Java Web サービス: WSDL 1.1 を理解してモデル化する

WSDL 1.1 による Web サービスの定義、そして WSDL 文書を Java 言語でモデル化して検証および変換する方法を学ぶ

Dennis Sosnoski
Architecture Consultant and Trainer
Sosnoski Software Associates Ltd

2011年 2月 08日

WSDL (Web Services Description Language) 2.0 が W3C (World Wide Web Consortium) 標準として承認されてから数年経った今でも、Web サービスの記述形式として最も広く使用されているのは、WSDL 1.1 です。けれどもその人気の反面、WSDL 1.1 はいくつかの問題を抱えています。例えば、多種多様なスキーマが使用されていること、Web サービス・スタックによって WSDL 文書の処理方法が異なることなどです。この記事では、WSDL 1.1 のサービス記述がどのように構造化されているかを説明し、WSDL 文書を検証して「ベスト・プラクティス」に沿った形に変換する Java™ ツールの基本構造を紹介します。

このシリーズの他の記事を見る

この連載について

Web サービスは、エンタープライズ・コンピューティングにおいて Java™ 技術が担う重大な役割の一部です。この連載では、XML および Web サービスのコンサルタントである Dennis Sosnoski が、Web サービスを使用する Java 開発者にとって重要になる主要なフレームワークと技術について説明します。この連載から、現場での最新の開発情報を入手して、それらを皆さんのプログラミング・プロジェクトにどのように利用できるかを知っておいてください。

エンタープライズ・アプリケーションのための Web サービスは、サービス定義の使用方法に大きく依存します。サービス定義とは、サービス・プロバイダーと潜在的なあらゆるコンシューマーとの間の基本契約を規定し、サービスが提供する関数のタイプ、そして各関数の一部として交換されるメッセージを細かく規定するものです。送信する実際のメッセージがサービス定義と一致している限り、サービス・プロバイダーとコンシューマーは、それぞれの側でのメッセージ交換を自由自在に実装することができます。このように、サービス定義によって XML メッセージ交換を指定するという点が、Web サービスを以前の分散プログラミング技術とは別のものにしています。

Web サービスを定義するには、これまで多種多様な手法が提案されてきましたが、そのうち最も広く採用されているのは WSDL 1.1 です。けれども WSDL 1.1 にはいくつかの欠点があります。例えば、構造があまりにも複雑なため、初心者には判読しにくいなどの問題です。さらに、公式に

決められた信頼できる定義がないことから、オリジナルの仕様文書の穴を埋めた「アドオン」解説が次々と追加されるという事態になっています。このことから、Web サービス・スタックは WSDL 1.1 文書をできるだけ柔軟に処理するようにしていますが、この柔軟性がかえって WSDL 1.1 を理解する上で混乱を招く原因となっています。なぜなら開発者は、最適な手法に関するガイダンスもなく、広範に及ぶ WSDL 構造を目にすることになるからです。

この記事では、WSDL 1.1 文書を理解する方法を説明し、さらに WSDL 文書を検証して標準形式に変換するための Java モデルの始めの部分を記載します。

WSDL 1.1 について

名前空間の使用について

この記事では、それぞれの名前空間に以下の接頭辞を使用します。

- `wsdl` 接頭辞は、WSDL 1.1 の名前空間 `http://schemas.xmlsoap.org/wsdl/` を表します。
- `soap` 接頭辞は、WSDL 1.1 の SOAP 1.1 拡張で使用する名前空間 `http://schemas.xmlsoap.org/wsdl/soap/` を表します。
- `xs` 接頭辞は、XML Schema 定義に使用される名前空間 `http://www.w3.org/2001/XMLSchema` を表します。

2001年の初めに公開された WSDL 1.1 は、表向きには 2007年に発表された W3C WSDL 2.0 勧告にその座を譲っています。WSDL 2.0 の構造は WSDL 1.1 よりも簡潔で、柔軟性という点でも勝っています。それにも関わらず、WSDL 2.0 は普及していません。それには、WSDL 2.0 が広範にサポートされていないという理由もありますが、逆に、普及していないために WSDL 2.0 をサポートする Web サービス・スタックの実装者に対するプレッシャーがほとんどないという、どちらが原因で、どちらが結果なのかわからない問題があります。一方、WSDL 1.1 には不備な点があるものの、ほとんどの目的を十分果たします。

