Web Service

# 标准通用语言

标准通用标记语言（Standard Generalized Markup language ），是一种定义电子文档结构和描述其内容的国际标准语言。

## html

HTML：HyperText Markup Language的简称，即超文本标记语言，标准通用标记语言下的一个子集，被设计用来显示数据。

## xml

Xml：是Extensible Markup Language的简称，即可扩展标记语言，标准通用标记语言下的一个子集 ，被设计用来传输和存储数据。

## soap

SOAP：是Simple Object Access Protocol简称，即简单对象访问协议，标准通用标记语言下的一个子集 ，被设计用来描述传递信息的格式；SOAP是一种轻量的、简单的、基于XML的协议。

其指导理念是“唯一一个没有发明任何新技术的技术”，是一种用于访问 Web 服务的协议。SOAP 基于XML 和 HTTP ，其通过XML 来实现消息描述，然后再通过 传输协议（这里即HTTP协议） 实现消息传输。

## RPC

rpc：Remote Procedure Call Protocol的简写，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。

# WebService

## web Service 是什么

Web service是一个平台独立的，低耦合的，自包含的、基于可编程的web的应用程序，可使用开放的XML标准来描述、发布、发现、协调和配置这些应用程序，用于开发分**布式的互操作的应用程序**。

### 跨平台和跨应用程序

有没有一种办法可以实现跨应用程序进行通信和跨平台进行通信呢？

换句话说，就是有什么办法可以实现我的应用程序 A 可以和应用程序 B 进行通信呢(跨应用程序，应用程序A和应用程序B不同)？

或者说是，我用 Java 写的应用程序和用 . Net 开发的应用程序之间进行通信呢（跨平台：java和.net 是不同的平台）？

拿腾讯 QQ 来说吧，我估计每一个人都用过腾讯 QQ 上面的天气预报吧 ！上面的这个天气预报功能是如何实现的呢？

有一种办法，那就是腾讯公司放个卫星上天，并且在公司中成立一个气象部门，天天关注于天气，然后每时每刻更新腾讯 QQ 上的这个天气预报信息，确实，这种办法确实行得通，不过，要真这样做的话，估计马化腾早就完蛋了。

### web Service 究竟是什么？

如果简单的说，Web Service就是一组函数库，不过它和我们平时概念中的函数库有些不同。

我们平时所说的函数库要么是自己写的（在自己的应用程序中写的一组函数库），要么是底层已存在的（如Win32 这种os API ），这两种函数库一个共同点，那就是它们都位于客户端本地，比如，您调用 的Win32 API；

我们之所以说 Web Service也是一组函数库，是因为它和上面提到的本地函数库本质完全一样，当然是有个区别的，就是它所处的位置：它不是位于本地，而是位于远程机器上（当然也可以是本地机器）。

因此 Web service，就是网络上不知道那个地方的一些函数，我们可通过网络来使用这些函数。好了，按照官方的正常做法，肯定要给这些“函数”起一个高大上的名字，又因为他们是位于web上，所以就叫“web service”，更官方的说：Web service是一种部署在 Web 上的对象或者是应用程序组件。

## web Service体系结构

在Web service的体系结构中，涉及到三个角色，一个是 Web 服务提供者，一个是 Web 服务中介者，还有一个就是 Web 服务请求者；同时还涉及到三类动作，即发布，查找，绑定。

### 三个角色

#### Web service提供者

可以发布 Web service，并且对使用自身服务的请求进行响应，Web 服务的拥有者，它会等待其他的服务或者是应用程序访问自己。

#### Web 服务请求者

Web 服务功能的使用者，它通过服务注册中心也就是 Web 服务中介者查找到所需要的服务，再利用 SOAP 消息向 Web 服务提供者发送请求以获得服务。

#### Web 服务中介者

也称为服务代理，用来注册已经发布的 Web服务提供者，并对其进行分类，同时提供搜索服务，简单来说的话，Web 服务中介者的作用就是把一个 Web 服务请求者和合适的 Web 服务提供者联系在一起，充当一个管理者的角色，一般是通过 UDDI来实现。

### 三个动作

#### 发布

通过发布操作，可以使 Web服务提供者向 Web 服务中介者注册自己的功能以及访问的接口。

#### 发现（查找）

使得 Web 服务请求者可以通过 Web 服务中介者来查找到特点的种类的 Web 服务。

#### 绑定

这里就是实现让服务请求者能够使用服务提供者提供的服务。

## web service 协议栈

|  |
| --- |
|  |

## WebServices三种基本元素

### wsdl

WSDL 即Web Services Description Language也就是 Web 服务描述语言。是基于 XML的用于描述 Web 服务以及如何访问 Web 服务的语言。为用户提供详细的接口说明书。

服务提供者通过服务描述将所有用于访问 Web服务的规范传送给服务请求者，要实现 Web服务体系结构的松散耦合，服务描述是一个关键，不管是请求者还是服务提供者，通过服务描述便可以不必了解对方的底层平台，编程语言等，服务描述与底层的 SOAP 基础结构相结合，足以封装服务请求者的应用程序和服务提供者的 Web服务之间的这个细节。

WSDL 描述了 Web服务的三个基本属性：

（1）提供了那些服务（函数）

（2）如何访问服务（函数）

（3）服务位于何处（通过URL）

### soap

简单对象访问协议是交换数据的一种协议规范，是一种轻量的、简单的、基于XML（标准通用标记语言下的一个子集）的协议，它被设计成在WEB上交换结构化的和固化的信息。

是一种用于访问 Web 服务的协议。因为 SOAP 基于XML 和传输协议（这里即Http协议） ，其通过XML 来实现消息描述，然后再通过传输协议（这里即Http协议） 实现消息传输。

SOAP 是用于在应用程序之间进行通信的一种通信协议。因为是基于 XML 和传输协议（这里即Http协议）的，所以独立于语言，独立于平台，并且因为 XML 的扩展性很好，所以基于 XML 的 SOAP 自然扩展性也不差。通过 SOAP 可以非常方便的解决互联网中消息互联互通的需求，其和其他的 Web 服务协议构建起 SOA 应用的技术基础。SOAP 协议的一个重要特点是它独立于底层传输机制，Web 服务应用程序可以根据需要选择自己的数据传输协议，可以在发送消息时来确定相应传输机制。

对于HTTP传输协议：由于 HTTP 协议本身的一些特点和局限性，使得当 SOAP 使用HTTP 绑定的 Web 服务可能无法满足某些企业应用的需求。因为，HTTP 不是一个可靠传输协议，所以有可能在传输过程中出现问题，然后 HTTP 协议是基于Request/Response 模型，也就是说客户端需要在等待响应消息接收完成后才能继续执行，而此时如果响应时间过长呢？基于上面的这些需求，便需要选择合适的传输协议了。

### Uddi

UDDI 即 Universal Description，Discovery and Integration，也就是通用的描述，发现以及整合。

WSDL 呢，用来描述了访问特定的 Web 服务的一些相关的信息，可以在互联网上，或者是在企业的不同部门之间，如何来发现我们所需要的 Web 服务呢？

而 Web 服务提供商又如何将自己开发的 Web 服务公布到因特网上，这就需要使用到 UDDI 了，UDDI的话，是一个跨产业，跨平台的开放性架构，可以帮助 Web 服务提供商在互联网上发布 Web 服务的信息。UDDI 是一种目录服务，企业可以通过 UDDI 来注册和搜索 Web 服务。简单来时候话，UDDI 就是一个目录，只不过在这个目录中存放的是一些关于 Web 服务的信息而已。并且 UDDI 通过SOAP 进行通讯，构建于 . Net 之上。

