Red Hat Fuse 7.11

Apache CXF 开发指南

使用 Apache CXF Web 服务开发应用程序

Last Updated: 2023-09-17

Red Hat Fuse 7.11 Apache CXF 开发指南

使用 Apache CXF Web 服务开发应用程序

法律通告

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。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。详情请查看我们的 CTO Chris Wright 信息。

部分 I. 编写 WSDL 合同

本节介绍如何使用 WSDL 定义 Web 服务接口。

第 1 章 介绍 WSDL 合同

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。文档具有逻辑部分和组件。合同的抽象部分在实施中立的数据类型和消息中定义服务。文档的声明部分定义了实施服务的端点如何与外界交互。

设计服务的建议方法是先在 WSDL 和 XML 架构中定义服务，然后再编写任何代码。手动编辑 WSDL 文档时，必须确保文档有效且正确。要这样做，您必须对 WSDL 有一定的了解。您可以在 W3C 网站 www.w3.org 找到标准。

1.1. WSDL 文档的结构

概述

WSDL 文件最简单的是根 **定义** 元素中包含的元素集合。这些元素描述服务以及如何实施该服务的端点。

WSDL 文档有两个不同的部分：

- 在实施中立地定义该服务 的逻辑部分
- 定义实施该服务的端点如何在网络中公开的 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该服务交换的消息。在 type 元素中，XML Schema 用于定义组成消息的数据的结构。**有多个 消息 元素用于定义服务所使用的消息的结构。portType 元素包含一个或多个 操作 元素，用于定义由服务公开的操作发送的消息。**

Concrete 部分

WSDL 文档的结合部分包含 绑定 和服务 元素。它描述了实施该服务的端点如何连接到外部世界。绑定 元素描述了如何 将消息 元素描述的数据单元映射到一个 Concrete 中，如 SOAP。服务 元素包含一个或多个 端口 元素，它们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 文档中引用的命名空间的简写定义。**
- 类型 - **形成服务所用消息的构建块的 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 -** 调用服务操作期间交换的消息描述。这些元素定义组成服务的操作参数。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 元素的集合。有关定义端口类型的详情，请参考 第 4 章 *定义逻辑接口*。

- **操作 -** 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。有关定义操作的详情请参考 “操作”一节。

- **绑定 -** 端点的数据格式规格。 binding 元素定义抽象消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具体位置。

- **服务 -** 相关端口 元素 的集合。这些元素是组织端点定义的软件仓库。

- **端口 -** 由绑定和物理地址定义的端点。这些元素将所有抽象定义整合在一起，以及定义传输详细信息，并定义服务所公开的物理端点。

1.3. 设计合同

要为服务设计 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. 定义您的服务使用的消息。
3. 定义服务的接口。
4. 定义每个接口使用的消息和线路上的数据的转换之间的绑定。
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

在使用 XML 架构的 WSDL 合同复杂数据类型中描述服务时，将定义为逻辑单元。

2.1. 逻辑数据单元简介

在定义服务时，您必须考虑的第一个操作是如何将用作公开操作参数的数据将被表示。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中定义其数据，这些应用程序可由任意数量的应用程序使用。这涉及两个步骤：

1. 将数据拆分为逻辑单元，这些单元可映射到服务物理实施所使用的数据类型中
2. 将逻辑单元合并到端点之间传递的消息中来执行操作

本章讨论第一步。*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

用于实施服务的接口定义了表示操作参数作为 XML 文档的数据。如果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消息中可组合的 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，以便它们从实施的角度有意义。

可用于定义服务数据单元的类型系统

根据 WSDL 规范，您可以使用您选择的任何类型的系统在 WSDL 合同中定义数据类型。但是，W3C 规格指出 XML 架构是 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 架构作为类型系统

XML 架构用于定义 XML 文档的结构方式。这可以通过定义组成文档的元素来实现。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，或者通过限制现有类型来定义它们。通过组合类型定义和元素定义，您可以创建包含复杂数据的内置 XML 文档。

在 WSDL XML Schema 中使用时，定义包含与服务交互数据的 XML 文档的结构。在定义服务使用的数据单元时，您可以将它们定义为类型，以指定消息部分的结构。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

在实施服务时，您可能需要只创建映射到您要使用的类型的逻辑数据单元。虽然此方法可以正常工作，并密切遵循构建 RPC 风格应用程序的模型，但最好构建一种面向服务的架构。

Web 服务互操作性组织的 WS-I 基本配置集提供了多个用于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。另外，W3C 还提供了以下准则，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础类型。

2.3. 在合同中添加数据单元

概述

根据您选择创建 WSDL 合同的方式，创建新的数据定义需要不同的知识。Apache CXF GUI 工具提供了多个有助于描述使用 XML 架构的数据类型。其他 XML 编辑器提供了不同级别的帮助。无论您选择的编辑器如何，最好了解生成的合同应该是什么样的。

流程

定义 WSDL 合同中使用的数据涉及以下步骤：

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

3. 创建一个 schema 元素，如 例 2.1 “WSDL 合同的架构条目”，作为 type 元素的子级。

targetNamespace 属性指定定义新数据类型的命名空间。最佳实践也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 合同的架构条目

```
<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，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种复杂类型，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 “定义枚举的类型”。

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

2.4. XML 架构简单类型

概述

如果消息部分是简单类型，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是利用简单类型来定义的。

输入简单类型

XML 架构简单类型主要放置在您的合同类型部分使用的 元素 元素中。它们也用于限制元素和扩展 元素 的基本 属性。

使用 xsd 前缀始终输入简单类型。例如，要指定某个元素类型为 int，您可以在其 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：浮点值

- **XSD:double**
- **XSD：布尔值**
- **XSD：字节**
- **xsd:unsignedByte**
- **XSD：整数**
- **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 架构提供了灵活的强大机制，用于从其简单数据类型构建复杂数据结构。您可以通过创建由元素和属性组成的序列来创建数据结构。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，如枚举类型、具有特定范围的值的数据类型，或者通过扩展或限制原语类型需要遵守特定模式的数据类型。

2.5.1. 定义数据结构

概述

在 XML 架构中，作为数据字段集合的数据单元是使用 complexType 元素来定义的。指定复杂类型需要三部分信息：

1. 定义的类型的名称在 complexType 元素的 name 属性中指定。
2. complexType 的第一个子元素描述了当 structure 字段在线路上时的行为。请参阅 “复杂的类型 varieties”一节。
3. 定义结构的每个字段都在作为 complexType 元素的 Getndchildren 的元素中定义。请参阅 “定义结构的部分”一节。

例如，在 XML 架构中作为带有两个元素的复杂类型来定义 例 2.3 “简单结构” 中的结构。

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” 中显示的结构的可能 XML 架构映射，在 例 2.4 “复杂类型” 中定义的结构会生成包含两个元素： name 和 age 的消息。

.

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>
```

复杂的类型 varieties

XML 架构有三种方法来描述复杂类型的字段在以 XML 文档表示并用线路传递时的方式。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	复杂的类型行为
序列	所有复杂类型的字段都可以存在，且它们必须按类型定义中指定的顺序。
all	所有复杂的类型字段都可以存在，但可以按任何顺序排列。
choice	只有结构中的其中一个元素可以放在消息中。

如果使用 选择 元素定义结构，如 例 2.5 “简单复杂的选择类型” 所示，它将生成一个包含 name 元素或 age 元素的消息。

例 2.5. 简单复杂的选择类型

```
<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 元素元素 定义组成结构的数据字段。每个 complexType 元素都应该至少包含一个 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，元素 元素有两个所需的属性：

- name 属性指定 data 字段的名称，它在定义的复杂类型中必须是唯一的。
- type 属性指定字段中存储的数据类型。类型可以是 XML 架构简单类型之一，也可以是在合同中定义的任何指定复杂类型。

除了名称和类型外，元素元素 还有两个常用的可选属性： minOcurrs 和 maxOccurs。 这些属性根据字段在结构中发生的次数进行绑定。默认情况下，每个字段仅在复杂类型中执行一次。使用这些属性，您可以更改字段必须或可以在结构中显示的次数。例如，您可以定义一个字段 previousJobs，它必须至少发生三次，且不超过七次，如 例 2.6 “具有发生限制的简单复杂类型” 所示。

例 2.6. 具有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，使用 minOccurs 来指定 age 字段是可选的，如 例 2.7 “简单的复杂类型，将 minOccurs 设为零” 所示。在这种情况下，可以省略这些 年龄，数据仍将有效。

例 2.7. 简单的复杂类型，将 minOccurs 设为零

```
<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，在下面的代码中的 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 属性 元素。属性 元素只能在所有、序列 或选择元素 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 - 指定用于标识属性的字符串的必需属性。**
- **Type - 指定存储在字段中的数据类型。这个类型可以是 XML 架构简单类型之一。**
- **使用 - 指定具有此属性时是否需要复杂类型的可选属性。有效值 是必需的 或 可选。默认值 是 属性是可选的。**

在 attribute 元素中，您可以指定可选的 default 属性，它允许您指定属性的默认值。

2.5.2. 定义数组

概述

Apache CXF 支持在合同中定义数组。首先定义一个复杂的类型，它 maxOccurs 属性的值大于一。第二个是使用 SOAP 阵列。SOAP 阵列提供添加功能，比如轻松地定义多维数组和传输稀疏填充阵列的功能。

复杂的类型数组

复杂的类型数组是序列复杂类型的特殊情况。您只需使用单个元素定义复杂类型，并为 maxOccurs 属性指定值。例如，要定义一系列调整浮动点号，您需要使用类似 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 阵列

SOAP 阵列通过除 SOAP-ENC:Array 基本类型使用 wsdl:arrayType 元素来定义。这个语法显示在 例 2.10 “使用 wsdl:arrayType 派生的 SOAP 阵列语法” 中。确保 definition 元素声明 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"。

例 2.10. 使用 wsdl:arrayType 派生的 SOAP 阵列语法

```
<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。要指定单一维度数组使用 []; 指定双维数组，使用 [][] 或 [,]。

例如： SOAP Array, SOAPStrings（如 例 2.11 “SOAP 阵列的定义” 所示）定义一个字符串的一个连续数组。wsdl:arrayType 属性指定数组元素的类型 xsd:string，以及维度的数量（ [] 表示一个维

度）。

例 2.11. SOAP 阵列的定义

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

您还可以使用简单元素描述 SOAP Array，如 SOAP 1.1 规范中所述。这个语法显示在 例 2.12 “使用元素派生的 SOAP 阵列的语法” 中。

例 2.12. 使用元素派生的 SOAP 阵列的语法

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

使用这种语法时，元素的 maxOccurs 属性必须始终设为 unbound。

2.5.3. 按扩展定义类型

与大多数主要的编码语言一样，XML Schema 也允许您创建数据类型来继承其某些元素的数据类型。这称为按照扩展来定义类型。例如，您可以创建一个名为 alienInfo 的新类型，它通过添加一个名为 planet 的新元素来扩展 例 2.4 “复杂类型” 中定义的 个人Info 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. 复杂的Content 元素指定新类型将具有多个元素。

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. 从派生新类型的类型（称为 基础 类型）在 extension 元素的 base 属性中指定。

4. 新类型的元素和属性在 extension 元素中定义，它们与用于常规复杂的类型相同。

例如： alienInfo 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 架构允许您通过限制 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类型 SSN，它是仅包含 9 个字符的字符串。通过限制简单类型来定义的新类型，可使用 simpleType 元素来定义。

按限制划分类型的定义需要三个操作：

1. 新类型的名称通过 simpleType 元素的 name 属性指定。

2. 从中派生新类型的简单类型（称为 基础类型 ）在 limits 元素中指定。请参阅 “指定基础类型”一节。

3. 规则（称为 facets ）定义基础类型所施加的限制是作为 限制 元素的子项的子项。请参阅 “定义限制”一节。

指定基础类型

基础类型是受限制来定义新类型的类型。它通过 limit 元素 来指定。限制 元素是 simpleType 元素的唯一子项，并且具有一个指定 基础 类型的属性(base)。基本类型可以是任何 XML 架构简单类型。

例如，通过限制 xsd:int 的值来定义新类型，您可以使用类似 例 2.14 “使用 int 作为基础类型” 中显示的定义。

例 2.14. 使用 int 作为基础类型

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

定义限制

定义对基本类型实施的限制的规则称为 *facets*。facets 是带有一条属性(value)的元素，它定义了如何强制实施 facet。可用的难题及其 有效值 设置取决于基本类型。例如，xsd:string 支持六方面面，其中包括：

- length
- minLength
- maxLength
- pattern

- whitespace
- 枚举

每个 facet 元素都是 限制 元素的子元素。

示例

例 2.15 “SSN 简单类型描述” 显示了一个简单类型 SSN 的示例，它代表了一个社交安全号。生成的 type 是 xxx-xx-xxxx 形式的字符串。<SSN>032-43-9876<SSN> 是此类型的元素的有效值，但 <SSN>032439876</SSN> 不可用。

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. 定义枚举的类型

概述

XML 架构中的枚举类型是根据限制的特殊定义情况。它们通过使用 枚举 面貌（受所有 XML 架构制语支持）加以说明。与大多数现代编程语言中枚举的类型一样，这种类型的变量只能具有其中一个指定值。

在 XML 架构中定义枚举

定义枚举的语法显示在 例 2.16 “枚举的语法” 中。

例 2.16. 枚举的语法

```
<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* 是任意一个或更大值，它指定枚举的每个具体案例的值。枚举的类型可以包含任意数量的问题单值，但由于从简单类型派生出，因此一次只有一个问题单值有效。

示例

例如，如果一个带有由 enumeration widgetSize 定义的元素的 XML 文档，如果它包括了 <widgetSize >big</widgetSize>，则它无效，但如果包含 <widgetSize>big,mungo</widgetSize>。例 2.17 “widgetSize enumeration”

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的成员的 element 相似，它们有三个属性：

- Name - 指定元素名称（在 XML 文档中出现）的必需属性。
- 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 type 具有内个定义，则可以省略此属性。
- nillable - 指定是否可以完全从文档中省略元素。如果 nillable 被设置为 true，则元素可以从使用 schema 生成的任何文档中省略。

元素也可以具有 内行 类型定义。in-line 类型通过 complexType 元素或 simpleType 元素来指定。指定数据类型是否复杂或简单后，您可以使用每种数据类型可用的工具定义任何类型的数据。不建议使用行

类型定义，因为它们没有重复使用。

第 3 章 定义服务使用的逻辑消息

摘要

服务由消息在调用操作时进行交换。在 WSDL 合同中，这些消息通过 消息 元素进行定义。消息由一个或多个利用 部分 元素定义的部分组成。

概述

服务的操作通过指定调用操作时交换的逻辑信息来定义。这些逻辑消息定义通过网络传输的数据作为 XML 文档。它们包含作为方法调用的一部分的所有参数。 使用您的合同中的 message 元素定义逻辑消息。每个逻辑消息由 部分 元素中定义的一个或多个部分组成。

虽然您的消息可以将每个参数列为单独的部分，但建议的做法是只使用一个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。输入消息定义在调用操作时服务接收的所有信息。输出消息定义在操作完成后服务返回的所有数据。故障消息定义服务在发生错误时返回的数据。

另外，每个操作都可以具有任意数量的故障消息。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，以供消费者了解错误。

用于与旧系统集成的消息设计

如果要将现有应用程序定义为服务，您必须确保实施操作方法使用的每个参数都在消息中表示。您还必须确保操作输出消息中包含返回值。

定义您的消息的一种方法是 RPC 样式。在使用 RPC 样式时，您可以在方法的参数列表中为每个参数定义一个消息。每种消息部分均基于合同 类型 元素中定义的类型。您的输入消息包含方法中每个输入参数的部分。您的输出消息会包括每个输出参数的一个部分，以及代表返回值的部分（如果需要）。如果参数既是输入参数和输出参数，它将被列为输入消息和输出消息的一部分。

当启用使用 Tibco 或 CORBA 等传输的旧系统时， RPC 样式消息定义很有用。这些系统围绕了程序和方法设计。因此，使用类似参数列表的消息可方便建模，用于调用的操作。RPC 样式也使得服务与它所公开的应用之间形成一个干净的映射。

SOAP 服务的消息设计

而 RPC 样式对现有系统建模非常有用，而该服务的社区很广泛地使用嵌套文档风格。在嵌套文档风格中，每个消息都有一个部分。该消息的部分引用合同 类型 元素中定义的打包程序元素。wrapper 元素具有以下特征：

- 它是包含一系列元素的复杂类型。更多信息请参阅 第 2.5 节 “定义复杂数据类型”。
- 如果它是一个用于输入信息的打包程序：
  - 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。
- 如果它是一个输出信息的打包程序：
  - 它为每个方法的输出参数和一个元素，每个方法的 inout 参数中有一个元素。
  - 第一个元素表示方法的返回参数。
  - 其名称将通过将响应 附加到 与打包程序关联的操作的名称。

消息命名

合同中的每个消息都必须在其命名空间中具有唯一的名称。建议您使用以下命名规则：

- 消息只应用于一个操作。
- 输入消息名是通过在操作名称中添加 Request 来形成的。

- 输出消息的名称是通过在操作名称中添加 Response 来形成的。
- 错误消息名称应该代表故障的原因。

消息部分

消息部分是逻辑消息的正式数据单元。每一部分使用 part 元素来定义，由 name 属性标识，另一个是 type 属性或指定其数据类型的 element 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
element="*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。

允许消息重复使用部分名称。例如，如果方法有参数 foo，它通过引用或是一个 in/out 传递，它可以是请求消息和响应信息的一部分，如 例 3.1 “重复使用的部分” 所示。

例 3.1. 重复使用的部分

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

示例

例如，假设您有一个存储个人信息的服务器，并提供了一个基于员工的 ID 编号返回员工数据的方法。查找数据的签名方法与 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 风格的 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

逻辑服务接口通过 portType 元素进行定义。

概述

逻辑服务接口通过 WSDL portType 元素进行定义。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输入、输出和错误消息来定义。当生成代码来实施 portType 元素中定义的服务接口时，每个操作都会转换成一个方法，其中包含由合同中指定的参数、输出和错误消息。

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，以包含接口定义，并为它指定唯一名称。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操作元素。请参阅 “操作”一节。
3. 对于每个操作，请指定用于代表操作参数列表、返回类型和异常的消息。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根元素。虽然许多 Web 服务实施直接将 portType 元素映射到生成的实现对象，但逻辑接口定义没有指定由实施的服务提供的确切功能。例如，名为 ticketSystem 的逻辑接口可导致一个实现，可以销售 concert ticket 或挂起问题单的问题。

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

WSDL 文档中的每个 portType 元素必须具有唯一的名称，使用 name 属性来指定，由一系列操作组成，它们在 操作 元素中描述。WSDL 文档可以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，对检查帐户余量的请求和跨小部件的订购都可以定义为操作。

portType 元素中定义的每个操作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

逻辑操作由代表端点之间沟通的逻辑消息的一组元素组成，以执行该操作。表 4.1 “操作消息元素” 中列出了可描述操作的元素。

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供应商的消息。此消息的部分内容对应于操作的输入参数。
output	指定服务提供商在响应请求时发送到客户端端点的消息。此消息的部分内容对应于服务提供商可更改的任何操作参数，例如通过引用传递的值。这包括操作的返回值。
fault	指定用于在端点之间通信错误条件的消息。

需要具有至少一个 输入 或一个 输出元素的操作。操作可以同时具有 输入和输出 元素，但每个元素只能有一个。操作不需要有任何 故障 元素，但在需要时也可具有任意数量的 故障 元素。

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组件数据格式时引用它。名称必须在包含的端口类型内唯一。
message	指定描述正在发送或接收的数据的抽象信息。**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，而 WSDL 会根据包含操作的名称提供默认命名方案。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 输入和输出 元素，则元素名称将分别默认为带有 Request 或 Response 附加到名称的操作的名称。

返回值

因为 operation 元素是操作期间传递的数据的一个抽象定义，所以 WSDL 不会提供为操作指定的返回值。如果方法返回一个值，它将映射到 output 元素，作为该消息的最后一个部分。

示例

例如，您可能有一个与 例 4.1 “personalInfo 查找接口” 中显示的接口类似。

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

本节介绍如何将 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端点可传输和接收的冲突格式。

概述

绑定在服务用于聚合的数据格式的逻辑消息之间提供桥接，该格式端点在物理世界中使用。它们描述了逻辑消息如何映射到端点在线路上使用的有效负载格式。在绑定中，指定了参数顺序、分散数据类型和返回值等详情。例如，消息的部分可以被重新排序，以反映 RPC 调用所需的顺序。根据绑定类型，您还可以识别哪些消息部分（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。端口类型是在两个逻辑服务之间一组交互的抽象定义。绑定（绑定） 是指在物理世界中实例化用于实施逻辑服务的消息的定义。然后，每个绑定都与一组网络详情相关联，该详细信息是结束一个端点的定义，这些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 要求绑定只能表示单一端口类型。例如，如果您具有与两个端口类型的合同，则无法编写将这两个绑定映射到统一数据格式。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，您可以将它映射到两个或更多个绑定中。每个绑定都可以更改消息的部分映射方式，或者为消息指定完全不同的有效负载格式。

WSDL 元素

使用 WSDL 绑定元素在合同中定义绑定。binding 元素由属性组成，如，其名称为提供 PortType 的引用的绑定和类型指定唯一名称。 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* 中所述。

实际映射在 binding 元素的子项中定义。这些元素根据您决定使用的有效负载格式类型而有所不同。以下部分阐述了不同的有效负载格式以及用于指定其映射的元素。

添加到合同中

Apache CXF 提供可为预定义服务生成绑定的命令行工具。

该工具将为您添加正确的元素到您的合同中。但是，建议您了解不同类型的绑定的工作原理。

您还可以使用任何文本编辑器将绑定添加到合同中。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
- pure 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，它不使用任何 SOAP 标头。但是，您可以使用任何文本或 XML 编辑器将 SOAP 标头添加到您的绑定中。

6.1. 添加 SOAP 1.1 BINDING

使用 wsdl2soap

要使用 wsdl2soap 生成 SOAP 1.1 绑定： **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，您将需要下载 Apache CXF 分发版本。

该命令具有以下选项：

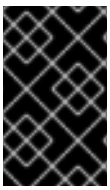
选项	解释
-i *port-type-name*	指定生成绑定的 **portType** 元素。
wsdlurl	包含 **portType** 元素定义的 WSDL 文件的路径和名称。

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*	当 风格为 RPC 时，指定 SOAP 正文命名空间。
-style (document/rpc)	指定 SOAP 绑定中使用的编码风格（document 或 RPC）。默认为文档。
-use (literal/encoded)	指定在 SOAP 绑定中使用的绑定使用（编码或字面处理）。默认为字面上的。
-v	显示工具的版本号。
-verbose	在代码生成过程中显示注释。
-quiet	在代码生成过程中禁止评论。

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 绑定。

示例

如果您的系统有一个接口，它会使用订购并提供单一操作来处理与 例 6.1 “排序系统接口” 中显示的 WSDL 片段中定义的订购。

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 orderWidgets的 SOAP 1.1 Binding” 中。**

例 6.2. 用于 orderWidgets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 document/literal 消息样式发送。

6.2. 在 SOAP 1.1 绑定中添加 SOAP 标头

概述

SOAP 标头通过在默认的 SOAP 1.1 绑定中添加 soap:header 元素来定义。soap:header 元素是 输入、输出和错误 元素的可选子元素。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 标头只能有一个消息部分，但您可以根据需要插入多个 SOAP 标头。

语法

定义 SOAP 标头的语法显示在 例 6.3 “SOAP 标头语法” 中。soap:header 的 message 属性是插入标头的部分所在消息的限定名称。part 属性是插入到 SOAP 标头中的消息部分的名称。由于 SOAP 标头始终为文档风格，因此必须使用元素定义插入到 SOAP 标头中的 WSDL 消息部分。消息 和 部分 属性共同描述要插入到 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>
```

除了强制 消息和 部分属性, soap:header 也支持 命名空间，使用，以及 encodingStyle 属性。这些属性对于 soap:header 的作用相同，因为它们为 soap:body。

在正文和标头间分割消息

插入到 SOAP 标头的消息部分可以是合同中任何有效的消息部分。它也可以是父消息的一部分，这些消息用作 SOAP 正文。由于您不太可能在同一消息中发送信息两次，因此 SOAP 绑定提供了指定插入到 SOAP 正文中的消息部分的方法。

soap:body 元素具有一个可选的属性(part)，它取一个以空格分隔的部分名称列表。当定义各个部分时，只有列出的消息部分才会插入到 SOAP 正文中。然后您可以将剩余的部分插入 SOAP 标头。

注意

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

示例

例 6.4 “带有 SOAP 头的 SOAP 1.1 绑定” 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，放置在请求的 SOAP 标头

中。SOAP 标头定义为来自 widgetKey 消息的 keyVal 部分。在这种情况下，您需要把 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. 带有 SOAP 头的 SOAP 1.1 绑定

```
<?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 “带有 SOAP 头的 SOAP 1.1 绑定”，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 信息

摘要

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

7.1. 将 SOAP 1.2 绑定添加到 WSDL 文档

使用 wsdl2soap

注意

要使用 wsdl2soap，您将需要下载 Apache CXF 分发版本。

要使用 wsdl2soap 生成 SOAP 1.2 绑定，请使用以下命令： **wsdl2soap-i***port-type-name*-b*binding-name*- SOAp12-d*output-directory*-n*soap-body-namespace*-style(document/rpc)-use(literal/encoded)-quiet*wdlurl* 工具有以下所需参数：

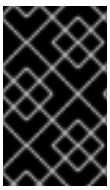
选项	解释
-i *port-type-name*	指定生成绑定的 **portType** 元素。
-soap12	指定生成的绑定使用 SOAP 1.2。
wsdlurl	包含 **portType** 元素定义的 WSDL 文件的路径和名称。

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*	当 风格为 RPC 时，指定 SOAP 正文命名空间。
-style (document/rpc)	指定 SOAP 绑定中使用的编码风格（document 或 RPC）。默认为文档。
-use (literal/encoded)	指定在 SOAP 绑定中使用的绑定使用（编码或字面处理）。默认为字面上的。
-v	显示工具的版本号。
-verbose	在代码生成过程中显示注释。
-quiet	在代码生成过程中禁止评论。

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。

示例

如果您的系统有一个接口，它会使用订购并提供单一操作来处理与 例 7.1 “排序系统接口” 中显示的 WSDL 片段中定义的订购。

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 “用于 orderWidgets 的 SOAP 1.2 绑定” 中。**

例 7.2. 用于 orderWidgets 的 SOAP 1.2 绑定

```
<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 document/literal 消息样式发送。

7.2. 在 SOAP 1.2 消息中添加标头

概述

SOAP 邮件标题通过在 SOAP 1.2 信息中添加 soap12:header 元素来定义。soap12:header 元素是绑定的、output 和 fault 元素的可选子项。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 标头只能有一个消息部分，但您可以根据需要插入多个标头。

语法

定义 SOAP 标头的语法显示在 例 7.3 “SOAP 标头语法” 中。

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>
```

表 7.1 “soap12:header Attributes” 描述了 soap12:header 元素的属性。

表 7.1. soap12:header Attributes

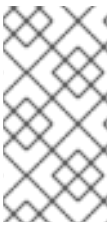
属性	描述
message	指定将要插入到标头中的消息的合格属性。
part	指定插入到 SOAP 标头中的消息部分的名称所需的属性。
使用	指定消息部分是否使用编码规则进行编码。如果设置为 **编码** 消息部分，则使用由 **encodingStyle** 属性的值指定的编码规则进行编码。如果设置为 **字面**，则消息部分由所引用的 schema 类型定义。
encodingStyle	指定用于构建消息的编码规则。
namespace	使用 **use="encoded"** 定义分配给标头元素序列化的命名空间。

在正文和标头间分割消息

插入到 SOAP 标头的消息部分可以是合同中任何有效的消息部分。它也可以是父消息的一部分，这些消息用作 SOAP 正文。由于您不太可能在同一消息中发送信息两次，因此 SOAP 1.2 绑定提供了指定插

入到 SOAP 正文中的消息部分的方法。

soap12:body 元素具有一个可选的属性(part)，它取一个以空格分隔的部分名称列表。当定义各个部分时，只有列出的消息部分插入到 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 会自动填写您的 SOAP 标头。

示例

例 7.4 “使用 SOAP Header 的 SOAP 1.2 绑定” 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，以便每个顺序都放置在请求的标头中的 xsd:base64binary 值。标头定义为来自 widgetKey 消息的 keyVal 部分。在这种情况下，您需要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 Header 的 SOAP 1.2 绑定

```
<?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 Header 的 SOAP 1.2 绑定” 以便标头值是输入和输出信息的一部分，如 例 7.5 “带有 SOAP 标头的 orderWidgets 的 SOAP 1.2 绑定” 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 属性指定该 keyVal 不应插入到正文中。但是，它将插入到标头中。

例 7.5. 带有 SOAP 标头的 orderWidgets 的 SOAP 1.2 绑定

```
<?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 SOAP 使用附件发送二进制数据

摘要

SOAP 附加功能提供了作为 SOAP 消息的一部分发送二进制数据的机制。将 SOAP 与附件一起使用需要将您的 SOAP 消息定义为 MIME 多部件消息。

概述

SOAP 消息通常不包含二进制数据。但是，W3C SOAP 1.1 规范允许使用 MIME 多部件/相关消息在 SOAP 消息中发送二进制数据。使用带有附件的 SOAP 调用这一技术。SOAP 附加在 W3C 的 SOAP 消息及附件 》中定义。

命名空间

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

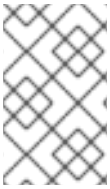
在下面的讨论中，假设此命名空间带有 mime 的前缀。设置此选项的 WSDL 定义 元素中的条目显示在 例 8.1 “合同中的 MIME 命名空间规格” 中。

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、输出和错误 元素的第一个子元素是描述 SOAP 消息正文的 soap:body 元素，表示数据。将 SOAP 与附件一起使用时，soap:body 元素将被 mime:multipartRelated 元素替代。



注意

WSDL 不支持 对故障 消息使用 mime:multipartRelated。

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

于描述消息的各部分。

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

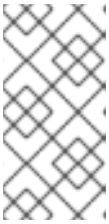
描述 MIME 多部件消息

使用 mime:multipartated 元素（包含 mime:part 元素）介绍了 MIME 多部件消息。要完全描述 MIME 多部件的信息，您必须执行以下操作：

1. 在您将作为 MIME 多部件消息的输入或输出消息内，添加 mime:mulipartRelated 元素作为保护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

如果合同有默认的 SOAP 绑定，您可以将来自默认绑定中对应的消息中的 soap:body 元素复制到 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：内容 属性

属性	描述 +

属性	描述 +
part	指定来自父消息定义中的 WSDL 消息 **部分** 的名称，该消息作为放置在线路上的 MIME 多部分消息的内容。 +
type	此消息中数据的 MIME 类型。MIME 类型定义为类型，使用语法类型/子 *类型*来定义。 + *有多个预定义的 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，在 多用途 Internet 邮件扩展(MIME) 形式中详细介绍：互联网消息正文和 多用途 Internet 邮件扩展(MIME) 第 2 部分：媒体类型。* +

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

示例

例 8.2 “使用带有附件的 SOAP 的合同” 显示 WSDL 片段定义以 JPEG 格式存储 Xrays 的服务。镜像数据 xRay 作为 xsd:base64binary 存储，并打包成 MIME 多部件消息的第二个部分 imageData。作为 SOAP 正文的一部分，输入消息的其余两个部分（ patientName 和 patientNumber 是 MIME 多部件）。

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 替换为附件作为 XML 消息的一部分发送二进制数据的机制。将 MTOM 与 Apache CXF 搭配使用需要把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 消息的一部分发送二进制数据的优化方法。与 SOAP 与附件不同，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 发送二进制数据并不需要在 SOAP 绑定中完全定义 MIME 多部件/Related 消息。但是，这样做需要您执行以下操作：

1. 给 您要发送的数据添加注解。

   您可以注解 WSDL 或实施数据的 Java 类。

2. 启用 运行时的 MTOM 支持。

   这可以通过编程方式或通过配置来完成。

3. 为 作为 附件传输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范畴。

9.2. 使用 MTOM 注解数据类型

概述

在 WSDL 中，当定义用于传输二进制数据的数据类型时，如镜像文件或声音文件，您可以定义类型为 xsd:base64Binary 数据的元素。默认情况下，任何类型为 xsd:base64Binary 的元素都会生成使用 MTOM 进行序列化的 byte[]。但是，代码生成器的默认行为不会充分利用序列化。

要充分利用 MTOM，您必须添加注解到服务的 WSDL 文档或用于实施二进制数据结构的 JAXB 类。将注解添加到 WSDL 文档中会强制代码生成器为二进制数据生成流传输数据处理程序。标注 JAXB 类涉及指定正确内容类型，并可能涉及更改包含二进制数据的字段的类型规格。

先 WSDL

例 9.1 “MTOM 信息” 显示 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。二进制文件字段旨在承载大镜像文件，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件发送，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，并指定该元素预期包含的 MIME 类型。您可以指定一个以逗号分隔的 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。某些 MIME 类型（如镜像的那些类型）定义了映射映射。

注意

MIME 类型由互联网编号分配机构(IANA)维护，在 多用途 Internet 邮件扩展(MIME)中详细介绍：互联网消息正文和 多用途 Internet 邮件扩展(MIME)第 2 部分：介质类型。

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” 显示如何使用 MTOM 从 例 9.1 “MTOM 信息” 修改 xRayType。

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 字节[]。相反，代码生成器会看到 xmime:expectedContentTypes 属性，并为 imageData 字段生成 DataHandler。

注意

您不需要更改 绑定 元素以使用 MTOM。当发送数据时，运行时会进行适当的更改。

Java 第一

如果您正在进行 Java 首个开发，您可以通过执行以下操作使 JAXB 类 MTOM 准备好：

1. 确保包含二进制数据的字段是 DataHandler。

2. 将 @XmlMimeType（） 注解添加到包含您要作为 MTOM 附加数据的字段。

例 9.3 “用于 MTOM 的 JAXB 类” 显示为使用 MTOM 标注的 JAXB 类。

例 9.3. 用于 MTOM 的 JAXB 类

```
@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 运行时不会启用 MTOM 支持。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。您可以以编程方式或使用配置来激活 MTOM 支持。

9.3.1. 使用 JAX-WS API

概述

服务供应商和消费者都必须启用 MTOM 优化功能。JAX-WS API 为每种类型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. 访问您发布的服务的 Endpoint 对象。

   访问 Endpoint 对象的最简单方法是在发布端点时。更多信息请参阅 第 31 章 *发布服务*。

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

   例 9.4. 从端点获取 SOAP Binding

   ```
   // 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，以访问 MTOM 属性。

3. 使用绑定的 setMTOMEnabled（） 方法将绑定的 MTOM enabled 属性设为 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

要让 MTOM 启用 JAX-WS 消费者，您必须执行以下操作：

1. 将使用者的代理转换为 BindingProvider 对象。

有关获取消费者代理的详情，请参考 第 25 章 *在没有 WSDL 合同的情况下开发消费者* 或 第 28 章 *从 WSDL 合同开发消费者*。

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

例 9.6. 从绑定 Provider获取 SOAP 绑定

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

3. 使用绑定的 setMTOMEnabled（） 方法将绑定 MTOM enabled 属性设为 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

如果使用 XML 发布您的服务，比如部署到容器时，您可以在端点的配置文件中启用端点的 MTOM 支持。有关配置端点的详情请参考 第 IV 部分 “配置 Web 服务端点”。

流程

MTOM 属性在您的端点的 jaxws:endpoint 元素内设置。启用 MTOM 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 添加条目 子元素。

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

4. **将 entry 元素的 value 属性设置为 true。**

示例

例 9.8 “配置启用 MTOM” 显示启用 **MTOM** 的端点。

例 9.8. 配置启用 MTOM

```
<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 绑定的一种替代方式，允许服务使用直接 XML 文档来交换数据，而无需增加 SOAP 信封。

XML 绑定命名空间

用于描述 XML 格式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

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. 在您的合同中添加标准 WSDL 绑定 元素以存放 XML 绑定，为绑定指定 一个唯一名称，并指定代表接口绑定的 WSDL portType 元素的名称。
3. 将 xformat: binding 子元素添加到绑定元素中，以标识消息作为纯 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。如需有关 rootNode 属性影响的更多信息，请参阅 “线路上的 XML 消息”一节。
5. 对于绑定接口中定义的每个操作，添加标准的 WSDL 操作 元素来保存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 子元素来代表操作使用的消息。

   这些元素对应于逻辑操作接口定义中定义的消息。

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

注意

如果有任何消息没有相关部分，例如返回无效操作的输出消息，则必须为消息设置 rootNode 属性，以确保 wire 上写入的消息是有效的，但为空，XML 文档。

线路上的 XML 消息

当您指定接口的信息要作为 XML 文档（没有 SOAP 信封）时，您必须小心谨慎，以确保当它们在线路上写入时，您的消息形成有效的 XML 文档。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。

解决了这两个问题的简单方法是，使用 global xformat:binding 元素上的可选 rootNode 属性或单个消息的 xformat:body 元素。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 属性时，在使用 doc 风格消息时，Apache CXF 将消息部分的 root 元素用作根元素，或者使用消息部分名称作为 rpc 样式消息时的 root 元素。

例如，如果 rootNode 属性未设置 例 10.1 “有效的 XML Binding Message” 中定义的消息，则会使用根元素 lineNumber 生成 XML 文档。

例 10.1. 有效的 XML Binding Message

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

对于一个部分的消息，Apache CXF 将始终生成有效的 XML 文档，即使未设置 rootNode 属性。但是，例 10.2 “无效的 XML Binding 消息” 中的信息会生成无效的 XML 文档。

例 10.2. 无效的 XML Binding 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 XML 绑定中指定的 rootNode 属性，Apache CXF 将生成与 例 10.3 “无效的 XML 文档” 中定义的消息类似的 XML 文档。例 10.2 “无效的 XML Binding 消息”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 “带有 rootNode 集的 XML Binding” Apache CXF 所示，则会将元素嵌套到指定的根元素中。在本例中，为整个绑定定义了 rootNode 属性，并指定根元素将命名为 entrants。

例 10.4. 带有 rootNode 集的 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>
```

从输入消息生成的 XML 文档类似于 例 10.5 “使用 rootNode 属性生成的 XML 文档”。请注意，XML 文档现在只有一个根元素。

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

您还可以使用消息中的 **xformat:body** 元素在消息绑定中为每个单独消息设置 **rootNode** 属性，或者覆盖特定消息的全局设置。例如，如果您想要 **例 10.4 “带有 rootNode 集的 XML Binding”** 中定义的输出消息与输入消息不同的 **root** 元素，您可以覆盖绑定的根元素，如 **例 10.6 “使用 xformat:body”** 所示。

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

这部分论述了如何将 Apache CXF 传输添加到 WSDL 文档。

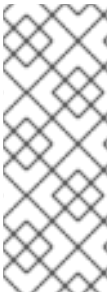
第 11 章 了解 WSDL 中如何定义端点

摘要

端点代表实例化服务。它们通过将绑定和用于公开端点的网络详情进行定义。

概述

端点可以被视为服务的物理清单。它结合了一个绑定，指定服务所使用的逻辑数据的物理表示，以及一组网络详细信息，它们用于定义用于使服务可以被其他端点联系的物理连接详情。



注意

CXF 提供程序是 CXF 消费者的服务器，对应于客户端。如果您在路由中使用 CXF(camel-cxf)组件作为起始端点，则端点既是 Camel 消费者和 CXF 供应商。如果您使用 Camel CXF 组件，作为路由中的结束端点，则端点既是 Camel 制作者和 CXF 消费者。

端点和服务

与绑定只能映射单个接口一样，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清单的票据销售服务。但是，您不能有一个端点，它同时引用了一个 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。端口 元素定义实际的端点。

WSDL 服务 元素具有一个属性，名称，用于指定唯一名称。service 元素用作相关端口元素集合的父元素。WSDL 不规范与 端口 元素相关。您可以以任何方式 将端口 元素关联。

WSDL 端口 元素具有 binding 属性，用于指定端点使用的绑定，是 wsdl:binding 元素的引用。它还包括 name 属性，这是在所有端口之间提供唯一名称的必需属性。port 元素是元素的父元素，用于指定端点使用的实际传输详细信息。以下部分介绍了用来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。

该工具将为您添加正确的元素到您的合同中。但是，建议您了解在定义端点工作中使用的不同传输方式。

您还可以使用任何文本编辑器在合同中添加端点。当您手动编辑合同时，您需要确保合同有效。

支持的传输

端点定义使用为每个传输定义的扩展 Apache CXF 进行构建。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。它为端点之间通信提供了一个标准化、强大且灵活的平台。由于这些因素是大多数 WS-* 规范的假设传输，并且是 RESTful 架构不可或缺的。

12.1. 添加基本 HTTP 端点

备选 HTTP 运行时

Apache CXF 支持以下替代 HTTP 运行时实施：

- 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 “配置网络运行时” 中详细介绍。

Netty HTTP URL

通常，HTTP 端点使用在类路径（Undertow 或 Netty）中包含哪些 HTTP 运行时。但是，如果 classpath 中同时包含 Undertow 运行时和 Netty 运行时，则需要在希望使用 Netty 运行时而明确指定，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，您可以通过指定要以下格式的端点 URL 来选择 Undertow 运行时：

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

有效负载类型

根据您使用的有效负载格式，可通过三种方式指定 HTTP 端点的地址。

- 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

从 Camel 2.16.0 发行版本，Apache Camel CXF Payload 支持开箱即用流缓存。

SOAP 1.1

当您通过 HTTP 发送 SOAP 1.1 消息时，必须使用 SOAP 1.1 地址 元素来指定端点的地址。它有一个属性（位置），用于指定端点的地址为 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 地址 元素来指定端点的地址。它有一个属性（位置），用于指定端点的地址为 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 地址 元素来指定端点的地址。它有一个属性（ 位置 ），用于指定端点的地址为 URL。HTTP 地址 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 Consumer 端点的机制

HTTP 使用者端点可指定多个 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，无论端点是否会请求一个 keep-alive，以及端点如何与代理交互。除了 HTTP 连接属性外，HTTP 消费者端点还可指定它的安全性。

消费者端点可使用两种机制配置：

- **配置**

- WSDL

12.2.2. 使用配置

命名空间

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。通常会使用前缀 http-conf 来引用。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 Bean 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。

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 消费者端点。http-conf:conduit 元素采用单个属性 name，用于指定与端点对应的 WSDL 端口 元素。name 属性的值为 *portQName*'.http-conduit'。例 12.5 “http-conf:conduit Element” 显示 http-conf:conduit 元素，用于在端点的目标命名空间中为端点添加由 WSDL 片段 &lt ;port binding="widgetSOAPBinding" name="widgetSOAPPort > 指定的端点配置。

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。表 12.1 “用于配置 HTTP 消费者端点的元素” 中描述的它们。

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、keep-alive 请求、内容类型等。请参阅 “client 元素”一节。
HTTP-conf:authorization	指定配置端点使用的基本身份验证方法的参数。首选方法是提供 **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	指定用于配置 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 参考或类名称，可以是预先选择或响应 401 HTTP 质询。
http-conf:trustDecider	指定检查 HTTP(S) **URLConnection** 对象的 bean 参考或类名称，以便在传输任何信息前与 HTTPS 服务供应商建立信任。

client 元素

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

表 12.2. HTTP 消费者配置属性

属性	描述
ConnectionTimeout	指定在超时前，使用者尝试建立连接的时间（以毫秒为单位）。默认值为 **30000**。 **0** 指定消费者继续无限期地发送请求。
ReceiveTimeout	指定使用者在超时前等待响应时间，以毫秒为单位。默认值为 **30000**。 **0** 指定使用者会无限期等待。

属性	描述
AutoRedirect	指定消费者是否自动遵循发布的服务器重定向。默认值为 **false**。
MaxRetransmits	指定消费者重新传输请求以满足重定向的最大次数。默认值为 **-1**，它指定允许无限的重新传输。
AllowChunking	指定使用者是否使用块发送请求。默认为 **true**，指定消费者在发送请求时使用块。 如果以下任意一个为 true，则无法使用块： ● **HTTP-conf:basicAuth** Supplier 配置为预先提供凭证。 ● **AutoRedirect** 设置为 **true**。 在这两种情况下，**AllowChunking** 的值会被忽略，并且不允许使用块。
accept	指定消费者准备好处理的介质类型。该值用作 HTTP **Accept** 属性的值。属性的值通过多用途互联网邮件扩展(MIME)类型来指定。
AcceptLanguage	指定接收响应的目的（例如，美国英语）。该值用作 HTTP **AcceptLanguage** 属性的值。 语言标签由国际机构用于标准(ISO)监管，通常通过组合语言代码（由 ISO-639 标准和国家代码确定），由 ISO-3166 标准决定，由连字符分隔。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编号授权机构(IANA)监管。该值用作 HTTP **AcceptEncoding** 属性的值。
ContentType	指定在邮件正文中发送的数据类型。使用多用途互联网邮件扩展(MIME)类型来指定介质类型。该值用作 HTTP **ContentType** 属性的值。默认为 **text/xml**。 对于 Web 服务，应将其设置为 **text/xml**。如果客户端将 HTML 表单数据发送到 CGI 脚本，则应该将其设置为 **application/x-www-form-url** 编码的。如果 HTTP POST 请求绑定到固定有效负载格式（而不是 SOAP），则内容类型通常会设置为 **application/octet-stream**。

属性	描述
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。 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的首选主机（即，用于虚拟服务器映射到同一互联网协议(IP)地址）。
连接	指定在每个请求/响应对话框后是否保持打开或关闭特定连接。有两个有效的值： ● **keep-Alive** - 指定使用者需要在初始请求/响应序列后保持打开的连接。如果服务器尊重，连接将保持打开，直到消费者将其关闭为止。 ● **关闭**（默认）- 指定到服务器的连接在每个请求/响应序列后关闭。
CacheControl	指定由涉及的链中必须遵循的行为的指令，增加从消费者到服务提供商的请求。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。在来自 World Wide Web Consortium(W3C)的 HTTP 规格中，也称为 **user-agent**。某些服务器根据发送请求的客户端进行优化。
请参阅	指定指示消费者在特定服务上发出请求的资源的 URL。该值用作 HTTP 推荐 **属性值**。 当一个请求是浏览器用户的结果时，可通过单击超链接而不是输入 URL 时使用此 HTTP 属性。这可允许服务器根据以前的任务流优化处理，并生成到资源的链接列表，以进行日志记录、优化缓存、过期或错误输入的链接等。但是，它通常用于 Web 服务应用程序。 如果 **AutoRedirect** 属性设为 **true**，并且请求将被重定向，则引用 **器** 属性中指定的任何值都会被覆盖。HTTP 推荐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详情请参考 第 12.6 节 “在拒绝模式中使用 HTTP 传输”。 您必须配置消费者端点，以便使用 WS-Addressing 进行分离端点。

属性	描述
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： • **HTTP**（默认） • **SOCKS**

示例

例 12.6 “HTTP 消费者端点配置” 显示 HTTP 消费者端点的配置，该端点需要在请求之间保持与提供程序打开的连接，每个调用只会重新传输请求一次，并且无法使用块流。

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 的更多信息，请参阅 第 16 章 *conduits*。

12.2.3. 使用 WSDL

命名空间

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。通常会使用前缀 http-conf 来引用。要使

用 HTTP 配置元素，您必须将 例 12.7 “HTTP Consumer WSDL Element 的命名空间” 中显示的行添加到端点的 WSDL 文档 的定义 元素中。

例 12.7. HTTP Consumer WSDL Element 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 元素的子项。它具有与配置文件中使用 的客户端 元素相同的属性。属性在 表 12.2 “HTTP 消费者配置属性” 中描述。

示例

例 12.8 “WSDL 以配置 HTTP 消费者端点” 显示配置 HTTP 消费者端点的 WSDL 片段，以指定它没有与缓存交互。

例 12.8. WSDL 以配置 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：客户端 缓存控制指令” 列出 HTTP 消费者支持的缓存控制指令。

表 12.3. HTTP-conf：客户端 缓存控制指令

指令	行为
no-cache	缓存无法使用特定的响应来满足后续的请求，而无需首先重新使用服务器重新显示该响应。如果使用此值指定特定的响应标头字段，则限制只适用于响应中的那些标头字段。如果没有指定响应标头字段，则限制将应用到整个响应。
no-store	缓存不得存储响应的任意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）。
max-stale	使用者可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以外的秒数，则用户仍然可以接受该响应。如果没有分配值，消费者可以接受任何年龄的过时的响应。
min-fresh	消费者希望对仍有至少指定的秒数而全新的响应。
no-transform	缓存不得修改提供程序和消费者之间的媒体类型或内容位置。
only-if-cached	缓存应只返回当前存储在缓存中的响应，而不是需要重新加载或重新验证的响应。
cache-extension	为其他缓存指令指定额外的扩展。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，以便应用程序不了解扩展指令可遵循标准指令强制的行为。

12.3. 配置服务提供商

12.3.1. HTTP 服务提供商的机制

HTTP 服务提供商端点可指定许多 HTTP 连接属性，包括是否遵守活动请求、与缓存交互的方式，以及允许它与消费者通信时所造成的错误。

可以使用两种机制配置服务供应商端点：

- **配置**

- WSDL

12.3.2. 使用配置

命名空间

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。通常会使用前缀 http-conf 来引用。要使用 HTTP 配置元素，您必须将 例 12.9 “HTTP Provider Configuration Namespace” 中显示的行添加到端点配置文件的 Bean 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。

例 12.9. HTTP Provider Configuration Namespace

```
<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 服务供应商端点。http-conf:destination 元素采用单个属性 name，用于指定与端点对应的 WSDL 端口 元素。name 属性的值采用以下形式的 *portQName*'.http-destination'。例 12.10 “HTTP-conf:destination Element” 显示 http-conf:destination 元素，用于为端点添加由 WSDL 片段 < port binding="widgetSOAPBinding" name="widgetSOAPPort > 指定的端点配置 http://widgets.widgetvendor.net。

例 12.10. HTTP-conf:destination Element

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

```
    ...
  </http-conf:destination>
...
```

http-conf:destination 元素有多个指定配置信息的子元素。表 12.4 “用于配置 HTTP 服务提供商端点的元素” 中描述的它们。

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	指定配置用于处理 HTTP 请求的上下文匹配策略的参数。
http-conf:fixedParameterOrder	指定此目的地处理的 HTTP 请求的参数顺序是否已解决。

server 元素

http-conf:server 元素用于配置服务提供商端点 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	设定连接超时前服务提供程序尝试接收请求的时间长度，以毫秒为单位。默认值为 **30000**。 **0** 指定供应商不会超时。
SuppressClientSendErrors	指定在收到请求时是否抛出异常。默认为 **false** ；遇到错误时会抛出异常。
SuppressClientReceiveErrors	指定在向消费者发送响应时是否遇到错误时是否抛出异常。默认为 **false** ；遇到错误时会抛出异常。
HonorKeepAlive	指定在发送响应后，服务提供商是否遵循连接是否保持打开状态。默认为 **false** ；忽略 keep-alive 请求。

属性	描述
RedirectURL	如果客户端请求的 URL 不再适合所请求的资源，则指定客户端请求应重定向到的 URL。在本例中，如果服务器响应的第一行中没有自动设置状态代码，则状态代码设置为 302，状态描述设置为 Object Moved。该值用作 HTTP RedirectURL 属性的值。
CacheControl	指定由涉及的链中必须遵循的行为的指令，由来自服务提供商的响应增加至消费者。请参阅 第 12.3.4 节 “服务提供商缓存控制指令”。
ContentLocation	设置在响应中发送资源所在的 URL。
ContentType	指定在响应中发送的信息的介质类型。使用多用途互联网邮件扩展(MIME)类型来指定介质类型。该值用作 HTTP ContentType 位置的值。
ContentEncoding	指定应用于由服务提供商发送的信息应用的任何额外内容编码。内容编码标签由互联网编号授权机构(IANA)监管。可能的内容编码值包括 zip、gzip、压缩、deflate 和 identity。这个值被用作 HTTP ContentEncoding 属性的值。 内容编码的主要用途是允许使用一些编码机制（如 zip 或 gzip）压缩文档。Apache CXF 对内容编码没有验证。用户负责确保应用程序级别支持指定的内容编码。
ServerType	指定发送响应的服务器类型。使用 form ***program-name*/*version*** 的值，如 **Apache/1.2.5**。

示例

例 12.11 “HTTP 服务提供商端点配置” 显示 HTTP 服务提供商端点的配置，该端点遵循保留请求并阻止所有通信错误。

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

命名空间

用于配置 HTTP 提供商端点的 WSDL 扩展元素在命名空间 http://cxf.apache.org/transports/http/configuration 中定义。通常会使用前缀 http-conf 来引用。要使用 HTTP 配置元素，您必须将 例 12.12 “HTTP Provider WSDL Element 的 Namespace” 中显示的行添加到端点的 WSDL 文档 的定义 元素中。

例 12.12. HTTP Provider WSDL Element 的 Namespace

```
<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。它具有与配置文件中使用 的服务器 元素相同的属性。属性在 表 12.5 “HTTP 服务提供商配置属性” 中描述。

示例

例 12.13 “WSDL 以配置 HTTP 服务供应商端点” 显示 WSDL 片段，它将配置 HTTP 服务提供商端点，指定它不会与缓存交互。

例 12.13. WSDL 以配置 HTTP 服务供应商端点

```
<service ... >
  <port ... >
    <soap:address ... />
```

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

12.3.4. 服务提供商缓存控制指令

表 12.6 “HTTP-conf：服务器 缓存控制指令” 列出 HTTP 服务提供商支持的缓存控制指令。

表 12.6. HTTP-conf：服务器 缓存控制指令

指令	行为
no-cache	缓存无法使用特定的响应来满足后续的请求，而无需首先重新使用服务器重新显示该响应。如果使用此值指定特定的响应标头字段，则限制只适用于响应中的那些标头字段。如果没有指定响应标头字段，则限制将应用到整个响应。
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，因为响应适合一个用户。如果使用此值指定特定的响应标头字段，则限制只适用于响应中的那些标头字段。如果没有指定响应标头字段，则限制将应用到整个响应。
no-store	缓存不得存储调用它的请求的任何部分或任何部分。
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位置。
must-revalidate	在后续响应中使用该条目前，缓存必须重新验证与响应相关的过期条目。
proxy-revalidate	与 must-revalidate 操作相同，但只能对共享缓存强制，并被私有未共享缓存忽略。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没有超过指定秒数的响应。
s-max-age	与 max-age 相似，但只能对共享缓存强制实施，并且被私有未共享缓存忽略。由 s-max-age 指定的年龄会覆盖由 max-age 指定的时间。当使用这个指令时，还必须使用 proxy-revalidate 指令。
cache-extension	为其他缓存指令指定额外的扩展。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，以便应用程序不了解扩展指令可遵循标准指令强制的行为。

12.4. 配置 UNDERTOW 运行时

概述

Undertow 运行时供 HTTP 服务提供商和 HTTP 用户使用分离端点。可以配置运行时的线程池，您可以通过 Undertow 运行时设置 HTTP 服务提供商的多个安全设置。

Maven 依赖项

如果将 Apache Maven 用作构建系统，您可以通过在项目的 pom.xml 文件中包含下列依赖项，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>
```

命名空间

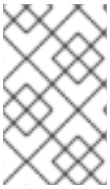
用于配置 Undertow 运行时的元素在命名空间 http://cxf.apache.org/transports/http-undertow/configuration 中定义。若要使用 Undertow 配置元素，您必须将 例 12.14 “Undertow 运行时配置命名空间” 中显示的行添加到端点配置文件的 Bean 元素中。在本例中，命名空间被分配了前缀 httpu。另外，您必须将配置元素的命名空间添加到 xsi:schemaLocation 属性。

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 元素是用来配置供应用使用的 Undertow 运行时的根元素。它具有单一必需属性 总线，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**，它是默认 总线实例的名称。

http:engine-factory 元素有三个子项，其中包含用于配置使用 **Undertow** 运行时工厂实例化的 **HTTP** 端口的信息。子对象在 表 12.7 “配置Undertow Runtime onnectionFactoryy 的元素” 中描述。

表 12.7. 配置Undertow Runtime onnectionFactoryy 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利用属性集合。它有一个属性 **id**，它指定可以引用该属性集的唯一标识符。
httpu:identifiedThreadingParameters	指定控制 Undertow 实例的线程池的可重复利用属性集合。它有一个属性 **id**，它指定可以引用该属性集的唯一标识符。 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**)，它指定具有嵌入式 **undertow** 和端口 的全局 **IP** 地址，用于指定由 **Undertow** 实例管理的端口数量。

注意

您可以为 **端口** 属性指定一个 **0** 值。**httpu:engine** 元素中指定的任何线程属性设为 **0**，用作未明确配置的所有 **Undertow** 侦听器的配置。

每个 **httpu:engine** 元素可以包含两个子项：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，也可以提供对父 **httpu:engine-factory** 元素中定义的一组配置属性的引用。

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

表 12.8. 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 **identifiedTLSServerParameters** 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 **指定的ThreadingParameters** 元素定义的一组属性。**id** 属性提供所 **识别的 ThreadingParameters** 元素的 id。

配置线程池

您可以通过以下任一方式配置 Undertow 实例的线程池的大小：

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

threadingParameters 有两个属性来指定线程池的大小。属性在 表 12.9 "配置 Undertow 线程池的属性" 中描述。

注意

httpu:identifiedThreadingParameters 元素具有一个子 线程参数 元素。

表 12.9. 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。如果未指定，则会选择 dafult 值。默认值为每个 CPU 内核的一个 I/O 线程。
minThreads	指定用于处理请求的最小线程数量。

属性	描述
最大线程数	指定用于处理请求的最大线程数量。

示例

例 12.15 “配置 Undertow 实例” 显示配置片段，在端口号 9001 上配置 Undertow 实例。

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，该处理程序将设置并发连接请求数和可通过 Undertow 服务器实例处理的最大队列大小。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. 用于配置请求限制处理程序的属性

属性	描述
maximumConcurrentRequests	指定由 Undertow 实例处理的最大并发请求数量。如果请求数超过这个限制，则请求会被放入。
queueSize	指定可通过 Undertow 实例处理的请求总数。如果请求数超过这个限制，则请求将被拒绝。

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. 配置网络运行时

概述

Netty 运行时供 HTTP 服务供应商和 HTTP 消费者使用分离端点。可以配置运行时的线程池，您还可以通过 Netty 运行时设置 HTTP 服务提供商的多个安全设置。

Maven 依赖项

如果将 Apache Maven 用作构建系统，您可以通过在项目的 pom.xml 文件中包括以下依赖项，将 Netty 运行时（用于定义 Web 服务端点）的服务器端实施添加到项目中：

```
<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 文件中包括以下依赖项，将 Netty 运行时（用于定义 Web 服务客户端）的客户端实施添加到项目中：

-

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

命名空间

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。通常会使用前缀 httpn 来引用。要使用 Netty 配置元素，您必须将 例 12.17 “Netty Runtime Configuration Namespace” 中显示的行添加到端点配置文件的 Bean 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。

例 12.17. Netty Runtime Configuration Namespace

```
<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 元素是用来配置应用程序使用的 Netty 运行时的根元素。它只有一个必需属性 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，它是默认 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 runtime factory 实例化的 HTTP 端口的信息。子对象在 表 12.11 “配置 Netty Runtime Factory 的元素” 中描述。

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。

元素	描述
httpn:identifiedTLSServerParameters	指定用于保护 HTTP 服务提供商的可重复利用属性集合。它有一个属性 **id**，它指定可以引用该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**，它指定可以引用该属性集的唯一标识符。 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 “配置网络运行时实例的属性” 显示 httpn:engine 元素支持的属性。

表 12.12. 配置网络运行时实例的属性

属性	描述
port	指定 Netty HTTP 服务器实例使用的端口。您可以为端口属性指定一个 **0** 值。引擎元素中指定的线程属性设为 0，其 port 属性设置为 0，所有未明确配置的 Netty 侦听器的配置。
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。
readIdleTime	指定 Netty 连接的最大读取闲置时间。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 元素具有一个用于配置安全属性的子元素，一个用于配置 Netty 实例的线程池。对于每种配置，您可以直接提供配置信息，也可以提供对父 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 “配置网络运行时实例的元素” 中。

表 12.13. 配置网络运行时实例的元素

元素	描述
httpn:tlsServerParameters	指定一组用于配置用于特定 Netty 实例的安全的属性。
httpn:tlsServerParametersRef	指的是由 **identifiedTLSServerParameters** 元素定义的一组安全属性。**id** 属性提供 **identifiedTLSServerParameters** 元素的 id。
httpn:threadingParameters	指定特定 Netty 实例使用的线程池大小。请参阅 “配置线程池”一节。
httpn:threadingParametersRef	指的是由 **指定的ThreadingParameters** 元素定义的一组属性。**id** 属性提供所 **识别的 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	指定设置 **ReuseAddress** TCP 套接字选项的布尔值。默认为 **false**。

配置线程池

您可以通过以下任一方式配置 Netty 实例的线程池的大小：

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

threadingParameters 元素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 元素具有一个子 线程参数 元素。

表 12.14. 配置网络线程池的属性

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

示例

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

例 12.18. 配置网络实例

```
<?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. 在拒绝模式中使用 HTTP 传输

概述

在普通的 HTTP 请求/响应场景中，请求和响应使用相同的 HTTP 连接发送。服务提供商处理请求并响应包含适当 HTTP 状态代码和响应内容的响应。如果是成功请求，HTTP 状态代码将设为 200。

在某些情况下，比如使用 WS-RM，或者请求执行延长的时间，因此分离请求和响应消息是合理的。在这种情况下，服务提供程序通过 HTTP 连接的后端通道将消费者发送 202 Accepted 响应到消费者，后者上接收了请求。然后，它会处理请求，并使用新的解除服务器→client HTTP 连接将响应发回到消费者。使用者运行时收到传入的响应，并在返回到应用代码之前将其与适当的请求相关联。

配置分离交互

以分离模式使用 HTTP 传输需要您执行以下操作：

1. 配置使用者以使用 WS-Addressing。

   请参阅 “配置端点以使用 WS-Addressing”一节。

2. 将使用者配置为使用分离端点。

请参阅 “配置使用者”一节。

3. 配置消费者与之交互的任何服务提供商，以使用 WS-Addressing。

请参阅 “配置端点以使用 WS-Addressing”一节。

配置端点以使用 WS-Addressing

指定使用者和任何服务提供程序使用 WS-Addressing。

您可以通过以下两种方式之一指定端点使用 WS-Addressing：

- 添加 wswa:UsingAddressing 元素到端点的 WSDL 端口 元素，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，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-Addressing 策略取代了 wswa:UsingAddressing WSDL 元素。

配置使用者

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 配置为使用已拒绝的 HTTP 端点” 显示设置 例 12.19 “使用 WSDL 激活 WS-Addressing” 中定义的端点的配置，以使用分离端点。用户现在收到所有响应 http://widgetvendor.net/widgetSellerInbox。

