Web 服務提示與技巧: JAX-RPC 與 JAX-WS 的比較

在 IBM Bluemix 雲平台上開發並部署您的下一個應用。

引言

Web 服務已經出現很久了。首先是 SOAP，但 SOAP 僅描述消息的情況，然後是 WSDL，WSDL 並不會告訴您如何使用 Java™ 編寫 Web 服務。在這種情況下，JAX-RPC 1.0 應運而生。經過數月使用之後，編寫此規範的 Java Community Process (JCP) 人員認識到需要對其進行一些調整，調整的結果就是 JAX-RPC 1.1。該規範使用大約一年之後，JCP 人員希望構建一個更好的版本：JAX-RPC 2.0。其主要目標是與行業方向保持一致，但行業中不僅只使用 RPC Web 服務，還使用面向消息的 Web 服務。因此從名稱中去掉了「RPC」，取而代之的是「WS」（當然表示的是 Web 服務）。因此 JAX-RPC 1.1 的後續版本是 JAX-WS 2.0——Java API for XML-based Web services。

哪些內容保持不變？

在列出 JAX-RPC 1.1 和 JAX-WS 2.0 的差異前，我們應該首先討論一下二者的相同之處。

- JAX-WS 仍然支持 SOAP 1.1 over HTTP 1.1，因此互操作性將不會受到影響，仍然可以在網上傳遞相同的消息。
- JAX-WS 仍然支持 WSDL 1.1，因此您所學到的有關該規範的知識仍然有用。WSDL 2.0 規範已經接近完成，但在 JAX-WS 2.0 相關工作結束時其工作仍在進行中。

區別何在？

- SOAP 1.2

  JAX-RPC 和 JAX-WS 都支持 SOAP 1.1。JAX-WS 還支持 SOAP 1.2。

- XML/HTTP

  WSDL 1.1 規範在 HTTP 綁定中定義，這意味著利用此規範可以在不使用 SOAP 的情況下通過 HTTP 發送 XML 消息。JAX-RPC 忽略了 HTTP 綁定。而 JAX-WS 添加了對其的支持。

- WS-I Basic Profile

  JAX-RPC 支持 WS-I Basic Profile (BP) V1.0。JAX-WS 支持 BP 1.1。（WS-I 即 Web 服務互操作性組織。）

- 新 Java 功能

  - JAX-RPC 映射到 Java 1.4。JAX-WS 映射到 Java 5.0。JAX-WS 依賴於 Java 5.0 中的很多新功能。
  - Java EE 5 是 J2EE 1.4 的後續版本，添加了對 JAX-WS 的支持，但仍然支持 JAX-RPC，這可能會對 Web 服務新手造成混淆。

- 數據映射模型

  - JAX-RPC 具有自己的映射模型，此模型大約涵蓋了所有模式類型中的 90%。它沒有涵蓋的部分映射到了 `javax.xml.soap.SOAPElement`。
  - JAX-WS 的數據映射模型是 JAXB。JAXB 可保證所有 XML 模式的映射。

- 接口映射模型

  JAX-WS 的基本接口映射模型與 JAX-RPC 的區別並不大，不過二者之間存在以下差異：

  - JAX-WS 的模型使用新的 Java 5.0 功能。
  - JAX-WS 的模型引入了異步功能。

- 動態編程模型

  - JAX-WS 的動態客戶機模型與 JAX-RPC 的對應模型差別很大。很多更改都是為了認可行業需求：

    - 引入了面向消息的功能。
    - 引入了動態異步功能。

  - JAX-WS 還添加了動態服務器模型，而 JAX-RPC 則沒有此模型。

- 消息傳輸優化機制（Message Transmission Optimization Mechanism，MTOM）

  JAX-WS 通過 JAXB 添加了對新附件規範 MTOM 的支持。Microsoft 從來沒有給 SOAP 添加附件規範；但似乎大家都支持 MTOM，因此應該能夠實現附件互操作性。

- 處理程序模型

  - 從 JAX-RPC 到 JAX-WS 的過程中，處理程序模型發生了很大的變化。
  - JAX-RPC 處理程序依賴於 SAAJ 1.2。JAX-WS 處理程序依賴於新的 SAAJ 1.3 規範。

在本文剩下的篇幅中，我們將討論 SOAP 1.2、XML/HTTP、WS-I Basic Profiles 和 Java 5。上面的其餘五個部分將分別在本系列技巧文章中單獨討論。

SOAP 1.2

從編程模型的角度而言，SOAP 1.1 和 SOAP 1.2 之間並沒有太多的差異。作為 Java 程序員，您只會在使用處理程序時遇到這些差異，我們將在以後的技巧文章中對如何處理這種情況進行討論。SAAJ 1.3 已更新以支持 SOAP 1.2。

XML/HTTP

與 SOAP 1.2 的更改類似，從編程模型的角度而言，SOAP/HTTP 和 XML/HTTP 消息之間並沒有太多的差異。作為 Java 程序員，您只會在使用處理程序時遇到這些差異，我們將在以後的技巧文章中對如何處理這種情況進行討論。HTTP 綁定具有自己的處理程序鏈和自己的一組消息上下文屬性。

WS-I Basic Profiles

JAX-RPC 1.1 支持 WS-I Basic Profile (BP) 1.0。從那時起，WS-I 人員就完成了 BP 1.1（以及關聯的 AP 1.0 和 SSBP 1.0）的開發。這些新概要闡明了一些小要點，更明確地定義了附件。JAX-WS 2.0 支持這些較新的概要。在大部分情況下，其間的差異並不會影響 Java 編程模型。不過附件除外。WS-I 不僅處理了有關附件的一些問題，而且還定義了自己的 XML 附件類型：wsi:swaRef。

很多人都被這些概要搞糊塗了。為了弄清楚其間的問題，將需要瞭解一下其相關歷史。

WS-I 的第一個基本概要 (BP 1.0) 在闡明各個規範方面做得非常不錯，但它並不完美。尤其對 SOAP with Attachments (Sw/A) 的支持仍然相當不明確。在第二個工作循環中，WS-I 人員將附件從基本概要 (BP 1.1) 中分離出來，並對第一版中一些沒有討論的內容進行了補充。當時他們還添加了兩個互不包括的基本概要補充文檔：AP 1.0 和 SSBP 1.0。AP 1.0 是附件概要 (Attachment Profile)，描述如何使用 Sw/A。SSBP 1.0 是簡單 SOAP 綁定概要

(Simple SOAP Binding Profile)，描述並不支持 Sw/A 的 Web 服務引擎（如 Microsoft 的 .NET）。WS-I 所提供的其他概要文件都是以這些基本概要文件為基礎構建的。

Java 5

對 Java 語言進行了一系列更改。JAX-WS 依賴於：Annotation、通用函數和執行程序。我們將在後續的技巧文章中具體討論 JAX-WS 如何依賴於這個新功能。有關 Java 的這些新功能的信息，請參見參考資料中的 Java 5 鏈接。

總結

JAX-WS 2.0 是 JAX-RPC 1.1 的後續版本。其中有些內容保持不變，但大部分編程模型都或多或少有些不同。本技巧文章中介紹的主題將在一系列技巧文章中展開討論，這個系列的文章對 JAX-WS 和 JAX-RPC 間的區別進行了詳細的討論，我們將在隨後的數月中陸續發佈。大致看來，可能會因為以下這些原因而決定從 JAX-RPC 遷移到 JAX-WS，或保持不變。

希望繼續使用 JAX-RPC 1.1 的原因：

- 如果您希望繼續使用現有的東西，JAX-RPC 將在今後一段時間內繼續得到支持。
- 如果您不希望升級到 Java 5。
- 如果您希望發送採用 SOAP 編碼的消息或創建 RPC/encoded 樣式的 WSDL。

升級到 JAX-WS 2.0 的原因：

- 如果您希望使用新的面向消息的 API。
- 如果您希望使用 MTOM 發送附件數據。
- 如果您希望通過 JAXB 更好地支持 XML 模式。
- 如果您希望在 Web 服務客戶機中採用異步編程模型。
- 如果您需要使用能夠處理 SOAP 1.2 消息的客戶機或服務。
- 如果您希望在 Web 服務中消除對 SOAP 的需求，而直接使用 XML/HTTP 綁定。
- 如果您喜歡使用領先的尖端技術。

Web 服務提示與技巧: JAX-RPC 與 JAX-WS，第 2 部分

引言

儘管 JAX-WS 2.0 中的一些方面是在 JAX-RPC 1.1 的基礎上進行的改進，但其他的部分卻是革命性的。例如，JAX-WS 沒有提供 XML 模式和 Java 之間的映射，而這是 JAX-RPC 1.1 中的一個重要特性。相反，JAX-WS 使用了另一種 JCP 定義的技術 JAXB (Java Architecture for XML Binding) 2.0，為其完成數據映射。JAX-WS 僅僅提供了 Web 服務調用模型。它不再關心表示應用程序數據的 Java Bean，而僅僅關注於將其提供給目標 Web 服務。在本文中，我們將比較 JAX-RPC 1.1 和 JAXB 2.0 的映射。

回頁首

從 XML 模式到 Java 的映射

在 JAX-RPC 和 JAX-WS/JAXB 中，將 XML 名稱映射到 Java 名稱的方式基本上是相同的，儘管在簡單類型的映射上有些細微的差異。表 1 介紹了這些差異，這些差異看起來非常顯著，然而除了日期/時間類型之外，其他的 XML 簡單類型都很少使用。

表 1. JAX-RPC 1.1 和 JAXB 2.0 中 XML 簡單類型映射的差異

類型	JAX-RPC 1.1	JAXB 2.0
xsd:anySimpleType	java.lang.String	java.lang.Object
xsd:duration	java.lang.String	javax.xml.datatype.Duration（新的類型）
xsd:dateTime	java.util.Calendar	javax.xml.datatype.XMLGregorianCalendar（新的類型）
xsd:time	java.util.Calendar	javax.xml.datatype.XMLGregorianCalendar
xsd:date	java.util.Calendar	javax.xml.datatype.XMLGregorianCalendar
xsd:gYearMonth	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gYear	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gMonthDay	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gMonth	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gDay	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:anyURI	java.net.URI	java.lang.String
xsd:NMTOKENS	java.lang.String[]	java.util.List
xsd:IDREF	java.lang.String	java.lang.Object

類型	JAX-RPC 1.1	JAXB 2.0
xsd:IDREFS	java.lang.String[]	java.util.List
xsd:ENTITY	not supported	java.lang.String
xsd:ENTITIES	not supported	java.util.List

在 JAX-RPC 和 JAXB 之間，簡單類型映射的純 Java 方面幾乎相同，但是 JAXB 映射還使用了新的 Java 註釋特性。清單 1 至 3 提供了一些簡單類型映射的示例。

清單 1. XML complexType 元素和屬性

```
<xsd:sequence>
  <xsd:element name="intField" type="xsd:int"/>
  <xsd:element name="intMinField" type="xsd:int" minOccurs="0"/>
  <xsd:element name="intNilField" type="xsd:int" nillable="true"/>
  <xsd:element name="stringField" type="xsd:string"/>
  <xsd:element name="stringMinField" type="xsd:string" minOccurs="0"/>
  <xsd:element name="stringNilField" type="xsd:string" nillable="true"/>
</xsd:sequence>
<xsd:attribute name="intAttr" type="xsd:int"/>
<xsd:attribute name="intAttrReq" type="xsd:int" use="required"/>
```

