Java Web 服务: WSDL 1.1 理解与建模

**简介：** 在 Web Services Description Language (WSDL) 2.0 被批准成为一个 World Wide Web Consortium (W3C) 标准的几年以后，WSDL 1.1 仍然是最广泛使用的 Web 服务描述方式。尽管很流行，但是 WSDL 1.1 仍然有一些问题，包括使用的模式很多，而且 Web 服务处理 WSDL 文档的方法各异。

企业应用程序的 Web 服务很大程度上依赖于服务定义的使用。服务定义规定了服务提供者和所有潜在用户之间的基本契约，详细规定了服务所提供的功能类型，以及每一个功能所交换的消息。服务提供者和用户可以采用任何他们喜欢的方式来实现各自的消息交换，只要他们所发送的实际消息符合服务定义的规定。使用规定 XML 消息交换的服务定义是 Web 服务与早期分布式编程的最显著区别。

被提议用来定义 Web 服务的技术有很多，但是最广泛使用的方法是 WSDL 1.1。WSDL 1.1 存在一些缺点，包括结构过于复杂，使得普通人很难读懂。它也缺少一种权威的正式定义，导致后来需要 “附加” 说明来修补原始规范文档的各种漏洞。相应的，Web 服务协议趋向于尽可能灵活地处理 WSDL 1.1 文档。这种灵活性肯定会增加理解 WSDL 1.1 的难度，因为开发人员会遇到大量的 WSDL 结构，但是无法确定哪种方法是最好的。

在本文中，您将了解如何理解 WSDL 1.1 文档，并且您将看到第一部分用于验证 WSDL 文档并将它们转换成标准形式的 Java 模型。

**认识 WSDL 1.1**

## 名称空间用法

本文使用：

* wsdl 前缀来表示 WSDL 1.1http://schemas.xmlsoap.org/wsdl/ 名称空间
* soap 前缀来表示 WSDL 1.1 扩展 SOAP 1.1 所使用的http://schemas.xmlsoap.org/wsdl/soap/ 名称空间
* xs 前缀来表示 XML 模式定义所使用的http://www.w3.org/2001/XMLSchema 名称空间

发布于 2001 年初的 WSDL 1.1 技术上已经被 2007 年发布的 W3C WSDL 2.0 推荐标准所替代。WSDL 2.0 提供了比 WSDL 1.1 更清晰的结构，并且更加灵活。但是 WSDL 2.0 遇到了一个鸡蛋相生的问题 — WSDL 2.0 并没有被广泛应用，因为它并没有得到广泛支持，同时由于没有广泛应用也使得支持它的 Web 服务实现也发展缓慢。虽然有缺陷，但是 WSDL 1.1 仍然适用于大多数场合。

原始的 WSDL 1.1 规范并没有明确定义将使用多少的特性。因为 WSDL 的关键是处理 SOAP 服务定义，它也包含了一些 SOAP 特性（如，RPC 编码）支持，后来证明这些是不需要的。Web Services Interoperability Organization (WS-I) 在 Basic Profile (BP) 中解决了这些问题，Basic Profile (BP) 定义了 Web 服务使用 SOAP 和 WSDL 的最佳实践方法。BP 1.0 是在 2004 年通过批准的，并在 2006 年升级到 BP 1.1。在本文中，我将基于 WS-I BP 指南来介绍 WSDL 1.1，忽略一些实际上已弃用的特性，如 SOAP RPC 编码。

XML 模式定义是用来定义 XML 文档结构的。WSDL 1.1 在最初的规范中包含了一个模式描述，但是这个模式在几个方面上与内容描述并不匹配。这个问题已经在后来的模式修订版本上得到纠正，但是 WSDL 1.1 文档仍然没有反映这个修改的更新。然后 WS-I BP 小组决定对 WSDL 模式进行更多的修改，所以它才为这个不稳定模式创建了一个最佳实践版本。根据其中一个模式版本编写的文档一般与其他版本是不兼容的（尽管使用相同的名称空间），但是幸好大多数 Web 服务工具实际上都会忽略模式而接受所有合理的东西。（见 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#resources) 中许多 WSDL 模式的链接。）

即使是 WSDL 1.1 模式的 WS-I BP 版本也不能保证 WSDL 1.1 文档符合规范要求。这个模式并不反映 WS-I BP 的所有约束，特别是关于组件的顺序。除此之外，XML 模式还无法处理许多很容易规定文档约束的类型（如替代属性，或从另一个模式上使用扩展元素）。所以检查一个 WSDL 1.1 文档是否符合 WSDL 1.1 规范（WS-I BP 所修订）远不止于打开 XML 模式验证。我将在本文后面继续讨论这个问题。首先，我将介绍 WSDL 1.1 服务描述的结构。