例 12.21. 将 Consumer 配置为使用已拒绝的 HTTP 端点

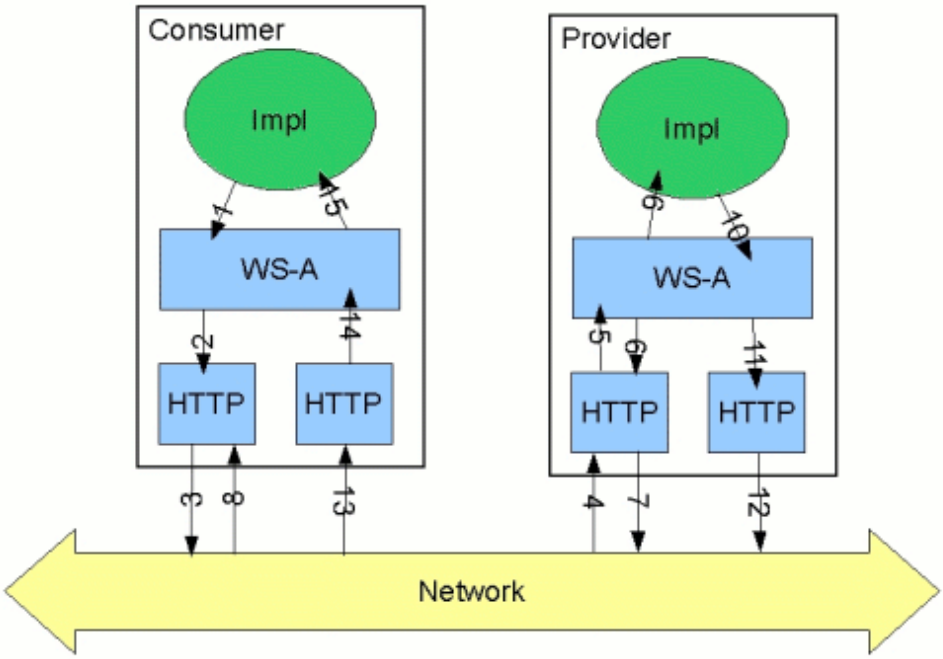
```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                          http://cxf.apache.org/schemas/configuration/http-conf.xsd
                         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 传输会增加对 HTTP 消息处理的额外复杂性。虽然为应用程序中的实施级别代码增加了复杂性是透明的，但了解调试原因可能很重要。

图 12.1 “拒绝 HTTP 传输中的消息流” 显示在分离模式下使用 HTTP 时的信息流。

图 12.1. 拒绝 HTTP 传输中的消息流



请求启动以下进程：

1. 使用者实施调用操作，并且生成请求消息。

2. WS-Addressing 层将 WS-A 标头添加到消息中。

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

3. 邮件将发送到服务供应商。

4. 服务提供商收到消息。

5. 来自消费者的请求消息被分配到提供商的 WS-A 层。

6. 由于 WS-A ReplyTo 标头没有设置为 anonymous，因此提供商会向一个 HTTP 状态代码发回一个消息，确认已接收了请求。

7. HTTP 层使用原始连接的 back-channel 将 202 Accepted 消息发回给消费者。

8. 使用者收到用于发送原始消息的 HTTP 连接后端频道的 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中。

12. HTTP 传输将响应发送到消费者的解除端点。

13. 消费者分离的端点接收来自服务提供商的响应。

14. 响应被分配给消费者的 WS-A 层，它使用 WS-A RelatesTo 标头与正确的请求相关联。

15. 关联的响应将返回到客户端实施，调用调用被取消阻塞。

第 13 章 使用 JMS 的 SOAP

摘要

Apache CXF 实施 W3C 标准 SOAP/JMS 传输。这个标准旨在提供更加可靠的 SOAP/HTTP 服务的替代方案。使用此传输的 Apache CXF 应用程序应该能够与实施 SOAP/JMS 标准的应用程序交互。传输在端点的 WSDL 中直接配置。

注意 ：在 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 协议的 SOAP 由 World Wide 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，并与也兼容的框架兼容。

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，并发送到指定目的地的 JMS 消息正文。

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. 另外，还可配置 JNDI 连接。
4. （可选）添加额外的 JMS 配置。

指定 JMS 传输类型

您可以将 SOAP 绑定配置为在指定 WSDL 绑定时使用 JMS 传输。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。差别在于地址规格。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

变体	描述
jndi	指定目的地名称是 JNDI 队列名称。使用这个变体时，您必须提供配置来访问 JNDI 供应商。
jndi-topic	指定目的地名称是 JNDI 主题名称。使用这个变体时，您必须提供配置来访问 JNDI 供应商。
队列	指定目的地是使用 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形式。
topic	指定目标为使用 JMS 解析的主题名称。提供的字符串传递到 **Session.createTopic（）**，以创建目的地的表示形式。

JMS URI *的选项* 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。

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，将使用 JMS URI 指定端点的目标目的地。JMS URI 也可以通过在 URI 中附加一个或多个选项来配置 JMS 连接。这些选项在 IETF 标准中详述，Java 消息服务 1.0 的 URI Scheme 进行了详细。它们可用于配置 JNDI 系统、要使用的回复目的地、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，您可以在 JMS URI 的末尾附加一个或多个选项，方法是使用问号(?)从目标地址中分离它们。多个选项由 符号（和）分隔。例 13.4 “JMS URI 选项的语法” 演示了在 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 属性设置为 URI 选项” 显示影响 JMS 传输层的 URI 选项。

表 13.2. JMS 属性设置为 URI 选项

属性	默认	描述
conduitIdSelectorPrefix		[可选] 一个字符串值，它前缀放在 conduit 所创建的所有 correlation ID 中。选择器可以使用它来侦听回复。
deliveryMode	**PERSISTENT**	指定是否使用 JMS PROJECTI **STENT** 或 **NON_PROJECTISTENT** 消息语义。如果是 BASEI **STENT** 交付模式，JMS 代理会在确认它们前将消息存储在持久性存储中；而 **NON_955ISTENT** 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提供程序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留下它的状态。
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： • 字节 • text • 二进制（二进制）
password		[可选] 指定用于创建连接的密码。不建议将此属性附加到 URI。
priority	**4**	指定 JMS 消息优先级，范围从 0（最低）到 9（最高）。
receiveTimout	**60000**	指定时间（以毫秒为单位），在使用请求/回复时，客户端将等待回复。

属性	默认	描述
reconnectOnException	**true**	[在 CXF 3.0] 中已弃用，指定传输是否应在异常时重新连接。 从 3.0 开始，当出现异常时，传输总是重新连接。
replyToName		[可选] 指定队列信息的回复目的地。回复目的地会出现在 **JMSReplyTo** 标头中。建议为具有请求性语义的应用设置此属性，因为如果未指定，JMS 提供程序将分配一个临时回复队列。 此属性的值会根据 JMS URI 中指定的变体进行解释： ● **JNDI** 变体 - 由 JNDI 解析的目标队列的名称 ● 使用 JMS 解析目标队列的队列变体名称
sessionTransacted	**false**	指定事务类型。有效值为： ● **true**- 资源本地事务 ● **false**-JTA 事务
timeToLive	**0**	指定 JMS 提供程序将丢弃消息的时间（以毫秒为单位）。值为 **0** 表示无限生命周期。
topicReplyToName		[可选] 指定主题信息的答复目的地。此属性的值会根据 JMS URI 中指定的变体进行解释： ● **JNDI-topic**- 由 JNDI 解析的目标主题的名称 ● **主题**- 由 JMS 解析的目标主题的名称

属性	默认	描述
使用ConduitIdSelector	**true**	指定 conduit 的 UUID 作为所有关联 ID 的前缀。 因为所有 conduits 都被分配一个唯一 UUID，请将此属性设置为 **true** 可让多个端点共享 JMS 队列或主题。
username		[可选] 指定用来创建连接的用户名。

JNDI 属性

表 13.3 “JNDI 属性设为 URI 选项” 显示可用于为此端点配置 JNDI 的 URI 选项。

表 13.3. JNDI 属性设为 URI 选项

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（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、蓝图或 JNDI 中搜索的 JTA 事务管理器的名称。如果找到事务管理器，则将启用 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。

其他 JNDI 属性

其属性 java.naming.factory.initial 和 java.naming.provider.url 是初始化任何 JNDI 提供者所需的标准属性。但是，除标准外，JNDI 提供程序还可能支持自定义属性。在本例中，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，您可以使用 JNDI 属性 java.naming.factory.control 在 JMS URI 中，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
```

示例

如果尚未配置 JMS 提供程序，则可以使用选项在 URI 中提供先决条件 JNDI 配置详情（请参阅 表 13.3 “JNDI 属性设为 URI 选项”）。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，可使用 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：

创建代表类端点地址的 JMS URI。

实例化 JaxWsServerFactoryBean 以发布该服务。

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。

消耗服务

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 JaxWsProxyFactoryBean 类，如 例 13.8 “消耗 SOAP/JMS 服务” 所示。

例 13.8. 消耗 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 “消耗 SOAP/JMS 服务” 中的代码执行以下操作：

创建代表类端点地址的 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中（在绑定范围、服务范围或端口范围）中，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，然后可用来查找 JMS 目的地。您也可以设置一些影响 JMS 传输层行为的属性。

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，请将以下设置添加到 wsdl:definitions 元素中：

```
<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	默认	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。

元素	默认	描述
soapjms:jndiConnectionFactoryName		指定 JMS 连接工厂的 JNDI 名称。
soapjms:deliveryMode	**PERSISTENT**	指定是否使用 JMS PROJECTI **STENT** 或 **NON_PROJECTISTENT** 消息语义。如果是 BASEI **STENT** 交付模式，JMS 代理会在确认它们前将消息存储在持久性存储中；而 **NON_955ISTENT** 消息仅保存在内存中。
soapjms:replyToName		[可选] 指定队列信息的回复目的地。回复目的地会出现在 **JMSReplyTo** 标头中。建议为具有请求性语义的应用设置此属性，因为如果未指定，JMS 提供程序将分配一个临时回复队列。 此属性的值会根据 JMS URI 中指定的变体进行解释： ● **JNDI** 变体 - 由 JNDI 解析的目标队列的名称 ● 使用 JMS 解析目标队列的队列变体名称
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 提供程序将丢弃消息的时间（以毫秒为单位）。值为 **0** 代表无限生命周期。

配置范围

WSDL 合同中的 WSDL 元素放置会影响合同中定义的端点更改的范围。SOAP/JMS WSDL 元素可以放置为 ws:binding 元素的子项、wsdl:service 元素或 wsdl:port 元素。SOAP/JMS 元素的父项决定将配置放入以下哪个范围：

绑定范围

您可以通过在 wsdl: *binding* 元素内放置扩展元素，将 JMS 传输配置为绑定范围。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在服务范围或端口范围中覆

盖。

服务范围

您可以通过将扩展元素放在 **wsdl:** ***service*** 元素内，将 JMS 传输配置为服务范围。此范围内的元素定义此服务中所有端点的默认配置。服务范围中的任何设置都可在端口范围中覆盖。

端口范围

您可以通过将扩展元素放在 **wsdl:** ***port*** 元素内，将 JMS 传输配置为端口范围。端口范围中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。

示例

例 13.9 “与 SOAP/JMS 配置进行 WSDL 合同” 显示 SOAP/JMS 服务的 WSDL 合同。它在绑定范围中配置 JNDI 层，即服务范围中的消息发送详情，以及端口范围中的回复目的地。

例 13.9. 与 SOAP/JMS 配置进行 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 “与 SOAP/JMS 配置进行 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 delivery 风格设置为非持久性，每个消息都设置为 live。

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 传输的通用实施。通用 JMS 传输不限于使用 SOAP 消息，并允许连接到使用 JMS 的任何应用程序。

注意 ：在 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，并使用将 JMS 消息与 text Message 或 ByteMessage 的正文交换的应用程序。

有两种方法可以启用和配置 JMS 传输：

- 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 使用单一 JMS 配置 bean 配置来配置 JMS 端点。bean 由 org.apache.cxf.transport.jms.JMSConfiguration 类实施。它可用于直接配置端点，或者配置 JMS conduit 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。要使用此命名空间，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 来指定 JMS 配置。an 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” 列出对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] 指定对定义 JMS ConnectionFactory 的 bean 的引用。
wrapInSingleConnectionFactory	**true** [pre v3.0]	**CXF 3.0 中删除** **预先** CXF 3.0 指定是否将 ConnectionFactory 与 Spring **SingleConnectionFactory** 换行。 在使用没有池连接的 ConnectionFactory 时启用此属性，因为它可以提高 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，使它能被重复使用。

属性	默认	描述
reconnectOnException	**false**	在 发生异常时，CXF 中已弃用 CXF 始终重新连接。 pre CXF 3.0 指定在出现异常时是否创建新连接。 使用 Spring **SingleConnectionFactory** 嵌套 ConnectionFactory 时： ● 对一个例外情况进行 **true** busybox，创建一个新连接 在使用 PooledConnectionFactory 时，不要启用这个选项，因为这个选项只返回池的连接，但不会重新连接。 ● 异常错误，错误，不要尝试重新连接
targetDestination		指定目的地的 JNDI 名称或供应商名称。
replyDestination		指定发送回复的 JMS 目的地的 JMS 名称。此属性允许使用用户定义的目的地来回复。详情请查看 第 14.6 节 “使用 Named Reply Destination”。
destinationResolver	DynamicDestinationResolver	指定对 Spring **DestinationResolver** 的引用。 此属性允许您定义目标名称如何解析到 JMS 目的地。有效值为： ● **DynamicDestinationResolver** sHistoryLimit->_<resolve 目标名称使用 JMS 提供程序的功能。 ● **JndiDestinationResolver** >_<-resolve 目标名称使用 JNDI。
transactionManager		指定对 Spring 事务管理器的引用。这可让服务参与 JTA 事务。

属性	默认	描述
taskExecutor	**SimpleAsyncTaskExecutor**	CXF 3.0 **中删除** **前 CXF 3.0** 指定对 Spring TaskExecutor 的引用。这在监听程序中用来决定如何处理传入的信息。
useJms11	**false**	CXF 3.0 CXF 3.0 **中删除** 仅支持 JMS 1.1 功能。 **前 CXF 3.0** 指定是否使用了 JMS 1.1 功能。有效值为： ● **true** abrt- theJMS 1.1 功能 ● **false** abrt- theJMS 1.0.2 功能
messageIdEnabled	**true**	CXF 3.0 **中删除** **前 CXF 3.0** 指定 JMS 传输是否希望 JMS 代理提供消息 ID。有效值为： ● **true** KUBECONFIG-broker 需要提供消息 ID ● **false** InventoryService-brokerbroker 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，代理会提供一个提示，它不需要生成消息 ID，或将其添加到端点的消息中。代理可以接受 hint，或者忽略它。

属性	默认	描述
messageTimestampEnabled	**true**	**CXF 3.0 中删除** **前 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： ● **true** abrt- thebroker 需要提供消息时间戳 ● **false** InventoryService-thebroker 不需要提供消息时间戳 在本例中，端点调用其消息 producer 的 **setDisableMessageTimestamp（）** 方法，值设为 **true**。然后，代理会提供一个提示，它不需要生成时间戳，或将其添加到端点的消息中。代理可以接受 hint，或者忽略它。
cacheLevel	**-1**（禁用功能）	**CXF 3.0 中删除** **前 CXF 3.0** 指定 JMS 侦听器容器可以应用的缓存级别。有效值为： ● **0** – CACHE_NONE ● **1** INVENTORYSERVICE-JAXBCACHE_CONNECTION ● **2** – CACHE_SESSION ● **3** – CACHE_CONSUMER ● **4** – CACHE_AUTO 详情请参阅 Class DefaultMessageListenerContainer
pubSubNoLocal	**false**	指定在使用主题时是否收到您自己的消息。 ● **true** NETWORK- thedo 没有接收您自己的信息 ● **false** WWPN-receive 询问您自己的信息
receiveTimeout	**60000**	指定时间（以毫秒为单位），以等待响应消息。

属性	默认	描述
explicitQosEnabled	**false**	指定每个消息是否明确设置 QoS 设置（如优先级、持久性、时间为实时），或使用默认值(**false**)。
deliveryMode	**2**	指定消息是否持久。有效值为： ● **1** (NON_955ISTENT)-messages 只保存内存 ● **2** (EFIISTENT)消息被保留到磁盘
priority	**4**	指定消息优先级。JMS 优先级值的范围是从 **0**（最低）到 **9**（最高）。详情请参阅您的 JMS 供应商的文档。
timeToLive	**0**（无限）	指定丢弃的消息前的时间（以毫秒为单位）。
sessionTransacted	**false**	指定是否使用了 JMS 事务。
concurrentConsumers	**1**	**CXF 3.0 中删除** **前 CXF 3.0** 指定侦听器的最小并发用户数。
maxConcurrentConsumers	**1**	**CXF 3.0 中删除** **前 CXF 3.0** 为监听器指定最大并发用户数。
messageSelector		指定用于过滤传入消息的选择器的字符串值。此属性允许多个连接共享队列。有关使用指定选择器的语法的更多信息，请参阅 JMS 1.1 规格。
subscriptionDurable	**false**	指定服务器是否使用持久订阅。
durableSubscriptionName		指定注册持久订阅的名称（字符串）。

属性	默认	描述
messageType	**text**	指定消息数据如何打包为 JMS 消息。有效值为： • 文本 abrt->_< 表明数据将被打包为 **text Message** • **byte** Equal-jaxbspec 表示数据将被打包为字节数组(**byte[]**) • **二进制** sHistoryLimit->_<spec 表示数据将被打包为 **ByteMessage**
pubSubDomain	**false**	指定目标目标是否为一个主题还是一个队列。有效值为： • **true** abrt- thetopic • **false** InventoryService-thequeue
jmsProviderTibcoEms	**false**	指定 JMS 提供程序是 Tibco EMS。 当设置为 **true** 时，安全性上下文中的主体会从 **JMS_TIBCO_SENDER** 标头填充。
useMessageIDAsCorrelationID	**false**	**CXF 3.0 中删除** 指定 JMS 是否将使用消息 ID 来关联消息。 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 目标可能具有的最大暂停数量。当当前数量超过指定的最大值时，JMSListenerContainer 将停止。

属性	默认	描述
reconnectPercentOfMax	**70**	CXF 3.0 指定当重启 JMSListenerContainer 时，会超过 **maxSuspendedContinuations**。 当其当前暂停持续数低于 **(maxSuspendedContinuations * reconnectPercentOfMax/100)** 的值时，监听程序容器会被重启。

如 例 14.2 “JMS 配置 bean” 所示，an 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 可以直接使用 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. 为该功能添加类型 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” 显示使用来自 例 14.2 “JMS 配置 bean” 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

JMSConfiguration Bean 可以使用 jms:jmsConfig-ref 元素应用到 JMS conduits 和 JMS 目的地。jms:jmsConfig-ref 元素的值是 JMSConfiguration bean 的 ID。

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

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，CXF 为每个消息创建一个新的 JMS 会话和 JMS producer。这是因为会话和制作者对象

都是安全线程的。创建制作者特别要大量时间，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，请参阅红帽 JBoss Fuse 事务指南中的"Appendix A 优化 JMS Single- 和 Multiple-Resource Transactions"的性能

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。尽可能使用 JMS MessageListener 异步实施请求/回复消息传递。

但是，当需要在端点间共享队列时，CXF 必须使用同步的 Consumer.receive（） 方法。此场景需要 MessageListener 来使用消息选择器来过滤消息。消息选择器必须预先识别，因此 MessageListener 只打开一次。

应该避免预先知道消息选择器的两个情况：

- 当 JMSMessageID 用作 JMSCorrelationID时

  如果 JMS 属性使用 ConduitIdSelector 和 conduitSelectorPrefix，则客户端不会设置 JMS 传输，则客户端不会设置 JMSCorrelationId。这会导致服务器使用请求消息的

JMSMessageId 作为 JMSCorrelationId。因为 JMSMessageID 无法预先识别，客户端必须使用 synchronous Consumer.receive（） 方法。

请注意，您必须在 IBM JMS 端点（默认）中使用 Consumer.receive（） 方法。

- 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 synchronous Consumer.receive（） 方法。

因此，一般规则是避免使用需要同步接收的设置。

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，使用本地资源的事务才会回滚 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，请根据需要配置端点，并将属性 sessionTrasnsacted 设置为 true。

注意

有关事务和池的更多信息，请参阅红帽 JBoss Fuse 事务指南。

JTA 事务

通过使用 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，它会在调用服务实施前启动事务。如果没有异常，则事务将被提交。否则，它将被回滚。

在 JTA 事务中，使用 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，因此消息

被消耗，数据就会被写入数据库，或者信息被回滚，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
```

    这个示例找到了 ID TransactionManager 的 bean。

- OSGi 参考方法
  - 使用 Blueprint 将事务管理器作为 OSGi 服务查找

```
<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。

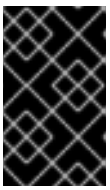
2. **配置 JCA 池的连接工厂**

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，它被提供给 JcaPooledConnectionFactory。然后，JcaPooledConnectionFactory 作为默认的 bean 提供了 ID ConnectionFactory。

请注意，JcaPooledConnectionFactory 类似于普通的 ConnectionFactory。但是，打开新的连接和会话时，它会检查 XA 事务（如果找到）将 JMS 会话作为 XA 资源注册。这允许 JMS 会话参与 JMS 事务。



重要

对 JMS 传输直接设置 XA ConnectionFactory 无法正常工作！

14.5. 使用 WSDL 配置 JMS

14.5.1. JMS WSDL 扩展名称

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” 中显示的行添加到合同的定义元素中。

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 元素及其子项提供，即 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 元素的子项来完成。WSDL 中使用的 jms:address 元素与配置文件中使用的相同。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 目的地是否为 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	指定将请求发送到的 JMS 目的地的名称。
jmsReplyDestinationName	指定发送回复的 JMS 目的地的 JMS 名称。此属性允许您使用用户定义的目的地来回复。详情请查看 第 14.6 节 “使用 Named Reply Destination”。
jndiDestinationName	指定绑定到请求发送到的 JMS 目的地的 JNDI 名称。
jndiReplyDestinationName	指定与发送回复的 JMS 目的地的 JNDI 名称。此属性允许您使用用户定义的目的地来回复。详情请查看 第 14.6 节 “使用 Named 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 代理时使用的密码。

jms:address WSDL 元素使用 jms:JMSNamingProperties 子元素，以指定连接 JNDI 提供程序所需要的其他信息。

指定 JNDI 属性

为提高与 JMS 和 JNDI 提供程序的互操作性，jms:address 元素包含一个子元素(jms:JMSNamingProperties)，它允许您指定用于在连接 JNDI 提供程序时使用的属性的值。jms:JMSNamingProperties 元素具有两个属性，即 name 和 value。name 指定要设置的属性名称。value 属性指定指定的值。JMS：JMSNamingProperties 元素也可用于规范供应商特定属性。

以下是您可以设置的通用 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消息的更多详细信息，请检查您的 JNDI 供应商的文档，并查阅 Java API 参考材料。

示例

例 14.6 “JMS WSDL 端口规格” 显示 JMS WSDL 端口 规格的示例。

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所使用的消息类型。JMS 使用者端点可以使用 JMS ByteMessage 或 JMS TextMessage。

在使用 ByteMessage 时，消费者端点使用 byte[] 作为数据存储在 JMS 邮件正文中的数据并检索数据。发送消息时，消息数据（包括任何格式信息）被打包成 字节[]，并在将消息正文放置在线路上前将其

放在消息正文中。收到消息时，使用者端点将尝试解包消息正文中存储的数据，就像将其打包在 字节[] 中一样。

在使用 text Message 时，使用者端点使用字符串作为来自消息正文的数据的方法。发送消息时，消息信息（包括任何格式的信息）将转换为字符串，并放入 JMS 消息正文。收到消息时，使用者端点将尝试解包在 JMS 消息正文中存储的数据，就像将其打包成字符串一样。

当原生 JMS 应用与 Apache CXF 消费者交互时，JMS 应用负责解释消息和格式化信息。例如，如果 Apache CXF 合同指定用于 JMS 端点的绑定是 SOAP，且消息将被打包为 text Message，接收 JMS 应用会得到一个包含所有 SOAP envelope 信息的文本信息。

指定消息类型

JMS 使用者端点接受的消息类型通过可选的 jms:client 元素进行配置。jms:client 元素是 WSDL 端口 元素的子项，并具有一条属性：

表 14.3. JMS 客户端 WSDL 扩展

messageType
指定消息数据如何打包为 JMS 消息。文本指定数据将打包为 **text Message**。二进制文件 指定数据将打包为 **ByteMessage**。

示例

例 14.7 “用于 JMS 消费者端点的 WSDL” 显示用于配置 JMS 消费者端点的 WSDL。

例 14.7. 用于 JMS 消费者端点的 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 提供程序配置

概述

JMS 提供程序端点具有许多可以配置的行为。它们是：

- 如何关联消息
- 使用持久订阅
- 如果服务使用本地 JMS 事务
- 端点使用的选择器

指定配置

提供商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 元素的子项，具有下列属性：

表 14.4. JMS 提供程序端点 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。有关使用指定选择器的语法的更多信息，请参阅 JMS 1.1 规格。
事务性	指定本地 JMS 代理是否围绕消息处理创建事务。默认值为 **false**。[a]
[a] 目前，运行时不支持将 **事务** 属性设置为 **true**。	

示例

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

例 14.8. 用于 JMS 供应商端点的 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 NAMED REPLY DESTINATION

概述

默认情况下，使用 JMS 的 Apache CXF 端点创建一个临时队列来回发送回复。如果要使用命名队列，您可以配置用于将回复作为端点 JMS 配置的一部分来发送回复的队列。

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的目的地的回复，它将在所有传出请求的 ReplyTo 字段中指定 属性的值。服务端点将使用 jndiReplyDestinationName 属性的值，当请求的 ReplyTo 字段中没有指定目的地时，将放置回复的位置。

示例

例 14.9 “使用 Named Reply Queue 的 JMS Consumer 规格” 显示 JMS 客户端端点的配置。

例 14.9. 使用 Named Reply Queue 的 JMS Consumer 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

如果您将 Apache ActiveMQ 用作 JMS 提供程序，则以特殊格式指定您的目的地的 JNDI 名称，可以动态为队列或主题创建 JNDI 绑定。这意味着 不需要使用 您的队列或主题的 JNDI 绑定来配置 JMS 供应商。

初始上下文工厂

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

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

例 15.1. SOAP/JMS WSDL 以连接到 Apache ActiveMQ

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

在 例 15.1 “SOAP/JMS WSDL 以连接到 Apache ActiveMQ” 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，InitialContext 实例中有一个预定义绑定，它会将 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” 控制 SOAP/JMS 扩展元素，用于指定 Apache ActiveMQ 连接工厂。

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 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。它们没有 抽象 JNDI 名称。

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

例 15.3. 带有动态创建队列的 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 JNDI 名称为 greeter.request.queue 的队列。如果不存在，它将创建一个新的队列，并将其绑定到 JNDI 名称 greeter.request.queue。

第 16 章 CONDUITS

摘要

不同的是实施出站连接的传输架构的低级别部分。其行为和生命周期可影响系统性能并处理负载。

概述

在 Apache CXF 运行时，管理客户端或出站传输详情。它们负责打开端口、建立出站连接、发送消息和侦听应用程序与单个外部端点之间的任何响应。如果应用程序连接到多个端点，它将为每个端点有一个双重实例。

每种传输类型使用 Conduit 接口实施自己的一致性。这允许应用程序级别功能和传输间的标准化接口。

通常，您只需要在配置客户端传输详情时考虑应用程序所使用的内容。运行时如何处理 conduits 的底层语义是，并非开发人员需要担心的。

但是，当了解 conduit 时，会非常有用：

- 实施自定义传输
- 管理有限资源的高级应用程序调整

CONDUIT 生命周期

conduits 由客户端实施对象管理。创建之后，在客户端实施对象期间持续存在。conduit 的生命周期为：

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 选择器对 conduit 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

施。否则，它会将客户端的引用传递给目标端点的 conduit。

3. 在需要时发送信息。

4. 当客户端实现对象被销毁时，与其关联的所有步骤都会被销毁。

CONDUIT WEIGHT

conduit 对象的 weight 取决于传输实施。HTTP conduits 非常轻便的权重。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

使用三个 Spring 配置元素之一来配置 JAX-WS 端点。正确的元素取决于您配置的端点的类型以及您想要使用哪些功能。对于消费者，您可以使用 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。您可以使用配置元素来覆盖合同中的信息。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激活一些高级功能，如 WS-RM。这可以通过为端点的配置元素提供子元素来实现。请注意，当处理使用 Java 优先方法开发的端点时，SEI 服务可能会因为端点的合同缺乏与要使用的绑定和传输有关的信息。

17.1. 配置服务供应商

17.1.1. 配置服务提供程序的元素

Apache CXF 有两个可用于配置服务供应商的元素：

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

两个元素之间的区别对于运行时主要是内部的。jaxws:endpoint 元素将属性注入 org.apache.cxf.jaxws.EndpointImpl 对象来支持服务端点。jaxws:server 元素将属性注入 org.apache.cxf.jaxws.support.support.JaxWsServerFactoryBean 对象来支持端点。EndpointImpl 对象将配置数据传递给 JaxWsServerFactoryBean 对象。JaxWsServerFactoryBean 对象用于创建实际服务对象。由于任一配置元素都会配置服务端点，您可以根据您首选的语法进行选择。

17.1.2. 使用 jaxws:endpoint Element

概述

jaxws:endpoint 元素是配置 JAX-WS 服务提供商的默认元素。其属性和子项指定实例化服务提供商所需的所有信息。许多属性映射到服务的合同中信息。子项用于配置拦截器和其他高级功能。

识别正在配置的端点

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施端点的类。这使用 jaxws:endpoint 元素的 implements or 属性来完成。

对于不同端点共享通用实现的实例，可以为每个端点提供不同的配置。有两种方法来区分配置中的特定端点：

- serviceName 属性和 endpointName 属性的组合

  serviceName 属性指定定义该服务端点的 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 *ns*:*name* 的格式指定为 QNames。*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，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
implementor	指定实施该服务的类。您可以使用类名称或 Spring bean 配置实施类类来指定实施类。此类必须在 classpath 上。
implementorClass	指定实施该服务的类。当提供给 **implementor** 属性的值作为使用 Spring AOP 包装的 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服务的文件夹。
endpointName	指定服务 **wsdl:port** 元素的 **name** 属性的值。它被指定为 QName，格式为 ***ns:name***，其中 *ns* 是 **wsdl:port** 元素的命名空间。
serviceName	指定服务 **wsdl:service** 元素的 **name** 属性的值。它被指定为 QName，格式为 ***ns:name***，其中 *ns* 是 **wsdl:service** 元素的命名空间。
publish	指定是否应该自动发布该服务。如果此项设为 **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。
总线	指定 Spring Bean 的 ID，配置用于管理服务端点的总线。这在将多个端点配置为使用一组通用的功能时，这很有用。
bindingUri	指定服务使用的消息绑定 ID。第 23 章 *Apache CXF Binding ID* 中提供了有效绑定 ID 列表。
name	指定服务 **wsdl:port** 元素的字符串指定 QName。它使用 ***{ns}localPart*** 格式指定为 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。abstract Bean 充当拥塞 Bean 定义且没有实例化的父项。默认值为 **false**。将其设置为 **true** 指示 bean 工厂不会实例化 bean。
dependent-on	指定在端点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 创建的用户，如 **Endpoint.publish（）** 或 **Service.getPort（）**。 默认值为 **false**。 将其设置为 **true** 执行以下操作： ● 通过将 **.jaxws-endpoint** 附加到其 id 来修改 bean 的内部名称 ● 使 bean 摘要
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，您可能需要使用多个 ns:*shortName*属性来 声明 endpointName 和 serviceName 属性使用的命名空间。

示例

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。这个示例假定您要对所有其他值使用默认值，或者实现在注解中指定了值。

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 “带有服务名称的 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，您必须指定要使用 serviceName 属性来实例化哪个服务定义。

例 17.2. 带有服务名称的 JAX-WS 端点配置

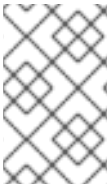
```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">

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

</beans>
```

xmlns:samp 属性指定定义了 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 属性设为 true。



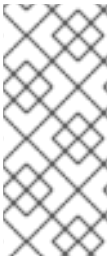
注意

这个 **HTTP/2 的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

为了提高性能，红帽建议在 Apache Karaf 上使用 servlet 传输(pax-web-undertow)，这样可启用对 web 容器的集中配置和调优，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 元素是配置 JAX-WS 服务提供商的一个元素。它将配置信息注入到 org.apache.cxf.jaxws.support.JaxWsServerFactoryBean。这是 Apache CXF 特定对象。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。

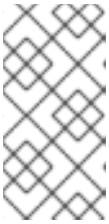
识别正在配置的端点

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施端点的类。这可以通过 jaxws:server 元素的 serviceBean 属性来完成。

对于不同端点共享通用实现的实例，可以为每个端点提供不同的配置。有两种方法来区分配置中的特定端点：

- serviceName 属性和 endpointName 属性的组合

  serviceName 属性指定定义该服务端点的 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 *ns*:*name* 的格式指定为 QNames。*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，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 Element 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 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。
serviceBean	指定实施该服务的类。您可以使用类名称或 Spring bean 配置实施类类来指定实施类。此类必须在 classpath 上。
serviceClass	指定实施该服务的类。当提供给 **implementor** 属性的值作为使用 Spring AOP 包装的 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服务的文件夹。
endpointName	指定服务 **wsdl:port** 元素的 **name** 属性的值。它被指定为 QName，格式为 ***ns:name***，其中 *ns* 是 **wsdl:port** 元素的命名空间。

属性	描述
serviceName	指定服务 **wsdl:service** 元素的 **name** 属性的值。它被指定为 QName，格式为 ***ns:name***，其中 *ns* 是 **wsdl:service** 元素的命名空间。
publish	指定是否应该自动发布该服务。如果此项设为 **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。
总线	指定 Spring Bean 的 ID，配置用于管理服务端点的总线。这在将多个端点配置为使用一组通用的功能时，这很有用。
bindingId	指定服务使用的消息绑定 ID。第 23 章 *Apache CXF Binding ID* 中提供了有效绑定 ID 列表。
name	指定服务 **wsdl:port** 元素的字符串指定 QName。它被指定为 QName，格式为 ***{ns}localPart***，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。abstract Bean 充当拥塞 Bean 定义且没有实例化的父项。默认值为 **false**。将其设置为 **true** 指示 bean 工厂不会实例化 bean。
dependent-on	指定在端点实例化前端点要实例化的 Bean 列表。
createdFromAPI	指定使用 Apache CXF API 创建的用户，如 **Endpoint.publish（）** 或 **Service.getPort（）**。 默认值为 **false**。 将其设置为 **true** 执行以下操作： ● 通过将 **.jaxws-endpoint** 附加到其 id 来修改 bean 的内部名称 ● 使 bean 摘要

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

示例

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. 为服务提供商添加功能

概述

jaxws:endpoint 和 jaxws:server 元素提供实例化服务提供商所需的基本配置信息。要向服务提供商添加功能或执行高级配置，您必须将子元素添加到配置中。

子元素允许您进行以下操作：

- 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 *启用可靠消息*
- 第 17.1.5 节 “在 JAX-WS 端点上启用 Schema 验证”

元素

表 17.3 “用于配置 JAX-WS 服务提供程序的元素” 描述 jaxws:endpoint 支持的子元素。

表 17.3. 用于配置 JAX-WS 服务提供程序的元素

元素	描述
jaxws:handlers	指定用于处理消息的 JAX-WS Handler 实施列表。有关 JAX-WS Handler 实施的更多信息，请参阅 第 43 章 *编写处理程序*。
jaxws:inInterceptors	指定处理入站请求的拦截器列表。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxws:inFaultInterceptors	指定处理入站错误信息的拦截器列表。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。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxws:outFaultInterceptors	指定处理出站错误信息的拦截器列表。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。使用 **org.apache.cxf.binding.Binding** factory 接口的实现来配置消息绑定。[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 定义进行指定。
jaxws:executor	指定用于该服务的 Java 执行程序。这使用嵌入式 bean 定义进行指定。
jaxws:features	指定 Beans 列表，用于配置 Apache CXF 的高级功能。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 接口的实施。[c]
jaxws:properties	指定传递给端点的属性的 Spring 映射。这些属性可用于控制启用 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。

[a] SOAP 绑定使用 **soap: SOApBinding** bean 进行配置。

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

[c] Invoker 实施控制如何调用服务。例如，它控制每个请求是否由服务实施的新实例处理，或在调用中保留状态。

17.1.5. 在 JAX-WS 端点上启用 Schema 验证

概述

您可以将 schema-validation-enabled 属性设置为在 jaxws:endpoint 元素上启用 schema 验证，或者在 jaxws:server 元素中启用 schema 验证。启用 schema 验证后，检查客户端和服务器间发送的消息，以符合 schema。默认情况下关闭 schema 验证，因为它对性能有显著影响。

示例

要在 JAX-WS 端点上启用架构验证，请在 jaxws:properties 子元素或 jaxws:server 元素中设置 schema-validation-enabled 属性。例如，要在 jaxws:endpoint 元素中启用 schema 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 “架构验证类型值”。

17.2. 配置 CONSUMER 端点

概述

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

要将其他功能（如 WS-RM）添加到添加到 jaxws:client 元素的使用者。子元素也用于配置端点的日志记录行为，并将其他属性注入到端点的实施中。

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 消费者所需的基本信息。您只需要为您要配置的特定属性提供值。大多数属性都有可识别的默认值，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	指定消费者发出请求的端点的 HTTP 地址。这个值覆盖合同中设定的值。
bindingId	指定消费者使用的消息绑定 ID。第 23 章 *Apache CXF Binding ID* 中提供了有效绑定 ID 列表。
总线	指定 Spring bean 配置总线管理端点的 ID。
endpointName	指定消费者发出请求的服务的 **wsdl:port** 元素的 **name** 属性的值。它被指定为 QName，格式为 ***ns*:*name***，其中 *ns* 是 **wsdl:port** 元素的命名空间。
serviceName	指定消费者发出请求的服务的 **wsdl:service** 元素的 **name** 属性的值。它被指定为 QName，格式为 ***ns*:*name***，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	指定消费者发出请求的服务的 **wsdl:port** 元素的字符串指定。它被指定为 QName，格式为 ***{ns}localPart***，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。abstract Bean 充当拥塞 Bean 定义且没有实例化的父项。默认值为 **false**。将其设置为 **true** 指示 bean 工厂不会实例化 bean。
dependent-on	指定在端点实例化前端点要实例化的 Bean 列表。
createdFromAPI	指定使用了 Apache CXF API（如 **Service.getPort（）** ）创建的用户。 默认值为 **false**。 将其设置为 **true** 执行以下操作： ● 通过将 **.jaxws-client** 附加到其 id 来修改 bean 的内部名称 ● 使 bean 摘要

除了 表 17.4 “用于配置 JAX-WS Consumer 的属性” 中列出的属性，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

要为消费者添加功能或执行高级配置，您必须将子元素添加到配置中。

子元素允许您进行以下操作：

- 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 *启用可靠消息*
- “在 JAX-WS 消费者上启用架构验证”一节

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

表 17.5. 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。使用 **org.apache.cxf.binding.Binding** factory 接口的实现来配置消息绑定。[a]
jaxws:dataBinding	指定实施端点使用的数据绑定的类。您使用嵌入式 bean 定义来指定此值。实施 JAXB 数据源的类是 **org.apache.cxf.jaxb.JAXBDataBinding**。
jaxws:features	指定 Beans 列表，用于配置 Apache CXF 的高级功能。您可以提供 bean 引用列表或嵌入式 Bean 列表。

元素	描述
jaxws:handlers	指定用于处理消息的 JAX-WS Handler 实施列表。有关 JAX-WS Handler 实现的更多信息，请参阅 第 43 章 *编写处理程序*。
jaxws:inInterceptors	指定进程入站响应的拦截器列表。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxws:inFaultInterceptors	指定处理入站错误信息的拦截器列表。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。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxws:outFaultInterceptors	指定处理出站错误信息的拦截器列表。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	指定一个 org.apache.cxf.endpoint.ConduitSelector 实现，供客户端使用。ConduitSelector 实施将覆盖用来选择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 进行配置。	

示例

例 17.5 “简单消费者配置” 显示了简单的使用者配置。

例 17.5. 简单消费者配置

```
<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 消费者上启用架构验证

要在 JAX-WS 使用者上启用架构验证，请在 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 “架构验证类型值”。

第 18 章 配置 JAX-RS 端点

摘要

本章论述了如何在 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，用于定义 XML 中的端点。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. 使用 jaxrs:server 元素的地址属性的 JAX-RS 服务的基本 URL。请注意，有两种不同的方法来指定地址 URL，这会影响端点的部署方式：

   - 作为相对 URL- 例如： /customers。在本例中，端点被部署到默认的 HTTP 容器中，通过将 CXF servlet 基本 URL 与指定的相对 URL 相结合，端点的基本 URL 会被隐式获得。

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

   - 作为绝对 URL 10.10.10.2-sHistoryLimit，例如 http://0.0.0.0:8200/cxf/customers。在本例中，为 JAX-RS 端点打开一个新的 HTTP 侦听器端口（如果尚未打开）。例如，在 Fuse 上下文中，会隐式创建新的 Undertow 容器以托管 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（对于多设备主机机器很有用）。

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 元素内列出它们。

蓝图示例

以下 Blueprint XML 示例演示了如何定义 JAX-RS 端点，该端点指定相对地址 /customers （因此将其部署到默认 HTTP 容器）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 Blueprint 中定义 JAX-RS 端点，通常需要以下 XML 命名空间：

prefix	命名空间
（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，用于指定相对地址 /customers （因此将其部署到默认 HTTP 容器）并由 服务实施。CustomerService 资源类：

```
<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，通常需要以下 XML 命名空间：

prefix	命名空间
（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） 明确指定 JAX-RS 根资源类，Spring XML 可让您配置自动发现，以便搜索特定 Java 软件包以获取资源类（由 @Path标注的类），并且所有发现的资源类都自动附加到端点。在这种情况下，您需要在 jaxrs:server 元素中仅指定 address 属性和 basePackages 属性。

例如，若要定义使用 a.b.c Java 软件包下的所有 JAX-RS 资源类的 JAX-RS 端点，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，在其在指定的 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 XML） 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） 创建一个单个 bean 实例，该实例在 Spring 容器的整个生命周期中使用。

prototype

每次将 bean 注入到另一个 bean 或 bean 获取 bean 时，或通过调用 bean registry 上的 getBean（） 来新建一个 bean 实例。

Request（请求）

（仅适用于 Web 感知型容器） 为 Bean 上调用的每个请求创建一个新的 bean 实例。

session

（只在一个 Web 感知容器中可用） 为单个 HTTP 会话生命周期创建一个新 Bean。

globalSession

（只在一个 Web 感知容器中可用） 为在 portlet 之间共享的单个 HTTP 会话的生命周期创建一个新 Bean。

如需有关 Spring 范围的更多详细信息，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，Spring 范围 无法正常工作。如果您在此例中指定 resource Bean 上的 scope 属性，则 scope 属性将有效忽略。

要让 bean 范围在 JAX-RS 服务器端点内正常工作，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 resourceanNames 属性，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>
```

上一示例配置两个资源 Bean，即 customerBean1 和 customerBean2。beanNames 属性指定为空格分隔的资源 bean ID 列表。

为了获得灵活性，您可以显式定义服务工厂对象的选项，当您使用 jaxrs:serviceFactories 元素配置 JAX-RS 服务器端点时。这种更详细的方法有优势，您可以将默认服务工厂实施替换为您的自定义实施，从而让您最终掌控 bean 生命周期。以下示例演示了如何使用此方法配置两个资源 Bean、customerBean1 和 customerBean2 ：

```
<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，则通常最好实施和注册 org.apache.cxf.service.Invoker bean（其中，可以通过引用来自 jaxrs:server/jaxrs:invoker 元素来注册实例）。

附加 WADL 文档

您可以使用 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 模式，用于描述 JAX-B 格式的消息内容，您可以通过 jaxrs:schemaLocations 将这些外部模式与 JAX-RS 服务器端点关联。

例如，如果您已将服务器端点与 WADL 文档相关联，并且您希望在传入消息上启用 schema 验证，您可以按如下方式指定相关的 XML 架构文件：

```
<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，如果您要将所有模式文件(*.xsd)包含在给定目录中，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 模式的任何种类的功能通常有用。

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编码消息正文的数据绑定。例如，若要指定 JAX-B 数据的绑定，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 数据的绑定，您可以按照如下所示配置 JAX-RS 端点：

```
<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 配置为将 JMS 消息库用作传输协议，而不使用 HTTP。由于 JMS 本身 不是 传输协议，因此实际的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，以下有关上例的几点：

- JMS 实施- JMS 实施由 ConnectionFactory bean 提供，后者实例化 Apache ActiveMQ 连接工厂对象。在实例化连接工厂后，它会作为默认的 JMS 实现层自动安装。

- JMS conduit 或 destination 对象-Apache CXF 隐式实例化 JMS conduit 对象（代表 JMS 消费者）或 JMS 目标对象（代表 JMS 提供程序）。这个对象必须由 QName 唯一标识，它通过 setttings xmlns:s="http://books.com" 定义（定义命名空间前缀）和 serviceName="s:BookService"（ 定义 QName）。

- 传输 ID- 要选择 JMS 传输，transportId 属性必须设置为 http://cxf.apache.org/transports/jms。

- JMS 地址- jaxrs:server/@address 属性使用标准化语法来指定要发送到的 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，使它自动将文件后缀（在 URL 中应用）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，考虑以下形式的 HTTP 请求：

```
GET /resource.xml
```

您可以配置 JAX-RS 服务器端点，以自动映射 .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>
```

当上述服务器端点收到 HTTP 请求时，它会自动创建类型为 application/xml 的新内容类型标头，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，以自动映射 .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>
```

当上述服务器端点收到 HTTP 请求时，它会自动创建一个新的接受语言标头，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 元素中可用的属性。

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） 启用自动发现功能，方法是指定以逗号分隔的 Java 软件包列表，这些列表用于发现 JAX-RS 根资源类和/或 JAX-RS 提供程序类。

属性	描述
beanNames	指定 JAX-RS root 资源 Bean ID 的以空格分开的列表。在 Spring XML 上下文中，可以通过设置 root 资源 **bean** 元素上的 **scope** 属性来定义根资源 Bean 的生命周期。
bindingId	指定服务使用的消息绑定 ID。第 23 章 *Apache CXF Binding ID* 中提供了有效绑定 ID 列表。
总线	指定 Spring Bean 的 ID，配置用于管理服务端点的总线。这在将多个端点配置为使用一组通用的功能时，这很有用。
docLocation	指定外部 WADL 文档的位置。
modelRef	将模型 schema 指定为类路径资源（例如，classpath **:/path/to/model.xml**的 URL）。有关如何定义 JAX-RS 模型架构的详细信息，请参阅 第 18.3 节 “使用模型架构定义 REST 服务”。
publish	指定是否应该自动发布该服务。如果设置为 **false**，则开发人员必须明确发布端点。
publishedEndpointUrl	指定 URL 基础地址，该地址将插入到自动生成的 WADL 接口的 **wadl:resources/@base** 属性中。
serviceAnnotation	**（仅限 Spring）** 指定 Spring 中自动发现的服务注解类名称。与 **basePackages** 属性结合使用时，此选项将限制自动发现类的集合，使其 **仅包含** 此注解类型所标注的类。猜！这是否正确？
serviceClass	指定 JAX-RS 根资源类的名称（实施 JAX-RS 服务）。在这种情况下，类由 Apache CXF 进行实例化，**而不是由** Blueprint 或 Spring 进行实例化。如果要在 Blueprint 或 Spring 中实例化类，请使用 **jaxrs:serviceBeans** 子元素。
serviceName	指定在使用 JMS 传输的特殊情况下，为 JAX-RS 端点指定服务 QName（使用 ***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（替换 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。

属性	描述
abstract	（仅限 Spring） 指定 bean 是否为抽象 bean。abstract Bean 充当拥塞 Bean 定义且没有实例化的父项。默认值为 **false**。将其设置为 **true** 指示 bean 工厂不会实例化 bean。
dependent-on	(Spring only) 指定端点在端点实例化之前要实例化的 Bean 列表，然后可以实例化该端点。

18.1.3. jaxrs:server Child Elements

子元素

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

表 18.2. JAX-RS 服务器端点子元素

元素	描述
jaxrs:executor	指定用于该服务的 Java **执行程序** （线程池实施）。这使用嵌入式 bean 定义进行指定。
jaxrs:features	指定 Beans 列表，用于配置 Apache CXF 的高级功能。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 定义进行指定。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。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxrs:inFaultInterceptors	指定处理入站错误信息的拦截器列表。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。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxrs:outFaultInterceptors	指定处理出站错误信息的拦截器列表。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 接口的实施。[a]

元素	描述
jaxrs:serviceFactories	为您提供对与此端点关联的 JAX-RS 根资源的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）用于创建 JAX-RS 根资源实例。
jaxrs:properties	指定传递给端点的属性的 Spring 映射。这些属性可用于控制启用 MTOM 支持等功能。
jaxrs:serviceBeans	此元素的子项是（an 元素）的实例或引用（**ref** 元素）JAX-RS 根资源。**请注意，如果 bean 元素中存在 scope 属性 （仅限 Spring）， 则忽略。**
jaxrs:modelBeans	由一个或多个 **org.apache.cxf.jaxrs.model.UserResource** Bean 组成，它们是资源模型的基本元素（与 **jaxrs:resource** 元素）的引用列表。详情请查看 第 18.3 节 “使用模型架构定义 REST 服务”。
jaxrs:model	在此端点中直接定义资源模型（即，这个 **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 “使用模型架构定义 REST 服务”。
jaxrs:providers	使您能够使用此端点注册一个或多个自定义 JAX-RS 提供程序。此元素的子项是（an 元素）的实例或引用（**ref** 元素）JAX-RS 提供程序。
jaxrs:extensionMappings	当 REST 调用的 URL 以文件扩展结尾时，您可以使用此元素将其与特定内容类型关联。例如，**.xml** 文件扩展名可以与 **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，您可以使用此元素将这个元素映射到特定的语言。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。

元素	描述
jaxrs:schemaLocations	指定验证 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，各自指定 XML 架构文件的位置（通常是 类路径 URL）。详情请查看 "模式验证"一节。
jaxrs:resourceComparator	可让您注册自定义资源比较器，它实施用于匹配特定资源类或方法的传入 URL 路径的算法。
jaxrs:resourceClasses	如果要从类名称创建多个资源，则只使用 **jaxrs:server/@serviceClass 属性**(blueprint) 而不是使用 jaxrs:server/@serviceClass 属性。**jaxrs:resourceClasses** 的子项必须是将 **name** 属性设置为资源名称 **的类** 元素。在这种情况下，类由 Apache CXF 进行实例化，**而不是由** Blueprint 或 Spring 进行实例化。

[a] Invoker 实施控制如何调用服务。例如，它控制每个请求是否由服务实施的新实例处理，或在调用中保留状态。

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

在 XML 语言（Blueprint XML 或 Spring XML）中实例化客户端代理的主要点是将其注入另一个 bean，然后使用客户端代理调用 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 客户端端点使用服务器端点 的不同 XML 命名空间来定义。下表显示了用于 XML 语言的命名空间：

XML 语言	客户端端点的命名空间
蓝图（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：

```
<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到 XML 配置中的其他 Bean 中。

address

address 属性指定 REST 调用的基本 URL。

serviceClass

serviceClass 属性指定根资源类（由 @Path无关）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。指定的类仅用于其元数据（通过 Java 反射和 JAX-RS 注释），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 元素中可用的属性。

表 18.3. JAX-RS 客户端端点属性

属性	描述
address	指定消费者发出请求的端点的 HTTP 地址。这个值覆盖合同中设定的值。
bindingId	指定消费者使用的消息绑定 ID。第 23 章 *Apache CXF Binding ID* 中提供了有效绑定 ID 列表。
总线	指定 Spring bean 配置总线管理端点的 ID。
inheritHeaders	如果从此代理创建子资源代理，则指定是否会继承为此代理设置的标头。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/密码验证的密码。
modelRef	将模型 schema 指定为类路径资源（例如，classpath **:/path/to/model.xml**的 URL）。有关如何定义 JAX-RS 模型架构的详细信息，请参阅 第 18.3 节 “使用模型架构定义 REST 服务”。
serviceClass	指定服务接口的名称或资源类（通过 **@PATH**标注），重新利用 JAX-RS 服务器实施。在这种情况下，指定的类 **不会** 直接调用（实际上是一个服务器类）。指定的类仅用于其元数据（通过 Java 反射和 JAX-RS 注释），用于动态构建客户端代理。
serviceName	指定在使用 JMS 传输的特殊情况下，为 JAX-RS 端点指定服务 QName（使用 ***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（替换 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 bean 是否为抽象 bean。abstract Bean 充当拥塞 Bean 定义且没有实例化的父项。默认值为 **false**。将其设置为 **true** 指示 bean 工厂不会实例化 bean。
dependent-on	(Spring only) 指定端点在实例化前需要被实例化的 Bean 列表，然后才能实例化它。

18.2.3. jaxrs:client Child Elements

子元素

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

表 18.4. JAX-RS 客户端端点子元素

元素	描述
jaxrs:executor	
jaxrs:features	指定 Beans 列表，用于配置 Apache CXF 的高级功能。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 定义进行指定。如需了解更多详细信息，请参阅 “指定数据绑定”一节。
jaxrs:inInterceptors	指定进程入站响应的拦截器列表。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxrs:inFaultInterceptors	指定处理入站错误信息的拦截器列表。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。更多信息请参阅 第 VII 部分 “开发 Apache CXF Interceptors”。
jaxrs:outFaultInterceptors	指定处理出站错误信息的拦截器列表。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	使您能够使用此端点注册一个或多个自定义 JAX-RS 提供程序。此元素的子项是（an 元素）的实例或引用（**ref** 元素）JAX-RS 提供程序。
jaxrs:modelBeans	由一个或多个 **org.apache.cxf.jaxrs.model.UserResource** Bean 组成，它们是资源模型的基本元素（与 **jaxrs:resource** 元素）的引用列表。详情请查看 第 18.3 节 “使用模型架构定义 REST 服务”。

元素	描述
jaxrs:model	在此端点中直接定义资源模型（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 “使用模型架构定义 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。
jaxrs:schemaLocations	指定验证 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，各自指定 XML 架构文件的位置（通常是 类路径 URL）。详情请查看 “模式验证”一节。

18.3. 使用模型架构定义 REST 服务

没有注解的 RESTful 服务

借助 JAX-RS 模型模式，可以在不 注解 Java 类的情况下定义 RESTful 服务。也就是说，不使用 @Path、@PathParam、@Consumes、@Consumes、@Produces 等等，直接发送到 Java 类（或接口），您可以提供单独的 XML 文件中的所有相关 REST 元数据。例如，在无法修改实现该服务的 Java 源时，这很有用。

模型 schema 示例

例 18.1 “JAX-RS 模型架构示例” 显示了一个模型架构示例，它定义了 BookStoreNoAnnotations root 资源类的服务元数据。

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 中定义对应的 JAX-RS 端点还是在 Spring XML 中定义。下表显示了用于 XML 语言的命名空间：

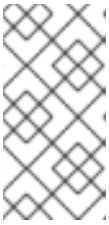
XML 语言	命名空间
蓝图（Blueprint）	http://cxf.apache.org/blueprint/jaxrs
Spring	http://cxf.apache.org/jaxrs

如何将模型模式附加到端点

要在端点上定义和附加模型模式，请执行以下步骤：

1. **使用您选择的注入平台（Blueprint XML 或 Spring XML）的适当 XML 命名空间定义模型架构。**

2. **将模型模式文件添加到您的项目的资源，以便架构文件可在最终软件包（JAR、WAR 或 OSGi 捆绑包文件）上提供。**



注意

另外，还可以使用端点的 jaxrs:model 子元素将模型模式直接嵌入到 JAX-RS 端点中。

3. **将端点配置为使用模型模式，方法是将端点的 modelRef 属性设置为 classpath 上模型架构的位置（使用 classpath URL）。**

4. **如有必要，使用 jaxrs:serviceBeans 元素明确实例化根资源。如果模型模式直接引用根资源类（而不是引用基本接口），您可以跳过这一步。**

引用类的模型架构配置

如果模型架构直接应用到根资源类，则不需要使用 jaxrs:serviceBeans 元素定义任何根资源anan，因为模型架构会自动实例化根资源 Bean。

例如，如果 customer-resources.xml 是一个模型的 schema，它将元数据与客户资源类相关联，您可以按照如下所示 实例化客户服务端点：

```
<jaxrs:server id="customerService"
              address="/customers"
              modelRef="classpath:/org/example/schemas/customer-resources.xml" />
```

配置模型架构引用接口

如果模型架构适用于 Java 接口（它是根资源的基本接口），则必须使用端点中的 jaxrs:serviceBeans 元素实例化根资源类。

例如，如果 customer-interfaces.xml 是一个模型 schema，它将元数据与客户接口相关联，您可以按照如下所示 实例化客户服务端点：

```
<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

模型架构的根元素。如果您需要引用模型模式（例如，使用 modelRef 属性的 JAX-RS 端点中），您应当设置此元素上的 id 属性。

model/resource

资源 元素用于将元数据与特定的根资源类（或对应的接口）关联。您可以在 resource 元素上定义以下属性：

属性	描述 +
name	将这个资源模型应用到的资源类（或对应接口）的名称。 +
path	映射到此资源的 REST URL 路径的组件。 +
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**。 +
produces	指定此资源生成的内容类型（互联网介质类型），如 **application/xml** 或 **application/json**。 +

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	此元素应用到的 Java 方法的名称。 +
path	映射到此方法的 REST URL 路径的组件。此属性值可以包括参数引用，例如： **path="/books/{id}/preceding"**，其中 **{id}** 会从路径中提取 **id** 参数的值。 +
verb	指定映射到此方法的 HTTP 动词。通常，: **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**，它会返回一个对子资源对象的引用（其中子资源类还必须使用 **资源元素提供元数据** ）。 +
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**。 +

属性	描述 +
produces	指定此操作生成的内容类型（互联网介质类型），如 **application/xml** 或 **application/json**。 +
Oneway	如果为 **true**，将操作配置为 **单向**，即不需要回复信息。默认值为 **false**。 +

model/resource/operation/param

param 元素用于从 REST URL 中提取值，并将其注入到其中一个方法参数中。您可以在 param 元素上定义以下属性：

属性	描述 +
name	此元素应用到的 Java 方法参数的名称。 +
type	指定从 REST URL 或消息中提取参数值的方式。可以将其设置为以下值之一： **PATH**、**QUERY**、**MATRIX**、HEAHER、**COOKIE**、**FORM**、**CONTEXT**、**REQUEST_BODY**。 **+**
defaultValue	要注入到参数的默认值，因为无法从 REST URL 或消息中提取值。 +

属性	描述 +
编码	如果为 **true**，则参数值以其 URI 编码形式注入（即，使用 **%n** 编码代码）。默认为 **false**。例如，当从 URL 路径中提取参数时，**/name/Joe%20Bloggs** 将被编码设为 **true**，该参数会作为 **Joe%20Bloggs** 注入到；否则，参数将作为 **Joe Bloggs** 注入。 +

第 19 章 APACHE CXF LOGGING

摘要

本章论述了如何在 Apache CXF 运行时配置日志。

19.1. APACHE CXF 日志概述

概述

Apache CXF 使用 Java 日志记录实用程序 java.util.logging。日志记录配置在一个日志配置文件中，该文件使用标准的 java.util.Properties 格式编写。要在应用上运行日志记录，您可以以编程方式指定日志记录，或者在启动应用程序时在命令中定义指向日志记录配置文件的属性。

默认属性文件

Apache CXF 附带默认的 logging.properties 文件，该文件位于您的 *InstallDir*/etc 目录中。此文件为日志消息和发布的消息级别配置输出目的地。默认配置会将日志记录器设置为打印带有 WARNING 级别标记的消息到控制台。您可以使用默认文件而无需更改任何配置设置，也可以更改配置设置以适应特定的应用程序。

日志记录功能

Apache CXF 包括日志记录功能，可插入您的客户端或您的服务以启用日志记录。例 19.1 “配置来启用日志记录” 显示启用日志记录功能的配置。

例 19.1. 配置来启用日志记录

```
<jaxws:endpoint...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

更多信息请参阅 第 19.6 节 “记录消息内容”。

从哪里开始？

要运行一个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” 中介绍的说明。

如需有关在 Apache CXF 中登录的更多信息，请阅读整个章节。

有关 java.util.logging 的更多信息

java.util.logging 实用程序是最常用的 Java 日志框架之一。在线提供了很多可用的信息，用于描述如何使用和扩展此框架。但作为起点，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 示例应用程序中日志消息的日志级别和输出目的地，请完成以下步骤：

1. 使用 *InstallDir*/samples/wsdl_first 目录中的 README.txt 文件的 java 部分，运行 demo 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** +
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/logging.properties demo.hw.server.Server &** +

默认的 logging.properties 文件位于 *InstallDir*/etc 目录中。它将 Apache CXF loggers 配置为向控制台打印 WARNING 级别日志消息。因此，您会看到到控制台的很少输出。

2. 如 README.txt 文件中所述，停止服务器。

3. 复制默认的 logging.properties 文件，将其命名为 mylogging.properties 文件，并将它保存为与默认 logging.properties 文件相同的目录中。

4. 通过编辑以下配置行，将 mylogging.properties 文件中的控制台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** +
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &** +

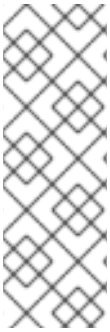
由于您已将全局日志和控制台日志记录器配置为记录级别 INFO 的消息，因此您会看到更多日志消息打印到控制台。

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 loggers

配置为向控制台打印 WARNING 级别的消息。如果此级别的日志记录适用于您的应用程序，在使用前不必对该文件进行任何更改。但是，您可以在日志消息中更改详细级别。例如，您可以把日志消息发送到控制台，或更改为文件或两者。另外，您可以在独立软件包的级别上指定日志记录。



注意

本节讨论在默认的 logging.properties 文件中显示的配置属性。然而，还有许多其他 java.util.logging 配置属性可以设置。有关 java.util.logging API 的更多信息，请参见 java.util.logging javadoc，网址为：http://download.oracle.com/javase/1.5/docs/api/java/util/logging/package-summary.html。

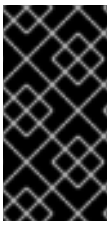
19.3.2. 配置日志输出

概述

Java 日志记录实用程序 java.util.logging 使用处理程序类输出日志消息。表 19.1 “Java.util.logging Handler Classes” 显示在默认的 logging.properties 文件中配置的处理程序。

表 19.1. Java.util.logging Handler Classes

处理器类	输出到
ConsoleHandler	将日志信息输出到控制台
FileHandler	输出日志消息到文件中



重要

处理程序类必须在系统类路径上，才能在启动时由 Java 虚拟机安装。设置 Apache CXF 环境时会完成此操作。

配置控制台处理程序

例 19.2 “配置控制台处理程序” 显示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
```

控制台处理程序也支持 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” 中显示的配置属性可以解释如下：

控制台处理程序支持单独的日志级别配置属性。这可让您将打印的日志信息限制为控制台，而全局日志记录设置可能会不同（请参阅 第 19.3.3 节 “配置日志记录级别”）。默认设置为 **WARNING**。

指定 **console handler** 类用于格式化日志消息的 **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
```

文件处理程序也支持 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” 中显示的配置属性可以解释如下：

指定输出文件的位置和模式。默认设置为您的主目录。

以字节为单位，日志记录器写入任何一个文件的最大数量。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。

指定循环要循环多少个输出文件。默认设置为 1。

指定文件处理器类格式化日志消息的 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程序和文件处理程序（以逗号分隔）来设置日志记录工具，以将日志消息输出到控制台和文件处理程序，如 配置两个控制台日志和文件 所示。

配置两个控制台日志和文件

```
Logging
```

```
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

19.3.3. 配置日志记录级别

日志级别

java.util.logging 框架支持以下日志级别，从最小到详细值：

- 严重
- WARNING

- INFO
- CONFIG
- FINE
- FINER
- FINEST

配置全局日志记录级别

要配置在所有日志记录器中记录的事件类型，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
```

在独立软件包中配置日志记录

level

java.util.logging 框架支持在单个软件包的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 软件包中的类中配置 SEVERE 级别的日志。

例 19.7. 在软件包级别配置日志记录

```
com.xyz.foo.level = SEVERE
```

19.4. 通过命令行启用日志记录

概述

您可以在启动应用程序时定义 java.util.logging.config.file 属性在应用程序上运行日志记录实用程

序。您可以指定默认的 logging.properties 文件，或者指定应用程序独有的 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，请在启动应用程序时添加 例 19.8 “在命令行上启动日志记录的标记” 中显示的标记。

例 19.8. 在命令行上启动日志记录的标记

```
-Djava.util.logging.config.file=myfile
```

19.5. 为 SUBSYSTEM 和 SERVICES 日志记录

概述

您可以使用 “在独立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 日志记录子系统列表。

表 19.2. Apache CXF Logging 子系统

子系统	描述
org.apache.cxf.aegis	Aegis binding
org.apache.cxf.binding.coloc	colocated binding
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
org.apache.cxf.binding.xml	XML 绑定

子系统	描述
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	用于 JAX-WS 风格的消息交换的前端、JAX-WS 处理程序处理以及与 JAX-WS 和配置相关的拦截器
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 保护 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	使用本地文件系统的传输实现
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-寻址实施
org.apache.cxf.ws.policy	ws-Policy 的实现
org.apache.cxf.ws.rm	ws-ReliableMessaging(WS-RM)实施
org.apache.cxf.ws.security.wss4j	WSS4J 安全实施

示例

WS-Addressing 示例包含在 *InstallDir*/samples/ws_addressing 目录中。日志记录在位于该目录中的 logging.properties 文件中配置。配置的相关行显示在 例 19.9 “为 WS-Addressing 配置日志记录” 中。

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” 中的配置启用跳过与 WS-Addressing 标头相关的日志消息，并以简洁的形式将它们显示到控制台中。

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. 记录消息内容

概述

您可以记录在服务和消费者之间发送的消息内容。例如，您可能要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

要记录在服务和消费者之间发送的信息，反之亦然，请完成以下步骤：

1. 将日志记录功能添加到您的端点配置中。
2. 将日志记录功能添加到您的消费者的配置。
3. 配置日志记录系统日志 INFO 级别信息。

将日志记录功能添加到端点

添加日志记录功能，您的端点的配置功能，如 例 19.10 “在端点配置中添加日志” 所示。

例 19.10. 在端点配置中添加日志

```
<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” 中显示的 XML 示例启用 SOAP 消息的日志记录。

在消费者中添加日志记录功能

添加日志记录功能，为您的客户端配置添加功能，如 例 19.11 “在客户端配置中添加日志” 所示。

例 19.11. 在客户端配置中添加日志

```
<jaxws:client ...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:client>
```

例 19.11 “在客户端配置中添加日志” 中显示的 XML 示例启用 SOAP 消息的日志记录。

将 logging 设置为 log INFO 级别信息

确保与您的服务关联的 logging.properties 文件配置为记录 INFO 级别的信息，如 例 19.12 “将日志级别设置为 INFO” 所示。

例 19.12. 将日志级别设置为 INFO

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

记录 SOAP 信息

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 “日志 SOAP 消息的端点配置” 中显示的 jaxws:features 元素添加到 wsdl_first 示例目录中的 cxf.xml 配置文件：

例 19.13. 日志 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. 这个示例使用默认的 **logging.properties** 文件，该文件位于 ***InstallDir*/etc** 目录中。制作此文件的副本，并将其命名为 **mylogging.properties**。

3. 在 **mylogging.properties** 文件中，通过编辑 **.level** 和 **java.util.logging.ConsoleHandler.level** 配置属性将日志记录级别更改为 **INFO** ：

```
.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** +
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &** +

5. 使用以下命令启动 **hello world** 客户端：

平台	命令 +

平台	命令 +
Windows	**java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.client.Client .\wsdl\hello_world.wsdl** +
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.client.Client ./wsdl/hello_world.wsdl** +

SOAP 信息被记录到控制台。

第 20 章 部署 WS-ADDRESSING

摘要

Apache CXF 支持 WS-WS 应用程序寻址。本章论述了如何在 Apache CXF 运行时环境中部署 WS-Addressing。

20.1. WS-ADDRESSING 简介

概述

WS-寻址是一个规范，它允许服务以传输中立的方式沟通寻址信息。它由两个部分组成：

- 用于将引用发送到 Web 服务端点的结构
- 组消息地址属性(MAP)，用于将寻址信息与特定消息关联

支持的规格

Apache CXF 支持 WS-Addressing 2004/08 规格和 WS-Addressing 2005/03 规范。

更多信息

有关 WS-Addressing 的详情，请参考 2004/08 提交地址 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 运行时使用拦截器截获并使用正在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，则会处理消息中包含的任何 WS-Addressing 信息。

WS-Addressing Interceptors

WS-Addressing 实施由两个拦截器组成，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	负责聚合用于传入消息地址属性(MAPs)的逻辑拦截器。
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 拦截器，必须将 WS-Addressing 拦截器添加到入站和出站拦截器链中。这可以通过以下之一完成：

- Apache CXF 功能
- RMAssertion 和 WS-Policy Framework
- 在 WS-Addressing 功能中使用 Policy Assertion

将 WS-Addressing 添加为功能

通过在客户端和服务器配置中添加 WS-Addressing 功能来启用 WS-Addressing，如 例 20.1 “client.xml 并为客户端配置添加 WS-Addressing 功能” 和 例 20.2 “server.xml 并添加到服务器配置的 WS-Addressing 功能” 所示。

例 20.1. client.xml 并为客户端配置添加 WS-Addressing 功能

```
<?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 并添加到服务器配置的 WS-Addressing 功能

```
<?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” 描述的两个属性。

表 20.2. WS-Addressing 属性

属性名称	值
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 中是否存在 **使用Addressing** 元素的布尔值 ; 也就是说，它不会阻止 WS-Addressing 标头的编码。

配置 WS-Addressing 属性

通过添加属性以及您想要将其设置为服务器或者客户端配置文件中的 WS-Addressing 功能来配置 WS-Addressing 属性。例如，以下配置提取服务器端点上的 allowDuplicates 属性设为 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，寻址策略断言，以启用非匿名响应，并嵌入到 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 启用可靠消息

摘要

Apache CXF 支持 WS-可靠消息传递(WS-RM)。本章论述了如何在 Apache CXF 中启用和配置 WS-RM。

21.1. WS-RM 简介

概述

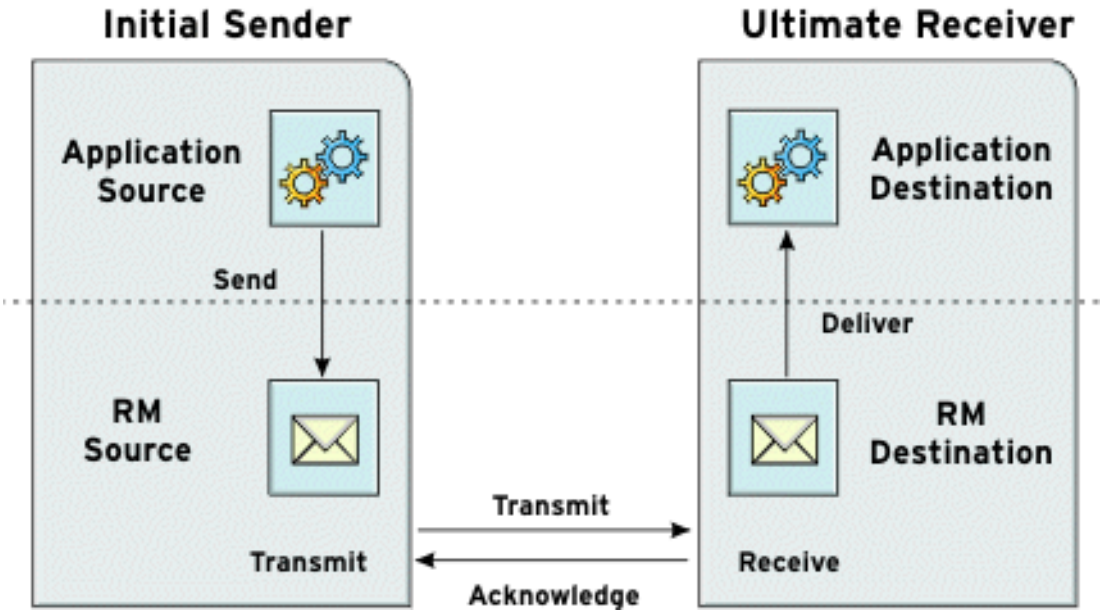
WS-ReliableMessaging(WS-RM)是一种保证在分布式环境中可靠地传输消息的协议。它可让消息在软件、系统或网络故障的分布式应用程序之间可靠地交付。

例如，可以使用 WS-RM 来确保在网络上以正确方式传送正确的消息。

WS-RM 的工作原理

WS-RM 确保在来源和目的地端点之间可靠传递消息。源是消息的初始发送者，目的地是最终接收器，如 图 21.1 “Web 服务可靠消息” 所示。

图 21.1. Web 服务可靠消息



可以描述 WS-RM 消息的流，如下所示：

1.