清單 2. 通過 JAX-RPC 1.1 映射為 Java Bean 屬性

```
private int intField;
   private Integer intMinField;
   private Integer intNilField;
   private String stringField;
   private String stringMinField;
   private String stringNilField;
   private Integer intAtt;
   private int intAttReq;
```

清單 3. 通過 JAXB 2.0 映射為 Java Bean 屬性

```
 protected int intField;
 protected Integer intMinField;

   @XmlElement(required = true, type = Integer.class, nillable = true)
   protected Integer intNilField;

   @XmlElement(required = true)
   protected String stringField;

   protected String stringMinField;
```

```
    @XmlElement(required = true, nillable = true)
    protected String stringNilField;

    @XmlAttribute
    protected Integer intAtt;

    @XmlAttribute(required = true)
    protected int intAttReq;
```

在 JAX-RPC 1.1 所生成的 Java Bean 中，您無法區分下面的差別：

- 元素字段和屬性字段
- 從 `minOccurs="0" type="xsd:int"` 映射的字段和從 `nillable="true" type="xsd:int"` 映射的字段
- 從 `type="xsd:string"` 映射的字段和從 `type="xsd:string" minOccurs="0"` 映射的字段

但由於 JAXB 使用了新的 Java 註釋，現在您可以區分出它們之間的差別。`@XmlElement` 和 `@XmlAttribute` 註釋具有一些選項。其中與本文相關的選項包括：

- Required：該元素是否必須存在？例如，`minOccurs` 是否不等於 1？
- Nillable：該字段是否包含 `nillable="true"` 屬性？

數組映射

在 JAX-RPC 和 JAXB 之間，從 XML 到 Java 的數組映射存在著差異，這是因為 JAXB 使用了新的泛型 Java 特性，如清單 4 至 6 所示。

清單 4. XML 數組

```
  <xsd:element name="intArrayField" type="xsd:int" minOccurs="0"
maxOccurs="unbounded"/>
```

清單 5. 通過 JAX-RPC 1.1 映射為 Java Bean 屬性

```
  private int[] intArrayField;
    public int[] getIntArrayField() {...}
    public void setIntArrayField(int[] intArrayField) {...}
    public int getIntArrayField(int i) {...}
    public void setIntArrayField(int i, int value) {...}
```

清單 6. 通過 JAXB 2.0 映射為 Java Bean 屬性

```
  @XmlElement(type = Integer.class)
    protected List<Integer> intArrayField;
    public List<Integer> getIntArrayField() {...}
```

注意訪問器方法中的差別。JAX-RPC 映射遵循 Java Bean 數組訪問器的嚴格定義。JAXB 映射並沒有映射為數組，所以有些不同。`getIntArrayField` 返回一個引用，不僅僅是快照，而是實際的列表。因此，您對返回的列表所做

的任何修改都將出現在屬性中。這就是為什麼對於數組屬性沒有 `set` 方法的原因。

複雜類型的映射

在 JAX-RPC 和 JAXB 中，對 `complexType` 的映射幾乎相同，除了這些字段在 JAX-RPC 中是私有的，而在 JAXB 中是保護的，並且 JAXB 映射添加了相應的註釋對字段順序進行描述。清單 7 至 9 提供了這兩種映射的示例和比較。

清單 7. XML 模式 complexType

```
  <xsd:element name="Phone" type="tns:Phone"/>
  <xsd:complexType name="Phone">
    <xsd:sequence>
      <xsd:element name="areaCode" type="xsd:string"/>
      <xsd:element name="exchange" type="xsd:string"/>
      <xsd:element name="number" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
```

清單 8. complexType 的 JAX-RPC 1.1 映射

```
  public class Phone {
    private String areaCode;
    private String exchange;
    private String number;

    ...
}
```

清單 9. complexType 的 JAXB 2.0 映射

```
  @XmlType(name = "Phone", propOrder = {
    "areaCode",
    "exchange",
    "number"
})
public class Phone {

    @XmlElement(required = true)
    protected String areaCode;
    @XmlElement(required = true)
    protected String exchange;
    @XmlElement(required = true)
    protected String number;

    ...
}
```

請注意，清單 7 中的模式使用了 `xsd:sequence`。如果它使用的是 `xsd:all`，那麼 `propOrder` 註釋應該為空：`@XmlType(name = "Phone", propOrder = {})`。

新的 ObjectFactory 類

JAXB 生成了一個文件 `ObjectFactory`，而 JAX-RPC 則沒有。每個包含 Java Bean 的目錄都會有一個 `ObjectFactory`。對於在模式相應的命名空間中定義的每種類型，`ObjectFactory` 類都為該類型提供了一種創建方法。對於每個元素，`ObjectFactory` 類提供了一種創建元素的方法，該方法返回一個 `javax.xml.bind.JAXBElement<Type>`。例如，對於清單 7 中的 Phone 模式，清單 10 顯示了所生成的 `ObjectFactory` 類，這個類具有一個返回 `Phone` 的實例和一個返回 `JAXBElement<Phone>` 的實例的方法。您仍然可以直接實例化該目錄中的 Java Bean，但最好是使用這個工廠。

清單 10. Phone 的 JAXB 2.0 ObjectFactory

```
import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.namespace.QName;

public class ObjectFactory {

    private final static QName _Phone_QNAME = new QName
    ("urn:types.MyServiceSample.ibm.com", "Phone");

    public ObjectFactory() {...}

    public Phone createPhone() {...}

    @XmlElementDecl(namespace = "urn:types.MyServiceSample.ibm.com", name = "Phone")
    public JAXBElement<Phone> createPhone(Phone value) {...}
}
```

回頁首

從 Java 到 XML 模式的映射

靈活且少量地使用這些新的 JAXB 定義的註釋，Java 可以以一種良好控制的方式映射到 XML，尤其是在對已提供的從 XML 到 Java 的映射進行逆映射的情況下。但是，如果使用沒有註釋的 Java 又會如何呢？

在 JAX-RPC 和 JAXB 中，將 Java 名稱映射到 XML 名稱基本上是相同的。也就是說，無論映射遵循 JAX-RPC 或 JAXB，Java 基本數據類型可以映射到相同的 XML 模式。JAX-RPC 定義了一組映射到 XML 的標準 Java 類。對於所有這些類（除了其中一個之外），JAXB 可以進行完全相同的映射，但是 JAXB 向映射類列表中添加了更多的類，如表 2 所示。

表 2. JAX-RPC 1.1 和 JAXB 2.0 之間在標準 Java 類映射上的差別

類型	JAX-RPC 1.1	JAXB 2.0
java.lang.String	xsd:string	xsd:string

類型	JAX-RPC 1.1	JAXB 2.0
java.math.BigInteger	xsd:integer	xsd:integer
java.math.BigDecimal	xsd:decimal	xsd:decimal
java.util.Calendar	xsd:dateTime	xsd:dateTime
java.util.Date	xsd:dateTime	xsd:dateTime
javax.xml.namespace.QName	xsd:QName	xsd:QName
java.net.URI	xsd:anyURI	xsd:string
javax.xml.datatype.XMLGregorianCalendar	n/a	xsd:anySimpleType
javax.xml.datatype.Duration	n/a	xsd:duration
java.lang.Object	n/a[1]	xsd:anyType
java.awt.Image	n/a[2]	xsd:base64Binary
javax.activation.DataHandler	n/a[2]	xsd:base64Binary
javax.xml.transform.Source	n/a[2]	xsd:base64Binary
java.util.UUID	n/a	xsd:string

表 2 中需要注意的地方：
1. 一些供應商將 java.lang.Object 映射為 xsd:anyType。
2. 在 JAX-RPC 中，這被映射為 MIME 綁定類型，即未定義的 XML 類型。

回頁首

結束語

我們比較了 JAX-RPC 1.1 和 JAXB 2.0 的映射。除了一些值得注意的地方外，它們基本上是類似的：

- 從 XML 到 Java：
  - 對一些簡單類型的映射存在差別
  - JAXB 添加了 ObjectFactory 類
  - JAXB 使用了新的 Java 語言註釋特性和泛型特性
- 從 Java 到 XML：
  - 基本數據類型的映射是相同的
  - 標準 Java 映射基本上相同

有關 JAXB 2.0 規範，還有很多的內容，特別是 Java 註釋的使用，但是這個主題已經超出了本文的範圍。有關 JAXB 2.0 的更多信息，請參見參考資料部分。

Web 服務提示與技巧: JAX-RPC 與 JAX-WS 的比較，第 3 部分

引言

從整體上看，Java API for XML-based RPC (JAX-RPC) 1.1 服務端點接口 (SEI) 和 Java API for XML Web Services (JAX-WS) 2.0 SEI 是非常相似的。本文將著重介紹它們之間的區別。儘管在結構上存在一定的區別，然而，提供反映 Web 服務契約的接口的目標是相同的。

比較 SEI 映射

清單 1 顯示了一個簡單的 HelloWorld Web 服務所使用的 WSDL。

清單 1. HelloWorld WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
    <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="urn:helloWorld/sample/ibm/com"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="HelloWorld"
    targetNamespace="urn:helloWorld/sample/ibm/com">
<wsdl:types>
  <xsd:schema targetNamespace="urn:helloWorld/sample/ibm/com"
      xmlns:tns="urn:helloWorld/sample/ibm/com"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="hello">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" nillable="true" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="helloResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="response" nillable="true" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>
<wsdl:message name="helloRequestMsg">
  <wsdl:part element="tns:hello" name="helloParameters" />
```

```
  </wsdl:message>
  <wsdl:message name="helloResponseMsg">
    <wsdl:part element="tns:helloResponse" name="helloResult" />
  </wsdl:message>
  <wsdl:portType name="HelloWorld">
    <wsdl:operation name="hello">
      <wsdl:input message="tns:helloRequestMsg" name="helloRequest" />
      <wsdl:output message="tns:helloResponseMsg" name="helloResponse" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="HelloWorldBinding" type="tns:HelloWorld">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"
/>
    <wsdl:operation name="hello">
      <soap:operation soapAction="urn:helloWorld/sample/ibm/com/hello" />
      <wsdl:input name="helloRequest">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="helloResponse">
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="HelloWorldService">
    <wsdl:port name="port" binding="tns:HelloWorldBinding">
      <soap:address location="http://tempuri.org/" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

清單 2 顯示了這個 WSDL 中 Java SEI 的 JAX-RPC 映射，其中不包含方法簽名（稍後您將看到有關這些內容的細節）。

清單 2. JAX-RPC HelloWorld SEI

```
package com.ibm.samples;
public interface HelloWorld extends java.rmi.Remote {
...
}
```

清單 3 顯示了用於相同的 WSDL 的 JAX-WS SEI。

清單 3. JAX-WS HelloWorld SEI

```
package com.ibm.samples.helloworld;

import javax.jws.WebService;

@WebService(name = "HelloWorld", targetNamespace = "urn:samples.ibm.com/HelloWorld")
```

```
public interface HelloWorld {
  ...
}
```

