定 **@WebService** 注释的 **wsdlLocation** 属性的值。 |
| **-v** | 显示工具的版本号。 |
| **-verbose\|-V** | 在代码生成过程中显示注释。 |
| **-quiet** | 在代码生成过程中抑制注释。 |
| **-allowElementReferences[=true]**, **-aer[=true]** | 如果为 **true**，在 JAX-WS 2.2 规范的第 2.3.1.2 (v)中禁止使用打包程序映射时给出的规则引用。默认为 **false**。 |
| **-asyncMethods[=*method1*,*method2*,...]** | 随后生成的 Java 类方法列表以允许客户端异步调用；类似于 JAX-WS 绑定文件中的 **enableAsyncMapping**。 |
| **-bareMethods[=*method1*,*method2*,...]** | 随后生成的 Java 类方法列表具有打包程序风格（请参阅以下），类似于 JAX-WS 绑定文件中的 **enableWrapperStyle**。 |
| **-mimeMethods[=*method1*,*method2*,...]** | 随后生成的 Java 类方法列表以启用 mime:content 映射，类似于 JAX-WS 绑定文件中的 **enableMIMEContent**。 |
| **-faultSerialVersionUID *fault-serialVersionUID*** | 如何生成错误异常。可能的值有：**NONE**、**TIMESTAMP**、**FQCN** 或特定数字。默认值为 **NONE**。 |
| **- *encoding*编码** | 指定生成 Java 代码时要使用的 Charset 编码。 |
| **-exceptionSuper** | 从 **wsdl:fault** 元素生成的 fault Bean 的 superclass（默认为 **java.lang.Exception**）。 |

| 选项 | 解释 |
| --- | --- |
| **-seiSuper *interfaceName*** | 为生成的 SEI 接口指定一个基本接口。例如，此选项可用于将 Java 7 **AutoCloseable** 接口添加为超级接口。 |
| **-mark-generated** | 将 @Generated 注释添加到生成的类。 |
| [a] 目前，Apache CXF 仅为代码生成器提供 WSDL 1.1 支持。 | |

### 44.3. JAVA2WS

**摘要**

**从 Java 代码生成 WSDL 文档**

**概要**

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>version</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <option>...</option>
         ...
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

**描述**

**java2ws 任务采用服务端点实施(SEI)，并生成用于实施 Web 服务的支持文件。它可以生成以下内容：**

- **WSDL 文档**

- 将服务部署为 POJO 所需的服务器代码
- 用于访问服务的客户端代码
- wrapper 和 fault Bean

所需的配置

该插件要求存在 **classNam**e 配置元素。元素的值是要处理的 **SEI** 的完全限定名称。

可选配置

下表中列出的配置元素可用于微调 **WSDL** 生成。

| 元素 | 描述 |
| --- | --- |
| **frontend** | 指定用于处理 SEI 并生成支持类的前端。**jaxws** 是默认值。还支持 **simple**。 |
| **dataBinding** | 指定用于处理 SEI 并生成支持类的数据绑定。使用 JAX-WS 前端时的默认设置是 **jaxb**。使用简单前端时的默认设置是 **aegis**。 |
| **genWsdl** | 指示工具在设置为 **true** 时生成 WSDL 文档。 |
| **genWrapperbean** | 指示工具在设置为 **true** 时生成 wrapper bean 和 fault Bean。 |
| **genClient** | 指示工具在设置为 **true** 时生成客户端代码。 |
| **genServer** | 指示工具在设置为 **true** 时生成服务器代码。 |
| **outputFile** | 指定生成的 WSDL 文件的名称。 |
| **classpath** | 指定处理 SEI 时搜索的类路径。 |
| **soap12** | 指定生成的 WSDL 文档是在设置为 **true** 时包含 SOAP 1.2 绑定。 |
| **targetNamespace** | 指定要在生成的 WSDL 文件中使用的目标命名空间。 |

| 元素 | 描述 |
| --- | --- |
| **serviceName** | 指定生成的 **service** 元素的 **name** 属性的值。 |

# 部分 VI. 开发 RESTFUL WEB 服务

本指南介绍了如何使用 JAX-RS API 实施 Web 服务。

# 第 45 章 RESTFUL WEB 服务简介

## 摘要

Representational State Transfer (REST)是一个软件架构，它围绕通过 HTTP 传输数据的数据中心，仅使用四个基本 HTTP 动词。它还使用任何额外的打包程序，如 SOAP 信封以及使用任何状态数据。

## 概述

*Representational State Transfer* (REST)是一个架构样式，首先在名为 Roy Fielding 的研究人员中描述。在 RESTful 系统中，服务器使用 URI 来公开资源，客户端使用四个 HTTP 动词来访问这些资源。当客户端收到它们处于状态的资源表示时。当访问新资源时，通常遵循链接、更改或转换其状态。为了工作，REST 假定资源能够使用普遍标准 grammar 表示。

全球 Web 是设计于 REST 原则的系统的最广泛示例。Web 浏览器充当访问 Web 服务器上托管的资源的客户端。资源使用 HTML 或 XML 图表来表示，所有 Web 浏览器都可以使用它们。浏览器还可以轻松遵循指向新资源的链接。

RESTful 系统的优点在于它们具有高度可扩展且高度灵活性。由于资源使用四个 HTTP 动词访问和操作，所以这些资源使用 URI 来公开，并且资源使用标准的 grammars 表示，所以客户端不会受服务器更改的影响。此外，RESTful 系统还可以充分利用 HTTP 的可扩展性功能，如缓存和代理。

## 基本 REST 原则

RESTful 架构遵循以下基本原则：

- 应用程序状态和功能被分成多个资源。
- 资源可以使用标准 URI 来寻址，它们可用作超媒体链接。
- 所有资源仅使用四个 HTTP 动词。
  - DELETE

- GET
- POST
- PUT

- 所有资源使用 HTTP 支持的 MIME 类型提供信息。
- 协议是无状态的。
- 响应是可缓存的。
- 协议是层次的。

### RESOURCES

*资源是* REST 的核心。资源是可以使用 URI 解决的信息源。在 Web 早期的早期，资源是大型静态文档。在现代 Web 中，资源可以是任何信息来源。例如，如果可通过 URI 访问它，Web 服务可以是资源。

RESTful 端点交换它们地址的资源的 *表示*。表示是一个文档，其中包含资源提供的数据。例如，提供客户记录的 Web 服务的方法将是资源，在服务和消费者之间交换的客户记录副本是资源的表示。

### REST 最佳实践

在设计 RESTful Web 服务时，请注意以下几点：

- 为您要公开的每个资源提供不同的 URI。

  例如，如果您要构建处理驱动记录的系统，每个记录都应该有一个唯一的 URI。如果系统也提供了关于 parking violations 和 speeding fines 的信息，每种类型的资源还应具有唯一的基础。例如，可以通过 /speedingfines/*driverID* 和 parking violations 访问速度较差，可以通过 /parkingfines/*driverID* 访问。

- 在 URI 中使用 nouns。

  使用 nounss 突出显示资源是什么，而不是操作。诸如 /ordering 等 URI 代表一个操作，而 /orders 则代表一个操作。

- 映射到 GET 的方法不应更改任何数据。

- 在您的响应中使用链接。

  在响应中放入其他资源的链接后，客户端可以更轻松地遵循数据链。例如，如果您的服务返回资源集合，客户端可以使用提供的链接访问每个独立资源。如果没有包括链接，客户端需要具有额外的逻辑才能遵循到特定节点的链。

- 使您的服务无状态。

  要求客户端或服务维护状态信息，强制两者间的紧密耦合。紧密耦合使升级和迁移变得更加困难。维护状态也可以使从通信错误恢复变得更加困难。

## 设计 RESTFUL WEB 服务

无论您用来实现 RESTful Web 服务的框架是什么，都应遵循几个步骤：

1. 定义服务将公开的资源。

   通常，服务会公开一个或多个以树形组织的资源。例如，驱动记录服务可以分为三个资源：

   - /license/*driverID*
   - /license/*driverID*/speedingfines
   - /license/*driverID*/parkingfines

2. 定义您要对每个资源执行的操作。

   例如，您可能希望更新不同地址或从驱动程序的记录中删除解析票据。

3. 将操作映射到适当的 HTTP 动词。

定义该服务后，您可以使用 Apache CXF 实现该服务。

### 使用 APACHE CXF 实施 REST

Apache CXF 为 *RESTFul Web 服务(JAX-RS)提供了 Java API*实施。JAX-RS 提供了使用注释将 POJO 映射到资源的标准化方法。

从抽象服务定义移动到使用 JAX-RS 实施的 RESTful Web 服务时，您需要执行以下操作：

1. 为代表服务资源树顶部的资源创建根资源类。

   请参阅 第 46.3 节 “根资源类”。

2. 将服务的其他资源映射到子资源。

   请参阅 第 46.5 节 “使用子资源”。

3. 创建方法来实现每个资源所使用的每个 HTTP 动词。

   请参阅 第 46.4 节 “使用资源方法”。

**注意**

Apache CXF 继续支持旧的 HTTP 绑定，将 Java 接口映射到 RESTful Web 服务。HTTP 绑定提供基本功能，并有许多限制。建议开发人员更新其应用以使用 JAX-RS。

#### 数据绑定

默认情况下，Apache CXF 将 Java 架构用于 XML Binding (JAXB)对象，将资源及其表示映射到 Java 对象。提供 Java 对象和 XML 元素之间的干净定义映射。

Apache CXF JAX-RS 实施还支持使用 *JavaScript 对象表示法* (JSON)交换数据。JSON 是一种流行数据格式，供 iwl 开发人员使用。JSON 和 JAXB 之间的数据处理由 Apache CXF 运行时处理。

# 第 46 章 创建资源

**摘要**

在 RESTful Web 服务中，所有请求都由资源处理。JAX-RS API 将资源实施为 Java 类。资源类是用一个或多个 JAX-RS 注释标注的 Java 类。使用 JAX-RS 实施的 RESTful Web 服务的核心是根资源类。root 资源类是指向由服务公开的资源树的入口点。它可以处理所有请求本身，或者它可能会提供对处理请求的子资源的访问。

## 46.1. 简介

**概述**

使用 JAX-RS API 实施的 RESTful Web 服务将响应作为 Java 类实施的资源的表示形式提供。*资源类*是使用 JAX-RS 注释来实施资源的类。对于大多数 RESTful Web 服务，需要访问的资源集合。资源类的注解提供资源 URI 以及每个操作处理的 HTTP 动词等信息。

资源类型

JAX-RS API 允许您创建两个基本类型的资源：

- 第 46.3 节 “根资源类” 是服务的资源树的入口点。它使用 @Path 注释进行解码，以定义服务中资源的基本 URI。
- 第 46.5 节 “使用子资源” 通过 root 资源访问。它们通过利用 @Path 注释进行解码的方法来实施。子资源的 @Path 注释定义了相对于根资源基本 URI 的 URI。

**Example**

例 46.1 “简单资源类” 显示简单的资源类。

**例 46.1. 简单资源类**

```
package demo.jaxrs.server;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;

@Path("/customerservice")
```

```
public class CustomerService
{
  public CustomerService()
  {
  }

  @GET
  public Customer getCustomer(@QueryParam("id") String id)
  {
    ...
  }

  ...
}
```

两个项目使 例 46.1 “简单资源类” 中定义的类成为资源类：

@Path 注释指定资源的基本 URI。

@GET 注释指定方法为资源实施 HTTP GET 方法。

## 46.2. 基本 JAX-RS 注释

### 概述

RESTful Web 服务实现所需的最基本信息有：

- 服务资源的 URI
- 类的方法如何映射到 HTTP 动词

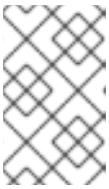
JAX-RS 定义一组提供此基本信息的注释。所有资源类必须至少有一个注解。

### 设置路径

@Path 注释指定资源的 URI。该注释由 javax.ws.rs.Path 接口定义，可用于分离资源类或资源方法。它接受字符串值作为其唯一参数。字符串值是一个 URI 模板，用于指定实施资源的位置。

URI 模板指定资源的相对位置。如 例 46.2 “URI 模板语法” 所示，模板可包含以下内容：

- 未处理的路径组件
- { }周围的参数标识符



注意

参数标识符可以包含正则表达式，以更改默认路径处理。

**例 46.2. URI 模板语法**

```
@Path("resourceName/{param1}/../{paramN}")
```

例如，URI 模板 widgets/{color}/{number} 将映射到 widgets/blue/12。color 参数的值分配给 蓝色。number 参数的值被分配 12。

URI 模板如何映射到完整的 URI，这取决于 @Path 注释的下降。如果它放在根资源类上，URI 模板是树中所有资源的根 URI，它将直接附加到发布该服务的 URI。如果注解解封子资源，它将相对于根资源 URI。

### 指定 HTTP 动词

JAX-RS 使用五个注释来指定将用于方法的 HTTP 动词：

- javax.ws.rs.DELETE 指定方法映射到 DELETE。
- javax.ws.rs.GET 指定方法映射到 GET。
- javax.ws.rs.POST 指定方法映射到 POST。

- javax.ws.rs.PUT 指定方法映射到 PUT。

- javax.ws.rs.HEAD 指定方法映射到 HEAD。

将方法映射到 HTTP 动词时，您必须确保映射有意义。例如，如果您映射要提交购买顺序的方法，您可以将它映射到 PUT 或 POST。将其映射到 GET 或 DELETE 会导致行为无法预计。

## 46.3. 根资源类

### 概述

root 资源类是实施 RESTful Web 服务的 JAX-RS 的入口点。它被解码为 @Path，用于指定服务实施的资源的根 URI。其方法可直接对资源实施操作，或提供对子资源的访问。

### 要求

要让类是根资源类，它必须满足以下条件：

- 类必须使用 @Path 注释进行解码。

  指定路径是服务实施的所有资源的根 URI。如果 root 资源类指定其路径是 小部件，其方法之一实施 GET 动词，则 小部件 上的 GET 会调用该方法。如果子资源指定其 URI 为 {id}，则子资源的完整 URI 模板是 widgets/{id}，它将处理对 URI （如 widgets/12 和 widgets/42 ）发出的请求。

- 类必须具有公共构造器，才能调用运行时。

  运行时必须能够为所有构造器的参数提供值。constructor 的参数可以包含使用 JAX-RS 参数注释的参数。有关参数注解的更多信息，请参阅 第 47 章 *将信息传递给资源类和方法*。

- 至少一个类方法使用 HTTP 动词注释或 @Path 注释进行解码。

### Example

例 46.3 “根资源类” 显示提供子资源访问权限的 root 资源类。

**例 46.3. 根资源类**

```
package demo.jaxrs.server;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

@Path("/customerservice/")
public class CustomerService
{
  public CustomerService()
  {
    ...
  }

  @GET
  public Customer getCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @DELETE
  public Response deleteCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @PUT
  public Response updateCustomer(Customer customer)
  {
    ...
  }

  @POST
  public Response addCustomer(Customer customer)
  {
    ...
  }

  @Path("/orders/{orderId}/")
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

}
```

例 46.3 “根资源类” 中的类满足 root 资源类的所有要求。

类使用 @Path 注释进行解码。服务公开的资源的 root URI 是 customerservice。

类具有公共构造器。在这种情况下，no 参数构造器用于简单。

类为资源实施四个 HTTP 动词。

该类还通过 getOrder （） 方法提供对子资源的访问。子资源的 URI （使用 @Path 注释指定）是 customerservice/order/*id*。子资源由 Order 类实现。

有关实现子资源的详情，请参考 第 46.5 节 “使用子资源”。

### 46.4. 使用资源方法

#### 概述

资源方法使用 JAX-RS 注释进行注释。它们有一个 HTTP 方法注解，指定方法进程的请求类型。JAX-RS 在资源方法上放置多个约束。

#### 常规限制

所有资源方法都必须满足以下条件：

- 它必须是公共。
- 它必须使用 “指定 HTTP 动词”一节 中描述的 HTTP 方法注解之一进行解码。
- 它不能有多个实体参数，如 “参数”一节 所述。

参数

资源方法参数采用两种形式：

- 实体参数- 未注解参数。其值从请求实体正文映射。entity 参数可以是您的应用程序具有实体供应商的任何类型的。它们通常是 JAXB 对象。

重要

资源方法 只能有一个 实体参数。

有关实体供应商的更多信息，请参阅 第 51 章 *实体支持*。

- 注释的参数-nnotated 参数使用其中一个 JAX-RS 注释来指定从请求中映射参数的值。通常，参数的值从请求 URI 的部分映射。

有关使用 JAX-RS 注释将请求数据映射到方法参数的更多信息，请参阅 第 47 章 *将信息传递给资源类和方法*。

例 46.4 “带有有效参数列表的资源方法” 显示具有有效参数列表的资源方法。

例 46.4. 带有有效参数列表的资源方法

```
@POST
@Path("disaster/monster/giant/{id}")
public void addDaikaiju(Kaiju kaiju,
                        @PathParam("id") String id)
{
  ...
}
```

例 46.5 “带有无效参数列表的资源方法” 显示带有无效参数列表的资源方法。它有两个没有注解的参数。

例 46.5. 带有无效参数列表的资源方法

```
@POST
@Path("disaster/monster/giant/")
```

```
public void addDaikaiju(Kaiju kaiju,
                        String id)
{
  ...
}
```

返回值

资源方法可以返回以下之一：

- void
- 应用具有实体提供程序的任何 Java 类

  有关实体供应商的更多信息，请参阅 第 51 章 *实体支持*。
- Response 对象

  有关 Response 对象的更多信息，请参阅 第 48.3 节 “微调应用程序的响应”。
- A GenericEntity<*T*> object

  有关 GenericEntity<*T*> 对象的更多信息，请参阅 第 48.4 节 “使用通用类型信息返回实体”。

所有资源方法都向请求者返回 HTTP 状态代码。当方法返回类型为 void 或返回的值为 null 时，资源方法会将 HTTP 状态代码设置为 204。当资源方法返回 null 以外的任何值时，它会将 HTTP 状态代码设置为 200。

## 46.5. 使用子资源

概述

可能需要由多个资源处理服务。例如，在处理服务最佳实践时，建议每个客户都作为唯一资源进行处理。每个顺序也会作为唯一资源进行处理。

使用 JAX-RS API，您将实施客户资源和子资源 的顺序。子资源是通过 root 资源类访问的资源。它们通过将 @Path 注释添加到资源类的方法来定义。子资源可以通过以下两种方式之一实现：

- 子资源方法- 间接为子资源实施 HTTP 动词，并使用 “指定 HTTP 动词”一节 中描述的其中一个注解进行解码。
- sub-resource locator- 指向实施子资源的类。

指定子资源

子资源通过减少带有 @Path 注释的方法来指定。子资源的 URI 如下：

1. 将子资源 @Path 注释的值附加到子资源的父资源的 @Path 注释的值。

   父资源的 @Path 注释可能位于资源类上，该方法返回包含子资源的类对象。

2. 重复上一步，直到达到 root 资源。

3. assembled URI 附加到部署该服务的基本 URI 中。

例如，例 46.6 “订购子资源” 中显示的子资源的 URI 可以是 baseURI/customerservice/order/12。

例 46.6. 订购子资源

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }
}
```

### 子资源方法

子资源方法使用 @Path 注释和其中一个 HTTP 动词注释进行解码。子资源方法直接负责处理使用指定的 HTTP 动词对资源发出的请求。

例 46.7 “子资源方法” 显示具有三个子资源方法的资源类：

- getOrder （） 处理 URI 与 /customerservice/orders/{orderId}/ 匹配的资源的 HTTP GET 请求。
- updateOrder （） 处理 URI 与 /customerservice/orders/{orderId}/ 匹配的资源的 HTTP PUT 请求。
- newOrder （） 处理位于 /customerservice/orders/ 的资源的 HTTP POST 请求。

**例 46.7. 子资源方法**

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @Path("/orders/{orderId}/")
  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
  {
    ...
  }

  @Path("/orders/")
  @POST
  public Order newOrder(Order order)
  {
    ...
  }
}
```

**注意**

具有相同 URI 模板的子资源方法等同于子资源 locator 返回的资源类。

**子资源 locators**

子资源 locator 与 HTTP 动词注解之一不分离，且不会直接处理子资源。相反，子资源 locator 返回可处理请求的资源类实例。

除了没有 HTTP 动词注解外，sub-resource locators 也不能有任何实体参数。子资源 locator 方法使用的所有参数都必须使用 第 47 章 将信息传递给资源类和方法 中描述的其中一个注解。

如 例 46.8 “子资源 locator 返回特定类” 所示，子资源 locator 允许您将资源封装为可重复使用的类，而不是将所有方法放在一个超级类中。processOrder （） 方法是一个子资源 locator。当请求在与 URI 模板 /orders/{orderId}/ 匹配的 URI 上时，它会返回 Order 类的实例。Order 类具有使用 HTTP 动词注解分离的方法。PUT 请求由 updateOrder （） 方法处理。

**例 46.8. 子资源 locator 返回特定类**

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  public Order processOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  ...
}

public class Order
{
  ...
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
  {
    ...
```

```
    }
}
```

子资源 locator 在运行时处理，以便它们可以支持 polymorphism。子资源 locator 的返回值可以是通用对象、抽象类或类层次结构的顶部。例如：如果您的服务需要处理 PayPal 订单和信用卡订单，则 例 46.8 “子资源 locator 返回特定类” 中的 processOrder （） 方法的签名可能会保持不变。您只需要实施两个类，即 ppOrder 和 ccOder，它们扩展了 Order 类。processOrder （） 的实现会根据需要的逻辑实例化子资源所需的实现。

### 46.6. 资源选择方法

#### 概述

给定 URI 可以映射到一个或多个资源方法。例如，URI customerservice/12/ma 可以与模板 @Path ("customerservice/{id}") 或 @Path ("customerservice/{id}/{state}/{state}") 匹配。JAX-RS 指定与请求匹配的资源方法的详细算法。该算法将规范化 URI、HTTP 动词和请求和响应实体的介质类型与资源类上的注解进行比较。

#### 基本选择算法

JAX-RS 选择算法分为三个阶段：

1. 确定 root 资源类。

   请求 URI 与使用 @Path 注释的所有类进行匹配。其 @Path 注释与请求 URI 相匹配的类是决定的。

   如果资源类 @Path 注释的值与整个请求 URI 匹配，则类的方法将用作第三个阶段的输入。

2. 确定对象将处理请求。

   如果请求 URI 大于所选类的 @Path 注释的值，则使用资源方法的 @Path 注解的值来查找可以处理请求的子资源。

   如果一个或多个子资源方法与请求 URI 匹配，则这些方法将用作第三个阶段的输入。

如果请求 URI 的唯一匹配是子资源定位器，则由子资源定位器创建的对象的资源方法与请求 URI 匹配。这个阶段会重复，直到子资源方法与请求 URI 匹配为止。

3. 选择处理请求的资源方法。

其 HTTP 动词注解与请求中的 HTTP 动词匹配的资源方法。另外，所选的资源方法必须接受请求实体正文的介质类型，并能够生成符合请求中指定的介质类型的响应。

从多个资源类中选择

选择算法的前两个阶段决定了将处理请求的资源。在某些情况下，资源由资源类实现。在其他情况下，它由使用同一 URI 模板的一个或多个子资源实现。当有多个资源与请求 URI 匹配时，资源类优先于子资源。

如果在资源类和子资源之间排序后，多个资源仍然与请求 URI 匹配，则以下条件用于选择单个资源：

1. 优先选择在其 URI 模板中具有最多文字字符的资源。

字面字符是不是模板变量一部分的字符。例如：/widgets/{id}/{color} 具有十个字面字符，/widgets/1/{color} 具有 eleven literal 字符。因此，请求 URI /widgets/1/red 将与具有 /widgets/1/{color} 的资源匹配，作为其 URI 模板。

**注意**

尾部斜杠(/)计为字面字符。因此，/joefred/ 将优先于 /joefred。

2. 首选资源在其 URI 模板中使用最多变量。

请求 URI /widgets/30/green 可以同时匹配 /widgets/{id}/{color} 和 /widgets/{amount}/。但是，将选择带有 URI 模板 /widgets/{id}/{color} 的资源，因为它有两个变量。

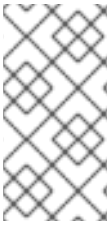
3. 首选使用包含正则表达式的最多变量的资源。

请求 URI /widgets/30/green 可以同时匹配 /widgets/{number}/{color} 和 /widgets/{id:.}/{color}*。但是，将选择带有 URI 模板 */widgets/{id:.}/{color} 的资源，因为它

有一个包含正则表达式的变量。

### 从多个资源方法中选择

在很多情况下，选择与请求 URI 匹配的资源会导致单个资源方法可以处理请求。该方法通过将请求中指定的 HTTP 动词与资源方法的 HTTP 动词匹配来确定。除了具有适当的 HTTP 动词注解外，所选的方法还必须处理请求中包含的请求实体，并且能够生成在请求的元数据中指定的正确响应类型。



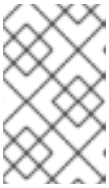
**注意**

资源方法可以处理的请求实体的类型由 @Consumes 注释指定。资源方法可以生成的响应类型使用 @Produces 注释来指定。

在选择资源时，会生成可处理请求的多种方法，该条件用于选择处理请求的资源方法：

1. 首选资源方法超过子资源。
2. 首选子资源方法而不是子资源定位器。
3. 首选使用 @Consumes 注释和 @Produces 注释中最具体值的方法。

   例如，具有注解 @Consumes ("text/xml") 的方法优先于带有注解 @Consumes ("textAttr") 的方法。这两种方法优先于没有 @Consumes 注释或注解 @Consumes ("\consider") 的方法。
4. 首选与请求正文实体的内容类型匹配的方法。

   **注意**

   请求正文实体的内容类型在 HTTP Content-Type 属性中指定。
5. 优先选择与作为响应接受的内容类型匹配的方法。

**注意**

在 HTTP Accept 属性中指定作为响应接受的内容类型。

自定义选择过程

在某些情况下，开发人员报告了选择多个资源类的方式有一定的限制。例如，如果给定资源类匹配，并且此类没有匹配的资源方法，则算法将停止执行。它永远不会检查剩余的匹配资源类。

Apache CXF 提供 org.apache.cxf.jaxrs.ext.ResourceComparator 接口，可用于自定义运行时如何处理多个匹配资源类。ResourceComparator 接口（如 例 46.9 “用于自定义资源选择的接口” 所示）需要实施的方法。一个比较两个资源类，另一个则比较两个资源方法。

**例 46.9. 用于自定义资源选择的接口**

```
package org.apache.cxf.jaxrs.ext;

import org.apache.cxf.jaxrs.model.ClassResourceInfo;
import org.apache.cxf.jaxrs.model.OperationResourceInfo;
import org.apache.cxf.message.Message;

public interface ResourceComparator
{
  int compare(ClassResourceInfo cri1,
              ClassResourceInfo cri2,
              Message message);

  int compare(OperationResourceInfo oper1,
              OperationResourceInfo oper2,
              Message message);
}
```

自定义实现在两个资源之间进行选择，如下所示：

- 如果第一个参数与第二个参数更匹配，则返回 1
- 如果第二个参数与第一个参数匹配，则返回 -1

如果返回 0，则运行时将继续默认的选择算法

*您可以通过将 resourceComparator 子添加到服务的 jaxrs:server 元素来注册自定义资源 Comparator 实现。*

# 第 47 章 将信息传递给资源类和方法


## 摘要

JAX-RS 指定多个注释，供开发人员控制传递到资源的信息的位置。注解符合 URI 中的常见 HTTP 概念，如列表参数。标准 API 允许对方法参数、bean 属性和资源类字段使用注解。Apache CXF 提供了一个扩展，允许将一系列参数注入 Bean。


## 47.1. 注入数据的基础知识

### 概述

使用 HTTP 请求消息中的数据初始化的参数、字段和 bean 属性，它们的值由运行时注入它们。注入的特定数据由 第 47.2 节 “使用 JAX-RS API” 中描述的一组注解指定。

JAX-RS 规范对数据注入时进行一些限制。它还对请求数据可以注入的对象类型进行一些限制。

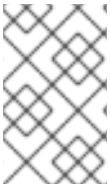
### 当数据被注入时

当请求因请求实例化时，请求数据会被注入到对象中。这意味着，只有与资源直接对应的对象才能使用注入注解。如 第 46 章 创建资源 所述，这些对象可以是带有 @Path 注释的 root 资源，也可以是从子资源 locator 方法返回的对象。

### 支持的数据类型

数据可以注入的特定一组数据类型取决于用来指定注入的数据源的注解。但是，所有注入注解至少支持以下数据类型：

- 原语，如 int、char 或 long
- 具有接受单个 String 参数的构造器的对象
- 具有静态 valueOf （） 方法的对象，它接受单个 String 参数
- List<T>, Set<T>, 或 SortedSet< T > objects，其中 T 满足列表中的其他条件



**注意**

如果注入注解对支持的数据类型有不同的要求，则会在讨论注解中突出显示差异。

## 47.2. 使用 JAX-RS API

### 47.2.1. JAX-RS 注解类型

标准 JAX-RS API 指定可用于将值注入字段、bean 属性和方法参数的注释。注解可以被分成三种不同的类型：

- 第 47.2.2 节 “从请求 URI 注入数据”
- 第 47.2.3 节 “从 HTTP 消息标头注入数据”
- 第 47.2.4 节 “从 HTML 表单注入数据”

### 47.2.2. 从请求 URI 注入数据

#### 概述

设计 RESTful Web 服务的最佳实践之一就是每个资源都应该有一个唯一的 URI。开发人员可以使用这个原则来为底层资源实施提供大量信息。在为资源设计 URI 模板时，开发人员可以构建模板，使其包含可注入到资源实施中的参数信息。开发人员还可以利用查询和列表参数将信息发送到资源实施中。

#### 从 URI 的路径获取数据

获取资源信息的更常见机制之一是通过为资源创建 URI 模板时使用的变量。这可以通过 javax.ws.rs.PathParam 注释来完成。@PathParam 注释具有一个参数，用于标识要从中注入数据的 URI 模板变量。

在 例 47.1 “从 URI 模板变量注入数据” 中，@PathParam 注释指定 URI 模板变量 颜色 的值被注入到 itemColor 字段中。

**例 47.1. 从 URI 模板变量注入数据**

```
import javax.ws.rs.Path;
```

```
import javax.ws.rs.PathParam
...

@Path("/boxes/{shape}/{color}")
class Box
{
  ...

  @PathParam("color")
  String itemColor;

  ...
}
```

*@PathParam 注释支持的数据类型与 "支持的数据类型"一节 中描述的数据类型不同。@PathParam 注释注入数据的实体必须是以下类型之一：*

- ***PathSegment***

  *该值将是路径匹配部分的最终片段。*

- ***List<PathSegment>***

  *该值将是与 named 模板参数匹配的 path segment 对应的 PathSegment 对象列表。*

- *原语，如 int、char 或 long*

- *具有接受单个 String 参数的构造器的对象*

- *具有静态 valueOf （） 方法的对象，它接受单个 String 参数*

*使用查询参数*

*在 Web 上传递信息的一个常见方法是使用 URI 中的查询参数。查询参数出现在 URI 的末尾，并用问号(?)与 URI 的资源位置部分分隔开。它们由一个或者多个名称值对组成，其中名称和值用等号(=)分隔。当指定多个查询参数时，对通过分号(;)或 符号（和）相互分隔。例 47.2 "带有查询字符串的 URI" 显示带有查询参数的 URI 语法。*

**例 47.2. 带有查询字符串的 URI**

```
http://fusesource.org?name=value;name2=value2;...
```

**注意**

**您可以使用分号或 符号来分隔查询参数，但不能同时使用两者。**

**javax.ws.rs.QueryParam 注释提取查询参数的值并将其注入 JAX-RS 资源。该注解使用一个参数来标识从中提取值的查询参数的名称，并注入到指定字段、bean 属性或参数中。@QueryParam 注释支持“支持的数据类型”一节 中描述的类型。**

**例 47.3 “使用查询参数中的数据的资源方法” 显示资源方法，将查询参数 id 的值注入方法的 id 参数。**

**例 47.3. 使用查询参数中的数据的资源方法**

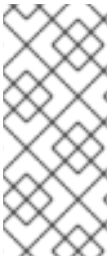
```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  @Path("/{type}")
  public void updateMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
    ...
  }
  ...
}
```

**要处理到 /monstersforhire/daikaiju?id=jonas 的 HTTP POST，将 updateMonster （） 方法的类型设为 daikaiju，ID 设置为 jonas。**

**使用列表参数**

**URI 列表参数（如 URI 查询参数）是名称/值对，可以提供额外的信息选择资源。与查询参数不同，**

列表参数可以在 URI 的任意位置显示，它们使用分号(;)与 URI 的层次结构路径段分隔开。/mostersforhire/daikaiju;id=jonas 有一个 matrix 参数，名为 id 和 /monstersforhire/japan；type=daikaiju/flying;wingspan=40 有两个列表参数，称为 type 和 wingspan。



注意

计算资源的 URI 时，不会评估列表参数。因此，用于查找用于处理请求 URI /monstersforhire/japan;type=daikaiju/flying;wingspan=40 是 /monstersforhire/japan/flying 的 URI。

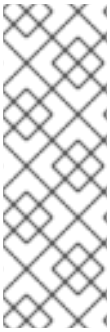
matrix 参数的值使用 javax.ws.rs.MatrixParam 注释注入到字段、参数或 bean 属性中。该注解使用一个参数来标识从中提取值的 matrix 参数的名称，并注入到指定字段、bean 属性或参数中。@MatrixParam 注释支持 “支持的数据类型”一节 中描述的类型。

例 47.4 “使用列表参数中的数据的资源方法” 显示一个资源方法，将 matrix 参数 type 和 id 的值注入方法的参数。

例 47.4. 使用列表参数中的数据的资源方法

```
import javax.ws.rs.MatrixParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  public void updateMonster(@MatrixParam("type") String type,
                            @MatrixParam("id") String id)
  {
    ...
  }
  ...
}
```

要处理到 /monstersforhire 的 HTTP POST，type=daikaiju;id=whale updateMonster （） 方法的 type 设置为 daikaiju，ID 设置为 whale。



*注意*

*JAX-RS 一次评估 URI 中的所有列表参数，因此无法对 URI 中的列表参数位置强制限制。例如： /monstersforhire/japan;type=daikaiju/flying;wingspan=40 , /monstersforhire/japan/flying;type=daikaiju;wingspan=40, 和 /monstersforhire/japan; type=daikaiju;wingspan=40/flying 都被视为相当于使用 JAX-RS API 实施的 RESTful Web 服务。*

*禁用 URI 解码*

*默认情况下，所有请求 URI 都解码。因此， URI /monster/night%20stalker 和 URI /monster/night stalker 等效。自动 URI 解码使得在 ASCII 字符集外轻松发送字符作为参数。*

*如果您不想自动解码 URI，您可以使用 javax.ws.rs.Encoded 注解来停用 URI 解码。注解可用于在以下级别取消激活 URI 解码：*

- *class level-Decorating a class with the @Encoded 注释会取消激活类中所有参数、字段和 bean 属性的 URI 解码。*
- *方法 level-Decorating 方法带有 @Encoded 注释可停用类所有参数的 URI 解码。*
- *参数/字段级别取消参数或字段（带有 @Encoded 注释）取消激活类所有参数的 URI 解码。*

*例 47.5 “禁用 URI 解码” 显示其 getMonster （） 方法不使用 URI 解码的资源。addMonster （） 方法只禁用 type 参数的 URI 解码。*

*例 47.5. 禁用 URI 解码*

```
@Path("/monstersforhire/")
public class MonsterService
{
  ...