RM 源会向 RM 目标发送 CreateSequence 协议消息。它包含接收确认的端点的引用（wsrm:AcksTo 端点）。

2. RM 目标将 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 源为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 和唯一的消息 ID。

4. RM 源会将每个消息传输到 RM 目的地。

5. RM 目标通过发送包含 RM SequenceAcknowledgement 标头的消息，确认来自 RM 源的收据。

6. RM 目的地以精确的方式向应用目的地传递消息。

7. RM 源重新传输一个尚未收到确认的消息。

第一个重新传输尝试是在基本重新传输间隔后进行的。默认情况下，以 exponential-off 间隔或以固定间隔形式进行重新传输尝试。如需了解更多详细信息，请参阅 第 21.5 节 “配置 WS-RM”。

整个过程会针对请求和响应消息进行对称进行的。也就是说，当响应消息时，服务器充当 RM 源，客户端充当 RM 目的地。

WS-RM 交付保证

WS-RM 可保证分布式环境中可靠的消息发送，无论所使用的传输协议如何。如果无法进行可靠交付，则源或目标端点会记录错误。

支持的规格

Apache CXF 支持 WS-RM 规范的以下版本：

WS-ReliableMessaging 1.0

（默认） Correspon ds 到 2005 年 2 月提交版本，现已过期。然而，出于向后兼容性的原因，这个版本被用作默认值。

WS-RM 版本 1.0 使用以下命名空间：

> http://schemas.xmlsoap.org/ws/2005/02/rm/

此版本的 WS-RM 可与以下 WS-Addressing 版本一起使用：

> http://schemas.xmlsoap.org/ws/2004/08/addressing (default)
> http://www.w3.org/2005/08/addressing

为了遵守 2005 年 2 月版本的 WS-RM，您很难使用第一个 WS-Addressing 版本（这是 Apache CXF 中的默认设置）。但大多数其他 Web 服务实施已切换到较新的 WS-Addressing 规格，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 服务可靠消息 规范。

WS-RM 版本 1.1 和 1.2 版本使用以下命名空间：

> 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

在供应商方面，Apache CXF 适应了客户端使用哪个版本的 WS-回复和相应的响应。

客户端

在客户端客户端上，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”），或在运行时覆盖 WS-RM 版本，使用运行时控制选项（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 功能作为拦截器实施。Apache CXF 运行时使用拦截器截获并使用正在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。如果应用程序拦截器链包含 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，在 表 21.1 “Apache CXF WS-ReliableMessaging Interceptors” 中描述。

表 21.1. Apache CXF WS-ReliableMessaging Interceptors

拦截器	描述
org.apache.cxf.ws.rm.RMOutInterceptor	涉及为外发消息提供可靠性保证的逻辑方面。 负责发送 **CreateSequence** 请求并等待它们的 **CreateSequenceResponse** 响应。 此外，还负责聚合序列属性-ID 和消息编号（用于应用消息）。
org.apache.cxf.ws.rm.RMInInterceptor	负责截获和处理 RM 协议消息和 **序列** 消息，这些消息由应用程序消息提供支持。
org.apache.cxf.ws.rm.RMCaptureInInterceptor	为持久性存储缓存传入的信息。
org.apache.cxf.ws.rm.RMDeliveryInterceptor	有助于向应用程序传输消息。
org.apache.cxf.ws.rm.soap.RMSoapInterceptor	负责将可靠性属性编码并解码为 SOAP 标头。
org.apache.cxf.ws.rm.RetransmissionInterceptor	负责创建应用程序消息的副本以备将来重新发送。

启用 WS-RM

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，在出站拦截器链中截获第一个应用程序消息时，RMOutInterceptor 会发送 CreateSequenceptor 请求，并等待处理原始应用程序消息，直到它收到 CreateSequence Response 响应。此外，WS-RM 拦截器将序列标头添加到应用程序消息中，并在目的地一侧从消息中提取它们。不需要对您的应用程序代码进行任何更改，以便可靠地交换消息。

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，您可以控制操作序列以及可靠交换的其他方面。例如，默认情况下，Apache CXF 尝试最大限度地提高序列生命周期，从而减少了带外 WS-RM 协议消息的开销。要强制每个应用程序消息使用单独的序列，请配置 WS-RM 源序列终止策略（将最大序列长度设置为 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，WS-RM 拦截器必须添加到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，所以 WS-Addressing 拦截器也必须存在于拦截器链中。

您可以通过以下两种方式之一来确保存在这些拦截器：

- 通过使用 Spring Bean，把它们添加到 assign 链中，以 显式。
- 隐式，使用 WS-Policy 断言，导致 Apache CXF 运行时代表您透明地添加拦截器。

Spring Bean：显式添加拦截器

要启用 WS-RM 和 WS-Addressing 拦截器到 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 *InstallDir*/samples/ws_rm 目录中的 WS-RM 示例中采用的方法。配置文件

ws-rm.cxf 显示了 WS-RM 和 WS-Addressing 拦截器作为 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” 中显示的代码可以按照以下方式解释：

Apache CXF 配置文件是一个 Spring XML 文件。您必须包含一个打开的 Spring Bean 元素，用于声明由 Bean 元素封装的子元素的命名空间和架构文件。

配置每个 WS-Addressing interceptors-MAPAggregator 和 MAPCodec。有关 WS-Addressing 的详情，请参考 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 拦截器添加到入站故障的拦截器链中。

将 WS-Addressing 和 WS-RM 拦截器添加到出站消息的拦截器链中。

将 WS-Addressing 和 WS-RM 拦截器添加到出站故障的拦截器链中。

WS-Policy 框架：隐式添加拦截器

WS-Policy 框架提供了您可使用 WS-Policy 的基础架构和 API。它符合 2006 年 11 月发布的 Web 服务策略 1.5-Framework 和 Web 服务策略 1.5-Attachment 规格。

要使用 Apache CXF WS-Policy 框架启用 WS-RM，请执行以下操作：

1. 在您的客户端和服务器端点中添加策略功能。例 21.2 “使用 WS-Policy 配置 WS-RM” 显示在 jaxws:feature 元素内嵌套的参考。referencean 指定 AddressingPolicy，它定义为同一配置文件内的独立元素。

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，它们可用作策略或策略参考元素附加到 WSDL 文件的附件点，如 例 21.3 “在您的 WSDL 文件中添加 RM 策略” 所示。**

例 21.3. 在您的 WSDL 文件中添加 RM 策略

```
<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多个消息上下文属性值，以便在运行时控制 WS-RM，且由 org.apache.cxf.rm.rm.RMManager 类中的公共常量定义的关键值。

运行时控制选项

下表列出了 org.apache.cxf.ws.rm.RMManager 类定义的密钥。

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）- 除非您使用 http://schemas.xmlsoap.org/ws/2005/02/rm/ RM 命名空间，否则此属性将被忽略。
WSRM_LAST_MESSAGE_PROPERTY	布尔值为 **true**，用于告知 WS-RM 代码是要发送的最后一个消息，允许代码关闭 WS-RM 序列和释放资源（自 CXF 的 3.0.0 版本起，WS-RM 默认关闭 RM 序列）。
WSRM_INACTIVITY_TIMEOUT_PROPERTY	长期不活跃超时（以毫秒为单位）。
WSRM_RETRANSMISSION_INTERVAL_PROPERTY	以毫秒为单位进行较长的基本重新传输间隔。
WSRM_EXPONENTIAL_BACKOFF_PROPERTY	布尔值 exponential back-off 标记。
WSRM_ACKNOWLEDGEMENT_INTERVAL_PROPERTY	长期确认间隔（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 的许多方面。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当前的 RM 状态到各个消息级别。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 端点确认时发出的通知。

JMX 控制示例

例如，如果您的客户端配置中启用了 JMX 服务器，您可以使用以下代码跟踪接收的最后确认号：

```
// 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 特定 WS-RM 属性

概述

要配置特定于 Apache CXF 的属性，请使用 rmManager Spring bean。在您的配置文件中添加以下内容：

- http://cxf.apache.org/ws/rm/manager 命名空间到您的命名空间列表。
- 适用于您要配置的特定属性的 rmManager Spring bean。

例 21.4 “配置 Apache CXF 特定 WS-RM 属性” 显示了一个简单的示例。

例 21.4. 配置 Apache CXF 特定 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
RMAsertion	类型为 RMA **sertion 的元素**
deliveryAssurance	Type **DeliveryAsuranceType** 的一个元素，用于描述应应用的交付保证
sourcePolicy	允许您配置 RM 源详情的 **SourcePolicyType** 的一个元素
destinationPolicy	允许您配置 RM 目的地详情的 **DestinationPolicyType** 的一个元素

示例

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 RMAsertion Children

表 21.3 “WS-Policy RMAsertion Element 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 Element 的子项

名称	描述
InactivityTimeout	指定在端点可以视为因为不活跃而终止的 RM 序列前必须收到消息的时长。
BaseRetransmissionInterval	设置确认在给定消息中的 RM Source 必须接收的时间间隔。如果在 **BaseRetransmissionInterval** 设置的时间内未收到确认，则 RM Source 将重新传输该消息。
ExponentialBackoff	指示将使用已知指数级 backoff 算法(Tanenbaum)调整重新传输间隔。 如需更多信息，请参阅 **Computer Networks**、Andrew S. Tanenbaum、Prentice Hall PTR, 2003。
AcknowledgementInterval	在 WS-RM 中，确认信息会在返回消息上发送或独立发送。如果无法发送确认消息的返回消息，则 RM Destination 可在发送独立确认前等待确认间隔。如果没有未确认的消息，RM Destination 可以选择不发送确认消息。

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 RMA sertion 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，则这是最好的方法。也就是说，如果要配置同一文件中特定于 Apache CXF 的属性和标准 WS-RM 策略属性。

例如，例 21.5 “使用 rmManager Spring Bean 中的 RMAsertion 配置 WS-RM 属性” 中的配置会显示：

- 标准 WS-RM 策略属性 BaseRetransmissionInterval 使用 rmManager Spring bean 中的 RMAsertion 配置。
- 在同一配置文件中配置 Apache CXF 特定 RM 属性 intraMessageThreshold。

例 21.5. 使用 rmManager Spring Bean 中的 RMAsertion 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 “在功能内将 WS-RM 属性配置为策略” 所示。

例 21.6. 在功能内将 WS-RM 属性配置为策略

```
<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。如果您希望服务互操作并使用 WS-RM 与部署到其他策略感知型 Web 服务堆栈的用户，最好采用这种方法。

例如，请参阅 “WS-Policy 框架：隐式添加拦截器”一节，其中在 WSDL 文件中配置了基本重新传输间隔。

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。如果您无法更改 WSDL 文件，这是个好方法。

例 21.7 “在外部附加中配置 WS-RM” 显示为特定 EPR 启用 WS-A 和 WS-RM（基本传输间隔为 30 秒）的外部附加。

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

本小节着重介绍从用例的角度配置 WS-RM 属性。其中属性是一个标准的 WS-RM 策略属性（在 http://schemas.xmlsoap.org/ws/2005/02/rm/policy/ 命名空间中定义），只会显示在 rmManager Spring bean 中的 RMAsertion 中设置它的示例。有关如何在功能内设置这样的属性（如策略）的详情；在 WSDL 文件中或外部附件中，请参阅 第 21.5.2 节 “配置标准 WS-RM 策略属性”。

涵盖以下用例：

- “基本重新传输间隔”一节
- “重新传输的 backoff”一节
- “确认的时间间隔”一节
- “最大未确认的消息阈值”一节
- “RM 序列的最大长度”一节

- “消息发送保障策略”一节

基本重新传输间隔

BaseRetransmissionInterval 元素指定在 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 基本重新传输间隔。

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 backoff

ExponentialBackoff 元素确定是否以指数级间隔执行对未确认消息的连续性重新传输尝试。

ExponentialBackoff 元素将启用此功能。默认情况下会使用 2 的 exponential backoff 比例。ExponentialBackoff 是一个标志。当元素存在时，会启用 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 Property” 演示了如何为重新传输设置 WS-RM exponential backoff。

例 21.9. 设置 WS-RM Exponential Backoff Property

```
<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>
```

确认的时间间隔

AcknowledgementInterval 元素指定 WS-RM 目标发送异步确认的时间间隔。除了同步确认外，还需要接收传入的消息。默认的异步确认间隔为 0 毫秒。这意味着，如果 AcknowledgementInterval 没有配置为特定值，确认将立即发送（即首次可用的商机）。

只有在满足以下条件时，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。
- 在确认间隔到期前不会发生对响应消息的确认机会。

例 21.10 “设置 WS-RM Acknowledgement Interval” 显示如何设置 WS-RM 确认间隔。

例 21.10. 设置 WS-RM Acknowledgement 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:AcknowledgementInterval Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

最大未确认的消息阈值

maxUnacknowledged 属性设置在序列被终止前每个序列的最大未确认消息数。

例 21.11 “设置 WS-RM Maximum Unacknowledged Message Threshold” 展示如何设置 WS-RM 的最大消息阈值。

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，这表示 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，并在收到之前发送的消息的所有确认后。使用 newsequence 发送新消息。

例 21.12 “设置 WS-RM 消息序列的最大长度” 演示了如何设置 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>
```

消息发送保障策略

您可以将 RM 目的地配置为使用以下发送保证策略：

- **AtMostOnce - RM 目标仅向应用程序目的地发送消息一次。如果发生错误后传送的信息超过。序列中的一些消息可能无法发送。**
- **AtLeastOnce - RM 目的地至少向应用目的地发送消息。发送的每个消息都会被发送，否则将引发错误。有些信息可能会多次传送。**
-

InOrder - RM 目标以发送顺序向应用程序目的地发送消息。这种交付保证可与 AtMostOnce 或 AtLeastOnce 保障相结合。

例 21.13 “设置 WS-RM Message Delivery Assurance 策略” 展示如何设置 WS-RM 消息发送保证。

例 21.13. 设置 WS-RM Message Delivery Assurance 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（如网络故障）提供了可靠性。WS-RM 持久性提供其它类型故障的可靠性，如 RM 源或 RM 目标崩溃。

WS-RM 持久性涉及将各种 RM 端点的状态存储在持久存储中。这可让端点在被重试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 持久存储基于 JDBC。为方便起见，Apache CXF 包括 Derby 用于开箱即用的部署。此外，持久存储也使用 Java API 进行公开。要实现自己的持久性机制，您可以使用这个 API 与首选 DB 来实施。

重要

WS-RM 持久性只支持单向调用，默认是禁用的。

如何使用

Apache CXF WS-RM 持久性的工作方式如下：

- 在 RM 源端点中，传出的消息会在传输前保留。在收到确认后，它将从持久性存储中驱除。
- 在从崩溃中恢复后，它会恢复永久性的消息并重新传输，直到所有消息都已确认。此时，RM 序列是关闭的。
- 在 RM 目标端点中，传入的信息会被保留，并在成功存储后发送确认。当成功分配消息时，它将从持久性存储中驱除。
- 在从崩溃中恢复后，它会恢复保留的消息并分配它们。它还会使 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，您必须指定为 WS-RM 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” 中显示的配置可启用基于 JDBC 的存储，该存储带有 Apache CXF。

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

属性名称	类型	默认设置
passWord	**字符串**	null
url	**字符串**	jdbc:derby:rmdb;create=true

例 21.15 “为 WS-RM Persistence 配置 JDBC 存储” 中显示的配置可启用与 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 Persistence 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

本章论述了如何在 Apache CXF 运行时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来避免分布式系统中的单点故障。您可以使用 来保护您的系统不受单一故障点 *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。两者结合为一个逻辑服务。客户端调用复制服务上的请求，而 Apache CXF 则向其中一个成员副本提供请求。到副本的路由对客户端是透明的。

具有静态故障转移的 HA

Apache CXF 支持使用静态故障转移的高可用性(HA)，其中副本详情是在服务 WSDL 文件中编码的。WSDL 文件包含多个端口，可以包含多个主机，可以针对同一服务包含多个主机。只要 WSDL 文件保持不变，集群中的副本数会一直保持静态。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

要使用静态故障转移启用 HA，您必须执行以下操作：

1. “在您的服务 WSDL 文件中编码副本详情”一节
2. “在您的客户端配置中添加集群功能”一节

在您的服务 WSDL 文件中编码副本详情

您必须在服务 WSDL 文件中对集群中的副本进行编码。例 22.1 “使用静态故障转移启用 HA：WSDL 文件” 显示 WSDL 文件提取，以定义三个副本的服务集群。

例 22.1. 使用静态故障转移启用 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 “使用静态故障转移启用 HA：WSDL 文件” 中显示的 WSDL 提取可以解释如下：

定义服务 **ClusterService**，它在三个端口上公开：

1. **Replica1**
2. **Replica2**
3. **Replica3**

定义 **Replica1**，以通过端口 **9001** 上的 **HTTP** 端点公开 **ClusterService** 作为 **SOAP**。

定义 **Replica2**，以在端口 **9002** 上通过 **HTTP** 端点公开 **ClusterService** 作为 **SOAP**。

定义 **Replica3**，以在端口 **9003** 上通过 **HTTP** 端点公开 **ClusterService** 作为 **SOAP**。

在您的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 “使用静态故障转移启用 HA：客户端配置” 所示。

例 22.2. 使用静态故障转移启用 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，如果与客户端通信不可用的原始服务不可用，则带有静态故障转移的 HA 会在选择副本服务时使用后续策略。后续策略在每次使用时都会以连续顺序选择一个副本服务。选择由 Apache CXF 的内部服务模型决定，生成确定性故障转移模式。

配置随机策略

在选择副本时，您可以使用静态故障转移配置 HA 来使用随机策略，而不是后续策略。random 策略在每次服务不可用时选择一个随机副本服务，或者出现故障。从集群中的surviv 成员中故障转移目标选择完全是随机的。

要配置随机策略，请将 例 22.3 “为静态故障转移配置 Random 策略” 中显示的配置添加到您的客户端配置文件中。

例 22.3. 为静态故障转移配置 Random 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 “为静态故障转移配置 Random 策略” 中显示的配置可以解释如下：

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 BINDING ID

绑定 ID 表

表 23.1. Message Bindings 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

为大项目手动创建捆绑包或捆绑包集合可能非常繁琐。Maven bundle 插件使作业通过自动化过程，并提供多个快捷方式来指定捆绑包清单的内容，从而更容易工作。

A.1. MAVEN BUNDLE PLUG-IN

红帽 Fuse OSGi 工具使用 Apache Felix 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的内容，实现 OSGi 捆绑清单的构造。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。该插件也具有用于捆绑清单中其他必要属性的默认值。

要使用 bundle 插件，请执行以下操作：

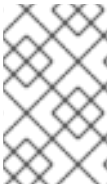
1. 第 A.2 节 “设置红帽 Fuse OSGi 项目” 捆绑包插件到您的项目的 POM 文件。
2. 第 A.3 节 “配置捆绑插件” 该插件用于正确填充捆绑包的清单。

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级别的项目。它不需要任何子项目。但是，它需要您执行以下操作：

1. 将 捆绑包插件添加到您的 POM。
2. 指示 Maven 将结果打包为 OSGi 捆绑包。



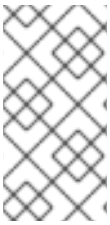
注意

您可以使用几个 Maven archetypes 来设置项目及适当的设置。

目录结构

创建 OSGi 捆绑包的项目可以是单一级别的项目。它只需要有一个顶层 POM 文件和一个 src 文件夹。与所有 Maven 项目中一样，您要将所有 Java 源代码放在 src/java 文件夹中，并且您将任何非 Java 资源放到 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 配置的红帽 Fuse OSGi 项目还包括一个 Bean，它还包含 src/resources/META-INF/spring 文件夹中的 Bean.xml 文件。

添加 bundle 插件

在使用捆绑插件前，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。

例 A.1 “将 OSGi 捆绑包插件添加到 POM” 显示将捆绑包插件添加到您的项目所需的 POM 条目。

例 A.1. 将 OSGi 捆绑包插件添加到 POM

```
...
<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 捆绑包插件添加到 POM” 中的条目执行以下操作：

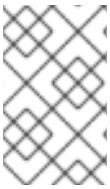
添加 Apache Felix 的依赖

将 bundle 插件添加到项目中

配置插件以使用项目的工件 ID 作为捆绑包的符号链接

配置插件以包含捆绑类导入的所有 Java 软件包；还可导入 org.apache.camel.osgi 软件包

配置插件以捆绑列出的类，但不将其包含在导出软件包列表中



注意

编辑配置以满足您的项目要求。

有关配置捆绑插件的详情请参考 第 A.3 节 “配置捆绑插件”。

激活捆绑包插件

要让 Maven 使用 bundle 插件，请指示它将项目的结果打包为捆绑包。通过将 POM 文件的 打包 元素设置为 bundle 来完成此操作。

有用的 Maven archetypes

有几种 Maven archetypes 可以生成预配置的项目来使用 bundle 插件：

- “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 创建通用项目以使用 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 CXF wsdl-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 创建了一个项目，用于构建部署到 Red Hat 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. 配置捆绑插件

概述

捆绑包插件需要很少的信息才能正常工作。所有所需属性都使用默认设置来生成有效的 OSGi 捆绑包。

虽然您可以使用默认值创建有效的捆绑包，但您可能需要修改某些值。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有：

- **Bundle-SymbolicName**

- bundle-Name
- bundle-Version
- export-Package
- private-Package
- import-Package

设置捆绑包的符号链接

默认情况下，Bundle 插件将 Bundle-SymbolicName 属性的值设置为 *groupId* + "." + *artifactId*，其例外情况如下：

- 如果 *groupId* 只有一个部分（无点），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，则捆绑包的符号链接是 org.apache.commons.logging。

- 如果 *artifactId* 等于 *groupId* 的最后部分，则使用 *groupId*。

  例如，如果 POM 指定组 ID 和工件 ID 作为 org.apache.maven:maven，则捆绑包的符号链接名为 org.apache.maven。

- 如果 *artifactId* 从 *groupId* 的最后部分开始，该部分会被删除。

  例如，如果 POM 指定组 ID 和工件 ID 作为 org.apache.maven:maven-core，则捆绑包的符号链接名为 org.apache.maven.core。

要为捆绑包符号名称指定您自己的值，在插件的 instructions 元素中添加 Bundle-SymbolicName 子，如 例 A.2 “设置捆绑包的符号链接” 所示。

例 A.2. 设置捆绑包的符号链接

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

设置捆绑包的名称

默认情况下，捆绑包的名称设置为 **${project.name}**。

要为捆绑包名称指定您自己的值，在插件的 **instructions** 元素中添加 **Bundle-Name** 子部分，如 例 A.3 “设置捆绑包的名称” 所示。

例 A.3. 设置捆绑包的名称

```
<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}**。所有横线(-)替换为句点(.)，数字最多可为四位数。例如，**4.2-SNAPSHOT** 变为 **4.2.0.SNAPSHOT**。

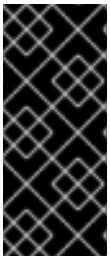
要为捆绑包版本指定您自己的值，将 **Bundle-Version** 子添加到插件的 **instructions** 元素中，如 例 A.4 “设置捆绑包的版本” 所示。

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，OSOS 清单的 Export-Package 列表由您的本地 Java 源代码中的所有软件包（在 src/main/java 下）填充（在 src/main/java下），但 默认软件包. 以及包含 .impl 或 .internal 的软件包。



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的 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非常大的软件包，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 child。

Export-Package 元素指定要包含在捆绑包中的软件包列表，并要导出。可以使用 * 通配符符号指定软件包名称。例如，条目 com.fuse.demo.* 包含项目类路径上以 com.fuse.demo 开头的所有软件包。

您可以指定要排除的软件包，用 ! 前缀。例如，条目 !com.fuse.demo.private 排除软件包 com.fuse.demo.private。

当排除软件包时，列表中的条目顺序非常重要。这个列表会根据开始的顺序处理，并忽略后续迭代条目。

例如，要包含以 com.fuse.demo 开头的所有软件包，除了软件包 com.fuse.demo.private 外，请使用以下内容列出软件包：

```
!com.fuse.demo.private,com.fuse.demo.*
```

但是，如果您使用 com.fuse.demo.*,!com.fuse.demo.private 列出软件包，那么捆绑包中包含 com.fuse.demo.private，因为它与第一个模式匹配。

指定私有软件包

如果要在导出捆绑包中指定要包含在捆绑包中的软件包列表，您可以在捆绑包插件配置中添加 Private-Package 指令。默认情况下，如果您没有指定 Private-Package 指令，则捆绑包中包含您本地 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。软件包会添加到捆绑包中并导出。

Private-Package 元素的工作方式与 导出-Package 元素类似，您会指定要包含在捆绑包中的软件包列表。bundle 插件使用列表来查找要包含在捆绑包中的项目的类路径上的所有类。这些软件包打包在捆绑包中，但不导出（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，Bundle 插件使用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，但您可能会发现您要导入不会自动添加到列表中的软件包的

实例。默认行为也可以导致导入不需要的软件包。

要指定要由捆绑包导入的软件包列表，请将 Import-Package child 添加到插件的 instructions 元素中。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，以确定是否存在所需的导入。要确保对捆绑包的内容进行了扫描，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详情，请参考：

- olink:OsgiDependencies/OsgiDependencies
- Apache Felix 文档
- Peter Kriens' aQute Software Consultancy 网站

部分 V. 使用 JAX-WS 开发应用程序

本指南介绍了如何使用标准 JAX-WS API 开发 Web 服务。

第 24 章 底层服务开发

摘要

有许多实例，您的 Java 代码已实现一组功能，您希望作为面向服务的应用程序的一部分公开。您可能还想避免使用 WSDL 来定义接口。使用 JAX-WS 注释，您可以添加 service 启用 Java 类所需的信息。您还可以创建一个 *可以代替 WSDL 合同的服务端点接口* (SEI)。如果您需要 WSDL 合同，Apache CXF 提供了从标注的 Java 代码生成合同的工具。

24.1. JAX-WS 服务开发简介

要创建从 Java 启动的服务，您必须执行以下操作：

1. 第 24.2 节 “创建 SEI” 一个 Service Endpoint Interface(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，但建议从接口操作。接口更适合与负责开发使用服务的应用程序的开发人员共享。这个接口比较小，不提供任何服务的实现详情。

2. 第 24.3 节 “为代码添加注解” 代码所需的注解。

3. 第 24.4 节 “生成 WSDL” 您的服务的 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* 服务作为服务提供商。

24.2. 创建 SEI

概述

服务端点接口 (SEI)是服务实施与在该服务上发出请求的消费者之间共享的 Java 代码。SEI 定义服务实施的方法，并提供了有关如何将服务公开为端点的详细信息。从 WSDL 合同开始时，SEI 由代码生成器生成。但是，从 Java 开始，开发人员负责创建 SEI。 创建 SEI 的基本模式有两个：

- 绿色字段开发 - 在这个模式中，您要开发没有现有 Java 代码或 WSDL 的新服务。最先创建 SEI。然后，您可以将 SEI 分发给负责实施使用 SEI 的服务供应商和消费者的任何开发人员。

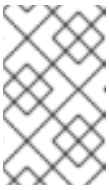
注意

推荐使用绿色字段服务开发的方法是创建一个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，您通常具有作为 Java 类实施的现有功能集，并且您要启用该服务。这意味着您必须执行两个操作：

  a. 创建一个 SEI，它只包含作为服务的一部分公开的操作。

  b. 修改现有的 Java 类，以便其实现 SEI。



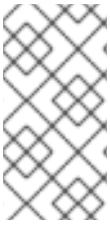
注意

虽然您可以将 JAX-WS 注释添加到 Java 类，但不建议这样做。

编写接口

SEI 是一个标准 Java 接口。它定义一组类实施的方法。它还可定义多个成员字段以及实施类可访问的常量。

如果是 SEI，定义的方法旨在映射到服务所公开的操作。SEI 与 wsdl:portType 元素对应。SEI 定义的方法与 wsdl:portType 元素中的 wsdl:operation 元素对应。



注意

JAX-WS 定义了一个注释，允许您指定不作为服务一部分公开的方法。但是，最好的做法是将这些方法从 SEI 离开。

例 24.1 “simple SEI” 显示用于库存更新服务的简单 SEI。

例 24.1. simple SEI

```
package com.fusesource.demo;

public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

实现接口

由于 SEI 是一个标准 Java 接口，因此实施它的类是标准 Java 类。如果从 Java 类开始，您必须修改才能实施接口。如果使用 SEI 启动，则实施类将实施 SEI。

例 24.2 “简单的实施类” 显示在 例 24.1 “simple SEI” 中实施接口的类。

例 24.2. 简单的实施类

```
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. 为代码添加注解

24.3.1. JAX-WS Annotations 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
- 如果某个操作是单向操作
- 服务使用的绑定风格
- 用于任何自定义例外的类名称
- 服务使用的命名空间

注意

大多数注解都有可识别的默认值，不需要为其提供值。但是，您在注解中提供的更多信息，会指定更好的服务定义。精心指定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. 必需的注解

概述

要从 Java 代码创建服务，您只需要向代码添加一个注解。您必须在 SEI 和实施类中添加 @WebService 注释。

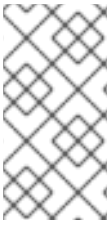
@WebService 注解

@WebService 注释由 javax.jws.WebService 接口定义，它被放在接口或要用作服务的类中。@webService 具有中描述的属性 表 24.1 “@webService 属性”

表 24.1. @webService 属性

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，该元素在 WSDL 合同中定义服务的接口。默认为在实施类的名称中添加 **PortType**。[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	指定已发布的服务的名称。此属性映射到定义所发布服务的 **wsdl:service** 元素的 **name** 属性。默认值是使用服务的实施类的名称。
wsdlLocation	指定存储服务的 WSDL 合同的 URL。这必须使用一个相对 URL 指定。默认为部署该服务的 URL。
endpointInterface	指定实施类实施的 SEI 的完整名称。仅在服务实施类中使用 属性时指定此属性。
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，用于指定公布的服务的端点详情。默认为在服务的实现类中添加 **Port**。

[a] 从 SEI 生成 WSDL 时，会使用接口名称来代替实施类名称。



注意

不需要为任何 @WebService 注释的属性提供值。但是，建议您提供尽可能多的信息。

为 **SEI** 标注

SEI 要求您添加 @WebService 注释。由于 SEI 是定义服务的合同，所以您应该在 @WebService 注释的属性中指定服务尽可能多的详细信息。

例 24.3 “带有 @WebService 注解的接口” 显示 例 24.1 “simple SEI” 中带有 @WebService 注释的接口。

例 24.3. 带有 @WebService 注解的接口

```
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 注解的接口” 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。

指定定义所发布服务的 wsdl:service 元素的值为 updateQuoteService。

指定该服务将在 http://demos.redhat.com/quoteExampleService?wsdl 发布其 WSDL 合同。

指定定义公开该服务的 端点的 name 属性的值是 updateQuotePort。

为服务实施添加注解

除了使用 @WebService 注释标注 SEI 外，还必须为服务实施类添加 @WebService 注释。当在服务实施类中添加注解时，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 注释足以启用 Java 接口或 Java 类，但它不完全描述服务将如何作为服务提供商公开。JAX-WS 编程模型使用许多可选注释来添加有关服务的详细信息，如其使用的绑定到 Java 代码。您可以将这些注解添加到服务的 SEI 中。

您在 SEI 中提供的更多详情是开发人员实施可使用它所定义的功能的应用程序。它还使 WSDL 文档更具体由工具生成。

概述

使用 Annotations 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 注解来指定多个绑定属性。这些属性直接与您在服务的 WSDL 合同中指定的属性对应。某些设置（如参数风格）可以限制如何实施方法。这些设置也可以影响在注解方法参数时使用哪些注解。

@SOAPBinding 注解

@SOAPBinding 注解由 javax.jws.SOAp.SOAPBinding 接口定义。它提供有关服务在部署时使用的 SOAP 绑定的详细信息。如果没有指定 @SOAPBinding 注解，则会使用嵌套的 doc/literal SOAP 绑定来发布服务。

您可以将 @SOAPBinding 注释放在 SEI 上，以及任何 SEI 的方法。在方法中使用时，将方法的 @SOAPBinding 注释设置具有优先权。

表 24.2 “@SOAPBinding 属性” 显示 @SOAPBinding 注释的属性。

表 24.2. @SOAPBinding 属性

属性	值	描述
样式	样式.DOCUMENT（默认） 样式.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，并出现在 **soap:body** 元素中的 wrapper 元素中。wrapper 元素中的消息部分对应于操作参数，且必须与操作中的参数的顺序相同。如果指定了 DOCUMENT 风格，则 SOAP 正文的内容必须是有效的 XML 文档，但其表单并没有严格限制。
使用	使用.LITERAL（默认） use.ENCODED[a]	指定传输 SOAP 消息的数据。
parameterStyle [b]	ParameterStyle.BARE ParameterStyle.WRAPPED（默认）	指定与 WSDL 合同中的消息部分对应的方法参数如何放入 SOAP 消息正文。如果指定了 BARE，则会将每个参数放在邮件正文中，作为消息根的子元素。如果指定了 WRAPPED，则所有输入参数都嵌套到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] 目前不支持 use.ENCODED。 [b] 如果将 **风格** 设置为 RPC，则必须使用 WRAPPED 参数样式。		

文档裸机风格参数

文档裸机风格是 Java 代码与服务生成的 XML 代码之间最直接映射。使用这种风格时，从操作参数列表中定义的输入和输出参数直接生成 schema 类型。

您需要使用 @SOAPBinding 注释将其样式设置为 Style.DOCUMENT，其 parameterStyle 属性设为 ParameterStyle.BARE 来指定您要使用裸机文档\literal 样式的样式。

要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。**

- 如果操作具有非 void 的返回类型，它不能具有任何输出或输入/输出参数。
- 如果操作的返回类型为 void，则不能有多个输出或输入/输出参数。

注意

使用 @WebParam 注释或 @WebResult 注释放入 SOAP 标头中的任何参数都不会根据允许的参数数目进行计数。

文档嵌套参数

文档嵌套风格允许更多 RPC，如 Java 代码和服务的 XML 表示之间的映射。使用这种风格时，方法的 parameter 列表中的参数会被绑定嵌套到单个元素中。这样做的缺点是，它在 Java 实施之间引入了一个额外的间接层，以及如何将消息放置在线路上。

要指定您要使用嵌套文档\literal 样式的 @SOAPBinding 注释，其 样式 属性设置为 Style.DOCUMENT，并将其 parameterStyle 属性设置为 ParameterStyle.WRAPPED。

您可以对使用 “@RequestWrapper 注释”一节 注解和 “@ResponseWrapper 注释”一节 注解生成的打包程序有一些控制。

示例

例 24.5 “使用 SOAP Binding 注解指定文档 Bare SOAP Binding” 显示使用文档裸机 SOAP 消息的 SEI。

例 24.5. 使用 SOAP Binding 注解指定文档 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

使用注释定义操作属性

当运行时将 Java 方法定义映射到 XML 操作定义时，它提供详情，例如：

- 在 XML 中交换的信息类似如下
- 如果消息可以优化为一条方法消息
- 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 SEI 中的方法中。@WebMethod 注释提供了通常在 ws:operation 元素中 代表的信息，用于描述与该方法关联的操作。

表 24.3 “@WebMethod 属性” 描述 @WebMethod 注释的属性。

表 24.3. @WebMethod 属性

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	指定是否应该将方法排除在服务接口中。默认值为 false。

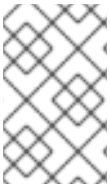
@RequestWrapper 注释

@RequestWrapper 注释由 javax.xml.ws.RequestWrapper 接口定义。它放置在 SEI 中的方法中。@RequestWrapper 注释指定 Java 类，用于实施请求消息交换方法参数的 wrapperan。它还指定了在处理请求消息时运行时使用的元素名称和命名空间。

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 XML representation 中的 wrapper 元素的本地名称。默认值可以是方法的名称，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 类的完整名称。



注意

只有 className 属性才是必需的。

重要

如果方法也带有 @SOAPBinding 注释，其 parameterStyle 属性设为 ParameterStyle.BARE，则忽略此注解。

@ResponseWrapper 注释

@ResponseWrapper 注释由 javax.xml.ws.ResponseWrapper 接口定义。它放置在 SEI 中的方法中。@ResponseWrapper 指定在消息交换中为方法参数实施打包程序的 Java 类。它还指定在处理响应消息和解包时运行时使用的元素名称和命名空间。

表 24.5 “@ResponseWrapper Properties” 描述 @ResponseWrapper 注释的属性。

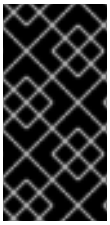
表 24.5. @ResponseWrapper Properties

属性	描述
localName	指定响应消息 XML representation 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 “@WebMethod 注释”一节 注解的 **operationName** 属性的值附加了 Response。

属性	描述
targetNamespace	指定定义 XML 打包程序元素的命名空间。默认值为 SEI 的目标命名空间。
className	指定实施 wrapper 元素的 Java 类的完整名称。

注意

只有 className 属性才是必需的。



重要

如果方法也带有 @SOAPBinding 注释，其 parameterStyle 属性设为 ParameterStyle.BARE，则忽略此注解。

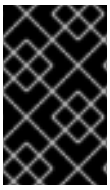
@WebFault 注释

@WebFault 注释由 javax.xml.ws.WebFault 接口定义。它会被置于您的 SEI 引发的异常。@WebFault 注释用于将 Java 异常映射到 wsdl:fault 元素。这些信息用于将例外置于可由服务及其消费者处理的表示中。

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	指定实施异常的 Java 类的全名。



重要

name 属性是必需的。

@Oneway 注解

@Oneway 注释由 javax.jws.Oneway 接口定义。它放置在 SEI 中不需要来自该服务的响应的方法。@Oneway 注释告知运行时，它可以通过不等待响应而优化方法执行，而且不会保留任何资源来处理响应。

此注解只能在满足以下条件的方法中使用：

- **它们返回 void**
- **它们没有实现 Holder 接口的参数**
- **它们不会抛出任何例外，可以传回给消费者**

示例

例 24.6 “带有注解的方法 SEI” 显示标有其方法的 SEI。

例 24.6. 带有注解的方法 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 Annotations 定义参数属性

SEI 中的方法参数对应于 wsdl:message 元素及其 wsdl:part 元素。JAX-WS 提供注解，供您描述

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中。如果参数将被放入 SOAP 标头，则 @WebParam 注释可以指定参数方向，以及生成的 wsdl:part 的其他属性。

表 24.7 “@WebParam 属性” 描述 @WebParam 注释的属性。

表 24.7. @WebParam 属性

属性	值	描述
name		指定参数的名称，因为它出现在生成的 WSDL 文档中。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 替换为基于零的参数索引（如 arg0、arg0 等）。
targetNamespace		指定参数的命名空间。它只适用于参数映射到 XML 元素的文档绑定。默认值是使用服务的命名空间。
模式	mode.IN（默认）[a] mode.OUT Mode.INOUT	指定参数方向。
header	false（默认） 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] 任何实现 Holder 接口的参数默认映射到 Mode.INOUT。		

@WebResult 注解

@WebResult 注释由 javax.jws.WebResult 接口定义。它放置在 SEI 中定义的方法中。@WebResult 注释允许您指定为方法返回值生成的 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	指定返回值的名称，因为它出现在生成的 WSDL 文档中。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。它仅用于记录绑定，其中返回值映射到 XML 元素。默认值是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。

示例

例 24.7 “完全注解的 SEI” 显示已完全注解的 SEI。

例 24.7. 完全注解的 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 Annotations

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。

此注解可让您添加文档，然后在 SEI 转换为 WSDL 后在 wsdl:documentation 元素内显示。默认情况下，文档元素显示在端口类型中，但您可以指定放置属性，使文档显示在 WSDL 文件的其他位置。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）** 指定在 WSDL 文件中显示此文档的位置。有关可能的放置值列表，请参阅 “在 WSDL 合同中放置”一节。
faultClass	**（可选）** 如果放置设置为 **FAULT_MESSAGE**、**PORT_TYPE_OPERATION_FAULT** 或 **BINDING_OPERATION_FAULT**，则必须将此属性设置为代表该故障的 Java 类。

@WSDLDocumentationCollection 注解

@WSDLDocumentationCollection 注释由 org.apache.cxf.annotations.WSDLDocumentationCollection 接口定义。它可以放在 SEI 或 SEI 方法

上。

此注解用于在单一位置或不同的位置插入多个文档元素。

在 WSDL 合同中放置

要指定在 WSDL 合同中应当出现文档的位置，您可以指定 放置 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” 展示如何将 @WSDL 文档 注释添加到 SEI 和其方法之一。

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 第 24.3.4.4 节 “使用文档生成的 WSDL” 中显示的 WSDL（在 第 24.3.4.3 节 “使用 @WSDL 文档” 中）生成时，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>
```

@WSDL 文档Collection 的示例

第 24.3.4.5 节 “使用 @WSDL 文档Collection” 展示如何将 @WSDLDocumentationCollection 注释添加到 SEI。

24.3.4.5. 使用 @WSDL 文档Collection

```
@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. 消息的 schema 验证

@SchemaValidation 注释

@SchemaValidation 注释由 org.apache.cxf.annotations.SchemaValidation 接口定义。它可以放在 SEI 和独立 SEI 方法上。

此注解打开发送到此端点的 XML 消息的 schema 验证。当您怀疑出现传入 XML 消息格式的问题时，这可用于测试目的。默认情况下禁用验证，因为它对性能有显著影响。

模式验证类型

架构验证 behaviour 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

"架构验证类型值" 显示可用验证类型的列表。

24.3.4.7. 架构验证类型值

类型	描述
IN	对客户端和服务器上的传入消息应用架构验证。
OUT	将架构验证应用到客户端和服务器上的传出消息。
两者	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 schema 验证。
REQUEST（请求）	将模式验证应用到 Request 消息，从而导致验证应用到传出客户端消息和传入的服务器消息。
RESPONSE	将模式验证应用到响应消息，从而导致验证应用到传入客户端消息和传出服务器消息。

示例

以下示例演示了如何根据 MyService SEI 为端点启用消息的 schema 验证。请注意，注解如何作为一个整体应用到 SEI，以及 SEI 中的个别方法。

```
@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 注解

@DataBinding 注解由 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，以替换默认的 JAXB 数据块。@DataBinding 注解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 数据绑定。

- 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，则需要在 cxf-rt-databinding-sdo 工件中添加依赖项。

- org.apache.cxf.aegis.databinding.AegisDatabinding

  如果要在 Maven 构建上下文中使用此数据绑定，则需要在 cxf-rt-databinding-aegis 工件中添加依赖项。

- org.apache.cxf.xmlbeans.XmlBeansDataBinding

  如果要在 Maven 构建上下文中使用此数据绑定，则需要在 cxf-rt-databinding-xmlbeans 工件中添加依赖项。

- org.apache.cxf.databinding.source.SourceDataBinding

这个数据绑定属于 Apache CXF 内核。

- org.apache.cxf.databinding.stax.StaxDataBinding

  这个数据绑定属于 Apache CXF 内核。

示例

第 24.3.4.9 节 “设置数据绑定” 展示如何将 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. 压缩信息

@GZIP 注解

@GZIP 注释由 org.apache.cxf.annotations.GZIP 接口定义。它放置在 SEI 上。

启用 GZIP 压缩信息。GZIP 是一个协商的增强。也就是说，客户端的初始请求将不会被 gzip 压缩，但会添加 Accept 标头；如果服务器支持 GZIP 压缩，其响应将被 gzip 压缩，任何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	超过此属性指定大小的消息 不会被 gzip 压缩。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。

为该消息启用 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。如需了解更多详细信息，请参阅 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，但会添加 Accept 标头。如果服务器支持 FastInfoset，响应将位于 FastInfoset 中，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，而不是协商。当为 **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” 显示如何为 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);
}
```

@FastInfoset 的考试

第 24.3.4.14 节 “启用 FastInfoset” 显示如何为 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 属性” 显示此注解中可以设置的可选属性。

24.3.4.16. @logging 属性

属性	描述
limit	指定在日志中截断信息的大小限制。默认为 64K。
inLocation	指定记录传入信息的位置。可以是 < **stderr>**、& **lt;stdout&** gt;、**<logger** > 或文件名。默认为 &lt **;logger>**。
outLocation	指定记录传出消息的位置。可以是 < **stderr>**、& **lt;stdout&** gt;、**<logger** > 或文件名。默认为 &lt **;logger>**。

示例

第 24.3.4.17 节 “使用注解进行日志配置” 演示了如何为 HelloWorld SEI 启用日志记录，其中传入消息发送到 < stdout> 和传出消息到 < logger>。

24.3.4.17. 使用注解进行日志配置

```
@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。策略通常来自 WS 规格和标准，它们通常通过定义 Wdl:policy 元素（在 WSDL 合同中出现）来设置。相反，属性是特定于 Apache CXF 的，它们通常通过在 Apache CXF Spring 配置文件中定义 jaxws:properties 元素进行设置。

不过，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。

此注解为端点添加特定于 Apache CXF 的配置设置。endpoint 属性也可以在 Spring 配置文件中指定。例如，若要在端点上配置 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>
```

另外，您可以通过在 SEI 中添加 @EndpointProperty 注解来指定 Java 中的上述配置设置，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 annotation

@EndpointProperties 注释由 org.apache.cxf.annotations.EndpointProperties 接口定义。它放置在 SEI 上。

此注释提供了一种将多个 @EndpointProperty 注释分组到列表中的方法。使用 @EndpointProperties 时，可以重新编写 第 24.3.4.20 节 “使用 @EndpointProperty Annotations 配置 WS-Security”，如 第 24.3.4.21 节 “使用 @EndpointProperties 注解配置 WS-Security” 所示。

24.3.4.21. 使用 @EndpointProperties 注解配置 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 来引用包含标准 wsdl:policy 元素的 XML 文件。如果从 SEI 生成 WSDL 合同（例如，使用 java2ws 命令行工具），您可以指定是否要将这个策略包含在 WSDL 中。

第 24.3.4.23 节 “@policy 属性” 显示 @Policy 注释支持的属性。

24.3.4.23. @policy 属性

属性	描述
uri	**（必需）** 文件的位置，包含策略定义。
includeInWSDL	**（可选）** 在生成 WSDL 时，是否将策略包含在生成的合同中。默认为 **true**。
placement	**（可选）** 指定在 WSDL 文件中显示此文档的位置。有关可能的放置值列表，请参阅 “在 WSDL 合同中放置”一节。
faultClass	**（可选）** 如果放置设置为 **BINDING_OPERATION_FAULT** 或 **PORT_TYPE_OPERATION_FAULT**，则必须设置此属性来指定这个策略应用到哪个故障。该值是代表故障的 Java 类。

@policies 注释

@Policies 注释由 org.apache.cxf.annotations.Policies 接口定义。它可以放在 SEI 或thse SEI 方法上。

此注释提供了一种将多个 @Policy 注释分组到列表中的方法。

在 WSDL 合同中放置

要指定在 WSDL 合同中应出现策略的位置，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中的 XML 文件中，该文件在 注解policies 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

注解您的代码后，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 插件的选项列表，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 qualified className。

示例

例 24.9 “从 SEI生成 WSDL” 显示在 例 24.7 “完全注解的 SEI” 中为 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 在没有 WSDL 合同的情况下开发消费者

摘要

您不需要 WSDL 合同来开发服务消费者。您可以从标注的 SEI 创建服务消费者。除了 SEI，您需要知道公开该服务的端点的地址，定义公开该服务的 service 元素的 QName，以及定义消费者发出请求的端点的 QName。此信息可以在 SEI 的注解中指定或单独提供。

25.1. JAVA-FIRST CONSUMER DEVELOPMENT

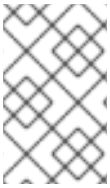
要创建没有 WSDL 合同的消费者，您必须执行以下操作：

1. 为服务创建一个 Service 对象，供使用者调用操作。
2. 向 Service 对象添加端口。
3. 使用 Service 对象的 getPort（） 方法获取该服务的代理。
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，它提供可让您获得由 wsdl:port 元素定义的端点的方法，这些元素是用于在服务上执行远程调用的代理。

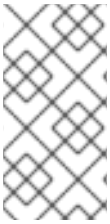


注意

Service 类提供抽象概念，允许客户端代码使用 Java 类型而不是使用 XML 文档。

create（）方法

Service 类有两个静态 create（） 方法，可用于创建新的 Service 对象。如 例 25.1 “service create（） 方法” 所示，create（） 方法采用 wsdl:service 元素的 QName 来代表 Service 对象，另一个则使用指定 WSDL 合同位置的 URI。



注意

所有服务发布他们的 WSDL 合同。对于 SOAP/HTTP 服务，URI 通常是附加了 ?wsdl 的服务的 URI。

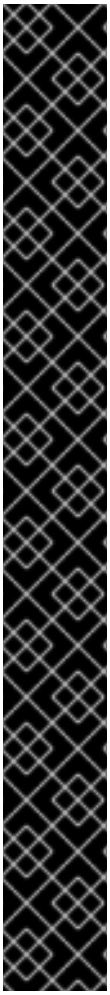
例 25.1. service create（） 方法

公共静态服务创建URLwsdlLocationQNameserviceNameWebServiceExceptionpublic 静态服务创建QNameserviceNameWebServiceException

serviceName 参数的值是一个 QName。其 namespace 部分的值是该服务的目标命名空间。该服务的目标命名空间在 @WebService 注释的 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. 将 Service 附加到 @WebService 注释的 name 属性的值。

2. 将 Service 附加到 SEI 的名称。



重要

在 OSGi 环境中部署的 CXF 消费者需要特殊处理，以避免发生类 NotFoundException的可能性。对于包含编程创建的 CXF 消费者的每个捆绑包，您需要创建一个单例 CXF 默认总线，并确保所有捆绑包的 CXF 使用者都使用它。如果没有这种保护机制，可以为一个捆绑包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，并且被分配了捆绑包 B 中创建的 CXF 默认总线。如果捆绑包 A 中的 CXF 总线需要配置额外功能（如 SSL 或 WS-Security），或者需要从捆绑包 A 中加载特定类或资源，则会失败。这是因为 CXF 总线实例将线程上下文类加载程序设置为创建它的捆绑包的类加载程序（本例中为 B）。此外，某些框架，比如 ws4j（在 CXF 中实施 WS-Security）使用 TCCL 从捆绑包内部加载资源，如 calback 处理程序类或其他属性文件。因为分配了捆绑包 B 的默认 CXF 总线及其 TCCL，因此 ws4j 层无法从捆绑包 A 加载所需资源，从而导致 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 “示例”一节 所示：

```
BusFactory.setThreadDefaultBus(BusFactory.newInstance().createBus());
```

示例

例 25.2 “创建服务 对象” 显示为 例 24.7 “完全注解的 SEI” 中显示的 SEI 创建 Service 对象的代码。

例 25.2. 创建服务 对象

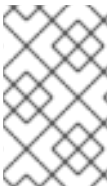
```
package com.fusesource.demo;

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 使用者使用。

使用 targetNamespace 属性和 @WebService 注解的 name 属性，为服务构建 QName。

调用单一参数 create（） 方法以创建新的 Service 对象。



注意

使用单一参数 create（） 可让您获得访问 WSDL 合同的任何依赖项。

25.3. 将端口添加到服务

概述

服务的端点信息在 wsdl:port 元素中定义，Service 对象会为 WSDL 合同中定义的每个端点创建一个代理实例。如果您在创建 Service 对象时不指定 WSDL 合同， Service 对象没有获取有关实施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（在 例 25.3 “addPort（） 方法” 中显示），在消费者实施中没有可用于 WSDL 合同的情况下使用。addPort（） 方法允许您为 Service 对象提供信息，后者通常存储在 wsdl:port 元素中，以便为服务实施创建代理。

例 25.3. addPort（） 方法

addPortQNameportNameStringbindingIdStringendpointAddressWebServiceException

portName 的值是一个 QName。其 namespace 部分的值是该服务的目标命名空间。该服务的目标

命名空间在 @WebService 注释的 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。
3. 将 Port 附加到 SEI 的名称。

bindingId 参数的值是一个字符串，它唯一标识端点使用的绑定类型。对于使用标准 SOAP 命名空间的 SOAP 绑定：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，地址是一个 HTTP 地址。HTTP 以外的传输使用不同的地址方案。

示例

例 25.4 “将端口添加到 服务对象” 显示在 例 25.2 “创建服务 对象” 中创建的服务 对象中 添加端口的代码。

例 25.4. 将端口添加到 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[])
  {
    ...
    QName portName = new QName("http://demo.redhat.com", "stockQuoteReporterPort");
    s.addPort(portName,
              "http://schemas.xmlsoap.org/soap/",
              "http://localhost:9000/StockQuote");
    ...
  }
}
```

例 25.4 “将端口添加到 服务对象” 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. 获取端点的代理

概述

服务代理是一个对象，提供由远程服务公开的所有方法，并处理进行远程调用所需的所有详情。Service 对象为通过 getPort（） 方法了解的所有端点提供服务代理。具有服务代理后，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（） 方法显示在 例 25.5 “getPort（） 方法” 中，为指定的端点返回服务代理。返回的代理与 SEI 相同。

例 25.5. getPort（） 方法

```
public<T> TgetPortQNameClass<T&gt;serviceEndpointInterfaceWebServiceException
```

portName 参数的值是一个 QName，用于标识 wsdl:port 元素，用于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

当您在没有 WSDL 合同的情况下工作时，portName 参数的值通常与调用 addPort（） 时用于 portName 参数的值相同。

示例

例 25.6 “获取服务代理” 显示为添加到 例 25.4 “将端口添加到 服务对象” 中的端点获取服务代理的代码。

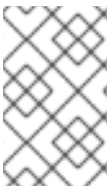
例 25.6. 获取服务代理

```
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. 实现 CONSUMER 的商业 LOGIC

概述

为远程端点实例化服务代理后，您可以像本地对象一样调用其方法。调用块直到远程方法完成。



注意

如果某个方法带有 @OneWay 注释，则调用会立即返回。

示例

例 25.7 “在没有 WSDL 合同的情况下实施的消费者” 显示 例 24.7 “完全注解的 SEI” 中定义的服务的使用者。

例 25.7. 在没有 WSDL 合同的情况下实施的消费者

```
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 “在没有 WSDL 合同的情况下实施的消费者” 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。

调用服务代理上的操作。

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。此合同在 wsdl:portType 元素中定义了一个接口 Greeter。该合同还定义了要在 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 接口定义以下操作：

sayHi - 单个输出参数，即 xsd:string。

greetMe - xsd:string 的输入参数，以及一个输出参数，即 xsd:string。

greetMeOneWay - 具有 xsd 的单个输入参数：string。由于此操作没有输出参数，所以它被优化为单向调用（即，使用者不会等待服务器的响应）。

pingMe - 没有输入参数，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层方法中，您从定义了服务提供商要实施的操作和方法的 WSDL 文档开始。使用 WSDL 文档，您可以为服务提供商生成起始点代码。在生成的代码中添加业务逻辑是使用常规 Java 编程 API 完成的。

27.1. JAX-WS 服务提供商开发概述

拥有 WSDL 文档后，开发 JAX-WS 服务提供商的过程如下：

1. 第 27.2 节 “生成 Starting Point Code” 起始点代码。
2. 实施 服务提供商的操作。
3. 第 31 章 *发布服务* 所实施的服务。

27.2. 生成 STARTING POINT CODE

概述

JAX-WS 指定从 WSDL 中定义的服务到将服务实施为服务提供商的 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会映射到 Java 类，遵循用于 XML Binding(JAXB)规格的 Java 架构定义的映射。由 wsdl:service 元素定义的端点也会生成成 Java 类，供使用者用于访问实施该服务的服务提供商。

cxf-codegen-plugin Maven 插件生成此代码。它还为您提供了为实施生成起始点代码的选项。代码生成器提供了控制生成的代码的多个选项。

运行代码生成器

例 27.1 “服务代码生成” 演示了如何使用代码生成器为服务生成起点代码。

例 27.1. 服务代码生成

```
<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>-server</extraarg>
              <extraarg>-impl</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

这会执行以下操作：

- **impl** 选项为 **WSDL** 合同中的每个 **wsdl:portType** 元素生成 **shell** 实施类。
- **server** 选项生成 一个简单的 **main**（），以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名为 ***outputDir*** 的目录。
- **WSDL** 元素指定 从中生成代码的 **WSDL** 合同。

有关代码生成器选项的完整列表请参考 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现的接口。您不应该编辑这个文件。
***serviceName*.java**	端点。此文件包含用于在服务上发出请求的 Java 类消费者。
***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**	允许您将服务提供商部署为独立进程的基本服务器主要线。更多信息请参阅 第 31 章 *发布服务*。

此外，代码生成器会为 WSDL 合同中定义的所有类型生成 Java 类。

生成的软件包

生成的代码会根据 WSDL 合同中使用的命名空间放入软件包中。为支持服务（基于 wsdl:portType 元素）、wsdl:service 元素和 wsdl:port 元素生成的类被放在软件包中，基于 WSDL 合同的目标命名空间。根据 type 元素的 targetNamespace 属性，为实施合同元素中定义的 类型 指定的类放置在软件包中。

映射算法如下：

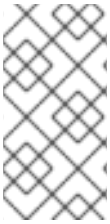
1. 领先的 http:// 或 urn:// 将从命名空间剥离。
2. 如果命名空间中的第一个字符串是有效的互联网域，例如，以 .com 或 .gov 结束，那么前导 www. 将从字符串中剥离，并且两个剩余的组件都是软盘。
3. 如果命名空间中的最终字符串以 .xxx 或 .xx 模式的文件扩展名结尾，则会剥离扩展。
4. 命名空间中的其余字符串附加到生成的字符串中，并以点分开。

5. 所有字母均由小写。

27.3. 实施服务提供商

生成实施代码

您生成了一个使用代码生成器的 **-impl** 标志来构建服务提供商的实施类。



注意

如果您的服务合同包含 XML 架构中定义的任何自定义类型，您必须确保生成并可用的类型类。

有关使用代码生成器的详情请参考 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 - 服务接口(SEI)。
- *portTypeName*Impl.java - 您要使用的类来实施服务所定义的操作。