其中存在三處區別：

- 包：目標命名空間是「urn:helloWorld/sample/ibm/com」。這兩個映射都使用了類似域名的字符串，並且顛倒了其中元素的順序。JAX-RPC 的映射在第一個斜槓處結束。JAX-WS 的映射則繼續該字符串，在第一個斜槓後面繼續添加信息。這兩種規範都允許從自定義的命名空間到包的映射。
- 註釋：JAX-WS 要求所有 SEI 必須包括 `@WebService` 註釋。正如本系列文章的第 1 部分中所介紹的，JAX-WS 支持 JSR-181 Web 服務元數據中定義的各種註釋。
- `java.rmi.Remote`：JAX-RPC SEI 擴展了 `java.rmi.Remote` 接口。而 JAX-WS 不需要這樣做。

在繼續學習操作映射的詳細內容之前，關於 SEI 本身還有一處需要說明的地方。儘管 JAX-WS 提供對具有 SEI 的 Web 服務的支持，但是對於所有的服務來說，這並不是強制的。使用 JAX-WS，JavaBean 可以作為 Web 服務實現而自行部署，不需要像 JAX-RPC 那樣要求 Bean 必須包括 SEI。所部署的沒有 SEI 的 JAX-WS 服務將被認為具有*隱式的 SEI*。

回頁首

比較操作映射

現在您已經看到了這些接口，下面再對各種操作的映射進行比較。可以使用不同的方法來設計表示具有類似語義的 Web 服務的 WSDL 文檔。文章 Which style of WSDL should I use? 提供了關於各種不同風格的 WSDL 文檔的概述，以及如何確定哪一種風格更適合您。

現在，讓我們來研究 JAX-RPC 和 JAX-WS 如何映射到每種 WSDL 風格。

研究 document/literal 包裝模式

清單 1 中的 WSDL 使用了 document/literal 包裝模式進行格式化。清單 4 和 5 是 JAX-RPC 和 JAX-WS 中對相同的包裝操作的映射。請注意，JAX-WS 向方法中添加了 `@RequestWrapper` 和 `@ResponseWrapper` 註釋。這些註釋為用作操作包裝器的元素和為這些包裝器元素生成的任何 Java Bean 提供了附加的元數據。這些註釋都是可選的。

清單 4. JAX-RPC 完整的 HelloWorld SEI

```
package com.ibm.samples;

public interface HelloWorld extends java.rmi.Remote {
    public java.lang.String hello(java.lang.String name) throws
java.rmi.RemoteException;
}
```

清單 5. JAX-WS 完整的 HelloWorld SEI

```
package com.ibm.samples.helloworld;

import javax.jws.WebMethod;
import javax.jws.WebParam;
```

```
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(name = "HelloWorld", targetNamespace = "urn:samples.ibm.com/HelloWorld")
public interface HelloWorld {

    @WebMethod(action = "urn:samples.ibm.com/HelloWorld/hello")
    @WebResult(name = "response", targetNamespace = "")
    @RequestWrapper(localName = "hello",
        targetNamespace = "urn:samples.ibm.com/HelloWorld",
        className = "com.ibm.samples.helloworld.Hello")
    @ResponseWrapper(localName = "helloResponse",
        targetNamespace = "urn:samples.ibm.com/HelloWorld",
        className = "com.ibm.samples.helloworld.HelloResponse")
    public String hello(
        @WebParam(name = "name", targetNamespace = "")
        String name);
}
```

正如您所看到的，JAX-WS 映射包含大量的註釋，但是當您仔細研究根簽名時會發現，唯一的區別是JAX-RPC 方法可以引發 `java.rmi.RemoteException`，而 JAX-WS 方法則不行。

研究 document/literal 模式

JAX-RPC 和 JAX-WS 都支持 document/literal 映射操作，但不支持包裝映射操作。要在 HelloWorld 示例中實現這一點，您需要刪除表示操作名的包裝器元素。清單 6 顯示了這個 WSDL 文檔中相關的部分，並與清單 1 中的 WSDL 進行對比。

清單 6. 文檔/文本 WSDL

```
<wsdl:types>
  <xsd:schema targetNamespace="urn:helloWorld/sample/ibm/com"
   xmlns:tns="urn:helloWorld/sample/ibm/com"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="hello" type="xsd:string"/>
      <xsd:element name="helloResponse" type="xsd:string"/>
  </xsd:schema>
</wsdl:types>

<wsdl:message name="helloRequestMsg">
  <wsdl:part element="tns:hello" name="helloParameters" />
</wsdl:message>

<wsdl:message name="helloResponseMsg">
  <wsdl:part element="tns:helloResponse" name="helloResult" />
</wsdl:message>
```

現在，讓我們來研究用於這個新的 WSDL 的 Java 映射。清單 7 和 8 分別顯示了 JAX-RPC 和 JAX-WS 映射。JAX-

RPC 映射是非常類似的！唯一的區別是參數名稱。與前面的情況一樣，如果不考慮註釋，JAX-RPC 映射和 JAX-WS 映射之間沒有什麼本質的區別。

清單 7. JAX-RPC 文檔/文本映射

```
public interface HelloWorld extends java.rmi.Remote {
    public java.lang.String hello(java.lang.String helloParameters)
        throws java.rmi.RemoteException;
}
```

清單 8. JAX-WS 文檔/文本映射

```
@WebService(name = "HelloWorld", targetNamespace = "urn:helloWorld/sample/ibm/com")
@SOAPBinding(parameterStyle = ParameterStyle.BARE)
public interface HelloWorld {

    @WebMethod(action = "urn:helloWorld/sample/ibm/com/hello")
    @WebResult(name = "helloResponse",
        targetNamespace = "urn:helloWorld/sample/ibm/com",
        partName = "helloResult")
    public String hello(
        @WebParam(name = "hello",
            targetNamespace = "urn:helloWorld/sample/ibm/com",
            partName = "helloParameters")
        String helloParameters);

}
```

請注意，對於 JAX-WS，沒有 `@RequestWrapper` 和 `@ResponseWrapper` 註釋。另外請注意，在接口級別上還出現了一個新的註釋，`@SOAPBinding`。這個註釋提供了關於參數風格的信息。如果沒有這個註釋，parameterStyle 屬性的缺省值為 wrapped，應該與清單 1 中的 WSDL 類似。

研究 RPC/literal 模式

下面的示例與前兩個示例有一些不同。對於 RPC/literal 風格的 WSDL，這些部分定義為類型而不是元素。清單 9 包含了相關的 WSDL 差別。

清單 9. RPC/文本 WSDL 更改

```
<wsdl:types/>

<wsdl:message name="helloRequestMsg">
  <wsdl:part name="helloParameters" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="helloResponseMsg">
  <wsdl:part name="helloResult" type="xsd:string"/>
</wsdl:message>
```

清單 10 和 11 中的 Java 映射反映了對 WSDL 的更改。同樣地，當去掉註釋後，可以得到完全相同的映射。

清單 10. JAX-RPC RPC/文本映射

```
public interface HelloWorld extends java.rmi.Remote {
    public java.lang.String hello(java.lang.String helloParameters)
        throws java.rmi.RemoteException;
}
```

清單 11. JAX-WS RPC/文本映射

```
@WebService(name = "HelloWorld", targetNamespace = "urn:helloWorld/sample/ibm/com")
@SOAPBinding(style = Style.RPC)
public interface HelloWorld {

    @WebMethod(action = "urn:helloWorld/sample/ibm/com/hello")
    @WebResult(name = "helloResult", partName = "helloResult")
    public String hello(
        @WebParam(name = "helloParameters", partName = "helloParameters")
        String helloParameters);

}
```

將這個 JAX-WS 接口與前面的進行比較，您將看到保留了 `@SOAPBinding` 註釋，但是現在它不是表示參數風格，而是表示 WSDL 風格。

研究 RPC/encoded 模式

對於 RPC/encoded 風格的操作，無法進行比較。JAX-WS 不支持包含*編碼的*數據表示的 WSDL 文檔的任何映射。這是因為 JAX-WS 與 WS-I 的 Basic Profile 1.1 的兼容性原因，其中不允許使用*編碼的* WSDL 文檔。有時候的確需要構建 RPC/encoded Web 服務，在這種情況下您應該堅持使用 JAX-RPC 映射，但是如果您希望編寫可互操作的 Web 服務，那麼您就不應該使用 RPC/encoded 模式。

回頁首

考慮其他的一些區別

JAX-WS 與 JAX-RPC 在操作映射方面的主要區別是前者引入了異步操作。任何具有雙向消息流的 WSDL 操作，或者其中的客戶端期待著接收響應的操作，都可以映射為異步的 Java 表示形式。有兩種不同的機制，帶回調的異步方式和異步輪詢，它們需要兩種不同的映射。後續的文章將描述這兩種類型的操作是如何工作的。本文只是向您介紹一個示例。清單 12 包含一個異步回調操作，其中 `javax.xml.ws.AsyncHandler` 對象為回調對象。清單 13 包含了一個異步輪詢操作映射。

清單 12. JAX-WS 異步回調

```
@WebMethod(action = "urn:samples.ibm.com/HelloWorld/hello")
@RequestWrapper(localName = "hello",
    targetNamespace = "urn:samples.ibm.com/HelloWorld",
```

```
    className = "com.ibm.samples.helloworld.Hello")
@ResponseWrapper(localName = "helloResponse",
    targetNamespace = "urn:samples.ibm.com/HelloWorld",
    className = "com.ibm.samples.helloworld.HelloResponse")
public Future<?> helloAsync(
    @WebParam(name = "name", targetNamespace = "")
    String name,
    @WebParam(name = "asyncHandler", targetNamespace = "")
    AsyncHandler<String> asyncHandler);
```

清單 13. JAX-WS 異步輪詢

```
@WebMethod(action = "urn:samples.ibm.com/HelloWorld/hello")
@RequestWrapper(localName = "hello",
    targetNamespace = "urn:samples.ibm.com/HelloWorld",
    className = "com.ibm.samples.helloworld.Hello")
@ResponseWrapper(localName = "helloResponse",
    targetNamespace = "urn:samples.ibm.com/HelloWorld",
    className = "com.ibm.samples.helloworld.HelloResponse")
public Response<String> helloAsync(
    @WebParam(name = "name", targetNamespace = "")
    String name);
```

在 JAX-RPC 中，沒有用於 WSDL 操作的異步映射，所以這裡沒有辦法進行比較。然而，重要的是，異步映射僅適用於客戶端。對於服務端點，不存在類似的異步映射，只能用於客戶端。

比較 IN/OUT 參數

JAX-RPC 和 JAX-WS 都支持稱為 IN/OUT 參數的參數。在清單 14 中，您可以看到在清單 1 的 WSDL 中添加了一個 IN/OUT 參數。請注意，不管對於輸入還是輸出，都出現了名為「inout」的參數。在這個場景中，JAX-RPC 和 JAX-WS 將該參數映射為一個 Holder 參數，但是對於不同的映射，其效果是不同的。

清單 14. 帶 IN/OUT 參數的 WSDL

```
<xsd:element name="hello">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" nillable="true" type="xsd:string" />
      <xsd:element name="inout" nillable="true" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="helloResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="response" nillable="true" type="xsd:string" />
      <xsd:element name="inout" nillable="true" type="xsd:string" />
    </xsd:sequence>
```

```
  </xsd:complexType>
</xsd:element>
```