UDDI最早由IBM、ARIA、MICROSOFT建立，现在参与的公司已愈300家。UDDI提供了一组基于标准的规范用于描述和发现服务，还提供类一组internet的实现。UDDI能够解决企业间遇到的大量的问题----互为对方提供服务。

我们平时开发出的web service 必须发布到UDDI吗？没必要，除非你的服务非常优秀，优秀到能给世界N百强企业提供服务，你觉得可能吗？

### 补充：wsdl与soap

wsdl和soap虽然是web service的两大基本要素，但是两者并没有必然的联系，都可以独立使用，他们交叉点在于wsdl在绑定具体的服务时可能会选择使用soap协议。

wsdl提供了一个统一的接口，目前已经成为一个国际上公认的标准，通过wsdl提供的接口可以使用不同类型资源的服务（如java、c#、C、C++等），因为wsdl是基于xml，与语言平台无关的。另外wsdl提供了binding和service元素，用以绑定接口到具体的服务，实现了接口与实现的分离。

wsdl与soap的关系就在于：wsdl绑定服务的时候可以设定使用的协议，协议可以是soap、http、smtp、ftp等任何一种传输协议，除此以外wsdl还可以绑定jms、ejb及local java等等（不过都是需要对binding和service元素做扩展的，，而且需要扩展服务器的功能以支持这种扩展）。

soap是一个用来封装服务请求方的请求和服务提供方的回复消息的协议。 至于wsdl是用来描述你公开的webservice的服务接口信息，客户方或者服务请求方要从这个服务接口的描述里面确定他要调用的方法以及传递参数的形式。