当初の WSDL 1.1 仕様は、どれだけの機能が使用されることになるかについて曖昧でした。WSDL の焦点は SOAP サービス定義と連携することに置かれていたため、初めは SOAP 機能 (rpc エンコーディングなど) のサポートも仕様に含まれていましたが、後になって、これらの機能は望ましくないことが判明しました。WS-I (Web Services Interoperability Organization) がこれらの問題に対処するために採った手段は、基本プロファイル (BP) に SOAP と WSDL を使用した Web サービスのベスト・プロクティスを定義するというものです。BP 1.0 は 2004年に承認され、2006年に BP 1.1 として更新されました。この記事では WS-I BP ガイドラインに基づく WSDL 1.1 について説明し、事実上廃止された機能 (SOAP 対応の rpc エンコーディングなど) については無視します。

XML 文書の構造を定義することになっているのは、XML Schema 定義です。WSDL 1.1 では当初の仕様にスキーマの記述を含めていましたが、そのスキーマは、いくつかの点でテキストの記述とは一致しませんでした。この不一致はその後のスキーマの変更で修正されたものの、WSDL 1.1 文書はこの変更を反映するように変更されませんでした。その後、WS-I BP グループは WSDL スキーマにさらに変更を加えることを決定し、この当てにならないスキーマのベスト・プラクティスとも言えるバージョンを作成しました。あるバーションのスキーマに対して作成された文書は、一般に、他のバージョンには対応しません (同じ名前空間を使っていても対応しません)。しかし幸いなことに、ほとんどの Web サービス・ツールは基本的にスキーマを無視し、妥当だと思われるものであれば何でも受け入れます (「参考文献」に、WSDL の多数のスキーマへのリンクが記載されています)。

WSDL 1.1 スキーマの WS-I BP バージョンでさえも、WSDL 1.1 文書が仕様に準拠していることを確実にする上では、それほど助けにはなりません。このスキーマは、特にコンポーネントの順序をはじめ、WS-I BP に定義された制約のすべてを反映していないからです。さらに、XML Schema は、文書に簡単に規定できるような多くの制約 (代替属性、つまり別のスキーマから必要となる拡張要素など) を処理することができません。したがって、WSDL 1.1 文書が (WS-I BP によって修正された) WSDL 1.1 仕様に準拠しているかどうかを調べるには、XML スキーマの妥当性検査を有効にするだけでは到底足りません。この話題については後で再び取り上げることとして、まずは WSDL 1.1 のサービス記述の構造を改めて説明します。

記述に使用するコンポーネント

WSDL 1.1 文書は常に、`<wsdl:definitions>` という便利な名前が付いたルート要素を使用します。このルート要素の中では、1 つの「パッシブ」な子要素 (単に個々の WSDL 1.1 文書を参照する要素) と、5 つの「アクティブ」な子要素 (実際にサービス記述を定義する要素) が以下の WSDL 1.1 名前空間に定義されます。

- `<wsdl:import>` は、この文書内に組み込む記述が含まれる個別の WSDL 1.1 文書を参照します。
- `<wsdl:types>` は、メッセージ交換に使用する XML の型または要素を定義します。
- `<wsdl:message>` は、実際のメッセージを XML の型または要素の観点から定義します。
- `<wsdl:portType>` は、サービスによって実装される抽象的な操作のセットを定義します。
- `<wsdl:binding>` は、`<wsdl:portType>` の実際の実装を、具体的なプロトコルとフォーマットを使用して定義します。
- `<wsdl:service>` は、全体としてのサービスを定義します。ここには通常、`<wsdl:binding>` 要素のアクセス情報を持つ 1 つ以上の `<wsdl:port>` 要素が含まれます。

上記の他、文書化を目的とした `<wsdl:document>` 要素もあります。この要素は、`<wsdl:definitions>` 要素の最初の子として、あるいは上記のどの要素の子要素としても使用することができます。

一般に、完全なサービス記述には、`<wsdl:import>` を除く上記の要素が少なくとも 1 つは必要です。ただし、これらの要素がすべて同じ文書内に含まれていなければならないというわけではありません。`<wsdl:import>` を使用して、複数の文書から完全な WSDL の記述を組み立てることも可能です。この場合、組織に合わせて記述を柔軟に分割することができます。例えば、最初の 3 つの記述要素 (`<wsdl:types>`、`<wsdl:message>`、`<wsdl:portType>`) が 1 つにまとまって完全なサービス・インターフェース記述になるので (おそらくアーキテクチャー・チームによって定義されます)、この 3 つを実装関連の `<wsdl:binding>` 要素と `<wsdl:service>` 要素から分けておくことが妥当な場合もあります。すべての主要な Web サービス・スタックは、複数の WSDL 文書に分割された記述をサポートします。

リスト 1 とリスト 2 に、2 つの WSDL 文書に分割された WSDL のサービス記述の例を記載します。この例では、インターフェース記述コンポーネントを BookServerInterface.wsdl ファイルにまとめ、実装コンポーネントを BookServerImpl.wsdl ファイルにまとめています。まず、リスト 1 に BookServerInterface.wsdl を記載します。

