Red Hat Fuse 7.13

Apache CXF 開発ガイド

Apache CXF Web サービスを使用したアプリケーションの開発

Last Updated: 2024-05-28

Red Hat Fuse 7.13 Apache CXF 開発ガイド

Apache CXF Web サービスを使用したアプリケーションの開発

法律上の通知

概要

Apache CXF を使用して Web サービスを開発するためのガイド

目次

多様性を受け入れるオープンソースの強化

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

パート I. WSDL コントラクトの作成

このパートでは、WSDL を使用して Web サービスインターフェイスを定義する方法について説明します。

第1章 WSDL コントラクトの紹介

概要

WSDL ドキュメントは、Web サービス記述言語といくつかの可能な拡張機能を使用してサービスを定義します。ドキュメントには論理的な部分と具体的な部分があります。コントラクトの抽象的な部分は、実装に中立なデータ型とメッセージの観点からサービスを定義します。ドキュメントの具体的な部分は、サービスを実装するエンドポイントがシステム外の世界とどのように相互作用するかを定義します。

サービスを設計するための推奨されるアプローチは、コードを記述する前に、WSDL および XML スキーマでサービスを定義することです。WSDL ドキュメントを手動で編集するときは、ドキュメントが有効であり、正しいことを確認する必要があります。これを行うには、WSDL にある程度精通している必要があります。この規格は、W3C Web サイト (www.w3.org) にあります。

1.1. WSDL ドキュメントの構造

概要

WSDL ドキュメントは、ルート **definition** 要素に含まれる要素のコレクションを指します。これらの要素は、サービスと、そのサービスを実装するエンドポイントにアクセスする方法を記述します。

WSDL ドキュメントには次の 2 つが含まれます。

- 実装中立的な用語でサービスを定義する 論理部分
- サービスを実装するエンドポイントがネットワーク上でどのように公開されるかを定義する 具体的な部分

論理部分

WSDL ドキュメントの論理部分には、**types**、**message**、および **portType** 要素が含まれます。サービスのインターフェイスとサービスによって交換されるメッセージについて説明します。**types** 要素内では、メッセージを設定するデータ構造を定義する XML スキーマが使用されます。サービスで使用されるメッセージの構造を定義するために、多くの **message** 要素が使用されます。**portType** 要素には、サービスによって公開される操作によって送信されるメッセージを定義する 1 つ以上の **operation** 要素が含まれます。

具体的な部分

WSDL ドキュメントの具体的な部分には、**binding** および **service** 要素が含まれます。サービスを実装するエンドポイントが、システムの外にどのように接続するかを説明します。**binding** 要素は、**message** 要素によって記述されるデータユニットが、SOAP などの具体的なオンザワイヤデータフォーマットにどのようにマッピングされるかを記述します。**service** 要素には、サービスを実装するエンドポイントを定義する 1 つ以上の **port** 要素が含まれます。

1.2. WSDL 要素

WSDL ドキュメントは、次の要素で設定されています。

- **definitions**: WSDL ドキュメントのルート要素。この要素の属性は、WSDL ドキュメントの名前、ドキュメントのターゲット名前空間、および WSDL ドキュメントで参照される名前空間の簡略定義を指定します。

- **types** - サービスで使用されるメッセージのビルディングブロックを形成するデータユニットの XML スキーマ定義。データ型の定義については、*2章論理データ単位の定義* を参照してください。
- **message** - サービス操作の呼び出し中に交換されるメッセージの説明。これらの要素は、サービスを設定する操作の引数を定義します。メッセージの定義については、*3章サービスで使用される論理メッセージの定義* を参照してください。
- **portType**: サービスの論理インターフェイスを記述する **operation** 要素のコレクションです。ポートタイプの定義については、*4章論理インターフェイスの定義* を参照してください。
- **operation** - サービスによって実行されるアクションの説明。操作は、操作が呼び出されたときに 2 つのエンドポイント間で渡されるメッセージによって定義されます。操作の定義については、「操作」 を参照してください。
- **binding** - エンドポイントの具体的なデータ形式の仕様。**binding** 要素は、抽象メッセージがエンドポイントによって使用される具体的なデータフォーマットにマップされる方法を定義します。この要素は、パラメーターの順序や戻り値などの詳細が指定される場所です。
- **service**: 関連する **port** 要素のコレクション。これらの要素は、エンドポイント定義を整理するためのリポジトリーです。
- **port** - バインディングおよび物理アドレスによって定義されるエンドポイント。これらの要素は、トランスポートの詳細の定義と組み合わされたすべての抽象的な定義をまとめ、サービスが公開される物理エンドポイントを定義します。

1.3. コントラクトの設計

サービスの WSDL コントラクトを設計するには、次の手順を実行する必要があります。

1. サービスで使用されるデータ型を定義します。
2. サービスで使用されるメッセージを定義します。
3. サービスのインターフェイスを定義します。
4. 各インターフェイスで使用されるメッセージと、ネットワーク上のデータの具体的な表現との間のバインディングを定義します。
5. 各サービスのトランスポートの詳細を定義します。

第2章 論理データ単位の定義

概要

WSDL コントラクトでサービスを記述する場合、複雑なデータ型は XML スキーマを使用して論理ユニットとして定義されます。

2.1. 論理データユニットの概要

サービスを定義するとき、最初に考慮しなければならないことは、公開された操作のパラメーターとして使用されるデータがどのように表されるかです。固定データ構造を使用するプログラミング言語で記述されたアプリケーションとは異なり、サービスは、任意の数のアプリケーションで使用できる論理ユニットでデータを定義する必要があります。これには 2 つのステップが含まれます。

1. データを論理ユニットに分割し、サービスの物理的な実装で使用されるデータ型にマッピングできるようにする
2. 操作を実行するためにエンドポイント間で渡されるメッセージに論理ユニットを結合する

この章では、最初のステップについて説明します。3章 *サービスで使用される論理メッセージの定義*は、2 番目のステップについて説明します。

2.2. データを論理データ単位にマッピング

概要

サービスの実装に使用されるインターフェイスは、操作パラメーターを表すデータを XML ドキュメントとして定義します。すでに実装されているサービスのインターフェイスを定義している場合は、実装されている操作のデータ型を、メッセージにアセンブルできる目立たない XML 要素に変換する必要があります。ゼロから始める場合は、メッセージの作成元となる設定ブロックを決定して、実装の観点から意味をなすようにする必要があります。

サービスデータユニットを定義するのに利用可能な型システム

WSDL 仕様に従って、WSDL コントラクトでデータ型を定義するために選択した任意の型システムを使用できます。ただし、W3C 仕様では、XML スキーマが WSDL ドキュメントに推奨される正規型システムであると規定されています。したがって、XML スキーマは Apache CXF の固有の型システムです。

型システムとしての XML スキーマ

XML スキーマは、XML ドキュメントの構造を定義するために使用されます。これは、ドキュメントを設定する要素を定義することによって行われます。これらの要素は、**xsd:int** などのネイティブ XML スキーマ型や、ユーザーが定義した型を使用できます。ユーザー定義型は、XML 要素の組み合わせを使用して構築されるか、既存の型を制限することによって定義されます。型定義と要素定義を組み合わせることで、複雑なデータを含むことができる複雑な XML ドキュメントを作成できます。

WSDL XML スキーマで使用される場合は、サービスとの対話に使用されるデータを保持する XML ドキュメントの構造を定義します。サービスで使用されるデータユニットを定義するときに、メッセージ部分の構造を指定するタイプとしてそれらを定義できます。データユニットをメッセージ部分を設定する要素として定義することもできます。

データユニットを作成する際の考慮事項

サービスの実装時に使用することを想定しているタイプに直接マップする論理データユニットを作成することを検討してください。このアプローチは機能し、RPC スタイルのアプリケーションを構築するモデルに厳密に従いますが、サービス指向アーキテクチャーの一部を構築するのに必ずしも理想的ではありません。

Web Services Interoperability Organization の WS-I 基本プロファイルは、データユニットを定義するためのいくつかのガイドラインを http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES で提供しています。さらに、W3C は、XML スキーマを使用して WSDL ドキュメントのデータ型を表すための次のガイドラインも提供します。

- 属性ではなく要素を使用します。
- 基本型としてプロトコル固有のタイプを使用しないでください。

2.3. コントラクトへのデータユニットの追加

概要

WSDL コントラクトの作成方法に応じて、新しいデータ定義を作成するには、さまざまな知識が必要です。Apache CXF GUI ツールは、XML スキーマを使用してデータ型を記述するための多くの支援を提供します。他の XML エディターは、さまざまなレベルの支援を提供します。選択するエディターに関係なく、結果として得られるコントラクトがどのようになるかについてある程度の知識を得ることが推奨されます。

手順

WSDL コントラクトで使用されるデータの定義には、次の手順が含まれます。

1. コントラクトで記述されているインターフェイスで使用されているすべてのデータ単位を判別します。
2. コントラクトに **types** 要素を作成します。
3. 例2.1「WSDL コントラクトのスキーマエントリー」 のように、**type** 要素の子として **schema** 要素を作成します。
   targetNamespace 属性は、新規データタイプを定義する namespace を指定します。ベストプラクティスは、ターゲット名前空間へのアクセスを提供する名前空間も定義することです。残りのエントリーは変更しないでください。

   例2.1 WSDL コントラクトのスキーマエントリー

   ```
   <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. 要素のコレクションである複合型ごとに、**complexType** 要素を使用してデータ型を定義します。「データ構造の定義」を参照してください。
5. 各配列について、**complexType** 要素を使用してデータタイプを定義します。「配列の定義」を参照してください。

6. 単純型から派生する複合型ごとに、**simpleType** 要素を使用してデータ型を定義します。「制限によるタイプの定義」 を参照してください。

7. 各列挙型について、**simpleType** 要素を使用してデータタイプを定義します。「列挙型の定義」 を参照してください。

8. 要素ごとに、**element** 要素を使用して定義します。「要素の定義」を参照してください。

2.4. XML スキーマの単純型

概要

メッセージ部分が単純型になる場合は、その型定義を作成する必要はありません。ただし、コントラクトで定義されたインターフェイスで使用される複合型は、単純型を使用して定義されます。

単純型の入力

XML スキーマの単純型は、主にコントラクトの types セクションで使用される **element** 要素に配置されます。これらは、**restriction** 要素および **extension** 要素の **base** 属性でも使用されます。

単純型は、常に **xsd** 接頭辞を使用して入力されます。たとえば、要素が **int** 型であることを指定するには、例2.2「単純型で要素の定義」 に示されるように **type** 属性に **xsd:int** を入力します。

例2.2 単純型で要素の定義

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

サポートされている XSD 単純型

Apache CXF は、次の XML スキーマの単純型をサポートしています。

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

- **xsd:byte**
- **xsd:unsignedByte**
- **xsd:integer**
- **xsd:positiveInteger**
- **xsd:negativeInteger**
- **xsd:nonPositiveInteger**
- **xsd:nonNegativeInteger**
- **xsd:decimal**
- **xsd:dateTime**
- **xsd:time**
- **xsd:date**
- **xsd:QName**
- **xsd:base64Binary**
- **xsd:hexBinary**
- **xsd:ID**
- **xsd:token**
- **xsd:language**
- **xsd:Name**
- **xsd:NCName**
- **xsd:NMTOKEN**
- **xsd:anySimpleType**
- **xsd:anyURI**
- **xsd:gYear**
- **xsd:gMonth**
- **xsd:gDay**
- **xsd:gYearMonth**
- **xsd:gMonthDay**

2.5. 複雑なデータ型の定義

概要

XML スキーマは、単純なデータ型から複雑なデータ構造を構築するための柔軟で強力なメカニズムを提供します。要素と属性のシーケンスを作成することにより、データ構造を作成できます。定義した型を拡張して、さらに複雑な型を作成することもできます。

複雑なデータ構造を構築することに加えて、列挙型、特定の範囲の値を持つデータ型、またはプリミティブ型を拡張または制限することによって特定のパターンに従う必要があるデータ型などの特殊な型を記述することもできます。

2.5.1. データ構造の定義

概要

XML スキーマでは、データフィールドのコレクションであるデータユニットは、**complexType** 要素を使用して定義されます。複合型を指定するには、次の 3 つの情報が必要です。

1. 定義された型の名前は **complexType** 要素の **name** 属性に指定されます。
2. **complexType** の最初の子要素は、それがネットワークに配置される際の構造のフィールドの動作を記述します。「複合型の種類」を参照してください。
3. 定義された構造の各フィールドは、**complexType** 要素の孫である **element** 要素で定義されます。「構造の部分を定義」 を参照してください。

たとえば、例2.3「簡易構造」 は、XML スキーマで 2 つの要素を持つ複合型として定義されています。

例2.3 簡易構造

```
struct personalInfo
{
  string name;
  int age;
};
```

例2.4「複合型」 は、例2.3「簡易構造」 に記載されている構造の可能な XML スキーママッピングの 1 つを表しています。例2.4「複合型」 で定義された構造によって、**name** および **age** の 2 つの要素が含まれるメッセージが生成されます。

.

例2.4 複合型

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

複合型の種類

XML スキーマには、複合型のフィールドが XML ドキュメントとして表され、ネットワーク上で渡され

るときにどのように編成されるかを記述する 3 つの方法があります。**complexType** 要素の最初の子要素は、どの複合型が使用されるかを判断します。表2.1「複合型記述子要素」 は、複合型の動作を定義するのに使用される要素を示しています。

表2.1 複合型記述子要素

要素	複雑型の動作
sequence	複合型のすべてのフィールドが存在する可能性があり、型定義で指定された順序である必要があります。
all	複合型のすべてのフィールドが存在する可能性がありますが、任意の順序で存在する可能性があります。
choice	構造内の要素の 1 つだけをメッセージに配置できます。

例2.5「簡易で複雑な choice 型」 に示されているように **choice** 要素を使用して構造が定義されている場合は、**name** 要素または **age** 要素のいずれかでメッセージを生成します。

例2.5 簡易で複雑な choice 型

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

構造の部分を定義

element 要素を使用して構造を設定するデータフィールドを定義します。すべての **complexType** 要素には、少なくとも 1 つの **element** 要素が含まれている必要があります。**complexType** 要素の各 **element** 要素は、定義したデータ構造のフィールドを表します。

データ構造のフィールドを完全に説明するために、**element** 要素には 2 つの必須属性があります。

- **name** 属性はデータフィールドの名前を指定し、定義された複合型内で一意である必要があります。
- **type** 属性は、フィールドに保存されたデータの型を指定します。タイプは、XML スキーマの単純タイプのいずれか、コントラクトで定義されている任意の名前付き複合タイプのいずれかです。

name と **type** 以外に、**element** 要素には、**minOcurrs** と **maxOccurs** の 2 つの一般的に使用される任意の属性があります。これらの属性は、構造内でフィールドが発生する回数に制限を設けます。デフォルトでは、各フィールドは複合型で 1 回だけ発生します。これらの属性を使用して、フィールドが構造体に表示される必要がある回数、または表示される回数を変更できます。たとえば、例2.6「発生制約のある簡易複合型」 に示されているように、**previousJobs** というフィールドを定義できます。これは 3 回以上、7 回以下発生する必要があります。

例2.6 発生制約のある簡易複合型

```
<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>
```

また、例2.7「minOccurs がゼロに設定された単純な複合型」 で示すように、**minOccurs** をゼロに設定することにより、**minOccurs** を使用して **age** フィールドを任意にすることもできます。この場合、**age** は省略できますが、データは引き続き有効となります。

例2.7 minOccurs がゼロに設定された単純な複合型

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

属性の定義

XML ドキュメントでは、属性は要素のタグに含まれています。たとえば、以下のコードの **complexType** 要素では、**name** は属性です。複合型の属性を指定するには、**complexType** 要素定義で **attribute** 要素を定義します。**attribute** 要素は、**all**、**sequence**、または **choice** 要素の後にのみ表示できます。複合型の属性ごとに **attribute** 要素を 1 つ指定します。**attribute** 要素は、**complexType** 要素の直接の子でなければなりません。

例2.8 属性を持つ複合型

```
<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>
```

前述のコードでは、**attribute** 要素は **personalInfo** 複合型に **age** 属性があることを指定します。**attribute** 要素には以下の属性があります。

- **name** - 属性を識別する文字列を指定する必須属性。
- **type** - フィールドに保存されたデータの型を指定します。タイプは、XML スキーマの単純型の 1 つにすることができます。

- **use** - 複合型にこの属性が必要であるかを指定する任意の属性。有効な値は **required** または **optional** です。デフォルトでは、この属性はオプションです。

attribute 要素では、任意の **default** 属性を指定できます。これにより、属性のデフォルト値を指定できます。

2.5.2. 配列の定義

概要

Apache CXF は、コントラクトで配列を定義する 2 つの方法をサポートしています。1 つ目は、値が 1 より大きい **maxOccurs** 属性のある 1 つの要素を持つ複合型を定義します。2 つ目は、**SOAP** 配列を使用することです。**SOAP** 配列は、多次元配列を簡単に定義したり、まばらに配置された配列を送信したりする機能などの追加機能を提供します。

複合型配列

複合型配列は、シーケンス複合型の特殊なケースです。単一要素で複合型を定義し、**maxOccurs** 属性の値を指定するだけです。たとえば、20 個の浮動小数点数の配列を定義するには、例2.9「複合型配列」 のような複合型を使用します。

例2.9 複合型配列

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

minOccurs 属性の値を指定することもできます。

SOAP 配列

SOAP 配列は、**wsdl:arrayType** 要素を使用して **SOAP-ENC:Array** 基本型から派生することによって定義されます。このための構文は 例2.10「wsdl:arrayType を使用して派生した SOAP 配列の構文」 に示されています。**definitions** 要素が **xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"** を宣言していることを確認します。

例2.10 wsdl:arrayType を使用して派生した SOAP 配列の構文

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

この構文を使用して、**TypeName** は新しく定義された配列型の名前を指定します。**ElementType** は、配列内の要素の型を指定します。**ArrayBounds** は、配列の次元数を指定します。1 次元配列を指定するには、**[]** を使用します。2 次元配列を指定するには、**[][]** または **[,]** を使用します。

たとえば、例2.11「SOAP 配列の定義」 に記載される SOAP 配列、SOAPStrings は文字列の 1 次元配列を定義します。**wsdl:arrayType** 属性は、配列要素の型 **xsd:string** を指定し、次元の数 **[]** は 1 次元を意味します。

例2.11 SOAP 配列の定義

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

また、SOAP 1.1 仕様で説明されているように、単純な要素を使用して SOAP 配列を記述することもできます。このための構文は 例2.12「要素を使用して派生した SOAP 配列の構文」 に示されています。

例2.12 要素を使用して派生した SOAP 配列の構文

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

この構文を使用する場合、要素の **maxOccurs** 属性は、常に **unbounded** に設定する必要があります。

2.5.3. 拡張による型の定義

ほとんどの主要なコーディング言語と同様に、XML スキーマを使用すると、他のデータ型から要素の一部を継承するデータ型を作成できます。これは、拡張による型の定義と呼ばれます。たとえば、**planet** という新しい要素を追加して、例2.4「複合型」 で定義される **personalInfo** 構造を拡張する、**alienInfo** という新しい型を作成できます。

拡張によって定義されるタイプには、次の 4 つの部分があります。

1. 型の名前は、**complexType** 要素の **name** 属性によって定義されます。
2. **complexContent** 要素は、新しい型に複数の要素があることを指定します。

注記

複合型に新しい属性を追加する場合にのみ、**simpleContent** 要素を使用できます。

3. 新しい型が派生される **ベース** 型と呼ばれる型は、**extension** 要素の **base** 属性で指定されます。

4. 新しい型の要素と属性は、通常の複合型と同様に **extension** 要素で定義されます。

たとえば、**alienInfo** は、例2.13「拡張子で定義されたタイプ」 のように定義されます。

例2.13 拡張子で定義されたタイプ

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

2.5.4. 制限によるタイプの定義

概要

XML スキーマを使用すると、XML スキーマの単純型の可能な値を制限することにより、新しい型を作成できます。たとえば、厳密に 9 文字の文字列である単純型 **SSN** を定義できます。単純型を制限することで定義された新しい型は、**simpleType** 要素を使用して定義されます。

制限による型の定義には、次の 3 つのことが必要です。

1. 新しい型の名前は、**simpleType** 要素の **name** 属性によって指定されます。

2. **ベース型** と呼ばれる新しい型が派生される単純型は、**restriction** 要素で指定されます。「基本型の指定」 を参照してください。

3. ベース型の制限を定義する **facet** というルールは、**restriction** 要素の子として定義されます。「制限の定義」 を参照してください。

基本型の指定

基本型は、新しいタイプを定義するために制限されているタイプです。これは **restriction** 要素を使用して指定されます。**restriction** 要素は **simpleType** 要素の唯一の子で、ベース型を指定する 1 つの属性 **base** を持ちます。基本型は、任意の XML スキーマの単純型にすることができます。

たとえば、**xsd:int** の値を制限して新しい型を定義するには、例2.14「基本型として int の使用」 のような定義を使用します。

例2.14 基本型として int の使用

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

制限の定義

基本型に課せられた制限を定義するルールは、**ファセット** と呼ばれます。ファセットとは、ファセットの適用方法を定義する 1 つの属性 **value** を持つ要素です。利用可能なファセットとその有効な **value** 設定は、ベース型によって異なります。たとえば、**xsd:string** は、以下を含む 6 つのファセットをサポートします。

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whitespace**
- **enumeration**

各ファセット要素は **restriction** 要素の子です。

例

例2.15「SSN の単純型の説明」 は、ソーシャルセキュリティー番号を表す単純型 **SSN** の例を示しています。作成される型は **xxx-xx-xxxx** 形式の文字列です。<SSN>032-43-9876<SSN> は、この型の要素に対する有効な値ですが、<SSN>032439876</SSN> は有効ではありません。

例2.15 SSN の単純型の説明

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

2.5.5. 列挙型の定義

概要

XML スキーマの列挙型は、制限による定義の特殊なケースです。これらは、すべての XML スキーマのプリミティブ型でサポートされる **enumeration** ファセットを使用して記述されます。最新のプログラミング言語の列挙型と同様に、この型の変数は、指定された値の 1 つのみを持つことができます。

XML スキーマでの列挙型の定義

列挙型を定義するための構文を 例2.16「列挙型の構文」 に示します。

例2.16 列挙型の構文

```
<simpleType name="EnumName">
```

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

EnumName は、列挙型の名前を指定します。**EnumType** は、ケース値のタイプを指定します。**CaseNValue** (**N** は 1 以上の任意の数値) は、列挙の特定の各ケースの値を指定します。列挙型は任意の数のケース値を持つことができますが、単純型から派生しているため、一度に有効なケース値は 1 つだけです。

例

たとえば、列挙 **widgetSize** で定義された要素を持つ XML ドキュメントは、例2.17「widgetSize 列挙型」 のように、<widgetSize>big</widgetSize> が含まれている場合に有効ですが、<widgetSize>big,mungo</widgetSize> が含まれている場合は有効になりません。

例2.17 widgetSize 列挙型

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

2.6. 要素の定義

XML スキーマの要素は、スキーマから生成された XML ドキュメントの要素のインスタンスを表します。最も基本的な要素は、単一の **element** 要素で設定されます。複合型のメンバーを定義する **element** 要素と同様に、以下の 3 つの属性があります。

- **name** - XML ドキュメントに表示される要素の名前を指定する必須属性。
- **type** - 要素の型を指定します。タイプは、任意の XML スキーマプリミティブ型またはコントラクトで定義された任意の名前付き複合型にすることができます。タイプにインライン定義がある場合、この属性は省略できます。
- **nillable** - ドキュメントから要素を完全に省略できるかどうかを指定します。**nillable** が **true** に設定されている場合、要素はスキーマを使用して生成したドキュメントから省略できます。

要素は、**in-line** 型定義を持つこともできます。インライン型は、**complexType** 要素または **simpleType** 要素のいずれかを使用して指定します。データの型が複雑か単純かを指定すると、各型のデータに使用できるツールを使用して、必要な任意のタイプのデータを定義できます。インライン型の定義は再利用できないため、推奨されません。

第3章 サービスで使用される論理メッセージの定義

概要

サービスは、その操作が呼び出されたときに交換されるメッセージによって定義されます。WSDL コントラクトでは、これらのメッセージは **message** 要素を使用して定義されます。メッセージは、**part** 要素を使用して定義される1つ以上のパーツで設定されます。

概要

サービスの操作は、操作が呼び出されたときに交換される論理メッセージを指定することによって定義されます。これらの論理メッセージは、ネットワークを介して渡されるデータを XML ドキュメントとして定義します。これらには、メソッド呼び出しの一部であるすべてのパラメーターが含まれています。 論理メッセージは、コントラクトの **message** 要素を使用して定義されます。各論理メッセージは、**part** 要素で定義された1つ以上のパーツで設定されます。

メッセージには各パラメーターを個別の部分としてリストできますが、推奨される方法は、操作に必要なデータをカプセル化する単一の部分のみを使用することです。

メッセージおよびパラメーターリスト

サービスによって公開される各操作は、1つの入力メッセージと1つの出力メッセージのみを持つことができます。入力メッセージは、操作が呼び出されたときにサービスが受け取るすべての情報を定義します。出力メッセージは、操作が完了したときにサービスが返すすべてのデータを定義します。障害メッセージは、エラーが発生したときにサービスが返すデータを定義します。

さらに、各操作には任意の数の障害メッセージを含めることができます。障害メッセージは、サービスでエラーが発生したときに返されるデータを定義します。これらのメッセージには通常、コンシューマーがエラーを理解するのに十分な情報を提供する部分が1つだけあります。

レガシーシステムと統合するためのメッセージデザイン

既存のアプリケーションをサービスとして定義する場合は、操作を実装するメソッドで使用される各パラメーターがメッセージで表されていることを確認する必要があります。また、戻り値が操作の出力メッセージに含まれていることを確認する必要があります。

メッセージを定義するための1つのアプローチは、RPC スタイルです。RPC スタイルを使用する場合は、メソッドのパラメーターリストのパラメーターごとに1つの部分を使用してメッセージを定義します。各メッセージパーツは、コントラクトの **types** 要素で定義された型に基づきます。入力メッセージには、メソッドの入力パラメーターごとに1つの部分が含まれています。出力メッセージには、出力パラメーターごとに1つの部分と、必要に応じて戻り値を表す部分が含まれます。パラメーターが入力パラメーターと出力パラメーターの両方である場合、それは入力メッセージと出力メッセージの両方の一部としてリストされます。

RPC スタイルのメッセージ定義は、Tibco や CORBA などのトランスポートを使用するレガシーシステムをサービス対応にする場合に役立ちます。これらのシステムは、手順と方法を中心に設計されています。そのため、呼び出される操作のパラメーターリストに似たメッセージを使用してモデル化するのが最も簡単です。RPC スタイルは、サービスとそれが公開しているアプリケーションの間のよりクリーンなマッピングも作成します。

SOAP サービスのメッセージデザイン

RPC スタイルは既存のシステムのモデリングに役立ちますが、サービスのコミュニティーはラップされ

たドキュメントスタイルを強く支持しています。ラップされたドキュメントスタイルでは、各メッセージは1つの部分で設定されます。メッセージのパーツは、コントラクトの **types** 要素で定義されたラッパー要素を参照します。ラッパー要素には次の特徴があります。

- これは、一連の要素を含む複合型です。詳細は、「複雑なデータ型の定義」 を参照してください。
- 入力メッセージのラッパーの場合:
  - メソッドの入力パラメーターごとに1つの要素があります。
  - その名前は、関連付けられている操作の名前と同じです。
- 出力メッセージのラッパーの場合:
  - メソッドの出力パラメーターごとに1つの要素があり、メソッドの inout パラメーターごとに1つの要素があります。
  - その最初の要素は、メソッドの戻りパラメーターを表します。
  - この名前は、ラッパーが関連付けられる操作の名前に **Response** を追加して生成されます。

メッセージの命名

コントラクト内の各メッセージは、その名前空間内で一意の名前を持っている必要があります。次の命名規則を使用することを推奨します。

- メッセージは、1回の操作でのみ使用する必要があります。
- 入力メッセージ名は、操作名の前に **Request** を追加して形成されます。
- 出力メッセージ名は、操作名の前に **Response** を追加して形成されます。
- 障害メッセージ名は、障害の理由を表す必要があります。

メッセージ部分

メッセージ部分は、論理メッセージの正式なデータ単位です。各パーツは **part** 要素を使用して定義され、**name** 属性と、データ型を指定する **type** 属性または **element** 属性によって識別されます。データ型の属性は、表3.1「part データ型の属性」 にリストされています。

表3.1 part データ型の属性

属性	説明
element="elem_name"	パーツのデータ型は、elem_name という要素によって定義されます。
type="type_name"	part のデータ型は、type_name と呼ばれる型によって定義されます。

メッセージは part の名前を再利用できます。たとえば、メソッドに参照によって渡されるか、入力/出力であるパラメーター **foo** がある場合、例3.1「再利用部分」 のように要求メッセージと応答メッセージの両方の一部にすることができます。

例3.1 再利用部分

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

例

たとえば、個人情報を保存し、従業員の ID 番号に基づいて従業員のデータを返すメソッドを提供するサーバーがあるとします。データを検索するためのメソッドシグネチャーは、例3.2「personalInfo ルックアップメソッド」 のようになります。

例3.2 personalInfo ルックアップメソッド

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

このメソッドシグネチャーは、例3.3「RPC WSDL メッセージ定義」 に示す RPC スタイルの WSDL フラグメントにマッピングできます。

例3.3 RPC WSDL メッセージ定義

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

また、例3.4「ラップされたドキュメントの WSDL メッセージ定義」 に示すラップされたドキュメントスタイルの WSDL フラグメントにマップすることもできます。

例3.4 ラップされたドキュメントの WSDL メッセージ定義

```
<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 論理インターフェイスの定義

概要

論理サービスインターフェイスは **portType** 要素を使用して定義されます。

概要

論理サービスインターフェイスは、WSDL **portType** 要素を使用して定義されます。**portType** 要素は、抽象操作定義のコレクションです。各操作は、操作が表すトランザクションを完了するために使用される入力、出力、および障害メッセージによって定義されます。**portType** 要素で定義されたサービスインターフェイスを実装するためにコードが生成されると、各操作はコントラクトで指定された入力、出力、および障害メッセージで定義されたパラメーターが含まれるメソッドに変換されます。

PROCESS

WSDL コントラクトで論理インターフェイスを定義するには、以下を実行する必要があります。

1. インターフェイス定義を含む **portType** 要素を作成し、一意の名前を付けます。「ポートタイプ」を参照してください。
2. インターフェイスで定義された各オペレーションに **operation** 要素を作成します。「操作」を参照してください。
3. 操作ごとに、操作のパラメーターリスト、戻りタイプ、および例外を表すために使用されるメッセージを指定します。「操作メッセージ」を参照してください。

ポートタイプ

WSDL **portType** 要素は、論理インターフェイス定義のルート要素です。多くの Web サービス実装は、**portType** 要素を生成された実装オブジェクトに直接マップしますが、論理インターフェイス定義は、実装されたサービスによって提供される正確な機能を指定しません。たとえば、ticketSystem という名前の論理インターフェイスは、コンサートチケットを販売するか、駐車違反切符を発行する実装になります。

portType 要素は、バインディングにマッピングされ、定義されたサービスを公開するエンドポイントによって使用される物理データを定義する WSDL ドキュメントの単位です。

WSDL ドキュメントの各 **portType** 要素には、**name** 属性を使用して指定された一意の名前が必要で、これは **operation** 要素に記載される操作のコレクションで設定されます。WSDL ドキュメントは、任意の数のポートの型を記述できます。

操作

WSDL **operation** 要素を使用して定義される論理操作は、2 つのエンドポイント間の対話を定義します。たとえば、当座預金口座の残高の確認とウィジェットの総額の注文の両方を操作として定義できます。

portType 要素内で定義された各操作には、**name** 属性を使用して指定された一意の名前が必要です。操作を定義するには、**name** 属性が必要です。

操作メッセージ

論理操作は、操作を実行するためにエンドポイント間で通信される論理メッセージを表す要素のセットで設定されます。操作を説明できる要素は、表4.1「操作メッセージ要素」 のとおりです。

表4.1操作メッセージ要素

要素	説明
入力 (input)	要求が行われたときにクライアントエンドポイントがサービスプロバイダーに送信するメッセージを指定します。このメッセージの一部は、操作の入力パラメーターに対応しています。
出力 (output)	要求に応答してサービスプロバイダーがクライアントエンドポイントに送信するメッセージを指定します。このメッセージの一部は、参照によって渡される値など、サービスプロバイダーによって変更される可能性のあるすべての操作パラメーターに対応しています。これには、操作の戻り値が含まれます。
fault	エンドポイント間でエラー状態を伝達するために使用されるメッセージを指定します。

操作は、少なくとも1つの **input** 要素または1つの **output** 要素を持つ必要があります。操作は **input** 要素と **output** 要素の両方を持つことができますが、各要素1つだけです。操作に **fault** 要素は必要はありませんが、必要な場合は任意の数の **fault** 要素を持つことができます。

要素には、表4.2「入力要素と出力要素の属性」 にリストされている 2 つの属性があります。

表4.2 入力要素と出力要素の属性

属性	説明
name	操作を具体的なデータ形式にマッピングするときに参照できるように、メッセージを識別します。名前は、囲んでいるポートタイプ内で一意である必要があります。
message	送信または受信されるデータを説明する抽象メッセージを指定します。**message** 属性の値は、WSDL ドキュメントに定義された抽象メッセージの1つの **name** 属性に対応している必要があります。

すべての **input** および **output** 要素の **name** 属性を指定する必要はありません。WSDL はエンクロージング操作の名前を基にデフォルトの命名スキームを提供します。操作で使用される要素が1つだけの場合、要素名はデフォルトで操作の名前になります。**input** および **output** 要素の両方が使用される場合、デフォルトの要素名は、名前にそれぞれ **Request** または **Response** のいずれかが付けられた操作の名前になります。

戻り値

operation 要素は、操作中に渡されるデータの抽象定義であるため、WSDL は操作に指定された戻り値を提供しません。メソッドが値を返す場合、そのメッセージの最後の部分として **output** 要素にマップされます。

例

たとえば、例4.1「personalInfo ルックアップインターフェイス」 のようなインターフェイスを使用している場合があります。

例4.1 personalInfo ルックアップインターフェイス

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

このインターフェイスは、例4.2「personalInfo ルックアップポートタイプ」 のポートタイプにマッピングできます。

例4.2 personalInfo ルックアップポートタイプ

```
<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>
```

パート II. WEB サービスバインディング

このパートでは、Apache CXF バインディングを WSDL ドキュメントに追加する方法について説明します。

第5章 WSDL のバインディングを理解

概要

バインディングは、サービスを定義するのに使用される論理メッセージを、エンドポイントが送受信できる具体的なペイロード形式にマップします。

概要

バインディングは、サービスによって使用される論理メッセージと、エンドポイントが物理的な世界で使用する具体的なデータ形式との間のブリッジを提供します。これらは、論理メッセージがエンドポイントによってネットワーク上で使用されるペイロード形式にどのようにマッピングされるかを説明します。パラメーターの順序、具体的なデータ型、戻り値などの詳細が指定されるのはバインディング内です。たとえば、メッセージの一部をバインディングで並べ替えて、RPC 呼び出しに必要な順序を反映させることができます。バインディングタイプに応じて、メソッドの戻りタイプを表すメッセージ部分がある場合は、それを識別することもできます。

ポートタイプとバインディング

ポートタイプとバインディングは直接関連しています。ポートタイプは、2 つの論理サービス間の一連の相互作用の抽象的な定義です。バインディングは、論理サービスの実装に使用されるメッセージが物理的な世界でどのようにインスタンス化されるかを具体的に定義したものです。次に、各バインディングは、ポートタイプによって定義された論理サービスを公開する 1 つのエンドポイントの定義を完了する一連のネットワーク詳細に関連付けられます。

エンドポイントが単一のサービスのみを定義するようにするために、WSDL では、バインディングが単一のポートタイプのみを表すことができる必要があります。たとえば、2 つのポートタイプとのコントラクトがある場合は、両方を具体的なデータ形式にマップする単一のバインディングを作成することはできません。2 つのバインディングが必要になります。

ただし、WSDL では、ポートタイプを複数のバインディングにマップできます。たとえば、コントラクトのポートタイプが単一の場合、それを 2 つ以上のバインディングにマップできます。各バインディングは、メッセージの一部のマッピング方法を変更したり、メッセージにまったく異なるペイロード形式を指定したりする可能性があります。

WSDL 要素

バインディングは、WSDL **binding** 要素を使用してコントラクトで定義されます。binding 要素は バインディングの一意の名前を指定する **name** や PortType への参照を提供する **type** などの属性で設定されます。この属性の値は、*4章 論理インターフェイスの定義* で説明されているように、バインディングをエンドポイントに関連付けるために使用されます。

実際のマッピングは、**binding** 要素の子で定義されます。これらの要素は、使用するペイロード形式のタイプによって異なります。次の章では、さまざまなペイロード形式とそれらのマッピングを指定するために使用される要素について説明します。

コントラクトへの追加

Apache CXF は、事前定義されたサービスインターフェイスのバインディングを生成できるコマンドラインツールを提供します。

ツールにより、コントラクトに適切な要素が追加されます。ただし、さまざまなタイプのバインディングがどのように機能するかについてある程度の知識を得ることが推奨されます。

任意のテキストエディターを使用して、コントラクトにバインディングを追加することもできます。コントラクトを手動で編集する場合は、コントラクトが有効であることを確認する責任があります。

サポートされているバインディング

Apache CXF は、次のバインディングをサポートしています。

- SOAP 1.1
- SOAP 1.2
- CORBA
- Pure XML

第6章 SOAP1.1 メッセージの使用

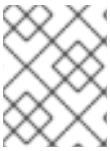
概要

Apache CXF は、SOAP ヘッダーを使用しない SOAP1.1 バインディングを生成するためのツールを提供します。ただし、任意のテキストまたは XML エディターを使用して、バインディングに SOAP ヘッダーを追加できます。

6.1. SOAP1.1 バインディングの追加

wsdl2soap の使用

wsdl2soap を使用して SOAP 1.1 バインディングを生成するには、コマンド **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** を使用します。



注記

wsdl2soap を使用するには、Apache CXF ディストリビューションをダウンロードする必要があります。

このコマンドには、以下のオプションがあります。

オプション	解釈
-i port-type-name	バインディングが生成される **portType** 要素を指定します。
wsdlurl	**portType** 要素定義が含まれる WSDL ファイルのパスと名前。

このツールには、次の任意の引数があります。

オプション	解釈
-b バインディング名	生成される SOAP バインディングの名前を指定します。
-d 出力ディレクトリー	生成される WSDL ファイルを配置するディレクトリーを指定します。
-o 出力ファイル	生成される WSDL ファイルの名前を指定します。
-n SOAP ボディー namespace	スタイルが RPC の場合は、SOAP ボディーの名前空間を指定します。

オプション	解釈
-style (document/rpc)	SOAP バインディングで使用するエンコードスタイル (ドキュメントまたは RPC) を指定します。デフォルトはドキュメントです。
-use (literal/encoded)	SOAP バインディングで使用するバインディングの使用 (エンコードまたはリテラル) を指定します。デフォルトはリテラルです。
-v	ツールのバージョン番号を表示します。
-verbose	コード生成プロセス中にコメントを表示します。
-quiet	コード生成プロセス中のコメントを非表示にします。

-iport-type-name および **wsdlurl** 引数が必要です。**-style rpc** 引数を指定すると、**-nsoap-body-namspace** 引数も必要になります。他のすべての引数は任意であり、任意の順序でリストできます。

重要

wsdl2soap は、**document/encoded** SOAP バインディングの生成をサポートしません。

例

システムに、注文を受け取り、注文を処理するための単一の操作を提供するインターフェイスがある場合、例6.1「注文システムインターフェイス」 に示されているものと同様の **WSDL** フラグメントで定義されます。

例6.1 注文システムインターフェイス

```
<?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>
```

orderWidgets 用に生成された SOAP バインディングは 例6.2「**orderWidgets** の SOAP 1.1 バインディング」 に記載されています。

例6.2 orderWidgets の SOAP 1.1 バインディング

```
<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>
```

このバインディングは、メッセージが **document/literal** メッセージスタイルを使用して送信されることを指定します。

6.2. SOAP1.1 バインディングへの SOAP ヘッダーの追加

概要

SOAP ヘッダーは、**soap:header** 要素をデフォルトの SOAP 1.1 バインディングに追加して定義されます。**soap:header** 要素は、バインディングの **input**、**output**、および **fault** 要素の任意の子です。SOAP ヘッダーは親メッセージの一部になります。SOAP ヘッダーは、メッセージとメッセージ部分を指定することによって定義されます。各 SOAP ヘッダーにはメッセージ部分を 1 つだけ含めることができますが、必要な数の SOAP ヘッダーを挿入できます。

構文

SOAP ヘッダーを定義するための構文を 例6.3「SOAP ヘッダー構文」 に示します。**soap:header** の **message** 属性は、ヘッダーに挿入される部分が取得されたメッセージの修飾名です。**part** 属性は、SOAP ヘッダーに挿入されたメッセージ部分の名前です。SOAP ヘッダーは常にドキュメントスタイル

であるため、SOAP ヘッダーに挿入される WSDL メッセージ部分は要素を使用して定義する必要があります。**message** と **part** 属性はともに、SOAP ヘッダーに挿入するデータを完全に記述します。

例6.3 SOAP ヘッダー構文

```
<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>
```

必須の **message** と **part** 属性と同様に、**soap:header** は、**namespace**、**use**、および **encodingStyle** 属性もサポートします。これらの属性は **soap:body** の場合と同様に **soap:header** でも機能します。

本文とヘッダーの間でメッセージを分割

SOAP ヘッダーに挿入されるメッセージ部分は、コントラクトからの任意の有効なメッセージ部分にすることができます。これは、SOAP 本体として使用されている親メッセージの一部である場合もあります。同じメッセージで情報を 2 回送信する可能性は低いため、SOAP バインディングは、SOAP 本文に挿入されるメッセージ部分を指定する手段を提供します。

soap:body 要素には、部分名のスペース区切りリストを取る任意の属性 **parts** があります。**parts** が定義されると、リストされたメッセージ部分のみが SOAP ボディーに挿入されます。その後、残りの部分を SOAP ヘッダーに挿入できます。

注記

親メッセージの一部を使用して SOAP ヘッダーを定義すると、Apache CXF が自動的に SOAP ヘッダーを入力します。

例

例6.4「SOAP 1.1 SOAP ヘッダーを使用したバインド」 は、例6.1「注文システムインターフェイス」 に示されている **orderWidgets** サービスの変更バージョンを示しています。このバージョンが変更され、各注文のリクエストおよび応答の SOAP ヘッダーに **xsd:base64binary** 値が配置されます。SOAP ヘッダーは、**widgetKey** メッセージの **keyVal** 部分として定義されます。この場合、SOAP ヘッダーは入力メッセージまたは出力メッセージの一部ではないため、アプリケーションロジックに SOAP ヘッダーを追加する必要があります。

例6.4 SOAP 1.1 SOAP ヘッダーを使用したバインド

```
<?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>
```

例6.4「SOAP 1.1 SOAP ヘッダーを使用したバインド」 を変更して、ヘッダー値が入力メッセージと出力メッセージの一部になるようにします。

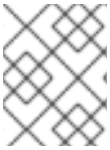
第7章 SOAP 1.2 メッセージの使用

概要

Apache CXF は、SOAP ヘッダーを使用しない SOAP 1.2 バインディングを生成するためのツールを提供します。任意のテキストまたは XML エディターを使用して、バインディングに SOAP ヘッダーを追加できます。

7.1. WSDL ドキュメントへの SOAP 1.2 バインディングの追加

wsdl2soap の使用



注記

wsdl2soap を使用するには、Apache CXF ディストリビューションをダウンロードする必要があります。

wsdl2soap を使用して SOAP 1.2 バインディングを生成するには、コマンド **wsdl2soap**-i**port-type-name**-b**binding-name**-soap12-d**output-directory**-o**output-file**-n**soap-body-namespace**-style (document/rpc)-use (literal/encoded)-v-verbose-quiet**wsdlurl** を使用します。ツールには以下の引数が必要です。

オプション	解釈
-i port-type-name	バインディングが生成される **portType** 要素を指定します。
-soap12	生成されたバインディングが SOAP 1.2 を使用することを指定します。
wsdlurl	**portType** 要素定義が含まれる WSDL ファイルのパスと名前。

このツールには、次の任意の引数があります。

オプション	解釈
-b バインディング名	生成される SOAP バインディングの名前を指定します。
-soap12	生成されたバインディングが SOAP 1.2 を使用することを指定します。
-d 出力ディレクトリー	生成される WSDL ファイルを配置するディレクトリーを指定します。
-o 出力ファイル	生成される WSDL ファイルの名前を指定します。

オプション	解釈
-n SOAP ボディー namespace	スタイルが RPC の場合は、SOAP ボディーの名前空間を指定します。
-style (document/rpc)	SOAP バインディングで使用するエンコードスタイル (ドキュメントまたは RPC) を指定します。デフォルトはドキュメントです。
-use (literal/encoded)	SOAP バインディングで使用するバインディングの使用 (エンコードまたはリテラル) を指定します。デフォルトはリテラルです。
-v	ツールのバージョン番号を表示します。
-verbose	コード生成プロセス中にコメントを表示します。
-quiet	コード生成プロセス中のコメントを非表示にします。

-i port-type-name および **wsdlurl** 引数が必要です。**-style rpc** 引数を指定すると、**-n** soap-body-namspace 引数も必要になります。他のすべての引数は任意であり、任意の順序でリストできます。

重要

wsdl2soap は、**document/encoded** SOAP 1.2 バインディングの生成をサポートしません。

例

システムに、注文を受け取り、注文を処理するための単一の操作を提供するインターフェイスがある場合、例7.1「注文システムインターフェイス」 に示されているものと同様の **WSDL** フラグメントで定義されます。

例7.1 注文システムインターフェイス

```
<?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>
```

orderWidgets 用に生成された SOAP バインディングは 例7.2「orderWidgets の SOAP 1.2 バインディング」 に記載されています。

例7.2 orderWidgets の SOAP 1.2 バインディング

```
<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>
```

このバインディングは、メッセージが **document/literal** メッセージスタイルを使用して送信されることを指定します。

7.2. SOAP 1.2 メッセージへのヘッダーの追加

概要

SOAP メッセージヘッダーは、**soap12:header** 要素を SOAP 1.2 メッセージに追加することで定義されます。**soap12:header** 要素は、バインディングの **input**、**output**、および **fault** 要素の任意の子です。SOAP ヘッダーは親メッセージの一部になります。SOAP ヘッダーは、メッセージとメッセージ部分を指定することによって定義されます。各 SOAP ヘッダーには 1 つのメッセージ部分しか含めることができませんが、必要な数のヘッダーを挿入できます。

構文

SOAP ヘッダーを定義するための構文を 例7.3「SOAP ヘッダー構文」 に示します。

例7.3 SOAP ヘッダー構文

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

soap12:header 要素の属性は、表7.1「**soap12:header** 属性」 で説明されています。

表7.1 soap12:header 属性

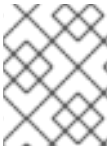
属性	説明
message	ヘッダーに挿入されている部分が取得されるメッセージの修飾名を指定する必須属性。
part	SOAP ヘッダーに挿入されるメッセージ部分の名前を指定する必須属性。
use	メッセージ部分をエンコード規則を使用してエンコードするかどうかを指定します。**encoded** に設定すると、メッセージ部分は、**encodingStyle** 属性の値で指定されたエンコーディングルールを使用してエンコードされます。**literal** に設定すると、メッセージ部分は参照されるスキーマ型によって定義されます。
encodingStyle	メッセージの作成に使用されるエンコード規則を指定します。
namespace	**use="encoded"** でシリアライズされたヘッダー要素に割り当てる namespace を定義します。

本文とヘッダーの間でメッセージを分割

SOAP ヘッダーに挿入されるメッセージ部分は、コントラクトからの任意の有効なメッセージ部分にすることができます。これは、SOAP 本体として使用されている親メッセージの一部である場合もあります。同じメッセージで情報を 2 回送信する可能性は低いため、SOAP 1.2 バインディングは、SOAP 本

文に挿入されるメッセージ部分を指定する手段を提供します。

soap12:body 要素には、部分名のスペース区切りリストを取る任意の属性 **parts** があります。**parts** が定義されると、リストされたメッセージ部分のみが SOAP 1.2 メッセージのボディーに挿入されます。その後、残りの部分をメッセージのヘッダーに挿入できます。



注記

親メッセージの一部を使用して SOAP ヘッダーを定義すると、Apache CXF が自動的に SOAP ヘッダーを入力します。

例

例7.4「SOAP ヘッダーを使用した SOAP 1.2 バインディング」 は、例7.1「注文システムインターフェイス」 に示されている **orderWidgets** サービスの変更バージョンを示しています。このバージョンは変更され、各注文のリクエストおよび応答のヘッダーに **xsd:base64binary** 値が配置されるようになります。ヘッダーは、**widgetKey** メッセージの **keyVal** 部分として定義されます。この場合、ヘッダーは入力メッセージまたは出力メッセージの一部ではないため、ヘッダーを作成するためのアプリケーションロジックを追加する必要があります。

例7.4 SOAP ヘッダーを使用した SOAP 1.2 バインディング

```
<?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>
```

例7.4「SOAP ヘッダーを使用した SOAP 1.2 バインディング」 を変更して、例7.5「SOAP ヘッダーを使用した orderWidgets の SOAP 1.2 バインディング」 に示すように、ヘッダー値が入力メッセージと出力メッセージの一部になるようにします。この場合、**keyVal** は入出力メッセージの一部です。**soap12:body** 要素では、**parts** 属性によって、**keyVal** がボディーに挿入されてはならないことが指定されます。ただし、ヘッダーに挿入されます。

例7.5 SOAP ヘッダーを使用した orderWidgets の SOAP 1.2 バインディング

```
<?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 添付ファイル付きの SOAP を使用したバイナリーデータの送信

概要

SOAP 添付ファイルは、SOAP メッセージの一部としてバイナリーデータを送信するためのメカニズムを提供します。添付ファイルで SOAP を使用するには、SOAP メッセージを MIME マルチパートメッセージとして定義する必要があります。

概要

通常、SOAP メッセージはバイナリーデータを伝送しません。ただし、W3C SOAP 1.1 仕様では、MIME マルチパート/関連メッセージを使用して SOAP メッセージでバイナリーデータを送信できます。この手法は、添付ファイル付きの SOAP を使用して呼び出されます。SOAP 添付ファイルは、W3C の SOAP Messages with Attachments Note で定義されています。

NAMESPACE

MIME マルチパート/関連メッセージの定義に使用される WSDL 拡張機能は、名前空間 **http://schemas.xmlsoap.org/wsdl/mime/** で定義されています。

以下の説明では、この namespace の前に **mime** 接頭辞が付くことを仮定しています。これを設定する WSDL **definitions** 要素のエントリーは 例8.1「コントラクトの MIME 名前空間の仕様」 に表示されます。

例8.1 コントラクトの MIME 名前空間の仕様

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

メッセージバインディングの変更

デフォルトの SOAP バインディングでは、**input**、**output**、および **fault** 要素の最初の子要素は、データを表す SOAP メッセージのボディーを記述する **soap:body** 要素です。添付ファイル付き SOAP を使用する場合、**soap:body** 要素は **mime:multipartRelated** 要素に置き換えられます。

注記

WSDL は、**fault** メッセージに対する **mime:multipartRelated** の使用をサポートしません。

mime:multipartRelated 要素は、メッセージボディーがバイナリーデータが含まれる可能性のあるマルチパートメッセージであることを Apache CXF に伝えます。要素の内容は、メッセージの部分とその内容を定義します。**mime:multipartRelated** 要素には、メッセージの個別部分を記述する 1 つ以上の **mime:part** 要素が含まれます。

最初の **mime:part** 要素には、通常デフォルトの SOAP バインディングに存在する **soap:body** 要素が含まれている必要があります。残りの **mime:part** 要素は、メッセージで送信される添付ファイルを定義します。

MIME マルチパートメッセージの説明

MIME マルチパートメッセージは、複数の **mime:part** 要素が含まれる **mime:multipartRelated** 要素を使用して記述されます。MIME マルチパートメッセージを完全に説明するには、次の手順を実行する必要があります。

1. MIME マルチパートメッセージとして送信する **input** または **output** メッセージで、エンクロージングメッセージの最初の子要素として **mime:mulipartRelated** 要素を追加します。
2. **mime:part** 子要素を **mime:multipartRelated** 要素に追加し、その **name** 属性を一意の文字列に設定します。
3. **soap:body** 要素を **mime:part** 要素の子として追加し、その属性を適切に設定します。

注記

コントラクトにデフォルトの SOAP バインディングがある場合、デフォルトのバインディングの対応するメッセージから **soap:body** 要素を MIME マルチパートメッセージにコピーできます。

4. 別の **mime:part** 子要素を **mime:multipartReleated** 要素に追加し、その **name** 属性を一意の文字列に設定します。
5. **mime:content** 子要素を **mime:part** 要素に追加し、メッセージのこの部分の内容を記述します。
MIME メッセージ部分の内容を完全に記述するために、**mime:content** 要素には次の属性があります。

表8.1mime:content 属性

属性	説明
part	親メッセージ定義から、ネットワーク上の MIME マルチパートメッセージのこの部分の内容として使用される WSDL メッセージ **part** の名前を指定します。 +
type	このメッセージ部分のデータの MIME タイプ。MIME タイプは構文 **type/subtype** を使用してタイプまたはサブタイプとして定義されます。 + **image/jpeg** および **text/plain** といった事前定義された MIME タイプは複数あります。MIME タイプは、Internet Assigned Numbers Authority (IANA) によって維持され、Multipurpose Internet Mail Extensions (MIME) パート 1: インターネットメッセージ本文 と Multipurpose Internet Mail Extensions (MIME) パート 2: メディアタイプ で詳細に説明されています。 +

6. 追加の MIME 部分ごとに、手順 [i303819] および [i303821] を繰り返します。

例

例8.2「添付ファイル付きの SOAP を使用したコントラクト」 は、X 線を JPEG 形式で保存するサービスを定義する WSDL フラグメントを示しています。イメージデータ **xRay** は、**xsd:base64binary** として保存され、MIME マルチパートメッセージの 2 番目の部分 **imageData** にパッケージ化されます。入力メッセージの残り 2 つの部分である **patientName** および **patientNumber** は、SOAP ボディーの一部として MIME マルチパートイメージの最初の部分で送信されます。

例8.2 添付ファイル付きの SOAP を使用したコントラクト

```
<?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 SOAP MTOM を使用したバイナリーデータの送信

概要

SOAP Message Transmission Optimization Mechanism (MTOM) は、XML メッセージの一部としてバイナリーデータを送信するためのメカニズムとして、SOAP を添付ファイルに置き換えます。Apache CXF で MTOM を使用するには、サービスのコントラクトに正しいスキーマタイプを追加し、MTOM の最適化を有効にする必要があります。

9.1. MTOM の概要

SOAP Message Transmission Optimization Mechanism (MTOM) は、SOAP メッセージの一部としてバイナリーデータを送信するための最適化された方法を指定します。添付ファイル付きの SOAP とは異なり、MTOM では、バイナリーデータを送信するために XML-binary Optimized Packaging (XOP) パッケージを使用する必要があります。MTOM を使用してバイナリーデータを送信する場合は、SOAP バインディングの一部として MIME マルチパート/関連メッセージを完全に定義する必要はありません。ただし、次のことを行う必要があります。

1. 添付ファイルとして送信するデータに アノテーション を付けます。
   データを実装する WSDL または Java クラスのいずれかにアノテーションを付けることができます。

2. ランタイムの MTOM サポートを 有効 にします。
   これは、プログラムで、または設定を通して行うことができます。

3. 添付ファイルとして渡されるデータの **DataHandler** を開発します。

注記

DataHandler の開発は、本ガイドの対象外です。

9.2. MTOM を使用するためのデータ型へのアノテーション

概要

WSDL では、イメージファイルやサウンドファイルなど、バイナリーデータのブロックに渡すデータ型を定義する場合、**xsd:base64Binary** 型のデータの要素を定義します。デフォルトでは、**xsd:base64Binary** 型のすべての要素によって、MTOM を使用してシリアライズ可能な **byte[]** が生成されます。ただし、コードジェネレーターのデフォルトの動作では、シリアル化を十分に活用していません。

MTOM を十分に活用するには、サービスの WSDL ドキュメントまたはバイナリーデータ構造を実装する JAXB クラスのいずれかにアノテーションを追加する必要があります。WSDL ドキュメントにアノテーションを追加すると、コードジェネレーターはバイナリーデータのストリーミングデータハンドラーを生成します。JAXB クラスにアノテーションを付けるには、適切なコンテンツタイプを指定する必要があり、バイナリーデータを含むフィールドのタイプ指定を変更することが必要になる場合もあります。

最初に WSDL

例9.1「MTOM へのメッセージ」 は、1つの文字列フィールド、1つの整数フィールド、およびバイナリーフィールドを含むメッセージを使用する Web サービスの WSDL ドキュメントを示しています。バイナリーフィールドは大きなイメージファイルを運ぶことを目的としているため、通常の SOAP メッ

セージの一部として送信することは適切ではありません。

例9.1 MTOM へのメッセージ

```
<?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>
```

MTOM を使用してメッセージのバイナリー部分を最適化された添付ファイルとして送信する場合は、**xmime:expectedContentTypes** 属性をバイナリーデータが含まれる要素に属性を追加する必要があります。この属性は、http://www.w3.org/2005/05/xmlmime 名前空間で定義され、要素に含まれると予想される MIME タイプを指定します。MIME タイプのコンマ区切りリストを指定できます。この属性の設定により、コードジェネレーターがデータの JAXB クラスを作成する方法が変更されます。ほとんどの MIME タイプでは、コードジェネレーターが DataHandler を作成します。イメージ用などの一部の MIME タイプには、マッピングが定義されています。

注記

MIME タイプは、Internet Assigned Numbers Authority (IANA) によって維持され、Multipurpose Internet Mail Extensions (MIME) パート 1: インターネットメッセージ本文 と Multipurpose Internet Mail Extensions (MIME) パート 2: メディアタイプ で詳細に説明されています。

ほとんどの場合、**application/octet-stream** を指定します。

例9.2「MTOM のバイナリーデータ」 は、MTOM を使用のするために 例9.1「MTOM へのメッセージ」 から **xRayType** を変更する方法を示しています。

例9.2 MTOM のバイナリーデータ

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

xRayType 用に生成された JAXB クラスには **byte[]** が含まれなくなりました。代わりに、コードジェネレーターは **xmime:expectedContentTypes** 属性を認識し、imageData フィールドの DataHandler を生成します。

注記

MTOM を使用するために **binding** 要素を変更する必要はありません。ランタイムは、データが送信されるときに適切な変更を行います。

最初に Java

Java の最初の開発を行っている場合は、次のようにして JAXB クラス MTOM を準備できます。

1. バイナリーデータを保持するフィールドが DataHandler であることを確認してください。
2. MTOM 添付ファイルとしてストリーミングするデータを含むフィールドに **@XmlMimeType()** アノテーションを追加します。

例9.3「MTOM の JAXB クラス」 は、MTOM を使用するためにアノテーションが付けられた JAXB クラスを示しています。

例9.3 MTOM の JAXB クラス

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

9.3. MTOM の有効化

デフォルトでは、Apache CXF ランタイムは MTOM サポートを有効にしません。通常の SOAP メッセージの一部として、または最適化されていない添付ファイルとしてすべてのバイナリーデータを送信します。MTOM サポートは、プログラムを用いて、または設定を使用してアクティベートできます。

9.3.1. JAX-WS API の使用

概要

サービスプロバイダーとコンシューマーの両方で、MTOM の最適化を有効にする必要があります。JAX-WS API は、各タイプのエンドポイントに異なるメカニズムを提供します。

Service provider

JAX-WS API を使用してサービスプロバイダーを公開した場合は、以下のようにランタイムの MTOM サポートを有効にします。

1. 公開されたサービスの **Endpoint** オブジェクトにアクセスします。
   Endpoint オブジェクトにアクセスするための最も簡単な方法として、エンドポイントをパブリッシュするときが挙げられます。詳細は、31章 *サービスの公開* を参照してください。
2. 例9.4「エンドポイントからの SOAP バインディングの取得」 で示すように、**getBinding()** メソッドを使用して **Endpoint** から SOAP バインディングを取得します。

   例9.4 エンドポイントからの SOAP バインディングの取得

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

MTOM プロパティーにアクセスするには、返されたバインディングオブジェクトを **SOAPBinding** オブジェクトにキャストする必要があります。

3. 例9.5「サービスプロバイダーの MTOM 有効プロパティーの設定」 に示されるように、バインディングの **setMTOMEnabled()** メソッドを使用して、バインディングの MTOM enabled プロパティーを **true** に設定します。

例9.5 サービスプロバイダーの MTOM 有効プロパティーの設定

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

コンシューマー

MTOM で JAX-WS コンシューマーを有効にするには、以下を行う必要があります。

1. コンシューマーのプロキシーを **BindingProvider** オブジェクトにキャストします。コンシューマープロキシーの取得は、25章*WSDL コントラクトなしでコンシューマーの開発* または 28章*WSDL コントラクトからのコンシューマーの開発* を参照してください。

2. 例9.6「**BindingProvider** からの SOAP バインディングの取得」 で示すように、**getBinding()** メソッドを使用して **BindingProvider** から SOAP バインディングを取得します。

例9.6 BindingProvider からの SOAP バインディングの取得

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

3. 例9.7「コンシューマーの MTOM 有効プロパティーの設定」 に示されるように、バインディングの **setMTOMEnabled()** メソッドを使用して、バインディングの MTOM enabled プロパティーを **true** に設定します。

例9.7 コンシューマーの MTOM 有効プロパティーの設定

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

9.3.2. 設定の使用

概要

コンテナーへのデプロイ時など、XML を使用してサービスを公開する場合、エンドポイントの MTOM サポートをエンドポイントの設定ファイルで有効にすることができます。エンドポイントの設定に関する詳細は、パートIV「Web サービスエンドポイントの設定」 を参照してください。

手順

MTOM プロパティーは、エンドポイントの **jaxws:endpoint** 要素内に設定されます。MTOM を有効にするには、以下を実行します。

1. **jaxws:property** 子要素をエンドポイントの **jaxws:endpoint** 要素に追加します。

2. **entry** 子要素を **jaxws:property** 要素に追加します。

3. **entry** 要素の **key** 属性を **mtom-enabled** に設定します。
4. **entry** 要素の **value** 属性を **true** に設定します。

例

例9.8「MTOM の有効化設定」 は、MTOM が有効なエンドポイントを示しています。

例9.8 MTOM の有効化設定

```
<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 XML ドキュメントの使用

概要

Pure XML ペイロードフォーマットは、SOAP エンベロープのオーバーヘッドなしでシンプルな XML ドキュメントを使用してサービスをデータの交換を可能にすることで、SOAP バインディングの代替手段を提供します。

XML バインディング名前空間

XML 形式のバインディングを記述するために使用される拡張機能は namespace http://cxf.apache.org/bindings/xformat で定義されます。Apache CXF ツールは、接頭辞 **xformat** を使用して XML バインディングエクステンションを表します。コントラクトに次の行を追加します。

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

ハンドエディティング

インターフェイスを純粋な XML ペイロードフォーマットにマッピングするには、以下を行います。

1. namespace 宣言を追加して、XML バインディングを定義するエクステンションを追加します。「XML バインディング名前空間」を参照してください。
2. XML バインディングを保持するための標準 WSDL **binding** 要素をコントラクトに追加し、バインディングに一意の **name** を付け、バインドされたインターフェイスを表す WSDL **portType** 要素の名前を指定します。
3. **xformat:binding** 子要素を **binding** 要素に追加し、メッセージが SOAP エンベロープなしで純粋な XML ドキュメントとして処理されることを特定します。
4. 必要に応じて、**xformat:binding** 要素の **rootNode** 属性を有効な QName に設定します。**rootNode** 属性の影響の詳細は、「ネットワーク上の XML メッセージ」 を参照してください。
5. バインドされたインターフェイスで定義された各操作に対して、標準の WSDL **operation** 要素を追加して、操作のメッセージのバインディング情報を保持します。
6. バインディングに追加された操作ごとに **input**、**output**、および **fault** 子要素を追加して、操作によって使用されるメッセージを表します。
   これらの要素は、論理操作のインターフェイス定義で定義されたメッセージに対応します。
7. オプションで、有効な **rootNode** 属性を持つ **xformat:body** 要素を、追加された **input**、**output**、および **fault** 要素に追加して、バインディングレベルで設定された **rootNode** の値をオーバーライドします。

注記

void を返す操作の出力メッセージなど、メッセージのパートがない場合は、ネットワークで書き込んだメッセージが有効な空の XML ドキュメントであるように、メッセージの **rootNode** 属性を設定する必要があります。

ネットワーク上の XML メッセージ

インターフェイスのメッセージを SOAP エンベロープを使わずに XML 文書として渡すように指定する場合、メッセージがワイヤー上に書かれるときに有効な XML 文書を形成するように注意する必要があります。また、XML ドキュメントを受信する Apache CXF 参加者が Apache CXF によって生成されたメッセージを認識できるようにする必要があります。

両問題を解決する簡単な方法は、グローバル **xformat:binding** 要素または個別のメッセージの **xformat:body** 要素のいずれかで、オプションの **rootNode** 属性を使用することです。**rootNode** 属性は、Apache CXF が生成した XML ドキュメントの root ノードとして機能する要素の QName を指定します。**rootNode** 属性が設定されていない場合、Apache CXF は doc スタイルのメッセージを使用する際はルート要素としてメッセージ部分のルート要素を使用し、rpc スタイルのメッセージを使用する際はルート要素としてメッセージ部分の名前を使用する要素を使用します。

たとえば、**rootNode** 属性が設定されていない場合、例10.1「有効な XML バインディングメッセージ」で定義されたメッセージは、ルート要素 **lineNumber** を持つ XML ドキュメントを生成します。

例10.1 有効な XML バインディングメッセージ

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

1 つの部分を持つメッセージでは、**rootNode** 属性が設定されていなくても、Apache CXF は常に有効な XML ドキュメントを生成します。ただし、例10.2「無効な XML バインディングメッセージ」 のメッセージは、無効な XML ドキュメントを生成します。

例10.2 無効な XML バインディングメッセージ

```
<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>
```

XML バインディングで指定された **rootNode** 属性がない場合、Apache CXF は 例10.2「無効な XML バインディングメッセージ」 で定義されたメッセージに対して 例10.3「無効な XML ドキュメント」 のような XML ドキュメントを生成します。生成された XML ドキュメントは、**pairName** および **entryNum** の 2 つのルート要素があるため無効です。

例10.3 無効な XML ドキュメント

```
<pairName>
```

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

例10.4「rootNode が設定された XML Binding」 に示されているように、**rootNode** 属性を設定すると、Apache CXF は指定されたルート要素の要素をラップします。この例では、**rootNode** 属性はバインディング全体に対して定義され、ルート要素に entrants という名前を指定しています。

例10.4 rootNode が設定された 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>
```

入力メッセージから生成された XML ドキュメントは 例10.5「rootNode 属性を使用して生成された XML ドキュメント」 に似ています。XML ドキュメントにはルート要素が 1 つしかないことに注意してください。

例10.5 rootNode 属性を使用して生成された XML ドキュメント

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

バインディングの ROOTNODE 属性設定の上書き

また、メッセージバインディング内の **xformat:body** 要素を使用して、個別のメッセージの **rootNode** 属性を設定したり、特定のメッセージのグローバル設定をオーバーライドしたりすることもできます。たとえば、例10.4「rootNode が設定された XML Binding」 で定義した出力メッセージに、入力メッセージとは異なるルート要素がある場合、例10.6「**xformat:body** の使用」 に示されるようにバインディングのルート要素を上書きできます。

例10.6 xformat:body の使用

```
<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>
```

パート III. WEB サービストランスポート

このパートでは、Apache CXF トランスポートを WSDL ドキュメントに追加する方法について説明します。

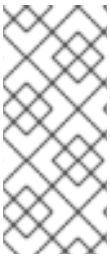
第11章 WSDL でエンドポイントを定義する方法について

概要

エンドポイントは、インスタンス化されたサービスを表します。これらは、バインディングと、エンドポイントを公開するために使用されるネットワークの詳細を組み合わせて定義されます。

概要

エンドポイントは、サービスの物理的なマニフェストとして考えることができます。これは、サービスが使用する論理データの物理表現を指定するバインディングと、サービスを他のエンドポイントによる問い合わせ可能にするために使用される物理接続の詳細を定義する一連のネットワーク詳細を組み合わせたものです。



注記

CXF プロバイダーは、クライアントに対応する CXF コンシューマーのサーバーです。ルートの開始エンドポイントとして CXF (**camel-cxf**) コンポーネントを使用している場合、エンドポイントは Camel コンシューマーと CXF プロバイダーの両方になります。Camel CXF コンポーネントをルートで終了エンドポイントとして使用する場合、エンドポイントは Camel プロデューサーおよび CXF コンシューマーの両方になります。

エンドポイントおよびサービス

バインディングが単一インターフェイスのみをマップできるのと同じ方法で、エンドポイントは単一のサービスにのみマッピングできます。ただし、サービスは任意の数のエンドポイントでマニフェストできます。たとえば、4 つの異なるエンドポイントによってマニフェストされたチケット公開サービスを定義できます。ただし、チケット販売サービスとウィジェット販売サービスの両方をマニフェストしたエンドポイントを 1 つ入れることはできませんでした。

WSDL 要素

エンドポイントは、WSDL **service** 要素と WSDL **port** 要素の組み合わせを使用してコントラクトに定義されます。**service** 要素は、関連する **port** 要素のコレクションです。**port** 要素は、実際のエンドポイントを定義します。

WSDL **service** 要素には一意の名前を指定する 1 つの属性 **name** があります。**service** 要素は、関連する **port** 要素のコレクションの親要素として使用されます。WSDL では、**port** 要素の関連性について何も指定しません。適切であると見なされる方法で **port** 要素を関連付けることができます。

WSDL **port** 要素には、エンドポイントで使用されるバインディングを指定する **binding** 属性があり、これは **wsdl:binding** 要素への参照になります。また、**name** 属性も含まれます。これは、すべてのポート間で一意の名前を提供する必須の属性です。**port** 要素は、エンドポイントによって使用される実際のトランスポートの詳細を指定する要素の親要素です。トランスポート詳細の指定に使用される要素は、以下のセクションで説明します。

コントラクトへのエンドポイントの追加

Apache CXF は、事前定義されたサービスインターフェイスとバインディングの組み合わせのエンドポイントを生成できるコマンドラインツールを提供します。

ツールにより、コントラクトに適切な要素が追加されます。ただし、エンドポイントの定義で異なるトランスポートがどのように使用されるかについてある程度理解していることが推奨されます。

テキストエディターを使用して、エンドポイントをコントラクトに追加することもできます。コントラクトを手動で編集する場合は、コントラクトが有効であることを確認する責任があります。

サポートされるトランスポート

エンドポイント定義は、Apache CXF がサポートするトランスポートごとに定義されたエクステンションを使用して構築されます。これには、以下のトランスポートが含まれます。

- HTTP
- CORBA
- Java Messaging Service

第12章 HTTP の使用

概要

HTTP は Web の基礎となるトランスポートです。エンドポイント間の通信に標準化された堅牢で柔軟なプラットフォームを提供します。これらの要素により、ほとんどの WS-* 仕様のトランスポートが想定され、RESTful アーキテクチャーに不可欠です。

12.1. 基本的な HTTP エンドポイントの追加

代替 HTTP ランタイム

Apache CXF は、以下の代替の HTTP ランタイム実装をサポートします。

- Undertow: 「Undertow ランタイムの設定」 で詳しく説明されています。
- Netty (「Netty ランタイムの設定」) で詳細に説明されています。

Netty HTTP URL

通常、HTTP エンドポイントはクラスパス (Undertow または Netty のいずれか) に含まれる HTTP ランタイムを使用します。ただし、Undertow ランタイムと Netty ランタイムの両方がクラスパスに含まれている場合は、Undertow ランタイムがデフォルトで使用されるため、Netty ランタイムをいつ使用するかを明示的に指定する必要があります。

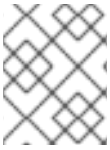
クラスパスで複数の HTTP ランタイムが使用可能な場合は、エンドポイント URL を次の形式で指定することにより、Undertow ランタイムを選択できます。

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

ペイロードのタイプ

使用しているペイロード形式に応じて、HTTP エンドポイントのアドレスを指定する 3 つの方法があります。

- SOAP 1.1 は標準化された **soap:address** 要素を使用します。
- SOAP 1.2 は **soap12:address** 要素を使用します。
- その他のペイロード形式はすべて **http:address element.** を使用します。



注記

Camel 2.16.0 リリースでは、Apache Camel CXF Payload は追加設定なしでストリームキャッシュをサポートします。

SOAP 1.1

HTTP 経由で SOAP 1.1 メッセージを送信する場合は、SOAP 1.1 **address** 要素を使用してエンドポイントのアドレスを指定する必要があります。エンドポイントのアドレスを URL として指定する 1 つの属性 **location** があります。SOAP 1.1 **address** 要素は、namespace http://schemas.xmlsoap.org/wsdl/soap/ で定義されています。

例12.1「SOAP 1.1 ポート要素」 は、HTTP 経由で SOAP 1.1 メッセージを送信するために使用される **port** 要素を示しています。

例12.1 SOAP 1.1 ポート要素

```
<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

HTTP 経由で SOAP 1.2 メッセージを送信する場合は、SOAP 1.2 **address** 要素を使用してエンドポイントのアドレスを指定する必要があります。エンドポイントのアドレスを URL として指定する 1 つの属性 **location** があります。SOAP 1.2 **address** 要素は、namespace http://schemas.xmlsoap.org/wsdl/soap12/ で定義されています。

例12.2「SOAP 1.2 ポート要素」 は、HTTP 経由で SOAP 1.2 メッセージを送信するために使用される **port** 要素を示しています。

例12.2 SOAP 1.2 ポート要素

```
<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>
```

その他のメッセージタイプ

メッセージが SOAP 以外のペイロードフォーマットにマッピングされている場合は、HTTP **address** 要素を使用して、エンドポイントのアドレスを指定する必要があります。エンドポイントのアドレスを URL として指定する 1 つの属性 **location** があります。HTTP **address** 要素は、namespace http://schemas.xmlsoap.org/wsdl/http/ で定義されています。

例12.3「HTTP ポート要素」 は、XML メッセージの送信に使用される **port** 要素を示しています。

例12.3 HTTP ポート要素

```
<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. コンシューマーの設定

12.2.1. HTTP コンシューマーエンドポイントのメカニズム

HTTP コンシューマーエンドポイントは、エンドポイントがリダイレクト応答を自動的に受け入れるかどうか、エンドポイントがチャンクを使用できるかどうか、エンドポイントがキープアライブを要求するかどうか、エンドポイントがプロキシーと対話する方法など、いくつかの HTTP 接続属性を指定できます。HTTP 接続プロパティーの他に、HTTP コンシューマーエンドポイントはセキュリティー保護する方法を指定できます。

コンシューマーエンドポイントは、以下の 2 つのメカニズムを使用して設定できます。

- 設定
- WSDL

12.2.2. 設定の使用

Namespace

HTTP コンシューマーエンドポイントの設定に使用される要素は、名前空間 http://cxf.apache.org/transports/http/configuration で定義されています。通常、接頭辞 **http-conf** を使用して参照されます。HTTP 設定要素を使用するには、例12.4「HTTP コンシューマー設定名前空間」 にある行をエンドポイント設定ファイルの **beans** 要素に追加する必要があります。また、設定要素の namespace を **xsi:schemaLocation** 属性に追加する必要があります。

例12.4 HTTP コンシューマー設定名前空間

```
<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 ランタイムまたは Netty ランタイム

http-conf namespace の要素を使用して、Undertow ランタイムまたは Netty ランタイムを設定できます。

conduit 要素

http-conf:conduit 要素とその子を使用して、HTTP コンシューマーエンドポイントを設定します。**http-conf:conduit** 要素は、エンドポイントに対応する WSDL **port** 要素を指定する単一の属性 **name** を取ります。**name** 属性の値は、**portQName**`.http-conduit` の形式を取ります。例12.5「**http-conf:conduit 要素**」 は、エンドポイントのターゲット namespace が http://widgets.widgetvendor.net の場合に WSDL フラグメント **<port binding="widgetSOAPBinding" name="widgetSOAPPort>** によって指定されるエンドポイントの設定を追加するために使用される **http-conf:conduit** 要素を示しています。

例12.5 http-conf:conduit 要素

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

http-conf:conduit 要素には、設定情報を指定する子要素があります。これらは、表12.1「HTTP コンシューマーエンドポイントの設定に使用する要素」 に説明があります。

表12.1 HTTP コンシューマーエンドポイントの設定に使用する要素

要素	説明
http-conf:client	タイムアウト、キープアライブリクエスト、コンテンツタイプなどの HTTP 接続プロパティーを指定します。「クライアント要素」を参照してください。
http-conf:authorization	エンドポイントがプリエンプションで使用する Basic 認証方法を設定するためのパラメーターを指定します。**http-conf:basicAuthSupplier** オブジェクトを指定することが推奨されます。
http-conf:proxyAuthorization	送信 HTTP プロキシーサーバーに対して Basic 認証を設定するためのパラメーターを指定します。
http-conf:tlsClientParameters	SSL/TLS の設定に使用するパラメーターを指定します。
http-conf:basicAuthSupplier	エンドポイントによって使用される基本認証情報または 401 HTTP チャレンジへの応答として、エンドポイントによって使用される基本認証情報を提供するオブジェクトの Bean 参照またはクラス名を指定します。
http-conf:trustDecider	情報送信前に HTTPS サービスプロバイダーとのコネクションに対して信頼を確立するために HTTP(S) **URLConnection** オブジェクトをチェックするオブジェクトの Bean 参照またはクラス名を指定します。

クライアント要素

http-conf:client 要素は、コンシューマーエンドポイントの HTTP コネクションのセキュリティー以外のプロパティーを設定するために使用されます。表12.2「HTTP コンシューマー設定の属性」 で説明されている属性で、接続のプロパティーを指定します。

表12.2 HTTP コンシューマー設定の属性

属性	説明
ConnectionTimeout	コンシューマーがタイムアウトするまでの接続の確立を試みる期間 (ミリ秒単位) を指定します。デフォルトは **30000** です。 **0** を指定すると、コンシューマーはリクエストの送信を無期限に続行します。
ReceiveTimeout	コンシューマーがタイムアウトするまでの応答を待つ期間 (ミリ秒単位) を指定します。デフォルトは **30000** です。 **0** を指定すると、コンシューマーは無期限に待機します。
AutoRedirect	コンシューマーが自動的に発行されたリダイレクトに従うかどうかを指定します。デフォルトは **false** です。
MaxRetransmits	コンシューマーがリダイレクトを満たすリクエストを再送信する最大回数を指定します。デフォルトは **-1** で、無制限の再送信が許可されることを指定します。
AllowChunking	コンシューマーがチャンクを使用して要求を送信するかどうかを指定します。デフォルトは **true** で、リクエストの送信時にコンシューマーがチャンクを使用することを指定します。 以下のいずれかが true の場合、チャンクは使用できません。 ● **http-conf:basicAuthSupplier** が、クレデンシャルを先制的に提供するように設定されている。 ● **AutoRedirect** は **true** に設定されている。 いずれの場合も、**AllowChunking** の値は無視され、チャンクは許可されません。
Accept	コンシューマーを処理する用意があるメディアタイプを指定します。値は HTTP **Accept** プロパティーの値として使用されます。属性の値は、多目的インターネットメール拡張 (MIME) タイプを使用して指定されます。

属性	説明
AcceptLanguage	応答を受信する目的でコンシューマーが好む言語 (たとえば、アメリカ英語) を指定します。この値は、HTTP **AcceptLanguage** プロパティーの値として使用されます。 言語タグは、国際標準化機構 (ISO) によって規制されており、通常、ISO-639 標準によって決定される言語コードと ISO-3166 標準によって決定される国コードをハイフンで区切って組み合わせることによって形成されます。たとえば、en-US はアメリカ英語を表します。
AcceptEncoding	コンシューマーを処理する用意があるコンテンツエンコーディングを指定します。コンテンツエンコーディングラベルは、Internet Assigned Numbers Authority (IANA) により規制されています。この値は、HTTP **AcceptEncoding** プロパティーの値として使用されます。
ContentType	メッセージのボディーに送信されるデータのメディアタイプを指定します。メディアタイプは、多目的インターネットメール拡張機能 (MIME) タイプを使用して指定されます。値は、HTTP **ContentType** プロパティーの値として使用されます。デフォルトは **text/xml** です。 Web サービスの場合、これを **text/xml** に設定する必要があります。クライアントが CGI スクリプトに HTML フォームデータを送信している場合、これを **application/x-www-form-urlencoded** に設定する必要があります。HTTP POST リクエストが固定されたペイロードフォーマットにバインドされている場合 (SOAP ではなく)、コンテンツ型は通常 **application/octet-stream** に設定されます。
ホスト	要求が呼び出されるリソースのインターネットホストおよびポート番号を指定します。値は HTTP **Host** プロパティーの値として使用されます。 通常この属性は必要ありません。これは、特定の DNS シナリオまたはアプリケーション設計でのみ必要です。たとえば、クライアントがクラスターに優先するホスト (つまり、同じインターネットプロトコル (IP) アドレスへのマッピング) を示します。

属性	説明
Connection	各リクエスト/レスポンスダイアログの後に、特定の接続を開いたままにするか、閉じるかどうかを指定します。有効な値は 2 つあります。 ● **Keep-Alive** (デフォルト) - 最初のリクエスト/応答シーケンスの後に、コネクションを開いたままにしておくことをコンシューマーが希望することを指定します。サーバーがそれに従うと、コンシューマーが閉じられるまで接続が開かれます。 ● **close** - 各リクエスト/応答シーケンスの後に、サーバーへのコネクションが閉じられるよう指定します。
CacheControl	コンシューマーからサービスプロバイダーへの要求を設定するチェーンに含まれるキャッシュが順守する必要のある動作に関するディレクティブを指定します。「コンシューマーキャッシュ制御ディレクティブ」を参照してください。
cookie	すべての要求で送信される静的クッキーを指定します。
BrowserType	要求の発信元のブラウザーに関する情報を指定します。World Wide Web コンソーシアム (W3C) の HTTP 仕様では、これは **ユーザーエージェント** とも呼ばれます。一部のサーバーは、リクエストを送信するクライアントに基づいて最適化されます。

属性	説明
Referer	特定のサービスで要求を行うようにコンシューマーに指示するリソースの URL を指定します。この値は、HTTP **Referer** プロパティーの値として使用されます。 この HTTP プロパティーは、要求が URL を入力せずにハイパーリンクの結果をクリックすると、使用されます。これにより、サーバーは以前のタスクフローに基づいて処理を最適化し、ログ記録、最適化されたキャッシュ、廃止されたリンクやタイプミスのあるリンクのトレースなどの目的でリソースへのバックリンクのリストを生成できます。ただし、通常は Web サービスアプリケーションでは使用されません。 **AutoRedirect** 属性が **true** に設定されていて、リクエストがリダイレクトされた場合、**Referer** 属性で指定された値はすべてオーバーライドされます。HTTP 参照プロパティーの値は、コンシューマーの元のリクエストをリダイレクトするサービスの URL に設定されます。
DecoupledEndpoint	別のプロバイダー→コンシューマー接続を介して応答を受信するための分離されたエンドポイントの URL を指定します。分離されたエンドポイントの使用の詳細については、「分離モードでの HTTP トランスポートの使用」 を参照してください。 分離されたエンドポイントが機能するには、WS-Addressing を使用するようにコンシューマーエンドポイントとサービスプロバイダーエンドポイントの両方を設定する必要があります。
ProxyServer	要求がルーティングされるプロキシーサーバーの URL を指定します。
ProxyServerPort	要求がルーティングされるプロキシーサーバーのポート番号を指定します。
ProxyServerType	要求のルーティングに使用されるプロキシーサーバータイプを指定します。有効な値は以下のとおりです。 • **HTTP**(デフォルト) • **SOCKS**

例

例12.6「HTTP コンシューマーエンドポイントの設定」 は、リクエスト間でプロバイダーへの接続を開いたままにし、呼び出しごとに 1 回だけリクエストを再送信し、チャンクストリームを使用できない HTTP コンシューマーエンドポイントの設定を示しています。

例12.6 HTTP コンシューマーエンドポイントの設定

```
<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>
```

補足情報

HTTP コンジットの詳細については、16章コンジット を参照してください。

12.2.3. WSDL の使用

Namespace

HTTP コンシューマーエンドポイントの設定に使用される WSDL 要素は、名前空間 **http://cxf.apache.org/transports/http/configuration** で定義されています。通常、接頭辞 **http-conf** を使用して参照されます。HTTP 設定要素を使用するには、例12.7「HTTP Consumer WSDL Element の名前空間」 にある行をエンドポイントの WSDL ドキュメントの **definitions** 要素に追加する必要があります。

例12.7 HTTP Consumer WSDL Element の名前空間

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

Undertow ランタイムまたは Netty ランタイム

http-conf namespace の要素を使用して、Undertow ランタイムまたは Netty ランタイムを設定できます。

クライアント要素

http-conf:client 要素は、WSDL ドキュメントで HTTP コンシューマーのコネクションプロパティーを指定するために使用されます。**http-conf:client** 要素は WSDL **port** 要素の子です。設定ファイルで使用される **client** 要素と同じ属性があります。属性については、表12.2「HTTP コンシューマー設定の属

性」 で説明されています。

例

例12.8「HTTP コンシューマーエンドポイントを設定する WSDL」 は、キャッシュと対話しないことを示す HTTP コンシューマーエンドポイントを設定する WSDL フラグメントを示しています。

例12.8 HTTP コンシューマーエンドポイントを設定する WSDL

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

12.2.4. コンシューマーキャッシュ制御ディレクティブ

表12.3「**http-conf:client** キャッシュ制御ディレクティブ」 に、HTTP コンシューマーがサポートするキャッシュ制御ディレクティブのリストを示します。

表12.3 http-conf:client キャッシュ制御ディレクティブ

ディレクティブ	動作
no-cache	キャッシュは、サーバーで最初にその応答を再検証しない限り、特定の応答を使用して後続の要求を満たすことはできません。この値で特定の応答ヘッダーフィールドが指定されていると、制限は応答内のそれらのヘッダーフィールドにのみ適用されます。応答ヘッダーフィールドが指定されていないと、制限は応答全体に適用されます。
no-store	キャッシュは、応答の一部またはそれを呼び出した要求の一部を格納してはなりません。
max-age	コンシューマーは、指定された秒単位の時間以下の応答を受け入れることができます。
max-stale	コンシューマーは、有効期限を超えた応答を受け入れることができます。値が max-stale に割り当てられている場合、それは、応答の有効期限を超えて、コンシューマーがその応答を受け入れることができる秒数を表します。値が割り当てられていない場合、コンシューマーは任意の年齢の古い応答を受け入れることができます。
min-fresh	コンシューマーは、少なくとも指定された秒数の間、まだ新鮮な応答を望んでいます。

ディレクティブ	動作
no-transform	キャッシュは、プロバイダーとコンシューマー間の応答でコンテンツのメディアタイプや場所を変更することはできません。
only-if-cached	キャッシュは、キャッシュに現在保存されている応答のみを返し、リロードまたは再検証が必要な応答は返しません。
cache-extension	他のキャッシュディレクティブへの追加の拡張機能を指定します。拡張機能は、情報提供または動作のいずれかになります。拡張ディレクティブは標準ディレクティブのコンテキストで指定されるため、拡張ディレクティブを理解していないアプリケーションは、標準ディレクティブで義務付けられている動作に準拠できます。

12.3. サービスプロバイダーの設定

12.3.1. HTTP サービスプロバイダーのメカニズム

HTTP サービスプロバイダーのエンドポイントは、キープアライブリクエストを受け入れるかどうか、キャッシュとの対話方法、コンシューマーとの通信におけるエラーの許容度など、いくつかの HTTP 接続属性を指定できます。

サービスプロバイダーエンドポイントは、以下の 2 つのメカニズムを使用して設定できます。

- 設定
- WSDL

12.3.2. 設定の使用

Namespace

HTTP プロバイダーエンドポイントの設定に使用される要素は、名前空間 http://cxf.apache.org/transports/http/configuration で定義されています。通常、接頭辞 **http-conf** を使用して参照されます。HTTP 設定要素を使用するには、例12.9「HTTP プロバイダー設定名前空間」 にある行をエンドポイント設定ファイルの **beans** 要素に追加する必要があります。また、設定要素の namespace を **xsi:schemaLocation** 属性に追加する必要があります。

例12.9 HTTP プロバイダー設定名前空間

```
<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 ランタイムまたは Netty ランタイム

http-conf namespace の要素を使用して、Undertow ランタイムまたは Netty ランタイムを設定できます。

destination 要素

http-conf:destination 要素およびその子を使用して HTTP サービスプロバイダーエンドポイントを設定します。**http-conf:destination** 要素は、エンドポイントに対応する WSDL **port** 要素を指定する単一の属性 **name** を取ります。**name** 属性の値は、**portQName`.http-destination`** の形式を取ります。例12.10「**http-conf:destination** 要素」 は、エンドポイントのターゲット namespace http://widgets.widgetvendor.net の場合に、WSDL フラグメント **<port binding="widgetSOAPBinding" name="widgetSOAPPort>** によって指定されるエンドポイントの設定を追加するために使用される **http-conf:destination** 要素を示しています。

例12.10 http-conf:destination 要素

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

http-conf:destination 要素には、設定情報を指定する複数の子要素があります。表12.4「HTTP サービスプロバイダーエンドポイントの設定に使用される要素」 に説明があります。

表12.4 HTTP サービスプロバイダーエンドポイントの設定に使用される要素

要素	説明
http-conf:server	HTTP 接続プロパティーを指定します。「サーバー要素」を参照してください。
http-conf:contextMatchStrategy	HTTP リクエストを処理するためにコンテキスト一致ストラテジーを設定するパラメーターを指定します。
http-conf:fixedParameterOrder	この宛先によって処理される HTTP リクエストのパラメーター順序が固定されるかどうかを指定します。

サーバー要素

http-conf:server 要素は、サービスプロバイターエンドポイントの HTTP コネクションのプロパティーを設定するために使用されます。表12.5「HTTP サービスプロバイダー設定属性」 で説明されている属性で、接続のプロパティーを指定します。

表12.5 HTTP サービスプロバイダー設定属性

属性	説明
ReceiveTimeout	接続がタイムアウトする前に、サービスプロバイダーが要求の受信を試行する時間の長さをミリ秒単位で設定します。デフォルトは **30000** です。 **0** はプロバイダーがタイムアウトしないことを指定します。
SuppressClientSendErrors	リクエストの受信時に例外が出力されるかどうかを指定します。デフォルトは **false** で、エラーが発生すると例外が出力されます。
SuppressClientReceiveErrors	コンシューマーへの応答の送信時にエラーが発生したときに例外を出力するかどうかを指定します。デフォルトは **false** で、エラーが発生すると例外が出力されます。
HonorKeepAlive	応答の送信後に、サービスプロバイダーが接続のリクエストを開いたままにするかどうかを指定します。デフォルトは **false** で、キープアライブ要求は無視されます。
RedirectURL	クライアント要求で指定された URL が要求されたリソースに対して適切でなくなった場合に、クライアント要求がリダイレクトされる URL を指定します。この場合、サーバー応答の最初の行にステータスコードが自動的に設定されていない場合、ステータスコードは 302 に設定され、ステータスの説明はオブジェクト移動に設定されます。値は HTTP RedirectURL プロパティーの値として使用されます。
CacheControl	サービスプロバイダーからコンシューマーへの応答を設定するチェーンに含まれるキャッシュが順守する必要のある動作に関するディレクティブを指定します。「サービスプロバイダーキャッシュ制御ディレクティブ」を参照してください。
ContentLocation	応答で送信されるリソースがある URL を設定します。
ContentType	応答で送信される情報のメディアタイプを指定します。メディアタイプは、多目的インターネットメール拡張機能 (MIME) タイプを使用して指定されます。値は、HTTP ContentType の場所の値として使用されます。

属性	説明
ContentEncoding	サービスプロバイダーによって送信される情報に適用されている追加のコンテンツエンコーディングを指定します。コンテンツエンコーディングラベルは、Internet Assigned Numbers Authority (IANA) により規制されています。コンテンツエンコーディング値には、zip、gzip、compress、deflate、および identity が含まれます。この値は、HTTP ContentEncoding プロパティーの値として使用されます。 コンテンツエンコーディングの主な用途は、zip や gzip などのエンコーディングメカニズムを使用してドキュメントを圧縮できるようにすることです。Apache CXF はコンテンツコードの検証を実行しません。指定されたコンテンツコーディングがアプリケーションレベルでサポートされていることを確認するのはユーザーの責任です。
ServerType	応答を送信しているサーバーのタイプを指定します。値は **program-name/version** の形式を取ります (例: **Apache/1.2.5**)。

例

例12.11「HTTP サービスプロバイダーエンドポイントの設定」 は、キープアライブ要求を尊重し、すべての通信エラーを抑制する HTTP サービスプロバイダーエンドポイントの設定を示しています。

例12.11 HTTP サービスプロバイダーエンドポイントの設定

```
<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. WSDL の使用

Namespace

HTTP プロバイダーエンドポイントの設定に使用される WSDL 要素は、名前空間 http://cxf.apache.org/transports/http/configuration で定義されています。通常、接頭辞 **http-conf** を使用して参照されます。HTTP 設定要素を使用するには、例12.12「HTTP プロバイダーの WSDL 要素の名前空間」 にある行をエンドポイントの WSDL ドキュメントの **definitions** 要素に追加する必要があります。

例12.12 HTTP プロバイダーの WSDL 要素の名前空間

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

Undertow ランタイムまたは Netty ランタイム

http-conf namespace の要素を使用して、Undertow ランタイムまたは Netty ランタイムを設定できます。

サーバー要素

http-conf:server 要素は、WSDL ドキュメントで HTTP サービスプロバイダーのコネクションプロパティーを指定するために使用されます。**http-conf:server** 要素は WSDL **port** 要素の子です。設定ファイルで使用される **server** 要素と同じ属性があります。属性については、表12.5「HTTP サービスプロバイダー設定属性」 で説明されています。

例

例12.13「HTTP サービスプロバイダーエンドポイントを設定する WSDL」 は、キャッシュと対話しないことを指定する HTTP サービスプロバイダーエンドポイントを設定する WSDL フラグメントを示しています。

例12.13 HTTP サービスプロバイダーエンドポイントを設定する WSDL

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

12.3.4. サービスプロバイダーキャッシュ制御ディレクティブ

表12.6「**http-conf:server** キャッシュ制御ディレクティブ」 HTTP サービスプロバイダーがサポートするキャッシュ制御ディレクティブをリスト表示します。

表12.6 http-conf:server キャッシュ制御ディレクティブ

ディレクティブ	動作

ディレクティブ	動作
no-cache	キャッシュは、サーバーで最初にその応答を再検証しない限り、特定の応答を使用して後続の要求を満たすことはできません。この値で特定の応答ヘッダーフィールドが指定されていると、制限は応答内のそれらのヘッダーフィールドにのみ適用されます。応答ヘッダーフィールドが指定されていないと、制限は応答全体に適用されます。
public	すべてのキャッシュは応答を保存できます。
プライベート	パブリック (**共有**) キャッシュは、応答が 1 人のユーザーを対象としているため、応答を保存できません。この値で特定の応答ヘッダーフィールドが指定されていると、制限は応答内のそれらのヘッダーフィールドにのみ適用されます。応答ヘッダーフィールドが指定されていないと、制限は応答全体に適用されます。
no-store	キャッシュは、応答の一部またはそれを呼び出した要求の一部を格納してはなりません。
no-transform	キャッシュは、サーバーとクライアント間の応答でコンテンツのメディアタイプまたは場所を変更してはなりません。
must-revalidate	キャッシュは、応答に関連する期限切れのエントリーを再検証してから、そのエントリーを後続の応答で使用できるようにする必要があります。
proxy-revalidate	must-revalidate と同じですが、共有キャッシュにのみ適用でき、プライベート非共有キャッシュでは無視される点が異なります。このディレクティブを使用する場合は、パブリックキャッシュディレクティブも使用する必要があります。
max-age	クライアントは、指定された秒数を超えない経過時間の応答を受け入れることができます。
s-max-age	max-age と同じですが、共有キャッシュにのみ適用でき、プライベート非共有キャッシュによって無視される点が異なります。s-max-age で指定された年齢は、max-age で指定された年齢を上書きします。このディレクティブを使用する場合は、proxy-revalidate ディレクティブも使用する必要があります。

ディレクティブ	動作
cache-extension	他のキャッシュディレクティブへの追加の拡張機能を指定します。拡張機能は、情報提供または動作のいずれかになります。拡張ディレクティブは標準ディレクティブのコンテキストで指定されるため、拡張ディレクティブを理解していないアプリケーションは、標準ディレクティブで義務付けられている動作に準拠できます。

12.4. UNDERTOW ランタイムの設定

概要

Undertow ランタイムは、分離されたエンドポイントを使用する HTTP サービスプロバイダーと HTTP コンシューマーによって使用されます。ランタイムのスレッドプールを設定できます。また、Undertow ランタイムを介して HTTP サービスプロバイダーのセキュリティー設定をいくつか設定することもできます。

Maven 依存関係

Apache Maven をビルドシステムとして使用する場合は、プロジェクトの **pom.xml** ファイルに以下の依存関係を追加して、Undertow ランタイムをプロジェクトに追加できます。

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

Namespace

Undertow ランタイムの設定に使用される要素は、名前空間 **http://cxf.apache.org/transports/http-undertow/configuration** で定義されています。Undertow 設定要素を使用するには、例 12.14「Undertow ランタイム設定名前空間」 にある行をエンドポイント設定ファイルの **beans** 要素に追加する必要があります。この例では、名前空間に接頭辞 **httpu** が割り当てられています。また、設定要素の namespace を **xsi:schemaLocation** 属性に追加する必要があります。

例12.14 Undertow ランタイム設定名前空間

```
<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 要素

httpu:engine-factory 要素は、アプリケーションによって使用される Undertow ランタイムの設定に使用されるルート要素です。この属性には単一の必須属性 **bus** があり、その値は、設定されている Undertow インスタンスを管理する **Bus** の名前です。

注記

値は通常、デフォルトの **Bus** インスタンスの名前である **cxf** です。

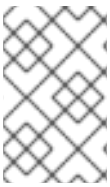
http:engine-factory 要素には、Undertow ランタイムファクトリーによってインスタンス化された HTTP ポートの設定に使用される情報が含まれる 3 つの子があります。子は、表12.7「Undertow ランタイムファクトリーを設定するための要素」 で説明されています。

表12.7 Undertow ランタイムファクトリーを設定するための要素

要素	説明
httpu:engine	特定の Undertow ランタイムインスタンスの設定を指定します。「engine 要素」を参照してください。
httpu:identifiedTLSServerParameters	HTTP サービスプロバイダーを保護するための再利用可能なプロパティーのセットを指定します。これには、プロパティーセットを参照できる一意の識別子を指定する単一の属性 **id** があります。
httpu:identifiedThreadingParameters	Undertow インスタンスのスレッドプールを制御するための再利用可能なプロパティーセットを指定します。これには、プロパティーセットを参照できる一意の識別子を指定する単一の属性 **id** があります。 「スレッドプールの設定」を参照してください。

engine 要素

httpu:engine 要素は、Undertow ランタイムの特定のインスタンスを設定するために使用されます。これには、Undertow インスタンスによって管理される **port** の数を指定する、組み込み undertow とポートを持つグローバル IP アドレスを指定する **host** の 2 つの属性があります。



注記

port 属性に **0** の値を指定することができます。**port** 属性が **0** に設定された **httpu:engine** 要素に指定されたスレッドプロパティーは、明示的に設定されていないすべての Undertow リスナーの設定として使用されます。

各 **httpu:engine** 要素には、セキュリティープロパティーを設定する子と Undertow インスタンスのスレッドプールを設定する子の 2 つの子があります。設定の各タイプに対して、設定情報を直接提供するか、親 **httpu:engine-factory** 要素で定義された設定プロパティーのセットへの参照を指定することもできます。

設定プロパティーの提供に使用される子要素は 表12.8「Undertow ランタイムインスタンスを設定するための要素」 で説明されています。

表12.8 Undertow ランタイムインスタンスを設定するための要素

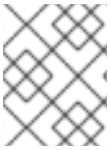
要素	説明
httpu:tlsServerParameters	特定の Undertow インスタンスに使用されるセキュリティーを設定するためのプロパティーのセットを指定します。
httpu:tlsServerParametersRef	**identifiedTLSServerParameters** 要素によって定義されたセキュリティープロパティーのセットを参照します。**id** 属性は、参照される **identifiedTLSServerParameters** 要素の ID を提供します。
httpu:threadingParameters	特定の Undertow インスタンスによって使用されるスレッドプールのサイズを指定します。「スレッドプールの設定」を参照してください。
httpu:threadingParametersRef	**identifiedThreadingParameters** 要素によって定義されるプロパティーのセットを参照します。**id** 属性は、参照される **identifiedThreadingParameters** 要素の ID を提供します。

スレッドプールの設定

Undertow インスタンスのスレッドプールのサイズは、以下のいずれかによって設定できます。

- **engine-factory** 要素の **identifiedThreadingParameters** 要素を使用して、スレッドプールのサイズを指定します。次に、t**hreadingParametersRef** 要素を使用して要素を参照します。
- **threadingParameters** 要素を使用して、スレッドプールのサイズを直接指定します。

threadingParameters には、スレッドプールのサイズを指定する属性が 2 つあります。属性については、表12.9「Undertow スレッドプールを設定するための属性」 で説明されています。



注記

httpu:identifiedThreadingParameters 要素には、単一の子 **threadingParameters** 要素があります。

表12.9 Undertow スレッドプールを設定するための属性

属性	説明
workerIOThreads	ワーカー用に作成される I/O スレッドの数を指定します。指定しない場合、デフォルト値が選択されます。デフォルト値は、CPU コアごとに 1 つの I/O スレッドです。
minThreads	リクエストの処理に Undertow インスタンスが使用できるスレッドの最小数を指定します。

属性	説明
maxThreads	リクエストの処理に Undertow インスタンスが使用できるスレッドの最大数を指定します。

例

例12.15「Undertow インスタンスの設定」 は、ポート番号 9001 で Undertow インスタンスを設定する設定フラグメントを示しています。

例12.15 Undertow インスタンスの設定

```
<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>
```

同時要求およびキューサイズの制限

Request Limiting Handler を設定して、同時接続リクエストの最大数と、Undertow サーバーインスタンスによって処理できるキューサイズに制限を設定できます。この設定の例を以下に示します。例12.16「接続要求およびキューサイズの制限」

表12.10 Request Limiting ハンドラーを設定するための属性

属性	説明
maximumConcurrentRequests	Undertow インスタンスで処理できる同時リクエストの最大数を指定します。リクエストの数がこの制限を超えると、リクエストはキューに入れられます。
queueSize	Undertow インスタンスによる処理のためにキューに入れられる可能性のあるリクエストの総数を指定します。リクエストの数がこの制限を超えると、リクエストは拒否されます。

例12.16 接続要求およびキューサイズの制限

```
<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. NETTY ランタイムの設定

概要

Netty ランタイムは、切り離されたエンドポイントを使用して HTTP サービスプロバイダーおよび HTTP コンシューマーによって使用されます。ランタイムのスレッドプールは設定でき、Netty ランタイムを介して HTTP サービスプロバイダーのセキュリティー設定を多数設定することもできます。

Maven 依存関係

Apache Maven をビルドシステムとして使用する場合は、プロジェクトの **pom.xml** ファイルに以下の依存関係を追加し、Netty ランタイムのサーバー側の実装 (Web サービスエンドポイントを定義するため) をプロジェクトに追加できます。

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

プロジェクトの **pom.xml** ファイルに以下の依存関係を含めることで、Netty ランタイムのクライアント側実装 (Web サービスクライアントを定義するため) をプロジェクトに追加できます。

```
<dependency>
   <groupId>org.apache.cxf</groupId>
```

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

Namespace

Netty ランタイムの設定に使用される要素は、名前空間 **http://cxf.apache.org/transports/http-netty-server/configuration** で定義されています。通常、接頭辞 **httpn** を使用して参照されます。Netty 設定要素を使用するには、例12.17「Netty ランタイム設定名前空間」 にある行をエンドポイント設定ファイルの **beans** 要素に追加する必要があります。また、設定要素の namespace を **xsi:schemaLocation** 属性に追加する必要があります。

例12.17 Netty ランタイム設定名前空間

```
<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 要素

httpn:engine-factory 要素は、アプリケーションによって使用される Netty ランタイムを設定するために使用されるルート要素です。1 つの必須属性 **bus** があります。この値は、設定される Netty インスタンスを管理する **Bus** の名前です。

注記

値は通常、デフォルトの **Bus** インスタンスの名前である **cxf** です。

httpn:engine-factory 要素には、Netty ランタイムファクトリーによってインスタンス化された HTTP ポートの設定に使用される情報が含まれる 3 つの子があります。子は、表12.11「Netty ランタイムファクトリーを設定するための要素」 で説明されています。

表12.11 Netty ランタイムファクトリーを設定するための要素

要素	説明
httpn:engine	特定の Netty ランタイムインスタンスの設定を指定します。「engine 要素」を参照してください。
httpn:identifiedTLSServerParameters	HTTP サービスプロバイダーを保護するための再利用可能なプロパティーのセットを指定します。これには、プロパティーセットを参照できる一意の識別子を指定する単一の属性 **id** があります。

要素	説明
httpn:identifiedThreadingParameters	Netty インスタンスのスレッドプールを制御するための再利用可能なプロパティーセットを指定します。これには、プロパティーセットを参照できる一意の識別子を指定する単一の属性 **id** があります。 「スレッドプールの設定」を参照してください。

engine 要素

httpn:engine 要素は、Netty ランタイムの特定のインスタンスを設定するために使用されます。表12.12「Netty ランタイムインスタンスを設定するための属性」 は、**httpn:engine** 要素によってサポートされる属性を示します。

表12.12 Netty ランタイムインスタンスを設定するための属性

属性	説明
port	Netty HTTP サーバーインスタンスによって使用されるポートを指定します。port 属性に **0** の値を指定できます。port 属性が **0** に設定された状態で engine 要素に指定されたスレッドプロパティーは、明示的に設定されていないすべての Netty リスナーの設定として使用されます。
host	Netty HTTP サーバーインスタンスによって使用されるリッスンアドレスを指定します。値はホスト名または IP アドレスになります。指定されていない場合、Netty HTTP サーバーはすべてのローカルアドレスをリッスンします。
readIdleTime	Netty 接続の最大読み取りアイドル時間を指定します。タイマーは、基礎となるストリームに読み取りアクションがある場合は常にリセットされます。
writeIdleTime	Netty 接続の最大書き込みアイドル時間を指定します。タイマーは、基礎となるストリームに書き込みアクションがある場合は常にリセットされます。
maxChunkContentSize	Netty 接続の集約された最大コンテンツサイズを指定します。デフォルト値は 10MB です。

httpn:engine 要素には、セキュリティープロパティーを設定する子要素が 1 つあります。また、Netty インスタンスのスレッドプールを設定するための 1 つの子要素。設定の各タイプに対して、設定情報を直接提供するか、親 **httpn:engine-factory** 要素で定義された設定プロパティーのセットへの参照を指定できます。

httpn:engine でサポートされる子要素が 表12.13「Netty ランタイムインスタンスを設定するための要素」 に表示されます。

表12.13 Netty ランタイムインスタンスを設定するための要素

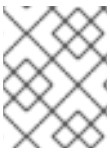
要素	説明
httpn:tlsServerParameters	特定の Netty インスタンスに使用されるセキュリティーを設定するための一連のプロパティーを指定します。
httpn:tlsServerParametersRef	**identifiedTLSServerParameters** 要素によって定義されたセキュリティープロパティーのセットを参照します。**id** 属性は、参照される **identifiedTLSServerParameters** 要素の ID を提供します。
httpn:threadingParameters	特定の Netty インスタンスが使用するスレッドプールのサイズを指定します。「スレッドプールの設定」を参照してください。
httpn:threadingParametersRef	**identifiedThreadingParameters** 要素によって定義されるプロパティーのセットを参照します。**id** 属性は、参照される **identifiedThreadingParameters** 要素の ID を提供します。
httpn:sessionSupport	**true** の場合、HTTP セッションのサポートを有効にします。デフォルトは **false** です。
httpn:reuseAddress	**ReuseAddress** TCP ソケットオプションを設定するブール値を指定します。デフォルトは **false** です。

スレッドプールの設定

Netty インスタンスのスレッドプールのサイズは、次のいずれかで設定できます。

- **engine-factory** 要素の **identifiedThreadingParameters** 要素を使用して、スレッドプールのサイズを指定します。次に、t**hreadingParametersRef** 要素を使用して要素を参照します。
- **threadingParameters** 要素を使用して、スレッドプールのサイズを直接指定します。

threadingParameters 要素には、表12.14「Netty スレッドプールを設定するための属性」 で説明されているようにスレッドプールのサイズを指定する 1 つの属性があります。



注記

httpn:identifiedThreadingParameters 要素には、単一の子 **threadingParameters** 要素があります。

表12.14 Netty スレッドプールを設定するための属性

属性	説明
threadPoolSize	Netty インスタンスがリクエストを処理するために使用できるスレッドの数を指定します。

例

例12.18「Netty インスタンスの設定」 は、さまざまな Netty ポートを設定する設定フラグメントを示しています。

例12.18 Netty インスタンスの設定

```
<?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. 分離モードでの HTTP トランスポートの使用

概要

通常の HTTP 要求/応答シナリオでは、要求と応答は同じ HTTP 接続を使用して送信されます。サービスプロバイダーは要求を処理し、適切な HTTP ステータスコードと応答の内容を含む応答で応答します。リクエストが成功した場合、HTTP ステータスコードは 200 に設定されます。

WS-RM を使用している場合や、要求の実行に長時間かかる場合など、場合によっては、要求メッセージと応答メッセージを分離することが理にかなっています。この場合、サービスプロバイダーは、要求が受信された HTTP 接続のバックチャネルを介して、202 Accepted 応答をコンシューマーに送信します。その後、リクエストを処理し、新しい分離された server→client HTTP 接続を使用して応答をコンシューマーに送信します。コンシューマーランタイムは着信応答を受信し、アプリケーションコードに戻る前に、それを適切な要求と関連付けます。

分離された対話の設定

分離モードで HTTP トランスポートを使用するには、次のことを行う必要があります。

1. WS-Addressing を使用するようにコンシューマーを設定します。
   「WS-Addressing を使用するためのエンドポイントの設定」を参照してください。

2. 分離されたエンドポイントを使用するようにコンシューマーを設定します。
   「コンシューマーの設定」を参照してください。

3. WS-Addressing を使用するためにコンシューマーが対話するサービスプロバイダーを設定します。
   「WS-Addressing を使用するためのエンドポイントの設定」を参照してください。

WS-Addressing を使用するためのエンドポイントの設定

コンシューマーおよびコンシューマーが対話するサービスプロバイダーが WS-Addressing を使用することを指定します。

エンドポイントが WS-Addressing を使用するように指定するには、次の 2 つの方法のいずれかを使用します。

- 例12.19「WSDL を使用した WS-Addressing のアクティブ化」 に示すように、**wswa:UsingAddressing** 要素をエンドポイントの WSDL **port** 要素に追加します。

例12.19 WSDL を使用した WS-Addressing のアクティブ化

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

- 例12.20「ポリシーを使用した WS-Addressing のアクティブ化」 に示されるように、WS-Addressing ポリシーをエンドポイントの WSDL **port** 要素に追加します。

例12.20 ポリシーを使用した WS-Addressing のアクティブ化

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

注記

WS-Addressing ポリシーは **wswa:UsingAddressing** WSDL 要素よりも優先されます。

コンシューマーの設定

http-conf:conduit 要素の **DecoupledEndpoint** 属性を使用して、分離されたエンドポイントを使用するようにコンシューマーエンドポイントを設定します。

例12.21「結合された HTTP エンドポイントを使用するためのコンシューマーの設定」 は、例12.19「WSDL を使用した WS-Addressing のアクティブ化」 で定義されたエンドポイントを設定し、分離したエンドポイントを使用する設定を示しています。コンシューマーは **http://widgetvendor.net/widgetSellerInbox** ですべての応答を受け取るようになりました。

例12.21 結合された HTTP エンドポイントを使用するためのコンシューマーの設定