你可以通过2.4.4.1对soap与wsdl进行理解。

#### wsdl绑定机制扩展

|  |
| --- |
| WSDL设计继承了以XML为基础的当代Web技术标准的开放设计理念。它允许通过扩展使用其它的类型定义语言(不光是XMI Schema)，允许使用多种网络传输协议和消息格式(不仅是在规范中定义的SOAP/HTTP、HTTP-GET/POST及MIME等)。WSDL也应用了当代软件工程中的复用理念，分离了抽象定义层和具体部署层，使得抽象定义层的复用性大大增加。  1、 SOAP绑定  WSDL包括用于SOAP 1.1终端的绑定，此绑定支持下列协议信息规范：  　　需要指示出此绑定是针对SOAP 1.1协议的。  　　为SOAP 终端指定地址的方式。  　　用于SOAPAction HTTP头的URI，而此头是绑定了SOAP 的HTTP头。  　　作为SOAP封装一部分传送的头列表定义。  SOAP绑定使用下列扩展元素对WSDL进行扩展：（注意黑体标注）  ＜definitions .... ＞  ＜binding .... ＞  ＜soap:binding style="rpc|document" transport="uri"＞ …1  ＜operation .... ＞  ＜soap:operation soapAction="uri" style="rpc|document"＞ …2  ＜input＞  ＜soap:body parts="nmtokens" use="litera|encoded"  encodingStyle="uri-list" namespace="uri"＞…3  ＜soap:header message="qname" fault="qname"/＞ …4  ＜/input＞  ＜output＞  ＜soap:body parts="nmtokens" use="litera|encoded"  encodingStyle="uri-list"? namespace="uri"/＞…3  ＜soap:header message="qname" fault="qname"/＞ …4  ＜/output＞  ＜fault＞  ＜soap:fault name="nmtoken" use="litera|encoded"  encodingStyle="uri-list" namespace="uri"＞…5  ＜/fault＞  ＜/operation＞  ＜/binding＞  ＜port .... ＞  ＜soap:address location="uri"/＞ …6  ＜/port＞  ＜/definitions＞    1）soap:binding  　　SOAP绑定元素的目的是指出绑定是针对SOAP协议格式的：Envelope, Header 和 Body。这个元素没有对编码或消息格式进行声明。在使用SOAP绑定时必须指出soap:binding元素。  2）soap:operation  　　soap:operation 元素从整体上为操作提供信息。  soapAction属性为此操作的SOAPAction头指定了值。此URI应当被直接用作SOAPAction头的值，在作出请求时不要试图将一个相对URI变成绝对URI。对于绑定了SOAP的HTTP协议来说，此值是必需的（它没有默认值）。对于其它的SOAP协议绑定，决对不能为其指定值，而soap:operation元素可以被忽略。  style属性指出操作是面向RPC（消息包含参数和返回值）的还是面向文档的（消息包含文档）。此信息可用于选择合适的编程模板。此属性的值将影响SOAP消息体的构建方式，如果没有指定值，它默认为soap:binding元素中指定的值。如果soap:binding元素没有指定样式，它假定为"文档"。  3）soap:body  　　soap:body元素指出了消息部分应如何在SOAP Body元素中表现。  　　消息的各个部分可以是抽象的类型定义，也可以是具体的模式定义。如果是抽象定义，根据编码格式定义的一套规则将对类型进行序列化。使用URI列表可以对每种编码格式进行标识，就象在SOAP规范中的那样。由于某些编码格式如SOAP编码允许某种给定的抽象类型的消息格式发生变化，那么读者应理解所有的格式变化："读者决定正确性"。为了避免对所有变化不必要的支持，可以对消息的定义具体化，然后暗示它的原始编码格式（如果存在）。在这种情况下，消息的创建者必须严格遵循特定的规范："作者决定正确性"。  　　soap:body提供的信息用于决定如何在SOAP消息的Body元素中组合不同的消息部分。soap:body元素可以用在面向RPC的消息也可以用在面向文档的消息中，但是封装操作的格式对Body节如何构建有重要影响：  　　如果操作格式是RPC，那么每个部分将是一个参数，或者是一个返回值，并且它们出现在body中的wrapper元素内。wrapper元素的名字与操作的名字相同，并且它的名称空间是namespace属性的值。每个消息部分（参数）在封装器下出现，它是由与调用中相应参数名称相同的存储器表示的。各部分以与调用参数相同的顺序进行排列。  　　如果操作格式是文档，那么就没有额外的封装器（wrappers），消息的parts直接出现在SOAP Body元素中。  　　在定义Body的内容和参数存储器元素时使用了相同的机制。  4）soap:header  　　soap:header元素允许在SOAP封装的头元素中对将要传送的头进行定义。它模拟了soap:body元素。并不需要使用soap:header费力地列出出现在SOAP封装中的所有头元素。例如，对WSDL的扩展暗示特定的头应加入到实际的有效负荷中，此时就不需要列出所有的头。  5）soap:fault  　　soap:fault元素指出了SOAP Fault Details元素的内容，它模仿了soap:body元素。  6）soap:address  　　SOAP地址绑定用于为端口指定地址。使用SOAP绑定的端口必须指定一个确切的地址，为地址指定的URI配制必须与soap:binding指定的传输相对应。  2、 HTTP GET 与 POST 绑定  为了描述Web浏览器与网站间的交互作用，WSDL包含了HTTP 1.1的 GET 和 POST版本的绑定。这种绑定允许应用程序而不浏览器与网站进行交互。可以指定下面的协议特定信息：  　　使用了HTTP GET 或 POST绑定的标记  　　端口地址  　　每种操作的相对地址（相对于端口定义的基地址）  HTTP GET/POST绑定使用下列元素扩展WSDL：（注意黑体标注）  ＜definitions .... ＞  ＜binding .... ＞  ＜http:binding verb="nmtoken"/＞…1  ＜operation .... ＞  ＜http:operation location="uri"/＞…2  ＜input .... ＞  ＜-- mime elements --＞  ＜/input＞  ＜output .... ＞  ＜-- mime elements --＞  ＜/output＞  ＜/operation＞  ＜/binding＞  ＜port .... ＞  ＜http:address location="uri"/＞…3  ＜/port＞  ＜/definitions＞  1）http:binding  　　http:binding元素指出此绑定使用了HTTP协议。必需的Verb属性的值指出了HTTP的版本，其值通常是GET或POST，但也可以使用其它值。注意，HTTP版本是大小写敏感的。  2）http:operation  　　Location属性为操作指定一个相对URI，此URI与http:address元素指定的URI结合在一起形成了HTTP请求的完整路径。此URI必须是一个相对的URI。  3）http:address  　　location属性为端口指定了基地址。其值是与http:operation绑定元素的位置（location）属性的值结合在一起的。    4）http:urlEncoded  　　UrlEncoded元素指出所有的消息part都按照标准的URI编码规则（名字=值…）编码进HTTP请求URI中。参数名与消息part的名相对应。使用"名字=值"对对part提供的值进行编码。通常使用GET指示URL编码，或使用POST指示一个FORM-POST。对于GET，字符"？"作为必需，将自动追加。  　　＜http:urlEncoded/＞  5）http:urlReplacement  　　http:urlReplacement元素指出所有的消息part都使用替代运算法则被编码进HTTP请求URI中：  　　http:operation 的相对URI值用于搜索一组探索方案。  　　在http:operation的值与http:address的位置属性的值结合前开始搜索。  　　每个消息part有一个搜索方式。搜索方式字符串是附加了"（"和"）"的消息part 的名字。对每个匹配，相应消息part的值在匹配处替代匹配。  　　匹配检查在任何值被替代前执行（已经替代的值不会引发另外的匹配检查）。  　　消息part不能有重复值。  　　＜http:urlReplacement/＞  3、 MIME 绑定  WSDL包含了以某种MIME格式将抽象类型与具体消息绑定在一起的方式。为下列MIME类型定义了绑定：  　　multipart/related  　　text/XML  　　application/x-www-form-urlencoded (此格式用于替代HTML中的表单)  　　其它 ( 由MIME类型字符串指定)  　　对MIME类型的定义的工作量很大，且是发展的，因此为每种MIME类型费力地定义XML语法不是WSDL的目标。在需要时，不排除增加额外的语法以定义另外的MIME类型。如果MIME类型字符串提供的信息足够描述内容，那么就可以使用下面定义的mime元素。  MIME绑定使用下列元素扩展WSDL：（注意黑体标注）  ＜mime:content part="nmtoken"? type="string"?/＞…1  ＜mime:multipartRelated＞…2  ＜mime:part＞ \* …3  ＜-- mime element --＞  ＜/mime:part＞  ＜/mime:multipartRelated＞  ＜mime:mimeXMLpart="nmtoken"?/＞ …4  1）mime:content  　　为了避免为每个MIME格式定义新的元素，如果不需要传送格式（而不是MIME类型字符串）的附加信息，就可以使用mime:content元素。  　　＜mime:content part="nmtoken"? type="string"?/＞  　　属性part用于指定消息part的名字。如果只有一个part，那么它的属性是可选的。属性type包含MIME类型的字符串。类型的值是由斜杠（/），每个值可以是一个通配符（\*）。如果没有指定类型属性，就表示所有的MIME类型是可选的。  　　如果返回格式是XML，但模式事先并不知道，通用mime元素可用于指示text/XML:  　　＜mime:content type="text/XML"/＞  　　通配符可用于指定某类MIME类型，对所有文本类型如下例所示：  　　＜mime:content type="text\*"/＞  　　＜mime:content/＞  2）mime:multipartRelated  　　multipart/related MIME类型任意的MIME格式化部分聚集为一个使用MIME 类型"multipart/related"的消息。  3）mime:part  　　元素mime:part描述了multipart/related消息的各个部分。在mime:part元素中出现的MIME元素用于为各部分指定具体的MIME类型。如果在一个mime:part元素中出现多个MIME元素，那么它们是可相互替换的。  4）mime:mimeXML为了指定与SOAP不兼容（没有SOAP封装）的XML负荷，但存在一个特殊的模式，就可以使用mime:mimeXml元素指定具体的模式。属性part引用定义了根XML元素具体模式的消息part。如果消息只有一个part，那么就可以忽略part属性。part使用element属性为单个part引用具体模式，或者使用type 属性为组合part进行引用。  5）soap:body  　　当SOAP请求使用MIME绑定时，将元素soap:body作为MIME元素使用是合法的。这指出内容的类型是"text/XML"，并且存在SOAP封装。 |

## soap文件详解

|  |
| --- |
| 协议结构（消息格式）：  <?xml  　version="1.0"?>  <!-- 必需；Envelope 英文译“信封”，该标记把此 XML 文档标识为一条 SOAP 消息-->  <soap:Envelope  　xmlns:soap="<http://www.w3.org/2001/12/soap-envelope>"  　soap:encodingStyle="<http://www.w3.org/2001/12/soap-encoding>">    <!-- 可选；消息头 -->  <soap:Header>  //...  </soap:Header>    <!--必需； 消息体；包含所有的调用和响应信息。 -->  <soap:Body>  //请求  //响应  <!--可选；处理此消息所发生错误的信息 … -->  <soap:Fault>  //...  </soap:Fault>  </soap:Body>  </soap:Envelope>  </xml> |
| 语法规则：  SOAP 消息必须用 XML 来编码   * SOAP 消息必须使用 SOAP Envelope 命名空间 * SOAP 消息必须使用 SOAP Encoding 命名空间 * SOAP 消息不能包含 DTD 引用 * SOAP 消息不能包含 XML 处理指令 |
| SOAP采用了已经广泛使用的两个协议:HTTP 和XML。HTTP用于实现 SOAP 的RPC 风格的传输, 而XML 是它的编码模式。采用几行代码和一个XML 解析器, HTTP 服务器( 如 IIS 或 Apache) 立刻成为SOAP 的 ORBS（**Object Request Broker**：对象请求代理）。SOAP 通讯协议使用 HTTP 来发送XML 格式的信息。HTTP与RPC 的协议很相似,它简单、 配置广泛,并且对防火墙比其它协议更容易发挥作用。HTTP 请求一般由 Web 服务器软件(如 IIS 或 Apache)来处理, 但越来越多的应用服务器产品正在支持HTTP。XML 作为一个更好的网络数据表达方式( NDR)。SOAP 把 XML 的使用代码化为请求和响应参数编码模式, 并用HTTP 作传输。具体地讲, 一个SOAP 方法可以简单地看作遵循SOAP编码规则的HTTP请求和响应, 一个 SOAP终端则可以看作一个基于HTTP 的URL, 它用来识别方法调用的目标。像CORBA/ IIOP一样, SOAP不需要具体的对象绑定到一个给定的终端, 而是由具体实现程序来决定怎样把对象终端标识符映像到服务器端的对象。 |
| 优点：   1. 简单的。客户端发送一个请求,调用相应的对象, 然后服务器返回结果。这些消息是XML 格式的,并且封装成符合HTTP 协议的消息。因此,它符合任何路由器、 防火墙或代理服务器的要求。 2. 完全和厂商无关。SOAP可以相对于平台、 操作系统、 目标模型和编程语言独立实现。另外,传输和语言绑定以及数据编码的参数选择都是由具体的实现决定的。 3. 与编程语言无关。SOAP 可以使用任何语言来完成，只要客户端发送正确SOAP 请求( 也就是说, 传递一个合适的参数给一个实际的远端服务器)。SOAP 没有对象模型,应用程序可以捆绑在任何对象模型中。 4. 与平台无关。SOAP 可以在任何操作系统中无需改动正常运行。 |