リスト 1. BookServerInterface.wsdl

```
<wsdl:definitions ... xmlns:tns="http://sosnoski.com/ws/library/BookServerInterface"
```

```
    targetNamespace="http://sosnoski.com/ws/library/BookServerInterface">
  <wsdl:document>Book service interface definition.</wsdl:document>
  <wsdl:types>
    <xs:schema ...
        targetNamespace="http://sosnoski.com/ws/library/BookServerInterface">
      <xs:import namespace="http://sosnoski.com/ws/library/types"
          schemaLocation="book-types.xsd"/>
      ...
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="getBookMessage">
    <wsdl:part name="part" element="tns:getBook"/>
  </wsdl:message>
  <wsdl:message name="getBookResponseMessage">
    <wsdl:part name="part" element="tns:getBookResponse"/>
  </wsdl:message>
  ...
  <wsdl:message name="addBookMessage">
    <wsdl:part name="part" element="tns:addBook"/>
  </wsdl:message>
  <wsdl:message name="addBookResponseMessage">
    <wsdl:part name="part" element="tns:addBookResponse"/>
  </wsdl:message>
  <wsdl:message name="addDuplicateFault">
    <wsdl:part name="fault" element="tns:addDuplicate"/>
  </wsdl:message>
  <wsdl:portType name="BookServerPortType">
    <wsdl:documentation>
      Book service implementation. This creates an initial library of books when the
      class is loaded, then supports method calls to access the library information
      (including adding new books).
    </wsdl:documentation>
    <wsdl:operation name="getBook">
      <wsdl:documentation>
        Get the book with a particular ISBN.
      </wsdl:documentation>
      <wsdl:input message="tns:getBookMessage"/>
      <wsdl:output message="tns:getBookResponseMessage"/>
    </wsdl:operation>
    ...
    <wsdl:operation name="addBook">
      <wsdl:documentation>Add a new book.</wsdl:documentation>
      <wsdl:input message="tns:addBookMessage"/>
      <wsdl:output message="tns:addBookResponseMessage"/>
      <wsdl:fault message="tns:addDuplicateFault" name="addDuplicateFault"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

リスト 2 には、もう一方の BookServerImpl.wsdl ファイルを記載します。ファイルの先頭近くにある `<wsdl:import>` 要素が、BookServerInterface.wsdl からインターフェース記述をインポートします。

リスト 2. BookServerImpl.wsdl

```
<wsdl:definitions ... xmlns:ins="http://sosnoski.com/ws/library/BookServerInterface"
    xmlns:tns="http://sosnoski.com/ws/library/BookServer"
    targetNamespace="http://sosnoski.com/ws/library/BookServer">
  <wsdl:document>
    Definition of actual book service implementation.
  </wsdl:document>
<wsdl:import namespace="http://sosnoski.com/ws/library/BookServerInterface"
      location="BookServerInterface.wsdl"/>
  <wsdl:binding name="BookServerBinding" type="ins:BookServerPortType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
```

```
    <wsdl:operation name="getBook">
      <soap:operation soapAction="urn:getBook"/>
      <wsdl:input>
        <soap:body/>
      </wsdl:input>
      <wsdl:output>
        <soap:body/>
      </wsdl:output>
    </wsdl:operation>
    ...
    <wsdl:operation name="addBook">
      <soap:operation soapAction="urn:addBook"/>
      <wsdl:input>
        <soap:body/>
      </wsdl:input>
      <wsdl:output>
        <soap:body/>
      </wsdl:output>
      <wsdl:fault name="addDuplicateFault">
        <soap:fault name="addDuplicateFault"/>
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="BookServer">
    <wsdl:port name="BookServerPort" binding="tns:BookServerBinding">
      <soap:address location="http://localhost:8080/cxf/BookServer"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

WSDL 1.1 名前空間での要素 (および属性) の定義の他、WSDL 1.1 は拡張要素も定義します。拡張要素は、特定のタイプのサービスに必要な追加情報を提供するために、WSDL 1.1 のサービス記述の特定の場所に挿入されるように意図されています。今でも広く使用されている WSDL 1.1 の拡張要素は唯一、SOAP 1.1 バインディング用の要素だけです (リスト 2 の `<wsdl:binding>` および `<wsdl:service>` 要素の中で使用されています)。これらの拡張要素はオリジナルの WSDL 1.1 仕様で定義され、その後、SOAP 1.2 バインディングに対応するために、2006年に別の仕様で定義されました。