```
<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">
```

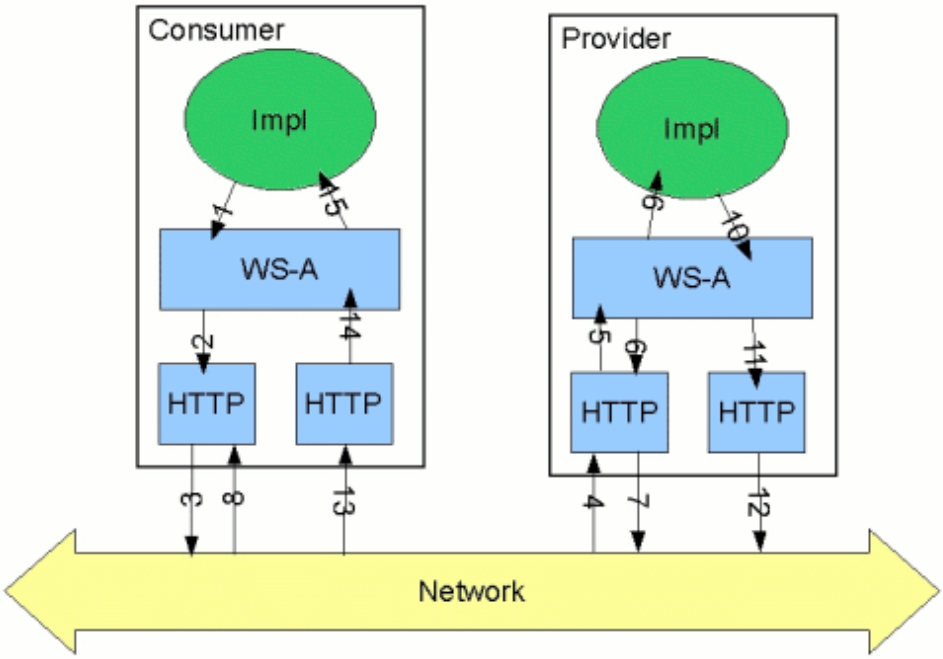
```
    <http:client DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
  </http:conduit>
</beans>
```

メッセージの処理方法

分離モードで HTTP トランスポートを使用すると、HTTP メッセージの処理にさらに複雑な層が追加されます。複雑さはアプリケーションの実装レベルのコードに対して透過的ですが、デバッグの理由で何が起こるかを理解しておくことが重要となります。

図12.1「デコードされた HTTP トランスポートのメッセージフロー」 は、分離モードで HTTP を使用する場合のメッセージのフローを示しています。

図12.1 デコードされた HTTP トランスポートのメッセージフロー



リクエストは以下のプロセスを開始します。

1. コンシューマーの実装は操作を呼び出し、リクエストメッセージが生成されます。
2. WS-Addressing レイヤーは WS-A ヘッダーをメッセージに追加します。
   分離されたエンドポイントがコンシューマーの設定に指定されると、切り離されたエンドポイントのアドレスは WS-A ReplyTo ヘッダーに配置されます。
3. メッセージはサービスプロバイダーに送信されます。
4. サービスプロバイダーはメッセージを受信します。

5. コンシューマーからの要求メッセージは、プロバイダーの WS-A レイヤーにディスパッチされます。

6. WS-A ReplyTo ヘッダーは anonymous に設定されていないため、プロバイダーは HTTP ステータスコードが 202 に設定されたメッセージを返信し、リクエストが受信されたことを承認します。

7. HTTP レイヤーは、元の接続のバックチャネルを使用して 202 Accepted メッセージをコンシューマーに送信します。

8. コンシューマーは、元のメッセージの送信に使用する HTTP 接続のバックチャネルで 202 Accepted 応答を受け取ります。
   コンシューマーが 202 Accepted 応答を受け取ると、HTTP 接続を閉じます。

9. 要求は、要求が処理されるサービスプロバイダーの実装に渡されます。

10. 応答の準備ができると、WS-A 層にディスパッチされます。

11. WS-A レイヤーは、WS-Addressing ヘッダーを応答メッセージに追加します。

12. HTTP トランスポートは、コンシューマーの分離されたエンドポイントに応答を送信します。

13. コンシューマーの分離されたエンドポイントは、サービスプロバイダーからの応答を受信します。

14. レスポンスはコンシューマーの WS-A レイヤーにディスパッチされ、WS-A RelatesTo ヘッダーを使用して適切なリクエストに相関します。

15. 相関応答がクライアント実装に返され、呼び出し呼び出しのブロックが解除されます。

第13章 SOAP OVER JMS の使用

概要

Apache CXF は、W3C 標準の SOAP/JMS トランスポートを実装しています。この標準は、SOAP/HTTP サービスのより堅牢な代替手段を提供することを目的としています。このトランスポートを使用する Apache CXF アプリケーションは、SOAP/JMS 標準も実装するアプリケーションと相互運用できる必要があります。トランスポートは、エンドポイントの WSDL で直接設定されます。

注: JMS 1.0.2 API のサポートは、CXF 3.0 で削除されました。Red Hat JBoss Fuse 6.2 以降 (CXF 3.0 を含む) を使用している場合、JMS プロバイダーは JMS 1.1 API をサポートする必要があります。

13.1. BASIC CONFIGURATION

概要

SOAP over JMS プロトコル は、World Wide Web Consortium (W3C) によって、ほとんどのサービスで使用される通常の SOAP/HTTP プロトコルにさらに信頼性の高いトランスポート層を提供する方法として定義されています。Apache CXF の実装は仕様に完全に準拠しており、準拠しているすべてのフレームワークと互換性がある必要があります。

このトランスポートは、JNDI を使用して JMS 宛先を検索します。操作が呼び出されると、要求は SOAP メッセージとしてパッケージ化され、JMS メッセージの本文で指定された宛先に送信されます。

SOAP/JMS トランスポートを使用するには:

1. トランスポートタイプが SOAP/JMS であることを指定します。
2. JMS URI を使用してターゲット宛先を指定します。
3. 必要に応じて、JNDI 接続を設定します。
4. オプションで、JMS 設定を追加します。

JMS トランスポートタイプの指定

WSDL バインディングを指定するときに JMS トランスポートを使用するように SOAP バインディングを設定します。**soap:binding** 要素の **トランスポート** 属性を **http://www.w3.org/2010/soapjms/** に設定します。例13.1「SOAP over JMS バインディング仕様」 は、SOAP/JMS を使用する WSDL バインディングを示しています。

例13.1 SOAP over JMS バインディング仕様

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

ターゲット宛先の指定

エンドポイントの WSDL ポートを指定するときに、JMS ターゲット宛先のアドレスを指定します。SOAP/JMS エンドポイントのアドレス仕様は、SOAP/HTTP エンドポイントと同じ **soap:address** 要素および属性を使用します。相違点はアドレス指定です。JMS エンドポイントは、URI Scheme for JMS 1.0 で定義されている JMS URI を使用します。 例13.2「JMS URI 構文」 は、JMS URI の構文を示しています。

例13.2 JMS URI 構文

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

表13.1「JMS URI バリアント」 JMS URI で利用可能なバリアントを説明します。

表13.1 JMS URI バリアント

バリアント	説明
jndi	宛先名が JNDI キュー名であることを指定します。このバリアントを使用する場合は、JNDI プロバイダーにアクセスするための設定を提供する必要があります。
jndi-topic	宛先名が JNDI トピック名であることを指定します。このバリアントを使用する場合は、JNDI プロバイダーにアクセスするための設定を提供する必要があります。
queue	宛先が JMS を使用して解決されたキュー名であることを指定します。提供される文字列は **Session.createQueue()** に渡され、宛先の表現を作成します。
topic	宛先が JMS を使用して解決されたトピック名であることを指定します。提供される文字列は **Session.createTopic()** に渡され、宛先の表現を作成します。

JMS URI の **オプション** 部分は、トランスポートを設定するために使用され、 「JMS URI」 で説明されています。

例13.3「SOAP/JMS エンドポイントアドレス」 は、JNDI を使用してターゲットの宛先が検索される SOAP/JMS エンドポイントの WSDL ポートエントリーを示しています。

例13.3 SOAP/JMS エンドポイントアドレス

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

JNDI および JMS トランスポートの設定

SOAP/JMS は、JNDI 接続と JMS トランスポートを設定するためのいくつかの方法を提供します。

- 「JMS URI」
- 「WSDL 拡張機能」

13.2. JMS URI

概要

SOAP/JMS を使用する場合、JMS URI を使用してエンドポイントのターゲット宛先を指定します。URI に 1 つ以上のオプションを追加すると、JMS 接続の設定に JMS URI を使用することもできます。これらのオプションは、IETF 標準、URI Scheme for Java Message Service 1.0 で説明されています。これらを使用して、JNDI システム、応答先、使用する配信モード、およびその他の JMS プロパティーを設定できます。

構文

例13.4「JMS URI オプションの構文」 にあるように、JMS URI の最後にオプションを 1 つ以上追加するには、宛先のアドレスに疑問符 (**?**) で区切ります。複数のオプションはアンパサンド (**&**) で区切られます。例13.4「JMS URI オプションの構文」 は、JMS URI で複数のオプションを使用するための構文を示しています。

例13.4 JMS URI オプションの構文

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

JMS プロパティー

表13.2「URI オプションとして設定された JMS プロパティー」 は、JMS トランスポート層に影響する URI オプションを示しています。

表13.2 URI オプションとして設定された JMS プロパティー

プロパティー	デフォルト	説明
conduitIdSelectorPrefix		[オプション] コンジットが作成するすべての相関 ID の前に付けられる文字列値。セレクターはこれを使用して応答をリッスンできます。

プロパティー	デフォルト	説明
deliveryMode	**PERSISTENT**	JMS **PERSISTENT** または **NON_PERSISTENT** メッセージセマンティクスを使用するかどうかを指定します。**PERSISTENT** 配信モードの場合、JMS ブローカーはメッセージを承認する前に永続ストレージに保存します。一方、**NON_PERSISTENT** メッセージはメモリーにのみ保持されます。
durableSubscriptionClientID		[オプション] オプション接続のクライアント識別子を指定します。このプロパティーは、プロバイダーがクライアントに代わって維持する状態に接続を関連付けるために使用されます。これにより、同じアイデンティティーを持つ後続のサブスクライバーが残りの状態でサブスクリプションを再開できます。
durableSubscriptionName		(必要に応じて) サブスクリプションの名前を指定します。
messageType	**byte**	CXF によって使用される JMS メッセージタイプを指定します。有効な値は以下のとおりです。 ● byte ● text ● binary
password		[オプション] 接続を作成するためのパスワードを指定します。URI にこのプロパティーを追加することは推奨されません。
priority	**4**	0(最低) から 9(最高) の範囲の JMS メッセージ優先度を指定します。
receiveTimout	**60000**	要求/応答交換が使用されるときにクライアントが応答を待機する時間をミリ秒単位で指定します。

プロパティー	デフォルト	説明
reconnectOnException	**true**	[CXF3.0 での非推奨] 例外が発生したときにトランスポートを再接続するかどうかを指定します。 3.0 の時点では、例外の発生時に常にトランスポートが再接続されます。
replyToName		[任意] キューメッセージの応答先を指定します。応答宛先が **JMSReplyTo** ヘッダーに表示されます。このプロパティーを設定することは、要求/応答セマンティクスを持つアプリケーションに推奨されます。これは、JMS プロバイダーが一時応答キューが指定されていない場合に割り当てるためです。 このプロパティーの値は、JMS URI で指定されたバリアントに従って解釈されます。 ● **jndi** バリアント - JNDI によって解決された宛先キューの名前 ● **queue** バリアント - JMS を使用して解決された宛先キューの名前
sessionTransacted	**false**	トランザクションタイプを指定します。有効な値は以下のとおりです。 ● **true** - リソース出力カルトランザクション ● **false** - JTA トランザクション
timeToLive	**0**	JMS プロバイダーがメッセージを破棄するまでの時間をミリ秒単位で指定します。**0** の値は無限に存在することを示します。

プロパティー	デフォルト	説明
topicReplyToName		[オプション] トピックメッセージの返信先を指定します。このプロパティーの値は、JMS URI で指定されたバリアントに従って解釈されます。 ● **jndi-topic** - JNDI によって解決された宛先トピックの名前 ● **topic** - JMS によって解決された宛先トピックの名前
useConduitIdSelector	**true**	コンジットの UUID をすべての相関 ID の接頭辞として使用するかどうかを指定します。 すべての Conduits に一意の UUID が割り当てられているため、このプロパティーを **true** に設定すると、複数のエンドポイントが JMS キューまたはトピックを共有できます。
username		[オプション] 接続の作成に使用するユーザー名を指定します。

JNDI プロパティー

表13.3「URI オプションとして設定可能な JNDI プロパティー」 は、このエンドポイントの JNDI を設定するために使用できる URI オプションを示しています。

表13.3 URI オプションとして設定可能な JNDI プロパティー

プロパティー	説明
jndiConnectionFactoryName	JMS 接続ファクトリーの JNDI 名を指定します。
jndiInitialContextFactory	JNDI プロバイダーの完全修飾クラス名を指定します (**javax.jms.InitialContextFactory** タイプである必要があります)。**java.naming.factory.initial** Java システムプロパティーの設定と同等です。
jndiTransactionManagerName	Spring、Blueprint、または JNDI で検索される JTA トランザクションマネージャーの名前を指定します。トランザクションマネージャーが見つかると、JTA トランザクションが有効になります。**sessionTransacted** JMS プロパティーを参照してください。

プロパティー	説明
jndiURL	JNDI プロバイダーを初期化する URL を指定します。**java.naming.provider.url** Java システムプロパティーの設定と同等です。

追加の JNDI プロパティー

プロパティー **java.naming.factory.initial** および **java.naming.provider.url** は、JNDI プロバイダーを初期化するために必要な標準のプロパティーです。ただし、JNDI プロバイダーが標準のプロパティーに加えてカスタムプロパティーをサポートする場合もあります。この場合、**jndi-PropertyName** 形式の URI オプションを設定することで、任意の JNDI プロパティーを設定できます。

たとえば、JNDI の SUN の LDAP 実装を使用している場合は、例13.5「JMS URI での JNDI プロパティーの設定」 のように JMS URI で JNDI プロパティー **java.naming.factory.control** を設定できます。

例13.5 JMS URI での JNDI プロパティーの設定

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

例

JMS プロバイダーが **まだ** 設定されていない場合は、オプションを使用して URI で必要な JNDI 設定の詳細を提供できます (表13.3「URI オプションとして設定可能な JNDI プロパティー」を参照)。たとえば、Apache ActiveMQ JMS プロバイダーを使用し、**test.cxf.jmstransport.queue** というキューに接続するようにエンドポイントを設定するには、例13.6「JNDI 接続を設定する JMS URI」 に示される URI を使用します。

例13.6 JNDI 接続を設定する JMS URI

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

サービスの公開

JAX-WS 標準 **publish()** メソッドを使用して SOAP/JMS サービスを公開することはできません。代わりに、例13.7「SOAP/JMS サービスの公開」 のように Apache CXF の **JaxWsServerFactoryBean** クラスを使用する必要があります。

例13.7 SOAP/JMS サービスの公開

```
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「SOAP/JMS サービスの公開」 のコードは、以下を行います。

エンドポイントのアドレスを表す JMS URI を作成します。

JaxWsServerFactoryBean をインスタンス化してサービスを公開します。

ファクトリー Bean の **address** フィールドをサービスの JMS URI で設定します。

ファクトリーによって作成されたサービスが SOAP/JMS トランスポートを使用するように指定します。

サービスの消費

標準の JAX-WS API を使用して SOAP/JMS サービスを使用することはできません。その代わりに、例13.8「SOAP/JMS サービスの使用」 のように Apache CXF の **JaxWsProxyFactoryBean** クラスを使用する必要があります。

例13.8 SOAP/JMS サービスの使用

```
// 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「SOAP/JMS サービスの使用」 のコードは、以下を行います。

エンドポイントのアドレスを表す JMS URI を作成します。

JaxWsProxyFactoryBean をインスタンス化してプロキシーを作成します。

ファクトリー Bean の **address** フィールドをサービスの JMS URI で設定します。

ファクトリーによって作成されたプロキシーが SOAP/JMS トランスポートを使用することを指定します。

13.3. WSDL 拡張機能

概要

バインディングスコープ、サービススコープ、またはポートスコープのいずれかで WSDL 拡張要素をコントラクトに挿入することにより、JMS トランスポートの基本設定を指定できます。WSDL 拡張機能を使用すると、JNDI **InitialContext** ブートストラップのプロパティーを指定でき、JMS 宛先の検索に使用できます。JMS トランスポート層の動作に影響を与えるプロパティーの一部を設定することもできます。

SOAP/JMS namespace

SOAP/JMS WSDL エクステンションは、**http://www.w3.org/2010/soapjms/** namespace で定義されます。WSDL コントラクトでそれらを使用するには、次の設定を **wsdl:definitions** 要素に追加します。

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

WSDL 拡張要素

表13.4「SOAP/JMS WSDL extension 要素」 は、JMS トランスポートの設定に使用できる WSDL 拡張要素すべてを示しています。

表13.4 SOAP/JMS WSDL extension 要素

要素	デフォルト	説明
soapjms:jndiInitialContextFactory		JNDI プロバイダーの完全修飾 Java クラス名を指定します。**java.naming.factory.initial** Java システムプロパティーの設定と同等です。
soapjms:jndiURL		JNDI プロバイダーを初期化する URL を指定します。**java.naming.provider.url** Java システムプロパティーの設定と同等です。
soapjms:jndiContextParameter		JNDI **InitialContext** を作成する追加のプロパティーを指定します。**name** および **value** 属性を使用してプロパティーを指定します。
soapjms:jndiConnectionFactoryName		JMS 接続ファクトリーの JNDI 名を指定します。

要素	デフォルト	説明
soapjms:deliveryMode	**PERSISTENT**	JMS **PERSISTENT** または **NON_PERSISTENT** メッセージセマンティクスを使用するかどうかを指定します。**PERSISTENT** 配信モードの場合、JMS ブローカーはメッセージを承認する前に永続ストレージに保存します。一方、**NON_PERSISTENT** メッセージはメモリーにのみ保持されます。
soapjms:replyToName		[任意] キューメッセージの応答先を指定します。応答宛先が **JMSReplyTo** ヘッダーに表示されます。このプロパティーを設定することは、要求/応答セマンティクスを持つアプリケーションに推奨されます。これは、JMS プロバイダーが一時応答キューが指定されていない場合に割り当てるためです。 このプロパティーの値は、JMS URI で指定されたバリアントに従って解釈されます。 ● **jndi** バリアント - JNDI によって解決された宛先キューの名前 ● **queue** バリアント - JMS を使用して解決された宛先キューの名前
soapjms:priority	**4**	0(最低) から 9(最高) の範囲の JMS メッセージ優先度を指定します。
soapjms:timeToLive	**0**	ミリ秒単位の時間。その後、JMS プロバイダーはメッセージを破棄します。**0** の値は無限の有効期間を表します。

設定スコープ

WSDL コントラクトに WSDL エレメントを配置すると、コントラクトで定義されたエンドポイントの設定変更のスコープに影響します。SOAP/JMS WSDL 要素は、**wsdl:binding** 要素、**wsdl:service** 要素、または **wsdl:port** 要素のいずれかの子として配置できます。SOAP/JMS 要素の親は、設定が配置されるスコープを次のスコープのどれに決定するかを決定します。

バインディングスコープ

wsdl:binding 要素内にエクステンション要素を配置することで、**バインディングスコープ** で JMS トランスポートを設定できます。このスコープの要素は、このバインディングを使用するすべてのエンドポイントのデフォルト設定を定義します。バインディングスコープの設定は、サービススコープまたはポートスコープで上書きできます。

サービス範囲

wsdl:service 要素内に extension 要素を配置することで、 **service scope**で JMS トランスポートを設定できます。このスコープの要素は、このサービスのすべてのエンドポイントのデフォルト設定を定義します。サービススコープの設定は、ポートスコープで上書きできます。

ポートのスコープ

wsdl:port 要素内に extension 要素を配置することで、 **ポートスコープで** JMS トランスポートを設定できます。ポートスコープの要素は、このポートの設定を定義します。これらは、サービススコープまたはバインディングスコープで定義された同じ拡張要素のデフォルトをオーバーライドします。

例

例13.9「SOAP/JMS 設定との WSDL コントラクト」 は、SOAP/JMS サービスの WSDL コントラクトを示しています。バインディングスコープで JNDI レイヤーを設定し、サービススコープでメッセージ配信の詳細を設定し、ポートスコープで応答先を設定します。

例13.9 SOAP/JMS 設定との WSDL コントラクト

```
<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「SOAP/JMS 設定との WSDL コントラクト」 の WSDL は以下を行います。

SOAP/JMS エクステンションの namespace を宣言します。

バインディングスコープで JNDI 接続を設定します。

JMS 配信スタイルを非永続に設定し、各メッセージを 1 分間存続させます。

ターゲットの宛先を指定します。

応答メッセージが **greeterReply.queue** キューで配信されるように JMS トランスポートを設定します。

第14章 汎用 JMS の使用

概要

Apache CXF は、JMS トランスポートの汎用実装を提供します。一般的な JMS トランスポートは、SOAP メッセージの使用に制限されておらず、JMS を使用する任意のアプリケーションに接続できます。

注: JMS 1.0.2 API のサポートは、CXF 3.0 で削除されました。Red Hat JBoss Fuse 6.2 以降 (CXF 3.0 を含む) を使用している場合、JMS プロバイダーは JMS 1.1 API をサポートする必要があります。

14.1. JMS を設定するためのアプローチ

Apache CXF 汎用 JMS トランスポートは JMS プロバイダーに接続し、JMS メッセージを **TextMessage** または **ByteMessage** のいずれかのボディーで交換するアプリケーションと動作します。

JMS トランスポートを有効にして設定するには、次の 2 つの方法があります。

- 「JMS 設定 Bean の使用」
- 「WSDL を使用した JMS の設定」

14.2. JMS 設定 BEAN の使用

概要

JMS 設定を簡素化し、より強力にするために、Apache CXF は単一の JMS 設定 Bean を使用して JMS エンドポイントを設定します。Bean は **org.apache.cxf.transport.jms.JMSConfiguration** クラスによって実装されます。エンドポイントを直接設定したり、JMS コンジットと宛先を設定するために使用できます。

設定名前空間

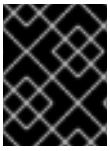
JMS 設定 Bean は、Spring p-namespace を使用して、設定を可能な限り単純にします。この名前空間を使用するには、例14.1「Spring p-namespace の宣言」 のように設定のルート要素で宣言する必要があります。

例14.1 Spring p-namespace の宣言

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

設定の指定

クラス **org.apache.cxf.transport.jms.JMSConfiguration** の Bean を定義して JMS 設定を指定します。Bean のプロパティーによってトランスポートの設定が提供されます。



重要

CXF 3.0 では、JMS トランスポートは Spring JMS に依存しなくなったため、SpringJMS 関連のオプションの一部が削除されました。

表14.1「一般的な JMS 設定プロパティー」 プロバイダーとコンシューマーの両方に共通するプロパティーをリスト表示します。

表14.1 一般的な JMS 設定プロパティー

プロパティー	デフォルト	説明
connectionFactory		[必須] JMSConnectionFactory を定義する Bean への参照を指定します。
wrapInSingleConnectionFactory	**true** [pre v3.0]	**CXF3.0 で削除** **pre CXF 3.0** では、ConnectionFactory を Spring **SingleConnectionFactory** でラップするかどうかを指定します。 JMS トランスポートのパフォーマンスを向上するため、接続をプールしない ConnectionFactory を使用する場合は、このプロパティーを有効にします。これは、JMS トランスポートが各メッセージの新しい接続を作成し、**SingleConnectionFactory** が接続のキャッシュに必要なため、再利用できるためです。

プロパティー	デフォルト	説明
reconnectOnException	**false**	**CXF 3.0 で非推奨** CXF は、例外が発生したときに常に再接続します。 **以前の CXF3.0** 例外が発生したときに新しい接続を作成するかどうかを指定します。 ConnectionFactory を Spring **SingleConnectionFactory** でラップする場合: ● **true** - 例外で、新しいコネクションを作成します。PooledConnectionFactory を使用する場合は、このオプションを有効にしないでください。このオプションは、プールされた接続のみを返し、再接続は返しません。 ● **false** - 例外で、再接続を試みないでください。
targetDestination		宛先の JNDI 名またはプロバイダー固有の名前を指定します。
replyDestination		応答の送信先となる JMS 宛先の JMS 名を指定します。このプロパティーでは、応答にユーザー定義の宛先を使用できます。詳細は「名前付き応答宛先の使用」 を参照してください。
destinationResolver	DynamicDestinationResolver	Spring **DestinationResolver** への参照を指定します。 このプロパティーにより、宛先名を JMS 宛先に解決する方法を定義できます。有効な値は以下のとおりです。 ● **DynamicDestinationResolver** - JMS プロバイダーの機能を使用して宛先名を解決します。 ● **JndiDestinationResolver** - JNDI を使用して宛先名を解決します。

プロパティー	デフォルト	説明
transactionManager		Spring トランザクションマネージャーへの参照を指定します。これにより、サービスは JTA トランザクションに参加できます。
taskExecutor	**SimpleAsyncTaskExecutor**	**CXF3.0 で削除** **CXF3.0 より前** の SpringTaskExecutor への参照を指定します。これは、受信メッセージの処理方法を決定するためにリスナーで使用されます。
useJms11	**false**	**CXF3.0 で削除** CXF3.0 は JMS1.1 機能のみをサポートします。 **CXF 3.0 より前** JMS 1.1 機能が使用されるかどうかを指定します。有効な値は以下のとおりです。 ● **true** - JMS 1.1 機能 ● **false** - JMS 1.0.2 機能
messageIdEnabled	**true**	**CXF3.0 で削除** **CXF 3.0** より前は、JMS トランスポートが JMS ブローカーにメッセージ ID を提供するかどうかを指定します。有効な値は以下のとおりです。 ● **true** - ブローカーはメッセージ ID を提供する必要があります。 ● **false** - ブローカーはメッセージ ID を提供する必要がありません。この場合、エンドポイントはメッセージプロデューサーの **setDisableMessageID()** メソッドを **true** の値で呼び出します。その後、ブローカーはメッセージ ID を生成したりエンドポイントのメッセージに追加したりする必要がないヒントを提供します。ブローカーはヒントを受け入れるか、無視します。

プロパティー	デフォルト	説明
messageTimestampEnabled	**true**	**CXF3.0 で削除** **CXF 3.0** JMS トランスポートが JMS ブローカーにメッセージタイムスタンプを提供するかどうかを指定します。有効な値は以下のとおりです。 ● **true** - ブローカーはメッセージのタイムスタンプを提供する必要があります。 ● **false** - ブローカーはメッセージのタイムスタンプを提供する必要はありません。 この場合、エンドポイントはメッセージプロデューサーの **setDisableMessageTimestamp()** メソッドを **true** の値で呼び出します。その後、ブローカーはタイムスタンプを生成したり、エンドポイントのメッセージに追加したりする必要がないヒントを提供します。ブローカーはヒントを受け入れるか、無視します。
cacheLevel	**-1** (機能の無効化)	**CXF3.0 で削除** **CXF 3.0 より前** JMS リスナーコンテナーが適用されるキャッシングのレベルを指定します。有効な値は以下のとおりです。 ● **0** - CACHE_NONE ● **1** - CACHE_CONNECTION ● **2** - CACHE_SESSION ● **3** - CACHE_CONSUMER ● **4** – CACHE_AUTO 詳細は、ClassDefaultMessageListenerContainer を参照してください。

プロパティー	デフォルト	説明
pubSubNoLocal	**false**	トピックの使用時に独自のメッセージを受信するかどうかを指定します。 ● **true** - 独自のメッセージを受け取りません。 ● **false** - 独自のメッセージを受け取ります。
receiveTimeout	**60000**	応答メッセージを待機する時間 (ミリ秒単位) を指定します。
explicitQosEnabled	**false**	QoS 設定 (優先度、永続性、ライブ時間など) が各メッセージに対して明示的に設定するか (**true**)、デフォルト値 (**false**) を使用するかどうかを指定します。
deliveryMode	**2**	メッセージが永続化されるかどうかを指定します。有効な値は以下のとおりです。 ● **1** (NON_PERSISTENT) - メッセージはメモリーのみを保持します。 ● **2** (PERSISTENT) - メッセージははディスクに永続化されます。
priority	**4**	メッセージの優先度を指定します。JMS 優先度の値の範囲は、**0** (最低) から **9** (最高) までです。詳細は、JMS プロバイダーのドキュメントを参照してください。
timeToLive	**0** (無期限)	送信されたメッセージが破棄されるまでの時間をミリ秒単位で指定します。
sessionTransacted	**false**	JMS トランザクションが使用されるかどうかを指定します。

プロパティー	デフォルト	説明
concurrentConsumers	**1**	**CXF3.0 で削除** **CXF 3.0 より前** リスナーの同時コンシューマーの最小数を指定します。
maxConcurrentConsumers	**1**	**CXF3.0 で削除** **CXF 3.0 より前** リスナーの同時コンシューマーの最大数を指定します。
messageSelector		受信メッセージのフィルターに使用するセレクターの文字列値を指定します。このプロパティーにより、複数の接続でキューを共有できます。メッセージセレクターを指定するために使用される構文の詳細については、JMS1.1 仕様 を参照してください。
subscriptionDurable	**false**	サーバーが永続サブスクリプションを使用するかどうかを指定します。
durableSubscriptionName		永続サブスクリプションの登録に使用される名前 (文字列) を指定します。
messageType	**text**	メッセージデータを JMS メッセージとしてパッケージ化する方法を指定します。有効な値は以下のとおりです。 ● **text** - データが **TextMessage** としてパッケージ化されることを指定します。 ● **byte** AsyncResult-AsyncResultspecifies は、データがバイトの配列 (**byte[]**) としてパッケージ化されるように指定します。 ● **binary** AsyncResult-AsyncResultspecifies は、データが **ByteMessage** としてパッケージ化されることを示します。

プロパティー	デフォルト	説明
pubSubDomain	**false**	ターゲットの宛先がトピックであるかキューであるかを指定します。有効な値は以下のとおりです。 ● **true** - トピック ● **false** - キュー
jmsProviderTibcoEms	**false**	JMS プロバイダーが TibcoEMS であるかどうかを指定します。 **true** に設定すると、セキュリティーコンテキストのプリンシパルが **JMS_TIBCO_SENDER** ヘッダーから入力されます。
useMessageIDAsCorrelationID	**false**	**CXF3.0 で削除** JMS がメッセージ ID を使用してメッセージを関連付けるかどうかを指定します。 **true** に設定すると、クライアントは生成された相関 ID を設定します。
maxSuspendedContinuations	**-1** (機能の無効化)	CXF 3.0 JMS 宛先が持つ可能性のある一時停止継続の最大数を指定します。現在の数が指定された最大数を超えると、JMSListenerContainer は停止します。
reconnectPercentOfMax	**70**	CXF 3.0 は、**maxSuspendedContinuations** を超過するために JMSListenerContainer を再起動するタイミングを指定します。 リスナーコンテナーは、現在の一時停止された継続回数が **(maxSuspendedContinuations * reconnectPercentOfMax/100)** の値を下回ると再起動されます。

例14.2「JMS 設定 Bean」 で示されているように、Bean のプロパティーは **bean** 要素の属性として指定されます。これらはすべて Spring **p** namespace で宣言されます。

例14.2 JMS 設定 Bean

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

設定をエンドポイントに適用する

JMSConfiguration Bean は、Apache CXF 機能メカニズムを使用してサーバーとクライアントエンドポイントの両方に直接適用できます。これを行うには、以下を行います。

1. エンドポイントの **address** 属性を **jms://** に設定します。
2. **jaxws:feature** 要素をエンドポイントの設定に追加します。
3. **org.apache.cxf.transport.jms.JMSConfigFeature** タイプの Bean を機能に追加します。
4. **bean** 要素の **p:jmsConfig-ref** 属性を **JMSConfiguration** Bean の ID に設定します。

例14.3「JAX-WS クライアントに JMS 設定を追加」 は、例14.2「JMS 設定 Bean」 からの JMS 設定を使用する JAX-WS クライアントを示しています。

例14.3 JAX-WS クライアントに JMS 設定を追加

```
<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>
```

設定のトランスポートへの適用

JMSConfiguration Bean は、**jmsConfig-ref** 要素を使用して JMS コンジェクトおよび JMS 宛先に適用できます。**jms:jmsConfig-ref** 要素の値は **JMSConfiguration** Bean の ID です。

例14.4「JMS コンジットへの JMS 設定の追加」 は、例14.2「JMS 設定 Bean」 の JMS 設定を使用する JMS 輻輳を示しています。

例14.4 JMS コンジットへの JMS 設定の追加

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

```
  <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

14.3. クライアント側 JMS パフォーマンスの最適化

概要

2 つの主要な設定がクライアントの JMS パフォーマンスに影響します。プーリングと同期受信です。

プーリング

クライアント側で、CXF はメッセージごとに新しい JMS セッションおよび JMS プロデューサーを作成します。これは、session および producer オブジェクトがスレッドセーフであるためです。プロデューサーの作成には、サーバーと通信する必要があるため、特に時間がかかります。

接続ファクトリーのプールは、接続、セッション、およびプロデューサーをキャッシュすることでパフォーマンスを向上します。

ActiveMQ の場合は、プーリングの設定が簡単です。以下に例を示します。

```
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);
```

プーリングの詳細は、Red Hat JBoss Fuse トランザクションガイド の付録 AJMS シングルおよびマルチリソーストランザクションのパフォーマンスの最適化を参照してください。

同期受信の回避

要求/応答交換の場合、JMS トランスポートは要求を送信してから応答を待ちます。可能な場合は、JMS **MessageListener** を使用してリクエスト/リプライメッセージングが非同期に実装されます。

ただし、エンドポイント間でキューを共有する必要がある場合は、CXF は同期 **Consumer.receive()** メソッドを使用する必要があります。このシナリオでは、**MessageListener** がメッセージセレクターを使用してメッセージをフィルターする必要があります。メッセージセレクターは事前に認識される必要があるため、**MessageListener** は一度だけ開かれます。

メッセージセレクターを事前に知ることができない 2 つのケースは避ける必要があります。

- **JMSMessageID** が **JMSCorrelationID** として使用される場合
  JMS プロパティーが **useConduitIdSelector** および **conduitSelectorPrefix** が JMS トランスポートに設定されていない場合、クライアントは **JMSCorrelationId** を設定しません。これにより、サーバーは **JMSCorrelationId** としてリクエストメッセージの **JMSMessageId** を使用し

ます。**JMSMessageID** は事前に認識できないため、クライアントは同期 **Consumer.receive()** メソッドを使用する必要があります。

IBM JMS エンドポイントで **Consumer.receive()** メソッドを使用する必要があることに注意してください (デフォルト)。

- ユーザーはリクエストメッセージに **JMStype** を設定し、カスタム **JMSCorrelationID** を設定します。
ここでも、カスタムの **JMSCorrelationID** は事前に認識できないため、クライアントは同期 **Consumer.receive()** メソッドを使用する必要があります。

したがって、一般的なルールは、同期受信の使用を必要とする設定の使用を避けることです。

14.4. JMS トランザクションの設定

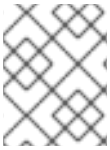
概要

CXF 3.0 は、一方向メッセージングを使用する場合、CXF エンドポイントでローカル JMS トランザクションと JTA トランザクションの両方をサポートします。

ローカルトランザクション

ローカルリソースを使用するトランザクションは、例外が発生した場合にのみ JMS メッセージをロールバックします。データベーストランザクションなどの他のリソースを直接調整することはありません。

ローカルトランザクションを設定するには、通常通りにエンドポイントを設定し、**sessionTrasnsacted** プロパティーを **true** に設定します。



注記

トランザクションとプーリングの詳細は、Red Hat JBoss Fuse トランザクションガイドを参照してください。

JTA トランザクション

JTA トランザクションを使用すると、任意の数の XA リソースを調整できます。CXF エンドポイントが JTA トランザクション用に設定されている場合、サービス実装を呼び出す前にトランザクションを開始します。例外が発生しなければ、トランザクションはコミットされます。それ以外の場合は、ロールバックされます。

JTA トランザクションでは、JMS メッセージが消費され、データがデータベースに書き込まれます。例外が発生すると、両方のリソースがロールバックされるため、メッセージが消費されてデータがデータベースに書き込まれるか、メッセージがロールバックされてデータがデータベースに書き込まれません。

JTA トランザクションの設定には、次の 2 つの手順が必要です。

1. トランザクションマネージャーの定義
    - Bean メソッド
        - トランザクションマネージャーの定義

```
<bean id="transactionManager"

class="org.apache.geronimo.transaction.manager.GeronimoTransactionManager"/>
```

- JMS URI でトランザクションマネージャーの名前を設定します

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

この例では、ID **TransactionManager** の Bean を検索します。

- OSGi 参照方法
  - ブループリントを使用して、トランザクションマネージャーを OSGi サービスとして検索します

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

  - JMS URI でトランザクションマネージャーの名前を設定します

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

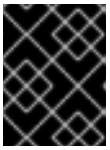
この例では、JNDI でトランザクションマネージャーを検索します。

2. JCA プール接続ファクトリーの設定
Spring を使用して JCA プールされた接続ファクトリーを定義します。

```
<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>
```

この例では、最初の Bean は **JcaPooledConnectionFactory** に指定される ActiveMQ XA 接続ファクトリーを定義します。**JcaPooledConnectionFactory** は id **ConnectionFactory** のデフォルトの Bean として提供されます。

JcaPooledConnectionFactory は通常の ConnectionFactory のようになることに注意してください。ただし、新しい接続とセッションが開かれると、XA トランザクションがチェックされ、見つかった場合は、JMS セッションが XA リソースとして自動的に登録されます。これにより、JMS セッションが JMS トランザクションに参加できるようになります。



重要

JMS トランスポートに XA ConnectionFactory を直接設定することはできません。

14.5. WSDL を使用した JMS の設定

14.5.1. JMS WSDL Extension Namespance

JMS エンドポイントを定義するための WSDL 拡張機能は、名前空間 http://cxf.apache.org/transports/jms で定義されています。JMS 拡張機能を使用するには、例14.5「JMS WSDL 拡張名前空間」 に示す行をコントラクトの定義要素に追加する必要があります。

例14.5 JMS WSDL 拡張名前空間

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

14.5.2. 基本の JMS 設定

概要

JMS アドレス情報は、**jms:address** 要素とその子 (jms**:JMSNamingProperties** 要素) を使用して提供されます。**jms:address** 要素の属性は、JMS ブローカーおよび宛先を識別するのに必要な情報を指定します。**jms:JMSNamingProperties** 要素は、JNDI サービスへの接続に使用される Java プロパティーを指定します。

重要

JMS 機能を使用して指定された情報は、エンドポイントの WSDL ファイルの情報を上書きします。

JMS アドレスの指定

JMS エンドポイントの基本設定は、**jms:address** 要素をサービスの **port** 要素の子として使用して行われます。WSDL で使用される **jms:address** 要素は、設定ファイルで使用されるものと同じです。その属性は、表14.2「JMS エンドポイント属性」 にリスト表示されます。

表14.2 JMS エンドポイント属性

属性	説明
destinationStyle	JMS 宛先が JMS キューまたは JMS トピックであるかどうかを指定します。
jndiConnectionFactoryName	JMS 宛先に接続するときに使用する JMS 接続ファクトリーにバインドされた JNDI 名を指定します。
jmsDestinationName	要求の送信先の JMS 宛先の JMS 名を指定します。
jmsReplyDestinationName	応答が送信される JMS 宛先の JMS 名を指定します。この属性を使用すると、ユーザー定義の宛先を返信に使用できます。詳細は 「名前付き応答宛先の使用」 を参照してください。
jndiDestinationName	要求の送信先の JMS 宛先にバインドされた JNDI 名を指定します。

属性	説明
jndiReplyDestinationName	応答が送信される JMS 宛先にバインドされた JNDI 名を指定します。この属性を使用すると、ユーザー定義の宛先を返信に使用できます。詳細は「名前付き応答宛先の使用」を参照してください。
connectionUserName	JMS ブローカーに接続するときに使用するユーザー名を指定します。
connectionPassword	JMS ブローカーに接続するときに使用するパスワードを指定します。

jms:address WSDL 要素は、**jms:JMSNamingProperties** 子要素を使用して、JNDI プロバイダーへの接続に必要な追加情報を指定します。

JNDI プロパティーの指定

JMS および JNDI プロバイダーとの相互運用性を高めるために、**jms:address** 要素には子要素 **jms:JMSNamingProperties** があり、JNDI プロバイダーへの接続時に使用されるプロパティーの設定に使用される値を指定できます。**jms:JMSNamingProperties** 要素には、**name** と **value** の 2 つの属性があります。**name** は設定するプロパティーの名前を指定します。**value** 属性は、指定されたプロパティーの値を指定します。**JMS:JMSNamingProperties** 要素は、プロバイダー固有のプロパティーの仕様にも使用できます。

以下は、設定可能な一般的な JNDI プロパティーのリストです。

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**

これらの属性で使用する情報の詳細については、JNDI プロバイダーのドキュメントを確認し、JavaAPI リファレンス資料を参照してください。

例

例14.6「JMS WSDL ポート仕様」 は、JMS WSDL **port** 仕様の例を示しています。

例14.6 JMS WSDL ポート仕様

```
<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 クライアント設定

概要

JMS コンシューマーエンドポイントは、使用するメッセージタイプを指定します。JMS コンシューマーエンドポイントは、JMS **ByteMessage** または JMS **TextMessage** を使用できます。

ByteMessage を使用する場合、コンシューマーエンドポイントは **byte[]** を、JMS メッセージボディーからデータを保存し、取得する方法として使用します。メッセージが送信されると、フォーマット情報を含むメッセージデータは **byte[]** にパッケージ化され、ネットワークに置かれる前にメッセージボディーに配置されます。メッセージが受信されると、コンシューマーエンドポイントは、メッセージボディーに格納されているデータを **byte[]** にパックしたかのようにアンマーシャルしようとします。

TextMessage を使用する場合、コンシューマーエンドポイントはメッセージボディーからデータを保存および取得するためのメソッドとして文字列を使用します。メッセージが送信されると、フォーマット固有の情報を含むメッセージ情報が文字列に変換され、JMS メッセージ本文に配置されます。メッセージを受け取ると、コンシューマーエンドポイントは、JMS メッセージボディーに格納されているデータを文字列にパックしたかのようにアンマーシャルしようとします。

ネイティブ JMS アプリケーションが Apache CXF コンシューマーと対話する場合、JMS アプリケーションはメッセージとフォーマット情報の解釈を担当します。たとえば、Apache CXF コントラクトで JMS エンドポイントに使用されるバインディングが SOAP であると指定され、メッセージが **TextMessage** としてパッケージ化されている場合、受信側の JMS アプリケーションは、すべての SOAP エンベロープ情報を含むテキストメッセージを取得します。

メッセージタイプの指定

JMS コンシューマーエンドポイントによって許可されるメッセージの型は、オプションの **jms:client** 要素を使用して設定されます。**jms:client** 要素は WSDL **port** 要素の子であり、属性が 1 つあります。

表14.3 JMS クライアントの WSDL 拡張

messageType
メッセージデータを JMS メッセージとしてパッケージ化する方法を指定します。**text** は、データを **TextMessage** としてパッケージ化することを指定します。**binary** は、データを **ByteMessage** としてパッケージ化することを指定します。

例

例14.7「JMS コンシューマーエンドポイントの WSDL」 は、JMS コンシューマーエンドポイントを設定するための WSDL を示しています。

例14.7 JMS コンシューマーエンドポイントの 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 プロバイダーの設定

概要

JMS プロバイダーエンドポイントには、設定可能な多くの動作があります。これには以下が含まれます。

- メッセージの相関方法
- 永続サブスクリプションの使用
- サービスがローカル JMS トランザクションを使用する場合
- エンドポイントによって使用されるメッセージセレクター

設定の指定

プロバイダーエンドポイントの動作は、オプションの **jms:server** 要素を使用して設定されます。**jms:server** 要素は WSDL **wsdl:port** 要素の子で、以下の属性があります。

表14.4 JMS プロバイダーエンドポイント WSDL 拡張

属性	説明
useMessageIDAsCorrealationID	JMS がメッセージ ID を使用してメッセージを関連付けるかどうかを指定します。デフォルトは **false** です。
durableSubscriberName	永続サブスクリプションの登録に使用される名前を指定します。
messageSelector	使用するメッセージセレクターの文字列値を指定します。メッセージセレクターを指定するために使用される構文の詳細については、JMS1.1 仕様 を参照してください。
transactional	ローカル JMS ブローカーがメッセージ処理に関するトランザクションを作成するかどうかを指定します。デフォルトは **false** です。[a]

[a] 現在、**transactional** 属性を **true** に設定するとランタイムはサポートされません。

例

例14.8「JMS プロバイダーエンドポイントの WSDL」 は、JMS プロバイダーエンドポイントを設定するための WSDL を示しています。

例14.8 JMS プロバイダーエンドポイントの 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. 名前付き応答宛先の使用

概要

デフォルトでは、JMS を使用する Apache CXF エンドポイントは、応答を送受信するための一時キューを作成します。名前付きキューを使用する場合は、エンドポイントの JMS 設定の一部として返信を送信するために使用されるキューを設定できます。

応答先名の設定

エンドポイントの JMS 設定の **jmsReplyDestinationName** 属性または **jndiReplyDestinationName** 属性のいずれかを使用して、応答宛先を指定します。クライアントエンドポイントは、指定された宛先で応答をリッスンし、すべての送信要求の **ReplyTo** フィールドに属性の値を指定します。サービスエンドポイントは、リクエストの **jndiReplyDestinationName** フィールドに宛先が指定されていない場合に、応答の送信先として **ReplyTo** 属性の値を使用します。

例

例14.9「名前付き応答キューを使用した JMS コンシューマー仕様」 は、JMS クライアントエンドポイントの設定を示しています。

例14.9 名前付き応答キューを使用した JMS コンシューマー仕様

```
<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 APACHE ACTIVEMQ との統合

概要

Apache ActiveMQ を JMS プロバイダーとして使用する場合、宛先の JNDI 名を、キューまたはトピックの JNDI バインディングを動的に作成する特別な形式で指定できます。つまり、キューまたはトピックの JNDI バインディングを使用して事前に JMS プロバイダーを設定する必要は**ありません**。

THE INITIAL CONTEXT FACTORY

Apache ActiveMQ と JNDI を統合するキーは、**ActiveMQInitialContextFactory** クラスです。このクラスは、JNDI **InitialContext** インスタンスを作成するために使用され、JMS ブローカーの JMS 宛先にアクセスするために使用できます。

例15.1「Apache ActiveMQ に接続するための SOAP/JMS WSDL」 は、Apache ActiveMQ と統合された JNDI **InitialContext** を作成する SOAP/JMS WSDL 拡張を示しています。

例15.1 Apache ActiveMQ に接続するための SOAP/JMS WSDL

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

例15.1「Apache ActiveMQ に接続するための SOAP/JMS WSDL」 では、Apache ActiveMQ クライアントは **tcp://localhost:61616** にあるブローカーポートに接続します。

接続ファクトリーの検索

JNDI **InitialContext** インスタンスの作成に加え、**javax.jms.ConnectionFactory** インスタンスにバインドされる JNDI 名を指定する必要があります。Apache ActiveMQ の場合、InitialContext **インスタンスには** 事前定義されたバインディングがあり、JNDI 名 **ConnectionFactory** を **ActiveMQConnectionFactory** インスタンスにマッピングします。例15.2「Apache ActiveMQ 接続ファクトリーを指定する SOAP/JMS WSDL」 ApacheActiveMQ 接続ファクトリーを指定するための SOAP/JMS 拡張要素を示します。

例15.2 Apache ActiveMQ 接続ファクトリーを指定する SOAP/JMS WSDL

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

動的宛先の構文

キューまたはトピックに動的にアクセスするには、宛先の JNDI 名を次のいずれかの形式の JNDI 複合名として指定します。

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

QueueName および **TopicName** は Apache ActiveMQ ブローカーが使用する名前です。JNDI 名は抽象化され **ません**。

例15.3「動的作成されたキューを含む WSDL ポート仕様」 は、動的に作成されたキューを使用する WSDL ポートを示しています。

例15.3 動的作成されたキューを含む WSDL ポート仕様

```
<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>
```

アプリケーションが JMS 接続を開くと、Apache ActiveMQ は JNDI 名 **greeter.request.queue** を持つキューが存在するかどうかを確認します。存在しない場合は、新しいキューが作成され、JNDI name **greeter.request.queue** にバインドします。

第16章 コンジット

概要

コンジットは、アウトバウンド接続を実装するために使用されるトランスポートアーキテクチャーの低レベルの部分です。それらの動作とライフサイクルは、システムのパフォーマンスと処理負荷に影響を与える可能性があります。

概要

コンジットは、Apache CXF ランタイムでクライアント側またはアウトバウンドのトランスポートの詳細を管理します。これらは、ポートのオープン、アウトバウンド接続の確立、メッセージの送信、およびアプリケーションと単一の外部エンドポイント間の応答のリッスンを担当します。アプリケーションが複数のエンドポイントに接続する場合、エンドポイントごとに 1 つのコンジットインスタンスがあります。

各トランスポートタイプは、コンジットインターフェイスを使用して独自のコンジットを実装します。これにより、アプリケーションレベルの機能とトランスポート間の標準化されたインターフェイスが可能になります。

一般に、クライアント側のトランスポートの詳細を設定するときに、アプリケーションで使用されているコンジットについてのみ心配する必要があります。ランタイムがコンジットを処理する方法の基本的なセマンティクスは、一般に、開発者が心配する必要のあるものではありません。

ただし、コンジットを理解することが役立つ場合があります。

- カスタムトランスポートの実装
- 限られたリソースを管理するための高度なアプリケーションチューニング

コンジットのライフサイクル

コンジットは、クライアント実装オブジェクトによって管理されます。作成されると、コンジットはクライアント実装オブジェクトの期間中存続します。コンジットのライフサイクルは次のとおりです。

1. クライアント実装オブジェクトが作成されると、**ConduitSelector** オブジェクトへの参照が付与されます。
2. クライアントがメッセージを送信する必要がある場合は、コンジットセレクターからのコンジットへのリクエストの参照です。
   メッセージが新しいエンドポイントに対するものである場合、コンジットセレクターは新しいコンジットを作成し、それをクライアント実装に渡します。それ以外の場合は、ターゲットエンドポイントのコンジットへの参照をクライアントに渡します。
3. コンジットは、必要に応じてメッセージを送信します。
4. クライアント実装オブジェクトが破棄されると、それに関連付けられているすべてのコンジットが破棄されます。

コンジットの重み

コンジットオブジェクトの重みは、トランスポートの実装によって異なります。HTTP コンジットは非常に軽量です。JMS コンジットは、JMS **Session** オブジェクトと 1 つ以上の **JMSListenerContainer** オブジェクトに関連付けられるため重くなります。

パート IV. WEB サービスエンドポイントの設定

このガイドでは、Red Hat Fuse で Apache CXF エンドポイントを作成する方法について説明します。

第17章 JAX-WS エンドポイントの設定

概要

JAX-WS エンドポイントは、3 つの Spring 設定要素のいずれかを使用して設定されます。正しい要素は、設定しているエンドポイントのタイプと使用する機能によって異なります。コンシューマーの場合は、**jaxws:client** 要素を使用します。サービスプロバイダーの場合は、**jaxws:endpoint** 要素または **jaxws:server** 要素のいずれかを使用できます。

エンドポイントの定義に使用される情報は、通常、エンドポイントのコントラクトで定義されます。設定要素を使用して、コントラクトの情報を上書きすることができます。設定要素を使用して、コントラクトで提供されていない情報を提供することもできます。

WS-RM などの高度な機能をアクティブにするには、設定要素を使用する必要があります。これは、エンドポイントの設定要素に子要素を提供することによって行われます。Java ファーストのアプローチを使用して開発されたエンドポイントを処理する場合、エンドポイントのコントラクトとして機能する SEI には、使用するバインディングとトランスポートのタイプに関する情報が不足している可能性があることに注意してください。

17.1. サービスプロバイダーの設定

17.1.1. サービスプロバイダー設定の要素

Apache CXF には、サービスプロバイダーの設定に使用できる 2 つの要素があります。

- 「jaxws:endpoint 要素の使用」
- 「jaxws:server 要素の使用」

2 つの要素の違いは、主にランタイムの内部にあります。**jaxws:endpoint** 要素は、サービスエンドポイントをサポートするために作成された **org.apache.cxf.jaxws.EndpointImpl** オブジェクトにプロパティーを挿入します。**jaxws:server** 要素は、エンドポイントをサポートするために作成された **org.apache.cxf.jaxws.support.JaxWsServerFactoryBean** オブジェクトにプロパティーをインジェクトします。**EndpointImpl** オブジェクトは、設定データを **JaxWsServerFactoryBean** オブジェクトに渡します。**JaxWsServerFactoryBean** オブジェクトは、実際のサービスオブジェクトを作成するために使用されます。どちらの設定要素もサービスエンドポイントを設定するため、好みの構文に基づいて選択できます。

17.1.2. jaxws:endpoint 要素の使用

概要

jaxws:endpoint 要素は、JAX-WS サービスプロバイダーを設定するためのデフォルトの要素です。その属性と子は、サービスプロバイダーをインスタンス化するために必要なすべての情報を指定します。属性の多くは、サービスのコントラクトの情報にマップされます。子は、インターセプターやその他の高度な機能を設定するために使用されます。

設定されているエンドポイントの特定

ランタイムが設定を適切なサービスプロバイダーに適用するには、ランタイムがそれを識別できる必要があります。サービスプロバイダーを識別するための基本的な手段は、エンドポイントを実装するクラスを指定することです。これは、**jaxws:endpoint** 要素の **implementor** 属性を使用して行われます。

異なるエンドポイントが共通の実装を共有する場合は、エンドポイントごとに異なる設定を提供することができます。設定で特定のエンドポイントを区別するには、次の 2 つの方法があります。

- **serviceName** 属性と **endpointName** 属性との組み合わせ
  serviceName 属性は、サービスのエンドポイントを定義する **wsdl:service** 要素を指定します。**endpointName** 属性は、サービスのエンドポイントを定義する特定の **wsdl:port** 要素を指定します。どちらの属性も、**ns:name** の形式で QNames として指定されます。**ns** は要素の namespace で、**name** は要素の **name** 属性の値です。

  注記

  wsdl:service 要素に **wsdl:port** 要素が 1 つしかない場合は、**endpointName** 属性を省略できます。

- **name** 属性
  name 属性は、サービスのエンドポイントを定義する特定の **wsdl:port** 要素の QName を指定します。QName は、**{ns}localPart** の形式で提供されます。**ns** は **wsdl:port** 要素の namespace で、**localPart** は **wsdl:port** 要素の **name** 属性の値です。

属性

jaxws:endpoint 要素の属性は、エンドポイントの基本プロパティーを設定します。これらのプロパティーには、エンドポイントのアドレス、エンドポイントを実装するクラス、およびエンドポイントをホストする **bus** が含まれます。

表17.1「jaxws:endpoint 要素を使用して JAX-WS サービスプロバイダーを設定するための属性」は、**jaxws:endpoint** 要素の属性について説明しています。

表17.1 jaxws:endpoint 要素を使用して JAX-WS サービスプロバイダーを設定するための属性

属性	説明
id	他の設定要素がエンドポイントを参照するのに使用できる一意の識別子を指定します。
implementor	サービスを実装するクラスを指定します。実装クラスを設定する Spring Bean へのクラス名または ID 参照のいずれかを使用して、実装クラスを指定できます。このクラスはクラスパス上にある必要があります。
implementorClass	サービスを実装するクラスを指定します。この属性は、**implementor** 属性に指定された値が Spring AOP を使用してラップされる Bean への参照である場合に便利です。
address	HTTP エンドポイントのアドレスを指定します。この値は、サービスコントラクトで指定された値を上書きします。
wsdlLocation	エンドポイントの WSDL コントラクトの場所を指定します。WSDL コントラクトの場所は、サービスのデプロイ元のフォルダーを基準にしています。

属性	説明
endpointName	サービスの **wsdl:port** 要素の **name** 属性値を指定します。これは、**ns:name** 形式を使用して QName として指定されます。ここで、ns は **wsdl:port** 要素の namespace です。
serviceName	サービスの **wsdl:service** 要素の **name** 属性値を指定します。これは、**ns:name** 形式を使用して QName として指定されます。ここで、**ns** は **wsdl:service** 要素の namespace になります。
publish	サービスを自動的に公開するかどうかを指定します。これを **false** に設定すると、開発者は 31章 *サービスの公開* に記載されているようにエンドポイントを明示的にパブリッシュする必要があります。
bus	サービスエンドポイントの管理に使用されるバスを設定する Spring Bean の ID を指定します。これは、共通の機能セットを使用するように複数のエンドポイントを設定する場合に役立ちます。
bindingUri	サービスが使用するメッセージバインディングの ID を指定します。有効なバインディング ID のリストは、23章*Apache CXF バインディング ID* に提供されています。
name	サービスの **wsdl:port** 要素の文字列化した QName を指定します。これは、**{ns}localPart** 形式を使用して QName として指定されます。**ns** は **wsdl:port** 要素の namespace で、**localPart** は **wsdl:port** 要素の **name** 属性の値です。
abstract	Bean が抽象 Bean であるかどうかを指定します。抽象 Bean は、具体的な Bean 定義の親として機能し、インスタンス化されません。デフォルトは **false** です。これを **true** に設定すると、Bean ファクトリーが Bean をインスタンス化しないように指示します。
depends-on	インスタンス化する前にエンドポイントがインスタンス化に依存する Bean のリストを指定します。

属性	説明
createdFromAPI	ユーザーが、**Endpoint.publish()** または **Service.getPort()** などの Apache CXF API を使用してその Bean を作成したことを指定します。 デフォルトは **false** です。 これを **true** に設定すると以下を行います。 ● **.jaxws-endpoint** を ID に追加して Bean の内部名を変更する ● Bean を抽象化します
publishedEndpointUrl	生成された WSDL の **address** 要素に配置される URL。この値が指定されない場合、**address** 属性の値が使用されます。この属性は、パブリック URL がサービスがデプロイされている URL と同じでない場合に役立ちます。

表17.1「jaxws:endpoint 要素を使用して JAX-WS サービスプロバイダーを設定するための属性」 にリスト表示される属性の他に、複数の **xmlns:shortName** 属性を使用して **endpointName** および **serviceName** 属性によって使用される namespace を宣言する必要がある場合があります。

例

例17.1「シンプルな JAX-WS エンドポイント設定」 は、エンドポイントが公開されるアドレスを指定する JAX-WS エンドポイントの設定を示しています。この例では、他のすべての値にデフォルトを使用するか、実装でアノテーションに値が指定されていることを前提としています。

例17.1 シンプルな JAX-WS エンドポイント設定

```
<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「サービス名を使用した JAX-WS エンドポイント設定」 は、コントラクトに 2 つのサービス定義が含まれている JAX-WS エンドポイントの設定を示しています。この場合、**serviceName** 属性を使用してインスタンス化するサービス定義を指定する必要があります。

例17.2 サービス名を使用した JAX-WS エンドポイント設定

```
<beans ...
```

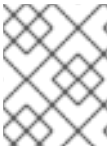
```
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">

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

</beans>
```

xmlns:samp 属性は、WSDL **service** 要素が定義される namespace を指定します。

例17.3「HTTP/2 が有効になっている JAX-WS エンドポイント設定」 は、HTTP/2 が有効になっているアドレスを指定する JAX-WS エンドポイントの設定を示しています。

Apache CXF 用の HTTP/2 の設定

Apache Karaf でスタンドアロン Apache CXF Undertow トランスポート (**http-undertow**) を使用する場合、HTTP/2 がサポートされます。HTTP/2 プロトコルを有効にするには、**jaxws:endpoint** 要素の **address** 属性を絶対 URL として設定し、 **org.apache.cxf.transports.http_undertow.EnableHttp2** プロパティーを **true** に設定する必要があります。



注記

この HTTP/2 実装は、プレーン HTTP または HTTPS を使用したサーバー側の HTTP/2 トランスポートのみをサポートします。

例17.3 HTTP/2 が有効になっている JAX-WS エンドポイント設定

```
<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>
```

注記

パフォーマンスを向上させるために、Red Hat は Apache Karaf でサーブレットトランスポート (**pax-web-undertow**) を使用することを推奨します。これにより、Web コンテナーの集中設定およびチューニングが可能になりますが、**pax-web-undertow** は HTTP/2 トランスポートプロトコルをサポートしません。

17.1.3. jaxws:server 要素の使用

概要

jaxws:server 要素は JAX-WS サービスプロバイダーを設定する要素です。**org.apache.cxf.jaxws.support.JaxWsServerFactoryBean** に設定情報を注入します。これは Apache CXF 固有のオブジェクトです。純粋な Spring アプローチを使用してサービスを構築している場合、サービスと対話するために Apache CXF 固有の API を使用する必要はありません。

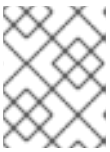
jaxws:server 要素の属性および子は、サービスプロバイダーをインスタンス化するために必要なすべての情報を指定します。属性は、エンドポイントをインスタンス化するために必要な情報を指定します。子は、インターセプターやその他の高度な機能を設定するために使用されます。

設定されているエンドポイントの特定

ランタイムが設定を適切なサービスプロバイダーに適用するには、ランタイムがそれを識別できる必要があります。サービスプロバイダーを識別するための基本的な手段は、エンドポイントを実装するクラスを指定することです。これは、**jaxws:server** 要素の **serviceBean** 属性を使用して行います。

異なるエンドポイントが共通の実装を共有する場合は、エンドポイントごとに異なる設定を提供することができます。設定で特定のエンドポイントを区別するには、次の 2 つの方法があります。

- **serviceName** 属性と **endpointName** 属性との組み合わせ
  serviceName 属性は、サービスのエンドポイントを定義する **wsdl:service** 要素を指定します。**endpointName** 属性は、サービスのエンドポイントを定義する特定の **wsdl:port** 要素を指定します。どちらの属性も、**ns:name** の形式で QNames として指定されます。**ns** は要素の namespace で、**name** は要素の **name** 属性の値です。

  注記

  wsdl:service 要素に **wsdl:port** 要素が 1 つしかない場合は、**endpointName** 属性を省略できます。

- **name** 属性
  name 属性は、サービスのエンドポイントを定義する特定の **wsdl:port** 要素の QName を指定します。QName は、**{ns}localPart** の形式で提供されます。**ns** は **wsdl:port** 要素の namespace で、**localPart** は **wsdl:port** 要素の **name** 属性の値です。

属性

jaxws:server 要素の属性は、エンドポイントの基本プロパティーを設定します。これらのプロパティーには、エンドポイントのアドレス、エンドポイントを実装するクラス、およびエンドポイントをホストする **bus** が含まれます。

表17.2「jaxws:server 要素を使用して JAX-WS サービスプロバイダーを設定するための属性」は、**jaxws:server** 要素の属性を説明します。

表17.2 jaxws:server 要素を使用して JAX-WS サービスプロバイダーを設定するための属性

属性	説明
id	他の設定要素がエンドポイントを参照するのに使用できる一意の識別子を指定します。
serviceBean	サービスを実装するクラスを指定します。実装クラスを設定する Spring Bean へのクラス名または ID 参照のいずれかを使用して、実装クラスを指定できます。このクラスはクラスパス上にある必要があります。
serviceClass	サービスを実装するクラスを指定します。この属性は、**implementor** 属性に指定された値が Spring AOP を使用してラップされる Bean への参照である場合に便利です。
address	HTTP エンドポイントのアドレスを指定します。この値は、サービスコントラクトで指定された値を上書きします。
wsdlLocation	エンドポイントの WSDL コントラクトの場所を指定します。WSDL コントラクトの場所は、サービスのデプロイ元のフォルダーを基準にしています。
endpointName	サービスの **wsdl:port** 要素の **name** 属性値を指定します。これは、**ns:name** 形式を使用して QName として指定されます。ここで、ns は **wsdl:port** 要素の namespace になります。
serviceName	サービスの **wsdl:service** 要素の **name** 属性値を指定します。これは、**ns:name** 形式を使用して QName として指定されます。ここで、ns は **wsdl:service** 要素の namespace になります。
publish	サービスを自動的に公開するかどうかを指定します。これを **false** に設定すると、開発者は 31章 *サービスの公開* に記載されているようにエンドポイントを明示的にパブリッシュする必要があります。
bus	サービスエンドポイントの管理に使用されるバスを設定する Spring Bean の ID を指定します。これは、共通の機能セットを使用するように複数のエンドポイントを設定する場合に役立ちます。
bindingId	サービスが使用するメッセージバインディングの ID を指定します。有効なバインディング ID のリストは、23章*Apache CXF バインディング ID* に提供されています。

属性	説明
name	サービスの **wsdl:port** 要素の文字列化した QName を指定します。これは、**{ns}localPart** 形式を使用して QName として指定されます。ここで、**ns** は **wsdl:port** 要素の namespace であり、**localPart** は **wsdl:port** 要素の **name** 属性の値になります。
abstract	Bean が抽象 Bean であるかどうかを指定します。抽象 Bean は、具体的な Bean 定義の親として機能し、インスタンス化されません。デフォルトは **false** です。これを **true** に設定すると、Bean ファクトリーが Bean をインスタンス化しないように指示します。
depends-on	エンドポイントをインスタンス化する前に、エンドポイントがインスタンス化に依存する Bean のリストを指定します。
createdFromAPI	ユーザーが、**Endpoint.publish()** または **Service.getPort()** などの Apache CXF API を使用してその Bean を作成したことを指定します。 デフォルトは **false** です。 これを **true** に設定すると以下を行います。 ● **.jaxws-endpoint** を ID に追加して Bean の内部名を変更する ● Bean を抽象化します

表17.2「jaxws:server 要素を使用して JAX-WS サービスプロバイダーを設定するための属性」 にリスト表示される属性の他に、複数の **xmlns:shortName** 属性を使用して **endpointName** および **serviceName** 属性によって使用される namespace を宣言する必要がある場合があります。

例

例17.4「シンプルな JAX-WS サーバー設定」 は、エンドポイントが公開されるアドレスを指定する JAX-WS エンドポイントの設定を示しています。

例17.4 シンプルな JAX-WS サーバー設定

```
<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. サービスプロバイダーへの機能の追加

概要

jaxws:endpoint および **jaxws:server** 要素は、サービスプロバイダーをインスタンス化するために必要な基本的な設定情報を提供します。サービスプロバイダーに機能を追加したり、高度な設定を実行したりするには、設定に子要素を追加する必要があります。

子要素を使用すると、次のことができます。

- 19章*Apache CXF ロギング*
- 59章*インターセプターを使用するためのエンドポイントの設定*
- 20章*WS-Addressing のデプロイ*
- 21章*信頼性の高いメッセージングの有効化*
- 「JAX-WS エンドポイントでスキーマ検証を有効にする」

要素

表17.3「JAX-WS サービスプロバイダーの設定に使用される要素」 では、**jaxws:endpoint** がサポートする子要素が説明されています。

表17.3 JAX-WS サービスプロバイダーの設定に使用される要素

要素	説明
jaxws:handlers	メッセージを処理するための JAX-WS ハンドラー実装のリストを指定します。JAX-WS ハンドラーの実装の詳細は、43章*ハンドラーの作成* を参照してください。
jaxws:inInterceptors	インバウンド要求を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxws:inFaultInterceptors	インバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxws:outInterceptors	アウトバウンド応答を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。

要素	説明
jaxws:outFaultInterceptors	アウトバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パート VII「Apache CXF インターセプターの開発」 を参照してください。
jaxws:binding	エンドポイントが使用するメッセージバインディングを設定する Bean を指定します。メッセージバインディング は、**org.apache.cxf.binding.BindingFactory** インターフェイスの実装を使用して設定されます。[a]
jaxws:dataBinding [b]	エンドポイントで使用されるデータバインディングを実装するクラスを指定します。これは、埋め込み Bean 定義を使用して指定されます。
jaxws:executor	サービスに使用される Java エグゼキュータを指定します。これは、埋め込み Bean 定義を使用して指定されます。
jaxws:features	Apache CXF の高度な機能を設定する Bean のリストを指定します。Bean 参照のリストまたは埋め込み Bean のリストのいずれかを提供できます。
jaxws:invoker	サービスが使用する org.apache.cxf.service.Invoker インターフェイスの実装を指定します。 [c]
jaxws:properties	エンドポイントに渡されるプロパティーの Spring マップを指定します。これらのプロパティーを使用して、MTOM サポートの有効化などの機能を制御できます。
jaxws:serviceFactory	サービスのインスタンス化に使用される **JaxWsServiceFactoryBean** オブジェクトを設定する Bean を指定します。

[a] SOAP バインディングは **soap:soapBinding** Bean を使用して設定されます。

[b] **jaxws:endpoint** 要素は **jaxws:dataBinding** 要素をサポートしません。

[c] Invoker 実装は、サービスの呼び出し方法を制御します。たとえば、各リクエストをサービス実装の新しいインスタンスで処理するかどうか、呼び出し間で状態を保持するかどうかを制御します。

17.1.5. JAX-WS エンドポイントでスキーマ検証を有効にする

概要

schema-validation-enabled プロパティーを設定して、**jaxws:endpoint** 要素または **jaxws: server** 要素でスキーマ検証を有効にできます。スキーマ検証が有効になっている場合、クライアントとサーバー

間で送信されるメッセージは、スキーマに準拠しているかどうかがチェックされます。スキーマ検証はパフォーマンスに大きな影響を与えるため、デフォルトではオフになっています。

例

JAX-WS エンドポイントでスキーマ検証を有効にするには、**jaxws:endpoint** 要素または **jaxws:server** 要素の **jaxws:properties** 子要素の **schema-validation-enabled** プロパティーを設定します。たとえば、**jaxws:endpoint** 要素でスキーマ検証を有効にするには、以下を実行します。

```
<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>
```

schema-validation-enabled プロパティーの許可される値のリストは、「スキーマ検証タイプの値」を参照してください。

17.2. コンシューマーエンドポイントの設定

概要

JAX-WS コンシューマーエンドポイントは、**jaxws:client** 要素を使用して設定されます。要素の属性は、コンシューマーを作成するために必要な基本情報を提供します。

WS-RM などのその他の機能をコンシューマーに追加するには、子を **jaxws:client** 要素に追加します。子要素は、エンドポイントのログ動作を設定したり、エンドポイントの実装に他のプロパティーを挿入したりするためにも使用されます。

基本的な設定プロパティー

表17.4「JAX-WS コンシューマーの設定に使用する属性」 で説明されている属性は、JAX-WS コンシューマーの設定に必要な基本情報を提供します。設定する特定のプロパティーの値のみを指定する必要があります。ほとんどのプロパティーには適切なデフォルトがあるか、エンドポイントのコントラクトによって提供される情報に依存しています。

表17.4 JAX-WS コンシューマーの設定に使用する属性

属性	説明
address	コンシューマーが要求を行うエンドポイントの HTTP アドレスを指定します。この値は、コントラクトで設定された値を上書きします。
bindingId	コンシューマーが使用するメッセージバインディングの ID を指定します。有効なバインディング ID のリストは、*23章Apache CXF バインディング ID* に提供されています。

属性	説明
bus	エンドポイントを管理するバスを設定する Spring Bean の ID を指定します。
endpointName	コンシューマーが要求するサービスの **wsdl:port** 要素の **name** 属性の値を指定します。これは、**ns:name** 形式を使用して QName として指定されます。ここで、ns は **wsdl:port** 要素の namespace になります。
serviceName	コンシューマーが要求するサービスの **wsdl:service** 要素の **name** 属性の値を指定します。これは、**ns:name** 形式を使用して QName として指定されます。ここで、ns は **wsdl:service** 要素の namespace になります。
username	単純なユーザー名/パスワード認証に使用されるユーザー名を指定します。
password	単純なユーザー名/パスワード認証に使用されるパスワードを指定します。
serviceClass	サービスエンドポイントインターフェイス (SEI) の名前を指定します。
wsdlLocation	エンドポイントの WSDL コントラクトの場所を指定します。WSDL コントラクトの場所は、クライアントのデプロイ元のフォルダーを基準にしています。
name	コンシューマーが要求するサービスの **wsdl:port** 要素の文字列化された QName を指定します。これは、**{ns}localPart** 形式を使用して QName として指定されます。ここで、ns は **wsdl:port** 要素の namespace であり、**localPart** は **wsdl:port** 要素の **name** 属性の値になります。
abstract	Bean が抽象 Bean であるかどうかを指定します。抽象 Bean は、具体的な Bean 定義の親として機能し、インスタンス化されません。デフォルトは **false** です。これを **true** に設定すると、Bean ファクトリーが Bean をインスタンス化しないように指示します。
depends-on	インスタンス化する前にエンドポイントがインスタンス化に依存する Bean のリストを指定します。

属性	説明
createdFromAPI	**Service.getPort()** などの Apache CXF API を使用して Bean を作成したユーザーを指定します。 デフォルトは **false** です。 これを **true** に設定すると以下を行います。 ● Bean の内部名を変更するには、**.jaxws-client** を id に追加します。 ● Bean を抽象化します

表17.4「JAX-WS コンシューマーの設定に使用する属性」 にリスト表示される属性の他に、複数の **xmlns:shortName** 属性を使用して **endpointName** および **serviceName** 属性によって使用される namespace を宣言する必要がある場合があります。

機能の追加

コンシューマーに機能を追加したり、高度な設定を行うには、設定に子要素を追加する必要があります。

子要素を使用すると、次のことができます。

- 19章*Apache CXF ロギング*
- 59章*インターセプターを使用するためのエンドポイントの設定*
- 20章*WS-Addressing のデプロイ*
- 21章*信頼性の高いメッセージングの有効化*
- 「JAX-WS コンシューマーでのスキーマ検証の有効化」

表17.5「コンシューマーエンドポイント設定の要素」 JAX-WS コンシューマーの設定に使用できる子要素の説明を示します。

表17.5 コンシューマーエンドポイント設定の要素

要素	説明
jaxws:binding	エンドポイントが使用するメッセージバインディングを設定する Bean を指定します。メッセージバインディング は、**org.apache.cxf.binding.BindingFactory** インターフェイスの実装を使用して設定されます。[a]
jaxws:dataBinding	エンドポイントで使用されるデータバインディングを実装するクラスを指定します。これは、埋め込み Bean 定義を使用して指定します。JAXB データバインディングを実装するクラスは **org.apache.cxf.jaxb.JAXBDataBinding** です。

要素	説明
jaxws:features	Apache CXF の高度な機能を設定する Bean のリストを指定します。Bean 参照のリストまたは埋め込み Bean のリストのいずれかを提供できます。
jaxws:handlers	メッセージを処理するための JAX-WS ハンドラー実装のリストを指定します。JAX-WS ハンドラーの実装に関する詳細は、43章*ハンドラーの作成* を参照してください。
jaxws:inInterceptors	インバウンド応答を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxws:inFaultInterceptors	インバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxws:outInterceptors	アウトバウンド要求を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxws:outFaultInterceptors	アウトバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxws:properties	エンドポイントに渡されるプロパティーのマップを指定します。
jaxws:conduitSelector	使用するクライアントの org.apache.cxf.endpoint.ConduitSelector 実装を指定します。ConduitSelector 実装は、送信要求の処理に使用される **Conduit** オブジェクトの選択に使用するデフォルトのプロセスを上書きします。

[a] SOAP バインディングは **soap:soapBinding** Bean を使用して設定されます。

例

例17.5「簡単なコンシューマー設定」 は、簡単なコンシューマー設定を示しています。

例17.5 簡単なコンシューマー設定

```
<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>
```

JAX-WS コンシューマーでのスキーマ検証の有効化

JAX-WS コンシューマーでスキーマ検証を有効にするには、**jaxws:client** 要素の **jaxws:properties** 子要素の **schema-validation-enabled** プロパティーを以下のように設定します。

```
<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>
```

schema-validation-enabled プロパティーの許可される値のリストは、「スキーマ検証タイプの値」を参照してください。

第18章 JAX-RS エンドポイントの設定

概要

この章では、Blueprint XML および Spring XML で JAX-RS サーバーエンドポイントをインスタンス化および設定する方法と、XML で JAX-RS クライアントエンドポイント (クライアントプロキシー Bean) をインスタンス化および設定する方法について説明します。

18.1. JAX-RS サーバーエンドポイントの設定

18.1.1. JAX-RS サーバーエンドポイントの定義

基本のサーバーエンドポイント定義

XML で JAX-RS サーバーエンドポイントを定義するには、少なくとも以下の項目を指定する必要があります。

1. XML でエンドポイントを定義するために使用される **jaxrs:server** 要素。**jaxrs:** namespace 接頭辞は、ブループリントと Spring の**異なる**名前空間にそれぞれマッピングされることに注意してください。

2. **jaxrs:server** 要素の **address** 属性を使用した JAX-RS サービスのベース URL。アドレス URL を指定する方法は 2 つあり、エンドポイントのデプロイ方法に影響を与えることに注意してください。

   - **相対 URL**(例: **/customers**)。この場合、エンドポイントはデフォルトの HTTP コンテナーにデプロイされ、エンドポイントのベース URL は、CXF サーブレットのベース URL と指定された相対 URL を組み合わせることによって暗黙的に取得されます。
     たとえば、JAX-RS エンドポイントを Fuse コンテナーにデプロイする場合、指定の **/customers** URL は URL **http://Hostname:8181/cxf/customers** に解決されます (コンテナーがデフォルトの **8181** ポートを使用していることを前提とします)。

   - **http://0.0.0.0:8200/cxf/customers** のように、**絶対 URL** を指定 (例:) として。この場合、JAX-RS エンドポイント用に新しい HTTP リスナーポートが開かれます (まだ開いていない場合)。たとえば、Fuse のコンテキストでは、JAX-RS エンドポイントをホストするために新しい Undertow コンテナーが暗黙的に作成されます。特別な IP アドレス **0.0.0.0** は、現在のホストに割り当てられたホスト名のいずれかに一致するワイルドカードとして機能します (マルチホームホストマシンで役に立ちます)。

3. JAX-RS サービスの実装を提供する 1 つ以上の JAX-RS ルートリソースクラス。リソースクラスを指定する最も簡単な方法は、**jaxrs:serviceBeans** 要素内にリソースクラスをリスト表示することです。

Blueprint の例

以下の Blueprint XML の例は、(デフォルトの HTTP コンテナーにデプロイするように) 相対アドレス **/customers** を指定し、**service.CustomerService** リソースクラスによって実装される、JAX-RS エンドポイントを定義する方法を示しています。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns: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>
```

Blueprint XML 名前空間

ブループリントで JAX-RS エンドポイントを定義するには、通常、少なくとも次の XML 名前空間が必要です。

接頭辞	Namespace
(デフォルト)	**http://www.osgi.org/xmlns/blueprint/v1.0.0**
cxf	**http://cxf.apache.org/blueprint/core**
jaxrs	**http://cxf.apache.org/blueprint/jaxrs**

Spring の例

以下の Spring XML の例は、相対アドレス **/customers** (デフォルトの HTTP コンテナーにデプロイされるように) を指定し、**service.CustomerService** リソースクラスによって実装される JAX-RS エンドポイントを定義する方法を示しています。

```
<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 名前空間

Spring で JAX-RS エンドポイントを定義するには、通常、少なくとも次の XML 名前空間が必要です。

接頭辞	Namespace
(デフォルト)	http://www.springframework.org/schema/beans
cxf	http://cxf.apache.org/core
jaxrs	http://cxf.apache.org/jaxrs

Spring XML の自動検出

(Spring only) JAX-RS ルートリソースクラスを明示的に指定する代わりに、Spring XML では自動検出を設定できます。これにより、特定の Java パッケージでリソースクラス (**@Path** によりアノテーションが付けられたクラス) が検索され、検出されたすべてのリソースクラスがエンドポイントに自動的に割り当てられます。この場合、**jaxrs:server** 要素に **address** 属性と **basePackages** 属性のみを指定する必要があります。

たとえば、**a.b.c** Java パッケージ下のすべての JAX-RS リソースクラスを使用する JAX-RS エンドポイントを定義するには、以下のように Spring XML でエンドポイントを定義できます。

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

自動検出メカニズムは、指定された Java パッケージで検出した JAX-RS プロバイダークラスも検出してエンドポイントにインストールします。

Spring XML でのライフサイクル管理

(Spring のみ) Spring XML では、**bean** 要素の **scope** 属性を設定することで、Bean のライフサイクルを制御できます。Spring では以下のスコープ値がサポートされます。

singleton

(デフォルト) 単一の Bean インスタンスを作成します。これはどこでも使用され、Spring コンテナーの存続期間全体にわたって持続します。

prototype

Bean が別の Bean に注入されるたびに、または Bean レジストリーで **getBean()** を呼び出して Bean を取得する際に、新しい Bean インスタンスを作成します。

request

(Web 対応コンテナーでのみ使用可能) Bean で呼び出されるすべての要求に対して新しい Bean インスタンスを作成します。

session

(Web 対応コンテナーでのみ使用可能) 単一の HTTP セッションの存続期間中、新しい Bean を作成します。

globalSession

(Web 対応コンテナーでのみ使用可能) ポートレット間で共有される単一の HTTP セッションの存続期間中、新しい Bean を作成します。

Spring スコープの詳細は、Bean スコープ に関する Spring フレームワークのドキュメントを参照してください。

jaxrs:serviceBeans 要素を使用して JAX-RS リソース Bean を指定すると、Spring スコープ **が正しく機能しない** ことに注意してください。この場合、リソース Bean で **scope** 属性を指定すると、**scope** 属性を効果的に無視されます。

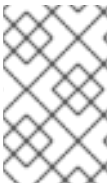
Bean スコープを JAX-RS サーバーエンドポイント内で適切に機能させるには、サービスファクトリーによって提供される間接化のレベルが必要です。Bean スコープを設定する最も簡単な方法は、以下のように **jaxrs:server** 要素で **beanNames** 属性を使用してリソース Bean を指定することです。

```
<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>
```

前述の例では、**customerBean1** と **customerBean2** の 2 つのリソース Bean を設定します。**beanNames** 属性は、リソース Bean ID のスペース区切りリストとして指定されます。

最終的な柔軟性を高めるために、**jaxrs:serviceFactories** 要素を使用して JAX-RS サーバーエンドポイントを設定する際に、サービスファクトリーオブジェクト **を明示的に定義** することができます。このより冗長なアプローチには、デフォルトのサービスファクトリー実装をカスタム実装に置き換えることができるという利点があります。これにより、Bean のライフサイクルを最終的に制御できます。以下の例は、このアプローチを使用して、2 つのリソース Bean (**customerBean1** と **customerBean2**) を設定する方法を示しています。

```
<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>
```



注記

シングルトン以外のライフサイクルを指定する場合は、org.apache.cxf.service.Invoker Bean を実装および登録することが推奨されます (インスタンスは **jaxrs:server/jaxrs:invoker** 要素から参照して登録できます)。

WADL ドキュメントの添付

任意で、**jaxrs:server** 要素の **docLocation** 属性を使用して、WADL ドキュメントを JAX-RS サーバーエンドポイントに関連付けることができます。以下に例を示します。

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

スキーマ検証

外部 XML スキーマがある場合、JAX-B 形式のメッセージコンテンツを記述するため、これらの外部スキーマを **jaxrs:schemaLocations** 要素を介して JAX-RS サーバーエンドポイントに関連付けることができます。

たとえば、サーバーエンドポイントを WADL ドキュメントに関連付けており、着信メッセージのスキーマ検証も有効にする場合は、次のように関連付けられた XML スキーマファイルを指定できます。

```
<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>
```

あるいは、特定のディレクトリーのスキーマファイル ***.xsd** をすべて含める場合は、以下のようにディレクトリー名を指定するだけです。

```
<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>
```

この方法でスキーマを指定すると、一般に、JAX-B スキーマへのアクセスを必要とするあらゆる種類の機能に役立ちます。

データバインディングの指定

jaxrs:dataBinding 要素を使用して、リクエストおよびリプライメッセージのメッセージボディーをエンコードするデータバインディングを指定できます。たとえば、JAX-B データバインディングを指定するには、次のように JAX-RS エンドポイントを設定できます。

```
<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>>
```

または、Aegis データバインディングを指定するには、JAX-RS エンドポイントを次のように設定できます。

```
<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>
```

JMS トランスポートの使用

HTTP の代わりに JMS メッセージングライブラリーをトランスポートプロトコルとして使用するように JAX-RS を設定することができます。JMS 自体はトランスポートプロトコルでは **ない** ため、実際のメッセージングプロトコルは設定する特定の JMS 実装によって異なります。

たとえば、次の Spring XML の例は、JMS トランスポートプロトコルを使用するように JAX-RS サーバーエンドポイントを設定する方法を示しています。

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

前の例について、次の点に注意してください。

- **JMS 実装**: JMS 実装は **ConnectionFactory** Bean によって提供され、Apache ActiveMQ 接続ファクトリーオブジェクトをインスタンス化します。接続ファクトリーをインスタンス化すると、デフォルトの JMS 実装レイヤーとして自動的にインストールされます。
- **JMS コンジットまたはデスティネーションオブジェクト**:Apache CXF は、JMS コンジットオブジェクト (JMS コンシューマーを表すため) または JMS デスティネーションオブジェクト (JMS プロバイダーを表すため) を暗黙的にインスタンス化します。このオブジェクトは、属性設定 **xmlns:s="http://books.com"** (namespace の接頭辞を定義) と **serviceName="s:BookService"** (QName を定義) で定義される QName によって一意に識別される必要があります。
- **Transport ID**: JMS トランポートを選択するには、**, the transportId** 属性を **http://cxf.apache.org/transports/jms** に設定する必要があります。
- **JMS アドレス**: **jaxrs:server/@address** 属性は標準化された構文を使用して、送信する JMS キューまたはトピックを指定します。この構文の詳細は、https://tools.ietf.org/id/draft-merrick-jms-uri-06.txt を参照してください。

拡張マッピングと言語マッピング

JAX-RS サーバーエンドポイントは、ファイル接尾辞 (URL に表示される) を MIME コンテンツタイプヘッダーに自動的にマップし、言語接尾辞を言語タイプヘッダーにマップするように設定できます。たとえば、次の形式の HTTP リクエストについて考えてみます。

```
GET /resource.xml
```

以下のように、**.xml** 接尾辞を自動的にマッピングするように JAX-RS サーバーエンドポイントを設定できます。

```
<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>
```

上記のサーバーエンドポイントが HTTP リクエストを受信すると、型 **application/xml** の新しいコンテンツ型ヘッダーを自動的に作成し、リソース URL から **.xml** 接尾辞を除去します。

言語マッピングについては、次の形式の HTTP リクエストを検討してください。

```
GET /resource.en
```

以下のように、**.en** 接尾辞を自動的にマッピングするように JAX-RS サーバーエンドポイントを設定できます。

```
<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>
```

上記のサーバーエンドポイントが HTTP リクエストを受信すると、値が **en-gb** の新しい受け入れ言語ヘッダーを自動的に作成し、リソース URL から **.en** 接尾辞を除去します。

18.1.2. jaxrs:server 属性

属性

表18.1「JAX-RS サーバーエンドポイントの属性」**jaxrs:server** 要素で利用可能な属性を説明します。

表18.1 JAX-RS サーバーエンドポイントの属性

属性	説明
id	他の設定要素がエンドポイントを参照するのに使用できる一意の識別子を指定します。
address	HTTP エンドポイントのアドレスを指定します。この値は、サービスコントラクトで指定された値を上書きします。
basePackages	**(Spring のみ)** JAX-RS ルートリソースクラスや JAX-RS プロバイダークラスを検出するために検索される Java パッケージのコンマ区切りリストを指定することにより、自動検出を有効にします。
beanNames	JAX-RS ルートリソース Bean の Bean ID のスペース区切りリストを指定します。Spring XML のコンテキストでは、ルートリソース **bean** 要素に **scope** 属性を設定することで、ルートリソース Bean のライフサイクルを定義できます。

属性	説明
bindingId	サービスが使用するメッセージバインディングの ID を指定します。有効なバインディング ID のリストは、23章*Apache CXF バインディング ID* に提供されています。
bus	サービスエンドポイントの管理に使用されるバスを設定する Spring Bean の ID を指定します。これは、共通の機能セットを使用するように複数のエンドポイントを設定する場合に役立ちます。
docLocation	外部の WADL ドキュメントの場所を指定します。
modelRef	モデルスキーマをクラスパスリソースとして指定します (例: **classpath:/path/to/model.xml** 形式の URL)。JAX-RS モデルスキーマを定義する方法の詳細については、「モデルスキーマでの REST サービスの定義」 を参照してください。
publish	サービスを自動的に公開するかどうかを指定します。**false** に設定すると、開発者はエンドポイントを明示的にパブリッシュする必要があります。
publishedEndpointUrl	自動生成される WADL インターフェイスの **wadl:resources/@base** 属性に挿入される URL ベースアドレスを指定します。
serviceAnnotation	**(Spring のみ)** Spring で自動検出するサービスアノテーションクラス名を指定します。**basePackages** プロパティーと組み合わせて使用すると、このオプションは自動検出されたクラスのコレクションが、このアノテーションタイプによってアノテーションが付けられたクラス **だけ** を含めるように制限されます。!これで正しいですか？
serviceClass	JAX-RS サービスを実装する JAX-RS ルートリソースクラスの名前を指定します。この場合、クラスは Blueprint や Spring では **なく** Apache CXF によってインスタンス化されます。Blueprint または Spring でクラスをインスタンス化する場合は、代わりに **jaxrs:serviceBeans** 子要素を使用します。
serviceName	JMS トランスポートが使用される特別なケースの JAX-RS エンドポイントのサービス QName を指定します (**ns:name** の形式を使用)。詳細は、「JMS トランスポートの使用」 を参照してください。
staticSubresourceResolution	**true** の場合、静的なサブリソースの動的解決を無効にします。デフォルトは **false** です。

属性	説明
transportId	(HTTP の代わりに) 非標準のトランスポート層を選択する場合。特に、このプロパティーを **http://cxf.apache.org/transports/jms** に設定すると、JMS トランスポートを選択できます。詳細は、「JMS トランスポートの使用」 を参照してください。
abstract	**(Spring のみ)** Bean が抽象 Bean であるかどうかを指定します。抽象 Bean は、具体的な Bean 定義の親として機能し、インスタンス化されません。デフォルトは **false** です。これを **true** に設定すると、Bean ファクトリーが Bean をインスタンス化しないように指示します。
depends-on	**(Spring のみ)** エンドポイントをインスタンス化する前に、エンドポイントがインスタンス化に依存する Bean のリストを指定します。

18.1.3. jaxrs:server 子要素

子要素

表18.2「JAX-RS サーバーエンドポイントの子要素」 に **jaxrs:server** 要素の子要素をまとめます。

表18.2 JAX-RS サーバーエンドポイントの子要素

要素	説明
jaxrs:executor	サービスに使用される Java **Executor** (スレッドプール実装) を指定します。これは、埋め込み Bean 定義を使用して指定されます。
jaxrs:features	Apache CXF の高度な機能を設定する Bean のリストを指定します。Bean 参照のリストまたは埋め込み Bean のリストのいずれかを提供できます。
jaxrs:binding	**使用されていません。**
jaxrs:dataBinding	エンドポイントで使用されるデータバインディングを実装するクラスを指定します。これは、埋め込み Bean 定義を使用して指定されます。詳細は、「データバインディングの指定」 を参照してください。
jaxrs:inInterceptors	インバウンド要求を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。

要素	説明
jaxrs:inFaultInterceptors	インバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxrs:outInterceptors	アウトバウンド応答を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxrs:outFaultInterceptors	アウトバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxrs:invoker	サービスが使用する org.apache.cxf.service.Invoker インターフェイスの実装を指定します。[a]
jaxrs:serviceFactories	このエンドポイントに関連付けられた JAX-RS ルートリソースのライフサイクルを最大限に制御できます。この要素の子 (**org.apache.cxf.jaxrs.lifecycle.ResourceProvider** 型のインスタンスでなければならない) は、JAX-RS ルートリソースインスタンスを作成するために使用されます。
jaxrs:properties	エンドポイントに渡されるプロパティーの Spring マップを指定します。これらのプロパティーを使用して、MTOM サポートの有効化などの機能を制御できます。
jaxrs:serviceBeans	この要素の子は、JAX-RS ルートリソースのインスタンス (**bean** 要素) または JAX-RS ルートリソースへの参照 (**ref** 要素) です。この場合、**scope** 属性 (**Spring のみ**) が **bean** 要素にある場合、この属性は無視される点に注意してください。
jaxrs:modelBeans	1 つまたは複数の **org.apache.cxf.jaxrs.model.UserResource** Bean への参照のリストで設定されます。この参照はリソースモデルの基本要素です (**jaxrs:resource** 要素に対応)。詳細は、「モデルスキーマでの REST サービスの定義」 を参照してください。

要素	説明
jaxrs:model	このエンドポイントに直接リソースモデルを定義します (つまり、この **jaxrs:model** 要素には 1 つまたは複数の **jaxrs:resource** 要素を含めることができます)。詳細は、「モデルスキーマでの REST サービスの定義」 を参照してください。
jaxrs:providers	1 つ以上のカスタム JAX-RS プロバイダーをこのエンドポイントに登録できます。この要素の子は、JAX-RS プロバイダーのインスタンス (**bean** 要素) または JAX-RS ルートリソースへの参照 (**ref** 要素) です。
jaxrs:extensionMappings	REST 呼び出しの URL がファイル拡張子で終わる場合、この要素を使用して、特定のコンテンツタイプに自動的に関連付けることができます。たとえば、**.xml** ファイル拡張子は **application/xml** コンテンツ型に関連付けることができます。詳細は、「拡張マッピングと言語マッピング」 を参照してください。
jaxrs:languageMappings	REST 呼び出しの URL が言語接尾辞で終わる場合、この要素を使用してこれを特定の言語にマップできます。たとえば、**.en** 言語接尾辞は **en-GB** 言語に関連付けることができます。詳細は、「拡張マッピングと言語マッピング」 を参照してください。
jaxrs:schemaLocations	XML メッセージの内容の検証に使用される 1 つ以上の XML スキーマを指定します。この要素には、1 つまたは複数の **jaxrs:schemaLocation** 要素を含めることができます。各要素は、XML スキーマファイルの場所を指定します (通常は **classpath** URL として)。詳細は、「スキーマ検証」 を参照してください。
jaxrs:resourceComparator	カスタムリソースコンパレータを登録できます。これは、着信 URL パスを特定のリソースクラスまたはメソッドに一致させるために使用されるアルゴリズムを実装します。
jaxrs:resourceClasses	**(Blueprint のみ)** クラス名から複数のリソースを作成する場合には、**jaxrs:server/@serviceClass** 属性の代わりに使用することができます。**jaxrs:resourceClasses** の子は、**name** 属性がリソースクラスの名前に設定された **class** 要素でなければなりません。この場合、クラスは Blueprint や Spring では **なく** Apache CXF によってインスタンス化されます。

[a] Invoker 実装は、サービスの呼び出し方法を制御します。たとえば、各リクエストをサービス実装の新しいインスタンスで処理するかどうか、呼び出し間で状態を保持するかどうかを制御します。

18.2. JAX-RS クライアントエンドポイントの設定

18.2.1. JAX-RS クライアントエンドポイントの定義

クライアントプロキシーの挿入

クライアントプロキシー Bean を XML 言語 (BlueprintXML または SpringXML) でインスタンス化する主なポイントは、別の Bean に挿入し、クライアントプロキシーを使用して REST サービスを呼び出すことができるようにすることです。XML でクライアントプロキシー Bean を作成するには、**jaxrs:client** 要素を使用します。

Namespaces

JAX-RS クライアントエンドポイントは、サーバーエンドポイントとは **異なる** XML 名前空間を使用して定義されます。次の表は、どの XML 言語にどの名前空間を使用するかを示しています。

XML 言語	クライアントエンドポイントの namespace
ブループリント	**http://cxf.apache.org/blueprint/jaxrs-client**
Spring	**http://cxf.apache.org/jaxrs-client**

基本のクライアントエンドポイント定義

次の例は、BlueprintXML または SpringXML でクライアントプロキシー Bean を作成する方法を示しています。

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

基本的なクライアントエンドポイントを定義するには、次の属性を設定する必要があります。

id

クライアントプロキシーの Bean ID を使用して、XML 設定内の他の Bean にクライアントプロキシーを挿入できます。

address

address 属性は REST 呼び出しのベース URL を指定します。

serviceClass

serviceClass 属性は、ルートリソースクラス (**@Path** によりアノテーションが付けられる) を指定して、REST サービスの説明を提供します。実際、これは **server** クラスですが、クライアントで直接使用されません。指定されたクラスは、クライアントプロキシーを動的に構築するのに使用されるメタデータ (Java リフレクションおよび JAX-RS アノテーションを介して) にのみ使用されます。

ヘッダーの指定

以下のように、**jaxrs:headers** 子要素を使用して、HTTP ヘッダーをクライアントプロキシーの呼び出しに追加できます。

```
<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 Client Endpoint 属性」 **jaxrs:client** 要素で利用可能な属性を説明します。

表18.3 JAX-RS Client Endpoint 属性

属性	説明
address	コンシューマーが要求を行うエンドポイントの HTTP アドレスを指定します。この値は、コントラクトで設定された値を上書きします。
bindingId	コンシューマーが使用するメッセージバインディングの ID を指定します。有効なバインディング ID のリストは、23章*Apache CXF バインディング ID* に提供されています。
bus	エンドポイントを管理するバスを設定する Spring Bean の ID を指定します。
inheritHeaders	このプロキシーからサブリソースプロキシーが作成された場合に、このプロキシーに設定されたヘッダーを継承するかどうかを指定します。デフォルトは **false** です。
username	単純なユーザー名/パスワード認証に使用されるユーザー名を指定します。
password	単純なユーザー名/パスワード認証に使用されるパスワードを指定します。
modelRef	モデルスキーマをクラスパスリソースとして指定します (例: **classpath:/path/to/model.xml** 形式の URL)。JAX-RS モデルスキーマを定義する方法の詳細については、「モデルスキーマでの REST サービスの定義」 を参照してください。

属性	説明
serviceClass	サービスインターフェイスまたはリソースクラス (**@PATH** によりアノテーションが付けられる) の名前を指定し、JAX-RS サーバー実装から再利用します。この場合、指定されたクラスは直接呼び出され**ません** (実際にはサーバークラスです)。指定されたクラスは、クライアントプロキシーを動的に構築するのに使用されるメタデータ (Java リフレクションおよび JAX-RS アノテーションを介して) にのみ使用されます。
serviceName	JMS トランスポートが使用される特別なケースの JAX-RS エンドポイントのサービス QName を指定します (**ns:name** の形式を使用)。詳細は、「JMS トランスポートの使用」 を参照してください。
threadSafe	クライアントプロキシーがスレッドセーフかどうかを指定します。デフォルトは **false** です。
transportId	(HTTP の代わりに) 非標準のトランスポート層を選択する場合。特に、このプロパティーを **http://cxf.apache.org/transports/jms** に設定すると、JMS トランスポートを選択できます。詳細は、「JMS トランスポートの使用」 を参照してください。
abstract	**(Spring のみ)** Bean が抽象 Bean であるかどうかを指定します。抽象 Bean は、具体的な Bean 定義の親として機能し、インスタンス化されません。デフォルトは **false** です。これを **true** に設定すると、Bean ファクトリーが Bean をインスタンス化しないように指示します。
depends-on	**(Spring のみ)** エンドポイントがインスタンス化される前にインスタンス化されたかどうかに依存する Bean のリストを指定します。

18.2.3. jaxrs:client 子要素

子要素

表18.4「JAX-RS Client Endpoint ポイントの子要素」 に **jaxrs:client** 要素の子要素をまとめます。

表18.4 JAX-RS Client Endpoint ポイントの子要素

要素	説明
jaxrs:executor	

要素	説明
jaxrs:features	Apache CXF の高度な機能を設定する Bean のリストを指定します。Bean 参照のリストまたは埋め込み Bean のリストのいずれかを提供できます。
jaxrs:binding	**使用されていません。**
jaxrs:dataBinding	エンドポイントで使用されるデータバインディングを実装するクラスを指定します。これは、埋め込み Bean 定義を使用して指定されます。詳細は、「データバインディングの指定」 を参照してください。
jaxrs:inInterceptors	インバウンド応答を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxrs:inFaultInterceptors	インバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxrs:outInterceptors	アウトバウンド要求を処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxrs:outFaultInterceptors	アウトバウンド障害メッセージを処理するインターセプターのリストを指定します。詳細は、パートVII「Apache CXF インターセプターの開発」 を参照してください。
jaxrs:properties	エンドポイントに渡されるプロパティーのマップを指定します。
jaxrs:providers	1 つ以上のカスタム JAX-RS プロバイダーをこのエンドポイントに登録できます。この要素の子は、JAX-RS プロバイダーのインスタンス (**bean** 要素) または JAX-RS ルートリソースへの参照 (**ref** 要素) です。
jaxrs:modelBeans	1 つまたは複数の **org.apache.cxf.jaxrs.model.UserResource** Bean への参照のリストで設定されます。この参照はリソースモデルの基本要素です (**jaxrs:resource** 要素に対応)。詳細は、「モデルスキーマでの REST サービスの定義」 を参照してください。

要素	説明
jaxrs:model	このエンドポイントに直接リソースモデルを定義します (つまり、**jaxrs:model** 要素には 1 つまたは複数の **jaxrs:resource** 要素を含めることができます)。詳細は、「モデルスキーマでの REST サービスの定義」 を参照してください。
jaxrs:headers	送信メッセージのヘッダーを設定するために使用されます。詳細は、「ヘッダーの指定」 を参照してください。
jaxrs:schemaLocations	XML メッセージの内容の検証に使用される 1 つ以上の XML スキーマを指定します。この要素には、1 つまたは複数の **jaxrs:schemaLocation** 要素を含めることができます。各要素は、XML スキーマファイルの場所を指定します (通常は **classpath** URL として)。詳細は、「スキーマ検証」 を参照してください。

18.3. モデルスキーマでの REST サービスの定義

アノテーションなしの RESTful サービス

JAX-RS モデルスキーマを使用すると、Java クラスにアノテーションを**付けずに** RESTful サービスを定義できます。つまり、Java クラス (またはインターフェイス) に直接 **@Path**、**@PathParam**、**@Consumes**、**@Produces** などのアノテーションを追加する代わりに、モデルスキーマを使用して、関連する REST メタデータをすべて別の XML ファイルで指定できます。これは、たとえば、サービスを実装する Java ソースを変更できない場合に役立ちます。

モデルスキーマの例

例18.1「サンプル JAX-RS モデルスキーマ」 は、**BookStoreNoAnnotations** ルートリソースクラスのサービスメタデータを定義するモデルスキーマの例を示しています。

例18.1 サンプル JAX-RS モデルスキーマ

```
<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>
```

Namespaces

モデルスキーマの定義に使用する XML 名前空間は、対応する JAX-RS エンドポイントを Blueprint XML で定義するか Spring XML で定義するかによって異なります。次の表は、どの XML 言語にどの名前空間を使用するかを示しています。

XML 言語	Namespace
ブループリント	**http://cxf.apache.org/blueprint/jaxrs**
Spring	**http://cxf.apache.org/jaxrs**

モデルスキーマをエンドポイントにアタッチする方法

モデルスキーマを定義してエンドポイントにアタッチするには、次の手順を実行します。

1. 選択したインジェクションプラットフォーム (Blueprint XML または Spring XML) に適切な XML 名前空間を使用して、モデルスキーマを定義します。
2. モデルスキーマファイルをプロジェクトのリソースに追加して、スキーマファイルが最終パッケージ (JAR、WAR、または OSGi バンドルファイル) のクラスパスで使用できるようにします。

注記

あるいは、エンドポイントの **jaxrs:model** 子要素を使用して、モデルスキーマを直接 JAX-RS エンドポイントに組み込むこともできます。

3. エンドポイントの **modelRef** 属性をクラスパス上のモデルスキーマの場所に設定し (クラスパス URL を使用)、モデルスキーマを使用するようにエンドポイントを設定します。
4. 必要な場合は、**jaxrs:serviceBeans** 要素を使用して、ルートリソースを明示的にインスタンス化します。モデルスキーマが (ベースインターフェイスを参照する代わりに) ルートリソースクラスを直接参照する場合は、この手順をスキップできます。

クラスを参照するモデルスキーマの設定

モデルスキーマがルートリソースクラスに直接適用される場合、モデルスキーマは自動的にルートリソース Bean をインスタンス化するため、**jaxrs:serviceBeans** 要素を使用してルートリソース Bean を定義する必要はありません。

たとえば、**customer-resources.xml** がメタデータをカスタマーリソースクラスに関連付けるモデルスキーマの場合、以下のように **customerService** サービスエンドポイントをインスタンス化できます。

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

インターフェイスを参照するモデルスキーマの設定

モデルスキーマが Java インターフェイス (ルートリソースのベースインターフェイス) に適用される場合、エンドポイントの **jaxrs:serviceBeans** 要素を使用してルートリソースクラスをインスタンス化する必要があります。

たとえば、**customer-interfaces.xml** がメタデータをカスタマーリソースクラスに関連付けるモデルスキーマの場合、以下のように **customerService** サービスエンドポイントをインスタンス化できます。

```
<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"/>
```

モデルスキーマリファレンス

モデルスキーマは、次の XML 要素を使用して定義されます。

model

モデルスキーマのルート要素。モデルスキーマを参照する必要がある場合は (たとえば、**modelRef** 属性を使用して JAX-RS エンドポイントから)、この要素の **id** 属性を設定する必要があります。

model/resource

resource 要素は、メタデータを特定のルートリソースクラス (または対応するインターフェイス) に関連付けるために使用されます。**resource** 要素に以下の属性を定義できます。

属性	説明 +
name	このリソースモデルが適用されるリソースクラス (または対応するインターフェイス) の名前。 +
path	このリソースにマップする REST URL パスのコンポーネント。 +
consumes	このリソースによって使用されるコンテンツの型 (インターネットメディアの型) を指定します (例: **application/xml** または **application/json**)。 +

属性	説明 +
produces	このリソースによって使用されるコンテンツの型 (インターネットメディアの型) を指定します (例: **application/xml** または **application/json**)。 +

model/resource/operation

operation 要素は、メタデータを Java メソッドに関連付けるために使用されます。 **operation** 要素に以下の属性を定義できます。

属性	説明 +
name	この要素が適用される Java メソッドの名前。 +
path	このメソッドにマップする REST URL パスのコンポーネント。この属性値には、パラメーターの参照を含めることができます (例: **path="/books/{id}/chapter"**)。ここで、**{id}** はパスから **id** パラメーターの値を抽出します。 +
verb	このメソッドにマップする HTTP 動詞を指定します。通常、**GET**、**POST**、**PUT**、または **DELETE** のいずれかです。HTTP の動詞が指定されて **いない** 場合、Java メソッドは **サブリソースロケーター** と考えられ、サブリソースオブジェクトへの参照を返します (サブリソースクラスにも **resource** 要素を使用してメタデータを指定する必要があります)。 +
consumes	この操作によって消費されるコンテンツタイプ (インターネットメディアタイプ) を指定します (例: **application/xml** や **application/json**)。 +
produces	この操作で生成されるコンテンツタイプ (インターネットメディアタイプ) を指定します (例:**application/xml** や **application/json**)。 +

属性	説明 +
oneway	**true** の場合、操作を **一方向**に設定します。つまり、リプライメッセージは不要になります。デフォルトは **false** です。 +

model/resource/operation/param

param 要素は、REST URL から値を抽出し、それをメソッドパラメーターのいずれかに注入するために使用されます。**param** 要素で以下の属性を定義できます。

属性	説明 +
name	この要素が適用される Java メソッドパラメーターの名前。 +
type	パラメーター値を REST URL またはメッセージから抽出する方法を指定します。**PATH**、**QUERY**、**MATRIX**、**HEADER**、**COOKIE**、**FORM**、**CONTEXT**、**REQUEST_BODY** のいずれかの値に設定できます。 +
defaultValue	REST URL またはメッセージから値を抽出できなかった場合に、パラメーターに挿入するデフォルト値。 +
encoded	**true** の場合、パラメーター値は URI でエンコードされた形式で挿入されます (つまり **%nn** エンコーディングを使用します)。デフォルトは **false** です。たとえば、URL パス (/**name**/**Joe%20Bloggs**) からパラメーターを抽出する際に、エンコードが **true** に設定されると、パラメーターは **Joe%20Bloggs** として注入されます。そうでない場合は、パラメーターは **Joe Bloggs** として注入されます。 +

第19章 APACHE CXF ロギング

概要

この章では、Apache CXF ランタイムでロギングを設定する方法について説明します。

19.1. APACHE CXF ロギングの概要

概要

Apache CXF は Java ロギングユーティリティー **java.util.logging** を使用します。ロギングは、標準の **java.util.Properties** 形式を使用して書き込まれるロギング設定ファイルで設定されます。アプリケーションでログを実行するには、プログラムでログを指定するか、アプリケーションの起動時にログ設定ファイルを指すコマンドでプロパティーを定義します。

デフォルトのプロパティーファイル

Apache CXF には、**InstallDir/etc** ディレクトリーにデフォルトの **logging.properties** ファイルが含まれています。このファイルは、ログメッセージの出力先と公開されるメッセージレベルの両方を設定します。デフォルト設定は、**WARNING** レベルでフラグが付けられたメッセージをコンソールへ出力するようにロガーを設定します。設定オプションを変更せずにデフォルトのファイルを使用することも、特定のアプリケーションに合わせて設定オプションを変更することもできます。

ロギング機能

Apache CXF には、ロギングを有効にするためにクライアントまたはサービスにプラグインできるロギング機能が含まれています。例19.1「ロギングの有効化設定」 は、ログ機能を有効にするための設定を示しています。

例19.1 ロギングの有効化設定

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

詳細は、「メッセージコンテンツのログ」 を参照してください。

どこから始めますか？

ロギングの簡単な例を実行するには、「ロギングを使用する簡単な例」。

Apache CXF でのロギングの動作の詳細については、この章全体をお読みください。

java.util.logging に関する詳細情報

java.util.logging ユーティリティーは、最も広く使用されている Java ロギングフレームワークの 1 つです。このフレームワークの使用方法と拡張方法を説明するオンラインで入手可能な情報はたくさんあります。ただし、開始点として、以下のドキュメントでは **java.util.logging** の詳細な概要を説明します。

- 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. ロギングを使用する簡単な例

ログレベルと出力先の変更

wsdl_first サンプルアプリケーションでログレベルとログメッセージの出力先を変更するには、以下のステップを実行してください。

1. **InstallDir/samples/wsdl_first** ディレクトリーの **README.txt** ファイルの **java を使用したデモの実行** に記載されているサンプルサーバーを実行します。**server start** コマンドは、 以下のようにデフォルトの **logging.properties** ファイルを指定することに注意してください。

プラットフォーム	コマンド +
Windows	**start 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 &** +

   デフォルトの **logging.properties** ファイルは **InstallDir/etc** ディレクトリーにあります。Apache CXF ロガーを設定し、**WARNING** レベルのログメッセージをコンソールに出力します。その結果、コンソールに印刷されるものはほとんどありません。

2. **README.txt** ファイルの説明に従ってサーバーを停止します。

3. デフォルトの **logging.properties** ファイルのコピーを作成します。名前が **mylogging.properties** ファイルになり、デフォルトの **logging.properties** ファイルと同じディレクトリーに保存します。

4. 以下の設定行を編集して、**mylogging.properties** ファイルのグローバルロギングレベルおよびコンソールロギングレベルを **INFO** に変更します。

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

5. 次のコマンドを使用してサーバーを再起動します。

プラットフォーム	コマンド +

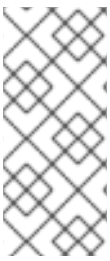
プラットフォーム	コマンド +
Windows	**start java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server** +
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &** +

レベル **INFO** のメッセージをログに記録するようにグローバルロギングとコンソールロガーを設定しているため、コンソールに多くのログメッセージが表示されます。

19.3. デフォルトのログ設定ファイル

19.3.1. ロギング設定の概要

デフォルトのロギング設定ファイル **logging.properties** は **InstallDir/etc** ディレクトリーにあります。Apache CXF ロガーを設定し、**WARNING** レベルのメッセージをコンソールに出力します。このレベルのログがアプリケーションに適している場合は、使用する前にファイルに変更を加える必要はありません。ただし、ログメッセージの詳細レベルを変更することはできます。たとえば、ログメッセージをコンソールに送信するか、ファイルに送信するか、またはその両方に送信するかを変更できます。さらに、個々のパッケージのレベルでロギングを指定できます。



注記

このセクションでは、デフォルトの **logging.properties** ファイルに表示される設定プロパティーを説明します。ただし、設定可能なその他の **java.util.logging** 設定プロパティーが多数あります。**java.util.logging** API の詳細は、http://download.oracle.com/javase/1.5/docs/api/java/util/logging/package-summary.html の **java.util.logging** を参照してください。

19.3.2. ロギング出力の設定

概要

Java ロギングユーティリティー **java.util.logging** は、ハンドラークラスを使用してログメッセージを出力します。表19.1「Java.util.logging Handler Classes」 は、デフォルトの **logging.properties** ファイルで設定されるハンドラーを示しています。

表19.1 Java.util.logging Handler Classes

ハンドラークラス	への出力
ConsoleHandler	ログメッセージをコンソールに出力します

ハンドラークラス	への出力
FileHandler	ログメッセージをファイルに出力します

重要

ハンドラークラスは、Java VM の起動時にインストールされるために、システムクラスパス上にある必要があります。これは、Apache CXF 環境を設定するときに行われます。

コンソールハンドラーの設定

例19.2「コンソールハンドラーの設定」 は、コンソールロガーを設定するためのコードを示しています。

例19.2 コンソールハンドラーの設定

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

コンソールハンドラーは、例19.3「コンソールハンドラーのプロパティー」 に示す設定プロパティーもサポートします。

例19.3 コンソールハンドラーのプロパティー

```
java.util.logging.ConsoleHandler.level = WARNING
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

例19.3「コンソールハンドラーのプロパティー」 に示す設定プロパティー次のように説明することができます。

コンソールハンドラーは、個別のログレベル設定プロパティーをサポートします。これにより、コンソールに出力されるログメッセージを制限できますが、グローバルログ設定は異なる場合があります (「ロギングレベルの設定」 を参照)。デフォルト設定は **WARNING** です。

コンソールハンドラークラスがログメッセージのフォーマットに使用する **java.util.logging** フォーマッタークラスを指定します。デフォルト設定は **java.util.logging.SimpleFormatter** です。

ファイルハンドラーの設定

例19.4「ファイルハンドラーの設定」 は、ファイルハンドラーを設定するコードを示しています。

例19.4 ファイルハンドラーの設定

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

ファイルハンドラーは、例19.5「ファイルハンドラーの設定プロパティー」 に示す設定プロパティーもサポートします。

例19.5 ファイルハンドラーの設定プロパティー

```
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「ファイルハンドラーの設定プロパティー」 に示す設定プロパティー次のように説明することができます:

出力ファイルの場所とパターンを指定します。デフォルト設定はホームディレクトリーです。

ロガーが任意の 1 つのファイルに書き込む最大量をバイト単位で指定します。デフォルト設定は **50000** です。ゼロに設定すると、ロガーが 1 つのファイルに書き込む量に制限はありません。

循環する出力ファイルの数を指定します。デフォルト設定は **1** です。

ファイルハンドラークラスがログメッセージのフォーマットに使用する **java.util.logging** フォーマッタークラスを指定します。デフォルト設定は **java.util.logging.XMLFormatter** です。

コンソールハンドラーとファイルハンドラーの両方を設定する

コンソールログとファイルの両方の設定 に示すように、コンソールハンドラーとファイルハンドラーをコンマで区切って指定することにより、ログメッセージをコンソールとファイルの両方に出力するようにログユーティリティーを設定できます。

コンソールログとファイルの両方の設定

```
Logging
```

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

19.3.3. ロギングレベルの設定

ロギングレベル

java.util.logging フレームワークは、最も詳細度の高いものから最も詳細なロギングレベルをサポートします。

- **SEVERE**
- **WARNING**
- **INFO**
- **CONFIG**
- **FINE**
- **FINER**
- **FINEST**

グローバルロギングレベルの設定

すべてのロガーでログに記録されるイベントのタイプを設定するには、例19.6「グローバルログレベルの設定」 に示すようにグローバルログレベルを設定します。

例19.6 グローバルログレベルの設定

```
.level= WARNING
```

個々のパッケージでのロギングの設定

```
level
```

java.util.logging フレームワークは、個別のパッケージレベルでのロギングの設定をサポートします。たとえば、例19.7「パッケージレベルでのロギングの設定」 で示されているコードの行は、**com.xyz.foo** パッケージのクラス上の **SEVERE** レベルでロギングを設定します。

例19.7 パッケージレベルでのロギングの設定

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

19.4. コマンドラインでのロギングの有効化

概要

アプリケーションの起動時に **java.util.logging.config.file** プロパティーを定義することで、アプリケーションでロギングユーティリティーを実行できます。デフォルトの **logging.properties** ファイル、またはそのアプリケーションに固有の **logging.properties** ファイルのいずれかを指定できます。

アプリケーションでのログ設定ファイルの指定

```
start-up
```

アプリケーションの起動時にロギングを指定するには、アプリケーションを起動するときに 例19.8「コマンドラインでロギングを開始するためのフラグ」 に示すフラグを追加します。

例19.8 コマンドラインでロギングを開始するためのフラグ

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

19.5. サブシステムとサービスのロギング

概要

「個々のパッケージでのロギングの設定」 に記載されている **com.xyz.foo.level** 設定プロパティーを使用して、指定された Apache CXF ロギングサブシステムの詳細なロギングを設定できます。

Apache CXF ロギングサブシステム

表19.2「Apache CXF ロギングサブシステム」 は、利用可能な Apache CXF ロギングサブシステムのリストを示しています。

表19.2 Apache CXF ロギングサブシステム

サブシステム	説明
org.apache.cxf.aegis	イージスバインディング
org.apache.cxf.binding.coloc	コロケーションバインディング
org.apache.cxf.binding.http	HTTP バインディング
org.apache.cxf.binding.jbi	JBI バインディング
org.apache.cxf.binding.object	Java オブジェクトバインディング
org.apache.cxf.binding.soap	SOAP バインディング
org.apache.cxf.binding.xml	XML バインディング
org.apache.cxf.bus	Apache CXF バス
org.apache.cxf.configuration	設定フレームワーク
org.apache.cxf.endpoint	サーバーとクライアントのエンドポイント
org.apache.cxf.interceptor	interceptors
org.apache.cxf.jaxws	JAX-WS スタイルのメッセージ交換、JAX-WS ハンドラー処理、および JAX-WS と設定に関連するインターセプターのフロントエンド
org.apache.cxf.jbi	JBI コンテナー統合クラス
org.apache.cxf.jca	JCA コンテナー統合クラス
org.apache.cxf.js	JavaScript フロントエンド
org.apache.cxf.transport.http	HTTP トランスポート
org.apache.cxf.transport.https	HTTPS を使用した HTTP トランスポートの安全なバージョン
org.apache.cxf.transport.jbi	JBI トランスポート
org.apache.cxf.transport.jms	JMS トランスポート

サブシステム	説明
org.apache.cxf.transport.local	ローカルファイルシステムを使用したトランスポートの実装
org.apache.cxf.transport.servlet	JAX-WS エンドポイントをサーブレットコンテナーにロードするための HTTP トランスポートとサーブレットの実装
org.apache.cxf.ws.addressing	WS-Addressing の実装
org.apache.cxf.ws.policy	WS-Policy の実装
org.apache.cxf.ws.rm	WS-ReliableMessaging (WS-RM) の実装
org.apache.cxf.ws.security.wss4j	WSS4J セキュリティーの実装