## wsdl文件

### WSDL的主要文档元素

WSDL文档可以分为两部分。顶部分由抽象定义组成，而底部分则由具体描述组成。

抽象部分以独立于平台和语言的方式定义SOAP消息，它们并不包含任何随 机器或语言而变的元素。这就定义了一系列服务，截然不同的应用都可以实现。

具体部分，如数据的序列化则归入底部分，因为它包含具体的定义。

在上述的文档元 素中，<types>、<message>、<portType>属于抽象定义 层，

而<binding>、<service>属于具体定义层。所有的抽象可以是单独存在于别的文件中，也可以从主文档中导入。 

### WSDL文档的结构实例解析

下面我们将通过一个实际的WSDL文档例子来详细说明各标签的作用及关系。

1. <?xml version="1.0" encoding="UTF-8"?>
2. <definitions
3. xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
4. xmlns:tns="http://www.jsoso.com/wstest"
5. xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6. xmlns="http://schemas.xmlsoap.org/wsdl/"
7. targetNamespace="http://www.jsoso.com/wstest"
8. name="Example">
9. <types>
10. <xsd:schema>
11. <xsd:import
12. namespace="http://www.jsoso.com/wstest"
13. schemaLocation="http://localhost:8080/hello?xsd=1"></xsd:import>
14. </xsd:schema>
15. </types>
16. <message name="toSayHello">
17. <part name="userName" type="xsd:string"></part>
18. </message>
19. <message name="toSayHelloResponse">
20. <part name="returnWord" type="xsd:string"></part>
21. </message>
22. <message name="sayHello">
23. <part name="person" type="tns:person"></part>
24. <part name="arg1" type="xsd:string"></part>
25. </message>
26. <message name="sayHelloResponse">
27. <part name="personList" type="tns:personArray"></part>
28. </message>
29. <message name="HelloException">
30. <part name="fault" element="tns:HelloException"></part>
31. </message>
32. <portType name="Example">
33. <operation name="toSayHello" parameterOrder="userName">
34. <input message="tns:toSayHello"></input>
35. <output message="tns:toSayHelloResponse"></output>
36. </operation>
37. <operation name="sayHello" parameterOrder="person arg1">
38. <input message="tns:sayHello"></input>
39. <output message="tns:sayHelloResponse"></output>
40. <fault message="tns:HelloException" name="HelloException"></fault>
41. </operation>
42. </portType>
43. <binding name="ExamplePortBinding" type="tns:Example">
44. <soap:binding
45. transport="http://schemas.xmlsoap.org/soap/http"
46. style="rpc"></soap:binding>
47. <operation name="toSayHello">
48. <soap:operation soapAction="sayHello"></soap:operation>
49. <input>
50. <soap:body use="literal"
51. namespace="http://www.jsoso.com/wstest"></soap:body>
52. </input>
53. <output>
54. <soap:body use="literal"
55. namespace="http://www.jsoso.com/wstest"></soap:body>
56. </output>
57. </operation>
58. <operation name="sayHello">
59. <soap:operation soapAction="sayHello"></soap:operation>
60. <input>
61. <soap:body use="literal"
62. namespace="http://www.jsoso.com/wstest"></soap:body>
63. </input>
64. <output>
65. <soap:body use="literal"
66. namespace="http://www.jsoso.com/wstest"></soap:body>
67. </output>
68. <fault name="HelloException">
69. <soap:fault name="HelloException" use="literal"></soap:fault>
70. </fault>
71. </operation>
72. </binding>
73. <service name="Example">
74. <port name="ExamplePort" binding="tns:ExamplePortBinding">
75. <soap:address location="http://localhost:8080/hello"></soap:address>
76. </port>
77. </service>
78. </definitions>

由于上面的示例XML较长，我们将其逐段分解讲解

#### WSDL文档的根元素：＜definitions＞

Xml代码

1. **<definitions**
2. xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
3. xmlns:tns="http://www.jsoso.com/wstest"
4. xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5. xmlns="http://schemas.xmlsoap.org/wsdl/"
6. targetNamespace="http://www.jsoso.com/wstest"
7. name="Example"**>**
8. ……
9. ……
10. **</definitions>**

 ＜definitions＞定义了文档中用到的各个xml元素的namespace缩写，也界定了本文档自己的 targetNamespace="http://www.jsoso.com/wstest"，这意味着其它的XML要引用当前XML中的元素时，要声 明这个namespace。注意xmlns:tns="http://www.jsoso.com/wstest"这个声明，它标示了使用tns这个前缀 指向自身的命名空间。

#### WSDL文档数据类型定义元素：<types>

Xml代码

1. **<types>**
2. **<xsd:schema>**
3. **<xsd:import**
4. namespace="http://www.jsoso.com/wstest"
5. schemaLocation="http://localhost:8080/hello?xsd=1"**></xsd:import>**
6. **</xsd:schema>**
7. **</types>**

 <types>标签定义了当前的WSDL文档用到的数据类型。要说明的是，为了最大程度的平台中立性，WSDL 使用 XML Schema 语法来定义数据类型。这些数据类型用来定义web service方法的参数和返回指。对于通用的原生数据类型如：integer , boolean , char , float等，在W3C的标准文档http://www.w3.org/2001/XMLSchema中已经做了定义。这里我们要引入的schema定义 schemaLocation="http://localhost:8080/hello?xsd=1"是我们自定义的Java对象类型。

#### WSDL文档消息体定义元素：< message >

Xml代码

1. **<message** name="toSayHello"**>**
2. **<part** name="userName" type="xsd:string"**></part>**
3. **</message>**
4. **<message** name="toSayHelloResponse"**>**
5. **<part** name="returnWord" type="xsd:string"**></part>**
6. **</message>**
7. **<message** name="sayHello"**>**
8. **<part** name="person" type="tns:person"**></part>**
9. **<part** name="arg1" type="xsd:string"**></part>**
10. **</message>**
11. **<message** name="sayHelloResponse"**>**
12. **<part** name="personList" type="tns:personArray"**></part>**
13. **</message>**
14. **<message** name="HelloException"**>**
15. **<part** name="fault" element="tns:HelloException"**></part>**
16. **</message>**