コンポーネントの詳細

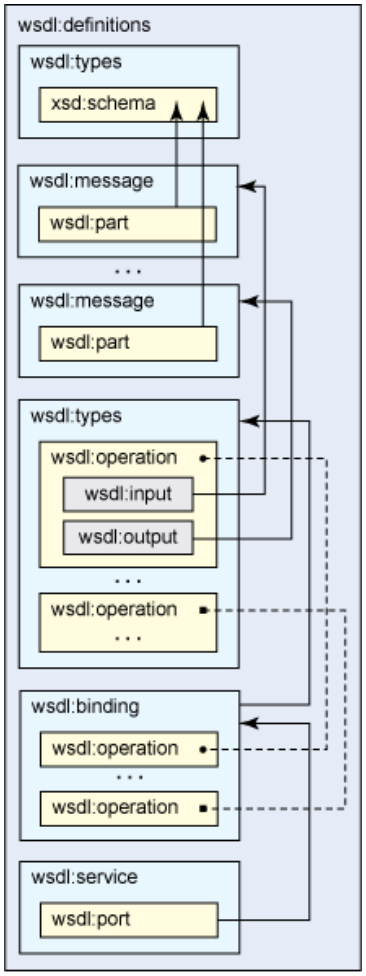
`<wsdl:types>` 要素は、メッセージに使用されるすべての XML 定義を、1 つ以上の `<xs:schema>` 要素でラッピングします (WSDL では、これらの定義に XML Schema に代わる手段も使用できますが、ほとんどのスタックがサポートするのは XML Schema のみです)。`<xs:schema>` 要素では、必要に応じて `<xs:import>` または `<xs:include>`、あるいはこの両方を使用して、WSDL 外部の他のスキーマを組み込むことができます (また、同じ WSDL に含まれる別のスキーマを参照することもできます)。

単一の `<wsdl:types>` 要素には、任意の数のスキーマ定義を含めることができます。したがって、WSDL 文書で複数の `<wsdl:types>` 要素を使用する理由はありません。リスト 1 では、`<wsdl:types>` 要素は BookServerInterface.wsdl ファイルの先頭近くにあります。

`<wsdl:import>` と `<wsdl:types>` を除き、WSDL 文書の他のすべての最上位レベルのコンポーネントには、必須の`##`属性を使用して個々に名前が付けられます。ドキュメント・ルート `<wsdl:definitions>` 要素で `targetNamespace` 属性を使用すると (通常は、こうすることがベスト・プラクティスなので従ってください)、上記の 2 つを除く他の最上位レベルのコンポーネントは、そのターゲット名前空間に定義されます。このことは、名前を定義するときには単純な

名前または名前の「ローカル」な部分だけを指定しますが、そのコンポーネントへの参照は、コンポーネントの名前を名前空間の接頭辞またはデフォルトの名前空間で修飾しなければならないことを意味します。図 1 に、WSDL コンポーネント間の最も重要なリンクを示します。実線は修飾名を使用する参照を表し、点線は名前空間の修飾なしで識別する場合に使用する名前を示します。

図 1. WSDL コンポーネント間のリンク



WSDL のサービス記述の中核となるのは、`<wsdl:message>` 要素によって表されるメッセージです。`<wsdl:message>` 要素は、クライアントとサービス・プロバイダーの間で交換される XML データを記述します。各 `<wsdl:message>` には `<wsdl:part>` 子要素が含まれないこともあれば、複数含まれることもあります (通常は 1 つ含まれます)。`part` 要素のそれぞれには、独自の `name` 属性 (`<wsdl:message>` 内で固有のもの) と、XML データのスキーマ定義を参照する `element` 属性または `type` 属性が必要です。リスト 1 に記載した BookServerInterface.wsdl では、`<wsdl:types>` 要素に続いて複数の `<wsdl:message>` 要素が示されています。

`<wsdl:portType>` 要素は、サービスの抽象インターフェースを、そのサービスとの間で受け渡すメッセージの観点から定義します。`<wsdl:portType>` 要素には任意の数の `<wsdl:operation>` 子要素が含まれます。各 `<wsdl:operation>` 子要素には、固有の `name` 属性が必要で (`<wsdl:portType>` 内で WS-I BP が `<wsdl:operation>` 子要素を一意に識別するために必要としています)、操作で使用

するメッセージを記述する 1 つ以上の子要素が含まれます。これらの子要素には、それぞれに異なる用途を表す 3 つのタイプがあります。

- `<wsdl:input>` : 操作に対する入力として、クライアントからサービス・プロバイダーに送信されるデータ
- `<wsdl:output>` : 操作の結果として、サービス・プロバイダーからクライアントに返されるデータ
- `<wsdl:fault>` : 処理中にエラーが発生した場合、サービス・プロバイダーからクライアントに返されるデータ