例

WS-Addressing サンプルは **InstallDir/samples/ws_addressing** ディレクトリーに含まれます。ロギングは、そのディレクトリーにある **logging.properties** ファイルに設定されます。関連する設定行は 例19.9「WS-Addressing のロギングの設定」 に示します。

例19.9 WS-Addressing のロギングの設定

```
java.util.logging.ConsoleHandler.formatter = demos.ws_addressing.common.ConciseFormatter
...
org.apache.cxf.ws.addressing.soap.MAPCodec.level = INFO
```

例19.9「WS-Addressing のロギングの設定」 での設定は、WS-Addressing ヘッダーに関連するログメッセージのスヌーピングを有効にし、簡潔な形式でコンソールに表示します。

このサンプルの実行については、**InstallDir/samples/ws_addressing** ディレクトリーにある **README.txt** ファイルを参照してください。

19.6. メッセージコンテンツのログ

概要

サービスとコンシューマーの間で送信されるメッセージの内容をログに記録できます。たとえば、サービスとコンシューマーの間で送信されている SOAP メッセージの内容をログに記録したい場合があります。

メッセージコンテンツロギングの設定

サービスとコンシューマーの間で送信されるメッセージをログに記録するには、またはその逆の場合は、次の手順を実行します。

1. エンドポイントの設定にログ機能を追加します。

2. ロギング機能をコンシューマーの設定に追加します。
3. ロギングシステムログ **INFO** レベルのメッセージを設定します。

エンドポイントへのロギング機能の追加

例19.10「エンドポイント設定へのロギングの追加」 に示すように、ログ機能をエンドポイントの設定に追加します。

例19.10 エンドポイント設定へのロギングの追加

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

例19.10「エンドポイント設定へのロギングの追加」 に示す XML の例 SOAP メッセージのロギングを有効にします。

ロギング機能をコンシューマーに追加する

例19.11「クライアント設定へのログの追加」 に示すように、クライアントの設定でログ機能を追加します。

例19.11 クライアント設定へのログの追加

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

例19.11「クライアント設定へのログの追加」 に示す XML の例 SOAP メッセージのロギングを有効にします。

INFO レベルのメッセージをログに記録するようにログを設定する

例19.12「ロギングレベルを INFO に設定する」 に示されるように、サービスに関連する **logging.properties** ファイルが **INFO** レベルのメッセージをログに記録するように設定されていることを確認します。

例19.12 ロギングレベルを INFO に設定する

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

SOAP メッセージのロギング

SOAP メッセージのロギングを確認するには、以下のように **InstallDir/samples/wsdl_first** ディレクトリーにある wsdl_first サンプルアプリケーションを変更します。

1. 例19.13「SOAP メッセージをログに記録するためのエンドポイント設定」 に記載されている **jaxws:features** 要素を、wsdl_first サンプルディレクトリーにある **cxf.xml** 設定ファイルに追加します。

例19.13 SOAP メッセージをログに記録するためのエンドポイント設定

```
<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. この例では、**InstallDir/etc** ディレクトリーにあるデフォルトの **logging.properties** ファイルを使用します。このファイルのコピーを作成し、これに **mylogging.properties** という名前を付けます。

3. **mylogging.properties** ファイルで、以下のように java. **level** と **java.util.logging.ConsoleHandler.level** 設定プロパティーを編集して、ロギングレベルを **INFO** に変更します。

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

4. 以下のように **cxf.xml** ファイルと **mylogging.properties** ファイルの両方で、新しい設定を使用してサーバーを起動します。

プラットフォーム	コマンド +
Windows	**start 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. 次のコマンドを使用して、hello world クライアントを起動します。

プラットフォーム	コマンド +
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 メッセージはコンソールに記録されます。

第20章 WS-ADDRESSING のデプロイ

概要

Apache CXF は、JAX-WS アプリケーションの WS-Addressing をサポートしています。この章では、Apache CXF ランタイム環境に WS-Addressing をデプロイする方法について説明します。

20.1. WS-ADDRESSING の概要

概要

WS-Addressing は、サービスがトランスポートニュートラルな方法でアドレス情報を通信できるようにする仕様です。これは 2 つの部分で設定されています。

- Web サービスエンドポイントへの参照を通信するための構造
- アドレス情報を特定のメッセージに関連付けるメッセージアドレス指定プロパティー (MAP) のセット

サポートされている仕様

Apache CXF は、WS-Addressing 2004/08 仕様と WS-Addressing 2005/03 仕様の両方をサポートしています。

その他の情報

WS-Addressing の詳細は、http://www.w3.org/Submission/ws-addressing/ で 2004/08 の投稿を参照してください。

20.2. WS-ADDRESSING インターセプター

概要

Apache CXF では、WS-Addressing 機能がインターセプターとして実装されています。Apache CXF ランタイムは、インターセプターを使用して、送受信されている生のメッセージを傍受して処理します。トランスポートはメッセージを受信すると、メッセージオブジェクトを作成し、そのメッセージをインターセプターチェーンを介して送信します。WS-Addressing インターセプターがアプリケーションのインターセプターチェーンに追加されると、メッセージに含まれる WS-Addressing 情報が処理されます。

WS-Addressing インターセプター

WS-Addressing の実装は、表20.1「WS-Addressing インターセプター」 で説明されているように 2 つのインターセプターで設定されています。

表20.1 WS-Addressing インターセプター

インターセプター	説明

インターセプター	説明
org.apache.cxf.ws.addressing.MAPAggregator	送信メッセージのメッセージアドレス指定プロパティー (MAP) の集約を担当する論理インターセプター。
org.apache.cxf.ws.addressing.soap.MAPCodec	メッセージアドレス指定プロパティー (MAPs) を SOAP ヘッダーとしてエンコードおよびデコードするプロトコル固有のインターセプター。

20.3. WS-ADDRESSING の有効化

概要

WS-Addressing を有効にするには、WS-Addressing インターセプターをインバウンドおよびアウトバウンドインターセプターチェーンに追加する必要があります。これは、次のいずれかの方法で行われます。

- Apache CXF の機能
- RMAssertion と WS-Policy フレームワーク
- WS-Addressing 機能でのポリシーアサーションの使用

機能としての WS-Addressing の追加

WS-Addressing を有効にするには、例20.1「client.xml およびクライアント設定への WS-Addressing 機能の追加」 と 例20.2「server.xml およびサーバー設定への WS-Addressing 機能の追加」 それぞれで示されるように WS-Addressing 機能をクライアントとサーバー設定に追加します。

例20.1 client.xml およびクライアント設定への WS-Addressing 機能の追加

```
<?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 およびサーバー設定への WS-Addressing 機能の追加

```
<?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. WS-ADDRESSING 属性の設定

概要

Apache CXF WS-Addressing 機能要素は、namespace **http://cxf.apache.org/ws/addressing** で定義されています。表20.2「WS-Addressing 属性」 で説明されている 2 つの属性をサポートします。

表20.2 WS-Addressing 属性

属性名	値
allowDuplicates	重複する MessageID が許容されるかどうかを決定するブール値。デフォルト設定は **true** です。
usingAddressingAdvisory	WSDL に **UsingAddressing** 要素が存在するかどうかを示すブール値。これは、WS-Addressing ヘッダーのエンコーディングを阻止しません。

WS-Addressing 属性の設定

サーバーまたはクライアント設定ファイルの WS-Addressing 機能に属性と設定する値を追加して、WS-Addressing 属性を設定します。たとえば、以下の設定抽出は、サーバーエンドポイントで **allowDuplicates** 属性を **false** に設定します。

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing" ...>
    <jaxws:endpoint ...>
      <jaxws:features>
        <wsa:addressing allowDuplicates="false"/>
      </jaxws:features>
    </jaxws:endpoint>
</beans>
```

機能に埋め込まれた WS-Policy アサーションの使用

例20.3「ポリシーを使用した WS-Addressing の設定」 では、匿名の応答を有効にするアドレスポリシーアサーションが **policies** 要素に組み込まれます。

例20.3 ポリシーを使用した WS-Addressing の設定

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       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 信頼性の高いメッセージングの有効化

概要

Apache CXF は、WS-Reliable Messaging (WS-RM) をサポートしています。この章では、Apache CXF で WS-RM を有効にして設定する方法について説明します。

21.1. WS-RM の概要

概要

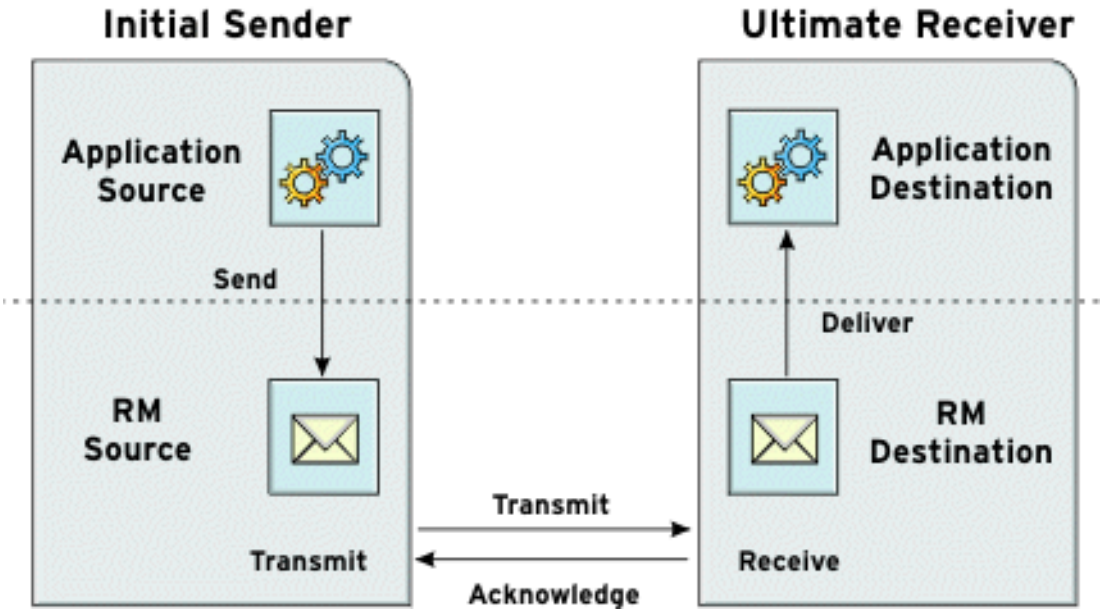
WS-ReliableMessaging (WS-RM) は、分散環境でのメッセージの信頼性の高い配信を保証するプロトコルです。これにより、ソフトウェア、システム、またはネットワークに障害が発生した場合でも、分散アプリケーション間でメッセージを確実に配信できます。

たとえば、WS-RM を使用すると、正しいメッセージがネットワークを介して 1 回だけ、正しい順序で配信されていることを確認できます。

WS-RM のしくみ

WS-RM は、送信元エンドポイントと宛先エンドポイント間のメッセージの信頼性の高い配信を保証します。図21.1「Web サービスの信頼できるメッセージング」 に示すように、送信元はメッセージの最初の送信者であり、宛先は最終的な受信者です。

図21.1 Web サービスの信頼できるメッセージング



WS-RM メッセージのフローは、次のように説明できます。

1. RM ソースは **CreateSequence** プロトコルメッセージを RM 宛先に送信します。これには、確認応答を受け取るエンドポイント (**wsrm:AcksTo** エンドポイント) の参照が含まれます。

2. RM 宛先は **CreateSequenceResponse** プロトコルメッセージを RM ソースに戻します。このメッセージには、RM シーケンスセッションのシーケンス ID が含まれています。

3. RM ソースは、アプリケーションソースが送信する各メッセージに RM **Sequence** ヘッダーを追加します。このヘッダーには、シーケンス ID と一意のメッセージ ID が含まれています。

4. RM 送信元は、各メッセージを RM 宛先に送信します。

5. RM 宛先は、RM **SequenceAcknowledgement** ヘッダーが含まれるメッセージを送信し、RM ソースからメッセージの受信を認識します。

6. RM 宛先は、メッセージを 1 回限りの順序でアプリケーション宛先に配信します。

7. RM ソースは、確認応答をまだ受信していないというメッセージを再送信します。
最初の再送信の試行は、基本再送信間隔の後に行われます。連続した再送信の試行は、デフォルトで、指数バックオフ間隔で、または代わりに、固定間隔で行われます。詳細は、「WS-RM の設定」 を参照してください。

このプロセス全体は、要求メッセージと応答メッセージの両方で対称的に発生します。つまり、応答メッセージの場合、サーバーは RM ソースとして機能し、クライアントは RM 宛先として機能します。

WS-RM 配信保証

WS-RM は、使用されるトランスポートプロトコルに関係なく、分散環境での信頼性の高いメッセージ配信を保証します。信頼できる配信が保証されない場合は、送信元エンドポイントまたは宛先エンドポイントのいずれかがエラーをログに記録します。

サポートされている仕様

Apache CXF は、次のバージョンの WS-RM 仕様をサポートしています。

WS-ReliableMessaging 1.0

(デフォルト) 2005 年 2 月の提出バージョン に対応しますが、現在は古くなっています。ただし、下位互換性のため、このバージョンがデフォルトとして使用されます。
WS-RM のバージョン 1.0 は、次の名前空間を使用します。

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

このバージョンの WS-RM は、次の WS-Addressing バージョンのいずれかで使用できます。

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

厳密に言えば、WS-RM の 2005 年 2 月の提出バージョンに準拠するには、これらの WS-Addressing バージョンの最初のバージョン (Apache CXF のデフォルト) を使用する必要があります。ただし、他のほとんどの Web サービス実装は最新の WS-Addressing 仕様に切り替えられているため、Apache CXF では WS-A バージョンを選択して、相互運用性を促進できます (「ランタイム制御」 を参照)。

WS-ReliableMessaging 1.1/1.2

公式の 1.1/1.2 Web サービスの信頼できるメッセージング 仕様に対応します。
WS-RM のバージョン 1.1 および 1.2 は、次の名前空間を使用します。

> http://docs.oasis-open.org/ws-rx/wsrm/200702

WS-RM の 1.1 および 1.2 バージョンは、次の WS-Addressing バージョンを使用します。

> http://www.w3.org/2005/08/addressing

WS-RM バージョンの選択

次のように、使用する WS-RM 仕様のバージョンを選択できます。

サーバー側

プロバイダー側では、Apache CXF は、クライアントが使用する WS-ReliableMessaging のバージョンに適応し、適切に応答します。

クライアント側の設定

クライアント側では、WS-RM バージョンは、クライアント設定で使用するネームスペースによって決定されます (「WS-RM の設定」 を参照)。または、ランタイム制御オプションを使用して、実行時に WS-RM バージョンをオーバーライドする (「ランタイム制御」 を参照)。

21.2. WS-RM インターセプター

概要

Apache CXF では、WS-RM 機能がインターセプターとして実装されます。Apache CXF ランタイムは、インターセプターを使用して、送受信されている生のメッセージを傍受して処理します。トランスポートはメッセージを受信すると、メッセージオブジェクトを作成し、そのメッセージをインターセプターチェーンを介して送信します。アプリケーションのインターセプターチェーンに WS-RM インターセプターが含まれている場合、アプリケーションは信頼性の高いメッセージングセッションに参加できます。WS-RM インターセプターは、メッセージチャンクの収集と集約を処理します。また、すべての確認応答および再送信ロジックを処理します。

Apache CXF -RM インターセプター

Apache CXF WS-RM の実装は、表21.1「Apache CXF WS-ReliableMessaging インターセプター」 で示されているように 4 つのインターセプターで設定されています。

表21.1 Apache CXF WS-ReliableMessaging インターセプター

インターセプター	説明
org.apache.cxf.ws.rm.RMOutInterceptor	送信メッセージの信頼性保証を提供する論理的な側面を扱います。 **CreateSequence** リクエストを送り、**CreateSequenceResponse 応答を** 待ちます。 また、アプリケーションメッセージのシーケンスプロパティー (ID とメッセージ番号) を集約するロールも果たします。
org.apache.cxf.ws.rm.RMInInterceptor	アプリケーションメッセージに Piggy backed である RM プロトコルメッセージと **SequenceAcknowledgement** メッセージの傍受および処理を行います。
org.apache.cxf.ws.rm.RMCaptureInInterceptor	永続ストレージ用の着信メッセージのキャッシュ。

インターセプター	説明
org.apache.cxf.ws.rm.RMDeliveryInterceptor	アプリケーションへのメッセージの InOrder 配信を保証します。
org.apache.cxf.ws.rm.soap.RMSoapInterceptor	信頼性プロパティーを SOAP ヘッダーとしてエンコードおよびデコードする責任があります。
org.apache.cxf.ws.rm.RetransmissionInterceptor	将来の再送信のためにアプリケーションメッセージのコピーを作成する責任があります。

WS-RM の有効化

インターセプターチェーンに WS-RM インターセプターが存在することで、必要に応じて WS-RM プロトコルメッセージが交換されます。たとえば、アウトバウンドインターセプターチェーンで最初のアプリケーションメッセージをインターセプトすると、 **RMOutInterceptor** は **CreateSequence** リクエストを送信し、**CreateSequenceResponse** 応答を受信するまで元のアプリケーションメッセージを処理するのを待ちます。さらに、WS-RM インターセプターは、シーケンスヘッダーをアプリケーションメッセージに追加し、宛先側でメッセージから抽出します。メッセージの交換を信頼できるものにするために、アプリケーションコードに変更を加える必要はありません。

WS-RM を有効にする方法の詳細については、 「WS-RM の有効化」 を参照してください。

WS-RM 属性の設定

設定を通じて、シーケンスの境界設定や信頼性の高い交換の他の側面を制御します。たとえば、デフォルトでは、Apache CXF はシーケンスの存続期間を最大化しようとするため、帯域外 WS-RM プロトコルメッセージによって発生するオーバーヘッドが削減されます。アプリケーションメッセージごとに別のシーケンスを使用するようにするには、WS-RM ソースシーケンス終了ポリシーを設定します (最大シーケンスの長さを **1** に設定)。

WS-RM 動作の設定の詳細については、 「WS-RM の設定」 を参照してください。

21.3. WS-RM の有効化

概要

信頼性の高いメッセージングを有効にするには、インバウンドとアウトバウンドの両方のメッセージと障害のインターセプターチェーンに WS-RM インターセプターを追加する必要があります。WS-RM インターセプターは WS-Addressing を使用するため、WS-Addressing インターセプターもインターセプターチェーンに存在する必要があります。

これらのインターセプターの存在を確認するには、次の 2 つの方法のいずれかを使用します。

- 明示的 に、Spring beans を使用してディスパッチチェーンに追加します
- 暗黙的 に、WS-Policy アサーションを使用します。これにより、Apache CXF ランタイムがユーザーに代わってインターセプターを透過的に追加します。

Spring beans: 明示的にインターセプターを追加する

WS-RM を有効にするには、WS-RM および WS-Addressing インターセプターを Apache CXF バス、または Spring bean 設定を使用するコンシューマーまたはサービスエンドポイントに追加します。これは、**InstallDir/samples/ws_rm** ディレクトリーにある WS-RM サンプルで取得した手法です。設定ファイル **ws-rm.cxf** では、WS-RM と WS-Addressing インターセプターが Spring Bean として 1 つずつ追加されています (例21.1「Spring Beans を使用した WS-RM の有効化」 を参照)。

例21.1 Spring Beans を使用した WS-RM の有効化

```
<?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「Spring Beans を使用した WS-RM の有効化」 に示すコードを次のように説明することができます。

Apache CXF 設定ファイルは Spring XML ファイルです。**beans** 要素によってカプセル化される子要素の namespace およびスキーマファイルを宣言するオープニング Spring **beans** 要素を含める必要があります。

各 WS-Addressing インターセプター (**MAPAggregator** および **MAPCodec**) を設定します。WS-Addressing の詳細については、20章*WS-Addressing のデプロイ* を参照してください。

WS-RM インターセプター - **RMOutInterceptor**、**RMInInterceptor**、および **RMSoapInterceptor** を設定します。

インバウンドメッセージのインターセプターチェーンに WS-Addressing および WS-RM インターセプターを追加します。

インバウンド障害のインターセプターチェーンに WS-Addressing および WS-RM インターセプターを追加します。

WS-Addressing および WS-RM インターセプターをアウトバウンドメッセージのインターセプターチェーンに追加します。

WS-Addressing および WS-RM インターセプターをアウトバウンド障害のインターセプターチェーンに追加します。

WS-Policy フレームワーク。暗黙的にインターセプターを追加する

WS-Policy フレームワークは、WS-Policy を使用できるようにするインフラストラクチャーと API を提供します。これは、Web サービスポリシー 1.5- フレームワーク および Web サービスポリシー 1.5- 添付ファイル の仕様の 2006 年 11 月のドラフト公開に準拠しています。

Apache CXF WS-Policy フレームワークを使用して WS-RM を有効にするには、次の手順を実行します。

1. ポリシー機能をクライアントとサーバーのエンドポイントに追加します。例21.2「WS-Policy を使用した WS-RM の設定」 は、**jaxws:feature** 要素内にネストされた参照 Bean を表示します。参照 Bean は **AddressingPolicy** を指定します。これは、同じ設定ファイル内で別の要素として定義されます。

   例21.2 WS-Policy を使用した WS-RM の設定

   ```
   <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. 例21.3「WSDL ファイルへの RM ポリシーの追加」 に示すように、信頼できるメッセージングポリシーを **wsdl:service** 要素またはポリシー参照要素として使用できる他の WSDL 要素に追加します。

例21.3 WSDL ファイルへの RM ポリシーの追加

```
<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. ランタイム制御

概要

org.apache.cxf.ws.rm.RMManager クラスのパブリック定数で定義されたキー値を使用して、クライアントコードで複数のメッセージコンテキストプロパティー値を設定して WS-RM を制御できます。

ランタイム制御オプション

以下の表は、**org.apache.cxf.ws.rm.RMManager** クラスによって定義されるキーのリストです。

キー	説明
WSRM_VERSION_PROPERTY	文字列の WS-RM バージョン名前空間 (**http://schemas.xmlsoap.org/ws/2005/02/rm/** または **http://docs.oasis-open.org/ws-rx/wsrm/200702**)。

キー	説明
WSRM_WSA_VERSION_PROPERTY	String WS-Addressing バージョンネームスペース (**http://schemas.xmlsoap.org/ws/2004/08/addressing** または **http://www.w3.org/2005/08/addressing**) - このプロパティーは、**http://schemas.xmlsoap.org/ws/2005/02/rm/** RM ネームスペースを使用していない限りは無視されます)。
WSRM_LAST_MESSAGE_PROPERTY	ブール値 **true** で、最後のメッセージが送信されたことを WS-RM コードに指示し、コードが WS-RM シーケンスおよびリリースリソースを表示できるようにします (C 3.0.0 バージョンの CXF より、クライアントを閉じると WS-RM はデフォルトで RM シーケンスを閉じます)。
WSRM_INACTIVITY_TIMEOUT_PROPERTY	ミリ秒単位の長い非アクティブタイムアウト。
WSRM_RETRANSMISSION_INTERVAL_PROPERTY	ミリ秒単位の長いベース再送信間隔。
WSRM_EXPONENTIAL_BACKOFF_PROPERTY	ブール値バックオフフラグ。
WSRM_ACKNOWLEDGEMENT_INTERVAL_PROPERTY	ミリ秒単位の長い確認応答間隔。

JMX を介した WS-RM の制御

Apache CXF の JMX 管理機能を使用して、WS-RM の多くの側面を監視および制御することもできます。JMX 操作の完全なリストは **org.apache.cxf.ws.rm.ManagedRMManager** および **org.apache.cxf.ws.rm.ManagedRMEndpoint** で定義されますが、これらの操作には現在の RM 状態を表示することが含まれます。JXM を使用して、WS-RM シーケンスを閉じたり終了したり、以前に送信されたメッセージがリモート RM エンドポイントによって確認されたときに通知を受信したりすることもできます。

JMX 制御の例

たとえば、クライアント設定で JMX サーバーを有効にしている場合は、次のコードを使用して、最後に受信した確認応答番号を追跡できます。

```
// 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. WS-RM の設定

21.5.1. Apache CXF 固有の WS-RM 属性の設定

概要

Apache CXF 固有の属性を設定するには、**rmManager** Spring Bean を使用します。以下を設定ファイルに追加します。

- 名前空間のリストへの **http://cxf.apache.org/ws/rm/manager** 名前空間。
- 設定する特定の属性用の **rmManager** Spring Bean。

例21.4「Apache CXF 固有の WS-RM 属性の設定」 簡単な例を示します。

例21.4 Apache CXF 固有の WS-RM 属性の設定

```
&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 の子

表21.2「rmManager Spring Bean の子」 は、**http://cxf.apache.org/ws/rm/manager** namespace で定義される **rmManager** Spring Bean の子要素を表示します。

表21.2 rmManager Spring Bean の子

要素	説明
RMAssertion	**RMAssertion** 型の要素
deliveryAssurance	適用すべき配信保証を記述する **DeliveryAssuranceType** の要素
sourcePolicy	RM ソースの詳細を設定できるようにするタイプ **SourcePolicyType** の要素
destinationPolicy	RM 宛先の詳細を設定できるようにするタイプ **DestinationPolicyType** の要素

例

例は、「未確認メッセージの最大しきい値」 を参照してください。

21.5.2. 標準の WS-RM ポリシー属性の設定

概要

次のいずれかの方法で、標準の WS-RM ポリシー属性を設定できます。

- 「rmManager Spring bean の RMAssertion」
- 「機能内のポリシー」
- 「WSDL ファイル」
- 「外部アタッチメント」

WS-Policy RMAssertion の子

表21.3「WS-Policy RMAssertion 要素の子」 は 、**http://schemas.xmlsoap.org/ws/2005/02/rm/policy** 名前空間で定義される要素を表示します。

表21.3 WS-Policy RMAssertion 要素の子

Name	説明
InactivityTimeout	エンドポイントが非アクティブのために RM シーケンスが終了したと見なす前に、メッセージを受信せずに経過する必要がある時間を指定します。

Name	説明
BaseRetransmissionInterval	特定のメッセージについて RM ソースが確認応答を受信する必要がある間隔を設定します。**BaseRetransmissionInterval** によって設定された時間内に確認応答が受信されなかった場合、RM ソースはメッセージを再送信します。
ExponentialBackoff	一般的に知られている指数バックオフアルゴリズム (Tanenbaum) を使用して再送信間隔が調整されることを示します。 詳細については、**Computer Networks**、Andrew S. Tanenbaum、Prentice Hall PTR、2003 を参照してください。
AcknowledgementInterval	WS-RM では、確認応答は返信メッセージで送信されるか、スタンドアロンで送信されます。確認応答を送信するための返信メッセージが利用できない場合、RM 宛先は、スタンドアロン確認応答を送信する前に、確認応答間隔まで待機できます。確認応答されていないメッセージがない場合、RM 宛先は確認応答を送信しないことを選択できます。

より詳細な参照情報

各要素のサブ要素と属性の説明を含む、より詳細な参照情報については、http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd を参照してください。

rmManager Spring bean の RMAssertion

標準の WS-RM ポリシー属性を設定するには、Apache CXF **rmManager** Spring Bean 内に **RMAssertion** を追加します。これは、すべての WS-RM 設定を同じ設定ファイルに保持する場合に最適なアプローチです。つまり、Apache CXF 固有の属性と標準の WS-RM ポリシー属性を同じファイルで設定する場合です。

たとえば、例21.5「rmManager Spring Bean で RMAssertion を使用して WS-RM 属性を設定する」 の設定では次を示します。

- **rmManager** Spring Bean 内で **RMAssertion** を使用して設定された標準の WS-RM ポリシー属性である **BaseRetransmissionInterval**。
- 同じ設定ファイルで設定された Apache CXF 固有の RM 属性 **intraMessageThreshold**。

例21.5 rmManager Spring Bean で RMAssertion を使用して WS-RM 属性を設定する

```
<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>
```

機能内のポリシー

例21.6「機能内のポリシーとしての WS-RM 属性の設定」 に示すように、機能内で標準の WS-RM ポリシー属性を設定できます。

例21.6 機能内のポリシーとしての WS-RM 属性の設定

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

WS-Policy フレームワークを使用して WS-RM を有効にする場合は、WSDL ファイルで標準の WS-RM ポリシー属性を設定できます。これは、サービスを相互運用して、他のポリシー対応 Web サービススタックにデプロイされたコンシューマーとシームレスに WS-RM を使用する場合に適したアプローチです。

例については、「WS-Policy フレームワーク。暗黙的にインターセプターを追加する」を参照してください。ここで、基本再送信間隔は WSDL ファイルで設定されます。

外部アタッチメント

標準の WS-RM ポリシー属性を外部の添付ファイルで設定できます。これは、WSDL ファイルを変更できない、または変更したくない場合に適したアプローチです。

例21.7「外部アタッチメントでの WS-RM の設定」は、特定の EPR に対して WS-A と WS-RM の両方 (基本再送信間隔 30 秒) を有効にする外部接続を示しています。

例21.7 外部アタッチメントでの WS-RM の設定

```
<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 設定のユースケース

概要

このサブセクションでは、ユースケースの観点から WS-RM 属性を設定することに焦点を当てます。属性が http://schemas.xmlsoap.org/ws/2005/02/rm/policy/ namespace で定義された標準の WS-RM ポリシー属性である場合、**rmManager** Spring Bean 内の **RMAssertion** で設定する例のみが表示されます。機能内のポリシーなどの属性を設定する方法の詳細については、WSDL ファイルまたは外部添付ファイルで、「標準の WS-RM ポリシー属性の設定」を参照してください。

次のユースケースについて説明します。

- 「基本再送間隔」
- 「再送信の指数バックオフ」
- 「確認間隔」
- 「未確認メッセージの最大しきい値」
- 「RM シーケンスの最大長」

- 「メッセージ配信保証ポリシー」

基本再送間隔

BaseRetransmissionInterval 要素は、まだ確認されていないメッセージを RM ソースが再送信する間隔を指定します。これは、http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd スキーマファイルで定義されています。デフォルト値は 3000 ミリ秒です。

例21.8「WS-RM ベースの再送間隔の設定」 WS-RM ベースの再送間隔を設定する方法を示します。

例21.8 WS-RM ベースの再送間隔の設定

```
<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>
```

再送信の指数バックオフ

ExponentialBackoff 要素は、承認されていないメッセージの連続する再送信試行が指数関数間隔で実行されるかどうかを判断します。

ExponentialBackoff 要素が存在すると、この機能が有効になります。デフォルトで **2** の指数バックオフ比率が使用されます。**ExponentialBackoff** はフラグです。要素が存在する場合、指数バックオフが有効になります。要素が存在しない場合、指数バックオフは無効になります。値は必要ありません。

例21.9「WS-RM 指数バックオフプロパティーの設定」 は、再送信のために WS-RM 指数バックオフを設定する方法を示しています。

例21.9 WS-RM 指数バックオフプロパティーの設定

```
<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>
```

確認間隔

AcknowledgementInterval 要素は、WS-RM 宛先が非同期確認応答を送信する間隔を指定します。これらは、着信メッセージの受信時に送信する同期確認応答に追加されます。デフォルトの非同期確認間隔は **0** ミリ秒です。つまり、**AcknowledgementInterval** が特定の値に設定されていない場合、確認応答は即座に送信されます (つまり、利用可能な最初の機会)。

非同期確認応答は、次の両方の条件が満たされた場合にのみ RM 宛先によって送信されます。

- RM 宛先は匿名以外の **wsrm:acksTo** エンドポイントを使用しています。
- 応答メッセージに確認応答をピギーバックする機会は、確認応答間隔が満了する前には発生しません。

例21.10「WS-RM 確認応答間隔の設定」 WS-RM 確認応答間隔を設定する方法を示します。

例21.10 WS-RM 確認応答間隔の設定

```
<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>
```

未確認メッセージの最大しきい値

maxUnacknowledged 属性は、シーケンスが終了する前に、シーケンスごとにアクセッジできる承認されていないメッセージの最大数を設定します。

例21.11「WS-RM の未確認メッセージの最大しきい値の設定」 は、WS-RM の未確認メッセージの最大しきい値を設定する方法を示しています。

例21.11 WS-RM の未確認メッセージの最大しきい値の設定

```
<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 シーケンスの最大長

maxLength 属性は、WS-RM シーケンスの最大長を設定します。デフォルト値は **0** で、WS-RM シーケンスの長さがバインドされないことを意味します。

この属性が設定されている場合、RM エンドポイントは、制限に達したとき、および以前に送信されたメッセージのすべての確認応答を受信した後に、新しい RM シーケンスを作成します。新しいメッセージは、newsequence を使用して送信されます。

例21.12「WS-RM メッセージシーケンスの最大長の設定」 RM シーケンスの最大長を設定する方法を示します。

例21.12 WS-RM メッセージシーケンスの最大長の設定

```
<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>
```

メッセージ配信保証ポリシー

次の配信保証ポリシーを使用するように RM 宛先を設定できます。

- **AtMostOnce**: RM 宛先はメッセージを 1 度のみアプリケーションの宛先に配信します。メッセージが複数回配信されると、エラーが発生します。シーケンス内の一部のメッセージが配信されない可能性があります。
- **AtLeastOnce**: RM 宛先は少なくとも 1 回アプリケーションの宛先にメッセージを配信します。送信されたすべてのメッセージが配信されるか、エラーが発生します。一部のメッセージは複数回配信される場合があります。
- **InOrder**: RM 宛先は、送信順にアプリケーションの宛先にメッセージを配信します。この配信保証は、**AtMostOnce** または **AtLeastOnce** 保証と組み合わせることができます。

例21.13「WS-RM メッセージ配信保証ポリシーの設定」 WS-RM メッセージ配信保証を設定する方法を示します。

例21.13 WS-RM メッセージ配信保証ポリシーの設定

```
<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. WS-RM 永続性の設定

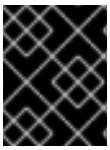
概要

この章ですでに説明した Apache CXF -RM 機能は、ネットワーク障害などの場合に信頼性を提供します。WS-RM の永続性は、RM ソースや RM 宛先のクラッシュなどの他のタイプの障害全体で信頼性を提供します。

WS-RM の永続性には、さまざまな RM エンドポイントの状態を永続ストレージに保存することが含まれます。これにより、エンドポイントは、メッセージが生まれ変わったときにメッセージの送受信を継続できます。

Apache CXF は、設定ファイルで WS-RM の永続性を有効にします。デフォルトの WS-RM 永続ストア

は JDBC ベースです。便宜上、Apache CXF には、すぐに使用できるデプロイメント用の Derby が含まれています。さらに、永続ストアも Java API を使用して公開されます。独自の永続化メカニズムを実装するには、この API と優先 DB を使用して永続化メカニズムを実装できます。



重要

WS-RM の永続性は一方向の通話でのみサポートされており、デフォルトでは無効になっています。

仕組み

Apache CXF WS-RM の永続性は次のように機能します。

- RM 送信元エンドポイントでは、送信メッセージは送信前に保持されます。確認応答を受信した後、永続ストアから削除されます。
- クラッシュからの回復後、永続化されたメッセージを回復し、すべてのメッセージが確認されるまで再送信します。その時点で、RM シーケンスは閉じられます。
- RM 宛先エンドポイントでは、着信メッセージが永続化され、ストアが成功すると、確認応答が送信されます。メッセージが正常にディスパッチされると、永続ストアから削除されます。
- クラッシュからの回復後、永続化されたメッセージを回復し、それらをディスパッチします。また、RM シーケンスを、新しいメッセージが受け入れられ、確認され、配信される状態にします。

WS- 永続性の有効化

WS-RM の永続性を有効にするには、WS-RM の永続ストアを実装するオブジェクトを指定する必要があります。独自に開発することも、Apache CXF に付属の JDBC ベースのストアを使用することもできます。

例21.14「デフォルトの WS-RM 永続ストアの設定」 に示す設定 Apache CXF に付属する JDBC ベースのストアを有効にします。

例21.14 デフォルトの WS-RM 永続ストアの設定

```
<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>
```

WS- 永続性の設定

Apache CXF に付属する JDBC ベースのストアは、表21.4「JDBC ストアのプロパティー」 に示すプロパティーをサポートします。

表21.4 JDBC ストアのプロパティー

属性名	型	デフォルト設定

属性名	型	デフォルト設定
driverClassName	**String**	**org.apache.derby.jdbc.EmbeddedDriver**
userName	**String**	null
passWord	**String**	null
url	**String**	jdbc:derby:rmdb;create=true

例21.15「WS-RM 永続性のための JDBC ストアの設定」 に示されている設定は、Apache CXF に同梱される JDBC ベースのストアを有効にし、**driverClassName** および **url** をデフォルト以外の値に設定します。

例21.15 WS-RM 永続性のための JDBC ストアの設定

```
<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 高可用性の有効化

概要

この章では、Apache CXF ランタイムでハイアベイラビリティを有効にして設定する方法について説明します。

22.1. 高可用性

概要

スケーラブルで信頼性の高いアプリケーションには、分散システムの単一障害点を回避するための高可用性が必要です。システムの単一障害点から、 **複製されたサービス**

レプリケートされたサービスは、同じサービスの複数のインスタンスまたは **レプリカ** で設定されます。これらが一緒になって、単一の論理サービスとして機能します。クライアントはレプリケートされたサービスでリクエストを呼び出し、Apache CXF はリクエストをメンバーレプリカの 1 つに配信します。レプリカへのルーティングは、クライアントに対して透過的です。

静的フェイルオーバーを備えた HA

Apache CXF は、レプリカの詳細がサービスの WSDL ファイルにエンコードされる静的フェイルオーバーを備えた高可用性 (HA) をサポートします。WSDL ファイルには複数のポートが含まれており、同じサービスに対して複数のホストを含めることができます。クラスター内のレプリカの数は、WSDL ファイルが変更されていない限り静的なままです。クラスターサイズを変更するには、WSDL ファイルを編集する必要があります。

22.2. 静的フェイルオーバーによる HA の有効化

概要

静的フェイルオーバーで HA を有効にするには、次の手順を実行する必要があります。

1. 「レプリカの詳細をサービス WSDL ファイルにエンコードします」
2. 「クラスタリング機能をクライアント設定に追加します」

レプリカの詳細をサービス WSDL ファイルにエンコードします

クラスター内のレプリカの詳細をサービス WSDL ファイルにエンコードする必要があります。例 22.1「静的フェイルオーバーによる HA の有効化。WSDL ファイル」 は、3 つのレプリカのサービスクラスターを定義する WSDL ファイル抽出を示しています。

例22.1 静的フェイルオーバーによる HA の有効化。WSDL ファイル

```
<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「静的フェイルオーバーによる HA の有効化。WSDL ファイル」 に示す WSDL 抽出は次のように説明することができます。

3 つのポートで公開されるサービス **ClusterService** を定義します。

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

Replica1 を定義して、**ClusterService** をポート **9001** で SOAP over HTTP エンドポイントとして公開します。

Replica2 を定義して、**ClusterService** をポート **9002** で SOAP over HTTP エンドポイントとして公開します。

Replica3 を定義して、**ClusterService** をポート **9003** で SOAP over HTTP エンドポイントとして公開します。

クラスタリング機能をクライアント設定に追加します

クライアント設定ファイルに、例22.2「静的フェイルオーバーによる HA の有効化。クライアント設定」 に示すようにクラスタリング機能を追加します。

例22.2 静的フェイルオーバーによる HA の有効化。クライアント設定

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       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. 静的フェイルオーバーを使用した HA の設定

概要

デフォルトでは、静的フェイルオーバーを使用する HA は、クライアントが通信している元のサービスが使用できなくなった場合、または失敗した場合に、レプリカサービスを選択するときにシーケンシャル戦略を使用します。シーケンシャルストラテジーは、使用されるたびに同じシーケンシャル順序でレプリカサービスを選択します。選択は Apache CXF の内部サービスモデルによって決定され、決定論的なフェイルオーバーパターンになります。

ランダム戦略の設定

レプリカを選択するときにシーケンシャル戦略の代わりにランダム戦略を使用するように静的フェイルオーバーを使用して HA を設定できます。ランダム戦略では、サービスが使用できなくなるか失敗するたびに、ランダムレプリカサービスが選択されます。クラスター内の存続メンバーからのフェイルオーバーターゲットの選択は完全にランダムです。

ランダム戦略を設定するには、クライアント設定ファイルに 例22.3「静的フェイルオーバーのランダム戦略の設定」 に示す設定を追加します。

例22.3 静的フェイルオーバーのランダム戦略の設定

```
<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「静的フェイルオーバーのランダム戦略の設定」 に示す設定は次のように説明することができます。

ランダムストラテジーを実装する **Random** bean および実装クラスを定義します。

レプリカを選択するときにランダム戦略が使用されることを指定します。

第23章 APACHE CXF バインディング ID

バインディング ID の表

表23.1 メッセージバインディングのバインディング ID

バインディング	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**

付録A MAVEN OSGI ツールの使用

概要

大規模なプロジェクトのバンドルまたはバンドルのコレクションを手動で作成するのは面倒な場合があります。Maven バンドルプラグインは、プロセスを自動化し、バンドルマニフェストのコンテンツを指定するためのいくつかのショートカットを提供することにより、作業を容易にします。

A.1. MAVEN バンドルプラグイン

Red Hat Fuse OSGi ツールは、Apache Felix の Maven バンドルプラグイン を使用します。バンドルプラグインは、Peter Kriens の bnd ツールに基づいています。バンドルにパッケージ化されているクラスの内容をイントロスペクトすることにより、OSGi バンドルマニフェストの構築を自動化します。バンドルに含まれるクラスの知識を使用すると、プラグインは適切な値を計算して、バンドルマニフェストの **Import-Packages** および **Export-Package** プロパティーにデータを投入することができます。プラグインには、バンドルマニフェストの他の必須プロパティーに使用されるデフォルト値もあります。

バンドルプラグインを使用するには、次の手順を実行します。

1. 「Red Hat FuseOSGi プロジェクトのセットアップ」 プロジェクトの POM ファイルへのバンドルプラグイン。
2. 「バンドルプラグインの設定」 バンドルのマニフェストを正しく設定するためのプラグイン。

A.2. RED HAT FUSEOSGI プロジェクトのセットアップ

概要

OSGi バンドルを構築するための Maven プロジェクトは、単純な単一レベルのプロジェクトにすることができます。サブプロジェクトは必要ありません。ただし、次のことを行う必要があります。

1. バンドルプラグインの POM への 追加
2. 結果を OSGi バンドルとしてパッケージ化するように Maven に 指示 します。

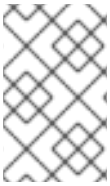
注記

適切な設定でプロジェクトをセットアップするために使用できる Maven アーキタイプがいくつかあります。

ディレクトリー構造

OSGi バンドルを構築するプロジェクトは、単一レベルのプロジェクトにすることができます。トップレベル POM ファイルと **src** フォルダーのみが必要になります。すべての Maven プロジェクトと同様に、すべての Java ソースコードを **src/java** フォルダーに配置し、Java 以外のリソースを **src/resources** フォルダーに配置します。

Java 以外のリソースには、Spring 設定ファイル、JBI エンドポイント設定ファイル、および WSDL コントラクトが含まれます。



注記

Apache CXF、Apache Camel、または別の Spring が設定されている Bean を使用する Red Hat Fuse OSGi プロジェクトには、**src/resources/META-INF/spring** フォルダーに **beans.xml** ファイルも含まれます。

バンドルプラグインの追加

バンドルプラグインを使用する前に、ApacheFelix への依存関係を追加する必要があります。依存関係を追加した後、バンドルプラグインを POM のプラグイン部分に追加できます。

例A.1「OSGi バンドルプラグインを POM に追加する」 は、バンドルプラグインをプロジェクトに追加するために必要な POM エントリーを示しています。

例A.1 OSGi バンドルプラグインを POM に追加する

```
...
<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「OSGi バンドルプラグインを POM に追加する」 内のエントリーは次を行います。

Apache Felix への依存関係を追加します

バンドルプラグインをプロジェクトに追加します

プロジェクトのアーティファクト ID をバンドルのシンボリック名として使用するようにプラグインを設定します

バンドルされたクラスによってインポートされたすべての Java パッケージを含めるようにプラグインを設定する。また、**org.apache.camel.osgi** パッケージをインポートする

リストされたクラスをバンドルするようにプラグインを設定しますが、エクスポートされたパッケージのリストにはそれらを含めません

注記

プロジェクトの要件を満たすように設定を編集します。

バンドルプラグインの設定の詳細には、「バンドルプラグインの設定」 を参照してください。

バンドルプラグインのアクティブ化

Maven にバンドルプラグインを使用させるには、プロジェクトの結果をバンドルとしてパッケージ化するように Maven に指示します。そのためには、POM ファイルの **packaging** 要素を **bundle** に設定します。

便利な Maven アーキタイプ

バンドルプラグインを使用するように事前設定されたプロジェクトを生成するために使用できる Maven アーキタイプがいくつかあります。

- 「Spring OSGi アーキタイプ」
- 「Apache CXF コードファーストの原型」
- 「Apache CXF wsdl - 原型アーキタイプ」
- 「Apache Camel アーキタイプ」

Spring OSGi アーキタイプ

Spring OSGi アーキタイプは、次のように、Spring DM を使用して OSGi プロジェクトを構築するための汎用プロジェクトを作成します。

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

次のコマンドを使用してアーキタイプを呼び出します。

```
mvn archetype:generate -DarchetypeGroupId=org.springframework.osgi -
DarchetypeArtifactId=spring-osgi-bundle-archetype -DarchetypeVersion=1.1.2 -DgroupId=groupId -
DartifactId=artifactId -Dversion=version
```

Apache CXF コードファーストの原型

次に示すように、Apache CXF コードファーストアーキタイプは、Java からサービスを構築するためのプロジェクトを作成します。

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2010.02.0-fuse-02-00
```

次のコマンドを使用してアーキタイプを呼び出します。

```
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 - 原型アーキタイプ

次に示すように、Apache CXF wsdl-first アーキタイプは、WSDL からサービスを作成するためのプロジェクトを作成します。

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2010.02.0-fuse-02-00
```

次のコマンドを使用してアーキタイプを呼び出します。

```
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 アーキタイプ

次に示すように、Apache Camel アーキタイプは、Red Hat Fuse にデプロイされるルートを構築するためのプロジェクトを作成します。

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2010.02.0-fuse-02-00
```

次のコマンドを使用してアーキタイプを呼び出します。

```
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. バンドルプラグインの設定

概要

バンドルプラグインを機能させるために必要な情報はほぼありません。必要なすべてのプロパティーは、デフォルト設定を使用して有効な OSGi バンドルを生成します。

デフォルト値のみを使用して有効なバンドルを作成できますが、値の一部を変更することを推奨します。プラグインの **instructions** 要素内のほとんどのプロパティーを指定できます。

設定プロパティー

一般的に使用される設定プロパティーのいくつかは次のとおりです。

- Bundle-SymbolicName
- Bundle-Name
- Bundle-Version
- Export-Package

- Private-Package
- Import-Package

バンドルのシンボリック名の設定

デフォルトでは、バンドルプラグインは **Bundle-SymbolicName** プロパティーの値を groupId + "." + artifactId に設定します。ただし、以下の例外があります。

- **groupId** にセクションが 1 つしかない (ドットがない) 場合には、クラスを含む最初のパッケージ名が返されます。
  たとえば、グループ ID が **commons-logging:commons-logging** の場合、バンドルのシンボリック名は **org.apache.commons.logging** です。
- **ArtifactId** が **groupId** の最後のセクションと同じ場合には、**groupId** が使用されます。
  たとえば、POM がグループ ID およびアーティファクト ID を **org.apache.maven:maven** として指定すると、バンドルのシンボリック名は **org.apache.maven** になります。
- **ArtifactId** が **groupId** の最後のセクションで始まる場合には、その部分は削除されます。
  たとえば、POM がグループ ID およびアーティファクト ID を **org.apache.maven:maven-core** として指定すると、バンドルのシンボリック名は **org.apache.maven.core** になります。

バンドルのシンボリック名に独自の値を指定するには、例A.2「バンドルのシンボリック名の設定」に示すように、プラグインの **instructions** 要素に **Bundle-SymbolicName** の子を追加します。

例A.2 バンドルのシンボリック名の設定

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

バンドル名の設定

デフォルトでは、バンドルの名前は **${project.name}** に設定されます。

バンドルの名前に独自の値を指定するには、例A.3「バンドル名の設定」に示すように、プラグインの **instructions** 要素に **Bundle-Name** の子を追加します。

例A.3 バンドル名の設定

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Name>JoeFred</Bundle-Name>
```

```
      ...
    </instructions>
  </configuration>
</plugin>
```

バンドルのバージョンの設定

デフォルトでは、バンドルのバージョンは **${project.version}** に設定されます。ダッシュ (**-**) はピリオド (**.**) に置き換えられ、数字は 4 桁に変換されます。たとえば、**4.2-SNAPSHOT** は **4.2.0.SNAPSHOT** になります。

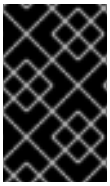
バンドルのバージョンに独自の値を指定するには、例A.4「バンドルのバージョンの設定」に示すように、プラグインの **instructions** 要素に **Bundle-Version** の子を追加します。

例A.4 バンドルのバージョンの設定

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

エクスポートされたパッケージの指定

デフォルトでは、OSGi マニフェストの **Export-Package** リストには、ローカルの Java ソースコード (**src/main/java** 下) のすべてのパッケージが反映されます。ただし、デフォルトのパッケージ、**.**、および **.impl** または **.internal** が含まれるすべてのパッケージを **除きます**。



重要

プラグイン設定で **Private-Package** 要素を使用し、エクスポートするパッケージのリストを指定しない場合、デフォルトの動作ではバンドルの **Private-Package** 要素にリストされているパッケージのみが含まれます。パッケージはエクスポートされません。

デフォルトの動作では、パッケージが非常に大きくなり、非公開にしておく必要のあるパッケージがエクスポートされる可能性があります。エクスポートされるパッケージのリストを変更するには、**Export-Package** の子をプラグインの **instructions** 要素に追加します。

Export-Package 要素は、バンドルに含まれるパッケージとエクスポートされるパッケージのリストを指定します。パッケージ名は、* ワイルドカードシンボルを使用して指定できます。たとえば、エントリー **com.fuse.demo.*** は、プロジェクトのクラスパスにある **com.fuse.demo** で始まるすべてのパッケージが含まれます。

除外するパッケージを指定するには、エントリーに **!** の接頭辞を追加します。たとえば、エントリー **!com.fuse.demo.private** は、パッケージ **com.fuse.demo.private** を除外します。

パッケージを除外する場合に、リスト内のエントリーの順序が重要です。リストは最初から順番に処理され、それ以降の矛盾するエントリーは無視されます。

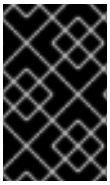
たとえば、パッケージ **com.fuse.demo.private** を除く **com.fuse.demo** で始まるすべてのパッケージを含めるには、以下のようにパッケージのリストを指定します。

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

ただし、**com.fuse.demo.*,!com.fuse.demo.private** を使用してパッケージをリスト表示すると、**com.fuse.demo.private** は最初のパターンと一致するため、バンドルに含まれます。

プライベートパッケージの指定

バンドルに追加するパッケージのリストを指定してそれらをエクスポート **しない** 場合、バンドルプラグイン設定に **Private-Package** 命令を追加できます。デフォルトでは、**Private-Package** 命令を指定しないと、ローカルの Java ソースのすべてのパッケージがバンドルに含まれます。



重要

パッケージが **Private-Package** 要素と **Export-Package** 要素の両方のエントリーと一致する場合は、**Export-Package** 要素が優先されます。パッケージがバンドルに追加され、エクスポートされます。

Private-Package 要素は、バンドルに含まれるパッケージのリストを指定する点で **Export-Package** 要素と同様に機能します。バンドルプラグインは、リストを使用して、バンドルに含まれるプロジェクトのクラスパスにあるすべてのクラスを検索します。これらのパッケージはバンドルにパッケージ化されますが、(**Export-Package** 命令で選択されない限り) エクスポートされません。

例A.5「プライベートパッケージのバンドルへの追加」 はバンドルにプライベートパッケージを含めるための設定です。

例A.5 プライベートパッケージのバンドルへの追加

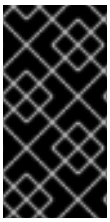
```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

インポートされたパッケージの指定

デフォルトでは、バンドルプラグインは OSGi マニフェストの **Import-Package** プロパティーに、バンドルのコンテンツによって参照されるすべてのパッケージのリストを反映させます。

ほとんどのプロジェクトでは通常、デフォルトの動作で十分ですが、リストに自動的に追加されないパッケージをインポートする場合があります。デフォルトの動作では、不要なパッケージがインポートされる可能性もあります。

バンドルによってインポートされるパッケージのリストを指定するには、**Import-Package** の子をプラグインの **instructions** 要素に追加します。パッケージリストの構文は、**Export-Package** 要素および **Private-Package** 要素の場合と同じです。



重要

Import-Package 要素を使用する場合、必要なインポートの有無を判断するのにプラグインはバンドルの内容を自動的にスキャンしません。バンドルの内容がスキャンされるようにするには、パッケージリストの最後のエントリーとして * を配置する必要があります。

例A.6「バンドルでインポートされたパッケージの指定」 は、バンドルでインポートされたパッケージを指定する設定です。

例A.6 バンドルでインポートされたパッケージの指定

```
<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>
```

補足情報

バンドルプラグインの設定の詳細には、以下を参照してください。

- olink:OsgiDependencies/OsgiDependencies
- Apache Felix のドキュメント
- Peter Kriens の aQute Software Consultancy Web サイト

パート V. JAX-WS を使用したアプリケーションの開発

このガイドでは、標準の JAX-WS API を使用して Web サービスを開発する方法について説明します。

第24章 ボトムアップサービス開発

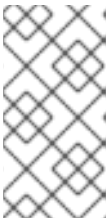
概要

サービス指向アプリケーションの一部として公開したい一連の機能をすでに実装している Java コードがある場合が多くあります。また、WSDL を使用してインターフェイスを定義することは避けたい場合もあります。JAX-WS アノテーションを使用して、Java クラスのサービスを有効にするために必要な情報を追加できます。WSDL コントラクトの代わりに使用できる **サービスエンドポイントインターフェイス** (SEI) を作成することもできます。WSDL コントラクトが必要な場合、Apache CXF には、アノテーション付きの Java コードからコントラクトを生成するためのツールが用意されています。

24.1. JAX-WS サービス開発の概要

Java から開始してサービスを作成するには、次のことを行う必要があります。

1. 「SEI の作成」 サービスとして公開するメソッドを定義するサービスエンドポイントインターフェイス (SEI)。



注記

Java クラスから直接作業することもできますが、インターフェイスから作業することを推奨します。インターフェイスは、サービスを使用するアプリケーションの開発を担当する開発者と共有するのに適しています。インターフェイスは小さく、サービスの実装の詳細は提供されません。

2. 「コードにアノテーションを付ける」 コードに必要なアノテーション。
3. 「WSDL の生成」 サービスの WSDL コントラクト。

注記

SEI をサービスのコントラクトとして使用する場合は、WSDL コントラクトを生成する必要はありません。

4. 31章 *サービスの公開* サービスプロバイダーとしてのサービス。

24.2. SEI の作成

概要

サービスエンドポイントインターフェイス (SEI) は、サービス実装とそのサービスで要求を行うコンシューマーの間で共有される Java コードの一部です。SEI は、サービスによって実装されるメソッドを定義し、サービスがエンドポイントとして公開される方法に関する詳細を提供します。WSDL コントラクトを開始する場合、SEI はコードジェネレーターによって生成されます。ただし、Java から開始する場合は、SEI を作成するのは開発者の責任です。 SEI を作成するには、次の 2 つの基本的なパターンがあります。

- グリーンフィールド開発 – このパターンでは、既存の Java コードまたは WSDL を使用せずに新しいサービスを開発しています。SEI を作成することから始めるのが最善です。次に、SEI を使用するサービスプロバイダーとコンシューマーの実装を担当する開発者に SEI を配布できます。

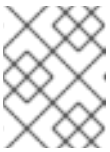
注記

グリーンフィールドサービス開発を行うための推奨される方法は、サービスとそのインターフェイスを定義する WSDL コントラクトを作成することから始めることです。26章 *開始点の WSDL コントラクト*を参照してください。

- サービスの有効化–このパターンでは、通常、Java クラスとして実装されている既存の機能セットがあり、サービスを有効にします。これは、次の 2 つのことを行う必要があることを意味します。

  a. サービスの一部として公開される予定の操作 **のみ** を含む SEI を作成します。

  b. SEI を実装するように、既存の Java クラスを変更します。



注記

JAX-WS アノテーションを Java クラスに追加することはできますが、推奨しません。

インターフェイスの作成

SEI は標準の Java インターフェイスです。クラスが実装する一連のメソッドを定義します。また、実装クラスがアクセスできるいくつかのメンバーフィールドと定数を定義することもできます。

SEI の場合、定義されたメソッドは、サービスによって公開される操作にマップされることを目的としています。SEI は **wsdl:portType** 要素に対応します。SEI で定義されたメソッドは、**wsdl:portType** 要素の **wsdl:operation** 要素に対応します。

注記

JAX-WS は、サービスの一部として公開されていないメソッドを指定できるようにするアノテーションを定義します。ただし、ベストプラクティスは、これらのメソッドを SEI から除外することです。

例24.1「シンプルな SEI」 は、株式更新サービスの簡単な SEI を示しています。

例24.1 シンプルな SEI

```
package com.fusesource.demo;

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

インターフェイスの実装

SEI は標準の Java インターフェイスであるため、SEI を実装するクラスは標準の Java クラスです。Java クラスから始める場合は、インターフェイスを実装するために Java クラスを変更する必要があります。SEI から始める場合、実装クラスは SEI を実装します。

例24.2「単純な実装クラス」 にインターフェイスを実装するためのクラスを示します例24.1「シンプルな SEI」。

例24.2 単純な実装クラス

```
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. コードにアノテーションを付ける

24.3.1. JAX-WS アノテーションの概要

JAX-WS アノテーションは、SEI を完全に指定されたサービス定義にマップするために使用されるメタデータを指定します。アノテーションで提供される情報には、次のものがあります。

- サービスのターゲット名前空間。
- リクエストメッセージを保持するために使用されるクラスの名前
- 応答メッセージを保持するために使用されるクラスの名前
- 操作が一方向操作の場合
- サービスが使用するバインディングスタイル
- カスタム例外に使用されるクラスの名前
- サービスによって使用されるタイプが定義される名前空間

注記

ほとんどのアノテーションには適切なデフォルトがあり、それらに値を指定する必要はありません。ただし、アノテーションで提供する情報が多いほど、サービス定義がより適切に指定されます。明確に指定されたサービス定義は、分散アプリケーションのすべての部分が連携する可能性を高めます。

24.3.2. 必要なアノテーション

概要

Java コードからサービスを作成するには、コードに 1 つのアノテーションを追加するだけで済みます。**@WebService** アノテーションを SEI と実装クラスの両方に追加する必要があります。

@WebService アノテーション

@WebService アノテーションは javax.jws.WebService インターフェイスによって定義され、サービスとして使用されるインターフェイスまたはクラスに配置されます。**@WebService** には、表 24.1「**@WebService** プロパティー」 で説明されているプロパティーがあります。

表24.1@WebService プロパティー

プロパティー	説明
name	サービスインターフェイスの名前を指定します。このプロパティーは、WSDL コントラクトのサービスのインターフェイスを定義する **wsdl:portType** 要素の **name** 属性にマッピングされます。デフォルトでは、**PortType** を実装クラスの名前に追加します。[a]
targetNamespace	サービスが定義されているターゲット名前空間を指定します。このプロパティーが指定されていない場合、ターゲット名前空間はパッケージ名から派生します。
serviceName	公開されたサービスの名前を指定します。このプロパティーは、公開されたサービスを定義する **wsdl:service** 要素の **name** 属性にマッピングされます。デフォルトでは、サービスの実装クラスの名前を使用します。
wsdlLocation	サービスの WSDL コントラクトが保存されている URL を指定します。これは、相対 URL を使用して指定する必要があります。デフォルトは、サービスがデプロイされている URL です。
endpointInterface	実装クラスが実装する SEI のフルネームを指定します。このプロパティーは、属性がサービス実装クラスで使用される場合にのみ指定されます。
portName	サービスが公開されるエンドポイントの名前を指定します。このプロパティーは、公開されたサービスのエンドポイントの詳細を指定する **wsdl:port** 要素の **name** 属性にマッピングされます。デフォルトは、サービスの実装クラスの名前に **Port** を追加します。

[a] SEI から WSDL を生成する場合、実装クラスの名前の代わりにインターフェイスの名前が使用されます。

注記

@WebService アノテーションのプロパティーに値を指定する必要はありません。ただし、できるだけ多くの情報を提供することを推奨します。

SEI にアノテーションを付ける

SEI では、**@WebService** アノテーションを追加する必要があります。SEI はサービスを定義するコントラクトなので、**@WebService** アノテーションのプロパティーにサービスの詳細をできるだけ多く指定する必要があります。

例24.3「**@WebService** アノテーションあるインターフェイス」は、**@WebService** アノテーションにより 例24.1「シンプルな SEI」 で定義したインターフェイスを示しています。

例24.3 @WebService アノテーションあるインターフェイス

```
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「**@WebService** アノテーションあるインターフェイス」 の **@WebService** アノテーションは以下を行います。

サービスインターフェイスを定義する **wsdl:portType** 要素の **name** 属性の値が **quoteUpdater** であることを指定します。

サービスのターゲット名前空間が http://demos.redhat.com であることを指定します。

公開されたサービスを定義する **wsdl:service** 要素の **name** の値が **updateQuoteService** であることを指定します。

サービスが WSDL コントラクトを http://demos.redhat.com/quoteExampleService?wsdl で公開することを指定します。

サービスを公開するエンドポイントを定義する **wsdl:port** 要素の **name** 属性の値が **updateQuotePort** であることを指定します。

サービス実装にアノテーションを付ける

また、SEI に **@WebService** アノテーションを付ける必要があります。また、サービス実装クラスに **@WebService** アノテーションを付ける必要があります。アノテーションをサービス実装クラスに追加する場合は、**endpointInterface** プロパティーのみを指定する必要があります。例24.4「アノテーション付きサービス実装クラス」 に示すようにプロパティーは、SEI のフルネームに設定する必要があります。

例24.4 アノテーション付きサービス実装クラス

```
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. 任意のアノテーション

概要

Java インターフェイスまたは Java クラスを有効にするサービスには **@WebService** アノテーションで十分ですが、サービスがサービスプロバイダーとして公開される方法を完全に説明しません。JAX-WS プログラミングモデルは、使用するバインディングなど、サービスに関する詳細を Java コードに追加するために、いくつかのオプションのアノテーションを使用します。これらのアノテーションをサービスの SEI に追加します。

SEI で提供する詳細が多いほど、開発者は、SEI が定義する機能を使用できるアプリケーションを実装しやすくなります。また、ツールによって生成された WSDL ドキュメントをより具体的にします。

概要

アノテーションを使用したバインディングプロパティーの定義

サービスに SOAP バインディングを使用している場合は、JAX-WS アノテーションを使用していくつかのバインディングプロパティーを指定できます。これらのプロパティーは、サービスの WSDL コントラクトで指定できるプロパティーに直接対応しています。パラメータースタイルなどの一部の設定では、メソッドの実装方法が制限される場合があります。これらの設定は、メソッドパラメーターにアノテーションを付けるときに使用できるアノテーションにも影響を与える可能性があります。

@SOAPBinding アノテーション

@SOAPBinding アノテーションは javax.jws.soap.SOAPBinding インターフェイスによって定義されます。デプロイ時にサービスによって使用される SOAP バインディングに関する詳細を提供します。**@SOAPBinding** アノテーションが指定されていない場合、サービスはラップされた doc/literal SOAP バインディングを使用して公開されます。

@SOAPBinding アノテーションを SEI および SEI の任意のメソッドに追加できます。メソッドで使用される場合は、メソッドの **@SOAPBinding** アノテーションの設定が優先されます。

表24.2「**@SOAPBinding プロパティー**」 **@SOAPBinding** アノテーションのプロパティーを表示します。

表24.2@SOAPBinding プロパティー

プロパティー	値	説明
style	Style.DOCUMENT (デフォルト) Style.RPC	SOAP メッセージのスタイルを指定します。RPC スタイルが指定されている場合、SOAP ボディーの各メッセージ部分はパラメーターまたは戻り値で、**soap:body** 要素内のラッパー要素に表示されます。ラッパー要素内のメッセージ部分は操作パラメーターに対応し、操作のパラメーターと同じ順序で表示される必要があります。DOCUMENT スタイルを指定する場合、SOAP 本体の内容は有効な XML ドキュメントである必要がありますが、その形式はそれほど厳密に制限されていません。
use	Use.LITERAL (デフォルト) Use.ENCODED[a]	SOAP メッセージのデータをストリーミングする方法を指定します。
parameterStyle[b]	ParameterStyle.BARE ParameterStyle.WRAPPED (デフォルト)	WSDL コントラクトのメッセージ部分に対応するメソッドパラメーターを SOAP メッセージ本文に配置する方法を指定します。BARE が指定されている場合、各パラメーターはメッセージルートの子要素としてメッセージ本文に配置されます。WRAPPED が指定されている場合、すべての入力パラメーターは要求メッセージで単一の要素にラップされ、すべての出力パラメーターは応答メッセージで単一の要素にラップされます。

[a] Use.ENCODED は現在サポートされていません。

[b] **style** を RPC に設定する場合は、WRAPPED パラメータースタイルを使用する必要があります。

ベアスタイルパラメーターを文書化する

ドキュメントベアスタイルは、Java コードと結果として得られるサービスの XML 表現との間の最も直接的なマッピングです。このスタイルを使用する場合、スキーマタイプは、操作のパラメーターリストで定義された入力パラメーターと出力パラメーターから直接生成されます。

ベア document\literal スタイルを使用するには、**@SOAPBinding** アノテーションの **style** プロパティーを Style.DOCUMENT に設定し、**parameterStyle** プロパティーを ParameterStyle.BARE に設定します。

ベアパラメーターを使用するときに操作がドキュメントスタイルの使用制限に違反しないようにするには、操作が次の条件に準拠している必要があります。

- 操作には、入力パラメーターまたは入出力パラメーターを 1 つだけ含める必要があります。
- 操作に **void** 以外の型がある場合、出力または入出力パラメーターを含めることはできません。
- 操作に **void** を返すタイプがある場合は、複数の出力または入出力パラメーターは必要ありません。

注記

@WebParam アノテーションまたは **@WebResult** アノテーションを使用して SOAP ヘッダーに配置されるパラメーターは、許可されるパラメーターの数にカウントされません。

ドキュメントでラップされたパラメーター

ドキュメントラップスタイルにより、Java コードと結果のサービスの XML 表現との間のマッピングのような RPC が可能になります。このスタイルを使用する場合、メソッドのパラメーターリスト内のパラメーターは、バインディングによって単一の要素にラップされます。これの欠点は、Java 実装とメッセージがネットワーク上に配置される方法との間に間接層が追加されることです。

ベア document\literal スタイルを使用するには、**@SOAPBinding** アノテーションの **style** プロパティーを Style.DOCUMENT に設定し、**parameterStyle** プロパティーを ParameterStyle.BARE に設定します。

「@RequestWrapper アノテーション」 アノテーションと「@ResponseWrapper アノテーション」 アノテーションを使用して、ラッパーの生成方法をある程度制御できます。

例

例24.5「SOAP バインディングアノテーションを使用したドキュメントベア SOAP バインディングの指定」 は、ドキュメントベア SOAP メッセージを使用する SEI を示しています。

例24.5 SOAP バインディングアノテーションを使用したドキュメントベア SOAP バインディングの指定

```
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
{
  ...
}
```

概要

アノテーションを使用した操作プロパティーの定義

ランタイムが Java メソッド定義を XML 操作定義にマップすると、次のような詳細が提供されます。

- 交換されたメッセージが XML でどのように表示されるか
- メッセージを一方向メッセージとして最適化できる場合
- メッセージが定義されている名前空間

@WebMethod アノテーション

@WebMethod アノテーションは javax.jws.WebMethod インターフェイスによって定義されます。これは、SEI のメソッドに配置されます。**@WebMethod** アノテーションは、通常、メソッドが関連付けられる操作を記述する **wsdl:operation** 要素で表される情報を提供します。

表24.3「**@WebMethod プロパティー**」 **@WebMethod** アノテーションのプロパティーを説明します。

表24.3 @WebMethod プロパティー

プロパティー	説明
operationName	関連する **wsdl:operation** 要素の **name** 値を指定します。デフォルト値はメソッドの名前です。
action	メソッド用に生成される **soap:operation** 要素の **soapAction** 属性の値を指定します。デフォルト値は空の文字列です。
exclude	メソッドをサービスインターフェイスから除外するかどうかを指定します。デフォルトは false です。

@RequestWrapper アノテーション

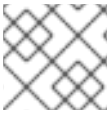
@RequestWrapper アノテーションは、javax.xml.ws.RequestWrapper インターフェイスによって定義されます。これは、SEI のメソッドに配置されます。**@RequestWrapper** アノテーションは、メッセージ変換を開始するリクエストメッセージのメソッドパラメーターのラッパー Bean を実装する Java クラスを指定します。また、リクエストメッセージをマーシャリングおよびアンマーシャリングするときにランタイムが使用する要素名と名前空間も指定します。

表24.4「**@RequestWrapper プロパティー**」 は **@RequestWrapper** アノテーションのプロパティーを説明します。

表24.4 @RequestWrapper プロパティー

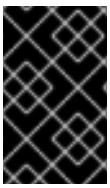
プロパティー	説明
localName	要求メッセージの XML 表現でラッパー要素のローカル名を指定します。デフォルト値はメソッドの名前または「@WebMethod アノテーション」 アノテーションの **operationName** プロパティーの値です。
targetNamespace	XML ラッパー要素が定義される名前空間を指定します。デフォルト値は、SEI のターゲット名前空間です。

プロパティー	説明
className	ラッパー要素を実装する Java クラスのフルネームを指定します。



注記

className プロパティーのみが必要になります。



重要

メソッドに **@SOAPBinding** アノテーションが付けられ、その **parameterStyle** プロパティーが **ParameterStyle.BARE** に設定されている場合、このアノテーションは無視されます。

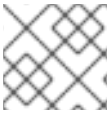
@ResponseWrapper アノテーション

@ResponseWrapper アノテーションは、javax.xml.ws.ResponseWrapper インターフェイスによって定義されます。これは、SEI のメソッドに配置されます。**@ResponseWrapper** は、メッセージ変換の応答メッセージのメソッドパラメーターのラッパー Bean を実装する Java クラスを指定します。また、応答メッセージをマーシャリングおよびアンマーシャリングするときにランタイムが使用する要素名と名前空間も指定します。

表24.5「**@ResponseWrapper** プロパティー」 は **@ResponseWrapper** アノテーションのプロパティーを説明します。

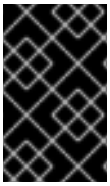
表24.5 @ResponseWrapper プロパティー

プロパティー	説明
localName	応答メッセージの XML 表現でラッパー要素のローカル名を指定します。デフォルト値は、Response が追加されたメソッドの名前、または Response が追加された「@WebMethod アノテーション」アノテーションの **operationName** プロパティーの値のいずれかです。
targetNamespace	XML ラッパー要素が定義されている名前空間を指定します。デフォルト値は、SEI のターゲット名前空間です。
className	ラッパー要素を実装する Java クラスのフルネームを指定します。



注記

className プロパティーのみが必要になります。



重要

メソッドに **@SOAPBinding** アノテーションが付けられ、その **parameterStyle** プロパティーが **ParameterStyle.BARE** に設定されている場合、このアノテーションは無視されます。

@WebFault アノテーション

@WebFault アノテーションは javax.xml.ws.WebFault インターフェイスによって定義されます。これは、SEI によって出力される例外に配置されます。**@WebFault** アノテーションを使用して、Java 例外を **wsdl:fault** 要素にマッピングします。この情報は、サービスとそのコンシューマーの両方が処理できる表現に例外をマーシャリングするために使用されます。

表24.6「**@WebFault プロパティー**」 **@WebFault** アノテーションのプロパティーを説明します。

表24.6 @WebFault プロパティー

プロパティー	説明
name	障害要素のローカル名を指定します。
targetNamespace	障害要素が定義されている名前空間を指定します。デフォルト値は、SEI のターゲット名前空間です。
faultName	例外を実装する Java クラスのフルネームを指定します。

重要

name プロパティーが必要です。

@Oneway アノテーション

@Oneway アノテーションは javax.jws.Oneway インターフェイスによって定義されます。これは、サービスからの応答を必要としない SEI のメソッドに配置されます。**@Oneway** アノテーションは、応答を待機せず、応答を処理するリソースを予約しないことで、メソッドの実行を最適化するようにランタイムに指示します。

このアノテーションは、次の条件を満たすメソッドでのみ使用できます。

- このページは **void** を返します。
- ホルダーインターフェイスを実装するパラメーターはありません
- コンシューマーに返すことができる例外を出力しません

例

例24.6「アノテーション付きメソッドを使用した SEI」 は、メソッドにアノテーションが付けられた SEI を示しています。

例24.6 アノテーション付きメソッドを使用した 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);
}
```

概要

アノテーションを使用したパラメータープロパティーの定義

SEI の method パラメーターは、**wsdl:message** 要素とその **wsdl:part** 要素に対応します。JAX-WS は、メソッドパラメーター用に生成された **wsdl:part** 要素を記述することのできるアノテーションを提供します。

@WebParam アノテーション

@WebParam アノテーションは、javax.jws.WebParam インターフェイスによって定義されます。これは、SEI で定義されたメソッドのパラメーターに配置されます。**@WebParam** アノテーションを使用すると、パラメーターの方向、パラメーターが SOAP ヘッダーに配置されるかどうか、および生成された **wsdl:part** の他のプロパティーを指定できます。

表24.7「**@WebParam** プロパティー」 **@WebParam** アノテーションのプロパティーを説明します。

表24.7 @WebParam プロパティー

プロパティー	値	説明
name		生成された WSDL ドキュメントに表示されるパラメーターの名前を指定します。RPC バインディングの場合、これはパラメーターを表す **wsdl:part** の名前です。ドキュメントバインディングの場合、これはパラメーターを表す XML 要素のローカル名です。JAX-WS 仕様によると、デフォルトは **argN** です。ここで、**N** はゼロベースの引数インデックスに置き換えてください (例: arg0、arg1 など)。

プロパティー	値	説明
targetNamespace		パラメーターの名前空間を指定します。これは、パラメーターが XML 要素にマップされるドキュメントバインディングでのみ使用されます。デフォルトでは、サービスの名前空間を使用します。
mode	Mode.IN (デフォルト)[a] Mode.OUT Mode.INOUT	パラメーターの方向を指定します。
header	false (デフォルト) true	パラメーターを SOAP ヘッダーの一部として渡すかどうかを指定します。
partName		パラメーターの **wsdl:part** 要素の **name** 属性の値を指定します。このプロパティーは、ドキュメントスタイルの SOAP バインディングに使用されます。

[a] ホルダーインターフェイスを実装するパラメーターはすべて、デフォルトで Mode.INOUT にマップされます。

@WebResult アノテーション

@WebResult アノテーションは、javax.jws.WebResult インターフェイスによって定義されます。これは、SEI で定義されたメソッドに配置されます。**@WebResult** アノテーションを使用すると、メソッドの戻り値に対して生成される **wsdl:part** のプロパティーを指定できます。

表24.8「**@WebResult** プロパティー」 **@WebResult** アノテーションのプロパティーを説明します。

表24.8 @WebResult プロパティー

プロパティー	説明
name	生成された WSDL ドキュメントに表示される戻り値の名前を指定します。RPC バインディングの場合、これは戻り値を表す **wsdl:part** の名前です。ドキュメントバインディングの場合、これは戻り値を表す XML 要素のローカル名です。デフォルト値は return です。
targetNamespace	戻り値の名前空間を指定します。これは、戻り値が XML 要素にマップされるドキュメントバインディングでのみ使用されます。デフォルトでは、サービスの名前空間を使用します。

プロパティー	説明
header	戻り値が SOAP ヘッダーの一部として渡されるかどうかを指定します。
partName	戻り値の **wsdl:part** 要素の **name** 属性の値を指定します。このプロパティーは、ドキュメントスタイルの SOAP バインディングに使用されます。

例

例24.7「完全にアノテーションが付けられた SEI」 は、完全にアノテーションが付けられた SEI を示しています。

例24.7 完全にアノテーションが付けられた 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 アノテーション

24.3.4.1. WSDL ドキュメント

@WSDLDocumentation アノテーション

@WSDLDocumentation アノテーションは、org.apache.cxf.annotations.WSDLDocumentation インターフェイスによって定義されます。SEI または SEI メソッドに配置できます。

このアノテーションを使用するとドキュメントを追加でき、SEI が WSDL に変換された後に **wsdl:documentation** 要素内に表示されます。デフォルトでは、ドキュメント要素はポートタイプ内に表示されますが、配置プロパティーを指定して、ドキュメントを WSDL ファイルの他の場所に表示することができます。「@WSDLDocumentation プロパティー」 は **@WSDLDocumentation** アノテーションがサポートするプロパティーを表示します。

24.3.4.2. @WSDLDocumentation プロパティー

プロパティー	説明
値	**(必須)** ドキュメントテキストを含む文字列。
placement	**(オプション)** WSDL ファイルのどこにこのドキュメントを表示するかを指定します。可能な配置値のリストについては、「WSDL コントラクトへの配置」を参照してください。
faultClass	**(任意)** 配置が **FAULT_MESSAGE**、**PORT_TYPE_OPERATION_FAULT**、または **BINDING_OPERATION_FAULT** に設定されている場合、このプロパティーも障害を表す Java クラスに設定する必要があります。

@WSDLDocumentationCollection annotation

@WSDLDocumentationCollection アノテーションは、org.apache.cxf.annotations.WSDLDocumentationCollection インターフェイスによって定義されます。SEI または SEI メソッドに配置できます。

このアノテーションは、単一の配置場所またはさまざまな配置場所に複数のドキュメント要素を挿入するために使用されます。

WSDL コントラクトへの配置

WSDL コントラクトでドキュメントが表示される場所を指定するには、タイプ **WSDLDocumentation.Placement** の **placement** プロパティーを指定します。プレースメントには、次のいずれかの値を指定できます。

- **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**

@WSDLDocumentation の例

「@WSDLDocumentation の使用」 に、**@WSDLDocumentation** アノテーションを SEI およびそのメソッドの 1 つに追加する方法を示します。

24.3.4.3. @WSDLDocumentation の使用

```
@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);
}
```

「ドキュメントで生成された WSDL」 で表示される WSDL は、「@WSDLDocumentation の使用」 の SEI から生成されるため、**ドキュメント** 要素のデフォルト配置は、それぞれ PORT_TYPE と **PORT_TYPE _OPERATION** になります。

24.3.4.4. ドキュメントで生成された WSDL

```
<wsdl:definitions ... >
  ...
  <wsdl:portType name="HelloWorld">
    <wsdl:documentation>A very simple example of an SEI</wsdl:documentation>
    <wsdl:operation name="sayHi">
      <wsdl:documentation>A traditional form of greeting</wsdl:documentation>
      <wsdl:input name="sayHi" message="tns:sayHi">
    </wsdl:input>
      <wsdl:output name="sayHiResponse" message="tns:sayHiResponse">
    </wsdl:output>
    </wsdl:operation>
```

```
  </wsdl:portType>
  ...
</wsdl:definitions>
```

@WSDLDocumentationCollection の例

「@WSDLDocumentationCollection を使用する」 では 、**@WSDLDocumentationCollection** アノテーションを SEI に追加する方法を説明します。

24.3.4.5. @WSDLDocumentationCollection を使用する

```
@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. メッセージのスキーマ検証

@SchemaValidation アノテーション

@SchemaValidation アノテーションは org.apache.cxf.annotations.SchemaValidation インターフェイスによって定義されます。これは、SEI および個々の SEI メソッドに配置できます。

このアノテーションは、このエンドポイントに送信される XML メッセージのスキーマ検証をオンにします。これは、着信 XML メッセージの形式に問題があると思われる場合のテスト目的に役立ちます。パフォーマンスに大きな影響を与えるため、デフォルトでは検証は無効になっています。

スキーマ検証タイプ

スキーマ検証の動作は、**type** パラメーターによって制御されます。この値は、**org.apache.cxf.annotations.SchemaValidation.SchemaValidationType** 型の列挙になります。「スキーマ検証タイプの値」 は使用可能な検証タイプのリストを示します。

24.3.4.7. スキーマ検証タイプの値

型	説明
IN	クライアントとサーバーの着信メッセージにスキーマ検証を適用します。

型	説明
OUT	クライアントとサーバー上の送信メッセージにスキーマ検証を適用します。
BOTH	クライアントとサーバーの着信メッセージと発信メッセージの両方にスキーマ検証を適用します。
NONE	すべてのスキーマ検証が無効になっています。
REQUEST	スキーマ検証を要求メッセージに適用します。つまり、検証を送信クライアントメッセージと受信サーバーメッセージに適用します。
RESPONSE	スキーマ検証を応答メッセージに適用します。つまり、検証を着信クライアントメッセージと発信サーバーメッセージに適用します。

例

次の例は、MyService SEI に基づいてエンドポイントのメッセージのスキーマ検証を有効にする方法を示しています。アノテーションを SEI 全体に適用する方法と、SEI の個々のメソッドに適用する方法に注意してください。

```
@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. データバインディングの指定

@DataBinding annotation

@DataBinding アノテーションは、org.apache.cxf.annotations.DataBinding インターフェイスによって定義されます。SEI に配置されます。

このアノテーションは、データバインディングを SEI に関連付けるために使用され、デフォルトの JAXB データバインディングを置き換えます。**@DataBinding** アノテーションの値は、データバインディングを提供するクラス **ClassName.class** である必要があります。

サポートされているデータバインディング

次のデータバインディングは現在、Apache CXF でサポートされています。

- **org.apache.cxf.jaxb.JAXBDataBinding**
  (デフォルト) 標準の JAXB データバインディング。
- **org.apache.cxf.sdo.SDODataBinding**
  Service Data Objects (SDO) データバインディングは、Apache Tuscany SDO 実装に基づいています。Maven ビルドのコンテキストでこのデータバインディングを使用する場合は、**cxf-rt-databinding-sdo** アーティファクトに依存関係を追加する必要があります。
- **org.apache.cxf.aegis.databinding.AegisDatabinding**
  Maven ビルドのコンテキストでこのデータバインディングを使用する場合は、**cxf-rt-databinding-aegis** アーティファクトに依存関係を追加する必要があります。
- **org.apache.cxf.xmlbeans.XmlBeansDataBinding**
  Maven ビルドのコンテキストでこのデータバインディングを使用する場合は、**cxf-rt-databinding-xmlbeans** アーティファクトに依存関係を追加する必要があります。
- **org.apache.cxf.databinding.source.SourceDataBinding**
  このデータバインディングは、Apache CXF コアに属しています。
- **org.apache.cxf.databinding.stax.StaxDataBinding**
  このデータバインディングは、Apache CXF コアに属しています。

例

「データバインディングの設定」 SDO バインディングを **HelloWorld** SEI に関連付ける方法を説明します。

24.3.4.9. データバインディングの設定

```
@WebService
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.10. メッセージの圧縮

@GZIP アノテーション

@GZIP アノテーションは org.apache.cxf.annotations.GZIP インターフェイスによって定義されます。SEI に配置されます。

メッセージの GZIP 圧縮を有効にします。GZIP はネゴシエートされた拡張機能です。つまり、クライアントからの最初の要求は gzipped ではありませんが、**Accept** ヘッダーが追加され、サーバーが GZIP 圧縮をサポートする場合、応答は gzipped になり、後続のリクエストも指定します。

「@GZIP Properties」 は、**@GZIP** アノテーションでサポートされるオプションのプロパティーを示しています。

24.3.4.11. @GZIP Properties

プロパティー	説明
threshold	このプロパティーで指定されたサイズよりも小さいメッセージは gzip 圧縮され **ません**。デフォルトは -1(制限なし) です。

@FastInfoset

@FastInfoset アノテーションは、org.apache.cxf.annotations.FastInfoset インターフェイスによって定義されます。SEI に配置されます。

メッセージに FastInfoset 形式を使用できるようにします。FastInfoset は、XML のバイナリーエンコーディング形式であり、メッセージサイズと XML メッセージの処理パフォーマンスの両方を最適化することを目的としています。詳細については、Fast Infoset に関する次の Sun の記事を参照してください。

FastInfoset は、ネゴシエートされた拡張機能です。つまり、クライアントからの最初のリクエストは FastInfoset 形式では追加されませんが、**Accept** ヘッダーが追加され、サーバーが FastInfoset をサポートしていると、応答は FastInfoset になり、後続のリクエストも行われます。

「@FastInfoset プロパティー」 は、**@FastInfoset** アノテーションでサポートされる任意のプロパティーを表示します。

24.3.4.12. @FastInfoset プロパティー

プロパティー	説明
force	ネゴシエートする代わりに、FastInfoset 形式の使用を強制するブールプロパティー。**true** の場合は、FastInfoset 形式の使用を強制し、それ以外はネゴシエートします。デフォルトは **false** です。

@GZIP の例

「GZIP の有効化」 は、**HelloWorld** SEI で GZIP 圧縮を有効にする方法を示します。

24.3.4.13. GZIP の有効化

```
@WebService
@GZIP
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

@FastInfoset の例

「FastInfoset の有効化」 は、**HelloWorld** SEI の FastInfoset 形式を有効にする方法を示します。

24.3.4.14. FastInfoset の有効化

```
@WebService
@FastInfoset
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.15. エンドポイントでのロギングを有効にする

@Logging annotation

@Logging アノテーションは、org.apache.cxf.annotations.Logging インターフェイスによって定義されます。SEI に配置されます。

このアノテーションにより、SEI に関連付けられているすべてのエンドポイントのロギングが可能になります。「@Logging Properties」 は、このアノテーションで設定できるオプションのプロパティーを示しています。

24.3.4.16. @Logging Properties

プロパティー	説明
limit	サイズ制限を指定します。これを超えると、ログでメッセージが切り捨てられます。デフォルトは 64K です。
inLocation	受信メッセージをログに記録する場所を指定します。**<stderr>**、**<stdout>**、**<logger>**、またはファイル名のいずれかを指定できます。デフォルトは **<logger>** です。
outLocation	送信メッセージをログに記録する場所を指定します。**<stderr>**、**<stdout>**、**<logger>**、またはファイル名のいずれかを指定できます。デフォルトは **<logger>** です。

例

「アノテーションを使用した設定のロギング」で、HelloWorld SEI でロギングを有効にする方法を説明します。ここでは、受信メッセージは **<stdout>** に送信され、発信メッセージは **<logger>** に送信されます。

24.3.4.17. アノテーションを使用した設定のロギング

```
@WebService
@Logging(limit=16000, inLocation="<stdout>")
public interface HelloWorld {
```

```
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.18. エンドポイントへのプロパティーとポリシーの追加

概要

プロパティーとポリシーの両方を使用して、設定データをエンドポイントに関連付けることができます。それらの本質的な違いは、**プロパティー** が Apache CXF 固有の設定メカニズムであるのに対し、**ポリシー** は標準の WSDL 設定メカニズムであるということです。ポリシーは通常 WS 仕様および標準に基づいており、通常は WSDL コントラクトに表示される **wsdl:policy** 要素を定義して設定されます。これとは対照的に、プロパティーは Apache CXF 固有のもので、通常は Apache CXF Spring 設定ファイルで **jaxws:properties** 要素を定義して設定されます。

ただし、ここで説明するように、アノテーションを使用して Java でプロパティー設定と WSDL ポリシー設定を定義することもできます。

24.3.4.19. プロパティーの追加

@EndpointProperty annotation

@EndpointProperty アノテーションは、**org.apache.cxf.annotations.EndpointProperty** インターフェイスによって定義されます。SEI に配置されます。

このアノテーションは、Apache CXF 固有の設定をエンドポイントに追加します。エンドポイントのプロパティーは、Spring 設定ファイルで指定することもできます。たとえば、エンドポイントに WS-Security を設定するには、以下のように Spring 設定ファイルの **jaxws:properties** 要素を使用してエンドポイントプロパティーを追加できます。

```
<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>
```

あるいは、「@EndpointProperty アノテーションを使用した WS-Security の設定」 に示すように、**@EndpointProperty** アノテーションを SEI に追加して、Java で前述の設定を指定できます。

24.3.4.20. @EndpointProperty アノテーションを使用した WS-Security の設定

```
@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 アノテーションは、**org.apache.cxf.annotations.EndpointProperties** インターフェイスによって定義されます。SEI に配置されます。

このアノテーションは、複数の **@EndpointProperty** アノテーションをリストにグループ化する方法を提供します。**@EndpointProperties** を使用すると、「@EndpointProperties アノテーションを使用した WS-Security の設定」 に示すように 「@EndpointProperty アノテーションを使用した WS-Security の設定」 を再書き込みできます。

24.3.4.21. @EndpointProperties アノテーションを使用した WS-Security の設定

```
@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. ポリシーの追加

@Policy アノテーション

@Policy アノテーションは **org.apache.cxf.annotations.Policy** インターフェイスによって定義されます。SEI または SEI メソッドに配置できます。

このアノテーションは、WSDL ポリシーを SEI または SEI メソッドに関連付けるために使用されます。ポリシーは、標準の **wsdl:policy** 要素を含む XML ファイルを参照する URI を提供することにより指定されます。SEI から WSDL コントラクトが生成される場合は (たとえば、**java2ws** コマンドラインツールを使用して)、WSDL にこのポリシーを追加するかどうかを指定できます。

「@Policy プロパティー」 に、**@Policy** アノテーションでサポートされるプロパティーを示します。

24.3.4.23. @Policy プロパティー

プロパティー	説明
uri	**(必須)** ポリシー定義を含むファイルの場所。
includeInWSDL	**(オプション)** WSDL を生成するときに、生成されたコントラクトにポリシーを含めるかどうか。デフォルトは **true** です。
placement	**(オプション)** WSDL ファイルのどこにこのドキュメントを表示するかを指定します。可能な配置値のリストについては、「WSDL コントラクトへの配置」を参照してください。
faultClass	**(任意)** 配置が **BINDING_OPERATION_FAULT** または **PORT_TYPE_OPERATION_FAULT** に設定されている場合、このプロパティーも障害を表す Java クラスに設定する必要があります。値は、障害を表す Java クラスです。

@Policies アノテーション

@Policies アノテーションは **org.apache.cxf.annotations.Policies** インターフェイスで定義されます。SEI または SEI メソッドに配置できます。

このアノテーションは、複数の **@Policy** アノテーションをリストにグループ化する方法を提供します。

WSDL コントラクトへの配置

ポリシーが WSDL コントラクトのどこに表示されるかを指定するには、**Policy.Placement** 型である **placement** プロパティーを指定します。プレースメントには、次のいずれかの値を指定できます。

```
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 の例

以下の例は、WSDL ポリシーを **HelloWorld** SEI に関連付け、ポリシーを **sayHi** メソッドに関連付ける方法を示しています。ポリシー自体は、**annotationpolicies** ディレクトリーの下に、ファイルシステムの XML ファイルに保存されます。

```
@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 の例

以下の例のように、**@Policies** アノテーションを使用して、複数の **@Policy** アノテーションをリストにグループ化できます。

```
@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. WSDL の生成

Maven の使用

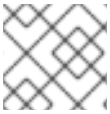
コードにアノテーションが付けられると、**java2ws** Maven プラグの **-wsdl** オプションを使用して、サービスの WSDL コントラクトを生成できます。**java2ws** Maven プラグインのオプションの詳細なリストは、「java2ws」 を参照してください。

例24.8「Java からの WSDL の生成」 では、**java2ws** Maven プラグインを設定して WSDL を生成する方法を示します。

例24.8 Java からの WSDL の生成

```
<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>
```



注記

className の値を修飾された className に置き換えます。

例

例24.9「SEI から生成された WSDL」 に示されている SEI 用に生成された WSDL コントラクトを示しています例24.7「完全にアノテーションが付けられた SEI」。

例24.9 SEI から生成された 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 WSDL コントラクトなしでコンシューマーの開発

概要

サービスコンシューマーを開発するために WSDL コントラクトは必要ありません。アノテーション付き SEI からサービスコンシューマーを作成できます。SEI に加えて、サービスを公開するエンドポイントが公開されるアドレス、サービスを公開するエンドポイントを定義するサービス要素の QName、およびコンシューマーがリクエストを行うエンドポイントを定義するポート要素の QName を知る必要があります。この情報は、SEI のアノテーションで指定することも、個別に提供することもできます。

25.1. JAVA-FIRST のコンシューマー開発

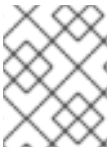
WSDL コントラクトなしでコンシューマーを作成するには、以下を実行する必要があります。

1. コンシューマーが操作を呼び出すサービスの **Service** オブジェクト を作成します。
2. ポートを **Service** オブジェクトに追加します。
3. **Service** オブジェクトの **getPort()** メソッドを使用してサービスの プロキシーを取得します。
4. コンシューマーのビジネスロジックを実装します。

25.2. SERIVCE オブジェクトの作成

概要

javax.xml.ws.Service クラスは、サービスを公開するすべてのエンドポイントの定義が含まれる **wsdl:service** 要素を表します。そのため、サービスのリモート呼び出しを行うためのプロキシーである **wsdl:port** 要素で定義されるエンドポイントを取得できるようにするメソッドを提供します。



注記

Service クラスは、XML ドキュメントの操作ではなく、クライアントコードが Java タイプと連携できるようにする抽象化を提供します。

The create() メソッド

Service クラスには、新しい **Service** オブジェクトの作成に使用できる 2 つの静的 **create()** メソッドがあります。例25.1「**Service create() メソッド**」 に示すように、両方の **create()** メソッドは **wsdl:service** 要素の QName を取得し、 **Service** オブジェクトは表すものと、WSDL コントラクトの場所を指定する URI を取得します。

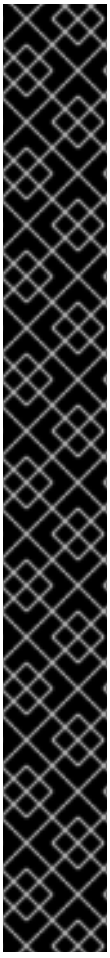
注記

すべてのサービスは WSDL コントラクトを公開します。SOAP/HTTP サービスの場合、URI は通常 **?wsdl** で追加されたサービスの URI です。

> **例25.1 Service create() メソッド**
>
> public static**ServicecreateURLwsdlLocationQNameserviceName**WebServiceExceptionpublic static**ServicecreateQNameserviceName**WebServiceException

serviceName パラメーターの値は QName です。その名前空間部分の値は、サービスのターゲット名前空間です。サービスのターゲット namespace は、 **@WebService** アノテーションの **targetNamespace** プロパティーで指定されます。QName の local part の値は、 **wsdl:service** 要素の **name** 属性の値です。この値は、次のいずれかの方法で決定できます。 .**@WebService** アノテーションの **serviceName** プロパティーで指定されます。

1. **@WebService** アノテーションの **name** プロパティーの値に **Service** を追加します。
2. **Service** を SEI の名前に追加します。



重要

OSGi 環境にデプロイされたプログラムで作成された CXF コンシューマーは、**ClassNotFoundException** が発生する可能性を回避するために特別な処理を必要とします。プログラムで作成された CXF コンシューマーを含むバンドルごとに、シングルトン CXF デフォルトバスを作成し、バンドルのすべての CXF コンシューマーがそれを使用するようにする必要があります。このセーフガードがないと、あるバンドルに別のバンドルで作成された CXF デフォルトバスが割り当てられ、継承するバンドルが失敗する可能性があります。

たとえば、バンドル A が CXF デフォルトバスを明示的に設定せず、バンドル B で作成された CXF デフォルトバスが割り当てられたとします。バンドル A の CXF バスを追加機能 (SSL や WS-Security など) で設定する必要がある場合、バンドル A のアプリケーションから特定のクラスまたはリソースをロードする必要がある場合、失敗します。これは、CXF バスインスタンスがスレッドコンテキストクラスローダー (TCCL) を、それを作成したバンドル (この場合はバンドル B) のバンドルクラスローダーとして設定するためです。さらに、wss4j (CXF で WS-Security を実装) などの特定のフレームワークは、TCCL を使用して、バンドル内から calback ハンドラークラスやその他のプロパティーファイルなどのリソースをロードします。バンドル A はバンドル B のデフォルト CXF バスおよびその TCCL に割り当てられていため、wss4j レイヤーはバンドル A から必要なリソースを読み込みできず、**ClassNotFoundException** エラーが発生します。

シングルトン CXF デフォルトバスを作成するには、「例」 に示すように、サービスオブジェクトを作成するメソッドの開始点に、**main** のコードを挿入します。

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

例

例25.2「**Serivce** オブジェクトの作成」に、例24.7「完全にアノテーションが付けられた SEI」 に記載されている SEI の **Service** オブジェクトを作成するコードを示します。

例25.2 Serivce オブジェクトの作成

```
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「**Serivce** オブジェクトの作成」 のコードは、以下を行います。

サービスのすべての CXF コンシューマーが使用できるシングルトン CXF デフォルトバスを作成します。

@WebService アノテーションの **targetNamespace** プロパティーと **name** プロパティーを使用して、サービスの QName をビルドする。

単一のパラメーター **create()** メソッドを呼び出して、新しい **Service** オブジェクトを作成する。

注記

single パラメーター **create()** を使用すると、WSDL コントラクトへのアクセス時に依存関係が解放されます。

25.3. サービスへのポートの追加

概要

サービスのエンドポイント情報は **wsdl:port** 要素で定義され、**Service** オブジェクトは WSDL コントラクトで定義された各エンドポイントのプロキシーインスタンスを作成します (指定されている場合)。**Service** オブジェクトの作成時に WSDL コントラクトを指定しない場合、**Service** オブジェクトにはサービスを実装するエンドポイントに関する情報がないため、プロキシーインスタンスを作成できません。この場合、**addPort()** メソッドを使用して **wsdl:port** 要素を表すのに必要な情報を **Service** オブジェクトに提供する必要があります。

addPort() メソッド

例25.3「**addPort()** メソッド」に示すように、**Service** クラスは **addPort()** メソッドを定義します。これは、コンシューマーの実装に利用できる WSDL コントラクトがない場合に使用されます。**addPort()** メソッドにより、サービス実装用のプロキシーを作成するのに必要な情報 (通常、**wsdl:port** 要素に保管される) を、**Service** オブジェクトに提供することができます。

例25.3 addPort() メソッド

addPortQName**portName**String**bindingId**String**endpointAddress**WebServiceException

portName の値は QName です。その名前空間部分の値は、サービスのターゲット名前空間です。サービスのターゲット namespace は、 **@WebService** アノテーションの **targetNamespace** プロパティーで指定されます。QName の local part の値は、**wsdl:port** 要素の **name** 属性の値です。この値は、次のいずれかの方法で決定できます。

1. **@WebService** アノテーションの **portName** プロパティーでこれを指定します。
2. **@WebService** アノテーションの **name** プロパティーの値に **Port** を追加します。
3. SEI の名前に **Port** を追加します。

bindingId パラメーターの値は、エンドポイントによって使用されるバインディングタイプを一意に識別する文字列です。SOAP バインディングでは、標準の SOAP namespace: **http://schemas.xmlsoap.org/soap/** を使用します。エンドポイントが SOAP バインディングを使用しない場合、**bindingId** パラメーターの値はバインディング開発者が決定されます。 **endpointAddress** パラメーターの値は、エンドポイントがパブリッシュされるアドレスです。SOAP/HTTP エンドポイントの場合、アドレスは HTTP アドレスです。HTTP 以外のトランスポートは、異なるアドレススキームを使用します。

例

例25.4「サービスのオブジェクトへの **ポートの追加**」 例25.2「**Serivce** オブジェクトの作成」 で作成した **Service** オブジェクトにポートを追加するためのコードを表示します。

例25.4 サービスのオブジェクトへの ポートの追加

```
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「サービスのオブジェクトへの **ポートの追加**」 のコードは、以下を行います。

portName パラメーターの QName を作成します。

addPort() メソッドを呼び出します。

エンドポイントが SOAP バインディングを使用することを指定します。

エンドポイントが公開されるアドレスを指定します。

25.4. エンドポイントのプロキシーの取得

概要

サービスプロキシーは、リモートサービスによって公開されるすべてのメソッドを提供し、リモート呼び出しを行うために必要なすべての詳細を処理するオブジェクトです。**Service** オブジェクトは、**getPort()** メソッドを使用して認識できるすべてのエンドポイントのサービスプロキシーを提供します。サービスプロキシーを取得したら、そのメソッドを呼び出すことができます。プロキシーは、サービスのコントラクトで指定された接続の詳細を使用して、呼び出しをリモートサービスエンドポイントに転送します。

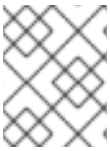
getPort() メソッド

例25.5「**getPort() メソッド**」に示すように、**getPort()** メソッドは、指定されたエンドポイントのサービスプロキシーを返します。返されるプロキシーは SEI と同じクラスです。

例25.5 getPort() メソッド

public**<T> TgetPortQNameportNameClass<T>serviceEndpointInterface**WebServiceException

portName パラメーターの値は、プロキシーが作成されるエンドポイントを定義する **wsdl:port** 要素を識別する QName です。**serviceEndpointInterface** パラメーターの値は、SEI の完全修飾名です。



注記

WSDL コントラクトを操作しない場合、**portName** パラメーターの値は通常 **addPort()** の呼び出し時に **portName** パラメーターに使用される値と同じです。

例

例25.6「サービスプロキシーの取得」 は、例25.4「サービスのオブジェクトへの **ポートの追加**」 に追加されたエンドポイントのサービスプロキシーを取得するためのコードを示します。

例25.6 サービスプロキシーの取得

```
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. コンシューマーのビジネスロジックの実装

概要

リモートエンドポイントのサービスプロキシーをインスタンス化すると、ローカルオブジェクトであるかのようにそのメソッドを呼び出すことができます。リモートメソッドが完了するまで、呼び出しはブロックされます。

注記

メソッドに **@OneWay** アノテーションが付けられている場合、呼び出しは即座に返します。

例

例25.7「WSDL コントラクトなしで実装されたコンシューマー」 は、例24.7「完全にアノテーションが付けられた SEI」 で定義されたサービスのコンシューマーを示しています。

例25.7 WSDL コントラクトなしで実装されたコンシューマー

```
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「WSDL コントラクトなしで実装されたコンシューマー」 のコードは、以下を行います。

Service オブジェクトを作成します。

エンドポイント定義を **Service** オブジェクトに追加します。

Service オブジェクトからサービスプロキシーを取得します。

サービスプロキシーで操作を呼び出します。

第26章 開始点の WSDL コントラクト

26.1. サンプル WSDL コントラクト

例26.1「HelloWorld WSDL コントラクト」 HelloWorld WSDL コントラクトを示しています。このコントラクトは、**wsdl:portType** 要素に単一のインターフェイス Greeter を定義します。コントラクトは、**wsdl:port** 要素にサービスを実装するエンドポイントも定義します。

例26.1 HelloWorld WSDL コントラクト

```
<?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 コントラクト」 で定義された Greeter インターフェイスは次の操作を定義します。

sayHi: 1 つの出力パラメーターの **xsd:string** を持つ。

greetMe: 入力パラメーターの **xsd:string** および出力パラメーターの **xsd:string** を持つ。

greetMeOneWay: 1 つの入力パラメーターの **xsd:string** を持つ。この操作には出力パラメーターがないため、一方向の呼び出しになるように最適化されています (つまり、コンシューマーはサーバーからの応答を待機しません)。

pingMe – 入力パラメーターと出力パラメーターはありませんが、障害例外が発生する可能性があります。

第27章 トップダウンサービス開発

概要

サービスプロバイダーを開発するトップダウンの方法では、サービスプロバイダーが実装する操作とメソッドを定義する WSDL ドキュメントから開始します。WSDL ドキュメントを使用して、サービスプロバイダーの開始点コードを生成します。生成されたコードへのビジネスロジックの追加は、通常の Java プログラミング API を使用して行われます。

27.1. JAX-WS サービスプロバイダー開発の概要

WSDL ドキュメントを入手したら、JAX-WS サービスプロバイダーを開発するプロセスは次のとおりです。

1. 「開始点コードの生成」 開始点コード。
2. サービスプロバイダーの操作を 実装 します。
3. 31章 *サービスの公開* 実装されたサービス。

27.2. 開始点コードの生成

概要

JAX-WS は、WSDL で定義されたサービスから、そのサービスをサービスプロバイダーとして実装する Java クラスへの詳細なマッピングを指定します。**wsdl:portType** 要素で定義される論理インターフェイスは、サービスエンドポイントインターフェイス (SEI) にマッピングされます。WSDL で定義された複合型はすべて、Java Architecture for XML Binding (JAXB) 仕様で定義されたマッピングに従って Java クラスにマップされます。**wsdl:service** 要素で定義されるエンドポイントも、サービスを実装するサービスプロバイダーにアクセスするためにコンシューマーによって使用される Java クラスに生成されます。

cxf-codegen-plugin Maven プラグインはこのコードを生成します。また、実装の開始点コードを生成するためのオプションも提供します。コードジェネレーターは、生成されたコードを制御するためのいくつかのオプションを提供します。

コードジェネレーターの実行

例27.1「サービスコード生成」 は、コードジェネレーターを使用してサービスの開始点コードを生成する方法を示しています。

例27.1 サービスコード生成

```
<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>
```

これは次のことを行います。

- **-impl** オプションは、WSDL コントラクトの各 **wsdl:portType** 要素にシェル実装クラスを生成します。
- **-server** オプションは、サービスプロバイダーをスタンドアロンのアプリケーションとして実行する単純な **main()** を生成します。
- **sourceRoot** は、生成されたコードが **outputDir** という名前のディレクトリーに書き込まれることを指定します。
- **WSDL** 要素は、コードが生成される WSDL コントラクトを指定します。

コードジェネレーターのオプションの完全なリストについては、「cxf-codegen-plugin」 を参照してください。

生成されたコード

表27.1「サービスプロバイダー用に生成されたクラス」 サービスプロバイダーを作成するために生成されるファイルについて説明します。

表27.1 サービスプロバイダー用に生成されたクラス

File	説明
portTypeName.java	SEI。このファイルには、サービスプロバイダーが実装するインターフェイスが含まれています。このファイルは編集しないでください。
serviceName.java	エンドポイント。このファイルには、コンシューマーがサービスで要求を行うために使用する Java クラスが含まれています。
portTypeNameImpl.java	スケルトン実装クラス。このファイルを変更して、サービスプロバイダーを構築します。

File	説明
portTypeNameServer.java	サービスプロバイダーをスタンドアロンプロセスとしてデプロイメントできるようにする基本的なサーバーメインライン。詳細は、31章 *サービスの公開* を参照してください。

さらに、コードジェネレーターは、WSDL コントラクトで定義されているすべてのタイプの Java クラスを生成します。

生成されたパッケージ

生成されたコードは、WSDL コントラクトで使用されている名前空間に基づいてパッケージに配置されます。(**wsdl:portType** 要素、**wsdl:service** 要素、および **wsdl:port** 要素に基づき) サービスをサポートするために生成されたクラスは、WSDL コントラクトのターゲット namespace に基づいてパッケージに配置されます。コントラクトの **types** 要素で定義された型を実装するために生成されたクラスは、**types** 要素の **targetNamespace** 属性に基づいてパッケージに配置されます。

マッピングアルゴリズムは次のとおりです。

1. 先頭の **http://** または **urn://** は名前空間から取り除かれます。
2. namespace の最初の文字列が有効なインターネットドメイン (例: **.com** または **.gov** で終わる) である場合、先頭の **www.** は文字列から取り除かれ、残りの 2 つのコンポーネントは反転される。
3. namespace の最後の文字列がパターン **.xxx** または **.xx** のファイル拡張子で終わる場合、エクステンションは取り除かれる。
4. 名前空間の残りの文字列は、結果の文字列に追加され、ドットで区切られます。
5. すべての文字は小文字になります。

27.3. サービスプロバイダーの実装

実装コードの生成

コードジェネレーターの **-impl** フラグを使用して、サービスプロバイダーをビルドするために使用される実装クラスを生成します。

注記

サービスのコントラクトに XML スキーマで定義されたカスタムタイプが含まれている場合は、タイプのクラスが生成され、使用可能であることを確認する必要があります。

コードジェネレーターの使用の詳細については、「cxf-codegen-plugin」 を参照してください。

生成されたコード

実装コードは 2 つのファイルで設定されています。

- **portTypeName.java**: サービスのサービスインターフェイス (SEI)
- **portTypeNameImpl.java**: サービスによって定義された操作を実装するために使用するクラス。

操作のロジックを実装する

サービスの操作のビジネスロジックを提供するには、**portTypeNameImpl.java** のスタブメソッドを完了します。通常、標準の Java を使用してビジネスロジックを実装します。サービスでカスタム XML スキーマタイプを使用する場合は、タイプごとに生成されたクラスを使用してそれらを操作する必要があります。いくつかの高度な機能にアクセスするために使用できる Apache CXF 固有の API もいくつかあります。

例

たとえば、で定義されたサービスの実装クラス 例26.1「HelloWorld WSDL コントラクト」 は、例27.2「グリーターサービスの実装」 のように見えるかもしれません。太字で強調表示されているコード部分のみをプログラマーが挿入する必要があります。

例27.2 グリーターサービスの実装

```
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 WSDL コントラクトからのコンシューマーの開発

概要

コンシューマーを作成する 1 つの方法は、WSDL コントラクトから開始することです。コントラクトは、コンシューマーが要求するサービスの操作、メッセージ、およびトランスポートの詳細を定義します。コンシューマーの開始点コードは、WSDL コントラクトから生成されます。コンシューマーが必要とする機能が、生成されたコードに追加されます。

28.1. スタブコードの生成

概要

cxf-codegen-plugin Maven プラグインは、WSDL コントラクトからスタブコードを生成します。スタブコードは、リモートサービスで操作を呼び出すために必要なサポートコードを提供します。

コンシューマーの場合、**cxf-codegen-plugin** Maven プラグインは以下のタイプのコードを生成します。

- スタブコード – コンシューマーを実装するためのサポートファイル。
- 開始点コード – リモートサービスに接続し、リモートサービスのすべての操作を呼び出すサンプルコード。

コンシューマーコードの生成

コンシューマーコードを生成するには、**cxf-codegen-plugin** Maven プラグインを使用します。例 28.1「コンシューマーコード生成」 コードジェネレーターを使用してコンシューマーコードを生成する方法を示します。

例28.1 コンシューマーコード生成

```
<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>
```

ここで、**outputDir** は、生成されたファイルが配置されるディレクトリーの場所であり、**wsdl** は WSDL コントラクトの場所を指定します。**-client** オプションは、コンシューマーの **main()** メソッドの開始点コードを生成します。

cxf-codegen-plugin Maven プラグインで利用可能な引数の完全リストは、「cxf-codegen-plugin」 を参照してください。

生成されたコード

コード生成プラグインは、例26.1「HelloWorld WSDL コントラクト」 に示すコントラクトに対して次の Java パッケージを生成します。

- **org.apache.hello_world_soap_http**: このパッケージは 、**http://apache.org/hello_world_soap_http** ターゲット名前空間から生成されます。この名前空間で定義されたすべての WSDL エンティティー (たとえば、Greeter ポートタイプや SOAPService サービス) は、この Java パッケージの Java クラスにマップされます。

- **org.apache.hello_world_soap_http.types**: このパッケージは **http://apache.org/hello_world_soap_http/types** ターゲット名前空間から生成されます。この namespace で定義されたすべての XML タイプ (HelloWorld コントラクトの **wsdl:types** 要素で定義されたもの) は、この Java パッケージの Java クラスにマップします。

cxf-codegen-plugin Maven プラグインによって生成されたスタブファイルは、以下のカテゴリーに分類されます。

- **org.apache.hello_world_soap_http** パッケージの WSDL エンティティーを表すクラス。次のクラスは、WSDL エンティティーを表すために生成されます。
  - greeter: Greeter **wsdl:portType** 要素を表す Java インターフェイス。JAX-WS の用語では、この Java インターフェイスはサービスエンドポイントインターフェイス (SEI) です。
  - **SOAPService** - SOAPService **wsdl:service** 要素を表す Java サービスクラス (**javax.xml.ws.Service** の拡張)
  - PingMeFault: pingMeFault **wsdl:fault** 要素を表す Java 例外クラス (**java.lang.Exception** を拡張)。

- **org.objectweb.hello_world_soap_http.types** パッケージの XML タイプを表すクラス。HelloWorld の例では、生成されるタイプは、要求メッセージと応答メッセージのさまざまなラッパーのみです。これらのデータ型の一部は、非同期呼び出しモデルに役立ちます。

28.2. コンシューマーの実装

概要

WSDL コントラクトから開始するときにコンシューマーを実装するには、次のスタブを使用する必要があります。

- サービスクラス

- SEI

これらのスタブを使用して、コンシューマーコードはサービスプロキシーをインスタンス化し、リモートサービスでリクエストを行います。また、コンシューマーのビジネスロジックも実装します。

生成されたサービスクラス

例28.2「生成されたサービスクラスの概要」に、生成されるサービスクラス **ServiceName_Service** の典型的な概要を示します。[2]これは、**javax.xml.ws.Service** ベースクラスを拡張します。

例28.2 生成されたサービスクラスの概要

```
@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「生成されたサービスクラスの概要」 の **ServiceName** クラスは、以下のメソッドを定義します。

- **ServiceName(URL wsdlLocation, QName serviceName)**: wsdlLocation から取得される WSDL コントラクトの QName **ServiceName** サービスを使用する **wsdl:service** 要素のデータに基づいてサービスオブジェクトを構築します。

- **ServiceName()**: デフォルトのコンストラクター。スタブコードの生成時 (例: **wsdl2java** ツールの実行時) に提供されたサービス名と WSDL コントラクトに基づいて、サービスオブジェクトを構築します。このコンストラクターを使用すると、WSDL コントラクトが指定された場所で引き続き使用可能であることが前提となります。

- **ServiceName(Bus bus)**: **(CXF 固有)** Service の設定に使用される Bus インスタンスを指定できるようにする追加のコンストラクター。これは、複数のバスインスタンスを異なるスレッドに関連付けることができるマルチスレッドアプリケーションのコンテキストで役立ちます。このコンストラクターは、指定したバスがこのサービスで使用されるバスであることを確認する簡単な方法を提供します。**wsdl2java** ツールを呼び出すときに **-fe cxf** オプションを指定する場合にのみ利用できます。

- **getPortName()**: **name** 属性が PortName と同じ **wsdl:port** 要素で定義されたエンドポイントのプロキシーを返します。getter メソッドは、**ServiceName** サービスで定義されるすべての **wsdl:port** 要素に対して生成されます。複数のエンドポイント定義が含まれる **wsdl:service** 要素により、複数の **getPortName()** メソッドを持つサービスクラスが生成されます。

サービスエンドポイントインターフェイス

元の WSDL コントラクトで定義されたすべてのインターフェイスについて、対応する SEI を生成できます。サービスエンドポイントインターフェイスは、**wsdl:portType** 要素の Java マッピングです。元の **wsdl:portType** 要素で定義された各操作は、SEI の対応するメソッドにマッピングされます。操作のパラメーターは次のようにマップされます。 .入力パラメーターはメソッド引数にマップされます。

1. 最初の出力パラメーターは戻り値にマップされます。
2. 複数の出力パラメーターがある場合は、2 番目と後続の出力パラメーターはメソッド引数にマッピングされます (さらに、この引数の値を **Holder** 型を使用して渡す必要があります)。

例として、例28.3「グリーターサービスエンドポイントインターフェイス」に、例26.1「HelloWorld WSDL コントラクト」で定義される **wsdl:portType** 要素から生成される Greeter SEI を示します。簡単にするために、例28.3「グリーターサービスエンドポイントインターフェイス」 標準の JAXB および JAX-WS アノテーションを省略します。

例28.3 グリーターサービスエンドポイントインターフェイス