实施操作的逻辑

为了为您的服务操作提供业务逻辑，可在 *portTypeName*Impl.java 中完成 stub 方法。您通常使用标准 Java 来实现业务逻辑。如果您的服务使用自定义 XML 架构类型，必须使用各种类型生成的类来操作它们。还有一些 Apache CXF 特定的 API 可用于访问某些高级功能。

示例

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必须由 mer 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、消息和传输详细信息。消费者的起点代码从 WSDL 合同生成。consumer 所需的功能添加到生成的代码中。

28.1. 生成 STUB CODE

概述

cxf-codegen-plugin Maven 插件从 WSDL 合同生成 stub 代码。stub 代码提供了在远程服务中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：

- 根代码 - 支持实施消费者的文件。
- 起始点代码 - 连接到远程服务的示例代码，并调用远程服务中的每个操作。

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来生成消费者代码。

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，而 *wsdl* 指定 WSDL 合同的位置。client 选项为 使用者的 main（） 方法生成起始点代码。

有关 cxf-codegen-plugin Maven 插件可用参数的完整列表请查看 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 - 此软件包从 http://apache.org/hello_world_soap_http 目标命名空间生成。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 类此 Java 软件包。
- org.apache.hello_world_soap_http.types - 此软件包从 http://apache.org/hello_world_soap_http/types 目标命名空间中生成。此命名空间中定义的所有 XML 类型（即，在 HelloWorld 合同的 wsdl:types 元素中定义的一切）都映射到该 Java 软件包中的 Java 类。

cxf-codegen-plugin Maven 插件生成的存根文件属于以下类别：

- 代表 org.apache.hello_world_SOAp_http 包中的 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 - 代表 SOAPService wsdl:service 元素的 Java 服务类（扩展 javax.xml.ws.Service）。
- PingMeFault - 代表 pingMeFault wsdl:fault 元素的 Java 异常类（扩展 java.lang.Exception）。

- 代表 org.objectweb.hello_world_ SOAp_http.types 软件包中的 XML 类型。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。其中一些数据类型对异步调用模型非常有用。

28.2. 实现消费者

概述

要在从 WSDL 合同开始实施消费者，您必须使用以下根根：

- Service class
- 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 Service 类概述” 显示了生成的服务类 *ServiceName*_Service的典型概述[2]，它扩展了 javax.xml.ws.Service 基础类。

例 28.2. 生成 Service 类概述

```
@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 Service 类概述” 中的 ***ServiceName*** 类定义以下方法：

- ***ServiceName*(URL wsdlLocation, QName serviceName) - 基于 wsdl:service 元素中具有来自 *wsdlLocation* 的 QName *ServiceName* 服务的数据构造服务对象。**
- ***ServiceName*（） - 默认构造器。它根据服务名称和生成 stub 代码时提供的 WSDL 合同来构建服务对象（例如，在运行 wsdl2java 工具时）。使用此构造器预测 WSDL 合同在指定位置仍然可用。**
- ***ServiceName*（Bus 总线） - （CXF 特定） 是一个额外的构造器，允许您指定用来配置该服务的总线实例。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方法，可确保您指定的总线是与此服务一起使用的总线。只有在调用 wsdl2java 工具时指定 -fe cxf 选项，才可用。**
- **get*PortName*（） - 为 wsdl:port 元素定义的端点返回代理，name 属性等于 *PortName*。为 *ServiceName* 服务定义的每个 wsdl:port 元素生成一个 getter 方法。包含多个端点定义的 wsdl:service 元素会生成带有多个 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： .输入参数映射到方法参数。

1. **第一个输出参数映射到返回值。**

2. 如果有多个输出参数，则第二个输出参数和后续输出参数映射到方法参数（此外，必须通过 Holder 类型传递这些参数的值）。

例如，例 28.3 “Greeter Service Endpoint Interface” 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 Service Endpoint Interface” 省略了标准 JAXB 和 JAX-WS 注释。

例 28.3. Greeter Service Endpoint Interface

```
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” 显示实现 HelloWorld 消费者的代码。使用者连接到 SOAPService 服务中的 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（） 方法进行，如下所示：

如果 Apache CXF 运行时类位于类路径上，则运行时会被隐式初始化。不需要调用特殊的函数来初始化 Apache CXF。

使用者需要一个字符串参数，为 HelloWorld 提供 WSDL 合同的位置。WSDL 合同的位置保存在 wsdlURL 中。

您可以使用需要 WSDL 合同的位置和服务名称的构造器创建服务对象。 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，该端口实施 Greeter 服务端点接口。

使用者调用 Greeter 服务端点接口支持的每个方法。

如果是 pingMe（） 方法，示例代码演示了如何捕获 pingMeFault 故障异常。

使用 -fe cxf 选项生成的客户端代理

如果您在 wsdl2java 中指定 -fe cxf 选项来生成客户端代理（因此选择 cxf frontend），则生成的客户端代理代码与 Java 7 更好地集成。在这种情况下，当您调用 get*ServiceName*Port（） 方法时，您会返回一个类型为 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

要了解这如何简化使用客户端代理的过程，请考虑使用标准 JAX-WS 代理对象编写的 Java 代码示例：

```
// 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();
```

并将前面的代码与以下等效的代码示例进行比较，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 属性以服务结尾，则不使用 _Service。

第 29 章 在运行时查找 WSDL

摘要

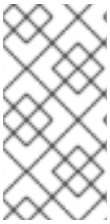
将 WSDL 文档的位置硬编码到应用程序中无法扩展。在实际部署环境中，您要在运行时解析 WSDL 文档的位置。Apache CXF 提供了很多可能实现的工具。

29.1. 用于查找 WSDL 文档的机制

在使用 JAX-WS API 开发消费者时，您必须提供用于定义服务的 WSDL 文档的硬编码路径。虽然这在小环境中非常正常，但使用硬编码路径在企业部署中不能正常工作。

要解决这个问题，Apache CXF 提供三种机制来删除使用硬编码路径的要求：

- 第 29.2 节 “使用注入注入代理”
- 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，因为它最容易实现。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. 使用注入注入代理

概述

通过 Apache CXF 使用 Spring Framework，可以避免使用 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到实施代码中。当运行时实例化实现对象时，也会根据配置为外部服务实例化代理。通过引用实例化代理来手动实现这个实现。

由于代理使用配置文件中的信息实例化，因此不需要硬编码 WSDL 位置。它可以在部署时更改。您还可以指定运行时应该搜索 WSDL 的类路径。

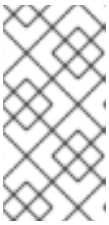
流程

要将外部服务的代理注入到服务提供商的实现中，请执行以下操作：

1. 在应用程序可以访问的已知位置部署所需的 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 架构文档放在 WAR WEB-INF/wsdl 文件夹中。



注意

如果您要将应用程序部署为 JAR 文件，建议将所有 WSDL 文档和 XML 架构文档放在 JAR 的 META-INF/wsdl 文件夹中。

2. 为 正在注入的代理配置 JAX-WS 客户端端点。

3. 使用 @Resource 注释，将代理 注入到 您的服务提供。

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。这会告知运行时，使用指定属性实例化 org.apache.cxf.jaxws.JaxWsClientProxy 对象。此对象是注入到服务供应商中的代理。

您至少需要为以下属性提供值：

- id- 表示用于标识要注入的客户端的 ID。
- serviceClass- 表示代理发出请求的服务的 SEI。

例 29.1 “配置将代理注入服务实施中” 显示 JAX-WS 客户端端点的配置。

■

例 29.1. 配置将代理注入服务实施中

```
<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 “配置将代理注入服务实施中” wsdlLocation 属性中，指示运行时从类路径加载 WSDL。如果 book.wsdl 位于 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 Consumer 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现中，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。配置的代理注入注解后立即声明的 BookService 对象。您可以使用此对象在代理的外部服务上调用。

29.3. 使用 JAX-WS 目录

概述

JAX-WS 规范强制所有实施支持：

在解析 Web 服务文档的任何 Web 服务文档（特别是 WSDL 和 XML 架构文档）时可使用标准目录功能。

这个目录工具使用由 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。

写入目录

JAX-WS 目录是 OASIS XML 目录 1.1 规范中定义的标准 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 “Common JAX-WS Catalog Elements” 列出用于 WSDL 位置解析的一些常见元素。

表 29.1. Common JAX-WS Catalog Elements

元素	描述
uri	将 URI 映射到备用 URI。
rewriteURI	重写 URI 的开头。例如，此元素允许您将以 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，您可以映射以 foo.xsd 结尾到 classpath:foo.xsd 的所有 URI。

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 的所有可用资源组装在一起。如果您的应用程序打包成单个 JAR 或 WAR，您可以将目录放在单个文件中。

如果您的应用程序被打包为多个 JAR，您可以将目录分成多个文件。每个目录文件都可以模块化，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 特定服务ContractResolver 接口的实现。您还需要用总线注册自定义解析器。

正确注册后，将使用自定义合同解析器来解决任何所需 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 接口” 所示，这个接口具有一个单一的方法 getContractLocation（），需要实施。getContractLocation（） 采用服务的 QName，并返回服务的 WSDL 合同 URI。

例 29.3. ServiceContractResolver 接口

```
public interface ServiceContractResolver
{
  URI getContractLocation(QName qname);
}
```

用于解析 WSDL 合同位置的逻辑特定于应用程序。您可以添加从 UDDI 注册表、数据库、文件系统中自定义位置或者您选择的任何其他机制解析合同位置的逻辑。

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之前，您必须使用合同解析器 registry 注册它。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 类中提供默认的实现。

要使用默认 registry 注册合同解析器，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（） 方法从总线中获取服务合同 registry。
3. 创建您的合同解析器的实例。
4. 使用 registry 的 register（） 方法，使用 registry 的 register（）注册您的合同解析器。

例 29.4 “注册合同解析器” 显示使用默认 registry 注册合同解析器的代码。

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。

创建合同解析器的实例。

使用 registry 注册合同解析器。

使用配置注册合同解析器

您还可以实施合同解析器，以便可以通过配置将其添加到客户端。实施合同解析器是为了使运行时读取配置并实例化解析器自行注册的方式。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。

要实施合同解析器，以便可以通过配置将其添加到客户端：

1. 在您的合同解析器实施中添加 init（） 方法。
2. 在您的 init（） 方法中添加相关的逻辑，使用合同解析器 registry 注册合同解析器，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 “可以使用配置注册的合同问题” 显示可以使用配置添加到客户端的合同解析器实施。

例 29.5. 可以使用配置注册的合同问题

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

要使用客户端注册合同解析器，您需要在客户端的配置中添加一个 **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” 显示添加由 **org.apache.cxf.demos.myContractResolver** 类实施的配置解析器的 **bean**。

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** 会调用每个合同解析器的 **getContractLocation（）** 方法。它会返回从注册的合同解析器返回的第一个 **URI**。

如果您注册了在已知共享文件系统上解析 **WSDL** 合同的合同解析器，则这是唯一使用合同解析器。但是，如果您随后使用 **UDDI registry** 注册了已解析 **WSDL** 位置的合同解析器，则 **registry** 可能会使用两个解析器查找服务的 **WSDL** 合同。注册表首先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，则 **registry** 会尝试使用 **UDDI** 合同解析器查找它。

第 30 章 通用故障处理

摘要

JAX-WS 规范定义两种类型的故障。个是通用的 JAX-WS 运行时异常。另一个是消息处理过程中引发的协议特定例外类别。

30.1. 运行时故障

概述

大部分 JAX-WS API 引发通用 javax.xml.ws.WebServiceException 异常。

引发 WebServiceException 的 API

表 30.1 “Throw WebServiceException 的 API” 列出一些可以引发通用 WebServiceException 异常的 JAX-WS API。

表 30.1. Throw WebServiceException 的 API

API	原因
Binding.setHandlerChain()	处理程序链配置中存在错误。
BindingProvider.getEndpointReference()	指定的类不会从 **W3CEndpointReference** 分配。
Dispatch.invoke()	Dispatch 实例的配置中存在错误，或者在与服务通信时发生错误。
Dispatch.invokeAsync()	Dispatch 实例配置中存在错误。
Dispatch.invokeOneWay()	Dispatch 实例的配置中存在错误，或者在与服务通信时发生错误。
LogicalMessage.getPayload()	当使用提供的 **JAXBContext** 来取消汇总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()	指定的类不会从 **W3CEndpointReference** 分配。

30.2. 协议故障

概述

当请求处理过程中出错时，会抛出协议异常。所有同步的远程调用可能会抛出协议异常。根本原因可以在消费者的消息处理链或服务提供商中实现。

JAX-WS 规范定义了一个通用协议例外。它还指定 SOAP 特定的协议异常和特定于 HTTP 的协议例外。

协议例外的类型

JAX-WS 规范定义三种类型的协议异常。您捕获的异常依赖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” 描述三种协议类型，以及何时抛出。

表 30.2. 通用协议例外的类型

例外类	Thrown
javax.xml.ws.ProtocolException	这个例外是通用协议异常。无论使用的协议是什么，都很难发现它。如果您使用 SOAP 绑定或 HTTP 绑定，它可以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	使用 SOAP 绑定时，远程调用会抛出这个异常。更多信息请参阅 “使用 SOAP 协议例外”一节。
javax.xml.ws.http.HTTPException	使用 Apache CXF HTTP 绑定来开发 RESTful Web 服务时会抛出这个异常。更多信息请参阅 第 VI 部分 “开发 RESTful Web 服务”。

使用 SOAP 协议例外

SOAPFaultException 异常嵌套 SOAP 错误。底层 SOAP 错误作为 javax.xml.SOAPFault 对象存储在 fault 字段中。

如果服务实施需要引发异常，它不适用于应用程序创建的任何自定义例外，它可以使用例外创建者将错误嵌套在 SOAPFaultException 中，并将其退回给消费者。例 30.1 “抛出 SOAP 协议例外” 显示在传递方法时用于创建和抛出 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，他们可以通过检查嵌套的 SOAPFault 异常来检索异常。如 例 30.2 “从 SOAP 协议例外获取故障” 所示，使用 SOAPFaultException 异常来检索 SOAPFaultException 异常的 getFault（） 方法。

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商的代码。

31.1. 发布服务时

Apache CXF 提供多种方法将服务作为服务提供商发布。您发布服务的方式取决于您所使用的部署环境。Apache CXF 支持的许多容器不需要编写发布端点的逻辑。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 类负责发布 JAX-WS 服务提供商。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. **当应用程序关闭时停止端点。**

Endpoint 类提供了创建和发布服务提供商的方法。它还提供了一种方式，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

服务提供商使用 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一个 Endpoint 对象：

- 静态端点创建对象实施器 此 create（） 方法返回指定服务实施的 端点。Endpoint 对象是使用实现类 javax.xml.ws.BindingType 注解提供的信息创建的（如果存在）。如果注解不存在，则 Endpoint 会使用默认的 SOAP 1.1/HTTP 绑定。

- 静态EndpointcreateURIbindingID对象实现器 This create（） 方法使用指定的绑定为指定的实现对象返回 Endpoint 对象。这个方法会覆盖 javax.xml.ws.BindingType 注解提供的绑定信息（如果存在）。如果 绑定ID 无法解析，或者它是 null，则使用 javax.xml.ws.BindingType 中指定的绑定来创建 端点。如果没有可以使用 bindingID 或 javax.xml.ws.BindingType，则 端点 将使用默认的 SOAP 1.1/HTTP 绑定来创建。

- 静态端点发布字符串地址对象实现器 The publish（） 方法为指定的实施创建一个 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，可扫描可用于实施的绑定列表。如果找到了 Endpoint 对象，则创建并发布该 Endpoint 对象。如果没有找到，则方法会失败。

  使用 publish（） 与调用其中一个 create（） 方法相同，然后调用 ???TITLE??? 中使用的 publish（） 方法。

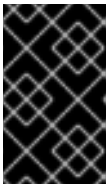
重要

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，并满足作为 SEI 实施的要求，或者必须是带有 javax.xml.ws.WebServiceProvider 的类实例，并实施 Provider 接口。

发布服务供应商

您可以使用以下 端点 方法之一发布服务提供商：

- 发布String地址 This publish（） 方法在指定的地址上发布该服务提供商。



重要

地址的 URL 方案必须与服务提供商的绑定之一兼容。

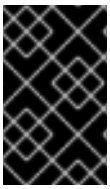
- 发布对象服务器Context 此 publish（） 方法基于指定服务器上下文提供的信息发布该服务提供商。服务器上下文必须为端点定义地址，上下文还必须与服务提供商的可用绑定之一兼容。

停止公布的服务供应商

当不再需要服务提供商时，您应该使用 stop（） 方法停止该服务。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其使用的所有资源。

例 31.1. 停止发布端点的方法

```
stop
```



重要

端点停止后，便无法重新发布。

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，您需要实现在应用的 main（） 方法中发布端点的逻辑。Apache CXF 为您提供了编写应用程序的 main（） 方法的两个选项。

- 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 标志使工具生成简单的服务器主线。生成的服务器主要线（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。

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。

根据端点的合同中 wsdl:port 元素的内容，创建端点的地址。

发布端点。

编写服务器主线

如果您使用了 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。要编写服务器主线，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. 创建发布服务提供商时要使用的可选服务器上下文。

3. “发布服务供应商”一节 服务提供商使用一个 publish（） 方法。

4. 当应用程序准备好退出时，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。

将服务提供商发布至 http://localhost:9000/SoapContext/SoapPort。

循环，直到服务器应关闭为止。

停止公布的端点。

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，您需要通过打包的捆绑包的生命周期协调发布和停止端点。您需要在捆绑包启动时发布端点，并希望在捆绑包停止时停止端点。

您可以通过实施 OSGi 捆绑包激活器将端点生命周期绑定到捆绑包的生命周期。捆绑激活器由 OSGi 容器用于在启动捆绑包时为其创建资源。容器也使用捆绑激活器在停止时清除捆绑包资源。

bundle activator 接口

您可以通过实施 org.osgi.framework.BundleActivator 接口为您的应用程序创建捆绑激活器。BundleActivator 接口在 例 31.4 “bundle Activator 接口” 中显示，它有两个需要实施的方法。

例 31.4. bundle Activator 接口

```
interface BundleActivator
{
  public void start(BundleContext context)
  throws java.lang.Exception;

  public void stop(BundleContext context)
  throws java.lang.Exception;
}
```

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，容器调用 stop（） 方法。这是您要停止端点的位置。

实施启动方法

捆绑包激活器的启动方法是您发布端点的位置。要发布您的端点，启动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. 创建发布服务提供商时要使用的可选服务器上下文。
3. “发布服务供应商”一节 服务提供商使用一个 **publish（） 方法。**

例 31.5 “发布端点的捆绑操作器启动方法” 显示发布服务提供商的代码。

例 31.5. 发布端点的捆绑操作器启动方法

```
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 “发布端点的捆绑操作器启动方法” 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。

将服务提供商发布到 http://localhost:9000/SoapContext/SoapPort。

实现停止方法

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。其实施中应包括停止应用程序发布的所有端点的逻辑。

例 31.6 “bundle Activator Stop Method 用于停止端点” 显示停止发布的端点的停止方法。

例 31.6. bundle Activator Stop Method 用于停止端点

```
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 stop(BundleContext context)
  {
    endpt.stop();
  }

  ...

}
```

告知容器

您必须添加相关容器，该容器应用程序捆绑包包含捆绑激活器。您可以通过在捆绑包的清单中添加

Bundle-Activator 属性进行此操作。此属性告知容器在激活捆绑包时要使用的类。其值是实施捆绑激活器的类的完全限定名称。

例 31.7 “bundle Activator Manifest Entry” 显示由类 com.widgetvendor.osgi.widgetActivator 实施的捆绑包的清单条目。

例 31.7. bundle Activator Manifest Entry

```
Bundle-Activator: com.widgetvendor.osgi.widgetActivator
```

第 32 章 基本数据绑定概念

摘要

有多个常规主题适用于 Apache CXF 如何处理类型映射。

32.1. 包含和导入架构定义

概述

Apache CXF 支持使用 include 和 import schema 标签，包含和导入 schema 定义。通过这些标签，您可以将外部文件或资源的定义插入到 schema 元素的范围。包括和导入的重要区别是：

- 包括在属于包含架构元素的同一目标命名空间中的定义中。
- 导入会进入属于包含架构元素的不同目标命名空间的定义中。

XSD:include 语法

include 指令使用以下语法：

```
<include schemaLocation="anyURI" />
```

任何URI 提供的引用的 schema 必须属于所属架构相同的目标命名空间，或者完全属于任何目标命名空间。如果引用的 schema 不属于任何目标命名空间，则在包含它时会自动将其添加到包含的 schema 命名空间中。

例 32.1 “包含 Another Schema 的 Schema 示例” 显示包含另一个 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” 显示所含架构文件的内容。

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 指令使用以下语法：

```
<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 “导入 Another Schema 的 Schema 示例” 演示了导入另一个 XML 架构的 XML 架构示例。

例 32.3. 导入 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 “导入架构示例” 显示导入的 schema 文件的内容。

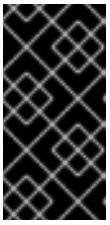
例 32.4. 导入架构示例

```
<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

使用在服务的 WSDL 文档中没有引用的 schema 文件中定义的类型是三个步骤：

1. 使用 xsd2wsdl 工具将架构文档转换为 WSDL 文档。
2. 使用生成的 WSDL 文档上的 wsdl2java 工具为类型生成 Java。



重要

您将收到 wsdl2java 工具的警告，表示 WSDL 文档没有定义任何服务。您可以忽略这个警告。

3. 将生成的类添加到类路径中。

32.2. XML 命名空间映射

概述

XML 架构类型、组和元素定义通过使用命名空间进行限定。命名空间可防止使用相同名称的实体间的命名冲突。Java 软件包具有类似目的。因此，Apache CXF 将架构文件的目标命名空间映射到包含实施架构文档中定义结构所需的类的软件包。

软件包命名

生成的软件包的名称派生自 schema 的目标命名空间，该算法来自以下算法：

1. URI 方案（如果存在）被剥离。

注意

Apache CXF 将仅剥离 http:、https: 和 urn: scheme。

例如，命名空间 http:\\www.widgetvendor.com\types\widgetTypes.xsd 变为 \\widgetvendor.com\types\widgetTypes.xsd。

2. 结尾的文件类型标识符（如果存在）会被剥离。

例如： \\www.widgetvendor.com\types\widgetTypes.xsd become \\widgetvendor.com\types\widgetTypes.

3. 生成的字符串使用 / 和 : 作为分隔符，被分为字符串列表。

因此，\\www.widgetvendor.com\types\widgetTypes 变成 list {"www.widegetvendor.com", "types", "widgetTypes"}.

4. 如果列表中的第一个字符串是互联网域名，则如下片段：

   a. 前导 www. 被剥离。

   b. 剩余的字符串将拆分为组件部分，使用 . 作为分隔符。

   c. 这个列表的顺序会被反转。

   因此, {"www.widegetvendor.com", "types", "widgetTypes"} become {"com", "widegetvendor", "types", "widgetTypes"}

   注意

   Internet 域名以下列之一结尾： .com、.net、.edu、.org、.gov 或两个字母国家代码之一。

5. 字符串转换为所有小写。

   因此, {"com", "widegetvendor", "types", "widgetTypes"} become {"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. 如果有任何字符串为 Java 关键字，则关键字前缀为下划线(_)。

   c. 如果有任何字符串以 numeral 开头，则该字符串的前缀为下划线(_)。

7. 字符串使用 . 作为分隔符来串联。

   因此, {"com", "widegetvendor", "types", "widgettypes"} 成为软件包名称 com.widgetvendor.types.widgettypes.

命名空间 http:\\www.widgetvendor.com\types\widgetTypes.xsd 中定义的 XML 架构结构映射到 Java 软件包 com.widgetvendor.types.widgettypes.widgettypes。

软件包内容

生成的 JAXB 生成的软件包包含以下内容：

- 类实施架构中定义的各种复杂类型

  有关复杂类型映射的详情，请参考 第 35 章 *使用复杂类型*。

- 使用枚举面市的任何简单类型进行 枚举 类型

  有关如何映射枚举的更多信息，请参阅 第 34.3 节 “枚举”。

- 一个公共 对象工厂 类，其中包含用于从架构中实例化对象的方法

  有关 ObjectFactory 类的更多信息，请参阅 第 32.3 节 “对象因素”。

- package-info.java 文件，提供有关软件包中类的元数据

32.3. 对象因素

概述

JAXB 使用对象工厂为实例化 JAXB 生成的结构实例提供机制。对象工厂包含在软件包范围内实例化定义的所有 XML 架构结构的方法。唯一的例外是，枚举不是对象工厂中的创建方法。

复杂的类型工厂方法

对于为实现 XML 模式复杂类型生成的每个 Java 类，对象工厂包含一个创建类实例的方法。这个方法采用以下形式：

```
typeName createtypeName();
```

例如，如果您的架构包含一个名为 widgetType 的复杂类型，Apache CXF 会生成名为 widget Type 的类来实现它。例 32.5 “复杂的类型对象 Factory Entry” 显示对象工厂中生成的创建方法。

例 32.5. 复杂的类型对象 Factory Entry

```
public class ObjectFactory
{
  ...
  WidgetType createWidgetType()
  {
    return new WidgetType();
  }
  ...
}
```

元素工厂方法

对于在 schema 全局范围中声明的元素，Apache CXF 会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 架构元素映射到 JAXBElement<T> 对象。创建方法采用以下形式：

```
public JAXBElement<elementType> createelementName(elementType value);
```

例如，如果您有一个名为 - sd:string 的名为 comment 的元素，Apache CXF 会生成对象工厂方法，如下所示 例 32.6 “元素对象 Factory Entry”

例 32.6. 元素对象 Factory Entry

```
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. 在 RUNTIME MARSHALLER 中添加类

概述

当 Apache CXF 运行时读取和写入 XML 数据时，它使用一个映射来将 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 架构 元素的目标命名空间中定义的所有类型。它还包含从导入到 WSDL 合同中任何模式的命名空间生成的类型。

除了应用程序 schema 元素所使用的命名空间以外的命名空间添加类型，可以使用 @XmlSeeAlso 注释来完成。如果您的应用程序需要处理在应用的 WSDL 文档范围外生成的类型，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。它包含要在 JAXB 上下文中包含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 {
        ...
    }
```

在您有权访问 JAXB 生成的类时，使用生成的 ObjectFactory 类来支持所需类型会更为高效。包含 对象 factory 类，包括对象工厂已知的所有类。

示例

例 32.8 “将类添加到 JAXB 上下文” 显示标记为 @XmlSeeAlso 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 架构元素用于在 XML 文档中定义元素实例。元素在 XML 架构文档的全局范围中定义，或者被定义为复杂类型的成员。在全局范围中定义时，Apache CXF 会将它们映射到一个 JAXB 元素类，从而更轻松地操作它们。

概述

XML 文档中的一个元素实例由 XML 架构文档中的全局范围 元素 定义，以便让 Java 开发人员更轻松地与元素一起工作，Apache CXF 将全局范围元素映射到一个特殊的 JAXB 元素类，或生成的 Java 类匹配其内容类型。

映射元素的方式取决于该元素是否使用 type 所引用的命名 类型 定义，或者使用内类型定义来定义该元素。使用行类型定义定义的元素映射到 Java 类。

建议使用命名类型定义元素，因为示例中的其他元素不重复使用行类型。

XML 架构映射

在 XML 架构中，利用 元素元素 来定义。元素 具有一条必需属性。name 指定在 XML 文档中出现的元素的名称。

除了 name 属性 元素 外，您还可以在 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，则需要在行类型定义中包含。
nillable	指定某个元素是否可完全离开文档。如果 nillable 被设置为 true，则元素可以从使用 schema 生成的任何文档中省略。

属性	描述
abstract	指定实例文档中是否可以使用元素。**true** 表示元素不能出现在实例文档中。相反，**替换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。有关此属性影响代码生成方式的详情，请参考 "抽象元素的 Java 映射"一节。
substitutionGroup	指定可用此元素替换的元素的名称。有关使用类型替换的详情，请参考 第 37 章 *element Substitution*。
default	指定元素的默认值。有关此属性影响代码生成方式的详情，请参考 "带有默认值的元素的 Java 映射"一节。
fixed	为元素指定固定值。

例 33.1 "简单 XML 架构元素定义" 显示了一个简单的元素定义。

例 33.1. 简单 XML 架构元素定义

```
<element name="joeFred" type="xsd:string" />
```

元素也可以使用内行类型定义来定义自己的类型。in-line 类型通过 complexType 元素或 simpleType 元素来指定。指定数据类型是否比较复杂或简单，您可以使用每种数据类型可用的工具定义任何类型的数据。

例 33.2 "XML 架构元素定义(in-line)类型" 显示包含行类型定义的元素定义。

例 33.2. XML 架构元素定义(in-line)类型

```
<element name="skate">
  <complexType>
    <sequence>
      <element name="numWheels" type="xsd:int" />
      <element name="brand" type="xsd:string" />
    </sequence>
  </complexType>
</element>
```

包含指定类型的元素的 JAVA 映射

默认情况下，全局定义的元素映射到 JAXBElement<T > 对象，其中 template 类由 element 元素 的 type 属性的值决定。对于 primitive 类型，模板类使用 “打包程序类”一节 中描述的打包程序类映射进行派生。对于复杂的类型，生成的 Java 类用于支持复杂类型，用作模板类。

为了支持映射并减轻开发人员对元素的 QName 有不必要的担心，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 “全球范围元素的对象因素方法” 所示。

例 33.3. 全球范围元素的对象因素方法

```
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，在 例 33.1 “简单 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 “简单元素的对象因素” 中显示。

例 33.4. 简单元素的对象因素

```
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” 演示了在 Java 中使用全局范围元素的示例。

例 33.5. 使用全局范围元素

```
JAXBElement<String> element = createJoeFred("Green");
String color = element.getValue();
```

使用 WSDL 中命名类型的元素

如果使用全局范围的元素来定义消息部分，则生成的 Java 参数不是 JAXBElement<T> 的实例。相反，它被映射到常规的 Java 类型或类。

根据 例 33.6 “使用元素作为消息部分的 WSDL” 中显示的 WSDL 片段，生成的方法具有类型为 String 的参数。

例 33.6. 使用元素作为消息部分的 WSDL

```
<?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 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

> 字符串中的字符串

具有内线类型的元素的 JAVA 映射

使用行类型定义元素时，它将按照与用于将其他类型映射到 Java 的规则相同的规则映射到 Java。简单类型的规则介绍了 第 34 章 *使用简单类型*。第 35 章 *使用复杂类型* 中描述了复杂类型的规则。

为带有行类型定义的元素生成 Java 类时，生成的类将被 @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 注解的属性” 中所述。

表 33.2. @XmlRootElement 注解的属性

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。如果此元素在目标命名空间中定义，则不指定该属性。

如果元素满足一个或多个条件，则不会使用 @XmlRootElement 注解：

- 元素的 nillable 属性被设置为 true
- 元素是替换组的头元素

  有关替换组的更多信息，请参阅 第 37 章 *element Substitution*。

抽象元素的 JAVA 映射

当元素的 abstract 属性设为 true 时，不会生成用于实例化类型的实例的对象工厂方法。如果该元素使用内行类型定义，则将生成支持内类型的 Java 类。

带有默认值的元素的 JAVA 映射

当使用元素 的默认 属性时，defaultValue 属性添加到生成的 @XmlElementDecl 注释。例如，在 例 33.8 “XML 架构元素，默认值” 中定义的元素会导致对象工厂方法在 例 33.9 “带有默认值的元素的对象因素方法” 中显示。

例 33.8. XML 架构元素，默认值

```
<element name="size" type="xsd:int" default="7"/>
```

例 33.9. 带有默认值的元素的对象因素方法

```
@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 架构简单类型是 XML 架构原语类型，如 xsd:int，或使用 simpleType 元素进行定义。它们用于指定不包含任何子或属性的元素。它们通常映射到原生 Java 结构，不需要生成特殊的类来实现它们。枚举的简单类型不会导致生成的代码，因为它们映射到 Java 枚举类型。

34.1. 原语类型

概述

使用其中一个 XML 架构原语类型定义消息部分时，生成的参数的类型会被映射到对应的 Java 原生类型。当映射在复杂类型的范围内定义的元素时，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 架构类型	Java 类型
XSD:string	字符串
XSD：整数	BigInteger
XSD:int	int
XSD:long	long
xsd:short	短
XSD:decimal	BigDecimal
XSD：浮点值	浮点值
XSD:double	double
XSD：布尔值	布尔值
XSD：字节	byte

XML 架构类型	Java 类型
xsd:QName	**QName**
xsd:dateTime	**XMLGregorianCalendar**
XSD:base64Binary	**byte[]**
xsd:hexBinary	**byte[]**
xsd:unsignedInt	**long**
xsd:unsignedShort	**int**
xsd:unsignedByte	**短**
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] 对于这种类型的属性。

打包程序类

将 XML 架构原语类型映射到 Java 原语类型，并不适用于所有可能的 XML 架构结构。有几个情况要求 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，其 nillable 属性设为 true，如下所示：**

```
<element name="finned" type="xsd:boolean"
        nillable="true" />
```

- 其 minOccurs 属性的 element 元素设为 0， 其 maxOccurs 属性设为 1，或其 maxOccurs 属性未指定，如下所示：

```
<element name="plane" type="xsd:string" minOccurs="0" />
```

- 使用 属性设置为 可选 属性或未指定的 attribute 元素，也没有指定其默认属性及其 固定 属性：

```
<element name="date">
  <complexType>
    <sequence/>
    <attribute name="calType" type="xsd:string"
              use="optional" />
  </complexType>
</element>
```

表 34.2 “原语架构类型到 Java Wrapper Class Mapping” 演示了 XML 架构制语类型如何在这些情形中被映射到 Java 打包程序类。

表 34.2. 原语架构类型到 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：浮点值	java.lang.Float
XSD:double	java.lang.Double
XSD：布尔值	java.lang.Boolean
XSD：字节	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. 由 RESTRICTION 定义的简单类型

概述

XML 架构允许您通过从其他原语类型或简单类型分离新类型来创建简单类型。简单类型通过 simpleType 元素进行描述。

新的类型通过限制一个或多个方面 的基本类型 来加以描述。这些限制会限制可以在新类型中保存的可能有效的值。例如，您可以定义一个简单类型 SSN，它是仅包含 9 个字符的字符串。

每个原语 XML 架构类型都有其自己的一组可选面。

流程

要定义您自己的简单类型，请执行以下操作：

1. 确定新简单类型的基本类型。
2. 根据所选基本类型的可用数据数据，确定定义新类型的限制。
3. 使用本节中显示的语法，将适当的 simpleType 元素输入到合同的 type 部分。

在 XML 架构中定义简单类型

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，由 **name** 属性的值来标识。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 元素内指定。可用的难题及其有效设置取决于基本类型。例如，**xsd:string** 有很多问题，其中包括：

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whitespace**

例 34.2 “后代码简单类型” 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后期代码。它只能包含两个大写字母。**TX** 是有效值，但 **tx** 或 **tX** 不是有效的值。

例 34.2. 后代码简单类型

```
<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（在 **例 34.2 “后代码简单类型”** 中）映射到一个 字符串，因为 **postalCode** 的基本类型是 **xsd:string**。例如，**例 34.3 “使用简单类型进行贡献请求”** 中显示的 **WSDL** 片段生成 **Java** 方法 **state**（），其使用参数 **postalCode (String)**。

例 34.3. 使用简单类型进行贡献请求

```
<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 不强制实施任何用于限制简单类型的问题。但是，您可以通过启用 schema 验证来配置 Apache CXF 端点来强制面临的问题。

要将 Apache CXF 端点配置为使用 schema 验证，将 schema-validation-enabled 属性设为 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详情，请参考 第 24.3.4.7 节 “架构验证类型值”。

34.3. 枚举

概述

在 XML 架构中，枚举的类型是一个简单的类型，它们使用 xsd:enumeration t 进行定义。与原子简单类型不同的是，它们映射到 Java 枚举。

在 XML 架构中定义枚举的类型

枚举是一个使用 xsd:enumeration facet 的简单类型。每个 xsd:enumeration facet 定义枚举类型的一个可能值。

例 34.5 “XML 架构定义的枚举” 显示枚举类型的定义。它具有以下可能的值：

- 大
- 大
- mungo
- gargantuan

例 34.5. XML 架构定义的枚举

```
<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 架构枚举（基本类型为 xsd:string ）的 XML 架构枚举：字符串自动映射到 Java 枚举类型。您可以使用 第 38.4 节 “自定义枚举映射” 中描述的自定义方法指示代码生成器与其他基本类型集成器映射到 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，则会在名称前加上一个 underscrore(_)。

2. 对于每个 枚举 面貌，根据 value 属性的值生成枚举常数。

   恒定名称通过将值中的所有小写字母转换为其大写等效符来派生。

3. 生成构造器，以取自枚举的基本类型所映射的 Java 类型。

4. 生成一个名为 value（） 的公共方法，以访问由 类型的实例表示的 facet 值。

   value（） 方法的返回类型是 XML 架构类型的基本类型。

5. 生成名为 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 架构类型的基础类型。

6. 该类使用 @XmlEnum 注释进行解码。

例 34.5 “XML 架构定义的枚举” 中定义的枚举类型映射到 例 34.6 “为字符串基础 XML 架构枚举的 Enumerated 类型” 中显示的 enum 类型。

例 34.6. 为字符串基础 XML 架构枚举的 Enumerated 类型

```
@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 架构支持定义数据类型的机制，这些类型是空格分隔的简单类型的列表。一个元素(primeList) 的示例显示了在 例 34.7 “列出类型示例” 中。

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。此模式的唯一变化是当消息部分直接映射到 XML 架构列表类型的实例。

在 XML 架构中定义列表类型

XML 架构列表类型是简单类型，因此使用 simpleType 元素来定义。用于定义列表类型的最常见语法显示在 例 34.8 “XML 架构列表类型语法” 中。

例 34.8. XML 架构列表类型语法

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value" />
    <facet value="value" />
    ...
  </list>
</simpleType>
```

为 ***atomicType*** 给出的值定义了列表中的元素类型。它只能是 XML 架构原子类型（如 xsd:int 或 xsd:string ）中构建的其中一个，也可以是不是列表的用户定义的简单类型。

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 “列出类型 Facets” 显示列表类型使用的 facets。

表 34.3. 列出类型 Facets

facet	效果
length	定义列表类型的实例中的元素数量。
minLength	定义列表类型中允许的最小元素数量。
maxLength	定义列表类型的实例允许的最大元素数。
枚举	定义列表类型的实例的允许值。
pattern	在列表类型的实例中定义元素的样式形式。使用正则表达式定义模式。

例如，在 例 34.7 “列出类型示例” 中显示的 simpleList 元素的定义显示在 例 34.9 “列表类型的定义” 中。

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 架构列表类型语法” 中显示的语法外，您还可以使用 例 34.10 “列出类型的备用语法” 中显示的常见语法定义列表类型。

例 34.10. 列出类型的备用语法

```
<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

当元素定义了列表类型时，列表类型映射到集合属性。collection 属性是 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 wrapper 类型映射的详情请参考 “打包程序类”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时，或映射到列表类型的元素时，生成的方法参数会被映射到数组而不是 List<T> 对象。阵列的基本类型是列表类型的基本类的 wrapper 类。

例如，例 34.11 “具有列表类型 Message 部分的 WSDL” 中的 WSDL 片段生成方法签名 例 34.12 “带有 List Type Parameter 的 Java 方法”。

例 34.11. 具有列表类型 Message 部分的 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 Parameter 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 架构中，联合是一个构造，它允许您描述类型，其数据可以是多个简单类型之一。例如，您可以定义一个类型，其值为整数 1，或 首先 定义字符串。unions 映射到 Java StringString。

在 XML 架构中定义

XML 架构表示法通过 simpleType 元素来定义。它们至少包含一个 用来定义 成员类型的联合元素。冲突的成员类型是有效的数据类型，可以存储在联合实例的实例中。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列表。例 34.13 “简单联合类型” 显示可以存储整数或字符串的联合定义。

例 34.13. 简单联合类型

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

除了将指定类型为 union 的成员类型外，您还可以将匿名简单类型定义为未发现的成员类型。这可以通过在 union 元素中添加匿名类型定义来实现。例 34.14 “Union(Aonymous 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 1 到 10。

例 34.14. Union(Aonymous 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。要让 Apache CXF 验证内容，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类型替换。但是，简单类型的默认映射到 Java 原语类型，但不完全支持简单类型替换。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可方便简单类型替换的 Java 类。

默认映射和编组

因为 Java 原语类型不支持类型替换，因此默认简单类型映射到 Java 原语类型提供了支持简单类型替换的问题。如果尝试将一个变量传递给一个变量，则 Java 虚拟机 会提示，即使定义类型允许的 schema 也会导致一个 int。

要解决 Java 类型系统的限制，Apache CXF 允许在元素的 xsi:type 属性的值满足以下任一条件时进行简单类型替换：

- **它指定与元素的 schema 类型兼容的原语类型。**
- **它指定从元素的 schema 类型限制限制的类型。**

- 它指定从元素的 schema 类型中由扩展生成的复杂类型。

当运行时进行类型替换时，它不会保留元素的 xsi:type 属性中指定的类型的任何知识。如果类型替换从复杂的类型到简单类型，则只保留与简单类型直接相关的值。由 扩展添加的任何其他元素和属性都将丢失。

支持丢失类型替换

您可以自定义生成简单类型，以方便以下列方式支持简单类型替换：

- 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命名简单类型创建 Java 值类。

  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 primitive 类型的所有实例映射到特定的对象类。

  更多信息请参阅 第 38.2 节 “指定 XML 架构 Primitive 的 Java 类”。

- 在您要自定义的特定元素中添加 baseType 自定义元素。

  baseType custom 元素允许您指定为表示属性生成的 Java 类型。为确保简单类型替换的最佳兼容性，请使用 java.lang.Object 作为基础类型。

  更多信息请参阅 第 38.6 节 “指定 Element 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。它们映射到可存放类型定义所代表数据的 Java 类。通常，映射是指包含一组代表元素和内容模型属性的属性的 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。更复杂的复杂类型由一组元素组成。

默认情况下，XML Schema 复杂类型映射到 Java 类，具有代表 XML 架构定义中列出的每个元素和属性的成员变量。该类为每个成员变量有 setters 和 getters。

在 XML 架构中定义

XML 架构复杂的类型是使用 complexType 元素定义的。complexType 元素嵌套了用来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，也可以作为元素的子元素匿名定义存储在元素中的信息结构。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 “定义元素在复杂类型中如何应用元素的元素” 中描述的子元素之一。这些元素决定了指定元素在类型的实例中如何出现。

表 35.1. 定义元素在复杂类型中如何应用元素的元素

元素	描述
all	作为复杂类型一部分定义的所有元素都必须出现在类型的实例中。但是，它们可能会以任何顺序出现。
choice	只有作为复杂类型一部分定义的一个元素才能出现在类型的实例中。
序列	作为复杂类型一部分定义的所有元素必须出现在类型的实例中，它们也必须按照类型定义中指定的顺序出现。

注意

如果复杂的类型定义只使用属性，则不需要 表 35.1 “定义元素在复杂类型中如何应用元素的元素” 中描述的元素之一。

在决定如何出现元素后，您可以通过向定义中添加一个或多个 元素 子来定义元素。

例 35.1 “XML Schema Complex Type” 显示了 XML 架构中的复杂类型定义。

例 35.1. XML Schema Complex Type

```
<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 类中的成员变量。此外，还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 类均带有 @XmlType 注释进行解码。如果映射适用于命名复杂类型，则注解 名称将设置为 complexType 元素的 name 属性的值。如果复杂的类型是作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 “具有内线类型的元素的 Java 映射”一节 所述，如果为作为元素定义的一部分定义的复杂类型生成，生成的类会与 @XmlRootElement 注释进行解码。

为了提供运行时，以指示应如何处理 XML 架构复杂类型元素，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 元素来定义。它们注解如下：

- @XmlType 注释的 propOrder 属性为空。
- 每个元素都会使用 @XmlElement 注释进行解码。
- @XmlElement 注释的 required 属性设置为 true。

例 35.2 “所有复杂类型的映射” 显示所有复杂类型以及两个元素的映射。

例 35.2. 所有复杂类型的映射

```
@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;
    }
}
```

Choice Complex Type

选择复杂类型通过选择元素来定义。它们注解如下：

- @XmlType 注释的 propOrder 属性根据它们出现在 XML 架构定义中的顺序列出元素

名称。

- 没有注解成员变量。

例 35.3 “选择复杂类型的映射” 显示了选择复杂类型的映射，以及两个元素。

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 元素定义序列复杂类型。它注解如下：

- @XmlType 注释的 propOrder 属性根据它们出现在 XML 架构定义中的顺序列出元素名称。

- 每个元素都会使用 @XmlElement 注释进行解码。

- @XmlElement 注释的 required 属性设置为 true。

例 35.4 “复杂类型映射” 显示 例 35.1 “XML Schema Complex Type” 中定义的复杂类型的映射。

例 35.4. 复杂类型映射

```
@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 元素范围内使用 属性元素和属性组 元素。在为 XML 文档属性声明定义结构时，提供了添加标签内指定的信息的方法，而不是标签包含的值。例如，在 XML 架构中描述 XML 元素 <value currency ="euro">410<\value> 时，使用 例 35.5 “XML 架构定义和属性” 所示的属性元素进行描述。

attributeGroup 元素允许您定义一组可重复使用的属性，这些属性可由架构定义的所有复杂类型引用。例如，如果您要定义一系列使用属性类别和 pubDate 的元素，您可以定义一个包含这些属性的属性组，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。

当描述用于开发应用程序逻辑中的数据类型时，使用 属性将其设置为 可选 或 所需的属性 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 属性的类中生成一个元素，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，名称，用于识别属性。它还有四个可选属性，如 表 35.2 “用于定义 XML 架构中的属性的可选属性” 所述。

表 35.2. 用于定义 XML 架构中的属性的可选属性

属性	描述
使用	指定是否需要属性。有效值 是必需的，可选，或禁止 使用。可选 是默认值。

属性	描述
type	指定属性可以获取的值类型。如果不使用的模式类型，则必须在命令行中定义属性。
default	指定用于属性的默认值。它仅在 **属性** 元素的 **use** 属性设置为 **可选** 时使用。
fixed	指定用于属性的固定值。它仅在 **属性** 元素的 **use** 属性设置为 **可选** 时使用。

例 35.5 “XML 架构定义和属性” 显示定义属性货币的属性元素，其值是字符串。

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 attribute 元素中省略了 type 属性，则数据的格式必须按行描述。例 35.6 “带有内数据描述的属性” 显示属性、类别 的 属性 元素，它可以取 autobiography、非 虚构值或虚构。

例 35.6. 带有内数据描述的属性

```
<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 架构中使用属性组

在复杂类型定义中使用属性组分为两个步骤：

1. 定义 属性组。

使用带有多个属性子元素的 attributeGroup 元素来定义 属性 组。attributeGroup 需要一个 name 属性，用于定义用于引用该属性组的字符串。属性 元素定义属性组的成员，并指定为 “在 XML 架构中定义属性”一节 所示。例 35.7 “属性组定义” 显示属性组 catalogIndecies 的描述。该属性组有两个成员： 类别 （可选）和 pubDate （需要）。

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。

您可以将 attributeGroup 元素用于 ref 属性，在复杂的类型定义中使用属性组。ref 属性的值是赋予您要用作类型定义一部分的属性组的名称。例如，如果要在复杂类型 dvdType 中使用属性组 catalogIndecies，您可以使用 <attributeGroup ref="catalogIndecies" />，如 例 35.8 “带有属性组的复杂类型” 所示。

例 35.8. 带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

属性映射到 Java，其成员元素的工作方式与映射到 Java 的方式相同。必要属性和可选属性映射到所生成的 Java 类中的成员变量。member 变量使用 @XmlAttribute 注释进行解码。如果需要 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” 所示，default 属性和 固定 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。此附加代码确保了如果没有设置值，则返回指定的值。

重要

fixed 属性处理与 default 属性相同。如果您希望 固定 属性被视为 Java 常量，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，就像在类型定义中明确使用组的成员一样。如果该属性组有三个成员，并且它用于复杂类型，则该类型的生成的类将包括 member 变量，以及 getter 和 setter 方法，用于属性组的每个成员。例如： 例 35.8 “带有属性组的复杂类型” 中定义的复杂类型，Apache CXF 生成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. 从简单类型分离复杂类型

概述

Apache CXF 支持从简单类型分离复杂类型。简单类型具有定义，但不定义 sub-elements 或 属性。因此，从简单类型分离复杂类型的主要原因之一就是将属性添加到简单类型。

从简单类型分离复杂类型的方法有两种：

- 按扩展
- 按限制

按扩展进行 Derivation

例 35.12 “从简单类型分离复杂类型（按扩展扩展）” 展示了复杂类型 国际Price 的示例，它通过 xsd:decimal primitive 类型的扩展来包括货币属性。

例 35.12. 从简单类型分离复杂类型（按扩展扩展）

```
<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。

根据限制进行调度

例 35.13 “从简单类型划分中分离复杂类型” 显示复杂类型 idType 的示例，它根据 xsd:string 的限制进行生成。定义的类型将 xsd:string的可能值限制为长度为 10 个字符的值。它还在类型中添加属性。

例 35.13. 从简单类型划分中分离复杂类型

```
<complexType name="idType">
  <simpleContent>
    <restriction base="xsd:string">
     <length value="10" />
     <attribute name="expires" type="xsd:dateTime" />
    </restriction>
  </simpleContent>
</complexType>
```

与在 例 35.12 “从简单类型分离复杂类型（按扩展扩展）” 中一样，新类型不包含任何子项 的简单内容元素信号。本例使用 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 注释进行解码的 Java 类。生成的类包含派生复杂类型的简单类型的成员变量 value。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，以及每个扩展简单类型的属性的 getter 和 setter 方法。

例 35.14 “idType Java Class” 显示 例 35.13 “从简单类型划分中分离复杂类型” 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. 从复杂类型分离复杂类型

概述

通过使用 XML 架构，您可以通过扩展或限制复杂 内容元素来生成新的复杂 类型。当生成 Java 类以表示派生复杂类型时，Apache CXF 扩展基础类型的类。这样，生成的 Java 代码会保留在 XML 架构中预期的继承层次结构。

模式语法

您可以使用 复杂的Content 元素以及 extension 元素或 limit 元素从其他复杂类型生成复杂类型。复杂的Content 元素指定包含的数据描述包含多个字段。extension 元素和 restrictions 元素（即 复杂 Content 元素的子项）指定正在修改的基础类型以创建新类型。基础类型由 base 属性指定。

扩展复杂类型

要扩展复杂类型，请使用 扩展 元素来定义组成新类型的额外元素和属性。在复杂类型描述中允许的所有元素都可以作为新类型的定义的一部分。例如，您可以向新类型添加匿名枚举，也可以使用 choice 元

素来指定仅新字段之一每次都有效。

例 35.15 “按扩展划分划分复杂类型” 显示定义两种复杂类型的 XML 架构片段，即 widgetOrderInfo 和 widgetOrderBillInfo。widgetOrderBillInfo 通过扩展 widgetOrderInfo 来包括两个新元素：orderNumber 和 amtDue。

例 35.15. 按扩展划分划分复杂类型

```
<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 limits 元素来限制基础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属性。例如，您可以在元素中添加 maxOccurs 属性来限制它可以被发生的次数。您还可以使用 固定 属性强制一个或多个元素具有预先确定的值。

例 35.16 “根据限制定义复杂类型” 演示了通过限制另一个复杂类型来定义复杂类型的示例。受限类型 wallawallaAddress 只能用于 Walla Walla、Washington 中的地址，因为 城市 元素、state 元素和 zipCode 元素的值会被修复。

例 35.16. 根据限制定义复杂类型

```
<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 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 类扩展了生成的 Java 类，以支持基础复杂类型。基本 Java 类也会修改为包含 @XmlSeeAlso 注释。base 类的 @XmlSeeAlso 注释列出了扩展基础类的所有类。

当通过扩展生成新复杂类型时，生成的类将包含所有添加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。

当根据限制派生出新复杂类型时，生成的类将没有新的成员变量。生成的类将只是一个 shell，它不提供任何附加功能。您完全是您确保强制实施 XML 架构中定义的限制。

例如，例 35.15 “按扩展划分划分复杂类型” 中的 schema 会产生两个 Java 类的生成：FlayOrderInfo 和 WidgetBillOrderInfo。widget OrderBillInfo 扩展 widget OrderInfo，因为 widgetOrderBillInfo 是由来自 widgetOrderInfo 的扩展。例 35.17 “WidgetOrderBillInfo” 显示 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. 发生限制

35.5.1. 模式元素支持限制

XML 架构允许您在组成复杂类型定义的四个 XML 架构元素中指定发生限制：

- 第 35.5.2 节 “All Element 上的冲突限制”

- 第 35.5.3 节 “对选择元素的发生限制”
- 第 35.5.4 节 “元素出现冲突限制”
- 第 35.5.5 节 “对序列的限制”

35.5.2. All Element 上的冲突限制

XML 架构

通过 all 元素定义的复杂类型不允许所有元素定义的结构出现多次。但是，您可以通过将 minOccurs 属性设置为 0 来为 all 元素定义的结构。

映射到 Java

将 all 元素的 minOccurs 属性设为 0 对生成的 Java 类没有影响。

35.5.3. 对选择元素的发生限制

概述

默认情况下，选择 元素的结果只能出现在复杂类型的实例中一次。您可以通过 minOccurs 属性及其 mxOccurs 属性来更改 选择 元素定义的结构的次数。通过使用这些属性，您可以指定选择类型可能会为零到复杂类型的实例中无限次。为选择类型选择的元素并不需要在每次发生次数时相同。

在 XML 架构中使用

minOccurs 属性指定选择类型必须出现的最小次数。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出现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。其值可以是任意非零、正整数或未绑定的 。将 maxOccurs 属性设置为 unbounded，指定选择类型可能会显示无限的次数。

例 35.18 “Choice Occurrence Constraints” 显示选择类型 ClubEvent 的定义，以及选择发生次数限制。选择类型总可以重复 0 以取消绑定次数。

例 35.18. Choice Occurrence Constraints

```
<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

与单个实例选择结构不同，XML 架构选择结构会多次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 “Choice Occurrence Constraints” 中定义的序列出现两次，则列表会有两个项目。

Java 类的成员变量的名称通过串联成员元素的名称来派生。元素名称由 Or 分隔，变量名称的第一个字母会被转换为小写。例如，从 例 35.18 “Choice Occurrence Constraints” 生成的成员变量将命名为 memberNameOrGuestName。

列表中存储的对象类型取决于成员元素的类型。例如：

- 如果生成的列表位于同一类型的成员元素，则生成的列表将包含 JAXBElement<T> 对象。JAXBElement<T> 对象的基本类型由 member 元素类型的典型映射决定。
- 如果成员元素是不同的类型，并且它们的 Java 表示实施了通用接口，则列表将包含通用接口的对象。
- 如果成员元素是不同的类型，并且它们的 Java 表示扩展了一个通用的基础类，则列表将包含常见基础类的对象。
- 如果没有满足其他条件，则列表将包含对象 对象。

生成的 Java 类将仅具有 member 变量的 getter 方法。getter 方法返回对实时列表的引用。对返回的列表进行的任何修改都将影响实际对象。

Java 类与 @XmlType 注释进行解码。注解的 name 属性设置为 XML 架构定义的父元素中的 name 属性的值。该注解的 propOrder 属性包含代表序列中的元素的单个成员变量。

代表选择结构中的成员变量通过 @XmlElements 注释来解码。@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成员元素。列表中 @XmlElement 注释的 name 属性设为 XML Schema 元素 的 name 属性的值，它们 类型 属性设置为 Java 类，从 XML Schema 元素 元素类型映射而生成的 Java 类。

例 35.19 “Java 代表带有 Occurrence Constraint 的选择结构” 显示 例 35.18 “Choice Occurrence Constraints” 中定义的 XML 架构选择结构的 Java 映射。

例 35.19. Java 代表带有 Occurrence Constraint 的选择结构

```
@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

如果仅指定了 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. 元素出现冲突限制

概述

您可以使用 element 的 minOccurs 属性和 maxOccurs 属性指定复杂类型中特定元素的多次。两个属性的默认值都是 1。

minOccurs 设置为 0

当您将复杂类型的 member 元素的 minOccurs 属性设置为 0 时，@XmlElement 注解会改变对应的 Java 成员变量。@XmlElement 注释的 required 属性设置为 false，而不是将其 required 属性设置为 true。

minOccurs 设置为大于 1 的值

在 XML 架构中，您可以通过将元素的 minOccurs 属性设置为大于一来，指定元素的 minOccurs 属性必须在类型的实例中多次发生。但是，生成的 Java 类不支持 XML 架构约束。Apache CXF 生成支持 Java 成员变量，就像未设置 minOccurs 属性一样。

带有 maxOccurs 集的元素

当希望一个成员元素在复杂类型的实例中显示多次时，您要将元素的 maxOccurs 属性设置为值大于 1。您可以将 maxOccurs 属性的值设置为 unbound，以指定 member 元素可能会显示为无限次数。

代码生成器将 maxOccurs 属性设为值大于 1 的成员元素映射到 List<T > 对象的 Java 成员变量。列表中的基础类由将元素类型映射到 Java 来确定。对于 XML 架构原语类型，使用打包程序类，如 “打包程序类”一节 所述。例如，如果 member 元素类型为 xsd:int，则生成的 member 变量是一个 List<Integer> 对象。

35.5.5. 对序列的限制

概述

默认情况下，序列 元素的内容只能在复杂类型的实例中出现一次。您可以使用 minOccurs 属性及其 maxOccurs 属性来更改由 sequence 元素定义的元素序列的次数。通过使用这些属性，您可以指定序列类型可能会为零到复杂类型的实例中无限次。

使用 XML 架构

minOccurs 属性指定在定义的复杂类型的实例中必须发生序列的最少次数。其值可以是任意正整数。将 minOcc urs 属性设置为 0。 指定序列不需要出现在复杂类型的实例中。

maxOccurs 属性指定在定义的复杂类型的实例中可以发生的次数的上限。其值可以是任意非零、正整数或未绑定的 。将 maxOccurs 属性设置为 unbounded，表示序列会出现无限次数。

例 35.20 “带有 Occurrence Constraints 的序列” 显示序列类型 CultureInfo 的定义，其顺序存在限

制。序列可重复 0 到 2 次。

例 35.20. 带有 Occurrence Constraints 的序列

```
<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，XML Schema 序列可能会多次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 35.20 “带有 Occurrence Constraints 的序列” 中定义的序列出现两次，则列表将是四项。

Java 类的成员变量的名称通过串联成员元素的名称来派生。元素名称由 And 分隔，变量名称的第一个字母会被转换为小写。例如，从 例 35.20 “带有 Occurrence Constraints 的序列” 生成的成员变量名为 nameAndLcid。

列表中存储的对象类型取决于成员元素的类型。例如：

- 如果生成的列表位于同一类型的成员元素，则生成的列表将包含 JAXBElement<T> 对象。JAXBElement<T> 对象的基本类型由 member 元素类型的典型映射决定。
- 如果成员元素是不同的类型，并且它们的 Java 表示实施了通用接口，则列表将包含通用接口的对象。
- 如果成员元素是不同的类型，并且它们的 Java 表示扩展了一个通用基础类，则该列表将包含通用基础类的对象。
- 如果没有满足其他条件，则列表将包含对象 对象。

生成的 Java 类仅具有 member 变量的 getter 方法。getter 方法返回对实时列表的引用。对返回的列表进行的任何修改都会影响实际对象。

Java 类与 @XmlType 注释进行解码。注解的 name 属性设置为 XML 架构定义的父元素中的 name 属性的值。该注解的 propOrder 属性包含代表序列中的元素的单个成员变量。

代表序列的 member 变量使用 @XmlElements 注释来解码序列中的元素。@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成员元素。列表中 @XmlElement 注释的 name 属性设为 XML Schema 元素 的 name 属性的值，它们 类型 属性设置为 Java 类，从 XML Schema 元素 元素类型映射而生成的 Java 类。

例 35.21 “带有 Occurrence 约束的序列代表” 显示 例 35.20 “带有 Occurrence Constraints 的序列” 中定义的 XML 架构序列的 Java 映射。

例 35.21. 带有 Occurrence 约束的序列代表

```
@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

如果仅指定了 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 架构模型组比较方便了快捷方式，可让您引用用户定义的复杂类型的元素组。例如，您可以定义

一组适用于应用程序中多种类型的元素，然后重复引用组。模型组使用 group 元素来定义，与复杂类型定义类似。模型组到 Java 的映射也与复杂类型的映射类似。

在 XML 架构中定义模型组

您可以使用 group 元素和 name 属性在 XML 架构中定义模型组。name 属性的值是一个字符串，用于在架构中指代组。group 元素（如 complexType 元素）可以具有 sequence 元素、all 元素或选择元素作为其即时子。

在子元素内，您可以使用 元素元素 定义组的成员。对于组的每个成员，指定一个 元素。组成员可以使用 element 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，并且其中一个元素最多可以发生三次，则定义一个含有三个 元素 的组，其中一个使用 maxOccurs="3"。例 35.22 “XML 架构模型组” 显示了一个含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，可用作复杂类型定义的一部分。要在复杂的类型定义中使用模型组，可将 group 元素与 ref 属性一起使用。ref 属性的值是被定义时给组的名称。例如：使用 例 35.22 “XML 架构模型组” 中定义的组，使用 {ref="tns:passenger" >，如 例 35.23 “带有模型组的复杂类型” 所示。

例 35.23. 带有模型组的复杂类型

```
<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，组将成为该类型的成员。因此，保留 的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 Model Group 的 Type 实例” 中显示了一个实例 保留 的示例。

例 35.24. 带有 Model Group 的 Type 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，模型组仅在复杂的类型定义中包含时映射到 Java 工件。为包含模型组的复杂类型生成代码时，Apache CXF 只是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进行注解。

例 35.25 “使用组输入” 显示为 例 35.23 “带有模型组的复杂类型” 中定义的复杂类型生成的 Java 类。

例 35.25. 使用组输入

```
@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 属性设置为大于一来来指定模型组会出现多次。要允许多次出现模型组 Apache CXF 将模型组映射到 List<T> 对象。List<T > 对象根据组的第一个子子规则生成：

- **如果使用 序列 元素定义组，** 请参阅 **第 35.5.5 节 “对序列的限制”。**

- 如果使用 选择 元素定义组，请参阅 第 35.5.3 节 “对选择元素的发生限制”。

第 36 章 使用通配符类型

摘要

当一个 schema 作者希望将绑定元素或属性映射到定义的类型时，会有一些实例。对于这些情况，XML 架构提供了三个用来指定通配符放置拥有者的机制。它们都以保留其 XML 架构功能的方式映射到 Java。