清單 15 提供了 Holder 參數的 JAX-RPC 映射，而清單 16 提供了 JAX-WS 映射。

清單 15. 帶 IN/OUT 參數的 JAX-RPC SEI

```
public interface HelloWorld extends java.rmi.Remote {
    public java.lang.String hello(
        java.lang.String name,
        javax.xml.rpc.holders.StringHolder inout) throws java.rmi.RemoteException;
}
```

清單 16. 帶 IN/OUT 參數的 JAX-WS SEI

```
@WebService(name = "HelloWorld", targetNamespace = "urn:helloWorld/sample/ibm/com")
public interface HelloWorld {

    @WebMethod(action = "urn:helloWorld/sample/ibm/com/hello")
    @RequestWrapper(localName = "hello",
        targetNamespace = "urn:helloWorld/sample/ibm/com",
        className = "helloworld.sample.ibm.com.Hello")
    @ResponseWrapper(localName = "helloResponse",
        targetNamespace = "urn:helloWorld/sample/ibm/com",
        className = "helloworld.sample.ibm.com.HelloResponse")
    public void hello(
        @WebParam(name = "name", targetNamespace = "")
        String name,
        @WebParam(name = "inout", targetNamespace = "", mode = Mode.INOUT)
        Holder<String> inout,
        @WebParam(name = "response", targetNamespace = "", mode = Mode.OUT)
        Holder<String> response);

}
```

對於 JAX-RPC，規範中為各種已知的類型定義了一組類作為 Holder 類。這些類型包括 `java.lang.String` 和其他基本數據類型。對於用戶定義的類型，JAX-RPC 需要生成能夠處理用戶定義類型的自定義的 Holder 類。另一方面，對於 JAX-WS，可以使用 Java 5 的泛型編程特性，提供適用於所有類型（包括用戶定義類型）的單個類。

另一個有趣的內容是返回類型的區別。JAX-WS 並不像 JAX-RPC 那樣保存返回類型，而是讓方法的返回類型為 void，並使用 Holder 來獲取返回值。根據 JAX-WS 中的規則，如果對於一個操作存在多個可以作為 OUT 參數的參數，那麼返回類型必須為 void，並且將所有的 OUT 參數映射為 Holder 類型。

回頁首

總結

上面的示例說明，儘管 JAX-RPC 和 JAX-WS 之間存在很多的區別，但是從 WSDL 到服務端點接口結構的映射是非常類似的。關鍵的區別包括：

- 包名不同。
- JAX-RPC 需要 java.rmi.Remote 和 java.rmi.RemoteException，而 JAX-WS 不需要。
- 對 Holder 進行了不同的定義。

即使存在這些類似的地方，但是有一個主要的區別，使得 JAX-WS SEI 不同於 JAX-RPC。JSR-181 註釋的使用，使得 JAX-WS SEI 可以以 Java 為中心或者以 WSDL 為中心來表示 Web 服務。可以包括大量的註釋，以便將 Java 信息映射回 WSDL 構件。JAX-RPC SEI 無法通過任何形式來提供這類信息。還有一些特性是 JAX-WS 所特有的，即異步調用模型，以及它不需要提前生成 SEI。另一方面，JAX-RPC 也具備一些 JAX-WS 所沒有的特性：它支持 RPC/encoded WSDL。

Web 服務提示與技巧: JAX-RPC 與 JAX-WS 的比較，第 4 部分

引言

JAX-RPC 1.1 和 JAX-WS 2.0 客戶機動態模型都提供一組類似的抽象步驟來進行調用。

1. 定義服務。
2. 從此服務創建動態調用對象。
3. 構建消息。
4. 調用操作。

儘管這兩個模型都採用相同的步驟，不過本文將說明兩個模型間細節的差異。我們將使用前一篇文章中所用的 HelloWorld Web 服務描述語言（Web Services Description Language，WSDL）。具體如清單 1 中所示。

清單 1. HelloWorld 服務的 WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
      <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:tns="urn:helloWorld/sample/ibm/com"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="HelloWorld"
      targetNamespace="urn:helloWorld/sample/ibm/com">
  <wsdl:types>
    <xsd:schema targetNamespace="urn:helloWorld/sample/ibm/com"
        xmlns:tns="urn:helloWorld/sample/ibm/com"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="hello">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" nillable="true" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="helloResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="response" nillable="true" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
```

```
  <wsdl:message name="helloRequestMsg">
    <wsdl:part element="tns:hello" name="helloParameters" />
  </wsdl:message>
  <wsdl:message name="helloResponseMsg">
    <wsdl:part element="tns:helloResponse" name="helloResult" />
  </wsdl:message>
  <wsdl:portType name="HelloWorld">
    <wsdl:operation name="hello">
      <wsdl:input message="tns:helloRequestMsg" name="helloRequest" />
      <wsdl:output message="tns:helloResponseMsg" name="helloResponse" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="HelloWorldBinding" type="tns:HelloWorld">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"
/>
    <wsdl:operation name="hello">
      <soap:operation soapAction="urn:helloWorld/sample/ibm/com/hello" />
      <wsdl:input name="helloRequest">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="helloResponse">
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="HelloWorldService">
    <wsdl:port name="port" binding="tns:HelloWorldBinding">
      <soap:address location="http://tempuri.org/" />
    </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

回頁首

JAX-RPC 的動態調用接口

JAX-RPC 的動態調用接口（Dynamic Invocation Interface，DII）是 `Call` 對象 (`javax.xml.rpc.Call`)。清單 2 中給出了使用 `Call` 對象調用清單 1 的 HelloWorld 服務的完整客戶機主類。還可以在清單 2 的介紹中看到對抽象步驟進行了說明。

清單 2. JAX-RPC 的 DII 客戶機

```
package com.ibm.samples.dii;

import javax.xml.namespace.QName;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
```

```
import javax.xml.rpc.encoding.XMLType;

public class HelloWorldClient {

    public static void main(String[] args) {
        try {
            // Define the service.
            QName serviceName = new QName(
                    "urn:helloWorld/sample/ibm/com",
                    "HelloWorldService");
            ServiceFactory factory = ServiceFactory.newInstance();
            Service service = factory.createService(serviceName);

            // Create the dynamic invocation object from this service.
            Call call = service.createCall();
            call.setTargetEndpointAddress(
                    "http://localhost:9081/HelloWorldService/services/port");

            // Build the message.
            QName operationName = new QName(
                    "urn:helloWorld/sample/ibm/com",
                    "hello");
            call.setOperationName(operationName);
            call.addParameter(
                    "name",             // parameter name
                    XMLType.XSD_STRING, // parameter XML type QName
                    String.class,       // parameter Java type class
                    ParameterMode.IN);  // parameter mode
            call.setReturnType(XMLType.XSD_STRING);
            call.setProperty(
                    Call.OPERATION_STYLE_PROPERTY,
                    "wrapped");

            // Invoke the operation.
            Object[] actualArgs = {"Georgia"};
            String response = (String) call.invoke(actualArgs);
            System.out.println("response = " + response);
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
    }
    }
```

下面讓我們瞭解一下這些抽象步驟的細節：

1. **定義服務。**通過使用 WSDL 的服務完全限定名稱構造 `javax.xml.rpc.Service` 對象。
2. **從此服務創建動態調用對象。**在 JAX-RPC 中，動態調用對象為 `javax.xml.rpc.Call`。

3. **構建消息。**在此步驟中，將使用關於操作的信息填充 `Call` 對象。此處需要提出的一點是：我們將調用 `call.setProperty(Call.OPERATION_STYLE_PROPERTY, "wrapped");`。`wrapped` 不是 JAX-RPC 為此屬性定義的值。JAX-RPC 僅定義 `rpc` 和 `document`。不過，`rpc` 實際上表示 *RPC/Encoded*，而 `document` 實際表示 *Document/Literal, Non-Wrapped*。自從 JAX-RPC 推出後，Document/Literal Wrapped 模式就成為了行業標準，因此其 `Call` 對象並不能很好地對其進行處理。可以完成此工作，但效果不太好。對此屬性進行擴展，以包括 `wrapped` 值，這是 IBM 填補此規範空白的方法，但並非標準擴展。
4. **調用操作。**此示例中的輸入是一個簡單的字符串，因此將使用字符串填充參數，並將其傳遞給調用函數。響應也是字符串，在此示例中將直接對其進行顯示。

JAX-WS 的動態 Dispatch 接口

JAX-WS 的 DII 是 `Dispatch` 對象 (`javax.xml.ws.Dispatch`)。清單 3 中給出了通過 `Dispatch` 對象調用清單 1 的 HelloWorld 服務的完整客戶機主類。可以在清單 3 的介紹中看到對抽象步驟進行了說明。

清單 3. JAX-WS 的 DII 客戶機

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import javax.xml.namespace.QName;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;

public class HelloWorldClient {

    public static void main(String[] args) {
        try {
            // Define the service.
            QName svcQname = new QName(
                    "urn:helloWorld/sample/ibm/com",
                    "HelloWorldService");
            QName portQName = new QName(
                    "urn:helloWorld/sample/ibm/com",
                    "port");
            Service svc = Service.create(svcQname);
            svc.addPort(
                    portQName,
                    SOAPBinding.SOAP11HTTP_BINDING,
                    "http://localhost:9080/JAXBSampleWebService/HelloWorldService");

            // Create the dynamic invocation object from this service.
            Dispatch<Source> dispatch = svc.createDispatch(
                    portQName,
```

```
                    Source.class,
                    Service.Mode.PAYLOAD);

            // Build the message.
            String content =
                    "<ns2:hello xmlns:ns2=\"urn:helloWorld/sample/ibm/com\">" +
                      "<name>Georgie</name>" +
                    "</ns2:hello>";
            ByteArrayInputStream bais = new
ByteArrayInputStream(content.getBytes());
            Source input = new StreamSource(bais);

            // Invoke the operation.
            Source output = dispatch.invoke(input);

            // Process the response.
            StreamResult result = new StreamResult(new ByteArrayOutputStream());
            Transformer trans = TransformerFactory.newInstance().newTransformer();
            trans.transform(response, result);
            ByteArrayOutputStream baos = (ByteArrayOutputStream)
result.getOutputStream();

            // Write out the response content.
            String responseContent = new String(baos.toByteArray());
            System.out.println(responseContent);
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

接下來讓我們看看此 JAX-WS 動態客戶機中的這些抽象步驟的細節：

1. **定義服務。**與 JAX-RPC 模型中類似，將基於完全限定名稱定義服務，但還要定義該服務的端口。
2. **從此服務創建動態調用對象。**在 JAX-WS 中，動態調用對象為 `javax.xml.ws.Dispatch`。這個類利用了 Java 5 中的新泛型功能支持多參數類型。
3. **構建消息。**在此處構建原始 SOAP 消息主題內容。
4. **調用操作。**您將 SOAP 主題的內容發送到服務，並從服務接收 SOAP 響應。

回頁首

模型間的差異

從抽象步驟來看，JAX-RPC 和 JAX-WS 的動態調用模型非常相似。但事實並非如此。看看每個示例的細節，就會發現這些模型實際上有很大的區別。接下來讓我們更為深入地分析一下其中的一些差異。