```
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;
}
```

コンシューマー主な機能

例28.4「コンシューマー実装コード」 は、HelloWorld コンシューマーを実装するコードを示しています。コンシューマーは、SOAPService サービスの SoapPort ポートに接続してから、Greeter ポートタイプでサポートされている各操作の呼び出しに進みます。

例28.4 コンシューマー実装コード

```
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「コンシューマー実装コード」 からの **Client.main()** メソッドは、以下のように行われます。

Apache CXF ランタイムクラスがクラスパス上にある場合、ランタイムは暗黙的に初期化されます。Apache CXF を初期化するために特別な関数を呼び出す必要はありません。

コンシューマーは、HelloWorld の WSDL コントラクトの場所を指定する単一の文字列引数を期待しています。WSDL コントラクトの場所は **wsdlURL** に保存されます。

WSDL コントラクトの場所とサービス名を必要とするコンストラクターを使用してサービスオブジェクトを作成します。 適切な **getPortName()** メソッドを呼び出して、必要なポートのインスタンスを取得します。この場合、SOAPService サービスは **Greeter** サービスエンドポイントインターフェイスを実装する SoapPort ポートのみをサポートします。

コンシューマーは、Greeter サービスエンドポイントインターフェイスでサポートされている各メソッドを呼び出します。

pingMe() メソッドの場合、サンプルコードは PingMeFault フォールト例外をキャッチする方法を示しています。

-fe cxf オプションで生成されたクライアントプロキシー

wsdl2java で **-fe cxf** オプションを指定してクライアントプロキシーを生成する場合 (**cxf** フロントエンドを選択して)、生成されるクライアントプロキシーコードは Java 7 とより適切に統合されます。この場合は、**getServiceNamePort()** メソッドを呼び出すと、SEI のサブインターフェイスである型を返し、以下の追加インターフェイスを実装します。

- **java.lang.AutoCloseable**
- **javax.xml.ws.BindingProvider** (JAX-WS 2.0)
- **org.apache.cxf.endpoint.Client**

これによりクライアントプロキシーの操作がどのように簡素化されるかを確認するには、標準の JAX-WS プロキシーオブジェクトを使用して記述された次の Java コードサンプルを検討してください。

```
// 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();
```

そして、前述のコードを **cxf** フロントエンドで生成したコードを使用して記述した以下の等価なコードサンプルと比較してください。

```
// Programming with proxy generated using '-fe cxf' option
//
try (ServiceNamePortTypeProxy port = service.getServiceNamePort()) {
    port.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, address);
    port.serviceMethod(...);
}
```

[2] **wsdl:service** 要素の **name** 属性が Service で終わる場合、_Service は使用されません。

第29章 実行時の WSDL の検索

概要

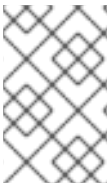
WSDL ドキュメントの場所をアプリケーションにハードコーディングすることはスケーラブルではありません。実際のデプロイメント環境では、実行時に WSDL ドキュメントの場所を解決できるようにする必要があります。Apache CXF は、これを可能にするための多くのツールを提供します。

29.1. WSDL ドキュメントを見つけるためのメカニズム

JAX-WS API を使用してコンシューマーを開発する場合は、サービスを定義する WSDL ドキュメントへのハードコードされたパスを提供する必要があります。これは小規模な環境では問題ありませんが、ハードコードされたパスを使用することは、エンタープライズデプロイメントではうまく機能しません。

この問題に対処するために、Apache CXF は、ハードコードされたパスを使用する必要をなくすための 3 つのメカニズムを提供します。

- 「インジェクションによるプロキシーのインスタンス化」
- 「JAX-WS カタログの使用」
- 「コントラクトリゾルバーの使用」



注記

プロキシーを実装コードに挿入するのが最も簡単なため、一般的に最適なオプションです。サービスプロキシーを挿入してインスタンス化するために必要なのは、クライアントエンドポイントと設定ファイルのみです。

29.2. インジェクションによるプロキシーのインスタンス化

概要

Apache CXF で Spring Framework を使用すると、JAX-WS API を使用してサービスプロキシーを作成する手間を省くことができます。これにより、設定ファイルでクライアントエンドポイントを定義してから、実装コードにプロキシーを直接挿入できます。ランタイムが実装オブジェクトをインスタンス化すると、設定に基づいて外部サービスのプロキシーもインスタンス化されます。実装は、インスタンス化されたプロキシーを参照して渡されます。

プロキシーは設定ファイルの情報を使用してインスタンス化されるため、WSDL の場所をハードコーディングする必要はありません。デプロイメント時に変更できます。ランタイムがアプリケーションのクラスパスで WSDL を検索するように指定することもできます。

手順

外部サービスのプロキシーをサービスプロバイダーの実装に挿入するには、次の手順を実行します。

1. 必要な WSDL ドキュメントを、アプリケーションのすべての部分がアクセスできる既知の場所にデプロイします。

注記

アプリケーションを WAR ファイルとしてデプロイする場合は、WAR の **WEB-INF/wsdl** フォルダーに WSDL ドキュメントと XML スキーマドキュメントをすべて配置することを推奨します。

注記

アプリケーションを JAR ファイルとしてデプロイする場合は、JAR の **META-INF/wsdl** フォルダーに WSDL ドキュメントと XML スキーマドキュメントをすべて配置することを推奨します。

2. 注入されるプロキシーの JAX-WS クライアントエンドポイントを 設定 します。
3. **@Resource** アノテーションを使用して、サービスプロバイダーにプロキシーを 注入します。

プロキシーの設定

アプリケーションの設定ファイルで **jaxws:client** 要素を使用して JAX-WS クライアントエンドポイントを設定します。これにより、指定のプロパティーの **org.apache.cxf.jaxws.JaxWsClientProxy** オブジェクトをインスタンス化するようにランタイムに指示します。このオブジェクトは、サービスプロバイダーに注入されるプロキシーです。

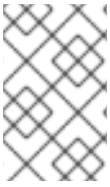
少なくとも、次の属性の値を指定する必要があります。

- **id**: 挿入されるクライアントを識別するために使用される ID を特定します。
- **serviceClass**: プロキシーがリクエストを行うサービスの SEI を指定します。

例29.1「サービス実装に注入されるプロキシーの設定」 は、JAX-WS クライアントエンドポイントの設定を示しています。

例29.1 サービス実装に注入されるプロキシーの設定

```
<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>
```



注記

例29.1「サービス実装に注入されるプロキシーの設定」では、**wsdlLocation** 属性がクラスパスから WSDL を読み込むようにランタイムに指示します。**books.wsdl** がクラスパスにある場合、ランタイムはそれを見つけることができます。

JAX-WS クライアントの設定の詳細については、「コンシューマーエンドポイントの設定」 を参照してください。

プロバイダー実装のコーディング

例29.2「サービス実装へのプロキシーの注入」 に示すように、**@Resource** を使用して、設定したプロキシーをリソースとしてサービス実装に注入します。

例29.2 サービス実装へのプロキシーの注入

```
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;

}
```

アノテーションの **name** プロパティーは JAX-WS クライアントの **id** 属性の値に対応します。設定されたプロキシーは、アノテーションの直後に宣言された **BookService** オブジェクトに注入されます。このオブジェクトを使用して、プロキシーの外部サービスを呼び出すことができます。

29.3. JAX-WS カタログの使用

概要

JAX-WS 仕様では、すべての実装が以下をサポートすることが義務付けられています。

Web サービスの記述の一部である Web サービス文書、特に WSDL および XML スキーマ文書を解決するときに使用される標準のカタログ機能。

このカタログ機能は、OASIS によって指定された XML カタログ機能を使用します。WSDL URI を取得するすべての JAX-WS API とアノテーションは、カタログを使用して WSDL ドキュメントの場所を解決します。

これは、WSDL ドキュメントの場所を特定のデプロイメント環境に合わせて書き換える XML カタログファイルを提供できることを意味します。

カタログを書く

JAX-WS カタログは、OASIS XML カタログ 1.1 仕様で定義されている標準の XML カタログです。マッピングを指定できます。

- ドキュメントのパブリック識別子および/または URI へのシステム識別子。
- リソースの URI から別の URI へ。

表29.1「一般的な JAX-WS カタログ要素」 は、WSDL ロケーション解決に使用されるいくつかの一般的な要素を示しています。

表29.1 一般的な JAX-WS カタログ要素

要素	説明
uri	URI を代替 URI にマップします。
rewriteURI	URI の先頭を書き換えます。たとえば、この要素を使用すると、http://cxf.apache.org で始まるすべての URI を **クラスパス** で始まる URI にマップできます。
uriSuffix	元の URI の接尾辞に基づいて、URI を代替 URI にマップします。たとえば、**foo.xsd** で終わるすべての URI を **classpath:foo.xsd** にマップできます。

カタログのパッケージ化

JAX-WS 仕様では、WSDL および XML Schema ドキュメントを解決するために使用されるカタログが **META-INF/jax-ws-catalog.xml** という名前の利用可能なすべてのリソースを使用してアセンブルされることを示しています。アプリケーションが単一の JAR または WAR にパッケージ化されている場合は、カタログを単一のファイルに配置できます。

アプリケーションが複数の JAR としてパッケージ化されている場合は、カタログを複数のファイルに分割できます。各カタログファイルは、特定の JAR のコードによってアクセスされる WSDL のみを処理するようにモジュール化できます。

29.4. コントラクトリゾルバーの使用

概要

実行時に WSDL ドキュメントの場所を解決するための最も複雑なメカニズムは、独自のカスタムコントラクトリゾルバーを実装することです。これには、Apache CXF 固有の ServiceContractResolver インターフェイスの実装を提供する必要があります。また、カスタムリゾルバーをバスに登録する必要があります。

適切に登録されると、カスタムコントラクトリゾルバーを使用して、必要な WSDL およびスキーマドキュメントの場所が解決されます。

コントラクトリゾルバーの実装

コントラクトリゾルバーは、org.apache.cxf.endpoint.ServiceContractResolver インターフェイスの実装です。例29.3「ServiceContractResolver Interface」 で示されているように、このインターフェイスには **getContractLocation()** の単一のメソッドがあります。このメソッドは実装する必要があります。**getContractLocation()** はサービスの QName を受け取り、サービスの WSDL コントラクトの URI を返します。

例29.3 ServiceContractResolver Interface

```
public interface ServiceContractResolver
```

```
{
  URI getContractLocation(QName qname);
}
```

WSDL コントラクトの場所を解決するために使用されるロジックは、アプリケーション固有です。UDDI レジストリー、データベース、ファイルシステム上のカスタムの場所、または選択したその他のメカニズムからコントラクトの場所を解決するロジックを追加できます。

コントラクトリゾルバーをプログラムで登録

Apache CXF ランタイムがコントラクトリゾルバーを使用する前に、コントラクトリゾルバーレジストリーに登録する必要があります。コントラクトリゾルバーレジストリーは、org.apache.cxf.endpoint.ServiceContractResolverRegistry インターフェイスを実装します。ただし、独自のレジストリーを実装する必要はありません。Apache CXF は、**org.apache.cxf.endpoint.ServiceContractResolverRegistryImpl** クラスでデフォルトの実装を提供します。

コントラクトリゾルバーをデフォルトのレジストリーに登録するには、次の手順を実行します。

1. デフォルトのバスオブジェクトへの参照を取得します。
2. バスの **getExtension()** メソッドを使用して、バスからサービスコントラクトレジストリーを取得します。
3. コントラクトリゾルバーのインスタンスを作成します。
4. レジストリーの **register()** メソッドを使用して、コントラクトリゾルバーをレジストリーに登録します。

例29.4「コントラクトリゾルバーの登録」 は、デフォルトのレジストリーにコントラクトリゾルバーを登録するためのコードを示しています。

例29.4 コントラクトリゾルバーの登録

```
BusFactory bf=BusFactory.newInstance();
Bus bus=bf.createBus();

ServiceContractResolverRegistry registry = bus.getExtension(ServiceContractResolverRegistry);

JarServiceContractResolver resolver = new JarServiceContractResolver();

registry.register(resolver);
```

例29.4「コントラクトリゾルバーの登録」 のコードは、以下を行います。

バスインスタンスを取得します。

バスのコントラクトリゾルバーレジストリーを取得します。

コントラクトリゾルバーのインスタンスを作成します。

コントラクトリゾルバーをレジストリーに登録します。

設定を使用したコントラクトリゾルバーの登録

コントラクトリゾルバーを実装して、設定を通じてクライアントに追加することもできます。コントラクトリゾルバーは、ランタイムが設定を読み取り、リゾルバーをインスタンス化するときに、リゾルバーがそれ自体を登録するように実装されます。ランタイムが初期化を処理するため、クライアントがコントラクトリゾルバーを使用する必要があるかどうかを実行時に決定できます。

設定を通じてクライアントに追加できるようにコントラクトリゾルバーを実装するには、次の手順を実行します。

1. コントラクトリゾルバー実装に **init()** メソッドを追加します。
2. 例29.4「コントラクトリゾルバーの登録」に示すように、コントラクトリゾルバーをコントラクトリゾルバーのレジストリーに登録する **init()** メソッドにロジックを追加します。
3. **@PostConstruct** アノテーションで **init()** メソッドを切り離します。

例29.5「設定を使用して登録できるサービスコントラクトリゾルバー」は、設定を使用してクライアントに追加できるコントラクトリゾルバーの実装を示しています。

例29.5 設定を使用して登録できるサービスコントラクトリゾルバー

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

クライアントにコントラクトリゾルバーを登録するには、クライアントの設定に **bean** 要素を追加する必要があります。**bean** 要素の **class** 属性は、コントラクトリゾルバーを実装するクラスの名前です。

例29.6「Bean によるコントラクトリゾルバーの設定」に、**org.apache.cxf.demos.myContractResolver** クラスによって実装される設定リゾルバーを追加する Bean を示します。

例29.6 Bean によるコントラクトリゾルバーの設定

```
<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>
```

コントラクト解決命令

新しいプロキシーが作成されると、ランタイムはコントラクトレジストリーリゾルバーを使用してリモートサービスの WSDL コントラクトを見つけます。コントラクトリゾルバーレジストリーは、リゾルバーが登録される順序で、各コントラクトリゾルバーの **getContractLocation()** メソッドを呼び出します。登録されたコントラクトリゾルバーの 1 つから返された最初の URI を返します。

よく知られている共有ファイルシステムで WSDL コントラクトを解決しようとするコントラクトリゾルバーを登録した場合、それが使用される唯一のコントラクトリゾルバーになります。ただし、UDDI レジストリーを使用して WSDL の場所を解決するコントラクトリゾルバーを後で登録した場合、レジストリーは両方のリゾルバーを使用してサービスの WSDL コントラクトを見つけることができます。レジストリーは、最初に共有ファイルシステムコントラクトリゾルバーを使用してコントラクトを見つけようとします。そのコントラクトリゾルバーが失敗した場合、レジストリーは UDDI コントラクトリゾルバーを使用してそれを見つけようとします。

第30章 一般的な障害処理

概要

JAX-WS 仕様では、2 種類の障害が定義されています。1 つは、一般的な JAX-WS ランタイム例外です。もう 1 つは、メッセージ処理中に出力されるプロトコル固有のクラスの例外です。

30.1. 実行時の障害

概要

ほとんどの JAX-WS API は、一般的な javax.xml.ws.WebServiceException 例外を出力します。

WebServiceException を出力する API

表30.1「WebServiceException を出力する API」 一般的な WebServiceException 例外を出力できるいくつかの JAX-WSAPI をリストします。

表30.1 WebServiceException を出力する API

API	理由
Binding.setHandlerChain()	ハンドラーチェーンの設定にエラーがあります。
BindingProvider.getEndpointReference()	指定のクラスが **W3CEndpointReference** から割り当てられない。
Dispatch.invoke()	Dispatch インスタンスの設定にエラーがあるか、サービスとの通信中にエラーが発生しました。
Dispatch.invokeAsync()	Dispatch インスタンスの設定にエラーがあります。
Dispatch.invokeOneWay()	Dispatch インスタンスの設定にエラーがあるか、サービスとの通信中にエラーが発生しました。
LogicalMessage.getPayload()	提供された **JAXBContext** を使用してペイロードをアンマーシャリングする際にエラーが発生した。WebServiceException の **cause** フィールドには、元の JAXBException が含まれます。
LogicalMessage.setPayload()	メッセージのペイロードの設定中にエラーが発生しました。**JAXBContext** の使用時に例外が出力された場合、WebServiceException の **cause** フィールドには元の JAXBException が含まれます。
WebServiceContext.getEndpointReference()	指定のクラスが **W3CEndpointReference** から割り当てられない。

30.2. プロトコル障害

概要

リクエストの処理中にエラーが発生すると、プロトコル例外が出力されます。すべての同期リモート呼び出しは、プロトコル例外を出力する可能性があります。根本的な原因は、コンシューマーのメッセージ処理チェーンまたはサービスプロバイダーのいずれかで発生します。

JAX-WS 仕様は、一般的なプロトコル例外を定義しています。また、SOAP 固有のプロトコル例外と HTTP 固有のプロトコル例外も指定します。

プロトコル例外の種類

JAX-WS 仕様では、3 種類のプロトコル例外が定義されています。どの例外をキャッチするかは、アプリケーションで使用されるトランスポートとバインディングによって異なります。

表30.2「一般的なプロトコルの例外の種類」 3 種類のプロトコル例外とそれらがいつ出力されるかについて説明します。

表30.2 一般的なプロトコルの例外の種類

例外クラス	投げられたとき
javax.xml.ws.ProtocolException	この例外は、一般的なプロトコル例外です。使用中のプロトコルに関係なくキャッチできます。SOAP バインディングまたは HTTP バインディングを使用している場合は、特定の障害タイプにキャストできます。XML バインディングを HTTP または JMS トランスポートと組み合わせて使用する場合、汎用プロトコル例外をより具体的な障害タイプにキャストすることはできません。
javax.xml.ws.soap.SOAPFaultException	この例外は、SOAP バインディングを使用するときにリモート呼び出しによって出力されます。詳細は、「SOAP プロトコル例外の使用」 を参照してください。
javax.xml.ws.http.HTTPException	この例外は、Apache CXF バインディングを使用して RESTful Web サービスを開発するときに出力されます。詳細は、パートVI「RESTful Web サービスの開発」 を参照してください。

SOAP プロトコル例外の使用

SOAPFaultException 例外は、SOAP 障害をラップします。根本の SOAP 障害は、javax.xml.soap.SOAPFault オブジェクトとして **fault** フィールドに保存されます。

サービス実装が、アプリケーション用に作成されたカスタム例外のいずれにも適合しない例外を出力する必要がある場合、例外作成者を使用して SOAPFaultException でフォールトをラップし、コンシューマーに出力します。例30.1「SOAP プロトコル例外の出力」 メソッドに無効なパラメーターが渡された場合に SOAPFaultException を作成して出力するためのコードを示しています。

例30.1 SOAP プロトコル例外の出力

```
public Quote getQuote(String ticker)
```

```
{
  ...
  if(tickers.length()<3)
  {
    SOAPFault fault = SOAPFactory.newInstance().createFault();
    fault.setFaultString("Ticker too short");
    throw new SOAPFaultException(fault);
  }
  ...
}
```

コンシューマーが SOAPFaultException 例外をキャッチすると、ラップされた SOAPFault 例外を調べることにより、例外の根本的な原因を取得できます。例30.2「SOAP プロトコル例外からの障害の取得」 にあるように、SOAPFault 例外は SOAPFaultException 例外の **getFault()** メソッドを使用して取得されます。

例30.2 SOAP プロトコル例外からの障害の取得

```
...
try
{
  proxy.getQuote(ticker);
}
catch (SOAPFaultException sfe)
{
  SOAPFault fault = sfe.getFault();
  ...
}
```

第31章 サービスの公開

概要

JAX-WS サービスをスタンドアロン Java アプリケーションとしてデプロイする場合は、サービスプロバイダーを公開するコードを明示的に実装する必要があります。

31.1. サービスを公開するタイミング

Apache CXF は、サービスをサービスプロバイダーとして公開するためのさまざまな方法を提供します。サービスを公開する方法は、使用しているデプロイメント環境によって異なります。Apache CXF でサポートされているコンテナーの多くは、エンドポイントを公開するための書き込みロジックを必要としません。2 つの例外があります:

- サーバーをスタンドアロン Java アプリケーションとしてデプロイする
- ブループリントなしでサーバーを OSGi コンテナーにデプロイする

サポートされているコンテナーにアプリケーションをデプロイする際の詳細については、パート IV「Web サービスエンドポイントの設定」 を参照してください。

31.2. サービスの公開に使用される API

概要

javax.xml.ws.Enddpoint クラスは JAX-WS サービスプロバイダーを公開する作業を行います。エンドポイントを公開するには、次の手順を実行します。

1. サービスプロバイダーの **Endpoint** オブジェクトを作成します。
2. エンドポイントを公開します。
3. アプリケーションがシャットダウンしたら、エンドポイントを停止します。

Endpoint クラスは、サービスプロバイダーを作成および公開するメソッドを提供します。また、単一のメソッド呼び出しでサービスプロバイダーを作成および公開できるメソッドも提供します。

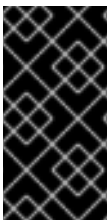
サービスプロバイダーのインスタンス化

サービスプロバイダーは、**Endpoint** オブジェクトを使用してインスタンス化されます。以下のメソッドのいずれかを使用して、サービスプロバイダーの **Endpoint** オブジェクトをインスタンス化します。

- static**EndpointcreateObjectimplementor** この **create()** メソッドは、指定されたサービス実装の **Endpoint** を返します。実装クラスの **javax.xml.ws.BindingType** アノテーションが存在する場合、これにより提供される情報を使用して **Endpoint** オブジェクトが作成されます。アノテーションが存在しない場合、**Endpoint** はデフォルトの SOAP 1.1/HTTP バインディングを使用します。
- static**EndpointcreateURIbindingIDObjectimplementor** この **create()** は、メソッドは、指定されたバインディングを用いた指定された実装オブジェクトの **Endpoint** オブジェクトを返します。**javax.xml.ws.BindingType** アノテーションが存在する場合、このメソッドはそのアノテーションにより提供されるバインディング情報を上書きします。**bindingID** を解決できない

場合や、**null** の場合、**javax.xml.ws.BindingType** で指定されたバインディングを使用して **Endpoint** を作成します。**bindingID** または **javax.xml.ws.BindingType** を使用しない場合は、デフォルトの SOAP 1.1/HTTP バインディングを使用して **Endpoint** が作成されます。

- static**Endpoint**publish**String**address**Object**implementor この **publish()** メソッドは、指定された実装の **Endpoint** オブジェクトを生成し、公開する。**Endpoint** オブジェクトに使用されるバインディングは、指定された **address** の URL スキームによって決定されます。実装で使用可能なバインディングのリストは、URL スキームをサポートするバインディングについてスキャンされます。これが見つかる場合、**Endpoint** オブジェクトが作成され、公開されます。見つからない場合、メソッドは失敗します。
publish() の使用は、**create()** メソッドのいずれかを呼び出してから、???TITLE??? で使用される **publish()** メソッドを呼び出します。



重要

Endpoint 作成メソッドに渡される実装オブジェクトは、**javax.jws.WebService** のアノテーションが付けられたクラスのインスタンスで SEI 実装の要件を満たすか、あるいは **javax.xml.ws.WebServiceProvider** のアノテーションが付けられたクラスのインスタンスでプロバイダーインターフェイスを実装する、のいずれかでなければなりません。

サービスプロバイダーの公開

以下の **Endpoint** メソッドのいずれかを使用してサービスプロバイダーをパブリッシュできます。

- **Publish**String**address** この **publish()** メソッドは、指定されたアドレスにサービスプロバイダーを公開します。

重要

address の URL スキームは、サービスプロバイダーのバインディングのいずれかと互換性がある必要があります。

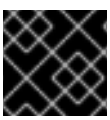
- **Publish**Object**server**Context This **publish()** メソッドは、指定のサーバーコンテキストで提供される情報に基づいてサービスプロバイダーを公開します。サーバーコンテキストはエンドポイントのアドレスを定義する必要があり、コンテキストはサービスプロバイダーの使用可能なバインディングの 1 つとも互換性がある必要があります。

公開されたサービスプロバイダーを停止する

サービスプロバイダーが必要なくなった場合は、その **stop()** メソッドを使用して停止する必要があります。**stop()** に示す 例31.1「公開されたエンドポイントを停止する方法」 メソッドは、エンドポイントをシャットダウンして、使用しているリソースをすべてクリーンアップします。

例31.1 公開されたエンドポイントを停止する方法

stop



重要

エンドポイントが停止すると、再公開できなくなります。

31.3. プレーン JAVA アプリケーションでのサービスの公開

概要

アプリケーションをプレーンな java アプリケーションとしてデプロイする場合は、アプリケーションの **main()** メソッドでエンドポイントを公開するロジックを実装する必要があります。Apache CXF は、アプリケーションの **main()** メソッドを記述する 2 つのオプションを提供します。

- **wsdl2java** ツールによって生成された **main()** メソッドを使用
- エンドポイントを公開するカスタム **main()** メソッドを作成します。

サーバーメインラインの生成

コードジェネレーター **-server** フラグは、ツールがシンプルなサーバーのメインラインを生成するようにします。例31.2「生成されたサーバーメインライン」に示すように、生成されたサーバーメインラインは、指定された WSDL コントラクトの **port** 要素ごとに 1 つのサービスプロバイダーを公開します。

詳細は、「cxf-codegen-plugin」 を参照してください。

例31.2「生成されたサーバーメインライン」 生成されたサーバーのメインラインを示しています。

例31.2 生成されたサーバーメインライン

```
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「生成されたサーバーメインライン」 のコードは、以下を行います。

サービスの実装オブジェクトのコピーをインスタンス化します。

エンドポイントのコントラクト内の **wsdl:port** 要素の **address** の子の内容に基づいて、エンドポイントのアドレスを作成する。

エンドポイントを公開します。

サーバーメインラインの作成

Java の最初の開発モデルを使用した場合、または生成されたサーバーのメインラインを使用したくない場合は、独自に作成できます。サーバーのメインラインを作成するには、次のことを行う必要があります。

1. 「サービスプロバイダーのインスタンス化」 サービスプロバイダーの **javax.xml.ws.Endpoint** オブジェクト。
2. サービスプロバイダーを公開するときに使用する任意のサーバーコンテキストを作成します。
3. 「サービスプロバイダーの公開」 **publish()** メソッドのいずれかを使用するサービスプロバイダー。
4. アプリケーションを終了する準備ができたら、サービスプロバイダーを停止します。

例31.3「カスタムサーバーメインライン」 は、サービスプロバイダーを公開するためのコードを示しています。

例31.3 カスタムサーバーメインライン

```
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「カスタムサーバーメインライン」 のコードは、以下を行います。

サービスの実装オブジェクトのコピーをインスタンス化します。

サービス実装の未公開の **Endpoint** を作成する。

サービスプロバイダーを http://localhost:9000/SoapContext/SoapPort に公開します。

サーバーをシャットダウンするまでループします。

公開されたエンドポイントを停止します。

31.4. OSGI コンテナーでのサービスの公開

概要

OSGi コンテナーにデプロイされるアプリケーションを開発するときは、エンドポイントの公開と停止を、それがパッケージ化されているバンドルのライフサイクルと調整する必要があります。バンドルの開始時にエンドポイントを公開し、バンドルの停止時にエンドポイントを停止する必要があります。

OSGi バンドルアクティベーターを実装することにより、エンドポイントのライフサイクルをバンドルのライフサイクルに結び付けます。バンドルアクティベーターは、開始時にバンドルのリソースを作成するために OSGi コンテナーによって使用されます。コンテナーは、バンドルアクティベータを使用して、停止時にバンドルリソースをクリーンアップします。

バンドルアクティベーターインターフェイス

org.osgi.framework.BundleActivator インターフェイスを実装することにより、アプリケーションのバンドルアクティベーターを作成します。例31.4「バンドルアクティベーターインターフェイス」 に示す BundleActivator インターフェイス、実装する必要のある 2 つのメソッドがあります。

例31.4 バンドルアクティベーターインターフェイス

```
interface BundleActivator
{
  public void start(BundleContext context)
  throws java.lang.Exception;

  public void stop(BundleContext context)
  throws java.lang.Exception;
}
```

start() メソッドは、バンドルの開始時にコンテナーによって呼び出されます。ここで、エンドポイントをインスタンス化して公開します。

stop() メソッドは、バンドルを停止する際にコンテナーによって呼び出されます。これは、エンドポイントを停止する場所です。

start メソッドの実装

バンドルアクティベーターの start メソッドは、エンドポイントを公開する場所です。エンドポイントを公開するには、start メソッドは次のことを行う必要があります。

1. 「サービスプロバイダーのインスタンス化」 サービスプロバイダーの **javax.xml.ws.Endpoint** オブジェクト。
2. サービスプロバイダーを公開するときに使用する任意のサーバーコンテキストを作成します。
3. 「サービスプロバイダーの公開」 **publish()** メソッドのいずれかを使用するサービスプロバイダー。

例31.5「エンドポイントを公開するためのバンドルアクティベーター開始メソッド」 は、サービスプロバイダーを公開するためのコードを示しています。

例31.5 エンドポイントを公開するためのバンドルアクティベーター開始メソッド

```
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「エンドポイントを公開するためのバンドルアクティベーター開始メソッド」 のコードは、以下を行います。

サービスの実装オブジェクトのコピーをインスタンス化します。

サービス実装の未公開の **Endpoint** を作成する。

http://localhost:9000/SoapContext/SoapPort でサービスプロバイダーを公開します。

stop メソッドの実装

バンドルアクティベーターの停止メソッドは、アプリケーションで使用されるリソースをクリーンアップする場所です。その実装には、アプリケーションによって公開されたすべてのエンドポイントを停止するためのロジックを含める必要があります。

例31.6「エンドポイントを停止するためのバンドルアクティベータ停止方法」 は、公開されたエンドポイントを停止するための停止メソッドを示しています。

例31.6 エンドポイントを停止するためのバンドルアクティベータ停止方法

```
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 stop(BundleContext context)
  {
    endpt.stop();
  }

  ...

}
```

コンテナーへの通知

アプリケーションのバンドルにバンドルアクティベーターが含まれていることをコンテナーに通知する必要があります。これを行うには、**Bundle-Activator** プロパティーをバンドルのマニフェストに追加します。このプロパティーは、バンドルをアクティブ化するときにバンドル内のどのクラスを使用するかをコンテナーに指示します。その値は、バンドルアクティベーターを実装するクラスの完全修飾名です。

例31.7「バンドルアクティベーターマニフェストエントリー」に、アクティベーターがクラス **com.widgetvendor.osgi.widgetActivator** で実装されているバンドルのマニフェストエントリーを示します。

例31.7 バンドルアクティベーターマニフェストエントリー

```
Bundle-Activator: com.widgetvendor.osgi.widgetActivator
```

第32章 基本的なデータバインディングの概念

概要

Apache CXF が型マッピングを処理する方法に適用される一般的なトピックがいくつかあります。

32.1. スキーマ定義の包含とインポート

概要

Apache CXF は、**include** および **import** スキーマタグを使用して、スキーマ定義の追加およびインポートをサポートします。これらのタグを使用すると、外部ファイルまたはリソースの定義をスキーマ要素のスコープに挿入できます。インクルードとインポートの本質的な違いは次のとおりです。

- インクルードは、囲んでいるスキーマ要素と同じターゲット名前空間に属する定義を取り込みます。
- インポートすると、囲んでいるスキーマ要素とは異なるターゲット名前空間に属する定義が取り込まれます。

xsd:include 構文

include ディレクティブの構文は次のとおりです。

```
<include schemaLocation="anyURI" />
```

anyURI によって指定された参照スキーマは、囲んでいるスキーマと同じターゲット名前空間に属しているか、ターゲット名前空間にまったく属していない必要があります。参照されるスキーマがどのターゲット名前空間にも属していない場合は、含まれるときに、それを囲むスキーマの名前空間に自動的に採用されます。

例32.1「別のスキーマを含むスキーマの例」 は、別の XML スキーマドキュメントを含む XML スキーマドキュメントの例を示しています。

例32.1 別のスキーマを含むスキーマの例

```
<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「含まれるスキーマの例」 は、含まれているスキーマファイルの内容を示します。

例32.2 含まれるスキーマの例

```
<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 syntax

import ディレクティブの構文は次のとおりです。

```
<import namespace="namespaceAnyURI"
        schemaLocation="schemaAnyURI" />
```

インポートされた定義は、**namespaceAnyURI ターゲット** 名前空間に属している必要があります。**namespaceAnyURI** が空白であるか、指定されていない場合、インポートされたスキーマ定義は修飾されません。

例32.3「別のスキーマをインポートするスキーマの例」 は、別の XML スキーマをインポートする XML スキーマの例を示しています。

例32.3 別のスキーマをインポートするスキーマの例

```
<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「インポートされたスキーマの例」 インポートされたスキーマファイルの内容を示します。

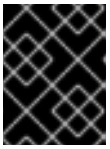
例32.4 インポートされたスキーマの例

```
<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>
```

参照されていないスキーマドキュメントの使用

サービスの WSDL ドキュメントで参照されていないスキーマドキュメントで定義されたタイプを使用することは、3 つのステップのプロセスです。

1. **xsd2wsdl** ツールを使用して、スキーマドキュメントを WSDL ドキュメントに変換します。
2. 生成された WSDL ドキュメントの **wsdl2java** ツールを使用してタイプの Java を生成します。



重要

WSDL ドキュメントがサービスを定義していないことを示す警告が **wsdl2java** ツールから表示されます。この警告は無視してかまいません。

3. 生成されたクラスをクラスパスに追加します。

32.2. XML 名前空間マッピング

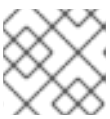
概要

XML スキーマのタイプ、グループ、および要素の定義は、名前空間を使用してスコープされます。名前空間は、同じ名前を使用するエンティティー間で発生する可能性のある名前の衝突を防ぎます。Java パッケージも同様の目的を果たします。したがって、Apache CXF は、スキーマドキュメントのターゲット名前空間を、スキーマドキュメントで定義された構造を実装するために必要なクラスを含むパッケージにマップします。

パッケージの命名

生成されたパッケージの名前は、次のアルゴリズムを使用してスキーマのターゲット名前空間から取得されます。

1. URI スキームが存在する場合は、削除されます。



注記

Apache CXF は、**http:**、**https:**、 および **urn:** スキームのみを削除します。

たとえば、名前空間 **http:\\www.widgetvendor.com\types\widgetTypes.xsd** は **\\widgetvendor.com\types\widgetTypes.xsd** になります。

2. 末尾のファイルタイプ識別子が存在する場合は、削除されます。
たとえば, **\\www.widgetvendor.com\types\widgetTypes.xsd** は **\\widgetvendor.com\types\widgetTypes** になります。

3. 生成される文字列は、/ および **:** を区切り文字として使用し、文字列のリストに分割されます。
そのため、**\\www.widgetvendor.com\types\widgetTypes** はリスト **{"www.widegetvendor.com", "types", "widgetTypes"}** になります。

4. リストの最初の文字列がインターネットドメイン名の場合、次のように分解されます。

   a. 先頭の **www.** は取り除かれます。

   b. 残りの文字列は、**.** を区切り文字として使用し、コンポーネント部分に分割されます。

   c. リストの順序が逆になります。
   そのため、**{"www.widegetvendor.com", "types", "widgetTypes"}** は **{"com", "widegetvendor", "types", "widgetTypes"}** になります

   注記

   インターネットドメイン名は **.com**、**.net**、**.edu**、**.org**、**.gov** のいずれか、または 2 文字の国コードのいずれかで終わります。

5. 文字列はすべて小文字に変換されます。
そのため **{"com", "widegetvendor", "types", "widgetTypes"}** は **{"com", "widegetvendor", "types", "widgettypes"}** になります。

6. 文字列は、次のように有効な Java パッケージ名コンポーネントに正規化されます。

   a. 文字列に特殊文字が含まれる場合、特殊文字はアンダースコア (_) に変換されます。

   b. 文字列のいずれかが Java キーワードである場合、キーワードの前にアンダースコア (_) が付けられます。

   c. 数字で始まる文字列のいずれかの場合は、文字列の前にアンダースコア (_) が付きます。

7. 文字列は、区切り文字として **.** を使用して連結されます。
そのため **{"com", "widegetvendor", "types", "widgettypes"}** は、パッケージ名 **com.widgetvendor.types.widgettypes** になります。

namespace **http:\\www.widgetvendor.com\types\widgetTypes.xsd** で定義された XML Schema コンストラクトは Java パッケージ **com.widgetvendor.types.widgettypes** にマッピングされます。

パッケージの内容

JAXB で生成されたパッケージには次のものが含まれています。

- スキーマで定義された各複合型を実装するクラス
複合型マッピングの詳細については、35章 *複雑なタイプの使用* を参照してください。

- **enumeration** ファセットを使用して定義された単純なタイプの列挙タイプ
列挙のマッピング方法の詳細については、「列挙」 を参照してください。

- スキーマからオブジェクトをインスタンス化するメソッドが含まれるパブリック **ObjectFactory** クラス
  ObjectFactory クラスの詳細は、「オブジェクトファクトリー」 を参照してください。
- パッケージ内のクラスに関するメタデータを提供する **package-info.java** ファイル

32.3. オブジェクトファクトリー

概要

JAXB は、オブジェクトファクトリーを使用して、JAXB で生成された構造のインスタンスをインスタンス化するためのメカニズムを提供します。オブジェクトファクトリーには、パッケージのスコープで XML スキーマで定義されたすべての設定をインスタンス化するためのメソッドが含まれています。唯一の例外は、列挙型がオブジェクトファクトリーで作成メソッドを取得しないことです。

複合型ファクトリーメソッド

XML スキーマ複合型を実装するために生成された Java クラスごとに、オブジェクトファクトリーにはクラスのインスタンスを作成するためのメソッドが含まれています。このメソッドの形式は次のとおりです。

> **typeName** create**typeName**();

たとえば、スキーマに **widgetType** という名前の複合型が含まれる場合、Apache CXF は **WidgetType** と呼ばれるクラスを生成して実装します。例32.5「複合型オブジェクトファクトリーエントリー」 は、オブジェクトファクトリーで生成された作成メソッドを示しています。

> **例32.5 複合型オブジェクトファクトリーエントリー**
>
> ```
> public class ObjectFactory
> {
>   ...
>   WidgetType createWidgetType()
>   {
>     return new WidgetType();
>   }
>   ...
> }
> ```

要素ファクトリーメソッド

スキーマのグローバルスコープで宣言されている要素の場合、Apache CXF はファクトリーメソッドをオブジェクトファクトリーに挿入します。33章*XML 要素の使用* で説明されているように、XML Schema 要素は **JAXBElement<T>** オブジェクトにマッピングされます。作成方法は次の形式を取ります。

> public JAXBElement<**elementType**> create**elementName**(**elementType** value);

たとえば、タイプ **xsd:string** の **comment** という名前の要素がある場合、Apache CXF は 例32.6「要素オブジェクトファクトリーエントリー」 に示されるオブジェクトファクトリーメソッドを生成します。

例32.6 要素オブジェクトファクトリーエントリー

```
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. ランタイムマーシャラーへのクラスの追加

概要

Apache CXF ランタイムが XML データを読み書きするとき、XML スキーマタイプをそれらの代表的な Java タイプに関連付けるマップを使用します。デフォルトでは、マップには WSDL コントラクトの **schema** 要素のターゲット namespace で定義されたすべてのタイプが含まれます。また、WSDL コントラクトにインポートされたスキーマの名前空間から生成されたタイプも含まれています。

アプリケーションの **schema** 要素によって使用されるスキーマ namespace 以外の名前空間から型を追加するには、**@XmlSeeAlso** アノテーションを使用します。アプリケーションがアプリケーションの WSDL ドキュメントの範囲外で生成されるタイプを操作する必要がある場合は、**@XmlSeeAlso** アノテーションを編集して JAXB マップに追加できます。

@XmlSeeAlso アノテーションを使用する

@XmlSeeAlso アノテーションは、サービスの SEI に追加できます。JAXB コンテキストに含めるクラスのコンマ区切りリストが含まれています。例32.7「JAXB コンテキストにクラスを追加するための構文」 は **@XmlSeeAlso** アノテーションを使用する構文を示しています。

例32.7 JAXB コンテキストにクラスを追加するための構文

```
import javax.xml.bind.annotation.XmlSeeAlso;
    @WebService()
    @XmlSeeAlso({Class1.class, Class2.class, ..., ClassN.class})
    public class GeneratedSEI {
        ...
    }
```

JAXB が生成したクラスにアクセスできる場合は、必要なタイプに対応するために生成された **ObjectFactory** クラスを使用することがより効率的になります。**ObjectFactory** クラスを含めるには、オブジェクトファクトリーに認識されるすべてのクラスが含まれます。

例

例32.8「JAXB コンテキストへのクラスの追加」 は、**@XmlSeeAlso** アノテーションが付けられた SEI を示しています。

例32.8 JAXB コンテキストへのクラスの追加

```
...
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 XML 要素の使用

概要

XML スキーマ要素は、XML ドキュメント内の要素のインスタンスを定義するために使用されます。要素は、XML スキーマドキュメントのグローバルスコープで定義されるか、複合型のメンバーとして定義されます。それらがグローバルスコープで定義されている場合、Apache CXF はそれらを JAXB 要素クラスにマップし、操作を容易にします。

概要

XML ドキュメントの要素インスタンスは、XML スキーマドキュメントのグローバルスコープの XML スキーマ **element** 要素によって定義されます。Java 開発者が要素を操作するのを容易にするために、Apache CXF はグローバルにスコープ設定された要素を特別な JAXB 要素クラスにマッピングするか、コンテンツ型に一致するように生成された Java クラスにマッピングします。

要素がどのようにマッピングされるかは、要素が **type** 属性によって参照される名前付き型を使用して要素を定義されるか、または要素がインライン型定義を使用して定義されるかによって異なります。インライン型定義で定義された要素は、Java クラスにマップされます。

インライン型はスキーマ内の他の要素で再利用できないため、要素は名前付き型を使用して定義することを推奨します。

XML スキーママッピング

XML Schema では、要素は **element** 要素を使用して定義されます。**element** 要素には必須属性が 1 つあります。**name** は、XML ドキュメントに表示される要素の名前を指定します。

name 属性 **要素** の他に、表33.1「要素の定義に使用される属性」 にリストされている任意の属性があります。

表33.1 要素の定義に使用される属性

属性	説明
type	要素の型を指定します。タイプは、任意の XML スキーマプリミティブ型またはコントラクトで定義された任意の名前付き複合型にすることができます。この属性が指定されていない場合は、インラインタイプ定義を含める必要があります。
nillable	要素をドキュメントから完全に除外できるかどうかを指定します。**nillable** が **true** に設定されている場合、要素はスキーマを使用して生成したドキュメントから省略できます。

属性	説明
abstract	要素をインスタンスドキュメントで使用できるかどうかを指定します。**true** は、要素がインスタンスドキュメントに表示されないことを示します。代わりに、この要素の QName が含まれる **substitutionGroup** 属性が含まれる別の要素がこの要素に表示される必要があります。この属性がコード生成にどのように影響するかについては、「抽象要素の Java マッピング」 を参照してください。
substitutionGroup	この要素で置き換えることができる要素の名前を指定します。タイプ置換の使用の詳細については、37章*元素置換* を参照してください。
default	要素のデフォルト値を指定します。この属性がコード生成にどのように影響するかについては、「デフォルト値を持つ要素の Java マッピング」 を参照してください。
固定:	要素の固定値を指定します。

例33.1「単純な XML スキーマ要素の定義」 は簡単な要素定義を示しています。

例33.1 単純な XML スキーマ要素の定義

```
<element name="joeFred" type="xsd:string" />
```

要素は、インライン型定義を使用して独自の型を定義することもできます。インライン型は、**complexType** 要素または **simpleType** 要素のいずれかを使用して指定します。データの型が複雑か単純かを指定すると、各型のデータに使用できるツールを使用して、必要な任意のタイプのデータを定義できます。

例33.2「インラインタイプの XML スキーマ要素定義」 は、インラインタイプ定義を含む要素定義を示しています。

例33.2 インラインタイプの XML スキーマ要素定義

```
<element name="skate">
  <complexType>
    <sequence>
      <element name="numWheels" type="xsd:int" />
      <element name="brand" type="xsd:string" />
    </sequence>
  </complexType>
</element>
```

名前付きタイプの要素の JAVA マッピング

デフォルトでは、グローバルに定義されている要素は **JAXBElement<T>** オブジェクトにマップされます。このオブジェクトは、**element** 要素の **type** 属性の値によって決定されます。プリミティブ型の場合、テンプレートクラスは、「ラッパークラス」 で説明されているラッパークラスマッピングを使用して派生します。複合型の場合、複合型をサポートするために生成された Java クラスがテンプレートクラスとして使用されます。

マッピングをサポートし、要素の QName に関する開発者の不必要な心配を軽減するために、例33.3「グローバルスコープ要素のオブジェクトファクトリーメソッド」 に示すように、グローバルに定義された要素ごとにオブジェクトファクトリーメソッドが生成されます。

例33.3 グローバルスコープ要素のオブジェクトファクトリーメソッド

```
public class ObjectFactory {

    private final static QName _name_QNAME = new QName("targetNamespace", "localName");

    ...

    @XmlElementDecl(namespace = "targetNamespace", name = "localName")
    public JAXBElement<type> createname(type value);

}
```

たとえば、例33.1「単純な XML スキーマ要素の定義」 で定義された要素結果は、例33.4「単純な要素のオブジェクトファクトリー」 に示すオブジェクトファクトリーメソッドになります。

例33.4 単純な要素のオブジェクトファクトリー

```
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「グローバルスコープ要素の使用」 は、Java でグローバルスコープの要素を使用する例を示しています。

例33.5 グローバルスコープ要素の使用

```
JAXBElement<String> element = createJoeFred("Green");
String color = element.getValue();
```

WSDL で名前付きタイプの要素を使用する

グローバルにスコープ設定された要素を使用してメッセージ部分を定義する場合、生成される Java パラメーターは **JAXBElement<T>** のインスタンスではありません。代わりに、通常の Java タイプまたはクラスにマップされます。

例33.6「メッセージ部分として要素を使用する WSDL」 に示される WSDL フラグメントの場合、結果のメソッドは **String** 型のパラメーターを持ちます。

例33.6 メッセージ部分として要素を使用する 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「グローバル要素を一部として使用する Java メソッド」に、**sayHi** 操作の生成されたメソッド署名を示します。

例33.7 グローバル要素を一部として使用する Java メソッド

```
StringsayHiStringin
```

インラインタイプの要素の JAVA マッピング

要素がインラインタイプを使用して定義されている場合、他のタイプを Java にマッピングするために使用されるのと同じルールに従って Java にマッピングされます。単純型の規則については、34章*単純型の使用* で説明しています。複合型のルールについては、35章*複雑なタイプの使用* で説明しています。

インラインの型定義を持つ要素に対して Java クラスが生成されると、生成されたクラスは **@XmlRootElement** アノテーションで装飾されます。**@XmlRootElement** アノテーションには、**name** と **namespace** の 2 つの便利なプロパティーがあります。属性については、表33.2「@XmlRootElement アノテーションのプロパティー」 で説明されています。

表33.2 @XmlRootElement アノテーションのプロパティー

プロパティー	説明
name	XML Schema **element** 要素の **name** 属性の値を指定します。
namespace	要素が定義されている名前空間を指定します。この要素がターゲット名前空間で定義されている場合、プロパティーは指定されていません。

要素が以下の条件を 1 つ以上満たす場合、**@XmlRootElement** アノテーションは使用されません。

- 要素の **nillable** 属性が **true** に設定されます。
- 要素は、置換グループのヘッド要素です
  置換グループの詳細については、37章*元素置換* を参照してください。

抽象要素の JAVA マッピング

要素の **abstract** 属性を **true** に設定すると、タイプのインスタンスをインスタンス化するオブジェクトファクトリーメソッドが生成されません。要素がインライン型を使用して定義されている場合、インライン型をサポートする Java クラスが生成されます。

デフォルト値を持つ要素の JAVA マッピング

要素の **default** 属性が使用される場合、**defaultValue** プロパティーが生成された **@XmlElementDecl** アノテーションに追加されます。たとえば、で定義された要素 例33.8「デフォルト値の XML スキーマ要素」 結果は、例33.9「デフォルト値を持つ要素のオブジェクトファクトリーメソッド」 に示すオブジェクトファクトリーメソッドになります。

例33.8 デフォルト値の XML スキーマ要素

```
<element name="size" type="xsd:int" default="7"/>
```

例33.9 デフォルト値を持つ要素のオブジェクトファクトリーメソッド

```
@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 単純型の使用

概要

XML Schema の単純なタイプは **xsd:int** などの XML Schema プリミティブタイプであるか、**simpleType** 要素を使用して定義されます。これらは、子や属性を含まない要素を指定するために使用されます。これらは通常、ネイティブ Java 構造にマップされ、それらを実装するために特別なクラスを生成する必要はありません。列挙された単純なタイプは Java **enum** タイプにマップされるため、生成されたコードでは発生しません。

34.1. プリミティブ型

概要

XML スキーマプリミティブ型の 1 つを使用してメッセージ部分が定義されると、生成されたパラメーターの型は、対応する Java ネイティブ型にマップされます。複合型のスコープ内で定義されている要素をマッピングする場合も、同じパターンが使用されます。結果のフィールドは、対応する Java ネイティブタイプです。

マッピング

表34.1「XML スキーマのプリミティブ型から Java ネイティブ型へのマッピング」 XML スキーマプリミティブ型と Java ネイティブ型の間のマッピングをリスト表示します。

表34.1 XML スキーマのプリミティブ型から Java ネイティブ型へのマッピング

XML スキーマタイプ	Java タイプ
xsd:string	**String**
xsd:integer	**BigInteger**
xsd:int	**int**
xsd:long	**long**
xsd:short	**short**
xsd:decimal	**BigDecimal**
xsd:float	**float**
xsd:double	**double**
xsd:boolean	**boolean**
xsd:byte	**byte**
xsd:QName	**QName**

XML スキーマタイプ	Java タイプ
xsd:dateTime	**XMLGregorianCalendar**
xsd:base64Binary	**byte[]**
xsd:hexBinary	**byte[]**
xsd:unsignedInt	**long**
xsd:unsignedShort	**int**
xsd:unsignedByte	**short**
xsd:time	**XMLGregorianCalendar**
xsd:date	**XMLGregorianCalendar**
xsd:g	**XMLGregorianCalendar**
xsd:anySimpleType [a]	**Object**
xsd:anySimpleType [b]	**String**
xsd:duration	**所要時間**
xsd:NOTATION	**QName**

[a] このタイプの要素の場合。

[b] このタイプの属性の場合。

ラッパークラス

XML スキーマプリミティブ型を Java プリミティブ型にマッピングすることは、考えられるすべての XML スキーマ構造に対して機能するわけではありません。いくつかのケースでは、XML スキーマプリミティブ型が Java プリミティブ型の対応するラッパー型にマップされる必要があります。ユースケースには以下が含まれます。

- 以下に示すように、その **nillable** 属性が **true** に設定された **element** 要素。

```
<element name="finned" type="xsd:boolean"
         nillable="true" />
```

- **minOccurs** 属性が **0**、**maxOccurs** 属性が **1** に設定されているか、**maxOccurs** 属性が指定されていない **element** 要素は、次のようになります。

```
<element name="plane" type="xsd:string" minOccurs="0" />
```

- 図のように，**use** 属性が **optional** または not specified に設定され， **default** 属性も **fixed** 属性も指定されていない **attribute** 要素です。

```
<element name="date">
  <complexType>
    <sequence/>
    <attribute name="calType" type="xsd:string"
               use="optional" />
  </complexType>
</element>
```

表34.2「プリミティブスキーマタイプから Java ラッパークラスへのマッピング」 は、これらの場合に XML スキーマプリミティブ型が Java ラッパークラスにマップされる方法を示しています。

表34.2 プリミティブスキーマタイプから Java ラッパークラスへのマッピング

スキーマタイプ	Java タイプ
xsd:int	**java.lang.Integer**
xsd:long	**java.lang.Long**
xsd:short	**java.lang.Short**
xsd:float	**java.lang.Float**
xsd:double	**java.lang.Double**
xsd:boolean	**java.lang.Boolean**
xsd:byte	**java.lang.Byte**
xsd:unsignedByte	**java.lang.Short**
xsd:unsignedShort	**java.lang.Integer**
xsd:unsignedInt	**java.lang.Long**
xsd:unsignedLong	**java.math.BigInteger**
xsd:duration	**java.lang.String**

34.2. 制限によって定義された単純なタイプ

概要

XML スキーマを使用すると、別のプリミティブ型または単純型から新しい型を派生させることにより、単純型を作成できます。単純なタイプは **simpleType** 要素を使用して記述されます。

新しいタイプは、1 つ以上のファセットで **基本タイプ** を制限することによって記述されます。これらのファセットは、新しいタイプに格納できる有効な値を制限します。例えば、ちょうど 9 文字の **string** である **SSN** というシンプルな型を定義することができます。

各プリミティブ XML スキーマタイプには、独自のオプションファセットのセットがあります。

手順

独自の単純型を定義するには、次のようにします。

1. 新しいシンプルタイプの基本タイプを決定します。
2. 選択した基本タイプで使用可能なファセットに基づいて、新しいタイプを定義する制限を決定します。
3. このセクションで説明する構文を使用して、適切な **simpleType** 要素をコントラクトの types セクションに入力します。

XML スキーマでの単純型の定義

例34.1「単純な型の構文」 は、単純型を記述するための構文を示しています。

例34.1 単純な型の構文

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value" />
    <facet value="value" />
    ...
  </restriction>
</simpleType>
```

タイプの description は **simpleType** 要素で囲まれ、**name** 属性の値で識別されます。新しい単純なタイプが定義されるベースタイプは、**xsd:restriction** 要素の **ベース** 属性で指定されます。各ファセット要素は **restriction** 要素内に指定されます。利用可能なファセットとその有効な設定は、ベース型によって異なります。たとえば、**xsd:string** には以下を含む数多くのファセットが含まれます。

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whitespace**

例34.2「郵便番号シンプルタイプ」 は、米国の州で使用される 2 文字の郵便番号を表す単純型の定義を示しています。大文字を 2 文字だけ含めることができます。**TX** は有効な値ですが、**tx** または **tX** は有効な値ではありません。

例34.2 郵便番号シンプルタイプ

```
<xsd:simpleType name="postalCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{2}" />
  </xsd:restriction>
</xsd:simpleType>
```

Java へのマッピング

Apache CXF は、ユーザー定義の単純型を単純型の基本型の Java 型にマップします。そのため、例34.2「郵便番号シンプルタイプ」 に示されているように、シンプルなタイプの **postalCode** を使用するメッセージは、**postalCode** のベースタイプが **xsd:string** であるため、**String** にマッピングされます。例えば、例34.3「シンプルタイプのクレジットリクエスト」 に示されている WSDL のフラグメントは、**String** のパラメーター **postalCode** を取る Java メソッド **state()** になります。

例34.3 シンプルタイプのクレジットリクエスト

```
<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>
```

ファセットの実施

デフォルトでは、Apache CXF は、単純型を制限するために使用されるファセットを強制しません。ただし、スキーマ検証を有効にすることで、ファセットを適用するように Apache CXF エンドポイントを設定できます。

スキーマ検証を使用するように Apache CXF エンドポイントを設定するには、**schema-validation-enabled** プロパティーを **true** に設定します。例34.4「スキーマ検証を使用するように設定されたサービスプロバイダー」 は、スキーマ検証を使用するサービスプロバイダーの設定を示しています

例34.4 スキーマ検証を使用するように設定されたサービスプロバイダー

```
<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>
```

スキーマ検証の設定の詳細については、「スキーマ検証タイプの値」 を参照してください。

34.3. 列挙

概要

XML Schema では、列挙型は **xsd:enumeration** ファセットを使用して定義された単純な型です。アトミック単純なタイプとは異なり、これらは Java **enum**s にマッピングされます。

XML スキーマで列挙型を定義する

列挙型は、**xsd:enumeration** ファセットを使用したシンプルなタイプです。各 **xsd:enumeration** ファセットは、列挙されたタイプの 1 つの可能な値を定義します。

例34.5「XML スキーマ定義の列挙」 は、列挙型の定義を示します。次の可能な値があります。

- **big**
- **large**
- **mungo**
- **gargantuan**

例34.5 XML スキーマ定義の列挙

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

Java へのマッピング

ベースタイプが **xsd:string** である XML Schema の列挙型は、自動的に Java の **enum** 型にマッピングされます。「列挙マッピングのカスタマイズ」 で説明されているカスタマイズを使用すると、コードジェネレーターに対し、他のベース型の列挙を Java **enum** 型にマッピングするように指示できます。

enum タイプは以下のように作成されます。

1. タイプの名前は、単純なタイプ定義の **name** 属性から取得され、Java 識別子に変換されます。一般に、これは XML スキーマの名前の最初の文字を大文字に変換することを意味します。XML スキーマ名の最初の文字が無効な文字である場合は、名前の前にアンダースコア (_) が追加されます。
2. **enumeration** ファセットごとに、列挙定数が **value** 属性の値に基づいて生成されます。定数の名前は、値内のすべての小文字を同等の大文字に変換することによって付けられます。
3. 列挙型の基本型からマップされた Java 型を取得するコンストラクターが生成されます。

4. **value()** というパブリックメソッドは、タイプのインスタンスによって表されるファセット値にアクセスするための生成されます。
   value() メソッドの戻り値タイプは、XML Schema 型のベースタイプです。

5. **fromValue()** というパブリックメソッドが生成され、ファセット値に基づいて列挙タイプのインスタンスを作成します。
   value() メソッドのパラメータータイプは、XML Schema 型のベースタイプです。

6. クラスは **@XmlEnum** アノテーションで禁止されています。

例34.5「XML スキーマ定義の列挙」で定義される列挙型は、例34.6「文字列ベースの XML スキーマ列挙用に生成された列挙型」に記載の **enum** 型にマッピングされます。

例34.6 文字列ベースの XML スキーマ列挙用に生成された列挙型

```
@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. リスト

概要

XML スキーマは、スペースで区切られた単純型のリストであるデータ型を定義するためのメカニズムをサポートしています。リスト型を使用した **primeList** の例は、例34.7「リストタイプの例」 に記載されています。

例34.7 リストタイプの例

```
<primeList>1 3 5 7 9 11 13<\primeList>
```

XML Schema リストタイプは通常 Java **List<T>** オブジェクトにマッピングされます。このパターンの唯一のバリエーションは、メッセージ部分が XML スキーマリストタイプのインスタンスに直接マップされる場合です。

XML スキーマでのリストタイプの定義

XML スキーマのリスト型は単純型で、**simpleType** 要素を使用して定義されます。リストタイプを定義するために使用される最も一般的な構文を 例34.8「XML スキーマリストタイプの構文」 に示します。

例34.8 XML スキーマリストタイプの構文

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value" />
    <facet value="value" />
    ...
  </list>
</simpleType>
```

atomType に指定された値は、リスト内の要素のタイプを定義します。指定できるのは、**xsd:int** または **xsd:string** などの組み込み XML スキーマアトミック型のいずれか、リストではないユーザー定義の単純型だけです。

リストタイプにリストされている要素のタイプを定義することに加えて、ファセットを使用してリストタイプのプロパティーをさらに制約することもできます。表34.3「リストタイプファセット」 リストタイプで使用されるファセットを示します。

表34.3 リストタイプファセット

ファセット	結果
length	リストタイプのインスタンスの要素数を定義します。
minLength	リストタイプのインスタンスで許可される要素の最小数を定義します。
maxLength	リストタイプのインスタンスで許可される要素の最大数を定義します。
enumeration	リストタイプのインスタンスの要素の許容値を定義します。

ファセット	結果
pattern	リストタイプのインスタンスの要素の字句形式を定義します。パターンは正規表現を使用して定義されます。

たとえば、例34.7「リストタイプの例」 に示される **simpleList** 要素の定義を、例34.9「リストタイプの定義」 に示します。

例34.9 リストタイプの定義

```
<simpleType name="primeListType">
  <list itemType="int"/>
</simpleType>
<element name="primeList" type="primeListType"/>
```

例34.8「XML スキーマリストタイプの構文」 に示す構文に加えて 例34.10「リストタイプの代替構文」 に示すあまり一般的でない構文を使用してリストタイプを定義することもできます。

例34.10 リストタイプの代替構文

```
<simpleType name="listType">
  <list>
    <simpleType>
      <restriction base="atomicType">
        <facet value="value"/>
        <facet value="value"/>
        ...
      </restriction>
    </simpleType>
  </list>
  </simpleType>
```

リストタイプ要素の Java へのマッピング

要素がリストタイプとして定義されている場合、リストタイプはコレクションプロパティーにマップされます。collection プロパティーは Java **List<T>** オブジェクトです。**List<T>** が使用するテンプレートクラスは、リストのベースタイプからマッピングされたラッパークラスです。たとえば、例34.9「リストタイプの定義」 で定義されたリストタイプは **List<Integer>** にマッピングされます。

ラッパータイプマッピングの詳細については、「ラッパークラス」 を参照してください。

リストタイプパラメーターの Java へのマッピング

メッセージ部分がリストタイプとして定義されている場合や、リストタイプの要素にマップされる場合、作成されるメソッドパラメーターは **List<T>** オブジェクトではなくアレイにマッピングされます。配列の基本型は、リスト型の基本クラスのラッパークラスです。

たとえば、例34.11「リストタイプのメッセージ部分を持つ WSDL」 の WSDL フラグメントは結果は、例34.12「リスト型パラメーターを使用した Java メソッド」 に示すメソッドシグネチャーになります。

例34.11 リストタイプのメッセージ部分を持つ 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 リスト型パラメーターを使用した Java メソッド

```
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. 組合

概要

XML スキーマでは、共用体は、データがいくつかの単純な型の 1 つである可能性がある型を記述できるようにする構造です。たとえば、値が整数 **1** または文字列 **first** のいずれかである型を定義できます。ユニオンは Java **String** にマッピングされます。

XML スキーマでの定義

XML スキーマユニオンは、**simpleType** 要素を使用して定義されます。これには、ユニオンのメンバー型を定義する **union** 要素が少なくとも 1 つ含まれます。ユニオンのメンバータイプは、ユニオンのインスタンスに格納できる有効なタイプのデータです。これらは **union** 要素の **memberTypes** 属性を使用して定義されます。**memberTypes** 属性の値には、1 つ以上の定義された単純な型名のリストが含まれます。例34.13「単純な共用体タイプ」 は、整数または文字列のいずれかを格納できるユニオンの定義を示しています。

例34.13 単純な共用体タイプ

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

名前付き型をユニオンのメンバー型として指定することに加えて、匿名の単純型をユニオンのメンバー型として定義することもできます。これは、**union** 要素内に匿名タイプ定義を追加することで行います。例34.14「匿名メンバータイプのユニオン」 は、有効な整数の可能な値を 1 から 10 の範囲に制限する匿名メンバータイプを含むユニオンの例を示しています。

例34.14 匿名メンバータイプのユニオン

```
<simpleType name="restrictedOrderNumUnion">
  <union memberTypes="xsd:string">
    <simpleType>
      <restriction base="xsd:int">
        <minInclusive value="1" />
        <maxInclusive value="10" />
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

Java へのマッピング

XML Schema の結合タイプは Java **String** オブジェクトにマッピングされます。デフォルトでは、Apache CXF は生成されたオブジェクトの内容を検証しません。Apache CXF でコンテンツを検証するには、「ファセットの実施」 で説明されているようにスキーマ検証を使用するようにランタイムを設定する必要があります。

34.6. 単純型置換

概要

XML では、**xsi:type** 属性を使用して、互換性のあるタイプ間で単純なタイプの置換が可能になります。ただし、単純型から Java プリミティブ型へのデフォルトのマッピングは、単純型の置換を完全にはサポートしていません。ランタイムは基本的な単純型置換を処理できますが、情報は失われます。コードジェネレーターをカスタマイズして、ロスレスの単純型置換を容易にする Java クラスを生成できます。

デフォルトのマッピングとマーシャリング

Java プリミティブ型は型置換をサポートしていないため、単純型から Java プリミティブ型へのデフォルトのマッピングでは、単純型置換をサポートする際に問題が発生します。型を定義するスキーマで許容される場合でも、**int** を必要とする変数に **short** を渡そうとした場合、Java 仮想マシンは動作を止めます。

Java 型システムによって課される制限を回避するために、要素の **xsi:type** 属性の値が以下のいずれかの条件を満たす場合、Apache CXF は単純型置換を許可します。

- 要素のスキーマ型と互換性のあるプリミティブ型を指定します。
- 要素のスキーマタイプから制限によって派生するタイプを指定します。
- 要素のスキーマタイプから拡張によって派生する複合タイプを指定します。

ランタイムが型置換を処理する場合、要素の **xsi:type** 属性に指定される型に関する情報が維持されません。型置換が複合型から単純型への場合、単純型に直接関連する値のみが保持されます。拡張機能によって追加された他の要素と属性はすべて失われます。

ロスレスタイプ置換のサポート

単純型の生成をカスタマイズして、次の方法で単純型置換のロスレスサポートを容易にすることができます。

- **globalBindings** カスタマイズ要素の **mapSimpleTypeDef** を **true** に設定します。
  これは、グローバルスコープで定義されたすべての名前付き単純型の Java 値クラスを作成するようにコードジェネレーターに指示します。

  詳細は、「単純型の Java クラスの生成」 を参照してください。

- **javaType** 要素を **globalBindings** カスタマイズ要素に追加します。
  これは、XML スキーマプリミティブ型のすべてのインスタンスを特定のクラスのオブジェクトにマップするようにコードジェネレーターに指示します。

  詳細は、「XML スキーマプリミティブの Java クラスの指定」 を参照してください。

- カスタマイズする特定の要素に **baseType** カスタマイズ要素を追加します。
  baseType カスタム要素を使用すると、プロパティーを表すために生成された Java タイプを指定できます。単純な型の置換に最適な互換性を確保するには、ベースタイプとして **java.lang.Object** を使用します。

  詳細は、「要素または属性の基本タイプの指定」 を参照してください。

第35章 複雑なタイプの使用

概要

複合型には複数の要素を含めることができ、それらは属性を持つことができます。それらは、型定義によって表されるデータを保持できる Java クラスにマップされます。通常、マッピングは、コンテンツモデルの要素と属性を表す一連のプロパティーを持つ Bean へのマッピングです。

35.1. 基本的な複合型マッピング

概要

XML スキーマの複合型は、単純な型よりも複雑な情報を含む構造を定義します。最も単純な複合型は、属性を持つ空の要素を定義します。より複雑な複合型は、要素のコレクションで設定されています。

デフォルトでは、XML スキーマ複合型は Java クラスにマップされ、XML スキーマ定義にリストされている各要素と属性を表すメンバー変数があります。クラスには、メンバー変数ごとにセッターとゲッターがあります。

XML スキーマでの定義

XML Schema の複雑なタイプは、**complexType** 要素を使用して定義されます。**complexType** 要素は、データの構造を定義するために使用される残りの要素をラップします。名前付き型定義の親要素として、または要素に保存されている情報の構造を匿名的に定義する **element** 要素の子として表示することができます。**complexType** 要素を使用して名前付き型を定義する場合は、**name** 属性を使用する必要があります。**name** 属性は、タイプを参照するための一意の識別子を指定します。

1 つ以上の要素を含む複合型定義には、表35.1「複合型で要素がどのように表示されるかを定義するための要素」 で説明されている子要素の 1 つがあります。これらの要素は、指定された要素がタイプのインスタンスでどのように表示されるかを決定します。

表35.1 複合型で要素がどのように表示されるかを定義するための要素

要素	説明
all	複合型の一部として定義されたすべての要素は、型のインスタンスに表示される必要があります。ただし、それらは任意の順序で表示できます。
choice	複合型の一部として定義された要素の 1 つだけが、型のインスタンスに表示されます。
sequence	複合型の一部として定義されたすべての要素は、型のインスタンスに表示される必要があり、型定義で指定された順序でも表示される必要があります。

注記

複合型定義が属性のみを使用する場合は、表35.1「複合型で要素がどのように表示されるかを定義するための要素」 で説明されている要素の 1 つは必要ありません。

要素の表示方法を決定したら、1つまたは複数の **element** 要素の子を定義に追加して、要素を定義します。

例35.1「XML スキーマ複合型」 は、XML スキーマでの複合型定義を示しています。

例35.1 XML スキーマ複合型

```
<complexType name="sequence">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="street" type="xsd:short" />
    <element name="city" type="xsd:string" />
    <element name="state" type="xsd:string" />
    <element name="zipCode" type="xsd:string" />
  </sequence>
</complexType>
```

Java へのマッピング

XML スキーマの複合型は Java クラスにマップされます。複合型定義の各要素は、Java クラスのメンバー変数にマップされます。複合型の要素ごとに、ゲッターメソッドとセッターメソッドも生成されます。

生成されたすべての Java クラスは **@XmlType** アノテーションでデコレートされます。マッピングが名前付きの複合型の場合は、アノテーション **name** は **complexType** 要素の **name** 属性の値に設定されます。複合型が要素定義の一部として定義されている場合、**@XmlType** アノテーションの **name** プロパティーは、**element** 要素の **name** 属性の値になります。

注記

「インラインタイプの要素の Java マッピング」 で説明されているように、生成されるクラスが要素定義の一部として定義された複合型に対して生成される場合、クラスに **@XmlRootElement** アノテーションが付けられます。

XML スキーマ複合型の要素を処理する方法を示すガイドラインをランタイムに提供するために、コードジェネレーターは、クラスとそのメンバー変数を装飾するために使用されるアノテーションを変更します。

すべての複合型

すべての複合型は、**all** 要素を使用して定義されます。それらは次のようにアノテーションが付けられています。

- **@XmlType** アノテーションの **propOrder** プロパティーは空です。
- 各要素は **@XmlElement** アノテーションで装備されています。
- **@XmlElement** アノテーションの **required** プロパティーが **true** に設定されています。

例35.2「すべての複雑なタイプのマッピング」 は、2 つの要素を持つすべての複合型のマッピングを示しています。

例35.2 すべての複雑なタイプのマッピング

```
@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;
    }
}
```

選択複合タイプ

選択複合型は、**choice** 要素を使用して定義されます。それらは次のようにアノテーションが付けられています。

- **@XmlType** アノテーションの **propOrder** プロパティーは、XML Schema 定義に表示される順序で要素の名前をリスト表示します。
- メンバー変数にはアノテーションが付けられていません。
  例35.3「選択複合型のマッピング」 は、2 つの要素を持つ選択複合型のマッピングを示しています。

例35.3 選択複合型のマッピング

```
@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;
    }

}
```

シーケンスコンプレックスタイプ

シーケンスのコンプレックスタイプは、**sequence** 要素を使用して定義します。次のようにアノテーションが付けられています。

- **@XmlType** アノテーションの **propOrder** プロパティーは、XML Schema 定義に表示される順序で要素の名前をリスト表示します。
- 各要素は **@XmlElement** アノテーションで装備されています。
- **@XmlElement** アノテーションの **required** プロパティーが **true** に設定されています。

例35.4「シーケンス複合型のマッピング」 は、例35.1「XML スキーマ複合型」 で定義された複合型のマッピングを示します。

例35.4 シーケンス複合型のマッピング

```
@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

概要

Apache CXF は **complexType** 要素のスコープ内での **attribute** 要素と **attributeGroup** 要素の使用をサポートしています。XML ドキュメントの属性宣言の構造を定義する場合、タグに含まれる値ではなく、タグ内で指定された情報を追加する手段を提供します。例えば、XML 要素の<value currency="euro">410<\value>を XML Schema で記述する場合、例35.5「XML スキーマの定義と属性」のように **attribute** 要素を用いて **currency** 属性を記述します。

attributeGroup 要素を使用すると、スキーマで定義されるすべてのコンプレックスタイプで参照できる再利用可能な属性のグループを定義できます。たとえば、すべての属性 **category** および **pubDate** を使用する一連の要素を定義する場合、これらの属性を持つ属性グループを定義して、それらを使用するすべての要素で参照することができます。これは、例35.7「属性グループの定義」 に示されています。

アプリケーションロジック開発で使用するデータタイプを記述する場合、**use** 属性は **optional** または **required** のいずれかに設定される属性は構造の要素として処理されます。複合型の説明に含まれる属性宣言ごとに、適切なゲッターメソッドとセッターメソッドとともに、属性のクラスに要素が生成されます。

XML スキーマでの属性の定義

XML Schema **属性** 要素には、属性の特定に使用される必須の属性である **name** が 1 つあります。また、表35.2「XML スキーマで属性を定義するために使用されるオプションの属性」 で説明されている 4 つのオプションの属性があります。

表35.2 XML スキーマで属性を定義するために使用されるオプションの属性

属性	説明
use	属性が必要かどうかを指定します。有効な値は **required**、**optional**、または **prohibited** です。**optional** はデフォルト値です。
type	属性が取ることができる値のタイプを指定します。使用しない場合は、属性のスキーマタイプをインラインで定義する必要があります。
default	属性に使用するデフォルト値を指定します。これは、**attribute** 要素の **use** 属性が **optional** に設定されている場合にのみ使用されます。
固定:	属性に使用する固定値を指定します。これは、**attribute** 要素の **use** 属性が **optional** に設定されている場合にのみ使用されます。

例35.5「XML スキーマの定義と属性」 は、値が文字列である属性通貨を定義する属性要素を示しています。

例35.5 XML スキーマの定義と属性

```
<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>
```

type 要素から **attribute** 属性が省略された場合、データの形式をインラインで記述する必要があります。例35.6「インラインデータの説明を含む属性」 は、**autobiography**、**non-fiction**、**fiction** のいずれかの値を取ることができる **category** という属性を持つ **attribute** 要素を示しています。

例35.6 インラインデータの説明を含む属性

```
<attribute name="category" use="required">
  <simpleType>
    <restriction base="xsd:string">
```

```
      <enumeration value="autobiography"/>
      <enumeration value="non-fiction"/>
      <enumeration value="fiction"/>
    </restriction>
  </simpleType>
</attribute>
```

XML スキーマでの属性グループの使用

複合型定義での属性グループの使用は、2 段階のプロセスです。

1. 属性グループを定義します。
   属性グループは、多くの **attribute** 子要素を持つ **attributeGroup** 要素を使用して定義されます。**attributeGroup** には、属性グループの参照に使用される文字列を定義する **name** 属性が必要です。**attribute** 要素は、属性グループのメンバーを定義するもので、「XML スキーマでの属性の定義」 のように指定します。例35.7「属性グループの定義」 に、属性グループ **catalogIndecies** の記述を示します。属性グループには、**category** (任意) および **pubDate** (必須) の 2 つのメンバーが含まれます。

   例35.7 属性グループの定義

   ```
   <attributeGroup name="catalogIndices">
     <attribute name="category" type="catagoryType" />
     <attribute name="pubDate" type="dateTime"
                use="required" />
   </attributeGroup>
   ```

2. 複合型の定義で属性グループを使用します。
   attributeGroup 属性とともに **ref** 要素を使用して、複合型定義で属性グループを使用します。**ref** 属性の値は、タイプ定義の一部として使用する属性グループの名前です。たとえば、複合型 **dvdType** で属性グループ **catalogIndecies** を使用する場合は、例35.8「属性を持つ複合型」に示されるように <attributeGroup ref="catalogIndecies" /> を使用します。

   例35.8 属性を持つ複合型

   ```
   <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>
   ```

属性を Java にマッピングする

属性は、メンバー要素が Java にマップされるのとほぼ同じ方法で Java にマップされます。必須属性とオプション属性は、生成された Java クラスのメンバー変数にマップされます。メンバー変数は **@XmlAttribute** アノテーションで禁止されています。属性が必須な場合、**@XmlAttribute** アノテーションの **required** プロパティーは **true** に設定されます。

例35.9「techDoc Description」 で定義された複合型に示す 例35.10「techDocJava クラス」 は Java クラスにマップされます。

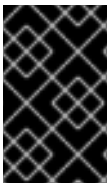
例35.9 techDoc Description

```
<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 techDocJava クラス

```
@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;
    }
}
```

例35.10「techDocJava クラス」 にあるように、**default** 属性と **fixed** 属性は、コードジェネレーターに対し、属性用に生成されたジッターメソッドにコードを追加するよう指示します。この追加コードは、値が設定されていない場合に指定された値が返されることを保証します。



重要

fixed 属性は **default** 属性と同じように扱われます。**fixed** 属性を Java 定数として処理させる場合は、「固定値属性マッピングのカスタマイズ」 に記載のカスタマイズを使用できます。

属性グループの Java へのマッピング

属性グループは、グループのメンバーが型定義で明示的に使用されているかのように Java にマップされます。属性グループに 3 つのメンバーがあり、それが複合型で使用されている場合、その型に対して生成されるクラスには、属性グループの各メンバーのゲッターメソッドとセッターメソッドとともにメンバー変数が含まれます。たとえば、例35.8「属性を持つ複合型」で定義される複合型の場合、例35.11「dvdType Java Class」に示されているように、属性グループのメンバーをサポートするために、Apache CXF はメンバー変数 **category** と **pubDate** が含まれるクラスを生成します。

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. 単純型から複雑型を導出する

概要

Apache CXF は、単純型からの複合型の派生をサポートしています。単純型には、定義上、サブ要素も属性もありません。したがって、単純型から複合型を導出する主な理由の 1 つは、単純型に属性を追加することです。

単純型から複合型を導出する方法は 2 つあります。

- 拡張子別
- 制限別

拡張による派生

例35.12「拡張による単純型からの複雑型の導出」に、currency 属性を追加するために、エクステンションにより **xsd:decimal** プリミティブ型から派生した複合型 **internationalPrice** の例を示します。

例35.12 拡張による単純型からの複雑型の導出

```
<complexType name="internationalPrice">
    <simpleContent>
        <extension base="xsd:decimal">
            <attribute name="currency" type="xsd:string"/>
        </extension>
    </simpleContent>
    </complexType>
```

simpleContent 要素は、新規タイプにサブ要素が含まれていないことを示します。**extension** 要素は、新しい型が **xsd:decimal** を拡張することを指定します。

制限による導出

例35.13「制限による単純型からの複雑型の導出」 は、**xsd:string** からの制限によって派生する複合型の **idType** の例を示しています。定義される型は、**xsd:string** の値を長さ 10 文字の値に制限します。また、タイプに属性を追加します。

例35.13 制限による単純型からの複雑型の導出

```
<complexType name="idType">
  <simpleContent>
    <restriction base="xsd:string">
     <length value="10" />
     <attribute name="expires" type="xsd:dateTime" />
    </restriction>
  </simpleContent>
</complexType>
```

例35.12「拡張による単純型からの複雑型の導出」 の場合のように、**simpleContent** 要素は新しい型に子が含まれていないことを示します。この例では、**restriction** 要素を使用して、新しいタイプで使用される可能な値を制限します。**attribute** 要素は、新しい型に要素を追加します。

Java へのマッピング

単純型から派生する複合型は、**@XmlType** アノテーションが付けられた Java クラスにマッピングされます。生成されるクラスには、複合型が派生する単純型のメンバー変数 **value** が含まれます。メンバー変数には **@XmlValue** アノテーションが付けられます。また、このクラスには **getValue()** メソッドと **setValue()** メソッドがあります。さらに、生成されたクラスには、単純型を拡張する属性ごとに、メンバー変数、および関連するゲッターメソッドとセッターメソッドがあります。

例35.14「idType Java Class」 に、例35.13「制限による単純型からの複雑型の導出」 で定義される **idType** 型に生成される Java クラスを示します。

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. 複合型からの複合型の導出

概要

XML スキーマを使用すると、**complexContent** 要素を使用して他の複雑な型を拡張するか、これを制限することで、新しい複雑なタイプを引き出すことができます。派生した複合型を表す Java クラスを生成する場合、Apache CXF は基本型のクラスを拡張します。このようにして、生成された Java コードは、XML スキーマで意図された継承階層を保持します。

スキーマ構文

complexContent 要素と **extension** 要素または **restriction** 要素のいずれかを使用して、複雑なタイプを他の複雑なタイプから派生します。**complexContent** 要素は、含まれるデータ記述に複数のフィールドが含まれていることを指定します。**extension** 要素と **complexContent** 要素の子である **restriction** 要素は、新しいタイプを作成するために変更されたベースタイプを指定します。ベースタイプは **base** 属性で指定されます。

複合型の拡張

コンプレックスタイプを拡張するには、**extension** 要素を使用して、新しいタイプを設定する追加の要素と属性を定義します。複合型の説明で許可されるすべての要素は、新しい型の定義の一部として許可されます。たとえば、新しい型に匿名の列挙を追加することや、**choice** 要素を使用して新しいフィールドの 1 つだけを一度に有効にできることを指定することが可能です。

例35.15「拡張による複合型の導出」に、**widgetOrderInfo** と **widgetOrderBillInfo** の 2 つの複合型を定義する XML スキーマフラグメントを示します。**widgetOrderBillInfo** は、**widgetOrderInfo** を拡張して **orderNumber** と **amtDue** の 2 つの新しい要素を含むようにしたものです。

例35.15 拡張による複合型の導出

```
<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>
```

複合型の制限

複合型を制限するには、**restriction** 要素を使用してベース型の要素または属性で使用可能な値を制限します。複合型を制限する場合は、基本型のすべての要素と属性をリストする必要があります。要素ごとに、定義に制限属性を追加できます。たとえば、**maxOccurs** 属性を要素に追加して、発生回数を制限できます。**fixed** 属性を使用して、1 つまたは複数の要素が事前に決定された値を持つように強制することもできます。

例35.16「制限による複合型の定義」 は、別の複合型を制限して複合型を定義する例を示しています。**city** 要素、**state** 要素、および **zipCode** 要素の値は固定なので、制限された型 **wallawallaAddress** は、Washington 州 Walla Walla のアドレスにのみ使用できます。

例35.16 制限による複合型の定義

```
<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>
```

-

Java へのマッピング

すべての複合型と同様に、Apache CXF は、別の複合型から派生した複合型を表すクラスを生成します。派生複合型用に生成された Java クラスは、基本複合型をサポートするために生成された Java クラスを拡張します。ベース Java クラスも **@XmlSeeAlso** アノテーションを組み込むように変更されます。ベースクラスの **@XmlSeeAlso** アノテーションは、ベースクラスを拡張するすべてのクラスをリスト表示します。

新しい複合型が拡張によって派生する場合、生成されたクラスには、追加されたすべての要素と属性のメンバー変数が含まれます。新しいメンバー変数は、他のすべての要素と同じマッピングに従って生成されます。

新しい複合型が制限によって派生した場合、生成されたクラスには新しいメンバー変数がありません。生成されたクラスは、追加機能を提供しないシェルになります。XML スキーマで定義された制限が適用されていることを確認するのは完全にあなた次第です。

たとえば、例35.15「拡張による複合型の導出」 のスキーマにより、**WidgetOrderInfo** および **WidgetBillOrderInfo** の 2 つの Java クラスが生成されます。 **WidgetOrderBillInfo** は **widgetOrderInfo** の拡張で派生しているため、**WidgetOrderBillInfo** は **WidgetOrderInfo** を拡張します。例35.17「WidgetOrderBillInfo」 は、**widgetOrderBillInfo** の生成されたクラスを示しています。

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. 発生の制約

35.5.1. 発生制約をサポートするスキーマ要素

XML スキーマを使用すると、複合型定義を設定する 4 つの XML スキーマ要素の出現制約を指定できます。

- 「すべての要素の発生制約」
- 「選択要素の発生制約」
- 「要素の発生制約」
- 「シーケンスの発生制約」

35.5.2. すべての要素の発生制約

XML スキーマ

all 要素で定義された複合型では、**all** 要素で定義された構造を複数回使用することはできません。ただし、**all** 要素の **minOccurs** 属性を **0** に設定することで、all 要素で定義される構造をオプションにすることができます。

Java へのマッピング

all 要素の **minOccurs** 属性を **0** に設定すると、生成された Java クラスには影響しません。

35.5.3. 選択要素の発生制約

概要

デフォルトでは、**choice** 要素の結果は、コンプレックスタイプのインスタンスで 1 回のみ表示されます。**choice** 要素で定義された構造を表現するために選ばれた要素の出現回数は、**minOccurs** 属性と **mxOccurs** 属性で変更することができます。これらの属性を使用して、複合型のインスタンスで選択型が 0 回から無制限に発生する可能性があることを指定できます。選択タイプ用に選択された要素は、タイプの出現ごとに同じである必要はありません。

XML スキーマでの使用

minOccurs 属性は、選択タイプを表示する必要のある最小回数を指定します。その値は任意の正の整数にすることができます。**minOccurs** 属性を **0** に設定すると、選択タイプはコンプレックスタイプのインスタンス内で表示する必要がないことを意味します。

maxOccurs 属性は、選択タイプを表示できる最大回数を指定します。この値には、ゼロ以外の正の整数または **unbounded** を設定できます。**maxOccurs** 属性を **unbounded** に設定すると、選択タイプが無限の回数を表示できることを示します。

例35.18「発生の制約の選択」に、choice の発生の制約が設定された choice 型の定義 **ClubEvent** を示します。全体的な選択タイプは、0 から無制限の時間まで繰り返すことができます。

例35.18 発生の制約の選択

```
<complexType name="ClubEvent">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="MemberName" type="xsd:string"/>
    <element name="GuestName" type="xsd:string"/>
  </choice>
</complexType>
```

Java へのマッピング

単一インスタンス選択構造とは異なり、複数回発生する可能性のある XML スキーマ選択構造は、単一のメンバー変数を使用して Java クラスにマップされます。この単一メンバー変数は、シーケンスの複数の発生に対するすべてのデータを保持する **List<T>** オブジェクトです。たとえば、例35.18「発生の制約の選択」 で定義されたシーケンスが 2 回発生した場合、リストには 2 つの項目が含まれます。

Java クラスのメンバー変数の名前は、メンバー要素の名前を連結することにより引き出されます。要素名は **Or** で区切られ、変数名の最初の文字は小文字に変換されます。例えば、例35.18「発生の制約の選択」 から生成されるメンバー変数の名前は **memberNameOrGuestName** となります。

リストに格納されるオブジェクトのタイプは、メンバー要素の型間の関係によって異なります。以下に例を示します。

- メンバー要素が同じタイプの場合は、生成されたリストに **JAXBElement<T>** オブジェクトが含まれます。**JAXBElement<T>** オブジェクトのベースタイプは、メンバー要素のタイプの通常のマッピングによって決定されます。
- メンバー要素が異なる型であり、それらの Java 表現が共通のインターフェイスを実装している場合、リストには共通のインターフェイスのオブジェクトが含まれます。
- メンバー要素が異なる型であり、それらの Java 表現が共通のベースクラスを拡張している場合、リストには共通のベースクラスのオブジェクトが含まれます。
- 他の条件が満たされていない場合は、リストに **Object** オブジェクトが含まれます。

生成された Java クラスには、メンバー変数のゲッターメソッドのみが含まれます。getter メソッドは、ライブリストへの参照を返します。返されたリストに変更を加えると、実際のオブジェクトに影響します。

Java クラスは **@XmlType** アノテーションで禁止されています。アノテーションの **name** プロパティーは、XML Schema 定義の親要素の **name** 属性の値に設定されます。アノテーションの **propOrder** プロパティーには、シーケンス内の要素を表す単一のメンバー変数が含まれます。

choice 構造の要素を表すメンバー変数には、**@XmlElements** アノテーションが付けられま

す。**@XmlElements** アノテーションには、**@XmlElement** アノテーションのコンマ区切りリストが含まれます。このリストには、型の XML スキーマ定義で定義されたメンバー要素ごとに **@XmlElement** アノテーションが 1 つ含まれます。リストの **@XmlElement** アノテーションは、**name** プロパティーが XML Schema **element** エレメントの **name** 属性の値に設定され、**type** プロパティーが XML Schema **element** エレメントのタイプをマッピングした結果の Java クラスに設定されています。

例35.19「発生制約のある選択構造の Java 表現」 は、例35.18「発生の制約の選択」 で定義されている XML スキーマ選択構造の Java マッピングを示しています。

例35.19 発生制約のある選択構造の Java 表現

```
@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 を 0 に設定

minOccurs 要素のみが指定され、その値が **0** の場合、コードジェネレーターは **minOccurs** 属性が設定されていないかのように Java クラスを生成します。

35.5.4. 要素の発生制約

概要

要素要素の **minOccurs** 属性と **maxOccurs** 属性を使用して、複雑なタイプで特定の **要素** が表示される回数を指定できます。両方の属性のデフォルト値は **1** です。

minOccurs を 0 に設定

複合型のメンバー要素の **minOccurs** 属性のいずれかを **0** に設定すると、対応する Java メンバー変数に付けられる **@XmlElement** アノテーションが変更されます。**required** プロパティーを **true** に設定する代わりに、**@XmlElement** アノテーションの **required** プロパティーが **false** に設定されます。

minOccurs が 1 より大きい値に設定されている

XML スキーマでは、要素の **minOccurs** 属性を複数の値に設定すると、タイプのインスタンスで

element 要素を複数回発生させるように指定できます。ただし、生成された Java クラスは XML スキーマ制約をサポートしません。Apache CXF は、**minOccurs** 属性が設定されていないかのように、サポートする Java メンバー変数を生成します。

maxOccurs が設定されている要素

メンバー要素をコンプレックスタイプのインスタンスで複数回表示するには、要素の **maxOccurs** 属性を 1 よりも大きい値に設定します。**maxOccurs** 属性の値を **unbounded** に設定し、member 要素に無制限の回数を表示できるように指定できます。

コードジェネレーターは、**maxOccurs** 属性が 1 を超える値に設定されたメンバー要素を **List<T>** オブジェクトである Java メンバー変数にマッピングします。リストの基本クラスは、要素のタイプを Java にマッピングすることによって決定されます。XML スキーマのプリミティブ型の場合には、ラッパークラスは 「ラッパークラス」 のように使用されます。たとえば、メンバー要素の型が **xsd:int** の場合、生成されるメンバー変数は **List<Integer>** オブジェクトになります。

35.5.5. シーケンスの発生制約

概要

デフォルトでは、**sequence** 要素の内容は、複合型のインスタンスで 1 回のみ表示されます。その **minOccurs** 属性およびその **maxOccurs** 属性を使用して、**sequence** 要素で定義された要素のシーケンスが表示を許される回数を変更することができます。これらの属性を使用して、複合型のインスタンスでシーケンス型が 0 回から無制限に発生する可能性があることを指定できます。

XML スキーマの使用

minOccurs 属性は、定義された複合型のインスタンスでシーケンスが発生しなければならない最小回数を指定します。その値は任意の正の整数にすることができます。**minOccurs** 属性を **0** に設定すると、シーケンスがコンプレックスタイプのインスタンス内で表示されないことを示しています。

maxOccurs 属性は、定義されたコンプレックスタイプのインスタンスでシーケンスが発生する回数の上限を指定します。この値には、ゼロ以外の正の整数または **unbounded** を設定できます。**maxOccurs** 属性を **unbounded** に設定すると、シーケンスが無限に見えるようになります。

例35.20「発生制約のあるシーケンス」に、シーケンスの発生の制約が設定された sequence 型の定義 **CultureInfo** を示します。シーケンスは 0 から 2 回繰り返すことができます。

例35.20 発生制約のあるシーケンス

```
<complexType name="CultureInfo">
  <sequence minOccurs="0" maxOccurs="2">
    <element name="Name" type="string"/>
    <element name="Lcid" type="int"/>
  </sequence>
</complexType>
```

Java へのマッピング

単一インスタンスシーケンスとは異なり、複数回発生する可能性のある XML スキーマシーケンスは、単一のメンバー変数を使用して Java クラスにマップされます。この単一メンバー変数は、シーケンスの複数の発生に対するすべてのデータを保持する **List<T>** オブジェクトです。たとえば、例35.20「発

生制約のあるシーケンス」 で定義されたシーケンスが 2 回発生した場合、リストには 4 つの項目が含まれます。

Java クラスのメンバー変数の名前は、メンバー要素の名前を連結することにより引き出されます。要素名は **And** で区切られ、変数名の最初の文字が小文字に変換されます。たとえば、例35.20「発生制約のあるシーケンス」 から生成されたメンバー変数の名前は **nameAndLcid** になります。

リストに格納されるオブジェクトのタイプは、メンバー要素の型間の関係によって異なります。以下に例を示します。

- メンバー要素が同じタイプの場合は、生成されたリストに **JAXBElement<T>** オブジェクトが含まれます。**JAXBElement<T>** オブジェクトのベースタイプは、メンバー要素のタイプの通常のマッピングによって決定されます。
- メンバー要素が異なる型であり、それらの Java 表現が共通のインターフェイスを実装している場合、リストには共通のインターフェイスのオブジェクトが含まれます。
- メンバー要素が異なる型であり、それらの Java 表現が共通のベースクラスを拡張している場合、リストには共通のベースクラスのオブジェクトが含まれます。
- 他の条件が満たされていない場合は、リストに **Object** オブジェクトが含まれます。

生成された Java クラスには、メンバー変数のゲッターメソッドのみがあります。getter メソッドは、ライブリストへの参照を返します。返されたリストに加えられた変更は、実際のオブジェクトに影響します。

Java クラスは **@XmlType** アノテーションで禁止されています。アノテーションの **name** プロパティーは、XML Schema 定義の親要素の **name** 属性の値に設定されます。アノテーションの **propOrder** プロパティーには、シーケンス内の要素を表す単一のメンバー変数が含まれます。

シーケンスの要素を表すメンバー変数には、**@XmlElements** アノテーションが付けられます。**@XmlElements** アノテーションには、**@XmlElement** アノテーションのコンマ区切りリストが含まれます。このリストには、型の XML スキーマ定義で定義されたメンバー要素ごとに **@XmlElement** アノテーションが 1 つ含まれます。リストの **@XmlElement** アノテーションは、**name** プロパティーが XML Schema **element** エレメントの **name** 属性の値に設定され、**type** プロパティーが XML Schema **element** エレメントのタイプをマッピングした結果の Java クラスに設定されています。

例35.21「発生制約のあるシーケンスの Java 表現」 は、例35.20「発生制約のあるシーケンス」 で定義された XML スキーマシーケンスの Java マッピングを示します。

例35.21 発生制約のあるシーケンスの Java 表現

```
@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 を 0 に設定

minOccurs 要素のみが指定され、その値が **0** の場合、コードジェネレーターは **minOccurs** 属性が設定されていないかのように Java クラスを生成します。

35.6. モデルグループの使用

概要

XML スキーマモデルグループは、ユーザー定義の複合型から要素のグループを参照できる便利なショートカットです。たとえば、アプリケーションの複数の型に共通する要素のグループを定義して、そのグループを繰り返し参照できます。モデルグループは **group** 要素を使用して定義され、複合型の定義に似ています。モデルグループの Java へのマッピングも、複合型のマッピングに似ています。

XML スキーマでのモデルグループの定義

group 要素と **name** 属性を使用して、XML スキーマでモデルグループを定義します。**name** 属性の値は、スキーマ全体でグループを参照するために使用される文字列です。**group** 要素は、**complexType** 要素と同様に、**sequence** 要素、**all** 要素、**choice** 要素を直接の子として持つことができます。

子要素内で、**element** 要素を使用してグループのメンバーを定義します。グループのメンバーごとに、1 つの **element** 要素を指定します。グループメンバーは、**minOccurs** や **maxOccurs** など、**element** 要素の標準的な属性を使用することができます。そのため、グループに 3 つの要素があり、その 1 つが最大 3 回発生できる場合は、グループに 3 つの **element** 要素を定義し、そのうちの 1 つには maxOccurs="3" を使用します。例35.22「XML スキーマモデルグループ」 は、3 つの要素を持つモデルグループを示しています。

例35.22 XML スキーマモデルグループ

```
<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>
```

タイプ定義でのモデルグループの使用

モデルグループが定義されると、それを複合型定義の一部として使用できます。コンプレックスタイプ定義でモデルグループを使用するには、**ref** 属性を指定して **group** 要素を使用します。**ref** 属性の値は、定義されたときにグループに指定された名前です。たとえば、例35.22「XML スキーマモデルグ

ループ」 で定義されたグループを使用するには 例35.23「モデルグループを持つ複合型」 に示すように、<group ref="tns:passenger" /> を使用します。

例35.23 モデルグループを持つ複合型

```
<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>
```

モデルグループがタイプ定義で使用される場合、グループはタイプのメンバーになります。したがって、**reservation** のインスタンスには 4 つのメンバー要素があります。最初の要素は **passenger** 要素で、例35.22「XML スキーマモデルグループ」 に示すグループで定義されるメンバー要素が含まれます。**reservation** のインスタンスの例は、例35.24「モデルグループを持つタイプのインスタンス」 に記載されています。

例35.24 モデルグループを持つタイプのインスタンス

```
<reservation>
  <passenger> <name>A. Smart</name> <clubNum>99</clubNum> <seatPref>isle1</seatPref>
</passenger>
  <origin>LAX</origin>
  <destination>FRA</destination>
  <fltNum>34567</fltNum>
</reservation>
```

Java へのマッピング

デフォルトでは、モデルグループは、複合型定義に含まれている場合にのみ Java アーティファクトにマップされます。モデルグループを含む複合型のコードを生成する場合、Apache CXF は、モデルグループのメンバー変数をその型用に生成された Java クラスに含めるだけです。モデルグループを表すメンバー変数には、モデルグループの定義に基づいてアノテーションが付けられます。

例35.25「グループで入力する」に、例35.23「モデルグループを持つ複合型」 で定義される複合型に生成される Java クラスを示します。

例35.25 グループで入力する

```
@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;
    }
```

複数回発生

group 要素の **maxOccurs** 属性を 1 より大きい値に設定すると、モデルグループが複数回表示されるように指定できます。Apache CXF がモデルグループを複数発生できるようにするには、モデルグループを **List<T>** オブジェクトにマッピングします。**List<T>** オブジェクトは、グループの最初の子のルールに従って生成されます。

- グループが **sequence** 要素を使用して定義される場合は、「シーケンスの発生制約」 を参照してください。
- グループが **choice** 要素を使用して定義されている場合は、「選択要素の発生制約」 を参照してください。

第36章 ワイルドカードタイプの使用

概要

スキーマ作成者がバインディング要素または属性を定義された型に延期したい場合があります。このような場合、XML スキーマはワイルドカードプレースホルダーを指定するための 3 つのメカニズムを提供します。これらはすべて、XML スキーマ機能を保持する方法で Java にマップされます。

36.1. 任意の要素の使用

概要

XML Schema **any** 要素は、複雑なタイプ定義にワイルドカードの配置ホルダーを作成するために使用されます。XML スキーマ **any** 要素に対して XML 要素がインスタンス化されると、有効な XML 要素を使用できます。**any** 要素は、インスタンス化された XML 要素の名前に制約をかけません。

たとえば、例36.1「任意の要素で定義された XML スキーマタイプ」 で定義された複合型が与えられた場合 例36.2「任意の要素を含む XML ドキュメント」 に示す XML 要素のいずれかをインスタンス化できます。

例36.1 任意の要素で定義された XML スキーマタイプ

```
<element name="FlyBoy">
  <complexType>
    <sequence>
      <any />
      <element name="rank" type="xsd:int" />
    </sequence>
  </complexType>
</element>
```

例36.2 任意の要素を含む XML ドキュメント

```
<FlyBoy>
  <learJet>CL-215</learJet>
  <rank>2</rank>
</element>
<FlyBoy>
  <viper>Mark II</viper>
  <rank>1</rank>
</element>
```

XML Schema **any** 要素は、Java **オブジェクト** オブジェクトまたは Java **org.w3c.dom.Element** オブジェクトにマッピングされます。

XML スキーマでの指定

シーケンスの複雑な型や複雑なタイプを定義するときに、**any** 要素を使用できます。多くの場合、**any** 要素は空の要素です。ただし、**annotation** 要素を子として取ることができます。

表36.1「XML スキーマの属性任意の要素」 は、**any** 要素の属性を記述します。

表36.1 XML スキーマの属性任意の要素

属性	説明
namespace	XML ドキュメントで要素をインスタンス化するために使用できる要素の名前空間を指定します。有効な値は、以下の通りです。 **##any** 任意の名前空間の要素を使用できることを指定します。これはデフォルトになります。 **##other** **親要素の名前空間以外** の任意の名前空間の要素を使用できることを指定します。 **##local** 名前空間のない要素を使用する必要があることを指定します。 **##targetNamespace** 親要素の名前空間の要素を使用する必要があることを指定します。 **URI のスペース区切りリスト #\#local および \#\#targetNamespace** リストされた名前空間のいずれかの要素を使用できることを指定します。
maxOccurs	要素のインスタンスが親要素に表示される最大回数を指定します。デフォルト値は **1** です。要素のインスタンスを無制限に表示可能であることを指定するには、属性の値を **unbounded** に設定します。
minOccurs	要素のインスタンスが親要素に表示される最小回数を指定します。デフォルト値は **1** です。
processContents	任意の要素をインスタンス化するために使用される要素を検証する方法を指定します。有効な値は以下のとおりです。 **strict** 要素を適切なスキーマに対して検証する必要があることを指定します。これはデフォルト値です。 **lax** 要素を適切なスキーマに対して検証する必要があることを指定します。検証できない場合、エラーは出力されません。 **skip** 要素を検証しないことを指定します。

例36.3「任意の要素で定義された複合型」に、**any** 要素で定義された複合型を示します。

例36.3 任意の要素で定義された複合型

```
<complexType name="surprisePackage">
  <sequence>
    <any processContents="lax" />
    <element name="to" type="xsd:string" />
    <element name="from" type="xsd:string" />
  </sequence>
</complexType>
```

Java へのマッピング

XML スキーマの **any** 要素により、**any** という名前の Java プロパティーが作成されます。プロパティーには、ゲッターメソッドとセッターメソッドが関連付けられています。作成されるプロパティーのタイプは、要素の **processContents** 属性の値によって異なります。**any** 要素の **processContents** 属性が **skip** に設定されている場合、要素は **org.w3c.dom.Element** オブジェクトにマッピングされます。**processContents** 属性の値がそれ以外の場合、**any** 要素は Java **Object** オブジェクトにマッピングされます。

生成されたプロパティーは **@XmlAnyElement** アノテーションで禁止されています。このアノテーションには、データをマーシャリングする際のランタイムに指示するオプションの **lax** プロパティーがあります。デフォルト値は **false** で、データを **org.w3c.dom.Element** オブジェクトに自動的にマーシャリングするようにランタイムに指示します。**lax** を **true** に設定: ランタイムが、データを JAXB タイプにマーシャリングしようとします。**any** 要素の **processContents** 属性が **skip** に設定されている場合は、**lax** プロパティーはデフォルト値に設定されます。**processContents** 属性の他のすべての値の場合、**lax** は **true** に設定されます。

例36.4「任意の要素を持つ Java クラス」 で定義された複合型がどのように定義されているかを示し、例36.3「任意の要素で定義された複合型」 Java クラスにマップされます。

例36.4 任意の要素を持つ Java クラス

```
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;
    }

}
```

マーシャリング

any 要素の Java プロパティーにその **lax** が **false** に設定されているか、プロパティーが指定されていない場合、ランタイムは XML データの JAXB オブジェクトへの解析を試行しません。データは常に DOM **Element** オブジェクトに保存されます。

any 要素の Java プロパティーの **lax** が **true** に設定されている場合、ランタイムは XML データを適切な JAXB オブジェクトにマーシャリングしようとします。ランタイムは、次の手順を使用して適切な JAXB クラスを識別しようとします。

1. XML 要素の要素タグをランタイムに認識されている要素のリストと照合します。一致するものが見つかった場合、ランタイムは XML データを要素の適切な JAXB クラスにマーシャリングします。
2. XML 要素の **xsi:type** 属性をチェックします。一致するものが見つかった場合、ランタイムは XML 要素をそのタイプの適切な JAXB クラスにマーシャリングします。
3. 一致するものが見つからない場合は、XML データを DOM **Element** オブジェクトにマーシャリングします。

通常、アプリケーションのランタイムは、そのコントラクトに含まれるスキーマから生成されたすべてのタイプを認識しています。これには、コントラクトの **wsdl:types** 要素で定義された型や、インクルージョンによってコントラクトに追加されたデータ型、他のスキーマのインポートによってコントラクトに追加されたデータ型が含まれます。「ランタイムマーシャラーへのクラスの追加」 で説明される **@XmlSeeAlso** アノテーションを使用して、追加の型をランタイムを認識させることもできます。

アンマーシャリング

any 要素の Java プロパティーの **lax** が **false** に設定されているか、プロパティーが指定されていない場合、ランタイムは DOM **Element** オブジェクトだけを受け入れます。他のタイプのオブジェクトを使用しようとすると、マーシャリングエラーが発生します。

any 要素の Java プロパティーの **lax** が **true** に設定されている場合、ランタイムは Java データ型とそれらが表す XML スキーマ構造との間の内部マッピングを使用して、ワイヤに書き込む XML 構造を判断します。ランタイムがクラスを認識し、それを XML スキーマ構造にマッピングできる場合は、データを書き出して **xsi:type** 属性を挿入し、要素に含まれるデータの型を特定します。

ランタイムが Java オブジェクトを既知の XML スキーマ構造にマップできない場合、マーシャリング例外が出力されます。「ランタイムマーシャラーへのクラスの追加」 で説明されている **@XmlSeeAlso** アノテーションを使用して、ランタイムのマッピングに型を追加できます。

36.2. XML スキーマ ANYTYPE タイプの使用

概要

XML スキーマ型 **xsd:anyType** は、すべての XML スキーマ型のルート型です。すべてのプリミティブは、すべてのユーザー定義の複合型と同様に、この型の派生物です。その結果、**xsd:anyType** として

定義された要素には、XML スキーマプリミティブの形式に加えてスキーマドキュメントで定義された複合型のデータが含まれます。

Java では、最も近い一致する型は **Object** クラスです。これは、他のすべての Java クラスがサブタイプ化されるクラスです。

XML スキーマでの使用

xsd:anyType 型は、他の XML スキーマ複合型と同様に使用します。**element** 要素の **type** 要素の値として使用することができます。他のタイプを定義する基本タイプとしても使用できます。

例36.5「ワイルドカード要素を持つ複合型」 に、**xsd:anyType** 型の要素が含まれる複合型の例を示します。

例36.5 ワイルドカード要素を持つ複合型

```
<complexType name="wildStar">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="ship" type="xsd:anyType" />
  </sequence>
</complexType>
```

Java へのマッピング

xsd:anyType 型の要素は、**Object** オブジェクトにマッピングされます。例36.6「ワイルドカード要素の Java 表現」 は、Java クラスへの 例36.5「ワイルドカード要素を持つ複合型」 のマッピングを示しています。

例36.6 ワイルドカード要素の Java 表現

```
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; }
}
```

このマッピングにより、ワイルドカード要素を表すプロパティーに任意のデータを配置できます。Apache CXF ランタイムは、データのマーシャリングとアンマーシャリングを処理して、使用可能な Java 表現にします。

マーシャリング

Apache CXF が XML データを Java 型にマーシャリングする場合、**anyType** 要素を既知の JAXB オブジェクトにマーシャリングしようとします。**anyType** 要素を JAXB で生成されたオブジェクトにマーシャリングできるかどうかを判断するために、ランタイムは要素の **xsi:type** 属性を調べて、要素内のデータ構築に使用される実際の型を判断します。**xsi:type** 属性が存在しない場合、ランタイムは、イントロスペクションによって要素の実際のデータ型の特定を試みます。要素の実際のデータ型が、アプリケーションの JAXB コンテキストで認識されている型の 1 つであると判断された場合、要素は適切な型の JAXB オブジェクトにマーシャリングされます。

ランタイムが要素の実際のデータタイプを判別できない場合や、要素の実際のデータ型が既知のタイプではない場合、ランタイムはコンテンツを **org.w3c.dom.Element** オブジェクトにマーシャルします。次に、DOM APIs を使用して要素のコンテンツを操作する必要があります。

アプリケーションのランタイムは通常、そのコントラクトに含まれるスキーマから生成されたすべてのタイプを認識しています。これには、コントラクトの **wsdl:types** 要素で定義されたタイプ、包含によりコントラクトに追加されたデータタイプ、他のスキーマドキュメントのインポートによりコントラクトに追加されたすべてのタイプが含まれます。「ランタイムマーシャラーへのクラスの追加」 で説明される **@XmlSeeAlso** アノテーションを使用して、追加の型をランタイムを認識させることもできます。

アンマーシャリング

Apache CXF が Java タイプを XML データにアンマーシャリングする場合、Java データタイプとそれらが表す XML スキーマ構造との間の内部マップを使用して、ワイヤに書き込む XML 構造を決定します。ランタイムがクラスを認識し、クラスを XML スキーマ構造にマッピングできる場合は、データを書き出して **xsi:type** 属性を挿入し、要素に含まれるデータの型を特定します。データが **org.w3c.dom.Element** オブジェクトに保存されている場合、ランタイムはオブジェクトによって表される XML 構造を書き込みますが、**xsi:type** 属性は含まれません。

ランタイムが Java オブジェクトを既知の XML スキーマ構造にマップできない場合、マーシャリング例外が出力されます。「ランタイムマーシャラーへのクラスの追加」 で説明されている **@XmlSeeAlso** アノテーションを使用して、ランタイムのマッピングに型を追加できます。

36.3. バインドされていない属性の使用

概要

XML スキーマには、複合型定義内の任意の属性のプレースホルダーを残すことができるメカニズムがあります。このメカニズムを使用すると、任意の属性を持つことができる複合型を定義できます。たとえば、3 つの属性を指定せずに、要素 <robot name="epsilon" />、<robot age="10000" />、または <robot type="weevil" /> を定義するタイプを作成できます。これは、データに柔軟性が必要な場合に特に役立ちます。

XML スキーマでの定義

宣言されていない属性は、**anyAttribute** 要素を使用して XML スキーマで定義されます。属性要素を使用できる場所であればどこでも使用できます。例36.7「宣言されていない属性を持つ複合型」 に示されているように、**anyAttribute** 要素には属性がありません。

例36.7 宣言されていない属性を持つ複合型

```
<complexType name="arbitter">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="rate" type="xsd:float" />
  </sequence>
  <anyAttribute />
</complexType>
```

定義された型 **arbitter** には 2 つの要素があり、任意の型の属性を 1 つ設定できます。例36.8「ワイルドカード属性で定義された要素の例」 に記載されている 3 つの要素は、すべて複合型 **arbitter** から生成できます。

例36.8 ワイルドカード属性で定義された要素の例

```
<officer rank="12"><name>...</name><rate>...</rate></officer>
<lawyer type="divorce"><name>...</name><rate>...</rate></lawyer>
<judge><name>...</name><rate>...</rate></judge>
```

Java へのマッピング

anyAttribute 要素を含む複合型が Java にマップされると、コードジェネレーターは、生成されたクラスに **otherAttributes** というメンバーを追加します。**otherAttributes** は **java.util.Map<QName, String>** タイプであり、マップのライブインスタンスを返すゲッターメソッドを持ちます。ゲッターから返されたマップはライブであるため、マップへの変更は自動的に適用されます。例36.9「宣言されていない属性を持つ複合型のクラス」 は、例36.7「宣言されていない属性を持つ複合型」 で定義された複合型に対して生成されたクラスを示します。

例36.9 宣言されていない属性を持つ複合型のクラス

```
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; }

}
```

宣言されていない属性の操作

生成されたクラスの **otherAttributes** メンバーには、**Map** オブジェクトが設定される必要があります。マップは **QNames** を使用してキーが付けられます。マップを取得すると、オブジェクトに設定されている任意の属性にアクセスして、オブジェクトに新しい属性を設定できます。

例36.10「宣言されていない属性の操作」 は、宣言されていない属性を操作するためのサンプルコードを示しています。

例36.10 宣言されていない属性の操作

```
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「宣言されていない属性の操作」 のコードは、以下を行います。

宣言されていない属性を含むマップを取得します。

属性を操作する QName を作成します。

属性の値をマップに設定します。

属性の 1 つの値を取得します。

第37章 元素置換

概要

XML スキーマ置換グループを使用すると、最上位またはヘッドの要素を置き換えることができる要素のグループを定義できます。これは、共通の基本タイプを共有する複数の要素がある場合、または交換可能である必要がある要素を持つ場合に役立ちます。

37.1. XML スキーマの置換グループ

概要

置換グループは XML スキーマの機能であり、そのスキーマから生成されたドキュメント内の別の要素を置き換えることができる要素を指定できます。置換可能な要素は head 要素と呼ばれ、スキーマのグローバルスコープで定義する必要があります。置換グループの要素は、head 要素と同じタイプ、または head 要素のタイプから派生したタイプである必要があります。

基本的に、置換グループを使用すると、汎用要素を使用して指定できる要素のコレクションを作成できます。たとえば、3 種類のウィジェットを販売する会社の注文システムを構築している場合、3 種類すべてのウィジェットの共通データのセットを含む汎用ウィジェット要素を定義できます。次に、ウィジェットのタイプごとに、より具体的なデータセットを含む置換グループを定義できます。コントラクトでは、ウィジェットのタイプごとに特定の順序付け操作を定義する代わりに、汎用ウィジェット要素をメッセージ部分として指定できます。実際のメッセージが作成されるとき、メッセージには置換グループの任意の要素を含めることができます。

構文

置換グループは、XML スキーマの **element** 要素の **substitutionGroup** 属性を使用して定義されます。**substitutionGroup** 属性の値は、定義される要素が置換する要素の名前です。たとえば、先頭要素が **widget** の場合は、属性 substitutionGroup="widget" を **woodWidget** という名前の要素に追加すると、**widget** 要素が使用されるあらゆる場所で **woodWidget** 要素を置換できることを指定します。これは、例37.1「代替グループの使用」 に示されています。

例37.1 代替グループの使用

```
<element name="widget" type="xsd:string" />
<element name="woodWidget" type="xsd:string"
        substitutionGroup="widget" />
```

タイプの制約

置換グループの要素は、head 要素と同じタイプであるか、head 要素のタイプから派生したタイプである必要があります。たとえば、先頭要素の型が **xsd:int** である場合、置換グループのすべてのメンバーは **xsd:int** 型または **xsd:int** から派生する型である必要があります。例37.2「複雑なタイプの置換グループ」 に示すような置換グループを定義することもできます。ここで、置換グループの要素は、head 要素のタイプから派生したタイプです。

例37.2 複雑なタイプの置換グループ

```
<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" />
```

置換グループの先頭要素 **widget** は、**widgetType** 型として定義されます。置換グループの各要素は、その型のウィジェットの注文に固有のデータを含むように **widgetType** を拡張します。

例37.2「複雑なタイプの置換グループ」のスキーマに基づいて、例37.3「置換グループを使用した XML ドキュメント」の **part** 要素は有効です。

例37.3 置換グループを使用した XML ドキュメント

```
<part>
  <widget>
    <shape>round</shape>
    <color>blue</color>
  </widget>
</part>
<part>
  <plasticWidget>
    <shape>round</shape>
    <color>blue</color>
    <moldProcess>sandCast</moldProcess>
  </plasticWidget>
```

```
</part>
<part>
  <woodWidget>
    <shape>round</shape>
    <color>blue</color>
    <woodType>elm</woodType>
  </woodWidget>
</part>
```

抽象的な頭の要素

スキーマを使用して作成されたドキュメントには表示されない抽象ヘッド要素を定義できます。抽象ヘッド要素は、ジェネリッククラスのより具体的な実装を定義するための基礎として使用されるため、Java の抽象クラスに似ています。抽象ヘッドはまた、最終製品での一般的な要素の使用を防ぎます。

例37.4「抽象ヘッド定義」 に示すように、**element** 要素の **abstract** 属性を **true** に設定して、抽象先頭要素を宣言します。このスキーマを使用すると、有効な **review** 要素に **positiveComment** 要素または **negativeComment** 要素のいずれかを含めることができますが、**comment** 要素を含めることはできません。