**服务描述组件**

WSDL 1.1 文档使用一个固定的根元素，一般命名为 <wsdl:definitions>。在根元素中，WSDL 1.1 名称空间定义了一个 “被动” 子元素（用来引用不同的 WSDL 1.1 文档）和五个 “主动” 子元素（这些元素就是实际的服务描述）：

* <wsdl:import> 引用另一个 WSDL 1.1 文档，将其描述加到本文档中。
* <wsdl:types> 定义消息交换所使用的 XML 类型和元素。
* <wsdl:message> 定义一个实际的消息，包含 XML 类型或元素。
* <wsdl:portType> 定义一个服务所实现的操作抽象集。
* <wsdl:binding> 定义 <wsdl:portType> 的一个使用特定协议和格式的具体实现。
* <wsdl:service> 定义一个服务整体，包括一个或多个包含 <wsdl:binding> 元素访问信息的 <wsdl:port> 元素。

另外还有一个可用于文档记录的 <wsdl:document> 元素，它可以是 <wsdl:definitions> 元素的第一个子元素，也可以是上面其他元素的第一个子元素。

一个完整的服务描述一般至少包含一个除 <wsdl:import> 元素之外的其他元素，但是所有这些元素不一定要位于同一个文档中。您可以使用 <wsdl:import> 元素将多个文档整合成为一个完整的 WSDL 描述，这使您能够灵活分割描述以满足您组织的要求。例如，前三个描述元素（<wsdl:types>、<wsdl:message> 和 <wsdl:portType>）一起构成了完整的服务接口描述（可能是由一个体系结构团队定义的），所以将它们保存到不同的面向实现的 <wsdl:binding> 和 <wsdl:service> 元素上是很有意义的。所谓以的主流 Web 服务协议都支持将描述分割到多个 WSDL 文档中。

清单 1 和 [清单 2](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing2) 是一个分割成两个 WSDL 文档的 WSDL 服务描述示例，其中接口描述组件位于 BookServerInterface.wsdl 文件，而实现组件则位于 BookServerImpl.wsdl。清单 1 是 BookServerInterface.wsdl：

**清单 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> |

清单 2 是 BookServerImpl.wsdl。开头的 <wsdl:import> 元素从 BookServerInterface.wsdl 导入接口描述。

**清单 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> |

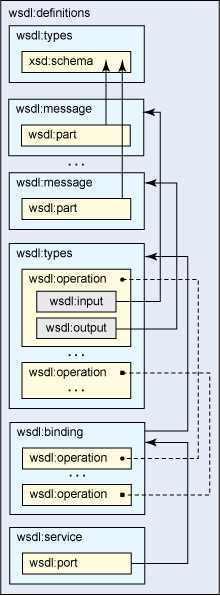
除了 WSDL 1.1 名称空间的元素（和属性）定义，WSDL 1.1 还定义了扩展元素。这些扩展元素是用在 WSDL 1.1 服务描述的特定位置以便为特定类型的服务提供必要的额外信息。仍然在广泛使用的 WSDL 1.1 扩展元素是那些用于 SOAP 1.1 绑定的元素（见 [清单 2](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing2)，<wsdl:binding> 和 <wsdl:service> 元素中的元素），它们是由原始的 WSDL 1.1 规范所定义的，而对于 SOAP 1.2 绑定的扩展元素则是在 2006 年发布的规范中定义的。

**组件细节**

<wsdl:types> 元素将消息所使用的所有 XML 定义封装到一个或多个 <xs:schema> 元素中。（WSDL 允许这些定义使用其他方法来替代 XML 模式，但是大多数协议都只支持 XML 模式。）<xs:schema> 元素可以使用 <xs:import> 和/或 <xs:include>将其他外部模式在需要时包含到 WSDL 中（以及引用同一个 WSDL 中包含的其他模式）。

因为一个 <wsdl:types> 可以包含任意数量的模式定义，因此一个 WSDL 文档完全不需要使用多个 <wsdl:types> 元素。在[清单 1](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing1) 中，<wsdl:types> 元素位于 BookServerInterface.wsdl 开头部分。

除了 <wsdl:import> 和 <wsdl:types>，一个 WSDL 文档的所有其他顶级组件都使用一个必需的 name 属性进行单独命名。如果您在文档根元素 <wsdl:definitions> 中使用 targetNamespace 属性（这是您通常应该采用的最佳实践方法），那么这些组件的名称是在该目标名称空间中定义的。这意味着在您定义名称时，您只需要指定名称的简单或 “本地” 部分，但是引用这个组件必须使用加上名称空间前缀或默认名称空间的完整名称。图 1 显示了 WSDL 组件之间的最重要联系，其中实线表示完整名称引用而虚线表示不带名称空间仅用于标识的名称：