  @GET
  @Encoded
  @Path("/{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
    ...
  }
```

```
    @PUT
    @Path("/{id}")
    public void addMonster(@Encoded @PathParam("type") String type,
                           @QueryParam("id") String id)
    {
      ...
    }
    ...
  }
```

*错误处理*

*如果尝试使用其中一个 URI 注入注解注入数据时出现错误，则会生成 WebApplicationException 异常嵌套原始异常。WebApplicationException 异常的状态设置为 404。*

### *47.2.3. 从 HTTP 消息标头注入数据*

*概述*

*在正常使用请求消息中的 HTTP 标头传递有关消息的通用信息、在传输中如何处理它，以及预期响应的详细信息。虽然通常识别和使用几个标准标头，但 HTTP 规格允许任何名称/值对用作 HTTP 标头。JAX-RS API 提供了将 HTTP 标头信息注入资源实施的简单机制。*

*最常用的 HTTP 标头之一是 cookie。Cookie 允许 HTTP 客户端和服务器在多个请求/响应序列之间共享静态信息。JAX-RS API 提供了注释，可直接将数据从 Cookie 注入资源实施。*

*从 HTTP 标头注入信息*

*javax.ws.rs.HeaderParam 注释用于将 HTTP 标头字段中的数据注入参数、字段或 bean 属性。它有一个单一参数，用于指定从中提取值并注入到资源实施中的 HTTP 标头字段的名称。关联的参数、字段或 bean 属性必须符合 "支持的数据类型"一节 中描述的数据类型。*

*注入 If-Modified-Since 标头 显示将 HTTP If-Modified-Since 标头的值注入类 oldestDate 字段的代码。*

*注入 If-Modified-Since 标头*

```
import javax.ws.rs.HeaderParam;
...
class RecordKeeper
```

```
{
  ...
  @HeaderParam("If-Modified-Since")
  String oldestDate;
  ...
}
```

### 从 Cookie 注入信息

Cookie 是特殊的 HTTP 标头类型。它们由一个或多个名称/值对组成，这些对传递到第一个请求上的资源实施。在第一个请求后，cookie 会在供应商和消费者之间返回，并显示每个消息。只有消费者，因为它们生成请求才能更改 Cookie。Cookie 通常用于跨多个请求/响应序列、存储用户设置和其他可保留数据维护会话。

javax.ws.rs.CookieParam 注释从 Cookie 的字段中提取值，并将它注入到资源实施中。它使用一个参数，用于指定要提取值的 Cookie 字段的名称。除了 “支持的数据类型”一节 中列出的数据类型外，@CookieParam 的实体也是 Cookie 对象。

例 47.6 “注入 Cookie” 显示将 handle cookie 的值注入 CB 类中的一个字段的代码。

**例 47.6. 注入 Cookie**

```
import javax.ws.rs.CookieParam;
...
class CB
{
  ...
  @CookieParam("handle")
  String handle;
  ...
}
```

### 错误处理

如果尝试使用其中一个 HTTP 消息注入数据时出现错误，则会生成 WebApplicationException 异常嵌套原始异常。WebApplicationException 异常的状态设置为 400。

## 47.2.4. 从 HTML 表单注入数据

*概述*

HTML 表单是一种从用户获取信息的简单方法，而且易于创建。表单数据可用于 HTTP GET 请求和 HTTP POST 请求：

*GET*

当作为 HTTP GET 请求的一部分发送表单数据时，数据将作为一组查询参数附加到 URI 中。"使用查询参数"一节 中讨论从查询参数注入数据。

*POST*

当将数据作为 HTTP POST 请求的一部分发送时，数据将放置在 HTTP 消息正文中。可以使用支持表单数据的常规实体参数来处理表单数据。也可以通过使用 @FormParam 注释来提取数据并注入资源方法参数来处理它。

*使用 @FormParam 注释来注入表单数据*

javax.ws.rs.FormParam 注释从表单数据中提取字段值，并将值注入到资源方法参数中。该注解使用一个参数，用于指定从中提取值的字段的键。关联的参数必须符合 "支持的数据类型"一节 中描述的数据类型。

*重要*

JAX-RS API Javadoc 表示 @FormParam 注释可以放在字段、方法和参数上。但是，只有在放在资源方法参数上时，@FormParam 注释才有意义。

*Example*

将表单数据注入资源方法参数 显示将数据注入其参数的资源方法。该方法假定客户端的表单包含三个字段，标题为、标签，以及包含字符串数据 的正文。

*将表单数据注入资源方法参数*

```
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;

...
```

```
@POST
public boolean updatePost(@FormParam("title") String title,
                          @FormParam("tags") String tags,
                          @FormParam("body") String post)
{
  ...
}
```

### 47.2.5. 指定要注入的默认值

#### 概述

要为更强大的服务实现提供，您可能需要确保可以将任何可选参数设置为默认值。这对从查询参数和列表参数获取的值特别有用，因为输入较长的 URI 字符串非常容易出错。您可能还想为从 Cookie 中提取的参数设置默认值，因为请求系统没有适当的信息来构造带有所有值的 Cookie。

javax.ws.rs.DefaultValue 注释可与以下注入注解结合使用：

- @PathParam
- @QueryParam
- @MatrixParam
- @FormParam
- @HeaderParam
- @CookieParam

@DefaultValue 注释指定在请求中没有与注入注释对应的数据时使用的默认值。

语法

设置参数默认值的语法 显示使用 @DefaultValue 注释的语法。

设置参数默认值的语法

```
import javax.ws.rs.DefaultValue;
 ...
 void resourceMethod(@MatrixParam("matrix")
                     @DefaultValue("value)
                     int someValue, ... )
 ...
```

该注解必须在 parameter、bean 或 field 之前，它将生效。与附带注入注释相关的 @DefaultValue 注释的位置无关紧要。

@DefaultValue 注释采用单个参数。如果无法根据注入注解提取正确的数据，则此参数值是将注入字段的值。该值可以是任意 String 值。该值应与相关字段的类型兼容。例如，如果关联的字段是 int 类型，则默认值 blue 会产生异常。

处理列表和集合

如果注解的参数的类型，bean 或字段是 List、Set 或 SortedSet，则生成的集合将从提供的默认值映射的一个条目。

Example

设置默认值 演示了使用 @DefaultValue 为注入值的字段指定默认值的两个示例。

设置默认值

```
import javax.ws.rs.DefaultValue;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/monster")
public class MonsterService
{

  @Get
  public Monster getMonster(@QueryParam("id") @DefaultValue("42") int id,
                            @QueryParam("type") @DefaultValue("bogeyman") String type)
  {
    ...
  }

  ...
}
```

*当 GET 请求发送到 baseURI/monster 时，会调用 设置默认值 中的 getMonster （） 方法。该方法需要两个查询参数： id 和 type，并附加到 URI 中。因此，使用 URI baseURI/monster?id=1&type=fomurllibiri 的 GET 请求会返回 FomedServiceSetiri，ID 为 one。*

*由于 @DefaultValue 注释放在这两个参数上，因此如果忽略了查询参数，getMonster （） 方法可以运行。发送到 baseURI/monster 的 GET 请求等同于使用 URI baseURI/monster?id=42&type=bogeyman 的 GET 请求。*

### 47.2.6. 将参数注入 Java Bean

**概述**

*当通过 REST 发布 HTML 表单时，服务器端的常见模式是创建一个 Java bean 来封装形式收到的所有数据（以及可能来自其他参数和 HTML 标头的数据）。通常，创建此 Java bean 分为两个步骤：资源方法通过注入接收表单值（例如，通过将 @FormParam 注解添加到其方法参数），而资源方法随后调用 bean 的构造器，以传递表单数据。*

*使用 JAX-RS 2.0 @BeanParam 注释，可以在单一步骤中实施此模式。表单数据可以直接注入到 bean 类的字段，而 bean 本身则由 JAX-RS 运行时自动创建。例如，这最容易解释。*

**注入目标**

*@BeanParam 注释可以附加到资源方法参数、资源字段或 bean 属性。但是，参数目标是可用于所有资源类生命周期的唯一目标类型。其他类型的目标仅限于每个请求的生命周期。表 47.1 "@BeanParam Injection Targets" 中总结了这种情况。*

*表 47.1. @BeanParam Injection Targets*

| 目标 | 资源类生命周期 |
| --- | --- |
| **参数** | All |
| **FIELD** | 按请求（默认） |
| **METHOD** (bean 属性) | 按请求（默认） |

*没有 BeanParam 注解的示例*

*以下示例演示了如何使用传统方法捕获 Java bean 中的表单数据（不使用 @BeanParam）：*

```
// Java
import javax.ws.rs.POST;
import javax.ws.rs.FormParam;
import javax.ws.rs.core.Response;
...
@POST
public Response orderTable(@FormParam("orderId")  String orderId,
                           @FormParam("color")    String color,
                           @FormParam("quantity") String quantity,
                           @FormParam("price")    String price)
{
    ...
    TableOrder bean = new TableOrder(orderId, color, quantity, price);
    ...
    return Response.ok().build();
}
```

*在本例中，orderTable 方法处理了一个表单，用于为周年网站订购表的数量。发布订购表单后，表单值将注入到 orderTable 方法的参数中，而 orderTable 方法会明确创建 TableOrder 类的实例，使用注入的形式数据。*

*带有 BeanParam 注解的示例*

*上一示例可以重构，以利用 @BeanParam 注释。使用 @BeanParam 方法时，表单参数可以直接注入到 bean 类的字段 TableOrder。实际上，您可以使用 bean 类中的任何标准 JAX-RS 参数注释：包括 @PathParam,@QueryParam,@FormParam,@MatrixParam,@CookieParam, 和 @HeaderParam。处理表单的代码可以被重构，如下所示：*

```
// Java
import javax.ws.rs.POST;
import javax.ws.rs.FormParam;
import javax.ws.rs.core.Response;
...
public class TableOrder {
    @FormParam("orderId")
    private String orderId;

    @FormParam("color")
    private String color;

    @FormParam("quantity")
    private String quantity;

    @FormParam("price")
    private String price;

    // Define public getter/setter methods
    // (Not shown)
    ...
}
...
@POST
public Response orderTable(@BeanParam TableOrder orderBean)
{
    ...
    // Do whatever you like with the 'orderBean' bean
    ...
    return Response.ok().build();
}
```

现在，表单注解已添加到 bean 类 TableOrder 中，您可以将资源方法签名中的所有 @FormParam 注解替换为单个 @BeanParam 注释，如下所示。现在，当表单发布到 orderTable 资源方法时，JAX-RS 运行时会自动创建一个 TableOrder 实例，orderBean，并在 bean 类上注入参数注解指定的所有数据。

### 47.3. 参数转换器

#### 概述

使用参数转换器，可以将参数（字符串类型）注入 任何类型的 字段、bean 属性或资源方法参数。通过实施和绑定合适的参数转换器，您可以扩展 JAX-RS 运行时，使其能够将参数 String 值转换为目标类型。

#### 自动转换

参数作为 String 的实例接收，因此始终可以将它们直接注入 String 类型的字段、bean 属性和方法参数。此外，JAX-RS 运行时具有将参数字符串自动转换为以下类型的功能：

1. 原语类型。

2. 具有接受单个 String 参数的构造器的类型。

3. 具有名为 valueOf 或 fromString 的静态方法的类型，它带有一个 String 参数，该参数返回一个类型实例。

4. 如果 T 是 2 或 3 中描述的类型之一，则列出 <T >、Set< T > 或 SortedSet <T>。

### 参数转换器

要将参数注入没有由自动转换涵盖的类型，您可以为 类型定义自定义参数转换器。参数转换器是 JAX-RS 扩展，允许您定义从 String 到自定义类型的转换，以及从自定义类型到 String 的反向方向。

### 工厂模式

JAX-RS 参数转换器机制使用工厂模式。因此，您可以按需注册参数转换器提供程序（类型为 javax.ws.rs.ext.ParamConverterProvider），而不是直接注册参数转换器（类型为 javax.ws.rs.ext.ParamConverterProvider）。

### ParamConverter 接口

javax.ws.rs.ext.ParamConverter 接口定义如下：

```
// Java
package javax.ws.rs.ext;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.ws.rs.DefaultValue;

public interface ParamConverter<T> {

    @Target({ElementType.TYPE})
    @Retention(RetentionPolicy.RUNTIME)
```

```
    @Documented
    public static @interface Lazy {}

    public T fromString(String value);

    public String toString(T value);
}
```

*要实现自己的 ParamConverter 类，您必须实施这个接口，覆盖 fromString 方法（将参数字符串转换为目标类型）和 toString 方法（将目标类型转换为字符串）。*

### *ParamConverterProvider 接口*

*javax.ws.rs.ext.ParamConverterProvider 接口定义如下：*

```
// Java
package javax.ws.rs.ext;

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

public interface ParamConverterProvider {
    public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation
annotations[]);
}
```

*要实现自己的 ParamConverterProvider 类，您必须实现这个接口，覆盖 getConverter 方法，这是创建 ParamConverter 实例的 factory 方法。*

### *绑定参数转换器供应商*

*要将参数转换器提供程序绑定到 JAX-RS 运行时（使其可用于应用程序），您必须使用 @Provider 注释给您的实施类添加注解，如下所示：*

```
// Java
...
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

@Provider
public class TargetTypeProvider implements ParamConverterProvider {
    ...
}
```

此注解可确保在部署的扫描阶段自动注册参数转换器供应商。

**Example**

以下示例演示了如何实现 **ParamConverterProvider** 和一个 **ParamConverter**，它能够将参数字符串转换为 **TargetType** 类型：

```
// Java
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.ext.ParamConverter;
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

@Provider
public class TargetTypeProvider implements ParamConverterProvider {

    @Override
    public <T> ParamConverter<T> getConverter(
        Class<T> rawType,
        Type genericType,
        Annotation[] annotations
    ) {
        if (rawType.getName().equals(TargetType.class.getName())) {
            return new ParamConverter<T>() {

                @Override
                public T fromString(String value) {
                    // Perform conversion of value
                    // ...
                    TargetType convertedValue = // ... ;
                    return convertedValue;
                }

                @Override
                public String toString(T value) {
                    if (value == null) { return null; }
                    // Assuming that TargetType.toString is defined
                    return value.toString();
                }
            };
        }
        return null;
    }

}
```

使用参数转换器

现在，您为 TargetType 定义了参数转换器，现在可以直接将参数注入 TargetType 字段和参数，例如：

```
// Java
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
...
@POST
public Response updatePost(@FormParam("target") TargetType target)
{
  ...
}
```

### 默认值的 lazy 转换

如果您为参数指定默认值（使用 @DefaultValue 注释），您可以选择默认值是否立即转换为目标类型（默认行为），或者是否仅在需要时转换默认值（延迟转换）。要选择 lazy 转换，请将 @ParamConverter.Lazy 注释添加到目标类型。例如：

```
// Java
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.ext.ParamConverter.Lazy;
...
@POST
public Response updatePost(
    @FormParam("target")
    @DefaultValue("default val")
    @ParamConverter.Lazy
    TargetType target)
{
  ...
}
```

## 47.4. 使用 APACHE CXF 扩展

### 概述

Apache CXF 为标准 JAX-WS 注入机制提供了一个扩展，供开发人员使用单个注解替换一系列注入注解。单个注解放置在包含使用注解提取数据的字段的 bean 中。例如，如果资源方法希望请求 URI 包含三个名为 id 的查询参数，输入，以及 大小，则可以使用单个 @QueryParam 注释，将所有参数注入到带有对应字段的 bean 中。

**注意**

请考虑使用 @BeanParam 注释（自 JAX-RS 2.0 起可用）。标准化的 @BeanParam 方法比专有 Apache CXF 扩展更灵活，因此推荐的替代方案。详情请查看 第 47.2.6 节 “将参数注入 Java Bean”。

**支持的注入注解**

此扩展不支持所有注入参数。它只支持以下内容：

- @PathParam
- @QueryParam
- @MatrixParam
- @FormParam

**语法**

要指示注解将使用串行注入到 bean 中，您需要执行两个操作：

1. 将注解的参数指定为空字符串。例如，@PathParam ("") 指定将一系列 URI 模板变量序列化为 bean。
2. 确保 annotated 参数是一个 bean，其中包含与要注入的值匹配的字段。

**Example**

例 47.7 “将查询参数注入 bean” 演示了将多个 Query 参数注入 bean 的示例。资源方法预期请求 URI 包含两个查询参数： type 和 id。其值注入到 Monster bean 的对应字段中。

**例 47.7. 将查询参数注入 bean**

```
import javax.ws.rs.QueryParam;
```

```
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  public void updateMonster(@QueryParam("") Monster bean)
  {
    ...
  }
  ...
}

public class Monster
{
  String type;
  String id;

  ...
}
```

# 第 48 章 将信息返回到 CONSUMER

**摘要**

RESTful 请求要求至少将 HTTP 响应代码返回到消费者。在很多情况下，可以通过返回普通 JAXB 对象或 GenericEntity 对象来满足请求。当资源方法需要返回其他元数据以及响应实体时，JAX-RS 资源方法可以返回包含任何需要 HTTP 标头或其他元数据的 Response 对象。

## 48.1. 返回类型

返回到使用者的信息决定了资源方法返回的确切对象类型。这似乎很明显，但 Java 返回对象和返回到 RESTful 使用者之间的映射不是一对一的。除了任何响应实体正文外，RESTful 用户还需要返回有效的 HTTP 返回代码。Java 对象中包含的数据到响应实体的映射由消费者将接受的 MIME 类型生效。

要解决将 Java 对象映射到 RESTful 响应消息时涉及的问题，资源方法可以返回四种 Java 构造：

- 第 48.2 节 “返回普通 Java 结构” 使用由 JAX-RS 运行时决定的 HTTP 返回代码返回基本信息。
- 第 48.2 节 “返回普通 Java 结构” 使用由 JAX-RS 运行时决定的 HTTP 返回代码返回复杂信息。
- 第 48.3 节 “微调应用程序的响应” 使用编程确定的 HTTP 返回状态返回复杂信息。Response 对象还允许指定 HTTP 标头。
- 第 48.4 节 “使用通用类型信息返回实体” 使用由 JAX-RS 运行时决定的 HTTP 返回代码返回复杂信息。GenericEnitity 对象为运行时组件提供对数据序列化的更多信息。

## 48.2. 返回普通 JAVA 结构

**概述**

在很多情况下，资源类可以返回标准 Java 类型、一个 JAXB 对象或应用具有实体提供程序的任何对象。在这些情况下，运行时使用所返回的对象的 Java 类来确定 MIME 类型信息。运行时还决定要发送给消费者的适当 HTTP 返回代码。

**可返回的类型**

资源方法可以返回 void 或提供实体写入器的任何 Java 类型。默认情况下，运行时有以下内容的供应商：

- Java 原语
- Java 原语 的数量 表示
- JAXB 对象

"原生支持的类型"一节 列出默认支持的所有返回类型。"自定义写入器"一节 描述如何实施自定义实体写入器。

MIME 类型

运行时首先检查 @Produces 注释的资源方法和资源类来确定返回的实体的 MIME 类型。如果找到，它将使用注解中指定的 MIME 类型。如果找不到由资源实现指定的资源，它依赖于实体供应商来确定正确的 MIME 类型。

默认情况下，运行时分配 MIME 类型，如下所示：

- Java 原语及其 数量 表示被分配一个 MIME 类型 application/octet-stream。
- 为 JAXB 对象分配 MIME 类型 application/xml。

应用程序可以通过实现自定义实体供应商来使用其他映射，如 "自定义写入器"一节 所述。

响应代码

当资源方法返回普通 Java 结构时，如果资源方法完成，则运行时会自动设置响应的状态代码，而不抛出异常。状态代码设置为如下：

- 204（无内容）- 资源方法的返回类型是 void

- 204（无内容）- 返回实体的值为 null
- 200(OK)- 返回实体的值不是 null

如果在资源方法完成前抛出异常，则返回状态代码会被设置，如 第 50 章 处理例外 所述。

## 48.3. 微调应用程序的响应

### 48.3.1. 构建响应的基础知识

#### 概述

当资源方法返回普通 Java 构造时，RESTful 服务通常需要更精确地控制返回到消费者的响应。JAX-RS Response 类允许资源方法对发送到消费者的返回状态进行一些控制，并在响应中指定 HTTP 消息标头和 Cookie。

响应 对象嵌套代表返回到消费者的实体的对象。响应 对象使用 ResponseBuilder 类作为工厂实例化。

ResponseBuilder 类也有许多方法用于操作响应的元数据。例如，ResonseBuilder 类包含设置 HTTP 标头和缓存控制指令的方法。

#### 响应和响应构建器之间的关系

Response 类具有受保护的构造器，因此无法直接实例化它们。它们使用 ResponseBuilder 类创建，该类由 Response 类括起。ResponseBuilder 类是所有信息的所有者，将被封装到从其创建的响应中。ResponseBuilder 类也具有负责在消息上设置 HTTP 标头属性的所有方法。

Response 类提供了一些简化设置正确响应代码并嵌套实体的方法。每个常见的响应状态代码都有方法。与包含实体正文或所需元数据的状态对应的方法包括允许直接设置信息到关联的响应构建器的版本。

ResponseBuilder 类的 build （） 方法返回一个响应对象，其中包含调用方法时响应构建器中存储的信息。返回响应对象后，响应构建器将返回到干净的状态。

#### 获取响应构建器

获取响应构建器的方法有两种：

- 使用 Response 类的静态方法，如 使用 Response 类获取响应构建器 所示。

使用 Response 类获取响应构建器

```
import javax.ws.rs.core.Response;

Response r = Response.ok().build();
```

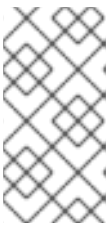
在获得响应构建器时，您无法访问实例的访问权限，您可以在多个步骤中操作。您必须将所有操作字符串为单一方法调用。

- 使用 Apache CXF 特定的 ResponseBuilderImpl 类。此类允许您直接使用响应构建器。但是，它要求您手动设置所有响应构建器信息。

例 48.1 “使用 ResponseBuilderImpl 类获取响应构建器” 演示了如何使用 ResponseBuilderImpl 类重写 使用 Response 类获取响应构建器。

例 48.1. 使用 ResponseBuilderImpl 类获取响应构建器

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(200);
Response r = builder.build();
```



注意

您还可以简单地将从 Response 类方法返回的 ResponseBuilder 分配给 ResponseBuilderImpl 对象。

更多信息

有关 Response 类的更多信息，请参阅 Response 类的 Javadoc。

有关 ResponseBuilder 类的更多信息，请参阅 ResponseBuilder 类的 Javadoc。

有关 Apache CXF ResponseBuilderImpl 类的更多信息，请参阅 ResponseBuilderImpl Javadoc。

### 48.3.2. 为常见用例创建响应

**概述**

Response 类提供处理 RESTful 服务需要更常见的响应的快捷方式。这些方法使用提供的值或默认值处理正确的标头。它们也会在适当的时候处理实体正文。

为成功请求创建响应

成功处理请求时，应用需要发送响应来确认请求已实现。该响应可能包含实体。

成功完成响应时最常见的响应是 OK。OK 响应通常包含一个与请求对应的实体。Response 类有一个超载的 ok （） 方法，将响应状态设置为 200，并将提供的实体添加到括起的响应构建器中。ok （） 方法有五个版本。最常用的变体有：

- response .ok （）- 创建状态为 200 和空实体正文的响应。
- response .ok (java.lang.Object entity)- 创建状态为 200 的响应，将提供的对象存储在响应实体正文中，并通过内省对象来确定实体介质类型。

创建具有 200 响应的响应 显示了创建具有 OK 状态的响应的示例。

创建具有 200 响应的响应

```
import javax.ws.rs.core.Response;
import demo.jaxrs.server.Customer;
...
```

```
Customer customer = new Customer("Jane", 12);

return Response.ok(customer).build();
```

对于请求者没有期望实体正文，则更适合发送 204 No Content 状态而不是 200 OK 状态。Response.noContent （） 方法将创建适当的响应对象。

创建具有 204 状态的响应 显示了创建具有 204 状态的响应的示例。

创建具有 204 状态的响应

```
import javax.ws.rs.core.Response;

return Response.noContent().build();
```

为重定向创建响应

Response 类提供了处理三个重定向响应状态的方法。

303 查看其他

当请求的资源需要永久重定向到新资源来处理请求时，303 See Other 状态很有用。

Response 类 seeOther （） 方法创建具有 303 状态的响应，并将新的资源 URI 放置到消息的 Location 字段中。seeOther （） 方法使用一个参数，该参数将新 URI 指定为 java.net.URI 对象。

304 not Modified

根据请求的性质，304 Not Modified 状态可用于不同的事情。它可用于表示请求的资源自以前的 GET 请求起没有改变。它还可用于表示修改资源的请求不会导致资源被更改。

Response class notModified （） 方法会创建一个具有 304 状态的响应，并在 HTTP 消息上设置修改后的日期属性。notModified （） 方法有三个版本：

- NotModified
- notModifiedjavax.ws.rs.core.Entitytag
- NotModifiedjava.lang.Stringtag

307 Temporary Redirect

当请求的资源需要将消费者定向到新资源时，307 Temporary Redirect 状态很有用，但希望消费者继续使用此资源来处理将来的请求。

Response class temporaryRedirect （） 方法会创建一个具有 307 状态的响应，并将新的资源 URI 放置到消息的 Location 字段中。temporaryRedirect （） 方法采用单个参数，该参数将新 URI 指定为 java.net.URI 对象。

创建具有 304 状态的响应 显示了创建具有 304 状态的响应的示例。

创建具有 304 状态的响应

```
import javax.ws.rs.core.Response;

return Response.notModified().build();
```

创建对信号错误的响应

Response 类提供了为两个基本处理错误创建响应的方法：

- serverError- 创建状态为 500 Internal Server Error 的响应。
- notAcceptablejava.util.List<javax.ws.rs.core.Variant>变体- 创建具有 406 Not Acceptable 状态的响应以及包含可接受的资源类型列表的实体正文。

创建具有 500 状态的响应 显示了创建具有 500 状态的响应的示例。

创建具有 500 状态的响应

```
import javax.ws.rs.core.Response;

return Response.serverError().build();
```

### 48.3.3. 处理更高级的响应

概述

响应 类方法为常见情况创建响应提供了简短的剪切。当您需要处理更复杂的情况，如指定缓存控制指令、添加自定义 HTTP 标头或发送由 Response 类处理的状态时，您需要使用 ResponseBuilder 类方法在使用 build （） 方法生成响应前填充响应。

如 “获取响应构建器”一节 所述，您可以使用 Apache CXF ResponseBuilderImpl 类创建可直接操作的响应构建器实例。

添加自定义标头

利用 ResponseBuilder 类的 header （） 方法，将自定义标头添加到响应中。header （） 方法采用两个参数：

- name-a 指定标头名称的字符串
- value- 包含标头中存储数据的 Java 对象

您可以通过重复调用 header （） 方法来对消息设置多个标头。

在响应中添加标头 显示向响应添加标头的代码。

在响应中添加标头

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.header("username", "joe");
Response r = builder.build();
```

添加 Cookie

利用 ResponseBuilder 类的 cookie （） 方法，将自定义标头添加到响应中。cookie （） 方法采用一个或多个 Cookie。每个 Cookie 存储在 javax.ws.rs.core.NewCookie 对象中。最容易使用的 NewCookie 类的 contructors 取两个参数：

- name-a 指定 Cookie 名称的字符串
- value- 指定 Cookie 值的字符串

您可以通过重复调用 cookie （） 方法来设置多个 Cookie。

在响应中添加 Cookie 显示向响应添加 Cookie 的代码。

在响应中添加 Cookie

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.NewCookie;

NewCookie cookie = new NewCookie("username", "joe");

Response r = Response.ok().cookie(cookie).build();
```

警告

使用 null 参数列表调用 cookie （） 方法，清除已与响应关联的任何 Cookie。

设置响应状态

当您要返回 Response 类帮助方法支持的状态以外的状态时，您可以使用 ResponseBuilder 类的 status （） 方法来设置响应的状态代码。status （） 方法有两个变体。一个用 int 来指定响应代码。另一个对象采用 Response.Status 对象来指定响应代码。

Response.Status 类是一个枚举在 Response 类中。它包含大多数定义的 HTTP 响应代码的条目。

在响应中添加标头 显示将响应状态设置为 404 Not Found 的代码。

在响应中添加标头

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(404);
Response r = builder.build();
```

设置缓存控制指令

ResponseBuilder 类的 cacheControl （） 方法允许您对响应设置缓存控制标头。cacheControl （） 方法采用 javax.ws.rs.CacheControl 对象，用于指定响应的缓存控制指令。

CacheControl 类具有与 HTTP 规范支持的所有缓存控制指令对应的方法。其中，指令是一个简单 on

或 off 值，即 setter 方法采用 布尔值。其中指令需要一个数字值，如 max-age 指令，则 setter 取一个 int 值。

在响应中添加标头 显示用于设置 no-store 缓存控制指令的代码。

在响应中添加标头

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.CacheControl;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

CacheControl cache = new CacheControl();
cache.setNoCache(true);

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.cacheControl(cache);
Response r = builder.build();
```

### 48.4. 使用通用类型信息返回实体

概述

在有些情况下，应用程序需要对返回的对象的 MIME 类型进行更多控制，或用于序列化响应的实体供应商。JAX-RS javax.ws.rs.core.GenericEntity<T > 类通过提供指定代表实体的通用对象类型的机制，提供对实体序列化的精细控制。

使用 GenericEntity<T> 对象

用于选择序列化响应的实体提供程序的标准之一是对象的通用类型。对象的通用类型代表对象的 Java 类型。当返回通用 Java 类型或 JAXB 对象时，运行时可以使用 Java 反映来确定通用类型。但是，当返回 JAX-RS Response 对象时，运行时无法确定嵌套实体的通用类型，对象的实际 Java 类被用作 Java 类型。

为确保实体供应商提供了正确的通用类型信息，可在将实体添加到返回的 Response 对象前嵌套在 GenericEntity<T > 对象中。

资源方法也可以直接返回 GenericEntity<T> 对象。实际上，这种方法很少被使用。通过反映未封装实体以及存储在 GenericEntity<T> 对象中的实体的通用类型信息所确定的 通用类型信息通常相同。

### 创建 GenericEntity<T> 对象

创建 GenericEntity<T> 对象的方法有两种：

1. 使用所包装实体，创建 GenericEntity<T > 类的子类。使用子类创建 GenericEntity<T> 对象 演示了如何创建一个 GenericEntity<T> 对象，其中包含一个类型 List<String& gt; 的实体，其通用类型将在运行时可用。

   #### 使用子类创建 GenericEntity<T> 对象

   ```
   import javax.ws.rs.core.GenericEntity;

   List<String> list = new ArrayList<String>();
   ...
   GenericEntity<List<String>> entity =
     new GenericEntity<List<String>>(list) {};
   Response response = Response.ok(entity).build();
   ```

   用于创建 GenericEntity<T> 对象的子类通常是匿名的。

2. 通过向实体提供通用类型信息来直接创建一个实例。例 48.2 “直接实例化 GenericEntity<T> 对象” 显示如何创建包含类型为 AtomicInteger 实体的响应。

   **例 48.2. 直接实例化 GenericEntity<T> 对象**

   ```
   import javax.ws.rs.core.GenericEntity;

   AtomicInteger result = new AtomicInteger(12);
   GenericEntity<AtomicInteger> entity =
     new GenericEntity<AtomicInteger>(result,
                                      result.getClass().getGenericSuperclass());
   Response response = Response.ok(entity).build();
   ```

## 48.5. 异步响应

### 48.5.1. 服务器上的异步处理

#### 概述

在服务器端异步处理调用的目的是，可以更有效地使用线程，并最终避免客户端连接尝试被拒绝的情况，因为所有服务器的请求线程都被阻止。异步处理调用时，请求线程将立即释放。

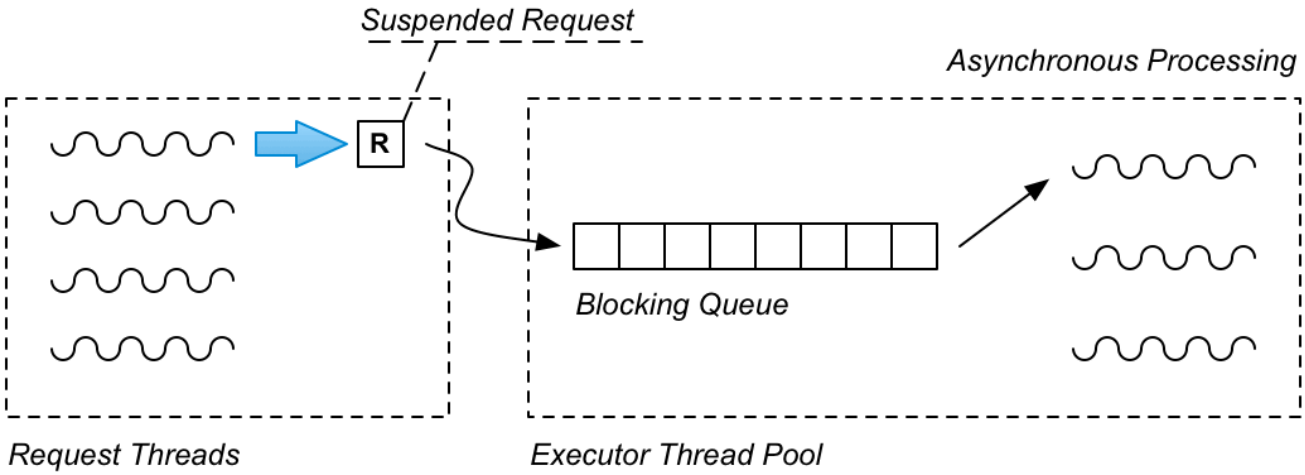
**注意**

请注意，即使在服务器端启用了异步处理，客户端仍会被阻止，直到它从服务器收到响应为止。如果要在客户端中看到异步行为，您必须实施客户端异步处理。请参阅 第 49.6 节 “客户端上的异步处理”。

#### 用于异步处理的基本模型

图 48.1 “同步处理的线程模型” 显示了在服务器端异步处理的基本模型概述。

图 48.1. 同步处理的线程模型




简而言之，请求在异步模型中按如下方式进行处理：

1. 在请求线程中调用异步资源方法（并接收对 **AsyncResponse** 对象的引用，稍后需要发送响应）。

2. 资源方法将暂停的请求封装在 **Runnable** 对象中，其中包含处理请求所需的所有信息和处理逻辑。

3. 资源方法将 Runnable 对象推送到 executor 线程池的阻塞队列中。

4. 资源方法现在可以返回，从而释放请求线程。

5. 当 Runnable 对象进入队列的顶部时，它由 executor 线程池中的一个线程处理。然后，使用封装的 AsyncResponse 对象将响应发回到客户端。

### 使用 Java executor 实现的线程池

java.util.concurrent API 是一个强大的 API，可让您非常轻松地创建完整的线程池实现。在 Java 并发 API 的术语中，线程池称为 executor。它只需要一行代码来创建完整的工作线程池，包括工作线程和提供它们的阻塞队列。

例如，要创建一个完整的工作线程池，如 图 48.1 “同步处理的线程模型” 中显示的 Executor Thread Pool，请创建一个 java.util.concurrent.Executor 实例，如下所示：

```
Executor executor = new ThreadPoolExecutor(
    5,                          // Core pool size
    5,                          // Maximum pool size
    0,                          // Keep-alive time
    TimeUnit.SECONDS,                   // Time unit
    new ArrayBlockingQueue<Runnable>(10)  // Blocking queue
);
```

此构造器创建一个具有五个线程的新线程池，由单个阻塞队列 fed 处理，最多可保存 10 个 可运行的对象。要将任务提交到线程池，请调用 executor.execute 方法，传递对 Runnable 对象（封装异步任务）的引用。