例37.4 抽象ヘッド定義

```
<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 の置換グループ

概要

JAXB 仕様で指定されるように、Apache CXF は、Java のネイティブクラス階層を使用し、これに **JAXBElement** クラスのワイルドカード定義に対するサポートを組み合わせて、置換グループをサポートします。置換グループのメンバーはすべて共通のベースタイプを共有する必要があるため、要素のタイプをサポートするために生成されたクラスも共通のベースタイプを共有します。さらに、Apache CXF は先頭要素のインスタンスを **JAXBElement<? extends T>** プロパティーにマッピングします。

生成されたオブジェクトファクトリーメソッド

置換グループを含むパッケージをサポートするために生成されたオブジェクトファクトリーには、置換グループ内の各要素のメソッドがあります。表37.1「JAXB 要素を宣言するためのプロパティーは置換グループのメンバーです」で説明されているように、先頭要素を除く置換グループの各メンバーについ

て、オブジェクトファクトリーメソッドに付けられる **@XmlElementDecl** アノテーションには 2 つの追加プロパティーが含まれます。

表37.1 JAXB 要素を宣言するためのプロパティーは置換グループのメンバーです

プロパティー	説明
substitutionHeadNamespace	ヘッド要素が定義されている名前空間を指定します。
substitutionHeadName	先頭要素の **name** 属性の値を指定します。

置換グループの **@XmlElementDecl** の先頭要素のオブジェクトファクトリーメソッドには、デフォルトの **namespace** プロパティーとデフォルトの **name** プロパティーのみが含まれます。

要素のインスタンス化メソッドに加えて、オブジェクトファクトリーには head 要素を表すオブジェクトをインスタンス化するためのメソッドが含まれています。置換グループのメンバーがすべて複合型である場合、オブジェクトファクトリーには、使用される各複合型のインスタンスをインスタンス化するためのメソッドも含まれています。

例37.5「置換グループのオブジェクトファクトリーメソッド」 は、例37.2「複雑なタイプの置換グループ」 で定義された置換グループのオブジェクトファクトリーメソッドを示します。

例37.5 置換グループのオブジェクトファクトリーメソッド

```
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);
    }

}
```

インターフェイスの置換グループ

置換グループの head 要素が操作のメッセージの 1 つでメッセージ部分として使用される場合、結果のメソッドパラメーターは、その要素をサポートするために生成されたクラスのオブジェクトになります。これは必ずしも **JAXBElement<? extends T>** クラスのインスタンスであるとは限りません。ランタイムは、Java のネイティブ型階層に依存して型置換をサポートし、Java はサポートされていない型を使用しようとする試みをすべてキャッチします。

ランタイムが要素の置換のサポートに必要なすべてのクラスを認識するために、SEI に **@XmlSeeAlso** アノテーションが付けられます。このアノテーションは、マーシャリングのためにランタイムに必要なクラスのリストを指定します。**@XmlSeeAlso** アノテーションの使用に関する詳細は、「ランタイムマーシャラーへのクラスの追加」 を参照してください。

例37.7「置換グループを使用して生成されたインターフェイス」 は、例37.6「置換グループを使用した WSDL インターフェイス」 に示されているインターフェイスに対して生成された SEI を示しています。インターフェイスは、例37.2「複雑なタイプの置換グループ」 で定義された置換グループを使用します。

例37.6 置換グループを使用した WSDL インターフェイス

```
<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 置換グループを使用して生成されたインターフェイス

```
@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「置換グループを使用して生成されたインターフェイス」 に示される SEI は、**@XmlSeeAlso** アノテーションにオブジェクトファクトリーをリスト表示します。名前空間のオブジェクトファクトリーをリスト表示すると、その名前空間に対して生成されたすべてのクラスにアクセスできます。

複雑なタイプの置換グループ

置換グループの先頭要素を複合型の要素として使用する場合、コードジェネレーターは要素を **JAXBElement<? extends T>** プロパティーにマッピングします。置換グループをサポートするために生成された生成されたクラスのインスタンスを含むプロパティーにはマップされません。

たとえば、例37.8「置換グループを使用した複合型」 で定義された複合型結果は、例37.9「置換グループを使用した複合型の Java クラス」 に示す Java クラスになります。複合型は、 例37.2「複雑なタイプの置換グループ」 で定義された置換グループを使用します。

例37.8 置換グループを使用した複合型

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element ref="xsd1:widget"/>
  </sequence>
</complexType>
```

例37.9 置換グループを使用した複合型の Java クラス

```
@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); }

}
```

置換グループプロパティーの設定

置換グループの操作方法は、コードジェネレーターがグループを単純な Java クラスにマッピングしたか、または **JAXBElement<? extends T>** クラスにマッピングしたかによって異なります。要素が生成された値クラスのオブジェクトに単純にマップされる場合、型階層の一部である他の Java オブジェクトを操作するのと同じ方法でオブジェクトを操作します。親クラスの代わりに任意のサブクラスを使用できます。オブジェクトを調べて正確なクラスを判別し、適切にキャストできます。

JAXB 仕様では、生成されたクラスのオブジェクトをインスタンス化するためにオブジェクトファクトリーメソッドを使用することを推奨しています。

コードジェネレーターが置換グループのインスタンスを保持するのに **JAXBElement<? extends T>** オブジェクトを作成する場合、**JAXBElement<? extends T>** オブジェクトに要素の値をラップする必要があります。これを行うための最良の方法は、オブジェクトファクトリーによって提供される要素作成メソッドを使用することです。それらは、その値に基づいて要素を作成するための簡単な手段を提供します。

例37.10「代替グループのメンバーの設定」 は、置換グループのインスタンスを設定するためのコードを示しています。

例37.10 代替グループのメンバーの設定

```
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「代替グループのメンバーの設定」 のコードは、以下を行います。

オブジェクトファクトリーをインスタンス化します。

PlasticWidgetType オブジェクトをインスタンス化します。

JAXBElement<PlasticWidgetType> オブジェクトをインスタンス化して、プラスチックウィジェット要素を保持します。

WidgetOrderInfo オブジェクトをインスタンス化します。

WidgetOrderInfo オブジェクトの **widget** をプラスチックウィジェット要素を保持する **JAXBElement** オブジェクトに設定します。

置換グループプロパティーの値を取得する

オブジェクトファクトリーの手法は、**JAXBElement<? extends T>** オブジェクトから要素の値を抽出する際には役立ちません。**JAXBElement<? extends T>** オブジェクトの **getValue()** メソッドを使用する必要があります。以下のオプションは、**getValue()** メソッドによって返されるオブジェクトの型を決定します。

- 使用できるすべてのクラスの **isInstance()** メソッドを使用して、要素の値オブジェクトのクラスを決定します。
- **JAXBElement<? extends T>** オブジェクトの **getName()** メソッドを使用して、要素の名前を決定します。
  getName() メソッドは QName を返します。要素のローカル名を使用して、値オブジェクトの適切なクラスを決定できます。
- **JAXBElement<? extends T>** オブジェクトの **getDeclaredType()** メソッドを使用して、値オブジェクトのクラスを決定します。
  getDeclaredType() メソッドは、要素の値オブジェクトの **Class** オブジェクトを返します。

警告

値オブジェクトの実際のクラスに関係なく、**getDeclaredType()** メソッドが先頭要素のベースクラスを返す可能性があります。

例37.11「代替グループのメンバーの値を得る」 は、置換グループから値を取得するコードを示しています。要素の値オブジェクトに適切なクラスを決定するために、このサンプルでは要素の **getName()** メソッドを使用します。

例37.11 代替グループのメンバーの値を得る

```
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. ウィジェットベンダーの例

37.3.1. ウィジェット注文インターフェイス

このセクションでは、実際のアプリケーションを解決するために Apache CXF で使用されている置換グループの例を示します。サービスとコンシューマーは、例37.2「複雑なタイプの置換グループ」 で定義されたウィジェット置換グループを使用して開発されます。サービスは、**checkWidgets** と **placeWidgetOrder** の 2 つの操作を提供します。例37.12「ウィジェット注文インターフェイス」 は、注文サービスのインターフェイスを示しています。

例37.12 ウィジェット注文インターフェイス

```
<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「ウィジェットの注文 SEI」 は、インターフェイス用に生成された Java SEI を示しています。

例37.13 ウィジェットの注文 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;
}
```

注記

この例は置換グループの使用のみを示しているため、一部のビジネスロジックは示されていません。

37.3.2. checkWidgets の操作

概要

checkWidgets は、置換グループのヘッドメンバーのパラメーターがある簡単な操作です。この操作は、置換グループのメンバーである個々のパラメーターを処理する方法を示しています。コンシューマーは、パラメーターが置換グループの有効なメンバーであることを確認する必要があります。サービスは、置換グループのどのメンバーが要求で送信されたかを適切に判別する必要があります。

コンシューマーの実装

生成されたメソッドシグネチャーは、置換グループの head 要素のタイプをサポートする Java クラスを使用します。置換グループのメンバー要素は、head 要素と同じ型であるか、head 要素の型から派生した型であるため、置換グループのメンバーをサポートするために生成された Java クラスは、head 要素をサポートするために生成された Java クラスから派生している。Java の型階層は、親クラスの代わりにサブクラスを使用することをネイティブにサポートします。

Apache CXF が置換グループの型および Java の型階層を生成する方法が原因で、クライアントは特別なコードを使用せずに **checkWidgets()** を呼び出すことができます。**checkWidgets()** を呼び出すロジックを開発する場合は、ウィジェットの置換グループに対応するために生成されたクラスの 1 つのオブジェクトを渡すことができます。

例37.14「**checkWidgets()** を呼び出すコンシューマー」 に、**checkWidgets()** を呼び出すコンシューマーを示します。

例37.14 checkWidgets() を呼び出すコンシューマー

```
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);
```

サービス実装

checkWidgets() のサービスの実装は、ウィジェットの記述を **WidgetType** オブジェクトとして取得し、ウィジェットのインベントリーを確認し、ストックされているウィジェットの数を返します。置換グループの実装に使用されるクラスはすべて同じベースクラスから継承されるため、JAXB 固有の API を使用せずに **checkWidgets()** を実装することができます。

widget の置換グループのメンバーをサポートするように生成されたクラスはすべて、**WidgetType** クラスを拡張します。このため、**instanceof** を使用して、渡されたウィジェットの型を決定し、単純に **widgetPart** オブジェクトをより制限のある型にキャストできます (適切であれば)。適切なタイプのオブジェクトを取得したら、適切な種類のウィジェットのインベントリーを確認できます。

例37.15「**checkWidgets()** のサービス実装」 は、可能な実装を示しています。

例37.15 checkWidgets() のサービス実装

```
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 操作

概要

placeWidgetOrder は、置換グループが含まれる 2 つの複合型を使用します。この操作は、Java 実装でそのような構造を使用することを示しています。コンシューマーとサービスの両方が、置換グループのメンバーを取得および設定する必要があります。

コンシューマーの実装

placeWidgetOrder() を呼び出すには、コンシューマーはウィジェットの置換グループの要素が 1 つ含まれるウィジェットの注文を構築する必要があります。ウィジェットをオーダーに追加する場合、コンシューマーは、置換グループの各要素に対して生成されたオブジェクトファクトリーメソッドを使用する必要があります。これにより、ランタイムとサービスが注文を正しく処理できるようになります。たとえば、プラスチックウィジェットに注文が置かれている場合、要素を注文に追加する前に **ObjectFactory.createPlasticWidget()** メソッドを使用して要素を作成します。

例37.16「代替グループメンバーの設定」に、**WidgetOrderInfo** オブジェクトの **widget** プロパティーを設定するためのコンシューマーコードを示します。

例37.16 代替グループメンバーの設定

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

サービス実装

placeWidgetOrder() メソッドは、**WidgetOrderInfo** オブジェクトの形式で注文を受け取り、順番を処理し、**WidgetOrderBillInfo** オブジェクトの形式でコンシューマーに請求書を返します。注文は、プレーンウィジェット、プラスチックウィジェット、または木製ウィジェットの場合があります。注文されたウィジェットの種類は、どの型のオブジェクトが **widgetOrderForm** オブジェクトの **widget** プロパティーに保存されているかによって決まります。**widget** プロパティーは 置換グループで、**widget** 要素、**woodWidget** 要素、または **plasticWidge**t 要素を含めることができます。

実装は、可能な要素のどれが順序で格納されるかを決定する必要があります。要素の QName を決定する **JAXBElement<? extends T>** オブジェクトの **getName()** メソッドを使用して、これを実現できます。次に、QName を使用して、置換グループ内のどの要素が順序になっているのかを判別できます。請求書に含まれる要素がわかれば、その値を適切なタイプのオブジェクトに抽出できます。

例37.17「**placeWidgetOrder()** の実装」 は、可能な実装を示しています。

例37.17 placeWidgetOrder() の実装

```
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() の実装**」 のコードは、以下を行います。

オブジェクトファクトリーをインスタンス化して要素を作成します。

WidgetOrderBillInfo オブジェクトをインスタンス化して、請求を保持します。

注文されたウィジェットの数を取得します。

注文に保存されている要素のローカル名を取得します。

要素が **woodWidget** 要素かどうかを確認します。

注文から適切なタイプのオブジェクトに要素の値を抽出します。

JAXBElement<T> オブジェクトを作成します。

bill オブジェクトの **widget** プロパティーを設定します。

bill オブジェクトの **amountDue** プロパティーを設定します。

第38章 タイプの生成方法のカスタマイズ

概要

デフォルトの JAXB マッピングは、XML スキーマを使用して Java アプリケーションのオブジェクトを定義するときに発生するほとんどのケースに対応しています。デフォルトのマッピングが不十分な場合、JAXB は広範なカスタマイズメカニズムを提供します。

38.1. タイプマッピングのカスタマイズの基本

概要

JAXB 仕様は、Java タイプを XML スキーマ構造にマップする方法をカスタマイズするいくつかの XML 要素を定義しています。これらの要素は、XML スキーマ構造とインラインで指定できます。XML スキーマ定義を変更できない、または変更したくない場合は、外部バインディングドキュメントでカスタマイズを指定できます。

Namespace

JAXB データバインディングのカスタマイズに使用される要素は、namespace **http://java.sun.com/xml/ns/jaxb** で定義されます。例38.1「JAXB カスタマイズ名前空間」 に示すような名前空間宣言を追加する必要があります。これは、JAXB のカスタマイズを定義するすべての XML ドキュメントのルート要素に追加されます。

例38.1 JAXB カスタマイズ名前空間

```
xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
```

バージョン宣言

JAXB カスタマイズを使用する場合は、使用されている JAXB バージョンを指定する必要があります。これには、外部バインディング宣言のルート要素に **jaxb:version** 属性を追加します。インラインのカスタマイズを使用している場合、カスタマイズを含む **schema** 要素 に **jaxb:version** 属性を含める必要があります。属性の値は常に **2.0** です。

例38.2「JAXB カスタマイズバージョンの指定」 に、**schema** 要素で使用される **jaxb:version** 属性の例を示します。

例38.2 JAXB カスタマイズバージョンの指定

```
< schema ...
      jaxb:version="2.0">
```

インラインカスタマイズの使用

コードジェネレーターが XML スキーマコンストラクトを Java コンストラクトにマップする方法をカスタマイズする最も直接的な方法は、カスタマイズ要素を XML スキーマ定義に直接追加することです。JAXB カスタマイズ要素は、変更される XML スキーマ構造の **xsd:appinfo** 要素内に配置されます。

例38.3「カスタマイズされた XML スキーマ」 は、インライン JAXB カスタマイズを含むスキーマの例を示しています。

例38.3 カスタマイズされた XML スキーマ

```
<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>
```

外部バインディング宣言の使用

タイプを定義する XML スキーマドキュメントに変更を加えることができない、または変更したくない場合は、外部バインディング宣言を使用してカスタマイズを指定できます。外部バインディング宣言は、多数のネストされた **jaxb:bindings** 要素で設定されます。例38.4「JAXB 外部バインディング宣言構文」 は、外部バインディング宣言の構文を示しています。

例38.4 JAXB 外部バインディング宣言構文

```
<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="schemaUri" | wsdlLocation="wsdlUri">
    <jaxb:bindings node="nodeXPath">
      binding declaration
    </jaxb:bindings>
    ...
  </jaxb:bindings>
<jaxb:bindings>
```

schemaLocation 属性と **wsdlLocation** 属性は、変更が適用されるスキーマドキュメントを特定するために使用されます。スキーマドキュメントからコードを生成する場合は、**schemaLocation** 属性を使用します。WSDL ドキュメントからコードを生成する場合は、**wsdlLocation** 属性を使用します。

node 属性は、修正する特定の XML スキーマ構造を特定するために使用されます。これは、XML スキーマ要素に解決される XPath ステートメントです。

例38.5「XML スキーマファイル」 に示されているスキーマドキュメント **widgetSchema.xsd** の場合、例38.6「外部バインディング宣言」 に記載の外部バインディング宣言により、複合型 **size** の生成が変更されます。

例38.5 XML スキーマファイル

```
<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 外部バインディング宣言

```
<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>
```

コードジェネレーターに対して外部バインディング宣言を使用するように指示するには、以下のように **wsdl2java** ツールの **-b binding-file** オプションを使用します。

```
wsdl2java -b widgetBinding.xml widget.wsdl
```

38.2. XML スキーマプリミティブの JAVA クラスの指定

概要

デフォルトでは、XML スキーマ型は Java プリミティブ型にマップされます。これは XML スキーマと Java の間の最も論理的なマッピングですが、アプリケーション開発者の要件を常に満たすとは限りません。XML スキーマプリミティブ型を追加情報を保持できる Java クラスにマップしたり、XML プリミティブ型を単純な型置換を可能にするクラスにマップしたりすることができます。

JAXB **javaType** カスタマイズ要素を使用すると、XML スキーマプリミティブ型と Java プリミティブ型間のマッピングをカスタマイズできます。これを使用して、グローバルレベルと個々のインスタンスレベルの両方でマッピングをカスタマイズできます。**javaType** 要素は、単純型定義の一部として、または複合型定義の一部として使用できます。

javaType カスタマイズ要素を使用する場合は、プリミティブ型の XML 表現とターゲット Java クラス間の変換方法を指定する必要があります。一部のマッピングには、デフォルトの変換方法があります。デフォルトのマッピングがない場合、Apache CXF は必要なメソッドの開発を容易にする JAXB メソッドを提供します。

構文

javaType カスタマイズ要素は、表38.1「XML スキーマタイプの Java クラスの生成をカスタマイズするための属性」 に記載されている 4 つの属性を取ります。

表38.1 XML スキーマタイプの Java クラスの生成をカスタマイズするための属性

属性	Required	説明
name	はい	XML スキーマプリミティブ型がマップされる Java クラスの名前を指定します。有効な Java クラス名または Java プリミティブ型の名前のいずれかである必要があります。このクラスが存在し、アプリケーションからアクセスできることを確認する必要があります。コードジェネレータはこのクラスをチェックしません。
xmlType	いいえ	カスタマイズされる XML スキーマプリミティブ型を指定します。この属性は、**javaType** 要素が **globalBindings** 要素の子として使用される場合にのみ使用されます。
parseMethod	いいえ	データの文字列ベースの XML 表現を Java クラスのインスタンスに解析するためのメソッドを指定します。詳細は、「コンバーターの指定」 を参照してください。
printMethod	いいえ	Java オブジェクトをデータの文字列ベースの XML 表現に変換するためのメソッドを指定します。詳細は、「コンバーターの指定」を参照してください。

javaType カスタマイズ要素は、3 つの方法で使用できます。

- XML スキーマプリミティブ型のインスタンスをすべて変更する場合: **globalBindings** カスタマイズ要素の子として使用されている場合、**javaType** 要素はスキーマドキュメントの XML スキーマ型のインスタンスをすべて変更します。この方法で使用する場合、変更中の XML スキーマプリミティブ型を識別する **xmlType** 属性の値を指定する必要があります。
  例38.7「グローバルプリミティブ型のカスタマイズ」に、スキーマの **xsd:short** の全インスタンスに **java.lang.Integer** を使用するようコードジェネレーターに指示する、インラインのグローバルカスタマイズを示します。

  例38.7 グローバルプリミティブ型のカスタマイズ

```
<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>
```

- 単純型定義を変更する場合: 名前付き単純型定義に適用される場合、**javaType** 要素は XML 単純型のすべてのインスタンスに生成されたクラスを変更します。**javaType** 要素を使用して単純型定義を変更する場合は、**xmlType** 属性を使用しないでください。
例38.8「シンプルタイプをカスタマイズするためのバインディングファイル」 に、**zipCode** という名前の単純型の生成を変更する外部バインディングファイルを示します。

例38.8 シンプルタイプをカスタマイズするためのバインディングファイル

```
<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>
```

- 複合型定義の要素または属性を変更する場合: **javaType** は、JAXB プロパティーのカスタマイズの一部として追加することで、複合型定義の個々の部分に適用できます。**javaType** 要素は、プロパティーの **baseType** 要素に子として配置されます。**javaType** 要素を使用して複合型定義の特定の部分を変更する場合、**xmlType** 属性を使用しないでください。
例38.9「複合型の要素をカスタマイズするためのバインディングファイル」 は、複合型の要素を変更するバインディングファイルを示しています。

例38.9 複合型の要素をカスタマイズするためのバインディングファイル

```
<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>
```

baseType 要素の使用に関する詳細は、「要素または属性の基本タイプの指定」 を参照してください。

コンバーターの指定

Apache CXF は、XML スキーマのプリミティブ型をランダムな Java クラスに変換できません。**javaType** 要素を使用して XML スキーマプリミティブ型のマッピングをカスタマイズする場合、コードジェネレーターは、カスタマイズされた XML スキーマプリミティブ型をマーシャリングおよびアンマーシャリングするために使用されるアダプタークラスを作成します。サンプルのアダプタークラスを 例38.10「JAXB アダプタークラス」 に示します。

例38.10 JAXB アダプタークラス

```
public class Adapter1 extends XmlAdapter<String, javaType>
{
  public javaType unmarshal(String value)
  {
    return(parseMethod(value));
  }

  public String marshal(javaType value)
  {
    return(printMethod(value));
  }
}
```

parseMethod および printMethod は、対応する **parseMethod** 属性および **printMethod** 属性の値に置き換えてください。値は、有効な Java メソッドを識別する必要があります。メソッドの名前は、次の 2 つの方法のいずれかで指定できます。

- **packagename.ClassName.methodName** の形式の完全修飾 Java メソッド名。
- **methodName** の形式の単純なメソッド名
  簡単なメソッド名のみを指定すると、コードジェネレーターは、**javaType** 要素の **name** 属性で指定されたクラスにメソッドが存在すると想定します。

重要

コードジェネレーターは、解析メソッドまたは印刷メソッドを生成 **しません**。あなたはそれらを供給する責任があります。解析メソッドと印刷メソッドの開発については、「コンバーターの実装」 を参照してください。

parseMethod 属性の値が指定されていない場合、コードジェネレーターは、**name** 属性で指定された Java クラスに最初のパラメーターが Java **String** オブジェクトのコンストラクターがあると想定します。生成されたアダプターの **unmarshal()** メソッドは、想定されたコンストラクターを使用して Java オブジェクトに XML データを設定します。

printMethod 属性の値が指定されていない場合、コードジェネレーターは、**name** 属性で指定された Java クラスに **toString()** メソッドがあると想定します。生成されたアダプターの **marshal()** メソッドは、想定された **toString()** メソッドを使用して Java オブジェクトを XML データに変換します。

javaType 要素の **name** 属性が Java プリミティブ型または Java プリミティブのラッパー型のいずれかを指定する場合、コードジェネレーターはデフォルトのコンバーターを使用します。デフォルトのコンバーターの詳細については、「デフォルトのプリミティブ型コンバーター」 を参照してください。

生成されるもの

「コンバーターの指定」 に記載のとおり、**javaType** カスタマイズ要素を使用すると、XML スキーマプリミティブ型の各カスタマイズに対してアダプタークラスの生成が 1 つトリガーされます。このアダプターの名前は、**AdapterN** のパターンを使用して順番に付けられます。2 つのプリミティブタイプのカスタマイズを指定する場合、コードジェネレーターは **Adapter1** と **Adapter2** の 2 つのアダプタークラスを作成します。

XML スキーマ構造用に生成されたコードは、影響を受ける XML スキーマ構造がグローバルに定義された要素であるか、複合型の一部として定義されているかによって異なります。

XML スキーマ構造がグローバルに定義された要素である場合、その型に対して生成されたオブジェクトファクトリーメソッドは、デフォルトのメソッドから次のように変更されます。

- メソッドには **@XmlJavaTypeAdapter** アノテーションが付けられます。
  アノテーションは、この要素のインスタンスを処理するときに使用するアダプタークラスをランタイムに指示します。アダプタークラスはクラスオブジェクトとして指定されます。

- デフォルトの型は、**javaType** 要素の **name** 属性で指定されたクラスに置き換えられます。

例38.11「グローバル要素のカスタマイズされたオブジェクトファクトリーメソッド」 は、例38.7「グローバルプリミティブ型のカスタマイズ」 に示すカスタマイズの影響を受ける要素のオブジェクトファクトリーメソッドを示します。

例38.11 グローバル要素のカスタマイズされたオブジェクトファクトリーメソッド

```
@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);
}
```

XML スキーマ構造が複合型の一部として定義されている場合、生成された Java プロパティーは次のように変更されます。

- プロパティーには **@XmlJavaTypeAdapter** アノテーションが付けられます。
  アノテーションは、この要素のインスタンスを処理するときに使用するアダプタークラスをランタイムに指示します。アダプタークラスはクラスオブジェクトとして指定されます。

- プロパティーの **@XmlElement** には、**type** プロパティーが含まれています。
  type プロパティーの値は、生成されるオブジェクトのデフォルトベース型を表すクラスオブジェクトです。XML スキーマプリミティブ型の場合、クラスは **String** になります。

- プロパティーには **@XmlSchemaType** アノテーションが付けられます。
  アノテーションは、設定の XML スキーマプリミティブ型を識別します。

- デフォルトの型は、**javaType** 要素の **name** 属性で指定されたクラスに置き換えられます。

例38.12「カスタマイズされた複合タイプ」 は、例38.7「グローバルプリミティブ型のカスタマイズ」に示すカスタマイズの影響を受ける要素のオブジェクトファクトリーメソッドを示します。

例38.12 カスタマイズされた複合タイプ

```
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;
    }

}
```

コンバーターの実装

parseMethod 属性と **printMethod** 属性で指定されたメソッドを呼び出す必要があることを除き、Apache CXF ランタイムは、XML プリミティブ型と **javaType** 要素によって指定された Java クラス間の変換方法を認識しません。ランタイムが呼び出すメソッドの実装を提供するのはあなたの責任です。実装されたメソッドは、XML プリミティブ型の字句構造を処理できる必要があります。

データ変換メソッドの実装を簡素化するために、Apache CXF は **javax.xml.bind.DatatypeConverter** クラスを提供します。このクラスは、すべての XML スキーマプリミティブ型を解析および出力するためのメソッドを提供します。解析メソッドは XML データの文字列表現を取り、表34.1「XML スキーマのプリミティブ型から Java ネイティブ型へのマッピング」 で定義されたデフォルトタイプのインスタンスを返します。print メソッドはデフォルトタイプのインスタンスを取り、XML データの文字列表現を返します。

DatatypeConverter クラスの Java ドキュメントは、https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/DatatypeConverter.html にあります。

デフォルトのプリミティブ型コンバーター

Java プリミティブ型、または Java プリミティブ型のラッパークラスのいずれかを **javaType** 要素の **name** 属性で指定する場合には、**parseMethod** 属性または **printMethod** 属性の値を指定する必要はありません。値が指定されていない場合、Apache CXF ランタイムはデフォルトのコンバーターを置き換

えます。

デフォルトのデータコンバーターは JAXB **DatatypeConverter** クラスを使用して XML データを解析します。デフォルトのコンバーターは、変換を機能させるために必要な型キャストも提供します。

38.3. 単純型の JAVA クラスの生成

概要

デフォルトでは、名前付きの単純型は、列挙型でない限り、生成された型にはなりません。単純型を使用して定義された要素は、Java プリミティブ型のプロパティーにマップされます。

型置換を使用する場合など、Java クラスに単純型を生成する必要がある場合があります。

コードジェネレーターに対し、グローバルに定義したすべての単純型のクラスを生成するように指示するには、**globalBindings** カスタマイズ要素の **mapSimpleTypeDef** を **true** に設定します。

カスタマイズの追加

名前付き単純型の Java クラスを作成するようにコードジェネレーターに指示するには、**globalBinding** 要素の **mapSimpleTypeDef** 属性を追加し、その値を **true** に設定します。

例38.13「SimpleTypes の Java クラスの生成を強制するインラインカスタマイズ」 は、コードジェネレーターに名前付きの単純型の Java クラスを生成させるインラインカスタマイズを示しています。

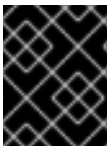
例38.13 SimpleTypes の Java クラスの生成を強制するインラインカスタマイズ

```
<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「定数の生成を強制するためのファイルのバインディング」 は、単純型の生成をカスタマイズする外部バインディングファイルを示しています。

例38.14 定数の生成を強制するためのファイルのバインディング

```
<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

このカスタマイズは、**グローバル** スコープで定義されている **名前付き** の単純型にのみ影響します。

生成されたクラス

単純型用に生成されるクラスには、**value** という 1 つのプロパティーがあります。**value** プロパティーは、「プリミティブ型」 のマッピングによって定義された Java 型です。生成されたクラスには **value** プロパティー getter と setter があります。

例38.16「シンプルタイプのカスタマイズされたマッピング」に、例38.15「カスタマイズされたマッピングのための単純なタイプ」 で定義される簡易型に生成される Java クラスを示します。

例38.15 カスタマイズされたマッピングのための単純なタイプ

```
<simpleType name="simpleton">
  <restriction base="xsd:string">
    <maxLength value="10"/>
  </restriction>
</simpleType>
```

例38.16 シンプルタイプのカスタマイズされたマッピング

```
@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. 列挙マッピングのカスタマイズ

概要

xsd:string 以外のスキーマタイプに基づく列挙型が必要な場合は、コードジェネレーターにマップするように指示する必要があります。生成される列挙定数の名前を制御することもできます。

カスタマイズは、1つ以上の **jaxb:typesafeEnum Member** 要素と共に **jaxb:typesafeEnum Class** 要素を使用して行われます。

コードジェネレーターのデフォルト設定では、列挙型のすべてのメンバーに対して有効な Java 識別子を作成できない場合もあります。**globalBindings** カスタマイズの属性を使用して、コードジェネレーターがこれを処理する方法をカスタマイズできます。

メンバー名カスタマイザー

列挙のメンバーを生成する際にコードジェネレーターが名前の競合に直面する場合や、列挙のメンバーに有効な Java 識別子を作成できない場合、デフォルトではコードジェネレーターが警告を生成し、列挙の Java **enum** 型を生成しません。

この動作は、**globalBinding** 要素の **typesafeEnumMemberName** 属性を追加することにより変更できます。**typesafeEnumMemberName** 属性の値については、表38.2「列挙型メンバー名の生成をカスタマイズするための値」 に記載されています。

表38.2 列挙型メンバー名の生成をカスタマイズするための値

値	説明
skipGeneration(デフォルト)	Java **enum** 型が生成されず、警告を生成することを指定します。
generateName	パターン **VALUE_N** に従ってメンバー名が生成されることを指定します。**N** は1から始まり、列挙のメンバーごとに増分されます。
generateError	列挙を Java **enum** 型にマッピングできない場合に、コードジェネレーターがエラーを生成することを指定します。

例38.17「タイプセーフなメンバー名を強制するためのカスタマイズ」 は、コードジェネレーターにタイプセーフなメンバー名を生成させるインラインカスタマイズを示しています。

例38.17 タイプセーフなメンバー名を強制するためのカスタマイズ

```
<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>
```

クラスカスタマイザ

jaxb:typesafeEnumClass 要素は、XML スキーマ列挙を Java **enum** 型にマッピングする必要があることを指定します。これには、表38.3「生成された列挙型クラスをカスタマイズするための属性」 で説明されている 2 つの属性があります。**jaxb:typesafeEnumClass** 要素がインラインで指定されている場合は、変更する単純型の **xsd:annotation** 要素内に配置する必要があります。

表38.3 生成された列挙型クラスをカスタマイズするための属性

属性	説明
name	生成される Java **enum** 型の名前を指定します。この値は有効な Java 識別子である必要があります。
map	列挙を Java **enum** 型へマッピングすべきかどうかを指定します。デフォルト値は **true** です。

メンバーカスタマイザー

jaxb:typesafeEnumMember 要素は、XML スキーマ **enumeration** ファセットと Java **enum** 型の定数の間のマッピングを指定します。カスタマイズする列挙では、**enumeration** ごとに 1 つの **jaxb:typesafeEnumMember** 要素を使用する必要があります。

インラインカスタマイズを使用する場合、この要素は次の 2 つの方法のいずれかで使用できます。

- 変更する **enumeration** ファセットの **xsd:annotation** 要素内に配置することができます。
- 列挙のカスタマイズに使用される **jaxb:typesafeEnumClass** 要素の子として、すべて配置することができます。

jaxb:typesafeEnumMember 要素には、必須の **name** 属性があります。**name** 属性は、生成される Java **enum** 型の定数の名前を指定します。その値は有効な Java 識別子でなければなりません。

jaxb:typesafeEnumMember 要素には、**value** 属性もあります。**value** は、**enumeration** ファセットを適切な **jaxb:typesafeEnumMember** 要素に関連付けるために使用されます。**value** 属性の値は、**enumeration** ファセットの **value** 属性の値のいずれかと一致する必要があります。この属性は、型の生成をカスタマイズするために外部バインディング仕様を使用する場合や、**jaxb:typesafeEnumMember** 要素を **jaxb:typesafeEnumClass** 要素の子としてグループ化する場合に必要です。

例

例38.18「列挙型のインラインカスタマイズ」 は、インラインカスタマイズを使用し、列挙型のメンバーを個別にカスタマイズした列挙型を示しています。

例38.18 列挙型のインラインカスタマイズ

```
<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「結合マッピングを使用した列挙型のインラインカスタマイズ」 は、インラインカスタマイズを使用し、メンバーのカスタマイズをクラスのカスタマイズに組み合わせた列挙型を示しています。

例38.19 結合マッピングを使用した列挙型のインラインカスタマイズ

```
<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「列挙をカスタマイズするためのバインディングファイル」 は、列挙型をカスタマイズする外部バインディングファイルを示しています。

例38.20 列挙をカスタマイズするためのバインディングファイル

```
<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. 固定値属性マッピングのカスタマイズ

概要

デフォルトでは、コードジェネレーターは、固定値を持つものとして定義された属性を通常のプロパティーにマップします。スキーマ検証を使用する場合、Apache CXF はスキーマ定義を適用できます (「スキーマ検証タイプの値」 を参照)。ただし、スキーマ検証を使用すると、メッセージの処理時間が長くなります。

固定値を持つ属性を Java にマップする別の方法は、それらを Java 定数にマップすることです。**globalBindings** カスタマイズ要素を使用して、固定値属性を Java 定数にマッピングするようコードジェネレーターに指示することができます。また、**property** 要素を使用して、固定値属性の Java 定数へのマッピングをよりローカルレベルでカスタマイズすることもできます。

グローバルカスタマイズ

この動作は、**globalBinding** 要素の **fixedAttributeAsConstantProperty** 属性を追加することにより変更できます。この属性を **true** に設定すると、**fixed** 属性を使用して定義された属性を Java 定数にマッピングするようにコードジェネレーターに指示します。

例38.21「定数の生成を強制するためのインラインカスタマイズ」 は、コードジェネレーターに固定値の属性の定数を生成させるインラインカスタマイズを示しています。

例38.21 定数の生成を強制するためのインラインカスタマイズ

```
<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「定数の生成を強制するためのファイルのバインディング」 は、固定属性の生成をカスタマイズする外部バインディングファイルを示しています。

例38.22 定数の生成を強制するためのファイルのバインディング

```
<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>
```

ローカルマッピング

property 要素の **fixedAttributeAsConstantProperty** 属性を使用して、属性ごとに属性のマッピングをカスタマイズできます。この属性を **true** に設定すると、**fixed** 属性を使用して定義された属性を Java 定数にマッピングするようにコードジェネレーターに指示します。

例38.23「定数の生成を強制するためのインラインカスタマイズ」 は、コードジェネレーターが固定値を持つ単一の属性の定数を生成するように強制するインラインカスタマイズを示しています。

例38.23 定数の生成を強制するためのインラインカスタマイズ

```
<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「定数の生成を強制するためのファイルのバインディング」 は、固定属性の生成をカスタマイズする外部バインディングファイルを示しています。

例38.24 定数の生成を強制するためのファイルのバインディング

```
<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 マッピング

デフォルトのマッピングでは、すべての属性がゲッターメソッドとセッターメソッドを使用して標準の Java プロパティーにマッピングされます。このカスタマイズが **fixed** 属性を使用して定義された属性に適用されると、例38.25「固定値属性の Java 定数へのマッピング」 に示されているように、属性は Java 定数にマッピングされます。

例38.25 固定値属性の Java 定数へのマッピング

```
@XmlAttribute
public final static type NAME = value;
```

タイプ は、 「プリミティブ型」 で説明されているマッピングを使用して、属性の基本タイプを Java タイプにマッピングすることによって決定されます。

NAME は、 **attribute** 要素の **name** 属性の値をすべて大文字に変換することで決定されます。

値 は **attribute** 要素の **fixed** 属性の値によって決定されます。

たとえば、例38.23「定数の生成を強制するためのインラインカスタマイズ」 で定義された属性 例38.26「Java 定数にマップされた固定値属性」 に示すようにマップされます。

例38.26 Java 定数にマップされた固定値属性

```
@XmlRootElement(name = "widgetAttr")
```

```
public class WidgetAttr {

   ...

  @XmlAttribute
  public final static int FIXER = 7;

  ...

}
```

38.6. 要素または属性の基本タイプの指定

概要

場合によっては、要素用に、または XML スキーマ複合型の一部として定義された属性用に生成されたオブジェクトのクラスをカスタマイズする必要があります。たとえば、より一般化されたクラスのオブジェクトを使用して、単純な型置換を可能にすることができます。

これを行う 1 つの方法は、JAXB 基本タイプのカスタマイズを使用することです。これにより、開発者は、ケースバイケースで、要素または属性を表すために生成されたオブジェクトのクラスを指定できます。基本型のカスタマイズにより、XML スキーマ構造と生成された Java オブジェクト間の代替マッピングを指定できます。この代替マッピングは、デフォルトの基本クラスの単純な特殊化または一般化にすることができます。XML スキーマプリミティブ型を Java クラスにマッピングすることもできます。

カスタマイズの使用法

JAXB ベース型のプロパティーを XML スキーマ構造に適用するには、JAXB **baseType** カスタマイズ要素を使用します。**baseType** カスタマイズ要素は JAXB **property** 要素の子であるため、適切にネスティングする必要があります。

XML スキーマ構造の Java オブジェクトへのマッピングをどのようにカスタマイズするかに応じて、**baseType** カスタマイズ要素の **name** 属性または **javaType** 子要素のいずれかを追加します。**name** 属性は、生成されたオブジェクトのデフォルトクラスを同じクラス階層内の別のクラスにマッピングするために使用されます。**javaType** 要素は、XML スキーマプリミティブ型を Java クラスにマッピングする場合に使用されます。

重要

同じ **baseType** カスタマイズ要素で **name** 属性と **javaType** 子要素の両方を使用することはできません。

デフォルトのマッピングを特殊化または一般化する

baseType カスタマイズ要素の **name** 属性は、生成されたオブジェクトのクラスを、同じ Java クラス階層内のクラスに再定義するために使用されます。この属性は、XML スキーマ構造がマップされる Java クラスの完全修飾名を指定します。指定する Java クラスは、コードジェネレーターが XML スキーマ構造用に通常生成する Java クラスのスーパークラスまたはサブクラスのいずれかである ***必要*** があります。Java プリミティブ型にマップする XML スキーマプリミティブ型の場合、カスタマイズの目的で、ラッパークラスがデフォルトの基本クラスとして使用されます。

たとえば、**xsd:int** として定義された要素は、**java.lang.Integer** をデフォルトのベースクラスとして使用します。**name** 属性の値には、**Number** または **Object** など **Integer** のスーパークラスを指定できます。

単純型置換の場合、最も一般的なカスタマイズは、プリミティブ型を **Object** オブジェクトにマッピングすることです。

例38.27「基本タイプのインラインカスタマイズ」に、複合型の 1 つの要素を Java **Object** オブジェクトにマッピングするインラインカスタマイズを示します。

例38.27 基本タイプのインラインカスタマイズ

```
<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「基本タイプをカスタマイズするための外部バインディングファイル」は、例38.27「基本タイプのインラインカスタマイズ」に示されているカスタマイズ用の外部バインディングファイルを示しています。

例38.28 基本タイプをカスタマイズするための外部バインディングファイル

```
<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>
```

結果の Java オブジェクトの **@XmlElement** アノテーションには **type** プロパティーが含まれます。**type** プロパティーの値は、生成されるオブジェクトのデフォルトベース型を表すクラスオブジェクトです。XML スキーマプリミティブ型の場合、クラスは対応する Java プリミティブ型のラッパークラスです。

例38.29「基本クラスが変更された Java クラス」は、例38.28「基本タイプをカスタマイズするための外部バインディングファイル」のスキーマ定義に基づいて生成されたクラスを示しています。

例38.29 基本クラスが変更された Java クラス

```
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;
    }

}
```

javaType での使用

javaType 要素は、XML スキーマのプリミティブ型を使用して定義された要素と属性をどのように Java オブジェクトにマッピングするかをカスタマイズするのに使用できます。**javaType** 要素を使用すると、単に **baseType** 要素の **name** 属性を使用するよりも柔軟性が高くなります。**javaType** 要素を使用すると、プリミティブ型をオブジェクトの任意のクラスにマッピングすることができます。

javaType 要素の使用の詳細な説明は、「XML スキーマプリミティブの Java クラスの指定」 を参照してください。

第39章 JAXBCONTEXT オブジェクトの使用

概要

JAXBContext オブジェクトにより、Apache CXF のランタイムが XML 要素と Java オブジェクト間でデータを変換できます。未加工の XML メッセージを扱うコンシューマーを実装する場合、アプリケーション開発者はメッセージハンドラーで JAXB オブジェクトを使用する **JAXBContext** オブジェクトをインスタンス化する必要があります。

概要

JAXBContext オブジェクトは、ランタイムで使用される低レベルオブジェクトです。これにより、ランタイムは XML 要素とそれに対応する Java 表現の間で変換できます。通常、アプリケーション開発者は **JAXBContext** オブジェクトを操作する必要はありません。XML データのマーシャリングとアンマーシャリングは、通常、JAX-WS アプリケーションのトランスポート層とバインディング層によって処理されます。

ただし、アプリケーションが XML メッセージのコンテンツを直接操作する必要がある場合があります。これらの 2 つの例では:

- 「コンシューマーでの XML の使用」
- *43章ハンドラーの作成*

利用可能な 2 つの **JAXBContext.newInstance()** メソッドのいずれかを使用して **JAXBContext** オブジェクトをインスタンス化する必要があります。

ベストプラクティス

JAXBContext オブジェクトのインスタンス化には、大量のリソースが必要です。アプリケーションが作成するインスタンスをできるだけ少なくすることを推奨します。これを実行する 1 つの方法は、アプリケーションが使用するすべての JAXB オブジェクトを管理できる単一の **JAXBContext** オブジェクトを作成し、アプリケーションの可能な限り多数の部分で共有することです。

JAXBContext オブジェクトはスレッドセーフです。

オブジェクトファクトリーを使用した JAXBCONTEXT オブジェクトの取得

JAXBContext クラスは、例39.1「クラスを使用した JAXB コンテキストの取得」に示される、JAXB オブジェクトを実装するクラスのリストを取る **newInstance()** メソッドを提供します。

> **例39.1 クラスを使用した JAXB コンテキストの取得**
>
> static**JAXBContextnewInstanceClass...classesToBeBound**JAXBException

返される **JAXBObject** オブジェクトは、メソッドに渡されるクラスによって実装される JAXB オブジェクトのデータをマーシャリングおよびアンマーシャリングできます。また、メソッドに渡されたクラスのいずれかから静的に参照されているクラスを操作することもできます。

アプリケーションによって使用されるすべての JAXB クラスの名前を **newInstance()** メソッドに渡すことはできますが、効率的ではありません。同じ目標を達成するためのより効率的な方法は、アプリケー

ション用に生成されたオブジェクトファクトリーまたはオブジェクトファクトリーを渡すことです。結果の **JAXBContext** オブジェクトは、指定されたオブジェクトファクトリーがインスタンス化できる JAXB クラスを管理できます。

パッケージ名を使用して JAXBCONTEXT オブジェクトを取得する

JAXBContext クラスは、例39.2「クラスを使用した JAXB コンテキストの取得」に示される、パッケージ名のコロン (**:**) 区切りリストを取る **newInstance()** メソッドを提供します。指定されたパッケージには、XML スキーマから派生した JAXB オブジェクトが含まれている必要があります。

> **例39.2 クラスを使用した JAXB コンテキストの取得**
>
> static**JAXBContextnewInstanceStringcontextPath**JAXBException

返される **JAXBContext** オブジェクトは、指定したパッケージのクラスによって実装されるすべての JAXB オブジェクトのデータをマーシャリングおよびアンマーシャリングできます。

第40章 非同期アプリケーションの開発

概要

JAX-WS は、サービスに非同期でアクセスするための簡単なメカニズムを提供します。SEI は、サービスに非同期でアクセスするために使用できる追加のメソッドを指定できます。Apache CXF コードジェネレーターは、追加のメソッドを生成します。ビジネスロジックを追加するだけです。

40.1. 非同期呼び出しのタイプ

通常の同期呼び出しモードに加えて、Apache CXF は 2 つの形式の非同期呼び出しをサポートします。

- ポーリングアプローチ: ポーリングアプローチを使用してリモート操作を呼び出すには、出力パラメーターを持たないが **javax.xml.ws.Response** オブジェクトを返すメソッドを呼び出します。**Response** オブジェクト (javax.util.concurrency.Future インターフェイスから継承) をポーリングして、応答メッセージが到達したかどうかを確認できます。

- コールバックアプローチ: コールバックアプローチを使用してリモート操作を呼び出すには、コールバックオブジェクト (**javax.xml.ws.AsyncHandler** 型) をパラメーターの 1 つとして参照するメソッドを呼び出します。応答メッセージがクライアントに到達すると、ランタイムが **AsyncHandler** オブジェクト上でコールバックし、応答メッセージの内容を提供します。

40.2. 非同期の例の WSDL

例40.1「非同期の例の WSDL コントラクト」 は、非同期の例で使用される WSDL コントラクトを示しています。コントラクトは、単一の操作 greetMeSometime を含む単一のインターフェイス GreeterAsync を定義します。

例40.1 非同期の例の WSDL コントラクト

```
<?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. スタブコードの生成

概要

非同期スタイルの呼び出しには、SEI で定義された専用の非同期メソッド用の追加のスタブコードが必要です。この特別なスタブコードは、デフォルトでは生成されません。非同期機能をオンにして必要なスタブコードを生成するには、WSDL 2.0 仕様のマッピングカスタマイズ機能を使用する必要があります。

カスタマイズにより、Maven コード生成プラグインがスタブコードを生成する方法を変更できます。特に、WSDL から Java へのマッピングを変更したり、特定の機能をオンにしたりすることができます。ここでは、カスタマイズを使用して非同期呼び出し機能をオンにします。カスタマイズは、**jaxws:bindings** タグを使用して定義するバインディング宣言を使用して指定されます (ここでは、**jaxws** 接頭辞は **http://java.sun.com/xml/ns/jaxws** namespace に関連付けられます)。バインディング宣言を指定する方法は 2 つあります。

外部バインディング宣言

外部バインディング宣言を使用する場合、**jaxws:bindings** 要素は WSDL コントラクトとは別のファイルで定義されます。スタブコードを生成するときに、コードジェネレーターへのバインディング宣言ファイルの場所を指定します。

埋め込みバインディング宣言

組み込みバインディング宣言を使用する場合は、**jaxws:bindings** 要素を直接 WSDL コントラクトに組み込み、それを WSDL 拡張として扱います。この場合、**jaxws:bindings** の設定は直接の親要素にのみ適用されます。

外部バインディング宣言の使用

非同期呼び出しをオンにするバインディング宣言ファイルのテンプレートを 例40.2「非同期バインディング宣言のテンプレート」 に示します。

例40.2 非同期バインディング宣言のテンプレート

```
<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>
```

ここで、**AffectedWSDL** は、このバインディング宣言の影響を受ける WSDL コントラクトの URL を指定します。**AffectedNode** は、WSDL コントラクトのどのノードがこのバインディング宣言の影響を受けるかを指定する XPath 値です。WSDL コントラクト全体が影響を受けるようにするには、**AffectedNode** を **wsdl:definitions** に設定できます。非同期呼び出し機能を有効にするには、**jaxws:enableAsyncMapping** 要素を **true** に設定します。

たとえば、GreeterAsync インターフェイスに対してのみ非同期メソッドを生成する場合は、前のバインディング宣言で <bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']"> を指定できます。

バインディング宣言がファイル **async_binding.xml** に保存されている場合、例40.3「コンシューマーコード生成」に記載されているように POM を設定します。

例40.3 コンシューマーコード生成

```
<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 オプションは、コードジェネレーターに対し、外部バインディングファイルの場所を指定します。

コードジェネレーターの詳細は、「cxf-codegen-plugin」 を参照してください。

埋め込みバインディング宣言の使用

また、**jaxws:bindings** 要素とその関連の **jaxws:enableAsynchMapping** 子要素を直接 WSDL に配置して、バインディングのカスタマイズをサービスを定義する WSDL ドキュメントに直接組み込むこともできます。また、**jaxws** 接頭辞の namespace 宣言を追加する必要もあります。

例40.4「非同期マッピングのための組み込みバインディング宣言を備えた WSDL」 は、操作の非同期マッピングをアクティブ化するバインディング宣言が埋め込まれた WSDL ファイルを示しています。

例40.4 非同期マッピングのための組み込みバインディング宣言を備えた 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>
```

バインディング宣言を WSDL ドキュメントに埋め込む場合、宣言を配置する場所を変更することで、宣言の影響を受けるスコープを制御できます。宣言が **wsdl:definitions** 要素の子として配置されると、コードジェネレーターは、WSDL ドキュメントに定義されたすべての操作に非同期メソッドを作成しま

す。**wsdl:portType** 要素の子として配置されると、コードジェネレーターは、インターフェイスに定義されたすべての操作に非同期メソッドを作成します。**wsdl:operation** 要素の子として配置されると、コードジェネレーターは、その操作にのみ非同期メソッドを作成します。

埋め込み宣言を使用する場合、コードジェネレーターに特別なオプションを渡す必要はありません。コードジェネレーターはそれらを認識し、それに応じて動作します。

生成されたインターフェイス

この方法でスタブコードを生成した後、GreeterAsync SEI (ファイル **GreeterAsync.java** 内) は 例 40.5「非同期呼び出しのメソッドを使用したサービスエンドポイントインターフェイス」に示すように定義されます。

例40.5 非同期呼び出しのメソッドを使用したサービスエンドポイントインターフェイス

```
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
    );
}
```

通常の同期メソッド **greetMeSometime()** のほかに、greetMeSometime 操作用に 2 つの非同期メソッドも生成されます。

- コールバックのアプローチ: public**Future<?>greetMeSomtimeAsyncjava.lang.StringrequestTypeAsyncHandler<GreetMeSomtimeResponse>asyncHandler**
- ポーリングのアプローチ: public**Response<GreetMeSomeTimeResponse>greetMeSometimeAsyncjava.lang.StringrequestType**

40.4. ポーリングアプローチを使用した非同期クライアントの実装

概要

ポーリングアプローチは、非同期アプリケーションを開発するための 2 つのアプローチのうちのより簡単なものです。クライアントが **OperationNameAsync()** という非同期メソッドを呼び出すと、応答を

ポーリングする **Response<T>** オブジェクトが返されます。クライアントが応答を待っている間に何をするかは、アプリケーションの要件によって異なります。ポーリングを処理するための 2 つの基本的なパターンがあります。

- ノンブロッキングポーリング: ノンブロッキング **Response<T>.isDone()** メソッドを呼び出すことで、結果が準備できているかどうかを定期的に確認します。結果の準備ができている場合、クライアントはそれを処理します。そうでない場合、クライアントは他のことを続けます。
- ブロッキングポーリング: **Response<T>.get()** をすぐに呼び出し、応答が到着するまでブロックします (任意でタイムアウトを指定します)。

非ブロッキングパターンを使用する

例40.6「非同期操作呼び出しのためのノンブロッキングポーリングアプローチ」 は、非ブロッキングポーリングを使用して、例40.1「非同期の例の WSDL コントラクト」 で定義されている greetMeSometime 操作で非同期呼び出しを行う方法を示しています。クライアントは非同期操作を呼び出し、結果が返されるかどうかを定期的にチェックします。

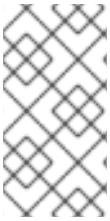
例40.6 非同期操作呼び出しのためのノンブロッキングポーリングアプローチ

```
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「非同期操作呼び出しのためのノンブロッキングポーリングアプローチ」 のコードは、以下を行います。

プロキシーで **greetMeSometimeAsync()** を呼び出す。

メソッド呼び出しは、**Response<GreetMeSometimeResponse>** オブジェクトを即座にクライアントに返します。Apache CXF ランタイムは、リモートエンドポイントからの応答受信および **Response<GreetMeSometimeResponse>** オブジェクト反映の詳細を処理します。



注記

ランタイムはリクエストをリモートエンドポイントの **greetMeSometime()** メソッドに送信し、呼び出しの非同期性の詳細を透過的に処理します。エンドポイント、したがってサービス実装は、クライアントが応答を待機する方法の詳細について心配することはありません。

返された **Response** オブジェクトの **isDone()** を確認して、応答が到達しているかどうかを確認します。

応答が到着していない場合、クライアントは再度チェックする前に作業を続行します。

応答が到達すると、クライアントは **get()** メソッドを使用して **Response** オブジェクトから取得します。

ブロッキングパターンの使用

ブロックポーリングパターンを使用する場合、**Response** オブジェクトの **isDone()** は呼び出されません。代わりに、リモート操作の呼び出し直後に **Response** オブジェクトの **get()** メソッドが呼び出されます。**get()** は、レスポンスが利用可能になるまでブロックします。

タイムアウトの制限を **get()** メソッドに渡すこともできます。

例40.7「非同期操作呼び出しのポーリングアプローチのブロック」 は、ブロッキングポーリングを使用するクライアントを示しています。

例40.7 非同期操作呼び出しのポーリングアプローチのブロック

```
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. コールバックアプローチを使用した非同期クライアントの実装

概要

非同期操作の呼び出しを行う別のアプローチは、コールバッククラスを実装することです。次に、コールバックオブジェクトをパラメーターとして受け取る非同期リモートメソッドを呼び出します。ランタイムは、コールバックオブジェクトへの応答を返します。

コールバックを使用するアプリケーションを実装するには、次のようにします。

1. AsyncHandler インターフェイスを実装するコールバッククラスを 作成 します。

注記

コールバックオブジェクトは、アプリケーションに必要な任意の量の応答処理を実行できます。

2. コールバックオブジェクトをパラメーターとして取り **Future<?>** オブジェクトを返す **operationNameAsync()** を使用して、リモート呼び出しを行います。

3. クライアントが応答データにアクセスする必要がある場合は、返された **Future<?>** オブジェクトの **isDone()** メソッドをポーリングして、リモートエンドポイントが応答を送信しているかどうかを確認できます。
コールバックオブジェクトがすべての応答処理を実行する場合、応答が到着したかどうかを確認する必要はありません。

コールバックの実装

コールバッククラスは、javax.xml.ws.AsyncHandler インターフェイスを実装する必要があります。インターフェイスは単一のメソッド **handleResponseResponse<T>res** を定義します。Apache CXF ランタイムは **handleResponse()** メソッドを呼び出して、レスポンスが到達していることをクライアントに通知します。例40.8「javax.xml.ws.AsyncHandler インターフェイス」 は、実装する必要のある AsyncHandler インターフェイスの概要を示しています。

例40.8 javax.xml.ws.AsyncHandler インターフェイス

```
public interface javax.xml.ws.AsyncHandler
{
  void handleResponse(Response<T> res)
}
```

例40.9「コールバックの実装クラス」 は、例40.1「非同期の例の WSDL コントラクト」 で定義された greetMeSometime 操作のコールバッククラスを示しています。

例40.9 コールバックの実装クラス

```
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「コールバックの実装クラス」 に示すコールバックの実装は次のことを行います。

リモートエンドポイントから返される応答を保持するメンバー変数 **response** を定義する。

handleResponse() を実装する。

この実装は、応答を抽出してそれをメンバー変数 **reply** に割り当てるだけです。

getResponse() という追加メソッドを実装する。

このメソッドは、**reply** からデータを抽出して返す便利な方法です。

コンシューマーの実装

例40.10「非同期操作コールのコールバックアプローチ」 は、コールバックアプローチを使用して、例40.1「非同期の例の WSDL コントラクト」 で定義された GreetMeSometime 操作への非同期呼び出しを行うクライアントを示しています。

例40.10 非同期操作コールのコールバックアプローチ

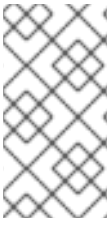
```
package demo.hw.client;

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「非同期操作コールのコールバックアプローチ」 のコードは、以下を行います。

コールバックオブジェクトをインスタンス化します。

プロキシーのコールバックオブジェクトを取得する **greetMeSometimeAsync()** を呼び出す。

メソッド呼び出しは、**Future<?>** オブジェクトを即座にクライアントに返します。Apache CXF ランタイムは、リモートエンドポイントからの応答受信、コールバックオブジェクトの **handleResponse()** メソッド呼び出し、および **Response<GreetMeSometimeResponse>** オブジェクト反映の詳細を処理します。



注記

ランタイムはリクエストをリモートエンドポイントの **greetMeSometime()** メソッドに送信し、呼び出しの非同期性の詳細をリモートエンドポイントが認識することなく処理します。エンドポイント、したがってサービス実装は、クライアントが応答を待機する方法の詳細について心配する必要はありません。

返された **Future<?>** オブジェクトの **isDone()** メソッドを使用して、応答がリモートエンドポイントから到達しているかどうかを確認する。

コールバックオブジェクトの **getResponse()** メソッドを呼び出し、応答データを取得する。

40.6. リモートサービスから返された例外をキャッチする

概要

非同期リクエストを行うコンシューマーは、同期リクエストを行うときに返されるのと同じ例外を受け取りません。非同期でコンシューマーに返される例外はすべて、ExecutionException 例外にラップされます。サービスによって出力される実際の例外は、ExecutionException 例外の **cause** フィールドに保存されます。

例外をキャッチする

リモートサービスによって生成された例外は、コンシューマーのビジネスロジックに応答を渡すメソッドによってローカルに出力されます。コンシューマーが同期要求を行うと、リモート呼び出しを行うメソッドが例外を出力します。コンシューマーが非同期リクエストを行う場合、Response<T> オブジェクトの **get()** メソッドが例外を出力します。コンシューマーは、応答メッセージを取得しようとするまで、要求の処理中にエラーが発生したことを検出しません。

JAX-WS フレームワークによって生成されるメソッドとは異なり、Response<T> オブジェクトの **get()** メソッドは、ユーザーがモデル化した例外や汎用 JAX-WS 例外を出力しません。代わりに、java.util.concurrent.ExecutionException 例外を出力します。

例外の詳細を取得する

フレームワークは、リモートサービスから返された例外を ExecutionException 例外の **cause** フィールドに保存します。リモート例外の詳細は、**cause** フィールドの値を取得し、保存した例外を調べて抽出されます。保存される例外は、任意のユーザー定義の例外または一般的な JAX-WS 例外の 1 つです。

例

例40.11「ポーリングアプローチを使用して例外をキャッチする」 は、ポーリングアプローチを使用して例外をキャッチする例を示しています。

例40.11 ポーリングアプローチを使用して例外をキャッチする

```
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「ポーリングアプローチを使用して例外をキャッチする」 のコードは、以下を行います。

Response<T> オブジェクトの **get()** メソッドへの呼び出しを try/catch ブロックにラップする。

ExecutionException 例外をキャッチします。

例外から **cause** フィールドを抽出する。

コンシューマーがコールバックアプローチを使用している場合、例外をキャッチするために使用されるコードは、サービスの応答が抽出されるコールバックオブジェクトに配置されます。

第41章 生の XML メッセージの使用

概要

高レベルの JAX-WSAPI は、データを JAXB オブジェクトにマーシャリングすることにより、開発者がネイティブ XML メッセージを使用するのを防ぎます。ただし、ネットワーク上を通過する生の XML メッセージデータに直接アクセスする方がよい場合があります。JAX-WS API は、生の XML へのアクセスを提供する 2 つのインターフェイスを提供します。Dispatch インターフェイスはクライアント側インターフェイスであり、Provider インターフェイスはサーバー側インターフェイスです。

41.1. コンシューマーでの XML の使用

概要

Dispatch インターフェイスは、生のメッセージを直接操作できる低レベルの JAX-WSAPI です。DOM オブジェクト、SOAP メッセージ、JAXB オブジェクトなど、さまざまな種類のメッセージまたはペイロードを受け入れて返します。これは低レベルの API であるため、Dispatch インターフェイスは、高レベルの JAX-WS API が実行するメッセージの準備を実行しません。Dispatch オブジェクトに渡すメッセージまたはペイロードが適切に構築されていること、および呼び出されているリモート操作に意味があることを確認する必要があります。

41.1.1. 使用モード

概要

ディスパッチオブジェクトには 2 つの **使用モード** があります。

- メッセージ モード
- メッセージペイロード モード (ペイロードモード)

Dispatch オブジェクトに指定する使用モードによって、ユーザーレベルのコードに渡される詳細の量が決まります。

メッセージモード

メッセージモード では、Dispatch オブジェクトは完全なメッセージで機能します。完全なメッセージには、バインディング固有のヘッダーとラッパーが含まれます。たとえば、SOAP メッセージを必要とするサービスと対話するコンシューマーは、Dispatch オブジェクトの **invoke()** メソッドに完全に指定された SOAP メッセージを提供する必要があります。**invoke()** メソッドも、完全に指定された SOAP メッセージを返します。コンシューマーコードは、SOAP メッセージのヘッダーと SOAP メッセージのエンベロープ情報を完成させて読み取る責任があります。

JAXB オブジェクトを操作する場合、メッセージモードは理想的ではありません。

Dispatch オブジェクトがメッセージモードを使用するように指定するには、Dispatch オブジェクトを作成するときに値 java.xml.ws.Service.Mode.MESSAGE を指定します。Dispatch オブジェクトの作成方法は 「ディスパッチオブジェクトの作成」 を参照してください。

ペイロードモード

メッセージペイロードモードとも呼ばれる **ペイロードモード** では、Dispatch オブジェクトはメッセージのペイロードのみを処理します。たとえば、ペイロードモードで動作する Dispatch オブジェクト

は、SOAP メッセージの本文でのみ機能します。バインディングレイヤーは、バインディングレベルのラッパーとヘッダーを処理します。**invoke()** メソッドから結果が返されると、バインディングレベルのラッパーとヘッダーはすでに取り除かれ、メッセージのボディーのみが残ります。

Apache CXF XML バインディングなど、特別なラッパーを使用しないバインディングを使用する場合は、ペイロードモードとメッセージモードで同じ結果が得られます。

Dispatch オブジェクトがペイロードモードを使用することを指定するには、Dispatch オブジェクトを作成するときに値 java.xml.ws.Service.Mode.PAYLOAD を指定します。Dispatch オブジェクトの作成方法は 「ディスパッチオブジェクトの作成」 を参照してください。

41.1.2. データ型

概要

Dispatch オブジェクトは低レベルのオブジェクトであるため、高レベルのコンシューマー API と同じ JAXB 生成タイプを使用するように最適化されていません。ディスパッチオブジェクトは、次のタイプのオブジェクトで機能します。

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource
- 「JAXB オブジェクトの使用」

ソースオブジェクトの使用

Dispatch オブジェクトは、javax.xml.transform.Source インターフェイスから派生したオブジェクトを受け入れて返します。ソースオブジェクトは、メッセージモードまたはペイロードモードのいずれかで、任意のバインディングによってサポートされます。

ソースオブジェクトは、XML ドキュメントを保持する低レベルのオブジェクトです。各 Source 実装は、保存された XML ドキュメントにアクセスし、その内容を操作するメソッドを提供します。次のオブジェクトは、ソースインターフェイスを実装します。

DOMSource

XML メッセージを Document Object Model (DOM) ツリーとして保持します。XML メッセージは、**getNode()** メソッドを使用してアクセスされる **Node** オブジェクトのセットとして保存されます。ノードは、**setNode()** メソッドを使用して更新または DOM ツリーに追加できます。

SAXSource

XML メッセージを Simple API for XML (SAX) オブジェクトとして保持します。SAX オブジェクトには、未加工のデータを保持する **InputSource** オブジェクトと、未加工のデータを解析する **XMLReader** オブジェクトが含まれます。

StreamSource

XML メッセージをデータストリームとして保持します。データストリームは、他のデータストリームと同じように操作できます。

汎用 Source オブジェクトを使用するように Dispatch オブジェクトを作成する場合、Apache CXF はメッセージを **SAXSource** オブジェクトとして返します。

この動作は、エンドポイントの **source-preferred-format** プロパティーを使用して変更できます。Apache CXF ランタイムの設定についての情報は、パートIV「Web サービスエンドポイントの設定」 を参照してください。

SOAPMessage オブジェクトの使用

以下の条件が満たされる場合、Dispatch オブジェクトは **javax.xml.soap.SOAPMessage** オブジェクトを使用できます。

- Dispatch オブジェクトは SOAP バインディングを使用しています
- Dispatch オブジェクトはメッセージモードを使用しています

SOAPMessage オブジェクトは SOAP メッセージを保持します。これらには、1 つの **SOAPPart** オブジェクトと 0 個以上の **AttachmentPart** オブジェクトが含まれています。**SOAPPart** オブジェクトには、SOAP メッセージの SOAP 固有部分が含まれます。これには、SOAP エンベロープ、SOAP ヘッダー、および SOAP メッセージボディーが含まれます。**AttachmentPart** オブジェクトには、アタッチメントとして渡されるバイナリーデータが含まれます。

DataSource オブジェクトの使用

ディスパッチオブジェクトは、次の条件が当てはまる場合に、javax.activation.DataSource インターフェイスを実装するオブジェクトを使用できます。

- Dispatch オブジェクトは HTTP バインディングを使用しています
- Dispatch オブジェクトはメッセージモードを使用しています

DataSource オブジェクトは、URL、ファイル、バイト配列など、さまざまなソースからの MIME タイプのデータを操作するためのメカニズムを提供します。

JAXB オブジェクトの使用

Dispatch オブジェクトは、生のメッセージを操作できる低レベルの API を目的としていますが、JAXB オブジェクトを操作することもできます。JAXB オブジェクトを操作するには、使用中の JAXB オブジェクトをマーシャリングおよびアンマーシャリングできる **JAXBContext** を Dispatch オブジェクトに渡す必要があります。**Dispatch** オブジェクトの作成時に **JAXBContext** が渡されます。

JAXBContext オブジェクトによって認識される任意の JAXB オブジェクトを、パラメーターとして **invoke()** メソッドに渡すことができます。返されたメッセージを **JAXBContext** オブジェクトによって認識される JAXB オブジェクトにキャストすることもできます。

JAXBContext オブジェクト作成の詳細は、39章*JAXBContext オブジェクトの使用* を参照してください。

41.1.3. ディスパッチオブジェクトの操作

手順

Dispatch オブジェクトを使用してリモートサービスを呼び出すには、次の手順に従う必要があります。

1. Dispatch オブジェクトを 作成 します。
2. リクエストメッセージを 設定 します。

3. 適切な **invoke()** メソッドを呼び出します。
4. 応答メッセージを解析します。

ディスパッチオブジェクトの作成

Dispatch オブジェクトを作成するには、次のようにします。

1. Dispatch オブジェクトが呼び出しを行うサービスを定義する **wsdl:service** 要素を表す **Service** オブジェクトを作成します。「Serivce オブジェクトの作成」を参照してください。
2. 例41.1「**createDispatch() メソッド**」 に示すように、 **Service** オブジェクトの **createDispatch()** メソッドを使用して、Dispatch オブジェクトを作成します。

例41.1 createDispatch() メソッド

public**Dispatch<T>createDispatchQNameportNamejava.lang.Class<T>typeService.Mod emode**WebServiceException

注記

JAXB オブジェクトを使用している場合、**createDispatch()** のメソッド署名は public**Dispatch<T>createDispatchQNameportNamejavax.xml.bind.JAXBCont extcontextService.Modemode**WebServiceException です。

表41.1「**createDispatch() のパラメーター**」で、**createDispatch()** メソッドのパラメーターを説明します。

表41.1 createDispatch() のパラメーター

パラメーター	説明
portName	Dispatch オブジェクトが呼び出しを行うサービスプロバイダーを表す **wsdl:port** 要素の QName を指定します。
type	Dispatch オブジェクトによって使用されるオブジェクトのデータ型を指定します。「データ型」を参照してください。JAXB オブジェクトを操作する場合、このパラメーターは JAXB オブジェクトのマーシャリングおよびアンマーシャリングに使用される **JAXBContext** オブジェクトを指定します。
mode	Dispatch オブジェクトの使用モードを指定します。「使用モード」を参照してください。

例41.2「ディスパッチオブジェクトの作成」に、ペイロードモードで **DOMSource** オブジェクトと連携する Dispatch オブジェクトを作成するコードを示します。

例41.2 ディスパッチオブジェクトの作成

```
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);
    ...
```

リクエストメッセージの作成

Dispatch オブジェクトを操作する場合、リクエストは最初から作成する必要があります。開発者は、Dispatch オブジェクトに渡されるメッセージが、対象のサービスプロバイダーが処理できる要求と一致することを確認する責任があります。これには、サービスプロバイダーが使用するメッセージと、サービスプロバイダーが必要とするヘッダー情報 (ある場合) に関する正確な知識が必要です。

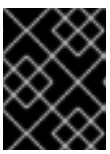
この情報は、メッセージを定義する WSDL ドキュメントまたは XML スキーマドキュメントによって提供されます。サービスプロバイダーは大きく異なりますが、従うべきガイドラインがいくつかあります。

- リクエストのルート要素は、呼び出される操作に対応する **wsdl:operation** 要素の **name** 属性の値に基づく。

警告

呼び出されるサービスが doc/literal ベアメッセージを使用する場合、リクエストのルート要素は、**wsdl:operation** 要素によって参照される **wsdl:part** 要素の **name** 属性の値に基づきます。

- リクエストのルート要素は名前空間で修飾されています。
- 呼び出されるサービスが rpc/literal メッセージを使用する場合、リクエストの最上位要素は名前空間で修飾されません。



重要

トップレベル要素の子は、名前空間で修飾されている場合があります。確実にするには、それらのスキーマ定義を確認する必要があります。

- 呼び出されるサービスが rpc/literal メッセージを使用する場合、最上位の要素を null にすることはできません。
- 呼び出されるサービスが doc/literal メッセージを使用する場合、メッセージのスキーマ定義によって、要素のいずれかが名前空間で修飾されているかどうかが判別されます。

サービスが XML メッセージを使用する方法の詳細については、WS-I 基本プロファイル を参照してください。

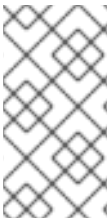
同期呼び出し

応答を生成する同期呼び出しを行うコンシューマーの場合は、例41.3「**Dispatch.invoke()** メソッド」に記載されている Dispatch オブジェクトの **invoke()** メソッドを使用します。

例41.3 Dispatch.invoke() メソッド

TinvokeTmsgWebServiceException

invoke() メソッドに渡される応答とリクエストの両方の型は、Dispatch オブジェクトの作成時に決定されます。たとえば、**createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE)** を使用して Dispatch オブジェクトを作成する場合、応答とリクエストの両方が **SOAPMessage** オブジェクトになります。



注記

JAXB オブジェクトを使用する場合、応答とリクエストの両方は、提供された **JAXBContext** オブジェクトがマーシャリングおよびアンマーシャリングできる任意の型にすることができます。また、応答と要求は異なる JAXB オブジェクトにすることができます。

例41.4「ディスパッチオブジェクトを使用した同期呼び出しの作成」に、**DOMSource** オブジェクトを使用してリモートサービスで同期呼び出しを行うコードを示します。

例41.4 ディスパッチオブジェクトを使用した同期呼び出しの作成

```
// 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);
```

非同期呼び出し

ディスパッチオブジェクトは、非同期呼び出しもサポートします。40章*非同期アプリケーションの開発*で説明されている高レベルの非同期 API と同様に、ディスパッチオブジェクトは、ポーリングアプローチとコールバックアプローチの両方を使用できます。

ポーリングアプローチを使用する場合、**invokeAsync()** メソッドは、応答が到達しているかどうかを確認するためにポーリングできる **Response<t>** オブジェクトを返します。例41.5「ポーリング用の **Dispatch.invokeAsync()** メソッド」 は、ポーリングアプローチを使用して非同期呼び出しを行うために使用されるメソッドのシグネチャーを示しています。

例41.5 ポーリング用の Dispatch.invokeAsync() メソッド

Response <T>invokeAsyncTmsg WebServiceException

非同期呼び出しにポーリングアプローチを使用する方法の詳細については、「ポーリングアプローチを使用した非同期クライアントの実装」 を参照してください。

コールバックアプローチを使用する場合は、**invokeAsync()** メソッドは、応答が返される時に処理する AsyncHandler 実装を受け取ります。例41.6「コールバックを使用した **Dispatch.invokeAsync()** メソッド」 は、コールバックアプローチを使用して非同期呼び出しを行うために使用されるメソッドのシグネチャーを示しています。

例41.6 コールバックを使用した Dispatch.invokeAsync() メソッド

Future<?>invokeAsyncTmsgAsyncHandler<T>handler WebServiceException

非同期呼び出しにコールバックアプローチを使用する方法の詳細については、「コールバックアプローチを使用した非同期クライアントの実装」 を参照してください。

注記

同期 **invoke()** メソッドと同様に、応答の型およびリクエストの型は、Dispatch オブジェクトの作成時に決定されます。

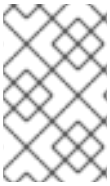
一方向の呼び出し

リクエストが応答を生成しない場合は、Dispatch オブジェクトの **invokeOneWay()** を使用してリモート呼び出しを行います。例41.7「**Dispatch.invokeOneWay()** メソッド」 は、このメソッドのシグネチャーを示しています。

例41.7 Dispatch.invokeOneWay() メソッド

invokeOneWayTmsgWebServiceException

リクエストのパッケージ化に使用されるオブジェクトのタイプは、Dispatch オブジェクトの作成時に決定されます。たとえば、**createDispatch(portName, DOMSource.class, Service.Mode.PAYLOAD)** を使用して Dispatch オブジェクトが作成されている場合、リクエストは **DOMSource** オブジェクトにパッケージ化されます。



注記

JAXB オブジェクトを使用する場合、応答とリクエストは、提供された **JAXBContext** オブジェクトがマーシャリングおよびアンマーシャリングできる任意の型にすることができます。

例41.8「ディスパッチオブジェクトを使用して一方向の呼び出しを行う」 は、JAXB オブジェクトを使用してリモートサービスで一方向の呼び出しを行うためのコードを示しています。

例41.8 ディスパッチオブジェクトを使用して一方向の呼び出しを行う

```
// 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. サービスプロバイダーでの XML の使用

概要

プロバイダーインターフェイスは、低レベルの JAX-WS API であり、メッセージを生の XML として直接処理するサービスプロバイダーを実装できます。メッセージは、プロバイダーインターフェイスを実装するオブジェクトに渡される前に JAXB オブジェクトにパッケージ化されません。

41.2.1. メッセージングモード

概要

プロバイダーインターフェイスを実装するオブジェクトには、2 つの **メッセージングモード** があります。

- メッセージ モード
- ペイロード モード

指定するメッセージングモードは、実装に渡されるメッセージングの詳細のレベルを決定します。

メッセージモード

メッセージモード を使用する場合、プロバイダーの実装は完全なメッセージで機能します。完全なメッセージには、バインディング固有のヘッダーとラッパーが含まれます。たとえば、SOAP バインディングを使用するプロバイダー実装は、完全に指定された SOAP メッセージとして要求を受信します。実装から返される応答は、完全に指定された SOAP メッセージである必要があります。

Provider 実装がメッセージモードを使用するように指定するには、例41.9「プロバイダー実装がメッセージモードを使用するように指定する」に示すように、値 java.xml.ws.Service.Mode.MESSAGE を **javax.xml.ws.ServiceMode** アノテーションの値として提供します。

例41.9 プロバイダー実装がメッセージモードを使用するように指定する

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
```

```
public class stockQuoteProvider implements Provider<SOAPMessage>
{
  ...
}
```

ペイロードモード

ペイロードモード では、プロバイダーの実装はメッセージのペイロードのみを処理します。たとえば、ペイロードモードで動作するプロバイダー実装は、SOAP メッセージの本文でのみ機能します。バインディングレイヤーは、バインディングレベルのラッパーとヘッダーを処理します。

Apache CXF XML バインディングなど、特別なラッパーを使用しないバインディングを使用する場合は、ペイロードモードとメッセージモードで同じ結果が得られます。

Provider 実装がペイロードモードを使用するように指定するには、例41.10「プロバイダー実装がペイロードモードを使用するように指定する」に示すように、値 java.xml.ws.Service.Mode.PAYLOAD を **javax.xml.ws.ServiceMode** アノテーションの値として提供します。

例41.10 プロバイダー実装がペイロードモードを使用するように指定する

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class stockQuoteProvider implements Provider<DOMSource>
{
  ...
}
```

@ServiceMode アノテーションの値を指定しない場合、Provider 実装はペイロードモードを使用します。

41.2.2. データ型

概要

これらは低レベルのオブジェクトであるため、プロバイダーの実装では、高レベルのコンシューマー API と同じ JAXB 生成タイプを使用できません。プロバイダーの実装は、次のタイプのオブジェクトで機能します。

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource

ソースオブジェクトの使用

プロバイダー実装は、javax.xml.transform.Source インターフェイスから派生したオブジェクトを受け入れて返すことができます。ソースオブジェクトは、XML ドキュメントを保持する低レベルのオブジェクトです。各 Source 実装は、保存された XML ドキュメントにアクセスしてその内容を操作するメソッドを提供します。次のオブジェクトは、ソースインターフェイスを実装します。

DOMSource

XML メッセージを Document Object Model (DOM) ツリーとして保持します。XML メッセージは、**getNode()** メソッドを使用してアクセスされる **Node** オブジェクトのセットとして保存されます。ノードは、**setNode()** メソッドを使用して更新または DOM ツリーに追加できます。

SAXSource

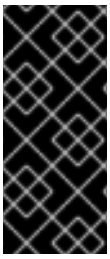
XML メッセージを Simple API for XML (SAX) オブジェクトとして保持します。SAX オブジェクトには、未加工のデータを保持する **InputSource** オブジェクトと、未加工のデータを解析する **XMLReader** オブジェクトが含まれます。

StreamSource

XML メッセージをデータストリームとして保持します。データストリームは、他のデータストリームと同じように操作できます。

汎用 Source オブジェクトを使用するように Provider オブジェクトを作成する場合、Apache CXF はメッセージを **SAXSource** オブジェクトとして返します。

この動作は、エンドポイントの **source-preferred-format** プロパティーを使用して変更できます。Apache CXF ランタイムの設定についての情報は、パートIV「Web サービスエンドポイントの設定」 を参照してください。



重要

Source オブジェクトを使用する場合、開発者は、必要なすべてのバインディング固有のラッパーがメッセージに追加されるようにする責任があります。たとえば、SOAP メッセージを期待するサービスと対話する場合、開発者は、必要な SOAP エンベロープが送信要求に追加され、SOAP エンベロープの内容が正しいことを確認する必要があります。

SOAPMessage オブジェクトの使用

以下の条件が満たされる場合、Provider 実装は **javax.xml.soap.SOAPMessage** オブジェクトを使用できます。

- プロバイダーの実装は SOAP バインディングを使用しています
- プロバイダーの実装はメッセージモードを使用しています

SOAPMessage オブジェクトは SOAP メッセージを保持します。これらには、1 つの **SOAPPart** オブジェクトと 0 個以上の **AttachmentPart** オブジェクトが含まれています。**SOAPPart** オブジェクトには、SOAP メッセージの SOAP 固有部分が含まれます。これには、SOAP エンベロープ、SOAP ヘッダー、および SOAP メッセージボディーが含まれます。**AttachmentPart** オブジェクトには、アタッチメントとして渡されるバイナリーデータが含まれます。

DataSource オブジェクトの使用

プロバイダーの実装では、次の条件が当てはまる場合に、javax.activation.DataSource インターフェイスを実装するオブジェクトを使用できます。

- 実装は HTTP バインディングを使用しています
- 実装はメッセージモードを使用しています

DataSource オブジェクトは、URL、ファイル、バイト配列など、さまざまなソースからの MIME タイプのデータを操作するためのメカニズムを提供します。

41.2.3. プロバイダーオブジェクトの実装

概要

プロバイダーインターフェイスは、比較的簡単に実装できます。実装する必要があるメソッドが **invoke()** の 1 つのみです。さらに、3 つの簡単な要件があります。

- 実装には **@WebServiceProvider** アノテーションが必要です。
- 実装には、デフォルトのパブリックコンストラクターが必要です。
- 実装は、プロバイダーインターフェイスの型付きバージョンを実装する必要があります。つまり、Provider<T> インターフェイスを実装することはできません。「データ型」 にリストされている具体的なデータ型を使用するバージョンのインターフェイスを実装する必要があります。たとえば、Provider<SAXSource> のインスタンスを実装できます。

プロバイダーインターフェイスの実装の複雑さは、要求メッセージを処理し、適切な応答を構築するロジックにあります。

メッセージの操作

高レベルの SEI ベースのサービス実装とは異なり、プロバイダー実装は生の XML データとして要求を受信し、生の XML データとして応答を送信する必要があります。これには、開発者が、実装されているサービスで使用されるメッセージについての深い知識を持っている必要があります。これらの詳細は通常、サービスを説明する WSDL ドキュメントに記載されています。

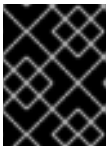
WS-I Basic Profile は、以下を含むサービスによって使用されるメッセージに関するガイドラインを提供します。

- リクエストのルート要素は、呼び出される操作に対応する **wsdl:operation** 要素の **name** 属性の値に基づく。

警告

サービスが doc/literal ベアメッセージを使用する場合、リクエストのルート要素は、**wsdl:operation** 要素によって参照される **wsdl:part** 要素の **name** 属性の値に基づきます。

- すべてのメッセージのルート要素は名前空間で修飾されています。
- サービスが rpc/literal メッセージを使用する場合、メッセージの最上位要素は名前空間で修飾されません。



重要

トップレベル要素の子は名前空間で修飾されている可能性がありますが、確実にするには、それらのスキーマ定義を確認する必要があります。

- サービスが rpc/literal メッセージを使用する場合、最上位の要素を null にすることはできません。

- サービスがドキュメント/リテラルメッセージを使用する場合、メッセージのスキーマ定義によって、要素のいずれかが名前空間で修飾されているかどうかが判別されます。

@WebServiceProvider アノテーション

JAX-WS にサービス実装として認識されるためには、Provider 実装に **@WebServiceProvider** アノテーションを付ける必要があります。

表41.2「**@WebServiceProvider** プロパティー」に、 **@WebServiceProvider** アノテーションに設定できるプロパティーを示します。

表41.2 @WebServiceProvider プロパティー

プロパティー	説明
portName	サービスのエンドポイントを定義する **wsdl:port** 要素の **name** 属性の値を指定します。
serviceName	サービスのエンドポイントが含まれる **wsdl:service** 要素の **name** 属性の値を指定します。
targetNamespace	サービスの WSDL 定義のターゲットネームスペースを指定します。
wsdlLocation	サービスを定義する WSDL ドキュメントの URI を指定します。

これらのプロパティーはすべてオプションであり、デフォルトでは空です。空のままにすると、Apache CXF は実装クラスからの情報を使用して値を作成します。

invoke() メソッドの実装

Provider インターフェイスには、実装する必要があるメソッド **invoke()** が 1 つだけあります。**invoke()** メソッドは、実装される Provider インターフェイスの型によって宣言されたオブジェクトの型にパッケージ化された受信リクエストを受け取り、同じ型のオブジェクトにパッケージ化された応答メッセージを返します。たとえば、Provider<SOAPMessage> インターフェイスの実装は、リクエストを **SOAPMessage** オブジェクトとして受け取り、応答を **SOAPMessage** オブジェクトとして返します。

プロバイダーの実装で使用されるメッセージングモードは、要求メッセージと応答メッセージに含まれるバインディング固有の情報の量を決定します。メッセージモードを使用する実装は、リクエストとともにすべてのバインディング固有のラッパーとヘッダーを受け取ります。また、バインディング固有のラッパーとヘッダーをすべて応答メッセージに追加する必要があります。ペイロードモードを使用する実装は、リクエストの本文のみを受信します。ペイロードモードを使用する実装によって返される XML ドキュメントは、要求メッセージの本文に配置されます。

例

例41.11「Provider<SOAPMessage> の実装」に、メッセージモードの **SOAPMessage** オブジェクトで動作する Provider 実装を示しています。

例41.11 Provider<SOAPMessage> の実装

```
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> の実装」 のコードは、以下を行います。

wsdl:service 要素の名前が **stockQuoteReporter** で、**wsdl:port** 要素が **stockQuoteReporterPort** という名前のサービスを実装する Provider オブジェクトを以下のクラスが実装することを指定する。

このプロバイダー実装がメッセージモードを使用することを指定します。

必要なデフォルトのパブリックコンストラクターを提供します。

SOAPMessage オブジェクトを受け取り、**SOAPMessage** オブジェクトを返す **invoke()** メソッドの実装を提供する。

着信 SOAP メッセージの本文から要求メッセージを抽出します。

要求メッセージのルート要素をチェックして、要求の処理方法を決定します。

応答の構築に必要なファクトリーを作成します。

応答の SOAP メッセージを作成します。

応答を **SOAPMessage** オブジェクトとして返す。

例41.12「Provider<DOMSource> の実装」に、ペイロードモードの **DOMSource** オブジェクトを使用する Provider 実装の例を示します。

例41.12 Provider<DOMSource> の実装

```
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> の実装」 のコードは、以下を行います。

wsdl:service 要素の名前が **stockQuoteReporter** で、**wsdl:port** 要素が **stockQuoteReporterPort** という名前のサービスを実装する Provider オブジェクトをクラスが実装することを指定する。

このプロバイダー実装がペイロードモードを使用することを指定します。

必要なデフォルトのパブリックコンストラクターを提供します。

DOMSource オブジェクトを受け取り、**DOMSource** オブジェクトを返す **invoke()** メソッドの実装を提供する。

第42章 コンテキストの使用

概要

JAX-WS は、コンテキストを使用して、メッセージングチェーンに沿ってメタデータを渡します。このメタデータは、そのスコープに応じて、実装レベルのコードからアクセスできます。また、実装レベルより下のメッセージを操作する JAX-WS ハンドラーからもアクセスできます。

42.1. コンテキストを理解する

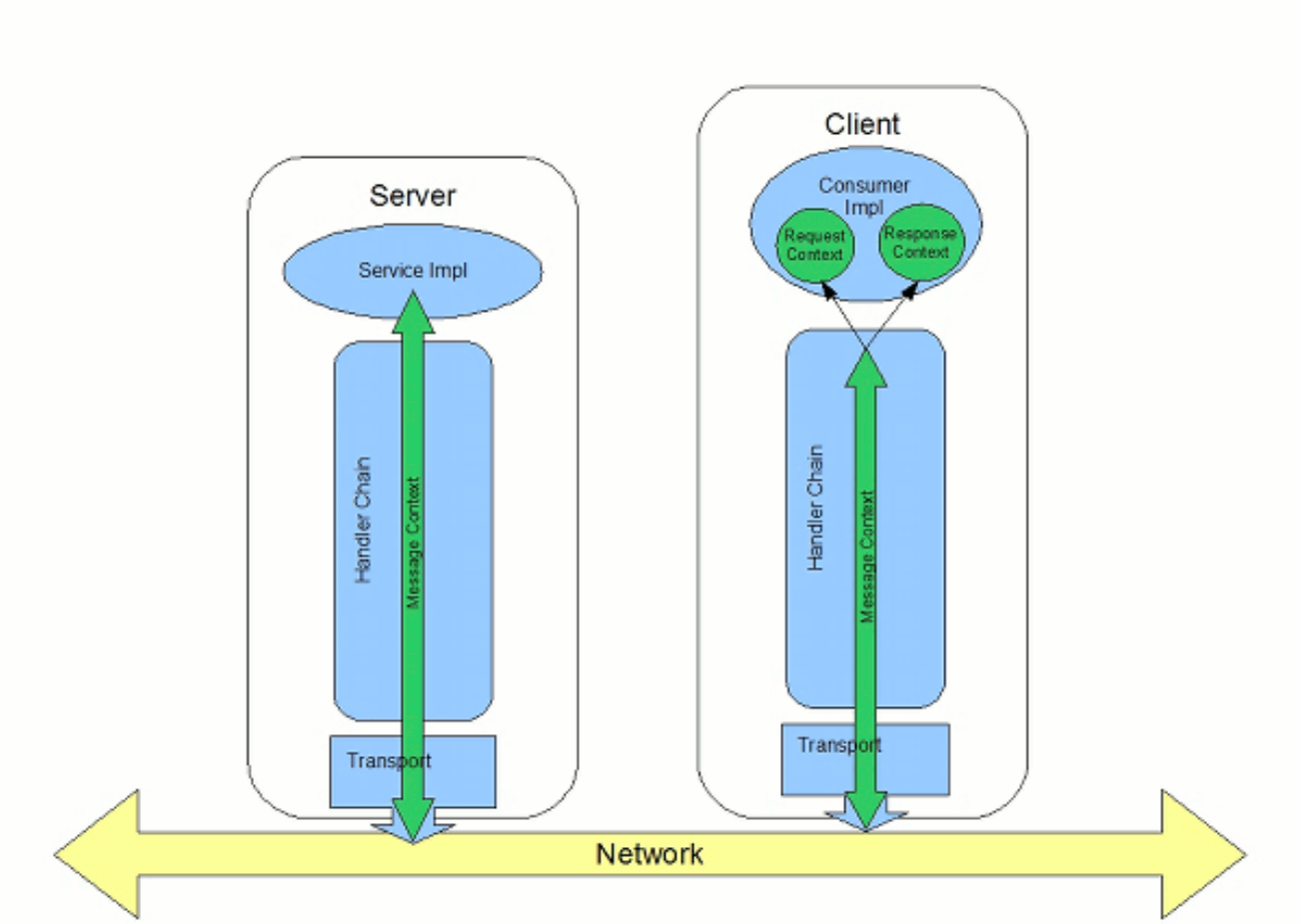
概要

多くの場合、メッセージに関する情報をアプリケーションの他の部分に渡す必要があります。Apache CXF は、コンテキストメカニズムを使用してこれを行います。コンテキストは、送信メッセージまたは受信メッセージに関連するプロパティーを保持するマップです。コンテキストに格納されるプロパティーは、通常、メッセージに関するメタデータと、メッセージの通信に使用される基になるトランスポートです。たとえば、HTTP 応答コードや JMS 相関 ID など、メッセージの送信に使用されるトランスポート固有のヘッダーは、JAX-WS コンテキストに格納されます。

コンテキストは、JAX-WS アプリケーションのすべてのレベルで使用できます。ただし、メッセージ処理スタックのどこでコンテキストにアクセスしているかによって、微妙に異なります。JAX-WS ハンドラーの実装は、コンテキストに直接アクセスでき、コンテキストに設定されているすべてのプロパティーにアクセスできます。サービス実装はコンテキストを注入することでそれらにアクセスし、**APPLICATION** スコープで設定されたプロパティーにのみアクセスできます。コンシューマーの実装は、**APPLICATION** スコープに設定されたプロパティーにのみアクセスできます。

図42.1「メッセージコンテキストとメッセージ処理パス」 コンテキストプロパティーが Apache CXF をどのように通過するかを示します。メッセージがメッセージングチェーンを通過すると、関連するメッセージコンテキストも一緒に通過します。

図42.1 メッセージコンテキストとメッセージ処理パス



プロパティーがコンテキストに保存される方法

メッセージコンテキストは、javax.xml.ws.handler.MessageContext インターフェイスのすべての実装です。MessageContext インターフェイスは、java.util.Map<String key, Object value> インターフェイスを拡張します。マップオブジェクトは、情報をキーと値のペアとして格納します。

メッセージコンテキストでは、プロパティーは名前と値のペアとして保存されます。プロパティーのキーは、プロパティーを識別する **String** です。プロパティーの値は、任意の Java オブジェクトに格納されている任意の値にすることができます。メッセージコンテキストから値が返される場合、アプリケーションは、期待するタイプを認識し、それに応じてキャストする必要があります。たとえば、プロパティーの値が **UserInfo** オブジェクトに保存される場合、プロパティーの値は **UserInfo** オブジェクトにキャストする必要がある **Object** オブジェクトとしてメッセージコンテキストから返されます。

メッセージコンテキストのプロパティーにもスコープがあります。スコープは、メッセージ処理チェーンのどこにプロパティーにアクセスできるかを決定します。

プロパティースコープ

メッセージコンテキストのプロパティーはスコープされます。プロパティーは、次のスコープのいずれかになります。

APPLICATION

APPLICATION としてスコープが設定されたプロパティーは、JAX-WS Handler 実装、コンシューマー実装コード、およびサービスプロバイダー実装コードで利用できます。ハンドラーがサービスプロバイダー実装にプロパティーを渡す必要がある場合は、プロパティーのスコープを

APPLICATION に設定します。コンシューマー実装またはサービスプロバイダー実装コンテキストのいずれかで設定されたすべてのプロパティーは、**APPLICATION** として自動的にスコープが設定されます。

HANDLER

HANDLER としてスコープ設定されたプロパティーは、JAX-WS Handler 実装でのみ利用できます。Handler 実装からメッセージコンテキストに保存されるプロパティーは、デフォルトで **HANDLER** としてスコープ設定されます。

メッセージコンテキストの **setScope()** メソッドを使用して、プロパティーのスコープを変更できます。例42.1「**MessageContext.setScope() メソッド**」 メソッドのシグネチャーを示します。

例42.1 MessageContext.setScope() メソッド

setScopeStringkeyMessageContext.Scopescopejava.lang.IllegalArgumentException

最初のパラメーターは、プロパティーのキーを指定します。2 番目のパラメーターは、プロパティーの新しいスコープを指定します。スコープは次のいずれかになります。

- **MessageContext.Scope.APPLICATION**
- **MessageContext.Scope.HANDLER**

ハンドラーのコンテキストの概要

JAX-WS ハンドラーインターフェイスを実装するクラスは、メッセージのコンテキスト情報に直接アクセスできます。メッセージのコンテキスト情報は、Handler 実装の **handleMessage()**、**handleFault()**、および **close()** メソッドに渡されます。

ハンドラーの実装は、スコープに関係なく、メッセージコンテキストに格納されているすべてのプロパティーにアクセスできます。また、論理ハンドラーは **LogicalMessageContext** と呼ばれる特別なメッセージコンテキストを使用します。**LogicalMessageContext** オブジェクトには、メッセージボディーの内容にアクセスするメソッドがあります。

サービス実装のコンテキストの概要

サービス実装は、メッセージコンテキストから **APPLICATION** としてスコープ設定されたプロパティーにアクセスできます。サービスプロバイダーの実装オブジェクトは、WebServiceContext オブジェクトを介してメッセージコンテキストにアクセスします。

詳細は、「サービス実装でのコンテキストの操作」 を参照してください。

コンシューマー実装のコンテキストの概要

コンシューマー実装は、メッセージコンテキストのコンテンツに間接的にアクセスできます。コンシューマーの実装には、2 つの別個のメッセージコンテキストがあります。

- リクエストコンテキスト–送信リクエストに使用されるプロパティーのコピーを保持します
- 応答コンテキスト–着信応答からのプロパティーのコピーを保持します

ディスパッチレイヤーは、コンシューマー実装のメッセージコンテキストとハンドラー実装で使用されるメッセージコンテキストの間でプロパティーを転送します。

要求がコンシューマー実装からディスパッチ層に渡されると、要求コンテキストの内容が、ディスパッ

チ層で使用されるメッセージコンテキストにコピーされます。サービスから応答が返されると、ディスパッチレイヤーはメッセージを処理し、適切なプロパティーをメッセージコンテキストに設定します。ディスパッチレイヤーが応答を処理すると、メッセージコンテキストの **APPLICATION** としてスコープ設定されたすべてのプロパティーをコンシューマー実装の応答コンテキストにコピーします。

詳細は、「コンシューマー実装でのコンテキストの操作」を参照してください。

42.2. サービス実装でのコンテキストの操作

概要

コンテキスト情報は、WebServiceContext インターフェイスを使用してサービス実装で利用できるようになります。WebServiceContext オブジェクトから、アプリケーションスコープの現在のリクエストのコンテキストプロパティーが反映された **MessageContext** オブジェクトを取得できます。プロパティーの値を操作することができ、それらは応答チェーンを介して伝播されます。

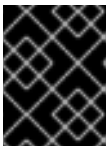
注記

MessageContext インターフェイスは、java.util.Map インターフェイスを継承します。その内容は、Map インターフェイスのメソッドを使用して操作できます。

コンテキストの取得

サービス実装でメッセージコンテキストを取得するには、次の手順を実行します。

1. WebServiceContext 型の変数を宣言します。
2. 変数に **javax.annotation.Resource** アノテーションを付け、コンテキスト情報が変数に注入されていることを示します。
3. **getMessageContext()** メソッドを使用して、WebServiceContext オブジェクトから **MessageContext** オブジェクトを取得します。



重要

getMessageContext() は、**@WebMethod** アノテーションが付けられたメソッドでのみ使用できます。

例42.2「サービス実装でのコンテキストオブジェクトの取得」 は、コンテキストオブジェクトを取得するためのコードを示しています。

例42.2 サービス実装でのコンテキストオブジェクトの取得

```
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();
  }

  ...
}
```

コンテキストからのプロパティーの読み取り

実装に MessageContext オブジェクトを取得したら、 例42.3「**MessageContext.get() メソッド**」 に記載されている **get()** メソッドを使用して、そこに保存されているプロパティーにアクセスできます。

例42.3 MessageContext.get() メソッド

VgetObjectkey

注記

この **get()** は、マップインターフェイスから継承されます。

key パラメーターは、コンテキストから取得するプロパティーを表す文字列です。**get()** は、プロパティーの適切な型にキャストする必要があるオブジェクトを返します。表42.1「サービス実装コンテキストで使用可能なプロパティー」 は、サービス実装のコンテキストで使用可能ないくつかのプロパティーを示しています。

重要

コンテキストから返されるオブジェクトの値を変更すると、コンテキスト内のプロパティーの値も変更されます。

例42.4「サービスのメッセージコンテキストからのプロパティーの取得」 は、呼び出された操作を表す WSDL **operation** 要素の名前を取得するコードを示します。

例42.4 サービスのメッセージコンテキストからのプロパティーの取得

```
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);
```

コンテキストでのプロパティーの設定

実装に MessageContext オブジェクトを取得したら、例42.5「**MessageContext.put() メソッド**」に記載されている **put()** メソッドを使用して、プロパティーを設定し、既存のプロパティーを変更できます。

例42.5 MessageContext.put() メソッド

Vput**K**key**V**value**C**lassCast**E**xception**I**llegal**A**rgument**E**xception**N**ull**P**ointer**E**xception

設定中のプロパティーがメッセージコンテキストにすでに存在する場合、**put()** メソッドは既存の値を新しい値に置き換え、古い値を返します。プロパティーがメッセージコンテキストにまだ存在しない場合は、**put()** メソッドはプロパティーを設定し、**null** を返します。

例42.6「サービスのメッセージコンテキストでのプロパティーの設定」 は、HTTP リクエストの応答コードを設定するためのコードを示しています。

例42.6 サービスのメッセージコンテキストでのプロパティーの設定

```
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));
```

サポートされているコンテキスト

表42.1「サービス実装コンテキストで使用可能なプロパティー」 サービス実装オブジェクトのコンテキストを介してアクセス可能なプロパティーをリスト表示します。

表42.1 サービス実装コンテキストで使用可能なプロパティー

プロパティー名	説明
org.apache.cxf.message.Message	
PROTOCOL_HEADERS[a]	トランスポート固有のヘッダー情報を指定します。この値は **java.util.Map<String, List<String>>** として保存されます。
RESPONSE_CODE	コンシューマーに返される応答コードを指定します。この値は **Integer** オブジェクトとして保存されます。
ENDPOINT_ADDRESS	サービスプロバイダーのアドレスを指定します。この値は **String** として保存されます。
HTTP_REQUEST_METHOD	リクエストとともに送信される HTTP 動詞を指定します。この値は **String** として保存されます。

プロパティー名	説明
PATH_INFO	要求されているリソースのパスを指定します。この値は **String** として保存されます。 パスは、ホスト名の後、クエリー文字列の前の URI の部分です。たとえば、エンドポイントの URI が http://cxf.apache.org/demo/widgets の場合、パスは **/demo/widgets** になります。
QUERY_STRING	リクエストの呼び出しに使用される URI に添付されているクエリーがある場合は、それを指定します。この値は **String** として保存されます。 クエリーは、URI の最後の **?** の後に表示されます。たとえば、http://cxf.apache.org/demo/widgets?color にリクエストが行われた場合、クエリーは **color** になります。
MTOM_ENABLED	サービスプロバイダーが SOAP 添付ファイルに MTOM を使用できるかどうかを指定します。この値は **Boolean** として保存されます。
SCHEMA_VALIDATION_ENABLED	サービスプロバイダーがスキーマに対してメッセージを検証するかどうかを指定します。この値は **Boolean** として保存されます。
FAULT_STACKTRACE_ENABLED	ランタイムが障害メッセージとともにスタックトレースを提供するかどうかを指定します。この値は **Boolean** として保存されます。
CONTENT_TYPE	メッセージの MIME タイプを指定します。この値は **String** として保存されます。
BASE_PATH	要求されているリソースのパスを指定します。この値は **java.net.URL** として保存されます。 パスは、ホスト名の後、クエリー文字列の前の URI の部分です。たとえば、エンドポイントの URL が http://cxf.apache.org/demo/widgets の場合、ベースパスは **/demo/widgets** になります。
ENCODING	メッセージのエンコーディングを指定します。この値は **String** として保存されます。
FIXED_PARAMETER_ORDER	パラメーターを特定の順序でメッセージに表示する必要があるかどうかを指定します。この値は **Boolean** として保存されます。

プロパティー名	説明
MAINTAIN_SESSION	コンシューマーが将来の要求のために現在のセッションを維持するかどうかを指定します。この値は **Boolean** として保存されます。
WSDL_DESCRIPTION	実装されているサービスを定義する WSDL ドキュメントを指定します。この値は **org.xml.sax.InputSource** オブジェクトとして保存されます。
WSDL_SERVICE	実装されているサービスを定義する **wsdl:service** 要素の修飾名を指定します。この値は **QName** として保存されます。
WSDL_PORT	サービスにアクセスするのに使用するエンドポイントを定義する **wsdl:port** 要素の修飾名を指定します。この値は **QName** として保存されます。
WSDL_INTERFACE	実装されているサービスを定義する **wsdl:portType** 要素の修飾名を指定します。この値は **QName** として保存されます。
WSDL_OPERATION	コンシューマーによって呼び出される操作に対応する **wsdl:operation** 要素の修飾名を指定します。この値は **QName** として保存されます。
javax.xml.ws.handler.MessageContext	
MESSAGE_OUTBOUND_PROPERTY	メッセージがアウトバウンドであるかどうかを指定します。この値は **Boolean** として保存されます。**true** は、メッセージがアウトバウンドであることを指定します。
INBOUND_MESSAGE_ATTACHMENTS	リクエストメッセージに含まれる添付ファイルが含まれます。この値は **java.util.Map<String, DataHandler>** として保存されます。 マップのキー値は、ヘッダーの MIME Content-ID です。
OUTBOUND_MESSAGE_ATTACHMENTS	応答メッセージの添付ファイルが含まれます。この値は **java.util.Map<String, DataHandler>** として保存されます。 マップのキー値は、ヘッダーの MIME Content-ID です。

プロパティー名	説明
WSDL_DESCRIPTION	実装されているサービスを定義する WSDL ドキュメントを指定します。この値は **org.xml.sax.InputSource** オブジェクトとして保存されます。
WSDL_SERVICE	実装されているサービスを定義する **wsdl:service** 要素の修飾名を指定します。この値は **QName** として保存されます。
WSDL_PORT	サービスにアクセスするのに使用するエンドポイントを定義する **wsdl:port** 要素の修飾名を指定します。この値は **QName** として保存されます。
WSDL_INTERFACE	実装されているサービスを定義する **wsdl:portType** 要素の修飾名を指定します。この値は **QName** として保存されます。
WSDL_OPERATION	コンシューマーによって呼び出される操作に対応する **wsdl:operation** 要素の修飾名を指定します。この値は **QName** として保存されます。
HTTP_RESPONSE_CODE	コンシューマーに返される応答コードを指定します。この値は **Integer** オブジェクトとして保存されます。
HTTP_REQUEST_HEADERS	リクエストの HTTP ヘッダーを指定します。この値は **java.util.Map<String, List<String>>** として保存されます。
HTTP_RESPONSE_HEADERS	応答の HTTP ヘッダーを指定します。この値は **java.util.Map<String, List<String>>** として保存されます。
HTTP_REQUEST_METHOD	リクエストとともに送信される HTTP 動詞を指定します。この値は **String** として保存されます。
SERVLET_REQUEST	サーブレットのリクエストオブジェクトが含まれます。この値は **javax.servlet.http.HttpServletRequest** として保存されます。
SERVLET_RESPONSE	サーブレットの応答オブジェクトが含まれます。この値は **javax.servlet.http.HttpResponse** として保存されます。

プロパティー名	説明
SERVLET_CONTEXT	サーブレットのコンテキストオブジェクトが含まれます。この値は **javax.servlet.ServletContext** として保存されます。
PATH_INFO	要求されているリソースのパスを指定します。この値は **String** として保存されます。 パスは、ホスト名の後、クエリー文字列の前の URI の部分です。たとえば、エンドポイントの URL が http://cxf.apache.org/demo/widgets の場合、パスは **/demo/widgets** になります。
QUERY_STRING	リクエストの呼び出しに使用される URI に添付されているクエリーがある場合は、それを指定します。この値は **String** として保存されます。 クエリーは、URI の最後の **?** の後に表示されます。たとえば、http://cxf.apache.org/demo/widgets?color にリクエストが行われた場合、クエリー文字列は **color** になります。
REFERENCE_PARAMETERS	WS-Addressing 参照パラメーターを指定します。これには、**wsa:IsReferenceParameter** 属性が **true** に設定されているすべての SOAP ヘッダーが含まれます。この値は **java.util.List** として保存されます。
org.apache.cxf.transport.jms.JMSConstants	
JMS_SERVER_HEADERS	JMS メッセージヘッダーが含まれます。詳細は、「JMS メッセージプロパティーの操作」 を参照してください。
[a] HTTP を使用する場合、このプロパティーは標準の JAX-WS 定義のプロパティーと同じです。	

42.3. コンシューマー実装でのコンテキストの操作

概要

コンシューマー実装は、BindingProvider インターフェイスを介してコンテキスト情報にアクセスできます。BindingProvider インスタンスは、次の 2 つの別個のコンテキストでコンテキスト情報を保持します。

- リクエストコンテキスト **リクエストコンテキスト** を使用すると、アウトバウンドメッセージに影響を与えるプロパティーを設定できます。リクエストコンテキストプロパティーは特定のポートインスタンスに適用され、一度設定されると、プロパティーが明示的にクリアされるまで、プロパティーはポートで行われる後続のすべての操作呼び出しに影響します。たとえば、リクエストコンテキストプロパティーを使用して、接続タイムアウトを設定したり、ヘッダーで送信するためのデータを初期化したりできます。

- 応答コンテキスト **応答コンテキスト** を使用すると、現在のスレッドから行われた最後の操作呼び出しへの応答によって設定されたプロパティー値を読み取ることができます。応答コンテキストのプロパティーは、操作が呼び出されるたびにリセットされます。たとえば、応答コンテキストプロパティーにアクセスして、最後の受信メッセージから受信したヘッダー情報を読み取ることができます。

重要

コンシューマー実装は、メッセージコンテキストのアプリケーションスコープに配置されている情報にのみアクセスできます。

コンテキストの取得

コンテキストは、javax.xml.ws.BindingProvider インターフェイスを使用して取得されます。BindingProvider インターフェイスには、コンテキストを取得するための 2 つのメソッドがあります。

- **getRequestContext()** 例42.7「**getRequestContext()** メソッド」に示されている **getRequestContext()** メソッドは、リクエストコンテキストを **Map** オブジェクトとして返します。返される **Map** オブジェクトを使用して、コンテキストの内容を直接操作できます。

  例42.7 getRequestContext() メソッド

  ```
  Map<String, Object>getRequestContext
  ```

- **getResponseContext()** 例42.8「**getResponseContext()** メソッド」に示されている **getResponseContext()** は、応答コンテキストを **Map** オブジェクトとして返します。返される **Map** オブジェクトの内容は、現在のスレッドで行われたモートサービスで最後に成功したリクエストからの応答コンテキストの内容の状態を反映しています。

  例42.8 getResponseContext() メソッド

  ```
  Map<String, Object>getResponseContext
  ```

プロキシーオブジェクトは BindingProvider インターフェイスを実装しているため、BindingProvider オブジェクトはプロキシーオブジェクトをキャストすることで取得できます。BindingProvider オブジェクトから取得されたコンテキストは、それを作成するために使用されたプロキシーオブジェクトで呼び出された操作に対してのみ有効です。

例42.9「コンシューマーの要求コンテキストの取得」 プロキシーのリクエストコンテキストを取得するためのコードを示しています。

例42.9 コンシューマーの要求コンテキストの取得

```
// Proxy widgetProxy obtained previously
BindingProvider bp = (BindingProvider)widgetProxy;
Map<String, Object> requestContext = bp.getRequestContext();
```

コンテキストからのプロパティーの読み取り

コンシューマーコンテキストは **java.util.Map<String, Object>** オブジェクトに保存されます。マップは、**String** オブジェクトであるキーと任意のオブジェクトが含まれる値を持ちます。応答コンテキストプロパティーのマッピングにあるエントリーにアクセスするには、**java.util.Map.get()** を使用します。

特定のコンテキストプロパティー **ContextPropertyName** を取得するには、例42.10「応答コンテキストプロパティーの読み取り」 のコードを使用します。

例42.10 応答コンテキストプロパティーの読み取り

```
// 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);
```

コンテキストでのプロパティーの設定

コンシューマーコンテキストは、**java.util.Map<String, Object>** オブジェクトに保存されるハッシュマップです。マップには、**String** オブジェクトであるキーと任意のオブジェクトである値が含まれます。コンテキストのプロパティーを設定するには、**java.util.Map.put()** メソッドを使用します。

要求コンテキストと応答コンテキストの両方でプロパティーを設定できますが、メッセージ処理に影響を与えるのは要求コンテキストに加えられた変更のみです。応答コンテキストのプロパティーは、現在のスレッドで各リモート呼び出しが完了するとリセットされます。

例42.11「リクエストコンテキストプロパティーの設定」 に示すコードは、BindingProvider.ENDPOINT_ADDRESS_PROPERTY の値を設定して、ターゲットサービスプロバイダーのアドレスを変更します。

例42.11 リクエストコンテキストプロパティーの設定

```
// 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

プロパティーがリクエストコンテキストに設定されると、その値は後続のすべてのリモート呼び出しに使用されます。値を変更すると、変更した値が使用されます。

サポートされているコンテキスト

Apache CXF は、コンシューマー実装で次のコンテキストプロパティーをサポートします。

表42.2 コンシューマーコンテキストのプロパティー

プロパティー名	説明
javax.xml.ws.BindingProvider	
ENDPOINT_ADDRESS_PROPERTY	ターゲットサービスのアドレスを指定します。この値は **String** として保存されます。
USERNAME_PROPERTY[a]	HTTP 基本認証に使用されるユーザー名を指定します。この値は **String** として保存されます。
PASSWORD_PROPERTY[b]	HTTP 基本認証に使用されるパスワードを指定します。この値は **String** として保存されます。
SESSION_MAINTAIN_PROPERTY[c]	クライアントがセッション情報を維持するかどうかを指定します。この値は **Boolean** オブジェクトとして保存されます。
org.apache.cxf.ws.addressing.JAXWSAConstants	
CLIENT_ADDRESSING_PROPERTIES	コンシューマーが目的のサービスプロバイダーに連絡するために使用する WS-Addressing 情報を指定します。この値は **org.apache.cxf.ws.addressing.AddressingProperties** として保存されます。
org.apache.cxf.transports.jms.context.JMSConstants	
JMS_CLIENT_REQUEST_HEADERS	メッセージの JMS ヘッダーが含まれます。詳細は、「JMS メッセージプロパティーの操作」 を参照してください。

[a] このプロパティーは、HTTP セキュリティー設定で定義されたユーザー名によって上書きされます。

[b] このプロパティーは、HTTP セキュリティー設定で定義されたパスワードによって上書きされます。

[c] Apache CXF はこのプロパティーを無視します。

42.4. JMS メッセージプロパティーの操作

概要

Apache CXF JMS トランスポートには、JMS メッセージのプロパティーを検査するために使用できるコンテキストメカニズムがあります。コンテキストメカニズムを使用して、JMS メッセージのプロパティーを設定することもできます。

42.4.1. JMS メッセージヘッダーの検査

概要

コンシューマーとサービスは、異なるコンテキストメカニズムを使用して JMS メッセージヘッダーのプロパティーにアクセスします。ただし、両方のメカニズムはヘッダープロパティーを **org.apache.cxf.transports.jms.context.JMSMessageHeadersType** オブジェクトとして返します。

サービスでの JMS メッセージヘッダーの取得

WebServiceContext オブジェクトから JMS メッセージヘッダーのプロパティーを取得するには、以下を行います。

1. 「コンテキストの取得」 で説明されているようにコンテキストを取得します。
2. org.apache.cxf.transports.jms.JMSConstants.JMS_SERVER_HEADERS パラメーターと共にメッセージコンテキストの **get()** メソッドを使用して、メッセージコンテキストからメッセージヘッダーを取得します。

例42.12「サービス実装での JMS メッセージヘッダーの取得」 は、サービスのメッセージコンテキストから JMS メッセージヘッダーを取得するためのコードを示しています。

例42.12 サービス実装での JMS メッセージヘッダーの取得

```
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);
      ...
    }
     ...
}
```

コンシューマーでの JMS メッセージヘッダープロパティーの取得

メッセージが JMS トランスポートから正常に取得されると、コンシューマーの応答コンテキストを使用して JMS ヘッダーのプロパティーを検査できます。さらに、「クライアント受信タイムアウト」 で説明されているように、クライアントがタイムアウトする前に応答を待機する時間の長さを設定または確認できます。 コンシューマーの応答コンテキストから JMS メッセージヘッダーを取得するには、次の手順を実行します。

1. 「コンテキストの取得」 で説明されているように応答コンテキストを取得します。

2. org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RESPONSE_HEADERS をパラメーターとしてコンテキストの **get()** メソッドを使用して、応答コンテキストから JMS メッセージヘッダーのプロパティーを取得します。

例42.13「コンシューマー応答ヘッダーからの JMS ヘッダーの取得」 は、コンシューマーの応答コンテキストから JMS メッセージヘッダープロパティーを取得するためのコードを示しています。

例42.13 コンシューマー応答ヘッダーからの JMS ヘッダーの取得

```
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「コンシューマー応答ヘッダーからの JMS ヘッダーの取得」 のコードは、以下を行います。

プロキシーを BindingProvider にキャストします。

応答コンテキストを取得します。

応答コンテキストから JMS メッセージヘッダーを取得します。

42.4.2. メッセージヘッダーのプロパティーの検査

標準 JMS ヘッダープロパティー

表42.3「JMS ヘッダーのプロパティー」 検査できる JMS ヘッダーの標準プロパティーをリスト表示します。

表42.3 JMS ヘッダーのプロパティー

プロパティー名	プロパティータイプ	ゲッターメソッド
相関 ID	**string**	**getJMSCorralationID()**
配信モード	**int**	**getJMSDeliveryMode()**
メッセージの有効期限	**long**	**getJMSExpiration()**
メッセージ ID	**string**	**getJMSMessageID()**
優先度	**int**	**getJMSPriority()**
再配信	**boolean**	**getJMSRedlivered()**
タイムスタンプ	**long**	**getJMSTimeStamp()**