**图 1. WSDL 组件之间的联系**  
 

<wsdl:message> 元素所表示的消息是 WSDL 服务描述的核心。<wsdl:message> 元素是客户端与服务提供商之间交换的 XML 数据描述，每一个 <wsdl:message> 都包含 0 到多个（一般是 1 个）<wsdl:part> 子元素。每一个 part 元素都必须有自己的name 属性（在 <wsdl:message> 中保持惟一），和一个 element 或一个引用 XML 数据模式定义的 type 属性。[清单 1](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing1) 显示了几个 <wsdl:message> 元素，它们位于 BookServerInterface.wsdl 的 <wsdl:types> 元素之后 。

<wsdl:portType> 元素定义了一个服务的抽象接口，即服务发送和接收的消息。<wsdl:portType> 元素包含了许多<wsdl:operation> 子元素。每一个 <wsdl:operation> 子元素都必须有一个 name 属性（WS-I BP 规定在 <wsdl:portType> 中是惟一的），并且包含一个或多个用于描述操作所使用的消息的子元素。这些子元素有三种，分别代表不同类型的用法：

* <wsdl:input> ：从客户端发送到服务提供商上作为操作输入的数据
* <wsdl:output> ：由服务提供商作为操作结果返回给客户端的数据
* <wsdl:fault> ：在处理出现错误时由服务提供商返回给客户端的数据

WSDL 1.1 定义了几种客户端与服务提供商交互模式，分别由不同的 <wsdl:input> 和 <wsdl:output> 子元素序列表示，但是并非所有模式定义都能够实现。WS-I BP 将模式限制为两种：“请求-响应” 操作，即一个 <wsdl:input> 后面紧跟一个<wsdl:output>；单向操作，即只有一个 <wsdl:input>。在 “请求-响应” 操作中（目前最常用的），<wsdl:input> 和<wsdl:output> 元素之后可以有任意多个 <wsdl:fault> 元素。

每一个 <wsdl:input>、<wsdl:output> 或 <wsdl:fault> 元素都会通过必需的 message 属性引用一个消息描述。这个引用是采用完整名称空间的，所以它通常需要包含一个前缀。您可以在 [清单 1](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing1) 中看到一些例子，如 getBook 操作描述中使用的<wsdl:input message="tns:getBookMessage"/> 元素。（tns 前缀是在根元素 <wsdl:definitions> 中定义的，其名称空间 URI 与 targetNamespace 属性相同。）

在大多数情况下您可以认为 <wsdl:portType> 在逻辑上是与 Java 接口等价的，其中 <wsdl:operation> 元素相当于方法，而<wsdl:input> 元素相当于方法参数，<wsdl:output> 元素是方法的返回值，而 <wsdl:fault 元素则是受检异常。从 WSDL 生成 Java 代码也采取这样的对应关系，大多数从 Java 代码生成 WSDL 的工具也采取同样的方法。

## SOAP 1.1 vs. 1.2

SOAP 1.1 从 2000 年规范发布以来就一直广泛应用于 Web Services。SOAP 1.2 是由 W3C 基于更广泛行业支持而开发并在 2007 年作为一个 W3C 官方标准发布的。SOAP 1.2 比 SOAP 1.1 更整洁且文档更齐全，1.1 中一些不好的方面已经被无情地去除。尽管结构上更整洁，但是对于大多数 Web 服务 而言，这两个版本之间的实际差别很小。可能 SOAP 1.2 的最重要特性是它是惟一官方支持使用 XML-binary Optimized Packaging (XOP) 和 SOAP Message Transmission Optimization Mechanism (MTOM) 的 SOAP 附件增强特性的版本。到目前为止，我一直在 Java Web Services 系列文章中使用 SOAP 1.1，因为一些较老的协议并不支持 SOAP 1.2，但是 1.2 可能是一种更好的 Web 服务全新部署方法。

<wsdl:binding> 元素表示 <wsdl:portType> 所定义抽象接口的一个实例，见 [清单 2](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing2) 所示的 BookServerImpl.wsdl 的开头部分。type属性指定了这个绑定所实现的端口类型全名。