 ＜message＞元素定义了web service函数的参数。＜message＞元素中的每个＜part＞子元素都和某个参数相符。输入参数在＜message＞元素中定义，与输出参数相 隔离，输出参数有自己的＜message＞元素。兼作输入、输出的参数在输入输出的＜message＞元素中有它们相应的＜part＞元素。输出 ＜message＞元素以"Response"结尾，对Java而言方法得返回值就对应一个输出的＜message＞。每个＜part＞元素都有名字和类 型属性，就像函数的参数有参数名和参数类型。   
  
在上面的文档中有两个输入参数、两个输出参数和一个错误参数（对应Java中的Exception）。   
  
 输入参数＜message＞的name属性分别命名为toSayHello，sayHello。   
toSayHello对应输入参数userName，参数类型为xsd:string，在Java语言中就是String；   
sayHello对应两个输入参数person和arg1，类型为tns:person和xsd:string。这里tns:person类型就是引用了< types >标签中的类型定义。   
  
 输出参数＜message＞的name属性分别命名为toSayHelloResponse和sayHelloResponse。   
这个名称和输入参数的＜message＞标签name属性对应，在其后面加上Response尾缀。   
toSayHelloResponse对应的返回值是returnWord，参数类型为xsd:string；   
sayHelloResponse对应的返回值是personList，参数类型为tns:personArray（自定义类型）；   
  
 错误参数＜message＞的name属性为HelloException。   
它的<part>子标签element而不是type来定义类型。   
  
以上的message标签的name属性通常使用web service函数方法名作为参照，错误参数标签则使用异常类名为参照。标签中的参数名称，即part子元素的name属性是可自定义的（下一章节详细说 明）。message标签的参数类型将引用types标签的定义。

#### WSDL文档函数体定义元素：< portType >

Xml代码

1. **<portType** name="Example"**>**
2. **<operation** name="toSayHello" parameterOrder="userName"**>**
3. **<input** message="tns:toSayHello"**></input>**
4. **<output** message="tns:toSayHelloResponse"**></output>**
5. **</operation>**
6. **<operation** name="sayHello" parameterOrder="person arg1"**>**
7. **<input** message="tns:sayHello"**></input>**
8. **<output** message="tns:sayHelloResponse"**></output>**
9. **<fault** message="tns:HelloException" name="HelloException"**></fault>**
10. **</operation>**
11. **</portType>**

在<operation>元素中，name属性表示服务方法名，parameterOrder属性表示方法的参数顺序，使用空格符分割多个参 数，如：“parameterOrder="person arg1”。<operation>元素的子标签<input>表示输入参数说明，它引用＜message＞标签中的输入参 数。<output>表示输出参数说明，它引用＜message＞标签中的输出参数。<fault>标签在Java方法中的特别 用来表示异常（其它语言有对应的错误处理机制），它引用＜message＞标签中的错误参数。

#### WSDL绑定实现定义元素：< binding >

1. **<binding** name="ExamplePortBinding" type="tns:Example"**>**
2. **<soap:binding**
3. transport="http://schemas.xmlsoap.org/soap/http"
4. style="rpc"**></soap:binding>**
5. **<operation** name="toSayHello"**>**
6. **<soap:operation** soapAction="sayHello"**></soap:operation>**
7. **<input>**
8. **<soap:body** use="literal"
9. namespace="http://www.jsoso.com/wstest"**></soap:body>**
10. **</input>**
11. **<output>**
12. **<soap:body** use="literal"
13. namespace="http://www.jsoso.com/wstest"**></soap:body>**
14. **</output>**
15. **</operation>**
16. **<operation** name="sayHello"**>**
17. **<soap:operation** soapAction="sayHello"**></soap:operation>**
18. **<input>**
19. **<soap:body** use="literal"
20. namespace="http://www.jsoso.com/wstest"**></soap:body>**
21. **</input>**
22. **<output>**
23. **<soap:body** use="literal"
24. namespace="http://www.jsoso.com/wstest"**></soap:body>**
25. **</output>**
26. **<fault** name="HelloException"**>**
27. **<soap:fault** name="HelloException" use="literal"**></soap:fault>**
28. **</fault>**
29. **</operation>**
30. **</binding>**

 <binding>标签是完整描述协议、序列化和编码的地方，<types>,<message>和<portType>标签处理抽象的数据内容，而<binding>标签是处理数据传输的物理实现。   
<binding>标签把前三部分的抽象定义具体化。   
  
首先<binding>标签使用<soap:binding>的transport和style属性定义了Web Service的通讯协议HTTP和SOAP的请求风格RPC。其次<operation>子标签将portType中定义的 operation同SOAP的请求绑定，定义了操作名称soapAction，输出输入参数和异常的编码方式及命名空间。

#### WSDL服务地址绑定元素：< service >

1. **<service** name="Example"**>**
2. **<port** name="ExamplePort" binding="tns:ExamplePortBinding"**>**
3. **<soap:address** location="http://localhost:8080/hello"**></soap:address>**
4. **</port>**
5. **</service>**

 service是一套＜port＞元素。在一一对应形式下，每个＜port＞元素都和一个location关联。如果同一个＜binding＞有多个＜port＞元素与之关联，可以使用额外的URL地址作为替换。   
  
一个WSDL文档中可以有多个＜service＞元素，而且多个＜service＞元素十分有用，其中之一就是可以根据目标URL来组织端口。在一个 WSDL文档中，＜service＞的name属性用来区分不同的service。在同一个service中，不同端口，使用端口的"name"属性区 分。   
  
简单的描述了WSDL对SOAP协议的支持，以及在Web Service中的作用。

## SOAP消息的绑定机制

以下请与2.6.2.5和2.4.4.1结合阅读。

摘要：Web服务的有效负载包装在SOAP消息中，而SOAP消息体通过绑定服务调用方式封装操作，绑定编码方式序列化参数。然而，调用方式和编码方式的不同组合就产生了多种SOAP绑定样式，而不同样式的SOAP消息构造各不相同，也有各自的优缺点。这就给服务的开发和调用带来了困难：如何选择服务绑定样式；如何根据绑定样式构造SOAP消息。

### 引言

Web服务技术解决了异构平台/系统之间应用的整合问题，为企业间应用集成提供了技术基础。Web服务互联互通是Web服务应用和集成的基础，保证Web服务之间互联互通的一致性是促进Web服务技术应用和发展的关键技术问题之一。由于Web服务采用WSDL (Web Services Description Language)语言定义消息格式和内容、采用SOAP(Simple Obiect  Access Protocol)协议进行消息通信。因此，Web服务之间互连互通的一致性就是WSDL/SOAP接口之间的一致性。

Web服务的有效负载通常包装在SOAP消息中，而SOAP消息结构由WSDL文档中的SOAP绑定定义确定。不同的调 用方式和编码方式通过组合可以产生多种绑定样式，而每种样式的应用场景和对应的SOAP消息结构并不相同。如果没有正确的构造SOAP消息，则无法正确交 换服务的有效负载，即使有效负载遵循提供者和使用者的公共XML模式也是如此。这就给服务的开发和调用带来了挑战：如何根据应用场景选择合适的服务绑定样 式？如何根据绑定样式构造正确的SOAP消息？因此，理解SOAP消息的绑定机制和各种服务绑定样式的消息构造，以及它们各自的优缺点就显得非常重要。尤 其是对于Web服务基础平台和工具（如Web服务运行平台、Web服务开发工具、Web服务测试工具等）的开发者而言就更加重要。