プロパティー名	プロパティータイプ	ゲッターメソッド
型	**string**	**getJMSType()**
有効期間	**long**	**getTimeToLive()**

オプションのヘッダープロパティー

さらに、**JMSMessageHeadersType.getProperty()** を使用して、JMS ヘッダーに保存されているすべてのオプションプロパティーを検証することもできます。オプションのプロパティーは、**org.apache.cxf.transports.jms.context.JMSPropertyType** の **List** として返されます。オプションのプロパティーは、名前と値のペアとして保存されます。

例

例42.14「JMS ヘッダープロパティーの読み取り」 は、応答コンテキストを使用して一部の JMS プロパティーを検査するためのコードを示しています。

例42.14 JMS ヘッダープロパティーの読み取り

```
// 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「JMS ヘッダープロパティーの読み取り」 のコードは、以下を行います。

メッセージの相関 ID の値を出力します。

メッセージの priority プロパティーの値を出力します。

メッセージの再配信されたプロパティーの値を出力します。

メッセージのオプションのヘッダープロパティーのリストを取得します。

プロパティーのリストをトラバースするために **Iterator** を取得する。

オプションのプロパティーのリストを繰り返し、それらの名前と値を出力します。

42.4.3. JMS プロパティーの設定

概要

概要

コンシューマーエンドポイントでリクエストコンテキストを使用すると、JMS メッセージヘッダープロパティーの数とコンシューマーエンドポイントのタイムアウト値を設定できます。これらのプロパティーは、1 回の呼び出しで有効です。サービスプロキシーで操作を呼び出すたびに、それらをリセットする必要があります。

サービスにヘッダープロパティーを設定することはできないことに注意してください。

JMS ヘッダーのプロパティー

表42.4「設定可能な JMS ヘッダープロパティー」 は、コンシューマーエンドポイントのリクエストコンテキストを使用して設定できる JMS ヘッダーのプロパティーをリスト表示します。

表42.4 設定可能な JMS ヘッダープロパティー

プロパティー名	プロパティータイプ	セッターメソッド
相関 ID	**string**	**setJMSCorralationID()**
配信モード	**int**	**setJMSDeliveryMode()**
優先度	**int**	**setJMSPriority()**
有効期間	**long**	**setTimeToLive()**

これらのプロパティーを設定するには、次のようにします。

1. **org.apache.cxf.transports.jms.context.JMSMessageHeadersType** オブジェクトを作成します。
2. 表42.4「設定可能な JMS ヘッダープロパティー」 で説明されている適切なセッターメソッドを使用して、設定する値を入力します。
3. org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_REQUEST_HEADERS を第 1 引数として、新しい **JMSMessageHeadersType** オブジェクトを第 2 引数として使用してリクエストコンテキストの **put()** メソッドを呼び出すことで、値をリクエストコンテキストに設定します。

オプションの JMS ヘッダープロパティー

オプションのプロパティーを JMS ヘッダーに設定することもできます。オプションの JMS ヘッダープロパティーは、他の JMS ヘッダープロパティーを設定するために使用される **JMSMessageHeadersType** オブジェクトに保存されます。それらは、**org.apache.cxf.transports.jms.context.JMSPropertyType** オブジェクトが含まれる **List** オブジェクトとして保存されます。オプションのプロパティーを JMS ヘッダーに追加するには、次の手順を実行します。

1. **JMSPropertyType** オブジェクトを作成します。
2. **setName()** を使用して、プロパティーの name フィールドを設定します。
3. **setValue()** を使用して、プロパティーの value フィールドを設定します。

4. **JMSMessageHeadersType.getProperty().add(JMSPropertyType)** を使用して、JMS メッセージヘッダーにプロパティーを追加します。

5. すべてのプロパティーがメッセージヘッダーに追加されるまで、この手順を繰り返します。

クライアント受信タイムアウト

JMS ヘッダーのプロパティーに加えて、コンシューマーエンドポイントがタイムアウトする前に応答を待機する時間を設定できます。
org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RECEIVE_TIMEOUT を第 1 引数として、コンシューマーが待機するミリ秒単位の時間を表す **long** を第 2 引数としてリクエストコンテキストの **put()** メソッドを呼び出すことで、値を設定します。

例

例42.15「リクエストコンテキストを使用した JMS プロパティーの設定」 は、リクエストコンテキストを使用していくつかの JMS プロパティーを設定するためのコードを示しています。

例42.15 リクエストコンテキストを使用した JMS プロパティーの設定

```
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「リクエストコンテキストを使用した JMS プロパティーの設定」 のコードは、以下を行います。

JMS プロパティーを変更するプロキシーの **InvocationHandler** を取得する。

InvocationHandler が **BindingProvider** であるかどうかを確認する。

返された **InvocationHandler** オブジェクトを **BindingProvider** オブジェクトにキャストし、リクエストコンテキストを取得する。

リクエストコンテキストを取得します。

JMSMessageHeadersType オブジェクトを作成し、新しいメッセージヘッダーの値を保持する。

相関 ID を設定します。

エクスパレーションプロパティーを 60 分に設定します。

新しい **JMSPropertyType** オブジェクトを作成する。

オプションのプロパティーの値を設定します。

オプションのプロパティーをメッセージヘッダーに追加します。

JMS メッセージヘッダー値をリクエストコンテキストに設定します。

クライアントの受信タイムアウトプロパティーを 1 秒に設定します。

第43章 ハンドラーの作成

概要

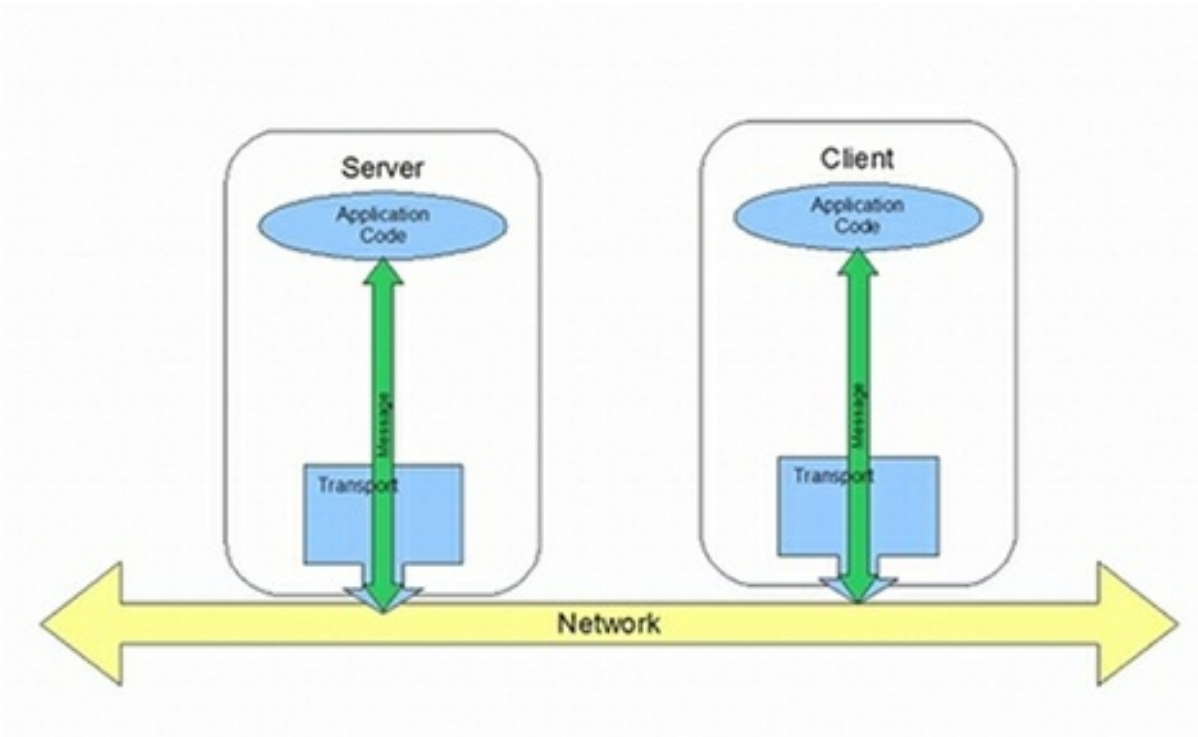
JAX-WS は、メッセージ処理モジュールをアプリケーションに追加するための柔軟なプラグインフレームワークを提供します。ハンドラーと呼ばれるこれらのモジュールは、アプリケーションレベルのコードから独立しており、低レベルのメッセージ処理機能を提供できます。

43.1. ハンドラー: はじめに

概要

サービスプロキシーがサービスで操作を呼び出すと、操作のパラメーターが Apache CXF に渡され、そこでメッセージに組み込まれ、ネットワークに配置されます。サービスがメッセージを受信すると、Apache CXF はネットワークからメッセージを読み取り、メッセージを再構築してから、操作の実装を担当するアプリケーションコードに操作パラメーターを渡します。アプリケーションコードが要求の処理を終了すると、応答メッセージは、要求を発信したサービスプロキシーへのトリップ時に同様の一連のイベントを実行します。これは、図43.1「メッセージ交換パス」 に示されています。

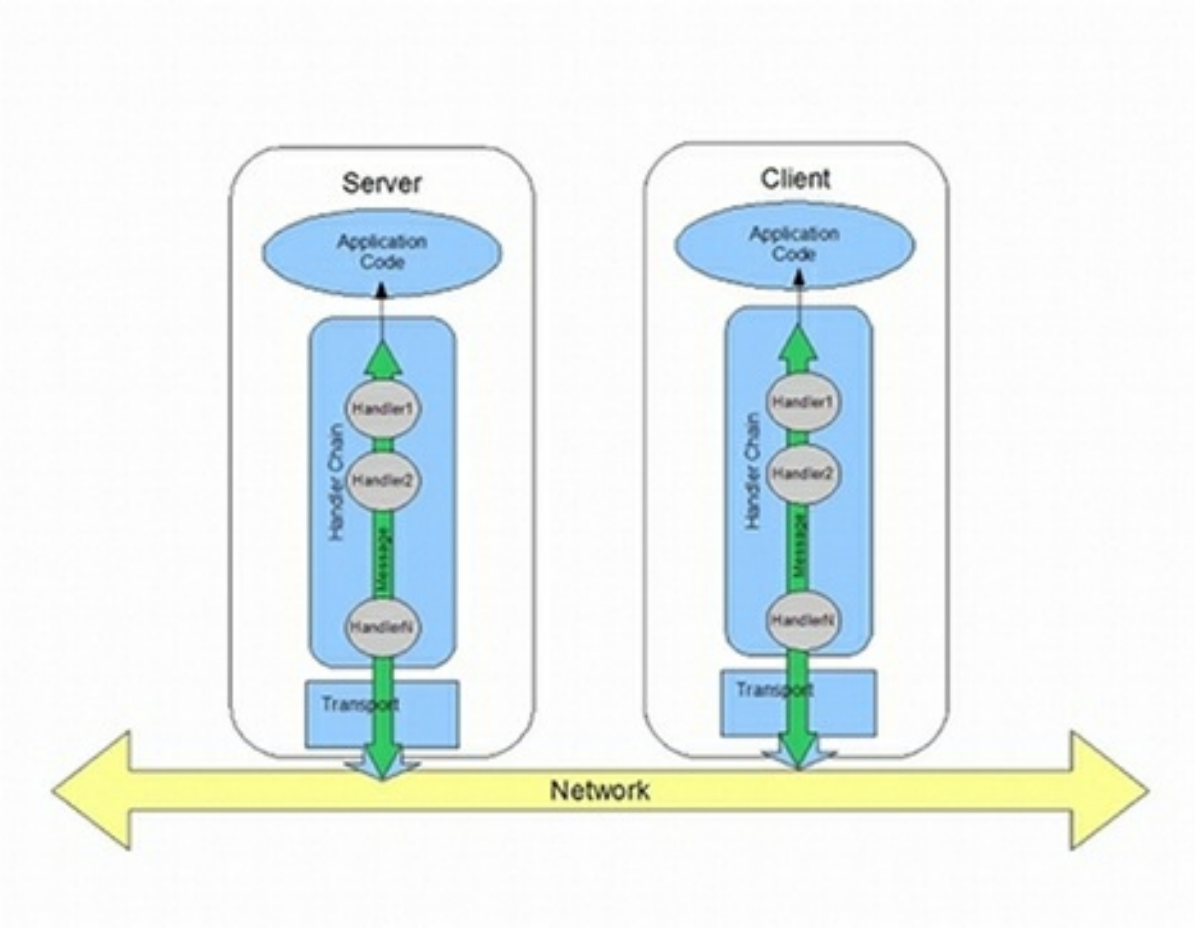
図43.1 メッセージ交換パス



JAX-WS は、アプリケーションレベルのコードとネットワークの間でメッセージデータを操作するためのメカニズムを定義します。たとえば、オープンネットワークを介して渡されるメッセージデータを独自の暗号化メカニズムを使用して暗号化することができます。データを暗号化および復号化する JAX-WS ハンドラーを作成できます。次に、ハンドラーをすべてのクライアントとサーバーのメッセージ処理チェーンに挿入できます。

図43.2「ハンドラーを使用したメッセージ交換パス」 に示すように、ハンドラーは、アプリケーションレベルのコードとメッセージをネットワークに配置するトランスポートコードの間を移動するチェーンに配置されます。

図43.2 ハンドラーを使用したメッセージ交換パス



ハンドラーの種類

JAX-WS 仕様では、次の 2 つの基本的なハンドラータイプが定義されています。

- 論理ハンドラー 論理ハンドラーは、メッセージペイロードとメッセージコンテキストに格納されているプロパティーを処理できます。たとえば、アプリケーションが純粋な XML メッセージを使用する場合、論理ハンドラーはメッセージ全体にアクセスできます。アプリケーションが SOAP メッセージを使用する場合、論理ハンドラーは SOAP 本体のコンテンツにアクセスできます。メッセージコンテキストに配置されない限り、SOAP ヘッダーまたは添付ファイルにアクセスすることはできません。
  論理ハンドラーは、ハンドラーチェーン上のアプリケーションコードの最も近くに配置されます。これは、メッセージがアプリケーションコードからトランスポートに渡されるときに最初に実行されることを意味します。ネットワークからメッセージを受信してアプリケーションコードに戻すと、論理ハンドラーが最後に実行されます。

- プロトコルハンドラー プロトコルハンドラーは、ネットワークから受信したメッセージ全体と、メッセージコンテキストに格納されているプロパティーを処理できます。たとえば、アプリケーションが SOAP メッセージを使用する場合、プロトコルハンドラーは、SOAP 本体、SOAP ヘッダー、および添付ファイルのコンテンツにアクセスできます。
  プロトコルハンドラーは、ハンドラーチェーンのトランスポートの最も近くに配置されます。これは、ネットワークからメッセージを受信したときに最初に実行されることを意味します。アプリケーションコードからネットワークにメッセージが送信されると、プロトコルハンドラーが最後に実行されます。

注記

Apache CXF でサポートされている唯一のプロトコルハンドラーは、SOAP に固有のものです。

ハンドラーの実装

2 つのハンドラータイプの違いは非常に微妙であり、共通のベースインターフェイスを共有しています。共通の親子関係があるため、論理ハンドラーとプロトコルハンドラーは、次のような実装が必要な多くのメソッドを共有します。

- handleMessage() **handleMessage()** メソッドは、すべてのハンドラーの中心的なメソッドです。これは、通常のメッセージの処理を担当するメソッドです。
- handleFault() **handleFault()** は、障害メッセージを処理するメソッドです。
- close() メッセージがチェーンの最後に到達すると、ハンドラーチェーンのすべての実行済みハンドラーで **close()** が呼び出されます。これは、メッセージ処理中に消費されたリソースをクリーンアップするために使用されます。

論理ハンドラーの実装とプロトコルハンドラーの実装の違いは、次の点にあります。

- 実装されている特定のインターフェイス
  すべてのハンドラーは、ハンドラーインターフェイスから派生したインターフェイスを実装します。論理ハンドラーは、LogicalHandler インターフェイスを実装します。プロトコルハンドラーは、ハンドラーインターフェイスのプロトコル固有の拡張機能を実装します。たとえば、SOAP ハンドラーは SOAPHandler インターフェイスを実装します。
- ハンドラーが利用できる情報の量
  プロトコルハンドラーは、メッセージの内容と、メッセージの内容とともにパッケージ化されているすべてのプロトコル固有の情報にアクセスできます。論理ハンドラーは、メッセージの内容にのみアクセスできます。論理ハンドラーには、プロトコルの詳細に関する知識がありません。

アプリケーションへのハンドラーの追加

アプリケーションにハンドラーを追加するには、次のことを行う必要があります。

1. ハンドラーをサービスプロバイダー、コンシューマー、またはその両方で使用するかどうかを決定します。
2. どのタイプのハンドラーがジョブに最も適切かを判断します。
3. 適切なインターフェイスを実装します。
   論理ハンドラーを実装するには、「論理ハンドラーの実装」 を参照してください。

   プロトコルハンドラーを実装するには、「プロトコルハンドラーの実装」 を参照してください。
4. ハンドラーを使用するようにエンドポイントを設定します。「ハンドラーを使用するためのエンドポイントの設定」を参照してください。

43.2. 論理ハンドラーの実装

概要

論理ハンドラーは、javax.xml.ws.handler.LogicalHandler インターフェイスを実装します。例43.1「論理ハンドラーの概要」に示した LogicalHandler インターフェイスは、**LogicalMessageContext** オブジェクトを **handleMessage()** メソッドと **handleFault()** メソッドに渡します。.コンテキストオブジェクトは、メッセージの **本文** およびメッセージ交換のコンテキストに設定されたプロパティーへのアクセスを提供します。

例43.1 論理ハンドラーの概要

```
public interface LogicalHandler extends Handler
{
  boolean handleMessage(LogicalMessageContext context);
  boolean handleFault(LogicalMessageContext context);
  void close(LogicalMessageContext context);
}
```

手順

論理ハンダーを実装するには、次のようにします。

1. ハンドラーに必要な 「ハンドラーの初期化」 ロジックの実装。
2. 「論理ハンドラーでのメッセージの処理」 ロジックの実装。
3. 「障害メッセージの処理」 ロジックの実装。
4. 終了時に 「ハンドラーを閉じる」 ハンドラーのロジックを実装する。
5. 「ハンドラーのリリース」 ハンドラーのリソースのロジックを破棄される前に実装する。

43.3. 論理ハンドラーでのメッセージの処理

概要

通常のメッセージ処理は、**handleMessage()** メソッドによって処理されます。

handleMessage() メソッドは、メッセージボディーおよびメッセージコンテキストに保存されたプロパティーへのアクセスを提供する **LogicalMessageContext** オブジェクトを受け取ります。

メッセージ処理の継続方法に応じて、**handleMessage()** メソッドは true または false のいずれかを返します。例外を出力することもできます。

メッセージデータの取得

論理メッセージハンドラーに渡される LogicalMessageContext オブジェクトにより、コンテキストの **getMessage()** メソッドを使用してメッセージボディーにアクセスできます。例43.2「論理ハンドラーでメッセージペイロードを取得する方法」に示した **getMessage()** メソッドは、LogicalMessage オブジェクトとしてメッセージペイロードを返します。

例43.2 論理ハンドラーでメッセージペイロードを取得する方法

LogicalMessagegetMessage

LogicalMessage オブジェクトを取得したら、それを使用してメッセージ本文を操作できます。例43.3「論理メッセージホルダー」 に示す LogicalMessage インターフェイス、実際のメッセージ本文を操作するためのゲッターとセッターがあります。

例43.3 論理メッセージホルダー

```
LogicalMessageSourcegetPayloadObjectgetPayloadJAXBContextcontextsetPayloadObjectpayloadJAXBContextcontextsetPayloadSourcepayload
```

重要

メッセージペイロードの内容は、使用中のバインディングのタイプによって決まります。SOAP バインディングは、メッセージの SOAP 本文へのアクセスのみを許可します。XML バインディングにより、メッセージ本文全体にアクセスできます。

メッセージ本文を XML オブジェクトとして操作する

論理メッセージのゲッターとセッターの 1 つのペアは、メッセージペイロードを **javax.xml.transform.dom.DOMSource** オブジェクトとして操作します。

パラメーターのない **getPayload()** メソッドは、**DOMSource** オブジェクトとしてメッセージペイロードを返します。返されるオブジェクトは、実際のメッセージペイロードです。返されたオブジェクトに変更を加えると、メッセージ本文がすぐに変更されます。

単一の Source オブジェクトを取る **setPayload()** メソッドを使用して、メッセージのボディーを **DOMSource** オブジェクトに置き換えることができます。

メッセージ本文を JAXB オブジェクトとして操作する

ゲッターとセッターのもう 1 つのペアを使用すると、メッセージペイロードを JAXB オブジェクトとして操作できます。**JAXBContext** オブジェクトを使用して、メッセージペイロードを JAXB オブジェクトに変換します。

JAXB オブジェクトを使用するには、次のようにします。

1. メッセージボディーのデータ型を管理できる **JAXBContext** オブジェクトを取得します。**JAXBContext** オブジェクト作成の詳細は、39章*JAXBContext オブジェクトの使用* を参照してください。
2. 例43.4「メッセージ本文を JAXB オブジェクトとして取得する」 に示すようにメッセージ本文を取得します。

   例43.4 メッセージ本文を JAXB オブジェクトとして取得する

   ```
   JAXBContext jaxbc = JAXBContext(myObjectFactory.class);
   Object body = message.getPayload(jaxbc);
   ```

3. 返されたオブジェクトを適切なタイプにキャストします。
4. 必要に応じてメッセージ本文を操作します。
5. 例43.5「JAXB オブジェクトを使用したメッセージ本文の更新」 に示すように、更新されたメッセージ本文をコンテキストに戻します。

例43.5 JAXB オブジェクトを使用したメッセージ本文の更新

```
message.setPayload(body, jaxbc);
```

コンテキストプロパティーの操作

論理ハンドラーに渡される論理メッセージコンテキストは、アプリケーションのメッセージコンテキストのインスタンスであり、そこに格納されているすべてのプロパティーにアクセスできます。ハンドラーは、**APPLICATION** スコープが設定されたプロパティーと **HANDLER** スコープが設定されたプロパティーの両方にアクセスできます。

アプリケーションのメッセージコンテキストと同様に、論理メッセージコンテキストは Java Map のサブクラスです。コンテキストに保存されたプロパティーにアクセスするには、Map インターフェイスから継承された **get()** メソッドと **put()** メソッドを使用します。

デフォルトでは、論理ハンドラー内からメッセージコンテキストで設定したプロパティーには、**HANDLER** のスコープが割り当てられます。アプリケーションコードがプロパティーにアクセスできるようにするには、コンテキストの **setScope()** メソッドを使用して、プロパティーのスコープを明示的に **APPLICATION.** に設定する必要があります。

メッセージコンテキストでのプロパティーの操作の詳細については、「コンテキストを理解する」 を参照してください。

メッセージの方向を決定する

多くの場合、メッセージがハンドラーチェーンを通過する方向を知ることが重要です。たとえば、着信要求からセキュリティートークンを取得し、発信応答にセキュリティートークンを添付するとします。

メッセージの方向は、メッセージコンテキストのアウトバウンドメッセージプロパティーに保存されます。例43.6「SOAP メッセージコンテキストからのメッセージの方向の取得」 に示すように、MessageContext.MESSAGE_OUTBOUND_PROPERTY キーを使用して、メッセージコンテキストからアウトバウンドメッセージプロパティーを取得します。

例43.6 SOAP メッセージコンテキストからのメッセージの方向の取得

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

このプロパティーは **Boolean** オブジェクトとして保存されます。オブジェクトの **booleanValue()** メソッドを使用して、プロパティーの値を決定できます。プロパティーが true に設定されている場合、メッセージはアウトバウンドです。プロパティーが false に設定されている場合、メッセージはインバウンドです。

戻り値の決定

handleMessage() メソッドがそのメッセージ処理をどのように完了するかが、メッセージ処理の実施方法に直接的な影響を及ぼします。次のいずれかのアクションを実行することで完了できます。

1. true を返す - メッセージ処理を正常に続行する必要があることを示す true シグナルを Apache CXF ランタイムに返します。次のハンドラーがある場合は、その **handleMessage()** が呼び出されます。

2. false を返す - 通常のメッセージ処理を停止する必要がある false シグナルを Apache CXF ランタイムに返します。ランタイムがどのように進行するかは、**現在のメッセージ** に使用されているメッセージ交換パターンによって異なります。
   要求/応答メッセージ交換の場合は、次のことが起こります。

   a. メッセージ処理の方向が逆になります。
      たとえば、要求がサービスプロバイダーによって処理されている場合、メッセージはサービスの実装オブジェクトへの進行を停止します。代わりに、リクエストを発信したコンシューマーに返すために、バインディングに向けて送り返されます。

   b. 新しい処理方向でハンドラーチェーンに沿って存在するメッセージハンドラーには、チェーン内で存在する順序で **handleMessage()** メソッドが呼び出されます。

   c. メッセージがハンドラーチェーンの最後に到達すると、メッセージがディスパッチされます。
      一方向のメッセージ交換の場合は、次のことが起こります。

   d. メッセージ処理が停止します。

   e. これまで呼び出されたすべてのメッセージハンドラーには、その **close()** メソッドが呼び出されます。

   f. メッセージが送信されます。

3. ProtocolException 例外を出力する -ProtocolException 例外、またはこの例外のサブクラスを出力すると、障害メッセージ処理が開始されていることを Apache CXF ランタイムに通知します。ランタイムがどのように進行するかは、**現在のメッセージ** に使用されているメッセージ交換パターンによって異なります。
   要求/応答メッセージ交換の場合は、次のことが起こります。

   a. ハンドラーがまだ障害メッセージを作成していない場合、ランタイムはメッセージを障害メッセージでラップします。

   b. メッセージ処理の方向が逆になります。
      たとえば、要求がサービスプロバイダーによって処理されている場合、メッセージはサービスの実装オブジェクトへの進行を停止します。代わりに、リクエストを発信したコンシューマーに返すために、バインディングに向けて送り返されます。

   c. 新しい処理方向でハンドラーチェーンに沿って存在するメッセージハンドラーには、チェーン内で存在する順序で **handleFault()** メソッドが呼び出されます。

   d. 障害メッセージがハンドラーチェーンの最後に到達すると、ディスパッチされます。
      一方向のメッセージ交換の場合は、次のことが起こります。

   e. ハンドラーがまだ障害メッセージを作成していない場合、ランタイムはメッセージを障害メッセージでラップします。

   f. メッセージ処理が停止します。

   g. これまで呼び出されたすべてのメッセージハンドラーには、その **close()** メソッドが呼び出されます。

   h. 障害メッセージが送信されます。

4. その他のランタイム例外を出力する -ProtocolException 例外以外のランタイム例外を出力すると、メッセージ処理が停止することを Apache CXF ランタイムに通知します。これまで呼び出されたすべてのメッセージハンドラーには **close()** メソッドが呼び出され、例外がディスパッ

チされます。メッセージが要求/応答メッセージ交換の一部である場合、例外がディスパッチされ、要求を発信したコンシューマーに返されます。

例

例43.7「論理メッセージハンドラーメッセージ処理」に、サービスコンシューマーによって使用される論理メッセージハンドラーの **handleMessage()** メッセージの実装を示します。リクエストは、サービスプロバイダーに送信される前に処理されます。

例43.7 論理メッセージハンドラーメッセージ処理

```
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「論理メッセージハンドラーメッセージ処理」 のコードは、以下を行います。

メッセージがアウトバウンド要求であるかどうかを確認します。

メッセージがアウトバウンド要求である場合、ハンドラーは追加のメッセージ処理を行います。

メッセージコンテキストからメッセージペイロードの LogicalMessage 表現を取得します。

実際のメッセージペイロードを JAXB オブジェクトとして取得します。

リクエストが正しいタイプであることを確認します。

そうである場合、ハンドラーはメッセージの処理を続行します。

合計の値をチェックします。

しきい値の 20 未満の場合、応答を作成してクライアントに返します。

応答を作成します。

false を返し、メッセージ処理を停止してクライアントに応答を返します。

メッセージのタイプが正しくない場合、ランタイム例外を出力します。

この例外はクライアントに返されます。

メッセージがインバウンド応答であるか、合計がしきい値を満たさない場合、true を返します。

メッセージ処理は正常に続行されます。

JAXB マーシャリングエラーが発生した場合、ProtocolException を出力します。

例外は、現在のハンドラーとクライアント間でハンドラーの **handleFault()** メソッドによって処理された後にクライアントに渡して戻されます。

43.4. プロトコルハンドラーの実装

概要

プロトコルハンドラーは、使用中のプロトコルに固有です。Apache CXF は、JAX-WS で指定されている SOAP プロトコルハンドラーを提供します。SOAP プロトコルハンドラーは、javax.xml.ws.handler.soap.SOAPHandler インターフェイスを実装します。

例43.8「SOAPHandler の概要」に示されている SOAPHandler インターフェイスは、メッセージへのアクセスを **SOAPMessage** オブジェクトとして提供する SOAP 固有のメッセージコンテキストを使用します。また、SOAP ヘッダーにアクセスすることもできます。

例43.8 SOAPHandler の概要

```
public interface SOAPHandler extends Handler
{
  boolean handleMessage(SOAPMessageContext context);
  boolean handleFault(SOAPMessageContext context);
  void close(SOAPMessageContext context);
  Set<QName> getHeaders()
}
```

SOAP 固有のメッセージコンテキストを使用する他に、SOAP プロトコルハンドラーでは追加のメソッド **getHeaders()** を実装する必要があります。この追加のメソッドは、ハンドラーが処理できるヘッダーブロックの QName を返します。

手順

論理ハンダーを実装するには、次のようにします。

1. ハンドラーに必要な 「ハンドラーの初期化」 ロジックの実装。
2. 「SOAP ハンドラーでのメッセージの処理」 ロジックを実装する。
3. 「障害メッセージの処理」 ロジックの実装。
4. **getHeaders()** メソッドを実装します。
5. 終了時に 「ハンドラーを閉じる」 ハンドラーのロジックを実装する。
6. 「ハンドラーのリリース」 ハンドラーのリソースのロジックを破棄される前に実装する。

getHeaders() メソッドの実装

例43.9「**SOAPHander.getHeaders() メソッド**」 で示されている **getHeaders()** メソッドは、ハンドラーが処理する SOAP ヘッダーを Apache CXF ランタイムに通知します。ハンドラーが理解する各 SOAP ヘッダーの外部要素の QName を返します。

例43.9 SOAPHander.getHeaders() メソッド

```
Set<QName>getHeaders
```

多くの場合、単に null を返すだけで十分です。しかし、アプリケーションがいずれかの SOAP ヘッダーの **mustUnderstand** 属性を使用する場合は、アプリケーションの SOAP ハンドラーが理解するヘッダーを指定することが重要になります。ランタイムは、**mustUnderstand** 属性が **true** に設定されたヘッダーのリストから、登録されたすべてのハンドラーが認識する SOAP ヘッダーのセットを確認します。フラグが立てられたヘッダーのいずれかが理解されたヘッダーのリストにない場合、ランタイムはメッセージを拒否し、SOAP は例外を理解する必要があります。

43.5. SOAP ハンドラーでのメッセージの処理

概要

通常のメッセージ処理は、**handleMessage()** メソッドによって処理されます。

handleMessage() メソッドは、**SOAPMessage** オブジェクトとしてのメッセージボディーおよびメッセージに関連付けられた SOAP ヘッダーへのアクセスを提供する **SOAPMessageContext** オブジェクトを受け取ります。さらに、コンテキストは、メッセージコンテキストに格納されているすべてのプロパティーへのアクセスを提供します。

メッセージ処理の継続方法に応じて、**handleMessage()** メソッドは true または false のいずれかを返します。例外を出力することもできます。

メッセージ本文の操作

SOAP メッセージコンテキストの **getMessage()** メソッドを使用して SOAP メッセージを取得することができます。これは、メッセージをライブ **SOAPMessage** オブジェクトとして返します。ハンドラー内のメッセージへの変更は、コンテキストに格納されているメッセージに自動的に反映されます。

既存のメッセージを新しいメッセージに置き換える場合は、コンテキストの **setMessage()** メソッドを使用できます。**setMessage()** メソッドは **SOAPMessage** オブジェクトを取ります。

SOAP ヘッダーの取得

SOAP メッセージのヘッダーには、**SOAPMessage** オブジェクトの **getHeader()** メソッドを使用してアクセスできます。これにより、SOAP ヘッダーが **SOAPHeader** オブジェクトとして返されます。このオブジェクトを、処理するヘッダー要素を探すために検査する必要があります。

SOAP メッセージコンテキストは、例43.10「**SOAPMessageContext.getHeaders() メソッド**」に示される **getHeaders()** メソッドを提供します。これは、指定された SOAP ヘッダーの JAXB オブジェクトが含まれる配列を返します。

> **例43.10 SOAPMessageContext.getHeaders() メソッド**
>
> **Ojbect[]getHeadersQNameheaderJAXBContextcontextbooleanallRoles**

要素の QName を使用してヘッダーを指定します。**allRoles** パラメーターを false に設定すると、返されるヘッダーをさらに制限することができます。これは、アクティブな SOAP ロールに適用可能な SOAP ヘッダーのみを返すようにランタイムに指示します。

ヘッダーが見つからない場合、メソッドは空の配列を返します。

JAXBContext オブジェクトのインスタンス化の詳細は、39章*JAXBContext オブジェクトの使用*を参照してください。

コンテキストプロパティーの操作

論理ハンドラーに渡される SOAP メッセージコンテキストは、アプリケーションのメッセージコンテキストのインスタンスであり、そこに格納されているすべてのプロパティーにアクセスできます。ハンドラーは、**APPLICATION** スコープが設定されたプロパティーと **Handler** スコープが設定されたプロパティーの両方にアクセスできます。

アプリケーションのメッセージコンテキストと同様に、SOAP メッセージコンテキストは JavaMap のサブクラスです。コンテキストに保存されたプロパティーにアクセスするには、Map インターフェイスから継承された **get()** メソッドと **put()** メソッドを使用します。

デフォルトでは、論理ハンドラー内からコンテキストで設定したプロパティーには、**HANDLER** のス

コープが割り当てられます。アプリケーションコードがプロパティーにアクセスできるようにするには、コンテキストの **setScope()** メソッドを使用して、プロパティーのスコープを明示的に **APPLICATION.** に設定する必要があります。

メッセージコンテキストでのプロパティーの操作の詳細については、「コンテキストを理解する」 を参照してください。

メッセージの方向を決定する

多くの場合、メッセージがハンドラーチェーンを通過する方向を知ることが重要です。たとえば、送信メッセージにヘッダーを追加し、受信メッセージからヘッダーを削除するとします。

メッセージの方向は、メッセージコンテキストのアウトバウンドメッセージプロパティーに保存されます。例43.11「SOAP メッセージコンテキストからのメッセージの方向の取得」 に示すように、MessageContext.MESSAGE_OUTBOUND_PROPERTY キーを使用して、メッセージコンテキストからアウトバウンドメッセージプロパティーを取得します。

例43.11 SOAP メッセージコンテキストからのメッセージの方向の取得

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

このプロパティーは **Boolean** オブジェクトとして保存されます。オブジェクトの **booleanValue()** メソッドを使用して、プロパティーの値を決定できます。プロパティーが true に設定されている場合、メッセージはアウトバウンドです。プロパティーが false に設定されている場合、メッセージはインバウンドです。

戻り値の決定

handleMessage() メソッドがそのメッセージ処理をどのように完了するかが、メッセージ処理の実施方法に直接的な影響を及ぼします。次のいずれかのアクションを実行することで完了できます。

1. true を返す - メッセージ処理を正常に続行する必要があることを示す true シグナルを Apache CXF ランタイムに返します。次のハンドラーがある場合は、その **handleMessage()** が呼び出されます。

2. false を返す - 通常のメッセージ処理を停止する false シグナルを Apache CXF ランタイムに返します。ランタイムがどのように進行するかは、**現在のメッセージ** に使用されているメッセージ交換パターンによって異なります。
   要求/応答メッセージ交換の場合は、次のことが起こります。

   a. メッセージ処理の方向が逆になります。
      たとえば、要求がサービスプロバイダーによって処理されている場合、メッセージはサービスの実装オブジェクトへの進行を停止します。代わりに、リクエストを発信したコンシューマーに返すために、バインディングに向けて送り返されます。

   b. 新しい処理方向でハンドラーチェーンに沿って存在するメッセージハンドラーには、チェーン内で存在する順序で **handleMessage()** メソッドが呼び出されます。

   c. メッセージがハンドラーチェーンの最後に到達すると、メッセージがディスパッチされます。
      一方向のメッセージ交換の場合は、次のことが起こります。

   d. メッセージ処理が停止します。

e. これまで呼び出されたすべてのメッセージハンドラーには、その **close()** メソッドが呼び出されます。

f. メッセージが送信されます。

3. ProtocolException 例外を出力する -ProtocolException 例外、またはこの例外のサブクラスを出力すると、障害メッセージ処理が開始されることを Apache CXF ランタイムに通知します。ランタイムがどのように進行するかは、**現在のメッセージ** に使用されているメッセージ交換パターンによって異なります。
要求/応答メッセージ交換の場合は、次のことが起こります。

a. ハンドラーがまだ障害メッセージを作成していない場合、ランタイムはメッセージを障害メッセージでラップします。

b. メッセージ処理の方向が逆になります。
たとえば、要求がサービスプロバイダーによって処理されている場合、メッセージはサービスの実装オブジェクトへの進行を停止します。リクエストを発信したコンシューマーに返すために、バインディングに向けて送り返されます。

c. 新しい処理方向でハンドラーチェーンに沿って存在するメッセージハンドラーには、チェーン内で存在する順序で **handleFault()** メソッドが呼び出されます。

d. 障害メッセージがハンドラーチェーンの最後に到達すると、ディスパッチされます。
一方向のメッセージ交換の場合は、次のことが起こります。

e. ハンドラーがまだ障害メッセージを作成していない場合、ランタイムはメッセージを障害メッセージでラップします。

f. メッセージ処理が停止します。

g. これまで呼び出されたすべてのメッセージハンドラーには、その **close()** メソッドが呼び出されます。

h. 障害メッセージが送信されます。

4. その他のランタイム例外を出力する -ProtocolException 例外以外のランタイム例外を出力すると、メッセージ処理が停止することを Apache CXF ランタイムに通知します。これまで呼び出されたすべてのメッセージハンドラーには **close()** メソッドが呼び出され、例外がディスパッチされます。メッセージが要求/応答メッセージ交換の一部である場合、例外がディスパッチされ、要求を発信したコンシューマーに返されます。

例

例43.12「SOAP ハンドラーでのメッセージの処理」に、SOAP メッセージを画面に出力する **handleMessage()** の実装を示します。

例43.12 SOAP ハンドラーでのメッセージの処理

```
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「SOAP ハンドラーでのメッセージの処理」 のコードは、以下を行います。

メッセージコンテキストからアウトバウンドプロパティーを取得します。

メッセージの方向をテストし、適切なメッセージを出力します。

コンテキストから SOAP メッセージを取得します。

メッセージをコンソールに出力します。

43.6. ハンドラーの初期化

概要

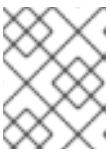
ランタイムがハンドラーのインスタンスを作成すると、ハンドラーがメッセージを処理するために必要なすべてのリソースが作成されます。これを行うためのすべてのロジックをハンドラーのコンストラクターに配置できますが、最適な場所ではない場合があります。ハンドラーフレームワークは、ハンドラーをインスタンス化するときに、いくつかのオプションの手順を実行します。オプションの手順で実行されるリソースインジェクションおよびその他の初期化ロジックを追加できます。

ハンドラーの初期化メソッドを提供する必要はありません。

初期化の順序

Apache CXF ランタイムは、次の方法でハンドラーを初期化します。

1. ハンドラーのコンストラクターが呼び出されます。
2. **@Resource** アノテーションで指定されたリソースが注入される。
3. **@PostConstruct** アノテーションが付けられたメソッドあれば、そのメソッドが呼び出される。



注記

@PostConstruct アノテーションが付けられたメソッドは、**void** の戻り値型を持ち、パラメーターを持ってはいけません。

4. ハンドラーは **Ready** 状態に配置されます。

43.7. 障害メッセージの処理

概要

ハンドラーは、メッセージの処理中に ProtocolException 例外が出力されると、障害メッセージを処理するために **handleFault()** メソッドを使用します。

handleFault() メソッドは、ハンドラーの型に応じて **LogicalMessageContext** オブジェクトまたは **SOAPMessageContext** オブジェクトのいずれかを受け取ります。受信したコンテキストは、ハンドラーの実装にメッセージペイロードへのアクセスを提供します。

障害メッセージ処理の継続方法に応じて、**handleFault()** メソッドは true または false のいずれかを返します。例外を出力することもできます。

メッセージペイロードの取得

handleFault() メソッドで受信されるコンテキストオブジェクトは、**handleMessage()** メソッドによって受信されるものに似ています。コンテキストの **getMessage()** メソッドを使用して、同じ方法でメッセージペイロードにアクセスします。唯一の違いは、コンテキストに含まれるペイロードです。

LogicalMessageContext の操作の詳細は、「論理ハンドラーでのメッセージの処理」を参照してください。

SOAPMessageContext の操作の詳細は、「SOAP ハンドラーでのメッセージの処理」を参照してください。

戻り値の決定

handleFault() メソッドがそのメッセージ処理をどのように完了するかが、メッセージ処理の実施方法に直接的な影響を及ぼします。次のいずれかのアクションを実行することで完了します。

true を返す

障害処理が正常に続行される必要があることを示す真のシグナルを返します。チェーンの次のハンドラーの **handleFault()** メソッドが呼び出されます。

false を返します

障害処理が停止したという false シグナルを返します。現在のメッセージの処理で呼び出されたハンドラーの **close()** メソッドが呼び出され、障害メッセージがディスパッチされます。

例外を出力する

例外を出力すると、障害メッセージの処理が停止します。現在のメッセージの処理で呼び出されたハンドラーの **close()** メソッドが呼び出され、例外がディスパッチされます。

例

例43.13「メッセージハンドラーでの障害の処理」に、メッセージのボディーを画面に出力する **handleFault()** の実装を示します。

例43.13 メッセージハンドラーでの障害の処理

```
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. ハンドラーを閉じる

ハンドラーチェーンがメッセージの処理が終了すると、ランタイムは実行された各ハンドラーの **close()** メソッドを呼び出します。これは、メッセージ処理中にハンドラーによって使用されたリソースをクリーンアップしたり、プロパティーをデフォルト状態にリセットしたりするのに適した場所です。

単一のメッセージ交換を超えてリソースを永続化する必要がある場合は、ハンドラーの **close()** メソッド内でリソースをクリーンアップしないでください。

43.9. ハンドラーのリリース

概要

ハンドラーがバインドされているサービスまたはサービスプロキシーがシャットダウンされると、ランタイムはハンドラーを解放します。ランタイムは、ハンドラーのデストラクタを呼び出す前に、オプションのリリースメソッドを呼び出します。このオプションの解放メソッドを使用して、ハンドラーによって使用されているリソースを解放したり、ハンドラーのデストラクタでは適切ではない他のアクションを実行したりできます。

ハンドラーのクリーンアップメソッドを提供する必要はありません。

リリースの順序

ハンドラーがリリースされると、次のようになります。

1. ハンドラーはアクティブなメッセージの処理を終了します。
2. ランタイムが、**@PreDestroy** アノテーションの付けられたメソッドを呼び出す。
   このメソッドは、ハンドラーによって使用されるすべてのリソースをクリーンアップする必要があります。
3. ハンドラーのデストラクタが呼び出されます。

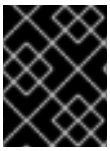
43.10. ハンドラーを使用するためのエンドポイントの設定

43.10.1. プログラムによる設定

43.10.1.1. コンシューマーへのハンドラーチェーンの追加

概要

ハンドラーチェーンをコンシューマーに追加するには、ハンドラーのチェーンを明示的に構築する必要があります。その後、サービスプロキシーの **Binding** オブジェクトにハンドラーチェーンを直接設定します。



重要

Spring の設定を使用して設定されたハンドラーチェーンは、プログラムで設定されたハンドラーチェーンをオーバーライドします。

手順

ハンドラーチェーンをコンシューマーに追加するには、次のようにします。

1. ハンドラーチェーンを保持する **List<Handler>** オブジェクトを作成します。
2. チェーンに追加される各ハンドラーのインスタンスを作成します。
3. インスタンス化された各ハンドラーオブジェクトを、ランタイムによって呼び出される順序でリストに追加します。
4. サービスプロキシーから Binding オブジェクトを取得します。
   Apache CXF は、**org.apache.cxf.jaxws.binding.DefaultBindingImpl** というバインディングインターフェイスの実装を提供します。
5. Binding オブジェクトの **setHandlerChain()** メソッドを使用して、プロキシーにハンドラーチェーンを設定します。

例

例43.14「コンシューマーへのハンドラーチェーンの追加」 は、ハンドラーチェーンをコンシューマーに追加するためのコードを示しています。

例43.14 コンシューマーへのハンドラーチェーンの追加

```
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「コンシューマーへのハンドラーチェーンの追加」 のコードは、以下を行います。

ハンドラーをインスタンス化します。

チェーンを保持する **List** オブジェクトを作成する。

ハンドラーをチェーンに追加します。

Binding オブジェクトを **DefaultBindingImpl** オブジェクトとしてプロキシーから取得する。

ハンドラーチェーンをプロキシーのバインディングに割り当てます。

43.10.1.2. サービスプロバイダーへのハンドラーチェーンの追加

概要

SEI または実装クラスのいずれかに **@HandlerChain** アノテーションを付けて、ハンドラーチェーンをサービスプロバイダーに追加します。アノテーションは、サービスプロバイダーが使用するハンドラーチェーンを定義するメタデータファイルを指します。

手順

ハンドラーチェーンをサービスプロバイダーに追加するには、次のようにします。

1. プロバイダーの実装クラスに **@HandlerChain** アノテーションを付けます。
2. ハンドラーチェーンを定義するハンドラー設定ファイルを作成します。

@HandlerChain アノテーション

javax.jws.HandlerChain アノテーションは、サービスプロバイダーの実装クラスに付けられます。アノテーションは、その **file** プロパティーで指定されたハンドラーチェーンの設定ファイルを読み込むようにランタイムに指示します。

アノテーションの **file** プロパティーは、読み込むハンドラーの設定ファイルを識別する 2 つの方法をサポートします。

- URL
- 相対パス名

例43.15「ハンドラーチェーンをロードするサービスの実装」に、**handlers.xml** と呼ばれるファイルに定義されたハンドラーチェーンを使用するサービスプロバイダーの実装を示します。**handlers.xml** は、サービスプロバイダーが実行されるディレクトリーに存在する必要があります。

例43.15 ハンドラーチェーンをロードするサービスの実装

```
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
{
...
}
```

ハンドラー設定ファイル

ハンドラー設定ファイルは、JSR 109 (Web Services for Java EE、バージョン 1.2) に付属する XML 文法を使用してハンドラーチェーンを定義します。この文法は **http://java.sun.com/xml/ns/javaee** で定義されます。

ハンドラー設定ファイルのルート要素は、**handler-chains** 要素です。**handler-chains** 要素には、1 つまたは複数の **handler-chain** 要素があります。

handler-chain 要素は、ハンドラーチェーンを定義します。表43.1「サーバーサイドハンドラーチェーンの定義に使用される要素」で、**handler-chain** 要素の子を説明します。

表43.1 サーバーサイドハンドラーチェーンの定義に使用される要素

要素	説明
handler	ハンドラーを説明する要素が含まれています。
service-name-pattern	ハンドラーチェーンがバインドされるサービスを定義する WSDL **service** 要素の QName を指定します。QName を定義する際に、* をワイルドカードとして使用できます。
port-name-pattern	ハンドラーチェーンがバインドされるエンドポイントを定義する WSDL **port** 要素の QName を指定します。QName を定義する際に、* をワイルドカードとして使用できます。
protocol-binding	ハンドラーチェーンが使用されるメッセージバインディングを指定します。バインディングは、URI として指定するか、**#\#SOAP11_HTTP**、**\##SOAP11_HTTP_MTOM**、**\##SOAP12_HTTP**、**\##SOAP12_HTTP_MTOM**、または **\#\#XML_HTTP** のエイリアスのいずれかを使用して指定します。 メッセージバインディング URI の詳細については、*23章Apache CXF バインディング ID* を参照してください。

handler-chain 要素に必要なのは、子要素としての **handler** 1 つだけです。ただし、完全なハンドラーチェーンを定義するのに、**handler** 要素を必要な数だけサポートします。チェーン内のハンドラーは、ハンドラーチェーン定義で指定された順序で実行されます。

重要

最終的な実行順序は、指定されたハンドラーを論理ハンドラーとプロトコルハンドラーに分類することによって決定されます。グループ内では、設定で指定された順序が使用されます。

protocol-binding などの他の子は、定義されたハンドラーチェーンのスコープを制限するために使用さ

れます。たとえば、**service-name-pattern** 要素を使用する場合、ハンドラーチェーンは WSDL **port** 要素が指定の WSDL **service** 要素の子であるサービスプロバイダーにのみ割り当てられます。**handler** 要素では、これらの制限用の子の 1 つしか使用できません。

handler 要素は、ハンドラーチェーンの個々のハンドラーを定義します。**handler-class** 子要素は、ハンドラーを実装するクラスの完全修飾名を指定します。**handler** 要素は、ハンドラーに一意の名前を指定するオプションの **handler-name** 要素を持つこともできます。

例43.16「ハンドラー設定ファイル」 は、単一のハンドラーチェーンを定義するハンドラー設定ファイルを示しています。チェーンは 2 つのハンドラーで設定されています。

例43.16 ハンドラー設定ファイル

```
<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 の設定

概要

ハンドラーチェーンを使用するようにエンドポイントを設定する最も簡単な方法は、エンドポイントの設定でチェーンを定義することです。これは、**jaxwxs:handlers** 子要素をエンドポイントを設定する要素に追加して行います。

重要

設定ファイルを介して追加されたハンドラーチェーンは、プログラムで設定されたハンドラーチェーンよりも優先されます。

手順

ハンドラーチェーンをロードするようにエンドポイントを設定するには、次のようにします。

1. エンドポイントに設定要素がまだない場合は、設定要素を追加します。
   Apache CXF エンドポイントの設定の詳細については、17章*JAX-WS エンドポイントの設定* を参照してください。
2. **jaxws:handlers** 子要素をエンドポイントの設定要素に追加します。
3. チェーンの各ハンドラーに対して、ハンドラーを実装するクラスを指定する **bean** 要素を追加します。

ハンドラー実装が複数の場所で使用される場合は、**ref** 要素を使用して **bean** 要素を参照できます。

ハンドラー要素

jaxws:handlers 要素は、エンドポイントの設定でハンドラーチェーンを定義します。これは、すべての JAX-WS エンドポイント設定要素の子として表示できます。以下のとおりです。

- **jaxws:endpoint** はサービスプロバイダーを設定します。
- **jaxws:server** もサービスプロバイダーを設定します。
- **jaxws:client** はサービスコンシューマーを設定します。

次の 2 つの方法のいずれかで、ハンドラーをハンドラーチェーンに追加します。

- 実装クラスを定義する **bean** 要素を追加する
- **ref** 要素を使用して、設定ファイルの他の場所から名前付き **bean** 要素を参照する

設定でハンドラーが定義される順序は、ハンドラーが実行される順序です。論理ハンドラーとプロトコルハンドラーを混在させると、順序が変更される場合があります。ランタイムは、設定で指定された基本的な順序を維持しながら、それらを適切な順序にソートします。

例

例43.17「Spring でハンドラーチェーンを使用するようにエンドポイントを設定する」 は、ハンドラーチェーンをロードするサービスプロバイダーの設定を示しています。

例43.17 Spring でハンドラーチェーンを使用するようにエンドポイントを設定する

```
<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 ツールリファレンス

44.1. プラグインのセットアップ

概要

Apache CXF プラグインを使用する前に、まず適切な依存関係とリポジトリーを POM に追加する必要があります。

依存関係

プロジェクトの POM に次の依存関係を追加する必要があります。

- JAX-WS フロントエンド

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-frontend-jaxws</artifactId>
  <version>version</version>
</dependency>
```

- HTTP トランスポート

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

- Undertow トランスポート

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

44.2. CXF-CODEGEN-PLUGIN

概要

WSDL ドキュメントから JAX-WS 準拠の Java コードを生成します

概要

基本例

以下の POM の抜粋は、**myService.wsdl** WSDL ファイルを処理する Maven **cxf-codegen-plugin** の設定方法の簡単な例を示しています。

```
<plugin>
```

```
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>3.3.6.fuse-7_13_0-00005-redhat-00001</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>target/generated/src/main/java</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>src/main/resources/wsdl/myService.wsdl</wsdl>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

基本の設定

前の例では、次の設定をカスタマイズできます

configuration/sourceRoot

生成された Java ファイルが保存されるディレクトリーを指定します。デフォルトは **target/generated-sources/cxf** です。

configuration/wsdlOptions/wsdlOption/wsdl

WSDL ファイルの場所を指定します。

説明

wsdl2java タスクは WSDL ドキュメントを取得し、サービスを実装する完全なアノテーションが付けられた Java コードを生成します。WSDL ドキュメントには有効な **portType** 要素が必要ですが、**binding** 要素または **service** 要素を含める必要はありません。オプションの引数を使用して、生成されたコードをカスタマイズできます。

WSDL オプション

プラグインを設定するには、1つ以上の **wsdlOptions** 要素が必要です。**wsdlOptions** 要素の **wsdl** 子要素が必要で、プラグインによって処理される WSDL ドキュメントを指定します。**wsdl** 要素に加えて、**wsdlOptions** 要素は WSDL ドキュメントの処理方法をカスタマイズできるさまざまな子要素を取ることができます。

複数の **wsdlOptions** 要素をプラグイン設定に追加できます。各要素は、処理用に単一の WSDL ドキュメントを設定します。

デフォルトのオプション

defaultOptions 要素はオプションの要素です。これは、指定されたすべての WSDL ドキュメントで使用されるオプションを設定するために使用できます。

重要

wsdlOptions 要素でオプションが複製する場合、**wsdlOptions** 要素の値が優先されます。

コード生成オプションの指定

汎用コード生成オプション (Apache CXF **wsdl2java** コマンドラインツールでサポートされるスイッチに対応) を指定するには、**extraargs** 要素を **wsdlOption** 要素の子として追加できます。たとえば、以下のように **-impl** オプションと **-verbose** オプションを追加できます。

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

スイッチが引数を取る場合、後続の **extraarg** 要素を使用してそれらを指定できます。たとえば、**jibx** データバインディングを指定するには、以下のようにプラグインを設定します。

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

バインディングファイルの指定

1 つまたは複数の JAX-WS バインディングファイルの場所を指定するには、**bindingFiles** 要素を **wsdlOption** の子要素として追加します。以下に例を示します。

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

特定の WSDL サービスのコードの生成

コードが生成される WSDL サービスの名前を指定するには、**serviceName** 要素を **wsdlOption** の子要素として追加できます (デフォルトでは WSDL ドキュメントのすべてのサービスのコードが生成されます)。以下に例を示します。

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <serviceName>MyWSDLService</serviceName>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

複数の WSDL ファイルのコード生成

複数の WSDL ファイルのコードを生成するには、WSDL ファイルの追加 **wsdlOption** 要素を挿入するだけです。すべての WSDL ファイルに適用される共通のオプションを指定する場合は、以下のように共通のオプションを **defaultOptions** 要素に配置します。

```
<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>
```

ワイルドカードマッチングを使用して複数の WSDL ファイルを指定することもできます。この場合、**wsdlRoot** 要素を使用して WSDL ファイルが含まれるディレクトリーを指定し、**include** 要素を使用して必要な WSDL ファイルを選択します。その際、* 文字を使用したワイルドカードがサポートされ

ます。たとえば、**src/main/resources/wsdl** ルートディレクトリーから **Service.wsdl** で終わるすべての WSDL ファイルを選択するには、プラグインを以下のように設定します。

```
<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>
```

Maven リポジトリーからの WSDL のダウンロード

Maven リポジトリーから WSDL ファイルを直接ダウンロードするには、**wsdlArtifact** 要素を **wsdlOption** 要素の子要素として追加し、以下のように Maven アーティファクトの変数を指定します。

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdlArtifact>
        <groupId>org.apache.pizza</groupId>
        <artifactId>PizzaService</artifactId>
        <version>1.0.0</version>
      </wsdlArtifact>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

エンコーディング

(JAXB 2.2 が必要です) 生成された Java ファイルに使用される文字エンコーディング (Charset) を指定するには、以下のように **encoding** 要素を **configuration** 要素の子要素として追加します。

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
    </wsdlOption>
  </wsdlOptions>
  <encoding>UTF-8</encoding>
</configuration>
...
```

別のプロセスのフォーク

fork 要素を **configuration** 要素の子として追加することで、コード生成用に別の JVM をフォークするように codegen プラグインを設定できます。fork 要素は、次のいずれかの値に設定できます。

once

単一の新しい JVM をフォークして、codegen プラグインの設定で指定されたすべての WSDL ファイルを処理します。

always

新しい JVM をフォークして、codegen プラグインの設定で指定された各 WSDL ファイルを処理します。

false

(デフォルト) フォークを無効にします。

codegen プラグインが別の JVM をフォークするよう設定される場合 (つまり、**fork** オプションが false 以外の値に設定された場合)、**additionalJvmArgs** 要素を使用してフォークされた JVM に追加の JVM 引数を指定できます。たとえば、以下のフラグメントでは、codegen プラグインを 1 つの JVM をフォークするよう設定します。これは、ローカルファイルシステムからの XML スキーマへのアクセスだけに制限されます (**javax.xml.accessExternalSchema** システムプロパティーを設定して)。

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

オプションの参照

コード生成プロセスの管理に使用するオプションは、以下の表で確認してください。

オプション	解釈
-fe|-frontend frontend	コードジェネレーターが使用するフロントエンドを指定します。設定可能な値は **jaxws**、**jaxws21**、および **cxf** です。JAX-WS 2.1 準拠のコードを生成するために、**jaxws21** フロントエンドが使用されます。**jaxws** フロントエンドの代わりに任意で使用できる **cxf** フロントエンドは、Service クラスの追加のコンストラクターを提供します。このコンストラクターを使用すると、サービスを設定するためのバスインスタンスを簡単に指定できます。デフォルトは **jaxws** です。
-db|-databinding databinding	コードジェネレーターが使用するデータバインディングを指定します。設定可能な値は **jaxb**、**xmlbeans**、**sdo** (**sdo-static** および **sdo-dynamic**)、ならびに **jibx** です。デフォルトは **jaxb** です。

オプション	解釈
-wv wsdlVersion	ツールで必要となる WSDL バージョンを指定します。デフォルトは **1.1** です。[a]
-p wsdlNamespace=PackageName	生成されたコードに使用するパッケージ名を 0 個以上、指定します。オプションで、WSDL 名前空間からパッケージ名へのマッピングを指定します。
-b bindingName	1 つ以上の JAXWS または JAXB バインディングファイルを指定します。バインディングファイルごとに、それぞれ **-b** フラグを使用します。
-sn serviceName	コードが生成される WSDL サービスの名前を指定します。デフォルトでは、WSDL ドキュメント内のすべてのサービスのコードが生成されます。
-reserveClass classname	**-autoNameResolution** と共に使用して、クラス生成時に使用 **しない** wsdl-to-java のクラス名を定義します。クラスが複数ある場合は、このオプションを複数回使用します。
-catalog catalogUrl	インポートされたスキーマと WSDL ドキュメントの解決に使用する XML カタログの URL を指定します。
-d 出力ディレクトリー	生成されたコードファイルを書き込むディレクトリーを指定します。
-compile	生成された Java ファイルをコンパイルします。
-classdir complile-class-dir	コンパイルされたクラスファイルが書き込まれるディレクトリーを指定します。
-clientjar jar-file-name	すべてのクライアントクラスと WSDL を含む JAR ファイルを生成します。このオプションが指定されている場合、**wsdlLocation** の指定は機能しません。
-client	クライアントメインラインの開始点コードを生成します。
-server	サーバーメインラインの開始点コードを生成します。
-impl	実装オブジェクトの開始点コードを生成します。

オプション	解釈
-all	すべての開始点コード (型、サービスプロキシー、サービスインターフェイス、サーバーメインライン、クライアントメインライン、実装オブジェクト、および Ant **build.xml** ファイル) を生成します。
-ant	Ant **build.xml** ファイルを生成します。
-autoNameResolution	バインディングのカスタマイズを使用せずに、名前の競合を自動的に解決します。
-defaultValues=DefaultValueProvider	生成されたクライアントと実装のデフォルト値を入力するようにツールに指示します。オプションで、デフォルト値の生成に使用されるクラス名を指定することもできます。デフォルトでは、**RandomValueProvider** クラスが使用されます。
-nexclude schema-namespace=java-packagename	コードの生成時に、指定された WSDL スキーマの名前空間を無視します。このオプションは複数回指定できます。また、オプションで、除外された名前空間で記述されたタイプで使用される Java パッケージ名を指定します。
-exsh (**true**/**false**)	拡張 soap ヘッダーメッセージバインディングの処理を有効または無効にします。デフォルトは false です。
-noTypes	タイプの生成をオフにします。
-dns (true/false)	デフォルトの名前空間パッケージ名マッピングのロードを有効または無効にします。デフォルトは true です。
-dex (true/false)	デフォルトの除外名前空間マッピングのロードを有効または無効にします。デフォルトは true です。
-xjc引数	JAXB データバインディングが使用されているときに XJC に直接渡される引数のコンマ区切りリストを指定します。設定可能なすべての XJC 引数のリストを取得するには、**-xjc-X** を使用します。
-noAddressBinding	JAX-WS 2.1 準拠のマッピングの代わりに、Apache CXF 独自の WS-Addressing タイプを使用するようにツールに指示します。
-validate [=all|basic|none]	コードを生成する前に、WSDL ドキュメントを検証するようにツールに指示します。

オプション	解釈
-keep	既存のファイルを上書きしないようにツールに指示します。
-wsdlLocation wsdlLocation	**@WebService** アノテーションの **wsdlLocation** プロパティーの値を指定します。
-v	ツールのバージョン番号を表示します。
-verbose|-V	コード生成プロセス中にコメントを表示します。
-quiet	コード生成プロセス中のコメントを非表示にします。
-allowElementReferences[=true], -aer[=true]	**true** の場合、JAX-WS 2.2 仕様のセクション 2.3.1.2(v) で指定されるルールを無視し、ラッパースタイルのマッピングの使用時に要素参照を許可しません。デフォルトは **false** です。
-asyncMethods[=method1,method2,...]	クライアント側の非同期呼び出しを許可する、その後生成される Java クラスメソッドのリスト。JAX-WS バインディングファイルの **enableAsyncMapping** と同様です。
-bareMethods[=method1,method2,...]	ラッパースタイル (下記を参照) を持つための、その後生成される Java クラスメソッドのリスト。JAX-WS バインディングファイルの **enableWrapperStyle** と同様です。
-mimeMethods[=method1,method2,...]	mime:content マッピングを有効にするための、その後生成される Java クラスメソッドのリスト。JAX-WS バインディングファイルの **enableMIMEContent** と同様です。
-faultSerialVersionUID fault-serialVersionUID	障害例外の suid を生成する方法。設定可能な値は、**NONE**、**TIMESTAMP**、**FQCN**、または特定の数値です。デフォルトは **NONE** です。
-encoding encoding	Java コードを生成するときに使用する Charset エンコーディングを指定します。
-exceptionSuper	**wsdl:fault** 要素から生成された障害 Bean のスーパークラス (デフォルトは **java.lang.Exception**)。
-seiSuper interfaceName	生成された SEI インターフェイスのベースインターフェイスを指定します。たとえば、このオプションを使用して、Java 7 **AutoCloseable** インターフェイスをスーパーインターフェイスとして追加できます。

オプション	解釈
-mark-generated	生成されたクラスに @Generated アノテーションを追加します。

[a] 現在、Apache CXF は、コードジェネレーターの WSDL1.1 のみをサポートします。

44.3. JAVA2WS

概要

Java コードから WSDL ドキュメントを生成します。

概要

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>version</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <option>...</option>
          ...
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

説明

java2ws タスクはサービスエンドポイント実装 (SEI) を受け取り、Web サービスの実装に使用されるサポートファイルを生成します。次を生成できます。

- WSDL ドキュメント
- サービスを POJO としてデプロイするために必要なサーバーコード
- サービスにアクセスするためのクライアントコード
- ラッパーとフォールト Bean

必須設定

プラグインに **className** 設定要素が存在する必要があります。要素の値は、処理される SEI の完全修飾名です。

任意の設定

次の表にリストされている設定要素を使用して、WSDL 生成を微調整できます。

要素	説明
frontend	SEI の処理とサポートクラスの生成に使用するフロントエンドを指定します。**jaxws** がデフォルトです。**simple** もサポートされています。
databinding	SEI の処理とサポートクラスの生成に使用されるデータバインディングを指定します。JAX-WS フロントエンドを使用する場合のデフォルトは **jaxb** です。単純なフロントエンドを使用する場合のデフォルトは **aegis** です。
genWsdl	**true** に設定した場合、WSDL ドキュメントを生成するようツールに指示します。
genWrapperbean	**true** に設定した場合、ラッパー Bean および障害 Bean を生成するようツールに指示します。
genClient	**true** に設定した場合、クライアントコードを生成するようツールに指示します。
genServer	**true** に設定した場合、サーバーコードを生成するようツールに指示します。
outputFile	生成される WSDL ファイルの名前を指定します。
classpath	SEI の処理時に検索されるクラスパスを指定します。
soap12	**true** に設定した場合、生成される WSDL ドキュメントに SOAP 1.2 バインディングが含まれることを指定します。
targetNamespace	生成された WSDL ファイルで使用するターゲット名前空間を指定します。
serviceName	生成された **service** 要素の **name** 属性の値を指定します。

パート VI. RESTFUL WEB サービスの開発

このガイドでは、JAX-RSAPI を使用して Web サービスを実装する方法について説明します。

第45章 RESTFUL WEB サービスの概要

概要

REST (Representational State Transfer) は、4 つの基本的な HTTP 動詞のみを使用して、HTTP を介したデータの送信を中心としたソフトウェアアーキテクチャースタイルです。また、SOAP エンベロープなどの追加のラッパーや状態データの使用を避けます。

概要

Representational State Transfer (REST) は、Roy Fielding という研究者による博士論文で最初に説明されたアーキテクチャースタイルです。RESTful システムでは、サーバーは URI を使用してリソースを公開し、クライアントは 4 つの HTTP 動詞を使用してこれらのリソースにアクセスします。クライアントがリソースの表現を受け取ると、クライアントはある状態に配置されます。通常はリンクをたどって新しいリソースにアクセスすると、状態が変化または遷移します。REST は、リソースが普及している標準文法を使用して表現できることを前提として機能させます。

World Wide Web は、REST の原則をもとに設計された、最も普及しているシステムの例です。Web ブラウザーは、Web サーバーでホストされているリソースにアクセスするクライアントとして機能します。リソースは、すべての Web ブラウザーが使用できる HTML または XML 文法を使用して表されます。ブラウザーは、新しいリソースへのリンクを簡単にたどることもできます。

RESTful システムの利点は、拡張性と柔軟性が高いことです。リソースは 4 つの HTTP 動詞を使用してアクセスおよび操作されるため、リソースは URI で公開され、標準の文法で表現されるので、サーバーに変更が加えられてもクライアントへの影響はありません。また、RESTful システムは、キャッシングやプロキシーなどの HTTP のスケーラビリティー機能を最大限に活用できます。

基本的な REST の原則

RESTful アーキテクチャーは、次の基本原則に準拠しています。

- アプリケーションの状態と機能は複数のリソースに分けられます。
- リソースは、ハイパーメディアリンクとして使用できる標準 URI でアドレス指定できます。
- すべてのリソースは、4 つの HTTP 動詞のみを使用します。
  - **DELETE**
  - **GET**
  - **POST**
  - **PUT**
- すべてのリソースは、HTTP でサポートされている MIME タイプを使用して情報を提供します。
- プロトコルはステートレスです。
- 応答はキャッシュ可能です。
- プロトコルは階層化されています。

RESOURCES

リソース は REST の中心です。リソースは、URI を使用してアドレス指定できる情報のソースです。Web 登場した頃は、リソースは主に静的なドキュメントでした。最近の Web では、リソースは任意の情報源はさまざまです。たとえば、URI を使用してアクセスできる場合には、Web サービスをリソースにすることができます。

RESTful エンドポイントは、アドレス指定するリソースの **表現** を交換します。表現は、リソースが提供するデータを含むドキュメントです。たとえば、顧客レコードにアクセスできるようにする Web サービスのメソッドはリソースで、サービスとコンシューマーの間で交換される顧客レコードのコピーはリソースの表現です。

REST のベストプラクティス

RESTful Web サービスを設計するときは、次の点に注意してください。

- 公開するリソースごとに個別の URI を指定します。
  たとえば、運転記録を処理するシステムを構築している場合には、各記録には一意の URI が必要です。システムが駐車違反やスピード違反の罰金に関する情報も提供する場合は、各タイプのリソースにも固有のベースが必要です。たとえば、スピード違反の罰金は **/speedingfines/driverID** を介してアクセスでき、駐車違反は **/parkingfines/driverID** を介してアクセスできます。

- URI で名詞を使用します。
  名刺を使用すると、リソースがモノで、アクションではないことが強調されます。**/ordering** などの URI はアクションを、**/orders** はモノを意味します。

- **GET** にマップするメソッドはデータを変更しないでください。

- 応答にはリンクを使用してください。
  応答に他のリソースへのリンクを含めると、クライアントが一連のデータを簡単にたどることができます。たとえば、サービスがリソースのコレクションを返す場合には、クライアントは提供されたリンクを使用して各リソースにアクセスするのが簡単になります。リンクが含まれていない場合には、クライアントは特定のノードへのチェーンをたどるためにロジックが追加で必要です。

- サービスをステートレスにします。
  クライアントまたはサービスで状態情報を維持させると、この 2 つが強制的に疎結合されます。疎結合であることが原因で、アップグレードと移行がより困難になります。また、状態を維持すると、通信エラーから回復するのがより困難になる可能性もあります。

RESTFUL WEB サービスの設計

RESTful Web サービスの実装に使用するフレームワークに関係なく、従う必要のあるステップがいくつかあります。

1. サービスが公開するリソースを定義します。
   一般的に、サービスは、ツリー編成されたリソースを 1 つ以上公開します。たとえば、運転記録サービスは次の 3 つのリソースに編成できます。

   - **/license/driverID**
   - **/license/driverID/speedingfines**
   - **/license/driverID/parkingfines**

2. 各リソースで実行できるようにするアクションを定義します。
   たとえば、ダイバーの住所を更新したり、運転免許証から駐車違反切符を削除したりできるようにする場合などです。

3. アクションを適切な HTTP 動詞にマップします。

サービスを定義したら、Apache CXF を使用してサービスを実装できます。

APACHE CXF を使用した REST の実装

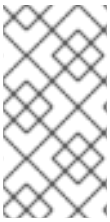
Apache CXF は、**RESTFul Web サービス** (JAX-RS) 用の JavaAPI の実装を提供します。JAX-RS は、アノテーションを使用して POJO をリソースにマップするための標準化された方法を提供します。

抽象サービス定義から JAX-RS を使用して実装された RESTful Web サービスに移行する場合は、次のことを行う必要があります。

1. サービスのリソースツリーの最上位を表すリソースのルートリソースクラスを作成します。
   「ルートリソースクラス」を参照してください。

2. サービスの他のリソースをサブリソースにマップします。
   「サブリソースの操作」を参照してください。

3. 各リソースで使用される各 HTTP 動詞を実装するメソッドを作成します。
   「リソースメソッドの操作」を参照してください。



注記

Apache CXF は、Java インターフェイスを RESTful Web サービスにマップする、以前の HTTP バインディングを引き続きサポートします。HTTP バインディングは基本的な機能を提供し、いくつかの制限があります。開発者は、JAX-RS を使用するようにアプリケーションを更新することを推奨します。

データバインディング

デフォルトでは、Apache CXF は Java Architecture for XML Binding (JAXB) オブジェクトを使用して、リソースとその表現を Java オブジェクトにマップします。Java オブジェクトと XML 要素の間で、正確かつ明確に定義されたマッピングを提供します。

Apache CXF JAX-RS 実装は、**JavaScript Object Notation** (JSON) を使用したデータ交換もサポートしています。JSON は、Ajax 開発者が使用する一般的なデータ形式です。JSON と JAXB の間のデータのマーシャリングは、Apache CXF ランタイムによって処理されます。

第46章 リソースの作成

概要

RESTful Web サービスでは、すべての要求はリソースによって処理されます。JAX-RS API は、リソースを Java クラスとして実装します。リソースクラスは、Java クラスで、1 つ以上の JAX-RS アノテーションが付けられます。JAX-RS を使用して実装された RESTful Web サービスのコアは、ルートリソースクラスです。ルートリソースクラスは、サービスによって公開されるリソースツリーへのエントリーポイントです。すべての要求自体を処理する場合もあれば、要求を処理するサブリソースにアクセスできるようにする場合もあります。

46.1. 概要

概要

JAX-RS API を使用して実装された RESTful Web サービスは、Java クラスによって実装されたリソースの表現として応答します。**リソースクラス** は、JAX-RS アノテーションを使用してリソースを実装するクラスです。ほとんどの RESTful Web サービスには、アクセスする必要のあるリソースのコレクションがあります。リソースクラスのアノテーションは、リソースの URI や各操作が処理する HTTP 動詞などの情報を提供します。

リソースの種類

JAX-RS API を使用すると、基本的なリソースタイプを 2 つ作成できます。

- 「ルートリソースクラス」 サービスのリソースツリーへのエントリーポイントです。**@Path** アノテーションが付けられ、サービスのリソースのベース URI を定義します。
- 「サブリソースの操作」 には、ルートリソースを使用してアクセスします。これらは、**@Path** アノテーションが付けられたメソッドによって実装されます。サブリソースの **@Path** アノテーションは、ルートリソースのベース URI に対する相対的な URI を定義します。

例

例46.1「単純なリソースクラス」 は単純なリソースクラスを示しています。

例46.1 単純なリソースクラス

```
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)
```

```
  {
    ...
  }

  ...
}
```

2 つの項目で、例46.1「単純なリソースクラス」 リソースクラスにクラスが定義されます。

@Path アノテーションは、リソースのベース URI を指定します。

@GET アノテーションは、メソッドがリソースの HTTP **GET** メソッドを実装することを指定します。

46.2. 基本的な JAX-RS アノテーション

概要

RESTful Web サービスの実装に必要な、最も基本的な情報は次のとおりです。

- サービスのリソースの URI
- クラスのメソッドが HTTP 動詞にどのようにマッピングされるか

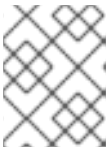
JAX-RS は、この基本情報を提供する一連のアノテーションを定義します。リソースクラスにはすべて、これらのアノテーションの内、少なくとも 1 つが必要です。

パスの設定

@Path アノテーションは、リソースの URI を指定します。アノテーションは javax.ws.rs.Path インターフェイスによって定義され、リソースクラスまたはリソースメソッドのいずれかを修飾するために使用できます。これには、唯一のパラメーターとして文字列値を使用できます。文字列値は、実装されたリソースの場所を指定する URI テンプレートです。

URI テンプレートは、リソースの相対的な場所を指定します。例46.2「URI テンプレート構文」 に示すように、テンプレートには次のものを含めることができます。

- 未処理のパスコンポーネント
- **{ }** で囲まれたパラメーター識別子



注記

パラメーター識別子には、デフォルトのパス処理を変更するための正規表現を含めることができます。

例46.2 URI テンプレート構文

```
@Path("resourceName/{param1}/../{paramN}")
```

たとえば、URI テンプレート **widgets/{color}/{number}** は **widgets/blue/12** にマップされます。**color** パラメーターの値は **blue** に割り当てられます。**number** パラメーターの値には 12 が割り当てられます。

URI テンプレートが完全な URI にマップされる方法は、**@Path** アノテーションが付けられたものによって異なります。ルートリソースクラスに配置されている場合には、URI テンプレートはツリー内のすべてのリソースのルート URI であり、サービスが公開されている URI に直接追加されます。サブリソースにアノテーションが付けられる場合は、ルートリソース URI と相対的なパスを使用します。

HTTP 動詞の指定

JAX-RS は、メソッドに使用される HTTP 動詞を指定するために 5 つのアノテーションを使用します。

- javax.ws.rs.DELETE は、メソッドが **DELETE** にマップすることを指定します。
- javax.ws.rs.GET は、メソッドが **GET** にマップすることを指定します。
- javax.ws.rs.POST は、メソッドが **POST** にマップすることを指定します。
- javax.ws.rs.PUT は、メソッドが **PUT** にマップすることを指定します。
- javax.ws.rs.HEAD は、メソッドが **HEAD** にマップすることを指定します。

メソッドを HTTP 動詞にマップする場合は、マッピングに意味があることを確認する必要があります。たとえば、発注書を送信する目的のメソッドをマップする場合、**PUT** または **POST** にマッピングします。**GET** または **DELETE** にマッピングすると、予期しない動作が発生します。

46.3. ルートリソースクラス

概要

ルートリソースクラスは、JAX-RS で実装された RESTful Web サービスへのエントリーポイントです。サービスによって実装されるリソースのルート URI を指定する **@Path** が付いています。そのメソッドは、リソースに対する操作を直接実装するか、サブリソースへのアクセスを提供します。

要件

クラスをルートリソースクラスに指定するには、次の基準を満たしている必要があります。

- クラスには **@Path** アノテーションを付ける必要があります。
  指定されたパスは、サービスで実装されるすべてのリソースのルート URI です。ルートリソースクラスで、パスが **widgets** で、そのメソッドの 1 つが **GET** 動詞を実装する場合は、**widgets** の **GET** がメソッドを呼び出します。サブリソースで、その URI が **{id}** であると指定されている場合には、サブリソースの完全な URI テンプレートは **widgets/{id}** で、**widgets/12** や **widgets/42** などの URI に対して行われた要求を処理します。
- ランタイムによる呼び出すには、クラスにパブリックコンストラクターが必要です。
  ランタイムは、コンストラクターのすべてのパラメーターの値を提供できる必要があります。コンストラクターのパラメーターには、JAX-RS パラメーターアノテーションで装飾されたパラメーターを含めることができます。パラメーターアノテーションの詳細については、47章 *リソースクラスとメソッドへの情報の受け渡し* を参照してください。
- クラスのメソッドの少なくとも 1 つに HTTP 動詞アノテーションまたは **@Path** アノテーションを付ける必要があります。

例

例46.3「ルートリソースクラス」 は、サブリソースへのアクセスを提供するルートリソースクラスを示しています。

例46.3 ルートリソースクラス

```
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「ルートリソースクラス」 のクラスは、ルートリソースクラスのすべての要件を満たしています。

クラスに **@Path** アノテーションが付けられます。サービスによって公開されるリソースのルート URI は **customerservice** です。

クラスにはパブリックコンストラクターがあります。この場合には、簡素化するために、引数なしのコンストラクターが使用されます。

このクラスは、リソースに 4 つの HTTP 動詞それぞれを実装します。

このクラスは、**getOrder()** メソッドを介してサブリソースへのアクセスも提供します。**@Path** アノテーションを使用して指定されるサブリソースの URI は **customerservice/order/id** です。サブリソースは **Order** クラスによって実装されます。

サブリソースの実装の詳細については、「サブリソースの操作」を参照してください。

46.4. リソースメソッドの操作

概要

リソースメソッドは、JAX-RS アノテーションを使用してアノテーションを付けます。それらには、メソッドが処理する要求のタイプを指定する HTTP メソッドアノテーションの 1 つが含まれます。JAX-RS は、リソースメソッドにいくつかの制約を課します。

一般的な制約

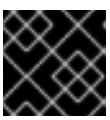
すべてのリソースメソッドは、次の条件を満たす必要があります。

- 公開されている。
- 「HTTP 動詞の指定」 で記載されている HTTP メソッドアノテーションの 1 つが付けられている。
- 「パラメーター」 で記載のエンティティーパラメーターを 1 つ以上指定しない。

パラメーター

リソースメソッドのパラメーターには、次の 2 つの形式があります。

- **エンティティーパラメーター**: エンティティーパラメーターにはアノテーションが付けられていません。このパラメーターの値は、要求エンティティーの本体からマッピングされます。エンティティーパラメーターは、アプリケーションにエンティティープロバイダーがあるタイプであればどれにでも指定できます。通常、これらは JAXB オブジェクトです。



重要

リソースメソッドは、エンティティーパラメーターを **1 つだけ** 指定できます。

エンティティープロバイダーの詳細は、51章 *エンティティーサポート* を参照してください。

- **アノテーション付きパラメーター**: アノテーション付きパラメーターは、JAX-RS アノテーションの 1 つを使用し、パラメーターの値が要求からどのようにマップされるかを指定します。通常、パラメーターの値は、要求 URI の一部からマップされます。
  要求データをメソッドパラメーターにマッピングするための JAX-RS アノテーションの使用の詳細は、47章 *リソースクラスとメソッドへの情報の受け渡し* を参照してください。

例46.4「有効なパラメーターリストを持つリソースメソッド」 は、パラメーターリストが有効なリソースメソッドを示しています。

例46.4 有効なパラメーターリストを持つリソースメソッド

```
@POST
@Path("disaster/monster/giant/{id}")
public void addDaikaiju(Kaiju kaiju,
                        @PathParam("id") String id)
{
  ...
}
```

例46.5「パラメーターリストが無効なリソースメソッド」 は、パラメーターリストが無効なリソースメソッドを示しています。アノテーションが付けられていないパラメーターが 2 つあります。

例46.5 パラメーターリストが無効なリソースメソッド

```
@POST
@Path("disaster/monster/giant/")
public void addDaikaiju(Kaiju kaiju,
                        String id)
{
  ...
}
```

戻り値

リソースメソッドは、次のいずれかを返すことができます。

- **void**
- アプリケーションにエンティティープロバイダーがある Java クラス
  エンティティープロバイダーの詳細は、51章 *エンティティーサポート* を参照してください。
- **Response** オブジェクト
  Response オブジェクトの詳細は、「アプリケーションの応答の微調整」を参照してください。
- **GenericEntity<T>** オブジェクト
  GenericEntity<T> オブジェクトの詳細は、「汎用型情報を含むエンティティーの返答」を参照してください。

すべてのリソースメソッドは、HTTP ステータスコードをリクエスターに返します。メソッドの戻り値

が **void** である場合、または返された値が **null** である場合、リソースメソッドは HTTP ステータスコードを **204** に設定します。リソースメソッドが **null** 以外の値を返すと、HTTP ステータスコードが **200** に設定されます。

46.5. サブリソースの操作

概要

サービスは複数のリソースで処理する必要がある場合が多いです。たとえば、注文処理サービスのベストプラクティスでは、各顧客を一意のリソースとして処理することを提案しています。各注文は、一意のリソースとしても処理されます。

JAX-RS API を使用して、顧客リソースと注文リソースを **サブ** リソースとして実装します。サブリソースは、ルートリソースクラスを介してアクセスされるリソースです。これらは、**@Path** アノテーションをリソースクラスのメソッドに追加して定義されます。サブリソースは、次の 2 つの方法のいずれかで実装できます。

- サブリソースメソッド: サブリソースの HTTP 動詞を直接実装し、「HTTP 動詞の指定」 で説明されているアノテーションの 1 つを付けます。
- サブリソースロケーター: サブリソースを実装するクラスを指します。

サブリソースの指定

サブリソースを指定するには、**@Path** アノテーションをメソッドに付けます。サブリソースの URI は次のように設定されます。

1. サブリソースの **@Path** アノテーションの値を、サブリソースの親リソースの **@Path** アノテーションの値に追加します。
   親リソースの **@Path** アノテーションは、サブリソースを含むクラスのオブジェクトを返すリソースクラスのメソッドにある場合があります。
2. ルートリソースに到達するまで、前の手順を繰り返します。
3. 組み立てられ URI は、サービスがデプロイされるベース URI に追加されます。

たとえば、例46.6「サブリソースの注文」 に示されているサブリソースの URI は、**baseURI/customerservice/order/12** である可能性があります。

例46.6 サブリソースの注文

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }
}
```

サブリソースメソッド

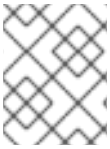
サブリソースメソッドには、**@Path** アノテーションと HTTP 動詞アノテーションのいずれが付けられます。れます。サブリソースメソッドは、指定された HTTP 動詞を使用してリソースに対して行われた要求の処理を直接担当します。

例46.7「サブリソースメソッド」 は、3 つのサブリソースメソッドを持つリソースクラスを示しています。

- **getOrder()** は、URI が **/customerservice/orders/{orderId}/** に一致するリソースの HTTP **GET** リクエストを処理します。
- **updateOrder()** は、URI が **/customerservice/orders/{orderId}/** に一致するリソースの HTTP **PUT** リクエストを処理します。
- **newOrder()** は、**/customerservice/orders/** でリソースの HTTP **POST** 要求を処理します。

例46.7 サブリソースメソッド

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



注記

URI テンプレートが同じサブリソースメソッドは、サブリソースロケーターによって返されるリソースクラスと同じです。

sub-resource ロケーター

サブリソースロケーターは、HTTP 動詞アノテーションの 1 つが追加されておらず、サブリソースでの要求は直接処理されません。代わりに、サブリソースロケーターは、要求を処理できるリソースクラスのインスタンスを返します。

HTTP 動詞アノテーションがないことに加えて、サブリソースロケーターはエンティティーパラメーター指定できません。サブリソースロケーターメソッドで使用されるすべてのパラメーターは、47章 *リソースクラスとメソッドへの情報の受け渡し* で説明されているアノテーションの 1 つを使用する必要があります。

例46.8「特定のクラスを返すサブリソースロケーター」 に示すように、サブリソースロケーターを使用すると、すべてのメソッドを 1 つのスーパークラスに配置する代わりに、リソースを再利用可能なクラスとしてカプセル化できます。**processOrder()** メソッドはサブリソースロケーターです。URI テンプレート **/orders/{orderId}/** に一致する URI でリクエストが実行されると、**Order** クラスのインスタンスが返されます。**Order** クラスには、HTTP 動詞アノテーションが付いたメソッドがあります。**PUT** リクエストは **updateOrder()** メソッドによって処理されます。

例46.8 特定のクラスを返すサブリソースロケーター

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

}
```

ポリモーフィズムをサポートできるように、サブリソースロケーターは実行時に処理されます。サブリソースロケーターの戻り値は、汎用 **Object**、抽象クラス、またはクラス階層の最上位になります。たとえば、サービスで PayPal の注文とクレジットカードの注文の両方を処理する必要がある場合、例46.8「特定のクラスを返すサブリソースロケーター」からの **processOrder()** メソッドの署名は変更さ

れません。**Order** クラスを拡張した **ppOrder** と **ccOder** の 2 つのクラスを実装することのみが必要になります。**processOrder()** の実装は、必要なロジックを基づいて、サブリソースの必要な実装をインスタンス化します。

46.6. リソースの選択方法

概要

特定の URI を 1 つ以上のリソースメソッドにマップすることが可能です。たとえば、URI **customerservice/12/ma** は **@Path("customerservice/{id}")** または **@Path("customerservice/{id}/{state}")** テンプレートと一致する可能性があります。JAX-RS は、要求とリソースメソッドと照合するための照合アルゴリズムを指定します。アルゴリズムは、正規化された URI、HTTP 動詞、および要求エンティティーと応答エンティティーのメディアタイプを、リソースクラスのアノテーションと比較します。

基本的な選択アルゴリズム

JAX-RS 選択アルゴリズムは、次の 3 つの段階に分けられます。

1. ルートリソースクラスを決定します。
   リクエスト URI は **@Path** アノテーションが付けられたすべてのクラスに対して照合されます。**@Path** アノテーションがリクエスト URI と一致するクラスが判断されます。

   リソースクラスの **@Path** アノテーションの値が要求 URI 全体と一致する場合、クラスのメソッドは 3 番目のステージへの入力として使用されます。

2. オブジェクトがリクエストを処理するかどうかを決定します。
   リクエスト URI が選択したクラスの **@Path** アノテーションの値より長い場合、リソースメソッドの **@Path** アノテーションの値は、要求を処理できるサブリソースの検索に使用されます。

   1 つ以上のサブリソースメソッドが要求 URI と一致する場合には、これらのメソッドは第 3 ステージの入力として使用されます。

   要求 URI に一致するのがサブリソースロケーターのみである場合には、サブリソースロケーターによって作成されたオブジェクトのリソースメソッドは、要求 URI と一致します。この段階は、サブリソースメソッドが要求 URI と一致するまで繰り返されます。

3. 要求を処理するリソースメソッドを選択します。
   HTTP 動詞アノテーションが要求の HTTP 動詞と一致するリソースメソッド。さらに、選択されたリソースメソッドは、要求エンティティー本体のメディアタイプを受け入れ、要求で指定されたメディアタイプに準拠する応答を生成できる必要があります。

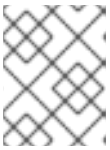
複数のリソースクラスからの選択

選択アルゴリズムの最初の 2 つの段階では、要求を処理するリソースを決定します。場合によっては、リソースはリソースクラスで実装されます。それ以外の場合は、同じ URI テンプレートを使用する 1 つ以上のサブリソースで実装されます。要求 URI に一致するリソースが複数ある場合には、サブリソースよりもリソースクラスが優先されます。

リソースクラスとサブリソースを並べ替えた後も複数のリソースが要求 URI に一致する場合は、次の基準を使用して単一のリソースを選択します。

1. URI テンプレートで最もリテラル文字が多いリソースを優先します。
   リテラル文字は、テンプレート変数の一部ではない文字です。たとえ

ば、**/widgets/{id}/{color}** には 10 個のリテラル文字があり、**/widgets/1/{color}** には 11 個のリテラル文字があります。したがって、リクエスト URI **/widgets/1/red** は、URI テンプレートとして **/widgets/1/{color}** を使用してリソースと照合されます。



注記

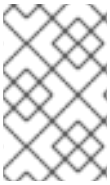
末尾のスラッシュ (/) はリテラル文字としてカウントされます。したがって、**/joefred/** は **/joefred** よりも優先されます。

2. URI テンプレートで変数が最も多いリソースを優先します。
要求 URI **/widgets/30/green** は、**/widgets/{id}/{color}** と **/widgets/{amount}/** の両方と一致する可能性があります。ただし、変数が 2 つあるため、URI テンプレート **/widgets/{id}/{color}** のリソースが選択されます。

3. 変数が最も多く、正規表現が含まれるリソースを優先します。
要求 URI **/widgets/30/green** は、**/widgets/{number}/{color}** と **/widgets/{id:.}/{color}* の両方と一致する可能性があります。ただし、正規表現を含む変数があるため、URI テンプレート */widgets/{id:.}/{color}** を含むリソースが選択されます。

複数のリソースメソッドからの選択

多くの場合に、要求 URI に一致するリソースを選択すると、要求を処理できるリソースメソッドが 1 つになります。メソッドは、要求で指定された HTTP 動詞をリソースメソッドの HTTP 動詞アノテーションと照合することによって決定されます。適切な HTTP 動詞アノテーションが存在することに加えて、選択されたメソッドは、要求内の要求エンティティーを処理でき、要求のメタデータで指定された適切なタイプの応答を生成できなければなりません。



注記

リソースメソッドが処理できるリクエストエンティティーの型は **@Consumes** アノテーションによって指定されます。リソースメソッドが生成できる応答の型は **@Produces** アノテーションを使用して指定されます。

リソースを選択すると、要求を処理できる複数のメソッドが生成されます。次の基準を使用して、要求を処理するリソースメソッドを選択します。

1. サブリソースよりもリソースメソッドを優先します。

2. サブリソースロケーターよりもサブリソースメソッドを優先します。

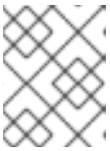
3. **@Consumes** アノテーションと **@Produces** アノテーションで最も具体的な値を使用するメソッドを優先します。
たとえば、アノテーション **@Consumes("text/xml")** を持つメソッドは、アノテーション **@Consumes("text/*")** を持つメソッドよりも優先されます。どちらのメソッドも、**@Consumes** アノテーションのないメソッドやアノテーション **@Consumes("*/*")** よりも優先されます。

4. リクエスト本文エンティティーのコンテンツタイプに最も近いメソッドを優先します。

注記

リクエストボディーエンティティーのコンテンツ型は HTTP **Content-Type** プロパティーで指定されます。

5. 応答として受け入れられるコンテンツタイプに最も近い方法を優先します。



注記

レスポンスとして受け入れられるコンテンツタイプは、HTTP **Accept** プロパティーで指定されます。

選択プロセスのカスタマイズ

開発者からの報告によると、このアルゴリズムでは、複数のリソースクラスを選択する方法に何らかの制限がある場合があります。たとえば、特定のリソースクラスが一致し、このクラスに一致するリソースメソッドがない場合に、このアルゴリズムは実行を停止します。一致していても残りのリソースクラスはチェックされません。

Apache CXF には、org.apache.cxf.jaxrs.ext.ResourceComparator インターフェイスが含まれており、ランタイムが複数の一致するリソースクラスの処理方法のカスタマイズに使用できます。例46.9「リソース選択をカスタマイズするためのインターフェイス」 に示される ResourceComparator インターフェイスには、実装する必要のあるメソッドが 2 つあります。1 つは 2 つのリソースクラスを、もう 1 つは 2 つのリソースメソッドを比較します。

例46.9 リソース選択をカスタマイズするためのインターフェイス

```
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);
}
```

カスタム実装は、次のように 2 つのリソースから選択します。

- 最初のパラメーターが 2 番目のパラメーターよりも一致している場合は、**1** を返します。
- 2 番目のパラメーターが最初のパラメーターよりも一致している場合は、**-1** を返します。

0 が返されると、ランタイムはデフォルトの選択アルゴリズムで続行されます。

カスタム ResourceComparator 実装を登録するには、子の **resourceComparator** をサービスの **jaxrs:server** 要素に追加します。

第47章 リソースクラスとメソッドへの情報の受け渡し

概要

JAX-RS は、開発者がリソースに渡される情報の出所を制御できるようにするアノテーションを複数指定します。アノテーションは、URI のマトリックスパラメーターなど、一般的な HTTP の概念に準拠しています。標準 API を使用すると、メソッドパラメーター、Bean プロパティー、およびリソースクラスフィールドでアノテーションを使用できます。Apache CXF には、一連のパラメーターを Bean に注入できるようにする拡張機能が含まれています。

47.1. データ注入の基本

概要

HTTP リクエストメッセージからのデータを使用して初期化されるパラメーター、フィールド、および Bean プロパティーには、ランタイムでそれらの値が注入されます。注入される特定のデータは、「JAX-RS API の使用」 で説明されている一連のアノテーションによって指定されます。

JAX-RS 仕様では、データが挿入されるタイミングにいくつかの制限があります。また、リクエストデータを挿入できるオブジェクトのタイプにも制限があります。

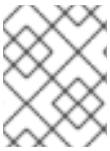
データが注入されるタイミング

要求データは、要求によりインスタンス化された時にオブジェクトに挿入されます。つまり、リソースに直接対応するオブジェクトのみがインジェクションアノテーションを使用できます。46章 *リソースの作成* で説明されているように、これらのオブジェクトは、**@Path** アノテーションが付けられたルートリソースまたはサブリソースロケーターメソッドから返されたオブジェクトのいずれかになります。

サポートされるデータタイプ

データを挿入できる特定のデータタイプセットは、挿入されたデータのソースを指定するために使用されるアノテーションによって異なります。ただし、すべてのインジェクションアノテーションは、少なくとも以下のデータタイプのセットをサポートします。

- **int**、**char**、または **long** などのプリミティブ
- 単一の **String** 引数を受け入れるコンストラクターを持つオブジェクト
- 単一の **String** 引数を受け入れる静的 **valueOf()** メソッドを持つオブジェクト
- List<**T**>、Set<**T**>、または SortedSet<**T**> オブジェクト。ここで **T** はリスト内の他の条件を満たします。



注記

インジェクションアノテーションがサポートされるデータ型に対する要件が異なる場合、アノテーションについては相違点が強調表示されます。

47.2. JAX-RS API の使用

47.2.1. JAX-RS アノテーションタイプ

標準の JAX-RS API は、値をフィールド、Bean プロパティー、およびメソッドパラメーターに注入するために使用できるアノテーションを指定します。アノテーションは、以下の 3 つのタイプに分割できます。

- 「リクエスト URI からのデータの注入」
- 「HTTP メッセージヘッダーからのデータの注入」
- 「HTML フォームからのデータの挿入」

47.2.2. リクエスト URI からのデータの注入

概要

RESTful Web サービスを設計するためのベストプラクティスの 1 つとして、各リソースに一意の URI を設定する必要があります。開発者はこの原則を使用して、基盤となるリソースの実装に適量の情報を提供できます。リソースの URI テンプレートを設計する場合には、開発者は、リソースの実装に注入できるパラメーター情報を含めるようにテンプレートを作成できます。また、開発者はクエリーおよびマトリックスパラメーターを使用して、情報をリソース実装にフィードすることもできます。

URI のパスからのデータ取得

リソースの URI テンプレートの作成に使用される変数でリソースに関する情報を取得することが一般的なメカニズムです。これは、**javax.ws.rs.PathParam** アノテーションを使用して実現されます。**@PathParam** アノテーションには、データの注入元となる URI テンプレート変数を識別する単一のパラメーターがあります。

例47.1「URI テンプレート変数からのデータの注入」 では、**@PathParam** アノテーションは URI テンプレート変数 **color** の値が **itemColor** フィールドにインジェクトされることを指定します。

例47.1 URI テンプレート変数からのデータの注入

```
import javax.ws.rs.Path;
import javax.ws.rs.PathParam
...

@Path("/boxes/{shape}/{color}")
class Box
{
  ...

  @PathParam("color")
  String itemColor;

  ...
}
```

@PathParam アノテーションでサポートされるデータ型は、「サポートされるデータタイプ」 で説明されているものとは異なります。**@PathParam** アノテーションがデータを注入するエンティティーは、以下の型のいずれかでなければなりません。

- PathSegment
  値は、パスに一致する部分の最後のセグメントになります。

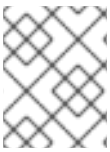
- List<PathSegment>
  この値は、名前付きの template パラメーターに一致するパスセグメントに対応する PathSegment オブジェクトのリストです。
- **int**、**char**、または **long** などのプリミティブ
- 単一の **String** 引数を受け入れるコンストラクターを持つオブジェクト
- 単一の **String** 引数を受け入れる静的 **valueOf()** メソッドを持つオブジェクト

クエリーパラメーターの使用

Web に情報を渡す一般的な方法は、URI で **クエリーパラメーター** を使用することです。クエリーパラメーターは URI の最後に表示され、疑問符 (**?**) で URI のリソース場所部分から区切られます。これらは、名前と値が等号 (**=**) で区切られた 1 つ以上の名前と値のペアで設定されます。複数のクエリーパラメーターを指定すると、ペアはセミコロン (**;**) またはアンパサンド (**&**) で区切られます。例47.2「クエリー文字列のある URI」 は、クエリーパラメーターが含まれる URI の構文を示しています。

例47.2 クエリー文字列のある URI

```
http://fusesource.org?name=value;name2=value2;...
```



注記

セミコロンまたはアンパサンドの **いずれか** を使用してクエリーパラメーターを区切ることができますが、両方は使用できません。

javax.ws.rs.QueryParam アノテーションは、クエリーパラメーターの値を抽出し、それを JAX-RS リソースに注入します。アノテーションでは、パラメーターを 1 つ使用できます。このパラメーターをもとに、値の抽出、指定されたフィールド、Bean プロパティー、またはパラメーターにインジェクトされるクエリーパラメーターの名前が識別されます。**@QueryParam** アノテーションは、「サポートされるデータタイプ」で説明されている型をサポートします。

例47.3「クエリーパラメーターのデータを使用したリソースメソッド」 は、クエリーパラメーター **id** の値をメソッドの **id** パラメーターに注入するリソースメソッドを示しています。

例47.3 クエリーパラメーターのデータを使用したリソースメソッド

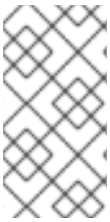
```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  @Path("/{type}")
  public void updateMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
```

```
    ...
  }
  ...
}
```

HTTP **POST** を /monstersforhire/daikaiju?id=jonas に処理するには、**updateMonster()** メソッドの **type** は **daikaiju** に設定され、**id** は **jonas** に設定されます。

マトリクスパラメーターの使用

URI クエリーパラメーターと同様に、URI マトリックスパラメーターは、リソース選択用の追加情報を提供できる名前と値のペアです。クエリーパラメーターとは異なり、マトリクスパラメーターは URI のどこにでも置くことができ、セミコロン (;) を使用して URI の階層パスセグメントから区別されます。**/mostersforhire/daikaiju;id=jonas** には **id** という 1 つのマトリクスパラメーターがあり、**/monstersforhire/japan;type=daikaiju/flying;wingspan=40** には **type** および **wingspan** という 2 つのマトリクスパラメーターがあります。



注記

リソースの URI を計算するときには、マトリックスパラメーターは評価されません。そのため、要求 URI **/monstersforhire/japan;type=daikaiju/flying;wingspan=40** は **/monstersforhire/japan/flying** の処理に適したリソースを見つけるために使用される URI です。

マトリクスパラメーターの値は、**javax.ws.rs.MatrixParam** アノテーションを使用してフィールド、パラメーター、または Bean プロパティーに注入されます。アノテーションでは、パラメーターを 1 つ使用できます。このパラメーターをもとに、値の抽出、指定されたフィールド、Bean プロパティー、またはパラメーターにインジェクトされるマトリックスパラメーターの名前が識別されます。**@MatrixParam** アノテーションは、「サポートされるデータタイプ」で説明されている型をサポートします。

例47.4「マトリックスパラメーターのデータを使用するリソースメソッド」 は、マトリクスパラメーター **type** および **id** の値をメソッドのパラメーターに注入するリソースメソッドを表しています。

例47.4 マトリックスパラメーターのデータを使用するリソースメソッド

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

HTTP **POST** を **/monstersforhire;type=daikaiju;id=whale** に処理するには、**updateMonster()** メソッドの **type** は **daikaiju** に設定され、**id** は **whale** に設定されます。

注記

JAX-RS は URI の全マトリックスパラメーターを一度に評価するため、URI のマトリックスパラメーターの場所に、制約を適用することはできません。たとえば、**/monstersforhire/japan;type=daikaiju/flying;wingspan=40** , **/monstersforhire/japan/flying;type=daikaiju;wingspan=40**, and **/monstersforhire/japan;type=daikaiju; wingspan=40/flying** はすべて、JAX-RS API を使用して実装される RESTful Web サービスと同等のものとして処理されます。

URI のデコードの無効化

デフォルトでは、すべてのリクエスト URI がデコードされます。そのため、URI **/monster/night%20stalker** と URI **/monster/night stalker** は同等です。自動 URI デコードを使用すると、ASCII 文字セット以外の文字をパラメーターとして送信しやすくなります。

URI を自動的にデコードしたくない場合は、**javax.ws.rs.Encoded** アノテーションを使用して URI のデコードを非アクティブ化できます。アノテーションは、以下のレベルで URI のデコードを非アクティブにするために使用できます。

- クラスレベル - **@Encoded** アノテーションでクラスをデコードすると、クラス内のすべてのパラメーター、フィールド、および Bean プロパティーの URI デコードが無効になります。
- メソッドレベル - **@Encoded** アノテーションでメソッドをデコードすると、クラスのすべてのパラメーターの URI デコードが無効になります。
- パラメーター/フィールドレベル - パラメーターまたはフィールドを **@Encoded** アノテーションでデコレートすると、クラスのすべてのパラメーターの URI デコードが無効になります。

例47.5「URI のデコードの無効化」は、**getMonster()** メソッドが URI デコードを使用しないリソースを示しています。**addMonster()** メソッドは、**type** パラメーターの URI デコードのみを無効にします。

例47.5 URI のデコードの無効化

```
@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)
```

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

エラー処理

URI インジェクションアノテーションのいずれかを使用してデータを注入しようとすると、元の例外をラップする WebApplicationException 例外が生成されます。WebApplicationException 例外のステータスは **404** に設定されます。

47.2.3. HTTP メッセージヘッダーからのデータの注入

概要

通常、要求メッセージの HTTP ヘッダーとメッセージに関する一般的な情報、送信方法、および予想される応答の詳細を渡します。いくつかの標準ヘッダーが一般的に認識されて使用されていますが、HTTP 仕様では、任意の名前/値ペアを HTTP ヘッダーとして使用できます。JAX-RS API には、HTTP ヘッダー情報をリソース実装に注入する簡単なメカニズムがあります。

最も一般的に使用される HTTP ヘッダーの 1 つがクッキーです。Cookie を使用すると、HTTP クライアントとサーバーが複数の要求/応答シーケンスで静的情報を共有できます。JAX-RS API には、クッキーから直接リソース実装にデータを注入するアノテーションがあります。

HTTP ヘッダーからの情報の注入

javax.ws.rs.HeaderParam アノテーションは、HTTP ヘッダーフィールドのデータをパラメーター、フィールド、または Bean プロパティーに注入するために使用されます。値が抽出され、リソース実装にインジェクトされる HTTP ヘッダーフィールドの名前を指定する単一のパラメーターがあります。関連するパラメーター、フィールド、または Bean プロパティーは、「サポートされるデータタイプ」 で説明されているデータタイプに準拠している必要があります。

If-Modified-Since ヘッダーの挿入 は、HTTP **If-Modified-Since** ヘッダーの値をクラスの **oldestDate** フィールドに注入するコードを示しています。

If-Modified-Since ヘッダーの挿入

```
import javax.ws.rs.HeaderParam;
...
class RecordKeeper
{
  ...
  @HeaderParam("If-Modified-Since")
  String oldestDate;
  ...
}
```

クッキーからの情報の挿入

クッキーは特別な HTTP ヘッダータイプです。これらは、最初の要求でリソース実装に渡される 1 つ以上の名前と値のペアで設定されます。最初の要求後に、クッキーは各クッキーでプロバイダーとコン

シューマーの間を行き来します。コンシューマーだけがリクエストを生成するため、Cookie を変更できます。クッキーは通常、複数の要求/応答シーケンス間でセッションを維持し、ユーザー設定やその他の永続化可能なデータを保存するために使用されます。

javax.ws.rs.CookieParam アノテーションは、クッキーのフィールドから値を抽出し、これをリソース実装に注入します。このアノテーションでは、値が抽出される Cookie のフィールドの名前を指定する単一のパラメーターを使用できます。「サポートされるデータタイプ」に示されているデータ型に加え、**@CookieParam** が付けられたエンティティーも **Cookie** オブジェクトになることができます。

例47.6「クッキーの挿入」は、**handle** クッキーの値を **CB** クラスのフィールドに注入するためのコードを示しています。

例47.6 クッキーの挿入

```
import javax.ws.rs.CookieParam;
...
class CB
{
  ...
  @CookieParam("handle")
  String handle;
  ...
}
```

エラー処理

HTTP メッセージのインジェクションアノテーションのいずれかを使用してデータを注入しようとすると、元の例外をラップする WebApplicationException 例外が生成されます。WebApplicationException 例外のステータスは **400** に設定されます。

47.2.4. HTML フォームからのデータの挿入

概要

HTML フォームでは、ユーザーから情報を簡単に取得する方法で、作成も簡単です。フォームデータは、HTTP **GET** リクエストおよび HTTP **POST** リクエストに使用できます。

GET
: フォームデータが HTTP **GET** リクエストの一部として送信されると、データはクエリーパラメーターのセットとして URI に追加されます。クエリーパラメーターからのデータの注入については、「クエリーパラメーターの使用」 で説明します。

POST
: フォームデータが HTTP **POST** の一部として送信されると、データは HTTP メッセージボディーに配置されます。フォームデータは、対象のフォームデータをサポートする通常のエンティティーパラメーターを使用して処理できます。また、**@FormParam** アノテーションを使用してデータを抽出し、リソースメソッドのパラメーターに注入することによって処理することもできます。

@FormParam アノテーションを使用したフォームデータの挿入

javax.ws.rs.FormParam アノテーションは、フィールド値をフォームデータから抽出し、値をリソースメソッドパラメーターに注入します。このアノテーションでは、値を抽出するフィールドのキーを指定する単一のパラメーターを指定できます。関連するパラメーターは、「サポートされるデータタイ

プ」 に記載のデータタイプに準拠する必要があります。

重要

JAX-RS API Javadoc には、**@FormParam** アノテーションをフィールド、メソッド、およびパラメーターに配置できると記載されています。ただし、**@FormParam** アノテーションは、リソースメソッドパラメーターに配置する場合にのみ意味があります。

例

リソースメソッドパラメーターへのフォームデータの挿入 は、フォームデータをパラメーターに注入するリソースメソッドを示しています。このメソッドは、クライアントのフォームに文字列データが含まれる **title**、**tags**、および **body** の 3 つのフィールドが含まれていることを前提としています。

リソースメソッドパラメーターへのフォームデータの挿入

```
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. 挿入するデフォルト値の指定

概要

より堅牢なサービス実装を提供するために、任意のパラメーターをデフォルト値に設定することができます。長い URI 文字列を入力するとエラーが発生しやすいため、これはクエリーパラメーターとマトリックスパラメーターから取得される値に特に役立ちます。また、要求側のシステムに適切な情報がないまま、すべての値を使用してクッキーを構築している可能性があるので、クッキーから抽出されたパラメーターにデフォルト値を設定すると良いでしょう。

javax.ws.rs.DefaultValue アノテーションは、以下のインジェクションアノテーションと併用できます。

- **@PathParam**
- **@QueryParam**
- **@MatrixParam**
- **@FormParam**
- **@HeaderParam**
- **@CookieParam**

@DefaultValue アノテーションは、インジェクションアノテーションに対応するデータがリクエストに存在しない場合に利用するデフォルト値を指定します。

構文

パラメーターのデフォルト値を設定するための構文に、**@DefaultValue** アノテーションを使用するための構文を示します。

パラメーターのデフォルト値を設定するための構文

```
import javax.ws.rs.DefaultValue;
  ...
  void resourceMethod(@MatrixParam("matrix")
                      @DefaultValue("value)
                      int someValue, ... )
  ...
```

このアノテーションは、パラメーター、Bean、フィールドの前に配置する必要があります。これは有効になります。付属のインジェクションアノテーションに対する **@DefaultValue** アノテーションの位置は重要ではありません。

@DefaultValue アノテーションは、単一のパラメーターを取ります。このパラメーターは、インジェクションアノテーションに基づいて適切なデータを抽出できない場合にフィールドに注入される値です。値は、任意の **String** 値にすることができます。この値は、関連付けられたフィールドのタイプと互換性がある必要があります。たとえば、関連付けられたフィールドが **int** 型である場合、**blue** のデフォルト値では例外が発生します。

リストとセットの処理

アノテーション付きのパラメーター、Bean、またはフィールドのタイプが List、Set、または SortedSet の場合に、生成されるコレクションには指定されたデフォルト値から 1 つのエントリーがマッピングされます。

例

デフォルト値の設定 は、**@DefaultValue** を使用して値が中ニュされるフィールドのデフォルト値を指定する 2 つの例を示しています。

デフォルト値の設定

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

```
  ...
}
```

デフォルト値の設定 の **getMonster()** メソッドは、**GET** リクエストが **baseURI/monster** に送信されると呼び出されます。メソッドでは、**id** と **type** の 2 つのクエリーパラメーターが URI に追加されることが想定されます。そのため、URI **baseURI/monster?id=1&type=fomóiri** を使用する **GET** リクエストは 1 の ID で Fomóiri を返します。

@DefaultValue アノテーションは両方のパラメーターに配置されるため、クエリーパラメーターが省略されると **getMonster()** メソッドが機能できます。**baseURI/monster** に送信された **GET** リクエストは、URI **baseURI/monster?id=42&type=bogeyman** を使用する **GET** リクエストと同等です。

47.2.6. Java Bean へのパラメーターの挿入

概要

REST 経由で HTML フォームを投稿する場合は、サーバー側で一般的なパターンで Java Bean を作成して、フォームで受け取ったすべてのデータをカプセル化します (場合によっては他のパラメーターや HTML ヘッダーからでもデータもカプセル化します)。通常、この Java Bean の作成は 2 段階のプロセスになります。リソースメソッドはインジェクションによって (たとえば、**@FormParam** メソッドパラメーターにアノテーションを追加することによって) フォームの値を受け取り、リソースメソッドは Bean のコンストラクタを呼び出し、フォームデータを渡します。

JAX-RS 2.0 **@BeanParam** アノテーションを使用すると、このパターンを 1 段階で実装できます。フォームデータは bean クラスのフィールドに直接挿入でき、Bean 自体は JAX-RS ランタイムによって自動的に作成されます。これは、例を使用することで最も簡単に説明できます。

挿入ターゲット

@BeanParam アノテーションは、リソースメソッドパラメーター、リソースフィールド、または Bean プロパティーに追加できます。ただし、パラメーターターゲットは、すべてのリソースクラスのライフサイクルで使用できる唯一の種類のターゲットです。他の種類のターゲットは、要求別のライフサイクルに制限されます。この状況は、表47.1「@BeanParam Injection Targets」 にまとめられています。

表47.1 @BeanParam Injection Targets

Target	リソースクラスのライフサイクル
PARAMETER	すべて
FIELD	per-request(デフォルト)
METHOD (Bean プロパティー)	per-request(デフォルト)

BeanParam アノテーションのない例

以下の例は、従来のアプローチ (**@BeanParam** を使用しない) を使用して Java Bean でフォームデータを取得する方法を表しています。

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

この例では、**orderTable** メソッドは家具の Web サイトからある数量のテーブルを注文するために使用されるフォームを処理します。注文フォームが投稿されると、フォームの値は **orderTable** メソッドのパラメーターに注入され、**orderTable** メソッドは注入されたフォームデータを使用して **TableOrder** クラスのインスタンスを明示的に作成します。

BeanParam アノテーションの使用例

上記の例は、**@BeanParam** アノテーションを利用するためにリファクタリングできます。**@BeanParam** のアプローチを使用する場合、フォームパラメーターは Bean クラスの **TableOrder** フィールドに直接注入できます。実際、**@PathParam**、**@QueryParam**、**@FormParam**、**@MatrixParam**、**@CookieParam**、および **@HeaderParam** などの、Bean クラスの標準の JAX-RS パラメーターアノテーションを使用できます。フォームを処理するコードは、以下のようにリファクタリングできます。

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

これでフォームアノテーションが Bean クラス TableOrder に追加され、以下のようにリソースメソッドの署名にある **@FormParam** アノテーションをすべて単一の **@BeanParam** アノテーションで置き換えることができるようになりました。フォームが **orderTable** リソースメソッドに投稿されると、JAX-RS ランタイムは **TableOrder** インスタンス **orderBean** を自動的に作成し、Bean クラスのパラメーターアノテーションが指定するすべてのデータを注入するようになりました。

47.3. パラメーターコンバーター

概要

パラメーターコンバーターを使用すると、パラメーター (**String** 型の) をフィールド、Bean プロパティー、またはリソースメソッド引数の **任意** の型に注入することができます。適切なパラメーターコンバーターを実装してバインディングすることで、JAX-RS ランタイムを拡張して、パラメーター文字列の値をターゲットタイプに変換できます。

自動変換

パラメーターは **String** のインスタンスとして受信されるため、**String** 型のフィールド、Bean プロパティー、およびメソッドパラメーターに常に注入することができます。さらに、JAX-RS ランタイムには、パラメーター文字列を以下の型に自動的に変換する機能があります。

1. プリミティブ型。
2. 単一の **String** 引数を受け入れるコンストラクターを持つ型。
3. 型のインスタンスを返す String 引数が 1 つあり、**valueOf** または **fromString** という名前の静的メソッドを持つ型。
4. **T** が 2 または 3 で説明する型のいずれかの場合、 **List<T>**、**Set<T>**、または **SortedSet<T>**。

パラメーターコンバーター

自動変換が対応していない型にパラメーターを注入するには、型のカスタム **パラメーターコンバーター** を定義できます。パラメーターコンバーターは、**String** からカスタムタイプへの変換、およびカスタムタイプから **String** への逆方向の変換を定義できる JAX-RS エクステンションです。

ファクトリーパターン

JAX-RS パラメーターコンバーターメカニズムは、ファクトリーパターンを使用します。したがって、パラメーターコンバーターを直接登録するのではなく、パラメーターコンバータープロバイダー (型 **javax.ws.rs.ext.ParamConverterProvider**) を登録する必要があり、オンデマンドでパラメーターコンバーター (型 **javax.ws.rs.ext.ParamConverter**) が作成されます。

ParamConverter インターフェイス

javax.ws.rs.ext.ParamConverter インターフェイスは以下のように定義されます。

```
// 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);
}
```

独自の **ParamConverter** クラスを実装するには、このインターフェイスを実装し、**fromString** メソッド (パラメーター文字列をターゲット型に変換) および **toString** メソッド (ターゲット型を文字列に変換) をオーバーライドする必要があります。

ParamConverterProvider interface

javax.ws.rs.ext.ParamConverterProvider インターフェイスは以下のように定義されます。

```
// 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[]);
}
```

独自の **ParamConverterProvider** クラスを実装するには、このインターフェイスを実装し、**ParamConverter** インスタンスを作成するファクトリーメソッドである **getConverter** メソッドをオーバーライドする必要があります。

パラメーターコンバータープロバイダーのバインディング

パラメーターコンバータープロバイダーを JAX-RS ランタイムに **バインド** (アプリケーションで利用可能) するには、以下のように実装クラスに **@Provider** アノテーションを付ける必要があります。

```
// Java
...
import javax.ws.rs.ext.ParamConverterProvider;
```

```
import javax.ws.rs.ext.Provider;

@Provider
public class TargetTypeProvider implements ParamConverterProvider {
    ...
}
```

このアノテーションを使用することで、パラメーターコンバータープロバイダーがデプロイメントのスキャンフェーズで自動的に登録されるようになります。

例

以下の例は、パラメーター文字列を **TargetType** 型に変換する機能を持つ **ParamConverterProvider** および **ParamConverter** を実装する方法を示しています。

```
// 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;
    }

}
```

パラメーターコンバーターの使用

TargetType のパラメーターコンバーターを定義したので、パラメーターを **TargetType** フィールドおよび引数に直接インジェクトできるようになりました。次に例を示します。

```
// Java
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
...
@POST
public Response updatePost(@FormParam("target") TargetType target)
{
  ...
}
```

デフォルト値の遅延変換

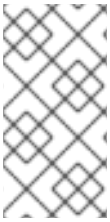
パラメーターのデフォルト値を指定する場合 (**@DefaultValue** アノテーションを使用して)、デフォルト値がターゲットの型にすぐ変換される (デフォルトの動作) か、必要な場合にのみ変換されるべきか (遅延変換) を選択することができます。遅延変換を選択するには、**@ParamConverter.Lazy** アノテーションをターゲット型に追加します。以下に例を示します。

```
// 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. APACHE CXF エクステンションの使用

概要

Apache CXF では、標準の JAX-WS インジェクションメカニズムへのエクステンションを提供しており、開発者が一連のインジェクションアノテーションを単一のアノテーションで置き換えることができます。アノテーションを使用して抽出されたデータのフィールドを含む Bean に、単一のアノテーションが配置されます。たとえば、リソースメソッドに **id**、**type**、および **size** という 3 つのクエリーパラメーターが含まれることが想定される場合、単一の **@QueryParam** アノテーションを使用して、対応するフィールドを持つ Bean にすべてのパラメーターを注入できます。



注記

代わりに **@BeanParam** アノテーションの使用を検討してください (JAX-RS 2.0 以降で利用可能)。標準的な **@BeanParam** のアプローチはプロプライエタリー Apache CXF エクステンションよりも柔軟性が高いため、推奨される代替手段となります。詳細は、「Java Bean へのパラメーターの挿入」 を参照してください。

サポートされるインジェクションアノテーション

このエクステンションは、すべてのインジェクションパラメーターをサポートする訳ではありません。以下のみをサポートします。

- **@PathParam**
- **@QueryParam**
- **@MatrixParam**
- **@FormParam**

構文

アノテーションが Bean へのシリアルインジェクションを使用することを指定するには、以下の 2 点を実行する必要があります。

1. アノテーションのパラメーターを空の文字列として指定します。たとえば、**@PathParam("")** は、URI テンプレート変数のシーケンスを Bean にシリアライズするように指定します。
2. アノテーション付きのパラメーターが、挿入される値と一致するフィールドを含む Bean であることを確認します。

例

例47.7「クエリーパラメーターの Bean へのインジェクト」 は、多数のクエリーパラメーターを Bean に挿入する例を示しています。リソースメソッドでは、リクエスト URI に **type** および **id** の 2 つのクエリーパラメーターが含まれることが想定されています。これらの値は、**Monster** Bean の対応するフィールドに注入されます。

例47.7 クエリーパラメーターの Bean へのインジェクト

```
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 コンシューマーへの情報の返却

概要

RESTful 要求では、少なくとも HTTP 応答コードをコンシューマーに返す必要があります。多くの場合、プレーンな JAXB オブジェクトまたは **GenericEntity** オブジェクトを返すことで要求を満たすことができます。リソースメソッドが応答エンティティーと共に追加のメタデータを返す必要がある場合、JAX-RS リソースメソッドは必要な HTTP ヘッダーまたは他のメタデータを含む **Response** オブジェクトを返すことができます。

48.1. 戻り値のタイプ

コンシューマーに返される情報によって、リソースメソッドが返すオブジェクトの正確なタイプが決まります。これは明白に思えるかもしれませんが、Java の戻りオブジェクトと RESTful コンシューマーに返されるものの間のマッピングは 1 対 1 ではありません。少なくとも、応答エンティティーのボディーに加えて、RESTful コンシューマーは有効な HTTP リターンコードを返す必要があります。Java オブジェクトに含まれるデータの応答エンティティーへのマッピングは、コンシューマーが受け入れる MIME タイプにより変化します。

Java オブジェクトを RESTful 応答メッセージにマッピングする時に発生する問題に対処するために、リソースメソッドは以下の 4 種類の Java コンストラクトを返すことができます。

- 「プレーンな Java コンストラクトを返します。」 JAX-RS ランタイムによって決定される HTTP リターンコードで基本情報を返します。
- 「プレーンな Java コンストラクトを返します。」 JAX-RS ランタイムによって決定される HTTP リターンコードで複雑な情報を返します。
- 「アプリケーションの応答の微調整」 プログラムにより決定された HTTP リターンステータスで複雑な情報を返します。**Response** オブジェクトにより、HTTP ヘッダーを指定することも可能です。
- 「汎用型情報を含むエンティティーの返答」 JAX-RS ランタイムによって決定される HTTP リターンコードで複雑な情報を返します。**GenericEnitity** オブジェクトは、データをシリアライズするランタイムコンポーネントに詳細情報を提供します。

48.2. プレーンな JAVA コンストラクトを返します。

概要

多くの場合、リソースクラスは標準の Java タイプ、JAXB オブジェクト、またはアプリケーションにエンティティープロバイダーがある任意のオブジェクトを返すことができます。このような場合には、ランタイムは返されるオブジェクトの Java クラスを使用して MIME タイプの情報を決定します。ランタイムは、コンシューマーに送信する適切な HTTP リターンコードも決定します。

戻り値の型

リソースメソッドは、**void** またはエンティティーライターが提供されている Java 型を返すことができます。デフォルトでは、ランタイムには以下のプロバイダーがあります。

- Java プリミティブ
- Java プリミティブの **Number** 表現

- JAXB オブジェクト

「ネイティブでサポートされるタイプ」 は、デフォルトでサポートされるすべての戻り値のタイプをリスト表示します。「カスタムライター」 は、カスタムエンティティーライターの実装方法を記述します。

MIME タイプ

ランタイムは、最初にリソースメソッドおよび **@Produces** アノテーションのリソースクラスをチェックして、返されるエンティティーの MIME タイプを決定します。見つかった場合は、アノテーションで指定された MIME タイプを使用します。リソース実装によって指定されたものが見つからない場合は、エンティティープロバイダーにより、適切な MIME タイプを決定します。

デフォルトでは、ランタイムは MIME タイプを以下のように割り当てます。

- Java プリミティブおよびその **Number** 表現には、MIME タイプの **application/octet-stream** が割り当てられます。
- JAXB オブジェクトには、**application/xml** の MIME タイプが割り当てられます。

アプリケーションは、「カスタムライター」 で説明されているようにカスタムエンティティープロバイダーを実装して、他のマッピングを使用できます。

レスポンスコード

リソースメソッドがプレーンな Java 構造を返す場合、リソースメソッドが例外を出力せずに完了すると、ランタイムは応答のステータスコードを自動的に設定します。ステータスコードは以下のように設定されます。

- **204** (コンテンツなし) - リソースメソッドの戻り値の型は **void** です。
- **204** (コンテンツなし) - 返されるエンティティーの値は **null** です。
- **200** (OK) - 返されるエンティティーの値は **null** ではありません。

リソースメソッドが完了する前に例外が出力された場合に、50章 *例外処理* で説明されているようにリターンステータスコードを設定します。

48.3. アプリケーションの応答の微調整

48.3.1. 応答構築の基本

概要

RESTful サービスでは、リソースメソッドがプレーンな Java コンストラクトを返す場合に、コンシューマーへ返される応答をより正確に制御する必要があります。JAX-RS **Response** クラスを使用すると、リソースメソッドがコンシューマーに送信される戻り値をある程度制御し、応答に HTTP メッセージヘッダーとクッキーを指定できます。

Response オブジェクトは、コンシューマーに返されるエンティティーを表すオブジェクトをラップします。**Response** オブジェクトは **ResponseBuilder** クラスをファクトリーとして使用してインスタンス化されます。

ResponseBuilder クラスには、応答のメタデータを操作するために使用されるメソッドも多数あります。たとえば、**ResonseBuilder** クラスには、HTTP ヘッダーとキャッシュ制御ディレクティブを設定するメソッドが含まれます。

応答と応答ビルダーの関係

Response クラスには保護されたコンストラクターがあるため、直接インスタンス化することはできません。これらは、**Response** クラスによって囲まれた **ResponseBuilder** クラスを使用して作成されます。**ResponseBuilder** クラスは、そこから作成された応答にカプセル化されるすべての情報のホルダーです。**ResponseBuilder** クラスには、メッセージに HTTP ヘッダープロパティーを設定するすべてのメソッドもあります。

Response クラスは、適切な応答コードの設定とエンティティーのラップを容易にするメソッドを提供します。一般的な応答ステータスコードごとにメソッドがあります。エンティティー本体または必要なメタデータを含むステータスに対応するメソッドには、関連する応答ビルダーに、情報を直接設定できるバージョンが含まれています。

ResponseBuilder クラスの **build()** メソッドは、メソッドが呼び出される際に応答ビルダーに保存されている情報が含まれる応答オブジェクトを返します。応答オブジェクトが返されると、応答ビルダーはクリーンな状態に戻ります。

応答ビルダーの取得

応答ビルダーを取得する方法は 2 つあります。

- **Response** クラスを使用した応答ビルダーの取得 に示すように、**Response** クラスの静的メソッドを使用します。

  Response クラスを使用した応答ビルダーの取得

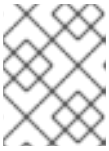
  ```
  import javax.ws.rs.core.Response;

  Response r = Response.ok().build();
  ```

  この方法で応答ビルダーを取得すると、複数の手順で操作できるインスタンスに、アクセスできなくなります。すべてのアクションを単一のメソッド呼び出しに配列する必要があります。

- Apache CXF 固有の **ResponseBuilderImpl** クラスの使用。このクラスを使用すると、応答ビルダーを直接操作できます。ただし、すべての応答ビルダーの情報を手動で設定する必要があります。
  例48.1「**ResponseBuilderImpl** クラスを使用した応答ビルダーの取得」は、**ResponseBuilderImpl** クラスを使用して **Response** クラスを使用した応答ビルダーの取得 を書き換える方法を表しています。

  例48.1ResponseBuilderImpl クラスを使用した応答ビルダーの取得

  ```
  import javax.ws.rs.core.Response;
  import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

  ResponseBuilderImpl builder = new ResponseBuilderImpl();
  builder.status(200);
  Response r = builder.build();
  ```



注記

Response クラスのメソッドから返される **ResponseBuilder** を **ResponseBuilderImpl** オブジェクトに割り当てることもできます。

補足情報

Response クラスの詳細は、**Response** クラスの Javadoc を参照してください。

ResponseBuilder クラスの詳細は、**ResponseBuilder** クラスの Javadoc を参照してください。

Apache CXF **ResponseBuilderImpl** クラスの詳細は、**ResponseBuilderImpl** Javadoc を参照してください。

48.3.2. 一般的なユースケースに対する応答の作成

概要

Response クラスは、RESTful サービスが必要とするより一般的な応答を処理するためのショートカットメソッドを提供します。これらのメソッドは、提供された値またはデフォルト値のいずれかを使用して適切なヘッダーの設定を処理します。また、必要に応じてエンティティーボディーの設定を処理します。

正常な要求の応答作成

要求が正常に処理されると、アプリケーションが応答を送信して、要求が完了したことを確認する必要があります。この応答にはエンティティーが含まれる可能性があります。

応答を正常に完了した場合の最も一般的な応答は **OK** です。**OK** 応答には通常、リクエストに対応するエンティティーが含まれます。**Response** クラスには、応答のステータスを **200** に設定し、指定されたエンティティーを囲まれた応答ビルダーに追加するオーバーロードされた **ok()** メソッドがあります。**ok()** メソッドには 5 つのバージョンがあります。最も一般的に使用されるバリアントは以下のとおりです。

- **Response.ok()** - **200** のステータスおよび空のエンティティーボディーでレスポンスを作成します。
- **Response.ok(java.lang.Object entity)** - **200** のステータスで応答を作成し、指定されたオブジェクトを応答エンティティーボディーに保存します。また、オブジェクトのイントロスペクションによるエンティティーメディア型を決定します。

200 応答での応答の作成 は、**OK** ステータスで応答を作成する例を示しています。

200 応答での応答の作成

```
import javax.ws.rs.core.Response;
import demo.jaxrs.server.Customer;
...