36.1. 使用 ANY ELEMENTS

概述

XML 架构 任何 元素用于在复杂类型定义中创建通配符拥有者。为 XML 架构 任何 元素实例化 XML 元素时，可以是任何有效的 XML 元素。any 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 “XML 架构类型通过 Any Element 定义” 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. XML 架构类型通过 Any Element 定义

```
<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 元素都映射到 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 元素是一个空元素。不过，它可以取 注释 元素作为子项。

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 文档中的元素的元素的命名空间。有效值为： **##any** 指定可以使用任意命名空间中的元素。这是默认值。 **##other** 指定可以使用 **父元素命名空间以外的任何命名空间中** 的元素。 **##local** 必须使用没有命名空间的元素。 **##targetNamespace** 指定必须使用父元素命名空间中的元素。 **空格分隔的 URI 列表 #\#local 和 \#\#targetNamespace** 指定可以使用任何列出的命名空间中的元素。
maxOccurs	指定元素中可以显示元素的最大次数。默认值为：**1**。要指定元素实例可能会出现无限次数，您可以将该属性的值设置为 **未绑定**。
minOccurs	指定元素中可以出现元素的最小次数。默认值为：**1**。
进程内容	指定实例化任何元素的元素应当如何进行验证。有效值为： **strict** 指定必须针对正确的 schema 验证元素。这是默认值。 **lax** 指定应针对正确的 schema 验证元素。如果无法验证它，则不会抛出错误。 **skip** 指定该元素不应经过验证。

例 36.3 “使用 Any Element 定义的复杂类型” 显示 使用任何 元素定义的复杂类型

例 36.3. 使用 Any Element 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 any 的 Java 属性。属性关联了 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 属性设置为跳过，则该 元素将映射到 org.w3c.dom.Element 对象。对于 processContents 属性的所有其他值，任何 元素都映射到 Java 对象。

生成的属性使用 @XmlAnyElement 注释进行解码。此注解具有一个可选的 lax 属性，指示运行时在划分数据时做什么。默认值为 false，它指示运行时自动将数据放入 org.w3c.dom.Element 对象中。将 lax 设置为 true 可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 属性设为其默认值。对于 processContents 属性的所有其他值，lax 设为 true。

例 36.4 “带有任何元素的 Java 类” 显示 例 36.3 “使用 Any Element 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 属性设为 false，或者未指定属性，则运行时不会试图将 XML 数据解析为 JAXB 对象。数据始终存储在 DOM Element 对象中。

如果任何 元素的 Java 属性设为 true，则运行时会尝试将 XML 数据放入适当的 JAXB 对象。运行时尝试使用以下步骤识别正确的 JAXB 类：

1. **它会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，则运行时摘要会将 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，则运行时摘要将 XML 元素放入该类型的正确 JAXB 类中。**

3. **如果无法找到与 XML 数据匹配的匹配，则将 XML 数据放入 DOM Element 对象。**

通常，应用程序的运行时会知道从其合同中包含的架构生成的所有类型。其中包括合同的 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向合同添加的任何类型。您还可以使用 @XmlSeeAlso 注释使运行时了解额外类型，如 第 32.4 节 “在 Runtime Marshaller 中添加类” 所述。

unmarshalling

如果任何 元素的 Java 属性设为 false，或者未指定属性，则运行时将仅接受 DOM Element 对象。尝试使用其他类型的对象将导致错误。

如果任何 元素的 Java 属性设为 true，则运行时会在 Java 数据类型和 XML 架构结构之间使用其代表的 XML 结构，以确定写入线的 XML 结构。如果运行时知道类并可以映射到 XML 架构结构，它将写入数据并插入 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，它将引发汇总异常。您可以使用 @XmlSeeAlso 注释（如 第 32.4 节 “在 Runtime Marshaller 中添加类” 中所述），向运行时添加类型。

36.2. 使用 XML 架构任何类型

概述

XML 架构类型 xsd:anyType 是所有 XML 架构类型的 root 类型。所有原语都是这种类型的衍生产品，因为所有用户定义的所有复杂类型。因此，作为 xsd:anyType 定义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，最匹配的类型是 Object 类。它是下级其他所有 Java 类的子类。

在 XML 架构中使用

您可以使用 xsd:anyType 类型作为其它 XML 架构复杂类型。它可以用作元素的 type 元素 的值。它还可用作定义其他类型的基本类型。

例 36.5 “带有通配符元素的复杂类型” 显示包含类型为 xsd:anyType 的复杂类型的示例。

例 36.5. 带有通配符元素的复杂类型

```
<complexType name="wildStar">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="ship" type="xsd:anyType" />
  </sequence>
</complexType>
```

映射到 Java

类型为 xsd:anyType 的元素映射到对象 对象。例 36.6 “Java 代表通配符元素” 显示 例 36.5 “带有通配符元素的复杂类型” 到 Java 类的映射。

例 36.6. Java 代表通配符元素

```
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卡元素的 属性中。Apache CXF 运行时处理数据碎片化和解包化，形成可用的 Java 表示。

marshalling

当 Apache CXF marshals XML 数据进入 Java 类型时，它会尝试 将任何Type 元素放入已知的 JAXB 对象。要确定如何将 任何Type 元素放入一个 JAXB 生成的对象，运行时将检查元素的 xsi:type 属性，以确定用于在元素中构造数据的实际类型。如果 xsi:type 属性不存在，则运行时会尝试按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决定为应用的 JAXB 上下文已知的类型之一，则该元素被放入正确类型的 JAXB 对象。

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。然后，您将需要使用 DOM APIs 与元素的内容一起工作。

应用的运行时通常知道从其合同中包含的架构生成的所有类型。其中包括合同中与 dl: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，以及通过导入其他架构文档添加到合同中的任何类型。您还可以使用 @XmlSeeAlso 注释使运行时了解额外类型，如 第 32.4 节 “在 Runtime Marshaller 中添加类” 所述。

unmarshalling

当 Apache CXF unmarshals Java 类型到 XML 数据时，它使用 Java 数据类型和 XML 架构结构之间的内部映射，以确定其对线的 XML 结构。如果运行时知道类，可以将类映射到 XML 架构结构，它将写入

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，运行时会写入对象代表的 XML 结构，但它不包含 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，它会抛出一个marshaling 异常。您可以使用 @XmlSeeAlso 注释（如 第 32.4 节 “在 Runtime Marshaller 中添加类” 中所述），向运行时添加类型。

36.3. 使用 UNBOUND 属性

概述

XML 架构有一个机制，允许您在复杂类型定义中保留任意属性的拥有者。通过使用这种机制，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。这在您的数据中的灵活性需要时特别有用。

在 XML 架构中定义

Undeclared 属性在 XML 架构中使用 anyAttribute 元素定义。它可用于在可以使用属性元素的位置。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 具有两个元素，并可具有任何类型的一个属性。例 36.8 “使用通配符属性定义的元素示例” 中显示的元素有三个元素，它们都可以从复杂的类型 arbitter 生成。

例 36.8. 使用通配符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 任何Attribute 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 java.util.Map<QName, String >，它是一种 getter 方法，它返回了映射的实时实例。由于 getter 返回的映射是实时的，对映射的任何修改都会被自动应用。例 36.9 “带有 Undeclared Attribute 的复杂类型” 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. 带有 Undeclared Attribute 的复杂类型

```
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; }

}
```

使用未压缩属性

生成的类 的另一个Attributes 成员应该被填充到 Map 对象。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 Attributes” 显示使用未压缩属性的代码示例。

例 36.10. 使用 Undeclared Attributes

```
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 Attributes” 中的代码执行以下操作：

获取含有未压缩属性的 map。

创建 QName 以用于属性。

将属性的值设置为映射。

检索其中一个属性的值。

第 37 章 ELEMENT SUBSTITUTION

摘要

借助 XML 架构替换组，您可以定义一组可替换顶层或 head 的元素的元素。当您拥有多个共享通用基础类型或需要交换的元素的元素时，这很有用。

37.1. 在 XML SCHEMA 中替换组

概述

替换组是 XML 模式的功能，允许您指定可替换从该架构生成的文档中另一元素的元素。可替换元素称为 head 元素，且必须在 schema 的全局范围内定义。替换组的元素必须是与 head 元素相同的类型，或者从 head 元素类型派生的类型。

本质上，替换组允许您构建一组可使用通用元素来指定的元素。例如，如果您为公司构建了一个排序系统，它销售三种类型的小部件，您可以定义一个包含全部三种小部件类型通用数据的通用小部件元素。然后，您可以定义一个替换组，其中包含每种类型的小部件的一组特定数据集。然后，您可以将通用 widget 元素指定为消息部分，而不为每种小部件定义特定排序操作。构建实际消息时，消息可以包含替换组的任何元素。

语法

替换组使用 XML 架构 元素的 replace Group 属性来定义。replace Group 属性的值是所定义元素的元素的名称。例如，如果您的 head 元素是 widget，请将属性 replaceGroup="widget" 添加到名为 woodWidget 的元素中，指定使用 widget 元素的任意位置，您可以替换 woodWidget 元素。这在 例 37.1 “使用 Substitution Group” 中显示。

例 37.1. 使用 Substitution Group

```
<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 元素类型相同，或者由 head 元素类型派生的类型。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类型 xsd:int 或从 xsd:int 派生的类型。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 小部件类型。 替换组的每个元素扩展 widgetType，使其包含特定于排序小部件类型的数据。

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>
```

摘要头元素

您可以定义一个抽象头元素，这些元素永远不会出现在使用您的 schema 生成的文档中。抽象元素与 Java 中的抽象类类似，因为它们用作定义更具体的通用类实施的基础。抽象头还阻止在最终产品中使用 generic 元素。

您可以通过将元素的 abstract 属性设置为 true 来声明 一个抽象 头 元素，如 例 37.4 “抽象头定义” 所示。使用此架构时，有效的 review 元素可以包含 正向 元素或 负请求 元素，但不能 包含注释 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. JAVA 中的替换组

概述

Apache CXF 按照 JAXB 规范中指定的，支持使用 Java 的原生类层次结构替换组，并组合使用 JAXBElement 类对通配符定义的支持。由于替换组的成员必须共享一个共同的基础类型，因此为支持元素类型生成的类也共享一个共同的基础类型。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

为支持包含替换组群的软件包生成的对象工厂有方法用于替换组中的每个元素。对于每个替换组的成员，除了 head 元素外，@XmlElementDecl 注解会包括两个额外的属性，如 表 37.1 “Declaring a JAXB Element 的属性是子组的成员” 所述。

表 37.1. Declaring a JAXB Element 的属性是子组的成员

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 命名空间 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 元素的对象的实例化方法。如果替换组的成员是所有复杂类型，则对象工厂还包含使用每种复杂类型的实例实例化方法。

例 37.5 “用于子组的对象因素方法” 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. 用于子组的对象因素方法

```
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);
    }

}
```

替换接口中的组

如果替换组的 head 元素用作操作消息之一的消息部分，则生成的方法参数将是为支持该元素而生成的类对象。它不一定是 JAXBElement< 实例？扩展 T> 类。运行时依赖于 Java 的原生类型层次结构来支持类型替换，Java 将捕获任何使用不支持的类型的尝试。

为确保运行时知道支持元素替换所需的所有类，使用 @XmlSeeAlso 注释来分离 SEI。此注解指定运行时用于汇总所需的类列表。有关使用 @XmlSeeAlso 注释的更多信息，请参阅 第 32.4 节 “在 Runtime Marshaller 中添加类”。

例 37.7 “使用替换组生成的接口” 显示为 例 37.6 “使用 Substitution Group 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 Substitution Group 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. 使用替换组生成的接口

```
@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 “使用替换组生成的接口” 中显示的 SEI 在 @XmlSeeAlso 注释中列出对象工厂。列出命名空间的对象工厂可访问该命名空间的所有生成类。

替换复杂类型的组

当替换组的 head 元素用作复杂类型中的元素时，代码生成器会将元素映射到 JAXBElement<? 扩展 T> 属性。它没有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 Substitution Group 的复杂类型” 中定义的复杂类型会导致 例 37.9 “使用子组进行复杂类型的 Java 类” 中显示的 Java 类。复杂的类型使用 例 37.2 “使用复杂类型替换组” 中定义的替换组。

例 37.8. 使用 Substitution Group 的复杂类型

```
<complexType name="widgetOrderInfo">
  <sequence>
```

```
    <element name="amount" type="xsd:int"/>
    <element ref="xsd1:widget"/>
  </sequence>
</complexType>
```

例 37.9. 使用子组进行复杂类型的 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

您如何使用替换组取决于代码生成器是否将组映射到直接 Java 类，还是一个 JAXBElement<？扩展 T > 类。当元素直接映射到生成值类的对象时，您可以像与作为类型层次结构一部分的其他 Java 对象一样使用对象。您可以替换父类的任何子类。您可以检查对象以确定其确切的类，并适当地进行广播。

JAXB 规范建议您使用对象工厂方法实例化所生成的类的对象。

当代码生成器创建 JAXBElement<?? 扩展 T > 对象来存放替换组的实例，您必须将元素的值嵌套在 JAXBElement<? 扩展 T> 对象。执行此操作的最佳方式是使用对象工厂提供的元素创建方法。它们提供了基于其值创建元素的简单方法。

例 37.10 “设置子组成员” 显示设置替换组的实例的代码。

例 37.10. 设置子组成员

```
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 “设置子组成员” 中的代码执行以下操作：

实例化对象工厂。

实例化 PlasticWidgetType 对象。

实例化一个 JAXBElement<PlasticWidgetType& gt; 对象以容纳 platformstic widget 元素。

实例化 小部件OrderInfo 对象。

将 widget OrderInfo 对象的 widget 设置为含有 plastic widget 元素的 JAXBElement 对象。

获取替换组属性值

从 JAXBElement<? 扩展 T > 对象提取元素的值时，对象工厂方法不帮助。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<? 扩展 T > 对象的 getName（） 方法，以确定元素的名称。

  getName（） 方法返回 QName。通过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<? 扩展 T > 对象的 getDeclaredType（） 方法来确定 value 对象的类。

  getDeclaredType（） 方法返回元素的值对象的 Class 对象。

警告

对于 head 元素，无论值对象的实际类如何，getDeclaredType（） 方法会返回基础类。

例 37.11 “获取子组成员组成员” 显示从替换组中检索值的代码。要确定元素的值对象的正确类，示例中使用了该元素的 getName（） 方法。

例 37.11. 获取子组成员组成员

```
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. 小部件顺序接口

本节演示了 Apache CXF 中用于解决真实应用的替换组示例。使用 例 37.2 “使用复杂类型替换组” 中定义的 widget 替换组来开发服务和消费者。该服务提供两个操作： checkWidget 和 placeWidgetOrder。例 37.12 “小部件顺序接口” 显示排序服务的接口。

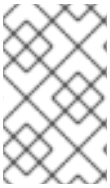
例 37.12. 小部件顺序接口

```
<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 “widget Ordering SEI” 显示为接口生成的 Java SEI。

例 37.13. widget Ordering 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

因为该示例只显示使用替换组，所以并没有显示一些业务逻辑。

37.3.2. checkWidgets Operation

概述

checkWidgets 是一个简单操作，其参数是替换组的头成员。此操作演示了如何处理作为替换组的成员的单个参数。使用者必须确保 参数是替换组群的有效成员。该服务必须正确确定在请求中发送替换组的成员。

消费者实施

生成的方法签名使用 Java 类来支持替换组的 head 元素类型。由于替换组的成员元素是与 head 元素类型相同的类型，或者由 head 元素类型派生的 Java 类，生成的 Java 类从生成的 Java 类中继承为支持 head 元素的 Java 类。Java 类型层次结构原生支持使用子类来代替父类。

由于 Apache CXF 如何生成替换组和 Java 类型层次结构的类型，客户端可以在不使用任何特殊代码的情况下调用 checkWidgets（）。当开发逻辑来调用 checkWidgets（） 时，您可以传递一个类的对象，以支持 widget 替换组。

例 37.14 “消费者检查 检查Widgets（）” 显示调用 checkWidgets（） 的使用者。

例 37.14. 消费者检查 检查Widgets（）

```
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);
```

服务实施

该服务的 checkWidgets（） 实施会将一个小部件描述作为 widget Type 对象获取，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的所有类都会从同一基础类继承，因此您可以在不使用任何 JAXB 特定的 API 的情况下实施 checkWidgets（）。

生成的所有类，以支持 widget 的 replace 组的成员扩展 widget Type 类。由于这一事实，您可以使用 instanceof 确定传递的小部件类型，只需在适当的时将 widgetPart 对象转换为更严格的类型。具有正确类型对象后，您可以检查正确类型的小部件。

例 37.15 “checkWidgets（）的服务实施” 显示可能的实施。

例 37.15. checkWidgets（）的服务实施

```
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 实施中使用此类结构。使用者和服务都必须获取和设置替换组的成员。

消费者实施

若要调用 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。在顺序中添加小部件时，使用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，如果将某一顺序放在了白板的小部件中，则使用 ObjectFactory.createPlasticWidget（） 方法在添加元素前创建该元素。

例 37.16 “设置子组成员” 显示用于设置 widget OrderInfo 对象的 widget 属性的使用者代码。

例 37.16. 设置子组成员

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

服务实施

placeWidgetOrder（） 方法以 小部件OrderInfo 对象的形式收到一个顺序，处理该顺序，并以 小部件OrderBillInfo 对象的形式向使用者返回计费。订购可以采用普通小部件、一个白板小部件或一个 wooden 小部件。排序小部件的类型由哪些对象类型存储在 widgetOrderForm 对象的 widget 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。

实施必须确定哪些元素按顺序存储。这可以使用 JAXBElement< 来完成？如何扩展 T& gt; 对象的 getName（） 方法，以确定该元素的 QName。然后，可以使用 QName 确定替换组中的哪个元素的顺序。知道 bill 中的元素后，您可以将其值提取到正确的对象类型。

例 37.17 “placeWidgetOrder（）的实施” 显示可能的实施。

例 37.17. placeWidgetOrder（）的实施

```
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 “placeWidgetOrder（）的实施” 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 小部件OrderBillInfo 对象以容纳 bill。

  获取排序的小部件数量。

  获取按顺序存储的元素的本地名称。

  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 JAXBElement<T> 对象，并放入 bill 中。

  设置 bill 对象的 widget 属性。

  设置 bill 对象的 amountDue 属性。

第 38 章 自定义类型是如何生成方式

摘要

默认的 JAXB 映射地址使用 XML 架构为 Java 应用定义对象时遇到的大部分情况。对于默认映射不足的实例，CIB 提供广泛的自定义机制。

38.1. 自定义类型映射的基础知识

概述

JAXB 规范定义了多种 XML 元素，这些元素自定义 Java 类型如何映射到 XML 架构结构。这些元素可以通过 XML 架构结构在命令行中指定。如果不能，或者不想修改 XML 架构定义，您可以在外部绑定文档中指定自定义。

命名空间

用于自定义 JAXB 数据源的元素在命名空间 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类似的命名空间。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，您必须指示要使用的 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。如果您使用在线自定义，则必须在包含自定义的 schema 元素中包含 jaxb:version 属性。该属性的值始终为 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

自定义代码生成器映射 XML 架构结构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 自定义元素放置在被修改的 XML 架构结构的 xsd:appinfo 元素中。

例 38.3 “自定义 XML 架构” 演示了一个含有内 JAXB 自定义的 schema 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

当您无法或不想更改定义您的类型的 XML 架构文档时，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 外部网络声明语法” 显示外部绑定声明的语法。

例 38.4. JAXB 外部网络声明语法

```
<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 属性用于标识要应用修改的架构文档。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 架构结构。它是解析为 XML 架构元素的 XPath 语句。

根据架构文档 widgetSchema.xsd （如 例 38.5 “XML 架构文件” 所示），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 bing 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 Schema 类型映射到 Java 原语类型。虽然这是 XML 架构与 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容纳额外信息的 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。它可用于在全局级别和个别实例级别自定义映射。您可以将 javaType 元素用作简单类型定义的一部分，也可以用作复杂类型定义的一部分。

当使用 javaType 自定义元素时，您必须指定将 primitive 类型的 XML representation 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 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 “用于自定义 Java 类生成 XML 架构类型的属性” 所述。

表 38.1. 用于自定义 Java 类生成 XML 架构类型的属性

属性	必填	描述
name	是	指定将 XML 架构原语类型映射到的 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。您必须确保这个类存在，且可以被您的应用程序访问。这个类的代码生成器没有检查。
xmlType	否	指定正在自定义的 XML 架构原语类型。此属性仅在 **javaType** 元素用作 **globalBindings** 元素的子级时使用。
parseMethod	否	指定负责将数据的基于字符串 XML representation 的方法解析为 Java 类的实例。更多信息请参阅 “指定转换器”一节。
printMethod	否	指定负责将 Java 对象转换为数据的字符串 XML 表示的方法。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 架构模拟类型的所有实例 - javaType 元素在 schema 文档中修改 XML 架构类型的所有实例（当作为 全局Bindings 自定义元素的子项）。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 架构原语类型。

例 38.7 “全球原语类型自定义” 显示在线全局自定义，指示代码生成器在架构中将 java.lang.Integer 用于 xsd:short 的所有实例。

例 38.7. 全球原语类型自定义

```
<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>
```

- 要修改简单类型定义 - javaType 元素会在应用到命名的简单类型定义时修改为 XML 简单类型所有实例生成的类。当使用 javaType 元素修改简单类型定义时，不要使用 xmlType 属性。

例 38.8 “用于自定义简单类型的绑定文件” 显示修改名为 zipCode 的简单类型的外部绑定文件。

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 可以应用到复杂类型定义的个别部分，方法是将其包含在 JAXB 属性自定义中。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，不要使用 xmlType 属性。

例 38.9 “在复杂类型自定义 Element 的绑定文件” 显示修改复杂类型的元素的绑定文件。

例 38.9. 在复杂类型自定义 Element 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 元素的详情请参考 第 38.6 节 “指定 Element 或属性的基本类型”。

指定转换器

Apache CXF 无法将 XML 架构原始类型转换为随机 Java 类。当您使用 javaType 元素自定义 XML 架构原语类型映射时，代码生成器会创建一个用于打包和解包自定义 XML 架构原语类型的适配器类。例 38.10 “JAXB Adapter 类” 显示了一个示例适配器类。

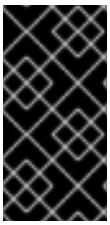
例 38.10. JAXB Adapter 类

```
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* 替换为对应的 parseMethod 属性的值和 printMethod 属性。该值必须识别有效的 Java 方法。您可以通过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*形式的简单方法名称

当您只提供简单方法名称时，代码生成器会假定 javaType 元素的 name 属性指定的类中存在方法。



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 “实现转换器”一节。

如果没有提供 parseMethod 属性的值，则 code 生成器会假定 name 属性指定的 Java 类具有构造器，其第一个参数是 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，则由 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 assumed 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 原语类型之一，则代码生成器将使用默认转换器。有关默认转换器的更多信息，请参阅 “默认原语类型转换器”一节。

生成的内容

如 “指定转换器”一节 所述，使用 javaType 自定义元素触发为每个 XML 架构小语自定义一个适配器类的生成。适配器使用模式 适配器*N* 按顺序命名。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生效的 XML 架构结构是全局定义的元素，或者定义为复杂类型的一部分。

当 XML 架构结构是全局定义的元素时，从默认方法中修改为类型生成的对象工厂方法，如下所示：

- 该方法与 @XmlJavaTypeAdapter 注释进行解码。

  该注解指示处理此元素实例时要使用的适配器类的运行时。适配器类指定为类对象。

- 默认类型由 javaType 元素的 name 属性指定的类替代。

例 38.11 “用于全局元素的自定义对象因素方法” 显示受 例 38.7 “全球原语类型自定义” 中显示的自定义影响元素的对象工厂方法。

例 38.11. 用于全局元素的自定义对象因素方法

```
@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 架构结构定义为复杂类型的一部分时，生成的 Java 属性将进行修改，如下所示：

- 属性被解码为 @XmlJavaTypeAdapter 注释。

  该注解指示处理此元素实例时要使用的适配器类的运行时。适配器类指定为类对象。

- 该属性的 @XmlElement 包含一个 类型 属性。

  type 属性的值是代表生成对象的默认基础类型的类对象。如果是 XML 架构原语类型，该类为 String。

- 属性通过 @XmlSchemaType 注释进行解码。

  该注释标识结构的 XML 架构原语类型。

- 默认类型由 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 “全球原语类型自定义” 中显示的自定义影响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;
    }

}
```

实现转换器

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，但它应调用由 parseMethod 属性和 printMethod 属性指定的方法。您需要提供运行时调用的方法实施。实施的方法必须能够使用 XML 原语类型的 lexical 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。解析方法使用 XML 数据的字符串表示，它们返回 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

当指定 Java 原语类型或 Java 原语类型 Wrapper 类时，在 javaType 元素的 name 属性中不需要指定 parse Method 属性的值。如果没有提供值，Apache CXF 运行时会替换默认转换器。

默认数据转换器使用 JAXB DatatypeConverter 类来解析 XML 数据。默认转换器也将提供进行转换所需的任何类型转换器。

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产生生成的类型，除非它们是枚举。使用简单类型定义的元素映射到 Java 原语类型的属性。

如果需要将简单类型生成到 Java 类，比如您希望使用类型替换时，会有一些实例。

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 自定义元素的 mapSimpleTypeDef 设置为 true。

添加自定义

要指示代码生成器为命名简单类型创建 Java 类，请添加 globalBinding 元素的 mapSimpleTypeDef 属性，并将其值设为 true。

例 38.13 “为 SimpleTypes 强制生成 Java 类自定义” 显示行自定义，它强制代码生成器为命名的简单类型生成 Java 类。

例 38.13. 为 SimpleTypes 强制生成 Java 类自定义

```
<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” 显示用于自定义生成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

此自定义仅影响在 全局范围 中定义的 指定 简单类型。

生成的类

为简单类型生成的类具有一个称为 **value** 的属性。**value** 属性是由 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

如果您希望根据 xsd:string 以外的模式类型枚举的类型，则必须指示代码生成器进行映射。您也可以控制生成的枚举常数。

自定义使用 jaxb:typesafeEnumClass 元素以及一个或多个 jaxb:typesafeEnumMember 元素来完成。

也可能会有实例，即代码生成器的默认设置无法为枚举器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的方式。

成员名称自定义器

如果代码生成器在生成枚举的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，则默认情况下代码生成器会生成警告，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。表 38.2 “自定义枚举 Member Name Generation 的值” 描述了 typesafeEnumMemberName 属性的值。

表 38.2. 自定义枚举 Member Name Generation 的值

值	描述
skipGeneration（默认）	指定没有生成 Java **enum** 类型并生成警告。

值	描述
generateName	指定成员名称将根据 **VALUE_*N*** 格式生成。*n* 从一个位置启动，会递增枚举的每个成员。
generateError	指定当代码生成器无法将枚举器映射到 Java 枚举类型时，它会生成错误。

例 38.17 “自定义来强制类型 Safe Member Names” 显示行自定义，它强制代码生成器生成类型安全成员名称。

例 38.17. 自定义来强制类型 Safe Member Names

```
<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 元素指定应将 XML 架构枚举元素映射到 Java enum 类型。它有两个属性，如 表 38.3 “自定义生成 Enumeration Class 的属性”。当指定 jaxb:typesafeEnumClass 元素时，它必须放在修改它的简单类型的 xsd:annotation 元素中。

表 38.3. 自定义生成 Enumeration Class 的属性

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 enumeration 是否应该映射到 Java 枚举 **类型**。默认值为 **true**。

成员自定义器

jaxb:typesafeEnumMember 元素指定 XML 架构 枚举 与 Java enum 类型常数之间的映射。您必须将一个 jaxb:typesafeEnumMember 元素用于自定义的 enumeration facet。

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 enumeration facet 的 xsd:annotation 元素中修改它。
- 它们都可放置为用于自定义 enumeration 的 jaxb:typesafeEnumClass 元素的子项。

jaxb:typesafeEnumMember 元素具有一个所需的 name 属性。name 属性指定生成的 Java enum 类型常数。这个值必须是有效的 Java 标识符。

jaxb:typesafeEnumMember 元素也具有 value 属性。该值 用于将 enumeration facet 与正确的 jaxb:typesafeEnumMember 元素相关联。value 属性的值必须与 enumeration facets' value 属性的值之一匹配。当使用外部绑定规格来自定义类型生成时，或者将 jaxb:typesafeEnumMember 元素作为 jaxb:typesafeEnumClass 元素的子项分组到 jaxb:typesafeEnumClass 元素时，需要此属性。

例子

例 38.18 “枚举类型的行自定义” 显示了一个枚举的类型，它使用在线自定义，并单独自定义枚举的成员。

例 38.18. 枚举类型的行自定义

```
<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 “使用组合映射的枚举类型的行自定义” 显示了一个枚举的类型，它使用在线自定义，并将成员的自定义组合到类自定义中。

例 38.19. 使用组合映射的枚举类型的行自定义

```
<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正常属性的固定值。使用 schema 验证时，Apache CXF 可以强制执行 schema 定义（请参阅 第 24.3.4.7 节 “架构验证类型值”）。但是，使用 schema 验证会增加消息处理时间。

映射具有固定值到 Java 的属性的另一种方法是将其映射到 Java 常量。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 常量。您还可以使用 attribute 元素在更本地化的级别上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 的固定AttributeAsConstantProperty 属性来更改此行为。将此属性设置为 true 可指示代码生成器将使用 固定 属性定义的任何属性映射到 Java 常数。

例 38.21 “自动自定义以强制生成 Constants” 显示行自定义，它强制代码生成器为具有固定值的属性生成常数。

例 38.21. 自动自定义以强制生成 Constants

```
<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” 显示用于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 属性自定义每个attribute 属性的属性映射。将此属性设置为 true 可指示代码生成器将使用 固定 属性定义的任何属性映射到 Java 常数。

例 38.23 “自动自定义以强制生成 Constants” 显示了一个无线自定义，它强制代码生成器生成带有固定值的单个属性的常量。

例 38.23. 自动自定义以强制生成 Constants

```
<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” 显示用于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。当此自定义应用到 使用固定 属性定义的属性时，属性将映射到 Java 常量，如 例 38.25 “将修复的 Value 属性映射到 Java Constant” 所示。

例 38.25. 将修复的 Value 属性映射到 Java Constant

```
@XmlAttribute
public final static type NAME = value;
```

type 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 由 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 属性值决定。

例如： 例 38.23 “自动自定义以强制生成 Constants” 中定义的属性映射，如 例 38.26 “修复了值属性映射到 Java Constant” 所示。

例 38.26. 修复了值属性映射到 Java Constant

```
@XmlRootElement(name = "widgetAttr")
public class WidgetAttr {

  ...

  @XmlAttribute
  public final static int FIXER = 7;

  ...

}
```

38.6. 指定 ELEMENT 或属性的基本类型

概述

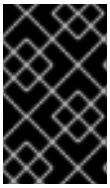
有时，您需要自定义为元素生成的对象的类，或用于定义为 XML 架构复杂类型一部分定义的属性。例如，您可能希望使用更常规的对象的类来代替简单类型。

执行此操作的一种方法是使用 JAXB 基础类型自定义。它在运行时允许开发人员指定生成的对象类，以表示元素或属性。通过基础类型自定义，您可以指定 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，也可以是默认的基础类的一般类别。它也可以是到 Java 类的 XML 架构原始类型的映射。

定制使用

要将 JAXB 基础类型属性应用到 XML Schema 结构，请使用 JAXB baseType 自定义元素。baseType custom 元素是 JAXB 属性 元素的子项，因此必须正确嵌套。

根据您要如何为 Java 对象定制 XML 架构结构映射，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 default 类映射到同一类层次结构中的另一类。您希望将 XML 架构原语类型映射到 Java 类时，将使用 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。

专用或常规默认映射

baseType custom 元素的 name 属性用于将生成的对象的类重新定义为同一 Java 类层次结构中的类。该属性指定将 XML 架构结构映射到的 Java 类的完全限定名称。指定的 Java 类必须是超级类或 Java 类的子类，代码生成器通常为 XML 架构结构生成。对于映射到 Java 原语类型的 XML 架构原语类型，打包程序类用作自定义目的的默认基础类。

例如，定义为 xsd:int 的元素使用 java.lang.Integer 作为其默认存储类。name 属性的值可以指定任何 Integer 的超类，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 “基本类型的在线自定义” 显示行自定义，它将复杂类型中的一个元素映射到 Java 对象对象。

例 38.27. 基本类型的在线自定义

```
<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 例 38.27 “基本类型的在线自定义” 中显示的自定义外部绑定文件。

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础类型的类对象。如果是 XML 架构原语类型，该类是对应的 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” 显示基于 例 38.28 “用于自定义基本类型的外部绑定文件” 中的 schema 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 的用法

javaType 元素可用于自定义如何使用 XML 架构原语类型定义元素和属性到 Java 对象。使用 javaType 元素提供比仅使用 baseType 元素的 name 属性更多的灵活性。javaType 元素允许您将

primitive 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 的运行时转换 XML 元素和 Java 对象之间的数据。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 消息的使用者时使用 JAXB 对象。

概述

JAXBContext 对象是由运行时使用的低级对象。它允许运行时在 XML 元素及其对应的 Java 表示之间转换。应用程序开发人员通常不需要使用 JAXBContext 对象。XML 数据的 marshaing 和 unmarshaling 通常由 JAX-WS 应用的传输和绑定层处理。

但是，当应用程序需要直接操作 XML 消息内容时，会有一些实例。在这两个实例中：

- 第 41.1 节 “在 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 对象是非常强化的资源，可以实例化。建议应用程序尽可能多地创建几个实例。执行此操作的一种方法是创建一个 JAXBContext 对象，该对象可以管理您的应用使用的所有 JAXB 对象，并在应用的许多部分中共享它。

JAXBContext 对象是安全线程。

使用对象工厂获取 JAXBCONTEXT 对象

JAXBContext 类提供了一个 newInstance（） 方法，显示在 例 39.1 “使用类获取 JAXB 上下文” 中，它取一个实施 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入方法的类实施的 JAXB 对象进行 marshal 和 unmarshal 数据。它还能够使用通过传入方法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，但它效率不高。实现相同目标的更有效的方法是在您的应用程序生成的对象工厂或对象因素。生成的 JAXBContext 对象可以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 newInstance（） 方法，显示在 例 39.2 “使用类获取 JAXB 上下文” 中，它使用冒号(:)顺序排列软件包名称的列表。指定的软件包应包含来自 XML 架构的 JAXB 对象。

例 39.2. 使用类获取 JAXB 上下文

```
staticJAXBContextnewInstanceStringcontextPathJAXBException
```

返回的 JAXBContext 对象将能够为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 提供以异步方式访问服务的简单机制。SEI 可以指定额外的方法，它们可用于异步访问服务。Apache CXF 代码生成器为您生成额外的方法。只需添加业务逻辑即可。

40.1. 同步调用的类型

除了通常的调用模式外，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，您可以调用一个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（继承来自 javax.util.concurrency.Future 接口）来检查响应消息是否已经到达。**

- **回调方法 - 要使用回调方法调用远程操作，您可以调用一个方法来引用回调对象（ javax.xml.ws.AsyncHandler 类型）作为其参数之一。当响应消息到达客户端时，运行时将调用 AsyncHandler 对象，并为其提供响应消息的内容。**

40.2. WSDL 用于异步示例

例 40.1 “用于同步示例的 WSDL 合同” 显示用于异步示例的 WSDL 合同。合同定义一个接口 GreeterAsync，它包含一个操作，即 greetMeSometime。

例 40.1. 用于同步示例的 WSDL 合同

```
<?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 CODE

概述

异步调用方式需要额外的 stub 代码用于 SEI 上定义的专用的异步方法。默认不生成此特殊存根代码。要切换异步功能并生成 requisite stub 代码，您必须使用 WSDL 2.0 规格中映射自定义功能。

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，并切换某些功能。在这里，自定义 用于在异步调用功能上切换。使用绑定声明指定自定义，您可以使用 jaxws:bindings 标签进行定义（其中 jaxws 前缀与 http://java.sun.com/xml/ns/jaxws 命名空间相关联）。指定绑定声明的方法有两种：

外部绑定声明

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。您在生成 stub 代码时，将绑定声明文件的位置指定为代码生成器。

内嵌绑定声明

在使用嵌入式绑定声明时，您直接嵌入了 jaxws:bindings 元素，直接包含在 WSDL 合同中，将其视为 WSDL 扩展。在这种情况下，jaxws:bindings 中的设置仅应用于 immediate 父元素。

使用外部绑定声明

例 40.2 “用于同步的绑定声明模板” 中会显示在异步调用中切换的绑定声明文件模板。

例 40.2. 用于同步的绑定声明模板

```
<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，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一个文件中, async_binding.xml 中，您需要设置 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查找外部绑定文件。

有关代码生成器的详情请参考 第 44.2 节 “cxf-codegen-plugin”。

使用嵌入的绑定声明

您还可以通过将 jaxws:bindings 元素及其关联的 jaxws:enableAsynchMapping 子项直接放入 WSDL，将绑定自定义直接嵌入到 WSDL 中。您还必须为 jaxws 前缀添加命名空间声明。

例 40.4 “用于同步映射的带有嵌入式绑定声明的 WSDL” 显示包含内嵌绑定声明的 WSDL 文件，它激活操作异步映射。

例 40.4. 用于同步映射的带有嵌入式绑定声明的 WSDL

```
<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。如果被放入为 **wsdl:portType** 元素的子项，则代码生成器会为接口中定义的所有操作创建异步方法。如果被置于一个 **wsdl:operation** 元素的子项，则代码生成器只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执行。

生成的接口

以这种方式生成存根代码后，GreeterAsync SEI（在文件 GreeterAsync.java中）被定义为 例 40.5 “带有同步调用的方法的服务端点接口”。

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，gre etMeSometime（） 还会为 greetMeSometime 操作生成两个异步方法：

- 回调方法公共Future<? >greetMeSomtimeAsyncjava.lang.StringrequestTypeAsyncHandler<GreetMeSomtime Response>asyncHandler

- 轮询方法公共Response<GreetMeSomeTimeResponse>greetMeSometimeAsyncjava.lang.StringrequestType

40.4. 使用轮询方法实施同步客户端

概述

轮询方法简单明了开发异步应用的两种方法。客户端调用名为 *OperationName*Async（） 的异步方法，并返回它轮询 响应的 Response<T > 对象。客户端在等待响应时做什么取决于应用程序的要求。处理轮询的基本模式有两种：

- 非阻塞轮询- 您可以通过调用非阻塞 Response<T>.isDone（） 方法来定期检查结果是否就绪。如果结果就绪，客户端会对其进行处理。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（），并阻止直到响应到达（可选指定超时）。

使用非阻塞模式

例 40.6 “对同步操作调用的非阻塞轮询方法” 演示了使用非阻塞轮询功能在 例 40.1 “用于同步示例的

WSDL 合同” 中定义的 greetMeSometime 操作上做出异步调用。客户端调用异步操作，并定期检查结果是否已返回。

例 40.6. 对同步操作调用的非阻塞轮询方法

```
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 “对同步操作调用的非阻塞轮询方法” 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& gt; 对象返回给客户端。Apache CXF 运行时处理从远程端点接收回复并填充 Response< GreetMeSometimeResponse > 对象的详情。

注意

运行时将请求传输到远程端点的 greetMeSometime（） 方法，并以透明方式处理调用的异步性质的详细信息。端点，因此服务实施不会担心客户端是否要等待响应的详细信息。

通过检查返回的 Response 对象的 isDone（） 检查响应是否已到达。

如果响应没有被到达，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 对象中检索它。

使用 blocking 模式

在使用块轮询模式时，Response 对象的 isDone（） 不会调用。而是在调用远程操作后立即调用 Response 对象的 get（） 方法。get（） 块到响应可用为止。

您还可以将超时限制传递给 get（） 方法。

例 40.7 “为异步操作调用阻止轮询方法” 显示使用块轮询的客户端。

例 40.7. 为异步操作调用阻止轮询方法

```
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"));
    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替代方法是实施回调类。然后，您可以调用将回调对象用作参数的异步远程方法。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响应处理。

2. 使用 ***operationName***Async（） 进行远程调用，它将回调对象用作参数，并返回 **Future<?>** 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<?& gt;** 对象的 **isDone（）** 方法，以查看远程端点是否已发送响应。

   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。接口定义了单一方法：处理 ResponseResponse<T> 重新调用 handleResponse（） 方法，以通知响应到达的客户端。例 40.8 “javax.xml.ws.AsyncHandler Interface” 展示了您必须实现的 AsyncHandler 接口的概述。

例 40.8. javax.xml.ws.AsyncHandler Interface

```
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 合同” 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：

定义一个成员变量 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，并将其分配给成员变量 回复。

实施名为 getResponse（） 的方法。

此方法是一种方便的方法，从 回复中提取数据 并返回。

实施消费者

例 40.10 “同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 合同” 中定义的 GreetMeSometime 操作进行异步调用的客户端。

例 40.10. 同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())
```

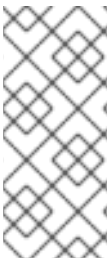
```
    {
      // Do some work
    }
    resp = callback.getResponse();
    ...
    System.exit(0);
  }
}
```

例 40.10 “同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（），它获取代理上的回调对象。

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细信息，调用回调对象的 handleResponse（） 方法，并填充 Response< GreetMeSomesponse> 对象。



注意

运行时会将请求传输到远程端点的 greetMeSometime（） 方法，并处理调用的异步性质而不了解远程端点的知识。端点，因此，服务实施不需要担心客户端是否要等待响应的详细信息。

使用返回的 Future<? > 对象的 isDone（） 方法检查响应是否从远程端点到达。

调用 callback 对象的 getResponse（） 方法以获取响应数据。

40.6. 从远程服务捕获异常

概述

在请求同步请求时，进行异步请求的用户不会接收到相同的例外。任何异步返回到消费者的异常都会在 ExecutionException 异常中嵌套。服务引发的实际异常保存在 ExecutionException 异常的 cause 字段中。

捕获异常

通过向消费者的业务逻辑传递响应的方法，会抛出远程服务生成的异常。当消费者发出同步请求时，进行远程调用的方法会引发异常。当消费者发出异步请求时，Response<T> 对象的 get（） 方法会抛出异常。在尝试检索响应消息前，使用者不会发现在处理请求时遇到错误。

与 JAX-WS 框架生成的方法不同，Response<T> 对象的 get（） 方法不会抛出用户模型异常或通用 JAX-WS 例外。相反，它会引发 java.util.concurrent.ExecutionException 异常。

获取例外详情

框架将从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，可以提取远程异常的详细信息。存储的例外可以是任何用户定义的例外，也可以是一个通用 JAX-WS 例外。

示例

例 40.11 “使用轮询方法捕获例外” 显示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。

捕获执行Exception 异常。

从异常中提取 原因 字段。

如果使用者使用回调方法，用于捕获异常的代码将放入提取服务的响应的回调对象中。

第 41 章 使用 RAW XML 消息

摘要

高级别 JAX-WS API 防止开发人员使用原生 XML 消息，将数据放入 JAXB 对象。然而，在有些情况下，最好直接访问传输的原始 XML 消息数据。JAX-WS API 提供了两个接口，它提供对原始 XML 的访问： Dispatch 接口是客户端侧接口，而 Provider 接口则是服务器端接口。

41.1. 在 CONSUMER 中使用 XML

摘要

Dispatch 接口是一个低级的 JAX-WS API，可让您直接使用原始消息。它接受并返回消息或有效负载，包括 DOM 对象、SOAP 消息和 JAXB 对象。由于它是低级 API，所以 Dispatch 接口不执行任何消息准备更高级别的 JAX-WS API 执行。您必须确保传递给 Dispatch 对象的消息或有效负载会被正确构建，并对被调用的远程操作有意义。

41.1.1. 使用模式

概述

分配对象有两个 *使用模式* ：

- 消息 模式
- 消息 Payload 模式（Payload 模式）

您为 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，这是完全指定的 SOAP 消息。invoke（） 方法还返回完全指定的 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。

在使用 JAXB 对象时，消息模式并不理想。

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，绑定级别打包程序和标头已经被剥离，并且只保留消息的正文。

在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提供相同的结果。

要指定 Dispatch 对象使用有效负载模式，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，所以它们不会被优化来使用与更高级别的消费者 API 相同的 JAXB 生成的类型。分配对象可用于以下类型对象：

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource
- “使用 JAXB 对象”一节

使用 Source 对象

Dispatch 对象接受并返回来自 javax.xml.transform.Source 接口的对象。源对象由任何绑定支持，也可以在消息模式或有效负载模式下支持。

源对象是存放 XML 文档的低级别对象。每个 Source 实施都提供访问存储的 XML 文档的方法，然后操作其内容。以下对象实施 Source 接口：

DOMSource

将 XML 消息作为文档对象模型(DOM)树保存。XML 消息存储为一组 Node 对象，这些对象通过 getNode（） 方法访问。可以使用 setNode（） 方法更新或添加到 DOM 树。

SAXSource

将 XML 消息作为 XML(SAX)对象的简单 API 保存。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

将 XML 消息作为数据流保存。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，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

当以下条件为 true 时，分配对象可以使用 javax.xml.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 envelope、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

当以下条件满足时，分配对象可以使用实现 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机制，用于处理各种来源中的数据，包括 URL、文件和字节数组。

使用 JAXB 对象

虽然 Dispatch 对象应该是低级 API，但可让您使用原始消息，但它们还允许您使用 JAXB 对象。若要搭配 JAXB 对象使用 Dispatch 对象，必须传递一个 JAXBContext，该对象可以使用的 marshal 和 unmarshal 和 unmarshal。在创建 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。您也可以将返回的消息转换为由 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，用于定义 Dispatch 对象要调用的服务。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”。

例 41.1. createDispatch（） 方法

```
publicDispatch<T&gt;createDispatchQNameportNamejava.lang.Class<T&gt;typeServic
e.ModemodeWebServiceException
```

注意

如果使用 JAXB 对象，则 createDispatch（） 的方法签名是：

```
publicDispatch<T&gt;createDispatchQNameportNamejavax.xml.bind.JAXB
ContextcontextService.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 **wsdl:port** 元素的 QName，它代表 Dispatch 对象发出调用的服务供应商。
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”。在使用 JAXB 对象时，此参数指定用于聚合和联合 JAXB 对象的 **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” 显示用于在有效负载模式下创建与 DOMSource 对象搭配使用的 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给 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，以及它所需的标题信息。

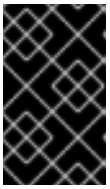
此信息可由 WSDL 文档或定义消息的 XML 架构文档提供。虽然服务提供商有很大不同，但遵循一些指南：

- 请求的根元素基于与正在调用 的操作 对应的 wsdl:operation 元素的值。

警告

如果正在调用的服务使用 doc/literal 裸机消息，则请求的根元素将基于 wsdl:operation 元素引用的 wsdl:part 元素的值。

- 请求的根元素是命名空间限定。
- 如果正在调用的服务使用 rpc/literal 消息，则请求中的顶层元素将不会被命名空间限定。



重要

顶级元素的子项可能是命名空间限定。要保证您必须检查其模式定义。

- 如果正在调用的服务使用 rpc/literal 消息，则所有顶级元素都不能为空。
- 如果正在调用的服务使用 doc/literal 消息，则消息的 schema 定义会确定任何元素是否合格。

有关服务如何使用 XML 消息的更多信息，请参阅 WS-I 基本配置集。

同步调用

对于执行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
```

在创建了 Dispatch 对象时，决定传递到 invoke（） 方法的响应和传递到 invoke（）方法的请求类

型。例如，如果您使用 createDispatch(portName, SOAPMessage.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，响应和请求可以是任何类型的 JAXBContext 对象，可以总结和解包。此外，响应和请求可以是不同的 JAXB 对象。

例 41.4 “使用 Dispatch 对象生成 Synchronous Invocation” 显示使用 DOMSource 对象对远程服务进行同步调用的代码。

例 41.4. 使用 Dispatch 对象生成 Synchronous Invocation

```
// 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* 中讨论的更高级别异步 API 一样，Dispatch 对象都可以使用轮询方法和回调方法。

使用轮询方法时，调用Async（） 方法会返回一个 Response<t& gt; 对象，该对象可以轮询来查看响应是否已到达。例 41.5 “用于轮询的 Dispatch.invokeAsync（） 方法” 显示使用轮询方法进行异步调用的方法的签名。

例 41.5. 用于轮询的 Dispatch.invokeAsync（） 方法

响应 <T> 调用AsyncTmsgWebServiceException

有关使用轮询方法进行异步调用的详情，请参考 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，调用Async（） 方法采取 AsyncHandler 实施，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的签名。

例 41.6. 使用回调的 Dispatch.invokeAsync（） 方法

```
future<?&gt; 调用AsyncTmsgAsyncHandler<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，响应的类型和请求类型决定。

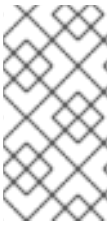
Oneway invocation

当请求没有生成响应时，使用 Dispatch 对象的 calls OneWay（）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

```
调用OneWayTmsgWebServiceException
```

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，则请求被打包为 DOMSource 对象。



注意

使用 JAXB 对象时，响应和请求可以是任何类型的 JAXBContext 对象，可以总结和解包。

例 41.8 “使用 Dispatch 对象以一个方式调用” 演示了使用 JAXB 对象对远程服务进行单向调用的代码。

例 41.8. 使用 Dispatch 对象以一个方式调用

```
// 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 接口是一个低级的 JAX-WS API，可让您实施一个服务提供商，以作为原始 XML 直接与消息一同工作。在实施提供程序接口的对象之前，消息不会打包成 JAXB 对象。

41.2.1. 消息传递模式

概述

实现提供程序接口的对象有两个 *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

在使用 *消息模式* 时，提供程序实施将适用于完整的消息。完整的消息包括任何绑定特定标头和打包程序。例如，一个使用 SOAP 绑定的提供程序实现会接收请求，作为完全指定的 SOAP 信息。从实施返回的所有响应都必须是完全指定的 SOAP 消息。

要指定 Provider 实施使用消息模式，方法是提供 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*，Provider 实施只用于消息有效负载。例如，在有效负载模式中工作的供应商实施只适用于 SOAP 消息的正文。绑定层处理任何绑定级别打包程序和标头。

在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提供相同的结果。

要指定 Provider 实施使用有效负载模式，方法是提供 java.xml.ws.Service.Mode.PAYLOAD 作为 javax.xml.ws.ServiceMode 注解的值，如 例 41.10 “指定提供程序实施使用 Payload Mode” 所示。

例 41.10. 指定提供程序实施使用 Payload Mode

```
@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 实施会使用有效负载模式。

41.2.2. 数据类型

概述

由于它们是低级对象，因此提供商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。提供程序实现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并返回从 javax.xml.transform.Source 接口派生的对象。源对象是存放 XML 文档的低级别对象。每个 Source 实施都提供了访问存储的 XML 文档和操作其内容的方法。以下对象实施 Source 接口：

DOMSource

将 XML 消息作为文档对象模型(DOM)树保存。XML 消息存储为一组 Node 对象，这些对象通过 getNode（） 方法访问。可以使用 setNode（） 方法更新或添加到 DOM 树。

SAXSource

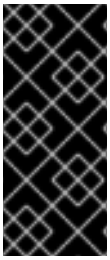
将 XML 消息作为 XML(SAX)对象的简单 API 保存。SAX 对象包含一个 InputSource 对象，其中包含原始数据和 XMLReader 对象来解析原始数据。

StreamSource

将 XML 消息作为数据流保存。数据流可以像任何其他数据流一样操作。

如果您创建您的 Provider 对象使其使用通用源对象，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

使用 Source 对象时，开发人员负责确保将所有必需的绑定打包程序添加到消息中。例如，当与预期 SOAP 消息的服务交互时，开发人员必须确保将所需的 SOAP信elope 添加至传出请求中，并且 SOAP envelope 的内容正确无误。

使用 SOAPMessage 对象

在以下情况下，供应商实施可以使用 javax.xml.SOAPMessage 对象：

- 提供程序实施使用 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 envelope、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

当以下条件满足时，提供程序实施可以使用实现 javax.activation.DataSource 接口的对象：

- 实现使用 HTTP 绑定
- 该实施使用消息模式

数据源对象提供了一种机制，用于处理各种来源中的数据，包括 URL、文件和字节数组。

41.2.3. 实施提供程序对象

概述

提供程序接口相对容易实现。它只有一个方法 调用（），必须实现。另外，它有三个简单要求：

- 实施必须具有 @WebServiceProvider 注释。

- 实施必须有一个默认的公共构造器。

- 实施必须实现输入的 Provider 接口版本。

  换句话说，您无法实现 Provider<T> 接口。您必须实现使用统一数据类型的接口版本，该版本包括在 第 41.2.2 节 “数据类型” 中列出的。例如，您可以实现一个 Provider<SAXSource> 实例。

实施提供程序接口的复杂性是在处理请求消息和构建正确响应的逻辑中。

使用消息

与基于 SEI 的更高级别的服务实现不同，提供商实现将请求接收为原始 XML 数据，并且必须发送响应作为原始 XML 数据。这要求开发人员对所实施服务所使用的消息的了解。这些详细信息通常可在描述该服务的 WSDL 文档中找到。

WS-I 基本 配置集提供有关服务使用的信息，包括：

- 请求的 root 元素基于与调用操作的 wsdl:operation 元素的 name 属性的值。

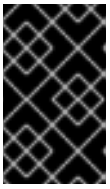
  

  警告

  如果服务使用 doc/literal 裸机消息，则请求的根元素将基于 wsdl:operation 元素引用的 wsdl:part 元素的值。

- 所有消息的根元素是命名空间合格的。

- 如果服务使用 rpc/literal 消息，则消息中的顶级元素不会具有命名空间限定性。



重要

顶级元素的子项可能具有命名空间授权，但必须检查其架构定义。

- 如果服务使用 rpc/literal 消息，则顶级元素都不能为空。
- 如果服务使用 doc/literal 消息，则消息的 schema 定义会确定任何元素是否合格。

@WebServiceProvider 注释

为了被 JAX-WS 认可为服务实施，提供程序实施必须与 @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	指定定义该服务端点的 **wsdl:port** 元素的 **name** 属性的值。
serviceName	指定包含该服务端点的 **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 服务的 WSDL 文件的 URI。

所有这些属性都是可选的，默认情况下为空。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，即 调用（），必须实现。invoke（） 方法接收发送到所实施提供程序接口类型声明的传入请求，并返回将响应消息打包为同一类型对象。例如，一个 Provider<SOAPMessage> 接口的实现会将请求接收为 SOAPMessage 对象，并返回响应作为 SOAPMessage 对象。

Provider 实施使用的消息传递模式决定了请求和响应消息包含的绑定特定信息的数量。使用消息模式实现的实现会接收所有绑定特定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中。使用有效负载模式实现的实施仅接收请求的正文。使用有效负载模式实现的 XML 文档被放入请求消息的正文中。

例子

例 41.11 “provider<SOAPMessage> 实现” 显示一个提供程序实施，它适用于消息模式下的 SOAPMessage 对象。

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：

指定以下类实施一个由 wsdl:service 元素命名为 stockQuoteReporter 的服务的 Provider 对象，其 wsdl:port 元素名为 stockQuoteReporterPort。

指定此提供者实施使用消息模式。

提供所需的默认公共构造器。

提供 调用（）方法的 实施，它采用 SOAPMessage 对象并返回 SOAPMessage 对象。

从传入 SOAP 消息正文中提取请求消息。

检查请求消息的根元素，以确定如何处理请求。

创建构建响应所需的因素。

为响应构建 SOAP 消息。

将响应返回为 SOAPMessage 对象。

例 41.12 “provider<DOMSource> 实现” 演示了在有效负载模式中使用 DOMSource 对象的提供程序实现示例。

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：

指定类实现实现服务的 Provider 对象，其 wsdl:service 元素命名为 stockQuoteReporter，其 wsdl:port 元素名为 stockQuoteReporterPort。

指定此提供程序的实现使用有效负载模式。

提供所需的默认公共构造器。

提供 调用（） 方法的实施，它采用 DOMSource 对象并返回 DOMSource 对象。

第 42 章 使用上下文

摘要

JAX-WS 使用上下文将元数据与消息传递链传递。此元数据取决于其范围，可用于实施级别代码。它还可访问在实施级别以下消息的 JAX-WS 处理程序。

42.1. 了解上下文

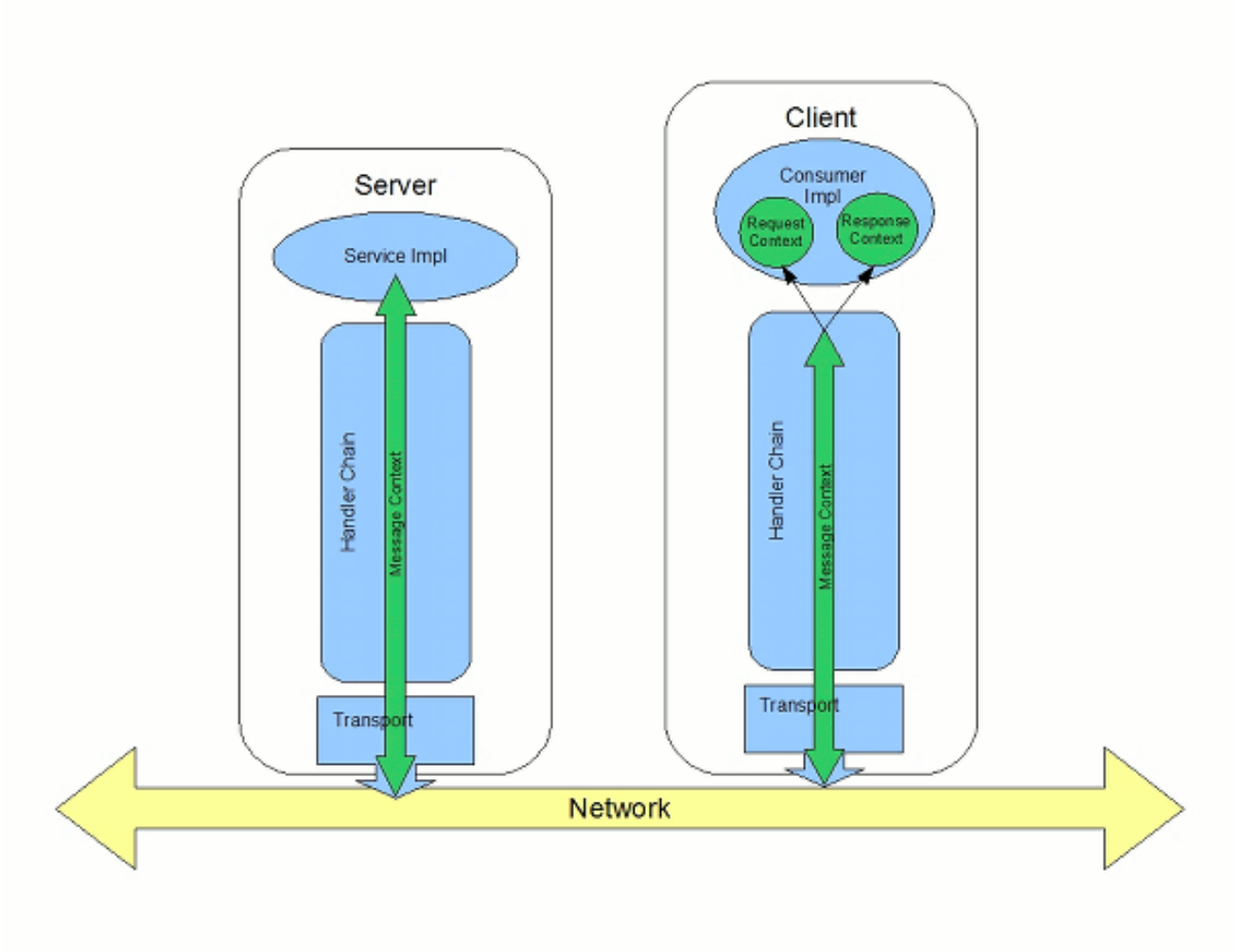
概述

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。上下文是保存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，以及用于发送消息的底层传输。例如，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。

上下文在所有级别上提供了 JAX-WS 应用。但是，它们根据您要访问上下文的消息处理堆栈中的方式有所不同。JAX-WS 处理程序实施可以直接访问上下文，并可访问他们中设置的所有属性。服务实施通过注入后访问上下文，只能访问在 APPLICATION 范围内设置的属性。消费者实施只能访问在 APPLICATION 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。当通过消息传递链的消息传递时，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



如何将属性存储在上下文中

message 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 键, Object value> 接口。将对象存储信息映射为键值对。

在消息上下文中，属性存储为名称/值对。属性的密钥是用于标识属性的 String。属性的值可以是存储在任何 Java 对象中的任何值。当值从消息上下文返回时，应用必须知道要期望的类型并相应地进行多播。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，作为对象对象，则必须转换回 UserInfo 对象。

消息上下文中的属性也具有范围。范围决定在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具有作用域。属性可以处于以下范围之一：

应用

属性范围为 APPLICATION 可供 JAX-WS Handler 实施、消费者实施代码和服务提供商实施代码使用。如果处理程序需要将属性传递到服务提供商实施，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上设置的所有属性自动作为 应用的范围。

处理程序

属性范围为 HANDLER 仅可用于 JAX-WS Handler 实施。默认情况下，存储在 Handler 实施的消息上下文中属性将限定为 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

first 参数指定属性的密钥。第二个参数指定属性的新范围。范围可以是：

- MessageContext.Scope.APPLICATION
- MessageContext.Scope.HANDLER

处理程序中上下文概述

实施 JAX-WS Handler 接口的类能够直接访问消息的上下文信息。消息的上下文信息传递到 Handler 实施的 handleMessage（）、handle Fault（） 和 close（） 方法。

处理程序实施可以访问消息上下文中存储的所有属性，而不考虑它们的范围。另外，逻辑处理程序使用专用的消息上下文，名为 LogicalMessageContext。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

服务实施可从消息上下文访问范围为 APPLICATION 的属性。服务提供商的实现对象通过 WebServiceContext 对象访问消息上下文。

更多信息请参阅 第 42.2 节 “在服务实现中使用上下文”。

消费者实施中的上下文概述

消费者实施可以间接访问消息上下文的内容。消费者实施有两个独立的消息上下文：

- 请求上下文 - 保留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

分配层传输消费者实施的消息上下文和处理程序使用的消息上下文之间的属性。

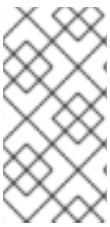
当请求从消费者实施传递到 allocated 层时，请求上下文的内容将复制到被分配层使用的消息上下文中。当从服务返回响应时，分配层将处理消息，并将适当的属性设置为其消息上下文。在分配层处理响应后，它会将所有属性范围都作为 APPLICATION 在其消息上下文中复制到消费者实施的响应上下文。

更多信息请参阅 第 42.3 节 “在消费者实现中使用上下文”。

42.2. 在服务实现中使用上下文

概述

使用 WebServiceContext 接口，将上下文信息提供给服务实施。从 WebServiceContext 对象，您可以获取填充了应用程序范围内当前请求的上下文属性的 MessageContext 对象。您可以操作属性的值，并通过响应链重新传播它们。



注意

MessageContext 接口继承自 java.util.Map 接口。其内容可以通过 Map 接口的方法操作。

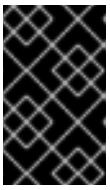
获取上下文

要在服务实现中获取信息上下文，请执行以下操作：

1. 声明类型为 WebServiceContext 的变量。

2. 使用 javax.annotation.Resource 注解来声明 变量的变量，以指明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（） 只能用于解码 @WebMethod 注释的方法。

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();
  }

  ...
}
```

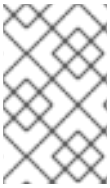
从上下文读取属性

为实现获得 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

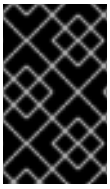
v获取对象密钥



注意

此 get（） 从 Map 接口继承。

key 参数是代表您要从上下文检索的属性的字符串。get（） 返回该属性必须转换为正确类型的对象。表 42.1 “服务实施上下文中可用的属性” 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值也会更改上下文中属性值。

例 42.4 “从服务的消息上下文获取保护” 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保护

```
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

为实现获得 MessageContext 对象后，您可以使用在 例 42.5 “MessageContext.put（） 方法” 中显示的 put（） 方法设置属性和更改现有属性。

例 42.5. MessageContext.put（） 方法

vputKkeyVvalueClassCastExceptionIllegalArgumentExceptionNullPointerException

如果消息上下文中已存在属性，put（） 方法将现有的值替换为新值，并返回旧值。如果消息上下文中不存在 属性，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保护” 显示为 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保护

```
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 “服务实施上下文中可用的属性” 列出服务实施对象中上下文访问的属性。

表 42.1. 服务实施上下文中可用的属性

属性名称	描述
org.apache.cxf.message.Message	
PROTOCOL_HEADERS[a]	指定传输特定标头信息。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。该值保存为 **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**。 该路径是主机名后面和任何查询字符串之前 URI 的部分。例如，如果端点的 URI 为 http://cxf.apache.org/demo/widgets，其路径是 **/demo/widgets**。
QUERY_STRING	指定查询（若有）附加到用于调用请求的 URI。该值存储为 **String**。 查询会显示在 URI 的末尾 **?**.例如，如果向 http://cxf.apache.org/demo/widgets?color 发出请求，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对 schema 验证消息。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故障消息。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**。 该路径是主机名后面和任何查询字符串之前 URI 的部分。例如，如果端点的 URL 为 http://cxf.apache.org/demo/widgets，基本路径为 **/demo/widgets**。
ENCODING	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。该值保存为 **org.xml.sax.InputSource** 对象。
WSDL_SERVICE	指定定义所实施服务的 **wsdl:service** 元素的限定名称。该值保存为 **QName**。
WSDL_PORT	指定用于定义访问该服务的端点的 **wsdl:port** 元素的限定名称。该值保存为 **QName**。

属性名称	描述
WSDL_INTERFACE	指定定义所实施服务的 **wsdl:portType** 元素的限定名称。该值保存为 **QName**。
WSDL_OPERATION	指定与消费者调用的操作对应的 **wsdl:operation** 元素的限定名称。该值保存为 **QName**。
javax.xml.ws.handler.MessageContext	
MESSAGE_OUTBOUND_PROPERTY	指定消息是否被出站。该值存储为 **布尔值**。**true** 指定信息是出站的。
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** >。 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** >。 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。该值保存为 **org.xml.sax.InputSource** 对象。
WSDL_SERVICE	指定定义所实施服务的 **wsdl:service** 元素的限定名称。该值保存为 **QName**。
WSDL_PORT	指定用于定义访问该服务的端点的 **wsdl:port** 元素的限定名称。该值保存为 **QName**。
WSDL_INTERFACE	指定定义所实施服务的 **wsdl:portType** 元素的限定名称。该值保存为 **QName**。
WSDL_OPERATION	指定与消费者调用的操作对应的 **wsdl:operation** 元素的限定名称。该值保存为 **QName**。
HTTP_RESPONSE_CODE	指定返回到消费者的响应代码。该值保存为 **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**。 该路径是主机名后面和任何查询字符串之前 URI 的部分。例如，如果端点的 URL 是 http://cxf.apache.org/demo/widgets，其路径是 **/demo/widgets**。
QUERY_STRING	指定查询（若有）附加到用于调用请求的 URI。该值存储为 **String**。 查询会显示在 URI 的末尾 **?**.例如，如果向 http://cxf.apache.org/demo/widgets?color 发出请求，则查询字符串为 **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 邮件标头。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. 在消费者实现中使用上下文