操作樣式

相當清楚的是，兩個模型之間的主要差異在於，JAX-RPC 採用遠程過程調用（Remote Procedure Call，RPC）模型

（即規範名稱中的 *RPC*），而 JAX-WS 的動態客戶機採用消息傳遞模型。對於 JAX-RPC，必須配置 `Call` 對象來顯式地從 WSDL 調用特定操作。在 JAX-WS 中，`Dispatch` 對象不知道所調用的是哪個操作，僅僅負責發送 XML 數據而已。

參數樣式

對於 JAX-RPC，所傳遞的參數類型保留規範定義的 XML 模式到 Java 的映射。對於 JAX-WS，可以採用兩種不同的方式使用 `Dispatch` 對象：作為基於 XML 的 `Dispatch`（如本文中所示）或作為基於 JAXB 的 `Dispatch`。基於 XML 的 `Dispatch` 接受基於以下對象類型的參數：`javax.xml.transform.Source`、`javax.xml.soap.SOAPMessage` 和 `javax.activation.DataSource`。基於 JAXB 的 `Dispatch` 要求用戶配置可用於封送和取消封送參數實例（JAXB Java Bean）的 `javax.xml.bind.JAXBContext`。

參數樣式之間的另一個主要差異在於所傳遞的內容。對於 JAX-RPC，參數始終作為請求的參數數據使用。而對於 JAX-WS，有兩種不同的參數模式：有效負載 (PAYLOAD) 和消息 (MESSAGE) 模式。PAYLOAD 模式表示參數僅僅代表 SOAP 主體的內容，而 MESSAGE 模式表示參數代表整個消息，包括 SOAP 信封。

調用模式

如上所述，JAX-RPC 和 JAX-WS 都通過 `invoke()` 方法提供了同步雙向調用。這兩個模型都提供了用於通過 `invokeOneWay` 方法調用單向 Web 服務操作的方法。兩個模型之間的主要差異在於，JAX-WS 還提供異步調用模型。其中包含異步回調模型和異步輪詢模型。JAX-RPC 並不提供異步調用選項。

服務器端動態編程模型

此處將不會詳細討論的另一個主要差異是，JAX-WS 添加了 JAX-RPC 永遠不會有的內容——動態服務器端編程模型。與客戶端模型類似，也可以將其配置為使用 PAYLOAD 或 MESSAGE 模式。

回頁首

總結

JAX-RPC 和 JAX-WS 都提供了動態客戶機模型。簡單而言，可以將二者視為對等。但仔細分析其細節，可以發現 JAX-RPC 的動態模型是 RPC 模型，而 JAX-WS 的動態模型為消息傳遞模型。JAX-WS API 可提供更大的靈活性，是 Web 服務編程模型發展的下一步目標。JAX-WS 還提供了異步支持和動態服務支持，而 JAX-RPC 並未對此進行定義。

Web 服务提示与技巧: JAX-RPC 与JAX-WS 的比较，第5部分

时间: 2010-10-09 19:50 来源: 互联网 作者: 互联网 点击: 1034 次

级别： 中级Russell Butek (butek@us.ibm.com), IT 专家, IBM Nicholas Gallardo (nlgallar@us.ibm.com), 软件工程师, I

级别： 中级

Russell Butek (butek@us.ibm.com), IT 专家,

IBM

Nicholas Gallardo (nlgallar@us.ibm.com), 软件工程师, IBM

2008 年 2 月 04 日

Java™ API for XML-based RPC (JAX-RPC) 支持

推荐内容

SystemImager安装

安装环境 SystemImager服务器：CentOS release5.5 (Fina...

JS实现遮罩层效果

function show(){ var cover = document.getElementById("cover")

How to update a module

I looked for instructions on how to update a Drupal module, but could

sqlite使用

sqlite常量的定义： const SQLITE_OK = 0; 返回成功 SQLITE_ERROR = 1;

bat文件

bat是dos下的批处理文件。批处理文件是无格式的文本文...

Oracle goldgate常用命令

公司使用Oracle 的goldgate已经半

SOAP with Attachments (Sw/A) 规范，而 Java API for XML Web Services (JAX-WS) 支持 Sw/A 以及新推出的消息传输优化机制（Message Transmission Optimization Mechanism，MTOM）规范。本文是本系列的第 5 部分，将通过分析 Web 服务描述语言（Web Services Description Language，WSDL）和映射 Java 接口示例对这两个附件模型进行比较。

引言

JAX-RPC 的附件模型为 Sw/A，自从编写 JAX-RPC 后，就推出了一个新的附件模型：MTOM。JAX-WS 和 JAX-RPC 一样提供了 Sw/A 支持，但还增加了对 MTOM 的支持。JAX-WS 通过 Java Architecture for XML Binding (JAXB) 规范支持 MTOM，此规范包括用于对 Sw/A 和 MTOM 附件进行封送和取消封送。在本文中，我们将通过示例对这两个模型进行分析。请注意： 本文仅对 WSDL 和 Java 编程模型进行比较，连接级别的消息比较需由读者自行完成。

回页首

JAX-RPC 和 Sw/A 示例

清单 1 给出了摘自多年前所撰写的关于附件提示（请参见参考资料）的 WSDL。清单 2 显示了对应的 Java 接口（即从此 WSDL 生成的 JAX-RPC 映射）。

清单 1. JAX-RPC Sw/A WSDL

```
<definitions
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://attachment.tip/"
```

```
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
      targetNamespace="http://attachment.tip/">
<types/>
<message name="sendImage">
  <part name="image" type="xsd:base64Binary"/>
</message>
<message name="sendImageResponse"/>
<message name="sendOctet">
  <part name="octet" type="xsd:base64Binary"/>
</message>
<message name="sendOctetResponse"/>
<portType name="AttachmentTip">
  <operation name="sendImage">
    <input message="tns:sendImage"/>
    <output message="tns:sendImageResponse"/>
  </operation>
  <operation name="sendOctet">
    <input message="tns:sendOctet"/>
    <output message="tns:sendOctetResponse"/>
  </operation>
</portType>
<binding name="AttachmentBinding" type="tns:AttachmentTip">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sendImage">
    <soap:operation soapAction=""/>
    <input>
```

```
      <mime:multipartRelated>
        <mime:part>
          <soap:body parts="" namespace="http://attachment.tip/" use="literal"/>
        </mime:part>
        <mime:part>
          <mime:content part="image" type="image/jpeg"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
    <output>
      <soap:body namespace="http://attachment.tip/" use="literal"/>
    </output>
  </operation>
  <operation name="sendOctet">
    <soap:operation soapAction=""/>
    <input>
      <mime:multipartRelated>
        <mime:part>
          <soap:body parts="" namespace="http://attachment.tip/" use="literal"/>
        </mime:part>
        <mime:part>
          <mime:content part="octet" type="application/octet-stream"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
    <output>
      <soap:body namespace="http://attachment.tip/" use="literal"/>
```

```
      </output>
    </operation>
  </binding>
  <service name="AttachmentService">
    <port binding="tns:AttachmentBinding" name="AttachmentTip">
      <soap:address location="http://localhost:9080/SwAService/services/AttachmentTip"/
    </port>
  </service>
  </definitions>
```

清单 2. JAX-RPC Sw/A Java 接口

```
package tip.attachment;

import java.awt.Image;
import java.rmi.Remote;
import java.rmi.RemoteException;

import javax.activation.DataHandler;

public interface AttachmentTip extends Remote {
    public void sendImage(Image image) throws RemoteException;
    public void sendOctet(DataHandler octet) throws RemoteException,
}
```

从 WSDL 生成的 Java 接口映射相当简单：sendImage 操作的 image/jpeg 部分映射为 java.awt.Image 参数；sendOctet 操作的 application/octet-stream 映射为 javax.activation.DataHandler 参数。不过此映射有一定的代价：多用途 Internet 邮件扩展（Multipurpose Internet Mail Extensions，MIME）类型信息不在 WSDL 接口部分中（portType、message 和 types 部分）。您必须自己对绑定进行分析，以找到 MIME 信息。这很不方便，因为相同 WSDL 接口的不同绑定可能会采用不同的类型，从而映射到不同的 Java 接口。如果从 WSDL 接口到 Java 接口之间存在一对一映射关系，这就要好得多。如果有多个绑定，每个都映射到一个实现，则所有绑定都实现相同的 Java 接口。使用 Sw/A 时不能做到这一点；从 WSDL 到 Java 的映射不是一对一的。

Sw/A 的第二个问题是，它本身对 Document/literal Wrapped WSDL 行业惯例遵循性并不好（您会注意到清单 1 中的 WSDL 是 rpc/literal WSDL）。要将 Sw/A 内容添加到 Document/literal Wrapped 消息，除了单个操作包装外，还需要指定其他消息部分。

MTOM 是否解决了这些问题？接下来我们就要对此进行讨论。

回页首

JAX-WS 和 MTOM 示例

在清单 3 中，我们将清单 1 的 Sw/A WSDL 修改为了等效的 MTOM WSDL。

清单 3. JAX-WS MTOM WSDL

```
<definitions
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://attachment.tip/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
      targetNamespace="http://attachment.tip/">
<types/>
<message name="sendImage">
  <part name="image" type="xsd:base64Binary"/>
</message>
<message name="sendImageResponse"/>
<message name="sendOctet">
  <part name="octet" type="xsd:base64Binary"/>
</message>
<message name="sendOctetResponse"/>
<portType name="AttachmentTip">
  <operation name="sendImage">
    <input message="tns:sendImage"/>
    <output message="tns:sendImageResponse"/>
  </operation>
  <operation name="sendOctet">
    <input message="tns:sendOctet"/>
    <output message="tns:sendOctetResponse"/>
  </operation>
</portType>
<binding name="AttachmentBinding" type="tns:AttachmentTip">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sendImage">
    <soap:operation soapAction=""/>
    <input>
      <soap:body namespace="http://attachment.tip/" use="literal"/>
    </input>
```

```
      <output>
        <soap:body namespace="http://attachment.tip/" use="literal"/>
      </output>
    </operation>
    <operation name="sendOctet">
      <soap:operation soapAction=""/>
      <input>
        <soap:body namespace="http://attachment.tip/" use="literal"/>
      </input>
      <output>
        <soap:body namespace="http://attachment.tip/" use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="AttachmentService">
    <port binding="tns:AttachmentBinding" name="AttachmentTip">
      <soap:address location="http://localhost:9080/MTOMService/services/AttachmentTip"/
    </port>
  </service>
</definitions>
```

清单 3 中的 MTOM WSDL 和清单 1 中的 Sw/A WSDL 的唯一区别在于绑定。MTOM 绑定不包含 MIME 信息。事实上，通过 WSDL 并不能说明您所处理的是附件，WSDL 的绑定看起来就像一个普通绑定。

清单 4 显示了对应的 Java 接口，即从此 WSDL 生成的 JAX-RPC 映射。

清单 4. JAX-WS MTOM Java 接口

```
package tip.attachment;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name = "AttachmentTip", targetNamespace = "http://attachment.tip/"
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface AttachmentTip {



    @WebMethod
    public void sendImage(
        @WebParam(name = "image", partName = "image")
        byte[] image);

    @WebMethod
    public void sendOctet(
        @WebParam(name = "octet", partName = "octet")
        byte[] octet);

}
```