WSDL 1.1 では、クライアントとサービス・プロバイダーとの対話に複数のパターンを定義しています。これらのパターンは、`<wsdl:input>` および `<wsdl:output>` 子要素の異なるシーケンスによって表されますが、パターンのすべてが、実装するのに十分なほど明確に定義されているわけではありません。そこで、WS-I BP ではこれらのパターンを 2 つだけに制限しています。1 つは `<wsdl:input>` の後に `<wsdl:output>` が続くリクエスト・レスポンス操作、もう 1 つは `<wsdl:input>` だけを指定した片方向の操作です。リクエスト・レスポンス操作 (群を抜いて最もよく使用されるタイプ) の場合、`<wsdl:input>` 要素および `<wsdl:output>` 要素の後に任意の数の `<wsdl:fault>` 要素を続けることができます。

`<wsdl:input>`、`<wsdl:output>`、または `<wsdl:fault>` の各要素は、必須の `message` 属性によってメッセージ記述を参照します。この参照は名前空間で修飾されるため、通常は接頭辞を組み込む必要があります。リスト 1 にはその一例として、`getBook` 操作の記述に `<wsdl:input message="tns:getBookMessage"/>` 要素が使用されているインスタンスがあります (`tns` 接頭辞は、`targetNamespace` 属性と同じ名前空間 URI を持つルート `<wsdl:definitions>` 要素に定義されています)。

`<wsdl:portType>` は、大部分の点で論理的に Java インターフェースに相当すると考えることができます。具体的には、`<wsdl:operation>` 要素はメソッドに相当し、`<wsdl:input>` 要素はメソッドのパラメーター、`<wsdl:output>` 要素はメソッドの戻り値、そして `<wsdl:fault>` 要素はチェック例外に相当すると考えられます。WSDL から Java コードを生成するときには、既存の Java コードから WSDL を生成するほとんどのツールと同じく、これらの対応付けを使用します。

SOAP 1.1 と SOAP 1.2 との違い

SOAP 1.1 は 2000年に公開されて以来、Web サービスに幅広く使用されています。SOAP 1.2 は、W3C を介した一層幅広い業界のサポートによって開発され、正式な W3C 標準として 2007年に公開されました。SOAP 1.2 は SOAP 1.1 よりも明確に文書化されているだけでなく、SOAP 1.1 の厄介な側面を取り除いて簡潔になっています。この一層簡潔な構造にも関わらず、ほとんどの Web サービスにとって、この 2 つの間に実用上の違いはほとんどありません。SOAP 1.2 の最も重要な特徴は、おそらくこれが唯一、XOP (XML-binary Optimized Packaging) および SOAP MTOM (Message Transmission Optimization Mechanism) によって提供される SOAP 添付の拡張サポートを使用する方法として正式にサポートされていることです。古いスタックには SOAP 1.2 をサポートしないものもあるので、連載「Java Web サービス」ではこれまで SOAP 1.1 を使用してきましたが、新しい Web サービス開発に使用するには、SOAP 1.2 のほうが賢い選択になると思います。

リスト 2 に記載した BookServerImpl.wsdl の最初に示されている `<wsdl:binding>` 要素は、`<wsdl:portType>` で定義された抽象インターフェースのインスタンスを表します。`type` 属性が、バインディングによって実装されるポート・タイプの修飾名を指定します。

<wsdl:binding> の子要素は、ポート・タイプの実装方法に関する詳細を指定します。WSDL 名前空間で <wsdl:portType> の子要素に対応する <wsdl:binding> の子要素は、name 属性の値として、名前空間修飾子による参照で指定した値ではなく、<wsdl:portType> 参照で指定したのと同じ値を使用する必要があります。この関連付けは、図 1 の <wsdl:operation> レベルに点線で示されています。名前による同じ関連付けは、<wsdl:operation> 要素の <wsdl:input>/<wsdl:output>/<wsdl:fault> 子要素に適用されます。同じ要素名を再使用するにも関わらず、これらの要素に含まれる内容は、<wsdl:portType> 要素の子になっているときと、<wsdl:binding> の子になっているときとでは、かなり違ってきます。