概述

消费者实施可以通过 **BindingProvider** 接口访问上下文信息。**BindingProvider** 实例在两个独立上下文中保存上下文信息：

-

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，一旦设置，属性会影响端口上进行的每次后续操作调用，直到显式清除属性等时间。例如，您可以使用请求上下文属性来设置连接超时或初始化要在标头中发送的数据。

- 响应上下文 通过 *响应上下文*，您可以读取响应从当前线程对最近一次操作调用设置的属性值。每个操作调用后都会重置响应上下文属性。例如，您可以访问一个响应上下文属性，以读取从上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范围中的信息才可以被消费者实施来访问。

获取上下文

上下文可以使用 javax.xml.ws.BindingProvider 接口获得。BindingProvider 接口有两种方法来获取上下文：

- getRequestContext() getRequestContext（） 方法（如 例 42.7 “getRequestContext（） 方法” 所示）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&gt;getRequestContext
```

- getResponseContext() getResponseContext（） 如 例 42.8 “getResponseContext（） 方法” 所示，将响应上下文返回为 Map 对象。returned Map 对象的内容反映了响应上下文的内容，来自当前线程中发出的远程服务的最新成功请求。

例 42.8. getResponseContext（） 方法

```
map<String, Object&gt;getResponseContext
```

由于代理对象实施 BindingProvider 接口，因此可通过广播代理对象获取 BindingProvider 对象。从 BindingProvider 对象获取的上下文仅适用于用于创建它的代理对象上调用的操作。

例 42.9 “获取消费者的请求上下文” 显示获取代理请求上下文的代码。

例 42.9. 获取消费者的请求上下文

```
// 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。该映射包含包含任意对象的 String 对象的键和值。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

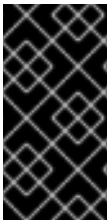
使用者上下文是存储在 java.util.Map<String, Object > 对象中的 hash map。该映射包含 String 对象和值为任意对象的键。要在上下文中设置属性，请使用 java.util.Map.put（） 方法。

虽然您可以在请求上下文和响应上下文中设置属性，但仅对请求上下文所做的更改对消息处理产生任何影响。当当前线程上完成各个远程调用时，响应上下文中的属性会重置。

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某个属性后，其值将用于所有后续远程调用。您可以更改值，然后使用更改的值。

支持的上下文

Apache CXF 在消费者实施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。该值存储为 **布尔值** 对象。
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-Addressing 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。更多信息请参阅 第 42.4 节 “使用 JMS 消息属性”。

[a] 此属性将被 HTTP 安全设置中定义的用户名覆盖。

[b] 此属性将被 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

消费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 消息标头

要从 WebServiceContext 对象获取 JMS 邮件标头属性，请执行以下操作：

1. **如 “获取上下文”一节 所述获取上下文。**
2. **使用消息上下文的 get（） 方法和参数 org.apache.cxf.transports.jms.JMSConstants.JMS_SERVER_HEADERS 获取消息上下文。**

例 42.12 “在服务实现中获取 JMS 消息标头” 显示用于从服务的消息上下文获取 JMS 邮件标头的代码：

> **例 42.12. 在服务实现中获取 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);
      ...
    }
     ...
}
```

在消费者中获取 JMS Message Header Properties

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 “客户端接收超时”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. **通过 org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RESPONSE_HEADERS 方法从响应上下文 获取 JMS 邮件标题属性。**

例 42.13 “从消费者响应标头中获取 JMS 标头” 显示用于从使用者的响应上下文获取 JMS 邮件标题属性的代码。

例 42.13. 从消费者响应标头中获取 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 “从消费者响应标头中获取 JMS 标头” 中的代码执行以下操作：

将代理转换为绑定Provider.

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 标头属性” 列出您可以检查的 JMS 标头中的标准属性。

表 42.3. JMS 标头属性

属性名称	属性类型	getter 方法
关联 ID	**字符串**	**getJMSCorralationID()**
交付模式	**int**	**getJMSDeliveryMode()**
消息过期	**long**	**getJMSExpiration()**
消息 ID	**字符串**	**getJMSMessageID()**
优先级	**int**	**getJMSPriority()**
redelivered	**布尔值**	**getJMSRedlivered()**
time Stamp	**long**	**getJMSTimeStamp()**
类型	**字符串**	**getJMSType()**
实时到实时	**long**	**getTimeToLive()**

可选标头属性

另外，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。可选属性存储为名称/值对。

示例

例 42.14 “读取 JMS 标头属性” 显示用于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：

打印消息的关联 ID 的值。

可打印消息的优先级属性的值。

打印消息的 redelivered 属性的值。

获取消息的可选标头属性列表。

获取一个 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 属性

摘要

在消费者端点中使用请求上下文，您可以设置多个 JMS 邮件标头属性和消费者端点的超时值。这些属性对单个调用有效。每次在服务代理中调用操作时，您必须重置它们。

请注意，您无法在服务中设置标头属性。

JMS 标头属性

表 42.4 “设定的 JMS 标头属性” 列出 JMS 标头中的属性，它们可通过使用者端点的请求上下文来设置。

表 42.4. 设定的 JMS 标头属性

属性名称	属性类型	setter Method
关联 ID	字符串	setJMSCorralationID()
交付模式	int	setJMSDeliveryMode()
优先级	int	setJMSPriority()
实时到实时	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 “设定的 JMS 标头属性” 中描述的适当的 setter 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，将新的 JMSMessageHeadersType 对象作为第二个参数。

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 标头属性存储在用于设置其他 JMS 标头属性的 JMSMessageHeadersType 对象中。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（） 设置属性的值字段。
4. 使用 JMSMessageHeadersType.getProperty（）.add(JMSPropertyType)，将 属性添加到 JMS 邮件标头中。
5. 重复此过程，直到所有属性都已添加到消息标头中。

客户端接收超时

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。您可以通过使用 org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RECEIVE_TIMEOUT 调用请求上下文的 put（） 方法来设置这个值，以毫秒表示您希望使用者作为第二个参数等待的时间。

示例

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 InvocationHandler。

检查，以查看 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 对象来容纳新的消息标头值。

设置正确的 ID。

将 Expiration 属性设为 60 分钟。

创建新的 JMSPropertyType 对象。

设置可选属性的值。

在消息标题中添加可选属性。

将 JMS 消息标头值设置为请求上下文。

将客户端接收超时属性设置为 1 秒。

第 43 章 编写处理程序

摘要

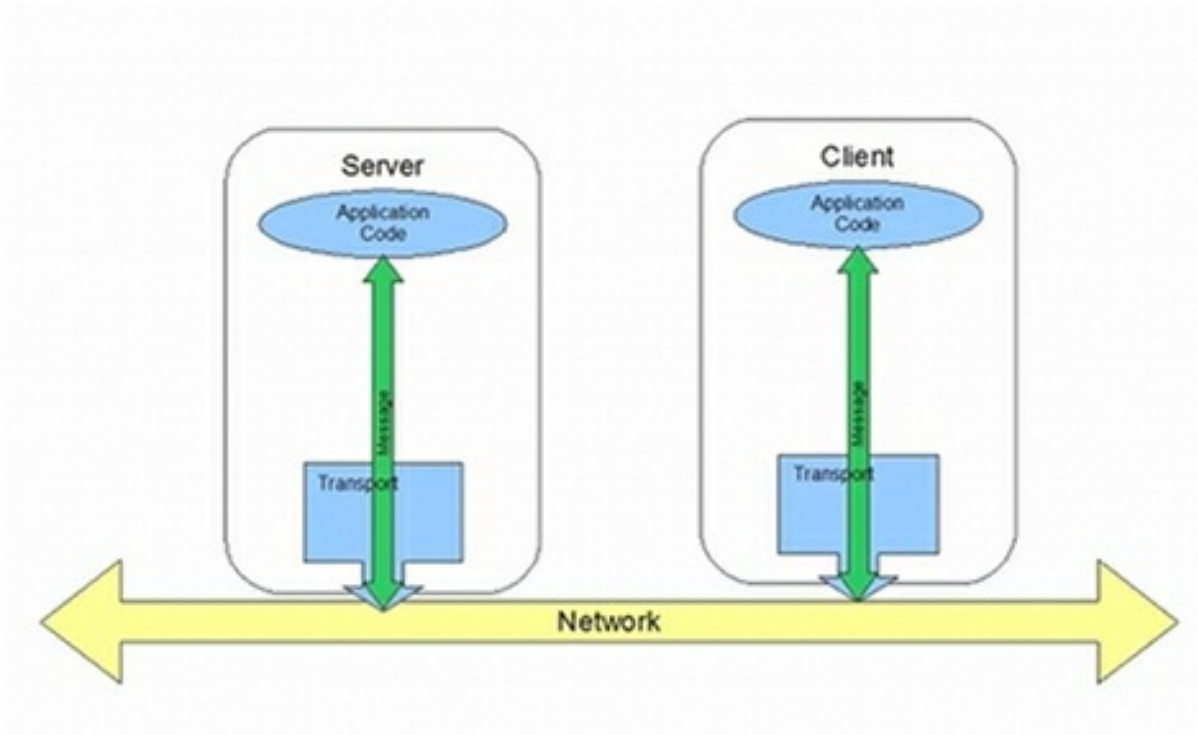
JAX-WS 提供了一个灵活的插件框架，可用于向应用添加消息处理模块。这些模块（称为处理程序）独立于应用程序级别代码，可以提供低级消息处理功能。

43.1. 处理程序：简介

概述

当服务代理在服务上调用操作时，操作的参数将传递到 Apache CXF（其中它们内置到消息中），并放置在线路上。当该服务收到消息时，Apache CXF 从线路读取消息，重新创建消息，然后将操作参数传递给负责实施操作的应用代码。当应用程序代码完成请求时，回复消息会发现其向源自该请求的服务代理的类似事件链。这在 图 43.1 “消息交换路径” 中显示。

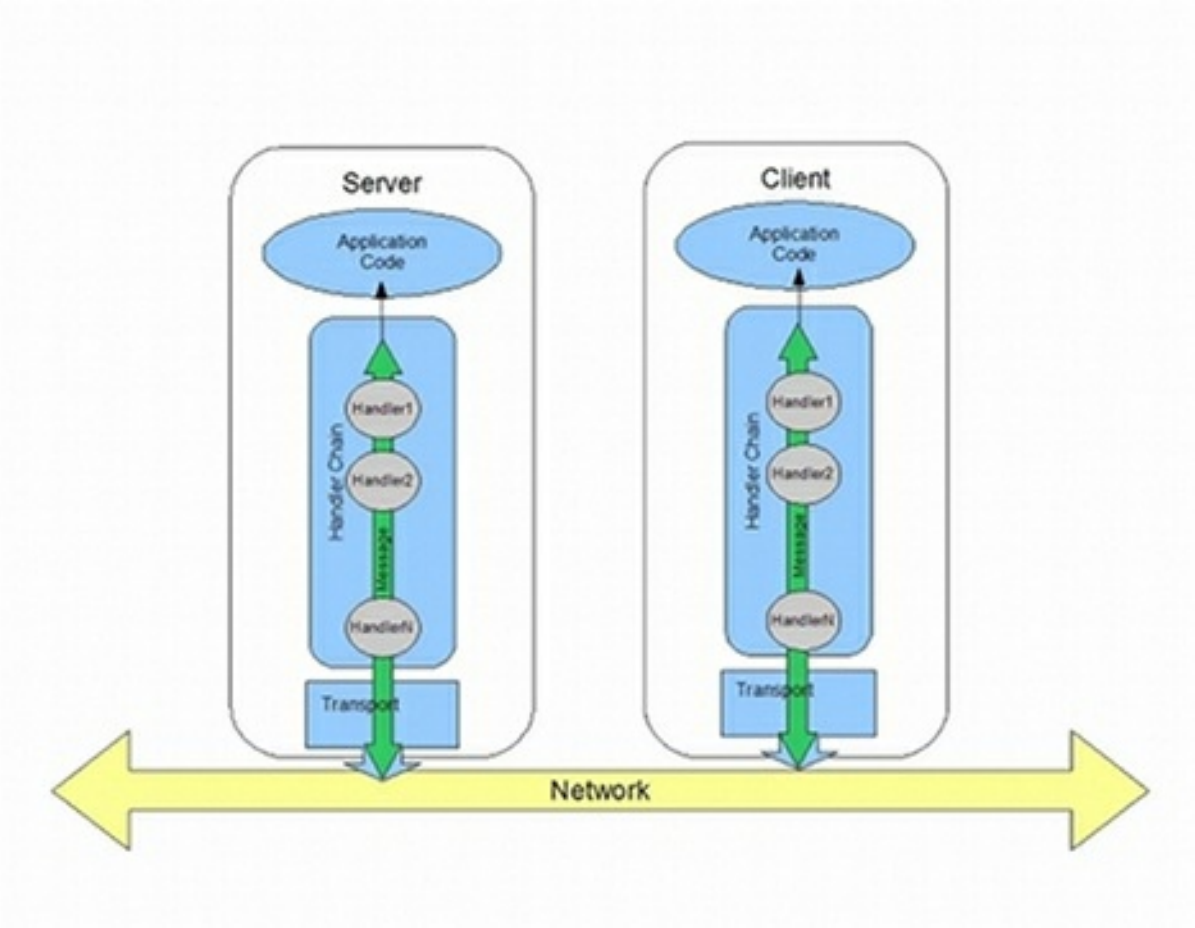
图 43.1. 消息交换路径



JAX-WS 定义了操作应用程序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，处理程序放置在应用程序级别代码和将消息放入网络的传输代码之间的遍历的链中。

图 43.2. 带有处理程序的消息交换路径



处理程序类型

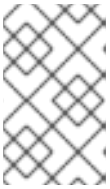
JAX-WS 规范定义了两个基本处理程序类型：

- 逻辑处理程序 逻辑处理程序可以处理消息有效负载以及消息上下文中存储的属性。例如，如果应用程序使用纯 XML 消息，逻辑处理程序可以访问整个消息。如果应用程序使用 SOAP 消息，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。

  逻辑处理程序被放在处理器链上的应用代码中。这意味着，当信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，以及消息上下文中存储的属性。例如，如果应用程序使用 SOAP 消息，协议处理程序将有权访问 SOAP 正文、SOAP 标头和任何附件。

  协议处理程序被置于处理器链上的传输中。这意味着，当从网络收到信息时，首先执行它们。当消息从应用程序代码发送到网络时，会最后执行协议处理程序。



注意

Apache CXF 支持的唯一协议处理程序特定于 SOAP。

处理程序实施

两个处理器类型之间的区别非常小，它们共享一个共同的基本接口。由于其常见父项，逻辑处理程序和协议处理程序共享了多种必须实施的方法，包括：

- **handleMessage()** handleMessage（） 方法是任意处理程序中的中央方法。它是负责处理普通消息的方法。
- **handleFault()** handleFault（） 是负责处理故障消息的方法。
- **close()** 当消息到达链的末尾时，对处理器链中执行的所有处理程序调用 close（）。它用于清理消息处理过程中消耗的所有资源。

逻辑处理程序的实现与协议处理程序的实现之间的不同：

- 所使用的特定接口

  所有处理程序都实施从 Handler 接口派生的接口。逻辑处理程序实施 LogicalHandler 接口。协议处理程序实施处理程序接口的协议特定扩展。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，以及所有使用消息内容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。

在应用程序中添加处理程序

要为应用程序添加处理器，您必须执行以下操作：

1. 确定处理程序是否将用于服务提供商、使用者或两者。

2. 确定哪种类型的处理程序最适合该作业。

3. 实施正确的接口。

   要实现逻辑处理程序，请参阅 第 43.2 节 “实施逻辑处理程序”。

   要实现协议处理程序，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 对象提供了对 消息正文 的访问，以及对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

要实现以下逻辑的手：

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 “在逻辑处理程序中处理消息” 逻辑。

3. 实施 第 43.7 节 “处理故障消息” 逻辑。

4. 在 第 43.8 节 “关闭处理程序” 完成后实现 处理程序的逻辑。

5. 为 第 43.9 节 “发布处理程序” 处理程序在销毁前实施任何逻辑。

43.3. 在逻辑处理程序中处理消息

概述

普通消息处理由 handleMessage（） 方法处理。

handleMessage（） 方法收到 LogicalMessageContext 对象，提供对消息正文和消息上下文中存储的任何属性的访问。

handleMessage（） 方法根据消息处理方式返回 true 或 false。它还可能会引发异常。

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 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 Payload 的方法” 所示）返回消息有效负载作为 LogicalMessage 对象。

> **例 43.2. 在逻辑处理程序中获取消息 Payload 的方法**
>
> ```
> LogicalMessagegetMessage
> ```

具有 LogicalMessage 对象后，就可以使用它来操作消息正文。LogicalMessage 接口（在 例 43.3 “逻辑消息冻结器” 所示）具有 getter 和 setters，以用于实际消息正文。

例 43.3. 逻辑消息冻结器

LogicalMessageSourcegetPayloadObjectgetPayloadJAXBContext上下文setPayloadObjectpayloadJAXBContextcontextsetPayloadSourcepayload

重要

消息有效负载的内容由正在使用的绑定类型决定。SOAP 绑定仅允许访问消息的 SOAP 正文。XML 绑定允许访问整个邮件正文。

将消息正文用作 XML 对象

逻辑消息的一对 getters 和 setters 将消息有效负载用作 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。对返回的对象进行的任何更改都会立即更改消息正文。

您可以使用使用单一 Source 对象的 setPayload（） 方法将消息的正文替换为 DOMSource 对象。

将消息正文用作 JAXB 对象

通过另一对 getter 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。

要使用您要执行以下操作的 JAXB 对象：

1. 获取可在消息正文中管理数据类型的 JAXBContext 对象。

   有关创建 JAXBContext 对象的详情，请参考 第 39 章 *使用 A JAXBContext 对象*。

2. 如 例 43.4 “获取消息正文作为 JAXB 对象” 所示获取消息正文。

例 43.4. 获取消息正文作为 JAXB 对象

```
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. 将更新的消息正文放回上下文，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消息上下文的实例，可访问其中存储的所有属性。处理程序能够访问 APPLICATION 范围和 HANDLER 范围的属性。

与应用程序的消息上下文一样，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，请使用 get（） 方法以及从 Map 接口继承的 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将被分配为 HANDLER 的范围。如果您希望应用程序代码能够访问您需要使用上下文的 setScope（） 方法的 属性，以显式将属性的范围设置为 APPLICATION。

有关在消息上下文中使用属性的更多信息，请参阅 第 42.1 节 “了解上下文”。

确定消息的方向

通常务必要知道消息通过处理程序链的方向。例如，您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 键从消息上下文中检索出站消息属性，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);
```

属性存储为 布尔值 对象。您可以使用对象的 booleanValue（） 方法来确定属性值。如果该属性设为 true，则消息被出站。如果该属性设为 false，则消息被入站。

确定返回值

handleMessage（） 方法如何完成其消息处理会对消息处理处理方式有直接影响。此操作可以通过执行以下操作之一完成：

1. 向 Apache CXF 运行时返回 true 信号，消息处理应正常继续。下一个处理程序（若有）对其 handleMessage（） 调用。

2. 向 Apache CXF 运行时返回错误错误信号，该信号必须停止正常消息处理。运行时进行的处理方式取决于用于 当前消息的消息 交换模式。

   请求响应消息会交换以下内容：

   a. 消息处理方向会被反转。

      例如，如果请求由服务提供商处理，则消息将停止进入服务的实施对象。相反，它会发回到发起请求的使用者的绑定。

   b. 任何驻留在处理程序链中的消息处理程序都会按照它们驻留在链中的顺序调用它们的 handleMessage（） 方法。

   c. 当消息到达处理器链的末尾时，它将被发送。

   对于单向信息会交换以下内容：

d.

消息处理停止.

e.

所有之前调用的消息处理程序都具有其 close（） 方法调用。

f.

消息被发送。

3.

引发 ProtocolException 异常异常，或者此异常的子类（此异常的子类）会发出错误消息处理的请求。运行时进行的处理方式取决于用于 当前消息的消息 交换模式。

请求响应消息会交换以下内容：

a.

如果处理程序还没有创建错误消息，则运行时会将消息嵌套在错误消息中。

b.

消息处理方向会被反转。

例如，如果请求由服务提供商处理，则消息将停止进入服务的实施对象。相反，它会发回到发起请求的使用者的绑定。

c.

任何驻留在处理程序链的任何消息处理程序都会在 其处理 方向中调用的顺序按照它们驻留在链中的顺序调用。

d.

当错误消息到达处理器链的末尾时，它将被发送。

对于单向信息会交换以下内容：

e.

如果处理程序还没有创建错误消息，则运行时会将消息嵌套在错误消息中。

f.

消息处理停止.

g.

所有之前调用的消息处理程序都具有其 close（） 方法调用。

h.
分配故障消息。

4.
引发任何其他运行时例外 - 加一个协议Exception 异常异常阻止了消息处理停止的 Apache CXF 运行时。所有之前调用的消息处理程序都具有 close（） 方法调用，并被分配异常。如果消息是请求响应消息交换的一部分，则会分配例外，使其返回到发起请求的消费者。

示例

例 43.7 “逻辑消息处理程序消息处理” 显示服务消费者使用的逻辑消息处理程序的 handleMessage（） 消息的实施。它会在请求发送到服务供应商之前处理请求。

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 representation。

获取实际消息 payload 作为 JAXB 对象。

检查以确保请求是正确类型。

如果出现这种情况，处理程序将继续处理消息。

检查 sum 的值。

如果小于 20 的阈值，则它构建响应并将其返回到客户端。

构建响应。

返回 false 以停止消息处理并返回对客户端的响应。

如果消息不是正确的类型，则抛出运行时异常。

这个异常将返回到客户端。

如果消息是入站响应，则返回 true，或者 sum 不符合阈值。

消息处理正常继续。

如果遇到 JAXB marshalling 错误，则抛出 ProtocolException。

异常会在由当前处理程序和客户端之间处理程序的 handleFault（） 方法处理后传回客户端。

43.4. 实施协议处理程序

概述

协议处理程序特定于使用中的协议。Apache CXF 提供由 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.SOAPHandler 接口。

SOAPHandler 接口（在 例 43.8 “SOAPHandler Synopsis” 所示）使用 SOAP特定的消息上下文，它提供对消息的访问作为 SOAPMessage 对象。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现一种称为 getHeaders（） 的其他方法。这个附加方法返回处理器可以处理的标头块的 QName。

流程

要实现逻辑手：

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 “在 SOAP 处理程序中处理消息” 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 第 43.8 节 “关闭处理程序” 完成后实现 处理程序的逻辑。
6. 为 第 43.9 节 “发布处理程序” 处理程序在销毁前实施任何逻辑。

实施 getHeaders（）方法

getHeaders（） 显示在 例 43.9 “SOAPHander.getHeaders（） 方法” 中，方法告知 Apache CXF 运行时，处理程序负责处理的 SOAP 标头。它返回每个 SOAP 的 outer 元素的 QNames，处理程序理解。

> **例 43.9. SOAPHander.getHeaders（） 方法**
>
> ```
> set<QName>getHeaders
> ```

对于很多情况来说，简单地返回 null 就足够了。但是，如果应用程序使用任何 SOAP 标头的 mustUnderstand 属性，那么务必要指定应用程序 SOAP 处理程序理解的标头。运行时会检查所有注册处理程序是否对标头列表了解的 SOAP 标头集，并将 mustUnderstand 属性设为 true。如果任何标记的标头都不在被理解的标头列表中，则运行时会拒绝该消息并抛出 SOAP 必须了解异常。

43.5. 在 SOAP 处理程序中处理消息

概述

普通消息处理由 handleMessage（） 方法处理。

handleMessage（） 方法收到 SOAPMessageContext 对象，以 SOAPMessage 对象和与消息关联的 SOAP 标头提供对消息正文的访问。另外，上下文提供对消息上下文中存储的任何属性的访问。

handleMessage（） 方法根据消息处理方式返回 true 或 false。它还可能会引发异常。

使用消息正文

您可以使用 SOAP 消息上下文的 getMessage（） 方法来获取 SOAP 消息。它将消息作为实时 SOAPMessage 对象返回。对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一个新的消息，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题元素。

SOAP 消息上下文提供 getHeaders（） 方法，如 例 43.10 “SOAPMessageContext.getHeaders（） Method”，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（） Method
>
> Ojbect[]getHeadersQName标头JAXBContext上下文布尔值allRoles

您可以使用元素的 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跃 SOAP 角色的 SOAP 标头。

如果没有找到标头，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

传递给逻辑处理程序的 SOAP 消息上下文是应用程序消息上下文的实例，可访问其中存储的所有属性。处理程序能够访问 APPLICATION 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 映射的子类。要访问存储在上下文中的属性，请使用 get（） 方法以及从 Map 接口继承的 put（） 方法。

默认情况下，您在逻辑处理程序内部的环境中设置的任何属性将被分配为 HANDLER。如果您希望应用程序代码能够访问您需要使用上下文的 setScope（） 方法的 属性，以显式将属性的范围设置为 APPLICATION。

有关在消息上下文中使用属性的更多信息，请参阅 第 42.1 节 “了解上下文”。

确定消息的方向

通常务必要知道消息通过处理程序链的方向。例如，您要将标头添加到传出消息，并从传入消息中剥离标头。

消息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 键从消息上下文中检索出站消息属性，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);
```

属性存储为 布尔值 对象。您可以使用对象的 booleanValue（） 方法来确定属性值。如果该属性设为

true，则消息被出站。如果该属性设为 false，则消息被入站。

确定返回值

handleMessage（） 方法如何完成其消息处理会对消息处理处理方式有直接影响。此操作可以通过执行以下操作之一完成：

1. 向 Apache CXF 运行时返回 true 信号，消息处理应正常继续。下一个处理程序（若有）对其 handleMessage（） 调用。

2. 向 Apache CXF 运行时返回 false 的信号，以停止正常消息处理。运行时进行的处理方式取决于用于 当前消息的消息 交换模式。

   请求响应消息会交换以下内容：

   a. 消息处理方向会被反转。

      例如，如果请求由服务提供商处理，则消息将停止进入服务的实施对象。相反，它将发回一个绑定，以返回源自请求的使用者。

   b. 任何驻留在处理程序链中的消息处理程序都会按照它们驻留在链中的顺序调用它们的 handleMessage（） 方法。

   c. 当消息到达处理器链的末尾时，它将被发送。

      对于单向信息会交换以下内容：

   d. 消息处理停止.

   e. 所有之前调用的消息处理程序都具有其 close（） 方法调用。

   f. 消息被发送。

3.

引发 ProtocolException 异常异常，或者此异常的子类发送一个 Apache CXF 运行时，会向 Apache CXF 运行时发出错误消息处理。运行时进行的处理方式取决于用于 当前消息的消息交换模式。

请求响应消息会交换以下内容：

a. 如果处理程序还没有创建错误消息，则运行时会将消息嵌套在错误消息中。

b. 消息处理方向会被反转。

例如，如果请求由服务提供商处理，则消息将停止进入服务的实施对象。它将发送回绑定以返回源自请求的使用者。

c. 任何驻留在处理程序链的任何消息处理程序都会在 其处理 方向中调用的顺序按照它们驻留在链中的顺序调用。

d. 当错误消息到达处理器链的末尾时，它将被发送。

对于单向信息会交换以下内容：

e. 如果处理程序还没有创建错误消息，则运行时会将消息嵌套在错误消息中。

f. 消息处理停止.

g. 所有之前调用的消息处理程序都具有其 close（） 方法调用。

h. 分配故障消息。

4. 引发任何其他运行时例外 - 加一个协议Exception 异常异常阻止了消息处理停止的 Apache CXF 运行时。所有之前调用的消息处理程序都具有 close（） 方法调用，并被分配异常。如果消息是请求响应消息交换的一部分，则分配异常以便将其返回到发起请求的消费者。

示例

例 43.12 “在 SOAP 处理程序中处理消息” 显示一个 **handleMessage（）** 实现，它将 SOAP 消息输出到屏幕。

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：

从消息上下文检索出站属性。

测试消息方向并打印适当的消息。

从上下文检索 SOAP 消息。

将消息打印到控制台。

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，它会创建处理邮件所需的所有资源。虽然您可以将用于执行此操作的所有逻辑放在处理器的构造器中，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许多可选步骤。您可以添加资源注入和其他初始化逻辑，这些逻辑将在可选步骤中执行。

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程序：

1. 处理程序的结构程序称为.
2. 由 @Resource 注释指定的所有资源都已注入。
3. 使用 @PostConstruct 注解进行解码（如果存在）的方法被调用。

注意

使用 @PostConstruct 注解分离的方法必须 无效 返回类型且没有参数。

4. 处理程序已变为 Ready 状态。

43.7. 处理故障消息

概述

在消息处理过程中引发 ProtocolException 异常时，处理程序使用 handleFault（） 方法来处理错误消息。

handleFault（） 方法根据处理程序类型接收 LogicalMessageContext 对象或 SOAPMessageContext 对象。收到的上下文授予处理程序的实现对消息有效负载的访问。

handleFault（） 方法返回 true 或 false，具体取决于故障消息处理是如何继续的。它还可能会引发异常。

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包含的有效负载。

有关使用 LogicalMessageContext 的更多信息，请参阅 第 43.3 节 “在逻辑处理程序中处理消息”。

有关使用 SOAPMessageContext 的更多信息，请参阅 第 43.5 节 “在 SOAP 处理程序中处理消息”。

确定返回值

handleFault（） 方法如何完成其消息处理会对消息处理方式进行直接影响。它通过执行以下操作之一完成：

返回 true

返回错误处理应正常继续的真正信号。将调用链中下一处理程序的 handleFault（） 方法。

返回错误

返回错误处理停止的假信号。调用在处理当前消息时调用的处理程序的 close（） 方法，并且发送错误消息。

抛出异常

引发异常将停止故障消息处理。调用在处理当前消息时调用的处理程序的 close（） 方法，并分配异常。

示例

例 43.13 “在消息处理程序中处理故障” 演示了 handleFault（） 的实施，它将消息正文打印到屏幕中。

例 43.13. 在消息处理程序中处理故障

```
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

当处理程序链的完成处理消息时，运行时将调用每个执行的处理程序的 close（） 方法。这是清理在消息处理期间由处理程序使用的所有资源，或将任何属性重置为默认状态的适当位置。

如果资源需要在单一消息交换之外保留，您不应在处理器的 close（） 方法中清除它。

43.9. 发布处理程序

概述

当处理程序被绑定为关闭的服务或服务代理时，运行时会发布处理程序。在调用处理程序的破坏性器之前，运行时将调用一个可选的发布方法。此可选版本方法可用于释放处理程序使用的任何资源，或者执行在处理程序的破坏性器中不适合的其他操作。

您不必为处理程序提供任何清理方法。

发行版本顺序

当处理程序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. 运行时调用带有 @PreDestroy 注释的方法。

   这个方法应该清理处理程序使用的所有资源。

3. 处理程序的 destructor 被调用。

43.10. 配置端点以使用处理程序

43.10.1. 程序配置

43.10.1.1. 在消费者中添加处理程序链

概述

添加处理程序链到消费者涉及明确构建处理程序链。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程序链会覆盖程序配置的处理程序链。

流程

将处理程序链添加到您执行以下操作的消费者中：

1. 创建 List<Handler& gt; 对象以容纳处理程序链。

2. 创建一个将添加到链中的每个处理程序的实例。

3. 根据运行时调用的顺序，将每个实例化处理程序对象添加到列表中。

4. 从服务代理获取 Binding 对象。

   Apache CXF 提供名为 org.apache.cxf.jaxws.binding.DefaultBindingImpl 的绑定接口实

施。

5. 使用 Binding 对象的 setHandlerChain（） 方法在代理上设置处理程序链。

示例

例 43.14 “在消费者中添加处理程序链” 显示将处理程序链添加到消费者的代码。

例 43.14. 在消费者中添加处理程序链

```
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 “在消费者中添加处理程序链” 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。

添加处理程序到 链。

从代理获取 Binding 对象作为 DefaultBindingImpl 对象。

将处理程序链分配给代理的绑定。

43.10.1.2. 在服务提供商中添加处理程序链

概述

您可以通过对带有 @HandlerChain 注释的 SEI 或实施类来向服务提供商添加处理程序链。该注释指向定义服务提供商所使用的处理程序链的 meta-data 文件。

流程

要将处理程序链添加到您执行以下操作的服务供应商中：

1. 使用 @HandlerChain 注释拒绝该供应商的实施类。
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 decorates service provider 的实施类。它指示运行时加载由其 file 属性指定的处理程序链配置文件。

该注解 的文件 属性支持两种方法来识别要加载的处理程序配置文件：

- a URL
- 相对路径名

例 43.15 “加载处理程序链的服务实现” 演示了服务提供商实施，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
{
...
}
```

处理器配置文件

处理程序配置文件使用 XML grammar 定义处理程序链，该链包含 JSR 109（用于 Java EE 版本 1.2 的 Web 服务）。这个mar在 http://java.sun.com/xml/ns/javaee 中定义。

处理程序配置文件的根元素是 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 “用于定义服务器范围处理程序链的元素” 描述 handler-chain 元素的子项。

表 43.1. 用于定义服务器范围处理程序链的元素

元素	描述
处理程序	包含描述处理程序的元素。
service-name-pattern	指定 WSDL 服务 元素的 QName，用于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，用于定义处理程序链的端点。在定义 QName 时，您可以使用 * 作为通配符。
协议绑定	指定使用处理程序链的消息绑定。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。** 有关消息绑定 URI 的更多信息，请参阅 第 23 章 *Apache CXF Binding ID*。

handler-chain 元素只需要将单一 handler 元素用作子项。不过，它可以根据需要支持许多 处理程序元素来定义完整的处理程序链。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

最后执行顺序将由将指定的处理程序排序为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。

其他子项（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服务提供程序，其 WSDL 端口 元素是指定 WSDL 服务 元素的子级。您只能在 handler 元素中使用其中一个限制的 children。

handler 元素在处理程序链中定义一个单个处理程序。其 handler-class 子元素指定类实施处理程序的完全限定名称。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。这可以通过在配置端点的元素中添加 jaxwxs:handlers 子级来完成。

重要

通过配置文件添加的处理程序链优先于程序配置的处理器链。

流程

要配置端点来加载处理器链，请执行以下操作：

1. 如果端点还没有配置元素，请添加。

   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。

   如果在多个位置中使用您的处理程序实施，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一个处理程序链。它可以作为所有 JAX-WS 端点配置元素的子项显示。这些是：

- jaxws:endpoint 配置服务供应商。
- jaxws:server 还配置服务提供商。
- jaxws:client 配置服务消费者。

您可以通过以下两种方式之一向处理器链添加处理程序：

- 添加定义实现类的 bean 元素

- 使用 ref 元素引用来自配置文件其他位置的命名 bean 元素

在配置中定义处理程序的顺序就是执行它们的顺序。如果您混合了逻辑处理程序和协议处理程序，则顺序可以被修改。运行时间会将它们排序成正确的顺序，同时保持配置中指定的基本顺序。

示例

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您的 POM。

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 提取显示了如何配置 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_11_1-00015-redhat-00002</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

指定要存储生成的 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，并从中生成经过完整注解的 Java 代码以实施服务。WSDL 文档必须具有有效的 portType 元素，但不需要包含 绑定 元素 或服务 元素。使用可选参数，您可以自定义生成的代码。

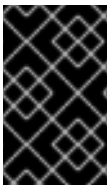
WSDL 选项

配置插件需要至少一个 wsdlOptions 元素。wsdlOptions 元素的 wsdl child 是必需的，并指定了由插件处理的 WSDL 文档。除了 wsdl 元素外，wsdlOptions 元素也可取多个可自定义 WSDL 文档的处理方式的子项。

插件配置中可以列出多个 wsdlOptions 元素。每一元素配置单个 WSDL 文档以用于处理。

默认选项

defaultOptions 元素是一个可选元素。它可用于设置所有指定的 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 元素中的值会优先使用。

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），您可以添加 extraargs 元素作为 wsdlOption 元素的子项。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** 元素来指定这些参数。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，您可以添加 **bindingFiles** 元素作为 **wsdlOption**的子项，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>
...
```

为特定的 WSDL 服务生成代码

要指定生成代码的 **WSDL** 服务的名称，您可以将 **serviceName** 元素添加为 **wsdlOption** 的子项（默认值是，在 **WSDL** 文档中为每个服务生成代码：

```
...
<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 common 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 * 字符进行通配符。例如，要从 src/main/resources/wsdl root 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，请添加 wsdlArtifact 元素作为 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>
...
```

encoding

（需要 JAXB 2.2） 来指定用于生成的 Java 文件的字符编码(Charset)，添加 编码 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>
...
```

对一个单独的进程进行分叉

您可以通过添加 fork 元素作为配置元素的子项，将 codegen 插件配置为生成代码的单独 JVM。fork 元素可设置为以下值之一：

once

派生一个新 JVM，以处理 codegen 插件配置中指定的所有 WSDL 文件。

always

派生一个新 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 插件配置为对一个独立的 JVM（即 fork 选项被设置为非错误值）进行分叉，您可以通过 additionalJvmArgs 元素为 fork 的 JVM 指定额外的 JVM 参数。例如，以下片段将 codegen 插件配置为 fork 一个 JVM，它限制为只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

下表描述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** frontend 用于生成 JAX-WS 2.1 兼容代码。**cxf** frontend（可以选择性地使用 **jaxws** frontend）为 Service 类提供额外的构造器。通过此构造器，您可以方便地指定用于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*	指定用于生成代码的零次或多个软件包名称。（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 JAXB 绑定文件。为每个绑定文件使用单独的 **-b** 标志。
-sn *serviceName*	指定要为其生成代码的 WSDL 服务的名称。默认值是在 WSDL 文档中为每个服务生成代码。

选项	解释
-reserveClass *classname*	与 **-autoNameResolution** 一起使用时，为 wsdl-to-java 定义在生成类时使用的 wsdl-to-java 的类名称。这个选项多次用于多个类。
-catalog *catalogUrl*	指定用于解析导入的 schema 和 WSDL 文档的 XML 目录的 URL。
-d *输出目录*	指定将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*	指定将编译类文件写入的目录。
-clientjar *jar-file-name*	生成包含所有客户端类和 WSDL 的 JAR 文件。指定这个选项时指定的 **wsdlLocation** 无法正常工作。
-client	为客户端主线生成起始点代码。
-server	为服务器主线生成起始点代码。
-impl	为实现对象生成起始点代码。
-all	生成所有起始点代码：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的名称。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 架构命名空间。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*	指定在使用 JAXB 数据绑定时要直接传递到 XJC 的以逗号分隔的参数列表。要获取所有可能的 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，而不是与 JAX-WS 2.1 兼容的映射。
-validate [=all|basic|none]	指示 工具在尝试生成任何代码前验证 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	在代码生成过程中禁止评论。
-allowElementReferences[=true], **-aer[=true]**	如果为 **true**，则忽略在 JAX-WS 2.2 规范第 2.3.1.2(v) 部分中提供的规则，不允许在使用 wrapper 风格的映射时引用元素。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；类似于在 JAX-WS 绑定文件中 **启用 AsyncMapping**。
-bareMethods[=*method1*,*method2*,...]	随后生成的 Java 类方法列表具有 wrapper 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，类似于在 JAX-WS 绑定文件中 **启用 MIMEContent**。
-faultSerialVersionUID *fault-serialVersionUID*	如何生成错误异常的问题。可能的值有：NONE、**TIMESTAMP**、**FQCN** 或特定数字。**默认为 NONE。**

选项	解释
-encoding *encoding*	指定生成 Java 代码时要使用的 Charset 编码。
-exceptionSuper	由 **wsdl:fault** 元素生成的用于错误 Bean 的超类（默认为 **java.lang.Exception**）。
-seiSuper *interfaceName*	为生成的 SEI 接口指定基本接口。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

synopsis

```
<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
- 用于访问该服务的客户端代码
- wrapper 和 faultan

所需配置

该插件要求 **className** 配置元素存在。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。也支持 **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 frontend 时的默认设置 **是**。
genWsdl	指示 工具在设置为 **true** 时生成 WSDL 文档。
genWrapperbean	指示 工具生成打包程序 bean，如果设为 **true** 时错误 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 classpath。

元素	描述
soap12	指定生成的 WSDL 文档在设置为 **true** 时包括 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。
serviceName	指定生成的 **service** 元素的 **name** 属性的值。

部分 VI. 开发 RESTFUL WEB 服务

本指南说明了如何使用 JAX-RS API 来实施 Web 服务。

第 45 章 RESTFUL WEB 服务简介

摘要

Representational State Transfer(REST)是一个软件架构风格，它围绕通过 HTTP 传输数据的中心，仅使用四个基本 HTTP 动词。它还允许使用任何额外打包程序（如 SOAP 信封和任何状态数据的使用）。

概述

Representational State Transfer (REST)是一种架构风格，首先由名为 Roy Fielding 的研究者进行代理解除。在 RESTful 系统中，服务器使用 URI 公开资源，客户端使用四个 HTTP 动词访问这些资源。客户端收到其处于状态的资源的表示形式。当它们访问新资源时，通常遵循链接、它们更改或转换，其状态为准。为了工作，REST 假设资源能够使用普遍的标准 getmmar 来表示。

World Wide Web 是设计基于 REST 原则的系统的最普遍性示例。Web 浏览器充当访问 Web 服务器上托管的资源的客户端。资源使用所有网页浏览器可以使用的 HTML 或 XMLmarmmarmmars 来表示。浏览器也可以轻松地跟踪新资源的链接。

RESTful 系统的优点在于它们具有高扩展性和高度灵活性。由于资源是使用四个 HTTP 动词访问的和操作，资源使用 URI 公开，资源使用标准 grammars 来表示，所以客户端不会受到更改服务器的影响。此外，RESTful 系统还可充分利用 HTTP 的可伸缩功能，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划分到资源中。
- 资源可以使用标准 URI 地址，这些 URI 可用作 hypermedia 链接。
- 所有资源仅使用四个 HTTP 动词。
  - 删除

- GET
- POST
- PUT

- 所有资源都使用 HTTP 支持的 MIME 类型提供信息。
- 协议是无状态的。
- 可缓存响应。
- 协议是分层的。

RESOURCES

资源是 REST 的中心。资源是可使用 URI 解决的信息来源。在 Web 早期，资源是大量静态文档。在现代 Web 中，资源可以是任何信息来源。例如，Web 服务可以是资源，如果使用 URI 进行访问。

RESTful 端点会 *交换* 其地址的资源的表述。表示是包含资源提供数据的文档。例如，提供客户记录访问的 Web 服务方法是资源，在服务之间交换的客户记录副本是该资源的表示。

REST 最佳实践

在设计 RESTful Web 服务时，请记住以下几点：

- 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，则每个记录应具有唯一的 URI。如果系统还提供有关分页违规和速度信息，则每种类型的资源也应具有唯一的基础。例如，可以通过 /speedingfines/*driverID* 访问速度良好，并通过 /parkingfines/*driverID*访问异常。

- 在您的 URI 中使用 nouns。

  使用 nouns 可突出显示资源是内容而不是操作的事实。URI，如 /ordering imply a action，而 /orders 则表示一个操作。

- 映射到 GET 的方法不应更改任何数据。

- 使用您的响应中的链接。

  在您的响应中放入到其他资源的链接可使客户端更轻松地跟踪数据链。例如，如果您的服务返回一组资源，则客户端使用提供的链接访问各个资源会更容易。如果没有包含链接，客户端需要额外的逻辑来遵循特定节点的链。

- 使您的服务处于无状态状态。

  要求客户端或服务维护状态信息，强制两者之间的紧密耦合。严格耦合使得升级和迁移更困难。维护状态还可以使通信错误恢复更为困难。

设计 RESTFUL WEB 服务

无论您用来实现 RESTful Web 服务的框架，应遵循以下几个步骤：

1. 定义服务将公开的资源。

   通常，服务会公开一个或多个资源，它们作为树形组织。例如，一个驱动记录服务可以分为三个资源：

   - /license/*driverID*
   - /license/*driverID*/speedingfines
   - /license/*driverID*/parkingfines

2. 定义您要能够对各个资源执行的操作。

   例如，您可能想要更新分离的地址或从驱动程序的记录中删除分页票据。

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 实现 REST

Apache CXF 为 *RESTFul Web 服务(JAX-RS)提供 Java API*实施。JAX-RS 提供了一种标准的方法，利用注释将 POJO 映射到资源。

从抽象服务定义移到使用 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别。

   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. 创建实施各个资源使用的每个 HTTP 动词的方法。

   请参阅 第 46.4 节 “使用资源方法”。

注意

Apache CXF 继续支持旧 HTTP 绑定，将 Java 接口映射到 RESTful Web 服务。HTTP 绑定提供基本功能，并有一些限制。我们鼓励开发人员更新自己的应用以使用 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 元素之间的清晰定义的映射。

Apache CXF JAX-RS 实施还支持使用 *JavaScript 对象表示法* (JSON)来交换数据。JSON 是 Ajax 开发人员使用的流行数据格式。JSON 和 JAXB 之间的数据放大由 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 Java 类，带有一个或多个 JAX-RS 注释标注。使用 JAX-RS 实施的 RESTful Web 服务的核心是根资源类。根资源类是服务所公开的资源树的入口点。它可以处理所有请求本身，也可能提供对处理请求的子资源的访问权限。

46.1. 简介

概述

使用 JAX-RS API 实施的 RESTful Web 服务以 Java 类实施的资源表示提供响应。*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，需要访问一组资源。资源类的注解提供资源 URI 和每个操作处理的 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。它通过 @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” 可通过 root 资源访问。它们通过利用 @Path 注释分离的方法实现。子资源的 @Path 注释定义相对于根资源的基本 URI 的 URI。

示例

例 46.1 “简单资源类” 显示一个简单资源类。

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 服务实施所需的最基本信息是：

- 服务资源的 URI
- 类方法如何映射到 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。该注解由 javax.ws.rs.Path 接口定义，可用于分离资源类型或资源方法。它使用字符串值作为其唯一参数。字符串值是一个 URI 模板，用于指定实现资源的位置。

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
- { }发布的参数标识符

注意

参数标识符可以包括正则表达式来更改默认路径处理。

例 46.2. URI 模板语法

```
@Path("resourceName/{param1}/../{paramN}")
```

例如，URI 模板 widget /{color}/{number} /{number} 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 取决于 @Path 注释的内容。如果放置在根资源类上，则 URI 模板是树中所有资源的根 URI，直接附加到发布该服务的 URI。如果注解分离子资源，它将相对于 root 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。

当您将方法映射到 HTTP 动词时，您必须确保映射是有意义的。例如，如果您映射了旨在提交购买订单的方法，您可以将它映射到 PUT 或 POST。把它映射到 GET 或 DELETE 会导致无法预计的行为。

46.3. 根资源类

概述

根资源类是 JAX-RS 的入口点，实施了 RESTful Web 服务。它通过 @Path 分离，用于指定由服务实施的资源的根 URI。其方法可以直接对资源实施操作，或者提供对子资源的访问权限。

要求

要使类成为根资源类，必须满足以下条件：

- 类必须与 @Path 注释进行解码。

  指定路径是服务实施的所有资源的根 URI。如果根资源类指定了其路径是 小部件，其方法之一可以实施 GET 动词，则 小部件 上的 GET 会调用该方法。如果子资源指定了其 URI 是 {id}，则子资源的完整 URI 模板为 widgets/{id}，它将处理对 URI（如 widgets/12 和 widgets/42） 的请求。

- 类必须具有公共构造器，供运行时调用。

  运行时必须能够为所有构造器参数提供值。constructor 的参数可以包括使用 JAX-RS 参数注释来分离的参数。有关参数注解的更多信息，请参阅 第 47 章 *将信息传入资源类和方法*。

- 至少一个类方法必须使用 HTTP 动词注释或 @Path 注释进行解码。

示例

例 46.3 “根资源类” 显示根资源类，提供对子资源的访问。

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” 中的类满足根资源类的所有要求。

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，使用 no 参数构造器进行简单性。

类实施资源的每个四个 HTTP 动词。

类还可以通过 getOrder（） 方法提供对子资源的访问权限。子资源的 URI（如 @Path 注释指定）是 customerservice/order/*id*。子资源由 Order 类实施。

有关实施子资源的更多信息，请参阅 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。它们具有一个 HTTP 方法注释，用于指定方法处理的请求类型。JAX-RS 在资源方法上放置几个限制。

常规限制

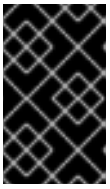
所有资源方法都必须满足以下条件：

- 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。实体参数可以是应用程序有实体提供程序的任何类型。通常它们是 JAXB 对象。



重要

资源方法 只能有一个 实体参数。

有关实体供应商的详情请参考 第 51 章 *实体支持*。

- 注释 的参数-Annotated 参数使用其中一个 JAX-RS 注释，用于指定如何从请求映射参数的值。通常，参数的值从请求 URI 的部分映射。

有关使用 JAX-RS 注解将请求数据映射到方法参数的更多信息，请参阅 第 47 章 *将信息传入资源类和方法*。

例 46.4 “带有有效参数列表的资源方法” 显示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” 显示具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
- 任何应用程序有实体提供程序的 Java 类

  有关实体供应商的详情请参考 第 51 章 *实体支持*。
- 响应 对象

  有关 响应 对象的更多信息，请参阅 第 48.3 节 “微调应用程序的响应”。
- GenericEntity<*T*> 对象

  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”。

所有资源方法将 HTTP 状态代码返回到请求者。当方法的返回类型为 void 或返回的值是 null 时，资源方法会将 HTTP 状态代码设置为 204。当资源方法返回除 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

服务可能需要由多个资源进行处理。例如，在订单处理服务最佳实践中，表示每个客户都将作为唯一资源进行处理。每个顺序也作为唯一资源进行处理。

使用 JAX-RS API，您可以将客户资源和订购资源 *实施为子资源*。子资源是一个通过根资源类访问的资源。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施：

- 子资源方法- 可为子资源实施 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。
- 子资源 locator-points 到实现子资源的类。

指定子资源

sub-resources 通过减去带有 @Path 注释的方法来指定。子资源的 URI 构造如下：

1. 将 sub-resource 的 @Path 注释的值附加到子资源的父资源 @Path 注释的值。

   父资源的 @Path 注释可能位于资源类中的方法上，该类返回包含子资源的类的对象。

2. 重复上一步，直到到达 root 资源。

3. 汇编 URI 附加到部署该服务的基本 URI 中。

例如，例 46.6 “order 子资源” 中显示的子资源的 URI 可以是 *baseURI*/customerservice/order/12。

例 46.6. order 子资源

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }
}
```

子资源方法

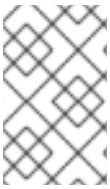
子资源方法使用 @Path 注释和 HTTP 动词注释之一进行解码。子资源方法直接负责处理使用指定的 HTTP 动词对资源发出请求。

例 46.7 “子资源方法” 显示具有三个子资源方法的资源类：

- getOrder（） 处理对 URI 匹配 /customerservice/orders/{orderId}/ 的资源的 HTTP GET 请求。
- updateOrder（） 处理 URI 匹配 /customerservice/orders/{orderId}/ 的资源的 HTTP PUT 请求。
- newOrder（） 处理 /customerservice/orders/ 中资源的 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由子资源 locator 返回的资源类。

子资源 locators

子资源 locators 不使用其中一个 HTTP 动词注解来解码，而不直接处理子资源上的请求。相反，子资源 locator 返回可以处理请求的资源类的实例。

除了没有 HTTP 动词注解外，sub-resource locators 还没有任何实体参数。子资源 locator 方法使用的所有参数都必须使用 第 47 章 *将信息传入资源类和方法* 中描述的其中一个注解。

如 例 46.8 “sub-resource 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。在与 URI 模板 /orders/{orderId}/ 匹配的 URI 上发出请求时，它会返回一个 Order 类的实例。Order 类具有 HTTP 动词注解分离的方法。PUT 请求由 updateOrder（） 方法处理。

例 46.8. sub-resource 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 locators 在运行时处理，以便它们能够支持 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 排序和信用卡订购，则 processOrder（） 方法的签名可能会保持不变。例 46.8 “sub-resource locator 返回特定类”您只需要实施两个类，即 ppOrder 和 ccOder，扩展 顺序 类。processOrder（） 的实施会根据需要什么逻辑来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}") 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 动词以及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. 确定根资源类。

   请求 URI 与带有 @Path 注释的所有类进行匹配。确定了 @Path 注释与请求 URI 匹配的类。

   如果资源类的 @Path 注释的值与整个请求 URI 匹配，则类的方法将用作输入到第三个阶段。

2. 确定对象将处理请求。

   如果请求 URI 超过所选类 @Path 注释的值，则使用资源方法的 @Path 注释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 的唯一匹配项是子资源定位器，则由子资源定位器创建的对象的资源方法与请求 URI 匹配。此阶段会重复，直到子资源方法与请求 URI 匹配。

3. 选择用于处理请求的资源方法。

HTTP 动词注释与请求中的 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确定将处理请求的资源。在某些情况下，资源由资源类实施。在其他情况下，它由使用相同 URI 模板的一个或多个子资源实施。当有多个与请求 URI 匹配的资源时，首选资源类超过子资源。

如果在资源类和子资源间排序后，如果多个资源仍与请求 URI 匹配，则使用以下条件来选择单个资源：

1. 首选使用其 URI 模板中的最多字面字符的资源。

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 /widgets/1/{color} 作为其 URI 模板的资源匹配。

注意

尾部斜杠(/)计算为字面字符。因此，/joefred/ 优先于 /joefred。

2. 首选使用其 URI 模板中的大部分变量的资源。

请求 URI /widgets/30/green 可以同时 /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。

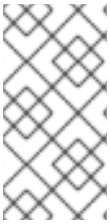
3. 优先使用包含正则表达式的大多数变量的资源。

请求 URI /widgets/30/green 可以和 /widgets/{number}/{color} 和 /widgets/{id:.}/{color}* 匹配。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，选择一个与请求 URI 匹配的资源会导致单个资源方法可以处理请求。方法通过通过资源方法的 HTTP 动词与请求中指定的 HTTP 动词匹配来确定。除了具有适当的 HTTP 动词注释外，所选的方法还必须能够处理请求中包含的请求实体，并且能够生成请求元数据中指定的正确类型的响应。



注意

资源方法可以处理的请求实体的类型由 @Consumes 注释指定。资源方法生成的响应类型可使用 @Produces 注释指定。

当选择一个资源时，可以生成多个可以处理请求的方法，用于选择处理请求的资源方法：

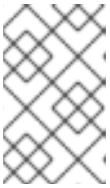
1. 优先选择资源方法而不是子资源。
2. 优先使用子资源方法，而不是子资源定位器。
3. 首选使用 @Consumes 注释中最常用的值的方法和 @Produces 注释。

   例如，具有注释 @Consumes("text/xml") 的方法优先于带有注释 @Consumes("text/*") 的方法。这两种方法都可以优先于不使用 @Consumes 注释的方法，或注解 @Consumes("*/*")。
4. 最好与请求正文实体的内容类型匹配。

   注意

   请求正文实体的内容类型在 HTTP Content-Type 属性中指定。
5. 最好与内容类型直接匹配作为响应的方法。



注意

HTTP Accept 属性中接受的内容类型作为响应指定。

自定义选择过程

在某些情况下，开发人员在选择多个资源类的方式中对算法有某种程度的限制。例如，如果一个给定资源类已匹配，如果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别。

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（在 例 46.9 “用于自定义资源选择的接口” 所示）必须具有需要实施的方法。其中一个比较两个资源类，另一个则比较两种资源方法。

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);
}
```

自定义实现选择两个资源之间的如下：

- **如果第一个参数与第二个参数相比，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，则运行时将继续使用默认选择算法

您可以通过将 resourceComparator 子添加到服务的 jaxrs:server 元素，注册自定义资源 Comparator 实施。

第 47 章 将信息传入资源类和方法

摘要

JAX-RS 指定多个注解，供开发人员控制传递给资源的信息来自的位置。该注解符合 URI 中的常见 HTTP 概念，如列表参数。标准 API 允许注解用于方法参数、bean 属性和资源类字段。Apache CXF 提供了一个扩展，允许将一系列参数注入到 Bean 中。

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，它们的值可以通过运行时注入它们。注入的特定数据由一组注解指定，如 第 47.2 节 “使用 JAX-RS API”。

在注入数据时，JAX-RS 规范对数据进行了一些限制。它还对对象类型施加一些限制，以请求数据可以注入到其中。

数据注入后

当请求因为请求实例化时，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* 中所述，这些对象可以是一个使用 @Path 注释的根本原因或从子资源 locator 方法返回的对象。

支持的数据类型

数据可注入的特定数据类型取决于用来指定注入数据的源的注解。但是，所有注入注解都支持以下一组数据类型：

- 原语，如 int、char 或 long
- 具有 buildor 的对象，它接受单个 String 参数
- 具有接受单个 String 参数的静态 valueOf（） 方法的对象

- list<*T*>、Set<*T*> 或 SortedSet< *T* > 对象，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，则会在讨论注解时突出显示区别。

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解。注解可以被分成三个不同的类型：

- 第 47.2.2 节 “注入请求 URI 中的数据”
- 第 47.2.3 节 “注入 HTTP 邮件标头中的数据”
- 第 47.2.4 节 “注入 HTML 格式的数据”

47.2.2. 注入请求 URI 中的数据

概述

设计 RESTful Web 服务的最佳实践之一是，每个资源应具有唯一的 URI。开发人员可以使用这个原则来为底层资源实施提供良好的信息。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。开发人员还可利用查询和列表参数，将信息馈送到资源实施中。

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 注解来完成。@PathParam 注释具有一个用于标识要注入数据的 URI 模板变量的单个参数。

在 例 47.1 “注入 URI 模板变量的数据” 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. 注入 URI 模板变量的数据

```
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后一个部分。

- **List<PathSegment>**

  该值将是与指定模板参数匹配的路径网段对象列表。

- **原语，如 int、char 或 long**

- **具有 buildor 的对象，它接受单个 String 参数**

- **具有接受单个 String 参数的静态 valueOf（） 方法的对象**

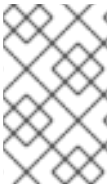
使用查询参数

在 Web 上传递信息的一种常见方法是使用 URI 中的 *查询参数*。查询参数出现在 URI 的末尾，并通过问号(?)与 URI 的资源位置部分隔开。它们由一个或多个名称值对组成，其中 name 和 value 用等号(=)

分开。当指定多个查询参数时，对通过分号（）或 符号（和）相互分离。 例 47.2 “带有查询字符串的 URI” 显示带有查询参数的 URI 的语法。

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，但不能两者。

javax.ws.rs.QueryParam 注解提取查询参数的值，并将其注入 JAX-RS 资源。该注释采用单个参数来标识从中获取并注入到指定字段、bean 属性或参数中的查询参数。@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” 显示一个资源方法，它将查询参数 id 的值注入到方法的 id 参数中。

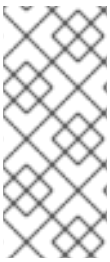
例 47.3. 使用查询参数中的数据的资源方法

```
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 HTTP POST 到 /monstersforhire/daikaiju?id=jonas the updateMonster（） 方法 的类型，并将 id 设置为 jonas。

使用列表参数

URI 列表参数（如 URI 查询参数）是 name/value 对，可以提供附加信息选择资源。与查询参数不同，列表参数可能会出现 URI 中的任何位置，它们使用分号(;)与 URI 的层次路径片段分离。/mostersforhire/daikaiju;id=jonas 有一个称为 id 和 /monstersforhire/japan;type=daikaiju/flying;wingspan=40 有两个列表参数，即 type 和 wingspan。



注意

在计算资源 URI 时，不会评估列表参数。因此，用于定位适当的资源以处理请求 URI /monstersforhire/japan;type=daikaiju/flying;wingspan=40 是 /monstersforhire/japan/flying。

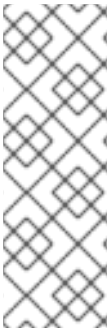
matrix 参数的值通过 javax.ws.rs.MatrixParam 注解注入字段、参数或 bean 属性。该注释采用单个参数来标识从中提取值并注入到指定字段、bean 属性或参数的 matrix 参数的名称。@MatrixParam 注释支持 “支持的数据类型”一节 中描述的类型。

例 47.4 “使用列表参数中的数据的资源方法” 显示资源方法，它将列表参数值注入方法参数和 id。

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 HTTP POST 到 /monstersforhire;type=daikaiju;id=whale，updateMonster（） 方法 的类型 被设置为 daikaiju，并将 id 设置为 whale。



注意

JAX-RS 一次性评估 URI 中的所有列表参数，因此无法对 URI 中的列表参数位置实施约束。例如 /monstersforhire/japan;type=daikaiju/flying;wingspan=40 , /monstersforhire/japan/flying;type=daikaiju;wspan=40, 和 /monstersforre/japan;type=daikaiju;wingspan=40/flying 均被视为使用 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 被解码。因此，URI /monster/night%20stalker 和 URI /monster/night stalker 具有等同的。自动 URI 解码可让您轻松在 ASCII 字符集外发送字符作为参数。

如果您不想自动解码 URI，您可以使用 javax.ws.rs.Encode 注解来 取消激活 URI 解码。该注解可用于在以下级别上取消激活 URI 解码：

- **类级别 - 使用 @En code 注释处理类可取消激活类中所有参数、字段和 bean 属性的 URI 解码。**
- **方法级别 - 使用 @En code 注释处理方法可取消激活类所有参数的 URI 解码。**
- **参数/字段级别-使用 @En code 注释停用类所有参数的 URI 解码参数或字段。**

例 47.5 “禁用 URI 解码” 显示 getMonster（） 方法不使用 URI 解码的资源。addMonster（） 方法只禁用 type 参数的 URI decoding。

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，会产生一个错误，则将生成 WebApplicationException 异常嵌套原始异常。WebApplicationException 异常的状态设置为 404。

47.2.3. 注入 HTTP 邮件标头中的数据

概述

在正常情况下，在请求消息中传递 HTTP 标头会传递消息的通用信息，如何在传输中处理它，以及有关预期响应的详情。虽然几种标准标头通常被识别和使用，但 HTTP 规范允许任何 name/value 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/响应序列间共享静态信息。JAX-RS API 提供了一个注释，将来自 Cookie 的数据直接注入到资源实施中。

注入 HTTP 标头的信息

javax.ws.rs.HeaderParam 注解用于将 HTTP 标头字段的数据注入到参数、字段或 bean 属性中。它具有一个参数，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 最旧的Date 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，它们传递到第一个请求的资源实施。在第一个请求后，cookie 在提供程序和消费者之间通过每条消息。只有消费者生成请求后才能更改 Cookie。Cookie 通常用于在多个请求/响应序列间维护会话，存储用户设置和其他可保留的数据。

javax.ws.rs.CookieParam 注解从 cookie 的项中提取值，并将其注入资源实施中。它取一个参数，用于指定要从中提取值的 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 的实体也可以是一个 Cookie 对象。

例 47.6 “注入 cookie” 显示将 handle cookie 的值注入到 CB 类中的字段的代码。

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 HTTP 消息注入注解注入数据时出现一个错误，则生成 WebApplicationException 异常嵌套原始异常。WebApplicationException 异常的状态设置为 400。

47.2.4. 注入 HTML 格式的数据

概述

HTML 表单是一种从用户获取信息的简单方法，也易于创建。表单数据可用于 HTTP GET 请求和 HTTP POST 请求：

GET

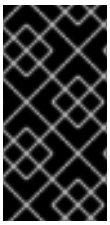
作为 HTTP GET 请求的一部分发送表单数据时，数据会作为一组查询参数附加到 URI。“使用查询参数”一节 中讨论从查询参数注入数据。

POST

当将数据作为 HTTP POST 请求的一部分发送时，数据会被放入 HTTP 消息正文中。可使用支持表单数据的常规实体参数处理表单数据。它还可通过使用 @FormParam 注释来提取数据并将组件注入资源方法参数来处理。

使用 @FormParam 注释来注入表单数据

javax.ws.rs.FormParam 注解从表单数据中提取字段值，并将值注入资源方法参数。该注释采用单个参数来指定从中提取值的字段键。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指出可以将 @FormParam 注释放在不同的字段、方法和参数上。但是，@FormParam 注释仅在放置在资源方法参数上时才有意义。