### 定义异步资源方法

要定义异步的资源方法，请使用 @Suspended 注释注入 javax.ws.rs.container.AsyncResponse 类型的参数，并确保方法返回 void。例如：

```
// Java
...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
```

```
@Path("/bookstore")
public class BookContinuationStore {
    ...
    @GET
    @Path("{id}")
    public void handleRequestInPool(@PathParam("id") String id,
                                    @Suspended AsyncResponse response) {
        ...
    }
    ...
}
```

请注意，资源方法必须返回 void，因为注入的 AsyncResponse 对象将在以后返回响应。

### AsyncResponse 类

javax.ws.rs.container.AsyncResponse 类在传入客户端连接上提供了一个抽象处理。当 AsyncResponse 对象注入资源方法时，底层 TCP 客户端连接最初处于 暂停状态。稍后，当您准备好返回响应时，您可以通过调用 AsyncResponse 实例在 AsyncResponse 实例中重新激活底层 TCP 客户端连接并返回响应。或者，如果您需要中止调用，您可以在 AsyncResponse 实例上调用 cancel。

### 将暂停请求封装为 Runnable

在 图 48.1 “同步处理的线程模型” 中显示的异步处理场景中，您将暂停的请求推送到队列，从中可以在稍后由专用线程池处理它。但是，为了实现这种方法，您需要有某种方式 在对象中封装 暂停的请求。暂停的请求对象需要封装以下事项：

- 传入请求中的参数（若有）。
- AsyncResponse 对象，它为传入的客户端连接提供句柄，以及发回响应的方法。
- 调用的逻辑。

封装这些事项的一种便捷方法是定义一个 可运行的 类来代表暂停的请求，其中 Runnable.run （） 方法封装了调用的逻辑。执行此操作的最简单方法是实现 Runnable 作为 本地类，如下例所示。

### 异步处理示例

要实现异步处理场景，资源方法的实现必须将 Runnable 对象（代表暂停的请求）传递给 executor 线程池。在 Java 7 和 8 中，您可以利用一些 novel 语法将 Runnable 类定义为本地类，如下例所示：

```
// Java
package org.apache.cxf.systest.jaxrs;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executor;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.CompletionCallback;
import javax.ws.rs.container.ConnectionCallback;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

import org.apache.cxf.phase.PhaseInterceptorChain;

@Path("/bookstore")
public class BookContinuationStore {

    private Map<String, String> books = new HashMap<String, String>();
    private Executor executor = new ThreadPoolExecutor(5, 5, 0, TimeUnit.SECONDS,
                                    new ArrayBlockingQueue<Runnable>(10));

    public BookContinuationStore() {
        init();
    }
    ...
    @GET
    @Path("{id}")
    public void handleRequestInPool(final @PathParam("id") String id,
                                    final @Suspended AsyncResponse response) {
        executor.execute(new Runnable() {
            public void run() {
                // Retrieve the book data for 'id'
                // which is presumed to be a very slow, blocking operation
                // ...
                bookdata = ...
                // Re-activate the client connection with 'resume'
                // and send the 'bookdata' object as the response
                response.resume(bookdata);
            }
        });
    }
    ...
}
```

请注意，资源方法参数 id 和 response 如何直接传递给 Runnable 本地类的定义。这个特殊语法可让您直接在 Runnable.run （） 方法中使用资源方法参数，而无需在本地类中定义对应的字段。

重要

要使这种特殊语法正常工作，资源方法参数 必须 声明为 最终 （这意味着在方法实施中不得更改这些资源）。

### 48.5.2. 超时和超时处理程序

#### 概述

异步处理模型还支持在 REST 调用时进行超时。默认情况下，超时会导致 HTTP 错误响应发送到客户端。但是，您也可以选择注册超时处理程序回调，供您自定义超时事件的响应。

#### 在没有处理程序的情况下设置超时示例

要定义简单的调用超时，而不指定超时处理程序，在 AsyncResponse 对象中调用 setTimeout 方法，如下例所示：

```
// Java
// Java
...
import java.util.concurrent.TimeUnit;
...
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
public class BookContinuationStore {
    ...
    @GET
    @Path("/books/defaulttimeout")
    public void getBookDescriptionWithTimeout(@Suspended AsyncResponse async) {
        async.setTimeout(2000, TimeUnit.MILLISECONDS);
        // Optionally, send request to executor queue for processing
        // ...
    }
    ...
}
```

请注意，您可以使用 java.util.concurrent.TimeUnit 类中的任何时间单位指定超时值。前面的示例没有显示将请求发送到 executor 线程池的代码。如果您只想测试超时的行为，您可以在资源方法正文中包含对 async.SetTimeout 的调用，并且每次调用时都会触发超时。

AsyncResponse.NO_TIMEOUT 值代表一个无限超时。

### 默认超时行为

默认情况下，如果触发了调用超时，JAX-RS 运行时会引发 ServiceUnavailableException 异常，并发回一个状态为 503 的 HTTP 错误响应。

### TimeoutHandler 接口

如果要自定义超时行为，您必须通过实施 TimeoutHandler 接口来定义超时处理器：

```
// Java
package javax.ws.rs.container;

public interface TimeoutHandler {
    public void handleTimeout(AsyncResponse asyncResponse);
}
```

当您在实现类中覆盖 handleTimeout 方法时，您可以选择以下方法来处理超时：

- 通过调用 asyncResponse.cancel 方法取消响应。
- 使用响应值调用 asyncResponse.resume 方法来发送响应。
- 通过调用 asyncResponse.setTimeout 方法来扩展等待周期。（例如，若要等待 10 秒，您可以调用 asyncResponse.setTimeout (10, TimeUnit.SECONDS)）。

### 使用处理程序设置超时示例

要使用超时处理程序定义调用超时，在 AsyncResponse 对象中调用 setTimeout 方法和 setTimeoutHandler 方法，如下例所示：

```
// Java
```

```
...
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
public class BookContinuationStore {
    ...
    @GET
    @Path("/books/cancel")
    public void getBookDescriptionWithCancel(@PathParam("id") String id,
                                             @Suspended AsyncResponse async) {
        async.setTimeout(2000, TimeUnit.MILLISECONDS);
        async.setTimeoutHandler(new CancelTimeoutHandlerImpl());
        // Optionally, send request to executor queue for processing
        // ...
    }
    ...
}
```

**其中，本例注册 CancelTimeoutHandlerImpl 超时处理程序的实例，以处理调用超时。**

**使用超时处理程序取消响应**

**CancelTimeoutHandlerImpl 超时处理程序定义如下：**

```
// Java
...
import javax.ws.rs.container.AsyncResponse;
...
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
public class BookContinuationStore {
    ...
    private class CancelTimeoutHandlerImpl implements TimeoutHandler {

        @Override
        public void handleTimeout(AsyncResponse asyncResponse) {
            asyncResponse.cancel();
        }

    }
    ...
}
```

在 AsyncResponse 对象上调用 cancel 的影响是向客户端发送 HTTP 503 (服务不可用)错误响应。您可以选择为 cancel 方法指定一个参数( int 或 java.util.Date 值)，它将用于在响应消息中设置 Retry-After: HTTP 标头。但是，客户端通常会忽略 Retry-After: 标头。

#### 在 Runnable 实例中处理取消的响应

如果您已将暂停请求封装为 Runnable 实例，该实例在 executor 线程池中排队以处理，您可能会发现，当线程池要处理请求时，您可能会发现 AsyncResponse 已取消。因此，您需要在 Runnable 实例中添加一些代码，这样就可以 使用取消的 AsyncResponse 对象来应对它。例如：

```
// Java
...
@Path("/bookstore")
public class BookContinuationStore {
    ...
    private void sendRequestToThreadPool(final String id, final AsyncResponse response) {

        executor.execute(new Runnable() {
            public void run() {
                if ( !response.isCancelled() ) {
                    // Process the suspended request ...
                    // ...
                }
            }
        });

    }
    ...
}
```

### 48.5.3. 处理丢弃的连接

#### 概述

您可以添加回调来处理客户端连接丢失的情况。

#### ConnectionCallback 接口

要为丢弃的连接添加回调，您必须实施 javax.ws.rs.container.ConnectionCallback 接口，其定义如下：

```
// Java
package javax.ws.rs.container;
```

```
public interface ConnectionCallback {
    public void onDisconnect(AsyncResponse disconnected);
}
```

**注册连接回调**

在实施连接回调后，必须通过调用其中一个寄存器方法来将其注册到当前的 AsyncResponse 对象。例如，要注册类型为 MyConnectionCallback 的连接回调：

```
asyncResponse.register(new MyConnectionCallback());
```

**连接回调的典型场景**

通常，实施连接回调的主要原因是释放与丢弃客户端连接关联的资源（您可以使用 AsyncResponse 实例作为键来识别需要释放的资源）。

### 48.5.4. 注册回调

**概述**

您可以选择将回调添加到 AsyncResponse 实例，以便在调用完成后获得通知。当调用此回调时，处理中有两个替代点，其中之一：

- 请求处理完成后，响应已发送到客户端，或者
- 请求处理完成后，一个未映射的 Throwable 会被传播到托管 I/O 容器。

**CompletionCallback 接口**

要添加完成回调，您必须实施 javax.ws.rs.container.CompletionCallback 接口，其定义如下：

```
// Java
package javax.ws.rs.container;

public interface CompletionCallback {
    public void onComplete(Throwable throwable);
}
```

通常，可丢弃 的参数为 null。但是，如果请求处理导致异常异常，则 抛出 包含未映射的异常实例。

*注册完成回调*

*在实施完成回调后，您必须通过调用其中一个寄存器方法来将其注册到当前的 AsyncResponse 对象。例如，要注册类型为 MyCompletionCallback 的完成回调：*

```
asyncResponse.register(new MyCompletionCallback());
```

# 第 49 章 JAX-RS 2.0 客户端 API

**摘要**

JAX-RS 2.0 定义功能齐全的客户端 API，可用于进行 REST 调用或任何 HTTP 客户端调用。这包括一个流畅的 API（简化构建请求）、用于解析消息的框架（基于称为 实体提供程序的插件），并支持客户端上的异步调用。

## 49.1. JAX-RS 2.0 客户端 API 简介

**概述**

JAX-RS 2.0 为 JAX-RS 客户端定义了一个流畅的 API，允许您逐步构建 HTTP 请求逐步，然后使用适当的 HTTP 动词(GET、POST、PUT 或 DELETE)调用请求。

**注意**

也可以在蓝图 XML 或 Spring XML 中定义 JAX-RS 客户端（使用 jaxrs:client 元素）。有关此方法的详情，请参考 第 18.2 节 “配置 JAX-RS 客户端端点”。

**依赖项**

要在应用程序中使用 JAX-RS 2.0 客户端 API，您必须将以下 Maven 依赖项添加到项目的 pom.xml 文件中：

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-client</artifactId>
  <version>3.3.6.fuse-7_13_0-00015-redhat-00001</version>
</dependency>
```

如果您计划使用异步调用功能（请参阅 第 49.6 节 “客户端上的异步处理”），您还需要以下 Maven 依赖项：

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-hc</artifactId>
  <version>3.3.6.fuse-7_13_0-00015-redhat-00001</version>
</dependency>
```

**客户端 API 软件包**

JAX-RS 2.0 客户端接口和类位于以下 Java 软件包中：

```
javax.ws.rs.client
```

在开发 JAX-RS 2.0 Java 客户端时，您通常还需要从 core 软件包访问类：

```
javax.ws.rs.core
```

### 简单的客户端请求示例

以下代码片段显示了一个简单示例，其中 JAX-RS 2.0 客户端 API 用于调用 http://example.org/bookstore JAX-RS 服务，使用 GET HTTP 方法调用：

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
Response res = client.target("http://example.org/bookstore/books/123")
    .request("application/xml").get();
```

### Fluent API

JAX-RS 2.0 客户端 API 设计为流畅的 API （有时称为域特定语言）。在流畅的 API 中，在单个声明中调用 Java 方法链，因此 Java 方法类似于来自简单语言的命令。在 JAX-RS 2.0 中，fluent API 用于构建和调用 REST 请求。

### 进行 REST 调用的步骤

使用 JAX-RS 2.0 客户端 API 时，在一系列步骤中构建并调用客户端调用，如下所示：

1. 引导客户端。
2. 配置目标。
3. 构建并发出调用。

4. 解析响应。

引导客户端

第一步是通过创建 javax.ws.rs.client.Client 对象来引导客户端。此 客户端 实例是一个相对重量的对象，它代表了支持 JAX-RS 客户端所需的技术堆栈（可能包括拦截器和其他 CXF 功能）。理想情况下，您应该在可以时重新使用客户端对象，而不是创建新对象。

要创建新的 Client 对象，请在 ClientBuilder 类上调用静态方法，如下所示：

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
...
Client client = ClientBuilder.newClient();
...
```

配置目标

通过配置目标，您可以有效地定义用于 REST 调用的 URI。以下示例演示了如何定义基本 URI、基础，然后使用 path (String) 方法向基本 URI 添加额外的路径片段：

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
```

构建并发出调用

这实际上分为两个步骤：首先，您构建了 HTTP 请求（包括标头、可接受的介质类型等）；另外，您可以调用相关的 HTTP 方法（可选，如果需要请求消息正文）。

例如，要创建和调用接受 application/xml 介质类型的请求：

```
// Java
import javax.ws.rs.core.Response;
...
Response resp = books.resolveTemplate("id", "123").request("application/xml").get();
```

### 解析响应

最后，您需要解析在上一步中获取的 resp ose、resp。通常，响应以 javax.ws.rs.core.Response 对象的形式返回，它封装 HTTP 标头和其他 HTTP 元数据，以及 HTTP 消息正文（若有）。

如果要以 String 格式访问返回的 HTTP 消息，您可以通过使用 String.class 参数调用 readEntity 方法来轻松这样做，如下所示：

```
// Java
...
String msg = resp.readEntity(String.class);
```

您可以通过将 String.class 指定为 readEntity 的参数，始终将响应的消息正文作为字符串访问。有关消息正文的更常规转换或转换，您可以提供一个 实体供应商 来执行转换。如需了解更多详细信息，请参阅 第 49.4 节 “解析请求和响应”。

## 49.2. 构建客户端目标

### 概述

创建初始 客户端 实例后，下一步是构建请求 URI。WebTarget 构建器类允许您配置 URI 的所有方面，包括 URI 路径和查询参数。

### WebTarget builder 类

javax.ws.rs.client.WebTarget 构建器类提供 fluent API 的一部分，可让您为请求构建 REST URI。

### 创建客户端目标

要创建 WebTarget 实例，请在 javax.ws.rs.client.Client 实例上调用其中一个 目标 方法。例如：

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
```

### 基本路径和路径片段

您可以使用 目标 方法指定所有 go 的完整路径；或者，您可以指定一个基本路径，然后使用 目标 方法

和路径方法的组合添加 路径 片段。将基本路径与路径段结合使用的好处是，您可以轻松地重复使用基本路径 WebTarget 对象来对稍有不同的目标进行多次调用。例如：

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget headers     = base.path("bookheaders");
// Now make some invocations on the 'headers' target...
...
WebTarget collections = base.path("collections");
// Now make some invocations on the 'collections' target...
...
```

### URI 模板参数

目标路径的语法也支持 URI 模板参数。也就是说，路径片段可以使用模板参数 {param} 来初始化，后者随后被解析为一个指定值。例如：

```
// Java
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
Response resp = books.resolveTemplate("id", "123").request("application/xml").get();
```

其中 resolveTemplate 方法将路径段 {id} 替换为值 123。

### 定义查询参数

查询参数可以附加到 URI 路径，其中查询参数的开头由单个 ? 字符标记。这个机制可让您使用语法设置一系列名称/值对： ?name1=value1&name2=value2&…

WebTarget 实例允许您使用 queryParam 方法定义查询参数，如下所示：

```
// Java
WebTarget target = client.target("http://example.org/bookstore/")
                         .queryParam("userId","Agamemnon")
                         .queryParam("lang","gr");
```

### 定义列表参数

matrix 参数与查询参数稍相似，但不是广泛支持，并使用不同的语法。要在 WebTarget 实例上定义 matrix 参数，请调用 matrixParam (String, Object) 方法。

## 49.3. 构建客户端调用

### 概述

构建目标 URI 后，使用 WebTarget 构建器类，下一步是配置请求的其他方面，如 HTTP 标头、Cookie 等等，因此使用 Invocation.Builder 类。构建调用的最后一步是调用适当的 HTTP 动词(GET、POST、PUT 或 DELETE)，并提供消息正文（如果需要）。

### invocation.Builder 类

javax.ws.rs.client.Invocation.Builder builder 类提供了 fluent API 的一部分，可让您构建 HTTP 消息的内容并调用 HTTP 方法。

#### 创建调用构建器

要创建 Invocation.Builder 实例，请在 javax.ws.rs.client.WebTarget 实例中调用其中一个 请求 方法。例如：

```
// Java
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.client.Invocation.Builder;
...
WebTarget books = client.target("http://example.org/bookstore/books/123");
Invocation.Builder invbuilder = books.request();
```

#### 定义 HTTP 标头

您可以使用标头方法在请求消息中添加 HTTP 标头，如下所示：

```
Invocation.Builder invheader = invbuilder.header("From", "fionn@example.org");
```

#### 定义 Cookie

您可以使用 cookie 方法在请求消息中添加 Cookie，如下所示：

```
Invocation.Builder invcookie = invbuilder.cookie("myrestclient", "123xyz");
```

*定义属性*

*您可以使用属性方法在此请求上下文中设置属性，如下所示：*

```
Invocation.Builder invproperty = invbuilder.property("Name", "Value");
```

*定义可接受的介质类型、语言或编码*

*您可以定义可接受的介质类型、语言或编码，如下所示：*

```
Invocation.Builder invmedia = invbuilder.accept("application/xml")
                                .acceptLanguage("en-US")
                                .acceptEncoding("gzip");
```

*调用 HTTP 方法*

*构建 REST 调用的过程通过调用 HTTP 方法终止，该方法执行 HTTP 调用。可以调用以下方法（从 javax.ws.rs.client.SyncInvoker 基础类）调用：*

```
get
post
delete
put
head
trace
options
```

*如果此列表中没有调用的特定 HTTP 动词，您可以使用通用方法方法调用任何 HTTP 方法。*

*输入的响应*

*所有 HTTP 调用方法都提供未输入的变体和类型变体（取一个额外的参数）。如果您使用默认 get（） 方法调用请求（无参数），则从调用中返回 javax.ws.rs.core.Response 对象。例如：*

```
Response res = client.target("http://example.org/bookstore/books/123")
                .request("application/xml").get();
```

*但是，也可以使用 get (Class<T>) 方法要求以特定类型返回响应。例如，要调用请求并要求以 BookInfo 对象返回响应：*

```
BookInfo res = client.target("http://example.org/bookstore/books/123")
                .request("application/xml").get(BookInfo.class);
```

但是，为了正常工作，您必须将合适的 实体供应商 注册到 Client 实例，该实例能够将响应格式 application/xml 映射到请求的类型。有关实体供应商的详情，请参阅 第 49.4 节 “解析请求和响应”。

在发布或放置中指定传出消息

对于在请求中包含消息正文（如 POST 或 PUT）的 HTTP 方法，您必须将消息正文指定为方法的第一个参数。邮件正文必须指定为 javax.ws.rs.client.Entity 对象，其中 实体 封装消息内容及其关联的介质类型。例如，要调用 POST 方法，其中消息内容作为 String 类型提供：

```
import javax.ws.rs.client.Entity;
...
Response res = client.target("http://example.org/bookstore/registerbook")
                .request("application/xml")
                .put(Entity.entity("Red Hat Install Guide", "text/plain"));
```

如有必要，Entity.entity （） 构造器方法将自动使用注册的实体提供程序将提供的消息实例映射到指定的介质类型。始终可以将消息正文指定为一个简单的 String 类型。

延迟调用

您可以选择创建 javax.ws.rs.client.Invocation 对象，而不必立即调用 HTTP 请求（例如，通过调用 get （） 方法）。Invocation 对象 封装 待处理调用的所有详情，包括 HTTP 方法。

以下方法可用于构建 Invocation 对象：

```
buildGet
buildPost
buildDelete
buildPut
build
```

例如，要创建一个 GET Invocation 对象并在以后调用它，您可以使用类似如下的代码：

```
import javax.ws.rs.client.Invocation;
import javax.ws.rs.core.Response;
...
Invocation getBookInfo = client.target("http://example.org/bookstore/books/123")
                .request("application/xml").buildGet();
```

```
...
// Later on, in some other part of the application:
Response = getBookInfo.invoke();
```

异步调用

JAX-RS 2.0 客户端 API 支持客户端上的异步调用。要进行异步调用，只需在以下 request () 链中调用 async () 方法。例如：

```
Future<Response> res = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get();
```

当您进行异步调用时，返回的值是一个 java.util.concurrent.Future 对象。有关异步调用的详情，请参考 第 49.6 节 “客户端上的异步处理”。

## 49.4. 解析请求和响应

概述

进行 HTTP 调用的一个重要方面是客户端必须能够解析传出请求消息和传入响应。在 JAX-RS 2.0 中，密钥概念是 Entity 类，它代表带有介质类型标记的原始消息。要解析原始消息，您可以注册多个 实体供应商，该用户能够将介质类型转换为或从特定的 Java 类型转换。

换句话说，在 JAX-RS 2.0 上下文中，实体表示原始消息，实体提供程序是提供解析原始消息的功能（基于介质类型）。

实体

Entity 是由元数据（媒体类型、语言和编码）增强的消息正文。Entity 实例以原始格式保存消息，并与特定介质类型相关联。要将 实体 对象的内容转换为 Java 对象，您需要 实体提供程序，该提供程序 能够将给定介质类型映射到所需的 Java 类型。

变体

javax.ws.rs.core.Variant 对象封装与 实体关联的元数据，如下所示：

- 媒体类型，

- 语言，
- 编码。

实际上，您可以将实体视为由 HTTP 消息内容组成，由 Variant 元数据增强。

### 实体供应商

实体提供程序是一个类，提供介质类型和 Java 类型之间的映射功能。实际上，您可以将实体供应商视为一个类，它提供解析特定介质类型的消息（或者可能有多个介质类型）。实体供应商有两个不同的变体：

**MessageBodyReader**

提供从介质类型映射到 Java 类型的能力。

**MessageBodyWriter**

提供从 Java 类型映射到介质类型的能力。

### 标准实体供应商

以下 Java 和介质类型组合的实体供应商作为标准提供：

**byte[]**

所有介质类型( （条件为 )。

**java.lang.String**

所有介质类型( （条件为 )。

**java.io.InputStream**

所有介质类型( （条件为 )。

**java.io.Reader**

所有介质类型( （条件为 )。

*java.io.File*

*所有介质类型( （条件为 )。*

*javax.activation.DataSource*

*所有介质类型( （条件为 )。*

*javax.xml.transform.Source*

*XML 类型(text/xml,application/xml,, 和 media type of the application ldapsearch+xml)。*

*javax.xml.bind.JAXBElement 和 application-supplied JAXB 类*

*XML 类型(text/xml,application/xml,, 和 media type of the application ldapsearch+xml)。*

*MultivaluedMap<String,String>*

*表单内容(application/x-www-form-urlencoded)。*

*StreamingOutput*

*所有介质类型(*authorize), MessageBodyWriter only.*

*java.lang.Boolean,java.lang.Character,java.lang.Number*

*仅适用于 text/plain。通过 boxing/unboxing 转换支持相应的原语类型。*

*响应对象*

*默认返回类型是 javax.ws.rs.core.Response 类型，它代表未输入的响应。Response 对象提供对完整的 HTTP 响应的访问，包括消息正文、HTTP 状态、HTTP 标头、媒体类型等。*

*访问响应状态*

*您可以通过 getStatus 方法（返回 HTTP 状态代码）访问响应状态：*

```
int status = resp.getStatus();
```

*或者 getStatusInfo 方法，它还提供 description 字符串：*

```
String statusReason = resp.getStatusInfo().getReasonPhrase();
```

访问返回的标头

您可以使用以下任一方法访问 HTTP 标头：

```
MultivaluedMap<String,Object>
getHeaders()

MultivaluedMap<String,String>
getStringHeaders()

String
getHeaderString(String name)
```

例如，如果您知道 Response 具有 Date 标头，您可以按照以下方式访问它：

```
String dateAsString = resp.getHeaderString("Date");
```

访问返回的 Cookie

您可以使用 getCookies 方法访问 Response 中设置的任何新 Cookie，如下所示：

```
import javax.ws.rs.core.NewCookie;
...
java.util.Map<String,NewCookie> cookieMap = resp.getCookies();
java.util.Collection<NewCookie> cookieCollection = cookieMap.values();
```

访问返回的消息内容

您可以通过调用 Response 对象的 readEntity 方法之一来访问返回的消息内容。readEntity 方法自动调用可用的实体提供程序，将消息转换为请求的类型（指定为 readEntity的第一个参数）。例如，将消息内容作为 String 类型访问：

```
String messageBody = resp.readEntity(String.class);
```

集合返回值

如果您需要将返回的消息作为 Java 通用类型（例如，列表 或 集合 类型）访问返回的消息，您可以使用 javax.ws.rs.core.GenericType<T > 构造来指定请求消息类型。例如：

```
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import javax.ws.rs.core.GenericType;
```

```
import java.util.List;
...
GenericType<List<String>> stringListType = new GenericType<List<String>>() {};

Client client = ClientBuilder.newClient();
List<String> bookNames = client.target("http://example.org/bookstore/booknames")
                .request("text/plain")
                .get(stringListType);
```

### 49.5. 配置客户端端点

#### 概述

可以通过注册和配置功能和提供程序来增加基础 **javax.ws.rs.client.Client** 对象的功能。

#### Example

以下示例显示了将配置为具有日志记录功能、自定义实体提供程序的客户端，并将 **prettyLogging** 属性设置为 **true** ：

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import org.apache.cxf.feature.LoggingFeature;
...
Client client = ClientBuilder.newClient();
client.register(LoggingFeature.class)
      .register(MyCustomEntityProvider.class)
      .property("LoggingFeature.prettyLogging","true");
```

#### 用于注册对象的可配置 API

客户端 类支持配置 **API** 来注册对象，它提供多个寄存器 方法变体。在大多数情况下，您将注册一个类或对象实例，如下例所示：

```
client.register(LoggingFeature.class)
client.register(new LoggingFeature())
```

有关 寄存器 变体的详情，请查看 配置文档 。

#### 您可以在客户端上配置什么？

您可以配置客户端端点的以下方面：

- 功能
- 供应商
- Properties
- 过滤器
- 拦截器

功能

javax.ws.rs.core.Feature 实际上是向 JAX-RS 客户端添加额外的功能或功能的插件。通常，功能会安装一个或多个拦截器，以提供所需的功能。

供应商

供应商是特定类型的客户端插件，提供映射功能。JAX-RS 2.0 规范定义了以下类型的提供程序：

实体供应商

实体提供程序 提供特定介质类型 Java 类型之间的映射功能。如需了解更多详细信息，请参阅 第 49.4 节 “解析请求和响应”。

映射供应商时出现异常

异常映射提供程序 将检查的运行时异常映射到 响应 实例。

上下文供应商

在服务器端使用上下文提供程序，为资源类和其他服务提供商提供上下文。

过滤器

JAX-RS 2.0 过滤器是一个插件，可让您访问消息处理管道的不同点(extension 点)的 URI、标头和各种上下文数据。详情请查看 第 61 章 JAX-RS 2.0 Filters 和 Interceptors。

拦截器

JAX-RS 2.0 拦截器是一个插件，可让您访问请求的消息正文或响应，因为它正在读取或写入。详情请查看 第 61 章 JAX-RS 2.0 Filters 和 Interceptors。

Properties

通过在客户端上设置一个或多个属性，您可以自定义注册的功能或注册提供程序的配置。

其他可配置类型

只能配置 javax.ws.rs.client.Client （和 javax.ws.rs.client.ClientBuilder）对象，以及 WebTarget 对象。当您更改 WebTarget 对象的配置时，底层客户端配置会深度复制，以提供新的 WebTarget 配置。因此，可以在不更改原始 Client 对象的配置的情况下更改 WebTarget 对象的配置。

## 49.6. 客户端上的异步处理

### 概述

JAX-RS 2.0 支持在客户端端处理调用。支持两种不同的异步处理样式：使用 java.util.concurrent.Future<V > 返回值；或者注册调用回调。

### 带有 Future<V> 返回值的异步调用

使用 Future<V > 方法异步处理，您可以异步调用客户端请求，如下所示：

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
Future<Response> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get();
```

```
...
// At a later time, check (and wait) for the response:
Response resp = futureResp.get();
```

您可以使用类似的方法来键入的响应。例如，若要获取类型为 **BookInfo** 的响应：

```
Client client = ClientBuilder.newClient();
Future<BookInfo> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get(BookInfo.class);
...
// At a later time, check (and wait) for the response:
BookInfo resp = futureResp.get();
```

### 带有调用回调的异步调用

您可以按照以下方法定义调用回调（使用 **javax.ws.rs.client.InvocationCallback<RESPONSE>**），而不使用 **future<V>**对象访问返回值，如下所示：

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
import javax.ws.rs.client.InvocationCallback;
...
Client client = ClientBuilder.newClient();
Future<Response> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get(
    new InvocationCallback<Response>() {
        @Override
        public void completed(final Response resp) {
            // Do something when invocation is complete
            ...
        }

        @Override
        public void failed(final Throwable throwable) {
            throwable.printStackTrace();
        }
    });
...
```

您可以使用类似的方法来输入的响应：

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
import javax.ws.rs.client.InvocationCallback;
...
Client client = ClientBuilder.newClient();
Future<BookInfo> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get(
    new InvocationCallback<BookInfo>() {
        @Override
        public void completed(final BookInfo resp) {
            // Do something when invocation is complete
            ...
        }

        @Override
        public void failed(final Throwable throwable) {
            throwable.printStackTrace();
        }
    });
...
```

# 第 50 章 处理例外

## 摘要

在可能的情况下，资源方法捕获的异常应该会导致向请求的消费者返回有用的错误。JAX-RS 资源方法可以抛出 WebApplicationException 异常。您还可以提供 ExceptionMapper<E> 实现来将例外映射到适当的响应。

## 50.1. JAX-RS EXCEPTION 类概述

### 概述

在 JAX-RS 1.x 中，唯一可用的例外是 WebApplicationException。从 JAX-WS 2.0 开始，定义了很多额外的 JAX-RS 异常类。

### JAX-RS 运行时级别异常

以下例外是仅由 JAX-RS 运行时丢弃（也就是说，不得 从应用级别代码抛出这些异常）：

#### ProcessingException

（仅限 JAX-RS 2.0） 可以在请求处理期间或 JAX-RS 运行时中响应期间抛出 javax.ws.rs.ProcessingException。例如，由于过滤器链或拦截器链处理中的错误，可能会抛出这个错误。

#### ResponseProcessingException

（仅 JAX-RS 2.0） javax.ws.rs.client.ResponseProcessingException 是 处理Exception 的子类，当 客户端侧 的 JAX-RS 运行时出现错误时，可以抛出它。

### JAX-RS 应用级别异常

应在应用程序级别代码中抛出（并捕获）以下例外：

#### WebApplicationException

javax.ws.rs.WebApplicationException 是一个通用应用级别 JAX-RS 异常，可以在服务器端的应用程序代码中抛出。此异常类型可以封装 HTTP 状态代码、错误消息和（可选）响应消息。详情请查看 第 50.2 节 “使用 WebApplicationException 异常报告”。

#### ClientErrorException

（仅限 JAX-RS 2.0） javax.ws.rs.ClientErrorException 异常类从 WebApplicationException 中继承，用于封装 HTTP 4xx 状态代码。

ServerErrorException

（仅限 JAX-RS 2.0） javax.ws.rs.ServerErrorException 异常类从 WebApplicationException 中继承，用于封装 HTTP 5xx 状态代码。

RedirectionException

（仅 JAX-RS 2.0） javax.ws.rs.RedirectionException 异常类继承了 WebApplicationException，用于封装 HTTP 3xx 状态代码。

## 50.2. 使用 WEBAPPLICATIONEXCEPTION 异常报告

> errors
> indexterm:[WebApplicationException]

概述

JAX-RS API 引入了 WebApplicationException 运行时异常，为资源方法提供简单方法，可以创建适合 RESTful 客户端使用的异常。WebApplicationException 例外可以包括定义实体正文的 Response 对象，以返回到请求的原始器。它还提供了指定要在不提供实体正文时返回到客户端的 HTTP 状态代码的机制。

创建一个简单的例外

创建 WebApplicationException 异常的最简单方法是使用 no 参数构造器，或将原始异常嵌套在 WebApplicationException 异常中。这两个构造器都会创建一个带有空响应的 WebApplicationException。

当抛出这些构造器之一创建的异常时，运行时会返回带有空实体正文和状态代码 500 Server Error 的响应。

设置返回到客户端的状态代码

当您要返回 500 以外的错误代码时，您可以使用四个 WebApplicationException 结构器之一来指定状态。例 50.1 “创建具有状态代码的 WebApplicationException” 中显示的两个构造器将返回状态取为整数。

> 例 50.1. 创建具有状态代码的 WebApplicationException

> WebApplicationExceptionintstatusWebApplicationExceptionjava.lang.Throwablecauseintstatus

例 50.2 “创建具有状态代码的 WebApplicationException” 中显示的另一个是响应状态，作为 Response.Status 实例。

> 例 50.2. 创建具有状态代码的 WebApplicationException
>
> WebApplicationExceptionjavax.ws.rs.core.Response.StatusstatusWebApplicationExceptionjava.lang.Throwablecausejavax.ws.rs.core.Response.Statusstatus

当抛出这些构造器之一创建的异常时，运行时会返回带有空实体正文和指定状态代码的响应。

提供实体正文

如果您希望发送消息并附带异常，您可以使用其中一个 WebApplicationException 结构器采用 Response 对象。运行时使用 Response 对象来创建发送到客户端的响应。响应中存储的实体映射到消息的实体正文，并且响应的 status 字段映射到消息的 HTTP 状态。

例 50.3 “发送一条例外信息” 显示向包含例外原因的客户端返回文本消息的代码，并将 HTTP 消息状态设置为 409 Conflict。

> 例 50.3. 发送一条例外信息
>
> ```
> import javax.ws.rs.core.Response;
> import javax.ws.rs.WebApplicationException;
> import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;
>
> ...
> ResponseBuilderImpl builder = new ResponseBuilderImpl();
> builder.status(Response.Status.CONFLICT);
> builder.entity("The requested resource is conflicted.");
> Response response = builder.build();
> throw WebApplicationException(response);
> ```

扩展通用异常

可以扩展 WebApplicationException 异常。这样，您可以创建自定义异常并消除一些样板代码。

例 50.4 “扩展 WebApplicationException” 显示一个新的异常，它会创建一个与 例 50.3 “发送一条例外信息” 中代码类似的响应。

**例 50.4. 扩展 WebApplicationException**

```
public class ConflicteddException extends WebApplicationException
{
  public ConflictedException(String message)
  {
    ResponseBuilderImpl builder = new ResponseBuilderImpl();
    builder.status(Response.Status.CONFLICT);
    builder.entity(message);
    super(builder.build());
     }
}