因为此 WSDL 是普通 WSDL，没有关于附件的信息，因此最终得到的 Java 接口也会反映出这一点。对于类

型为 base64Binary（或 hexBinary）的部分，JAX-WS 将其映射为 byte[] 类型的参数。对于 MTOM，会从 WSLD 提取类型的全部 MIME 信息，并需要由客户机或服务器运行时对内容进行恰当的格式化。

回页首

Sw/A 和 MTOM 的比较

JAX-RPC Sw/A 部分指出了 Sw/A 的两个缺点：

MIME 类型信息在 WSDL 绑定中，而不是 WSDL 接口中。

很难创建 Document/literal Wrapped 附件 WSDL。

让我们首先讨论一下 Document/literal Wrapped 样式。

Document/literal Wrapped 样式与附件

出于比较目的，我们将清单 3 中的 MTOM WSDL 保留为 rpc/literal WSDL；但要将此 WSDL 转换为 Document/literal Wrapped WSDL 很容易。下一部分的清单 5 显示了清单 3 中的 WSDL 的 Document/literal Wrapped 等效 WSDL（有关各种 WSDL 样式之间的区别的更多信息，请参见参考资料部分）。

绑定中的 MIME 类型信息

因为 MTOM WSDL 的绑定没有任何 MIME 信息，因此不用分析绑定来确定部分类型。在 WSDL 中有生成 Java 接口所需要的全部信息。不过，正如您通过比较清单 2 和清单 4 可以看到的，我们丢失了一些信息。这两个 WSDL 的接口部分都显示操作中的数据类型为 base64Binary，而 base64Binary 又映射到 byte[]。不过，在 JAX-RPC Sw/A WSDL 中，我们通过绑定了解到部分类型为 MIME 映像和 MIME 八进制流。在 JAX-WS MTOM WSDL 中，此信息丢失了。这可能让人感觉是件坏事，但有一个好处，即接口得到了完全的清理。无论绑定的是什么，接口始终都是一样的。事实上，客户机和服务器代码库的实现者不应考虑参数是否是附件的问题。这仅是 SOAP 消息的一个细节，WSDL 到 Java 映射的编写者已经尽量让程序员不考虑 SOAP 消息的细节。

不过，如果您真的希望知道 MIME 类型（即希望找回所丢失的信息），JAX-WS 提供了一个属性，您可将其注入 XML 元素：expectedContentTypes 中。清单 5 中突出显示的部分就是这个特殊的属性。清单 6 显示

了对应的 Java 接口。如果忽略注释，此 Java 接口就完全与清单 2 中的接口相同。

清单 5. JAX-WS MIME 属性

```
<definitions
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://attachment.tip/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://attachment.tip/">
  <types>
    <xsd:schema
            xmlns:tns="http://attachment.tip/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://attachment.tip/">
      <xsd:element name="sendImage" type="tns:sendImage"/>
      <xsd:complexType name="sendImage">
        <xsd:sequence>
          <xsd:element
                  xmlns:ns1="http://www.w3.org/2005/05/xmlmime"
                ns1:expectedContentTypes="image/*"
                  name="image"
                  type="xsd:base64Binary"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="sendImageResponse" type="tns:sendImageResponse"/>
      <xsd:complexType name="sendImageResponse">
```

```
      <xsd:sequence/>
    </xsd:complexType>
    <xsd:element name="sendOctet" type="tns:sendOctet"/>
    <xsd:complexType name="sendOctet">
      <xsd:sequence>
        <xsd:element
               xmlns:ns1="http://www.w3.org/2005/05/xmlmime"
              ns1:expectedContentTypes="application/octet-stream"
               name="octet"
               type="xsd:base64Binary"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="sendOctetResponse" type="tns:sendOctetResponse"/>
    <xsd:complexType name="sendOctetResponse">
      <xsd:sequence/>
    </xsd:complexType>
  </xsd:schema>
</types>
<message name="sendImage">
  <part name="parameters" element="tns:sendImage"/>
</message>
<message name="sendImageResponse">
  <part name="parameters" element="tns:sendImageResponse"/>
</message>
<message name="sendOctet">
  <part name="parameters" element="tns:sendOctet"/>
</message>
```

```
<message name="sendOctetResponse">
  <part name="parameters" element="tns:sendOctetResponse"/>
</message>
<portType name="AttachmentTip">
  <operation name="sendImage">
    <input message="tns:sendImage"/>
    <output message="tns:sendImageResponse"/>
  </operation>
  <operation name="sendOctet">
    <input message="tns:sendOctet"/>
    <output message="tns:sendOctetResponse"/>
  </operation>
</portType>
<binding name="AttachmentBinding" type="tns:AttachmentTip">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sendImage">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="sendOctet">
    <soap:operation soapAction=""/>
```

```
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="AttachmentService">
    <port binding="tns:AttachmentBinding" name="AttachmentTip">
      <soap:address location="http://localhost:9080/MTOMService/services/AttachmentTip"/
    </port>
  </service>
</definitions>
```

请注意，这个新属性完全包含在 WSDL 接口中。该属性不在绑定中，因此所有类型信息都在您所期望的位置。

清单 6. 从 MIME 属性映射得到的 JAX-WS Java 接口

```
package tip.attachment;

import java.awt.Image;
import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebParam;
```

```
import javax.jws.WebService;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(name = "AttachmentTip", targetNamespace = "http://attachment.tip/"
public interface AttachmentTip {

    @WebMethod
    @RequestWrapper(localName = "sendImage",
            targetNamespace = "http://attachment.tip/",
            className = "tip.attachment.SendImage")
    @ResponseWrapper(localName = "sendImageResponse",
            targetNamespace = "http://attachment.tip/",
            className = "tip.attachment.SendImageResponse")
    public void sendImage(
        @WebParam(name = "image", targetNamespace = "")
        Image image);

    @WebMethod
    @RequestWrapper(localName = "sendOctet",
            targetNamespace = "http://attachment.tip/",
            className = "tip.attachment.SendOctet")
    @ResponseWrapper(localName = "sendOctetResponse",
            targetNamespace = "http://attachment.tip/",
            className = "tip.attachment.SendOctetResponse")
    public void sendOctet(
        @WebParam(name = "octet", targetNamespace = "")
```

```
        DataHandler octet);
}
```

正如前面所提到的，JAX-WS 依赖于 JAXB 处理其大部分 Web 服务内容的数据绑定。expectedContentTypes 元素的映射在 JAXB 2.0 规范的附录 H 中定义（请参见参考资料）。JAXB 2.0 映射与 JAX-RPC 映射类似。表 1 对这些映射进行了比较。

表 1. MIME 类型到 Java 类型的映射

MIME 类型	JAX-RPC 映射	JAX-WS/JAXB 映射
image/gif	java.awt.Image	java.awt.Image
image/jpg	java.awt.Image	java.awt.Image
text/plain	java.lang.String	javax.xml.transform.Source
text/xml	javax.xml.transform.Source	javax.xml.transform.Source
application/xml	javax.xml.transform.Source	javax.xml.transform.Source
multipart/*	javax.mail.internet.MimeMultipart	javax.activation.DataHandler
所有其他类型	javax.activation.DataHandler	javax.activation.DataHandler

启用/禁用附件支持

MTOM 的另外一个好处是，您能够启用或禁用它。对于 Sw/A，如果某一方不支持发送 Sw/A 附件，就不能遵守 WSDL 定义的契约。另一方面，如果使用清单 3 或清单 5 中给出的类似的 MTOM WSDL，客户机就可以选择将数据作为 MTOM 附件发送或在 SOAP 消息中以内联方式发送。无论客户机选择哪个选项，都仍然可以与 Web 服务交互。MTOM 只是发送内容的一个优化选项，而不是像 Sw/A 一样犟制要求使用。

回页首

Sw/A 和 JAX-WS

JAX-WS 仍然支持 Sw/A 模型。缺省情况下，JAX-WS 将 Sw/A 附件映射到 Java 接口上的 byte[]，就像

清单 4 中的示例一样。要获得在 JAX-RPC 中使用的映射，可以使用 enableMIMEContent WSDL 绑定定义（有关更多信息，请参见 JAX-WS 规范的 8.7.5 节）。清单 7 显示了与清单 2 中的接口的 JAX-RPC 版本等效的 JAX-WS 版本。

清单 7. JAX-WS Java 接口，Sw/A 附件从 mime:content 映射

```
package tip.attachment;

import java.awt.Image;
import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name = "AttachmentTip", targetNamespace = "http://attachment.tip/"
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface AttachmentTip {



    @WebMethod
    public void sendImage(
        @WebParam(name = "image", partName = "image")
        Image image);

    @WebMethod
    public void sendOctet(
```

```
        @WebParam(name = "octet", partName = "octet")
        DataHandler octet);

}
```

回页首

总结

JAX-RPC 支持 Sw/A 模型。JAX-WS 也支持 Sw/A，但另外还支持新的 MTOM 模型。从多方面而言，MTOM 都比 Sw/A 提高了不少：

创建 Java 接口所必需的全部信息现在都在 WSDL 接口中。

MTOM 可以在 Document/literal Wrapped WSDL 中使用。

MTOM 允许对附件进行优化，但并不像 Sw/A 一样对附件有羁制要求。

参考资料

学习

您可以参阅本文在 developerWorks 全球站点上的 英文原文 。

本系列文章的其它文章：

第 1 部分

第 2 部分

第 3 部分

第 4 部分

获得关于以下方面的规范、API 类和 Javadoc：

JAX-RPC 1.1

JAX-WS 2.0

JAXB 2.0

了解 Java 5 的所有功能。

阅读 WSDL 1.1 规范。

查看 SOAP V1.2 primer。

了解 WSDL 的不同样式，特别是 Document/literal Wrapped 模式。

了解关于 Sw/A 和 JAX-RPC 的更多信息。

您可以通过使用 IBM® WebSphere® Application Server Version 6.1 Feature Pack for Web Services 开始评估 JAX-WS 功能。

IBM developerWorks SOA and Web services 专区提供了大量的文章，以及关于如何开发 Web 服务应用程序的初级、中级和高级教程。

使用 IBM SOA Sandbox 进行试验！通过 IBM SOA 进行实际的亲手实践来提高您的 SOA 技能。

IBM SOA 网站提供 SOA 的概述，并介绍 IBM 是如何帮助您实现 SOA 的。

了解关于 developerWorks 技术事件和网络广播的最新消息。请特别关注以下 SOA 和 Web 服务技术讲

座：

Building SOA solutions and managing the service lifecycle

SCA/SDO:To drive the next generation of SOA

访问 Safari 书店，浏览有关这些技术主题以及其他方面的书籍。

查看快速的 Web 服务按需演示。

获取SOA and Webservices 专区的 RSS。（了解关于 RSS 的更多信息。）

获得产品和技术

使用 IBM 试用软件开发您的下一个项目，可下载或索取 DVD 光盘。

讨论

参与论坛讨论。

参与 developerWorks Blog，从而加入到 developerWorks 社区中来，其中包括以下与 SOA 和 Web 服务相关的 Blogs：

Sandy Carter 的 Service Oriented Architecture -- Off the Record

Ali Arsanjani 的 Best Practices in Service-Oriented Architecture

Bobby Woolf 的 WebSphere SOA and J2EE in Practice

Eoin Lane 博士的 Building SOA applications with patterns

Kerrie Holley 的 Client Insights, Concerns and Perspectives on SOA

Simon Johnston 的 Service-Oriented Architecture and Business-Level Tooling

Sanjay Bose 的 SOA, ESB and Beyond

作者简介

Russell Butek 是 IBM 的一名 SOA 和 Web 服务顾问。他曾是 IBM WebSphere Web 服务引擎的开发人员之一。他也是 JAX-RPC Java Specification Request (JSR) 专家组的成员。他参与了 Apache 的 AXIS SOAP 引擎的实现，并推动 AXIS 1.0 遵守 JAX-RPC。