示例

将表单数据注入资源方法参数 显示将表单数据注入参数的资源方法。这个方法假定客户端表单包含三个字段：标题、标签 和 正文，其中包含字符串数据。

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

要为更可靠的服务实现提供，您可能需要确保将任何可选参数设置为默认值。这在输入长 URI 字符串以来从查询参数和列表参数中获取的值特别有用。您可能还想为从 cookie 中提取的参数设置默认值，因为请求的系统可能没有正确的信息来构造带有所有值的 cookie。

javax.ws.rs.DefaultValue 注解可以和以下注入注解结合使用：

- @PathParam
- @QueryParam
- @MatrixParam
- @FormParam
- @HeaderParam
- @CookieParam

@DefaultValue 注释指定在请求中不存在与注入注释对应的数据时将使用的默认值。

语法

设置参数的默认值的语法 显示使用 @DefaultValue 注释的语法。

设置参数的默认值的语法

```
import javax.ws.rs.DefaultValue;
  ...
  void resourceMethod(@MatrixParam("matrix")
                      @DefaultValue("value)
                      int someValue, ... )
  ...
```

该注释必须在参数、bean 或 字段之前使用，它将起作用。相对于带注入注解的 @DefaultValue 注释的位置无关紧要。

@DefaultValue 注释使用一个参数。此参数是根据注入注解提取正确的数据，则将注入字段的值。该值可以是任意 String 值。该值应当与关联字段的类型兼容。例如，如果关联的字段是 int，则默认值 blue 会导致异常。

处理列表和集合

如果注解参数的类型，an 或字段是 List、Set 或 SortedSet，则生成的集合将具有从提供的默认值映射的单个条目。

示例

设置默认值 显示使用 @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，调用 设置默认值 中的 getMonster（） 方法。这个方法需要两个查询参数，即 id 和 type，并附加到 URI。因此，使用 URI *baseURI*/monster?id=1&type=fomóiri 的 GET 请求会返回 Fomóiri，其 ID 为 1。

由于 @DefaultValue 注释都放在这两个参数上，因此如果查询参数被省略，则 getMonster（） 方法可以正常使用。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，服务器端的一个通用模式是创建一个 Java Bean，以封装以形式接收的所有数据（以及来自其他参数和 HTML 标头的数据）。通常，创建此 Java bean 将是一个两个步骤：资源方法通过注入（例如，将 @FormParam 注释添加到其方法参数），然后调用 bean 的构造器，以表单数据传递。

使用 JAX-RS 2.0 @BeanParam 注释，可以在一个步骤中实施此模式。表单数据可以直接注入到 bean 类的字段，并且 bean 本身由 JAX-RS 运行时自动创建。例如，这最易于说明。

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，参数目标是唯一可与所有资源类生命周期一起使用的目标类型。其他类型的目标仅限于每个请求的生命周期。这个情况在 表 47.1 “@BeanParam Injection Targets” 中进行了概述。

表 47.1. @BeanParam Injection Targets

目标	资源类别生命周期
参数	All
字段	per-request（默认）
METHOD （属性）	per-request（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以 Java Bean 捕获表单数据（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，用于排序来自 furniture 网站的表数量。当订购表单后，表单值将注入到 orderTable 方法的参数中，使用注入的表单数据显式创建 TableOrder 类的实例。

带有 BeanParam 注解的示例

上一示例可以重构为利用 @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重构如下：

```
// 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，您可以使用单个 @BeanParam 注释替换资源方法签名中的所有 @FormParam 注释，如所示。现在，当表单发布到 orderTable 资源方法中时，JAX-RS 运行时会自动创建 TableOrder 实例，orderBean，并注入 bean 类上参数注解指定的所有数据。

47.3. 参数转换器

概述

使用参数转换器时，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，您可以扩展 JAX-RS 运行时，使它能够将参数 String 值转换为目标类型。

自动转换

参数作为 String 的实例接收，因此您可以将其直接注入到字段、bean 属性和 String 类型的方法参数。此外，JAX-RS 运行时还具备将参数字符串自动转换为以下类型的功能：

1. 原语类型。

2. 具有可接受单个 字符串 参数的构造器的类型。

3. 具有名为 valueOf 或 来自String 的静态方法类型，它带有一个 String 参数，该参数返回类型为实例。

4. 如果 T 是 2 或 3 中描述的类型之一，请列出 & lt;T >、Set< T > 或 SortedSet <T>。

参数转换器

要将参数注入自动转换没有涵盖的类型，您可以为类型定义自定义 *参数转换器*。参数转换器是一种 JAX-RS 扩展，可让您定义从 String 转换到自定义类型，并在反向方向从自定义类型转换到 字符串。

factory pattern

JAX-RS 参数转换器机制采用工厂模式。因此，您不必直接注册参数转换器，而是必须根据 需要注册参数转换器（类型、Java x.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);
}
```

-

要实现您自己的参数，您必须实现这个接口，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[]);
}
```

要实施自己的 ParamConverterProvider 类，您必须实施这个接口，覆盖 getConverter 方法，它是一种工厂方法来创建 ParamConverter 实例。

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（使它可用于您的应用，您必须使用 @Provider 注释标注您的实施类），如下所示：

```
// Java
...
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

@Provider
public class TargetTypeProvider implements ParamConverterProvider {
    ...
}
```

此注解可确保您的参数转换器供应商会在部署的扫描阶段自动注册。

示例

以下示例演示了如何使用 ParamConverterProvider 和 ParamConverter（将参数字符串转换为 TargetType 类型）和 ParamConverter ：

```
// 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，您已为 **TargetType** 定义了参数转换器，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

如果您为您的参数指定默认值（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），还是只在需要时转换默认值(lazy conversion)。要选择 lazy conversion，将 @ParamConverter.Lazy 注解添加到目标类型。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 提供标准的 JAX-WS 注入机制的扩展，允许开发人员使用单一注解替换一系列注入注解。单个注解放置在一个 bean 上，其中包含使用注解提取的数据字段。例如，如果资源方法预期一个请求 URI 中包含名为 id 的三个查询参数，则键入，它可使用一个 @QueryParam 注释来将所有参数注入到 Bean 中并包含相应字段。

注意

请考虑使用 @BeanParam 注释（自 JAX-RS 2.0 起可用）。标准化的 @BeanParam 方法比专用 Apache CXF 扩展更灵活，因此建议的替代方案。详情请查看 第 47.2.6 节 “将参数注入 Java Bean”。

支持的注入注解

这个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

要指定注解将使用串行注入程序，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("") 指定一系列 URI 模板变量被序列化为 bean。
2. 确保注解的参数是一个 bean，其中包含与要注入的值匹配的字段。

示例

例 47.7 “将查询参数注入 Bean” 显示将多个 Query 参数注入到 bean 的示例。资源方法预期请求 URI 包含两个查询参数： type 和 id。其值注入了 Monster an 的对应字段。

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 请求要求至少有 HTTP 响应代码返回给使用者。在很多情况下，可以通过返回普通的 JAXB 对象或通用性对象来满足请求。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所需 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 返回对象之间的映射和返回到 RESTful 消费者之间的映射并非一。至少，除任何响应实体正文外，RESTful 消费者还需要返回有效的 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接受的 MIME 类型生效。

为了解决将 Java 对象映射到 RESTful 响应消息的问题，可以返回四种 Java 结构类型：

- 第 48.2 节 “返回普通 Java 结构” 使用由 JAX-RS 运行时确定的 HTTP 返回代码返回基本信息。
- 第 48.2 节 “返回普通 Java 结构” 通过由 JAX-RS 运行时确定的 HTTP 返回代码返回复杂信息。
- 第 48.3 节 “微调应用程序的响应” 通过编程方式确定的 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” 通过由 JAX-RS 运行时确定的 HTTP 返回代码返回复杂信息。GenericEnitity 对象为运行时组件提供序列化数据的更多信息。

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，运行时使用返回的对象的 Java 类决定 MIME 类型信息。该运行时还决定要发送到消费者的适当 HTTP 返回代码。

可返回的类型

资源方法可以返回 void 或提供实体写器的任何 Java 类型。默认情况下，运行时具有以下供应商：

- Java 原语
- Java 原语的数字表
- 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作者。

MIME 类型

运行时首先检查 @Produces 注释的资源方法和资源类，从而确定返回的实体的 MIME 类型。如果找到了一个，它将使用注解中指定的 MIME 类型。如果找不到资源实现指定的，它依赖于实体提供商来确定正确的 MIME 类型。

默认情况下，运行时分配 MIME 类型，如下所示：

- Java 原语及其数字表示被分配一个 MIME 类型的应用程序 /octet-stream。
- 为 JAXB 对象分配 MIME 类型 application/xml。

应用程序可以通过实施自定义实体提供程序来使用其他映射，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，而不会抛出异常。状态代码设置如下：

- 204（无内容）- 资源类型为 void
- 204（无内容）- 返回的实体的值是 null
- 200(OK)- 返回的实体的值是 null

如果在资源方法完成返回状态代码前抛出异常，如 第 50 章 *处理异常* 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

RESTful 服务通常需要更精确地控制返回到消费者的响应，而在资源方法返回普通 Java 结构时要被允许。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 对象嵌套了代表消费者返回实体的对象。使用 Response Builder 类作为工厂来实例化响应对象。

ResponseBuilder 类也有许多用于操作响应的元数据的方法。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

响应 类具有受保护的构造器，因此无法直接实例化。它们使用由 Response 类包含的 ResponseBuilder 类创建。ResponseBuilder 类是所有信息的拥有者，它们将封装在从它创建的响应中。ResponseBuilder 类也具有负责设置消息上的 HTTP 标头属性的所有方法。

Response 类提供一些方法来轻松设置正确的响应代码并嵌套该实体。各个常见响应状态代码有方法。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将信息设置为关联的响应构建器的版本。

ResponseBuilder 类的 build（） 方法会返回一个响应对象，其中包含调用方法时在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();
```

当获得响应构建器时，您无法访问多个步骤的实例。您必须将所有操作的字符串到一个方法调用中。

- 使用 Apache CXF 特定 ResponseBuilderImpl 类。此类允许您直接与响应构建器一起使用。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” 显示如何使用 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

您也可以简单地将 ResponseBuilder 返回的 Response Builder 方法分配到 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。

如需有关 Apache CXF ResponseBuilderIml 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 类提供了处理 RESTful 服务所需的更常见响应的快捷方式方法。这些方法使用提供的值或默认值来处理正确的标头。它们也会在适当时处理填充实体正文。

为成功请求创建响应

当成功处理请求时，应用程序需要发送响应，以确认请求已实现。该响应可以包含实体。

成功完成响应时最常见的响应是 OK。OK 响应通常包含与请求对应的实体。Response 类具有超载的 ok（） 方法，它将响应状态设置为 200，并将提供的实体添加到已括起的响应构建器中。ok（） 方法有五个版本。最常用的变体是：

- response .ok（）- 创建状态 200 和空实体正文的响应。
- response .ok（java.lang.Object 实体）- 创建具有 200 状态的响应，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媒体类型。

创建具有 200 响应的响应 演示了创建具有 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();
```

对于请求者不是预期实体正文，可能更适合发送 204 No Content 状态，而不是 200 OK 状态。Response.noContent（） 方法将创建相应的响应对象。

创建具有 204 状态的响应 显示创建具有 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();
```

创建重定向响应

Response 类提供了处理三种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将消费者重定向到新资源以处理请求时，303 See Other status 非常有用。

Response 类 seeOther（） 方法创建了具有 303 状态的响应，并将新资源 URI 放置到消息的 Location 字段中。seeOther（） 方法采用单个参数，它将新 URI 指定为 java.net.URI 对象。

304 未修改

304 Not Modified 状态可用于不同的内容，具体取决于请求的性质。它可用于表示请求的资源自以前的 GET 请求以来没有变化。它还可用来表明修改资源的请求不会导致资源被改变。

Response classes notModified（） 方法创建了具有 304 状态的响应，并设置 HTTP 消息修改的日期属性。notModified（） 方法有三个版本：

- notModified
- notModifiedjavax.ws.rs.core.Entitytag
- notModifiedjava.lang.Stringtag

307 临时重定向

当请求的资源需要将使用者定向到新资源时，307 临时重定向 状态非常有用，但希望消费者继续使用该资源来处理将来的请求。

Response 类 temporaryRedirect（） 方法创建了具有 307 状态的响应，并将新资源 URI 放置到消息的 Location 字段中。temporaryRedirect（） 方法采用单一参数，它将新 URI 指定为 java.net.URI 对象。

创建具有 304 状态的响应 显示创建具有 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 无可接受的状态以及包含可接受资源类型列表的实体正文。

创建具有 500 个状态的响应 显示创建具有 500 状态的响应的示例。

创建具有 500 个状态的响应

```
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 类方法提供了为常见情况创建响应的短差。当您需要解决更复杂的情况，如指定缓存控制指令、添加自定义 HTTP 标头或发送未由 Response 类处理的状态时，您需要使用 ResponseBuilder 类方法来填充响应，然后才能使用 build（） 方法来生成响应对象。

如 “获取响应构建器”一节 所述，您可以使用 Apache CXF ResponseBuilderImpl 类来创建可直接操作的响应构建器实例。

添加自定义标头

使用 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（） 方法在消息上设置多个标头。

在响应中添加标头 显示向响应中添加标头的代码。

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

使用 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 cookies。每个 Cookie 都存储在 javax.ws.rs.core.NewCookie 对象中。NewCookie 类的最简单方法是使用两个参数：

- name-a 指定 Cookie 名称的字符串
- value-a 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 显示将 Cookie 添加到响应中的代码。

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（） 方法将擦除已经与响应关联的 cookie。

设置响应状态

当您想要返回除 Response 类帮助程序方法所支持的其中一个状态以外的状态时，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。需要一个 int 来指定响应代码。另一个需要 Response.Status 对象来指定响应代码。

Response.Status 类是存储在 Response 类中的枚举者。它含有大多数定义的 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（） 方法允许您设置响应上的缓存控制标头。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。如果指令是简单的 on 或 off 值，则 setter 方法采用 布尔值。如果指令需要数字值，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

有时，应用程序需要对所返回对象的 MIME 类型进行更多控制，或者用于序列化响应的实体提供商。JAX-RS javax.ws.rs.core.GenericEntity<T > 类通过提供指定代表该实体的通用类型的机制，提供对实体的串行控制。

使用通用性<T>对象

用于选择序列化响应的实体提供程序的一种条件是对象的通用类型。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，运行时无法决定嵌套实体的通用类型，对象的实际 Java 类用作 Java 类型。

为确保实体提供程序以正确的通用类型信息提供，在将请求对象添加到 Response 对象前，可以将实体 嵌套在 GenericEntity<T > 对象中。

资源方法也可以直接返回 通用Entity<T> 对象。在实践中，这个方法很少被使用。通过反映未打包的实体确定的通用类型信息，为在通用性< T> 对象中嵌套的实体存储的通用 类型信息通常相同。

创建通用性<T> 对象

创建 通用Entity<T> 对象有两种方法：

1. 使用被嵌套的实体，创建 GenericEntity<T > 类的子类。使用子类创建通用性<T> 对象 演示了如何创建 GenericEntity<T> 对象，其中包含一个类型为 List<String& gt; 的实体，其通用类型将在运行时可用。

   使用子类创建通用性<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，直接创建实例。例 48.2 “直接实例化 GenericEntity<T> 对象” 显示如何创建包含 AtomicInteger 的实体的响应。

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

服务器端异步调用的目的是实现更有效的线程使用，并最终避免因为所有服务器的请求线程都被阻断而拒绝客户端连接尝试的情况。当异步处理调用时，请求线程会立即释放。

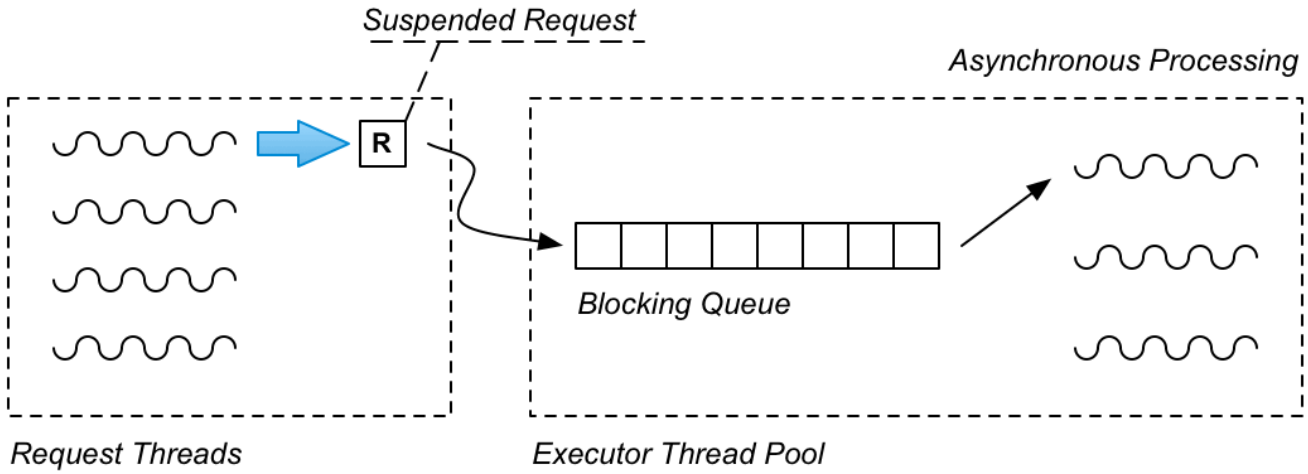
注意

请注意，即使服务器端启用了异步处理，客户端仍会被阻断，直到服务器收到响应为止。如果要在客户端中看到异步行为，则必须实现客户端异步处理。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” 展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



在概述中，请求按照异步模型中的如下处理：

1. 异步资源方法在请求线程内调用（并接收对 AsyncResponse 对象的引用，稍后需要它来发回响应）。

2. 资源方法将暂停的请求封装在可运行对象中，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 线程池的块队列中。

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，它将由 executor 线程池中的其中一个线程处理。封装的 AsyncResponse 对象随后用于向客户端发送响应。

使用 Java 执行程序进行线程池实施

java.util.concurrent API 是一个强大的 API，可让您轻松创建完整的线程池实施。在 Java 并发 API 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，包括工作线程和馈送的队列。

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，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
);
```

此构造器创建一个新的线程池，其中包含五个线程，由单一块队列 fed，其中可容纳最多 10 个 可运行 对象。要向线程池提交任务，请调用 executor.execute 方法，传递对 可运行的 对象的引用（封装异步任务）。

定义异步资源方法

要定义异步的资源方法，请使用 @Suspended 注释，注入类型为 javax.ws.rs.container.AsyncResponse 的参数，并确保方法返回了。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，因为稍后将使用注入的 AsyncResponse 对象来返回响应。

AsyncResponse 类

javax.ws.rs.container.AsyncResponse 类在传入的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挂起状态。稍后，当您准备好返回响应时，可以通过调用 AsyncResponse 实例来重新激活底层 TCP 客户端连接并返回响应。或者，如果您需要中止调用，可以在 AsyncResponse 实例中调用 cancel。

封装一个挂起的请求作为可运行

在 图 48.1 “同步处理的线程模型” 中显示的异步处理场景中，您要将挂起的请求推送到队列，以便稍后被专用线程池处理。为使此方法正常工作，您需要有某种方式在对象中 封装 已暂停的请求。暂停的请求对象需要封装以下操作：

- 来自传入请求的参数（若有）。
- AsyncResponse 对象在传入的客户端连接上提供句柄，以及发回响应的方法。

- 调用的逻辑。

封装这些问题的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（） 方法封装调用的逻辑。执行此操作的最好方法是将 Runnable 作为本地类实施，如下例所示。

异步处理示例

要实施异步处理场景，资源方法的实施必须将可运行的对象（代表暂停的请求）传递到 executor 线程池。在 Java 7 和 8 中，您可以利用一些 novel 语法将可运行类定义为一个本地类，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);
            }
        });
    }
    ...
}
```

注意资源方法参数、ID 和 响应 方式如何直接传递到可运行本地类的定义中。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

为了实现这种特殊语法，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它们）。

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 调用中设定超时。默认情况下，超时会导致将 HTTP 错误响应发回到客户端。但是，您还可以选择注册超时处理程序回调，它可让您自定义对超时事件的响应。

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，但不指定超时处理程序，调用 AsyncResponse 对象上的 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来指定超时值。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，您可以在资源方法正文中包含对 async.SetTimeout 的调用，并且每次调用时将触发超时。

AsyncResponse.NO_TIMEOUT 值代表无限超时。

默认超时(haviour)

默认情况下，如果触发调用超时，JAX-RS 运行时会引发 ServiceUnavailableException 异常，并发回一个 HTTP 错误响应，其状态为 503。

TimeoutHandler 接口

如果要自定义超时，您必须通过实施 TimeoutHandler 接口来定义超时处理程序：

```
// Java
package javax.ws.rs.container;

public interface TimeoutHandler {
    public void handleTimeout(AsyncResponse asyncResponse);
}
```

当您覆盖实现类中的 handleTimeout 方法时，您可以选择以下方法来处理超时：

- 通过调用 asyncResponse.cancel 方法来取消响应。

- 通过调用 asyncResponse.resume 方法以及响应值来发送响应。

- 通过调用 asyncResponse.setTimeout 方法来扩展等待的周期。（例如，要等待更多 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 对象上调用 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
        // ...
    }
    ...
}
```

在本例中，注册了 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 的效果是向客户端发送 HTTP 503（服务不可用）错误响应。您可以选择为 cancel 方法指定参数（即 int 或 java.util.Date 值），用于在响应消息中设置 Retry-After: HTTP 标头。客户端通常会忽略 Retry-After: 标头。

在可运行的实例中处理已取消响应

如果您封装了作为可运行的实例暂停的请求（在 executor 线程池中处理），您可能会发现，线程池开始处理的时间可能会取消。 因此，您很难在可运行的实例中添加一些代码，这使它能够应对取消的 AsyncResponse 对象。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

可以添加一个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，您必须实现 javax.ws.rs.container.ConnectionCallback 接口，该接口定义如下：

```
// 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，您必须调用其中一个 寄存器 方法将它注册到当前的 AsyncResponse 对象。例如，若要注册类型为 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，以标识需要释放的资源）。

48.5.4. 注册回调

概述

您可以选择将回调添加到 AsyncResponse 实例，以便在调用完成时获得通知。当可以调用此回调时，处理中有两个替代点，分别是：

- 请求处理完成后，响应已被发送回客户端，或者
- 请求处理完成后，向托管 I/O 容器传播一个未映射的 Throwable。

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，可迭代 的参数为 null。但是，如果请求处理导致了未映射异常，则可 抛出 包含未映射异常实例。

注册完成回调

在实施完回调后，您必须调用其中一个 寄存器 方法将它注册到当前的 AsyncResponse 对象。例如，若要注册类型为 MyCompletionCallback 的完成回调：

```
asyncResponse.register(new MyCompletionCallback());
```

第 49 章 JAX-RS 2.0 CLIENT API

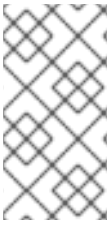
摘要

JAX-RS 2.0 定义了一个功能齐全的客户端 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（以简化构建请求）、用于解析消息的框架（基于称为 *实体提供商 的插件*），以及对客户端上异步调用的支持。

49.1. JAX-RS 2.0 客户端 API 简介

概述

JAX-RS 2.0 定义了用于 JAX-RS 客户端的流畅 API，它可让您构建 HTTP 请求逐步构建，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 Blueprint XML 或 Spring XML 中定义 JAX-RS 客户端（使用 jaxrs:client 元素）。有关这个方法的详情，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_11_1-00015-redhat-00002</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_11_1-00015-redhat-00002</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，您通常需要访问核心软件包中的类：

```
javax.ws.rs.core
```

简单客户端请求的示例

以下代码片段演示了一个简单的示例，其中 JAX-RS 2.0 客户端 API 用于对 http://example.org/bookstore JAX-RS 服务进行调用，并通过 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 设计为一个 *流畅的 API*（有时称为域特定语言）。在流畅的 API 中，以单一声明调用一个 Java 方法链，因此 Java 方法从简单语言类似命令。在 JAX-RS 2.0 中，流畅的 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，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型对象，它代表支持 JAX-RS 客户端（包括拦截器和其他 CXF 功能）所需的技术堆栈。理想情况下，您应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使用 path(String) 方法定义基本 URI 基础 URI，然后向基本 URI 添加额外路径片段：

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
```

构建并调用

这实际上分为两个步骤：首先，您先构建 HTTP 请求（包括标头、接受媒体类型等）；然后，您可以调用相关的 HTTP 方法（可选提供请求消息正文（如果需要）。

例如，要创建并调用接受 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，您需要解析上一步中获得的重新启动。通常，以 javax.ws.rs.core.Response 对象的形式返回响应，它封装了 HTTP 标头和其他 HTTP 元数据，以及 HTTP 邮件正文（若有）。

如果要以 String 格式访问返回的 HTTP 消息，您可以使用 String.class 参数调用 readEntity 方法，如下所示：

```
// Java
...
String msg = resp.readEntity(String.class);
```

您可以始终将响应的消息正文作为 String 进行访问，方法是将 String.class 指定为 readEntity 的参数。有关更常规转换或转换邮件正文，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 class

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

若要创建 WebTarget 实例，可在 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一个完整的路径，使用 target 方法；或者，您可以使用 target 方法和路径方

法的组合添加 路径 片段。将基本路径与路径片段组合的优点是，您可以轻松地在稍有不同目标上为多个调用重新使用基本路径 WebTarget 对象。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，可以使用模板参数 {*param*} 来初始化路径片段，然后被解析为指定的值。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 方法将路径片段 {id} 替换为值 123。

定义查询参数

查询参数可以附加到 URI 路径中，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

矩阵参数与查询参数类似，但不被广泛支持，并使用不同的语法。要在 WebTarget 实例上定义 matrix 参数，调用 matrixParam(String, Object) 方法。

49.3. 构建客户端调用

概述

在构建目标 URI 后，使用 WebTarget 构建器类后，下一步是配置请求的其他方面，如 HTTP 标头、cookies 等。构建调用的最后一个步骤是调用适当的 HTTP 动词（GET、POST、PUT 或 DELETE），并在需要时提供一个消息正文。

invocation.Builder 类

javax.ws.rs.client.Invocation.Builder 构建器类提供 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

若要创建 Invocation.Builder 实例，可在 javax.ws.rs.client.WebTarget 实例上调用 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 cookies

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 属性方法在这个请求的上下文中设置属性，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（继承于 javax.ws.rs.client.SyncInvoker 基础类）：

```
get
post
delete
put
head
trace
options
```

如果您想要调用的特定 HTTP 动词不在此列表中，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 调用方法均提供无类型变体和输入变体（使用额外参数）。如果使用默认的 get（） 方法（不使用参数）调用请求，则 javax.ws.rs.core.Response 对象从调用返回。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>) 方法请求将响应返回为特定类型的值。例如，要调用请求并要求将响应返回为 BookInfo 对象：

```
BookInfo res = client.target("http://example.org/bookstore/books/123")
                    .request("application/xml").get(BookInfo.class);
```

但是，为了解决这个问题，您必须将合适的 *实体提供程序* 注册到 客户端 实例，后者能够将响应格式 application/xml 映射到请求的类型。有关实体供应商的详情，请参考 第 49.4 节 “解析请求和响应”。

在 post 或 put 中指定传出消息

对于在请求中包含消息正文的 HTTP 方法（如 POST 或 PUT），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媒体类型。例如，要调用 POST 方法，消息内容以 String 类型形式提供：

```
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，实体.entity（） 构造器方法将使用注册的实体提供商自动将提供的消息实例映射到指定的介质类型。总是可以将消息正文指定为一个简单的 String 类型。

延迟调用

除了立即调用 HTTP 请求（例如，调用 get（） 方法），您可以选择创建 javax.ws.rs.client.Invocation 对象，稍后可以调用该请求。Invocation 对象 封装 待处理调用的所有详情，包括 HTTP 方法。

可以使用以下方法构建 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 支持客户端侧的异步调用。要进行异步调用，只需在以下 request（） 方法链中调用 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，返回的值为 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 调用的一个基本方面是客户端必须能够解析传出请求消息和传入的响应。在 JAX-RS 2.0 中，关键概念是 实体 类，它代表一个带有介质类型标记的原始消息。为了解析原始消息，您可以注册多个 *实体提供程序*，其具有将介质类型转换为特定 Java 类型的能力。

换句话说，在 JAX-RS 2.0 上下文中，实体 是原始消息的表示，实体提供程序是提供解析原始消息（基于介质类型）的功能。

实体

实体 是元数据增强的消息正文（媒体类型、语言和编码）。实体 实例以原始格式保存消息，并与特定介质类型关联。要将 实体 对象的内容转换为需要 *实体提供商的* Java 对象，它可以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，并由 变体 元数据增强。

实体提供程序

实体提供程序是提供介质类型和 Java 类型之间的映射功能的类。实际上，您可以将实体提供程序视为一个类，能够解析特定介质类型（或可能有多个介质类型）的信息。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 类型的功能。

MessageBodyWriter

提供从 Java 类型映射到介质类型的功能。

标准实体供应商

以下 Java 和介质组合的实体供应商以标准形式提供：

byte[]

所有介质类型(*/*)。

java.lang.String

所有介质类型(*/*)。

java.io.InputStream

所有介质类型(*/*)。

java.io.Reader

所有介质类型(*/*)。

java.io.File

所有介质类型(*/*)。

javax.activation.DataSource

所有介质类型(*/*)。

javax.xml.transform.Source

XML 类型（text/xml、application/xml 和 media type of the application/*+xml）。

javax.xml.bind.JAXBElement 和 application-supplied JAXB 类

XML 类型（text/xml、application/xml 和 media type of the application/*+xml）。

MultivaluedMap<String,String>

表格内容（应用程序/x-www-form-url 编码的）。

StreamingOutput

所有介质类型(*/*)、仅 MessageBodyWriter.

java.lang.Boolean,java.lang.Character,java.lang.Number

仅针对 文本/说明。对应的原语类型通过框/取消框进行转换支持。

响应对象

默认的返回类型是 javax.ws.rs.core.Response 类型，它代表了一个未输入的响应。Response 对象提供对完整 HTTP 响应的访问，包括邮件正文、HTTP 状态、HTTP 标头、媒体类型等。

访问响应状态

您可通过 getStatus 方法访问响应状态（这将返回 HTTP 状态代码）：

```
int status = resp.getStatus();
```

或者，通过 getStatusInfo 方法，它还提供了描述字符串：

```
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 有 Date 标头，您可以按照如下所示访问它：

```
String dateAsString = resp.getHeaderString("Date");
```

访问返回的 cookies

您可以使用 getCookies 方法访问 Response 上设置的任何新 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 对象上的 readEntity 方法之一来访问返回的消息内容。readEntity 方法会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，作为 String 类型访问消息内容：

```
String messageBody = resp.readEntity(String.class);
```

集合返回值

如果您需要以 Java 通用类型来访问返回的消息，如 List 或 Collection 类型，您可以使用 javax.ws.rs.core.GenericType<T > construct 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

通过注册和配置功能和提供程序，可以添加 base javax.ws.rs.client.Client 对象的功能。

示例

以下示例显示了配置为具有日志记录功能、自定义实体提供程序以及将 prettyLogging 属性设置为 true 的客户端：

```
// 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

Client 类支持 Configurable API 用于注册对象，它提供一些变体的 寄存器 方法。在大多数情况下，您可以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 Configurable 参考文档。

您可以对客户端配置什么？

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 实际上是一个插件，为 JAX-RS 客户端添加额外的功能或功能。通常，一个功能会安装一个或多个拦截器以提供所需的功能。

供应商

提供程序是特定的客户端插件，可提供映射功能。JAX-RS 2.0 规范定义以下类型的提供程序：

实体提供程序

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”。

异常映射供应商

例外映射提供程序 将选定的运行时异常映射到 响应 的实例。

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 过滤器是一种插件，可让您访问消息处理管道的各种点（extension 点）上的 URI、标头和各种上下文数据。详情请查看 第 61 章 *JAX-RS 2.0 过滤器和拦截器*。

拦截器

JAX-RS 2.0 拦截器是一种插件，可让您在请求或写出时访问请求的消息正文。详情请查看 第 61 章 *JAX-RS 2.0 过滤器和拦截器*。

Properties

通过在客户端上设置一个或多个属性，您可以自定义注册功能或注册的供应商的配置。

其他可配置类型

不可能配置 javax.ws.rs.client.Client （和 javax.ws.rs.client.client.ClientBuilder）对象，还是一个 WebTarget 对象。当您更改 WebTarget 对象的配置时，底层客户端配置会深入复制，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 支持在客户端一侧进行异步调用。支持两种不同的异步处理方式：使用 java.util.concurrent.Future<V> 返回值；或通过注册调用回调。

使用 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();
```

您可以对输入的响应使用类似的方法。例如，若要获得类型为 **BookInfo** 的回答：

```
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();
```

使用调用回调的异步调用

您可以使用 **Future<V>** 对象访问返回值，而是定义调用回调（使用 **javax.ws.rs.client.InvocationCallback<RESPONSE>**），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();
        }
    });
...
```

您可以对输入的响应使用类似的方法：

```
// 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 处理异常

摘要

当可能时，资源方法发现的异常应该会导致向请求消费者返回有用的错误。JAX-RS 资源方法可以引发 WebApplicationException 异常。您还可以提供 ExceptionMapper<E> 实现来将异常映射到适当的响应。

50.1. JAX-RS EXCEPTION 类概述

概述

在 JAX-RS 1.x 中，唯一可用的异常类是 WebApplicationException。从 JAX-WS 2.0 开始，定义了很多额外的 JAX-RS 异常类。

JAX-RS 运行时级别例外

以下例外旨在仅由 JAX-RS 运行时抛出（也就是说，不得 从应用级别代码中丢弃这些例外）：

ProcessingException

（JAX-RS 2.0 仅） javax.ws.rs.ProcessingException 可在请求处理期间引发，或者在 JAX-RS 运行时进行响应处理。例如，因为过滤器链或拦截器链处理中的错误，可能会引发此错误。

ResponseProcessingException

（仅） javax.ws.rs.client.Response ProcessingException 的子类 processingException，可以在 客户端的 JAX-RS 运行时中出现错误时抛出。

JAX-RS 应用级别例外

以下例外旨在在应用程序级别代码中抛出（并发现）：

WebApplicationException

javax.ws.rs.WebApplicationException 是一个通用应用级别 JAX-RS 异常，它可以抛出服务器侧的应用程序代码中。此异常类型可以封装 HTTP 状态代码、错误消息，以及（可选）响应消息。详情请查看 第 50.2 节 “使用 WebApplicationException 例外报告”。

ClientErrorException

(JAX-RS 2.0 only) javax.ws.rs.ClientErrorException 异常类从 WebApplicationException 继

承，用于封装 HTTP 4xx 状态代码。

ServerErrorException

(JAX-RS 2.0) javax.ws.rs.ServerErrorException 异常类从 WebApplicationException 继承，用于封装 HTTP 5xx 状态代码。

RedirectionException

(JAX-RS 2.0) javax.ws.rs.RedirectionException 异常类从 WebApplicationException 继承，用于封装 HTTP 3xx 状态代码。

50.2. 使用 WEBAPPLICATIONEXCEPTION 例外报告

errors
indexterm:[WebApplicationException]

概述

JAX-RS API 引入了 WebApplicationException 运行时异常，为创建适合 RESTful 客户端使用的异常提供简单方法。WebApplicationException 异常可以包含 Response 对象，用于定义实体正文以返回请求的来源器。它还提供了在不提供实体正文时指定要返回到客户端的 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 argument constructor 或 constructor 将原始异常嵌套在 WebApplicationException 异常中。两个构造器都使用空响应来创建 WebApplicationException。

抛出一个由其中任何一个构造器创建的异常时，运行时会返回一个空实体正文和 500 Server Error 状态代码的响应。

设置返回到客户端的状态代码

当您想要返回 500 以外的错误代码时，您可以使用以下四个 WebApplicationException 构造器之一来指定状态。例 50.1 “创建具有状态代码的 WebApplicationException” 中显示的两个构造器以整数形式取返回状态。

例 50.1. 创建具有状态代码的 WebApplicationException

WebApplicationExceptionintstatusWebApplicationExceptionjava.lang.Throwablecauseintstatu

s

另外两个（如 例 50.2 “创建具有状态代码的 WebApplicationException” 所示）将响应状态用作 Response.Status 的实例。

例 50.2. 创建具有状态代码的 WebApplicationException

WebApplicationExceptionjavax.ws.rs.core.Response.StatusstatusWebApplicationExceptionjava.lang.Throwable会导致javax.ws.rs.core.Response.Status状态

丢弃其中一个构造器创建的异常时，运行时会返回一个空实体正文和指定状态代码的响应。

提供实体正文

如果您希望消息与例外一起发送，您可以使用一个使用 Response 对象的 WebApplicationException 构造器之一。运行时使用 Response 对象来创建发送到客户端的响应。存储在响应中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 “发送消息异常” 显示将文本消息返回到包含异常原因的客户端的代码，并将 HTTP 消息状态设置为 409 Conflict。

例 50.3. 发送消息异常

```
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);
```

扩展通用例外

可以扩展 WebApplicationException 异常。这样，您可以创建自定义例外并消除一些样子代码。

例 50.4 “扩展 WebApplicationException” 显示与 例 50.3 “发送消息异常” 中代码类似的响应的新例外。

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

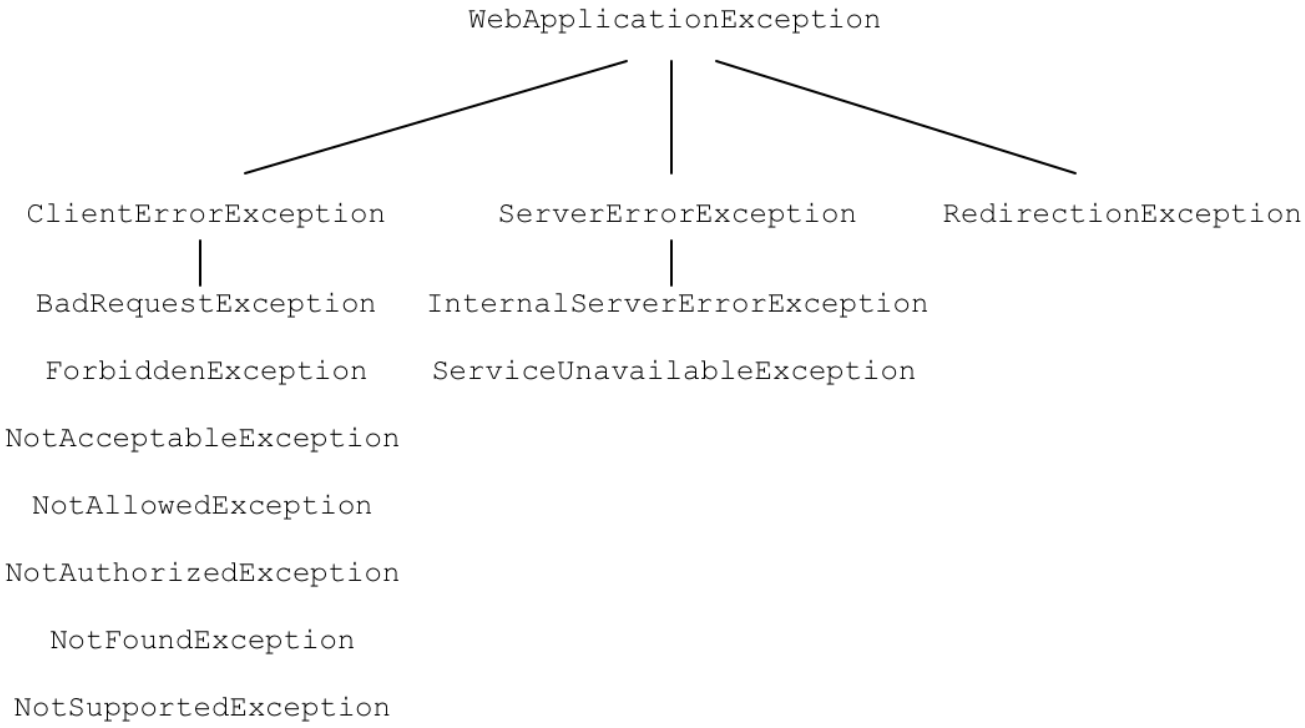
概述

JAX-RS 2.0 引入了多个特定的 HTTP 例外类型，您可以在应用程序代码中抛出（除现有的 WebApplicationException 异常类型外）。这些异常类型可用于嵌套标准 HTTP 状态代码，包括 HTTP 4xx 状态代码（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 应用异常层次结构” 显示 JAX-RS 2.0 支持的应用级别例外的层次结构。

图 50.1. JAX-RS 2.0 应用异常层次结构



WebApplicationException 类

javax.ws.rs.WebApplicationException 异常类（自 JAX-RS 1.x 开始可用）位于 JAX-RS 2.0 层次结构的基础，在 第 50.2 节 “使用 WebApplicationException 例外报告” 中详细介绍。

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）。这个类的构造器使用一个 URI 参数，它指定重定向位置。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。

服务器异常子类

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. 将例外映射到响应

概述

存在引发 WebApplicationException 异常的实例是不现实或不可能的。例如，您可能不希望捕获所有可能的例外情况，然后为它们创建一个 WebApplicationException。您可能还想使用自定义例外，以便更轻松地处理应用程序代码。

若要处理这些情况，您可以使用 JAX-RS API 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供应商。当与 Apache CXF 运行时注册时，每当引发 E 类型的异常时，将使用自定义提供程序。

如何选择异常映射程序

在两个情况下，使用异常映射程序：

- 如果引发任何例外或其子类，则运行时将检查相应的异常映射程序。如果指定映射程序会抛出特定异常，则会选择一个例外映射程序。如果没有为引发特定例外的异常映射程序，则会选择异常最接近的超级分类的异常映射程序。

- 默认情况下，WebApplicationException 将由默认 mapper WebApplicationExceptionMapper 处理。即使注册了额外的自定义映射程序，这可能会正确处理 WebApplicationException 异常（例如，自定义 RuntimeException mapper），也不会 使用自定义映射程序，并将改为使用 WebApplicationExceptionMapper。

  这种行为可以改变，但通过将 Message 对象上的 default.wae.mapper.least.specific 属性设为 true 来更改。启用此选项后，默认的 WebApplicationExceptionMapper 会被重新委派到最低优先级，以便可以处理 WebApplicationException 异常（带有自定义 mapper）。例如，如果启用此选项，可以通过注册自定义 RuntimeException mapper 来捕获 WebApplicationException 异常。请参阅 “为 WebApplicationException 注册异常映射程序”一节。

如果没有找到异常的 mapper，则异常嵌套在 ServletException 异常中，并传递到容器运行时。容器运行时将决定如何处理异常。

实施异常映射程序

通过实施 javax.ws.rs.ext.ExceptionMapper<E> 接口来创建异常映射程序。如 例 50.5 “异常映射程序界面” 所示，接口具有一个单一方法 toResponse（），它使用原始例外作为参数并返回 Response 对象。

例 50.5. 异常映射程序界面

```
public interface ExceptionMapper<E extends java.lang.Throwable>
{
  public Response toResponse(E exception);
}
```

例外映射程序创建的 Response 对象与任何其他 Response 对象一样由运行时处理。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。

异常映射程序实现由运行时被视为提供程序。因此，它们必须与 @Provider 注释进行解码。

如果在异常 mapper 正在构建 Response 对象时出现异常，则运行时将向使用者发送状态为 500 Server Error 的响应。

例 50.6 “将例外映射到响应” 显示一个例外映射程序，它截获 Spring AccessDeniedException 异常异常，并生成带有 403 Forbidden 状态和空实体正文的响应。

例 50.6. 将例外映射到响应

```
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();
  }

}
```

该运行时将捕获任何 AccessDeniedException 例外，并创建没有实体正文和 403 状态的 Response 对象。然后，运行时将处理 Response 对象，因为它会正常的响应。结果是消费者会收到状态 403 的 HTTP 响应。

注册异常映射程序

在 JAX-RS 应用可以使用例外映射程序之前，例外映射程序必须使用运行时进行注册。异常映射程序使用应用配置文件中的 jaxrs:providers 元素在运行时注册。

jaxrs:providers 元素是 jaxrs:server 元素的子级，含有 bean 元素的列表。每个 bean 元素都定义一个例外映射程序。

例 50.7 “使用运行时注册异常映射程序” 显示配置为使用自定义例外映射 SecurityExceptionMapper 的 JAX-RS 服务器。

例 50.7. 使用运行时注册异常映射程序

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

为 WebApplicationException 注册异常映射程序

为 WebApplicationException 异常异常异常注册是一个特殊情况，因为这种异常类型由默认的 WebApplicationExceptionMapper 自动处理。通常，即使您注册了应该处理 WebApplicationException 的自定义映射程序，它仍然将继续由默认的 WebApplicationExceptionMapper 处理。要更改这个默认行为，您需要将 default.wae.mapper.least.specific 属性设置为 true。

例如，以下 XML 代码显示了如何在 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 运行时支持开箱即用 MIME 类型和 Java 对象之间的有限映射。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

该运行时依赖于 JAX-RS MessageBodyReader 和 MessageBodyWriter 实现，以在 HTTP 消息及其 Java 表示之间序列化和反序列化数据。读取器和写入器可以限制其能够处理的 MIME 类型。

运行时为多个常用映射提供读取器和写入器。如果应用程序需要更高级的映射，开发人员可以提供消息 BodyReader 接口和/或 MessageBodyWriter 接口的自定义实现。在应用程序启动时，自定义读取器和写入器会在运行时注册。

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] 此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。它们实施 javax.ws.rs.ext.MessageBodyReader 接口。

接口在 例 51.1 “消息读取器界面” 中显示，有两个需要实现的方法：

例 51.1. 消息读取器界面

```
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（） 方法决定了读者是否能够读取数据流并创建正确类型的实体表示。如果读取器可以创建正确类型的实体，则方法返回为 true。

表 51.2 “用于确定读取器是否可生成实体的参数” 描述 isReadable（） 方法的参数。

表 51.2. 用于确定读取器是否可生成实体的参数

参数	类型	描述
type	**class<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，如果消息正文被转换为方法参数，则该值将是 **Method.getGenericParameterTypes（）** 方法返回的方法参数类型。
annotations	annotation[]	指定用于存储该实体创建的对象的声明上的注解列表。例如，如果消息正文被转换为方法参数，这将是 **Method.getParameterAnnotations（）** 方法返回的注解。
mediaType	**MediatType**	指定 HTTP 实体的 MIME 类型。

readFrom()

readFrom（） 方法读取 HTTP 实体，并将其涉及所需的 Java 对象。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，方法应该会抛出 IOException 异常。如果发生需要 HTTP 错误响应的错误，应当抛出带 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	**class<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，如果消息正文被转换为方法参数，则该值将是 **Method.getGenericParameterTypes（）** 方法返回的方法参数类型。

参数	类型	描述
annotations	annotation[]	指定用于存储该实体创建的对象的声明上的注解列表。例如，如果消息正文被转换为方法参数，这将是 **Method.getParameterAnnotations（）** 方法返回的注解。
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

这个方法不应该关闭输入流。

在将 MessageBodyReader 实施用作实体读取器之前，必须先将它与 javax.ws.rs.ext.Provider 注解进行解码。@Provider 注释提醒提供额外功能的运行时。这种实现还必须与运行时注册，如 “注册读者和写器”一节 所述。

默认情况下，自定义实体提供商处理所有 MIME 类型。您可以使用 javax.ws.rs.Consumes 注解限制自定义实体读取器处理的 MIME 类型。@Consumes 注释指定自定义实体提供程序所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，则实体提供商不会选择为可能的 reader。

例 51.2 “XML 源实体读取器” 显示实体读取者消耗 XML 实体并将其存储在 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。

接口在 例 51.3 “消息写入程序接口” 中显示，有三个需要实现的方法：

例 51.3. 消息写入程序接口

```
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（） 方法决定了实体作者是否可以将 Java 类型映射到正确的实体类型。如果写入器可以进行映射，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	**class<T>**	指定正在写入对象的 Java 类。
genericType	类型	通过反映资源方法返回类型或通过返回的实例检查，指定要写入的 Java 类型。**GenericEntity** 类（在 第 48.4 节 “使用通用类型信息返回实体” 所述）提供了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，则该值将写入到 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	**class<T>**	指定正在写入对象的 Java 类。
genericType	类型	通过反映资源方法返回类型或通过返回的实例检查，指定要写入的 Java 类型。**GenericEntity** 类（在 第 48.4 节 “使用通用类型信息返回实体” 所述）提供了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，并将实体写入到输出流。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。如果发生需要 HTTP 错误响应的错误，应当抛出带 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	**class<T>**	指定正在写入对象的 Java 类。
genericType	类型	通过反映资源方法返回类型或通过返回的实例检查，指定要写入的 Java 类型。**GenericEntity** 类（在 第 48.4 节 “使用通用类型信息返回实体” 所述）提供了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。
entityStream	**OutputStream**	指定将实体写入的输出流。

在消息BodyWriter 实施可用作实体写器之前，它必须通过 javax.ws.rs.ext.Provider 注释来解码。@Provider 注释提醒提供额外功能的运行时。这种实现还必须与运行时注册，如 “注册读者和写器”一节 所述。

默认情况下，自定义实体提供商处理所有 MIME 类型。您可以使用 javax.ws.rs.Produces 注释限制自定义实体写入器的 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，则实体提供商不会选择作为可能的写入器。

例 51.4 “XML 源实体作者” 显示取源对象并生成 XML 实体的实体作者。

例 51.4. XML 源实体作者

```
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;
  }
}
```

注册读者和写器

在 JAX-RS 应用可以使用任何自定义实体提供程序之前，必须将自定义提供程序注册到运行时。提供程序使用应用配置文件中的 jaxrs:providers 元素或使用 JAXRSServeronnectionFactoryyBean 类通过运行时注册。

jaxrs:providers 元素是 jaxrs:server 元素的子级，含有 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” 显示配置为使用一组自定义实体提供程序的 JAX-RS 服务器。

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，用于将实例化的实体提供程序添加到应用程序中。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 标头和其他未使用其他注入注解无法立即可用的详细信息。Apache CXF 提供特殊的类，它将所有可能的上下文信息放在单个对象中。

52.1. 上下文简介

上下文注解

您指定上下文信息是使用 javax.ws.rs.core.Context 注解注入字段或资源方法参数。为其中一个上下文类型添加字段或参数将指示运行时将选择适当的上下文信息注入注解字段或参数。

上下文类型

表 52.1 “上下文类型” 列出可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）	用于确定最佳表示变体或确定是否设置了一组前提条件的信息
SecurityContext	有关请求者安全（包括验证方案）的信息，如果请求频道安全，以及用户原则

可在使用上下文信息的位置

上下文信息适用于 JAX-RS 应用的以下部分：

- 资源类
- 资源方法

- 实体提供程序
- 异常映射程序

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，包括实体提供商和异常映射程序都是如此。

添加上下文

借助 JAX-RS 框架，开发人员可以扩展可使用上下文机制注入的信息类型。您可以通过实施 Context<T> 对象并将其注册到运行时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。大多数信息可以使用方法参数（如 第 47.2.2 节 “注入请求 URI 中的数据” 所述）来访问。但是，使用参数会强制对 URI 如何处理某些限制。使用参数访问 URI 的片段不提供对完整请求 URI 的资源访问权限。

您可以通过将 URI 上下文注入资源来提供对完整请求 URI 的访问权限。URI 作为 UriInfo 对象提供。UriInfo 界面以多种方式模拟 URI。它还可以将 URI 作为 UriBuilder 对象提供，允许您构建 URI 来返回到客户端。

:experimental:

52.2.1. 注入 URI 信息

概述

当作为 UriInfo 对象的类字段或方法参数使用 @Context 注释时，当前请求的 URI 上下文将注入到 UriInfo 对象中。

示例

将 URI 上下文注入类字段 显示通过注入 URI 上下文填充的字段的类。

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 和所选资源的 URI 的路径片段的访问。此信息可用于许多目的，例如根据 URI 或计算 URI 做出处理决策，以作为响应的一部分返回。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资金，并且基本 URI 包含 .co.uk 扩展是否可能决定我们英国的 Pounds。

UriInfo 接口提供了访问 URI 的部分方法：

- 基本 URI
- 资源路径
- 完整 URI

获取 Base URI

基础 URI 是发布该服务的根 URI。它不包含在 service 的 @Path 注释中指定的任何 URI 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被发布到 http://fusesource.org，在 http://fusesource.org/montersforhire/nightstalker?12 基本 URI 中进行了请求将为 http://fusesource.org。

表 52.2 “访问资源基本 URI 的方法” 描述返回基本 URI 的方法。

表 52.2. 访问资源基本 URI 的方法

方法	Desription
URIgetBaseUri	将服务的基本 URI 返回为 URI 对象。
UriBuildergetBaseUriBuilder	将基础 URI 返回为 javax.ws.rs.core.UriBuilder 对象。UriBuilder 类对于为服务实施的其他资源创建 URI 很有用。

获取路径

请求 URI *的路径* 部分是用来选择当前资源的 URI 的部分。它不包括基本 URI，但不包括 URI 中包含的任何 URI 模板变量和列表参数。

路径值取决于所选的资源。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 上进行。
- putterPath — /mostersforhire/911

  PUT 请求在 /monstersforhire/911 上进行。

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	将资源的路径作为已解码的 URI 返回。
字符串getPath布尔值解码	返回资源的路径。指定 **false** 可禁用 URI 解码。

方法	Desription
list<PathSegment>getPathSegments	将解码的路径作为 javax.ws.rs.core.PathSegment 对象的列表返回。路径的每个部分（包括列表参数）都放入列表中的唯一条目。 例如，资源路径 **box/round#tall** 将导致列表，包括三个条目：**框**、**舍入** 和 **tall**。
list<PathSegment>getPathSegments布尔值解码	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）都放入列表中的唯一条目。指定 **false** 可禁用 URI 解码。 例如，资源路径 **box#tall/round** 的结果是带有三个条目的列表：**框**、**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。不同之处在于，请求 URI 包含附加到 URI 的任何查询参数，绝对路径不包含查询参数。

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** 类对于为服务实施的其他资源创建 URI 很有用。
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 的请求，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。通常，这些变量路径片段作为资源方法的参数使用，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，在 例 52.1 “从 URI 上下文返回路径参数的方法” 中显示，它会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&gt;getPathParametersMultivaluedMap<java.lang.String,
java.lang.String&gt;getPathParametersbooleandecode
```

没有利用任何参数的 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)。

值以模板标识符作为键存储在映射中。例如，如果资源的 URI 模板为 /{color}/box/{note}，则返回的映射有两个条目，其键为 color 和 note。

示例

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

子类和实施注释接口的类可以继承 JAX-RS 注释。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可以专门满足特殊需求的通用对象。JAX-RS 允许将类与资源映射时使用的注释从超级类继承来保持此能力。

JAX-RS 的注释继承也扩展为支持接口。实施类继承其所实施的接口上使用的 JAX-RS 注释。

JAX-RS 继承规则提供了一种机制来覆盖继承的注释。但是，无法从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

资源类从其实现的接口继承任何 JAX-RS 注释。资源类也从其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 am 接口继承的注解。

在 例 53.1 “注解继承” 中显示的代码示例中，Kad jin 类的 getMonster（） 方法继承了 Kai ju 接口的 @Path、@GET 和 @PathParam 注释。

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

覆盖继承注解

覆盖继承的注解与提供新注解一样简单。如果子类或实施类为方法提供任何自己的 JAX-RS 注释，则会忽略该方法的所有 JAX-RS 注释。

在 例 53.2 “覆盖注解继承” 中显示的代码示例中，Kadjin 类的 getMonster（） 方法不会继承 Kaiju 界面中的任何注解。实施类覆盖 @Produces 注释，该注释会导致接口中的所有注解被忽略。

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)允许您通过扩展公布的 JAX-RS 服务端点来生成 OpenAPI 文档。

Spring Boot 和 Karaf 实现同时支持 OpenApiFeature。

54.1. OPENAPIFEATURE 选项

您可以在 OpenApiFeature 中使用以下选项：

表 54.1. OpenApiFeature 操作

名称	描述	默认
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
customizer	自定义器类实例	null
description	description+	null
filterClass	一个安全过滤器++	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	当生成 openapi.json 时，正确打印 JSON 文档++	true
propertiesLocation	属性文件位置	**/swagger.properties**

名称	描述	默认
readAllResources	通读所有操作（没有 @Operation++）	true
resourceClasses	必须扫描++ 的资源类列表	null
resourcePackages	一个软件包名称列表，其中必须扫描资源++	null
runAsFilter	以过滤器的形式运行该功能	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： openapi-configuration.yaml openapi-configuration.json openapi.yaml openapi.json	true
scannerClass	JAX-RS API 扫描程序类的名称，用于范围应用、资源软件包、资源类和类路径扫描，请参阅 Resource Scanning 部分	null
securityDefinitions	安全定义列表+	null
supportSwaggerUi	启用/关闭 SwaggerUI 支持	null(== true)
swaggerUiConfig	Swagger UI 配置	null
swaggerUiMavenGroupAndArtifact	用于查点 SwaggerUI 的 Maven 工件	null
swaggerUiVersion	SwaggerUI 的版本	null
termsOfServiceUrl	服务 URL 条款+	null
title	标题+	null

名称	描述	默认
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 使用多个服务器端点）。另外，您可能想要将扫描属性设置为 false。	false
version	版本+	null

+ 选项在 OpenAPI 类中定义

++ 选项在 SwaggerConfiguration 类中定义

54.2. QUARKUS 实施

本节论述了如何使用在 JAR 文件中定义 REST 服务的 OpenApiFeature，并部署到 Karaf 容器的 Fuse 中。

54.2.1. Quickstart 示例

您可以从 Fuse Software Downloads 页面下载 Red Hat Fuse Quickstart。

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：

- **通过将 CXF 类(org.apache.cxf.jaxrs.openapi.OpenApiFeature)添加到 < jaxrs:server > 定义，修改用于定义 CXF 服务的 XML 文件。**

  例如，请参阅 55.4 示例 XML 文件。

- 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,ApiResponse, ApiResponse ,ApiResponses 等等。

详情请查看 https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X。

例如，请参阅 55.5 示例资源类。

  - 添加 OpenAPI 注释到注释的端点（@PATH、@PUT、@POST、@GET、@Produces、@Consumes、@DELETE、@PathParam 等）。

例如，请参阅 55.5 示例资源类。

55.4 示例 XML 文件

```
<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 示例资源类

```
.
.
.