...
throw ConflictedException("The requested resource is conflicted.");
```

## 50.3. JAX-RS 2.0 EXCEPTION 类型

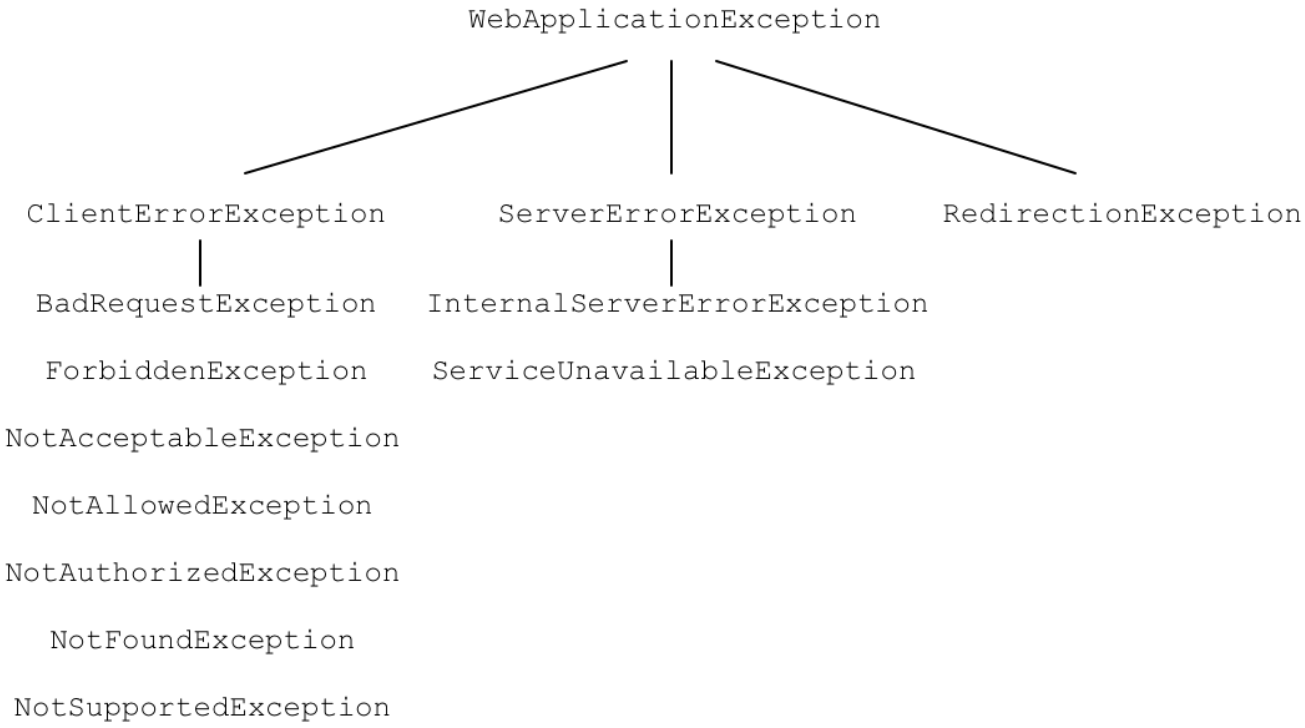
### 概述

JAX-RS 2.0 引入了一些特定的 HTTP 异常类型，您可以在应用程序代码中抛出（和捕获）（除现有的 WebApplicationException 异常类型之外）。这些例外类型可用于嵌套标准 HTTP 状态代码，可以是 HTTP 客户端错误(HTTP 4xx 状态代码)或 HTTP 服务器错误(HTTP 5xx 状态代码)。

### 例外层次结构

图 50.1 “JAX-RS 2.0 应用例外层次结构” 显示 JAX-RS 2.0 中支持的应用级别异常的层次结构。

*图 50.1. JAX-RS 2.0 应用例外层次结构*




***WebApplicationException 类***

*javax.ws.rs.WebApplicationException 异常类（自 JAX-RS 1.x 起可用）位于 JAX-RS 2.0 异常层次结构的基础，在 第 50.2 节 “使用 WebApplicationException 异常报告” 中进行了详细介绍。*

***ClientErrorException 类***

*javax.ws.rs.ClientErrorException 异常类用于封装 HTTP 客户端错误(HTTP 4xx 状态代码)。在应用程序代码中，您可以抛出此异常或其子类之一。*

***ServerErrorException 类***

*javax.ws.rs.ServerErrorException 异常类用于封装 HTTP 服务器错误(HTTP 5xx 状态代码)。在应用程序代码中，您可以抛出此异常或其子类之一。*

***RedirectionException 类***

*javax.ws.rs.RedirectionException 异常类用于封装 HTTP 请求重定向(HTTP 3xx 状态代码)。此类的构造器采用 URI 参数，该参数指定了重定向位置。重定向 URI 可以通过 getLocation （） 方法访问。通常，HTTP 重定向在客户端上是透明的。*

***客户端异常子类***

您可以在 JAX-RS 2.0 应用程序中引发以下 HTTP 客户端异常(HTTP 4xx 状态代码)：

**BadRequestException**

封装 400 Bad Request HTTP 错误状态。

**ForbiddenException**

封装 403 Forbidden HTTP 错误状态。

**NotAcceptableException**

封装 406 Not Acceptable HTTP 错误状态。

**NotAllowedException**

封装 405 方法 Not Allowed HTTP 错误状态。

**NotAuthorizedException**

封装 401 Unauthorized HTTP 错误状态。在以下情况下可能会引发这个异常：

- 客户端没有发送所需的凭证（在 HTTP Authorization 标头中），或者
- 客户端显示凭据，但凭据无效。

**NotFoundException**

封装 404 Not Found HTTP 错误状态。

**NotSupportedException**

封装 415 Unsupported Media Type HTTP 错误状态。

服务器例外子类

您可以在 JAX-RS 2.0 应用程序中引发以下 HTTP 服务器例外(HTTP 5xx 状态代码)：

**InternalServerErrorException**

封装 500 Internal Server Error HTTP 错误状态。

ServiceUnavailableException

封装 503 Service Unavailable HTTP 错误状态。

## 50.4. 将例外映射到 RESPONSES

### 概述

有些实例会抛出 WebApplicationException 异常不现实或不可能。例如，您可能不希望捕获所有可能的异常，然后为它们创建一个 WebApplicationException。您可能还希望使用自定义例外来更轻松地处理应用程序代码。

为了处理这些情况，JAX-RS API 允许您实施自定义异常提供程序，该提供程序生成要发送到客户端的 Response 对象。通过实施 ExceptionMapper<E> 接口来创建自定义异常提供程序。当使用 Apache CXF 运行时注册时，每当抛出类型 E 异常时，将使用自定义供应商。

### 如何选择例外映射程序

两个情况下都会使用例外映射器：

- 当抛出任何异常或其子类时，运行时将检查适当的异常映射器。如果处理特定的异常抛出，则会选择一个异常映射器。如果抛出的具体例外映射程序没有例外，则会选择例外的超级类例外映射器。

- 默认情况下，WebApplicationException 将由默认的映射程序 WebApplicationExceptionMapper 处理。即使注册了额外的自定义映射器，它可能会处理 WebApplicationException 异常（例如，自定义 RuntimeException 映射器），也不会 使用自定义映射器，而是使用 WebApplicationExceptionMapper。

  但是，可以通过在 Message 对象中将 default.wae.mapper.least.specific 属性设为 true 来更改此行为。启用此选项后，默认的 WebApplicationExceptionMapper 会重新分配给最低优先级，以便处理带有自定义例外映射器的 WebApplicationException 异常。例如，如果启用了此选项，可以通过注册自定义 RuntimeException 映射器来捕获 WebApplicationException 异常。请参阅 “为 WebApplicationException 注册异常映射器”一节。

如果没有为异常找到异常，则会将异常嵌套在 ServletException 异常中，并传递到容器运行时。然后，容器运行时将确定如何处理异常。

实施异常映射器

异常映射程序是通过实现 javax.ws.rs.ext.ExceptionMapper<E> 接口来创建的。如 例 50.5 “例外映射器接口” 所示，接口具有单一方法 toResponse （），它将原始异常取为参数并返回 Response 对象。

例 50.5. 例外映射器接口

```
public interface ExceptionMapper<E extends java.lang.Throwable>
{
  public Response toResponse(E exception);
}
```

异常映射程序创建的 Response 对象由运行时处理，就像任何其他 Response 对象一样。生成的对消费者的响应将包含在 Response 对象中封装的状态、标头和实体正文。

异常映射程序实现被视为运行时的供应商。因此，它们必须使用 @Provider 注释进行解码。

如果在构建 Response 对象时出现异常，则运行时会向消费者发送状态为 500 Server Error 的响应。

例 50.6 “将异常映射到响应” 显示一个例外映射程序，它截获 Spring AccessDeniedException 异常，并生成带有 403 Forbidden 状态和空实体正文的响应。

例 50.6. 将异常映射到响应

```
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;

import org.springframework.security.AccessDeniedException;

@Provider
public class SecurityExceptionMapper implements ExceptionMapper<AccessDeniedException>
{

  public Response toResponse(AccessDeniedException exception)
  {
    return Response.status(Response.Status.FORBIDDEN).build();
  }

}
```

运行时会捕获任何 AccessDeniedException 异常，并创建没有实体正文和 403 状态的 Response 对象。然后，运行时会将 Response 对象处理为正常响应。结果是消费者将收到 HTTP 响应，状态为 403。

### 注册异常映射器

在 JAX-RS 应用可以使用异常映射程序之前，必须将异常映射器注册到运行时。异常映射程序使用应用程序配置文件中的 jaxrs:providers 元素与运行时注册。

jaxrs:providers 元素是 jaxrs:server 元素的子级，包含 bean 元素的列表。每个 bean 元素定义一个异常映射器。

例 50.7 “将异常映射器注册到运行时” 显示 JAX-RS 服务器，配置为使用自定义例外映射器 SecurityExceptionMapper。

**例 50.7. 将异常映射器注册到运行时**

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

### 为 WebApplicationException 注册异常映射器

为 WebApplicationException 异常注册例外映射程序是一个特殊情况，因为此异常类型由默认的 WebApplicationExceptionMapper 自动处理。通常，即使您注册了您要处理 WebApplicationException 的自定义映射程序，它仍将继续由默认的 WebApplicationExceptionMapper 处理。要更改此默认行为，您需要将 default.wae.mapper.least.specific 属性设为 true。

例如，以下 XML 代码演示了如何在 JAX-RS 端点上启用 default.wae.mapper.least.specific 属性：

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
    </jaxrs:providers>
```

```
    <jaxrs:properties>
      <entry key="default.wae.mapper.least.specific" value="true"/>
    </jaxrs:properties>
  </jaxrs:server>
</beans>
```

*您还可以在拦截器中设置 **default.wae.mapper.least.specific** 属性，如下例所示：*

```
// Java
public void handleMessage(Message message)
{
    m.put("default.wae.mapper.least.specific", true);
    ...
```

# 第 51 章 实体支持

## 摘要

Apache CXF 运行时支持 MIME 类型和 Java 对象之间的有限数量的映射。开发人员可以通过实施自定义读取器和写入器来扩展映射。自定义读取器和写入器在启动时使用运行时注册。

## 概述

运行时依赖于 JAX-RS MessageBodyReader 和 MessageBodyWriter 实现来序列化 HTTP 消息及其 Java 表示之间的数据。读取器和写入器可以限制他们能够处理的 MIME 类型。

运行时为多个常见映射提供读取器和写入器。如果应用程序需要更多的映射，开发人员可以提供 MessageBodyReader 接口和/或 MessageBodyWriter 接口的自定义实现。自定义读取器和写入器在应用启动时使用运行时注册。

## 原生支持的类型

表 51.1 “原生支持的实体映射” 列出 Apache CXF 提供的实体映射。

表 51.1. 原生支持的实体映射

| Java 类型 | MIME 类型 |
| --- | --- |
| 原语类型 | **text/plain** |
| **java.lang.Number** | **text/plain** |
| **byte[]** | ***/*** |
| **java.lang.String** | ***/*** |
| **java.io.InputStream** | ***/*** |
| **java.io.Reader** | ***/*** |
| **java.io.File** | ***/*** |
| **javax.activation.DataSource** | ***/*** |
| **javax.xml.transform.Source** | **text/xml,application/xml,application/*+xml** |

| Java 类型 | MIME 类型 |
| --- | --- |
| **javax.xml.bind.JAXBElement** | **text/xml,application/xml,application/*+xml** |
| JAXB 注解的对象 | **text/xml,application/xml,application/*+xml** |
| javax.ws.rs.core.MultivaluedMap<String, String> | **application/x-www-form-urlencoded** [a] |
| javax.ws.rs.core.StreamingOutput | */* [b] |

[a] 此映射用于处理 HTML 表单数据。

[b] 这个映射只支持将数据返回到消费者。

### *自定义读取器*

*自定义实体读取器负责将传入的 HTTP 请求映射到服务实施可以操作的 Java 类型中。它们实施 javax.ws.rs.ext.MessageBodyReader 接口。*

*例 51.1 “消息读取器接口” 中显示的接口有两个需要实现的方法：*

**例 51.1. 消息读取器接口**

```
package javax.ws.rs.ext;

public interface MessageBodyReader<T>
{
  public boolean isReadable(java.lang.Class<?> type,
                            java.lang.reflect.Type genericType,
                            java.lang.annotation.Annotation[] annotations,
                            javax.ws.rs.core.MediaType mediaType);

  public T readFrom(java.lang.Class<T> type,
                    java.lang.reflect.Type genericType,
                    java.lang.annotation.Annotation[] annotations,
                    javax.ws.rs.core.MediaType mediaType,
                    javax.ws.rs.core.MultivaluedMap<String, String> httpHeaders,
                    java.io.InputStream entityStream)
  throws java.io.IOException, WebApplicationException;
}
```

#### *isReadable()*

*isReadable （） 方法决定了读取器是否能够读取数据流，并创建正确的实体表示类型。如果读取器可以创建正确的实体类型，则方法返回为 true。*

*表 51.2 “用于确定读取器是否可以生成实体的参数” 描述 isReadable （） 方法的参数。*

*表 51.2. 用于确定读取器是否可以生成实体的参数*

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| **type** | **类<T>** | 指定用于存储实体的对象的实际 Java 类。 |
| **genericType** | 类型 | 指定用于存储实体的对象的 Java 类型。例如，如果要将消息正文转换为 method 参数，则该值将是 **Method.getGenericParameterTypes （）** 方法返回的 method 参数的类型。 |
| **annotations** | Annotation[] | 指定为存储实体而创建的对象声明的注解列表。例如，如果消息正文要转换为 method 参数，这将是 **Method.getParameterAnnotations （）** 方法返回的该参数上的注解。 |
| **mediaType** | **MediatType** | 指定 HTTP 实体的 MIME 类型。 |

***readFrom()***

***readFrom （） 方法读取 HTTP 实体，并将其覆盖到所需的 Java 对象中。如果读取成功，则方法返回包含该实体的创建的 Java 对象。如果在读取输入流时发生错误，方法应抛出 IOException 异常。如果发生错误需要 HTTP 错误响应，应抛出带有 HTTP 响应的 WebApplicationException。***

*表 51.3 “用于读取实体的参数” 描述 readFrom （） 方法的参数。*

*表 51.3. 用于读取实体的参数*

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| **type** | **类<T>** | 指定用于存储实体的对象的实际 Java 类。 |

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| **genericType** | 类型 | 指定用于存储实体的对象的 Java 类型。例如，如果要将消息正文转换为 method 参数，则该值将是 **Method.getGenericParameterTypes （）** 方法返回的 method 参数的类型。 |
| **annotations** | Annotation[] | 指定为存储实体而创建的对象声明的注解列表。例如，如果消息正文要转换为 method 参数，这将是 **Method.getParameterAnnotations （）** 方法返回的该参数上的注解。 |
| **mediaType** | **MediatType** | 指定 HTTP 实体的 MIME 类型。 |
| **httpHeaders** | MultivaluedMap<String, String> | 指定与实体关联的 HTTP 消息标头。 |
| **entityStream** | **InputStream** | 指定包含 HTTP 实体的输入流。 |

***重要***

*这个方法不应关闭输入流。*

*在 MessageBodyReader 实现可用作实体阅读器之前，它必须使用 javax.ws.rs.ext.Provider 注解进行分离。@Provider 注释提醒提供的实施提供了额外功能的运行时。这个实现也必须使用运行时注册，如 "注册阅读器和作者"一节 所述。*

*默认情况下，自定义实体供应商处理所有 MIME 类型。您可以使用 javax.ws.rs.Consumes 注解限制自定义实体读取器将处理的 MIME 类型。@Consumes 注释指定自定义实体提供程序读取的、以逗号分隔的 MIME 类型列表。如果实体不是指定的 MIME 类型，则不会选择实体提供程序作为可能的读取器。*

*例 51.2 "XML 源实体读取器" 显示使用 XML 实体的实体读取器，并将它们存储在 Source 对象中。*

> ***例 51.2. XML 源实体读取器***

```
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamSource;

import org.w3c.dom.Document;
import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Consumes({"application/xml", "application/*+xml", "text/xml", "text/html" })
public class SourceProvider implements MessageBodyReader<Object>
{
  public boolean isReadable(Class<?> type,
                            Type genericType,
                            Annotation[] annotations,
                            MediaType mt)
  {
    return Source.class.isAssignableFrom(type) || XMLSource.class.isAssignableFrom(type);
  }

  public Object readFrom(Class<Object> source,
                         Type genericType,
                         Annotation[] annotations,
                         MediaType mediaType,
                         MultivaluedMap<String, String> httpHeaders,
                         InputStream is)
  throws IOException
  {
    if (DOMSource.class.isAssignableFrom(source))
    {
      Document doc = null;
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder;
      try
      {
        builder = factory.newDocumentBuilder();
        doc = builder.parse(is);
      }
      catch (Exception e)
      {
        IOException ioex = new IOException("Problem creating a Source object");
        ioex.setStackTrace(e.getStackTrace());
        throw ioex;
       }
```

```
        return new DOMSource(doc);
      }
      else if (StreamSource.class.isAssignableFrom(source) ||
Source.class.isAssignableFrom(source))
      {
        return new StreamSource(is);
      }
      else if (XMLSource.class.isAssignableFrom(source))
      {
        return new XMLSource(is);
      }

      throw new IOException("Unrecognized source");
    }
  }
```

### 自定义写入器

自定义实体作者负责将 Java 类型映射到 HTTP 实体。它们实施 **javax.ws.rs.ext.MessageBodyWriter** 接口。

例 51.3 “消息写入器接口” 中显示的接口有三个需要实现的方法：

**例 51.3. 消息写入器接口**

```
package javax.ws.rs.ext;

public interface MessageBodyWriter<T>
{
  public boolean isWriteable(java.lang.Class<?> type,
                             java.lang.reflect.Type genericType,
                             java.lang.annotation.Annotation[] annotations,
                             javax.ws.rs.core.MediaType mediaType);

  public long getSize(T t,
                      java.lang.Class<?> type,
                      java.lang.reflect.Type genericType,
                      java.lang.annotation.Annotation[] annotations,
                      javax.ws.rs.core.MediaType mediaType);

  public void writeTo(T t,
                      java.lang.Class<?> type,
                      java.lang.reflect.Type genericType,
                      java.lang.annotation.Annotation[] annotations,
                      javax.ws.rs.core.MediaType mediaType,
                      javax.ws.rs.core.MultivaluedMap<String, Object> httpHeaders,
```

```
                java.io.OutputStream entityStream)
    throws java.io.IOException, WebApplicationException;
}
```

### *isWriteable()*

***isWriteable （） 方法确定实体写入器是否可以将 Java 类型映射到正确的实体类型。如果写入器可以进行映射，则方法会返回 true。***

***表 51.4 “用于读取实体的参数” 描述 isWritable （） 方法的参数。***

***表 51.4. 用于读取实体的参数***

| 参数 | 类型 | 描述 |
|---|---|---|
| **type** | **类<T>** | 指定正在写入对象的 Java 类。 |
| **genericType** | 类型 | 指定要编写的对象的 Java 类型，通过反映资源方法返回类型或通过检查返回的实例来获取。**GenericEntity** 类（如第 48.4 节 “使用通用类型信息返回实体” 所述）提供对控制这个值的支持。 |
| **annotations** | Annotation[] | 指定返回实体的方法上的注解列表。 |
| **mediaType** | **MediatType** | 指定 HTTP 实体的 MIME 类型。 |

### *getSize()*

***getSize （） 方法在 writeTo （） 之前调用。它返回所写入实体的长度（以字节为单位）。如果返回正值，则该值将写入 HTTP 消息的 Content-Length 标头中。***

***表 51.5 “用于读取实体的参数” 描述 getSize （） 方法的参数。***

***表 51.5. 用于读取实体的参数***

| 参数 | 类型 | 描述 |
|---|---|---|
| **t** | generic | 指定正在写入的实例。 |
| **type** | **类<T>** | 指定正在写入对象的 Java 类。 |

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| **genericType** | 类型 | 指定要编写的对象的 Java 类型，通过反映资源方法返回类型或通过检查返回的实例来获取。**GenericEntity** 类（如 第 48.4 节 “使用通用类型信息返回实体” 所述）提供对控制这个值的支持。 |
| **annotations** | Annotation[] | 指定返回实体的方法上的注解列表。 |
| **mediaType** | **MediatType** | 指定 HTTP 实体的 MIME 类型。 |

***writeTo()***

***writeTo （） 方法将 Java 对象转换为所需的实体类型，并将实体写入输出流。如果在将实体写入输出流时发生错误，则方法应抛出 IOException 异常。如果发生错误需要 HTTP 错误响应，应抛出带有 HTTP 响应的 WebApplicationException。***

***表 51.6 “用于读取实体的参数” 描述 writeTo （） 方法的参数。***

***表 51.6. 用于读取实体的参数***

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| **t** | generic | 指定正在写入的实例。 |
| **type** | **类<T>** | 指定正在写入对象的 Java 类。 |
| **genericType** | 类型 | 指定要编写的对象的 Java 类型，通过反映资源方法返回类型或通过检查返回的实例来获取。**GenericEntity** 类（如 第 48.4 节 “使用通用类型信息返回实体” 所述）提供对控制这个值的支持。 |
| **annotations** | Annotation[] | 指定返回实体的方法上的注解列表。 |
| **mediaType** | **MediatType** | 指定 HTTP 实体的 MIME 类型。 |
| **httpHeaders** | MultivaluedMap<String, Object> | 指定与实体关联的 HTTP 响应标头。 |

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| **entityStream** | **OutputStream** | 指定将实体写入的输出流。 |

*在 MessageBodyWriter 实现可用作实体写入器之前，它必须使用 javax.ws.rs.ext.Provider 注解进行分离。@Provider 注释提醒提供的实施提供了额外功能的运行时。这个实现也必须使用运行时注册，如 "注册阅读器和作者"一节 所述。*

*默认情况下，自定义实体供应商处理所有 MIME 类型。您可以使用 javax.ws.rs.Produces 注解限制自定义实体写器将处理的 MIME 类型。@Produces 注释指定自定义实体提供程序生成的以逗号分隔的 MIME 类型列表。如果实体不是指定的 MIME 类型，则不会选择实体提供程序作为可能的写器。*

*例 51.4 "XML 源实体编写器" 显示接受 Source 对象并生成 XML 实体的实体编写器。*

**例 51.4. XML 源实体编写器**

```
import java.io.IOException;
import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyWriter;
import javax.ws.rs.ext.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;

import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Produces({"application/xml", "application/*+xml", "text/xml" })
public class SourceProvider implements MessageBodyWriter<Source>
{

  public boolean isWriteable(Class<?> type,
                             Type genericType,
                             Annotation[] annotations,
                             MediaType mt)
```

```
    {
      return Source.class.isAssignableFrom(type);
    }

    public void writeTo(Source source,
                        Class<?> clazz,
                        Type genericType,
                        Annotation[] annotations,
                        MediaType mediatype,
                        MultivaluedMap<String, Object> httpHeaders,
                        OutputStream os)
    throws IOException
    {
      StreamResult result = new StreamResult(os);
      TransformerFactory tf = TransformerFactory.newInstance();
      try
      {
        Transformer t = tf.newTransformer();
        t.transform(source, result);
      }
      catch (TransformerException te)
      {
        te.printStackTrace();
        throw new WebApplicationException(te);
      }
    }

    public long getSize(Source source,
                        Class<?> type,
                        Type genericType,
                        Annotation[] annotations,
                        MediaType mt)
    {
      return -1;
    }
  }
```

**注册阅读器和作者**

**在 JAX-RS 应用可以使用任何自定义实体提供程序之前，必须在运行时注册自定义提供程序。提供程序使用应用配置文件中的 jaxrs:providers 元素或使用 JAXRSServerFactoryBean 类通过运行时注册。**

**jaxrs:providers 元素是 jaxrs:server 元素的子级，包含 bean 元素的列表。每个 bean 元素定义一个实体提供程序。**

**例 51.5 “使用运行时注册实体供应商” 显示 JAX-RS 服务器，配置为使用一组自定义实体提供程序。**

**例 51.5. 使用运行时注册实体供应商**

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="isProvider" class="com.bar.providers.InputStreamProvider"/>
      <bean id="longProvider" class="com.bar.providers.LongProvider"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

**JAXRSServerFactoryBean 类是 Apache CXF 扩展，提供对配置 API 的访问。它有一个 setProvider () 方法，允许您将实例化实体供应商添加到应用程序中。例 51.6 “以编程方式注册实体供应商” 显示用于以编程方式注册实体提供程序的代码。**

**例 51.6. 以编程方式注册实体供应商**

```
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
...
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
...
SourceProvider provider = new SourceProvider();
sf.setProvider(provider);
...
```

# 第 52 章 获取和使用上下文信息

**摘要**

上下文信息包括资源 URI、HTTP 标头和其他无法使用其他注入注解的详细信息。Apache CXF 提供特殊的类，将所有可能的上下文信息放入单个对象。

## 52.1. 上下文简介

### 上下文注解

您可以使用 javax.ws.rs.core.Context 注释来指定上下文信息要注入到字段或资源方法参数中。注解某个上下文类型的字段或参数将指示运行时将适当的上下文信息注入注释的字段或参数。

### 上下文类型

表 52.1 “上下文类型” 列出可以注入的上下文信息的类型，以及支持它们的对象。

**表 52.1. 上下文类型**

| 对象 | 上下文信息 |
| --- | --- |
| UriInfo | 完整请求 URI |
| httpHeaders | HTTP 消息标头 |
| Request（请求） | 可用于确定最佳表示变体或确定是否设置了一组条件的信息 |
| securityContext | 有关请求者安全的信息，包括正在使用的身份验证方案，如果请求频道安全，以及用户原则 |

### 可以使用上下文信息的位置

上下文信息可用于 JAX-RS 应用程序的以下部分：

- 资源类
- 资源方法

- 实体供应商
- 例外映射器

影响范围

使用 @Context 注释注入的所有上下文信息都特定于当前请求。在所有情况下，包括实体供应商和异常映射程序都是如此。

添加上下文

JAX-RS 框架允许开发人员扩展可以使用上下文机制注入的信息类型。您可以通过实施 Context<T> 对象并使用运行时注册来添加自定义上下文。

## 52.2. 使用完整请求 URI

摘要

请求 URI 包含大量信息。其中大多数信息都可使用方法参数访问，如 第 47.2.2 节 “从请求 URI 注入数据” 所述，但使用参数会强制处理 URI 的某些限制。使用参数访问 URI 的片段也不提供对完整请求 URI 的资源访问。

您可以通过将 URI 上下文注入资源来提供对完整请求 URI 的访问。URI 作为 UriInfo 对象提供。UriInfo 接口以多种方式处理 URI。它还可以将 URI 提供为 UriBuilder 对象，允许您构建 URI 以返回到客户端。

:experimental:

### 52.2.1. 注入 URI 信息

概述

当属于 UriInfo 对象的 class 字段或 method 参数与 @Context 注释分离时，当前请求的 URI 上下文将注入到 UriInfo 对象中。

Example

*将 URI 上下文注入类字段* 通过注入 URI 上下文来显示带有字段的类。

**将 URI 上下文注入类字段**

```
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo requestURI;
  ...
}
```

### 52.2.2. 使用 URI

**概述**

使用 URI 上下文的一个主要优点是它提供对服务的基本 URI 和所选资源 URI 的路径段的访问。此信息可用于很多目的，如基于 URI 或计算 URI 作为响应的一部分返回的处理决策。例如，如果请求的基本 URI 包含 .com 扩展，服务可能会决定使用美国美元，以及基础 URI 是否包含 .co.uk 扩展，则可能会决定我们 British Pounds。

UriInfo 接口提供了访问 URI 部分的方法：

- 基本 URI
- 资源路径
- 完整的 URI

### 获取基本 URI

基础 URI 是发布该服务的 root URI。它不包含任何服务的 @Path 注释中指定的 URI 的任何部分。例如，如果实施 例 47.5 “禁用 URI 解码” 中定义的资源的服务已发布到 http://fusesource.org，并且在 http://fusesource.org/montersforhire/nightstalker?12 上发出了一个请求，基本 URI 为 http://fusesource.org。

表 52.2 “访问资源基础 URI 的方法” 描述返回基本 URI 的方法。

表 52.2. 访问资源基础 URI 的方法

| 方法 | Desription |
| --- | --- |
| URIgetBaseUri | 将服务的基本 URI 返回为 **URI** 对象。 |
| UriBuildergetBaseUriBuilder | 将基本 URI 返回为 **javax.ws.rs.core.core.UriBuilder** 对象。**UriBuilder** 类可用于为服务实施的其他资源创建 URI。 |

### 获取路径

请求 URI 的路径 部分是用于选择当前资源的 URI 部分。它不包括基本 URI，但不包含任何 URI 模板变量和 URI 中包含的列表参数。

路径的值取决于所选资源。例如，在 获取资源路径 中定义的资源的路径将是：

- rootPath — /monstersforhire/
- getterPath — /mostersforhire/nightstalker

  GET 请求在 /monstersforhire/nightstalker 上发出。
- putterPath — /mostersforhire/911

  PUT 请求是在 /monstersforhire/911 上进行的。

### 获取资源路径

```
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo rootUri;

  ...

  @GET
  public List<Monster> getMonsters(@Context UriInfo getUri)
  {
    String rootPath = rootUri.getPath();
    ...
  }

  @GET
  @Path("/{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @Context UriInfo getUri)
  {
    String getterPath = getUri.getPath();
    ...
  }

  @PUT
  @Path("/{id}")
  public void addMonster(@Encoded @PathParam("type") String type,
                         @Context UriInfo putUri)
  {
    String putterPath = putUri.getPath();
    ...
  }
  ...
}
```

*表 52.3 “访问资源路径的方法” 描述返回资源路径的方法。*

**表 52.3. 访问资源路径的方法**

| 方法 | Desription |
| --- | --- |
| **字符串getPath** | 将资源的路径返回为解码的 URI。 |
| **字符串getPath布尔值解码** | 返回资源的路径。指定 **false** 禁用 URI 解码。 |

| 方法 | Desription |
| --- | --- |
| **List<PathSegment>getPathSegments** | 将解码的路径返回为 javax.ws.rs.core.PathSegment 对象列表。路径的每个部分（包括列表参数）都放在列表中的唯一条目中。<br><br>例如，资源路径 **框/round#tall** 将导致包含三个条目的列表： **box**、**round**、和 **tall**。 |
| **List<PathSegment>getPathSegmentsbooleandecode** | 将路径返回为 javax.ws.rs.core.PathSegment 对象列表。路径的每个部分（包括列表参数）都放在列表中的唯一条目中。指定 **false** 禁用 URI 解码。<br><br>例如，资源路径 **框#tall/round** 将导致含有三个条目的列表： **box**、**tall** 和 **round**。 |

*获取完整请求 URI*

*表 52.4 “访问完整请求 URI 的方法” 描述返回完整请求 URI 的方法。您可以选择返回请求 URI 或资源的绝对路径。区别在于，请求 URI 包含附加到 URI 的任何查询参数，绝对路径不包括查询参数。*

*表 52.4. 访问完整请求 URI 的方法*

| 方法 | Desription |
| --- | --- |
| **URIgetRequestUri** | 返回完整的请求 URI，包括查询参数和列表参数，作为 **java.net.URI** 对象。 |
| **UriBuildergetRequestUriBuilder** | 返回完整的请求 URI，包括查询参数和列表参数，作为 **javax.ws.rs.UriBuilder** 对象。**UriBuilder** 类可用于为服务实施的其他资源创建 URI。 |
| **URIgetAbsolutePath** | 返回完整的请求 URI，包括列表参数，作为 **java.net.URI** 对象。绝对路径不包括查询参数。 |
| **UriBuildergetAbsolutePathBuilder** | 返回完整的请求 URI，包括列表参数，作为 **javax.ws.rs.UriBuilder** 对象。绝对路径不包括查询参数。 |

*对于使用 URI http://fusesource.org/montersforhire/nightstalker?12 发出请求，getRequestUri （） 方法将返回 http://fusesource.org/montersforhire/nightstalker?12。getAbsolutePath （） 方法将返回 http://fusesource.org/montersforhire/nightstalker。*

### *52.2.3. 获取 URI 模板变量的值*

概述

如 “设置路径”一节 所述，资源路径可以包含动态绑定到值的变量片段。这些变量路径片段通常用作资源方法的参数，如 “从 URI 的路径获取数据”一节 所述。但是，您也可以通过 URI 上下文访问它们。

获取路径参数的方法

UriInfo 接口提供了两种方法，如 例 52.1 “从 URI 上下文返回路径参数的方法” 所示，它返回一个路径参数列表。

例 52.1. 从 URI 上下文返回路径参数的方法

MultivaluedMap<java.lang.String, java.lang.String>getPathParametersMultivaluedMap<java.lang.String, java.lang.String>getPathParameters布尔值解码

不使用任何参数的 getPathParameters （） 方法会自动解码路径参数。如果要禁用 URI 解码，请使用 getPathParameters (false)。

这些值存储在映射中，将其模板标识符用作键。例如，如果资源的 URI 模板为 /{color}/box/{note}，则返回的映射将有两个带有键 颜色 的条目，并注意。

Example

例 52.2 “从 URI 上下文中提取路径参数” 显示使用 URI 上下文检索路径参数的代码。

例 52.2. 从 URI 上下文中提取路径参数

```
import javax.ws.rs.Path;
import javax.ws.rs.Get;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.core.MultivaluedMap;

@Path("/monstersforhire/")
public class MonsterService
```

```
  @GET
  @Path("/{type}/{size}")
  public Monster getMonster(@Context UriInfo uri)
  {
    MultivaluedMap paramMap = uri.getPathParameters();
    String type = paramMap.getFirst("type");
    String size = paramMap.getFirst("size");
  }
}
```

# 第 53 章 注解继承

## 摘要

JAX-RS 注释可以通过子类和实施注释的类来继承。继承机制允许子类和实施类覆盖从其父级继承的注解。

## 概述

继承是 Java 中更强大的机制之一，因为它允许开发人员创建通用对象，然后可以专门满足特定的需求。JAX-RS 允许用于将类映射到资源从超级类继承的标注来保持此电源。

JAX-RS 的注解继承也扩展为支持接口。实施类继承其实施的接口中使用的 JAX-RS 注释。

JAX-RS 继承规则提供覆盖继承注释的机制。但是，无法从超级类或接口继承的构造中完全删除 JAX-RS 注释。

### 继承规则

资源类从它实施的接口继承任何 JAX-RS 注释。资源类也从它们扩展的任何超级类继承任何 JAX-RS 注释。从超级类继承的注解优先于从接口继承的注解。

在 例 53.1 “注解继承” 中显示的代码示例，K Kaijin 类的 getMonster （） 方法从 Kaiju 接口继承了 @Path、@GET、@GET、@PathParam 注解。

**例 53.1. 注解继承**

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  public Monster getMonster(@PathParam("id") int id);
  ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{
```