WSDL によって定義された拡張は、<wsdl:binding> に記述されます。<soap:binding> 子要素は、SOAP サービスを定義する際に使用されます (WS-I BP で許可しているサービスのタイプは唯一、SOAP サービスだけですが、WSDL 1.1 では HTTP バインディングも許可します)。この <soap:binding> 要素は、必須の transport 属性を使用して、バインディングで使用するトランスポートのタイプを定義します (リスト 2 の値、http://schemas.xmlsoap.org/soap/http に示されているように、WS-I BP で選択が許可されているのは HTTP だけです)。オプションの style 属性では、XML データ表現として rpc スタイルを使用するか、文書スタイルを使用するかを選択することができます (最もよく使用されるデフォルトの document は、タイプの定義ではなくスキーマ要素の定義を使用するメッセージに相当します)。

<wsdl:binding> に含まれる各 <wsdl:operation> 子要素の中で、その操作を呼び出すリクエストを識別するために SOAPAction の値を指定するには、<soap:operation> 要素を使用することができます (この要素を使用して、<soap:binding> 要素で決定された rpc スタイルまたは文書スタイルの選択を変更することもできますが、WS-I BP ではこの用法を禁止しています)。<wsdl:input>/<wsdl:output>/<wsdl:fault> 子要素のそれぞれには、別の拡張要素が含まれます。リスト 2 では、<wsdl:input> にも、<wsdl:output> にも、拡張要素として <soap:body> が含まれています (これは、メッセージ・データを SOAP メッセージ本体に含めて送信することを意味します。データ、さらにエラーを SOAP ヘッダーに含めて送信することもできますが、賢明な方法ではないと思います)。また、<wsdl:fault> でも、同等の <soap:fault> が使用されています。

最後に説明する WSDL のサービス記述のコンポーネントは、<wsdl:service> 要素です。この要素は <wsdl:port> 要素のグループで構成され、各 <wsdl:port> 要素がアクセス・アドレスを <wsdl:binding> に関連付けます。アクセス・アドレスは、ネストされた <soap:address> 拡張要素によって指定されます。

WSDL の処理

WSDL 1.1 文書のスキーマおよび規則のさまざまなバリエーションを考えれば、多くの文書が WS-I BP で定義されたベスト・プラクティスの形式と一致しないのも当然です。ベスト・プラクティスの形式のさまざまな変化形を対象としたあらゆる Web サービス・スタックによるサポートが、旧式の、あるいは誤った構成の使用をこれまで存続させてきたため、業界全体に悪い慣例が広がっています。そして私も、この悪影響から免れてはいません。この連載のサンプル・コードのために用意した WSDL 文書を再び見直してみて、完全に正しい文書が 1 つもないことに驚きました。

この記事を執筆するにあたり、WSDL 文書をベスト・プラクティスのルールと照らし合わせて検証するためのツールがあれば、人々の役に立つだろうと考えました。元の WSDL にエラーがない

限り、このような検証用のツールが、WSDL 文書をベスト・プラクティスの形式に変換するための小さな一歩となるはずです。このツールを実現するには、当初計画していたよりもかなりの作業が必要になることがわかったので、連載の今後 2 回の記事にわたり、このモデルの詳細を開発していきます。

Java 言語で WSDL 文書を処理するためのモデルは、今まで数多く組み立てられてきました。これには、JSR 110 リファレンス実装 (「参考文献」を参照) として広範に使用されている、WSDL4J (Web Services Description Language for Java Toolkit) も含まれますが、いずれのモデルも、私が達成したいと考えている 2 つの目標を完全に満たしそうにはありませんでした。というのも 2 つの目標のうちの 1 つは、不完全な形式の WSDL 文書を読み取って、エラーとベスト・プラクティスからの逸脱の両方をレポートすること、そしてもう 1 つは、ベスト・プラクティスの形式に再フォーマット化したエラーのない WSDL 文書を作成することだからです。例えば WSDL4J は、入力での要素の順序を維持しないため、順序付けに関する問題をレポートすることになります。さらに、このモデルはスキーマ定義を処理しないので、`<wsdl:part>` 要素からの参照をチェックするにはそのまま使用することはできません。私に与えられた選択肢は、もっと現実的な目標を設定するか、独自のモデルを作成するかのどちらかでしたが、私が選んだのは当然、独自のモデルを作成することです。

WSDL モデル

妥当性検査と検証の違い

この記事では、WSDL 文書の正確さを調べるという意味で、「検証 (verification)」という用語を使用しています。verification の代わりに使用できる「検証 (validation)」という用語は、一般に、XML 文書をスキーマ定義に照らし合わせてチェックするという意味で「妥当性検査 (validation)」として使用されているためです。