<wsdl:binding> 的子元素说明规定了这个端口类型是如何实现的。WSDL 名称空间的子元素对应于 <wsdl:portType> 所指定端口类型，并且必须使用相同的 name 属性值 — 和 <wsdl:portType> 引用一样不是 完整名称空间引用。[图 1](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#fig1) 在 <wsdl:operation> 上使用虚线表示这个连接。相同的名称连接也用在 <wsdl:operation> 的子元素 <wsdl:input>/<wsdl:output>/<wsdl:fault> 上。尽管重用了相同的元素名称，但如果这些元素是 <wsdl:binding> 元素的子元素而不是 <wsdl:portType> 元素的子元素，那么它的内容是差别很大的。

<wsdl:binding> 是 WSDL 所定义的扩展发挥作用的地方。<soap:binding> 子元素将用于定义一个 SOAP 服务（这是 WS-I BP 所支持的惟一服务类型，尽管 WSDL 1.1 也支持 HTTP 绑定）。这个 <soap:binding> 元素会使用必需的 transport 属性来定义该绑定所使用的传输类型。（如 [清单 2](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing2) 中http://schemas.xmlsoap.org/soap/http 的值所示，HTTP 是 WS-I BP 所支持的惟一选择。）您可以使用可选的 style 属性来选择 RPC 或文档类型作为 XML 数据表现（默认最常用的是 document，它通过模式元素定义与消息保持一致，而不是类型定义）。

在 <wsdl:binding> 的每一个 <wsdl:operation> 子元素中，必须使用一个 <soap:operation> 元素指定一个 SOAPAction 值，用来确定调用这个操作的请求（也可能用于重写 <soap:binding> 元素所指定的 RPC 或文档类型，但是 WS-I 禁止这个用法）。每一个 <wsdl:input>/<wsdl:output>/<wsdl:fault> 子元素都包含另一个扩展元素，在 [清单 2](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing2) 所示情况中，<wsdl:input> 或 <wsdl:output> 中总是 <soap:body>（表示消息数据中在 SOAP 消息主体中发送的 — 在 SOAP 头中也可能发送数据，甚至错误，但是我认为这种不是一种好的做法），而在 <wsdl:fault> 则使用等价的 <soap:fault>。

WSDL 服务描述的最后一个组件是 <wsdl:service> 元素，这是由一组 <wsdl:port> 元素组成的。每一个 <wsdl:port> 元素都会通过一个 <wsdl:binding> 关联一个访问地址。这个访问地址是由内嵌的 <soap:address> 扩展元素指定的。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#ibm-pcon)

**WSDL 处理**

由于 WSDL 1.1 文档的模式和规则变化很多，所以很自然许多文档都不符合 WS-I BP 所规定的最佳形式。所有 Web 服务工具对许多最佳形式差异的支持助长了过时或错误结构的使用，从而导致行业中遍布许多不好的实践方法。而我也肯定免不了受到影响 — 检查我在本系列文章所提供的示例 WSDL 文档，我意外地发现没有一个代码是完全正确的。

所以当我准备撰写本篇文章时，我原本认为加入一个人们可用来验证 WSDL 文档是否符合最佳实践规则的工具是很不错的。只要原始 WSDL 文件没有错误，将 WSDL 文档转换成符合最佳实践方法的形式似乎是很简单的过程。但是结果证明这个过程比我最初设想的工作量要大得多，所以我将在本系列文章的后续两篇文章中详细介绍这个模型。

采用 Java 语言实现的 WSDL 文档处理模型有很多，包括广泛使用的 Web Services Description Language for Java Toolkit (WSDL4J)，这是 JSR 110 的参考实现（见 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#resources)）。但是似乎其中没有一个模型符合我真正希望实现的两个目标：第一，读取所有形式上不完全合理的 WSDL 文档，并报告错误及与最佳实践的差异；第二，重新编写符合最佳实践格式的正确 WSDL 文档。例如，WSDL4J 不会保持元素输入的顺序，这样我就会报告排序问题，也不会处理模式定义，所以它们不会直接用来检查 <wsdl:part> 元素的引用。所以我可能会告诉更现实地设定我的目标，或者编写自己的模型。很自然，我会选择编写自己的模型。

**WSDL 模型**

## 确认（Validation） vs. 验证（Verification）

我将在本文中使用验证 这个词来表示检查一个 WSDL 文档的正确性，因为它的同义词确认 通常在 XML 文档中用来表示根据模式定义检查文档的意思。

之前我已经将部分 WSDL 模型作为 JiBX/WS 项目的一部分来实现 JiBX 数据绑定。这个模型只是用作输出，并且它只包含相对较少的类，它们在某些情况下会包含 WSDL XML 结构内嵌元素的数据（包含一个 <wsdl:part> 子元素的 <wsdl:message>，和<wsdl:binding> 元素中包含一个 <soap:body> 或 <soap:fault> 的<wsdl:input>、<wsdl:output> 和 <wsdl:fault> 等）。这种压缩的类结构使它很容易创建这种结构所支持的 WSDL 文档子集，但是当我开始考虑基于该模型开发一个验证和重建工具时，我认识到支持可能设计不当的 WSDL 输入将需要一种更接近于 XML 表现的模型。