```
  public Monster getMonster(int id)
  {
    ...
  }
  ...
}
```

### 覆盖继承的注解

覆盖继承的注解与提供新注解一样容易。如果子类或实施类为方法提供自己的 JAX-RS 注释，则忽略该方法的所有 JAX-RS 注释。

在 例 53.2 “覆盖注解继承” 中显示的代码示例中，K Kaijin 类的 getMonster （） 方法不会继承 Kaiju 接口中的任何注解。实施类覆盖 @Produces 注释，该注释导致来自接口的所有注释被忽略。

**例 53.2. 覆盖注解继承**

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  @Produces("text/xml");
  public Monster getMonster(@PathParam("id") int id);
  ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{

  @GET
  @Path("/{id}")
  @Produces("application/octect-stream");
  public Monster getMonster(@PathParam("id") int id)
  {
    ...
  }
  ...
}
```

# 第 54 章 使用 OPENAPI 支持扩展 JAX-RS 端点

**摘要**

CXF OpenApiFeature (org.apache.cxf.jaxrs.openapi.OpenApiFeature)允许您通过使用简单的配置扩展发布的 JAX-RS 服务端点来生成 OpenAPI 文档。

Spring Boot 和 Karaf 实现都支持 OpenApiFeature。

## 54.1. OPENAPIFEATURE 选项

您可以使用 OpenApiFeature 中的以下选项。

表 54.1. OpenApiFeature 操作

| Name | 描述 | default |
|---|---|---|
| **configLocation** | OpenAPI 配置位置 | null |
| **contactEmail** | 联系电子邮件+ | null |
| **contactName** | 联系名称+ | null |
| **contactUrl** | 联系链接+ | null |
| **customr** | customr 类实例 | null |
| **description** | description+ | null |
| **filterClass** | 安全过滤器++ | null |
| **ignoredRoutes** | 扫描所有资源时排除特定路径（请参阅 **scanAllResources**）++ | null |
| **许可证** | 许可证+ | null |
| **licenseUrl** | 许可证 URL+ | null |
| **prettyPrint** | 在生成 openapi.json 时，pretty-print JSON 文档++ | true |
| **propertiesLocation** | 属性文件位置 | **/swagger.properties** |

| Name | 描述 | default |
| --- | --- | --- |
| **readAllResources** | 也读取没有 @Operation++ 的所有操作 | true |
| **resourceClasses** | 必须扫描++ 的资源类列表 | null |
| **resourcePackages** | 必须扫描资源的软件包名称列表++ | null |
| **runAsFilter** | 将该功能作为过滤器运行 | false |
| **扫描** | 自动扫描所有 JAX-RS 资源 | true |
| **scanKnownConfigLocations** | 扫描已知的 OpenAPI 配置位置 (classpath 或文件系统)，它们是：<br>openapi-configuration.yaml<br>openapi-configuration.json<br>openapi.yaml<br>openapi.json | true |
| **scannerClass** | JAX-RS API 扫描程序类的名称，用于限制应用、资源软件包、资源类和类路径扫描，请参阅 资源扫描 部分 | null |
| **securityDefinitions** | 安全定义列表+ | null |
| **supportSwaggerUi** | 打开/关闭 SwaggerUI 支持 | null (== true) |
| **swaggerUiConfig** | Swagger UI 配置 | null |
| **swaggerUiMavenGroupAndArtifact** | 用于固定 SwaggerUI 的 Maven 工件 | null |
| **swaggerUiVersion** | SwaggerUI 的版本 | null |
| **termsOfServiceUrl** | 服务 URL 的条款+ | null |
| **title** | 标题+ | null |
| **useContextBasedConfig** | 如果设置，则会为每个 OpenApiContext 实例生成唯一的上下文 Id （请参阅 使用多服务器端点）。另外，您可能想将扫描属性设置为 false。 | false |

| Name | 描述 | default |
| --- | --- | --- |
| **version** | version+ | null |

*+ 选项在 OpenAPI 类中定义*

*++ 选项在 SwaggerConfiguration 类中定义*

### *54.2. KARAF 实施*

*本节论述了如何使用 OpenApiFeature 在 JAR 文件中定义 REST 服务，并部署到 Karaf 容器上的 Fuse。*

#### *54.2.1. Quickstart 示例*

*您可以从 Fuse Software Downloads 页面下载红帽 Fuse 快速入门。*

*Quickstart zip 文件包含用于快速入门的 /cxf/rest/ 目录，它演示了如何使用 CXF 创建 RESTful (JAX-RS) Web 服务以及如何启用 OpenAPI 并注解 JAX-RS 端点。*

#### *54.2.2. 启用 OpenAPI*

*启用 OpenAPI 涉及：*

- *通过在 <jaxrs :server> 定义中添加 CXF 类 (org.apache.cxf.jaxrs.openapi.OpenApiFeature)来修改 定义 CXF 服务的 XML 文件。*

  *例如，请参阅 55.4 示例 XML 文件示例。*

- *在 REST 资源类中：*

  - *为服务所需的每个注解导入 OpenAPI 注解：*

```
import io.swagger.annotations.*
```

- 

其中 * = Api,ApiOperation,ApiParam,ApiResponse,ApiResponses, 等。

详情请参阅 https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X。

例如，请参阅 55.5 资源类示例。

- 将 OpenAPI 注解添加到 JAX-RS 注释的端点 (@PATH、@PUT、@POST、@GET、@Produces、@Consumes、@DELETE、@PathParam 等等)。

例如，请参阅 55.5 资源类示例。

### 55.4 示例 XML 文件示例

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

   <jaxrs:server id="customerService" address="/crm">
      <jaxrs:serviceBeans>
         <ref component-id="customerSvc"/>
      </jaxrs:serviceBeans>
      <jaxrs:providers>
        <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
      </jaxrs:providers>
      <jaxrs:features>
      <bean class="org.apache.cxf.jaxrs.openapi.OpenApiFeature">
        <property name="title" value="Fuse:CXF:Quickstarts - Customer Service" />
        <property name="description" value="Sample REST-based Customer Service" />
        <property name="version" value="${project.version}" />
      </bean>
      </jaxrs:features>
   </jaxrs:server>
```

```
    <cxf:bus>
        <cxf:features>
          <cxf:logging />
        </cxf:features>
        <cxf:properties>
            <entry key="skip.default.json.provider.registration" value="true" />
        </cxf:properties>
    </cxf:bus>

    <bean id="customerSvc" class="org.jboss.fuse.quickstarts.cxf.rest.CustomerService"/>

</blueprint>
```

### 55.5 资源类示例

```
.
.
.

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;

.
.
.

@Path("/customerservice/")
@Api(value = "/customerservice", description = "Operations about customerservice")
public class CustomerService {

    private static final Logger LOG =
        LoggerFactory.getLogger(CustomerService.class);

    private MessageContext jaxrsContext;
    private long currentId                = 123;
```

```
private Map<Long, Customer> customers = new HashMap<>();
private Map<Long, Order> orders       = new HashMap<>();

public CustomerService() {
    init();
}


@GET
@Path("/customers/{id}/")
@Produces("application/xml")
@ApiOperation(value = "Find Customer by ID", notes = "More notes about this
    method", response = Customer.class)
@ApiResponses(value = {
        @ApiResponse(code = 500, message = "Invalid ID supplied"),
        @ApiResponse(code = 204, message = "Customer not found")
        })
public Customer getCustomer(@ApiParam(value = "ID of Customer to fetch",
    required = true) @PathParam("id") String id) {
    LOG.info("Invoking getCustomer, Customer id is: {}", id);
    long idNumber = Long.parseLong(id);
    return customers.get(idNumber);
}


@PUT
@Path("/customers/")
@Consumes({ "application/xml", "application/json" })
@ApiOperation(value = "Update an existing Customer")
@ApiResponses(value = {
        @ApiResponse(code = 500, message = "Invalid ID supplied"),
        @ApiResponse(code = 204, message = "Customer not found")
        })
public Response updateCustomer(@ApiParam(value = "Customer object that needs
    to be updated", required = true) Customer customer) {
    LOG.info("Invoking updateCustomer, Customer name is: {}", customer.getName());
    Customer c = customers.get(customer.getId());
    Response r;
    if (c != null) {
        customers.put(customer.getId(), customer);
        r = Response.ok().build();
    } else {
        r = Response.notModified().build();
    }

    return r;
}

@POST
@Path("/customers/")
@Consumes({ "application/xml", "application/json" })
@ApiOperation(value = "Add a new Customer")
@ApiResponses(value = { @ApiResponse(code = 500, message = "Invalid ID
    supplied"), })
public Response addCustomer(@ApiParam(value = "Customer object that needs to
    be updated", required = true) Customer customer) {
```

```
        LOG.info("Invoking addCustomer, Customer name is: {}", customer.getName());
        customer.setId(++currentId);

        customers.put(customer.getId(), customer);
        if (jaxrsContext.getHttpHeaders().getMediaType().getSubtype().equals("json"))
    {
            return Response.ok().type("application/json").entity(customer).build();
        } else {
            return Response.ok().type("application/xml").entity(customer).build();
        }
    }

    @DELETE
    @Path("/customers/{id}/")
    @ApiOperation(value = "Delete Customer")
    @ApiResponses(value = {
            @ApiResponse(code = 500, message = "Invalid ID supplied"),
            @ApiResponse(code = 204, message = "Customer not found")
            })
    public Response deleteCustomer(@ApiParam(value = "ID of Customer to delete",
        required = true) @PathParam("id") String id) {
        LOG.info("Invoking deleteCustomer, Customer id is: {}", id);
        long idNumber = Long.parseLong(id);
        Customer c = customers.get(idNumber);

        Response r;
        if (c != null) {
            r = Response.ok().build();
            customers.remove(idNumber);
        } else {
            r = Response.notModified().build();
        }

        return r;
    }

.
.
.

}
```

### 54.3. SPRING BOOT 实施

本节论述了如何在 Spring Boot 中使用 Swagger2Feature。

请注意，对于 OpenAPI 3 实现，使用 OpenApiFeature (org.apache.cxf.jaxrs.openapi.OpenApiFeature)。

### 54.3.1. Quickstart 示例

Quickstart 示例(https://github.com/fabric8-quickstarts/spring-boot-cxf-jaxrs)演示了如何在 Spring Boot 中使用 Apache CXF。Quickstart 使用 Spring Boot 配置包含启用了 Swagger 的 CXF JAX-RS 端点的应用程序。

### 54.3.2. 启用 Swagger

启用 Swagger 涉及：

- 在 REST 应用程序中：
  - 导入 Swagger2Feature：

```
import org.apache.cxf.jaxrs.swagger.Swagger2Feature;
```

  - 在 CXF 端点中添加 Swagger2Feature：

```
endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
```

    例如，请参阅 示例 55.1 示例 REST 应用程序。

- 在 Java 实现文件中，为服务所需的每个注解导入 Swagger API 注解：

```
import io.swagger.annotations.*
```

  其中 * = Api,ApiOperation,ApiParam,ApiResponse,ApiResponses, 等。

  详情请查看 https://github.com/swagger-api/swagger-core/wiki/Annotations。

  例如，请参阅 Java 实现文件示例 55.2 示例。

-

*在 Java 文件中，将 Swagger 注释添加到 JAX-RS 注释 (@PATH,@PUT,@POST,@GET,@Produces,@Consumes,@DELETE,@PathParam, 等)。*

*例如，请参阅 Java 文件示例 55.3 示例。*

**示例 55.1 示例 REST 应用程序**

```
package io.fabric8.quickstarts.cxf.jaxrs;

import java.util.Arrays;

import org.apache.cxf.Bus;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.swagger.Swagger2Feature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SampleRestApplication {

    @Autowired
    private Bus bus;

    public static void main(String[] args) {
        SpringApplication.run(SampleRestApplication.class, args);
    }

    @Bean
    public Server rsServer() {
        // setup CXF-RS
        JAXRSServerFactoryBean endpoint = new JAXRSServerFactoryBean();
        endpoint.setBus(bus);
        endpoint.setServiceBeans(Arrays.<Object>asList(new HelloServiceImpl()));
        endpoint.setAddress("/");
        endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
        return endpoint.create();
    }
}
```

**Java 实现文件示例 55.2 示例**

```
import io.swagger.annotations.Api;

@Api("/sayHello")
public class HelloServiceImpl implements HelloService {

    public String welcome() {
        return "Welcome to the CXF RS Spring Boot application, append /{name} to call the hello
service";
    }

    public String sayHello(String a) {
        return "Hello " + a + ", Welcome to CXF RS Spring Boot World!!!";
    }

}
```

***Java 文件示例 55.3 示例***

```
package io.fabric8.quickstarts.cxf.jaxrs;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.springframework.stereotype.Service;

@Path("/sayHello")
@Service
public interface HelloService {

    @GET
    @Path("")
    @Produces(MediaType.TEXT_PLAIN)
    String welcome();

    @GET
    @Path("/{a}")
    @Produces(MediaType.TEXT_PLAIN)
    String sayHello(@PathParam("a") String a);

}
```

### 54.4. 访问 OPENAPI 文档

当 OpenApiFeature 启用 OpenAPI 时，OpenAPI 文档位于由服务端点位置组成的位置 URL 中，后跟 /openapi.json 或 /openapi.yaml。

例如，对于在 http://host:port/context/services/ 中发布的 JAX-RS 端点，其中 context 是 Web 应用上下文，/services 是 servlet URL，其 OpenAPI 文档位于 http://host:port/context/services/openapi.json 和 http://host:port/context/services/openapi.yaml。

如果 OpenApiFeature 处于活跃状态，则 CXF Services 页面会链接到 OpenAPI 文档。

在上例中，您将进入 http://host:port/context/services/services，然后按照链接返回 OpenAPI JSON 文档。

如果需要从另一个主机上的 OpenAPI UI 访问定义，您可以添加 cxf-rt-rs-security-cors 中的 CrossOriginResourceSharingFilter。

### 54.5. 通过反向代理访问 OPENAPI

如果要通过反向代理访问 OpenAPI JSON 文档或 OpenAPI UI，请设置以下选项：

- 将 CXFServlet use-x-forwarded-headers init 参数设置为 true。
  - 在 Spring Boot 中，使用 cxf.servlet.init 为参数名称添加前缀：

```
cxf.servlet.init.use-x-forwarded-headers=true
```

  - 在 Karaf 中，将以下行添加到 installDir/etc/org.apache.cxf.osgi.cfg 配置文件中：

```
cxf.servlet.init.use-x-forwarded-headers=true
```

    注：如果您还没有在 etc 目录中有一个 org.apache.cxf.osgi.cfg 文件，您可以创建一个。

-

**如果您为 OpenApiFeature basePath 选项指定一个值，且您要防止 OpenAPI 缓存 basePath 值，请将 OpenApiFeature usePathBasedConfig 选项设置为 TRUE:**

```
<bean class="org.apache.cxf.jaxrs.openapi.OpenApiFeature">
    <property name="usePathBasedConfig" value="TRUE" />
</bean>
```

# 部分 VII. 开发 APACHE CXF INTERCEPTORS

本指南论述了如何编写 Apache CXF 拦截器，这些拦截器可以在消息中执行预处理和后处理。

# 第 55 章 APACHE CXF 运行时中的拦截器


## 摘要

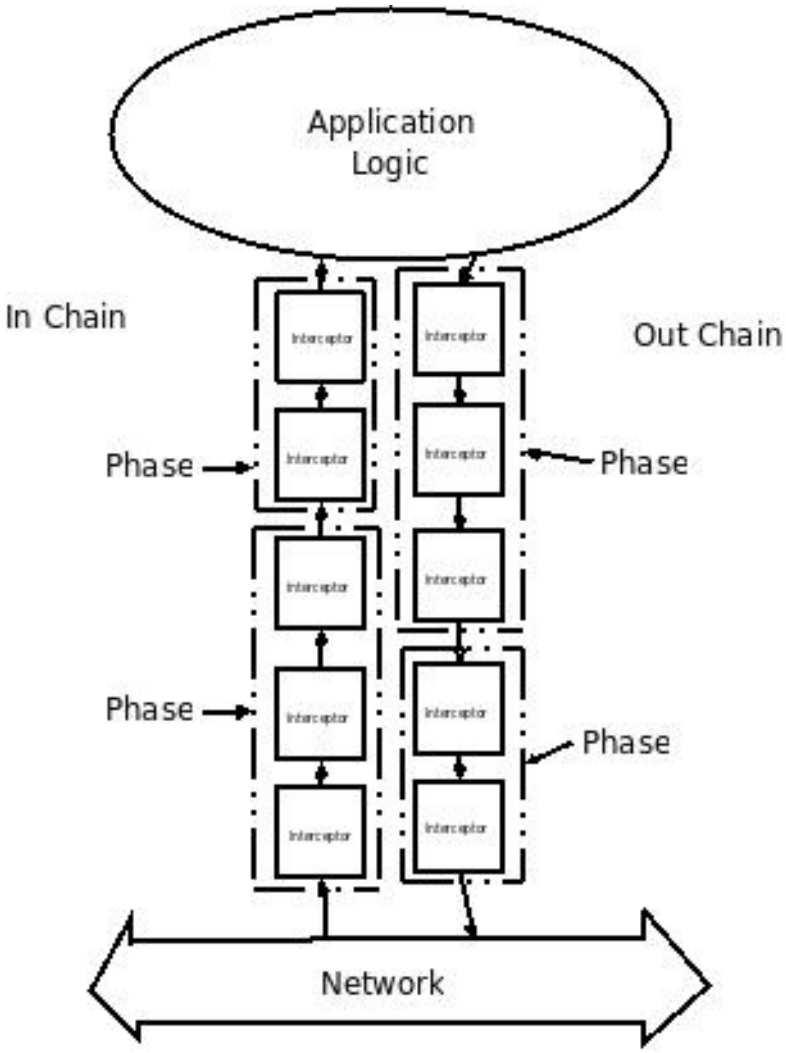
Apache CXF 运行时中的大多数功能都由拦截器实现。Apache CXF 运行时创建的每个端点都有三个潜在的拦截器链来处理消息。这些链中的拦截器负责在线路传输的原始数据之间转换消息，以及由端点实施代码处理的 Java 对象。拦截器被组织为阶段，以确保以正确的顺序进行处理。


## 概述

Apache CXF 处理消息的很大部分。当消费者对远程服务进行调用时，运行时需要将数据放入服务可以使用的消息并将其放在线上。服务供应商必须 unmarshal 消息，执行其业务逻辑，并将响应放入适当的消息格式。然后，消费者必须 unmarshal 响应消息，将其与正确的请求相关联，并将其传递回消费者的应用程序代码。除了基本的 marshaling 和 unmarshaling 外，Apache CXF 运行时还可以执行一些其他操作及消息数据。例如，如果已激活 WS-RM，则运行时必须在 marshaling 和 unmarshaling 前处理消息块和确认消息。如果激活了安全性，则运行时必须验证消息的凭据，作为消息处理序列的一部分。

图 55.1 “Apache CXF 拦截器链” 显示请求消息在由服务提供商接收时采用的基本路径。

图 55.1. Apache CXF 拦截器链




### APACHE CXF 中的消息处理

当 Apache CXF 开发使用者调用远程服务时，启动以下消息处理序列：

1. Apache CXF 运行时会创建一个出站拦截器链来处理请求。

2. 如果调用启动双向消息交换，则运行时会创建一个入站拦截器链和一个故障处理拦截器链。

3. 请求消息通过出站拦截器链按顺序传递。

   链中的每个拦截器对消息执行一些处理。例如，Apache CXF 提供 SOAP 拦截器会打包 SOAP 信封中的消息。

4. 如果出站链上的任何拦截器创建错误条件，则链将被 unwound 并且控制返回到应用程序级别

代码。

拦截器链通过在之前调用的所有拦截器上调用 fault 处理方法。

5. 请求将发送到适当的服务提供商。

6. 收到响应时，它将按顺序通过入站拦截器链传递。

注意

如果响应是错误消息，它将被传递给故障处理拦截器链。

7. 如果入站链上的任何拦截器创建错误条件，则链为 unwound。

8. 当消息到达入站拦截器链的末尾时，它将传回到应用程序代码。

当 Apache CXF 开发的服务供应商从消费者收到请求时，会出现类似的过程：

1. Apache CXF 运行时会创建一个入站拦截器链来处理请求消息。

2. 如果请求是双向消息交换的一部分，则运行时还会创建一个出站拦截器链和一个故障处理拦截器链。

3. 请求通过入站拦截器链按顺序传递。

4. 如果入站链上的任何拦截器创建错误条件，则链将被取消设置，并将故障分配给消费者。

拦截器链通过在之前调用的所有拦截器上调用 fault 处理方法。

5. 当请求到达入站拦截器链的末尾时，它将传递给服务实施。

6. 当响应就绪时，它将按顺序通过出站拦截器链传递。

**注意**

如果响应是例外的，它将通过故障处理拦截器链传递。

7. 如果出站链上的任何拦截器创建错误条件，则链为 **unwound**，并会分配错误消息。

8. 请求到达出站链的末尾后，它将分配给消费者。

拦截器

Apache CXF 运行时中的所有消息处理都由 拦截器 完成。拦截器是在传递给应用程序层之前可以访问消息数据的 POJO。他们可以执行很多操作，包括：转换消息、去除消息的标头或验证消息数据。例如，拦截器是否可以从消息中读取安全标头，对外部安全服务验证凭证，并确定消息处理是否可以继续。

拦截器可用的消息数据由以下几个因素决定：

- 拦截器的链
- 拦截器的阶段
- 前面在链中发生的其他拦截器

阶段

拦截器分为几个 阶段。阶段是具有通用功能的拦截器的逻辑分组。每个阶段负责特定类型的消息处理。例如，处理传递给应用程序层的 **marshaled Java** 对象的拦截器都会在同一阶段中发生。

拦截器链

阶段被聚合为 拦截器链。拦截器链是根据消息是入站还是出站排序的拦截器阶段列表。

使用 Apache CXF 创建的每个端点都有三个拦截器链：

- 入站消息的链
- 用于出站消息的链
- 错误消息的链

拦截器链主要基于端点使用的绑定和传输进行构建。添加其他运行时功能（如安全或日志记录）还会在链中添加拦截器。开发人员也可以使用配置将自定义拦截器添加到链中。

开发拦截器

无论其功能是什么，都遵循相同的基本步骤：

1. 第 56 章 Interceptor API

   Apache CXF 提供了多个抽象拦截器，以便更轻松地开发自定义拦截器。

2. 第 57.2 节 “指定拦截器的阶段”

   拦截器要求消息的某些部分可用，并需要数据采用特定格式。消息内容和数据格式由拦截器的阶段部分决定。

3. 第 57.3 节 “在阶段限制拦截器放置”

   通常，阶段内拦截器的顺序不重要。然而，在某些情况下，可能务必要确保拦截器在之前或之后在同一阶段中执行。

4.
第 58.2 节 “处理消息”

5.
第 58.3 节 “出错后取消卷”

如果在执行拦截器后在活跃拦截器链中发生错误，则会调用其故障处理逻辑。

6.
第 59 章 配置端点以使用拦截器

# 第 56 章 INTERCEPTOR API

## 摘要

拦截器实施 PhaseInterceptor 接口，它扩展了基本拦截器接口。此接口定义了 Apache CXF 运行时用来控制拦截器执行的许多方法，不适用于应用程序开发人员实施。为简化拦截器开发，Apache CXF 提供了很多可扩展的抽象拦截器实现。

## 接口

Apache CXF 中的所有拦截器都实现 例 56.1 “基本拦截器接口” 中显示的基本 Interceptor 接口。

**例 56.1. 基本拦截器接口**

```
package org.apache.cxf.interceptor;

public interface Interceptor<T extends Message>
{

  void handleMessage(T message) throws Fault;

  void handleFault(T message);

}
```

Interceptor 接口定义了开发人员为自定义拦截器实施的两种方法：

### handleMessage()

handleMessage （） 方法在拦截器中执行大多数工作。它在策略链中的每个拦截器上调用，并接收正在处理的消息的内容。开发人员在此方法中实施拦截器的消息处理逻辑。有关实现 handleMessage （） 方法的详情，请参考 第 58.2 节 “处理消息”。

### handleFault()

当正常消息处理中断时，在拦截器上调用 handleFault （） 方法。运行时会调用每个调用的拦截器的 handleFault （） 方法，因为它取消了拦截器链。有关实现 handleFault （） 方法的详情，请参考 第 58.3 节 “出错后取消卷”。

大多数拦截器不直接实现 Interceptor 接口。相反，它们实现了 例 56.2 “阶段拦截器接口” 中显示的 PhaseInterceptor 接口。PhaseInterceptor 接口添加了四个允许拦截器链的方法。

**例 56.2. 阶段拦截器接口**

```
package org.apache.cxf.phase;
...

public interface PhaseInterceptor<T extends Message> extends Interceptor<T>
{

    Set<String> getAfter();

    Set<String> getBefore();

    String getId();

    String getPhase();

}
```

**抽象拦截器类**

开发人员应扩展 **AbstractPhaseInterceptor** 类，而不是直接实现 **PhaseInterceptor** 接口。这个抽象类为 **PhaseInterceptor** 接口的阶段管理方法提供实施。**AbstractPhaseInterceptor** 类也提供 **handleFault （） 方法的默认实现。**

开发人员需要提供 **handleMessage （） 方法的实施。它们也可以为 handleFault （） 方法提供不同的实现。开发者提供的实现可以使用通用 org.apache.cxf.message.Message 接口提供的方法操作消息数据。**

对于使用 **SOAP** 消息的应用程序，**Apache CXF** 提供了一个 **AbstractSoapInterceptor** 类。扩展此类提供了 **handleMessage （） 方法和 handleFault （） 方法，作为 org.apache.cxf.binding.soap.SoapMessage 对象访问消息数据。SoapMessage 对象有从消息中检索 SOAP 标头、SOAP 信封和其他 SOAP 元数据的方法。**

# 第 57 章 确定拦截器何时被调用

## 摘要

拦截器分为几个阶段。拦截器运行的阶段决定了它可以访问的消息数据的一部分。拦截器可以确定其与同一阶段中其他拦截器的关系位置。拦截器的阶段及其在阶段的位置被设置为拦截器逻辑的一部分。

## 57.1. 指定拦截器位置

在开发自定义拦截器时，首先要考虑的是拦截器所属的消息处理链中。开发人员可以通过以下两种方式之一控制消息处理链中的位置：

- 指定拦截器的阶段
- 在阶段内对拦截器的位置指定限制

通常，指定拦截器位置的代码放置在拦截器的构造器中。这使得运行时可以实例化拦截器，并放在拦截器链中的正确位置，而无需在应用级别代码中出现任何明确的操作。

## 57.2. 指定拦截器的阶段

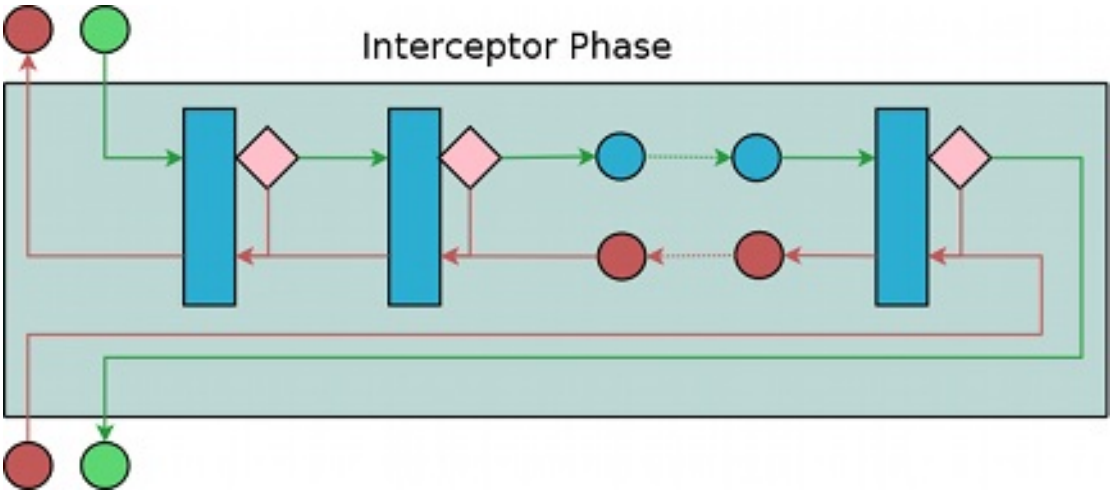
### 概述

拦截器分为几个阶段。拦截器的阶段决定了在消息处理序列中调用的时间。开发人员指定拦截器的阶段。阶段使用框架提供的常量值来指定。

### 阶段

阶段是拦截器的逻辑集合。如 图 57.1 “拦截器阶段” 所示，阶段内的拦截器会按顺序调用。

图 57.1. 拦截器阶段




阶段以有序列表链接，组成拦截器链，并在消息处理过程中提供定义的逻辑步骤。例如，入站拦截器链的 RECEIVE 阶段有一个拦截器组使用从线路获取的原始消息数据来处理传输级别详情。

但是，并没有执行任何阶段可以执行的操作。我们建议一个阶段中的拦截器遵循阶段进取的任务。

Apache CXF 定义的完整阶段列表可在 第 62 章 Apache CXF 消息处理阶段 中找到。

### 指定阶段

Apache CXF 提供用于指定阶段的 org.apache.cxf.Phase 类。类是常量的集合。Apache CXF 定义的每个阶段在 阶段 类中都有对应的常数。例如，RECEIVE 阶段由值 Phase.RECEIVE 指定。

### 设置阶段

拦截器的阶段在拦截器的构造器中设置。AbstractPhaseInterceptor 类定义了三个用于实例化拦截器的构造器：

- 公共 AbstractPhaseInterceptor (String phase)- 将拦截器的阶段设置为指定的阶段，并将拦截器的 id 设置为拦截器的类名称。

  该构造器将满足大多数用例。

- 公共 AbstractPhaseInterceptor (String id, String phase)- 将拦截器的 id 设置为作为第一个参数传递的字符串，并将拦截器的阶段设置为第二个字符串。

- 公共 AbstractPhaseInterceptor (String phase, boolean uniqueId)-specifies the interceptor should using a unique, system generated id.如果 uniqueId 参数为 true，则拦截器的 id 将由系统计算。如果 uniqueId 参数为 false，则拦截器的 id 被设置为拦截器的类名称。

设置自定义拦截器阶段的建议方法是使用 super （） 方法将阶段传递给 AbstractPhaseInterceptor 构造器，如 例 57.1 “设置拦截器的阶段” 所示。

**例 57.1. 设置拦截器的阶段**

```
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
 {

    public StreamInterceptor()
   {
     super(Phase.PRE_STREAM);
    }
}
```

例 57.1 “设置拦截器的阶段” 中显示的 StreamInterceptor interceptor 放置在 PRE_STREAM 阶段。

### 57.3. 在阶段限制拦截器放置

#### 概述

将拦截器放在阶段可能无法对放置进行精细的控制，以确保拦截器正常工作。例如，如果需要使用 SAAJ API 检查消息的 SOAP 标头，则需要在拦截器后将消息转换为 SAAJ 对象后运行。有些情况下，一个拦截器消耗另一个拦截器需要的消息的一部分。在这些情况下，开发人员可以提供在拦截器之前必须执行的拦截器列表。开发人员也可以提供在拦截器之后必须执行的拦截器列表。

重要

运行时只能在拦截器的阶段使用这些列表。如果开发人员从早期阶段将拦截器放在必须在当前阶段之后执行的拦截器列表中，则运行时将忽略请求。

#### 添加到链前

开发拦截器时出现的一个问题是，拦截器所需的数据并不总是存在。当链中的一个拦截器消耗后续拦

截器所需的消息数据时，可能会发生这种情况。开发人员可以通过修改其拦截器来控制自定义拦截器消耗的内容，并可能会解决这个问题。但是，这并不总是可行，因为 Apache CXF 使用了多个拦截器，开发人员无法修改它们。

另一种解决方法是确保在任何使用定制拦截器所需的消息数据的拦截器之前放置自定义拦截器。执行此操作的最简单方法是将其放在早期阶段，但并不总是可能。对于需要在一个或多个拦截器之前放置拦截器，Apache CXF 的 AbstractPhaseInterceptor 类提供了两个 addBefore （） 方法。

如 例 57.2 “在其他拦截器前添加拦截器的方法” 所示，一个使用单个拦截器 ID，另一个使用拦截器 ID。您可以进行多个调用来继续向列表添加拦截器。

例 57.2. 在其他拦截器前添加拦截器的方法

```
publicaddBeforeStringipublicaddBeforeCollection<String>i
```

如 例 57.3 “指定在当前拦截器后必须运行的拦截器列表” 所示，开发人员在自定义拦截器的 constuctor 中调用 addBefore （） 方法。

例 57.3. 指定在当前拦截器后必须运行的拦截器列表

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addBefore(HolderOutInterceptor.class.getName());
    }

...

}
```

大多数拦截器使用它们的类名称进行拦截器 ID。

将 添加到链后

拦截器所需的另一个原因是数据没有被放在消息对象中。例如，拦截器可能希望将消息数据用作 SOAP 消息，但在消息转换为 SOAP 消息之前，它将无法正常工作。开发人员可以通过修改其拦截器来

控制自定义拦截器消耗的内容，并可能会解决这个问题。但是，这并不总是可行，因为 Apache CXF 使用了多个拦截器，开发人员无法修改它们。

另一种解决方法是确保将自定义拦截器放在拦截器或拦截器后，生成自定义拦截器所需的消息数据。执行此操作的最简单方法是将其放在后续阶段，但并不总是可能。AbstractPhaseInterceptor 类为需要在一个或多个其它拦截器之后放置拦截器的情况提供两个 addAfter （） 方法。

如 例 57.4 “在其他拦截器后添加拦截器的方法” 所示，一个方法采用单个拦截器 ID，另一个方法采用拦截器 ID 的集合。您可以进行多个调用来继续向列表添加拦截器。

**例 57.4. 在其他拦截器后添加拦截器的方法**

```
publicaddAfterStringipublicaddAfterCollection<String>i
```

如 例 57.5 “指定在当前拦截器前必须运行的拦截器列表” 所示，开发人员在自定义拦截器的 constuctor 中调用 addAfter （） 方法。

**例 57.5. 指定在当前拦截器前必须运行的拦截器列表**

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addAfter(StartingOutInterceptor.class.getName());
    }

...