以前、JiBX/WS プロジェクトの一環として、JiBX データ・バインディングで使用する部分的な WSDL モデルを実装しました。このモデルは出力専用に設計されており、必要となるクラスは比較的少数です。これらのクラスは、場合によっては WSDL XML 構造のネストされた要素からのデータを結合します (例えば、`<wsdl:message>` は単一の子 `<wsdl:part>` と組み合わせられ、`<wsdl:binding>` 内に含まれる `<wsdl:input>`、`<wsdl:output>`、および `<wsdl:fault>` は、`<soap:body>` または `<soap:fault>` 要素と組み合わせられるなどです)。この簡潔なクラス構造では、サポートする WSDL 文書のサブセットを簡単に構成することができます。けれども、この部分的な WSDL モデルに基づいて検証およびツールの再構成を行なうことを検討した初めの段階で、完全に構造化されていない可能性のある WSDL の入力をサポートするには、XML 表現により近いモデルが必要だということに気付きました。

WSDL 1.1 の WS-I BP スキーマからコードを生成するという選択肢もありましたが、この方法を検討してみると、生成されたクラスをそのまま直接使用すると面倒なことになるとわかりました。このスキーマには重複したタイプが含まれるだけでなく、異なるメッセージ交換パターンを表すために厄介な構造が使用されているからです (その後、これらのパターンのいくつかは、WS-I BP テキストで禁止されました)。

最後に私が到達した結論は、クラスを手動で構成することです。最終的な結果は、スキーマからコードを生成した場合とほとんど同じになり、しかも不要な重複と複雑さが取り除かれることになりました。JiBX データ・バインディングは同じクラスに対する複数のバインディングをサポートするため、WSDL 1.1 のあらゆるバージョンで許容される全オプションに対処する入力バイン

ディングをセットアップできたと同時に、ベスト・プラクティスの形式でのみ WSDL を出力する出力バインディングを構成することができました。

リスト 3 に、`Definitions` クラスから、ルート `<wsdl:definitions>` 要素に対応する部分を抜粋して記載します。

リスト 3. `Definitions` クラス (一部抜粋)

```
public class Definitions extends ElementBase
{
    /** Enumeration of child elements, in expected order. */
    static enum AddState {
        invalid, imports, types, message, portType, binding, service };

    /** List of allowed attribute names. */
    public static final StringArray s_allowedAttributes =
        new StringArray(new String[] { "name", "targetNamespace" });

    /** Validation context in use. */
    private ValidationContext<ElementBase,Definitions> m_validationContext;

    /** Current state (used for checking order in which child elements are added). */
    private AddState m_state;

    /** Name for this definitions. */
    private String m_name;

    /** Target namespace for WSDL. */
    private String m_targetNamespace;

    /** List of all import child elements. */
    private List<Import> m_imports = new ArrayList<Import>();

    /** List of all types child elements. */
    private List<Types> m_types = new ArrayList<Types>();

    /** List of all message child elements. */
    private List<Message> m_messages = new ArrayList<Message>();

    /** List of all portType child elements. */
    private List<PortType> m_portTypes = new ArrayList<PortType>();

    /** List of all binding child elements. */
    private List<Binding> m_bindings = new ArrayList<Binding>();

    /** List of all services child elements. */
    private List<Service> m_services = new ArrayList<Service>();

    /** Map from qualified name to message in this definition. */
    private Map<QName,Message> m_nameMessageMap =
        new HashMap<QName,Message>();

    /** Map from qualified name to port type in this definition. */
    private Map<QName,PortType> m_namePortTypeMap =
        new HashMap<QName,PortType>();

    /** Map from qualified name to message in this definition. */
    private Map<QName,Binding> m_nameBindingMap =
        new HashMap<QName,Binding>();

    /** Map from qualified name to service in this definition. */
    private Map<QName,Service> m_nameServiceMap =
        new HashMap<QName,Service>();
    ...
    /**
```

```
     * Check state transitions between different types of child elements.
     * If the elements are not in the expected order,
     * this flags the first out-of-order element for reporting.
     * @param state new add state
     * @param comp element component
     */
    private void checkAdd(AddState state, ElementBase comp) {
        if (m_state != state) {
            if (m_state == null || (m_state != AddState.invalid &&
                state.ordinal() > m_state.ordinal())) {

                // advanced on to another type of child element
                m_state = state;

            } else if (state.ordinal() < m_state.ordinal()) {

                // report child element out of order
                m_validationContext.addWarning
                    ("Child element of wsdl:definitions out of order", comp);
                m_state = AddState.invalid;
            }
        }
    }
    ...
    /**
     * Add an unmarshalled wsdl:message child element. This also indexes the message by
     * name for validation access.
     *
     * @param child
     */
    public void addMessage(Message child) {
        checkAdd(AddState.message, child);
        m_messages.add(child);
        addName(child.getName(), child, m_nameMessageMap);
    }
    ...
```