Customer customer = new Customer("Jane", 12);

return Response.ok(customer).build();
```

要求元がエンティティーボディーを期待しない場合は、**200 OK** のステータスではなく **204 No Content** ステータスを送信する方が適切な場合があります。**Response.noContent()** メソッドにより、適切な応答オブジェクトが作成されます。

204 ステータスでの応答の作成 は、**204** ステータスで応答を作成する例を示しています。

204 ステータスでの応答の作成

```
import javax.ws.rs.core.Response;

return Response.noContent().build();
```

リダイレクトの応答の作成

Response クラスは、3 つのリダイレクト応答ステータスを処理するメソッドを提供します。

303 See Other

303 See Other ステータスは、リクエストされたリソースが要求を処理するためにコンシューマーを新しいリソースに永続的にリダイレクトする必要がある場合に便利です。
Response クラスの **seeOther()** メソッドは、**303** のステータスで応答を作成し、新しいリソース URI をメッセージの **Location** フィールドに配置します。**seeOther()** メソッドは、新しい URI を **java.net.URI** オブジェクトとして指定する単一のパラメーターを取ります。

304 Not Modified

304 Not Modified ステータスは、リクエストの性質に応じてさまざまなものに使用できます。前回の **GET** リクエスト以降、要求されたリソースが変更されていないことを示すことができます。リソース変更要求で、リソースが変更されなかったと示すために使用することもできます。
Response クラスの **notModified()** メソッドは **304** ステータスで応答を作成し、HTTP メッセージに変更された日付プロパティーを設定します。**notModified()** メソッドには 3 つのバージョンがあります。

- **notModified**
- **notModifiedjavax.ws.rs.core.Entitytag**
- **notModifiedjava.lang.Stringtag**

307 Temporary Redirect

307 Temporary Redirect ステータスは、要求されたリソースがコンシューマーを新しいリソースに転送する必要があるものの、このリソースを引き続き使用して今後の要求を処理する場合に役立ちます。
Response クラスの **temporaryRedirect()** メソッドは、**307** のステータスで応答を作成し、新しいリソース URI をメッセージの **Location** フィールドに配置します。**temporaryRedirect()** メソッドは、新しい URI を **java.net.URI** オブジェクトとして指定する単一のパラメーターを取ります。

304 ステータスでの応答の作成 は、**304** ステータスで応答を作成する例を示しています。

304 ステータスでの応答の作成

```
import javax.ws.rs.core.Response;

return Response.notModified().build();
```

エラーを示す応答の作成

Response クラスは、2 つの基本的な処理エラーに対する応答を作成するメソッドを提供します。

- **serverError** - **500 Internal Server Error** のステータスでレスポンスを作成します。
- **notAcceptablejava.util.List<javax.ws.rs.core.Variant>variants** - **406 Not Acceptable** ステータスと、許容可能なリソース型のリストが含まれるエンティティーボディーでレスポンスを作成します。

500 ステータスでの応答の作成 は、**500** ステータスで応答を作成する例を示しています。

500 ステータスでの応答の作成

```
import javax.ws.rs.core.Response;

return Response.serverError().build();
```

48.3.3. より高度な応答の処理

概要

Response クラスメソッドは、一般的なケースの応答を作成するためのショートカットを提供します。キャッシュ制御ディレクティブの指定、カスタム HTTP ヘッダーの追加、または **Response** クラスで処理されていないステータスの送信など、より複雑なケースに対応する必要がある場合は、**build()** メソッドを使用してレスポンスオブジェクトを生成する前に、**ResponseBuilder** クラスメソッドを使用してレスポンスを入力する必要があります。

「応答ビルダーの取得」 で説明されているように、Apache CXF **ResponseBuilderImpl** クラスを使用して直接操作できるレスポンスビルダーインスタンスを作成できます。

カスタムヘッダーの追加

カスタムヘッダーは、**ResponseBuilder** クラスの **header()** メソッドを使用してレスポンスに追加されます。**header()** メソッドは、2 つのパラメーターを取ります。

- **name** - ヘッダー名を指定する文字列
- **value** - ヘッダーに格納されたデータを含む Java オブジェクト

header() メソッドを繰り返し呼び出すと、メッセージに複数のヘッダーを設定することができます。

応答へのヘッダーの追加 は、応答にヘッダーを追加するコードを示します。

応答へのヘッダーの追加

```
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();
```

クッキーの追加

カスタムヘッダーは、**ResponseBuilder** クラスの **cookie()** メソッドを使用してレスポンスに追加されます。**cookie()** メソッドは1つ以上のクッキーを取ります。各クッキーは **javax.ws.rs.core.NewCookie** オブジェクトに保存されます。最も簡単に使用できる **NewCookie** クラスのコンストラクターは、2つのパラメーターを取ります。

- **name** - クッキーの名前を指定する文字列
- **value** - クッキーの値を指定する文字列

cookie() メソッドを繰り返し呼び出して、複数のクッキーを設定できます。

応答へのクッキーの追加 は、クッキーを応答に追加するコードを示します。

応答へのクッキーの追加

```
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

null パラメーターリストで **cookie()** メソッドを呼び出すと、すでに応答に関連付けられているクッキーが消去されます。

応答ステータスの設定

Response クラスのヘルパーメソッドでサポートされないステータスを返す場合は、**ResponseBuilder** クラスの **status()** メソッドを使用してレスポンスのステータスコードを設定することができます。**status()** メソッドには2つのバリアントがあります。1つは、レスポンスコードを指定する **int** を取ります。もう1つは **Response.Status** オブジェクトを取り、応答コードを指定します。

Response.Status クラスは、**Response** クラスで囲まれた列挙型です。定義されたほとんどの HTTP 応答コードに、エントリーがあります。

応答へのヘッダーの追加 に、応答ステータスを **404 Not Found** に設定するコードを示します。

応答へのヘッダーの追加

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(404);
Response r = builder.build();
```

キャッシュ制御ディレクティブの設定

ResponseBuilder クラスの **cacheControl()** メソッドを使用すると、レスポンスにキャッシュ制御ヘッダーを設定することができます。**cacheControl()** メソッドは、応答のキャッシュ制御ディレクティブを指定する **javax.ws.rs.CacheControl** オブジェクトを取ります。

CacheControl クラスには、HTTP 仕様でサポートされるすべてのキャッシュ制御ディレクティブに対応するメソッドがあります。ディレクティブが簡単な on または off の値である場合、setter メソッドは **boolean** の値を取ります。ディレクティブは **max-age** ディレクティブなどの数値が必要であるため、setter は **int** の値を取ります。

応答へのヘッダーの追加 は、**no-store** キャッシュ制御ディレクティブを設定するコードを表しています。

応答へのヘッダーの追加

```
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. 汎用型情報を含むエンティティーの返答

概要

アプリケーションで、返されたオブジェクトの MIME タイプや、応答のシリアル化に使用するエンティティープロバイダーを詳細に制御する必要がある場合があります。JAX-RS **javax.ws.rs.core.GenericEntity<T>** クラスは、エンティティーを表すオブジェクトの汎用型を指定するメカニズムを提供することで、エンティティーのシリアライズに対する詳細な制御を可能にします。

GenericEntity<T> オブジェクトの使用

応答をシリアライズするエンティティープロバイダーの選択に使用される基準の 1 つは、オブジェクトの汎用型です。オブジェクトの汎用型はオブジェクトの Java 型を表します。一般的な Java 型または JAXB オブジェクトが返されると、ランタイムは Java リフレクションを使用して汎用型を判断できます。ただし、JAX-RS **Response** オブジェクトが返されると、ランタイムはラップされたエンティティーの汎用型を決定できず、オブジェクトの実際の Java クラスが Java 型として使用されます。

エンティティープロバイダーに正しい汎用型情報が提供されるようにするため、エンティティーは返された **Response** オブジェクトに追加される前に、**GenericEntity<T>** オブジェクトでラップされます。

リソースメソッドは、**GenericEntity<T>** オブジェクトを直接返すこともできます。実際には、このアプローチはほぼ使用されません。通常、ラップされていないエンティティーのリフレクションによって決定される汎用型情報や、**GenericEntity<T>** オブジェクトでラップされたエンティティーに格納される汎用型情報は、通常同じです。

GenericEntity<T> オブジェクトの作成

GenericEntity<T> オブジェクトを作成する方法は 2 つあります。

1. ラップされるエンティティーを使用して **GenericEntity<T>** クラスのサブクラスを作成します。サブクラスを使用した GenericEntity<T> オブジェクトの作成 は、実行時に利用可能な汎用型を持つ型 **List<String>** のエンティティーが含まれる **GenericEntity<T>** オブジェクトを作成する方法を示しています。

   サブクラスを使用した GenericEntity<T> オブジェクトの作成

   ```
   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();
   ```

   GenericEntity<T> オブジェクトの作成に使用されるサブクラスは通常、匿名です。

2. エンティティーに汎用型情報を指定して、インスタンスを直接作成します。例 48.2「GenericEntity<T> オブジェクトの直接インスタンス化」 は、**AtomicInteger** 型のエンティティーが含まれる応答を作成する方法を示しています。

   例48.2 GenericEntity<T> オブジェクトの直接インスタンス化

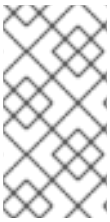
   ```
   import javax.ws.rs.core.GenericEntity;

   AtomicInteger result = new AtomicInteger(12);
   GenericEntity<AtomicInteger> entity =
     new GenericEntity<AtomicInteger>(result,
                                      result.getClass().getGenericSuperclass());
   Response response = Response.ok(entity).build();
   ```

48.5. 非同期応答

48.5.1. サーバー上の非同期処理

概要

サーバー側での呼び出しの非同期処理の目的は、スレッドをより効率的に使用できるようにすることで、最終的に、サーバーの要求スレッドがブロックされていることが原因でクライアントの接続試行が拒否されるというシナリオを回避することです。呼び出しが非同期的に処理されると、要求スレッドはほぼすぐに解放されます。



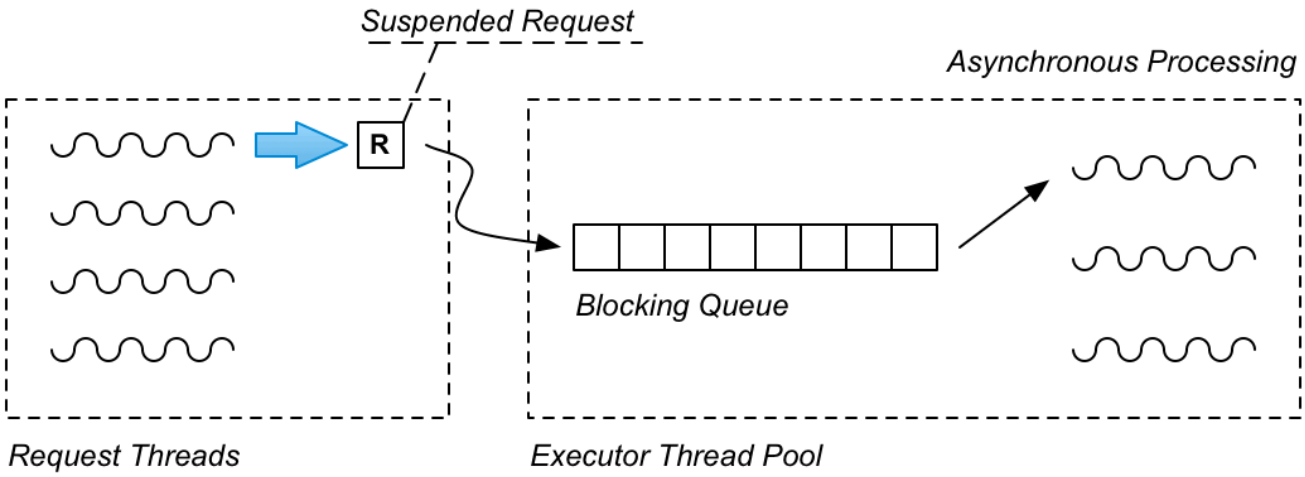
注記

サーバー側で非同期処理が有効になっている場合でも、サーバーから応答を受信するまでクライアントは、**ブロックされたまま** になることに注意してください。クライアント側で非同期動作を実行するには、クライアント側の非同期処理を実装する必要があります。「クライアントの非同期処理」を参照してください。

非同期処理の基本モデル

図48.1「非同期処理のスレッドモデル」 は、サーバー側での非同期処理の基本モデルの概要を示しています。

図48.1 非同期処理のスレッドモデル



つまり、リクエストは非同期モデルで以下のように処理されます。

1. 非同期リソースメソッドはリクエストスレッド内で呼び出されます (その後応答の送信に必要な **AsyncResponse** オブジェクトへの参照を受け取ります)。
2. リソースメソッドは、**Runnable** オブジェクトで一時停止されたリクエストをカプセル化します。これには、リクエストを処理するのに必要なすべての情報および処理ロジックが含まれます。
3. リソースメソッドは、Runnable オブジェクトをエグゼキュータースレッドプールのブロッキングキューにプッシュします。
4. リソースメソッドが返されるようになったため、要求スレッドが解放されます。
5. **Runnable** オブジェクトがキューの先頭に到達すると、エグゼキュータースレッドプールのスレッドの1つによって処理されます。カプセル化された **AsyncResponse** オブジェクトは、応答をクライアントに送り返すために使用されます。

Java エグゼキューターを使用したスレッドプールの実装

java.util.concurrent API は、完全なスレッドプールの実装を非常に簡単に作成できる強力な API です。Java 同時実行 API の用語では、スレッドプールは **エグゼキューター** と呼ばれます。作業スレッドやそれを提供するブロッキングキューなど、完全な作業スレッドプールを作成するには、コード1行のみが必要です。

たとえば、図48.1「非同期処理のスレッドモデル」 に記載されている **Executor Thread Pool** のような完全な作業スレッドプールを作成するには、以下のように **java.util.concurrent.Executor** インスタンスを作成します。

```
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
);
```

このコンストラクタは、5 つのスレッドを持つ新しいスレッドプールを作成し、最大 10 個の **Runnable** オブジェクトを保持できる単一のブロッキングキューによって供給されます。スレッドプールにタスクを送信するには、**executor.execute** メソッドを呼び出して、**Runnable** オブジェクト (非同期タスクをカプセル化する) への参照を渡します。

非同期リソースメソッドの定義

非同期であるリソースメソッドを定義するには、**@Suspended** アノテーションを使用して型 **javax.ws.rs.container.AsyncResponse** の引数を注入し、メソッドが **void** を返すことを確認します。以下に例を示します。

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

注入された **AsyncResponse** オブジェクトは後でレスポンスを返すために使用されるため、リソースメソッドは **void** を返す必要があることに注意してください。

AsyncResponse クラス

javax.ws.rs.container.AsyncResponse クラスは、受信クライアントコネクションにおける抽象ハンドルを提供します。**AsyncResponse** オブジェクトがリソースメソッドに注入されると、ベースとなる TCP クライアントコネクションは最初は **一時停止** 状態になります。後で応答を返す準備ができたら、**AsyncResponse** インスタンスで **resume** を呼び出すことで、基礎となる TCP クライアントコネクションを再度アクティブにし、レスポンスを返すことができます。または、呼び出しを中止する必要がある場合は、**AsyncResponse** インスタンスで **cancel** を呼び出すことができます。

一時停止した要求の Runnable としてのカプセル化

図48.1「非同期処理のスレッドモデル」 の非同期処理シナリオでは、一時停止された要求をキューにプッシュします。そのキューから、後で専用のスレッドプールで処理できます。ただし、このアプローチを機能させるには、オブジェクトで一時停止された要求を **カプセル化** する方法が必要です。一時停止された要求オブジェクトは以下をカプセル化する必要があります。

- 受信要求からのパラメーター (存在する場合)。
- **AsyncResponse** オブジェクト。受信クライアントコネクションのハンドルと、レスポンスの返信方法を提供します。

- 呼び出しのロジック。

これらをカプセル化する便利な方法は、**Runnable** クラスを定義して一時停止されたリクエストを表します。ここで、**Runnable.run()** メソッドは呼び出しのロジックをカプセル化します。これを実行する最も一般的な方法は、以下の例のように **Runnable** をローカルクラスとして実装することです。

非同期処理の例

非同期処理シナリオを実装するには、リソースメソッドの実装は **Runnable** オブジェクト (一時停止されたリクエストを表します) をエグゼキュータースレッドプールに渡す必要があります。Java 7 および 8 では、次の例に示すように、いくつかの新しい構文を利用して、**Runnable** クラスをローカルクラスとして定義できます。

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

リソースメソッド引数 **id** および **response** が、**Runnable** ローカルクラスの定義に直接渡される方法に注意してください。この特別な構文を使用すると、ローカルクラスの対応するフィールドを定義しなくても、**Runnable.run()** メソッドでリソースメソッドの引数を直接使用できます。

重要

この特別な構文を機能させるには、リソースメソッドパラメーターを **final** として宣言する **必要** があります (メソッド実装で変更できないことを意味します)。

48.5.2. タイムアウトおよびタイムアウトハンドラー

概要

非同期処理モデルには、REST 呼び出しにタイムアウトを指定するサポートがあります。デフォルトでは、タイムアウトになると HTTP エラー応答がクライアントに送信されます。ただし、タイムアウトハンドラーコールバックを登録するオプションもあります。これにより、タイムアウトイベントへの応答をカスタマイズできます。

ハンドラーなしでタイムアウトを設定する例

タイムアウトハンドラーを指定せずに単純な呼び出しタイムアウトを定義するには、以下の例のように **AsyncResponse** オブジェクトで **setTimeout** メソッドを呼び出します。

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

java.util.concurrent.TimeUnit クラスの任意の時間単位を使用して、タイムアウト値を指定できることに注意してください。上記の例では、要求をエグゼキュータースレッドプールに送信するコードは提示されていません。タイムアウトの動作をテストするだけであれば、リソースメソッド本文に **async.SetTimeout** への呼び出しのみを含めると、タイムアウトは呼び出しごとにトリガーされます。

AsyncResponse.NO_TIMEOUT の値は無限のタイムアウトを表します。

デフォルトのタイムアウト動作

デフォルトでは、呼び出しタイムアウトがトリガーされると、JAX-RS ランタイムが **ServiceUnavailableException** 例外を発生させ、ステータス **503** で HTTP エラーの応答を返します。

TimeoutHandler インターフェイス

タイムアウトの動作をカスタマイズする場合は、**TimeoutHandler** インターフェイスを実装してタイムアウトハンドラーを定義する必要があります。

```
// Java
package javax.ws.rs.container;

public interface TimeoutHandler {
    public void handleTimeout(AsyncResponse asyncResponse);
}
```

実装クラスで **handleTimeout** メソッドを上書きする場合は、タイムアウトを処理する次の方法のいずれかを選択できます。

- **asyncResponse.cancel** メソッドを呼び出すことで、レスポンスを取り消します。
- レスポンス値で **asyncResponse.resume** メソッドを呼び出すことで、レスポンスを送信します。
- **asyncResponse.setTimeout** メソッドを呼び出すことで、待機期間を延長します。たとえば、さらに 10 秒間待つには、 **asyncResponse.setTimeout(10, TimeUnit.SECONDS)** を呼び出しできます。

ハンドラーでタイムアウトを設定する例

タイムアウトハンドラーで呼び出しタイムアウトを定義するには、以下の例のように **AsyncResponse** オブジェクトの **setTimeout** メソッドと **setTimeoutHandler** メソッドの両方を呼び出します。

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

この例では、呼び出しタイムアウトを処理するために **CancelTimeoutHandlerImpl** タイムアウトハンドラーのインスタンスを登録します。

タイムアウトハンドラーを使用した応答の取り消し

CancelTimeoutHandlerImpl タイムアウトハンドラーは以下のように定義されます。

```
// 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();
        }

    }
    ...
}
```

AsyncResponse オブジェクト上で **cancel** を呼び出す効果は、クライアントに HTTP 503 (**Service unavailable**) エラーの応答を送信することです。任意で、**cancel** メソッド (**int** または **java.util.Date** の値) の引数を指定できます。これは応答メッセージで **Retry-After:** HTTP ヘッダーを設定するために使用されます。ただし、クライアントは多くの場合で **Retry-After:** ヘッダーを無視します。

Runnable インスタンスでの取り消し済みの応答への対応

エグゼキュータースレッドプールで処理のためにキューに格納された **Runnable** インスタンスとして、一時停止されたリクエストをカプセル化した場合、スレッドプールがリクエストを処理するまでに

AsyncResponse がキャンセルされる可能性があります。このため、**Runnable** インスタンスにコードを追加する必要があります。これにより、キャンセルされた **AsyncResponse** オブジェクトに対応できるようになります。以下に例を示します。

```
// 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. 切断された接続の処理

概要

クライアント接続が失われた場合に対処するためにコールバックを追加することが可能です。

ConnectionCallback インターフェイス

切断されたコネクションのコールバックを追加するには、以下のように **javax.ws.rs.container.ConnectionCallback** インターフェイスを実装する必要があります。

```
// Java
package javax.ws.rs.container;

public interface ConnectionCallback {
    public void onDisconnect(AsyncResponse disconnected);
}
```

接続コールバックの登録

接続コールバックを実装したら、**register** メソッドの1つを呼び出して、現在の **AsyncResponse** オブジェクトに登録する必要があります。たとえば、型 **MyConnectionCallback** の接続コールバックを登録するには、次のコマンドを実行します。

```
asyncResponse.register(new MyConnectionCallback());
```

接続コールバックの一般的なシナリオ

通常、接続コールバックを実装する主な理由は、切断されたクライアントコネクション (解放が必要なリソースを識別するためのキーとして **AsyncResponse** インスタンスを使える) に関連付けられたリソースを解放することです。

48.5.4. コールバックの登録

概要

オプションで、呼び出しが完了したときに通知されるように、**AsyncResponse** インスタンスにコールバックを追加できます。このコールバックの呼び出しが可能な場合の処理には、代替えポイントが 2 つあります。

- 要求の処理が完了し、応答がすでにクライアントに送信された後。
- リクエスト処理を終了して、マッピングされていない **Throwable** がホスト I/O コンテナーに伝播された後。

CompletionCallback インターフェイス

完了コールバックを追加するには、以下のように定義された **javax.ws.rs.container.CompletionCallback** インターフェイスを実装する必要があります。

```
// Java
package javax.ws.rs.container;

public interface CompletionCallback {
    public void onComplete(Throwable throwable);
}
```

通常、**throwable** 引数は **null** です。ただし、リクエスト処理によってマッピングされていない例外が発生した場合は、**throwable** にはマッピングされていない例外インスタンスが含まれます。

完了コールバックの登録

完了コールバックを実装したら、**register** メソッドの 1 つを呼び出して、現在の **AsyncResponse** オブジェクトに登録する必要があります。たとえば、型 **MyCompletionCallback** の完了コールバックを登録するには、次のコマンドを実行します。

```
asyncResponse.register(new MyCompletionCallback());
```

第49章 JAX-RS 2.0 CLIENT API

概要

JAX-RS 2.0 は、REST 呼び出しや HTTP クライアント呼び出しを行うために使用できるフル機能のクライアント API を定義します。これには、Fluent API(要求の構築を簡素化するための)、メッセージを解析するフレームワーク (**エンティティープロバイダー**と呼ばれるタイプのプラグイン)、クライアント側での非同期呼び出しのサポートが含まれます。

49.1. JAX-RS 2.0 クライアント API の概要

概要

JAX-RS 2.0 は JAX-RS クライアントの Fluent API を定義し、HTTP 要求をステップバイステップで構築してから、適切な HTTP 動詞 (GET、POST、PUT または DELETE) を使用して要求を呼び出すことができます。

注記

Blueprint XML または Spring XML で JAX-RS クライアントを定義することもできます (**jaxrs:client** 要素を使用)。このアプローチの詳細は、「JAX-RS クライアントエンドポイントの設定」 を参照してください。

依存関係

アプリケーションで JAX-RS 2.0 クライアント API を使用するには、以下の Maven 依存関係をプロジェクトの **pom.xml** ファイルに追加する必要があります。

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-client</artifactId>
  <version>3.3.6.fuse-7_13_0-00005-redhat-00001</version>
</dependency>
```

非同期呼び出し機能を使用する予定 (「クライアントの非同期処理」を参照) を使用する場合は、以下の Maven 依存関係も必要になります。

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-hc</artifactId>
  <version>3.3.6.fuse-7_13_0-00005-redhat-00001</version>
</dependency>
```

クライアント API パッケージ

JAX-RS 2.0 クライアントインターフェイスとクラスは、以下の Java パッケージにあります。

```
javax.ws.rs.client
```

JAX-RS 2.0 Java クライアントを開発する場合には、通常はコアパッケージからクラスにアクセスする必要もあります。

```
javax.ws.rs.core
```

シンプルなクライアント要求の例

次のコードフラグメントは、JAX-RS 2.0 クライアント API を使用して **http://example.org/bookstore** JAX-RS サービスで呼び出しを行い、GET HTTP メソッドで呼び出す簡単な例を示しています。

```
// 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 クライアント API は **Fluent API**(ドメイン固有言語と呼ばれることもあります) として設計されています。Fluent API では、Java メソッドが単純な言語のコマンドであるかのように、Java メソッドのチェーンが単一のステートメントで呼び出されます。JAX-RS 2.0 では、Fluent API を使用して REST 要求を作成して呼び出します。

REST 呼び出しを実行する手順

JAX-RS 2.0 クライアント API を使用すると、クライアント呼び出しは以下のような一連のステップでビルドされ、呼び出されます。

1. クライアントをブートストラップします。
2. ターゲットを設定します。
3. 呼び出しを構築して実行します。
4. 応答を解析します。

クライアントのブートストラップ

最初のステップでは、**javax.ws.rs.client.Client** オブジェクトを作成し、クライアントをブートストラップします。この **Client** インスタンスは比較的重いオブジェクトで、JAX-RS クライアント (場合によってはインターセプターや追加の CXF 機能を含む) をサポートするために必要なテクノロジーのスタックを表します。理想的には、新規オブジェクトを作成する代わりに、可能な場合はクライアントオブジェクトを再利用する必要があります。

新しい **Client** オブジェクトを作成するには、以下のように **ClientBuilder** クラスで静的メソッドを呼び出します。

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
...
Client client = ClientBuilder.newClient();
...
```

ターゲットの設定

ターゲットを設定すると、REST 呼び出しに使用される URI を効果的に定義します。次の例は、**path(String)** メソッドを使用して、ベース URI **base** を定義し、ベース URI にパスセグメントを追加する方法を示しています。

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
```

呼び出しの構築および実行

これは実際には 2 つのステップを 1 つにまとめたものです。最初に、HTTP 要求 (ヘッダー、受け入れられたメディアタイプなどを含む) を構築し、次に、関連する HTTP メソッドを呼び出します (必要に応じて、オプションで要求メッセージ本文を提供します)。

たとえば、**application/xml** メディア型を受け入れるリクエストを作成および呼び出すには、以下を実行します。

```
// Java
import javax.ws.rs.core.Response;
...
Response resp = books.resolveTemplate("id", "123").request("application/xml").get();
```

応答を解析します

最後に、前のステップで取得したレスポンス **resp** を解析する必要があります。通常、レスポンスは **javax.ws.rs.core.Response** オブジェクトの形式で返されます。これは、HTTP ヘッダーと、他の HTTP メタデータおよび HTTP メッセージボディー (ある場合) ををカプセル化します。

String 形式で返された HTTP メッセージにアクセスする場合、以下のように **String.class** 引数を指定して **readEntity** メソッドを呼び出して、簡単にアクセスできます。

```
// Java
...
String msg = resp.readEntity(String.class);
```

String.class を **readEntity** の引数として指定すると、レスポンスのメッセージボディーに **String** として常にアクセスできます。メッセージボディーを一般的に変換する場合には、**エンティティープロバイダー** を指定して変換を実行できます。詳細は、「要求と応答の解析」 を参照してください。

49.2. クライアントターゲットの構築

概要

最初の **Client** インスタンスの作成後、次のステップはリクエスト URI を構築することです。**WebTarget** ビルダークラスを使用すると、URI パスやクエリーパラメーターを含む URI のすべての側面を設定できます。

WebTarget ビルダークラス

javax.ws.rs.client.WebTarget ビルダークラスは、リクエストの REST URI を構築できるようにする Fluent API の一部を提供します。

クライアントターゲットを作成する

WebTarget インスタンスを作成するには、**javax.ws.rs.client.Client** インスタンス上で **target** メソッドのいずれかを呼び出します。以下に例を示します。

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
```

ベースパスおよびパスセグメント

target メソッドを使用して、完全なパスをすべて一度に指定できます。または、基本パスを指定してから、**target** メソッドと **path** メソッドの組み合わせを使用して、パスセグメントを 1 つずつ追加することもできます。ベースパスとパスセグメントを組み合わせる利点は、**WebTarget** ベースパスオブジェクトを、わずかに異なるターゲット上で複数の呼び出しに対して簡単に再使用できることです。以下に例を示します。

```
// 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 テンプレートパラメーター

ターゲットパスの構文も URI テンプレートパラメーターをサポートします。つまり、パスセグメントはテンプレートパラメーター **{param}** で初期化でき、その後に指定値に解決されます。以下に例を示します。

```
// 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();
```

ここで、**resolveTemplate** メソッドは、パスセグメント **{id}** を値 **123** に置き換えます。

クエリーパラメーターの定義

クエリーパラメーターは URI パスに追加できます。ここでは、URI パスでは、クエリーパラメーターの先頭に 1 つの **?** 文字が付けられます。このメカニズムでは、構文 **?name1=value1&name2=value2&…** を使用して一連の名前/値のペアを設定することができます。

WebTarget インスタンスでは、以下のように **queryParam** メソッドを使用してクエリーパラメーターを定義できます。

```
// Java
WebTarget target = client.target("http://example.org/bookstore/")
                        .queryParam("userId","Agamemnon")
                        .queryParam("lang","gr");
```

マトリックスパラメーターの定義

マトリックスパラメーターはクエリーパラメーターと若干似ていますが、広くサポートされておらず、異なる構文は使用します。**WebTarget** インスタンスでマトリックス パラメーターを定義するには、**matrixParam(String, Object)** メソッドを呼び出します。

49.3. クライアント呼び出しのビルド

概要

WebTarget ビルダークラスを使用してターゲット URI を構築した後、次の手順では HTTP ヘッダーやクッキーなどの他のものを設定します。これは、**Invocation.Builder** クラスを使用します。呼び出しの構築における最後のステップは、適切な HTTP 動詞 (GET、POST、PUT、または DELETE) を呼び出して、必要に応じてメッセージボディーを指定します。

invocation.Builder クラス

javax.ws.rs.client.Invocation.Builder ビルダークラスは、Fluent API の一部を提供し、HTTP メッセージの内容を構築し、HTTP メソッドを呼び出すことができます。

呼び出しビルダーを作成します。

Invocation.Builder インスタンスを作成するには、**javax.ws.rs.client.WebTarget** インスタンス上で **request** メソッドのいずれかを呼び出します。以下に例を示します。

```
// 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();
```

HTTP ヘッダーの定義

以下のように、**header** メソッドを使用してリクエストメッセージに HTTP ヘッダーを追加できます。

```
Invocation.Builder invheader = invbuilder.header("From", "fionn@example.org");
```

クッキーの定義

以下のように、**cookie** メソッドを使用してクッキーをリクエストメッセージに追加できます。

```
Invocation.Builder invcookie = invbuilder.cookie("myrestclient", "123xyz");
```

プロパティーの定義

以下のように、プロパティーメソッドを使用して、このリクエストのコンテキストでプロパティーを設定できます。

```
Invocation.Builder invproperty = invbuilder.property("Name", "Value");
```

許可されるメディアタイプ、言語、またはエンコーディングの定義

以下のように、受け入れられるメディアタイプ、言語、またはエンコードを定義できます。

```
Invocation.Builder invmedia = invbuilder.accept("application/xml")
                                        .acceptLanguage("en-US")
                                        .acceptEncoding("gzip");
```

HTTP メソッドを呼び出します。

REST 呼び出しを構築するプロセスは、HTTP 呼び出しを実行する HTTP メソッドを呼び出して終了します。以下のメソッド (**javax.ws.rs.client.SyncInvoker** ベースクラスから継承) を呼び出すことができます。

```
get
post
delete
put
head
trace
options
```

呼び出す特定の HTTP 動詞がこのリストにない場合は、汎用 **method** メソッドを使用して HTTP メソッドを呼び出すことができます。

タイプが指定された応答

すべての HTTP 呼び出しメソッドには、型なしのバリアントと型付きバリアント (追加の引数を取る) が用意されています。デフォルトの **get()** メソッド (引数を取らない) を使用してリクエストを呼び出すと、**javax.ws.rs.core.Response** オブジェクトがその呼び出しから返されます。以下に例を示します。

```
Response res = client.target("http://example.org/bookstore/books/123")
                     .request("application/xml").get();
```

ただし、**get(Class<T>)** メソッドを使用して、レスポンスを特定の型として返すように要求することもできます。たとえば、リクエストを呼び出して、**BookInfo** オブジェクトとしてレスポンスを返すように要求するには、次のようにします。

```
BookInfo res = client.target("http://example.org/bookstore/books/123")
                     .request("application/xml").get(BookInfo.class);
```

ただし、これを機能させるには、**application/xml** レスポンス形式をマッピングできる **Client** インスタンスで、適切な**エンティティープロバイダー** をリクエストされた型に登録する必要があります。エンティティープロバイダーの詳細は、「要求と応答の解析」 を参照してください。

送信メッセージの POST または PUT の指定

HTTP メソッド (POST または PUT など) の要求にメッセージボディーが含まれる場合に、そのメッセージボディーをメソッドの最初の引数として指定する必要があります。メッセージボディーは **javax.ws.rs.client.Entity** オブジェクトとして指定する必要があります。ここで、**Entity** はメッセージの内容とそれに関連するメディア型をカプセル化します。たとえば、メッセージの内容が **String** 型として提供される POST メソッドを呼び出すには、以下のようにします。

```
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"));
```

必要に応じて、**Entity.entity()** コンストラクターメソッドが、登録済みエンティティープロバイダーを使用して、提供されたメッセージインスタンスを自動的に指定のメディア型にマップします。メッセージボディーは、常に単純な **String** 型として指定できます。

遅延呼び出し

すぐに HTTP リクエストを呼び出す代わりに (**get()** メソッドを呼び出すなど)、後で呼び出すことのできる **javax.ws.rs.client.Invocation** オブジェクトを作成するオプションがあります。**Invocation** オブジェクトは、HTTP メソッドを含む、保留中の呼び出しの詳細を **すべて** カプセル化します。

以下のメソッドは、**Invocation** オブジェクトの構築に使用できます。

```
buildGet
buildPost
buildDelete
buildPut
build
```

たとえば、GET **Invocation** オブジェクトを作成し、後で呼び出すには、以下のようなコードを使用することができます。

```
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();
```

非同期呼び出し

JAX-RS 2.0 クライアント API は、クライアント側での非同期呼び出しをサポートします。非同期呼び出しを行うには、**request()** の後のメソッドのチェーンで **async()** メソッドを呼び出します。以下に例を示します。

```
Future<Response> res = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get();
```

非同期呼び出しを行うと、戻り値は **java.util.concurrent.Future** オブジェクトになります。非同期呼び出しの詳細は、「クライアントの非同期処理」 を参照してください。

49.4. 要求と応答の解析

概要

HTTP 呼び出しを行う際の重要な側面は、クライアントが送信要求メッセージと受信応答を解析できる必要があることです。JAX-RS 2.0 では、主要な概念は、メディアタイプでタグ付けされた raw メッセージを表す **Entity** クラスです。未加工メッセージを解析するために、メディアタイプと特定の Java タイプとの間で変換できる複数の **エンティティープロバイダー** を登録できます。

つまり、JAX-RS 2.0 のコンテキストでは、**Entity** は raw メッセージの表現で、エンティティープロバイダーは (メディア型に基づく) raw メッセージを解析する機能を提供するプラグインです。

Entities

Entity は、メタデータ (メディア型、言語、およびエンコーディング) によって拡張されたメッセージボディーです。**Entity** インスタンスはメッセージを raw 形式で保持し、特定のメディア型に関連付けられます。**Entity** オブジェクトのコンテンツを、 Java オブジェクトに変換するには、指定のメディア型を必要な Java 型にマッピングできる **エンティティープロバイダー** が必要です。

バリアント

javax.ws.rs.core.Variant オブジェクトは、以下のように **Entity** に関連付けられたメタデータをカプセル化します。

- メディアタイプ
- 言語
- エンコーディング

実質的に、**Entity** は HTTP メッセージの内容で設定され、 **Variant** メタデータによって拡張されると考えることができます。

エンティティープロバイダー

エンティティープロバイダーは、メディアタイプと Java 型間のマッピング機能を提供するクラスです。実際には、エンティティープロバイダーは、特定のメディアタイプ (または複数のメディアタイプ) のメッセージを解析できるクラスと考えることができます。エンティティープロバイダーには、以下の 2 つの異なるバリアントがあります。

MessageBodyReader
: メディアタイプから Java 型へのマッピング機能を提供します。

MessageBodyWriter
: Java 型からメディアタイプへのマッピング機能を提供します。

標準のエンティティープロバイダー

次の Java とメディアタイプの組み合わせのエンティティープロバイダーが標準として提供されています。

byte[]
: すべてのメディア型 (*/*)

java.lang.String
: すべてのメディア型 (*/*)

java.io.InputStream
: すべてのメディア型 (*/*)

java.io.Reader
: すべてのメディア型 (*/*)

java.io.File
: すべてのメディア型 (*/*)

javax.activation.DataSource
: すべてのメディア型 (*/*)

javax.xml.transform.Source
: XML 型 (**text/xml**、**application/xml**、および **application/*+xml** フォームのメディア型)。

javax.xml.bind.JAXBElement およびアプリケーションが提供する JAXB クラス
: XML 型 (**text/xml**、**application/xml**、および **application/*+xml** フォームのメディア型)。

MultivaluedMap<String,String>
: フォームコンテンツ (**application/x-www-form-urlencoded**)。

StreamingOutput
: すべてのメディア型 (*/*)、**MessageBodyWriter** のみ。

java.lang.Boolean、java.lang.Character、java.lang.Number
: **text/plain** のみ対象。ボックス/アンボックス変換でサポートされるプリミティブタイプ。

Response オブジェクト

デフォルトの戻り値型は **javax.ws.rs.core.Response** 型で、型付けされていない応答を表します。**Response** オブジェクトは、メッセージボディー、HTTP ステータス、HTTP ヘッダー、メディア型など、完全な HTTP 応答へのアクセスを提供します。

応答ステータスへのアクセス

応答の状態には、**getStatus** メソッド (HTTP ステータスコードを返す) からアクセスできます。

```
int status = resp.getStatus();
```

または、説明文字列も提供する **getStatusInfo** メソッドでアクセスすることもできます。

```
String statusReason = resp.getStatusInfo().getReasonPhrase();
```

返されるヘッダーへのアクセス

以下の方法のいずれかを使用して HTTP ヘッダーにアクセスできます。

```
MultivaluedMap<String,Object>
getHeaders()

MultivaluedMap<String,String>
getStringHeaders()

String
getHeaderString(String name)
```

たとえば、**Response** に **Date** ヘッダーがあることが分かっている場合は、以下のようにアクセスできます。

```
String dateAsString = resp.getHeaderString("Date");
```

返されるクッキーへのアクセス

以下のように、**getCookies** メソッドを使用して、**Response** で設定された新しいクッキーにアクセスできます。

```
import javax.ws.rs.core.NewCookie;
...
java.util.Map<String,NewCookie> cookieMap = resp.getCookies();
java.util.Collection<NewCookie> cookieCollection = cookieMap.values();
```

返されるメッセージコンテンツへのアクセス

Response オブジェクトの **readEntity** メソッドのいずれかを呼び出すと、返されるメッセージコンテンツにアクセスできます。**readEntity** メソッドは利用可能なエンティティープロバイダーを自動的に呼び出し、メッセージを要求された型 (**readEntity** の最初の引数として指定) に変換します。たとえば、メッセージコンテンツに **String** 型としてアクセスするには、次のように指定します。

```
String messageBody = resp.readEntity(String.class);
```

コレクションの戻り値

返されたメッセージを Java 汎用型 (例: **List** または **Collection** 型) としてアクセスする必要がある場合、**javax.ws.rs.core.GenericType<T>** コンストラクションを使用してリクエストメッセージ型を指定できます。以下に例を示します。

```
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. クライアントエンドポイントの設定

概要

機能およびプロバイダーの登録および設定により、ベース **javax.ws.rs.client.Client** オブジェクトの機能を拡張できます。

例

以下の例は、ロギング機能、カスタムエンティティープロバイダー、および **prettyLogging** プロパティーを **true** に設定するよう設定されたクライアントを示しています。

```
// 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");
```

オブジェクトを登録する設定可能な API

Client クラスはオブジェクトの登録用に **Configurable** API をサポートします。これにより、**register** メソッドの複数のバリアントが提供されます。ほとんどの場合に、以下の例のようにクラスまたはオブジェクトインスタンスを登録します。

```
client.register(LoggingFeature.class)
client.register(new LoggingFeature())
```

register バリアントの詳細は、**Configurable** に関する参考ドキュメントを参照してください。

クライアントで設定できる内容

クライアントエンドポイントの以下の機能を設定できます。

- 機能
- プロバイダー
- プロパティー
- Filters
- インターセプター

機能

javax.ws.rs.core.Feature は事実上、追加の機能を JAX-RS クライアントに追加するプラグインです。多くの場合に、機能は、必要な機能を提供するためのインターセプターを 1 つ以上インストールします。

プロバイダー

プロバイダーは、マッピング機能を提供する特定の種類のクライアントプラグインです。JAX-RS 2.0 仕様は、以下のようなプロバイダーを定義します。

エンティティープロバイダー

エンティティープロバイダー は、特定のメディアタイプ間のマッピング機能を提供します。詳細は、「要求と応答の解析」 を参照してください。

例外マッピングプロバイダー

例外マッピングプロバイダー は、チェックされたランタイム例外を **Response** のインスタンスにマッピングします。

コンテキストプロバイダー

コンテキストプロバイダーはサーバー側で使用され、リソースクラスや他のサービスプロバイダーのコンテキストを提供します。

Filters

JAX-RS 2.0 フィルターは、メッセージ処理パイプラインのさまざまなポイント (拡張ポイント) の URI、ヘッダー、およびその他のコンテキストデータへのアクセスを可能にするプラグインです。詳細は、61章*JAX-RS 2.0 フィルターおよびインターセプター* を参照してください。

インターセプター

JAX-RS 2.0 インターセプターは、読み取りまたは書き込み時に要求または応答のメッセージボディーへのアクセスを許可するプラグインです。詳細は、61章*JAX-RS 2.0 フィルターおよびインターセプター* を参照してください。

プロパティー

クライアントに 1 つ以上のプロパティーを設定すると、登録された機能または登録されたプロバイダーの設定をカスタマイズできます。

その他の設定可能なタイプ

javax.ws.rs.client.Client (および **javax.ws.rs.client.ClientBuilder**) オブジェクトだけでなく、**WebTarget** オブジェクトも設定することができます。**WebTarget** オブジェクトの設定を変更すると、基盤のクライアント設定はディープコピーされ、新しい **WebTarget** 設定が指定されます。そのため、元の **Client** オブジェクトの設定を変更せずに、**WebTarget** オブジェクトの設定を変更することができます。

49.6. クライアントの非同期処理

概要

JAX-RS 2.0 は、クライアント側での呼び出しの非同期処理をサポートします。非同期処理の 2 種類のスタイルがサポートされます。これは、**java.util.concurrent.Future<V>** 戻り値を使用するか、呼び出しコールバックを登録します。

Future<V> の戻り値での非同期呼び出し

Future<V> アプローチを非同期処理に使用する場合は、以下のようにクライアント要求を非同期的に呼び出すことができます。

```
// 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();
```

型付きの応答と同様の方法を使用できます。たとえば、**BookInfo** 型のレスポンスを取得するには、以下を指定します。

```
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();
```

呼び出しコールバックを使用した非同期呼び出し

Future<V> オブジェクトを使用して戻り値にアクセスする代わりに、以下のように呼び出しコールバック (**javax.ws.rs.client.InvocationCallback<RESPONSE>** を使用) を定義できます。

```
// 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();
```

```
        }
    });
...
```

型付きの応答と同様の方法を使用できます。

```
// 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 例外処理

概要

可能な場合に、リソースメソッドがキャッチした例外をトリガーとして、有用なエラーが要求元コンシューマーに返されるはずです。JAX-RS リソースメソッドは WebApplicationException 例外を出力できます。例外を適切な応答にマッピングするために ExceptionMapper<E> 実装を指定することもできます。

50.1. JAX-RS 例外クラスの概要

概要

JAX-RS 1.x では、利用可能な例外クラスは **WebApplicationException** のみです。しかし、JAX-WS 2.0 以降、多くの JAX-RS 例外クラスが追加で定義されています。

JAX-RS ランタイムレベルの例外

以下の例外は、JAX-RS ランタイムでだけ出力されるはずです (つまり、これらの例外はアプリケーションレベルのコードからは**出力しない**でください)。

ProcessingException

(**JAX-RS 2.0 のみ**) リクエストの処理中または JAX-RS ランタイムでの応答処理中に **javax.ws.rs.ProcessingException** を出力できます。たとえば、フィルターチェーンまたはインターセプターチェーン処理のエラーが原因で、このエラーが出力される可能性があります。

ResponseProcessingException

(**JAX-RS 2.0 のみ**) **javax.ws.rs.client.ResponseProcessingException** は **ProcessingException** のサブクラスであり、**クライアント側** の JAX-RS ランタイムでエラーが発生したときに出力される可能性があります。

JAX-RS アプリケーションレベルの例外

以下の例外は、アプリケーションレベルのコードで出力 (およびキャッチ) されるように設計されています。

WebApplicationException

javax.ws.rs.WebApplicationException は、汎用アプリケーションレベルの JAX-RS 例外で、サーバー側のアプリケーションコードで出力できます。この例外タイプは、HTTP ステータスコード、エラーメッセージ、および応答メッセージをカプセル化 (任意) できます。詳細は、「WebApplicationException 例外を使用したレポート」 を参照してください。

ClientErrorException

(**JAX-RS 2.0 のみ**) **javax.ws.rs.ClientErrorException** 例外クラスは **WebApplicationException** から継承し、HTTP **4xx** ステータスコードをカプセル化するために使用されます。

ServerErrorException

(**JAX-RS 2.0 のみ**) **javax.ws.rs.ServerErrorException** 例外クラスは **WebApplicationException** から継承し、HTTP **5xx** ステータスコードをカプセル化するために使用されます。

RedirectionException

(**JAX-RS 2.0 のみ**) **javax.ws.rs.RedirectionException** 例外クラスは **WebApplicationException** から継承し、HTTP **3xx** ステータスコードをカプセル化するために使用されます。

50.2. WEBAPPLICATIONEXCEPTION 例外を使用したレポート

errors
indexterm:[WebApplicationException]

概要

JAX-RS API には WebApplicationException ランタイム例外が導入され、リソースメソッドが RESTful クライアントが消費するのに適した例外を簡単に作成できるようになりました。
WebApplicationException 例外には、リクエスト元に返されるエンティティーボディーを定義する **Response** オブジェクトを含めることができます。また、エンティティーボディーが指定されていない場合に、クライアントに HTTP ステータスコードを指定するメカニズムも提供します。

簡単な例外の作成

WebApplicationException 例外を作成する最も簡単な方法として、no の引数コンストラクターまたは WebApplicationException 例外で元の例外をラップするコンストラクターのいずれかを使用します。どちらのコンストラクターも、空の応答で WebApplicationException を作成します。

これらのコンストラクターのいずれかによって作成された例外が出力されると、ランタイムはエンティティーボディーが空でステータスコードが **500 Server Error** のレスポンスを返します。

クライアントに返されるステータスコードの設定

500 以外のエラーコードを返す場合は、ステータスを指定できるようにする 4 つの WebApplicationException コンストラクターのいずれかを使用できます。例50.1「ステータスコードでの WebApplicationException の作成」 に示されているこれらのコンストラクターの 2 つを整数として返します。

例50.1 ステータスコードでの WebApplicationException の作成

WebApplicationExceptionintstatusWebApplicationExceptionjava.lang.Throwablecauseintstatus

例50.2「ステータスコードでの WebApplicationException の作成」 に記載されている他の 2 つは、レスポンスステータスを **Response.Status** のインスタンスとしてし取ります。

例50.2 ステータスコードでの WebApplicationException の作成

WebApplicationExceptionjavax.ws.rs.core.Response.StatusstatusWebApplicationExceptionjava.lang.Throwablecausejavax.ws.rs.core.Response.Statusstatus

これらのコンストラクターのいずれかによって作成された例外が出力されると、ランタイムはエンティティーボディーが空で指定したステータスコードがレスポンスを返します。

エンティティーボディーの指定

例外とともにメッセージを送信する場合は、**Response** オブジェクトを取る WebApplicationException コンストラクターのいずれかを使用することができます。ランタイムは、**Response** オブジェクトを使用してクライアントに送信される応答を作成します。応答に保存されているエンティティーはメッセー

ジのエンティティーボディーに、応答のステータスフィールドはメッセージの HTTP ステータスにマップされます。

例50.3「例外を含めたメッセージの送信」 は、例外の理由を含むテキストメッセージをクライアントに返すコードを示しています。HTTP メッセージのステータスは **409 Conflict** に設定されます。

例50.3 例外を含めたメッセージの送信

```
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);
```

汎用例外の拡張

WebApplicationException 例外を拡張できます。これにより、カスタム例外を作成して、一部のボイラープレートコードを削除できるようになります。

例50.4「Extending WebApplicationException」 では、例50.3「例外を含めたメッセージの送信」 にコードと同様の応答を作成する新しい例外を紹介しています。

例50.4 Extending 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 例外タイプ

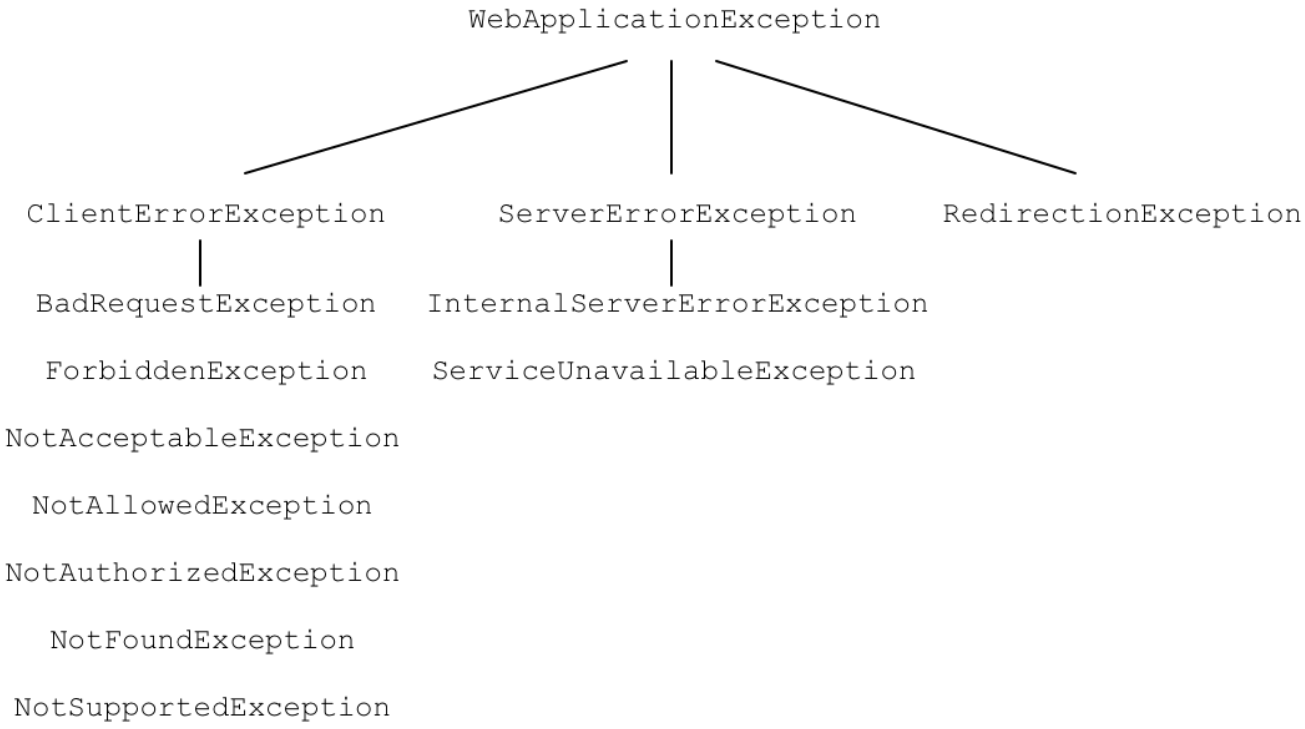
概要

JAX-RS 2.0 では、アプリケーションコードで出力 (およびキャッチ) できる特定の HTTP 例外型が導入されています (既存の **WebApplicationException** 例外型に加えて)。これらの例外型は、HTTP クライアントエラー (HTTP **4xx** ステータスコード) または HTTP サーバーエラー (HTTP **5xx** ステータスコード) のいずれかに対して、標準の HTTP ステータスコードをラッピングするために使用できます。

例外の階層

図50.1「JAX-RS 2.0 Application Exception Hierarchy」 では、JAX-RS 2.0 でサポートされるアプリケーションレベルの例外の階層を紹介しています。

図50.1 JAX-RS 2.0 Application Exception Hierarchy



WebApplicationException クラス

javax.ws.rs.WebApplicationException 例外クラス (JAX-RS 1.x 以降で使用可能) は JAX-RS 2.0 例外階層のベースにあり、「WebApplicationException 例外を使用したレポート」で詳細に説明されています。

ClientErrorException クラス

javax.ws.rs.ClientErrorException 例外クラスは、HTTP クライアントエラー (HTTP **4xx** ステータスコード) をカプセル化するために使用されます。アプリケーションコードでは、この例外またはサブクラスの 1 つを出力できます。

ServerErrorException クラス

javax.ws.rs.ServerErrorException 例外クラスは、HTTP サーバーエラー (HTTP **5xx** ステータスコード) をカプセル化するために使用されます。アプリケーションコードでは、この例外またはサブクラスの 1 つを出力できます。

RedirectionException クラス

javax.ws.rs.RedirectionException 例外クラスは、HTTP リクエストのリダイレクト (HTTP **3xx** ステータスコード) をカプセル化するために使用されます。このクラスのコンストラクターは、リダイレクト先を指定する URI 引数を使用できます。リダイレクト URI は **getLocation()** メソッドからアクセスできます。通常、HTTP リダイレクトはクライアント側では透過的です。

クライアント例外サブクラス

JAX-RS 2.0 アプリケーションで、以下の HTTP クライアント例外 (HTTP **4xx** ステータスコード) を発生させることができます。

BadRequestException

400 Bad Request HTTP エラーステータスをカプセル化します。

ForbiddenException

403 Forbidden HTTP エラーステータスをカプセル化します。

NotAcceptableException

406 Not Acceptable HTTP エラーステータスをカプセル化します。

NotAllowedException

405 Method Not Allowed HTTP エラーステータスをカプセル化します。

NotAuthorizedException

401 Unauthorized HTTP エラーステータスをカプセル化します。この例外は、以下のいずれかの場合に発生する可能性があります。

- クライアントが (HTTP **Authorization** ヘッダーで) 必要なクレデンシャルを送信しなかった。
- クライアントはクレデンシャルを提示したが、認証情報が有効ではなかった。

NotFoundException

404 Not Found HTTP エラーステータスをカプセル化します。

NotSupportedException

415 Unsupported Media Type HTTP エラーステータスをカプセル化します。

サーバー例外サブクラス

JAX-RS 2.0 アプリケーションで、以下の HTTP サーバー例外 (HTTP **5xx** ステータスコード) を発生させることができます。

InternalServerErrorException

500 Internal Server Error HTTP エラーステータスをカプセル化します。

ServiceUnavailableException

503 Service Unavailable HTTP エラーステータスをカプセル化します。

50.4. 例外の応答へのマッピング

概要

WebApplicationException 例外の出力が現実的でない場合や、不可能な場合があります。たとえば、可能な例外をすべてキャッチし、そのような例外に対して WebApplicationException を作成する必要がない場合があります。また、アプリケーションコードの操作を容易にするカスタムの例外を使用することもできます。

このようなケースを処理するために、JAX-RS API では、クライアントに送信する **Response** オブジェクトを生成するカスタム例外プロバイダーを実装できます。カスタム例外プロバイダーは、ExceptionMapper<E> インターフェイスを実装して作成されます。Apache CXF ランタイムで登録されると、型 **E** の例外が出力されるたびにカスタムプロバイダーが使用されます。

例外マッパーの選択方法

例外マッパーは 2 つのケースで使用されます。

- 例外またはそのサブクラスの 1 つが出力されると、ランタイムは適切な例外マッパーをチェックします。特定の例外出力を処理する場合は、例外マッパーが選択されます。出力された特定の例外の例外マッパーがない場合には、例外の中で最も近いスーパークラスの例外マッパーが選択されます。

- デフォルトでは、WebApplicationException はデフォルトマッパー **WebApplicationExceptionMapper** によって処理されます。WebApplicationException 例外を処理する可能性がある追加のカスタムマッパーが登録されていても (カスタム **RuntimeException** マッパーなど)、カスタムマッパーは **使用されず**、代わりに **WebApplicationExceptionMapper** が使用されます。
  ただし、この動作は **Message** オブジェクトの **default.wae.mapper.least.specific** プロパティーを **true** に設定して変更できます。このオプションを有効にすると、カスタム例外マッパーを使用して WebApplicationException 例外を処理できるようにするため、デフォルトの **WebApplicationExceptionMapper** の優先順位は最低になります。たとえば、このオプションが有効な場合に、カスタム **RuntimeException** マッパーを登録することで WebApplicationException 例外をキャッチできます。「WebApplicationException の例外マッパーの登録」を参照してください。

例外に、例外マッパーがない場合には、この例外は ServletException 例外でラップされ、コンテナーランタイムに渡されます。その後、コンテナーランタイムは例外の処理方法を決定します。

例外マッパーの実装

例外マッパーは、javax.ws.rs.ext.ExceptionMapper<E> インターフェイスを実装して作成されます。例 50.5「例外マッパーのインターフェイス」で示されているように、インターフェイスには単一のメソッド **toResponse()** があります。これは元の例外をパラメーターとして取り、**Response** オブジェクトを返します。

例50.5 例外マッパーのインターフェイス

```
public interface ExceptionMapper<E extends java.lang.Throwable>
{
  public Response toResponse(E exception);
}
```

例外マッパーによって作成された **Response** オブジェクトは、他の **Response** オブジェクトのようにランタイムによって処理されます。コンシューマーへの結果となる応答には、**Response** オブジェクトにカプセル化されたステータス、ヘッダー、およびエンティティーボディーが含まれます。

例外マッパー実装はランタイムによってプロバイダーとみなされます。そのため、**@Provider** アノテーションを付ける必要があります。

例外マッパーによる **Response** オブジェクトの構築中に例外が発生すると、ランタイムは **500 Server Error** のステータスで応答をコンシューマーに送信します。

例50.6「例外の応答へのマッピング」は、Spring AccessDeniedException 例外を遮断し、 **403 Forbidden** ステータスと空のエンティティーボディーで応答を生成する例外マッパーを示しています。

例50.6 例外の応答へのマッピング

```
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();
  }

}
```

ランタイムは AccessDeniedException 例外をキャッチし、空のエンティティーボディーと **403** のステータスで **Response** オブジェクトを作成します。ランタイムは、通常のレスポンスの場合と同様に **Response** オブジェクトを処理します。その結果、コンシューマーはステータスが **403** の HTTP 応答を受信します。

例外マッパーの登録

JAX-RS アプリケーションが例外マッパーを使用する前に、例外マッパーをランタイムに登録する必要があります。例外マッパーは、アプリケーションの設定ファイルの **jaxrs:providers** 要素を使用してランタイムで登録されます。

jaxrs:providers 要素は **jaxrs:server** 要素の子で、**bean** 要素のリストが含まれます。各 **bean** 要素は 1 つの例外マッパーを定義します。

例50.7「ランタイムへの例外マッパーの登録」は、カスタム例外マッパー **SecurityExceptionMapper** を使用するように設定された JAX-RS サーバーを示しています。

例50.7 ランタイムへの例外マッパーの登録

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

WebApplicationException の例外マッパーの登録

WebApplicationException 例外の例外マッパーを登録することは、この例外型はデフォルトの **WebApplicationExceptionMapper** によって自動的に処理されるため、特殊なケースです。通常、**WebApplicationException** を処理する予定のカスタムマッパーを登録する場合でも、引き続きデフォルトの **WebApplicationExceptionMapper** によって処理されます。このデフォルトの動作を変更するには、**default.wae.mapper.least.specific** プロパティーを **true** に設定する必要があります。

たとえば、以下の XML コードは JAX-RS エンドポイントで **default.wae.mapper.least.specific** プロパティーを有効にする方法を示しています。

```
<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>
```

以下の例のように、インターセプターで **default.wae.mapper.least.specific** プロパティーを設定することもできます。

```
// Java
public void handleMessage(Message message)
{
    m.put("default.wae.mapper.least.specific", true);
    ...
```

第51章 エンティティーサポート

概要

Apache CXF ランタイムは、追加設定なしで MIME タイプと Java オブジェクトとの間のマッピングを少数サポートします。開発者はカスタムリーダーとライターを実装してマッピングを拡張できます。カスタムリーダーとライターは、起動時にランタイムに登録されます。

概要

ランタイムは、JAX-RS MessageBodyReader と MessageBodyWriter 実装に依存して、HTTP メッセージと Java 表現間でデータをシリアライズおよびデシリアライズします。リーダーとライターは、処理可能な MIME タイプを制限できます。

ランタイムは、共通マッピングのリーダーおよびライターを多数提供します。アプリケーションにより高度なマッピングが必要な場合に、開発者は MessageBodyReader インターフェイスおよび MessageBodyWriter インターフェイスのカスタム実装を指定できます。カスタムリーダーとライターは、アプリケーションの起動時にランタイムに登録されます。

ネイティブでサポートされるタイプ

表51.1「ネイティブでサポートされるエンティティーマッピング」 では、追加設定なしの Apache CXF によって提供されるエンティティーマッピングをリスト表示します。

表51.1 ネイティブでサポートされるエンティティーマッピング

Java タイプ	MIME タイプ
プリミティブ型	**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**
javax.xml.bind.JAXBElement	**text/xml、application/xml、application/*+xml**
JAXB アノテーションが付けられたオブジェクト	**text/xml、application/xml、application/*+xml**

Java タイプ	MIME タイプ
javax.ws.rs.core.MultivaluedMap<String, String>	**application/x-www-form-urlencoded** [a]
javax.ws.rs.core.StreamingOutput	*/* [b]

[a] このマッピングは、HTML フォームデータの処理に使用されます。

[b] このマッピングは、データをコンシューマーに返す場合にのみサポートされます。

カスタムリーダー

カスタムエンティティーリーダーは、受信 HTTP 要求を、サービスの実装が操作できる Java タイプにマッピングします。javax.ws.rs.ext.MessageBodyReader インターフェイスを実装します。

例51.1「メッセージリーダーインターフェイス」 に示されるインターフェイスには、実装が必要な 2 つのメソッドがあります。

例51.1 メッセージリーダーインターフェイス

```
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()

この **isReadable()** メソッドは、リーダーがデータストリームを読み取り、適切な型のエンティティー表現を作成できるかどうかを判断します。リーダーが適切な型のエンティティーを作成できる場合は、メソッドが **true** を返します。
表51.2「リーダーがエンティティーを生成できるかどうかを決定するために使用されるパラメーター」では **isReadable()** メソッドのパラメーターについて説明します。

表51.2 リーダーがエンティティーを生成できるかどうかを決定するために使用されるパラメーター

パラメーター	型	説明
type	**Class<T>**	エンティティーを保存するために使用されるオブジェクトの実際の Java クラスを指定します。
genericType	型	エンティティーを保存するために使用されるオブジェクトの Java タイプを指定します。たとえば、メッセージボディーがメソッドパラメーターに変換される場合、値は **Method.getGenericParameterTypes()** メソッドによって返されるメソッドパラメーターの型になります。
annotations	Annotation[]	エンティティーの保存用に作成されたオブジェクトの宣言に、アノテーションのリストを指定します。たとえば、メッセージボディーがメソッドパラメーターに変換される場合は、**Method.getParameterAnnotations()** メソッドによって返されるパラメーターのアノテーションになります。
mediaType	**MediatType**	HTTP エンティティーの MIME タイプを指定します。

readFrom()

readFrom() メソッドは HTTP エンティティーを読み取り、目的の Java オブジェクトに変換します。読み取りに成功すると、メソッドはエンティティーを含め、作成された Java オブジェクトを返します。入力ストリームの読み取り中にエラーが発生した場合に、メソッドは IOException 例外を出力する必要があります。HTTP エラー応答を必要とするエラーが発生した場合は、HTTP 応答のある WebApplicationException が発生します。
表51.3「エンティティーの読み取りに使用されるパラメーター」では **readFrom()** メソッドのパラメーターについて説明します。

表51.3 エンティティーの読み取りに使用されるパラメーター

パラメーター	型	説明
type	**Class<T>**	エンティティーを保存するために使用されるオブジェクトの実際の Java クラスを指定します。

パラメーター	型	説明
genericType	型	エンティティーを保存するために使用されるオブジェクトの Java タイプを指定します。たとえば、メッセージボディーがメソッドパラメーターに変換される場合、値は **Method.getGenericParameterTypes()** メソッドによって返されるメソッドパラメーターの型になります。
annotations	Annotation[]	エンティティーの保存用に作成されたオブジェクトの宣言に、アノテーションのリストを指定します。たとえば、メッセージボディーがメソッドパラメーターに変換される場合は、**Method.getParameterAnnotations()** メソッドによって返されるパラメーターのアノテーションになります。
mediaType	**MediatType**	HTTP エンティティーの MIME タイプを指定します。
httpHeaders	MultivaluedMap<String, String>	エンティティーに関連付けられた HTTP メッセージヘッダーを指定します。
entityStream	**InputStream**	HTTP エンティティーが含まれる入力ストリームを指定します。

重要

このメソッドでは入力ストリームは終了されないはずです。

MessageBodyReader 実装をエンティティーリーダーとして使用するには、**javax.ws.rs.ext.Provider** アノテーションを付ける必要があります。**@Provider** アノテーションは、提供された実装が追加の機能を提供することをランタイムに警告します。この実装は、「リーダーおよびライターの登録」 の説明に従ってランタイムにも登録する必要があります。

デフォルトでは、カスタムエンティティープロバイダーはすべての MIME タイプを処理します。**javax.ws.rs.Consumes** アノテーションを使用して、カスタムエンティティーリーダーが処理する MIME タイプを制限できます。**@Consumes** アノテーションは、カスタムエンティティープロバイダーが読み取る MIME タイプのコンマ区切りリストを指定します。エンティティーが指定された MIME タイプでない場合に、エンティティープロバイダーはリーダー候補として選択されません。

例51.2「XML ソースエンティティーリーダー」 では、消費された XML エンティティーを使用し、それらを Source オブジェクトに保存するエンティティーリーダーを紹介しています。

例51.2 XML ソースエンティティーリーダー

```
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");
  }
}
```

カスタムライター

カスタムエンティティーライターは、Java タイプを HTTP エンティティーにマッピングします。javax.ws.rs.ext.MessageBodyWriter インターフェイスを実装します。

例51.3「メッセージライターインターフェイス」 に示されるインターフェイスには、実装が必要な 3 つのメソッドがあります。

例51.3 メッセージライターインターフェイス

```
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() メソッドは、エンティティーライターが Java 型を適切なエンティティー型にマップできるかどうかを判断します。ライターがマッピングを実行できる場合、メソッドは **true** を返します。
表51.4「エンティティーの読み取りに使用されるパラメーター」では **isWritable()** メソッドのパラメーターについて説明します。

表51.4 エンティティーの読み取りに使用されるパラメーター

パラメーター	型	説明
type	**Class<T>**	書き込まれるオブジェクトの Java クラスを指定します。
genericType	型	リソースメソッドの戻り型の反映または返されたインスタンスの検査のいずれかによって取得される、書き込まれるオブジェクトの Java 型を指定します。「汎用型情報を含むエンティティーの返答」で説明されている **GenericEntity** クラスは、この値を制御するためのサポートを提供します。
annotations	Annotation[]	エンティティーを返すメソッドのアノテーションのリストを指定します。
mediaType	**MediatType**	HTTP エンティティーの MIME タイプを指定します。

getSize()

getSize() メソッドは **writeTo()** の前に呼び出されます。このメソッドでは、書き込まれているエンティティーの長さ (バイト単位) を返します。正の値が返された場合、値は HTTP メッセージの **Content-Length** ヘッダーに書き込まれます。
表51.5「エンティティーの読み取りに使用されるパラメーター」では **getSize()** メソッドのパラメーターについて説明します。

表51.5 エンティティーの読み取りに使用されるパラメーター

パラメーター	型	説明
t	generic	書き込まれるインスタンスを指定します。

パラメーター	型	説明
type	**Class<T>**	書き込まれるオブジェクトの Java クラスを指定します。
genericType	型	リソースメソッドの戻り型の反映または返されたインスタンスの検査のいずれかによって取得される、書き込まれるオブジェクトの Java 型を指定します。「汎用型情報を含むエンティティーの返答」で説明されている **GenericEntity** クラスは、この値を制御するためのサポートを提供します。
annotations	Annotation[]	エンティティーを返すメソッドのアノテーションのリストを指定します。
mediaType	**MediatType**	HTTP エンティティーの MIME タイプを指定します。

writeTo()

writeTo() メソッドは Java オブジェクトを目的のエンティティー型に変換し、エンティティーを出力ストリームに書き込みます。エンティティーを出力ストリームに書き込むときにエラーが発生した場合に、メソッドは IOException 例外を出力する必要があります。HTTP エラー応答を必要とするエラーが発生した場合は、HTTP 応答のある WebApplicationException が発生します。
表51.6「エンティティーの読み取りに使用されるパラメーター」では **writeTo()** メソッドのパラメーターについて説明します。

表51.6 エンティティーの読み取りに使用されるパラメーター

パラメーター	型	説明
t	generic	書き込まれるインスタンスを指定します。
type	**Class<T>**	書き込まれるオブジェクトの Java クラスを指定します。

パラメーター	型	説明
genericType	型	リソースメソッドの戻り型の反映または返されたインスタンスの検査のいずれかによって取得される、書き込まれるオブジェクトの Java 型を指定します。「汎用型情報を含むエンティティーの返答」で説明されている **GenericEntity** クラスは、この値を制御するためのサポートを提供します。
annotations	Annotation[]	エンティティーを返すメソッドのアノテーションのリストを指定します。
mediaType	**MediatType**	HTTP エンティティーの MIME タイプを指定します。
httpHeaders	MultivaluedMap<String, Object>	エンティティーに関連付けられた HTTP 応答ヘッダーを指定します。
entityStream	**OutputStream**	エンティティーが書き込まれる出力ストリームを指定します。

MessageBodyWriter 実装をエンティティーライターとして使用するには、**javax.ws.rs.ext.Provider** アノテーションを付ける必要があります。**@Provider** アノテーションは、提供された実装が追加の機能を提供することをランタイムに警告します。この実装は、「リーダーおよびライターの登録」 の説明に従ってランタイムにも登録する必要があります。

デフォルトでは、カスタムエンティティープロバイダーはすべての MIME タイプを処理します。**javax.ws.rs.Produces** アノテーションを使用して、カスタムエンティティーライターが処理する MIME タイプを制限できます。**@Produces** アノテーションは、カスタムエンティティープロバイダーが生成する MIME タイプのコンマ区切りリストを指定します。エンティティーが指定された MIME タイプでない場合に、エンティティープロバイダーはライター候補として選択されません。

例51.4「XML ソースエンティティーライター」 では、ソースオブジェクトを取得して XML エンティティーを生成するエンティティーライターを紹介しています。

例51.4 XML ソースエンティティーライター

```
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;
  }
}
```

リーダーおよびライターの登録

JAX-RS アプリケーションがカスタムエンティティープロバイダーを使用できるようにするには、カスタムプロバイダーをランタイムに登録する必要があります。プロバイダーは、アプリケーションの設定ファイルの **jaxrs:providers** 要素を使用するか、**JAXRSServerFactoryBean** クラスを使用してランタイムに登録されます。

jaxrs:providers 要素は **jaxrs:server** 要素の子で、**bean** 要素のリストが含まれます。各 **bean** 要素は 1 つのエンティティープロバイダーを定義します。

例51.5「ランタイムを使用したエンティティープロバイダーの登録」 では、カスタムエンティティープロバイダーのセットを使用するよう設定された JAX-RS サーバーを示しています。

例51.5 ランタイムを使用したエンティティープロバイダーの登録

```
<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 クラスは、設定 API へのアクセスを提供する Apache CXF エクステンションです。インスタンス化されたエンティティープロバイダーをアプリケーションに追加できるようにする **setProvider()** メソッドがあります。例51.6「エンティティープロバイダーのプログラムによる登録」 は、エンティティープロバイダーをプログラムで登録するためのコードを示します。

例51.6 エンティティープロバイダーのプログラムによる登録

```
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
...
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
...
SourceProvider provider = new SourceProvider();
sf.setProvider(provider);
...
```

第52章 コンテキスト情報の取得と使用

概要

コンテキスト情報には、リソースの URI、HTTP ヘッダー、およびその他のインジェクションアノテーションを使用して読み取りできないその他の情報が含まれます。Apache CXF は、考えられるすべてのコンテキスト情報を 1 つのオブジェクトに統合する特別なクラスを提供します。

52.1. コンテキストの概要

コンテキストのアノテーション

javax.ws.rs.core.Context アノテーションを使用して、コンテキスト情報がフィールドまたはリソースメソッドパラメーターに注入されるよう指定します。コンテキスト型のいずれかのフィールドまたはパラメーターにアノテーションを付けると、適切なコンテキスト情報をアノテーション付きフィールドまたはパラメーターに注入するように、ランタイムに指示が出されます。

コンテキストのタイプ

表52.1「コンテキストタイプ」 は、挿入可能なコンテキスト情報のタイプと、その情報をサポートするオブジェクトをリスト表示します。

表52.1 コンテキストタイプ

Object	コンテキスト情報
UriInfo	完全リクエスト URI
HttpHeaders	HTTP メッセージヘッダー
Request	最適な表現バリアントを判別するため、または一連の前提条件が設定されているかどうかを判別するために使用できる情報
SecurityContext	使用中の認証スキーム、要求チャネルが安全であるかどうか、およびユーザー原則など、要求者のセキュリティーに関する情報

コンテキスト情報を使用できる場所

コンテキスト情報は、JAX-RS アプリケーションの以下の部分で利用できます。

- リソースクラス
- サブリソースメソッド
- エンティティープロバイダー
- 例外マッパー

範囲

@Context アノテーションを使用して注入されたすべてのコンテキスト情報は現在のリクエストに固有です。これは、エンティティープロバイダーや例外マッパーを含むすべてのケースに該当します。

コンテキストの追加

JAX-RS フレームワークにより、開発者はコンテキストメカニズムを使用して注入できる情報タイプを拡張できます。Context<T> オブジェクトを実装し、ランタイムに登録して、カスタムコンテキストを追加します。

52.2. 完全な要求 URI の操作

概要

要求 URI には大量の情報が含まれます。この情報は、「リクエスト URI からのデータの注入」 に記載のメソッドパラメーターを使用してアクセスできますが、パラメーターを使用すると URI の処理方法に特定の制約が強制的に課されます。URI のセグメントにアクセスするためにパラメーターを使用しても、完全な要求 URI へのリソースにはアクセスできません。

URI コンテキストをリソースに注入することで、完全な要求 URI にアクセスできます。URI は UriInfo オブジェクトとして提供されます。UriInfo インターフェイスは、URI を複数の方法で破棄する関数を提供します。また、URI を UriBuilder オブジェクトとして提供して、クライアントに返す URI を作成することもできます。

:experimental:

52.2.1. URI 情報の挿入

概要

UriInfo オブジェクトであるクラスフィールドまたはメソッドパラメーターに **@Context** アノテーションが付けられると、現在のリクエストの URI コンテキストが UriInfo オブジェクトに注入されます。

例

URI コンテキストのクラスフィールドへの注入 は、URI コンテキストを挿入してフィールドにデータが投入されたクラスを示しています。

URI コンテキストのクラスフィールドへの注入

```
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. URI の操作

概要

URI コンテキストを使用する主な利点の 1 つとして、サービスのベース URI と選択したリソースの URI のパスセグメントにアクセスできることが挙げられます。この情報は、URI に基づいた処理の決定や、応答の一部として返す URI の計算など、多くの目的に役立ちます。たとえば、要求のベース URI に **.com** 拡張が含まれる場合に、このサービスでは米国ドルを、ベース URI に **.co.uk** 拡張が含まれる場合には、英国ポンドを使用するように選択できます。

UriInfo インターフェイスは、URI の一部にアクセスするためのメソッドを提供します。

- ベース URI
- リソースパス
- 完全な URI

ベース URI の取得

ベース URI は、サービスが公開されるルート URI です。サービスの **@Path** アノテーションで指定された URI の一部は含まれません。たとえば、例47.5「URI のデコードの無効化」 で定義されたリソースを実装するサービスが http://fusesource.org に公開され、リクエストが http://fusesource.org/montersforhire/nightstalker?12 で行われた場合、ベース URI は http://fusesource.org になります。

表52.2「リソースのベース URI にアクセスするメソッド」 は、ベース URI を返すメソッドを示します。

表52.2 リソースのベース URI にアクセスするメソッド

メソッド	詳細
URIgetBaseUri	サービスのベース URI を **URI** オブジェクトとして返します。
UriBuildergetBaseUriBuilder	ベース URI を **javax.ws.rs.core.UriBuilder** オブジェクトとして返します。**UriBuilder** クラスは、サービスによって実装された他のリソースの URI を作成するのに役立ちます。

パスの取得

要求 URI の **パス** 部分は、現在のリソースの選択に使用された URI の部分になります。ベース URI は含まれませんが、URI に含まれる URI テンプレート変数とマトリックスパラメーターは含まれます。

パスの値は選択したリソースによって異なります。たとえば、リソースのパスの取得 で定義されたリソースのパスは以下のようになります。

- **rootPath**: /monstersforhire/
- **getterPath**: /mostersforhire/nightstalker
  GET リクエストは /monstersforhire/nightstalker で実行されました。
- **putterPath**: /mostersforhire/911
  PUT リクエストは /monstersforhire/911 で実行されました。

リソースのパスの取得

```
@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「リソースのパスにアクセスするメソッド」では、リソースパスを返すメソッドについて説明します。

表52.3 リソースのパスにアクセスするメソッド

メソッド	詳細
StringgetPath	リソースのパスをデコードされた URI として返します。
StringgetPathbooleandecode	リソースのパスを返します。**false** を指定すると、URI のデコードが無効になります。

メソッド	詳細
List<PathSegment>getPathSegments	デコードされたパスを javax.ws.rs.core.PathSegment オブジェクトのリストとして返します。マトリックスパラメーターを含むパスの各部分は、リストの一意のエントリーに配置されます。 たとえば、リソースパス **box/round#tall** の場合は、**box**、**round**、および **tall** の 3 つのエントリーのリストになります。
List<PathSegment>getPathSegmentsbooleandecode	パスを javax.ws.rs.core.PathSegment オブジェクトのリストとして返します。マトリックスパラメーターを含むパスの各部分は、リストの一意のエントリーに配置されます。**false** を指定すると、URI のデコードが無効になります。 たとえば、リソースパス **box#tall/round** の場合は、**box**、**tall**、および **round** の 3 つのエントリーのリストになります。

完全な要求 URI の取得

表52.4「完全な要求 URI にアクセスする方法」 では、完全なリクエスト URI を返すメソッドを説明します。要求 URI またはリソースの絶対パスを返すオプションがあります。相違点は、要求 URI には URI に追加されるクエリーパラメーターが含まれ、絶対パスにはクエリーパラメーターが含まれないことです。

表52.4 完全な要求 URI にアクセスする方法

メソッド	詳細
URIgetRequestUri	クエリーパラメーターおよびマトリックスパラメーターを含む完全なリクエスト URI を **java.net.URI** オブジェクトとして返します。
UriBuildergetRequestUriBuilder	クエリーパラメーターおよびマトリックスパラメーターを含む完全なリクエスト URI を **javax.ws.rs.UriBuilder** オブジェクトとして返します。**UriBuilder** クラスは、サービスによって実装された他のリソースの URI を作成するのに役立ちます。
URIgetAbsolutePath	マトリックスパラメーターを含む完全なリクエスト URI を **java.net.URI** オブジェクトとして返します。絶対パスにはクエリーパラメーターは含まれません。

メソッド	詳細
UriBuildergetAbsolutePathBuilder	マトリックスパラメーターを含む完全なリクエスト URI を **javax.ws.rs.UriBuilder** オブジェクトとして返します。絶対パスにはクエリーパラメーターは含まれません。

URI http://fusesource.org/montersforhire/nightstalker?12 を使用して行われるリクエストの場合は、**getRequestUri()** メソッドは http://fusesource.org/montersforhire/nightstalker?12 を返します。**getAbsolutePath()** メソッドは http://fusesource.org/montersforhire/nightstalker を返します。

52.2.3. URI テンプレート変数の値の取得

概要

「パスの設定」 で説明されているように、リソースパスには、値に動的にバインドされた変数セグメントを含めることができます。多くの場合、これらの変数パスセグメントは、「URI のパスからのデータ取得」 で説明されているリソースメソッドへのパラメーターとして使用されます。ただし、URI コンテキストを使用してそれらにアクセスすることもできます。

パスパラメーターを取得する方法

UriInfo インターフェイスは、(例52.1「URI コンテキストからパスパラメーターを返すメソッド」 のように) パスパラメーターのリストを返す 2 つのメソッドを提供します。

例52.1 URI コンテキストからパスパラメーターを返すメソッド

```
MultivaluedMap<java.lang.String,
java.lang.String>getPathParametersMultivaluedMap<java.lang.String,
java.lang.String>getPathParametersbooleandecode
```

パラメーターを取らない **getPathParameters()** メソッドは、パスパラメーターを自動的にデコードします。URI のデコードを無効にする場合は、**getPathParameters(false)** を使用します。

値は、テンプレート識別子をキーとして使用してマップに保存されます。たとえば、リソースの URI テンプレートが **/{color}/box/{note}** の場合、返されるマップには **color** キーと **note** キーを持つ 2 つのエントリーがあります。

例

例52.2「URI コンテキストからのパスパラメーターの抽出」 は、URI コンテキストを使用してパスパラメーターを取得するためのコードを示しています。

例52.2 URI コンテキストからのパスパラメーターの抽出

```
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 アノテーションの継承

概要

JAX-RS アノテーションは、アノテーション付きインターフェイスを実装するサブクラスとクラスによって継承できます。継承メカニズムにより、サブクラスと実装クラスは、親から継承されたアノテーションを上書きできます。

概要

継承は、開発者が特定のニーズに対応するために特化できる汎用オブジェクトを作成できるため、Java で最も強力なメカニズムの 1 つです。JAX-RS は、クラスをリソースにマッピングする時に使用されるアノテーションをスーパークラスから継承できるようにすることで、この機能を維持します。

JAX-RS のアノテーション継承は、インターフェイスのサポートも拡張します。実装クラスは、実装するインターフェイスで使用される JAX-RS アノテーションを継承します。

JAX-RS 継承ルールは、継承されたアノテーションをオーバーライドするメカニズムを提供します。ただし、JAX-RS アノテーションをスーパークラスまたはインターフェイスから継承するコンストラクトから完全に削除することはできません。

継承ルール

リソースクラスは、実装するインターフェイスから JAX-RS アノテーションを継承します。リソースクラスは、拡張しているスーパークラスから JAX-RS アノテーションも継承します。スーパークラスから継承されたアノテーションは、インターフェイスから継承されたアノテーションよりも優先されます。

例53.1「アノテーションの継承」のコード例では、**Kaijin** クラスの **getMonster()** メソッドは、Kaiju インターフェイスから **@Path**、**@GET**、および **@PathParam** アノテーションを継承します。

例53.1 アノテーションの継承

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

継承されたアノテーションの上書き

継承されたアノテーションの上書きは、新しいアノテーションを指定するのと同じくらい簡単です。サブクラスまたは実装クラスがメソッドに独自の JAX-RS アノテーションを指定した場合に、そのメソッドのすべての JAX-RS アノテーションは無視されます。

例53.2「アノテーションの継承の上書き」 のコード例では、**Kaijin** クラスの **getMonster()** メソッドは Kaiju インターフェイスからのアノテーションを継承しません。実装クラスは **@Produces** アノテーションをオーバーライドします。これにより、インターフェイスからのすべてのアノテーションが無視されます。

例53.2 アノテーションの継承の上書き

```
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 OPENAPI サポートによる JAX-RS エンドポイントの拡張

概要

CXF OpenApiFeature (**org.apache.cxf.jaxrs.openapi.OpenApiFeature**) を使用すると、パブリッシュ済みの JAX-RS サービスエンドポイントを単純な設定で拡張することによって OpenAPI ドキュメントを生成できます。

OpenApiFeature は、Spring Boot および Karaf 実装の両方でサポートされます。

54.1. OPENAPIFEATURE オプション

OpenApiFeature では、以下のオプションを使用できます。

表54.1 OpenApiFeature 操作

名前	説明	デフォルト
configLocation	OpenAPI 設定の場所	null
contactEmail	連絡先メール+	null
contactName	連絡先名+	null
contactUrl	連絡先リンク+	null
customizer	カスタマイザークラスインスタンス	null
description	説明+	null
filterClass	セキュリティーフィルター++	null
ignoredRoutes	すべてのリソースをスキャンする際に、特定のパスを除外します (**scanAllResources** を参照)++	null
license	ライセンス+	null
licenseUrl	ライセンス URL+	null
prettyPrint	openapi.json を生成する際に、JSON ドキュメント++ をプリティープリントで出力します。。	true
propertiesLocation	プロパティーファイルの場所	**/swagger.properties**
readAllResources	@Operation++ なしですべての操作も読み取ります。	true

名前	説明	デフォルト
resourceClasses	スキャンする必要があるリソースクラスのリスト	null
resourcePackages	リソースのスキャンが必要なパッケージ名のリスト++	null
runAsFilter	機能をフィルターとして実行します。	false
scan	すべての JAX-RS リソースを自動的にスキャンします。	true
scanKnownConfigLocations	既知の OpenAPI 設定の場所 (クラスパスまたはファイルシステム) をスキャンします。 openapi-configuration.yaml openapi-configuration.json openapi.yaml openapi.json	true
scannerClass	アプリケーション、リソースパッケージ、リソースクラス、およびクラスパススキャンのスコープを設定するために使用される JAX-RS API スキャナークラスの名前。詳細は、リソーススキャン のセクションを参照してください。	null
securityDefinitions	セキュリティー定義のリスト+	null
supportSwaggerUi	SwaggerUI サポートのオン/オフを切り替えます	null(== true)
swaggerUiConfig	Swagger UI の設定	null
swaggerUiMavenGroupAndArtifact	SwaggerUI を特定するための Maven アーティファクト	null
swaggerUiVersion	SwaggerUI のバージョン	null
termsOfServiceUrl	サービス URL+	null
title	タイトル+	null

名前	説明	デフォルト
useContextBasedConfig	設定されている場合に、それぞれの OpenApiContext インスタンスに対して一意の Context Id が生成されます (複数のサーバーエンドポイントの使用 を参照)。また、scan プロパティーを false に設定することもできます。	false
version	バージョン+	null

+ オプションは OpenAPI クラスで定義されます。

++ オプションは SwaggerConfiguration クラスで定義されます。

54.2. KARAF 実装

ここでは、REST サービスが JAR ファイルで定義され、Fuse on Karaf コンテナーにデプロイされる OpenApiFeature を使用する方法について説明します。

54.2.1. クイックスタートの例

Fuse Software Downloads ページから Red Hat Fuse **quickstarts** をダウンロードできます。

Quickstart zip ファイルには、CXF を使用して RESTful (JAX-RS) Web サービスを作成する方法と、OpenAPI を有効にして JAX-RS エンドポイントにアノテーションを付ける方法を実証するクイックスタートの **/cxf/rest/** ディレクトリーが含まれています。

54.2.2. OpenAPI の有効化

OpenAPI を有効にするには、以を行います。

- CXF クラス (**org.apache.cxf.jaxrs.openapi.OpenApiFeature**) を **<jaxrs:server>** 定義に追加して CXF サービスを定義する XML ファイルを変更します。
  例は、例 55.4: XML ファイル を参照してください。
- REST リソースクラスで以下を行います。
  - サービスで必要な各アノテーションの OpenAPI アノテーションをインポートします。

    ```
    import io.swagger.annotations.*
    ```

    ここで、* は **Api**、**ApiOperation**、**ApiParam**、**ApiResponse**、**ApiResponses** などです。

    詳細は、**https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X** を参照してください。

    例は、例 55.5: リソースクラスの例 を参照してください。

- OpenAPI アノテーションを JAX-RS でアノテーションが付けられたエンドポイント (**@PATH**、**@PUT**、**@POST**、**@GET**、**@Produces**、**@Consumes**、**@DELETE**、**@PathParam** など) に追加します。

例は、例 55.5: リソースクラスの例 を参照してください。

例 55.4: XML ファイル

```
<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>
```

例 55.5: リソースクラスの例

```
.
.
.
```

```
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;

.
.
.

@Path("/customerservice/")
@Api(value = "/customerservice", description = "Operations about customerservice")
public class CustomerService {

    private static final Logger LOG =
        LoggerFactory.getLogger(CustomerService.class);

    private MessageContext jaxrsContext;
    private long currentId                = 123;
    private Map<Long, Customer> customers = new HashMap<>();
    private Map<Long, Order> orders       = new HashMap<>();

    public CustomerService() {
        init();
    }


    @GET
    @Path("/customers/{id}/")
    @Produces("application/xml")
    @ApiOperation(value = "Find Customer by ID", notes = "More notes about this
        method", response = Customer.class)
    @ApiResponses(value = {
            @ApiResponse(code = 500, message = "Invalid ID supplied"),
            @ApiResponse(code = 204, message = "Customer not found")
            })
    public Customer getCustomer(@ApiParam(value = "ID of Customer to fetch",
        required = true) @PathParam("id") String id) {
        LOG.info("Invoking getCustomer, Customer id is: {}", id);
        long idNumber = Long.parseLong(id);
        return customers.get(idNumber);
    }


    @PUT
```

```
@Path("/customers/")
@Consumes({ "application/xml", "application/json" })
@ApiOperation(value = "Update an existing Customer")
@ApiResponses(value = {
        @ApiResponse(code = 500, message = "Invalid ID supplied"),
        @ApiResponse(code = 204, message = "Customer not found")
        })
public Response updateCustomer(@ApiParam(value = "Customer object that needs
    to be updated", required = true) Customer customer) {
    LOG.info("Invoking updateCustomer, Customer name is: {}", customer.getName());
    Customer c = customers.get(customer.getId());
    Response r;
    if (c != null) {
        customers.put(customer.getId(), customer);
        r = Response.ok().build();
    } else {
        r = Response.notModified().build();
    }

    return r;
}

@POST
@Path("/customers/")
@Consumes({ "application/xml", "application/json" })
@ApiOperation(value = "Add a new Customer")
@ApiResponses(value = { @ApiResponse(code = 500, message = "Invalid ID
    supplied"), })
public Response addCustomer(@ApiParam(value = "Customer object that needs to
    be updated", required = true) Customer customer) {
    LOG.info("Invoking addCustomer, Customer name is: {}", customer.getName());
    customer.setId(++currentId);

    customers.put(customer.getId(), customer);
    if (jaxrsContext.getHttpHeaders().getMediaType().getSubtype().equals("json"))
{
        return Response.ok().type("application/json").entity(customer).build();
    } else {
        return Response.ok().type("application/xml").entity(customer).build();
    }
}

@DELETE
@Path("/customers/{id}/")
@ApiOperation(value = "Delete Customer")
@ApiResponses(value = {
        @ApiResponse(code = 500, message = "Invalid ID supplied"),
        @ApiResponse(code = 204, message = "Customer not found")
        })
public Response deleteCustomer(@ApiParam(value = "ID of Customer to delete",
    required = true) @PathParam("id") String id) {
    LOG.info("Invoking deleteCustomer, Customer id is: {}", id);
    long idNumber = Long.parseLong(id);
    Customer c = customers.get(idNumber);

    Response r;
```

```
        if (c != null) {
            r = Response.ok().build();
            customers.remove(idNumber);
        } else {
            r = Response.notModified().build();
        }

        return r;
    }

.
.
.