}
```

大多数拦截器使用它们的类名称进行拦截器 ID。

# 第 58 章 实施 INTERCEPTORS PROCESSING LOGIC

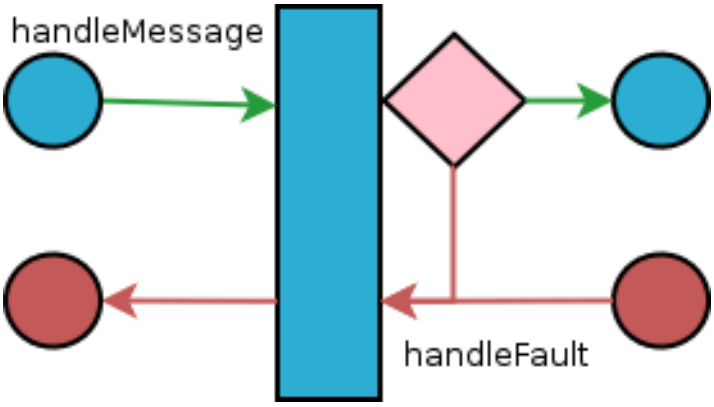
**摘要**

拦截器直接实施。它们批量处理逻辑位于 handleMessage （） 方法中。此方法接收消息数据并根据需要对其进行操作。开发人员可能还希望添加一些特殊的逻辑来处理故障处理情况。

## 58.1. 拦截器流

图 58.1 “通过拦截器流” 显示通过拦截器的进程流。

图 58.1. 通过拦截器流




在正常消息处理中，仅调用 handleMessage （） 方法。handleMessage （） 方法放置拦截器的消息处理逻辑。

如果在拦截器的 handleMessage （） 方法中发生错误，或者拦截器链中的任何后续拦截器，则会调用 handleFault （） 方法。在出现错误时，handleFault （）方法可用于清理。它还可用于更改错误消息。

## 58.2. 处理消息

### 概述

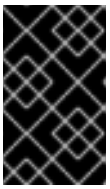
在正常消息处理中，调用拦截器的 handleMessage （） 方法。它接收消息数据作为一个 Message 对象。除了消息的实际内容外，Message 对象还可以包含与消息或消息处理状态相关的多个属性。Message 对象的确切内容取决于链中当前拦截器前面的拦截器。

*获取消息内容*

*Message 接口提供了两种方法，可用于提取消息内容：*

- *公共<T> T getContentjava.lang.Class<T> 格式 The getContent （） 方法在指定类的对象中返回消息的内容。如果内容不作为指定类的实例使用，则返回 null。可用内容类型列表由拦截器链中的位置和拦截器链的方向决定。*

- *公共集合<Attachment&gt ;getAttachments （） 方法返回 Java Collection 对象，其中包含与消息关联的任何二进制附加。attachments 存储在 org.apache.cxf.message.Attachment 对象中。attachment 对象提供管理二进制数据的方法。*



*重要*

*attachments 仅在附加处理拦截器执行后可用。*

*确定消息的方向*

*可以通过查询消息交换来确定消息的方向。消息交换将入站消息和出站消息存储在单独的属性中。[3]*

*使用消息的 getExchange （） 方法检索与消息关联的消息交换。如 例 58.1 “获取消息交换” 所示，getExchange （） 不使用任何参数，并返回消息交换作为 org.apache.cxf.message.Exchange 对象。*

> *例 58.1. 获取消息交换*
>
> *ExchangegetExchange*

*Exchange 对象有四个方法，如 例 58.2 “从消息交换中获取消息” 所示，用于获取与交换关联的消息。每个方法都返回消息作为 org.apache.cxf.Message 对象，如果消息不存在，它将返回 null。*

> *例 58.2. 从消息交换中获取消息*
>
> *MessagegetInMessageMessagegetInFaultMessageMessagegetOutMessageMessagegetOutFaultMessage*

例 58.3 “检查证书链的方向” 显示用于确定当前消息是否出站性的代码。该方法获取消息交换，并检查当前消息是否与交换的出站消息相同。它还会根据交换出站故障故障消息检查当前消息，到出站故障拦截器链上的错误消息。

**例 58.3. 检查证书链的方向**

```
public static boolean isOutbound()
{
  Exchange exchange = message.getExchange();
      return message != null
            && exchange != null
            && (message == exchange.getOutMessage()
               || message == exchange.getOutFaultMessage());
}
```

**Example**

例 58.4 “消息处理方法示例” 显示处理 **zip** 压缩消息的拦截器的代码。它将检查消息的方向，然后执行适当的操作。

**例 58.4. 消息处理方法示例**

```
import java.io.IOException;
import java.io.InputStream;
import java.util.zip.GZIPInputStream;

import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
{

  ...

  public void handleMessage(Message message)
  {

      boolean isOutbound = false;
      isOutbound = message == message.getExchange().getOutMessage()
            || message == message.getExchange().getOutFaultMessage();

      if (!isOutbound)
      {
        try
        {
          InputStream is = message.getContent(InputStream.class);
```

```
            GZIPInputStream zipInput = new GZIPInputStream(is);
            message.setContent(InputStream.class, zipInput);
          }
          catch (IOException ioe)
          {
            ioe.printStackTrace();
          }
        }
        else
        {
         // zip the outbound message
        }
    }
  ...
}
```

### 58.3. 出错后取消卷

#### 概述

当执行拦截器链期间发生错误时，运行时会停止遍历拦截器链，并通过调用已执行的链中任何拦截器的 handleFault （） 方法来遍历链。

handleFault （） 方法可用于在正常消息处理过程中清理拦截器使用的任何资源。它还可用于回滚只有在消息处理成功完成时才应遵循的任何操作。如果故障消息将传递到出站故障处理拦截器链，则 handleFault （） 方法也可用于向故障消息添加信息。

#### 获取消息有效负载

handleFault （） 方法接收与正常消息处理中使用的 handleMessage （） 方法相同的 Message 对象。“获取消息内容”一节 中描述了从 Message 对象获取消息内容。

#### Example

例 58.5 “处理未卷拦截器链” 显示用来确保在拦截器链 unwound 时，用来确保原始 XML 流重新放入消息的代码。

例 58.5. 处理未卷拦截器链

```
@Override
```

```
public void handleFault(SoapMessage message)
{
  super.handleFault(message);
  XMLStreamWriter writer = (XMLStreamWriter)message.get(ORIGINAL_XML_WRITER);
  if (writer != null)
  {
    message.setContent(XMLStreamWriter.class, writer);
  }
}
```

---

*[3]*
*它还会单独存储入站和出站错误。*

# 第 59 章 配置端点以使用拦截器

## 摘要

当消息交换中包含拦截器时，拦截器会添加到端点中。端点的拦截器链由 Apache CXF 运行时中的多个组件的拦截器链组成。拦截器在端点的配置或其中一个运行时组件的配置中指定。可以使用配置文件或拦截器 API 添加拦截器。

## 59.1. 决定附加拦截器的位置

### 概述

主机拦截器链有很多运行时对象。它们是：

- 端点对象
- service 对象
- proxy 对象
- 用于创建端点或代理的 factory 对象
- 绑定
- 中央 总线 对象

开发人员可以将自己的拦截器附加到这些对象。附加拦截器的最常见对象是总线和单个端点。选择正确的对象需要了解如何组合这些运行时对象来创建端点。根据设计，每个 cxf 相关捆绑包都有自己的 cxf 总线。因此，如果在总线中配置拦截器，并且同一 Blueprint 上下文上的服务被导入或创建到另一个捆绑包中，则不会处理拦截器。相反，您可以将拦截器直接配置为导入的服务中的 JAXWS 客户端或端点。

### 端点和代理

将拦截器附加到端点或代理是放置拦截器的最精细的方法。任何直接附加到端点或代理的拦截器都只影响特定的端点或代理。这是附加特定于服务特定影响的拦截器的好位置。例如，如果开发人员希望公开

一个服务实例，该服务将单位从指标转换为 imperial，他们可以直接将拦截器附加到一个端点。

因素

使用 Spring 配置将拦截器附加到用于创建端点或代理的工厂中，与将拦截器直接附加到端点或代理可以正常工作。但是，当拦截器以编程方式附加到工厂时，会传播到工厂创建的每个端点或代理。

绑定

将拦截器附加到绑定可让开发人员指定一组应用到使用该绑定的所有端点的拦截器。例如，如果开发人员希望强制所有使用原始 XML 绑定的端点包含特殊的 ID 元素，他们可以附加负责将元素添加到 XML 绑定中的拦截器。

总线

附加拦截器的最常见位置是总线。当拦截器附加到总线时，拦截器会被传播到该总线管理的所有端点。在创建共享一组类似拦截器的多个端点的应用程序中，将拦截器附加到总线非常有用。

合并附加点

因为端点的最终拦截器链集合是列出的对象贡献的拦截器链，所以列出的对象可以在单一端点配置中合并。例如，如果应用生成多个端点，它们都需要检查验证令牌来检查该拦截器，则拦截器将附加到应用的总线。如果其中一个端点还需要一个拦截器，该拦截器将 Standards 转换为美元，则转换拦截器将直接附加到特定的端点。

## 59.2. 使用配置添加拦截器

概述

将拦截器附加到端点的最简单方法是使用配置文件。要附加到端点的每个拦截器都使用标准 Spring bean 配置。然后，可以使用 Apache CXF 配置元素将拦截器的 bean 添加到正确的拦截器链中。

具有关联拦截器链的每个运行时组件都可以使用专用的 Spring 元素进行配置。组件的每个元素都有一组标准的子项，用于指定其拦截器链。每个与组件关联的拦截器链都有一个子级。要添加到链中的拦截器的 Bean 的子项列表。

### 配置元素

表 59.1 “拦截器链配置元素” 描述将拦截器附加到运行时组件的四个配置元素。

表 59.1. 拦截器链配置元素

| 元素 | 描述 |
| --- | --- |
| **inInterceptors** | 包含 Bean 列表，配置拦截器以添加到端点的入站拦截器链中。 |
| **outInterceptors** | 包含 Bean 列表，配置拦截器以添加到端点的出站拦截器链中。 |
| **inFaultInterceptors** | 包含 Bean 列表，配置拦截器以添加到端点的入站故障处理拦截器链中。 |
| **outFaultInterceptors** | 包含 Bean 列表，配置拦截器以添加到端点的出站故障处理拦截器链中。 |

所有拦截器链配置元素都取一个 **list** 子元素。**list** 元素有一个子对象，用于附加到链的每个拦截器。可以使用 **bean** 元素直接配置拦截器或 **ref** 元素来指定拦截器，该元素引用配置拦截器的 **bean** 元素。

### 例子

例 59.1 “将拦截器附加到总线” 显示将拦截器附加到总线的入站拦截器链的配置。

**例 59.1. 将拦截器附加到总线**

```
&lt;beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="
      http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
      http://cxf.apache.org/transports/http/configuration
http://cxf.apache.org/schemas/configuration/http-conf.xsd
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"&gt;
  ...
  &lt;bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor"/&gt;

  &lt;cxf:bus&gt;
    *&lt;cxf:inInterceptors&gt;
      &lt;list&gt;
        &lt;ref bean="GZIPStream"/&gt;
      &lt;/list&gt;
```

```
    &lt;/cxf:inInterceptors&gt;*
  &lt;/cxf:bus&gt;
&lt;/beans&gt;
```

*例 59.2 “将拦截器附加到 JAX-WS 服务提供商” 显示将拦截器附加到 JAX-WS 服务的出站拦截器链的配置。*

**例 59.2. 将拦截器附加到 JAX-WS 服务提供商**

```
&lt;beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"&gt;

  &lt;jaxws:endpoint ...&gt;
    *&lt;jaxws:outInterceptors&gt;
      &lt;list&gt;
        &lt;bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor" /&gt;
      &lt;/list&gt;
    &lt;/jaxws:outInterceptors&gt;*
  &lt;/jaxws:endpoint&gt;
&lt;/beans&gt;
```

**更多信息**

有关使用 Spring 配置配置端点的更多信息，请参阅 第 IV 部分 “配置 Web 服务端点”。

## 59.3. 以编程方式添加拦截器

### 59.3.1. 添加拦截器的方法

可以使用以下任一方法之一，以编程方式将拦截器附加到端点：

- InterceptorProvider API

- Java 注解

使用 InterceptorProvider API 可让开发人员将拦截器附加到具有拦截器链的任何运行时组件，但它需要使用底层 Apache CXF 类。Java 注释只能添加到服务接口或服务实施中，但开发人员可以在 JAX-WS API 或 JAX-RS API 中保留。

### 59.3.2. 使用拦截器供应商 API

#### 概述

拦截器可以使用实现 InterceptorProvider 接口的任何组件注册，这些组件在 拦截器供应商接口 中显示。

#### 拦截器供应商接口

```
package org.apache.cxf.interceptor;

import java.util.List;

public interface InterceptorProvider
{
    List<Interceptor<? extends Message>> getInInterceptors();

    List<Interceptor<? extends Message>> getOutInterceptors();

    List<Interceptor<? extends Message>> getInFaultInterceptors();

    List<Interceptor<? extends Message>> getOutFaultInterceptors();
}
```

接口中的四个方法允许您检索每个端点的拦截器链作为 Java List 对象。开发人员可使用 Java List 对象提供的方法，向任何链添加或删除拦截器。

#### 流程

要使用 InterceptorProvider API 将拦截器附加到运行时组件的拦截器链，您必须：

1. 使用将拦截器附加到的链访问运行时组件。

   开发人员必须使用 Apache CXF 特定的 API 从标准 Java 应用程序代码访问运行时组件。运行时组件通常通过将 JAX-WS 或 JAX-RS 工件转换为底层 Apache CXF 对象来访问。

2. 创建拦截器的实例。

3. 使用正确的 get 方法来检索所需的拦截器链。

4. 使用 List 对象的 add （） 方法将拦截器附加到拦截器链。

   此步骤通常与检索拦截器链结合使用。

### 将拦截器附加到消费者

以编程方式将拦截器附加到消费者 显示将拦截器附加到 JAX-WS 消费者的入站拦截器链的代码。

#### 以编程方式将拦截器附加到消费者

```
package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import org.apache.cxf.endpoint.Client;

public class Client
{
  public static void main(String args[])
  {
    QName serviceName = new QName("http://demo.eric.org", "stockQuoteReporter");
    Service s = Service.create(serviceName);

    QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
    s.addPort(portName, "http://schemas.xmlsoap.org/soap/", "http://localhost:9000/EricStockQuote");

    quoteReporter proxy = s.getPort(portName, quoteReporter.class);

    Client cxfClient = (Client) proxy;
```

```
    ValidateInterceptor validInterceptor = new ValidateInterceptor();
    cxfClient.getInInterceptor().add(validInterceptor);

    ...
  }
}
```

以编程方式将拦截器附加到消费者 中的代码执行以下操作：

为消费者创建 JAX-WS Service 对象。

在提供消费者目标地址的 Service 对象中添加一个端口。

创建用于在服务提供商上调用方法的代理。

将代理发送到 org.apache.cxf.endpoint.Client 类型。

创建拦截器的实例。

将拦截器附加到入站拦截器链。

### 将拦截器附加到服务提供商

以编程方式将拦截器附加到服务提供商 显示将拦截器附加到服务提供商的出站拦截器链的代码。

### 以编程方式将拦截器附加到服务提供商

```
package com.fusesource.demo;
import java.util.*;

import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
```

```
import org.apache.cxf.frontend.EndpointImpl;

public class stockQuoteReporter implements quoteReporter
{
  ...
  public stockQuoteReporter()
  {
    ServerFactoryBean sfb = new ServerFactoryBean();
    Server server = sfb.create();
    EndpointImpl endpt = server.getEndpoint();

    AuthTokenInterceptor authInterceptor = new AuthTokenInterceptor();

    endpt.getOutInterceptor().add(authInterceptor);
  }
}
```

*以编程方式将拦截器附加到服务提供商* **中的代码执行以下操作：**

*创建一个* **ServerFactoryBean** *对象，它提供对底层* **Apache CXF** *对象的访问。*

*获取* **Apache CXF** *用于代表端点的* **Server** *对象。*

*获取服务提供商的* **Apache CXF EndpointImpl** *对象。*

*创建拦截器的实例。*

*将拦截器附加到端点；s* **outbound interceptor** *链。*

### *将拦截器附加到总线*

*将拦截器附加到总线* *显示将拦截器附加到总线入站拦截器链的代码。*

### *将拦截器附加到总线*

```
import org.apache.cxf.BusFactory;
org.apache.cxf.Bus;

...

Bus bus = BusFactory.getDefaultBus();

WatchInterceptor watchInterceptor = new WatchInterceptor();

bus..getInInterceptor().add(watchInterceptor);

...
```

将拦截器附加到总线 中的代码执行以下操作：

获取运行时实例的默认总线。

创建拦截器的实例。

将拦截器附加到入站拦截器链。

WatchInterceptor 将附加到运行时实例创建的所有端点的入站拦截器链。

### 59.3.3. 使用 Java 注解

**概述**

Apache CXF 提供四个 Java 注解，供开发人员指定端点使用的拦截器链。与将拦截器附加到端点的其它方法不同，注解会附加到应用程序级别的工件中。使用的工件决定了注解的影响范围。

**放置注解的位置**

注解可以放置在以下工件中：

- 定义端点端点的服务端点接口(SEI)

如果注解放置在 SEI 上，则实施接口的所有服务提供商以及所有使用 SEI 创建代理的用户都会受到影响。

- 服务实现类

  如果注解放置在实施类上，则所有使用实现类的服务供应商都会受到影响。

注解

注解都包括在 org.apache.cxf.interceptor 软件包中，并在 表 59.2 “拦截器链注解” 中描述。

表 59.2. 拦截器链注解

| 注解 | 描述 |
| --- | --- |
| InInterceptors | 指定入站拦截器链的拦截器。 |
| OutInterceptors | 指定出站拦截器链的拦截器。 |
| InFaultInterceptors | 指定入站故障拦截器链的拦截器。 |
| OutFaultInterceptors | 指定出站故障拦截器链的拦截器。 |

列出拦截器

拦截器列表以完全限定类名称列表的形式使用 在链注解中列出拦截器的语法 中显示的语法指定。

在链注解中列出拦截器的语法

```
interceptors={"interceptor1", "interceptor2", ..., "interceptorN"}
```

Example

将拦截器附加到服务实现 显示将两个拦截器附加到使用 SayHiImpl 提供逻辑的端点的入站拦截器链的注解。

*将拦截器附加到服务实现*

```
import org.apache.cxf.interceptor.InInterceptors;

@InInterceptors(interceptors={"com.sayhi.interceptors.FirstLast",
"com.sayhi.interceptors.LogName"})
public class SayHiImpl implements SayHi
{
  ...
}
```

# 第 60 章 在 FLY 操作拦截器链

## 摘要

拦截器可以重新配置端点的拦截器链，作为其消息处理逻辑的一部分。它可以添加新的拦截器、删除拦截器、重新排序拦截器，甚至挂起拦截器链。任何针对的操作都特定于调用，因此每次在消息交换中涉及端点时，都会使用原始链。

## 概述

拦截器链仅在创建消息交换时才有效。每个消息都包含对负责处理它的拦截器链的引用。开发人员可以使用此引用来更改消息的拦截器链。由于链是每个交换的，对消息的拦截器链所做的任何更改都不会影响其他消息交换。

### 链生命周期

拦截器链和链中的拦截器会根据每个调用进行实例化。当调用端点以参与消息交换时，所需的拦截器链会与其拦截器的实例一起实例化。当完成导致创建拦截器链的消息交换时，链及其拦截器实例将被销毁。

这意味着，您对拦截器链或拦截器字段所做的任何更改都不会在消息交换之间保留。因此，如果拦截器仅在活跃的链中放置另一个拦截器，则只有活跃的链生效。未来任何消息交换都将从 pristine 状态创建，具体由端点的配置决定。它还意味着，开发人员无法在拦截器中设置标记，该标志会改变将来的消息处理。

如果拦截器需要与将来的实例一起传递信息，可以在消息上下文中设置属性。该上下文在消息交换之间有效。

### 获取拦截器链

更改消息的拦截器链的第一步是获取拦截器链。这可以通过在 例 60.1 “获取拦截器链的方法” 中显示的 Message.getInterceptorChain （） 方法完成。拦截器链返回为 org.apache.cxf.interceptor.InterceptorChain 对象。

> **例 60.1. 获取拦截器链的方法**
>
> InterceptorChaingetInterceptorChain

### 添加拦截器

InterceptorChain 对象有两个方法，如 例 60.2 “将拦截器添加到拦截器链的方法” 所示，用于将拦截器添加到拦截器链。一个允许您添加单个拦截器，另一个允许您添加多个拦截器。

**例 60.2. 将拦截器添加到拦截器链的方法**

```
添加Interceptor<? extends Message>iaddCollection<Interceptor<? extends Message>>i
```

例 60.3 “将拦截器添加到拦截器链(on-the-fly)链” 显示将单个拦截器添加到消息的拦截器链的代码。

**例 60.3. 将拦截器添加到拦截器链(on-the-fly)链**

```
void handleMessage(Message message)
{
  ...
  AddledIntereptor addled = new AddledIntereptor();
  InterceptorChain chain = message.getInterceptorChain();
  chain.add(addled);
  ...
}
```

例 60.3 “将拦截器添加到拦截器链(on-the-fly)链” 中的代码执行以下操作：

实例化要添加到链中的拦截器副本。

**重要**

添加到链的拦截器应位于与当前拦截器相同的阶段或后一个阶段，而不是当前的拦截器。

获取当前消息的拦截器链。

向链中添加新的拦截器。

*删除拦截器*

*InterceptorChain 对象有一个方法，如 例 60.4 “从拦截器链中删除拦截器的方法” 所示，用于从拦截器链中删除拦截器。*

*例 60.4. 从拦截器链中删除拦截器的方法*

*删除Interceptor<? extends Message>i*

*例 60.5 “从拦截器链(on-the-fly)中删除拦截器” 显示从消息的拦截器链中删除拦截器的代码。*

*例 60.5. 从拦截器链(on-the-fly)中删除拦截器*

```
void handleMessage(Message message)
{
  ...
  Iterator<Interceptor<? extends Message>> iterator =
          message.getInterceptorChain().iterator();
  Interceptor<?> removeInterceptor =  null;
  for (; iterator.hasNext(); ) {
    Interceptor<?> interceptor = iterator.next();
    if (interceptor.getClass().getName().equals("InterceptorClassName")) {
      removeInterceptor = interceptor;
      break;
    }
  }

  if (removeInterceptor !=  null) {
    log.debug("Removing interceptor {}",removeInterceptor.getClass().getName());
    message.getInterceptorChain().remove(removeInterceptor);
  }
  ...
}
```

*其中 InterceptorClassName 是您要从链中删除的拦截器的类名称。*

# 第 61 章 JAX-RS 2.0 FILTERS 和 INTERCEPTORS

*摘要*

*JAX-RS 2.0 为 REST 调用定义了安装过滤器和拦截器的标准 API 和语义。过滤器和拦截器通常用于提供日志记录、身份验证、授权、消息压缩、消息加密等功能。*

## 61.1. JAX-RS 过滤器和 INTERCEPTORS 简介

### 概述

*本节概述了 JAX-RS 过滤器和拦截器的处理管道，突出显示可能安装过滤器链或拦截器链的扩展点。*

#### 过滤器

*JAX-RS 2.0 过滤器 是一类插件，使开发人员能够访问通过 CXF 客户端或服务器传递的所有 JAX-RS 消息。过滤器适合处理与消息关联的元数据：HTTP 标头、查询参数、介质类型和其他元数据。过滤器具有中止消息调用的功能（例如，对于安全插件很有用）。*

*如果您愿意，您可以在每个扩展点处安装多个过滤器，在这种情况下，在链中执行过滤器（执行顺序未定义，除非为每个安装的过滤器指定了 优先级值 ）。*

#### 拦截器

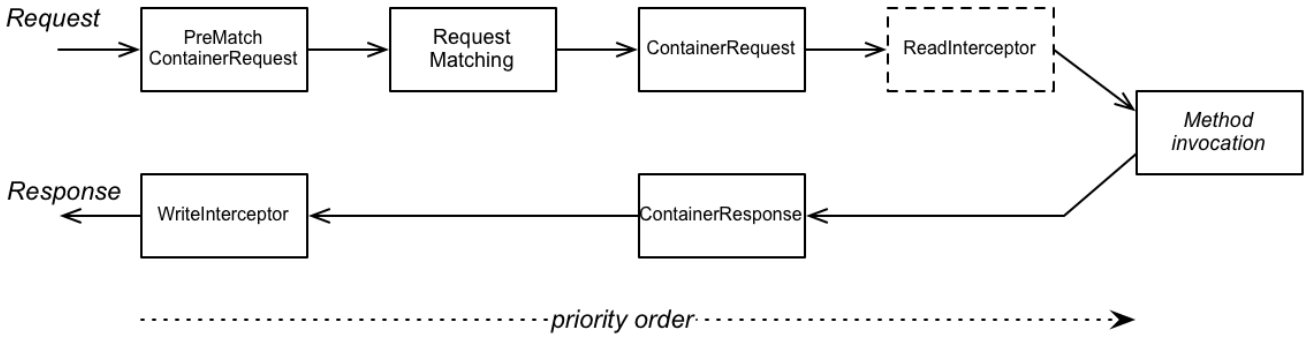
*JAX-RS 2.0 拦截器 是一类插件，使开发人员能够访问消息正文，因为它正在读取或写入。拦截器嵌套在 MessageBodyReader.readFrom 方法调用中（用于读取拦截器）或 MessageBodyWriter.writeTo 方法调用（用于 writer interceptors）。*

*如果您愿意，您可以在每个扩展点处安装多个拦截器，在这种情况下，在链中执行拦截器（执行顺序未定义，除非为每个安装的拦截器指定了 优先级值 ）。*

#### 服务器处理管道

*图 61.1 “server-Side Filter 和 Interceptor Extension Points” 显示了在服务器端安装的 JAX-RS 过滤器和拦截器的处理管道的概述。*

*图 61.1. server-Side Filter 和 Interceptor Extension Points*




*服务器扩展点*

*在服务器处理管道中，您可以在以下扩展点中添加过滤器（或拦截器）：*

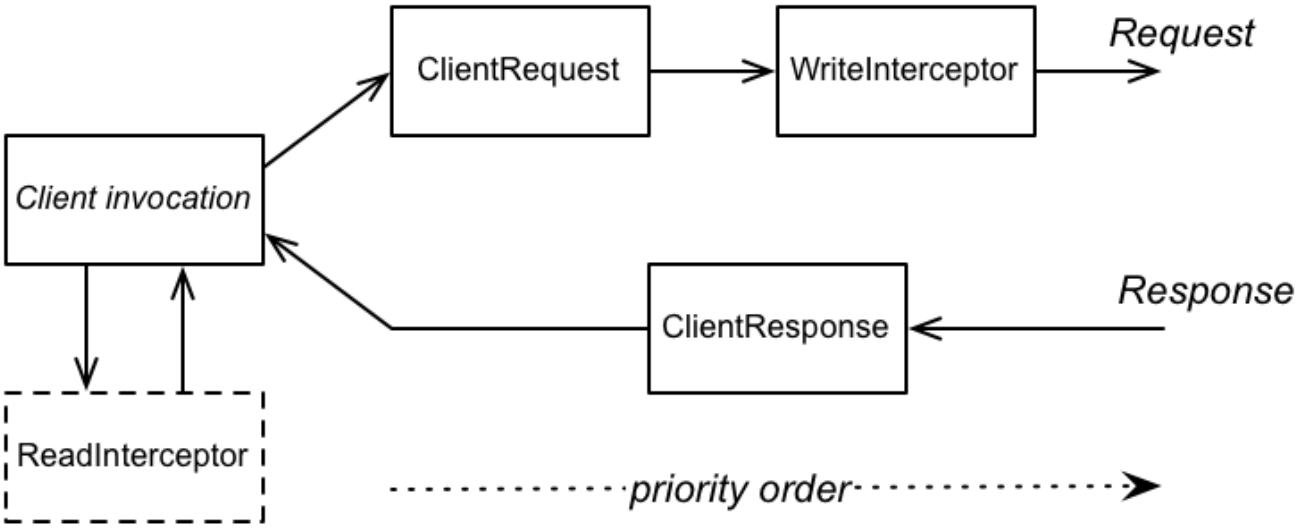
1. ***PreMatchContainerRequest** 过滤器*
2. ***ContainerRequest** 过滤器*
3. ***ReadInterceptor***
4. ***ContainerResponse** 过滤器*
5. ***WriteInterceptor***

*请注意，发生资源匹配 **前** 会达到 **PreMatchContainerRequest** 扩展点，因此此时将无法使用一些上下文元数据。*

*客户端处理管道*

*图 61.2 “client-Side Filter 和 Interceptor Extension Points” 显示在客户端端安装的 **JAX-RS** 过滤器和拦截器的处理管道的概述。*

*图 61.2. client-Side Filter 和 Interceptor Extension Points*




*客户端扩展点*

在客户端处理管道中，您可以在以下扩展点中添加过滤器（或拦截器）：

1. **ClientRequest** 过滤器
2. **WriteInterceptor**
3. **ClientResponse filter**
4. **ReadInterceptor**

*过滤和拦截器顺序*

如果您在同一扩展点安装多个过滤器或拦截器，过滤器的执行顺序取决于分配给它们的优先级（在 **Java** 源中使用 **@Priority** 注解）。优先级以整数值表示。通常，优先级较高的过滤器会更接近服务器端的资源方法调用；而具有 较低优先级 数字的过滤器则更接近客户端调用。换句话说，过滤器和拦截器 在请求 消息上执行，按优先级号的 降序 执行；而对响应消息的过滤器和拦截器操作则按优先级号 降序 执行。

*过滤类*

可以实施以下 **Java** 接口以创建自定义 **REST** 消息过滤器：

- javax.ws.rs.container.ContainerRequestFilter
- javax.ws.rs.container.ContainerResponseFilter
- javax.ws.rs.client.ClientRequestFilter
- javax.ws.rs.client.ClientResponseFilter

拦截器类

可以实施以下 Java 接口以创建自定义 REST 消息拦截器：

- javax.ws.rs.ext.ReaderInterceptor
- javax.ws.rs.ext.WriterInterceptor

### 61.2. 容器请求过滤器

#### 概述

本节介绍如何实施和注册容器请求过滤器，该过滤器用于截获 server （容器）端的传入请求消息。容器请求过滤器通常用于处理服务器端的标头，并可用于任何类型的通用请求处理（即处理独立于特定资源方法的处理）。

此外，容器请求过滤器是特殊情况的一部分，因为它可以在两个不同的扩展点上安装：PreMatchContainerRequest （资源匹配步骤前）和 ContainerRequest （资源匹配步骤之后）。

#### ContainerRequestFilter 接口

javax.ws.rs.container.ContainerRequestFilter 接口定义如下：

```
// Java
...
package javax.ws.rs.container;
```

```
import java.io.IOException;

public interface ContainerRequestFilter {
    public void filter(ContainerRequestContext requestContext) throws IOException;
}
```

通过实现 **ContainerRequestFilter** 接口，您可以在服务器端为以下扩展点创建一个过滤器：

- **PreMatchContainerRequest**
- **ContainerRequest**

### ContainerRequestContext 接口

**ContainerRequestFilter** 的过滤器 方法接收类型为 **javax.ws.rs.container.ContainerRequestContext** 的单个参数，它可用于访问传入的请求消息及其关联的元数据。**ContainerRequestContext** 接口定义如下：

```
// Java
...
package javax.ws.rs.container;

import java.io.InputStream;
import java.net.URI;
import java.util.Collection;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import javax.ws.rs.core.Cookie;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.UriInfo;


public interface ContainerRequestContext {

    public Object getProperty(String name);

    public Collection getPropertyNames();

    public void setProperty(String name, Object object);
```

```
    public void removeProperty(String name);

    public UriInfo getUriInfo();

    public void setRequestUri(URI requestUri);

    public void setRequestUri(URI baseUri, URI requestUri);

    public Request getRequest();

    public String getMethod();

    public void setMethod(String method);

    public MultivaluedMap getHeaders();

    public String getHeaderString(String name);

    public Date getDate();

    public Locale getLanguage();

    public int getLength();

    public MediaType getMediaType();

    public List getAcceptableMediaTypes();

    public List getAcceptableLanguages();

    public Map getCookies();

    public boolean hasEntity();

    public InputStream getEntityStream();

    public void setEntityStream(InputStream input);

    public SecurityContext getSecurityContext();

    public void setSecurityContext(SecurityContext context);

    public void abortWith(Response response);
}
```

***PreMatchContainerRequest* 过滤器的实现示例**

*要为* ***PreMatchContainerRequest*** *扩展点（即，在资源匹配之前执行过滤器的位置）实现容器请求过滤器，请定义一个实施* ***ContainerRequestFilter*** *接口的类，确保为类添加* ***@PreMatching*** *注解（选择* ***PreMatchContainerRequest*** *扩展点）。*

*例如，以下代码显示了在 PreMatchContainerRequest 扩展点中安装的简单容器请求过滤器示例，其优先级为 20：*

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.annotation.Priority;
import javax.ws.rs.ext.Provider;

@PreMatching
@Priority(value = 20)
@Provider
public class SamplePreMatchContainerRequestFilter implements
  ContainerRequestFilter {

  public SamplePreMatchContainerRequestFilter() {
    System.out.println("SamplePreMatchContainerRequestFilter starting up");
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    System.out.println("SamplePreMatchContainerRequestFilter.filter() invoked");
  }
}
```

***ContainerRequest 过滤器的实现示例***

*要为 ContainerRequest 扩展点实施容器请求过滤器（即，过滤器在资源匹配 后执行 的位置），请定义一个实施 ContainerRequestFilter 接口的类，而无需 @PreMatching 注解。*

*例如，以下代码显示了在 ContainerRequest 扩展点中安装的简单容器请求过滤器示例，其优先级为 30 ：*

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;

@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {

  public SampleContainerRequestFilter() {
    System.out.println("SampleContainerRequestFilter starting up");
```

```
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    System.out.println("SampleContainerRequestFilter.filter() invoked");
  }
}
```

**注入 ResourceInfo**

在 ContainerRequest 扩展点（即发生资源匹配后），可以通过注入 ResourceInfo 类来访问匹配的资源类和资源方法。例如，以下代码演示了如何注入 ResourceInfo 类作为 ContainerRequestFilter 类的字段：

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;
import javax.ws.rs.core.Context;

@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {

  @Context
  private ResourceInfo resinfo;

  public SampleContainerRequestFilter() {
    ...
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    String resourceClass = resinfo.getResourceClass().getName();
    String methodName    = resinfo.getResourceMethod().getName();
    System.out.println("REST invocation bound to resource class: " + resourceClass);
    System.out.println("REST invocation bound to resource method: " + methodName);
  }
}
```

**中止调用**

通过创建合适的容器请求过滤器实施，可以中止服务器端调用。通常，这对在服务器端实施安全功能很有用：例如，为了实现身份验证功能或授权功能。如果传入的请求无法成功进行身份验证，您可以在容器请求过滤器内中止调用。

例如，以下预匹配功能尝试从 URI 的查询参数中提取用户名和密码，并调用验证方法来检查用户名和密码凭证。如果身份验证失败，则调用在 ContainerRequestContext 对象上的 abortWith 中止，传递要返回到客户端的错误响应。

```
// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.ext.Provider;

@PreMatching
@Priority(value = 20)
@Provider
public class SampleAuthenticationRequestFilter implements
  ContainerRequestFilter {

  public SampleAuthenticationRequestFilter() {
    System.out.println("SampleAuthenticationRequestFilter starting up");
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    ResponseBuilder responseBuilder = null;
    Response response = null;

    String userName = requestContext.getUriInfo().getQueryParameters().getFirst("UserName");
    String password = requestContext.getUriInfo().getQueryParameters().getFirst("Password");
    if (authenticate(userName, password) == false) {
      responseBuilder = Response.serverError();
      response = responseBuilder.status(Status.BAD_REQUEST).build();
      requestContext.abortWith(response);
    }
  }

  public boolean authenticate(String userName, String password) {
    // Perform authentication of 'user'
    ...
  }
}
```

### 绑定服务器请求过滤器

要 绑定 服务器请求过滤器（即要将它安装到 Apache CXF 运行时中），请执行以下步骤：

1. 将 @Provider 注释添加到容器请求过滤器类，如以下代码片段所示：

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;

@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {
  ...
}
```

当容器请求过滤器实施加载到 Apache CXF 运行时中时，REST 实施会自动扫描加载的类，以搜索标有 @Provider 注释( 扫描阶段)。

2. 在 XML 中定义 JAX-RS 服务器端点（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”）时，将服务器请求过滤器添加到 jaxrs:providers 元素中的供应商列表中。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
    ...
    <jaxrs:server id="customerService" address="/customers">
      ...
      <jaxrs:providers>
        <ref bean="filterProvider" />
      </jaxrs:providers>
      <bean id="filterProvider"
class="org.jboss.fuse.example.SampleContainerRequestFilter"/>

    </jaxrs:server>

</blueprint>
```