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 引导实现

这部分论述了如何在 Spring Boot 中使用 Swagger2Feature。

请注意，对于 OpenAPI 3 的实现，请使用 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)演示了如何将 Apache CXF 与 Spring Boot 搭配使用。Quickstart 使用 Spring Boot 配置一个启用了 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;
```

  - 将 Swagger2Feature 添加到 CXF 端点：

```
endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
```

    例如，请参阅 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,ApiResponse, ApiResponse ,ApiResponses 等等。

详情请查看 https://github.com/swagger-api/swagger-core/wiki/Annotations。

例如，请参阅 55.2 示例 Java 实施文件。

- 在 Java 文件中，将 Swagger 注释添加到 JAX-RS 注释的端点（@PATH,@PUT,@POST,@GET,@Produces,@Consumes,@DELETE,@PathParam 等等）。

  例如，请参阅 55.3 示例 Java 文件。

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();
    }
}
```

55.2 示例 Java 实施文件

```
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!!!";
    }

}
```

55.3 示例 Java 文件

```
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 OpenAPI 由 OpenApiFeature 启用时，OpenAPI 文档位于由服务端点位置组成的位置 URL，后跟 /openapi.json 或 /openapi.yaml。

例如，对于在 http://host:port/context/services/ 上发布的 JAX-RS 端点，其中 上下文是 Web 应用上下文，/services 则是 servlet URL，其 OpenAPI 文档可在 http://host:port/context/services/openapi.json 和 http://host:port/context/services/openapi.yaml。

如果 OpenApiFeature 处于活跃状态，CXF Services 页会链接到 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 CORS 支持从另一主机上的 OpenAPI UI 访问定义，您可以在 cxf-rt-rs-security-cors 中添加 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
```

注意：如果您的 etc 目录中还没有 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，而您希望防止 OpenAPI 缓存 basePath 值，则将 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

本指南介绍了如何编写可以在消息上执行前和后处理的 Apache CXF 拦截器。

第 55 章 APACHE CXF 运行时的拦截器

摘要

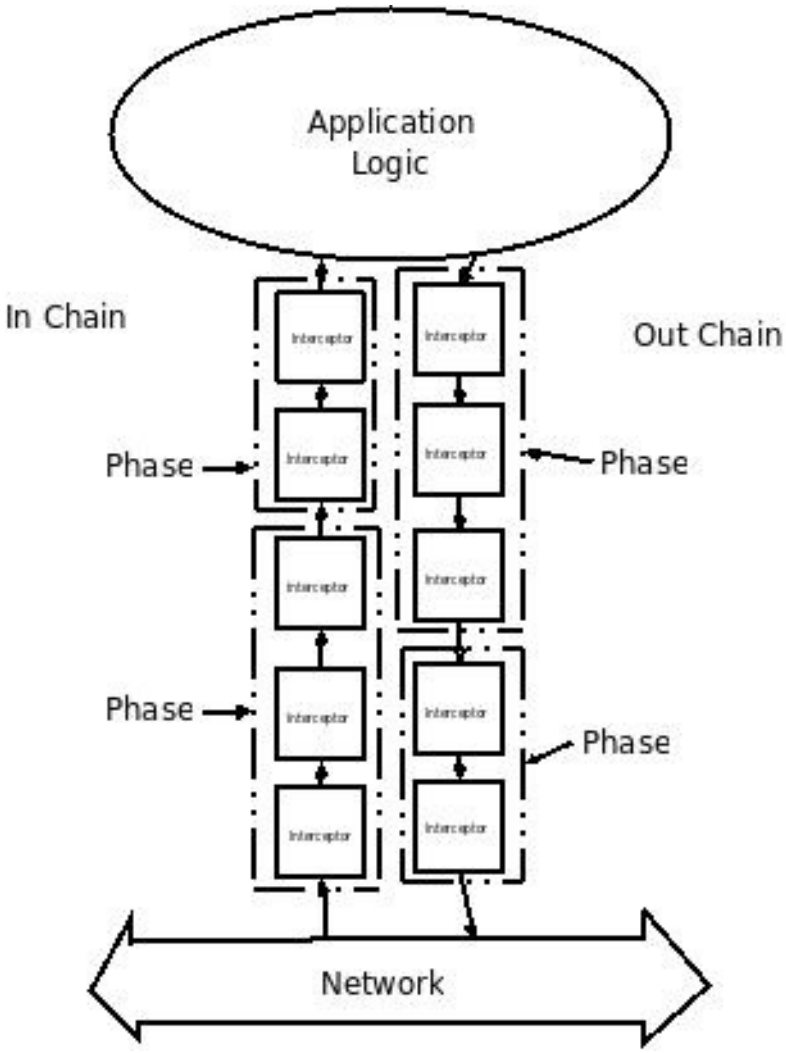
Apache CXF 运行时的大部分功能由拦截器实施。Apache CXF 运行时创建的每个端点包含三个处理消息的潜在拦截器链。这些链中的拦截器负责在线路和端点实施代码处理的 Java 对象之间转换原始数据传输的消息。拦截器被分为阶段，以确保处理正确顺序正确。

概述

Apache CXF 确实需要处理消息的一大部分。当消费者向远程服务调用时，运行时需要将数据放入服务可以使用的消息中，并将其放置在线路上。服务提供商必须联合该消息，执行其业务逻辑，并将响应引入到适当的消息格式。然后，消费者必须解除响应消息，将其与正确的请求相关联，然后将它传回消费者的应用代码。除了基本 marshaing 和 unmarshaling，Apache CXF 运行时还可通过消息数据执行许多其他事务。例如，如果激活 WS-RM，则运行时必须处理消息块并在放送和解放消息前确认消息。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” 显示服务提供程序收到请求时请求消息的基本路径。

图 55.1. Apache CXF 拦截器链



APACHE CXF 中的消息处理

当 Apache CXF 开发的消费者调用远程服务时，以下消息处理序列已启动：

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故障处理拦截器链。

3. 请求消息会按顺序通过出站拦截器链传递。

   链中的每个拦截器都会对消息执行一些处理。例如，Apache CXF 提供的 SOAP 拦截器打包了 SOAP envelope 中的消息。

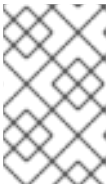
4. 如果出站链上的任何拦截器都创建一个错误，则链没有找到，并且控制权返回到应用程序级别

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，从而无法发现。

5. 请求被分配给适当的服务供应商。

6. 收到响应后，它将按顺序通过入站拦截器链进行传递。



注意

如果响应是错误消息，它将传递到 fault 处理拦截器链。

7. 如果入站链上的任何拦截器都创建一个错误条件，则链将不找到。

8. 当消息到达入站拦截器链的末尾时，它会返回到应用程序代码。

当 Apache CXF 开发的服务供应商从消费者收到请求时，会发生类似的过程：

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，则运行时还会创建出站拦截器链和故障处理拦截器链。

3. 请求通过入站拦截器链顺序传递。

4. 如果入站链上的任何拦截器创建了错误条件，则链将不找到，并将错误分配给消费者。

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，从而无法发现。

5. 当请求到达入站拦截器链的末尾时，它将传递到服务实施。

6. 当响应就绪时，它会按顺序通过出站拦截器链传递。

注意

如果响应是异常，它将通过错误处理拦截器链进行传递。

7. 如果出站链上的任何拦截器创建了错误条件，则链将不找到，并且会分配错误消息。

8. 请求到达出站链的末尾后，它将被分配给消费者。

拦截器

Apache CXF 运行时中的所有消息处理都由 *拦截器* 来完成。拦截器是可在将消息数据传递给应用程序层前访问消息数据的 POJO。它们可以做多个事情，包括：转换消息的消息、剥离消息的标头，或者验证消息数据。例如，拦截器可以读取消息的安全标头，针对外部安全服务验证凭证，并决定消息处理是否可以继续。

拦截器可用的消息数据由多个因素决定：

- 拦截器链
- 拦截器阶段
- 前面在链中发生的其他拦截器

阶段

拦截器分为 *阶段*。阶段是包含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发生。

拦截器链

阶段会聚合为 *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 创建的每个端点有三个拦截器链：

- 入站消息的链
- 出站消息的链
- 用于错误消息的链

拦截器链主要由端点选择使用的绑定和传输组成。添加其他运行时功能，如安全或日志记录，也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

无论它的功能如何，开发拦截器始终遵循相同的基本步骤：

1. 第 56 章 *Interceptor API*

   Apache CXF 提供多个抽象拦截器，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”

   拦截器需要某些消息的部分可用，并且需要以特定格式提供数据。信息的内容及数据格式部分由拦截器的阶段决定。

3. 第 57.3 节 “在阶段限制拦截器放置”

   通常，阶段内拦截器的排序并不重要。但是，在某些情况下，确保拦截器在相同阶段之前或之后执行也可能非常重要。

4. 第 58.2 节 “处理消息”

5. 第 58.3 节 “出错后 Unwinding”

   如果在拦截器执行后在活跃的拦截器链中发生错误，则会调用其错误处理逻辑。

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。此接口定义了很多 Apache CXF 的运行时用于控制拦截器执行的方法，并不适用于开发人员实现。为简化拦截器开发，Apache CXF 提供了多个可以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所需的两种方法：

handleMessage()

handleMessage（） 方法在拦截器中执行大多数工作。它在消息链中的每个拦截器调用，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（） 方法的详细信息，请参阅 第 58.2 节 “处理消息”。

handleFault()

当普通消息处理被中断时，在拦截器中调用 handleFault（） 方法。运行时调用每个调用拦截器的 handleFault（） 方法，因为它会取消截取拦截器链。有关实现 handleFault（） 方法的详细信息，请参阅 第 58.3 节 “出错后 Unwinding”。

大多数拦截器不会直接实施 Interceptor 接口。相反，它们实施 例 56.2 “阶段拦截器接口” 中显示的 PhaseInterceptor 接口。PhaseInterceptor 接口添加了四个方法来允许拦截器链。

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();

}
```

摘要拦截器类

开发人员不应直接扩展 **PhaseInterceptor** 接口，而是扩展 **AbstractPhaseInterceptor** 类。此抽象类为 **PhaseInterceptor** 接口的阶段管理方法提供实施。**AbstractPhaseInterceptor** 类还提供了 **handleFault（） 方法的默认实施。**

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。开发人员提供的实施可以使用通用 org.apache.cxf.message.Message.Message 接口提供的方法操控消息数据。**

对于使用 **SOAP** 消息的应用程序，**Apache CXF** 提供了一个 **AbstractSoapInterceptor** 类。扩展这个类提供 **handleMessage（） 方法和 handleFault（） 方法，具有将消息数据的访问权限为 org.apache.cxf.binding. SOAp.SoapMessage 对象。SoapMessage 对象具有从消息中检索 SOAP 标头、SOAP envelope 和其他 SOAP 元数据的方法。**

第 57 章 确定 INTERCEPTOR 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内容。拦截器可以决定其在同一阶段与其他拦截器的关系的位置。拦截器的阶段及其在阶段的位置设置为拦截器的构造器逻辑的一部分。

57.1. 指定 INTERCEPTOR 位置

在开发自定义拦截器时，需要考虑的第一个操作是消息处理链中拦截器所属的位置。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拦截器位置：

- 指定拦截器的阶段
- 指定阶段内拦截器的位置限制

通常，指定拦截器的位置的代码放置在拦截器的构造器中。这样，运行时可以实例化拦截器，并在拦截器链中放入正确的位置，而无需在应用程序级别代码中进行任何显式操作。

57.2. 指定拦截器的阶段

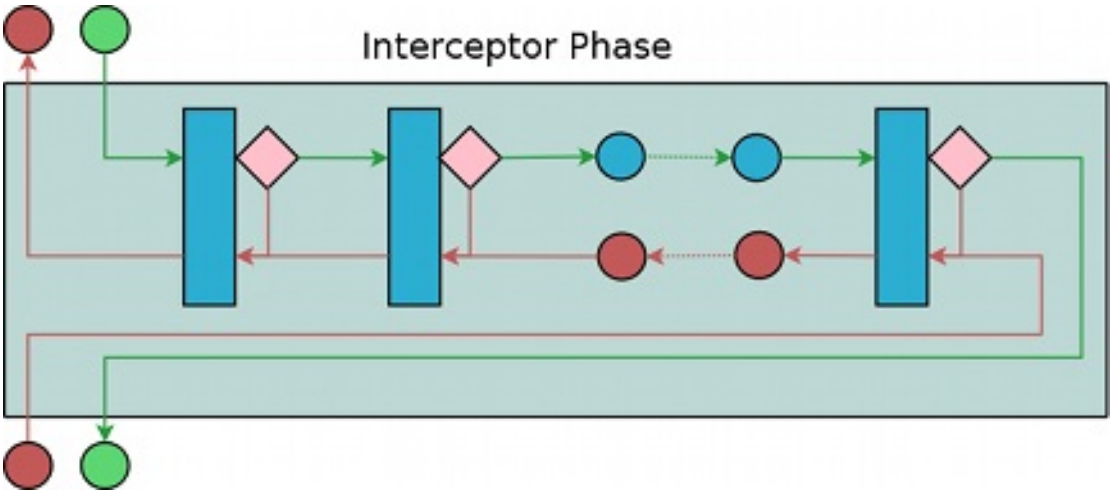
概述

拦截器分为几个阶段。拦截器的阶段决定了它所调用的消息处理序列中的时间。开发人员指定拦截器它的构造器。使用框架提供的常量值来指定阶段。

阶段

阶段是拦截器的逻辑集合。如 图 57.1 “一个拦截器阶段” 所示，阶段中的拦截器会按顺序调用。

图 57.1. 一个拦截器阶段



这些阶段以排序列表中链接在一起，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 阶段组使用原始消息数据从线路获取时会处理传输级别详情。

然而，这里没有执行任何阶段可以执行的操作。建议在阶段内拦截器遵循阶段要编写的任务。

Apache CXF 定义的阶段完整列表可在 第 62 章 *Apache CXF 消息处理阶段* 中找到。

指定阶段

Apache CXF 提供 org.apache.cxf.Phase 类来指定阶段。类是常量的集合。Apache CXF 定义的每个阶段在 阶段 类都有一个相应的常量。例如，RECEIVE 阶段由 value 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 类定义了三个构造器用于实例化拦截器：

- 公共 AbstractPhaseInterceptor(String phase)- 将拦截器的阶段设置为指定阶段，并将拦截器的 id 设置为拦截器的类名称。

  此构造器将满足大多数用例。

- 公共 AbstractPhaseInterceptor(String id, String phase)- 将拦截器的 id 设置为在第一个参数传递的字符串，以及拦截器阶段到第二个字符串。

- 公共 AbstractPhaseInterceptor(String phase, boolean uniqueId)- 表示拦截器是否应使用唯一系统生成的 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 设置为拦截器的类名称。

设置自定义拦截器的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 拦截器被放入 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足够精细地控制其放置，以确保拦截器正常工作。例如，如果某一拦截器需要使用 SAAJ API 检查消息的 SOAP 标头，则需要在将消息转换为 SAAJ 对象的拦截器后运行。在某些情况下，一个拦截器会消耗另一个拦截器需要的信息的一部分。在这种情况下，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拦截器列表，该列表必须在拦截器之后执行。

重要

运行时只能在拦截器的阶段遵从这些列表。如果开发人员将拦截器从早期阶段放置于必须在当前阶段之后必须执行的拦截器，则运行时将忽略请求。

先添加到链中

开发拦截器时会出现一个问题，即拦截器所需的数据并不总是存在。当链中的一个拦截器会消耗以后

拦截器所需的消息数据时，会出现这种情况。开发人员可以通过修改其拦截器来控制消耗哪些自定义拦截器，并可能解决这个问题。但是，因为 Apache CXF 使用了很多拦截器，所以开发人员无法修改它们。

另一种解决方案是确保在任何使用自定义拦截器所需消息数据的拦截器之前放置自定义拦截器。最简单的方法是将其放置在早期阶段，但并不总是可能。对于需要放置拦截器时，需要放在一个或多个其他拦截器之前，Apache CXF 的 AbstractPhaseInterceptor 类提供两个 addBefore（） 方法。

如 例 57.2 “在其他拦截器前添加拦截器的方法” 所示，使用一个拦截器 id，另一个则使用拦截器 id。您可以进行多个调用来继续将拦截器添加到列表中。

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 “指定必须在当前拦截器之后运行的拦截器列表” 所示，开发人员在自定义拦截器的 Constuctor 中调用 addBefore（） 方法。

例 57.3. 指定必须在当前拦截器之后运行的拦截器列表

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addBefore(HolderOutInterceptor.class.getName());
    }

...

}
```

大多数拦截器都将其类名称用于拦截器 id。

添加到链的后面

拦截器需要的数据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，但如果消息被放入到 SOAP 消息前，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消耗哪些自定义拦截器，并可能解决这个问题。但是，因为 Apache CXF 使用了很多拦截器，所以开发人员无法修改它们。

另一种解决方案是确保将自定义拦截器放在拦截器或拦截器后，它会生成自定义拦截器所需的消息数据。最简单的方法是将其放置在后续的阶段，但并不总是可能。AbstractPhaseInterceptor 类为当一个或者其它拦截器后需要放置拦截器时，提供两个 addAfter（） 方法。

如 例 57.4 “其他拦截器后添加拦截器的方法” 所示，一种方法采用单个拦截器 id，另一个方法取一组拦截器 ID。您可以进行多个调用来继续将拦截器添加到列表中。

例 57.4. 其他拦截器后添加拦截器的方法

```
publicaddAfterStringipublicaddAfterCollection<String>i
```

如 例 57.5 “指定必须在当前拦截器前运行的拦截器列表” 所示，开发人员在自定义拦截器的 Constuctor 中调用 addAfter（） 方法。

例 57.5. 指定必须在当前拦截器前运行的拦截器列表

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addAfter(StartingOutInterceptor.class.getName());
    }

...

}
```

大多数拦截器都将其类名称用于拦截器 id。

第 58 章 实现拦截器处理日志

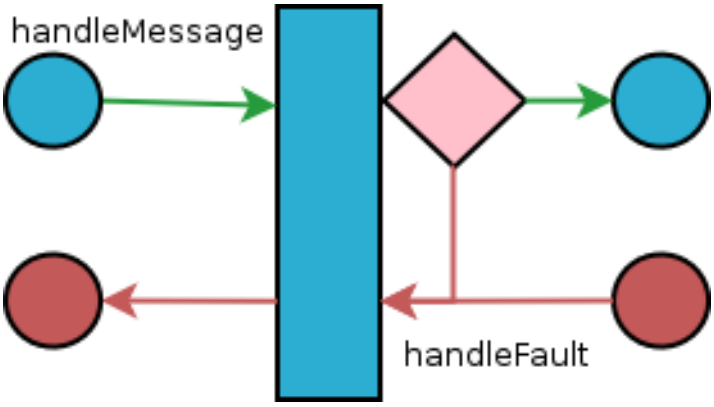
摘要

拦截器直接实现。大部分其处理逻辑在 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，只有 handleMessage（） 方法被调用。handleMessage（） 方法是放置拦截器的消息处理逻辑的位置。

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。handleFault（） 方法有助于在出错时清除拦截器。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。它将消息数据作为 Message 对象接收。除了消息的实际内容外，Message 对象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之前的拦截器。

获取消息内容

Message 接口提供了可在提取消息内容时使用的方法：

- public<T> T getContentjava.lang.Class<T> format The getContent（） 方法返回指定类对象中的消息内容。如果内容作为指定类的实例不可用，则返回 null。可用内容类型列表由拦截器链上的拦截器位置和拦截器链的方向决定。

- publicCollection<Attachment& gt; getAttachments（） 方法返回一个包含与消息关联的任何二进制附件的 Java Collection 对象。该附件存储在 org.apache.cxf.message.Attachment 对象中。附加对象提供管理二进制数据的方法。

重要

Attachments 仅在附件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]

与消息关联的消息交换是通过消息的 getExchange（） 方法来检索。如 例 58.1 “获取消息交换” 所示，getExchange（） 不会取任何参数，并将消息交换返回为 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，在 例 58.2 “从消息交换获取信息” 中显示，用于获取与交换关联的信息。每个方法都将返回消息作为 org.apache.cxf.Message 对象，如果消息不存在，则返回 null。

例 58.2. 从消息交换获取信息

```
MessagegetInMessageMessagegetInFaultMessagegetOutMessagegetOutFaultMessage
```

例 58.3 “检查消息链的方向” 显示用于确定当前消息是否出站代码。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。它还会针对交换出站错误消息检查当前的消息，到出站错误拦截器链上的错误消息。

例 58.3. 检查消息链的方向

```
public static boolean isOutbound()
{
  Exchange exchange = message.getExchange();
      return message != null
            && exchange != null
            && (message == exchange.getOutMessage()
               || message == exchange.getOutFaultMessage());
}
```

示例

例 58.4 “消息处理方法示例” 显示处理 zip 压缩信息的拦截器的代码。它将检查消息的方向，然后执行相应的操作。

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 UNWINDING

概述

在执行拦截器链的过程中发生错误时，运行时会通过调用已执行的链中的 handleFault（） 方法来停止遍历拦截器链。

handleFault（） 方法可用于清理在正常消息处理过程中拦截器使用的所有资源。它还可用于回滚只能在消息处理成功完成时省略的所有操作。如果错误消息将传递到出站错误处理拦截器链，则可以使用 handleFault（） 方法将信息添加到故障消息。

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 所述从 Message 对象获取消息内容。

示例

例 58.5 “处理未缓解的拦截器链” 显示代码，用于确保当拦截器链未找到时，原始 XML 流重新置于消息中。

例 58.5. 处理未缓解的拦截器链

```
@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故障。

第 59 章 配置端点以使用拦截器

摘要

在消息交换中包含时，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构建。拦截器在端点的配置或其中一个运行时组件配置中指定。可以使用配置文件或拦截器 API 添加拦截器。

59.1. 确定附加拦截器的位置

概述

托管拦截器链的多个运行时对象。它们是：

- endpoint 对象
- service 对象
- 代理对象
- 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任何这些对象。要附加拦截器的最常用对象是总线和单个端点。选择正确的对象时，需要了解这些运行时对象是如何组合来使端点。根据设计，每个 cxf 相关捆绑包都有自己的 cxf 总线。因此，如果在总线中配置了拦截器，且同一 blueprint 上下文的服务被导入或被创建到另一个捆绑包中，则拦截器不会被处理。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方法。任何直接连接到端点的拦截器，或代理只能

影响特定的端点或代理。这是附加特定于服务特定警告的拦截器的良好情况。例如，如果某个开发人员想要公开一个服务实例，它将指标的单元转换为不误地可以将拦截器直接附加到一个端点。

因素

使用 Spring 配置将拦截器附加到用于创建端点的工厂中，或者代理具有与将拦截器直接附加到端点或代理时的影响。但是，当拦截器被附加到工厂时，附加到工厂的拦截器会传播到由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于使用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 绑定的端点都包含特殊的 ID 元素，他们可以将这个拦截器附加到 XML 绑定中。

总线

附加拦截器的最常规位置是总线。当拦截器附加到总线时，拦截器会传播到由该总线管理的所有端点。在创建多个端点的应用程序中，将拦截器附加到总线会很有用，后者共享一组类似的拦截器。

组合附加点

因为端点的最终拦截器链是由列出的对象贡献的拦截器链的修改，所以列出的几个对象可以在单一端点配置中合并。例如，如果应用程序生成的多个端点都需要检查验证令牌的拦截器，则该拦截器会附加到应用程序的总线中。如果其中一个端点还需要将 Euros 转换为收号，转换拦截器会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 进行配置。然后，拦截器 bean 可以使用 Apache CXF 配置元素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均可使用专用的 Spring 元素进行配置。每个组件的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项。拦截器的子项列表添加到

链中。

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 配置拦截器列表，以添加到端点的入站拦截器链中。
outInterceptors	包含 Bean 配置拦截器列表，以添加到端点的出站拦截器链中。
inFaultInterceptors	包含 Bean 配置拦截器的列表，以添加到端点的入站错误处理拦截器链中。
outFaultInterceptors	包含 Bean 配置拦截器列表，以添加到端点的出站错误处理拦截器链中。

所有拦截器链配置元素都取一个 列表 子元素。list 元素有一个子项，用于附加到链的每个拦截器。拦截器可以通过直接配置拦截器或引用配置拦截器的 bean 元素来指定。

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

拦截器可以通过以下方法之一附加到端点中：

- InterceptorProvider API
- Java 注解

使用 InterceptorProvider API，开发人员可以将拦截器附加到具有拦截器链的任何运行时组件，但它需要处理底层 Apache CXF 类。Java 注解只能添加到服务接口或服务实施中，但它们允许开发人员保留在 JAX-WS API 或 JAX-RS API 中。

59.3.2. 使用拦截器供应商 API

概述

拦截器可以使用任何实现在 拦截器供应商接口 中显示的 InterceptorProvider 接口的组件注册。

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();
}
```

通过接口中的四个方法，您可以检索端点的拦截器链作为 Java List 对象。通过使用 Java List 对象提供的方法，开发人员可以在任何链中添加和移除拦截器。

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. 使用拦截器要附加到的链，访问运行时组件。

   开发人员必须使用 Apache CXF 特定的 API 访问标准 Java 应用代码的运行时组件。通常可以通过将 JAX-WS 或 JAX-RS 工件转换为基础 Apache CXF 对象来访问运行时组件。

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合并。

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：

为消费者创建一个 JAX-WS Service 对象。

在提供消费者的目标地址的 Service 对象中添加端口。

创建用于在服务提供商上调用方法的代理。

将代理转换为 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供应商

以编程方式将拦截器附加到服务供应商 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供应商

```
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供应商 中的代码执行以下操作：

创建一个 ServerFactoryBean 对象，用于提供对底层 Apache CXF 对象的访问权限。

获取 Apache CXF 用来代表端点的服务器对象。

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；出站拦截器链。

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 将附加到由运行时实例创建的所有端点的入站拦截器链中。

59.3.3. 使用 Java 注解

概述

Apache CXF 提供四个 Java 注解，允许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他方法不同，注解附加到应用程序级别的工件。所用的构件决定了注解的作用范围。

放置注解的位置

注解可以放在以下工件中：

- 定义端点端点接口(SEI)

  如果注解放置在 SEI 上，则实施该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。

- 服务实施类

  如果注解放在实施类上，则使用实施类的所有服务提供商都将受到影响。

注解

该注解都位于 org.apache.cxf.interceptor 软件包中，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	指定出站错误拦截器链的拦截器。

列出拦截器

拦截器列表使用 在链注解中列出拦截器的语法 中显示的语法指定为完全限定类名称列表。

在链注解中列出拦截器的语法

```
interceptors={"interceptor1", "interceptor2", ..., "interceptorN"}
```

示例

将拦截器附加到服务实现中 显示将两个拦截器附加到使用 SayHiImpl 所提供的逻辑的入站拦截器链的注解。

将拦截器附加到服务实现中

```
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 INTERCEPTOR CHAINS

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。任何通常操作都是特定于调用的，因此每次涉及消息交换时，都会使用原始链。

概述

只有在消息交换被创建时，拦截器链才会有效。每个消息都包含对负责进行处理的拦截器链的引用。开发人员可以使用此引用来更改消息的拦截器链。因为链是按更改的，对消息拦截器链的任何更改都不会影响其他消息交换。

链生命周期

链中拦截器链和拦截器会基于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拦截器的实例实例化。当导致创建拦截器链的消息交换完成后，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的字段所做的任何更改都不会在消息交换中保留。因此，如果拦截器将另一个拦截器放在活跃链中，则只影响活跃链。将来的消息交换都将从 pristine 状态创建，具体由端点的配置决定。这也意味着开发人员无法在修改将来的消息处理的拦截器中设置标记。

如果拦截器需要与将来的实例一起传递信息，它可以在消息上下文中设置属性。该上下文在消息交换之间保留。

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。这通过使用 例 60.1 “获取拦截器链的方法” 中显示的 Message.getInterceptorChain（） 方法来实现。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”，用于将拦截器添加到拦截器链中。您可以使用一个拦截器添加单个拦截器，您可以添加多个拦截器。

例 60.2. 将拦截器添加到拦截器链的方法

添加Interceptor<? 扩展消息>iaddCollection<Interceptor<? 扩展 Message>i

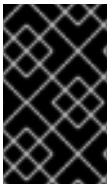
例 60.3 “将拦截器添加到拦截器链中” 显示将单一拦截器添加到消息拦截器链的代码。

例 60.3. 将拦截器添加到拦截器链中

```
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中” 中的代码执行以下操作：

实例化拦截器的副本来添加到链中。



重要

添加到链中的拦截器应该处于与当前拦截器相同的阶段，或者与当前拦截器不同。

获取当前消息的拦截器链。

将新拦截器添加到链中。

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”，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<? 扩展消息>i

例 60.5 “从拦截器链中删除拦截器” 显示从消息拦截器链中删除拦截器的代码。

例 60.5. 从拦截器链中删除拦截器

```
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 过滤器和拦截器

摘要

JAX-RS 2.0 定义在 REST 调用的处理管道中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 过滤器和拦截器介绍

概述

本节概述了 JAX-RS 过滤器和拦截器的处理管道，强调可以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* 是一种插件类型，使开发人员能够访问通过 CXF 客户端或服务器的所有 JAX-RS 消息。过滤器适合处理与消息关联的元数据：HTTP 标头、query 参数、介质类型和其他元数据。过滤器具有中止消息调用的能力（例如，对于安全插件很有用）。

如果您类似，您可以在每个扩展点中安装多个过滤器，在这种情况下，过滤器在链中执行（执行顺序是未定义，除非为每个已安装的过滤器指定了一个 优先级值 ）。

拦截器

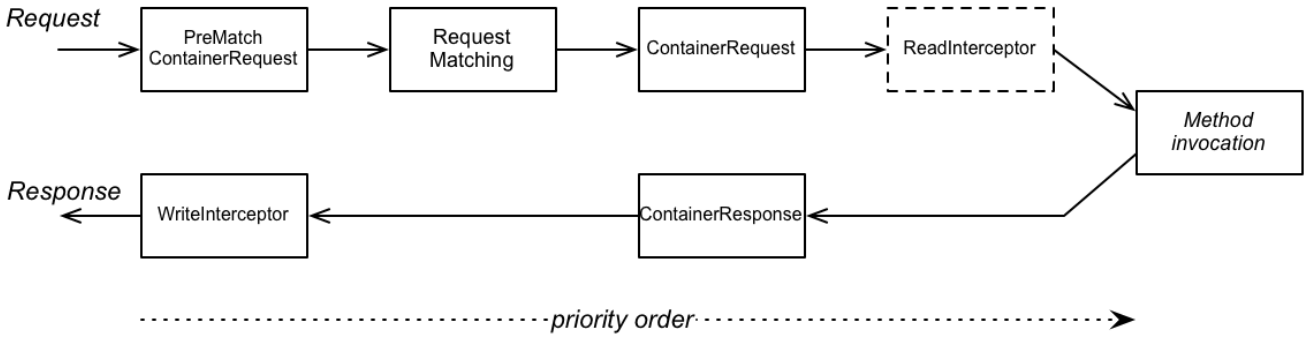
JAX-RS 2.0 *拦截器* 是类型插件，使开发人员能够像被读取或写入时赋予对消息正文的访问权限。拦截器围绕 MessageBodyReader.readFrom 方法调用（用于读取拦截器）或 MessageBodyWriter.writeTo 方法调用（用于 writer 拦截器）。

如果您类似，您可以在每个扩展点上安装多个拦截器，在这种情况下，拦截器是未定义的（除非您为每个安装的拦截器指定 优先级值 ）。

服务器处理管道

图 61.1 “server-Side Filter 和 Interceptor Extension Points” 演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，您可以在以下任意扩展点中添加过滤器（或拦截器）：

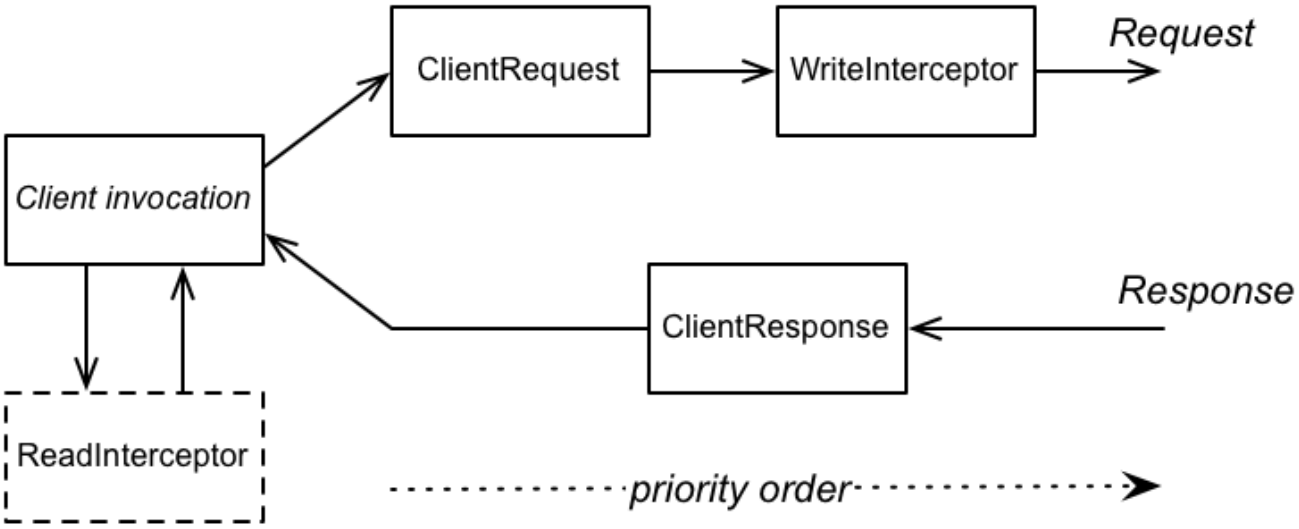
1. PreMatchContainerRequest 过滤器
2. containerRequest 过滤器
3. ReadInterceptor
4. ContainerResponse 过滤器
5. WriteInterceptor

请注意，在发生资源匹配 前，会达到 PreMatchContainerRequest 扩展点，因此此时一些上下文元数据将不可用。

客户端处理管道

图 61.2 “客户端过滤和 Interceptor Extensension Point” 显示用于 JAX-RS 过滤器和客户端上安装的拦截器的处理管道的概述。

图 61.2. 客户端过滤和 Interceptor Extensension Point



客户端扩展点

在客户端处理管道中，您可以在以下任意扩展点中添加过滤器（或拦截器）：

1. ClientRequest 过滤器
2. WriteInterceptor
3. ClientResponse 过滤器
4. ReadInterceptor

过滤和拦截器顺序

如果您在相同扩展点中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 注释）。优先级以整数值表示。通常，具有 较高优先级 号的过滤器会更接近服务器上的资源方法调用；而具有 较低优先级 号的过滤器则接近客户端调用。换句话说，对 请求 消息执行的过滤器和拦截器会按优先级编号 降序执行；而过滤器和拦截器对 响应 消息执行，则以降序为 优先级编号执行。

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* 用于截获服务器（容器）侧的传入请求消息。容器请求过滤器通常用于处理服务器端上的标头，并可用于任何种类的通用请求处理（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对象，因为它可以安装两个不同的扩展点：PreMatchContainerRequest （在资源匹配步骤之前）和 ContainerRequest （在资源匹配步骤后）。

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;
}
```

通过实施 ContainerRequestFilter 接口，您可以为服务器端的以下扩展点创建一个过滤器：

- **PreMatchContainerRequest**
- **containerRequest**

ContainerRequestContext 接口

ContainerRequestFilter 的 过滤器 方法接收单个参数，即 javax.ws.rs.container.ContainerRequestContext，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 扩展点实施容器请求过滤器（即，在资源匹配前执行过滤器），定义一个实施 ContainerRequestFilter 接口的类，确保为类添加 @PreMatching 注解（以选择 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，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，在资源匹配后执行过滤器），定义一个实施 ContainerRequestFilter 接口的类，而无需 @PreMatching 注释。

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，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将 ResourceInfo 类注入 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

通过创建适合容器请求过滤器的实施，可以中止服务器端调用。通常，这可用于在服务器端实施安全功能：例如，实施身份验证功能或授权功能。如果传入的请求无法验证成功，您可以在容器请求过滤器中中止调用。

例如，以下预先匹配功能尝试从 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据。如果身份验证失败，则调用 ContainerRequestContext 对象上的 abortWith 会中止，并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，将其安装到 Apache CXF 运行时），请执行以下步骤：

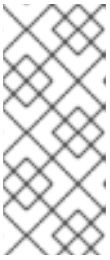
1. **将 @Provider 注释添加到容器请求过滤器类，如以下代码片段所示：**

```
// 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现加载到 Apache CXF 运行时，REST 实现会自动扫描加载的类，以搜索标有 @Provider 注释（*扫描阶段*）的类。

2. **在 XML 中定义 JAX-RS 服务器端点时（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”），将服务器请求过滤器添加到 jaxrs:providers 元素中的提供程序列表中。**

```
<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

这一步是 Apache CXF 的非标准要求。根据 JAX-RS 标准严格讲，@Provider 注释应当全部满足绑定过滤器所必需的。但在实践中，标准方法有些不灵活，当大型项目中纳入多个库时，可能会导致供应商冲突。

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，一般可用于任何种类的通用响应处理。

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 扩展点创建过滤器，它会在调用执行后过滤响应消息。

注意

通过容器响应过滤器，您可以同时访问请求消息（通过 requestContext 参数）和响应消息（通过 responseContext 信息），但在此阶段只能修改响应。

ContainerResponseContext 接口

ContainerResponseFilter 的 过滤 方法接收两个参数：类型为 javax.ws.rs.container.ContainerRequestContext 的参数（请参阅 “ContainerRequestContext 接口”一节）；以及类型为 javax.ws.rs.container.ContainerResponseContext 的参数，它可用于访问传出响应消息及其相关的元数据。

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);
}
```

实施示例

要为 ContainerResponse 扩展名点实施容器响应过滤器（即，在服务器端执行调用后执行过滤器），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，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，将其安装到 Apache CXF 运行时），请执行以下步骤：

1. **将 @Provider 注释添加到容器响应过滤器类，如以下代码片段中所示：**

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

当将容器响应过滤器实现加载到 Apache CXF 运行时，REST 实现会自动扫描加载的类以搜索标有 @Provider 注释（*扫描阶段*）的类。

2. **在 XML 中定义 JAX-RS 服务器端点时（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”），将服务器响应过滤器添加到 jaxrs:providers 元素中的提供程序列表中。**

```
<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

这一步是 **Apache CXF** 的非标准要求。根据 **JAX-RS** 标准严格讲，**@Provider** 注释应当全部满足绑定过滤器所必需的。但在实践中，标准方法有些不灵活，当大型项目中纳入多个库时，可能会导致供应商冲突。

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* 用于截获客户端上的传出请求消息。客户端请求过滤器通常用于处理标头，并可用于任何种类的通用请求处理。

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**，您可以为客户端一侧的 **ClientRequest** 扩展点创建一个过滤器，在向服务器发送消息前过滤请求消息。

ClientRequestContext 接口

ClientRequestFilter 的 过滤器 方法接收单个参数，即 javax.ws.rs.client.ClientRequestContext，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);
}
```

实施示例

要为 ClientRequest 扩展点实施客户端请求过滤器（即，在发送请求消息前执行过滤器），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，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

通过实施适当的客户端请求过滤器，可以中止客户端调用。例如，您可以实施客户端过滤器来检查请求是否已正确格式化，并在需要时中止请求。

以下测试代码 总是 中止请求，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，您可以直接在 javax.ws.rs.client.client.Client 对象上注册客户端请求过滤器，或者在 javax.ws.rs.client.WebTarget 对象上注册。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，因此只有特定 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，以将它应用到使用 客户端 对象进行的所有调用；以及如何注册 **TestAbortClientRequest** 过滤器，使它仅应用到 **rest/TestAbortClientRequest** 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* 用于截获客户端上的传入响应消息。客户端响应过滤器可用于客户端上任意一种通用响应处理。

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**，您可以为客户端一侧的 **ClientResponse** 扩展点创建一个过滤器，在从服务器接收响应消息后过滤它。

ClientResponseContext 接口

ClientResponseFilter 的 过滤 方法接收两个参数：类型为 javax.ws.rs.client.client.ClientRequestContext （请参阅 “ClientRequestContext 接口”一节）和类型为 javax.ws.rs.client.client.ClientResponseContext 的参数，它可用于访问传出响应消息及其相关的元数据。

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);
}
```

实施示例

要为 ClientResponse 扩展名点实施客户端响应过滤器（即，在收到响应消息后执行过滤器），请定义一个实施 ClientResponseFilter 接口的类。

例如，以下代码显示了一个在 ClientResponse 扩展点中安装的简单客户端响应过滤器，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，您可以直接在 javax.ws.rs.client.client.Client 对象上注册客户端响应过滤器，或者在 javax. ws.rs.client.WebTarget 对象上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，因此只有特定 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. 实体读者

概述

本节介绍如何实施和注册 *实体读取拦截器*，这可让您在客户端或服务器端读取消息正文时截获输入流。这通常对请求正文的通用转换（如加密和解密）或压缩和解压缩非常有用。

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）在服务器端读取或客户端。您可以在以下任一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，则实体读取器在应用程序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，相关正文可能无法由匹配的资源访问，在这种情况下，读者拦截器没有调用。

- 客户端侧- 如果作为客户端拦截器，则实体读取器拦截器会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明确访问响应消息（例如，通过调用 Response.getEntity 方法），则 reader 拦截器不会被调用。

ReaderInterceptorContext 接口

ReaderInterceptor 的 aroundReadFrom 方法接收一条类型为 javax.ws.rs.ext.ReaderInterceptorContext 的参数，它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 接口还支持从基础 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（优先级 10）的实体读取器拦截器示例，它替代了进入响应的消息正文中的所有 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();
  }
}
```

服务器端实现示例

要为服务器端实施实体读取器拦截器，请定义一个实施 ReaderInterceptor 接口的类，并使用 @Provider 注释标注。

例如，以下代码显示了服务器端的实体读取器拦截器（优先级为 10），它替代了传入请求消息正文中的所有 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();
  }
}
```

在客户端上绑定读者拦截器

使用 JAX-RS 2.0 客户端 API，您可以直接在 javax.ws.rs.client.client.Client 对象或 javax.ws.rs.client.WebTarget 对象上注册实体读取器。同样，这意味着读者可以选择性地应用到不同的范围，因此只有某些 URI 路径受拦截器影响。

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”。

在服务器端绑定读器

要在服务器端 绑定 读取器（即，将其安装到 Apache CXF 运行时），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 {
  ...
}
```

当读取拦截器实现加载到 Apache CXF 运行时，REST 实现会自动扫描加载的类以搜索标有 @Provider 注释（*扫描阶段*）的类。

2. **在 XML 中定义 JAX-RS 服务器端点时（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”），将读取拦截器添加到 jaxrs:providers 元素中的提供程序列表中。**

```
<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

这一步是 Apache CXF 的非标准要求。根据 JAX-RS 标准严格讲，@Provider 注释应是绑定拦截器所必需的。但在实践中，标准方法有些不灵活，当大型项目中纳入多个库时，可能会导致供应商冲突。

61.7. 实体作者 INTERCEPTOR

概述

本节介绍如何实施和注册 *实体作者*，它可让您在客户端或服务器端写入消息正文时截获输出流。这通常对请求正文的通用转换（如加密和解密）或压缩和解压缩非常有用。

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）在服务器端编写或客户端。您可以在以下任一上下文中使用实体写器：

- 服务器端- 如果作为服务器端拦截器，实体写入器拦截器会在响应消息正文之前被截取并发回到客户端。
- 客户端侧- 如果作为客户端拦截器，实体写入器拦截器会在请求消息正文被放入并发送到服务器前截获请求消息正文。

WriterInterceptorContext 接口

WriterInterceptor 的 aroundWriteTo 方法接收一个类型为 javax.ws.rs.ext.WriterInterceptorContext 的参数，它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 接口还支持从基础 InterceptorContext 接口继承的方法。有关 InterceptorContext 的定义，请参阅 “InterceptorContext 接口”一节。

客户端实现示例

要为客户端实施实体作者，请定义一个实施 WriterInterceptor 接口的类。

例如，以下代码显示了客户端（优先级 10）的实体写器拦截器示例，它会将额外行文本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();
  }
}
```

服务器端实现示例

要为服务器端实施实体写器拦截器，请定义一个实施 WriterInterceptor 接口的类，并将其标上 @Provider 注释。

例如，以下代码显示了服务器端的实体写器拦截器（优先级为 10），它会将额外行文本附加到传出请求的消息正文：

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

在客户端上绑定写入器

使用 JAX-RS 2.0 客户端 API，您可以直接在 javax.ws.rs.client.Client 对象或 javax.ws.rs.client.WebTarget 对象上注册实体写器拦截器。同样，这意味着 writer 拦截器可以选择性地应用到不同的范围，因此只有某些 URI 路径会受到拦截器影响。

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”。

在服务器端绑定写入器

要在服务器端 绑定 写入器（即，将其安装到 Apache CXF 运行时），请执行以下步骤：

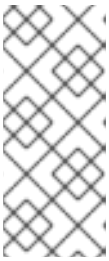
1. 将 @Provider 注释添加到 writer 拦截器类，如以下代码片段所示：

```
// 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 writer 拦截器实现加载到 Apache CXF 运行时，REST 实现会自动扫描加载的类以搜索标有 @Provider 注释（ *扫描阶段*）的类。

2. 在 XML 中定义 JAX-RS 服务器端点时（例如，请参阅 第 18.1 节 “配置 JAX-RS 服务器端点”），将写入器拦截器添加到 jaxrs:providers 元素中的提供程序列表中。

```
<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

这一步是 **Apache CXF** 的非标准要求。根据 **JAX-RS** 标准严格讲，**@Provider** 注释应是绑定拦截器所必需的。但在实践中，标准方法有些不灵活，当大型项目中纳入多个库时，可能会导致供应商冲突。

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为过滤器和拦截器添加 **@Provider** 注释。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一种替代方法，供您选择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，您必须实施自定义 **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

您实现一个动态功能，如下所示：

1. 实施一个或多个容器过滤器或容器拦截器，如前面所述。但不要 用 @Provider 注释（否则为他们进行全局绑定，使动态功能有效到期）。

2. 通过实施 DynamicFeature 类来创建您自己的动态功能，覆盖配置方法。

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要调用哪些资源方法。您可以使用这些信息来决定一些过滤器或拦截器。

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. 记得使用 @Provider 注释给动态功能类添加注解，以确保它在部署的扫描阶段被使用。

动态功能示例

以下示例演示了如何定义用于为 MyResource 类（或子类注释）的任何方法注册 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());
    }
}
```

动态绑定进程

JAX-RS 标准要求 DynamicFeature.configure 方法为每个资源方法调用一次。这意味着每个资源方法都可以由动态功能安装过滤器或拦截器，但它由动态功能负责决定是否在每个情况下注册过滤器或拦截器。换句话说，动态功能支持的绑定粒度是单个资源方法的级别。

FeatureContext 接口

FeatureContext 接口（允许您在配置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	执行传输特定处理，比如为二进制附加确定 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 信息，构建会添加正确的拦截器。SOAP 邮件标题也在此阶段处理。
PRE_PROTOCOL	执行协议级别处理。这包括处理 WS-* 标头和 SOAP 消息属性的处理。
USER_PROTOCOL	
POST_PROTOCOL	
UNMARSHAL	将消息数据解放到应用程序级别代码使用的对象中。
PRE_LOGICAL	处理未汇总的消息数据。
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，在此阶段调用服务实施。在客户端，响应被移回应用程序。
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	在链中执行之后阶段所需的任何集合。
PRE_LOGICAL	对从应用程序级别传递的未汇总的数据执行处理。
USER_LOGICAL	
POST_LOGICAL	
PREPARE_SEND	打开连接以在线路上写入消息。
PRE_STREAM	执行所需的处理，以将条目的消息准备到数据流中。
PRE_PROTOCOL	开始处理协议特定信息。
写	写入协议消息。
PRE_MARSHAL	放大消息。
MARSHAL	
POST_MARSHAL	
USER_PROTOCOL	处理协议消息。
POST_PROTOCOL	
USER_STREAM	处理字节级别消息。
POST_STREAM	
SEND	发送消息并关闭传输流。

重要

出站拦截器链具有一组镜像结束阶段，其名称会附加有 _ENDING。结束阶段使用拦截器，要求在有线上写入数据前执行一些终端操作。

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**	调用服务的正确方法。

出站

Apache CXF 默认不将任何内核拦截器添加到出站拦截器链中。端点的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 端点入站消息链的拦截器。

表 63.2. 入站 JAX-WS 拦截器

类	阶段	描述
HolderInterceptor	**PRE_INVOKE**	在消息中为任何 out 或 in/out 参数创建拥有者对象。
WrapperClassInInterceptor	**POST_LOGICAL**	将嵌套的 doc/literal 消息的部分解包到相应的对象数组中。
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 处理程序完成后，该消息会一起传递给入站链上的下一拦截器。

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 处理程序通过消息结束时，信息会一起传递给链中的下一个拦截器。

表 63.3 “出站 JAX-WS 拦截器” 列出添加到 JAX-WS 端点出站消息链的拦截器。

表 63.3. 出站 JAX-WS 拦截器

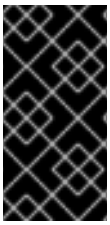
类	阶段	描述
HolderOutInterceptor	**PRE_LOGICAL**	从拥有者对象中删除任何 out 和 in/out 参数的值，并在消息的参数列表中添加值。
WebFaultOutInterceptor	**PRE_PROTOCOL**	处理出站错误消息。
WrapperClassOutInterceptor	**PRE_LOGICAL**	在消息中添加前，请确保已换行为 doc/literal 消息和 rpc/literal 消息。
LogicalHandlerOutInterceptor	**PRE_MARSHAL**	将消息处理传递到端点使用的 JAX-WS 逻辑处理程序。当 JAX-WS 处理程序完成后，该消息会一起传递给出站链上的下一拦截器。
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 端点入站消息链的拦截器。

表 63.4. 入站 JAX-RS 拦截器

类	阶段	描述
JAXRSInInterceptor	**PRE_STREAM**	选择根资源类，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至 ServiceInvokerInterceptor 拦截器。在 JAXRSInterceptor 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 端点的出站消息链的拦截器。

表 63.5. 出站 JAX-RS 拦截器

类	阶段	描述
JAXRSOutInterceptor	**MARSHAL**	将响应置于正确的格式以进行传输。

63.3. 消息绑定

SOAP

表 63.6 “入站 SOAP 拦截器” 使用 SOAP Binding 时，列出将拦截器添加到端点的入站消息链。

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。如果消息是错误消息，则正常处理将中止，并且启动错误处理。
MustUnderstandInterceptor	**PRE_PROTOCOL**	处理必须了解标头。
RPCInInterceptor	**UNMARSHAL**	Unmarshals rpc/literal 消息。如果消息是裸机，则会将消息传递给 **BareInterceptor** 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操作的唯一操作。
SoapHeaderInterceptor	**UNMARSHAL**	将映射映射到操作参数的 SOAP 标头绑定到适当的对象。
AttachmentInterceptor	**接收**	解析 mime boundaries 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** 对象的集合中。
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数据中读取的数据绑定。
StaxInterceptor	**POST_STREAM**	从消息创建 **XMLStreamReader** 对象。
URIMappingInterceptor	**UNMARSHAL**	处理 HTTP GET 方法的处理。
SwAInterceptor	**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” 使用 SOAP Binding 时，列出将拦截器添加到端点的出站消息链。

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**	删除只标记为入站的所有 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 以及处理消息中可能存在的任何附件所需的 mime 项。
BareOutInterceptor	**MARSHAL**	写入消息部分。

类	阶段	描述
StaxOutInterceptor	**PRE_STREAM**	从消息创建 **XMLStreamWriter** 对象。
WrappedOutInterceptor	**MARSHAL**	打包出站消息参数。
SoapOutInterceptor	**写**	在消息中写入 **soap:envelope** 元素和标头块的元素。另外，同时为剩余的拦截器写入一个空 **soap:body** 元素以填充。
SwAOutInterceptor	**PRE_LOGICAL**	删除所有将打包为 SOAP 附加的二进制数据，并将其存储用于后续处理。

XML

表 63.8 “入站 XML 拦截器” 使用 XML Binding 时，列出将拦截器添加到端点的入站消息链。

表 63.8. 入站 XML 拦截器

类	阶段	描述
AttachmentInterceptor	**接收**	解析 mime boundaries 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** 对象的集合中。
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。
StaxInterceptor	**POST_STREAM**	从消息创建 **XMLStreamReader** 对象。
URIMappingInterceptor	**UNMARSHAL**	处理 HTTP GET 方法的处理。
XMLMessageInInterceptor	**UNMARSHAL**	解放 XML 消息。

表 63.9 “出站 XML 拦截器” 使用 XML Binding 时，列出将拦截器添加到端点的出站消息链。

表 63.9. 出站 XML 拦截器

类	阶段	描述
StaxOutInterceptor	PRE_STREAM	从消息创建 XMLStreamWriter 对象。
WrappedOutInterceptor	MARSHAL	打包出站消息参数。
XMLMessageOutInterceptor	MARSHAL	放大传输消息。

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链的拦截器。

表 63.10. 入站 CORBA 拦截器

类	阶段	描述
CorbaStreamInInterceptor	PRE_STREAM	反序列化 CORBA 消息。
BareInterceptor	UNMARSHAL	对消息部分进行反序列化。

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的拦截器。

表 63.11. 出站 CORBA 拦截器

类	阶段	描述
CorbaStreamOutInterceptor	PRE_STREAM	对消息进行序列化。
BareOutInterceptor	MARSHAL	写入消息部分。
CorbaStreamOutEndingInterceptor	USER_STREAM	为消息创建一个可流对象，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 “入站日志拦截器” 列出添加到端点入站消息链以支持日志的拦截器。

表 63.12. 入站日志拦截器

类	阶段	描述
LoggingInterceptor	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以支持日志的拦截器。

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。

有关日志记录的更多信息，请参阅 第 19 章 *Apache CXF Logging*。

WS-Addressing

表 63.14 “入站 WS-Addressing 拦截器” 使用 WS-Addressing 时，列出将拦截器添加到端点的入站消息链。

表 63.14. 入站 WS-Addressing 拦截器

类	阶段	描述
MAPCodec	PRE_PROTOCOL	解码消息寻址属性。

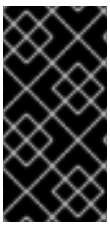
表 63.15 “出站 WS-Addressing 拦截器” 使用 WS-Addressing 时，列出添加到端点出站消息链的拦截器。

表 63.15. 出站 WS-Addressing 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-Addressing 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 拦截器也将添加到拦截器链中。

表 63.16 “入站 WS-RM 拦截器” 使用 WS-RM 时，列出将拦截器添加到端点的入站消息链。

表 63.16. 入站 WS-RM 拦截器

类	阶段	描述
RMInInterceptor	**PRE_LOGICAL**	处理消息部分和确认消息的聚合。
RMSoapInterceptor	**PRE_PROTOCOL**	编码和解码来自消息的 WS-RM 属性。

表 63.17 “出站 WS-RM 拦截器” 使用 WS-RM 时，列出将拦截器添加到端点的出站消息链。

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。同时处理确认和重新发送请求的处理。
RMSoapInterceptor	**PRE_PROTOCOL**	编码和解码来自消息的 WS-RM 属性。

有关 WS-RM 的更多信息，请参阅 第 21 章 *启用可靠消息*。

第 64 章 拦截器供应商

概述

拦截器供应商是在 Apache CXF 运行时中附加有拦截器链的对象。它们都实施 org.apache.cxf.interceptor.InterceptorProvider 接口。开发人员可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 验证是一个 Java 标准，您可以通过将 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)- 这是原始 Bean 验证 1.0(JSR-303)标准的演进，您可以在运行时声明可以检查的限制，可以使用 Java 注解来声明可进行检查的约束。您可以使用注解对 Java 代码的以下部分定义限制：

- 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某些标准 bean 验证限制的 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 schema 验证？

在某些方面，bean 验证和 schema 验证非常类似。使用 XML 模式配置端点是在 Web 服务端点上在运行时验证消息的一种良好方法。XML 模式可以检查许多与传入和传出消息验证相同的限制。但是，对于以下一个或多个原因，bean 验证有时会是一个有用的替代选择：

- Bean 验证可让您独立于 XML 模式定义限制（例如，在代码第一服务开发时很有用）。
- 如果您当前的 XML 模式太小，可以使用 bean 验证来定义更严格的限制。
- an 验证可让您定义自定义约束，这可能无法使用 XML 模式语言进行定义。

依赖项

Bean Validation 1.1(JSR-349)标准仅定义 API，而非实施。因此，必须在两个部分提供依赖项：

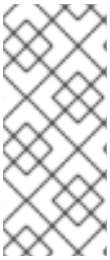
- 核心依赖项- 提供 bean 验证 1.1 API、Java 统一表达式语言 API 和实施。
- Hibernate 验证器依赖项- 提供 bean 验证 1.1 的实施。

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 和实施。此表达式语言由 bean 验证在内部使用，但在应用程序编程级别上并不重要。

Hibernate 验证器依赖项

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 环境中的验证供应商

解决验证提供程序的默认机制涉及扫描 classpath 来查找提供程序资源。然而，如果 OSGi(Apache Karaf)环境不起作用，这种机制 不起作用，因为验证提供程序（例如，Hibernate 验证器）被打包到单独的捆绑包中，因此您的应用程序类路径中不可用。在 OSGi 上下文中，需要将 Hibernate 验证器与您的应用程序捆绑包相连接，并且 OSGi 需要一些帮助才能成功完成此操作。

在 OSGi 中显式配置验证供应商

在 OSGi 的上下文中，您需要明确配置验证供应商，而不必依赖于自动发现。例如，如果您使用通用验证功能（请参阅 “bean 验证功能”一节）来启用 bean 验证，则必须使用验证供应商进行配置，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，它会打包 Hibernate 验证提供程序。

HibernateValidationProviderResolver 类示例

以下代码示例演示了如何定义自定义 HibernateValidationProviderResolver，它可解析 Hibernate 验证器：

```
// 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时，配置为使用 Maven bundle 插件时，您的应用程序将会在部署时附加到 Hibernate 验证捆绑包（假设您已将 Hibernate 验证捆绑包部署到 OSGi 容器）。

65.2. 使用 BEAN 验证开发服务

65.2.1. 为 Service Bean 标注

概述

使用 bean 验证开发服务的第一个步骤是将相关验证注解应用到代表服务的 Java 类或接口。验证注解可让您应用约束来将限制应用到方法参数、返回值和类字段，然后在运行时检查该服务。

验证简单的输入参数

要验证服务方法的参数-其中参数简单 Java 类型，您可以应用 bean 验证 API 中的任何约束注解（javax.validation.constraints 软件包）。例如，以下代码示例测试了 nullness（@NotNull 注

释）、id 字符串与 \\d+ 正则表达式匹配（@Pattern 注释），以及 名称 字符串在范围 1 到 50 的长度：

```
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();
}
```

验证复杂的输入参数

要验证复杂的输入参数（对象实例），请将 @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 注释不自行指定任何限制。使用 @Valid 注解 Book 参数时，您可以有效地告知验证引擎在 Book 类的定义中（递归）查找验证限制。在本例中，Book 类定义了在其 id 和 name 字段上的验证限制，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Response）

要将验证应用到常规方法返回值（非Response），请在方法签名前添加注解。例如，要测试 nullness（@NotNull 注释）的返回值，并以递归方式测试验证约束（@Valid 注释），请注解 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，您可以使用与非接收情况下相同的注解。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 验证的标准注释” 显示 Bean Validation 规范中定义的标准注释，它们可用于定义字段和方法返回值和参数（标准注解没有可以在类级别上应用）。

表 65.1. Bean 验证的标准注释

注解	适用于	描述
@AssertFalse	布尔值,布尔值	检查注解的元素是否为 **false**。
@AssertTrue	布尔值,布尔值	检查注解的元素是否为 **true**。

注解	适用于	描述
@DecimalMax(value=, inclusive=)	**BigDecimal**,**BigInteger**,**CharSequence**, byte**, short**, int ,**int**,**long** 和 primitive type wrappers	当 **inclusive=false** 时，检查注解的值是否小于指定的最大值。否则，检查该值是否小于或等于指定的最大值。**value** 参数指定最大为 **BigDecimal** 字符串格式。
@DecimalMin(value=, inclusive=)	**BigDecimal**,**BigInteger**,**CharSequence**, byte**, short**, int ,**int**,**long** 和 primitive type wrappers	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** 参数指定最小值，采用 **BigDecimal** 字符串格式。
@Digits(integer=, fraction=)	**BigDecimal**,**BigInteger**,**CharSequence**, byte**, short**, int ,**int**,**long** 和 primitive type wrappers	检查注释的值是否为具有最多 **整数** 数和 **部分** 部分数字的数字。
@Future	**java.util.Date**, **java.util.Calendar**	检查注解日期是否在将来。
@Max(value=)	**BigDecimal**,**BigInteger**,**CharSequence**, byte**, short**, int ,**int**,**long** 和 primitive type wrappers	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**, short**, int ,**int**,**long** 和 primitive type wrappers	检查注解的值是否大于或等于指定最小值。
@NotNull	任何类型	检查注释的值不是 **null**。
@Null	任何类型	检查注释的值是否为 **null**。
@Past	**java.util.Date**, **java.util.Calendar**	检查注解的日期是否位于过去。
@Pattern(regex=, flag=)	**CharSequence**	检查注释的字符串 **是否与** 正则表达式正则表达式匹配。
@Size(min=, max=)	**CharSequence**,**Collection**,**Map** and 数组	检查注解的集合、映射或数组的大小（包括在 **min** 和 **max** （含）之间。

注解	适用于	描述
@Valid	任何非保护类型	在注释的对象上递归执行验证。如果对象是集合或数组，则会以递归方式验证元素。如果对象是映射映射，则将递归验证值元素。

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 自行定义自定义约束注解。有关如何在 Hibernate 验证器实现中进行此操作的详情，请参考 Hibernate 验证器 参考指南 中的 创建自定义约束 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 验证的拦截器在 JAX-WS 端点和 JAX-RS 1.1 端点上都很常见（但 JAX-RS 2.0 端点则使用不同的拦截器类）。

命名空间

在本节所示的 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，对于 Blueprint 或 Spring，如下表所示：

XML 语言	命名空间
蓝图（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 验证最简单的方法是在端点中添加 *bean 验证功能*。bean 验证功能由以下类实施：

org.apache.cxf.validation.BeanValidationFeature

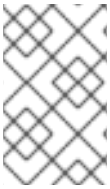
通过将此功能类实例添加到 JAX-WS 端点（可以通过 Java API，或通过 XML 中的 jaxws:features 子元素的 jaxws:endpoint ）添加实例，您可以在端点上启用 bean 验证。此功能安装两个拦截器：即验证传入的消息数据的拦截器；以及验证返回值的 Out 拦截器（其中使用默认配置参数创建拦截器）。

带有 bean 验证功能的 JAX-WS 配置示例

以下 XML 示例演示了如何在 JAX-WS 端点中启用 bean 验证功能，方法是将 commonValidationFeature bean 添加到端点，作为 JAX-WS 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，请参阅 “HibernateValidationProviderResolver 类示例”一节。在 OSGi 环境(Apache Karaf)环境上下文中，只需要配置 beanValidationProvider。



注意

请记住，根据上下文，将 jaxws 前缀映射到蓝图或 Spring 的相应 XML 命名空间。

常见 bean 验证 1.1 拦截器

如果要对 bean 验证的配置拥有更精细的控制，您可以单独安装拦截器，而不必使用 bean 验证功能。通过放置 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，根据验证限制验证资源方法参数。如果验证失败，则引发 javax.validation.ConstraintViolationException 异常。要安装此拦截器，请通过 XML 中的 jaxws:inInterceptors 子元素将其添加到端点（或 XML 中的 jaxrs:inInterceptors 子元素）。

org.apache.cxf.validation.BeanValidationOutInterceptor

在 JAX-WS（或 JAX-RS 1.1）端点中安装时，根据验证限制验证响应值。如果验证失败，则引发 javax.validation.ConstraintViolationException 异常。要安装此拦截器，请通过 XML 中的 jaxws:outInterceptors 子元素将其添加到端点（或 XML 中的 jaxrs:outInterceptors 子元素）。

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显式将相关 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，请参阅 “HibernateValidationProviderResolver 类示例”一节。在 OSGi 环境(Apache Karaf)环境上下文中，只

需要配置 beanValidationProvider。

配置 BeanValidationProvider

org.apache.cxf.validation.BeanValidationProvider 是一个嵌套 bean 验证实施（验证供应商）的简单打包程序类。通过覆盖默认的 BeanValidationProvider 类，您可以自定义 bean 验证的实施。BeanValidationProvider bean 可以覆盖以下一个或多个供应商类：

javax.validation.ParameterNameProvider

提供方法和构造器参数的名称。请注意，需要这个类，因为 Java 反映 API 不允许您 访问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。通过实施您自己的 ValidationProvider 类，您可以为您自己的类定义自定义验证规则。这种机制可以让您扩展 bean 验证框架。

javax.validation.ValidationProviderResolver

实施一种机制来发现 ValidationProvider 类，并返回发现的类的列表。默认解析器会在 classpath 上查找 META-INF/services/javax.validation.spi.ValidationProvider 文件，其中应包含 ValidationProvider 类的列表。

javax.validation.ValidatorFactory

返回 javax.validation.Validator 实例的工厂。

org.apache.cxf.validation.ValidationConfiguration

CXF 打包程序类，允许您从验证供应商层覆盖更多类。

要自定义 BeanValidationProvider，请将自定义 BeanValidationProvider 实例传递到验证拦截器的构造器，并传输到验证拦截器的结构器。 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 服务端点上启用 bean 验证，该端点在 Blueprint XML 或 Spring XML 中定义。用于执行 bean 验证的拦截器在 JAX-WS 端点和 JAX-RS 1.1 端点上都很常见（但 JAX-RS 2.0 端点则使用不同的拦截器类）。

命名空间

在本节所示的 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，对于 Blueprint 或 Spring，如下表所示：

XML 语言	命名空间
蓝图（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 验证最简单的方法是在端点中添加 *bean 验证功能*。bean 验证功能由以下类实施：

org.apache.cxf.validation.BeanValidationFeature

通过将此功能类的实例添加到 JAX-RS 端点（可以通过 Java API 或通过 jaxrs:features 子元素的 jaxrs:server in XML 中启用 bean 验证），即可在端点上启用 bean 验证。此功能安装两个拦截器：即验证传入的消息数据的拦截器；以及验证返回值的 Out 拦截器（其中使用默认配置参数创建拦截器）。

验证异常映射器

JAX-RS 端点还需要配置 *验证异常映射程序*，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

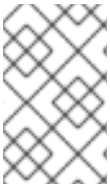
根据 JAX-RS 2.0 规范实施验证例外映射：任何输入参数验证违反情况都映射到 HTTP 状态代码 400 Bad Request；以及任何返回值验证违反（或内部验证违反情况）都映射到 HTTP 状态代码 500 Internal Server Error。

JAX-RS 配置示例

以下 XML 示例演示了如何在 JAX-RS 端点中启用 bean 验证功能，方法是添加 commonValidationFeature bean 作为 JAX-RS 功能，并通过添加 exceptionMapper an 作为 JAX-RS 供应商：

```
<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，请参阅 “HibernateValidationProviderResolver 类示例”一节。在 OSGi 环境(Apache Karaf)环境上下文中，只需要配置 beanValidationProvider。



注意

记得根据情况将 jaxrs 前缀映射到蓝图或 Spring 的相应 XML 命名空间。

常见 bean 验证 1.1 拦截器

您可以选择安装 bean 验证拦截器来获取对验证实施的更精细的控制，而不是使用 bean 验证功能。JAX-RS 使用与 JAX-WS 相同的拦截器进行此目的，请参阅 “常见 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 示例演示了如何在 JAX-RS 端点中启用 bean 验证功能，方法是显式将相关的 In interceptor bean 和 Out interceptor bean 添加到服务器端点：

```
<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，请参阅 “HibernateValidationProviderResolver 类示例”一节。在 OSGi 环境(Apache Karaf)环境上下文中，只需要配置 beanValidationProvider。

配置 BeanValidationProvider

您可以将自定义 BeanValidationProvider 实例注入到验证拦截器中，如 “配置 BeanValidationProvider”一节 所述。

65.3.3. JAX-RS 2.0 配置

概述

与 JAX-RS 1.1 不同（与 JAX-WS 共享通用验证拦截器），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

通过将此功能类实例添加到 JAX-RS 端点（通过 Java API 或通过 jaxrs:features 子元素的 jaxrs:server in XML 中），您可以对 JAX-RS 2.0 服务器端点启用 bean 验证。此功能安装两个拦截器：即验证传入的消息数据的拦截器；以及验证返回值的 Out 拦截器（其中使用默认配置参数创建拦截器）。

验证异常映射器

JAX-RS 2.0 使用与 JAX-RS 1.x 相同的验证异常 mapper 类：

org.apache.cxf.jaxrs.validation.ValidationExceptionMapper

根据 JAX-RS 2.0 规范实施验证例外映射：任何输入参数验证违反情况都映射到 HTTP 状态代码 400 Bad Request; 以及任何返回值验证违反（或内部验证违反情况）都映射到 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，则您还应注册 org.apache.cxf.jaxrs.validation.JAXRSBeanValidationInvoker instance- using the endpoint configuration- invoker 元素，以确保正确调用验证。

有关 JAX-RS 服务生命周期管理的更多详情，请参阅 “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添加 jaxrsValidationFeature bean 作为 JAX-RS 功能并将 exceptionMapper bean 作为 JAX-RS 提供者添加到 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，请参阅“HibernateValidationProviderResolver 类示例”一节。在 OSGi 环境(Apache Karaf)环境上下文中，只需要配置 beanValidationProvider。

注意

记得根据情况将 jaxrs 前缀映射到蓝图或 Spring 的相应 XML 命名空间。

常见 bean 验证 1.1 拦截器

如果要对 bean 验证的配置拥有更精细的控制，您可以单独安装 JAX-RS 拦截器，而不使用 bean 验证功能。配置以下的是一个或多个 JAX-RS 拦截器：

org.apache.cxf.validation.JAXRSBeanValidationInInterceptor

在 JAX-RS 2.0 服务器端点中安装时，根据验证限制验证资源方法参数。如果验证失败，则引发 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，根据验证限制验证响应值。如果验证失败，则引发 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 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，请参阅“HibernateValidationProviderResolver 类示例”一节。在 OSGi 环境(Apache Karaf)环境上下文中，只需要配置 beanValidationProvider。

配置 BeanValidationProvider

您可以将自定义 BeanValidationProvider 实例注入到验证拦截器中，如 “配置 BeanValidationProvider”一节 所述。

配置 JAXRSParameterNameProvider

org.apache.cxf.jaxrs.validation.jaxRSParameterNameProvider 类是 javax.validation.ParameterNameProvider 接口的实施，它可用于在 JAX-RS 2.0 端点上下文中提供方法和构造器参数。