}
```

54.3. SPRING BOOT 実装

本セクションでは、Spring Boot で Swagger2Feature を使用する方法について説明します。

OpenAPI 3 実装では、OpenApiFeature (**org.apache.cxf.jaxrs.openapi.OpenApiFeature**) を使用することに注意してください。

54.3.1. クイックスタートの例

クイックスタートサンプル (**https://github.com/fabric8-quickstarts/spring-boot-cxf-jaxrs**) は、Spring Boot で Apache CXF を使用する方法を実証します。クイックスタートは Spring Boot を使用して、Swagger が有効な CXF JAX-RS エンドポイントが含まれるアプリケーションを設定します。

54.3.2. Swagger の有効化

Swagger を有効にするには、以下の操作が必要です。

- REST アプリケーションで以下を行います。
  - Swagger2Feature をインポートする。

    ```
    import org.apache.cxf.jaxrs.swagger.Swagger2Feature;
    ```

  - Swagger2Feature を CXF エンドポイントに追加する。

    ```
    endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
    ```

    例は、例 55.1: REST アプリケーションの例 を参照してください。
- Java 実装ファイルで、サービスが必要とする各アノテーションの Swagger API アノテーションをインポートします。

  ```
  import io.swagger.annotations.*
  ```

  ここで、* は **Api**、**ApiOperation**、**ApiParam**、**ApiResponse**、**ApiResponses** などです。

  詳細は、**https://github.com/swagger-api/swagger-core/wiki/Annotations** を参照してください。

例は、例 55.2: Java 実装ファイル を参照してください。

- Java ファイルで、Swagger アノテーションを JAX-RS でアノテーションが付けられたエンドポイント
(**@PATH**、**@PUT**、**@POST**、**@GET**、**@Produces**、**@Consumes**、**@DELETE**、**@PathParam** など) に追加します。
例は、例 55.3: Java ファイル を参照してください。

例 55.1: REST アプリケーションの例

```
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();
    }
}
```

例 55.2: Java 実装ファイル

```
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!!!";
    }

}
```

例 55.3: Java ファイル

```
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. OPENAPI ドキュメントへのアクセス

OpenAPI が OpenApiFeature で有効にされると、サービスエンドポイントの場所の後に **/openapi.json** または **/openapi.yaml** が続いて URL が設定される場所で OpenAPI ドキュメントを利用できます。

たとえば、**http://host:port/context/services/** でパブリッシュされる JAX-RS エンドポイントの場合 (ここで、**context** は Web アプリケーションコンテキストで、**/services** はサーブレット URL)、その OpenAPI ドキュメンテーションは **http://host:port/context/services/openapi.json** および **http://host:port/context/services/openapi.yaml** で利用できます。

OpenApiFeature がアクティブな場合には、CXF Services ページに OpenAPI ドキュメントへのリンクが追加されます。

上記の例では、**http://host:port/context/services/services** にアクセスし、OpenAPI JSON ドキュメントを返すリンクをたどります。

別のホストの OpenAPI UI から定義にアクセスするために CORS サポートが必要な場合は、**cxf-rt-rs-security-cors** から **CrossOriginResourceSharingFilter** を追加できます。

54.5. リバースプロキシーを介した OPENAPI へのアクセス

OpenAPI JSON ドキュメントまたは OpenAPI UI にリバースプロキシー経由でアクセスする場合は、以下のオプションを設定します。

- **CXFServlet use-x-forwarded-headers** init パラメーターを **true** に設定します。
  - Spring Boot で、パラメーター名に **cxf.servlet.init** を付けます。

    ```
    cxf.servlet.init.use-x-forwarded-headers=true
    ```

  - Karaf で、以下の行を **installDir/etc/org.apache.cxf.osgi.cfg** 設定ファイルに追加します。

    ```
    cxf.servlet.init.use-x-forwarded-headers=true
    ```

    注記: **etc** ディレクトリーに **org.apache.cxf.osgi.cfg** ファイルがない場合は、作成できます。

- OpenApiFeature **basePath** オプションの値を指定し、OpenAPI が **basePath** 値をキャッシュしないようにする場合は、OpenApiFeature **usePathBasedConfig** オプションを **TRUE** に設定します。

```
<bean class="org.apache.cxf.jaxrs.openapi.OpenApiFeature">
    <property name="usePathBasedConfig" value="TRUE" />
</bean>
```

パート VII. APACHE CXF インターセプターの開発

このガイドでは、メッセージの前処理と後処理を実行できる Apache CXF インターセプターを記述する方法について説明します。

第55章 APACHE CXF RUNTIME のインターセプター

概要

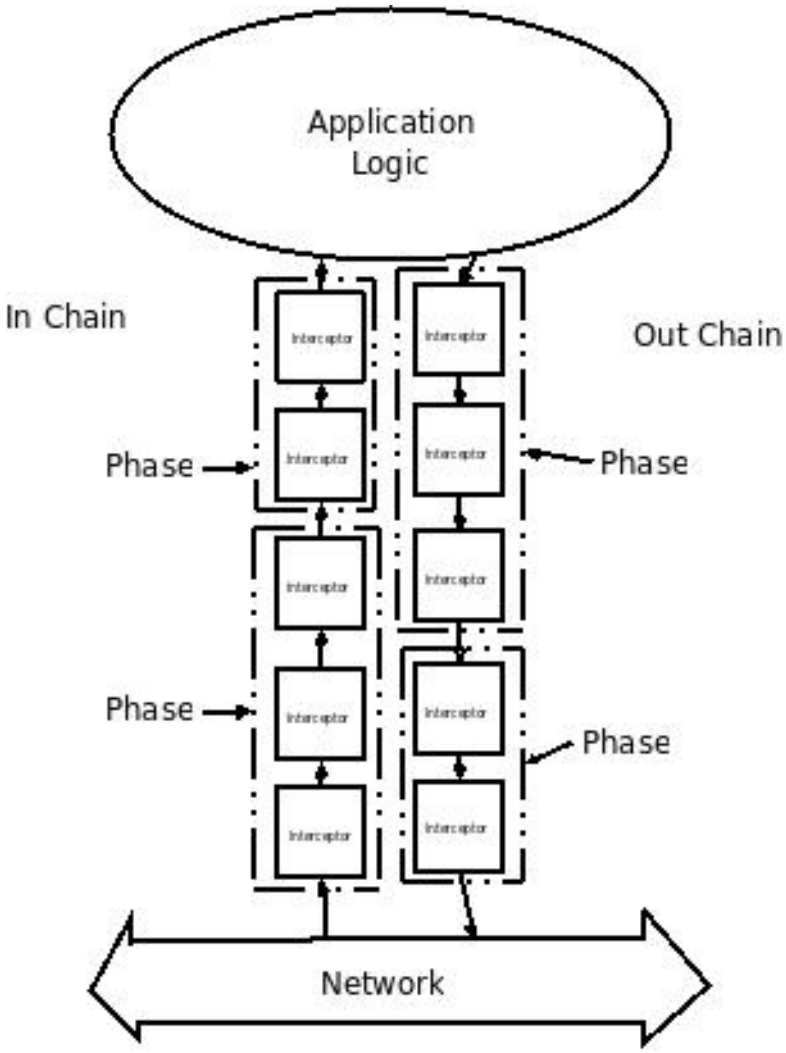
Apache CXF ランタイムの機能のほとんどは、インターセプターで実装されます。Apache CXF ランタイムで作成されるすべてのエンドポイントには、メッセージ処理に候補として 3 つのインターセプターチェーンがあります。これらのチェーンのインターセプターは、ネットワーク全体で転送された生データと、エンドポイントの実装コードによって処理される Java オブジェクトの間でメッセージを変換します。インターセプターは、適切な順序で処理が行われるように複数の段階に分けて編成されます。

概要

Apache CXF 機能の大部分は、メッセージの処理を伴います。コンシューマーがリモートサービスで呼び出しを行う場合に、ランタイムはサービスが消費できるメッセージに、データをマーシャリングし、ネットワーク上に配置する必要があります。サービスプロバイダーは、メッセージをアンマーシャリングしてビジネスロジックを実行し、応答を適切なメッセージ形式でマーシャリングする必要があります。次に、コンシューマーは応答メッセージをアンマーシャリングし、適切なリクエストに関連付けて、コンシューマーのアプリケーションコードに戻す必要があります。Apache CXF ランタイムは、基本的なマーシャリングおよびアンマーシャリングの他に、メッセージデータでさまざまなことを実行できます。たとえば、WS-RM がアクティベートされると、ランタイムはメッセージをマーシャリングおよびアンマーシャリングする前に、メッセージチャンクや、確認メッセージを処理する必要があります。セキュリティーがアクティベートされると、ランタイムはメッセージ処理シーケンスの一部としてメッセージのクレデンシャルを検証する必要があります。

図55.1「Apache CXF インターセプターチェーン」 は、要求メッセージがサービスプロバイダーによって受信されるときにたどる基本的なパスを示しています。

図55.1 Apache CXF インターセプターチェーン



APACHE CXF でのメッセージ処理

Apache CXF で開発されたコンシューマーがリモートサービスを呼び出すと、以下のメッセージ処理シーケンスが開始されます。

1. Apache CXF ランタイムは、要求を処理するアウトバウンドインターセプターチェーンを作成します。

2. 呼び出しが双方向のメッセージ交換を開始する場合、ランタイムはインバウンドインターセプターチェーンと障害処理インターセプターチェーンを作成します。

3. 要求メッセージは、アウトバウンドインターセプターチェーンを介して順次渡されます。
チェーンの各インターセプターは、メッセージに処理を実行します。たとえば、Apache CXF が提供する SOAP インターセプターは、SOAP エンベロープでメッセージをパッケージ化します。

4. アウトバウンドチェーンのインターセプターのいずれかがエラー条件を作成すると、チェーンは巻き戻され、制御はアプリケーションコードに戻ります。
インターセプターチェーンは、以前に呼び出されたすべてのインターセプターで障害処理メソッドを呼び出すことで戻されます。

5. リクエストが適切なサービスプロバイダーにディスパッチされます。

6. 応答を受け取ると、インバウンドインターセプターチェーンを介して順次渡されます。

注記

応答がエラーメッセージである場合には、障害処理インターセプターチェーンに渡されます。

7. インバウンドチェーン上のインターセプターのいずれかがエラー条件を作成すると、チェーンは巻き戻されます。

8. メッセージがインバウンドインターセプターチェーンの最後に到達すると、アプリケーションコードに戻ります。

Apache CXF で開発されたサービスプロバイダーがコンシューマーから要求を受信すると、同様のプロセスが実行されます。

1. Apache CXF ランタイムは、インバウンドインターセプターチェーンを作成し、要求メッセージを処理します。

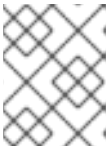
2. 要求が双方向メッセージ交換の一部である場合には、ランタイムはアウトバウンドインターセプターチェーンと障害処理インターセプターチェーンも作成します。

3. 要求はインバウンドインターセプターチェーンを介して順次渡されます。

4. インバウンドチェーンのインターセプターのいずれかがエラー条件を作成する場合に、チェーンは巻き戻され、障害はコンシューマーにディスパッチされます。
インターセプターチェーンは、以前に呼び出されたすべてのインターセプターで障害処理メソッドを呼び出すことで戻されます。

5. 要求がインバウンドインターセプターチェーンの最後に到達すると、サービス実装に渡されます。

6. 応答の準備ができると、アウトバウンドインターセプターチェーンを介して順次渡されます。



注記

応答が例外である場合に、応答は障害処理インターセプターチェーンを介して渡されます。

7. アウトバウンドチェーンのインターセプターのいずれかがエラー条件を作成する場合には、チェーンは巻き戻されて、障害メッセージがディスパッチされます。

8. 要求がアウトバウンドチェーンの最後に到達すると、コンシューマーにディスパッチされます。

インターセプター

Apache CXF ランタイムのすべてのメッセージ処理は、**インターセプター** によって行われます。インターセプターとは、アプリケーション層に渡す前にメッセージデータにアクセスできる POJO です。メッセージの変換、メッセージのヘッダーの削除、メッセージデータの検証など、さまざまなことを実行できます。たとえば、インターセプターはメッセージのセキュリティーヘッダーを読み取り、外部セキュリティーサービスに対して認証情報を検証し、メッセージ処理を継続できるかどうかを決定できます。

インターセプターで使用できるメッセージデータは、複数の要因によって決定されます。

- インターセプターのチェーン

- インターセプターのフェーズ
- チェーンの初期に発生する他のインターセプター

フェーズ

インターセプターは **フェーズ** 別に編成されます。フェーズは、共通の機能を持つインターセプターの論理グループです。各フェーズは、特定のタイプのメッセージ処理を行います。たとえば、インターセプターがマーシャリングされた Java オブジェクトを処理して、アプリケーション層に渡すのは、同じフェーズで行われます。

インターセプターチェーン

フェーズは **インターセプターチェーン** に集約されます。インターセプターチェーンは、メッセージがインバウンドか、アウトバウンドであるかどうかに基づいて順序付けされるインターセプターフェーズのリストです。

Apache CXF を使用して作成された各エンドポイントには、3 つのインターセプターチェーンがあります。

- インバウンドメッセージのチェーン
- アウトバウンドメッセージのチェーン
- エラーメッセージのチェーン

インターセプターチェーンは、主にエンドポイントによって使用されるバインディングおよびトランスポートの選択に基づいて構築されます。セキュリティーやロギングなどの他のランタイム機能もチェーンに追加します。開発者は設定を使用して、カスタムインターセプターをチェーンに追加することもできます。

インターセプターの開発

機能に関係なくインターセプターを開発する場合は、常に同じ基本手順に従います。

1. *56章 インターセプター API*
   Apache CXF は、カスタムインターセプターの開発を簡素化する抽象インターセプターを多数提供します。
2. 「インターセプターのフェーズの指定」
   インターセプターはメッセージの特定の部分が利用可能で、またデータを特定の形式で要求する必要があります。メッセージとデータの形式の内容は、インターセプターのフェーズによって部分的に決定されます。
3. 「フェーズでのインターセプター配置の制限」
   通常、フェーズ内のインターセプターの順序は重要ではありません。ただし、特定の状況では、同じフェーズで他のインターセプターの前または後にインターセプターが実行されることが重要です。
4. 「メッセージの処理」
5. 「エラー後の巻き戻し」
   インターセプターの実行後にアクティブなインターセプターチェーンでエラーが発生した場合に、その障害処理ロジックが呼び出されます。

6. 59章 *インターセプターを使用するためのエンドポイントの設定*

第56章 インターセプター API

概要

インターセプターは、ベースインターセプターインターフェイスを拡張する PhaseInterceptor インターフェイスを実装します。このインターフェイスは、インターセプターの実行を制御するために Apache CXF のランタイムによって使用されるメソッドを複数定義し、アプリケーション開発者が実装するのには適していません。インターセプターの開発を簡素化するために、Apache CXF は、拡張可能な多数の抽象インターセプター実装を提供します。

インターフェイス

Apache CXF のすべてのインターセプターは、例56.1「ベースインターセプターインターフェイス」 で紹介しているベースインターセプターインターフェイスを実装します。

例56.1 ベースインターセプターインターフェイス

```
package org.apache.cxf.interceptor;

public interface Interceptor<T extends Message>
{

  void handleMessage(T message) throws Fault;

  void handleFault(T message);

}
```

インターセプターインターフェイスは、開発者がカスタムインターセプターに実装する必要のある 2 つのメソッドを定義します。

handleMessage()

handleMessage() メソッドは、インターセプターでほとんどの作業を行います。メッセージチェーンの各インターセプターで呼び出され、処理されたメッセージの内容を受信します。開発者は、このメソッドにインターセプターのメッセージ処理ロジックを実装します。**handleMessage()** メソッドの実装に関する詳細は、「メッセージの処理」を参照してください。

handleFault()

通常のメッセージ処理が中断された場合に、インターセプターで **handleFault()** メソッドが呼び出されます。ランタイムは、インターセプターチェーンをアンワインドするため、呼び出された各インターセプターの **handleFault()** メソッドを逆順で呼び出します。**handleFault()** メソッドの実装に関する詳細は、「エラー後の巻き戻し」を参照してください。

インターセプターの多くでは、インターセプターインターフェイスは直接実装されません。代わりに、例56.2「フェーズインターセプターインターフェイス」 に示される PhaseInterceptor インターフェイスを実装します。PhaseInterceptor インターフェイスは、4 つのメソッドを追加し、インターセプターがインターセプターチェーンに参加できるようにします。

例56.2 フェーズインターセプターインターフェイス

```
package org.apache.cxf.phase;
...
```

```
public interface PhaseInterceptor<T extends Message> extends Interceptor<T>
{

    Set<String> getAfter();

    Set<String> getBefore();

    String getId();

    String getPhase();

}
```

抽象インターセプタークラス

PhaseInterceptor インターフェイスを直接実装する代わりに、開発者は **AbstractPhaseInterceptor** クラスを拡張する必要があります。この抽象クラスは、PhaseInterceptor インターフェイスのフェーズ管理メソッドの実装を提供します。**AbstractPhaseInterceptor** クラスは、**handleFault()** メソッドのデフォルト実装も提供します。

開発者は **handleMessage()** メソッドの実装を提供する必要があります。**handleFault()** メソッドに異なる実装を指定することもできます。開発者が提供する実装は、汎用の org.apache.cxf.message.Message インターフェイスで提供されるメソッドを使用してメッセージデータを操作できます。

SOAP メッセージと動作するアプリケーションでは、Apache CXF は **AbstractSoapInterceptor** クラスを提供します。このクラスを拡張すると、**handleMessage()** メソッドと **handleFault()** メソッドは、**org.apache.cxf.binding.soap.SoapMessage** オブジェクトとしてメッセージデータにアクセスできるようになります。**SoapMessage** オブジェクトには、SOAP ヘッダー、SOAP エンベロープ、およびその他の SOAP メタデータをメッセージから取得するメソッドがあります。

第57章 インターセプターが呼び出されるタイミングの決定

概要

インターセプターはフェーズ別に編成されます。インターセプターが実行されるフェーズによって、メッセージデータのどの部分にアクセスできるかが決まります。インターセプターは、同じフェーズの他のインターセプターとの関係でその位置を判別できます。インターセプターのフェーズとフェーズ内のその場所は、インターセプターのコンストラクターロジックの一部として設定されます。

57.1. インターセプターの場所の指定

カスタムインターセプターを開発する場合に、まず考慮すべきことは、メッセージ処理チェーンのどこにがインターセプターが属するかです。開発者は、以下の 2 つの方法のいずれかで、メッセージ処理チェーンでインターセプターの位置を制御できます。

- インターセプターフェーズの指定
- フェーズ内のインターセプターの位置への制約指定

通常、インターセプターの場所を指定するコードはインターセプターのコンストラクターに配置されます。これにより、ランタイムは、アプリケーションレベルのコードで明示的なアクションを実行しなくても、インターセプターをインスタンス化してインターセプターチェーンの適切な場所に配置できます。

57.2. インターセプターのフェーズの指定

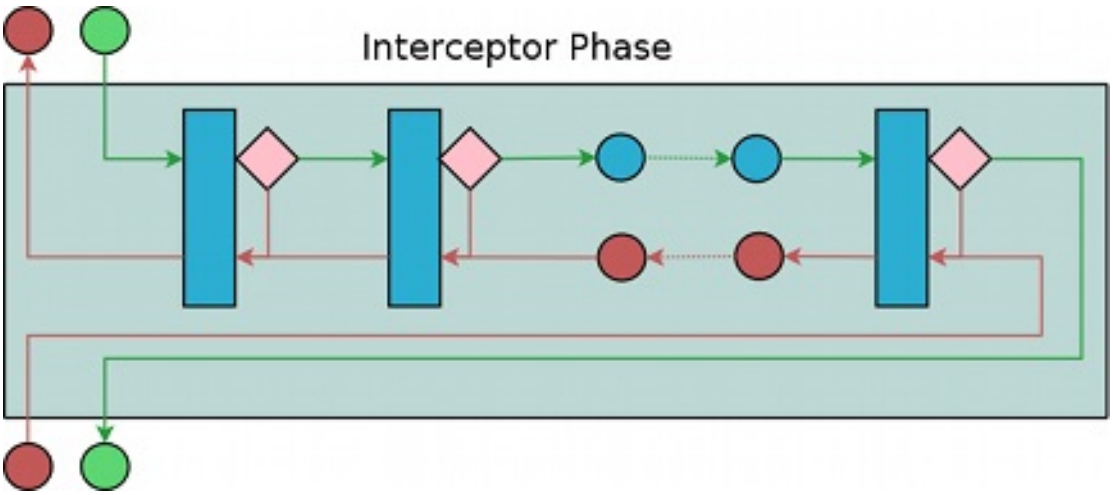
概要

インターセプターはフェーズ別に編成されます。インターセプターのフェーズは、呼び出されるメッセージ処理シーケンスのタイミングを決定します。開発者は、インターセプターのフェーズとそのコンストラクターを指定します。フェーズは、フレームワークによって提供される定数値を使用して指定されます。

フェーズ

フェーズはインターセプターの論理コレクションです。図57.1「インターセプターフェーズ」 で示されているように、フェーズ内のインターセプターは順次呼び出されます。

図57.1インターセプターフェーズ



フェーズは順序付けされたリストでリンクされ、インターセプターチェーンを形成すると共に、メッセージ処理手順で定義された論理ステップを提供します。たとえば、インバウンドインターセプターチェーンの **RECEIVE** フェーズにあるインターセプターのグループは、ネットワークから取得した RAW メッセージデータを使用してトランスポートレベルの詳細を処理します。

ただしどのフェーズでも、作業内容を強制的に実行することはできません。フェーズ内のインターセプターは、フェーズに含まれるタスクに準拠することが推奨されます。

Apache CXF によって定義されるフェーズの完全なリストは、62章*Apache CXF Message Processing フェーズ*を参照してください。

フェーズの指定

Apache CXF は、フェーズの指定に使用する **org.apache.cxf.Phase** クラスを提供します。クラスは定数のコレクションです。Apache CXF によって定義される各フェーズの **Phase** クラスに対応する定数があります。たとえば、RECEIVE フェーズは Phase.RECEIVE 値によって指定されます。

フェーズの設定

インターセプターのフェーズは、インターセプターのコンストラクターに設定されます。**AbstractPhaseInterceptor** クラスは、インターセプターをインスタンス化するためのコンストラクターを 3 つ定義します。

- **public AbstractPhaseInterceptor(String phase)** - インターセプターのフェーズを指定されたフェーズに設定し、インターセプターの ID をインターセプターのクラス名に自動的に設定します。
  このコンストラクターはほとんどのユースケースに対応します。

- **public AbstractPhaseInterceptor(String id, String phase)** - インターセプターの ID を最初のパラメーターとして渡された文字列に設定し、インターセプターのフェーズを 2 番目の文字列に設定します。

- **public AbstractPhaseInterceptor(String phase, boolean uniqueId)** - インターセプターが一意のシステム生成された ID を使用するかどうかを指定します。**uniqueId** パラメーターが **true** の場合、インターセプターの ID はシステムによって計算されます。**uniqueId** パラメーターが **false** の場合、インターセプターの ID はインターセプターのクラス名に設定されます。

カスタムインターセプターのフェーズの設定方法として、例57.1「インターセプターフェーズの設定」のように **super()** メソッドを使用して **AbstractPhaseInterceptor** コンストラクターにフェーズを渡すことが推奨されます。

例57.1 インターセプターフェーズの設定

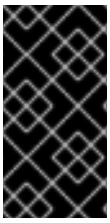
```
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
 {

    public StreamInterceptor()
   {
     super(Phase.PRE_STREAM);
    }
}
```

例57.1「インターセプターフェーズの設定」 に示す **StreamInterceptor** インターセプターは、PRE_STREAM フェーズに配置されます。

57.3. フェーズでのインターセプター配置の制限

概要

インターセプターをフェーズに配置すると、インターセプターが適切に機能できる程度に、その配置を十分に細かく制御できない場合があります。たとえば、SAAJ API を使用してメッセージの SOAP ヘッダーを検査するのに必要なインターセプターは、メッセージを SAAJ オブジェクトに変換するインターセプターの後に実行する必要があります。また、別のインターセプターに必要なメッセージの一部が、他のインターセプターにより消費される場合もあります。この場合、開発者はインターセプターよりも前に実行する必要があるインターセプターのリストを指定できます。開発者はインターセプターの後に実行する必要があるインターセプターのリストを指定することもできます。



重要

ランタイムでは、インターセプターのフェーズ内でのみ、このリストは受け入れられます。開発者が、現在のフェーズの後に実行する必要があるインターセプターのリストに前のフェーズのインターセプターを配置した場合には、ランタイムはその要求を無視します。

チェーンの事前追加

インターセプターの開発時に発生する問題の 1 つは、インターセプターが必要とするデータが常に存在するとは限らないことです。これは、チェーン内の 1 つのインターセプターが、後のインターセプターで必要となるメッセージデータを消費する場合に発生する可能性があります。開発者は、カスタムインターセプターの消費内容を制御し、インターセプターを変更して問題を修正できます。ただし、Apache CXF では多数のインターセプターが使用されており、開発者はそれらを変更できないため、これが常に可能であるとは限りません。

別の解決策として、カスタムインターセプターが必要とするメッセージデータを消費するインターセプターの前にカスタムインターセプターが配置されるようにすることです。これを行う最も簡単な方法は、初期のフェーズに配置することですが、それが常に可能であるとは限りません。あるインターセプターを 1 つ以上の他のインターセプターの前に配置する必要がある場合、Apache CXF の **AbstractPhaseInterceptor** クラスは 2 つの **addBefore()** メソッドを提供します。

例57.2「他のインターセプターの前にインターセプターを追加するメソッド」 のように、1 つは単一のインターセプター ID を取り、もう 1 つはインターセプター ID のコレクションを取ります。複数の呼び出しを実行して、リストへのインターセプターの追加を継続できます。

例57.2 他のインターセプターの前にインターセプターを追加するメソッド

public**addBeforeString**ipublic**addBeforeCollection<String>**i

例57.3「現在のインターセプターの後に実行する必要があるインターセプターのリストの指定」 にあるように、開発者はカスタムインターセプターのコンストラクターで **addBefore()** メソッドを呼び出します。

例57.3 現在のインターセプターの後に実行する必要があるインターセプターのリストの指定

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addBefore(HolderOutInterceptor.class.getName());
    }

...

}
```

ほとんどのインターセプターは、インターセプター ID にクラス名を使用します。

チェーンの事後追加

インターセプターに必要なデータが存在しないもう 1 つの理由は、データがメッセージオブジェクトに配置されていないことです。たとえば、インターセプターはメッセージデータを SOAP メッセージとして処理する場合がありますが、メッセージが SOAP メッセージに変換される前にチェーンに配置されている場合は機能しません。開発者は、カスタムインターセプターの消費内容を制御し、インターセプターを変更して問題を修正できます。ただし、Apache CXF では多数のインターセプターが使用されており、開発者はそれらを変更できないため、これが常に可能であるとは限りません。

別の解決策として、カスタムインターセプターが必要とするメッセージデータを消費するインターセプターの後にカスタムインターセプターが配置されるようにすることです。これを行う最も簡単な方法は、後のフェーズに配置することですが、それが常に可能であるとは限りません。**AbstractPhaseInterceptor** クラスは、他のインターセプターの後にインターセプターを配置する必要がある場合に 2 つの **addAfter()** メソッドを提供します。

例57.4「他のインターセプターの後にインターセプターを追加するメソッド」 のように、1 つのメソッドが単一のインターセプター ID を取り、もう 1 つのメソッドはインターセプター ID のコレクションを取ります。複数の呼び出しを実行して、リストへのインターセプターの追加を継続できます。

例57.4 他のインターセプターの後にインターセプターを追加するメソッド

public**addAfterStringi**public**addAfterCollection<String>i**

例57.5「現在のインターセプターの前に実行する必要があるインターセプターのリストの指定」 にあるように、開発者はカスタムインターセプターのコンストラクターで **addAfter()** メソッドを呼び出します。

例57.5 現在のインターセプターの前に実行する必要があるインターセプターのリストの指定

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addAfter(StartingOutInterceptor.class.getName());
    }
```

```
...

}
```

ほとんどのインターセプターは、インターセプター ID にクラス名を使用します。

第58章 インターセプター処理ログの実装

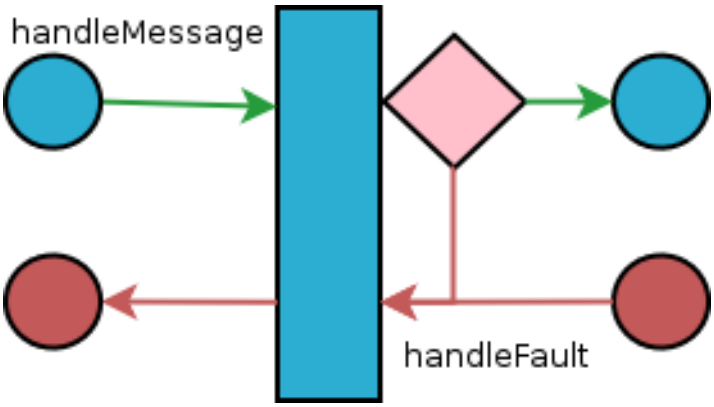
概要

インターセプターは実装が簡単です。処理ロジックの大部分は **handleMessage()** メソッドにあります。このメソッドはメッセージデータを受け取り、必要に応じて操作します。開発者は、障害処理のケースを扱う特別なロジックを追加することもできます。

58.1. インターセプターフロー

図58.1「インターセプター経由のフロー」 は、インターセプターを使用してプロセスフローを表示します。

図58.1 インターセプター経由のフロー



通常のメッセージ処理では、**handleMessage()** メソッドのみが呼び出されます。**handleMessage()** メソッドは、インターセプターのメッセージ処理ロジックを配置する場所です。

インターセプターまたはインターセプターチェーン内の後続のインターセプターの **handleMessage()** メソッドでエラーが発生した場合、**handleFault()** メソッドが呼び出されます。**handleFault()** メソッドは、エラーが発生した際にインターセプターの後にクリーンアップする場合に便利です。障害メッセージの変更にも使用できます。

58.2. メッセージの処理

概要

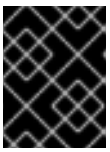
通常のメッセージ処理では、インターセプターの **handleMessage()** メソッドが呼び出されます。メッセージデータを Message オブジェクトとして受信します。メッセージの実際の内容とともに、Message オブジェクトには、メッセージやメッセージ処理の状態に関連する多くのプロパティーが含まれる場合があります。Message オブジェクトの正確な内容は、チェーンの現在のインターセプターの前にあるインターセプターによって異なります。

メッセージコンテンツの取得

Message インターフェイスには、メッセージの内容の抽出に使用できるメソッドが 2 つあります。

- public**<T> TgetContentjava.lang.Class<T> format getContent()** メソッドは指定されたクラスのオブジェクトにあるメッセージの内容を返します。指定されたクラスのインスタンスとしてコンテンツを利用できない場合は、null を返します。利用可能なコンテンツタイプのリストは、インターセプターチェーン上のインターセプターの場所とインターセプターチェーンの方向によって決まります。

- publicCollection<Attachment>getAttachments **getAttachments()** メソッドは、メッセージに関連するバイナリーアタッチメントを含む Java **Collection** オブジェクトを返します。添付ファイルは org.apache.cxf.message.Attachment オブジェクトに保存されます。attachment オブジェクトは、バイナリーデータを管理するメソッドを提供します。



重要

添付ファイルは、添付ファイルの処理インターセプターの実行後にのみ利用できます。

メッセージの方向の決定

メッセージの方向は、メッセージ交換を照会することで判断できます。メッセージ交換は、インバウンドメッセージとアウトバウンドメッセージを別のプロパティーに保存します。[3]

メッセージに関連付けられたメッセージエクスチェンジは、メッセージの **getExchange()** メソッドを使用して取得されます。例58.1「メッセージ交換の取得」にあるように、**getExchange()** はパラメーターを取らず、メッセージエクスチェンジを org.apache.cxf.message.Exchange オブジェクトとして返します。

例58.1 メッセージ交換の取得

ExchangegetExchange

Exchange オブジェクトには、例58.2「メッセージエクスチェンジからのメッセージの取得」 に示される 4 つのメソッドがあり、交換に関連するメッセージを取得します。各メソッドはメッセージを org.apache.cxf.Message オブジェクトとして返します。存在しない場合は null を返します。

例58.2 メッセージエクスチェンジからのメッセージの取得

MessagegetInMessageMessagegetInFaultMessageMessagegetOutMessageMessagegetOutFaultMessage

例58.3「証明書チェーンの方向の確認」 では、現在のメッセージがアウトバウンドであるかを判断するコードを紹介します。このメソッドはメッセージ交換を取得し、現在のメッセージが交換のアウトバウンドメッセージと同じかどうかを確認します。また、現在のメッセージを、アウトバウンド障害インターセプターチェーン上のエラーメッセージと交換するアウトバウンド障害メッセージと照合します。

例58.3 証明書チェーンの方向の確認

```
public static boolean isOutbound()
{
   Exchange exchange = message.getExchange();
       return message != null
             && exchange != null
             && (message == exchange.getOutMessage()
                || message == exchange.getOutFaultMessage());
}
```

例

例58.4「メッセージ処理メソッドの例」 は、zip 圧縮メッセージを処理するインターセプターのコードを示しています。メッセージの方向を確認してから、適切なアクションを実行します。

例58.4 メッセージ処理メソッドの例

```
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. エラー後の巻き戻し

概要

インターセプターチェーンの実行中にエラーが発生した場合、ランタイムはインターセプターチェーンのトラバースを停止し、すでに実行されたチェーンでインターセプターの **handleFault()** メソッドを呼び出すことでチェーンをアンワインドします。

handleFault() メソッドを使用して、通常のメッセージ処理中にインターセプターによって使用されるすべてのリソースをクリーンアップできます。また、メッセージ処理が正常に完了した場合にのみ有効なアクションをロールバックするために使用することもできます。障害メッセージがアウトバウンド障害処理のインターセプターチェーンに渡される場合、**handleFault()** メソッドを使用して情報を障害メッセージに追加することもできます。

メッセージペイロードの取得

handleFault() メソッドは、通常のメッセージ処理で使用される **handleMessage()** メソッドと同じ Message オブジェクトを受け取ります。Message オブジェクトからメッセージコンテンツを取得する方法は、「メッセージコンテンツの取得」 で説明します。

例

例58.5「巻き戻しインターセプターチェーンの処理」 は、インターセプターチェーンが巻き戻しされるときに、元の XML ストリームがメッセージに確実に戻るようにするために使用されるコードです。

例58.5 巻き戻しインターセプターチェーンの処理

```
@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] また、インバウンドおよびアウトバウンドの障害を個別に保存します。

第59章 インターセプターを使用するためのエンドポイントの設定

概要

インターセプターは、メッセージ交換に含まれる場合にはエンドポイントに追加されます。エンドポイントのインターセプターチェーンは、Apache CXF ランタイムの多数のコンポーネントのインターセプターチェーンをもとに構築されます。インターセプターは、エンドポイントの設定またはランタイムコンポーネントの 1 つの設定のいずれかで指定されます。インターセプターは、設定ファイルまたはインターセプター API を使用して追加できます。

59.1. インターセプターを割り当てる場所の決定

概要

インターセプターチェーンをホストするランタイムオブジェクトが多数あります。これには以下が含まれます。

- エンドポイントオブジェクト
- サービスオブジェクト
- プロキシーオブジェクト
- エンドポイントまたはプロキシーの作成に使用されるファクトリーオブジェクト
- バインディング
- 中央 **Bus** オブジェクト

開発者は独自のインターセプターをこれらのオブジェクトのいずれかに割り当てることができます。インターセプターを割り当てる最も一般的なオブジェクトはバスおよび個別のエンドポイントです。適切なオブジェクトを選択するには、これらのランタイムオブジェクトを組み合わせてエンドポイントを設定する方法を理解する必要があります。設計内容に準拠し、各 cxf 関連のバンドルには独自の cxf バスがあります。したがって、インターセプターがバスで設定され、同じ Blueprint コンテキストのサービスがインポートされたり、作成されると、インターセプターは処理されません。代わりに、インターセプターをインポートしたサービスの JAXWS クライアントまたはエンドポイントに直接設定できます。

エンドポイントおよびプロキシー

インターセプターを配置するための最もきめ細かい方法として、インターセプターをエンドポイントまたはプロキシーのいずれかに接続することが挙げられます。エンドポイントまたはプロキシーに直接接続されたインターセプターの影響があるのは、特定のエンドポイントまたはプロキシーだけです。これは、サービスの特定のインスタンスに固有のインターセプターに割り当てるのに適した場所です。たとえば、開発者が単位をメートル法からインペリアル法に変換するサービスのインスタンスを公開する場合に、インターセプターを 1 つのエンドポイントに直接接続できます。

ファクトリー

Spring 設定を使用して、エンドポイントまたはプロキシーの作成に使用されるファクトリーにインターセプターを割り当てると、インターセプターを直接エンドポイントまたはプロキシーに割り当てることと同じ効果があります。ただし、インターセプターがプログラム的にファクトリーに割り当てられると、ファクトリーに割り当てられたインターセプターはファクトリーで作成されたすべてのエンドポイントまたはプロキシーに伝播されます。

バインディング

バインディングにインターセプターをアタッチすると、開発者は、バインディングを使用する全エンドポイントに適用されるインターセプターをまとめて指定できます。たとえば、開発者が生の XML バインディングを使用するすべてのエンドポイントに特別な ID 要素を強制的に追加する場合に、XML バインディングへの要素を追加するインターセプターをアタッチできます。

バス

インターセプターを割り当てる最も一般的な場所はバスです。インターセプターがバスに割り当てられている場合に、インターセプターはそのバスで管理される全エンドポイントに伝播されます。インターセプターをバスに割り当てると、同様のインターセプターを共有する複数のエンドポイントを作成するアプリケーションでの使用に役立ちます。

割り当てポイントの統合

エンドポイントのインターセプターチェーンの最終セットは、リストされたオブジェクトで提供されるインターセプターチェーンの融合であるため、リストされたオブジェクトのいくつかを 1 つのエンドポイントの設定に統合できます。たとえば、アプリケーションが複数のエンドポイントを生成し、そのすべてが検証トークンをチェックするインターセプターを必要とする場合に、そのインターセプターはアプリケーションのバスに接続されます。これらのエンドポイントの 1 つでユーロをドルに変換するインターセプターも必要な場合は、変換インターセプターは特定のエンドポイントに直接割り当てられます。

59.2. 設定を使用したインターセプターの追加

概要

インターセプターをエンドポイントに割り当てる最も簡単な方法は、設定ファイルを使用することです。エンドポイントにアタッチされる各インターセプターは、標準の Spring Bean を使用して設定されます。その後、インターセプターの Bean は Apache CXF 設定要素を使用して適切なインターセプターチェーンに追加できます。

ランタイムコンポーネントに、関連付けられたインターセプターチェーンが含まれる場合には、特殊な Spring 要素を使用して設定変更ができます。コンポーネントの各要素には、インターセプターチェーンを指定するための標準的な子セットがあります。コンポーネントに関連付けられたインターセプターチェーンごとに 1 つの子があります。子は、チェーンに追加するインターセプターの Bean をリスト表示します。

設定要素

表59.1「インターセプターチェーン設定要素」 では、インターセプターをランタイムコンポーネントにアタッチするための 4 つの設定要素を説明します。

表59.1インターセプターチェーン設定要素

要素	説明
inInterceptors	エンドポイントのインバウンドインターセプターチェーンに追加するインターセプターを設定する Bean のリストが含まれます。

要素	説明
outInterceptors	エンドポイントのアウトバウンドインターセプターチェーンに追加するインターセプターを設定する Bean のリストが含まれます。
inFaultInterceptors	エンドポイントのインバウンド障害処理インターセプターチェーンに追加するインターセプターを設定する Bean のリストが含まれます。
outFaultInterceptors	エンドポイントのアウトバウンド障害処理インターセプターチェーンに追加するインターセプターを設定する Bean のリストが含まれます。

インターセプターチェーン設定要素はすべて **list** 子要素を取ります。**list** 要素には、チェーンにアタッチされるインターセプターごとに子が 1 つあります。インターセプターは、インターセプターを直接設定する **bean** 要素または、インターセプターを設定する **bean** 要素を参照する **ref** 要素を使用して指定できます。

例

例59.1「インターセプターのバスへの割り当て」 は、バスのインバウンドインターセプターチェーンにインターセプターを割り当てるための設定です。

例59.1 インターセプターのバスへの割り当て

```
&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「JAX-WS サービスプロバイダーへのインターセプターの割り当て」 は、インターセプターを JAX-WS サービスのアウトバウンドインターセプターチェーンに割り当てるための設定です。

例59.2 JAX-WS サービスプロバイダーへのインターセプターの割り当て

```
&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;
```

補足情報

Spring 設定を使用してエンドポイントを設定する方法は、パートIV「Web サービスエンドポイントの設定」 を参照してください。

59.3. プログラムによるインターセプターの追加

59.3.1. インターセプターの追加方法

インターセプターは、以下のいずれかの方法を使用してプログラム的にエンドポイントに割り当てることができます。

- InterceptorProvider API
- Java アノテーション

InterceptorProvider API を使用すると、開発者はインターセプターチェーンを持つ任意のランタイムコンポーネントにインターセプターを割り当てることができますが、基礎となる Apache CXF クラスを使用する必要があります。Java アノテーションはサービスインターフェイスまたはサービス実装にのみ追加できますが、開発者は JAX-WS API または JAX-RS API 内だけで作業を行えるようにできます。

59.3.2. インターセプタープロバイダー API の使用

概要

インターセプターは、インターセプタープロバイダーインターフェイス に示されている InterceptorProvider インターフェイスを実装するコンポーネントであればどれでも登録できます。

インターセプタープロバイダーインターフェイス

```
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();
}
```

インターフェイスの 4 つのメソッドにより、エンドポイントのインターセプターチェーンを Java **List** オブジェクトとして取得できます。Java **List** オブジェクトによって提供されるメソッドを使用することで、開発者は任意のチェーンにインターセプターを追加および削除できます。

手順

InterceptorProvider API を使用してインターセプターをランタイムコンポーネントのインターセプターチェーンに割り当てるには、以下を実行する必要があります。

1. インターセプターが割り当てられるチェーンを使用してランタイムコンポーネントへのアクセスを取得します。
   開発者は、Apache CXF 固有の API を使用して標準の Java アプリケーションコードからランタイムコンポーネントにアクセスする必要があります。ランタイムコンポーネントは通常、JAX-WS または JAX-RS アーティファクトを基盤の Apache CXF オブジェクトにキャストすることでアクセスできます。

2. インターセプターのインスタンスを作成します。

3. 適切な get メソッドを使用して、目的のインターセプターチェーンを取得します。

4. **List** オブジェクトの **add()** メソッドを使用してインターセプターをインターセプターチェーンにアタッチします。
   通常、このステップはインターセプターチェーンの取得と組み合わされます。

インターセプターをコンシューマーに取り付け

プログラムでコンシューマーにインターセプターを取り付け は、JAX-WS コンシューマーのインバウンドインターセプターチェーンにインターセプターを割り当てるコードです。

プログラムでコンシューマーにインターセプターを取り付け

```
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);

    ...
  }
}
```

プログラムでコンシューマーにインターセプターを取り付け のコードは、以下を行います。

コンシューマーの JAX-WS **Service** オブジェクトを作成します。

コンシューマーのターゲットアドレスを提供する **Service** オブジェクトにポートを追加します。

サービスプロバイダーでメソッドを呼び出すために使用されるプロキシーを作成します。

プロキシーを **org.apache.cxf.endpoint.Client** 型にキャストします。

インターセプターのインスタンスを作成します。

インターセプターをインバウンドインターセプターチェーンに割り当てます。

インターセプターのサービスプロバイダーへの割り当て

プログラムを使用したインターセプターのサービスプロバイダーへの割り当て は、サービスプロバイダーのアウトバウンドインターセプターチェーンにインターセプターを割り当てるコードです。

プログラムを使用したインターセプターのサービスプロバイダーへの割り当て

```
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);
  }
}
```

プログラムを使用したインターセプターのサービスプロバイダーへの割り当て のコードは、以下を行います。

基盤となる Apache CXF オブジェクトへのアクセスを提供する **ServerFactoryBean** オブジェクトを作成します。

Apache CXF がエンドポイントを表すために使用する **Server** オブジェクトを取得します。

サービスプロバイダーの Apache CXF **EndpointImpl** オブジェクトを取得します。

インターセプターのインスタンスを作成します。

エンドポイントのアウトバウンドインターセプターチェーンにインターセプターを割り当てます。

バスへのインターセプターの取り付け

バスへのインターセプターの取り付け は、バスのインバウンドインターセプターチェーンにインターセプターを割り当てるコードです。

バスへのインターセプターの取り付け

```
import org.apache.cxf.BusFactory;
org.apache.cxf.Bus;

...

Bus bus = BusFactory.getDefaultBus();

WatchInterceptor watchInterceptor = new WatchInterceptor();

bus..getInInterceptor().add(watchInterceptor);

...
```

バスへのインターセプターの取り付け のコードは、以下を行います。

ランタイムインスタンスのデフォルトのバスを取得します。

インターセプターのインスタンスを作成します。

インターセプターをインバウンドインターセプターチェーンに割り当てます。

WatchInterceptor は、ランタイムインスタンスによって作成されるすべてのエンドポイントのインバウンドインターセプターチェーンにアタッチされます。

59.3.3. Java アノテーションの使用

概要

Apache CXF は、開発者がエンドポイントによって使用されるインターセプターチェーンを指定できるように、4 つの Java アノテーションを提供します。インターセプターをエンドポイントに割り当てる他の手段とは異なり、アノテーションはアプリケーションレベルのアーティファクトに割り当てられます。使用するアーティファクトにより、アノテーションの影響範囲が決まります。

アノテーションを配置する場所

アノテーションは以下のアーティファクトに配置できます。

- エンドポイントを定義するサービスエンドポイントインターフェイス (SEI)
  アノテーションが SEI に配置されると、インターフェイスを実装するサービスプロバイダーすべてと、プロキシーの作成に SEI を使用するコンシューマーすべてに影響があります。
- サービス実装クラス
  アノテーションが実装クラスに配置されると、実装クラスを使用するすべてのサービスプロバイダーが影響を受けます。

アノテーション

アノテーションはすべて **org.apache.cxf.interceptor** パッケージにあり、表59.2「インターセプターチェーンアノテーション」 で説明されています。

表59.2 インターセプターチェーンアノテーション

Annotation	説明
InInterceptors	インバウンドインターセプターチェーンのインターセプターを指定します。
OutInterceptors	アウトバウンドインターセプターチェーンのインターセプターを指定します。
InFaultInterceptors	インバウンドフォールトインターセプターチェーンのインターセプターを指定します。
OutFaultInterceptors	アウトバウンド障害インターセプターチェーンのインターセプターを指定します。

インターセプターのリスト表示

インターセプターのリストは、チェーンアノテーションでインターセプターをリスト表示する構文 に記載されている構文を使用して、完全修飾クラス名のリストとして指定されます。

チェーンアノテーションでインターセプターをリスト表示する構文

```
interceptors={"interceptor1", "interceptor2", ..., "interceptorN"}
```

例

サービス実装へのインターセプターの割り当て は、**SayHiImpl** によって提供されるロジックを使用するエンドポイントのインバウンドインターセプターチェーンに 2 つのインターセプターをアタッチするアノテーションを示しています。

サービス実装へのインターセプターの割り当て

```
import org.apache.cxf.interceptor.InInterceptors;

@InInterceptors(interceptors={"com.sayhi.interceptors.FirstLast",
"com.sayhi.interceptors.LogName"})
public class SayHiImpl implements SayHi
{
  ...
}
```

第60章 オンザフライでのインターセプターチェーンの操作

概要

インターセプターは、エンドポイントのインターセプターチェーンをメッセージ処理ロジックの一部として再設定できます。新しいインターセプターの追加、インターセプターの削除、インターセプターの並べ替え、インターセプターの一時停止などを行うことができます。すべてのオンザフライ操作は呼び出し固有であるため、エンドポイントがメッセージ交換に関与するたびに元のチェーンが使用されます。

概要

インターセプターチェーンは、その作成のきっかけとなったメッセージ交換の間だけ存続します。各メッセージには、メッセージを処理するインターセプターチェーンへの参照が含まれています。開発者は、この参照を使用してメッセージのインターセプターチェーンを変更できます。チェーンは交換ごとに行われるため、メッセージのインターセプターチェーンに加えられた変更は他のメッセージ交換には影響しません。

チェーンライフサイクル

インターセプターチェーンとチェーン内のインターセプターは、呼び出しごとにインスタンス化されます。メッセージ交換に参加するエンドポイントが呼び出されると、必要なインターセプターチェーンがそのインターセプターのインスタンスと共にインスタンス化されます。インターセプターチェーンの作成の原因となったメッセージエクスチェンジが完了すると、チェーンとそのインターセプターインスタンスが破棄されます。

つまり、インターセプターチェーンまたはインターセプターのフィールドへの変更は、別のメッセージ交換には永続されません。そのため、インターセプターが別のインターセプターをアクティブなチェーンに配置すると、アクティブなチェーンのみが有効になります。今後のメッセージ交換は、エンドポイントの設定によって決定される、初期の状態から作成されます。また、開発者は今後のメッセージ処理を変更するインターセプターにフラグを設定できなくなります。

インターセプターが今後のインスタンスに情報を渡す必要がある場合は、メッセージコンテキストにプロパティーを設定できます。コンテキストは、メッセージエクスチェンジ全体で保持されます。

インターセプターチェーンの取得

メッセージのインターセプターチェーンを変更する最初のステップはインターセプターチェーンを取得します。これは、例60.1「インターセプターチェーンを取得するメソッド」 に記載されている **Message.getInterceptorChain()** メソッドを使用して行われます。インターセプターチェーンは org.apache.cxf.interceptor.InterceptorChain オブジェクトとして返されます。

> **例60.1 インターセプターチェーンを取得するメソッド**
>
> **InterceptorChaingetInterceptorChain**

インターセプターの追加

例60.2「インターセプターチェーンにインターセプターを追加するメソッド」 にあるように、InterceptorChain オブジェクトには、インターセプターチェーンにインターセプターを追加するためのメソッドが 2 つあります。1 つは単一のインターセプターを追加でき、もう 1 つは複数のインターセプターを追加できます。

例60.2 インターセプターチェーンにインターセプターを追加するメソッド

addInterceptor<? extends Message>iaddCollection<Interceptor<? extends Message>>i

例60.3「オンザフライでのインターセプターチェーンへのインターセプターの追加」 は、メッセージのインターセプターチェーンに単一のインターセプターを追加するためのコードです。

例60.3 オンザフライでのインターセプターチェーンへのインターセプターの追加

```
void handleMessage(Message message)
{
  ...
  AddledIntereptor addled = new AddledIntereptor();
  InterceptorChain chain = message.getInterceptorChain();
  chain.add(addled);
  ...
}
```

例60.3「オンザフライでのインターセプターチェーンへのインターセプターの追加」 のコードは、以下を行います。

チェーンに追加するインターセプターのコピーをインスタンス化します。

重要

チェーンに追加されるインターセプターは、現在のインターセプターと同じフェーズか、現在のインターセプターよりも後続フェーズのいずれかである必要があります。

現在のメッセージのインターセプターチェーンを取得します。

新しいインターセプターをチェーンに追加します。

インターセプターの削除

InterceptorChain オブジェクトには、例60.4「インターセプターチェーンからインターセプターを削除するメソッド」 にあるように、インターセプターチェーンからインターセプターを削除するためのメソッドが 1 つあります。

例60.4 インターセプターチェーンからインターセプターを削除するメソッド

removeInterceptor<? extends Message>i

例60.5「オンザフライでのインターセプターチェーンからのインターセプターの削除」 は、メッセージのインターセプターチェーンからインターセプターを削除するコードです。

例60.5 オンザフライでのインターセプターチェーンからのインターセプターの削除

```
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);
  }
  ...
}
```

ここで、**InterceptorClassName** は、チェーンから削除するインターセプターのクラス名です。

第61章 JAX-RS 2.0 フィルターおよびインターセプター

概要

JAX-RS 2.0 は、REST 呼び出しの処理パイプラインにフィルターとインターセプターをインストールする標準の API とセマンティクスを定義します。フィルターとインターセプタは通常、ロギング、認証、承認、メッセージ圧縮、メッセージ暗号化などの機能を提供するために使用されます。

61.1. JAX-RS フィルターおよびインターセプターの概要

概要

このセクションでは、JAX-RS フィルターおよびインターセプターの処理パイプラインを概説して、フィルターチェーンまたはインターセプターチェーンをインストールできる拡張ポイントに焦点を当てます。

Filters

JAX-RS 2.0 **フィルター** は、CXF クライアントまたはサーバーを通過するすべての JAX-RS メッセージへの開発者アクセスを許可するプラグインの一種です。フィルターは、メッセージに関連付けられたメタデータ (HTTP ヘッダー、クエリーパラメーター、メディアタイプ、その他のメタデータ) を処理するのに適しています。フィルターには、メッセージ呼び出しを中断する機能があります (セキュリティープラグインなどに役立ちます)。

必要に応じて、各拡張ポイントに複数のフィルターをインストールできます。その場合には、フィルターがチェーンで実行されます (ただし、インストールしたフィルターごとに **priority** の値が指定されていない限り、実行の順番は未定義のままです)。

インターセプター

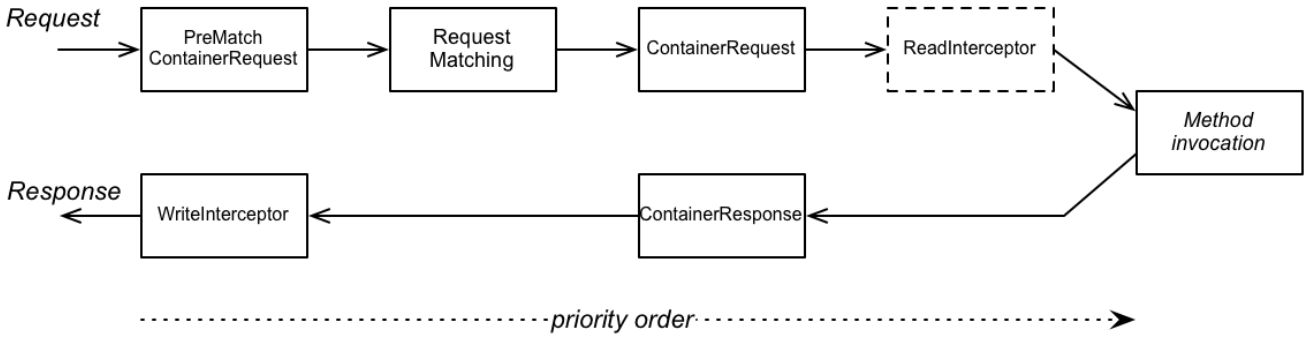
JAX-RS 2.0 **インターセプター** は、開発者が読み取りまたは書き込み中のメッセージボディーへのアクセス権限を付与するプラグインの一種です。インターセプターは、**MessageBodyReader.readFrom** メソッド呼び出し (リーダーインターセプターの場合) または **MessageBodyWriter.writeTo** メソッド呼び出し (ライターインターセプターの場合) のいずれかにラップされます。

必要に応じて、各拡張ポイントに複数のインターセプターをインストールできます。その場合には、インターセプターがチェーンで実行されます (ただし、インストールしたフィルターごとに **priority** の値が指定されていない限り、実行の順番は未定義のままです)。

サーバー処理パイプライン

図61.1「サーバー側のフィルターおよびインターセプター拡張ポイント」 は、サーバー側にインストールされている JAX-RS フィルターおよびインターセプターの処理パイプラインの概要を示しています。

図61.1 サーバー側のフィルターおよびインターセプター拡張ポイント



サーバー拡張ポイント

サーバー処理パイプラインでは、以下のエクステンションポイントのいずれかでフィルター (またはインターセプター) を追加できます。

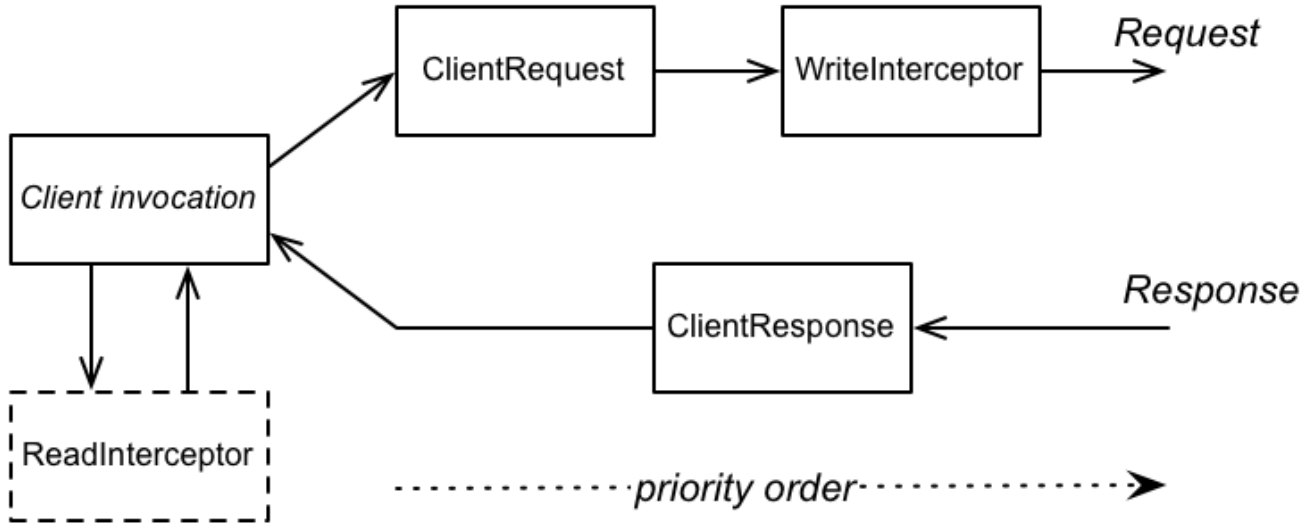
1. **PreMatchContainerRequest** フィルター
2. **ContainerRequest** フィルター
3. **ReadInterceptor**
4. **ContainerResponse** フィルター
5. **WriteInterceptor**

PreMatchContainerRequest エクステンションポイントは、リソース一致の発生 **前** に到達されるため、この時点で一部のコンテキストメタデータが使用できなくなることに注意してください。

クライアント処理パイプライン

図61.2「クライアント側のフィルターおよびインターセプター拡張ポイント」 は、クライアント側にインストールされている JAX-RS フィルターおよびインターセプターの処理パイプラインの概要を示しています。

図61.2 クライアント側のフィルターおよびインターセプター拡張ポイント



クライアント拡張ポイント

クライアント処理パイプラインでは、以下のエクステンションポイントのいずれかでフィルター (またはインターセプター) を追加できます。

1. **ClientRequest** フィルター
2. **WriteInterceptor**
3. **ClientResponse** フィルター
4. **ReadInterceptor**

フィルターおよびインターセプターの順序

複数のフィルターまたはインターセプターを同じ拡張ポイントにインストールする場合、フィルターの実行順序はそれらに割り当てられた優先順位 (Java ソースの **@Priority** アノテーションを使用) によって異なります。優先度は整数値として表されます。通常、優先度が **高い** フィルターはサーバー側でリソースメソッド呼び出しの近くに配置され、優先順位が **低い** フィルターはクライアント呼び出しの近くに配置されます。つまり、**要求** メッセージに作用するフィルターとインターセプターは、優先順位番号の **昇順** で実行されますが、**応答** メッセージに作用するフィルターとインターセプターは、優先順位番号の **降順** で実行されます。

フィルタークラス

以下の Java インターフェイスを実装して、カスタム REST メッセージフィルターを作成できます。

- **javax.ws.rs.container.ContainerRequestFilter**
- **javax.ws.rs.container.ContainerResponseFilter**
- **javax.ws.rs.client.ClientRequestFilter**
- **javax.ws.rs.client.ClientResponseFilter**

インターセプタークラス

カスタム REST メッセージインターセプターを作成するには、以下の Java インターフェイスを実装できます。

- **javax.ws.rs.ext.ReaderInterceptor**
- **javax.ws.rs.ext.WriterInterceptor**

61.2. コンテナー要求フィルター

概要

本セクションでは、**コンテナー要求フィルター**を実装して登録する方法を説明します。このフィルターは、サーバー (コンテナー) 側の受信要求メッセージをインターセプトするために使用されます。コンテナー要求フィルターは、サーバー側でヘッダーを処理するために使用されることが多く、あらゆる種類の汎用要求処理 (つまり、呼び出された特定のリソースメソッドから独立している処理) に使用できます。

さらに、コンテナーリクエストフィルターは、**PreMatchContainerRequest** (リソース一致ステップの前) と **ContainerRequest** (リソース一致ステップの後) の 2 つの異なる拡張ポイントにインストールできるため、特別なケースになります。

ContainerRequestFilter インターフェイス

javax.ws.rs.container.ContainerRequestFilter インターフェイスは以下のように定義されます。

```
// Java
...
package javax.ws.rs.container;

import java.io.IOException;

public interface ContainerRequestFilter {
    public void filter(ContainerRequestContext requestContext) throws IOException;
}
```

ContainerRequestFilter インターフェイスを実装することで、サーバー側で以下のエクステンションポイントのいずれかにフィルターを作成できます。

- **PreMatchContainerRequest**
- **ContainerRequest**

ContainerRequestContext インターフェイス

ContainerRequestFilter の **filter** メソッドは、**javax.ws.rs.container.ContainerRequestContext** 型の引数を 1 つ受け取り、これは受信リクエストメッセージとその関連メタデータにアクセスするために使用できます。**ContainerRequestContext** インターフェイスは以下のように定義されます。

```
// 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 フィルターの実装例

PreMatchContainerRequest エクステンションポイントのコンテナー要求フィルター (つまり、リソース一致の前にフィルターが実行される場合) を実装するには、**ContainerRequestFilter** インターフェイスを実装するクラスを定義し、クラスに **@PreMatching** アノテーションを付けます (**PreMatchContainerRequest** 拡張ポイントを選択するため)。

たとえば、以下のコードは、**PreMatchContainerRequest** エクステンションポイントにインストールされる単純なコンテナーリクエストフィルターの例を示しています。ここでは、優先度は 20 です。

```
// 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 フィルターの実装例

ContainerRequest エクステンションポイント (つまり、リソース一致の **後** にフィルターが実行される場合) のコンテナー要求フィルターを実装するには、**@PreMatching** アノテーション **なし** で **ContainerRequestFilter** インターフェイスを実装するクラスを定義します。

たとえば、以下のコードは、**ContainerRequest** エクステンションポイントにインストールされる単純なコンテナーリクエストフィルターの例を示しています。ここで、優先度は 30 になります。

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

ResourceInfo の注入

ContainerRequest エクステンションポイント (つまりリソース一致発生 **後**) では、**ResourceInfo** クラスを注入することで、一致したリソースクラスとリソースメソッドにアクセスできます。たとえば、以下のコードは、**ResourceInfo** クラスを **ContainerRequestFilter** クラスのフィールドとしてインジェクトする方法を示しています。

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

呼び出しの中止

コンテナー要求フィルターに適した実装を作成して、サーバー側の呼び出しを中止できます。通常、認証機能や承認機能を実装する場合など、これはサーバー側でセキュリティー機能を実装するのに役立ちます。受信要求が認証に失敗した場合には、コンテナー要求フィルター内から呼び出しを中止できます。

たとえば、以下の事前照合機能は URI のクエリーパラメーターからユーザー名とパスワードを抽出し、認証メソッドを呼び出してユーザー名とパスワードの認証情報を確認します。認証に失敗すると、**ContainerRequestContext** オブジェクトで **abortWith** を呼び出すことで呼び出しが中断され、クライアントに返されるエラー応答を渡します。

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

サーバー要求フィルターのバインド

サーバー要求フィルターを **バインド** する (Apache CXF ランタイムにインストールする) には、以下の手順を実行します。

1. 以下のコードフラグメントで示されるように、**@Provider** アノテーションをコンテナーリクエストフィルタークラスに追加します。

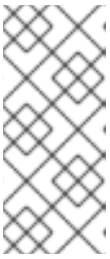
```
// Java
package org.jboss.fuse.example;

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 {
  ...
}
```

コンテナーリクエストフィルター実装が Apache CXF ランタイムにロードされると、REST 実装はロードされたクラスを自動的にスキャンし、**@Provider** アノテーション (**スキャンフェーズ**) の付いたクラスを検索します。

2. XML で JAX-RS サーバーエンドポイントを定義する場合 (例: 「JAX-RS サーバーエンドポイントの設定」)、**jaxrs:providers** 要素のプロバイダーリストにサーバー要求フィルターを追加します。

```
<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>
```



注記

この手順は、Apache CXF の非標準要件です。厳密に言うと、JAX-RS 標準によれば、フィルターをバインドするために必要なのは **@Provider** アノテーションのみです。しかし実際には、標準的なアプローチはやや柔軟性がなく、大規模なプロジェクトに多くのライブラリーが含まれている場合は、プロバイダーの衝突につながる可能性があります。

61.3. コンテナー応答フィルター

概要

本セクションでは、**コンテナー応答フィルター**を実装して登録する方法を説明します。このフィルターは、サーバー側で送信応答メッセージをインターセプトするために使用されます。コンテナー応答フィルターは、応答メッセージでのヘッダーの自動設定に使用でき、通常、あらゆる種類の汎用的な応答処理に使用できます。

ContainerResponseFilter インターフェイス

javax.ws.rs.container.ContainerResponseFilter インターフェイスは以下のように定義されます。

```
// Java
```

```
...
package javax.ws.rs.container;

import java.io.IOException;

public interface ContainerResponseFilter {
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext
responseContext)
            throws IOException;
}
```

ContainerResponseFilter を実装することで、サーバー側で **ContainerResponse** エクステンションポイントのフィルターを作成できます。これは、呼び出しの実行 **後** にレスポンスメッセージをフィルタリングします。

注記

コンテナーレスポンスフィルターでは、リクエストメッセージ (**requestContext** 引数経由) とレスポンスメッセージ (**responseContext** メッセージ経由) の両方にアクセスできますが、この段階ではレスポンスのみを変更できます。

ContainerResponseContext インターフェイス

ContainerResponseFilter の **filter** メソッドは、型 **javax.ws.rs.container.ContainerRequestContext** の引数 (「ContainerRequestContext インターフェイス」を参照) と型 **javax.ws.rs.container.ContainerResponseContext** の引数 (送信レスポンスメッセージとその関連データにアクセスするために使用できる) の 2 つの引数を受け取ります。

ContainerResponseContext インターフェイスは以下のように定義されます。

```
// 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);
}
```

サンプル実装

ContainerResponse エクステンションポイントのコンテナーレスポンスフィルター (つまり、呼び出しがサーバー側に実行された後にフィルターが実行された場合) を実装するには、**ContainerResponseFilter** インターフェイスを実装するクラスを定義します。

たとえば、以下のコードは、**ContainerResponse** エクステンションポイントにインストールされる単純なコンテナーリクエストフィルターの例を示しています。ここで、優先度は 10 になります。

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

サーバーの応答フィルターのバインド

サーバー応答フィルターを **バインド** する (Apache CXF ランタイムにインストールする) には、以下の手順を実行します。

1. 以下のコードフラグメントで示されるように、**@Provider** アノテーションをコンテナーレスポンスフィルタークラスに追加します。

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

コンテナーレスポンスフィルター実装が Apache CXF ランタイムにロードされると、REST 実装はロードされたクラスを自動的にスキャンし、**@Provider** アノテーション (**スキャンフェーズ**) の付いたクラスを検索します。

2. XML で JAX-RS サーバーエンドポイントを定義する場合 (例: 「JAX-RS サーバーエンドポイントの設定」)、**jaxrs:providers** 要素のプロバイダーリストにサーバーレスポンスフィルターを追加します。

```
<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>
```

注記

この手順は、Apache CXF の非標準要件です。厳密に言うと、JAX-RS 標準によれば、フィルターをバインドするために必要なのは **@Provider** アノテーションのみです。しかし実際には、標準的なアプローチはやや柔軟性がなく、大規模なプロジェクトに多くのライブラリーが含まれている場合は、プロバイダーの衝突につながる可能性があります。

61.4. クライアント要求フィルター

概要

本セクションでは、**クライアント要求フィルター**を実装して登録する方法を説明します。このフィルターは、クライアント側で送信要求メッセージをインターセプトするために使用されます。クライアント要求フィルターはヘッダーの処理によく使用され、あらゆる種類の汎用的な要求処理に使用できま

す。

ClientRequestFilter インターフェイス

javax.ws.rs.client.ClientRequestFilter インターフェイスは以下のように定義されます。

```
// 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;
}
```

ClientRequestFilter を実装することにより、クライアント側に **ClientRequest** エクステンションポイントのフィルターを作成できます。これは、メッセージをサーバーに送信する前にリクエストメッセージをフィルタリングします。

ClientRequestContext インターフェイス

ClientRequestFilter の **filter** メソッドは、**javax.ws.rs.client.ClientRequestContext** 型の引数を 1 つ受け取り、これは送信リクエストメッセージとその関連メタデータにアクセスするために使用できます。**ClientRequestContext** インターフェイスは以下のように定義されます。

```
// 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);
}
```

サンプル実装

ClientRequest エクステンションポイント (リクエストメッセージを送信する前にフィルターが実行される場所) にクライアントリクエストフィルターを実装するには、**ClientRequestFilter** インターフェイスを実装するクラスを定義します。

たとえば、以下のコードは、**ClientRequest** エクステンションポイントにインストールされる単純なクライアントリクエストフィルターの例を示しています。ここで、優先度は 20 になります。

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

呼び出しの中止

適切なクライアント要求フィルターを実装することで、クライアント側の呼び出しを中止できます。たとえば、クライアント側のフィルターを実装して、要求が正しくフォーマットされているかどうかを確認し、必要に応じて要求を中止できます。

以下のテストコードは**常に** リクエストを中止し、**BAD_REQUEST** HTTP ステータスをクライアント呼び出しコードに返します。

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

クライアント要求フィルターの登録

JAX-RS 2.0 クライアント API を使用すると、クライアントリクエストフィルターを **javax.ws.rs.client.Client** オブジェクトまたは **javax.ws.rs.client.WebTarget** オブジェクトに直接登録できます。そのため、クライアント要求フィルターはオプションで異なるスコープに適用できるので、このフィルターの影響を受けるのは、特定の URI パスのみです。

たとえば、次のコードは、**client** オブジェクトを使用して行われたすべての呼び出しに適用されるように **SampleClientRequestFilter** フィルターを登録する方法、および **rest/TestAbortClientRequest** のサブパスにのみ適用されるように **TestAbortClientRequestFilter** フィルターを登録する方法を示しています。

```
// 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. クライアント応答フィルター

概要

本セクションでは、**クライアント応答フィルター** を実装して登録する方法を説明します。これは、クライアント側で受信応答メッセージをインターセプトするために使用されます。クライアント応答フィルターは、クライアント側でのあらゆる種類の汎用的な応答処理に使用できます。

ClientResponseFilter インターフェイス

javax.ws.rs.client.ClientResponseFilter インターフェイスは以下のように定義されます。

```
// Java
package javax.ws.rs.client;
...
import java.io.IOException;

public interface ClientResponseFilter {
    void filter(ClientRequestContext requestContext, ClientResponseContext responseContext)
            throws IOException;
}
```

ClientResponseFilter を実装することで、クライアント側で **ClientResponse** エクステンションポイントのフィルターを作成できます。これは、サーバーから受信した後にレスポンスメッセージをフィルタリングします。

ClientResponseContext インターフェイス

ClientResponseFilter の **filter** メソッドは、型 **javax.ws.rs.client.ClientRequestContext** の引数 (「ClientRequestContext インターフェイス」を参照) と型 **javax.ws.rs.client.ClientResponseContext** の引数 (送信レスポンスメッセージとその関連データにアクセスするために使用できる) の 2 つの引数を受け取ります。

ClientResponseContext インターフェイスは以下のように定義されます。

```
// 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);
}
```

サンプル実装

ClientResponse エクステンションポイントのクライアントレスポンスフィルター (つまり、サーバーからレスポンスメッセージを受信した後にフィルターが実行された場合) を実装するには、**ClientResponseFilter** インターフェイスを実装するクラスを定義します。

たとえば、以下のコードは、**ClientResponse** エクステンションポイントにインストールされる単純なクライアントレスポンスフィルターの例を示しています。ここで、優先度は 20 になります。

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

クライアント応答フィルターの登録

JAX-RS 2.0 クライアント API を使用すると、クライアントレスポンスフィルターを **javax.ws.rs.client.Client** オブジェクトまたは **javax.ws.rs.client.WebTarget** オブジェクトに直接登録できます。そのため、クライアント要求フィルターはオプションで異なるスコープに適用できるので、このフィルターの影響を受けるのは、特定の URI パスのみです。

たとえば、以下のコードは、**client** オブジェクトを使用して実行されたすべての呼び出しに適用されるように **SampleClientResponseFilter** フィルターを登録する方法を示しています。

```
// 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. エンティティーリーダーインターセプター

概要

本セクションでは、**エンティティーリーダーインターセプター** を実装し、登録する方法を説明します。これにより、クライアント側またはサーバー側でメッセージボディーを読み取る際に入力ストリームをインターセプトできます。これは通常、暗号化や復号などの要求ボディーの汎用的な変換や圧縮や圧縮解除に役立ちます。

ReaderInterceptor インターフェイス

javax.ws.rs.ext.ReaderInterceptor インターフェイスは以下のように定義されます。

```
// Java
...
package javax.ws.rs.ext;

public interface ReaderInterceptor {
    public Object aroundReadFrom(ReaderInterceptorContext context)
            throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

ReaderInterceptor インターフェイスを実装することで、メッセージボディー (**Entity** オブジェクト) をサーバー側またはクライアント側で読み取る際にインターセプトできます。エンティティーリーダーインターセプターは、以下のいずれかのコンテキストで使用できます。

- **サーバー側**: サーバー側のインターセプターとしてバインドされた場合に、エンティティーリーダーインターセプターは、(一致したリソース内の) アプリケーションコードによってアクセス

されたタイミングで、要求メッセージのボディーをインターセプトします。REST 要求のセマンティクスによっては、一致したリソースがメッセージ本文にアクセスできない場合があり、そのような場合にはリーダーインターセプターは呼び出されません。

- **クライアント側**: クライアント側のインターセプターとしてバインドされた場合に、エンティティーリーダーインターセプターは、クライアントコードによってアクセスされたタイミングで応答メッセージのボディーをインターセプトします。クライアントコードが明示的にレスポンスメッセージにアクセスしない場合 (**Response.getEntity** メソッドの呼び出しなど)、リーダーインターセプターは呼び出されません。

ReaderInterceptorContext インターフェイス

ReaderInterceptor の **aroundReadFrom** メソッドは、メッセージボディー (**Entity** オブジェクト) とメッセージメタデータの両方にアクセスするために使用できる **javax.ws.rs.ext.ReaderInterceptorContext** 型の引数を 1 つ受け取ります。

ReaderInterceptorContext インターフェイスは以下のように定義されます。

```
// 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 インターフェイス

ReaderInterceptorContext インターフェイスは、ベース **InterceptorContext** インターフェイスから継承されたメソッドもサポートします。

InterceptorContext インターフェイスは以下のように定義されます。

```
// 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);
}
```

クライアント側での実装例

クライアント側にエンティティーリーダーインターセプターを実装するには、**ReaderInterceptor** インターフェイスを実装するクラスを定義します。

たとえば、以下のコードは、クライアント側のエンティティーリーダーインターセプター (優先度が 10) の例を示しています。これは、受信応答のメッセージボディーで **COMPANY_NAME** の全インスタンスを **Red Hat** に置き換えます。

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

サーバー側での実装例

サーバー側のエンティティーリーダーインターセプターを実装するには、**ReaderInterceptor** インターフェイスを実装し、**@Provider** アノテーションを付けるクラスを定義します。

たとえば、以下のコードは、サーバー側のエンティティーリーダーインターセプター (優先度が 10) の例を示しています。これは、受信リクエストのメッセージボディーで **COMPANY_NAME** の全インスタンスを **Red Hat** に置き換えます。

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

クライアント側でのリーダーインターセプターのバインド

JAX-RS 2.0 クライアント API を使用すると、エンティティーリーダーインターセプターを

javax.ws.rs.client.Client オブジェクトまたは **javax.ws.rs.client.WebTarget** オブジェクトに直接登録できます。実質的に、リーダーインターセプターはオプションで異なるスコープに適用できるため、インターセプターの影響を受けるのは、特定の URI パスのみです。

たとえば、以下のコードは、**client** オブジェクトを使用して実行されたすべての呼び出しに適用されるように **SampleClientReaderInterceptor** インターセプターを登録する方法を示しています。

```
// 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);
```

JAX-RS 2.0 クライアントでインターセプターを登録する方法は、「クライアントエンドポイントの設定」 を参照してください。

サーバー側でのリーダーインターセプターのバインド

サーバー側でリーダーインターセプターを **バインド** する (つまり Apache CXF ランタイムにインストールする) には、以下の手順を実行します。

1. 以下のコードフラグメントで示されるように、**@Provider** アノテーションをリーダーインターセプタークラスに追加します。

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

   リーダーインターセプター実装が Apache CXF ランタイムにロードされると、REST 実装はロードされたクラスを自動的にスキャンし、**@Provider** アノテーション (**スキャンフェーズ**) の付いたクラスを検索します。

2. XML で JAX-RS サーバーエンドポイントを定義する場合 (例: 「JAX-RS サーバーエンドポイントの設定」)、**jaxrs:providers** 要素のプロバイダーリストにリーダーインターセプターを追加します。

   ```
   <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>
```

注記

この手順は、Apache CXF の非標準要件です。厳密に言うと、JAX-RS 標準によれば、インターセプターをバインドするために必要なのは **@Provider** アノテーションのみです。しかし実際には、標準的なアプローチはやや柔軟性がなく、大規模なプロジェクトに多くのライブラリーが含まれている場合は、プロバイダーの衝突につながる可能性があります。

61.7. エンティティーライターインターセプター

概要

本セクションでは、**エンティティーライターインターセプター** を実装し、登録する方法を説明します。これにより、クライアント側またはサーバー側でメッセージボディーを書き込む際に出力ストリームをインターセプトできます。これは通常、暗号化や復号などの要求ボディーの汎用的な変換や圧縮や圧縮解除に役立ちます。

WriterInterceptor インターフェイス

javax.ws.rs.ext.WriterInterceptor インターフェイスは以下のように定義されます。

```
// Java
...
package javax.ws.rs.ext;

public interface WriterInterceptor {
    void aroundWriteTo(WriterInterceptorContext context)
            throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

WriterInterceptor インターフェイスを実装することで、メッセージボディー (**Entity** オブジェクト) をサーバー側またはクライアント側で書き込む際にインターセプトできます。エンティティーライターインターセプターは、以下のいずれかのコンテキストで使用できます。

- **サーバー側**: サーバー側のインターセプターとしてバインドされた場合に、エンティティーライターインターセプターは、マーシャリングされ、クライアントに送信される直前に応答メッセージボディーをインターセプトします。

- **クライアント側**: クライアント側のインターセプターとしてバインドされた場合に、エンティティーライターインターセプターは、マーシャリングされてサーバーに送信される直前に要求メッセージのボディーをインターセプトします。

WriterInterceptorContext インターフェイス

WriterInterceptor の **aroundWriteTo** メソッドは、メッセージボディー (**Entity** オブジェクト) とメッセージメタデータの両方にアクセスするために使用できる **javax.ws.rs.ext.WriterInterceptorContext** 型の引数を 1 つ受け取ります。

WriterInterceptorContext インターフェイスは以下のように定義されます。

```
// 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 インターフェイス

WriterInterceptorContext インターフェイスは、ベース **InterceptorContext** インターフェイスから継承されたメソッドもサポートします。**InterceptorContext** の定義については、「InterceptorContext インターフェイス」を参照してください。

クライアント側での実装例

クライアント側にエンティティーライターインターセプターを実装するには、**WriterInterceptor** インターフェイスを実装するクラスを定義します。

たとえば、以下のコードは、クライアント側 (優先度が 10) のエンティティーライターインターセプターの例を示しています。これは、送信リクエストのメッセージボディーにテキストの行を余分に追加します。

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

サーバー側での実装例

サーバー側のエンティティーライターインターセプターを実装するには、**WriterInterceptor** インターフェイスを実装し、**@Provider** アノテーションを付けるクラスを定義します。

たとえば、以下のコードは、サーバー側 (優先度が 10) のエンティティーライターインターセプターの例を示しています。これは、送信要求のメッセージボディーにテキストの行を余分に追加します。

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

クライアント側のライターインターセプターのバインド

JAX-RS 2.0 クライアント API を使用すると、エンティティーライターインターセプターを **javax.ws.rs.client.Client** オブジェクトまたは **javax.ws.rs.client.WebTarget** オブジェクトに直接登録できます。実質的に、ライターインターセプターはオプションで異なるスコープに適用できるため、インターセプターの影響を受けるのは、特定の URI パスのみです。

たとえば、以下のコードは、**client** オブジェクトを使用して実行されたすべての呼び出しに適用されるように **SampleClientReaderInterceptor** インターセプターを登録する方法を示しています。

```
// 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);
```

JAX-RS 2.0 クライアントでインターセプターを登録する方法は、「クライアントエンドポイントの設定」 を参照してください。

サーバー側でのライターインターセプターのバインド

サーバー側でライターインターセプターを **バインド** する (Apache CXF ランタイムにインストールする) には、以下の手順を実行します。

1. 以下のコードフラグメントで示されるように、**@Provider** アノテーションをライターインターセプタークラスに追加します。

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

ライターインターセプター実装が Apache CXF ランタイムにロードされると、REST 実装はロードされたクラスを自動的にスキャンし、**@Provider** アノテーション (**スキャンフェーズ**) の付いたクラスを検索します。

2. XML で JAX-RS サーバーエンドポイントを定義する場合 (例: 「JAX-RS サーバーエンドポイントの設定」)、**jaxrs:providers** 要素のプロバイダーリストにライターインターセプターを追加します。

```
<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>
```

注記

この手順は、Apache CXF の非標準要件です。厳密に言うと、JAX-RS 標準によれば、インターセプターをバインドするために必要なのは **@Provider** アノテーションのみです。しかし実際には、標準的なアプローチはやや柔軟性がなく、大規模なプロジェクトに多くのライブラリーが含まれている場合は、プロバイダーの衝突につながる可能性があります。

61.8. 動的バインディング

概要

コンテナーフィルターとコンテナーインターセプターをリソースにバインディングする標準的な方法では、フィルターとインターセプターに **@Provider** アノテーションを付けます。これにより、バインディングは **グローバル**: になります。つまり、フィルターとインターセプターはサーバー側の **すべて** のリソースクラスおよびリソースメソッドにバインドされます。

動的バインディングはサーバー側でバインディングする別の方法で、インターセプターおよびフィルターが適用されるリソースメソッドを選択できます。フィルターおよびインターセプターの動的バインディングを有効にするには、以下に記載されているように、カスタム **DynamicFeature** インターフェイスを実装する必要があります。

DynamicFeature インターフェイス

DynamicFeature インターフェイスは、以下のように **javax.ws.rx.container** パッケージで定義されます。

```
// 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);
}
```

動的機能の実装

以下のように動的機能を実装します。

1. 上記のように、1つ以上のコンテナーフィルターまたはコンテナーインターセプターを実装します。ただし、**@Provider** アノテーションを**付けないでください** (そうしないと、グローバルにバインドされ、動的機能が事実上無関係になります)。
2. **DynamicFeature** クラスを実装して、**configure** メソッドをオーバーライドすることで、独自の動的な機能を作成します。
3. **configure** メソッドでは、**resourceInfo** 引数を使用して、この機能が呼び出されているリソースクラスとリソースメソッドを検出できます。この情報は、フィルターまたはインターセプターの一部を登録するかどうかを決定するベースとして使用できます。
4. フィルターまたはインターセプターを現在のリソースメソッドに登録する場合は、**context.register** メソッドのいずれかを呼び出します。
5. 動的機能クラスに **@Provider** アノテーションを付けて、デプロイメントのスキャンフェーズ中にこれが選択されるようにします。

動的機能の例

以下の例は、**@GET** アノテーションが付けられた **MyResource** クラス (またはサブクラス) のメソッドに **LoggingFilter** フィルターを登録する動的な機能を定義する方法を示しています。

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

動的バインディングプロセス

JAX-RS 標準では、**DynamicFeature.configure** メソッドを **リソースメソッドごとに1度だけ呼び出す** 必要があります。つまり、すべてのリソースメソッドは動的機能によってフィルターまたはインターセ

プターをインストールする可能性がありますが、動的機能を使用して、それぞれの場合にフィルターまたはインターセプターを登録するかどうかを決定します。つまり、動的機能でサポートされるバインディングは、個々のリソースメソッドのレベルで設定されます。

FeatureContext インターフェイス

FeatureContext インターフェイス (**configure** メソッドでフィルターおよびインターセプターを登録可能) は、以下のように **Configurable<>** のサブインターフェイスとして定義されます。

```
// Java
package javax.ws.rs.core;

public interface FeatureContext extends Configurable<FeatureContext> {
}
```

Configurable<> インターフェイスは、以下のようにフィルターおよびインターセプターを 1 つのリソースメソッドに登録するさまざまなメソッドを定義します。

```
// 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 MESSAGE PROCESSING フェーズ

インバウンドフェーズ

表62.1「インバウンドメッセージ処理フェーズ」 は、インバウンドインターセプターチェーンで利用可能なフェーズをリスト表示します。

表62.1インバウンドメッセージ処理フェーズ

フェーズ	説明
RECEIVE	バイナリー割り当ての MIME 境界の決定など、トランスポート固有の処理を実行します。
PRE_STREAM	トランスポートによって受信される生のデータストリームを処理します。
USER_STREAM	
POST_STREAM	
READ	要求が SOAP または XML メッセージであるかどうかを判別し、ビルドによって適切なインターセプターが追加されます。SOAP メッセージヘッダーもこのフェーズで処理されます。
PRE_PROTOCOL	プロトコルレベルの処理を実行します。これには、WS-* ヘッダーの処理や SOAP メッセージプロパティーの処理が含まれます。
USER_PROTOCOL	
POST_PROTOCOL	
UNMARSHAL	メッセージデータをアプリケーションレベルのコードで使用されるオブジェクトにアンマーシャリングします。
PRE_LOGICAL	アンマーシャリングされたメッセージデータを処理します。
USER_LOGICAL	
POST_LOGICAL	
PRE_INVOKE	

フェーズ	説明
INVOKE	メッセージをアプリケーションコードに渡します。サーバー側では、このフェーズでサービス実装が呼び出されます。クライアント側では、応答がアプリケーションに戻されます。
POST_INVOKE	アウトバウンドインターセプターチェーンを呼び出します。

アウトバウンドフェーズ

表62.2「インバウンドメッセージ処理フェーズ」 は、インバウンドインターセプターチェーンで利用可能なフェーズをリスト表示します。

表62.2 インバウンドメッセージ処理フェーズ

フェーズ	説明
SETUP	チェーン内の後続のフェーズで必要な設定を実行します。
PRE_LOGICAL	アプリケーションレベルから渡されたアンマーシャリングデータで処理を実行します。
USER_LOGICAL	
POST_LOGICAL	
PREPARE_SEND	ワイヤリング上のメッセージを書き込むための接続を開きます。
PRE_STREAM	データストリームへのエントリーの準備に必要な処理を実行します。
PRE_PROTOCOL	プロトコル固有の情報の処理を開始します。
WRITE	プロトコルメッセージを書き込みます。
PRE_MARSHAL	メッセージをマーシャリングします。
MARSHAL	
POST_MARSHAL	
USER_PROTOCOL	プロトコルメッセージを処理します。
POST_PROTOCOL	

フェーズ	説明
USER_STREAM	バイトレベルのメッセージを処理します。
POST_STREAM	
SEND	メッセージを送信し、トランスポートストリームを閉じます。

重要

アウトバウンドインターセプターチェーンには、名前の末尾に **_ENDING** が付いた終了フェーズのミラーセットがあります。終了フェーズは、データがワイヤリングに書き込まれる前に何らかの終了アクションが必要なインターセプターとして使用します。

第63章 APACHE CXF が提供するインターセプター

63.1. コア APACHE CXF インターセプター

受信

表63.1「コアインバウンドインターセプター」 は、すべての Apache CXF エンドポイントに追加されるコアインバウンドインターセプターをリスト表示します。

表63.1 コアインバウンドインターセプター

クラス	フェーズ	説明
ServiceInvokerInterceptor	**INVOKE**	サービスに適切なメソッドを呼び出します。

Outbound

Apache CXF ではデフォルトで、コアインターセプターチェーンにコアインターセプターは追加されません。エンドポイントのアウトバウンドインターセプターチェーンの内容は、使用中の機能によって異なります。

63.2. フロントエンド

JAX-WS

表63.2「インバウンド JAX-WS インターセプター」 は、JAX-WS エンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.2 インバウンド JAX-WS インターセプター

クラス	フェーズ	説明
HolderInInterceptor	**PRE_INVOKE**	メッセージ内の out または in/out パラメーターのホルダーオブジェクトを作成します。
WrapperClassInInterceptor	**POST_LOGICAL**	ラップされたドキュメント/リテラルメッセージの一部をオブジェクトの適切な配列にアンラップします。
LogicalHandlerInInterceptor	**PRE_PROTOCOL**	エンドポイントによって使用される JAX-WS 論理ハンドラーにメッセージ処理を渡します。JAX-WS ハンドラーが完了すると、メッセージはインバウンドチェーンの次のインターセプターに渡されます。

クラス	フェーズ	説明
SOAPHandlerInterceptor	**PRE_PROTOCOL**	エンドポイントによって使用される JAX-WS SOAP ハンドラーにメッセージ処理を渡します。SOAP ハンドラーがメッセージを完了すると、メッセージはチェーン内の次のインターセプターに渡されます。

表63.3「アウトバウンド JAX-WS インターセプター」 は、JAX-WS エンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.3 アウトバウンド JAX-WS インターセプター

クラス	フェーズ	説明
HolderOutInterceptor	**PRE_LOGICAL**	ホルダーオブジェクトから out および in/out パラメーターの値を削除し、その値をメッセージのパラメーターリストに追加します。
WebFaultOutInterceptor	**PRE_PROTOCOL**	アウトバウンド障害メッセージを処理します。
WrapperClassOutInterceptor	**PRE_LOGICAL**	メッセージに追加される前に、ラップされたドキュメント/リテラルメッセージと rpc/literal メッセージが適切にラップされていることを確認します。
LogicalHandlerOutInterceptor	**PRE_MARSHAL**	エンドポイントによって使用される JAX-WS 論理ハンドラーにメッセージ処理を渡します。JAX-WS ハンドラーが完了すると、メッセージはアウトバウンドチェーンの次のインターセプターに渡されます。
SOAPHandlerInterceptor	**PRE_PROTOCOL**	エンドポイントによって使用される JAX-WS SOAP ハンドラーにメッセージ処理を渡します。SOAP ハンドラーがメッセージの処理を終了すると、チェーン内の次のインターセプターに渡されます。

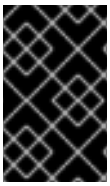
クラス	フェーズ	説明
MessageSenderInterceptor	**PREPARE_SEND**	Destination オブジェクトにコールバックして、出力ストリームやヘッダーなどを設定し、送信トランスポートを準備します。

JAX-RS

表63.4「インバウンド JAX-RS インターセプター」 は、JAX-RS エンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.4 インバウンド JAX-RS インターセプター

クラス	フェーズ	説明
JAXRSInInterceptor	**PRE_STREAM**	ルートリソースクラスを選択し、設定された JAX-RS 要求フィルターを呼び出し、ルートリソースで呼び出すメソッドを決定します。



重要

JAX-RS エンドポイントのインバウンドチェーンは、**ServiceInvokerInInterceptor** インターセプタに直接スキップします。**JAXRSInInterceptor** の後に他のインターセプターは呼び出されません。

表63.5「アウトバウンド JAX-RS インターセプター」 は、JAX-RS エンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.5 アウトバウンド JAX-RS インターセプター

クラス	フェーズ	説明
JAXRSOutInterceptor	**MARSHAL**	応答を適切な形式に変換して送信します。

63.3. メッセージのバインディング

SOAP

表63.6「インバウンド SOAP インターセプター」 は、SOAP バインディングの使用時にエンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.6 インバウンド SOAP インターセプター

クラス	フェーズ	説明
CheckFaultInterceptor	**POST_PROTOCOL**	メッセージが障害メッセージかどうかを確認します。メッセージが障害メッセージである場合には、通常の処理は中止され、障害処理が開始されます。
MustUnderstandInterceptor	**PRE_PROTOCOL**	解釈する必要のあるヘッダーを処理します。
RPCInInterceptor	**UNMARSHAL**	rpc/literal メッセージをアンマーシャリングします。メッセージがベアである場合、メッセージは **BareInInterceptor** オブジェクトに渡され、メッセージ部分をデシリアライズします。
ReadsHeadersInterceptor	**READ**	SOAP ヘッダーを解析し、メッセージオブジェクトに保存します。
SoapActionInInterceptor	**READ**	SOAP アクションヘッダーを解析し、対象アクションの一意の操作を検索しようとします。
SoapHeaderInterceptor	**UNMARSHAL**	操作パラメーターにマッピングする SOAP ヘッダーを適切なオブジェクトにバインドします。
AttachmentInInterceptor	**RECEIVE**	mime 境界の mime ヘッダーを解析し、**ルート** 部分を検索し、入力ストリームをこれにリセットして、他の部分を **Attachment** オブジェクトのコレクションに保存します。
DocLiteralInInterceptor	**UNMARSHAL**	SOAP ボディーの最初の要素を調べ、適切な操作を決定し、データバインディングを呼び出してデータを読み取ります。
StaxInInterceptor	**POST_STREAM**	メッセージから **XMLStreamReader** オブジェクトを作成します。
URIMappingInterceptor	**UNMARSHAL**	HTTP GET メソッドの処理を行います。

クラス	フェーズ	説明
SwAInInterceptor	**PRE_INVOKE**	バイナリーの SOAP 添付ファイルに必要な MIME ハンドラーを作成し、そのデータをパラメーターリストに追加します。

表63.7「アウトバウンド SOAP インターセプター」 は、SOAP バインディングの使用時にエンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.7 アウトバウンド SOAP インターセプター

クラス	フェーズ	説明
RPCOutInterceptor	**MARSHAL**	rpc スタイルのメッセージを送信用にマーシャルします。
SoapHeaderOutFilterInterceptor	**PRE_LOGICAL**	インバウンドのみとしてマークされている SOAP ヘッダーをすべて削除します。
SoapPreProtocolOutInterceptor	**POST_LOGICAL**	SOAP バージョンと SOAP アクションヘッダーを設定します。
AttachmentOutInterceptor	**PRE_STREAM**	メッセージに含まれる可能性のある添付ファイルの処理に必要な添付マーシャラーと、mime stuff を設定します。
BareOutInterceptor	**MARSHAL**	メッセージの部分を書き込みます。
StaxOutInterceptor	**PRE_STREAM**	メッセージから **XMLStreamWriter** オブジェクトを作成します。
WrappedOutInterceptor	**MARSHAL**	アウトバウンドメッセージパラメーターをラップします。
SoapOutInterceptor	**WRITE**	メッセージのヘッダーブロックの **soap:envelope** 要素を書き込みます。また、残りのインターセプターの入力のために空の **soap:body** 要素も書き込みます。
SwAOutInterceptor	**PRE_LOGICAL**	SOAP 添付ファイルとしてパッケージ化されるバイナリーデータを削除し、後で処理できるように保存します。

XML

表63.8「インバウンド XML インターセプター」 は、XML バインディングの使用時にエンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.8 インバウンド XML インターセプター

クラス	フェーズ	説明
AttachmentInInterceptor	**RECEIVE**	mime 境界の mime ヘッダーを解析し、**ルート** 部分を検索し、入力ストリームをこれにリセットして、他の部分を **Attachment** オブジェクトのコレクションに保存します。
DocLiteralInInterceptor	**UNMARSHAL**	メッセージボディーの最初の要素を調べ、適切な操作を決定してから、データバインディングを呼び出してデータを読み取ります。
StaxInInterceptor	**POST_STREAM**	メッセージから **XMLStreamReader** オブジェクトを作成します。
URIMappingInterceptor	**UNMARSHAL**	HTTP GET メソッドの処理を行います。
XMLMessageInInterceptor	**UNMARSHAL**	XML メッセージをアンマーシャリングします。

表63.9「アウトバウンド XML インターセプター」 は、XML バインディングの使用時にエンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.9 アウトバウンド XML インターセプター

クラス	フェーズ	説明
StaxOutInterceptor	**PRE_STREAM**	メッセージから **XMLStreamWriter** オブジェクトを作成します。
WrappedOutInterceptor	**MARSHAL**	アウトバウンドメッセージパラメーターをラップします。
XMLMessageOutInterceptor	**MARSHAL**	送信用にメッセージをマーシャリングします。

CORBA

表63.10「インバウンド CORBA インターセプター」 は、COBRA バインディングの使用時にエンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.10 インバウンド CORBA インターセプター

クラス	フェーズ	説明
CorbaStreamInInterceptor	**PRE_STREAM**	CORBA メッセージをデシリアライズします。
BareInInterceptor	**UNMARSHAL**	メッセージの部分をデシリアライズします。

表63.11「アウトバウンド CORBA インターセプター」 は、COBRA バインディングの使用時にエンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.11 アウトバウンド CORBA インターセプター

クラス	フェーズ	説明
CorbaStreamOutInterceptor	**PRE_STREAM**	メッセージをシリアライズします。
BareOutInterceptor	**MARSHAL**	メッセージの部分を書き込みます。
CorbaStreamOutEndingInterceptor	**USER_STREAM**	メッセージのストリーム可能なオブジェクトを作成し、メッセージコンテキストに保存します。

63.4. その他の機能

Logging

表63.12「インバウンドロギングインターセプター」 は、ロギングをサポートするために、エンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.12 インバウンドロギングインターセプター

クラス	フェーズ	説明
LoggingInInterceptor	**RECEIVE**	生のメッセージデータをロギングシステムに書き込みます。

表63.13「アウトバウンドロギングインターセプター」 は、ロギングをサポートするために、エンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.13 アウトバウンドロギングインターセプター

クラス	フェーズ	説明
LoggingOutInterceptor	**PRE_STREAM**	ロギングシステムにアウトバウンドメッセージを書き込みます。

ロギングの詳細は、19章*Apache CXF ロギング*を参照してください。

WS-Addressing

表63.14「インバウンド WS-Addressing インターセプター」 は、WS-Addressing の使用時に、エンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.14 インバウンド WS-Addressing インターセプター

クラス	フェーズ	説明
MAPCodec	**PRE_PROTOCOL**	メッセージアドレス指定プロパティーをデコードします。

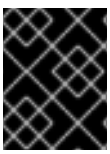
表63.15「アウトバウンド WS-Addressing インターセプター」 は、WS-Addressing の使用時に、エンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.15 アウトバウンド WS-Addressing インターセプター

クラス	フェーズ	説明
MAPAggregator	**PRE_LOGICAL**	メッセージのメッセージアドレス指定プロパティーを集約します。
MAPCodec	**PRE_PROTOCOL**	メッセージのアドレス指定プロパティーをエンコードします。

WS-Addressing の詳細には、20章*WS-Addressing のデプロイ* を参照してください。

WS-RM



重要

WS-RM は WS-Addressing に依存するため、WS-Addressing インターセプターもすべてインターセプターチェーンに追加されます。

表63.16「インバウンド WS-RM インターセプター」 は、WS-RM の使用時に、エンドポイントのインバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.16 インバウンド WS-RM インターセプター

クラス	フェーズ	説明
RMInInterceptor	**PRE_LOGICAL**	メッセージ部分と確認応答メッセージの集約を処理します。
RMSoapInterceptor	**PRE_PROTOCOL**	メッセージから WS-RM プロパティーをエンコードおよびデコードします。

表63.17「アウトバウンド WS-RM インターセプター」 は、WS-RM の使用時に、エンドポイントのアウトバウンドメッセージチェーンに追加されたインターセプターをリスト表示します。

表63.17 アウトバウンド WS-RM インターセプター

クラス	フェーズ	説明
RMOutInterceptor	**PRE_LOGICAL**	メッセージのチャンク作成とそのチャンクの送信を処理します。確認応答と再送要求の処理にも対応します。
RMSoapInterceptor	**PRE_PROTOCOL**	メッセージから WS-RM プロパティーをエンコードおよびデコードします。

WS-RM の詳細は、*21章 信頼性の高いメッセージングの有効化* を参照してください。

第64章 インターセプタープロバイダー

概要

インターセプタープロバイダーは、インターセプターチェーンが割り当てられた Apache CXF ランタイムのオブジェクトです。これらはすべて org.apache.cxf.interceptor.InterceptorProvider インターフェイスを実装します。開発者は独自のインターセプターを任意のインターセプタープロバイダーに割り当てることができます。

プロバイダーのリスト

以下のオブジェクトはインターセプタープロバイダーです。

- **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**

パート VIII. APACHE CXF の機能

本ガイドでは、Apache CXF の高度な各種機能を有効にする方法を説明します。

第65章 BEAN VALIDATION

概要

Bean Validation は、Java 規格で、Java アノテーションをサービスクラスまたはインターフェイスに追加することでランタイム制約を定義できるようにします。Apache CXF はインターセプターを使用して、この機能を Web サービスメソッド呼び出しと統合します。

65.1. 概要

概要

Bean Validation 1.1(JSR-349): 元の Bean Validation 1.0 (JSR-303) の後継であり、Java アノテーションを使用してランタイム時にチェックできる制約を宣言できます。アノテーションを使用して、Java コードの以下の部分に制約を定義できます。

- Bean クラスのフィールド。
- メソッドおよびコンストラクターパラメーター。
- メソッドの戻り値。

アノテーション付きクラスの例

以下の例は、標準の Bean Validation 制約がある、アノテーション付きの Java クラスです。

```
// 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 Validation またはスキーマの検証

場合によっては、Bean Validation とスキーマ検証は非常に似ています。XML スキーマでのエンドポイント設定は、Web サービスエンドポイントでランタイム時にメッセージを検証するように適切に確立された方法です。XML スキーマは、受信および送信メッセージに対して、Bean Validation と同じ制約の多くを確認できます。しかし、Bean Validation は、以下のような理由で便利な代替手段となる場合があります。

- Bean Validation を使用すると、XML スキーマとは独立して制約を定義できる (コードファーストサービス開発の場合などに便利です)。

- 現在の XML スキーマが lax 以外の場合は、Bean Validation を使用してより厳密な制約を定義できる。
- Bean Valudation を使用すると、カスタム制約を定義できますが、XML スキーマ言語を使用して定義できない場合があります。

依存関係

Bean Validation 1.1(JSR-349) 標準は、実装ではなく API のみを定義します。依存関係は、以下の 2 つの部分で指定する必要があります。

- **コア依存関係**- Bean Validation 1.1 API、Java 統合式言語 API および実装を提供します。
- **Hibernate Validator の依存関係**- Bean Validation 1.1 の実装を提供します。

コアの依存関係

Bean バリデーションを使用するには、以下のコア依存関係をプロジェクトの Maven **pom.xml** ファイルに追加する必要があります。

```
<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>
```

注記

javax.el/javax.el-api および **org.glassfish/javax.el** 依存関係は、Java 統合式言語の API および実装を提供します。この式言語は、Bean Validation によって内部で使用されますが、アプリケーションプログラミングレベルでは重要ではありません。

Hibernate Validator の依存関係

Bean バリデーションの Hibernate Validator 実装を使用するには、以下の追加の依存関係をプロジェクトの Maven **pom.xml** ファイルに追加します。

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-validator</artifactId>
   <version>5.0.3.Final</version>
</dependency>
```

OSGi 環境における検証プロバイダーの解決

検証プロバイダーを解決するデフォルトのメカニズムには、クラスパスをスキャンしてプロバイダーリソースを検索する必要があります。ただし、OSGi (Apache Karaf) 環境の場合、検証プロバイダー (Hibernate バリデーターなど) は別のバンドルにパッケージ化されているため、アプリケーションのクラスパスで自動的に使用できないため、このメカニズムは**機能しません**。OSGi のコンテキストでは、Hibernate バリデーターをアプリケーションバンドルに接続する必要があり、OSGi にはこれを正常に実行するためのサポートが若干必要です。

OSGi での検証プロバイダーの明示設定

OSGi のコンテキストでは、自動検出に依存するのではなく、検証プロバイダーを明示的に設定する必要があります。たとえば、共通の検証機能 (「Bean 検証機能」を参照) を使用して Bean Validation を有効にする場合は、以下のように検証プロバイダーで設定する必要があります。

```
<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"/>
```

HibernateValidationProviderResolver は、Hibernate 検証プロバイダーをラッピングするカスタムクラスです。

HibernateValidationProviderResolver クラスの例

以下のコード例は、Hibernate バリデーターを解決するカスタム **HibernateValidationProviderResolver** を定義する方法を示しています。

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

Maven バンドルプラグインを使用するように設定された Maven ビルドシステムに前述のクラスをビルドすると、アプリケーションはデプロイ時に Hibernate バリデーターバンドルに接続されます (Hibernate validator バンドルが OSGi コンテナーにデプロイされていることを前提とします)。

65.2. BEAN VALIDATION を使用したサービスの開発

65.2.1. サービス Bean のアノテーション

概要

Bean 検証でサービスを開発する最初のステップは、関連する検証アノテーションをサービスを表す Java クラスまたはインターフェイスに適用することです。検証アノテーションを使用すると、メソッドパラメーター、戻り値、およびクラスフィールドに制約を適用でき、サービスが呼び出されるたびに、実行時にチェックされます。

単純な入力パラメーターの検証

パラメーターが簡単な Java 型である場合にサービスメソッドのパラメーターを検証するには、Bean バリデーション API (**javax.validation.constraints** パッケージ) から制約アノテーションを適用できます。たとえば、以下のコード例は、nullness (**@NotNull** アノテーション) の両方のパラメーターをテストし、**id** 文字列が **\\d+** 正規表現 (**@Pattern** アノテーション) と一致するかどうか、および **name** 文字列の長さが 1 から 50 の範囲にあるかどうかをテストします。

```
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();
}
```

複雑な入力パラメーターの検証

複雑な入力パラメーター (オブジェクトインスタンス) を検証するには、以下の例のように **@Valid** アノテーションをパラメーターに適用します。

```
import javax.validation.Valid;
...
@POST
@Path("/books")
public Response addBook( @Valid Book book ) {
    // do some work
    return Response.created().build();
}
```

@Valid アノテーションは、単独では制約を指定しません。Book パラメーターに **@Valid** のアノテーションを付けると、検証制約を探すために (再帰的に) **Book** クラスの定義内を調べるように、検証エンジンを効果的に指示します。この例では、以下のように、**Book** クラスは **id** および **name** フィールド

の検証制約で定義されます。

```
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;

    // ...
}
```

戻り値の検証 (応答以外)

通常のメソッドの戻り値 (応答以外) に検証を適用するには、メソッド署名の前にアノテーションを追加します。たとえば、nullness (**@NotNull** アノテーション) の戻り値をテストし、検証の制約を再帰的にテストするには (**@Valid** アノテーション)、以下のように **getBook** メソッドにアノテーションを付けます。

```
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 );
}
```

戻り値 (応答) の検証

javax.ws.rs.core.Response オブジェクトを返すメソッドに検証を適用するには、Response 以外のケースと同じアノテーションを使用できます。以下に例を示します。

```
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. 標準アノテーション

Bean 検証の制約

表65.1「Bean Validation の標準アノテーション」 は、Bean Validation 仕様で定義されている標準のアノテーションで、フィールドやメソッドの戻り値およびパラメーターの制約定義に使用できます (標準アノテーションはクラスレベルで適用できません)。

表65.1 Bean Validation の標準アノテーション

Annotation	適用先	説明
@AssertFalse	**Boolean**、**boolean**	アノテーションが付けられた要素が **false** であることを確認します。
@AssertTrue	**Boolean**、**boolean**	アノテーションが付けられた要素が **true** であることを確認します。
@DecimalMax(value=, inclusive=)	**BigDecimal**、**BigInteger**、**CharSequence**、**byte**、**short**、**int**、**long**、およびプリミティブ型ラッパー	**inclusive=false** の場合、アノテーションが付けられた値が指定された最大値を下回ることを確認します。そうでない場合は、値が指定の最大値以下であることを確認します。**value** パラメーターは、**BigDecimal** 文字列形式で最大値を指定します。
@DecimalMin(value=, inclusive=)	**BigDecimal**、**BigInteger**、**CharSequence**、**byte**、**short**、**int**、**long**、およびプリミティブ型ラッパー	**inclusive=false** の場合、アノテーションが付けられた値が指定された最小値を上回ることを確認します。そうでない場合は、値が指定の最小値以上であることを確認します。**value** パラメーターは **BigDecimal** 文字列形式で最小値を指定します。
@Digits(integer=, fraction=)	**BigDecimal**、**BigInteger**、**CharSequence**、**byte**、**short**、**int**、**long**、およびプリミティブ型ラッパー	アノテーションが付けられた値が、整数部の桁数が **integer** までで、小数部の桁数が **fraction** までの数値であるかどうかを確認します。
@Future	**java.util.Date**, **java.util.Calendar**	アノテーション付き日付が未来の日付であるかどうかを確認します。
@Max(value=)	**BigDecimal**、**BigInteger**、**CharSequence**、**byte**、**short**、**int**、**long**、およびプリミティブ型ラッパー	アノテーション付きの値が指定の最大値以下であるかを確認します。
@Min(value=)	**BigDecimal**、**BigInteger**、**CharSequence**、**byte**、**short**、**int**、**long**、およびプリミティブ型ラッパー	アノテーション付きの値が指定の最小値以上であるかを確認します。

Annotation	適用先	説明
@NotNull	任意のタイプ	アノテーションが付けられた値が **null** でないことを確認します。
@Null	任意のタイプ	アノテーションが付けられた値が **null** であることを確認します。
@Past	**java.util.Date**, **java.util.Calendar**	アノテーションが付けられた日付が過去のものであるかどうかを確認します。
@Pattern(regex=, flag=)	**CharSequence**	指定されたフラグの一致を考慮して、アノテーションが付けられた文字列が正規表現 **regex** に一致するかどうかをチェックします。
@Size(min=, max=)	**CharSequence**、**Collection**、**Map**、および配列	アノテーションが付けられたコレクション、マップ、または配列のサイズが **min** 以上 **max** 以下であるかどうかを確認します。
@Valid	プリミティブ以外のタイプ	アノテーション付きオブジェクトに対して再帰的に検証を実行します。オブジェクトがコレクションかアレイの場合は、要素は再帰的に検証されます。また、オブジェクトがマップの場合、値要素が再帰的に検証されます。

65.2.3. カスタムアノテーション

Hibernate でのカスタム制約の定義

Bean Validation API で独自のカスタム制約アノテーションを定義できます。Hibernate バリデーターの実装でこれを行う方法は、Hibernate Validator Reference Guide の Creating custom constraints の章を参照してください。

65.3. BEAN VALIDATION の設定

65.3.1. JAX-WS の設定

概要

ここでは、Blueprint XML または Spring XML で定義される JAX-WS サービスエンドポイントで Bean Validation を有効にする方法を説明します。Bean Validation の実行に使用されるインターセプターは、JAX-WS エンドポイントと JAX-RS 1.1 エンドポイントの両方に共通です (JAX-RS 2.0 エンドポイントは異なるインターセプタークラスを使用します)。

Namespaces

このセクションに記載されている XML の例では、以下の表で示すように、必ず **jaxws** namespace 接頭辞を Blueprint または Spring のいずれかの適切な namespace にマッピングしてください。

XML 言語	Namespace
ブループリント	**http://cxf.apache.org/blueprint/jaxws**
Spring	**http://cxf.apache.org/jaxws**

Bean 検証機能

JAX-WS エンドポイントで最も簡単に Bean 検証を有効化する方法は、**Bean Validation 機能** をエンドポイントに追加することです。Bean Validation 機能は、以下のクラスによって実装されます。

org.apache.cxf.validation.BeanValidationFeature

この機能クラスのインスタンスを JAX-WS エンドポイントに追加することにより (Java API または XML の **jaxws:endpoint** の **jaxws:features** 子要素を経由)、エンドポイントで Bean バリデーションを有効にできます。この機能は、着信メッセージデータを検証する **In** インターセプターと、戻り値を検証する **Out** インターセプター (インターセプターはデフォルトの設定パラメーターで作成されます) の 2 つのインターセプターをインストールします。

Bean Validation 機能を使用した JAX-WS の設定例

以下の XML の例は、**commonValidationFeature** Bean を JAX-WS 機能としてエンドポイントに追加することで、JAX-WS エンドポイントで Bean バリデーション機能を有効にする方法を示しています。

```
<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"/>
```

HibernateValidationProviderResolver クラスの実装例は、「HibernateValidationProviderResolver クラスの例」 を参照してください。OSGi 環境 (Apache Karaf) のコンテキストで **beanValidationProvider** のみを設定する必要があります。

注記

コンテキストに応じて、必ず **jaxws** 接頭辞を Blueprint または Spring のいずれかの適切な XML namespace にマップしてください。

一般的な Bean Validation 1.1 インターセプター

Bean Validation の設定をより詳細に制御する場合は、Bean Validation 機能を使用する代わりにインターセプターを個別にインストールできます。Bean Validation 機能の代わりに、以下のインターセプターのいずれかまたは両方を設定できます。

org.apache.cxf.validation.BeanValidationInInterceptor

JAX-WS(または JAX-RS 1.1) エンドポイントにインストールする場合は、検証制約に対してリソースメソッドパラメーターを検証します。検証に失敗した場合は、**javax.validation.ConstraintViolationException** 例外を発生させます。このインターセプターをインストールするには、XML の **jaxws:inInterceptors** 子要素 (または XML の **jaxrs:inInterceptors** 子要素) を介してエンドポイントに追加します。

org.apache.cxf.validation.BeanValidationOutInterceptor

JAX-WS(または JAX-RS 1.1) エンドポイントにインストールする場合は、検証制約に対して応答値を検証します。検証に失敗した場合は、**javax.validation.ConstraintViolationException** 例外を発生させます。このインターセプターをインストールするには、XML の **jaxws:outInterceptors** 子要素 (または XML の **jaxrs:outInterceptors** 子要素) を介してエンドポイントに追加します。

Bean Validation インターセプターを使用した JAX-WS の設定例

以下の XML の例は、関連する **In** インターセプター Bean と **Out** インターセプター Bean をエンドポイントに明示的に追加することで、JAX-WS エンドポイントで Bean Validation 機能を有効にする方法です。

```
<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"/>
```

HibernateValidationProviderResolver クラスの実装例は、「HibernateValidationProviderResolver クラスの例」 を参照してください。OSGi 環境 (Apache Karaf) のコンテキストで **beanValidationProvider** のみを設定する必要があります。

BeanValidationProvider の設定

org.apache.cxf.validation.BeanValidationProvider は、Bean バリデーション実装 (**バリデーションプロバイダー**) をラップする単純なラッパークラスです。デフォルトの **BeanValidationProvider** クラスを上書きすると、Bean バリデーションの実装をカスタマイズできます。**BeanValidationProvider** Bean を使用すると、以下のプロバイダークラスを 1 つ以上上書きできます。

javax.validation.ParameterNameProvider

メソッドおよびコンストラクターパラメーターの名前を指定します。Java リフレクション API はメソッドパラメーターまたはコンストラクターパラメーターの名前に**アクセスできない** ため、このクラスが必要になることに注意してください。

javax.validation.spi.ValidationProvider<T>

指定された型 **T** の Bean バリデーションの実装を提供します。独自の **ValidationProvider** クラスを実装して、独自のクラスのカスタム検証ルールを定義できます。このメカニズムにより、Bean Validation フレームワークを効果的に拡張することができます。

javax.validation.ValidationProviderResolver

ValidationProvider クラスを検出するメカニズムを実装し、検出されたクラスのリストを返します。デフォルトのリゾルバーは、**ValidationProvider** クラスのリストが含まれる必要があるクラスパス上の **META-INF/services/javax.validation.spi.ValidationProvider** ファイルを探します。

javax.validation.ValidatorFactory

javax.validation.Validator インスタンスを返すファクトリー。

org.apache.cxf.validation.ValidationConfiguration

検証プロバイダーレイヤーからより多くのクラスをオーバーライドできる CXF ラッパークラス。

BeanValidationProvider をカスタマイズするには、カスタム **BeanValidationProvider** インスタンスを検証 **In** インターセプターのコンストラクターと検証 **Out** インターセプターのコンストラクターに渡します。以下に例を示します。

```
<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 設定

概要

ここでは、Blueprint XML または Spring XML で定義される JAX-RS サービスエンドポイントで Bean Validation を有効にする方法を説明します。Bean Validation の実行に使用されるインターセプターは、JAX-WS エンドポイントと JAX-RS 1.1 エンドポイントの両方に共通です (JAX-RS 2.0 エンドポイントは異なるインターセプタークラスを使用します)。

Namespaces

このセクションに記載されている XML の例では、以下の表で示すように、必ず **jaxws** namespace 接頭辞を Blueprint または Spring のいずれかの適切な namespace にマッピングしてください。

XML 言語	Namespace
ブループリント	**http://cxf.apache.org/blueprint/jaxws**
Spring	**http://cxf.apache.org/jaxws**

Bean 検証機能

JAX-RS エンドポイントで最も簡単に Bean 検証を有効化する方法は、**Bean Validation 機能** をエンドポイントに追加することです。Bean Validation 機能は、以下のクラスによって実装されます。

org.apache.cxf.validation.BeanValidationFeature

この機能クラスのインスタンスを JAX-RS エンドポイントに追加することにより (Java API または XML の **jaxrs:server** の **jaxrs:features** 子要素を経由)、エンドポイントで Bean バリデーションを有効にできます。この機能は、着信メッセージデータを検証する **In** インターセプターと、戻り値を検証する **Out** インターセプター (インターセプターはデフォルトの設定パラメーターで作成されます) の 2 つのインターセプターをインストールします。

検証例外マッパー

JAX-RS エンドポイントは、**検証例外マッパー** (HTTP エラー応答にマッピング) を設定する必要もあります。この検証例外マッパーは、HTTP エラー応答に対する検証例外をマッピングします。以下のクラスは JAX-RS の検証例外マッピングを実装します。

org.apache.cxf.jaxrs.validation.ValidationExceptionMapper

JAX-RS 2.0 仕様に従って検証例外マッピングを実装します。入力パラメーター検証違反は HTTP ステータスコード **400 Bad Request** にマップされます。戻り値の検証違反 (または内部検証違反) は HTTP ステータスコード **500 Internal Server Error** にマップされます。

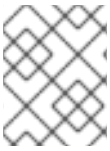
JAX-RS 設定のサンプル

以下の XML の例は、**commonValidationFeature** Bean を JAX-RS 機能として追加し、**exceptionMapper** Bean を JAX-RS プロバイダーとして追加することで、JAX-RS エンドポイントで Bean バリデーション機能を有効にする方法を示しています。

```
<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"/>
```

HibernateValidationProviderResolver クラスの実装例は、「HibernateValidationProviderResolver クラスの例」 を参照してください。OSGi 環境 (Apache Karaf) のコンテキストで **beanValidationProvider** のみを設定する必要があります。



注記

コンテキストに応じて、必ず **jaxrs** 接頭辞を Blueprint または Spring のいずれかの適切な XML namespace にマップしてください。

一般的な Bean Validation 1.1 インターセプター

Bean バリデーション機能を使用する代わりに、任意で Bean 検証インターセプターをインストールして検証実装をより詳細に制御することができます。JAX-RS は、この目的のために JAX-WS と同じインターセプターを使用します (「一般的な Bean Validation 1.1 インターセプター」 参照)。

Bean Validation インターセプターを使用した JAX-RS 設定の例

以下の XML の例は、関連する **In** インターセプター Bean と **Out** インターセプター Bean をサーバーエンドポイントに明示的に追加することで、JAX-RS エンドポイントで Bean Validation 機能を有効にする方法です。

```
<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"/>
```

HibernateValidationProviderResolver クラスの実装例は、「HibernateValidationProviderResolver クラスの例」 を参照してください。OSGi 環境 (Apache Karaf) のコンテキストで **beanValidationProvider** のみを設定する必要があります。

BeanValidationProvider の設定

「BeanValidationProvider の設定」 の説明にあるように、カスタム **BeanValidationProvider** インスタンスを検証インターセプターに注入できます。

65.3.3. JAX-RS 2.0 の設定

概要

JAX-RS 1.1(JAX-WS と共通の検証インターセプターを共有する) とは異なり、JAX-RS 2.0 設定は、JAX-RS2.0 に固有の専用の検証インターセプタークラスに依存しています。

Bean 検証機能

JAX-RS 2.0 の場合には、以下のクラスで実装される専用の Bean Validation 機能があります。

org.apache.cxf.validation.JAXRSBeanValidationFeature

この機能クラスのインスタンスを JAX-RS エンドポイントに追加することにより (Java API または XML の **jaxrs:server** の **jaxrs:features** 子要素を経由)、JAX-RS 2.0 サーバーエンドポイントで Bean バリデーションを有効にできます。この機能は、着信メッセージデータを検証する **In** インターセプターと、戻り値を検証する **Out** インターセプター (インターセプターはデフォルトの設定パラメーターで作成されます) の 2 つのインターセプターをインストールします。

検証例外マッパー

JAX-RS 2.0 は JAX-RS 1.x と同じ検証例外マッパークラスを使用します。

org.apache.cxf.jaxrs.validation.ValidationExceptionMapper

JAX-RS 2.0 仕様に従って検証例外マッピングを実装します。入力パラメーター検証違反は HTTP ステータスコード **400 Bad Request** にマップされます。戻り値の検証違反 (または内部検証違反) は HTTP ステータスコード **500 Internal Server Error** にマップされます。

Bean Validation 呼び出し元

デフォルト以外のライフサイクルポリシー (Spring ライフサイクル管理を使用する場合など) で JAX-RS サービスを設定する場合には、エンドポイント設定で **jaxrs:invoker** 要素を使用して、サービスエンドポイントで **org.apache.cxf.jaxrs.validation.JAXRSBeanValidationInvoker** インスタンスも登録し、Bean バリデーションが正しく呼び出されるようにする必要があります。

JAX-RS サービスライフサイクル管理の詳細は、「Spring XML でのライフサイクル管理」 を参照してください。

Bean バリデーション機能を使用した JAX-RS 2.0 の設定例

以下の XML の例は、**jaxrsValidationFeature** Bean を JAX-RS 機能として追加し、**exceptionMapper** Bean を JAX-RS プロバイダーとして追加することで、JAX-RS 2.0 エンドポイントで Bean バリデーション機能を有効にする方法を示しています。

```
<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"/>
```

HibernateValidationProviderResolver クラスの実装例は、「HibernateValidationProviderResolver クラスの例」 を参照してください。OSGi 環境 (Apache Karaf) のコンテキストで **beanValidationProvider** のみを設定する必要があります。

注記

コンテキストに応じて、必ず **jaxrs** 接頭辞を Blueprint または Spring のいずれかの適切な XML namespace にマップしてください。

一般的な Bean Validation 1.1 インターセプター

Bean Validation の設定をより詳細に制御する場合は、Bean Validation 機能を使用する代わりに JAX-RS インターセプターを個別にインストールできます。以下の JAX-RS インターセプターの 1 つまたは両方を設定します。

org.apache.cxf.validation.JAXRSBeanValidationInInterceptor

JAX-RS 2.0 サーバーエンドポイントにインストールする場合は、リソースメソッドパラメーターを検証制約に対して検証します。検証に失敗した場合は、**javax.validation.ConstraintViolationException** 例外を発生させます。このインターセプターをインストールするには、XML の **jaxrs:inInterceptors** 子要素を介してエンドポイントに追加します。

org.apache.cxf.validation.JAXRSBeanValidationOutInterceptor

JAX-RS 2.0 エンドポイントにインストールする場合は、検証制約に対して応答値を検証します。検証に失敗した場合は、**javax.validation.ConstraintViolationException** 例外を発生させます。このインターセプターをインストールするには、XML の **jaxrs:inInterceptors** 子要素を介してエンドポイントに追加します。

Bean Validation インターセプターを使用した JAX-RS 2.0 の設定例

以下の XML の例は、関連する **In** インターセプター Bean と **Out** インターセプター Bean をサーバーエンドポイントに明示的に追加することで、JAX-RS 2.0 エンドポイントで Bean Validation 機能を有効にする方法です。

```
<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"/>
```

HibernateValidationProviderResolver クラスの実装例は、「HibernateValidationProviderResolver クラスの例」 を参照してください。OSGi 環境 (Apache Karaf) のコンテキストで **beanValidationProvider** のみを設定する必要があります。

BeanValidationProvider の設定

「BeanValidationProvider の設定」 の説明にあるように、カスタム **BeanValidationProvider** インスタンスを検証インターセプターに注入できます。

JAXRSParameterNameProvider の設定

org.apache.cxf.jaxrs.validation.JAXRSParameterNameProvider クラスは **javax.validation.ParameterNameProvider** インターフェイスの実装で、JAX-RS 2.0 エンドポイントのコンテキストでメソッドおよびコンストラクターの名前を提供できます。