**注意**

此步骤是 Apache CXF 的非标准要求。根据 JAX-RS 标准，严格说，@Provider 注释应当是绑定过滤器所需的所有内容。但是在实践中，标准方法有些不灵活，当许多库包含在大型项目中时，可能会导致禁止提供程序。

## 61.3. 容器响应过滤器

### 概述

本节介绍如何实施和注册 容器响应过滤器，该过滤器 用于截获服务器端的传出响应消息。容器响应过滤器可用于在响应消息中自动填充标头，通常可用于任何类型的通用响应处理。

### ContainerResponseFilter 接口

javax.ws.rs.container.ContainerResponseFilter 接口定义如下：

```
// Java
...
package javax.ws.rs.container;

import java.io.IOException;

public interface ContainerResponseFilter {
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext
responseContext)
            throws IOException;
}
```

通过实施 ContainerResponseFilter，您可以为服务器端的 ContainerResponse 扩展点创建一个过滤器，它会在调用执行后过滤响应消息。

**注意**

容器响应过滤器可让您访问请求消息（通过 requestContext 参数）和响应消息（通过 responseContext 消息），但在此阶段只能修改响应。

### ContainerResponseContext 接口

ContainerResponseFilter 的过滤器 方法接收两个参数：参数是 javax.ws.rs.container.ContainerRequestContext （请参阅 “ContainerRequestContext 接口”一节），以及类型为 javax.ws.rs.container.ContainerResponseContext 的参数，可用于访问传出响应消息及其相关元数据。

ContainerResponseContext 接口定义如下：

```
// Java
```

```
...
package javax.ws.rs.container;

import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.net.URI;
import java.util.Date;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.Link;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.MessageBodyWriter;

public interface ContainerResponseContext {

    public int getStatus();

    public void setStatus(int code);

    public Response.StatusType getStatusInfo();

    public void setStatusInfo(Response.StatusType statusInfo);

    public MultivaluedMap<String, Object> getHeaders();

    public abstract MultivaluedMap<String, String> getStringHeaders();

    public String getHeaderString(String name);

    public Set<String> getAllowedMethods();

    public Date getDate();

    public Locale getLanguage();

    public int getLength();

    public MediaType getMediaType();

    public Map<String, NewCookie> getCookies();

    public EntityTag getEntityTag();

    public Date getLastModified();

    public URI getLocation();

    public Set<Link> getLinks();
```

```
    boolean hasLink(String relation);

    public Link getLink(String relation);

    public Link.Builder getLinkBuilder(String relation);

    public boolean hasEntity();

    public Object getEntity();

    public Class<?> getEntityClass();

    public Type getEntityType();

    public void setEntity(final Object entity);

    public void setEntity(
            final Object entity,
            final Annotation[] annotations,
            final MediaType mediaType);

    public Annotation[] getEntityAnnotations();

    public OutputStream getEntityStream();

    public void setEntityStream(OutputStream outputStream);
}
```

*实现**示例***

***要为 ContainerResponse 扩展点实施容器响应过滤器（即，在服务器端执行过滤器后执行过滤器），请定义一个实施 ContainerResponseFilter 接口的类。***

***例如，以下代码显示了在 ContainerResponse 扩展点中安装的简单容器响应过滤器示例，其优先级为 10：***

```
// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
@Priority(value = 10)
public class SampleContainerResponseFilter implements ContainerResponseFilter {

  public SampleContainerResponseFilter() {
    System.out.println("SampleContainerResponseFilter starting up");
```

```
    }

    @Override
    public void filter(
        ContainerRequestContext requestContext,
        ContainerResponseContext responseContext
    )
    {
      // This filter replaces the response message body with a fixed string
      if (responseContext.hasEntity()) {
        responseContext.setEntity("New message body!");
      }
    }
}
```

*绑定服务器响应过滤器*

*要 绑定 服务器响应过滤器（即要将它安装到 Apache CXF 运行时中），请执行以下步骤：*

1. *将 @Provider 注释添加到容器响应过滤器类，如以下代码片段所示：*

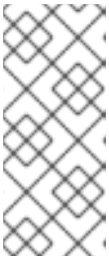
```
// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
@Priority(value = 10)
public class SampleContainerResponseFilter implements ContainerResponseFilter {
  ...
}
```

*当容器响应过滤器实施加载到 Apache CXF 运行时中时，REST 实施会自动扫描加载的类，以搜索标有 @Provider 注释( 扫描阶段)。*

2. *在 XML 中定义 JAX-RS 服务器端点（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”）时，将服务器响应过滤器添加到 jaxrs:providers 元素中的供应商列表中。*

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

```
>
   ...
   <jaxrs:server id="customerService" address="/customers">
     ...
     <jaxrs:providers>
       <ref bean="filterProvider" />
     </jaxrs:providers>
     <bean id="filterProvider"
class="org.jboss.fuse.example.SampleContainerResponseFilter"/>

   </jaxrs:server>

</blueprint>
```



**注意**

此步骤是 Apache CXF 的非标准要求。根据 JAX-RS 标准，严格说，@Provider 注释应当是绑定过滤器所需的所有内容。但是在实践中，标准方法有些不灵活，当许多库包含在大型项目中时，可能会导致禁止提供程序。

## 61.4. 客户端请求过滤器

### 概述

本节介绍如何实施和注册客户端请求过滤器，该过滤器用于在客户端一侧截获传出请求消息。客户端请求过滤器通常用于处理标头，并可用于任何类型的通用请求处理。

### ClientRequestFilter 接口

javax.ws.rs.client.ClientRequestFilter 接口定义如下：

```
// Java
package javax.ws.rs.client;
...
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.client.ClientRequestContext;
...
public interface ClientRequestFilter {
    void filter(ClientRequestContext requestContext) throws IOException;
}
```

通过实施 ClientRequestFilter，您可以为客户端端的 ClientRequest 扩展点创建一个过滤器，该过滤器会在向服务器发送消息前过滤请求消息。

***ClientRequestContext 接口***

***ClientRequestFilter 的过滤器 方法接收类型为 javax.ws.rs.client.ClientRequestContext 的单个参数，它可用于访问传出请求消息及其关联的元数据。ClientRequestContext 接口定义如下：***

```
// Java
...
package javax.ws.rs.client;

import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.net.URI;
import java.util.Collection;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import javax.ws.rs.core.Configuration;
import javax.ws.rs.core.Cookie;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.MessageBodyWriter;

public interface ClientRequestContext {

    public Object getProperty(String name);

    public Collection<String> getPropertyNames();

    public void setProperty(String name, Object object);

    public void removeProperty(String name);

    public URI getUri();

    public void setUri(URI uri);

    public String getMethod();

    public void setMethod(String method);

    public MultivaluedMap<String, Object> getHeaders();

    public abstract MultivaluedMap<String, String> getStringHeaders();

    public String getHeaderString(String name);

    public Date getDate();

    public Locale getLanguage();
```

```
    public MediaType getMediaType();

    public List<MediaType> getAcceptableMediaTypes();

    public List<Locale> getAcceptableLanguages();

    public Map<String, Cookie> getCookies();

    public boolean hasEntity();

    public Object getEntity();

    public Class<?> getEntityClass();

    public Type getEntityType();

    public void setEntity(final Object entity);

    public void setEntity(
            final Object entity,
            final Annotation[] annotations,
            final MediaType mediaType);

    public Annotation[] getEntityAnnotations();

    public OutputStream getEntityStream();

    public void setEntityStream(OutputStream outputStream);

    public Client getClient();

    public Configuration getConfiguration();

    public void abortWith(Response response);
}
```

*实现示例*

***要为 ClientRequest 扩展点实施客户端请求过滤器（即，在发送请求消息之前执行过滤器），请定义一个实施 ClientRequestFilter 接口的类。***

***例如，以下代码显示了在 ClientRequest 扩展点中安装的简单客户端请求过滤器示例，其优先级为 20：***

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.annotation.Priority;
```

```
@Priority(value = 20)
public class SampleClientRequestFilter implements ClientRequestFilter {

  public SampleClientRequestFilter() {
    System.out.println("SampleClientRequestFilter starting up");
  }

  @Override
  public void filter(ClientRequestContext requestContext) {
    System.out.println("ClientRequestFilter.filter() invoked");
  }
}
```

**中止调用**

**可以通过实施合适的客户端请求过滤器来中止客户端调用。例如，您可以实施客户端侧过滤器来检查请求是否已正确格式化，如有必要，终止请求。**

**以下测试代码始终中止请求，并将 BAD_REQUEST HTTP 状态返回到客户端调用代码：**

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
import javax.annotation.Priority;

@Priority(value = 10)
public class TestAbortClientRequestFilter implements ClientRequestFilter {

  public TestAbortClientRequestFilter() {
    System.out.println("TestAbortClientRequestFilter starting up");
  }

  @Override
  public void filter(ClientRequestContext requestContext) {
    // Test filter: aborts with BAD_REQUEST status
    requestContext.abortWith(Response.status(Status.BAD_REQUEST).build());
  }
}
```

**注册客户端请求过滤器**

**使用 JAX-RS 2.0 客户端 API，您可以在 javax.ws.rs. client.Client 对象或 javax.ws.rs.client.WebTarget 对象上直接注册客户端请求过滤器。实际上，这意味着客户端请求过滤器可以选择性地应用到不同的范围，以便只有特定的 URI 路径会受到过滤器的影响。**

例如，以下代码演示了如何注册 SampleClientRequestFilter 过滤器，使其适用于使用 客户端 对象进行的所有调用；以及如何注册 TestAbortClientRequestFilter 过滤器，使其仅适用于 rest/TestAbortClientRequest RequestRequestRequest 的子路径。

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(new SampleClientRequestFilter());
WebTarget target = client
  .target("http://localhost:8001/rest/TestAbortClientRequest");
target.register(new TestAbortClientRequestFilter());
```

### 61.5. 客户端响应过滤器

#### 概述

本节介绍如何实施和注册客户端 响应过滤器，该过滤器 用于截获客户端端的传入响应消息。客户端响应过滤器可用于客户端端的任何通用响应处理。

#### ClientResponseFilter 接口

javax.ws.rs.client.ClientResponseFilter 接口定义如下：

```
// Java
package javax.ws.rs.client;
...
import java.io.IOException;

public interface ClientResponseFilter {
    void filter(ClientRequestContext requestContext, ClientResponseContext responseContext)
            throws IOException;
}
```

通过实施 ClientResponseFilter，您可以在客户端端为 ClientResponse 扩展点创建一个过滤器，它会在从服务器接收响应消息后过滤响应消息。

#### ClientResponseContext 接口

**ClientResponseFilter 的过滤器 方法接收两个参数：参数是 javax.ws.rs.client.ClientRequestContext （请参阅 “ClientRequestContext 接口”一节），以及类型为 javax.ws.rs.client.ClientResponseContext 的参数，可用于访问传出响应消息及其相关元数据。**

**ClientResponseContext 接口定义如下：**

```
// Java
...
package javax.ws.rs.client;

import java.io.InputStream;
import java.net.URI;
import java.util.Date;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.Link;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;

public interface ClientResponseContext {

    public int getStatus();

    public void setStatus(int code);

    public Response.StatusType getStatusInfo();

    public void setStatusInfo(Response.StatusType statusInfo);

    public MultivaluedMap<String, String> getHeaders();

    public String getHeaderString(String name);

    public Set<String> getAllowedMethods();

    public Date getDate();

    public Locale getLanguage();

    public int getLength();

    public MediaType getMediaType();

    public Map<String, NewCookie> getCookies();

    public EntityTag getEntityTag();

    public Date getLastModified();
```

```
    public URI getLocation();

    public Set<Link> getLinks();

    boolean hasLink(String relation);

    public Link getLink(String relation);

    public Link.Builder getLinkBuilder(String relation);

    public boolean hasEntity();

    public InputStream getEntityStream();

    public void setEntityStream(InputStream input);
}
```

*实现示例*

*要为 **ClientResponse** 扩展点实施客户端响应过滤器（即，在从服务器接收响应消息后执行过滤器），请定义一个实施 **ClientResponseFilter** 接口的类。*

*例如，以下代码显示了在 **ClientResponse** 扩展点中安装的简单客户端响应过滤器示例，其优先级为 **20**：*

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientResponseContext;
import javax.ws.rs.client.ClientResponseFilter;
import javax.annotation.Priority;

@Priority(value = 20)
public class SampleClientResponseFilter implements ClientResponseFilter {

  public SampleClientResponseFilter() {
    System.out.println("SampleClientResponseFilter starting up");
  }

  @Override
  public void filter(
      ClientRequestContext requestContext,
      ClientResponseContext responseContext
  )
  {
    // Add an extra header on the response
    responseContext.getHeaders().putSingle("MyCustomHeader", "my custom data");
  }
}
```

注册客户端响应过滤器

使用 JAX-RS 2.0 客户端 API，您可以在 javax.ws.rs. client.Client 对象或 javax.ws.rs.client.WebTarget 对象上直接注册客户端响应过滤器。实际上，这意味着客户端请求过滤器可以选择性地应用到不同的范围，以便只有特定的 URI 路径会受到过滤器的影响。

例如，以下代码演示了如何注册 SampleClientResponseFilter 过滤器，使其适用于使用 client 对象进行的所有调用：

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(new SampleClientResponseFilter());
```

### 61.6. ENTITY READER INTERCEPTOR

概述

本节介绍如何在客户端或服务器端读取消息正文时，实施和注册 实体阅读器拦截器。这通常可用于请求正文的通用转换，如加密和解密、压缩和解压缩。

ReaderInterceptor 接口

javax.ws.rs.ext.ReaderInterceptor 接口定义如下：

```
// Java
...
package javax.ws.rs.ext;

public interface ReaderInterceptor {
    public Object aroundReadFrom(ReaderInterceptorContext context)
            throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

通过实施 ReaderInterceptor 接口，您可以截获消息正文(实体 对象)，因为它在服务器端或客户端上读取。您可以在以下任何一个上下文中使用实体读取器：

- 服务器端- 如果作为服务器端拦截器绑定，实体读取器在应用代码访问时截获请求消息正文（在匹配的资源中）。根据 REST 请求的语义，消息正文可能无法被匹配资源访问，在这种情况下，读取器不会被调用。

- client side - 如果作为客户端侧拦截器绑定，实体读取器在客户端代码访问时截获响应消息正文。如果客户端代码没有显式访问响应消息（例如，通过调用 Response.getEntity 方法），则读取器拦截器不会被调用。

### ReaderInterceptorContext 接口

ReaderInterceptor 的 aroundReadFrom 方法接收类型为 javax.ws.rs.ext.ReaderInterceptorContext 的参数，可用于访问消息正文(实体 对象)和消息元数据。

ReaderInterceptorContext 接口定义如下：

```
// Java
...
package javax.ws.rs.ext;

import java.io.IOException;
import java.io.InputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MultivaluedMap;

public interface ReaderInterceptorContext extends InterceptorContext {

    public Object proceed() throws IOException, WebApplicationException;

    public InputStream getInputStream();

    public void setInputStream(InputStream is);

    public MultivaluedMap<String, String> getHeaders();
}
```

### InterceptorContext 接口

ReaderInterceptorContext 接口也支持从基础 InterceptorContext 接口继承的方法。

InterceptorContext 接口定义如下：

```
// Java
...
package javax.ws.rs.ext;

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.util.Collection;

import javax.ws.rs.core.MediaType;

public interface InterceptorContext {

    public Object getProperty(String name);

    public Collection<String> getPropertyNames();

    public void setProperty(String name, Object object);

    public void removeProperty(String name);

    public Annotation[] getAnnotations();

    public void setAnnotations(Annotation[] annotations);

    Class<?> getType();

    public void setType(Class<?> type);

    Type getGenericType();

    public void setGenericType(Type genericType);

    public MediaType getMediaType();

    public void setMediaType(MediaType mediaType);
}
```

**客户端上的实现示例**

**要为客户端实施实体读取器，请定义一个实施 ReaderInterceptor 接口的类。**

**例如，以下代码显示了客户端侧的实体读者拦截器示例（优先级为 10），在传入响应的消息正文中替换 红帽 的所有 COMPANY_NAME 实例：**

```
// Java
package org.jboss.fuse.example;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
```

```
import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
public class SampleClientReaderInterceptor implements ReaderInterceptor {

  @Override
  public Object aroundReadFrom(ReaderInterceptorContext interceptorContext)
        throws IOException, WebApplicationException
  {
    InputStream inputStream = interceptorContext.getInputStream();
    byte[] bytes = new byte[inputStream.available()];
    inputStream.read(bytes);
    String responseContent = new String(bytes);
    responseContent = responseContent.replaceAll("COMPANY_NAME", "Red Hat");
    interceptorContext.setInputStream(new ByteArrayInputStream(responseContent.getBytes()));

    return interceptorContext.proceed();
  }
}
```

**服务器端的实现示例**

**要为服务器端实施实体读取器，请定义一个实施 ReaderInterceptor 接口的类，并使用 @Provider 注释给它添加注解。**

**例如，以下代码显示了服务器端的实体读者拦截器示例（优先级为 10），在传入请求的消息正文中替换 红帽 的所有 COMPANY_NAME 实例：**

```
// Java
package org.jboss.fuse.example;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
@Provider
public class SampleServerReaderInterceptor implements ReaderInterceptor {

  @Override
  public Object aroundReadFrom(ReaderInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
```

```
    InputStream inputStream = interceptorContext.getInputStream();
    byte[] bytes = new byte[inputStream.available()];
    inputStream.read(bytes);
    String requestContent = new String(bytes);
    requestContent = requestContent.replaceAll("COMPANY_NAME", "Red Hat");
    interceptorContext.setInputStream(new ByteArrayInputStream(requestContent.getBytes()));

    return interceptorContext.proceed();
  }
}
```

**在客户端绑定读取器**

使用 JAX-RS 2.0 客户端 API，您可以在 javax.ws.rs.client.Client 对象或 javax.ws.rs.client.WebTarget 对象上直接注册实体 reader 拦截器。这表示，读者拦截器可以选择性地应用到不同的范围，以便只有特定的 URI 路径会受到拦截器的影响。

例如，以下代码演示了如何注册 SampleClientReaderInterceptor 拦截器，使其适用于使用 client 对象进行的所有调用：

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(SampleClientReaderInterceptor.class);
```

有关使用 JAX-RS 2.0 客户端注册拦截器的详情，请参考 第 49.5 节 “配置客户端端点”。

**在服务器端绑定读取器**

要在服务器端 绑定 读取器（即，要将其安装到 Apache CXF 运行时中），请执行以下步骤：

1. 将 @Provider 注释添加到 reader 拦截器类，如以下代码片段所示：

```
// Java
package org.jboss.fuse.example;
...
import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
```

```
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
@Provider
public class SampleServerReaderInterceptor implements ReaderInterceptor {
  ...
}
```

*当读卡器实施加载到 Apache CXF 运行时中时，REST 实施会自动扫描加载的类，以搜索标有 @Provider 注释( 扫描阶段)。*

2. *在 XML 中定义 JAX-RS 服务器端点（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”）时，将 reader 拦截器添加到 jaxrs:providers 元素中的供应商列表中。*

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
    ...
    <jaxrs:server id="customerService" address="/customers">
      ...
      <jaxrs:providers>
        <ref bean="interceptorProvider" />
      </jaxrs:providers>
      <bean id="interceptorProvider"
class="org.jboss.fuse.example.SampleServerReaderInterceptor"/>

    </jaxrs:server>

</blueprint>
```

**注意**

*此步骤是 Apache CXF 的非标准要求。根据 JAX-RS 标准，严格说，@Provider 注释应全部是绑定拦截器所必需的。但是在实践中，标准方法有些不灵活，当许多库包含在大型项目中时，可能会导致禁止提供程序。*

### 61.7. ENTITY WRITER INTERCEPTOR

**概述**

*本节介绍如何在客户端或服务器端 写入消息正文时截获实体拦截器。这通常可用于请求正文的通用转换，如加密和解密、压缩和解压缩。*

### WriterInterceptor 接口

javax.ws.rs.ext.WriterInterceptor 接口定义如下：

```
// Java
...
package javax.ws.rs.ext;

public interface WriterInterceptor {
    void aroundWriteTo(WriterInterceptorContext context)
            throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

通过实施 WriterInterceptor 接口，您可以截获消息正文(实体 对象)，因为它在服务器端或客户端上写入。您可以在以下任何一个上下文中使用实体拦截器：

- 服务器端- 如果作为服务器端拦截器绑定，实体写器将截获响应消息正文，紧接在响应消息正文后再发送回客户端。
- client side- 如果作为客户端侧拦截器绑定，实体写入器会截获请求消息正文，紧接在请求消息正文被封锁并发送到服务器。

### WriterInterceptorContext 接口

WriterInterceptor 的 aroundWriteTo 方法收到类型为 javax.ws.rs.ext.WriterInterceptorContext 的参数，可用于访问消息正文(实体 对象)和消息元数据。

WriterInterceptorContext 接口定义如下：

```
// Java
...
package javax.ws.rs.ext;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MultivaluedMap;

public interface WriterInterceptorContext extends InterceptorContext {
```

```
    void proceed() throws IOException, WebApplicationException;

    Object getEntity();

    void setEntity(Object entity);

    OutputStream getOutputStream();

    public void setOutputStream(OutputStream os);

    MultivaluedMap<String, Object> getHeaders();
}
```

### *InterceptorContext 接口*

***WriterInterceptorContext 接口也支持从基础 InterceptorContext 接口继承的方法。有关 InterceptorContext 的定义，请参阅 “InterceptorContext 接口”一节。***

#### *客户端上的实现示例*

***要为客户端实施实体拦截器，请定义一个实现 WriterInterceptor 接口的类。***

***例如，以下代码显示了客户端侧的实体写器（优先级为 10）的示例，它将额外文本行附加到传出请求的消息正文中：***

```
// Java
package org.jboss.fuse.example;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
public class SampleClientWriterInterceptor implements WriterInterceptor {

  @Override
  public void aroundWriteTo(WriterInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
    OutputStream outputStream = interceptorContext.getOutputStream();
    String appendedContent = "\nInterceptors always get the last word in.";
    outputStream.write(appendedContent.getBytes());
    interceptorContext.setOutputStream(outputStream);
```

```
    interceptorContext.proceed();
  }
}
```

**服务器端的实现示例**

**要为服务器端实施实体拦截器，请定义一个实施 WriterInterceptor 接口的类，并使用 @Provider 注释给它添加注解。**

**例如，以下代码显示了服务器端的实体写器示例（优先级为 10），这会将额外的文本行附加到传出请求的消息正文中：**

```
// Java
package org.jboss.fuse.example;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
@Provider
public class SampleServerWriterInterceptor implements WriterInterceptor {

  @Override
  public void aroundWriteTo(WriterInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
    OutputStream outputStream = interceptorContext.getOutputStream();
    String appendedContent = "\nInterceptors always get the last word in.";
    outputStream.write(appendedContent.getBytes());
    interceptorContext.setOutputStream(outputStream);

    interceptorContext.proceed();
  }
}
```

**在客户端绑定 writer 拦截器**

**使用 JAX-RS 2.0 客户端 API，您可以在 javax.ws.rs. client.Client 对象或 javax.ws.rs.client.WebTarget 对象上直接注册实体拦截器。实际上，这意味着 writer interceptor 可以选择性地应用到不同的范围，以便只有特定的 URI 路径会受到拦截器的影响。**

**例如，以下代码演示了如何注册 SampleClientReaderInterceptor 拦截器，使其适用于使用 client 对**

象进行的所有调用：

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(SampleClientReaderInterceptor.class);
```

有关使用 JAX-RS 2.0 客户端注册拦截器的详情，请参考 第 49.5 节 “配置客户端端点”。

在服务器端绑定写器

要在服务器端 绑定 拦截器（即要将它安装到 Apache CXF 运行时中），请执行以下步骤：

1. 将 @Provider 注释添加到 writer interceptor 类，如以下代码片段所示：

```
// Java
package org.jboss.fuse.example;
...
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
@Provider
public class SampleServerWriterInterceptor implements WriterInterceptor {
  ...
}
```

当写器实施加载到 Apache CXF 运行时中时，REST 实施会自动扫描加载的类，以搜索标有 @Provider 注解( 扫描阶段)。

2. 在 XML 中定义 JAX-RS 服务器端点（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”）时，将 writer interceptor 添加到 jaxrs:providers 元素中的供应商列表中。

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

```
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
    ...
    <jaxrs:server id="customerService" address="/customers">
      ...
      <jaxrs:providers>
        <ref bean="interceptorProvider" />
      </jaxrs:providers>
      <bean id="interceptorProvider"
class="org.jboss.fuse.example.SampleServerWriterInterceptor"/>

    </jaxrs:server>

</blueprint>
```

**注意**

此步骤是 Apache CXF 的非标准要求。根据 JAX-RS 标准，严格说，@Provider 注释应全部是绑定拦截器所必需的。但是在实践中，标准方法有些不灵活，当许多库包含在大型项目中时，可能会导致禁止提供程序。

### 61.8. 动态绑定

#### 概述

将容器过滤器和容器拦截器绑定到资源的标准方法是使用 @Provider 注解来注解过滤器和拦截器。这样可确保绑定为 全局 ：即，过滤器和拦截器绑定到服务器端的每个资源类和资源方法。

动态绑定是在服务器端绑定的替代方法，允许您选择并选择将拦截器和过滤器应用到哪些资源方法。要为过滤器和拦截器启用动态绑定，您必须实现自定义 DynamicFeature 接口，如下所述。

#### DynamicFeature 接口

DynamicFeature 接口在 javax.ws.rx.container 软件包中定义，如下所示：

```
// Java
package javax.ws.rs.container;

import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.WriterInterceptor;
```

```
public interface DynamicFeature {
    public void configure(ResourceInfo resourceInfo, FeatureContext context);
}
```

### 实施动态功能

您实现动态功能，如下所示：

1. 如前面所述，实现一个或多个容器过滤器或容器拦截器。但是，请勿 通过 @Provider 注释给他们添加注释（否则，它们将全局绑定，使动态功能有效不相关）。

2. 通过实施 DynamicFeature 类，覆盖 configure 方法来创建您自己的动态功能。

3. 在配置 方法中，您可以使用 resourceInfo 参数来发现哪个资源类以及正在调用此功能的资源方法。您可以使用此信息来决定注册某些过滤器或拦截器的基础。

4. 如果您决定使用当前资源方法注册过滤器或拦截器，您可以通过调用其中一个 context.register 方法来实现。

5. 记住要使用 @Provider 注释给动态功能类添加注释，以确保它在部署的扫描阶段获取。

### 动态功能示例

以下示例演示了如何为带有 @GET 标注的 MyResource 类（或子类）定义注册 LoggingFilter 过滤器的动态功能：

```
// Java
...
import javax.ws.rs.container.DynamicFeature;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.ext.Provider;

@Provider
public class DynamicLoggingFilterFeature implements DynamicFeature {
  @Override
  void configure(ResourceInfo resourceInfo, FeatureContext context) {
    if (MyResource.class.isAssignableFrom(resourceInfo.getResourceClass())
      && resourceInfo.getResourceMethod().isAnnotationPresent(GET.class)) {
```

```
        context.register(new LoggingFilter());
    }
}
```

*动态绑定过程*

**JAX-RS 标准要求为每个资源方法运行一次 DynamicFeature.configure 方法。这意味着，每个资源方法都可能具有由动态功能安装的过滤器或拦截器，但取决于每个情况下是否注册过滤器或拦截器。换句话说，动态功能支持的绑定的粒度是单个资源方法的级别。**

**FeatureContext 接口**

**FeatureContext 接口（允许您在 configure 方法中注册过滤器和拦截器）被定义为 Configurable <> 的子接口，如下所示：**

```
// Java
package javax.ws.rs.core;

public interface FeatureContext extends Configurable<FeatureContext> {
}
```

**Configurable<> 接口定义了各种在单一资源方法上注册过滤器和拦截器的方法，如下所示：**

```
// Java
...
package javax.ws.rs.core;

import java.util.Map;

public interface Configurable<C extends Configurable> {
    public Configuration getConfiguration();
    public C property(String name, Object value);
    public C register(Class<?> componentClass);
    public C register(Class<?> componentClass, int priority);
    public C register(Class<?> componentClass, Class<?>... contracts);
    public C register(Class<?> componentClass, Map<Class<?>, Integer> contracts);
    public C register(Object component);
    public C register(Object component, int priority);
    public C register(Object component, Class<?>... contracts);
    public C register(Object component, Map<Class<?>, Integer> contracts);
}
```

# 第 62 章 APACHE CXF 消息处理阶段

## 入站阶段

*表 62.1 “入站消息处理阶段” 列出入站拦截器链中可用的阶段。*

*表 62.1. 入站消息处理阶段*

| 阶段 | 描述 |
| --- | --- |
| 接收 | 执行特定于传输的处理，如确定二进制附件的 MIME 边界。 |
| PRE_STREAM | 处理传输接收的原始数据流。 |
| USER_STREAM | |
| POST_STREAM | |
| READ | 确定请求是否为 SOAP 或 XML 消息，并构建会添加正确的拦截器。此阶段也会处理 SOAP 消息标头。 |
| PRE_PROTOCOL | 执行协议级别处理。这包括处理 WS114 标头和 SOAP 消息属性的处理。 |
| USER_PROTOCOL | |
| POST_PROTOCOL | |
| UNMARSHAL | 将消息数据解压缩到应用级别代码使用的对象中。 |
| PRE_LOGICAL | 处理 unmarshalled 消息数据。 |
| USER_LOGICAL | |
| POST_LOGICAL | |
| PRE_INVOKE | |
| 调用 | 将消息传递给应用程序代码。在服务器端，服务实施在此阶段被调用。在客户端，响应被移回到应用程序。 |
| POST_INVOKE | 调用出站拦截器链。 |

## 出站阶段

*表 62.2 “入站消息处理阶段” 列出入站拦截器链中可用的阶段。*

*表 62.2. 入站消息处理阶段*

| 阶段 | 描述 |
| --- | --- |
| 设置 | 执行链中后续阶段所需的任何集合。 |
| PRE_LOGICAL | 对从应用程序级别传递的 unmarshalled 数据执行处理。 |
| USER_LOGICAL | |
| POST_LOGICAL | |
| PREPARE_SEND | 打开连接，以在 wire 上写入消息。 |
| PRE_STREAM | 执行所需的处理，以准备消息进入数据流中。 |
| PRE_PROTOCOL | 开始处理协议特定信息。 |
| 写 | 编写协议消息。 |
| PRE_MARSHAL | marshals 消息。 |
| MARSHAL | |
| POST_MARSHAL | |
| USER_PROTOCOL | 处理协议消息。 |
| POST_PROTOCOL | |
| USER_STREAM | 处理字节级消息。 |
| POST_STREAM | |
| SEND | 发送消息并关闭传输流。 |

***重要***

***出站拦截器链有一组镜像结束阶段，其名称附加了 _ENDING。结束阶段使用拦截器，在数据在线上写入前需要进行一些终端操作。***

# 第 63 章 APACHE CXF PROVIDED INTERCEPTORS

## 63.1. CORE APACHE CXF INTERCEPTORS

### 入站

表 63.1 “内核入站拦截器” 列出添加到所有 Apache CXF 端点的核心入站拦截器。

表 63.1. 内核入站拦截器

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **ServiceInvokerInterceptor** | 调用 | 在服务上调用正确的方法。 |

### 出站

默认情况下，Apache CXF 不会将任何内核拦截器添加到出站拦截器链中。端点出站拦截器链的内容取决于所使用的功能。

## 63.2. FRONT-ENDS

### JAX-WS

表 63.2 “入站 JAX-WS 拦截器” 列出添加到 JAX-WS 端点的入站消息队列中的拦截器。

表 63.2. 入站 JAX-WS 拦截器

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **HolderInInterceptor** | **PRE_INVOKE** | 为消息中的任何 out 或 in/out 参数创建拥有者对象。 |
| **WrapperClassInInterceptor** | **POST_LOGICAL** | 将嵌套的 doc/literal 消息的部分封装到适当的对象数组中。 |
| **LogicalHandlerInInterceptor** | **PRE_PROTOCOL** | 将消息处理传递到端点使用的 JAX-WS 逻辑处理程序。当 JAX-WS 处理程序完成时，消息会随同传递给入站链上的下一个拦截器。 |

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **SOAPHandlerInterceptor** | **PRE_PROTOCOL** | 将消息处理传递到端点使用的 JAX-WS SOAP 处理程序。当 SOAP 处理程序使用消息完成时，消息将与链中的下一个拦截器一起传递。 |

*表 63.3 “出站 JAX-WS 拦截器” 列出添加到 JAX-WS 端点的出站消息队列中的拦截器。*

*表 63.3. 出站 JAX-WS 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **HolderOutInterceptor** | **PRE_LOGICAL** | 从拥有者对象中删除 out 和 in/out 参数的值，并将值添加到消息的参数列表中。 |
| **WebFaultOutInterceptor** | **PRE_PROTOCOL** | 处理出站故障消息。 |
| **WrapperClassOutInterceptor** | **PRE_LOGICAL** | 在添加到消息前，请确保正确嵌套了 doc/literal 消息和 rpc/literal 消息。 |
| **LogicalHandlerOutInterceptor** | **PRE_MARSHAL** | 将消息处理传递到端点使用的 JAX-WS 逻辑处理程序。当 JAX-WS 处理程序完成时，消息将随同传递给出站链上的下一个拦截器。 |
| **SOAPHandlerInterceptor** | **PRE_PROTOCOL** | 将消息处理传递到端点使用的 JAX-WS SOAP 处理程序。当 SOAP 处理程序完成消息处理时，它将一起传递给链中的下一个拦截器。 |
| **MessageSenderInterceptor** | **PREPARE_SEND** | 调用回 Destination 对象，使其设置输出流、标头等，以准备传出传输。 |

***JAX-RS***

*表 63.4 “入站 JAX-RS 拦截器” 列出添加到 JAX-RS 端点入站消息队列中的拦截器。*

*表 63.4. 入站 JAX-RS 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **JAXRSInInterceptor** | **PRE_STREAM** | 选择 root 资源类，调用任何配置的 JAX-RS 请求过滤器，并确定要在 root 资源上调用的方法。 |

***重要***

***JAX-RS 端点的入站链直接跳过 ServiceInvokerInInterceptor 拦截器。在 JAXRSInInterceptor 后，不会调用其他拦截器。***

*表 63.5 “出站 JAX-RS 拦截器” 列出添加到 JAX-RS 端点的出站消息队列中的拦截器。*

*表 63.5. 出站 JAX-RS 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **JAXRSOutInterceptor** | **MARSHAL** | 将响应放入正确的传输格式。 |

### *63.3. 消息绑定*

#### *SOAP*

*表 63.6 “入站 SOAP 拦截器” 在使用 SOAP Binding 时，列出添加到端点入站消息链中的拦截器。*

*表 63.6. 入站 SOAP 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **CheckFaultInterceptor** | **POST_PROTOCOL** | 检查消息是否为 fault 消息。如果消息是故障消息，则正常处理将中止并启动故障处理。 |
| **MustUnderstandInterceptor** | **PRE_PROTOCOL** | 进程必须理解标头。 |
| **RPCInInterceptor** | **UNMARSHAL** | Unmarshals rpc/literal 消息。如果消息是裸机的，则会将消息传递给 **BareInInterceptor** 对象，以反序列化消息部分。 |
| **ReadsHeadersInterceptor** | **READ** | 解析 SOAP 标头并将其存储在消息对象中。 |