Nick Gallardo 担任 IBM WebSphere 平台软件工程师，主要负责 Web 服务支持的各个方面。在此之前，他从事 IBM WebSphere 和 Tivoli 平台中其他方面的工作。Nick 于 2001 加入 IBM，此前他曾在德克萨斯州奥斯汀市两家不同的初创技术型公司从事开发工作。

上一篇：基于REST 的Web 服务：基础

下一篇：面向资源与面向活动的Web 服务

发表评论

请自觉遵守互联网相关的政策法规，严禁发布色情、暴力、反动的言论。

评价：◉ 中立 ○ 好评 ○ 差评

表情：

◉ ○ ○ ○
○ ○ ○ ○
○ ○ ○

验证码： FRAN ☐ 匿名？ 发表评论

最新评论 进入详细评论页>>

JAX-WS Hello World Example – RPC Style

By mkyong | November 16, 2010 | Updated : August 29, 2012 | Viewed : times

JAX-WS is bundled with JDK 1.6, which makes Java web service development easier to develop. This tutorial shows you how to do the following tasks:

1. Create a SOAP-based RPC style web service endpoint by using JAX-WS.
2. Create a Java web service client manually.
3. Create a Java web service client via **wsimport** tool.
4. Create a Ruby web service client.

You will be surprise of how simple it is to develop a RPC style web service in JAX-WS.

Note
In general words, "*web service endpoint*" is a service which published outside for user to access; where "*web service client*" is the party who access the published service.

JAX-WS Web Service End Point

The following steps showing how to use JAX-WS to create a RPC style web service endpoint.

1. Create a Web Service Endpoint Interface

File : HelloWorld.java

```java
package com.mkyong.ws;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
```

```
//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.RPC)
public interface HelloWorld{

        @WebMethod String getHelloWorldAsString(String name);

}
```


2. Create a Web Service Endpoint Implementation

File : HelloWorldImpl.java

Java

```
package com.mkyong.ws;

import javax.jws.WebService;

//Service Implementation
@WebService(endpointInterface = "com.mkyong.ws.HelloWorld")
public class HelloWorldImpl implements HelloWorld{
```

```
        @Override
        public String getHelloWorldAsString(String name) {
                return "Hello World JAX-WS " + name;
        }

}
```

3. Create a Endpoint Publisher

File : HelloWorldPublisher.java

Java

```
package com.mkyong.endpoint;

import javax.xml.ws.Endpoint;
import com.mkyong.ws.HelloWorldImpl;

//Endpoint publisher
public class HelloWorldPublisher{

        public static void main(String[] args) {
           Endpoint.publish("http://localhost:9999/ws/hello", new HelloWorldImpl());
    }

}
```

Run the endpoint publisher, and your “**hello world web service**” is deployed in URL

“**http://localhost:9999/ws/hello**“.

4. Test It

You can test the deployed web service by accessing the generated WSDL (Web Service Definition Language) document via this URL “**http://localhost:9999/ws/hello?wsdl**” .

Web Service Clients

Ok, web service is deployed properly, now let’s see how to create web service client to access to the published service.

1. Java Web Service Client

Without tool, you can create a Java web service client like this :

Java

```java
package com.mkyong.client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import com.mkyong.ws.HelloWorld;

public class HelloWorldClient{
```

```
        public static void main(String[] args) throws Exception {

        URL url = new URL("http://localhost:9999/ws/hello?wsdl");

        //1st argument service URI, refer to wsdl document above
        //2nd argument is service name, refer to wsdl document above
        QName qname = new QName("http://ws.mkyong.com/", "HelloWorldImplService");

        Service service = Service.create(url, qname);

        HelloWorld hello = service.getPort(HelloWorld.class);

        System.out.println(hello.getHelloWorldAsString("mkyong"));

    }

}
```

Output

```
Hello World JAX-WS mkyong
```
Bash

2. Java Web Service Client via wsimport tool

Alternative, you can use "**wsimport**" tool to parse the published wsdl file, and generate necessary client files (stub) to access the published web service.

Where is wsimport?
This **wsimport** tool is bundle with the JDK, you can find it at "*JDK_PATH/bin*" folder.

Issue "**wsimport**" command.

```Bash
wsimport -keep http://localhost:9999/ws/hello?wsdl
```

It will generate necessary client files, which is depends on the provided wsdl file. In this case, it will generate one interface and one service implementation file.

File : HelloWorld.java

```Java
package com.mkyong.ws;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.1.1 in JDK 6
 * Generated source version: 2.1
 *
```

```
 */
@WebService(name = "HelloWorld", targetNamespace = "http://ws.mkyong.com/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface HelloWorld {

    /**
     *
     * @param arg0
     * @return
     *     returns java.lang.String
     */
    @WebMethod
    @WebResult(partName = "return")
    public String getHelloWorldAsString(
        @WebParam(name = "arg0", partName = "arg0")
        String arg0);

}
```

File : HelloWorldImplService.java

Java

```
package com.mkyong.ws;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
```

```
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;

/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.1.1 in JDK 6
 * Generated source version: 2.1
 *
 */
@WebServiceClient(name = "HelloWorldImplService",
        targetNamespace = "http://ws.mkyong.com/",
        wsdlLocation = "http://localhost:9999/ws/hello?wsdl")
public class HelloWorldImplService
    extends Service
{

    private final static URL HELLOWORLDIMPLSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:9999/ws/hello?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        HELLOWORLDIMPLSERVICE_WSDL_LOCATION = url;
    }

    public HelloWorldImplService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
```

```
    }

    public HelloWorldImplService() {
        super(HELLOWORLDIMPLSERVICE_WSDL_LOCATION,
                        new QName("http://ws.mkyong.com/", "HelloWorldImplService"));
    }

    /**
     *
     * @return
     *     returns HelloWorld
     */
    @WebEndpoint(name = "HelloWorldImplPort")
    public HelloWorld getHelloWorldImplPort() {
        return (HelloWorld)super.getPort(
                        new QName("http://ws.mkyong.com/", "HelloWorldImplPort"),
                        HelloWorld.class);
    }

    /**
     *
     * @param features
     *  A list of {@link javax.xml.ws.WebServiceFeature} to configure on the proxy.
     *  Supported features not in the <code>features</code> parameter will have their de
     * @return
     *  returns HelloWorld
     */
    @WebEndpoint(name = "HelloWorldImplPort")
    public HelloWorld getHelloWorldImplPort(WebServiceFeature... features) {
        return (HelloWorld)super.getPort(
```

```
                        new QName("http://ws.mkyong.com/", "HelloWorldImplPort"),
                        HelloWorld.class,
                        features);
    }

}
```

Now, create a Java web service client which depends on the above generated files.

Java

```java
package com.mkyong.client;

import com.mkyong.ws.HelloWorld;
import com.mkyong.ws.HelloWorldImplService;

public class HelloWorldClient{

        public static void main(String[] args) {

                HelloWorldImplService helloService = new HelloWorldImplService();
                HelloWorld hello = helloService.getHelloWorldImplPort();

                System.out.println(hello.getHelloWorldAsString("mkyong"));

    }

}
```

Here's the output

```
Hello World JAX-WS mkyong
```

Bash

3. Ruby Web Service Client

Often time, web service development is mixed use with other programming language. So, here's a Ruby web service client example, which is used to access the published JAX-WS service.

```
# package for SOAP-based services
require 'soap/wsdlDriver'

wsdl_url = 'http://localhost:9999/ws/hello?wsdl'

service = SOAP::WSDLDriverFactory.new(wsdl_url).create_rpc_driver

# Invoke service operations.
data1 = service.getHelloWorldAsString('mkyong')

# Output results.
puts "getHelloWorldAsString : #{data1}"
```

Output

```
getHelloWorldAsString : Hello World JAX-WS mkyong
```

Bash

Tracing SOAP Traffic

From top to bottom, showing how SOAP envelope flows between client and server. See #1 web service client again :

```java
URL url = new URL("http://localhost:9999/ws/hello?wsdl");
QName qname = new QName("http://ws.mkyong.com/", "HelloWorldImplService");
Service service = Service.create(url, qname);

HelloWorld hello = service.getPort(HelloWorld.class);

System.out.println(hello.getHelloWorldAsString("mkyong"));
```

Note
To monitor SOAP traffic is very easy, see this guide – "How to trace SOAP message in Eclipse IDE".

1. Request a WSDL file

First, client send a wsdl request to service endpoint, see HTTP traffic below :

Client send request :

```bash
```

```
GET /ws/hello?wsdl HTTP/1.1
User-Agent: Java/1.6.0_13
Host: localhost:9999
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

Server send response :

Markup

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml;charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>

<definitions
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://ws.mkyong.com/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="http://ws.mkyong.com/"
        name="HelloWorldImplService">

        <types></types>

<message name="getHelloWorldAsString">
        <part name="arg0" type="xsd:string"></part>
</message>
<message name="getHelloWorldAsStringResponse">
```

```
        <part name="return" type="xsd:string"></part>
</message>

<portType name="HelloWorld">
        <operation name="getHelloWorldAsString" parameterOrder="arg0">
                <input message="tns:getHelloWorldAsString"></input>
                <output message="tns:getHelloWorldAsStringResponse"></output>
        </operation>
</portType>

<binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">

        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"></soa
        <operation name="getHelloWorldAsString">
                <soap:operation soapAction=""></soap:operation>
                <input>
                        <soap:body use="literal" namespace="http://ws.mkyong.com/"></soa
                </input>
                <output>
                        <soap:body use="literal" namespace="http://ws.mkyong.com/"></soa
                </output>
        </operation>

</binding>

<service name="HelloWorldImplService">
<port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
<soap:address location="http://localhost:9999/ws/hello"></soap:address>
</port>
</service>
```

```
</definitions>
```

2. hello.getHelloWorldAsString()

A second call, client put method invoke request in SOAP envelope and send it to service endpoint. At the service endpoint, call the requested method and put the result in a SOAP envelope and send it back to client.