本文将详细介绍SOAP消息的绑定机制和根据绑定样式构造SOAP消息，以及解决互操作性问题的相关规范。本文的主要目的是为了帮助开发者更加全面透彻地理解SOAP消息的绑定和构造。

### 消息绑定

SOAP Body提供了一种消息交换的机制，是SOAP消息的实际负载，可包含任意内容。SOAP消息体(SOAP Body)通过绑定服务调用方式(RPC或者Document)封装操作，绑定编码方式(Encoded或者Literal)序列化参数。SOAP消息的 绑定样式由style、use和encodingStyle这三个属性共同设置。style属性指定服务的调用方式，是采用RPC方式还是 Document方式；use属性指定消息的编码方式，是采用Encoded方式还是采用Literal方式；而encodingStyle属性指定具体 编码规则，例如可以指定SOAP编码规则、XML Schema编码规则等等，通常情况下都是采用XML Schema。

#### style属性

style属性描述了服务的调用方式，取值为”rpc”或者”document”，默认值为”document”。适用于SOAP Body元素的子元素（也可能是孙级元素）。此选项指定为WSDL文档中的soap:binding元素（通常情况下）或者soap:operation 的 style 属性。

##### RPC

RPC: (Remote Procedure Call) 采用客户端/服务器方式(请求/响应)，发送请求到服务器端，服务端执行方法后返回结果。优点是跨语言跨平台，缺点是编译时无法排错，只能在运行时检查。

SOAP消息本质上是一种从发送方到接收方的单向传输，但是SOAP经常组合到实现请求/响应机制中。要让RPC使用SOAP，必须遵循几条规则：

* 首先，请求和响应消息必须被编码成结构类型。
* 其次，对一个操作的每一个输入参数，都必须有一个同名元素（或输入结构的成员）作为参数。
* 最后，对每一个输出参数，都必须有一个名称匹配的元素（或输出结构的成员）。

style=”rpc”指明遵从SOAP标准，在SOAP Body中封装RPC调用的请求和返回操作。对该类消息的约束是必须把“操作的名称”作为“封装了操作的请求和响应消息负载”的“根元素名称”。对于SOAP请求消息，请求操作的名称是根据WSDL文档中的wsdl:operation元素命名；对于SOAP响应消息，响应操作的名称是请求操作的名称后面追加"Response"构成。操作元素中的每个子元素表示一个参数，根据WSDL文档中的wsdl:part元素命名。RPC请求/响应 消息的格式(省略了名称空间限定)如下：

|  |  |
| --- | --- |
| RPC请求消息格式 | RPC响应消息格式 |
| <SOAP信封>  <SOAP头部>       ……  </SOAP头部>   <SOAP消息体>  <请求操作名称>            <参数名1>值</参数名1>            <参数名2>值</参数名2>             ......     </请求操作名称>  </SOAP消息体>  </SOAP信封> | <SOAP信封>  <SOAP头部>    ……  </SOAP头部>  <SOAP消息体>         <响应操作名称>            <参数名1>值</参数名1>            <参数名2>值</参数名2>                ......          </响应操作名称>   </SOAP消息体>  </SOAP信封> |

##### Document

  与RPC相比较，Document方式在XML文件中不是做远程方法的映射，而是一份完整的自包含的业务文档。当服务器端收到这份文档后，先进行预处理（比如词汇的翻译和映射），然后再构造出返回消息。这个构造返回消息的过程中，往往不再是简简单单的一个方法调用，而是多个对象协同完成一个事务的处 理，再将结果返回。

采用Document方式构造消息，SOAP Body元素中的内容完全由WSDL 中的 XML Schema定义，于是可以使用XML Schema验证SOAP Body元素中的内容。SOAP Body的子元素指定为消息部分在WSDL文档中定义的 wsdl:part 元素，该元素指向 XML Schema元素声明。通常wsdl:message不会包含多个wsdl:part 元素，所以SOAP Body 内容是真正的 XML 文档，尽管 WSDL 本身不禁止包含多个wsdl:part元素。Document请求/响应消息的格式(省略了名称空间限定)如下：

|  |  |
| --- | --- |
| Document请求消息格式 | Document响应消息格式 |
| <SOAP信封>  <SOAP头部>       ……  </SOAP头部>   <SOAP消息体>     <输入消息>     ….    </输入消息>  ….  </SOAP消息体>  </SOAP信封> | <SOAP信封>  <SOAP头部>   ……  </SOAP头部>  <SOAP消息体>     <输出消息>     ….    </输出消息>  ….  </SOAP消息体>  </SOAP信封> |

#### use属性

use属性描述了消息序列化的方式，取值为”encoded”或者”literal”，默认值是”literal”。 如果use=”encoded”，设置encodingStyle属性的值指定编码规则。如果use=”literal”，可以不用设置 encodingStyle属性的值，通常情况都是使用默认的XML Schema。适用于出现在下一个级别的 Web 服务方法参数（或返回值）。此选项指定为WSDL文档中的 soap:body、soap:header、soap:fault和soap:headerfault元素的 use 属性。

##### Encoded

SOAP 编码使用 XML Schema的一个子集在 XML 文档与其所表示的数据之间进行绑定。SOAP 编码还使用href属性对文档中多次出现的元素的引用。SOAP编码和XML Schema编码完全不同的地方是数组。SOAP规范的5.4.2节指定了一种特别的机制来表示 XML 中的编程语言数组，它使用一种特殊的SOAP-ENC:Array类型。而XML Schema是通过设置element的minOccurs和maxOccurs属性值表示数组。例如使用这两种编码方式对整形数组的定义如下：

|  |
| --- |
| SOAP编码定义数组  <complexType name="ArrayOfInt">    <complexContent>       <restriction base="SOAP-ENC:Array">  <sequence>      <element name=”item” type=”xsd:int” minOccurs=”0”  maxOccurs=”unbounded”/>  </sequence>          <attribute ref="SOAP-ENC:arrayType"  wsdl:arrayType="xsd:ArrayOfInt[]"/>       </restriction>    </complexContent>  </complexType> |
| XML Schema定义数组  <complexType name="ArrayOfInt">  <complexContent>  <sequence>    <element name=”item” type=”xsd:int” minOccurs=”0” maxOccurs=”unbounded”/>  </sequence>   </complexContent>  </complexType> |

假设一个加法服务的定义为public  int[]  add(int a[],int b[])，使用Encoded方式编码add方法的消息格式如下：

|  |
| --- |
| <op:add xmlns:op=”http://act.buaa.edu.cn/add”  xmlns: SOAP-ENC =” http://schemas.xmlsoap.org/soap/encoding/”  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  xmlns:xsi=” http://www.w3.org/2001/XMLSchema-instance” >    < a  SOAP-ENC:arrayType=”xsd:int[2]” xsi:type=”SOAP-ENC:array”>    <item xsi:type=”xsd:int”>45</item>  <item xsi:type=”xsd:int”>36</item>  </a>  <b  SOAP-ENC:arrayType=”xsd:int[2]” xsi:type=”SOAP-ENC:array”>    <item xsi:type=”xsd:int”>235</item>  <item xsi:type=”xsd:int”>67</item>  </b>  </op:add> |