このモデルが一般的な形式の入力とベスト・プラクティスの形式の出力の両方をサポートする仕組みは、リスト 3 の子要素データの編成を見れば明らかです。このモデルは、あらゆるタイプの子要素を網羅した単一のリストを使用するのではなく、タイプごとに個別のリストを使用します。入力 JiBX バインディングは子要素を順不同のセットとして扱い、子要素がアンマーシャルされるたびに、その要素のタイプに固有の設定メソッドを呼び出します。`<wsdl:message>` 子要素に使用している `addMessage()` 設定メソッドからわかるように、設定メソッドは、子要素の以前の値を置換するのではなく、タイプ別のリストにインスタンスを追加します。また、要素が期待される順序になっていない場合に備え、設定メソッドのそれぞれが状態チェックも実行します。

拡張属性および拡張要素 (基本的には、WSDL 1.1 名前空間を使用しない任意の属性または要素) は、どの WSDL 要素にも含めることができます。連載の以前の記事で説明した、WSDL 文書に組み込まれている WS-Policy 構成は、このような拡張要素の一例であり、実際にはポリシー参照です。これらの拡張要素でのベスト・プラクティスは、WSDL 1.1 名前空間にある子要素の前に拡張要素を配置することなので、出力バインディングではそのように処理しています。リスト 3 には示されていませんが、入力バインディングは WSDL 要素クラスの基底クラスからのコードを使用して拡張要素および拡張属性を処理し、要素を任意の順序で受け入れます (要素が WSDL 1.1 名前空間からの要素に続いている場合は、警告を生成します)。

このモデルは、それぞれに固有のクラス一式を持つ拡張名前空間ごとの個別のバインディングを使用して、既知の拡張要素を処理します。連載「Java Web サービス」の次回の記事では、これら

の拡張要素の処理について詳しく説明するとともに、ソース・コードの詳細についても取り上げます。

モデルを検証する

WSDL データの基本的な検証は、要素に対応するオブジェクトがアンマーシャルされて WSDL 文書のツリー構造に追加される時点で行われます。これは、リスト 3 の終わりの `addMessage()` コード内に示されています。このコードは、`checkAdd()` メソッドを使って子要素の順序をチェックし、`addName()` メソッドを使って有効な名前が指定されていること (テキストが `NCName` スキーマ型と一致し、値が要素タイプ内で一意であること) を確認した上で、名前をオブジェクトにマッピングします。けれどもこの検証では、要素の最も基本的な情報を個別にチェックしているにすぎません。各要素のその他のプロパティーと、要素間の相互関係をチェックするための検証コードも必要です。

JiBX では、アンマーシャリングおよびマーシャリングのプロセスの一環としてユーザー拡張フックを呼び出すことができます。この WSDL モデルでは、そのような拡張フックとして、事後設定メソッドを使用して検証ロジックを実行します。事後設定メソッドは、関連付けられたオブジェクトのアンマーシャリングが完了した後に呼び出されるため、オブジェクト検証チェックを実行するのに役立つ場合がよくあります。WSDL を検証するのに最も簡単な方法となるのは、ルート `<wsdl:definitions>` 要素に対する単一の事後設定メソッドで、オブジェクトの検証全体を実行することです。この方法を使用すれば、コンポーネントが期待される順序でない場合に発生する、WSDL 文書のコンポーネントへの前方参照の問題を回避することができます。

今後の拡張

この記事では、WSDL の基本的な構造と使用方法を説明するとともに、WSDL 文書の検証およびベスト・プラクティスの形式への変換の両方をサポートすることを目的とした WSDL 対応の Java データ・モデルを紹介しました。

連載の次回の記事では、このトピックをさらに詳しく掘り下げて、WS-Policy および WS-SecurityPolicy アサーションを作成する際によくある問題を検討します。さらに、WSDL モデルと検証処理について詳細に説明するなかで、WSDL に組み込まれた WS-Policy/WS-SecurityPolicy アサーションを組み込むように、このモデルを拡張します。

著者について

Dennis Sosnoski

Dennis Sosnoski は Java ベースの SOA および Web サービスを専門とするコンサルタント兼トレーナーです。専門家としてのソフトウェア開発経験は 30 年以上に渡り、この 10 年間はサーバー・サイドの XML 技術や Java 技術に注力しています。オープンソースのJiBX XML Data Binding フレームワークや、それに関連した JiBX/WS Web サービス・フレームワークの開発リーダーを務め、さらに Apache Axis2 Web サービス・フレームワークのコミッターでもあります。彼は JAX-WS 2.0 および JAXB 2.0 仕様のエキスパート・グループの一員でもありました。連載「Java Web サービス」の内容は、彼の SOA および Web サービスのトレーニング・クラスがベースとなっています。