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **SoapActionInInterceptor** | **READ** | 解析 SOAP 操作标头，并尝试查找该操作的唯一操作。 |
| **SoapHeaderInterceptor** | **UNMARSHAL** | 将映射到操作参数的 SOAP 标头绑定到适当的对象。 |
| **AttachmentInInterceptor** | **接收** | 解析 mime 边界的 mime 标头，找到 **根** 部分，并将输入流重置为它，并将其他部分存储在 **Attachment** 对象集合中。 |
| **DocLiteralInInterceptor** | **UNMARSHAL** | 检查 SOAP 正文中的第一个元素，以确定适当的操作并调用要在数据中读取的数据绑定。 |
| **StaxInInterceptor** | **POST_STREAM** | 从消息创建一个 **XMLStreamReader** 对象。 |
| **URIMappingInterceptor** | **UNMARSHAL** | 处理 HTTP GET 方法的处理。 |
| **SwAInInterceptor** | **PRE_INVOKE** | 为二进制 SOAP 附加创建所需的 MIME 处理程序，并将数据添加到参数列表中。 |

*表 63.7 “出站 SOAP 拦截器” 在使用 SOAP Binding 时，列出添加到端点出站消息链中的拦截器。*

*表 63.7. 出站 SOAP 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **RPCOutInterceptor** | **MARSHAL** | 用于传输的 marshals rpc 样式消息。 |
| **SoapHeaderOutFilterInterceptor** | **PRE_LOGICAL** | 删除所有标记为入站的 SOAP 标头。 |
| **SoapPreProtocolOutInterceptor** | **POST_LOGICAL** | 设置 SOAP 版本和 SOAP 操作标头。 |
| **AttachmentOutInterceptor** | **PRE_STREAM** | 设置附件 marshalers 和处理消息中可能出现的任何附件所需的 mimeullers。 |
| **BareOutInterceptor** | **MARSHAL** | 写入消息部分。 |

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **StaxOutInterceptor** | **PRE_STREAM** | 从消息创建一个 **XMLStreamWriter** 对象。 |
| **WrappedOutInterceptor** | **MARSHAL** | 包装出站消息参数。 |
| **SoapOutInterceptor** | **写** | 在消息中写入 **soap:envelope** 元素和标头块的元素。另外，为剩余的拦截器写入一个空的 **soap:body** 元素来填充。 |
| **SwAOutInterceptor** | **PRE_LOGICAL** | 删除将打包为 SOAP 附加的任何二进制数据，并存储它以便稍后进行处理。 |

***XML***

*表 63.8 “入站 XML 拦截器” 在使用 XML Binding 时，列出添加到端点入站消息队列中的拦截器。*

***表 63.8. 入站 XML 拦截器***

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **AttachmentInInterceptor** | **接收** | 解析 mime 边界的 mime 标头，找到根部分，并将输入流重置为它，然后将其他部分存储在 **Attachment** 对象集合中。 |
| **DocLiteralInInterceptor** | **UNMARSHAL** | 检查消息正文中的第一个元素，以确定适当的操作，然后调用数据中读取的数据绑定。 |
| **StaxInInterceptor** | **POST_STREAM** | 从消息创建一个 **XMLStreamReader** 对象。 |
| **URIMappingInterceptor** | **UNMARSHAL** | 处理 HTTP GET 方法的处理。 |
| **XMLMessageInInterceptor** | **UNMARSHAL** | 取消处理 XML 消息。 |

*表 63.9 “出站 XML 拦截器” 在使用 XML Binding 时，列出添加到端点出站消息链中的拦截器。*

***表 63.9. 出站 XML 拦截器***

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **StaxOutInterceptor** | **PRE_STREAM** | 从消息创建一个 **XMLStreamWriter** 对象。 |
| **WrappedOutInterceptor** | **MARSHAL** | 包装出站消息参数。 |
| **XMLMessageOutInterceptor** | **MARSHAL** | marshals 消息传输。 |

*CORBA*

*表 63.10 “入站 CORBA 拦截器” 在使用 CORBA Binding 时，列出添加到端点入站消息队列中的拦截器。*

*表 63.10. 入站 CORBA 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **CorbaStreamInInterceptor** | **PRE_STREAM** | 反序列化 CORBA 消息。 |
| **BareInInterceptor** | **UNMARSHAL** | 反序列化消息部分。 |

*表 63.11 “出站 CORBA 拦截器” 在使用 CORBA Binding 时，列出添加到端点出站消息链中的拦截器。*

*表 63.11. 出站 CORBA 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **CorbaStreamOutInterceptor** | **PRE_STREAM** | 序列化消息。 |
| **BareOutInterceptor** | **MARSHAL** | 写入消息部分。 |
| **CorbaStreamOutEndingInterceptor** | **USER_STREAM** | 为消息创建可流对象，并将其存储在消息上下文中。 |

## *63.4. 其他功能*

*日志记录*

*表 63.12 “入站日志记录拦截器” 列出添加到端点入站消息队列中来支持日志记录的拦截器。*

*表 63.12. 入站日志记录拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **LoggingInInterceptor** | **接收** | 将原始消息数据写入日志记录系统。 |

*表 63.13 “出站日志拦截器” 列出添加到端点出站消息链中的拦截器来支持日志记录。*

*表 63.13. 出站日志拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **LoggingOutInterceptor** | **PRE_STREAM** | 将出站消息写入日志记录系统。 |

*有关日志记录的详情请参考 第 19 章 Apache CXF Logging。*

### *WS-Addressing*

*表 63.14 “入站 WS-地址拦截器” 使用 WS-Addressing 时，列出添加到端点入站消息链中的拦截器。*

*表 63.14. 入站 WS-地址拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **MAPCodec** | **PRE_PROTOCOL** | 解码消息寻址属性。 |

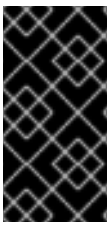
*表 63.15 “出站 WS-地址拦截器” 在使用 WS-Addressing 时，列出添加到端点出站消息链中的拦截器。*

*表 63.15. 出站 WS-地址拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **MAPAggregator** | **PRE_LOGICAL** | 聚合消息寻址属性。 |
| **MAPCodec** | **PRE_PROTOCOL** | 对消息寻址属性进行编码。 |

*有关 WS-寻址的更多信息，请参阅 第 20 章 部署 WS-Addressing。*

### WS-RM



*重要*

*WS-RM 依赖于 WS-Addressing，因此所有 WS-Addressing 拦截器也会添加到拦截器链中。*

*表 63.16 “入站 WS-RM 拦截器” 在使用 WS-RM 时，列出添加到端点入站消息链中的拦截器。*

*表 63.16. 入站 WS-RM 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **RMInInterceptor** | **PRE_LOGICAL** | 处理消息部分的聚合和确认消息。 |
| **RMSoapInterceptor** | **PRE_PROTOCOL** | 从消息对 WS-RM 属性进行编码和解码。 |

*表 63.17 “出站 WS-RM 拦截器” 在使用 WS-RM 时，列出添加到端点出站消息链中的拦截器。*

*表 63.17. 出站 WS-RM 拦截器*

| 类 | 阶段 | 描述 |
| --- | --- | --- |
| **RMOutInterceptor** | **PRE_LOGICAL** | 处理消息的块和块的传输。另外，还处理确认和重新发送请求的处理。 |
| **RMSoapInterceptor** | **PRE_PROTOCOL** | 从消息对 WS-RM 属性进行编码和解码。 |

*有关 WS-RM 的详情，请参考 第 21 章 启用可靠的消息传递。*

# 第 64 章 拦截器供应商

## 概述

拦截器供应商是 Apache CXF 运行时中的对象，它们附加了拦截器链。它们都实施 org.apache.cxf.interceptor.InterceptorProvider 接口。开发人员可以将自己的拦截器附加到任何拦截器供应商。

## 供应商列表

以下对象是拦截器供应商：

- AddressingPolicyInterceptorProvider
- ClientFactoryBean
- ClientImpl
- ClientProxyFactoryBean
- CorbaBinding
- CXFBusImpl
- org.apache.cxf.jaxws.EndpointImpl
- org.apache.cxf.endpoint.EndpointImpl
- ExtensionManagerBus
- JAXRSClientFactoryBean

- *JAXRSServerFactoryBean*
- *JAXRSServiceImpl*
- *JaxWsClientEndpointImpl*
- *JaxWsClientFactoryBean*
- *JaxWsEndpointImpl*
- *JaxWsProxyFactoryBean*
- *JaxWsServerFactoryBean*
- *JaxwsServiceBuilder*
- *MTOMPolicyInterceptorProvider*
- *NoOpPolicyInterceptorProvider*
- *ObjectBinding*
- *RMPolicyInterceptorProvider*
- *ServerFactoryBean*
- *ServiceImpl*

- *SimpleServiceBuilder*
- *SoapBinding*
- *WrappedEndpoint*
- *WrappedService*
- *XMLBinding*

# 部分 VIII. APACHE CXF 功能

本指南论述了如何启用 Apache CXF 的各种高级功能。

# 第 65 章 BEAN VALIDATION

**摘要**

Bean 验证是一种 Java 标准，允许您将 Java 注解添加到服务类或接口来定义运行时限制。Apache CXF 使用拦截器将此功能与 Web 服务方法调用集成。

## 65.1. 简介

### 概述

Bean Validation 1.1 (JSR-349)- 是原始 Bean Validation 1.0 (JSR-303)标准支持的演进，您能使用 Java 注释声明可在运行时检查的约束。您可以使用注解来定义 Java 代码以下部分的约束：

- bean 类中的字段。
- 方法和构造器参数.
- 方法返回值。

### 注解的类示例

以下示例显示了带有一些标准 bean 验证约束的 Java 类：

```
// Java
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Max;
import javax.validation.Valid;
...
public class Person {
    @NotNull private String firstName;
    @NotNull private String lastName;
    @Valid @NotNull private Person boss;

    public @NotNull String saveItem( @Valid @NotNull Person person, @Max( 23 ) BigDecimal age )
{
        // ...
    }
}
```

**Bean 验证或模式验证？**

在某些情况下，bean 验证和模式验证非常相似。使用 XML 模式配置端点是建立良好的方法，可在 Web 服务端点上在运行时验证消息。XML 模式可以检查与传入和传出消息上的 bean 验证相同的限制。但是，因为以下一个或多个原因，bean 验证有时可能是一个有用的替代方案：

- Bean 验证可让您独立于 XML 模式定义约束（例如，在代码优先服务开发时很有用）。
- 如果您的当前 XML 模式太 lax，您可以使用 bean 验证来定义严格的限制。
- Bean 验证可让您定义自定义约束，这可能无法使用 XML 模式语言定义。

**依赖项**

Bean Validation 1.1 (nouveau-349)标准仅定义 API，而非实施。因此，依赖项必须在两个部分中提供：

- 核心依赖项- 提供 bean 验证 1.1 API、Java 统一表达式语言 API 和实现。
- Hibernate Validator 依赖项- 提供 bean 验证 1.1 的实现。

**核心依赖项**

要使用 bean 验证，您必须在项目的 Maven pom.xml 文件中添加以下核心依赖项：

```
<dependency>
   <groupId>javax.validation</groupId>
   <artifactId>validation-api</artifactId>
   <version>1.1.0.Final</version>
</dependency>
<dependency>
   <groupId>javax.el</groupId>
   <artifactId>javax.el-api</artifactId>
   <!-- use 3.0-b02 version for Java 6 -->
   <version>3.0.0</version>
</dependency>
<dependency>
   <groupId>org.glassfish</groupId>
   <artifactId>javax.el</artifactId>
```

```
    <!-- use 3.0-b01 version for Java 6 -->
    <version>3.0.0</version>
</dependency>
```

**注意**

**javax.el/javax.el-api 和 org.glassfish/javax.el 依赖项提供了 Java 统一表达式语言的 API 和实现。此表达式语言由 bean 验证在内部使用，但在应用编程级别并不重要。**

**Hibernate Validator 依赖项**

**要使用 bean 验证的 Hibernate Validator 实现，您必须将以下额外依赖项添加到项目的 Maven pom.xml 文件中：**

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.3.Final</version>
</dependency>
```

**解决 OSGi 环境中的验证提供程序**

**解析验证提供程序的默认机制涉及扫描 classpath 来查找供应商资源。但是，如果是 OSGi (Apache Karaf)环境，这种机制 不起作用，因为验证提供程序（如 Hibernate 验证器）被打包在单独的捆绑包中，因此不能在应用程序类路径中自动可用。在 OSGi 的上下文中，Hibernate 验证器需要与您的应用程序捆绑包有线，而 OSGi 需要一些帮助才能成功做到这一点。**

**在 OSGi 中明确配置验证供应商**

**在 OSGi 的上下文中，您需要明确配置验证提供程序，而不依赖于自动发现。例如，如果您使用通用验证功能（请参阅 "Bean 验证功能"一节）启用 bean 验证，则必须使用验证供应商配置它，如下所示：**

```
<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

**其中 HibernateValidationProviderResolver 是一个自定义类，用于打包 Hibernate 验证提供程序。**

**HibernateValidationProviderResolver 类示例**

以下代码示例演示了如何定义可解析 Hibernate 验证器的自定义 **HibernateValidationProviderResolver** ：

```
// Java
package org.example;

import static java.util.Collections.singletonList;
import org.hibernate.validator.HibernateValidator;
import javax.validation.ValidationProviderResolver;
import java.util.List;

/**
 * OSGi-friendly implementation of {@code javax.validation.ValidationProviderResolver} returning
 * {@code org.hibernate.validator.HibernateValidator} instance.
 *
 */
public class HibernateValidationProviderResolver implements ValidationProviderResolver {

    @Override
    public List getValidationProviders() {
        return singletonList(new HibernateValidator());
    }
}
```

当您在 Maven 构建系统中构建前面的类（配置为使用 Maven 捆绑包插件）时，应用程序将在部署时连接到 Hibernate 验证器捆绑包（假设您已将 Hibernate 验证器捆绑包部署到 OSGi 容器）。

## 65.2. 使用 BEAN 验证开发服务

### 65.2.1. 注解服务 Bean

**概述**

使用 bean 验证开发服务的第一步是将相关的验证注解应用到代表您的服务的 Java 类或接口。验证注解可让您将约束应用到方法参数、返回值和类字段，然后在运行时检查它们，每次调用服务时。

**验证简单的输入参数**

要验证服务方法的参数（其中参数是简单的 Java 类型），您可以应用来自 bean 验证 API 的任何约束注解(javax.validation.constraints 软件包)。例如，以下代码示例测试 nullness (@NotNull 注释)，id 字符串是否与 \\d+ 正则表达式匹配(@Pattern 注释)，以及 名称 字符串的长度是范围 1 到 50 ：

```
import javax.validation.constraints.NotNull;
```

```
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
...
@POST
@Path("/books")
public Response addBook(
        @NotNull @Pattern(regexp = "\\d+") @FormParam("id") String id,
        @NotNull @Size(min = 1, max = 50) @FormParam("name") String name) {
    // do some work
    return Response.created().build();
}
```

### 验证复杂输入参数

要验证复杂的输入参数（对象实例），将 @Valid 注释应用到参数，如下例所示：

```
import javax.validation.Valid;
...
@POST
@Path("/books")
public Response addBook( @Valid Book book ) {
    // do some work
    return Response.created().build();
}
```

@Valid 注释没有指定任何约束。当您使用 @Valid 为 Book 参数标注时，您将有效地告知验证引擎在 Book 类的定义内查找验证约束。在本例中，Book 类定义了其 id 和 name 字段的验证限制，如下所示：

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
...
public class Book {
    @NotNull @Pattern(regexp = "\\d+") private String id;
    @NotNull @Size(min = 1, max = 50) private String name;

    // ...
}
```

### 验证返回值（非响应）

要将验证应用到常规方法返回值(non-Response)，请在方法签名前添加注解。例如，要测试 nullness 的返回值(@NotNull 注释)，并且以递归方式测试验证约束(@Valid 注释)，请注释 getBook 方法，如下所示：

```
import javax.validation.constraints.NotNull;
import javax.validation.Valid;
...
```

```
@GET
@Path("/books/{bookId}")
@Override
@NotNull @Valid
public Book getBook(@PathParam("bookId") String id) {
    return new Book( id );
}
```

*验证返回值(Response)*

*要将验证应用到返回 javax.ws.rs.core.Response 对象的方法，您可以使用与非响应案例相同的注解。例如：*

```
import javax.validation.constraints.NotNull;
import javax.validation.Valid;
import javax.ws.rs.core.Response;
...
@GET
@Path("/books/{bookId}")
@Valid @NotNull
public Response getBookResponse(@PathParam("bookId") String id) {
    return Response.ok( new Book( id ) ).build();
}
```

### 65.2.2. 标准注解

**Bean 验证限制**

*表 65.1 “Bean 验证的标准注解”* 显示 **Bean Validation** 规范中定义的标准注解，可用于定义字段和方法返回值和参数的限制（任何标准注解都可在类级别上应用）。

*表 65.1. Bean 验证的标准注解*

| 注解 | 适用于 | 描述 |
|---|---|---|
| **@AssertFalse** | 布尔值,**boolean** | 检查被注释的元素是否为 **false**。 |
| **@AssertTrue** | 布尔值,**boolean** | 检查注解的元素是否为 **true**。 |
| **@DecimalMax (value=, inclusive=)** | **bigdecimal**,**BigInteger**,**CharSequence**, byte ,**byte**,**short**,**int**, int,**long** 和 primitive 类型打包程序 | 当 **inclusive=false** 时，检查注解的值是否小于指定的最大值。否则，检查值是否小于或等于指定的最大值。**value** 参数指定 bigDecimal 字符串 **格式** 的最大值。 |

| 注解 | 适用于 | 描述 |
| --- | --- | --- |
| **@DecimalMin (value=, inclusive=)** | **bigdecimal**,**BigInteger**,**CharSequence**, byte ,**byte**,**short**,**int**, int,**long** 和 primitive 类型打包程序 | 当 **inclusive=false** 时，检查注解的值是否大于指定最小值。否则，检查值是否大于或等于指定最小值。**value** 参数指定 **BigDecimal** 字符串格式的最小值。 |
| **@Digits(integer=, fraction=)** | **bigdecimal**,**BigInteger**,**CharSequence**, byte ,**byte**,**short**,**int**, int,**long** 和 primitive 类型打包程序 | 检查注解的值是否为数字，最多为 **整数** 数字和 **分数** 分数。 |
| **@Future** | **java.util.Date**, **java.util.Calendar** | 检查注解的日期是否在以后。 |
| **@Max(value=)** | **bigdecimal**,**BigInteger**,**CharSequence**, byte ,**byte**,**short**,**int**, int,**long** 和 primitive 类型打包程序 | 检查注解的值是否小于或等于指定的最大值。 |
| **@Min(value=)** | **bigdecimal**,**BigInteger**,**CharSequence**, byte ,**byte**,**short**,**int**, int,**long** 和 primitive 类型打包程序 | 检查注解的值是否大于或等于指定最小值。 |
| **@NotNull** | 任何类型 | 检查注解的值是否不是 **null**。 |
| **@Null** | 任何类型 | 检查注解的值是否为 **null**。 |
| **@Past** | **java.util.Date**, **java.util.Calendar** | 检查注解的日期是过去的。 |
| **@Pattern(regex=, flag=)** | **CharSequence** | 检查注解的字符串是否与考虑给定标志 **匹配的正则表达式** 匹配。 |
| **@Size(min=, max=)** | **CharSequence**,**Collection**,**Map** 和 array | 检查注解的集合、映射或数组的大小是否在 **min** 和 **max** （包含）之间。 |
| **@Valid** | 任何非原始类型 | 在注释的对象上递归执行验证。如果对象是集合或数组，则元素会递归验证。如果对象是一个映射，则值元素会递归验证。 |

### *65.2.3. 自定义注解*

#### *在 Hibernate 中定义自定义限制*

*可以使用 bean 验证 API 定义您自己的自定义约束注解。有关如何在 Hibernate 验证器实现中执行此操作的详情，请参考 Hibernate Validator 参考指南 中的 创建自定义限制 章节。*

### 65.3. 配置 BEAN 验证

#### 65.3.1. JAX-WS 配置

##### 概述

本节论述了如何在 JAX-WS 服务端点中启用 bean 验证，该端点在 Blueprint XML 或 Spring XML 中定义。用于执行 bean 验证的拦截器常见于 JAX-WS 端点和 JAX-RS 1.1 端点(JAX-RS 2.0 端点使用不同的拦截器类)。

##### 命名空间

在本节中显示的 XML 示例中，您必须记住将 jaxws 命名空间前缀映射到适当的命名空间，可以是 Blueprint 或 Spring，如下表所示：

| XML 语言 | Namespace |
| --- | --- |
| Blueprint | http://cxf.apache.org/blueprint/jaxws |
| Spring | http://cxf.apache.org/jaxws |

##### Bean 验证功能

在 JAX-WS 端点上启用 bean 验证的最简单方法是将 bean 验证功能 添加到端点。bean 验证功能由以下类实现：

**org.apache.cxf.validation.BeanValidationFeature**

通过将此功能类的实例添加到 JAX-WS 端点（通过 Java API 或通过 jaxws:features 子元素的 jaxws:endpoint 子元素在端点上启用 bean 验证），您可以在端点上启用 bean 验证。这个功能会安装两个拦截器：一个用来验证传入的消息数据的 In interceptor，以及一个验证返回值的 Out interceptor （使用默认配置参数创建拦截器）。

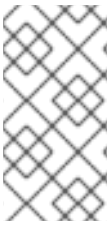
##### 带有 bean 验证功能的 JAX-WS 配置示例

以下 XML 示例演示了如何通过将 commonValidationFeature bean 作为 JAX-WS 功能添加到端点，在 JAX-WS 端点中启用 bean 验证功能：

```
<jaxws:endpoint xmlns:s="http://bookworld.com"
```

```
            serviceName="s:BookWorld"
            endpointName="s:BookWorldPort"
            implementor="#bookWorldValidation"
            address="/bwsoap">
    <jaxws:features>
        <ref bean="commonValidationFeature" />
    </jaxws:features>
</jaxws:endpoint>

<bean id="bookWorldValidation"
class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>

<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

*有关 HibernateValidationProviderResolver 类的示例实现，请参阅 “HibernateValidationProviderResolver 类示例”一节。只需要在 OSGi 环境(Apache Karaf)环境中配置 beanValidationProvider。*



**注意**

*请记住，根据上下文，将 jaxws 前缀映射到 Blueprint 或 Spring 的适当 XML 命名空间。*

**通用 Bean 验证 1.1 拦截器**

*如果要对 bean 验证的配置具有更精细的控制，您可以单独安装拦截器，而不使用 bean 验证功能。要代替 bean 验证功能，您可以配置以下一个或多个拦截器：*

**org.apache.cxf.validation.BeanValidationInInterceptor**

*在 JAX-WS （或 JAX-RS 1.1）端点中安装时，根据验证约束验证资源方法参数。如果验证失败，请引发 javax.validation.ConstraintViolationException 异常。要安装此拦截器，请通过 XML 中的 jaxws:inInterceptors 子元素（或 XML 中的 jaxrs:inInterceptors 子元素）将其添加到端点中。*

**org.apache.cxf.validation.BeanValidationOutInterceptor**

*在 JAX-WS （或 JAX-RS 1.1）端点中安装时，根据验证约束验证响应值。如果验证失败，请引发 javax.validation.ConstraintViolationException 异常。要安装此拦截器，请通过 XML 中的 jaxws:outInterceptors 子元素（或 XML 中的 jaxrs:outInterceptors 子元素）将其添加到端点中。*

*带有 bean 验证拦截器的 JAX-WS 配置示例*

*以下 XML 示例演示了如何通过将相关的 In interceptor bean 和 Out interceptor bean 添加到端点，在 JAX-WS 端点中启用 bean 验证功能：*

```
<jaxws:endpoint xmlns:s="http://bookworld.com"
                serviceName="s:BookWorld"
                endpointName="s:BookWorldPort"
                implementor="#bookWorldValidation"
                address="/bwsoap">
    <jaxws:inInterceptors>
        <ref bean="validationInInterceptor" />
    </jaxws:inInterceptors>

    <jaxws:outInterceptors>
        <ref bean="validationOutInterceptor" />
    </jaxws:outInterceptors>
</jaxws:endpoint>

<bean id="bookWorldValidation"
class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider"/>
</bean>
<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

*有关 HibernateValidationProviderResolver 类的示例实现，请参阅 “HibernateValidationProviderResolver 类示例”一节。只需要在 OSGi 环境(Apache Karaf)环境中配置 beanValidationProvider。*

*配置 BeanValidationProvider*

*org.apache.cxf.validation.BeanValidationProvider 是一个简单的打包程序类，它打包了 bean 验证实施(验证提供程序)。通过覆盖默认 BeanValidationProvider 类，您可以自定义 bean 验证的实现。BeanValidationProvider bean 允许您覆盖以下一个或多个供应商类：*

*javax.validation.ParameterNameProvider*

*为方法和构造器参数提供名称。请注意，需要这个类，因为 Java 反映 API 不为您提供方法参数或构造器参数的名称。*

*javax.validation.spi.ValidationProvider<T>*

为指定类型 T 提供 bean 验证的实现。通过实施自己的 ValidationProvider 类，您可以为您自己的类定义自定义验证规则。这种机制可让您有效地扩展 bean 验证框架。

*javax.validation.ValidationProviderResolver*

实施用于发现 ValidationProvider 类的机制，并返回已发现类的列表。默认解析器在 classpath 上查找 META-INF/services/javax.validation.spi.ValidationProvider 文件，该文件应包含 ValidationProvider 类列表。

*javax.validation.ValidatorFactory*

返回 javax.validation.Validator 实例的工厂。

*org.apache.cxf.validation.ValidationConfiguration*

CXF 打包程序类，可让您覆盖验证供应商层中的更多类。

要自定义 BeanValidationProvider，请将自定义 BeanValidationProvider 实例传递给验证机构以及验证 Out interceptor 的构造器。例如：

```
<bean id="validationProvider" class="org.apache.cxf.validation.BeanValidationProvider" />

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
   <property name="provider" ref="validationProvider" />
</bean>

<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
   <property name="provider" ref="validationProvider" />
</bean>
```

### 65.3.2. JAX-RS 配置

*概述*

本节论述了如何在 JAX-RS 服务端点中启用 bean 验证，该端点在 Blueprint XML 或 Spring XML 中定义。用于执行 bean 验证的拦截器常见于 JAX-WS 端点和 JAX-RS 1.1 端点(JAX-RS 2.0 端点使用不同的拦截器类)。

*命名空间*

在本节中显示的 XML 示例中，您必须记住将 jaxws 命名空间前缀映射到适当的命名空间，可以是 Blueprint 或 Spring，如下表所示：

| XML 语言 | Namespace |
| --- | --- |
| Blueprint | http://cxf.apache.org/blueprint/jaxws |
| Spring | http://cxf.apache.org/jaxws |

**Bean 验证功能**

在 JAX-RS 端点上启用 bean 验证的最简单方法是向端点添加 bean 验证功能。bean 验证功能由以下类实现：

**org.apache.cxf.validation.BeanValidationFeature**

通过将此功能类的实例添加到 JAX-RS 端点（通过 Java API 或通过 XML 中的 jaxrs:server 的 jaxrs:features 子元素），您可以在端点上启用 bean 验证。这个功能会安装两个拦截器：一个用来验证传入的消息数据的 In interceptor，以及一个验证返回值的 Out interceptor （使用默认配置参数创建拦截器）。

**验证例外映射器**

JAX-RS 端点还需要配置 验证异常映射器，它负责将验证例外映射到 HTTP 错误响应。以下类为 JAX-RS 实施验证异常映射：

**org.apache.cxf.jaxrs.validation.ValidationExceptionMapper**

根据 JAX-RS 2.0 规范实施验证异常映射：任何输入参数验证违反情况都会映射到 HTTP 状态代码 400 Bad Request ；以及任何返回值验证违反（或内部验证违反）映射到 HTTP 状态代码 500 Internal Server Error。

**JAX-RS 配置示例**

以下 XML 示例演示了如何通过将 commonValidationFeature bean 添加为 JAX-RS 功能，并通过将 exceptionMapper bean 添加为 JAX-RS 提供程序，在 JAX-RS 端点中启用 bean 验证功能：

```
<jaxrs:server address="/bwrest">
   <jaxrs:serviceBeans>
      <ref bean="bookWorldValidation"/>
   </jaxrs:serviceBeans>
   <jaxrs:providers>
      <ref bean="exceptionMapper"/>
   </jaxrs:providers>
   <jaxrs:features>
```

```
        <ref bean="commonValidationFeature" />
    </jaxrs:features>
</jaxrs:server>

<bean id="bookWorldValidation"
class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>
<beanid="exceptionMapper"class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

*有关 HibernateValidationProviderResolver 类的示例实现，请参阅 “HibernateValidationProviderResolver 类示例”一节。只需要在 OSGi 环境(Apache Karaf)环境中配置 beanValidationProvider。*

**注意**

*请记住，根据上下文，将 jaxrs 前缀映射到 Blueprint 或 Spring 的适当 XML 命名空间。*

### 通用 Bean 验证 1.1 拦截器

*您可以选择安装 bean 验证拦截器来更精细地控制验证实现，而不使用 bean 验证功能。JAX-RS 使用与 JAX-WS 相同的拦截器来实现此目的，请参阅 “通用 Bean 验证 1.1 拦截器”一节*

#### 带有 bean 验证拦截器的 JAX-RS 配置示例

*以下 XML 示例演示了如何通过将相关的 In interceptor bean 和 Out interceptor bean 添加到服务器端点，在 JAX-RS 端点中启用 bean 验证功能：*

```
<jaxrs:server address="/">
    <jaxrs:inInterceptors>
        <ref bean="validationInInterceptor" />
    </jaxrs:inInterceptors>

    <jaxrs:outInterceptors>
        <ref bean="validationOutInterceptor" />
    </jaxrs:outInterceptors>

    <jaxrs:serviceBeans>
```

```
    ...
    </jaxrs:serviceBeans>

    <jaxrs:providers>
        <ref bean="exceptionMapper"/>
    </jaxrs:providers>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

有关 **HibernateValidationProviderResolver** 类的示例实现，请参阅 *"HibernateValidationProviderResolver 类示例"一节*。只需要在 **OSGi** 环境**(Apache Karaf)**环境中配置 **beanValidationProvider**。

**配置 BeanValidationProvider**

您可以将自定义 **BeanValidationProvider** 实例注入验证拦截器，如 *"配置 BeanValidationProvider"一节* 所述。

#### 65.3.3. JAX-RS 2.0 配置

**概述**

与 **JAX-RS 1.1**（与 **JAX-WS** 共享通用验证拦截器）不同，**JAX-RS 2.0** 配置依赖于特定于 **JAX-RS 2.0** 的专用验证拦截器类。

**Bean** 验证功能

对于 **JAX-RS 2.0**，有一个专用的 **bean** 验证功能，它由以下类实施：

**org.apache.cxf.validation.JAXRSBeanValidationFeature**

通过将此功能类的实例添加到 JAX-RS 端点（通过 Java API 或通过 XML 中的 jaxrs:server 的 jaxrs:features 子元素），您可以在 JAX-RS 2.0 服务器端点上启用 bean 验证。这个功能会安装两个拦截器：一个用来验证传入的消息数据的 In interceptor，以及一个验证返回值的 Out interceptor（使用默认配置参数创建拦截器）。

### 验证例外映射器

JAX-RS 2.0 使用与 JAX-RS 1.x 相同的验证异常映射器类：

**org.apache.cxf.jaxrs.validation.ValidationExceptionMapper**

根据 JAX-RS 2.0 规范实施验证异常映射：任何输入参数验证违反情况都会映射到 HTTP 状态代码 400 Bad Request ；以及任何返回值验证违反（或内部验证违反）映射到 HTTP 状态代码 500 Internal Server Error。

### Bean 验证调用器

如果您使用非默认生命周期策略（例如，使用 Spring 生命周期管理）配置 JAX-RS 服务，您还应注册 org.apache.cxf.jaxrs.validation.JAXRSBeanValidationInvoker 实例，使用带有服务端点的端点配置中的 jaxrs:invoker 元素，以确保正确调用 bean 验证。

有关 JAX-RS 服务生命周期管理的详情，请参考 “Spring XML 中的生命周期管理”一节。

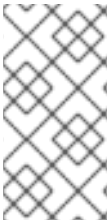
### 带有 bean 验证功能的 JAX-RS 2.0 配置示例

以下 XML 示例演示了如何通过将 jaxrsValidationFeature bean 添加为 JAX-RS 功能，并通过将 exceptionMapper bean 添加为 JAX-RS 提供程序，在 JAX-RS 端点中启用 bean 验证功能：

```
<jaxrs:server address="/">
    <jaxrs:serviceBeans>
    ...
    </jaxrs:serviceBeans>
    <jaxrs:providers>
        <ref bean="exceptionMapper"/>
    </jaxrs:providers>
    <jaxrs:features>
        <ref bean="jaxrsValidationFeature" />
    </jaxrs:features>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>
<bean id="jaxrsValidationFeature" class="org.apache.cxf.validation.JAXRSBeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>
```

```
<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

*有关 HibernateValidationProviderResolver 类的示例实现，请参阅 "HibernateValidationProviderResolver 类示例"一节。只需要在 OSGi 环境(Apache Karaf)环境中配置 beanValidationProvider。*



*注意*

*请记住，根据上下文，将 jaxrs 前缀映射到 Blueprint 或 Spring 的适当 XML 命名空间。*

*通用 Bean 验证 1.1 拦截器*

*如果要对 bean 验证的配置具有更精细的控制，您可以单独安装 JAX-RS 拦截器，而不使用 bean 验证功能。配置以下 JAX-RS 拦截器中的一个或两个：*

*org.apache.cxf.validation.JAXRSBeanValidationInInterceptor*

*在 JAX-RS 2.0 服务器端点中安装时，会根据验证约束验证资源方法参数。如果验证失败，请引发 javax.validation.ConstraintViolationException 异常。要安装此拦截器，请通过 XML 中的 jaxrs:inInterceptors 子元素将其添加到端点。*

*org.apache.cxf.validation.JAXRSBeanValidationOutInterceptor*

*在 JAX-RS 2.0 端点中安装时，根据验证约束验证响应值。如果验证失败，请引发 javax.validation.ConstraintViolationException 异常。要安装此拦截器，请通过 XML 中的 jaxrs:inInterceptors 子元素将其添加到端点。*

*带有 bean 验证拦截器的 JAX-RS 2.0 配置示例*

*以下 XML 示例演示了如何通过显式将相关的 In interceptor bean 和 Out interceptor bean 添加到服务器端点，在 JAX-RS 2.0 端点中启用 bean 验证功能：*

```
<jaxrs:server address="/">
    <jaxrs:inInterceptors>
        <ref bean="validationInInterceptor" />
    </jaxrs:inInterceptors>

    <jaxrs:outInterceptors>
        <ref bean="validationOutInterceptor" />
    </jaxrs:outInterceptors>
```

```
    <jaxrs:serviceBeans>
    ...
    </jaxrs:serviceBeans>

    <jaxrs:providers>
        <ref bean="exceptionMapper"/>
    </jaxrs:providers>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="validationInInterceptor"
class="org.apache.cxf.jaxrs.validation.JAXRSBeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="validationOutInterceptor"
class="org.apache.cxf.jaxrs.validation.JAXRSBeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

*有关 HibernateValidationProviderResolver 类的示例实现，请参阅“HibernateValidationProviderResolver 类示例”一节。只需要在 OSGi 环境(Apache Karaf)环境中配置 beanValidationProvider。*

### 配置 BeanValidationProvider

*您可以将自定义 BeanValidationProvider 实例注入验证拦截器，如“配置 BeanValidationProvider”一节所述。*

### 配置 JAXRSParameterNameProvider

*org.apache.cxf.jaxrs.validation.JAXRSParameterNameProvider 类是 javax.validation.ParameterNameProvider 接口的实现，可用于在 JAX-RS 2.0 端点上下文中提供方法和构造参数的名称。*