##### Literal

与Encoded相比，Literal采用XML Schema编码，而不是SOAP编码；Literal编码不需要数据类型属性。数据根据 WSDL 文档中指定的 XML Schema定义或导入 WSDL 文档的 XML Schema定义逐字进行格式设置。使用Literal方式编码add方法的消息格式如下：

|  |
| --- |
| <op:add xmlns:op=”http://act.buaa.edu.cn/add”>  <a>  <item>45</item>  <item>36</item>  </a>  <b>  <item>235</item>  <item>67</item>  </b>  </op:bdd> |

#### encodingStyle属性

WSDL规范中的encodingStyle属性和SOAP规范中的encodingStyle 属性定义是一样的。encodingStyle属性的值是一个URIs列表，由单个空格隔开，每个URI代表着一种消息的编码规则，它们按照编码的限制从 强到弱排序。encodingStyle属性是用来指定SOAP消息的编码规则，也就是序列化的格式和类型系统。如果 use=”encoded”，encodingStyle属性的值为”http://schemas.xmlsoap.org/soap /encoding/”，则表示使用SOAP规范的第5节的编码，这一节定义了将编程语言的类型映射到 XML 的基本机制。如果use=”literal”则使用外部提供的类型系统，也可以通过encodingStyle属性来指定Schema，但是通常情况下使 用XML Schema来编码SOAP消息。

这其中的缘故是因为SOAP规范是在采用XML Schema规范之前编写的。因此，原始的SOAP规范必须提供一种方法来指明编码类型信息。然而，自从采用XML Schema之后，大多数语言使用自己的从XML Schema到编程语言类型之间的映射（或序列化规则），这使得SOAP编码变得过时。因此，推荐不要采用SOAP编码，而是采用使用Literal编 码，在Literal映射中由 XML Schema 文档（通常是WSDL文档的形式）从外部指定映射。

### 消息构造

style和use属性都有两个值，通过它们之间的不同组合，就可以产生四种绑定样式，分别是rpc-literal、rpc-encoded、 document-literal和document-encoded。为了使Document绑定样式能够支持RPC绑定样式的调用，增加了一种包装 (Wrapped)样式，它只不过是对Document样式的使用进行了约束。这样就又增加了两种绑定样式， document-literal-wrapped和document-encoded-wrapped，一共合起来就有6种绑定样式。下面通过一个加法服务为例来说明在不同绑定样式下的SOAP消息构造，服务的定义如下：

|  |
| --- |
| //Java Code  public  int add(int  a , int  b){  return a+b;  } |

从服务的定义可以看出，输入操作的操作名为add，有a和b两个整数类型的输入参数。根据WSDL规范返回操作的操作名应该为addResponse，有一个整数类型的输出参数，假设命名为return。

操作的序列化是需要名称空间限定的。如果是RPC调用方式，则名称空间是soap :body的namespace属性值；如果是Document 调用方式，名称空间则是输入消息引用的Schema的targetNamespace属性值。

参数的序列化是否需要名称空间限定取决于Schema定义时 elementFormDefault属性的值。如果值为”qualified”则表示需要限定，名称空间为Schema的 targetNamespace属性值；如果值为”unqualified”表示不用限定，默认值为”unqualified”。

操作序列化后的元素作为 SOAP消息体的子元素，而每个参数序列化后的元素都是作为操作元素的子元素，排列的顺序和操作的参数定义顺序一样。

假定以下所有SOAP消息都是在如下条件下构造：

* 如果use= ”encoded”则encodingStyle的值为”http://schemas.xmlsoap.org/soap /encoding/”，如果use= ”literal”则使用” http://www.w3.org/2001/XMLSchema”编码。
* 如果style=”rpc”则wsdl:part 部分的类型引用使用type属性，如果style=”document”则使用element属性。

#### rpc-literal绑定样式

|  |
| --- |
| WSDL描述：  <wsdl:definitions xmlns:wsdl= " http://schemas.xmlsoap.org/wsdl/"     xmlns:ns="http://act.buaa.edu.cn/add"  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  targetNamespace="http://act.buaa.edu.cn/add">  <wsdl:message name=”addRequest”>    <wsdl:part name=”a” type=”xsd:int”/>      <wsdl:part name=”b” type=”xsd:int ”/>  </wsdl:message>  <wsdl:message name=”addReponse”>      <wsdl:part name=”return” type=”xsd:int”/>  </wsdl:message>  <!--Just assume it's rpc-literal. -->  …… |
| SOAP请求消息：  <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">     <env:Body>       <op: add  xmlns:op=“http://act.buaa.edu.cn/add”>         <a>12</a>  <b>45</b>       </op: add >     </env:Body>  </env:Envelope> |

优点：WSDL 描述和SOAP消息基本达到了尽可能地简单易懂的要求；服务的操作名出现在SOAP消息的Body中，这样接收者就可以很轻松地把消息发送到方法的实现；没有类型编码，提高了吞吐量，减少了处理的性能开销。

缺点：在RPC模型中XML仅仅被用于描述方法的信息，不能充分利用XML的功能去描述和验证一份业务文档 ；不能使用Schema简单地检验此消 息的有效性，因为只有参数a和b是Schema 中定义的内容，其余的 env:body 内容(例如add元素)都来自 WSDL 定义；无法直接从消息中获得参数的类型信息；RPC样式对请求/响应消息的模式捆绑，使得服务与客户端之间耦合性增加，一旦方法发生变化，客户端就需要做 相应的改动；相对异步调用方式而言，RPC样式下服务调用通常是同步的。

#### rpc-encoded 绑定样式

  与rpc-literal绑定样式唯一的区别就是请求消息的参数部分编码方式不一样。消息中的每个参数都有类型编码（比如 <a xsi:type=”int”>12</a>），直接从消息就可以知道参数的数据类型。但是这些类型信息降低了吞吐量和消息处理的性 能。尤其是在大数据的情况下，性能开销明显。

|  |
| --- |
| 参照rpc-literal绑定样式的WSDL文档 |
| SOAP请求消息：  <env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"         xmlns:xsd="<http://www.w3.org/2001/XMLSchema>"  xmlns:xsi=” http://www.w3.org/2001/XMLSchema-instance” >    <env:Body>       <op: add xmlns:op=“http://act.buaa.edu.cn/add”>          < a  xsi:type=”xsd:int”>12</a>  <b  xsi:type=”xsd:int”>45</b>       </op: add >     </env:Body>  </env:Envelope> |

rpc-encoded绑定样式主要在重载、数据图形和多态的情况下使用。WSDL 允许重载的操作，但是当参数个数相同的情况下，因为Literal编码方式没有类型信息，无法定位方法，所以rpc-literal就不支持这样的重载。 数据图形的标准方式是使用href 标记，它是 rpc-encoded的样式。多态所生成的 SOAP 消息必须包含类型编码信息，这样接收终端才能知道它所接收的是父类的哪一个扩展。