从 WSDL 1.1 的 WS-I BP 模式生成代码则是另一种方法。当我尝试这样做时，我认识到只使用所生成的类可能会很麻烦，因为这个模式包含了过多的类型，以及一些适合用来表现不同消息交换模式的结构（其有一些是后来 WS-I BP 规定中禁止使用的）。

所以我最终只是手动地创建这些类，虽然最终结果与我之前从模式生成代码，然后去掉不必要的重复内容并简化之后的结果是非常相似的。JiBX 数据绑定支持给相同的类提供多重绑定，所以我能够创建输入绑定来处理所有版本 WSDL 都支持的全部方法，同时配置输出绑定只输出最佳形式的 WSDL。

清单 3 显示了 Definitions 类中对应于根元素 <wsdl:definitions> 的部分内容：

**清单 3. Definitions class （部分）**

|  |
| --- |
| 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);  }  ... |

[清单 3](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing3) 中的子元素数据组织显示该模型是如何支持普通形式输入和最佳形式输出的。它不使用一个包含所有类型的子元素清单，而是为每一种类型创建不同的清单。JiBX 绑定输入将子元素作为无序集处理，在每一个子元素乱序时调用该类元素特有的设置方法。设置方法会将这个实例添加一个有类型的清单，而不替换任何之前已有的值，您可以参考处理<wsdl:message> 子元素所使用的 addMessage() 设置方法。在获取元素可能不是预期的顺序，所以每一个设置方法也会在每一次获取时检查元素状态。

所有的 WSDL 元素都允许添加扩展属性和元素（实际上是任何不使用 WSDL 1.1 名称空间的属性或元素）。本系列的前几篇文章的 WSDL 文档中所使用的 WS 策略配置就是一种这样的扩展元素，它们原来是实际的策略引用。这些扩展元素的最佳实践方法是它们用来引用 WSDL 1.1 名称空间中任意子元素的，并且这也是在输出绑定中对它们进行处理的方法。输入绑定会使用 WSDL 元素类的基类代码来处理扩展元素和属性，[清单 3](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing3) 没有包含这部分代码。而且它不关心元素的顺序（如果它们在 WSDL 1.1 名称空间的一个元素之后，那么就会出现一条警告）。

这个模型会为每一个扩展名称空间使用不同的绑定来处理已知的扩展元素，每一个扩展名称空间均对应不同集合的类。我将在下一篇 Java Web Services 文章对这些扩展元素进行更详细的介绍，同时提供更详细的源代码。

**验证模型**

WSDL 数据的一些基本验证是作为与 WSDL 文档树结构的元素对应的无序对象执行的，如 [清单 3](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#listing3) 末尾的 addMessage() 所示。这段代码使用 checkAdd() 方法来检查子元素的顺序，并使用 addName() 方法来保证提供了一个有效名称（与 NCName 模式类型相匹配，并且其值在该元素类型中是惟一的），并将这个名称映射到对象上。但是这只是单独地检查元素的最基本信息；我们还需要更多的验证代码来检查每一个元素的其他属性，以及元素之间的相互关系。

JiBX 允许您调用作为编组（marshalling）和解组（unmarshalling）过程一部分的用户扩展元素。WSDL 模型会使用一个这样的扩展元素，即一个后设方法，来运行这个验证逻辑。在相关对象的乱序完成之后它就会调用一个后设方法，所以执行对象验证检查通常是有用的。在验证 WSDL 时，最早的方法是是在一个后设方法之外执行根元素 <wsdl:definitions>的所有对象验证。这种方法可以避免向前引用了不在预期顺序的 WSDL 文档组件的问题。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jws20/index.html#ibm-pcon)

**更多的扩展**

在本文中，您了解了 WSDL 的基本结构和用法，以及用于支持 WSDL 文档验证并将它们转换成最佳形式文档的 WSDL Java 数据模型。

[本系列](http://www.ibm.com/developerworks/cn/java/j-jws/) 的下一篇文章会进一步讨论这个话题，讨论编写 WS-Policy 和 WS-SecurityPolicy 断言时经常遇到的问题。它也会介绍 WSDL 模型和更深层次的验证过程，包括扩展这个模型以包含 WSDL 中嵌入的 WS-Policy/WS-SecurityPolicy 断言。