Client send request :

Markup

```
POST /ws/hello HTTP/1.1
SOAPAction: ""
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.
Content-Type: text/xml; charset=utf-8
User-Agent: Java/1.6.0_13
Host: localhost:9999
Connection: keep-alive
Content-Length: 224

<?xml version="1.0" ?>
        <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
                <S:Body>
                        <ns2:getHelloWorldAsString xmlns:ns2="http://ws.mkyong.com/">
                                <arg0>mkyong</arg0>
                        </ns2:getHelloWorldAsString>
                </S:Body>
        </S:Envelope>
```

Server send response :

Markup

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8

<?xml version="1.0" ?>
        <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
                <S:Body>
                        <ns2:getHelloWorldAsStringResponse xmlns:ns2="http://ws.mkyong.c
                                <return>Hello World JAX-WS mkyong</return>
                        </ns2:getHelloWorldAsStringResponse>
                </S:Body>
        </S:Envelope>
```

Done, any comments are appreciated.

Download Source Code

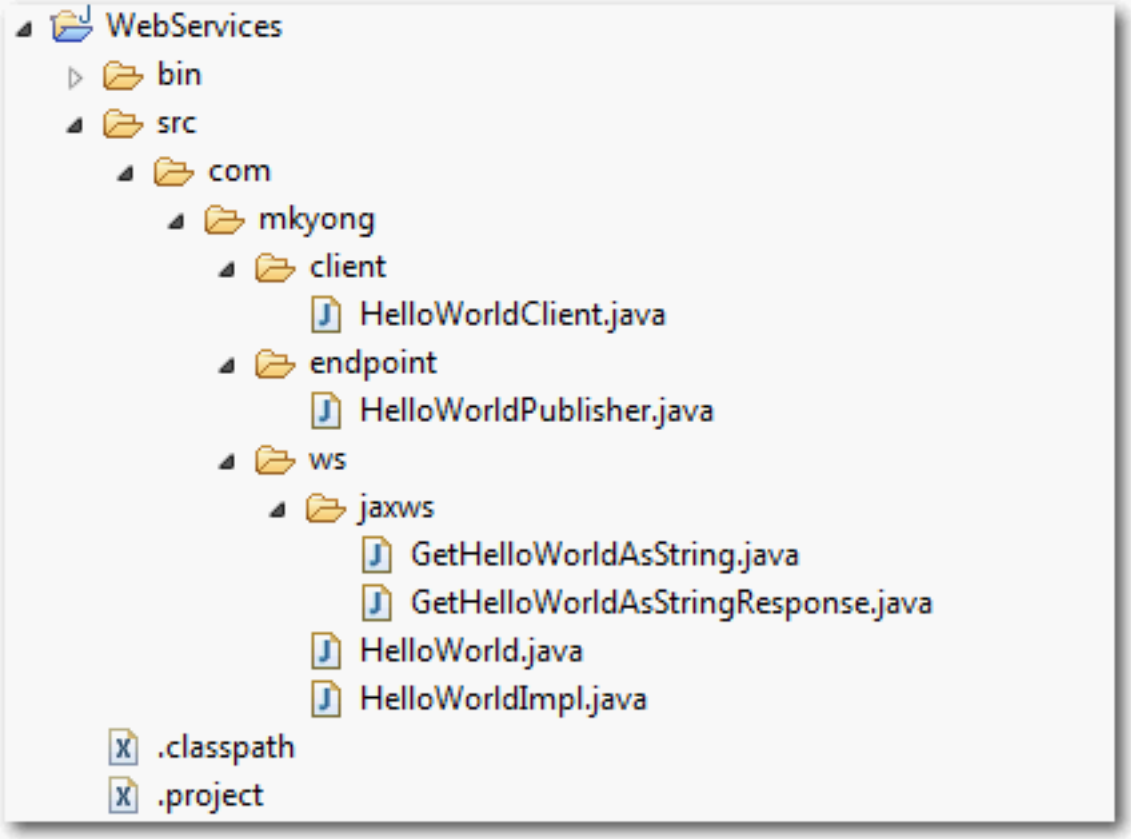
Download It – JAX-WS-HelloWorld-RPC-Example.zip (14KB)

JAX-WS Hello World Example – Document Style

By mkyong | November 20, 2010 | Updated : August 29, 2012 | Viewed : times

In this tutorial, we show you how to use JAX-WS to create a SOAP-based web service (document style) endpoint. Compare with RPC style, it need some extra efforts to get it works.

Directory structure of this example



JAX-WS Web Service End Point

Here are the steps to create a document style web service in JAX-WS.

1. Create a Web Service Endpoint Interface

Actually, annotated with `@SOAPBinding` is optional, because the default style is document.

File : HelloWorld.java

Java

```
package com.mkyong.ws;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.DOCUMENT, use=Use.LITERAL) //optional
public interface HelloWorld{

        @WebMethod String getHelloWorldAsString(String name);

}
```

Note
In JAX-WS development, convert from "*RPC style*" to "*Document style*" is very easy, just change the `@SOAPBinding` style option.

2. Create a Web Service Endpoint Implementation

File : HelloWorldImpl.java

Java

```java
package com.mkyong.ws;

import javax.jws.WebService;

//Service Implementation
@WebService(endpointInterface = "com.mkyong.ws.HelloWorld")
public class HelloWorldImpl implements HelloWorld{

        @Override
        public String getHelloWorldAsString(String name) {
                return "Hello World JAX-WS " + name;
        }

}
```

3. Create a Endpoint Publisher.

File : HelloWorldPublisher.java

```java
package com.mkyong.endpoint;

import javax.xml.ws.Endpoint;
import com.mkyong.ws.HelloWorldImpl;

//Endpoint publisher
public class HelloWorldPublisher{

        public static void main(String[] args) {
           Endpoint.publish("http://localhost:9999/ws/hello", new HelloWorldImpl());
    }

}
```

Wait, when you run the end point publisher, you will hits following error message :

```java
Wrapper class com.mkyong.ws.jaxws.GetHelloWorldAsString is not found.
    Have you run APT to generate them?
```

See this article. You need to use "**wsgen**" tool to generate necessary JAX-WS portable artifacts. Let move to next step.

4. wsgen command

Document style requires extra classes to run, you can use "**wsgen**" to generate all necessary Java artifacts (mapping classes, wsdl or xsd schema). The "**wsgen**" command is required to read a service endpoint implementation class :

```bash
wsgen -keep -cp . com.mkyong.ws.HelloWorldImpl
```

It will generate two classes, copy it to your "**package.jaxws**" folder.

File : GetHelloWorldAsString.java

```java
package com.mkyong.ws.jaxws;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "getHelloWorldAsString", namespace = "http://ws.mkyong.com/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getHelloWorldAsString", namespace = "http://ws.mkyong.com/")
public class GetHelloWorldAsString {

    @XmlElement(name = "arg0", namespace = "")
    private String arg0;
```

```
    /**
     *
     * @return
     *     returns String
     */
    public String getArg0() {
        return this.arg0;
    }

    /**
     *
     * @param arg0
     *     the value for the arg0 property
     */
    public void setArg0(String arg0) {
        this.arg0 = arg0;
    }

}
```

File : GetHelloWorldAsStringResponse.java

Java

```
package com.mkyong.ws.jaxws;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
```

```java
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "getHelloWorldAsStringResponse", namespace = "http://ws.mkyong.co
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getHelloWorldAsStringResponse", namespace = "http://ws.mkyong.com/")
public class GetHelloWorldAsStringResponse {

    @XmlElement(name = "return", namespace = "")
    private String _return;

    /**
     *
     * @return
     *     returns String
     */
    public String getReturn() {
        return this._return;
    }

    /**
     *
     * @param _return
     *     the value for the _return property
     */
    public void setReturn(String _return) {
        this._return = _return;
    }

}
```

Note
The “wsgen” tool is available in the “JDK_Path\bin\” folder. For detail, please read this JAX-WS : wsgen tool example article.

5. Done

Done, publish it and test it via URL : *http://localhost:9999/ws/hello?wsdl*.

Web Service Client

Create a web service client to access your published service.

File : HelloWorldClient.java

```java
package com.mkyong.client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import com.mkyong.ws.HelloWorld;

public class HelloWorldClient{

        public static void main(String[] args) throws Exception {
```

```
        URL url = new URL("http://localhost:9999/ws/hello?wsdl");
        QName qname = new QName("http://ws.mkyong.com/", "HelloWorldImplService");

        Service service = Service.create(url, qname);

        HelloWorld hello = service.getPort(HelloWorld.class);

         System.out.println(hello.getHelloWorldAsString("mkyong"));

    }

}
```

Output

```
Hello World JAX-WS mkyong
```
Bash

Tracing SOAP Traffic

From top to bottom, showing how SOAP envelope flows between client and server in this document style web service.

1. Request a WSDL file

First, client send a wsdl request to service endpoint :

Client send request :

```Bash
GET /ws/hello?wsdl HTTP/1.1
User-Agent: Java/1.6.0_13
Host: localhost:9999
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

Server send response :

```Markup
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml;charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>
        <!-- Published by JAX-WS RI at http://jax-ws.dev.java.net.
                RI's version is JAX-WS RI 2.1.1 in JDK 6. -->
        <!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net.
                RI's version is JAX-WS RI 2.1.1 in JDK 6. -->
<definitions
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://ws.mkyong.com/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="http://ws.mkyong.com/"
        name="HelloWorldImplService">
```

```
<types>
<xsd:schema>
        <xsd:import namespace="http://ws.mkyong.com/"
                schemaLocation="http://localhost:9999/ws/hello?xsd=1"></xsd:import>
</xsd:schema>
</types>

<message name="getHelloWorldAsString">
        <part name="parameters" element="tns:getHelloWorldAsString"></part>
</message>
<message name="getHelloWorldAsStringResponse">
        <part name="parameters" element="tns:getHelloWorldAsStringResponse"></part>
</message>

<portType name="HelloWorld">
        <operation name="getHelloWorldAsString">
                <input message="tns:getHelloWorldAsString"></input>
                <output message="tns:getHelloWorldAsStringResponse"></output>
        </operation>
</portType>

<binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">

        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document">
        </soap:binding>
        <operation name="getHelloWorldAsString">
                <soap:operation soapAction=""></soap:operation>
                <input>
                        <soap:body use="literal"></soap:body>
                </input>
```

```
                <output>
                        <soap:body use="literal"></soap:body>
                </output>
        </operation>

</binding>

<service name="HelloWorldImplService">

<port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">

<soap:address location="http://localhost:9999/ws/hello"></soap:address>

</port>
</service>
</definitions>
```

2. getHelloWorldAsString(String name)

A second call, client put method invoke request in SOAP envelope and send it to service endpoint. At the service endpoint, call the requested method and put the result in a SOAP envelope and send it back to client.

Client send request :

Markup

```
POST /ws/hello HTTP/1.1
SOAPAction: ""
```

```
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.
Content-Type: text/xml; charset=utf-8
User-Agent: Java/1.6.0_13
Host: localhost:9999
Connection: keep-alive
Content-Length: 224

<?xml version="1.0" ?>
        <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
                <S:Body>
                        <ns2:getHelloWorldAsString xmlns:ns2="http://ws.mkyong.com/">
                                <arg0>mkyong</arg0>
                        </ns2:getHelloWorldAsString>
                </S:Body>
        </S:Envelope>
```

Server send response :

Markup

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8

<?xml version="1.0" ?>
        <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
                <S:Body>
                        <ns2:getHelloWorldAsStringResponse xmlns:ns2="http://ws.mkyong.c
                                <return>Hello World JAX-WS mkyong</return>
                        </ns2:getHelloWorldAsStringResponse>
```

```
            </S:Body>
        </S:Envelope>
```

Download Source Code

Download It – JAX-WS-HelloWorld-Document-Example.zip (10KB)

Tags : hello world | jax-ws | web services

Share this article on

Twitter | Facebook | Google+