#### document /literal绑定样式

  请求消息的参数元素添加了名称空间限定，这是因为输入消息在WSDL文档的Schema中定义，而且Schema的 elementFormDefault=”qualified”。也就是说参数元素必须使用Schema的名称空 间"http://act.buaa.edu.cn/add"进行限定。

|  |
| --- |
| WSDL 描述：  <wsdl:definitions xmlns:wsdl= " http://schemas.xmlsoap.org/wsdl/"  xmlns:ns="http://act.buaa.edu.cn/add"  targetNamespace=" http://act.buaa.edu.cn/add ">  <wsdl :types>  <xs:schema elementFormDefault="qualified"  targetNamespace="http://act.buaa.edu.cn/add"  xmlns:xs="http://www.w3.org/2001/XMLSchema" >         <xs:element name="a"  type="xs:int " />         <xs:element name="b"  type="xs:int " />         <xs:element name="return"  type="xs:int " />  </xs :schema>  </wsdl :types>  <wsdl:message name=”addRequest”>       <wsdl:part  name=”parameter1” element=”ns:a”/>  <wsdl:part  name=”parameter2” element=”ns:b”/>  </wsdl:message>  <wsdl:message name=”addReponse”>       <wsdl:part name=”parameter” element=”ns:return”/>  </wsdl:message>  <!--Just assume it's  document-literal. -->  …… |
| SOAP 请求消息：  <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"  xmlns:op=“http://act.buaa.edu.cn/add”>  <env:Body>      <op:a>12</op:a>  <op:b>45</op:b>  </env:Body>  </env:Envelope> |

优点：没有操作和类型编码信息，减少了消息的数据量，提高了消息处理性能；env:Body中每项内容都定义在Schema中，可以用任何XML检验器检验此消息的有效性。

缺点：WSDL文档变得比较复杂，这不过是一个非常小的缺点，因为 WSDL 并没有打算由人来读取；SOAP消息中没有提供服务操作的名称，一些特定的程序代码在分发消息时可能会变得复杂，并且有时变得不可能。如果使用HTTP作 为底层传输协议，可以使用SOAPAction属性绑定操作的名称来解决消息分发的问题。虽然大多数情况下都是使用HTTP协议来传输SOAP消息，但是 这种方法绑定了底层传输协议，限制了其他传输协议的使用。

#### document /encoded绑定样式

|  |
| --- |
| 参照document-literal绑定样式的WSDL文档 |
| SOAP 请求消息：  <env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  xmlns:xsi=” http://www.w3.org/2001/XMLSchema-instance”         xmlns:op=“http://act.buaa.edu.cn/add”>     <env:Body>         <op:a  xsi:type=”xsd:int”>12</op:a>  <op:b  xsi:type=”xsd:int”>45</op:b>     </env:Body>  </env:Envelope> |

与document-literal绑定样式唯一的区别就是请求消息的参数部分编码方式不一样。引入了类型编码，降低了吞吐量和消息处理的性能。这种绑定样式不被大多数Web服务实现平台支持。

#### document-literal-wrapped绑定样式

|  |
| --- |
| WSDL描述：  <wsdl:definitions  xmlns:wsdl= " http://schemas.xmlsoap.org/wsdl/"  xmlns:ns="http://act.buaa.edu.cn/add"  targetNamespace=" http://act.buaa.edu.cn/add ">  <wsdl :types>  <xs:schema elementFormDefault="qualified" targetNamespace="http://act.buaa.edu.cn/xsd" xmlns:xs="http://www.w3.org/2001/XMLSchema" >         <xs:element name="add">            <xs:complexType>              <xs:sequence>                 <xs:element name="a"  type="xs:int " />                 <xs:element name="b"  type="xs:int " />              </xs:sequence>            </xs:complexType>         </xs:element>         <xs:element name="addResponse">            <xs:complexType>              <xs:sequence>                 <xs:element name="return"  type="xs:int " />              </xs:sequence>           </xs:complexType>         </xs:element>  </xs :schema>  </wsdl :types>  <wsdl:message name=”addRequest”>      <wsdl:part  name=”parameter” element=”ns:add”/>  </wsdl:message>  <wsdl:message name=”addReponse”>      <wsdl:part name=”parameter” element=”ns:addResponse”/>  </wsdl:message>  <!--Just assume it's  document-literal-wrapped. -->  …… |
| SOAP请求消息：  <env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">     <env:Body>       <op: add  xmlns:op=“http://act.buaa.edu.cn/add”>         <op:a>12</op:a>  <op:b>45</op:b>       </op: add >    </env:Body>  </env:Envelope> |

此时SOAP Body中第一个元素的名称并不是操作的名称，而是Schema中的元素的名称。Schema中的元素的名称可以与操作名相同，也可以不同。如果取相同则是一种将操作名放入SOAP消息的巧妙方式。

虽然此SOAP 消息看起来与 rpc-literal的 SOAP 消息是完全一样的(如果不考虑名称空间限定)，但是这两种消息之间存在着微妙的区别。在 rpc-literal的 SOAP 消息中，<env:Body>下的<op:add>根元素是操作的名称。在document-literal-wrapped的 SOAP 消息中，<op:add>元素是单个输入消息的组成部分引用的元素的名称。

document-literal-wrapped样式规定Document绑定操作的输入消息和输出消息都只有一个wsdl:part部分；该部分使用element属性引用一个元素；该元素是复杂类型并且没有属性；该元素的名称和操作的名称必须一样。

优点：包装行为吸取了RPC样式的一个重要优点，即RPC样式中SOAP消息体可以直接通过与之关联的服务操作名称来命名，同时又摒弃了RPC样式 的不足之处；可以利用WSDL文档类型部分的Schema文档直接来验证SOAP消息体；只要在Schema中定义了明确的数据结构，如何构建SOAP消 息体具有很大的灵活性；由于业务数据是自包含的，显然文档模型更利于采用异步处理。

 缺点：WSDL 较为复杂, 但是这仍然是一个非常小的缺点。使得服务调用的复杂度有所增加，尤其是在API级别。针对程序开发人员来说，基于包装的document绑定样式的服务编 写客户端代码也许就变成了一项极具挑战性的工作。在SOAP消息体的XML包装元素中必须拥有一个服务操作的名称，因此包装版本不支持重载的服务操作。实 际上，针对一个既定的元素名称也只能够有一个服务操作。

#### document-encoded-wrapped绑定样式

|  |
| --- |
| 参照document-literal-wrapped绑定样式的WSDL文档 |
| SOAP请求报文  <env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  xmlns:xsi=” http://www.w3.org/2001/XMLSchema-instance” >    <env:Body>       <op: add  xmlns:op=“http://act.buaa.edu.cn/add”>          <op:a  xsi:type=”xsd:int”>12</op:a>  <op:b  xsi:type=”xsd:int”>45</op:b>       </op: add >     </env:Body>  </env:Envelope> |

与document -literal-wrapped绑定样式唯一的区别就是请求消息的参数部分编码方式不一样。引入了类型编码，降低了吞吐量和消息处理的性能。

### 互操作性

虽然Web服务解决了异构平台/系统之间应用的互操作性，但是不同Web服务实现平台的差异性带来了新的互操作性问题。主要原因如下：

* 不同的标准化组织规定了许多不同的Web服务标准，各标准也有许多不同的版本。
* SDL/SOAP协议的某些规定的模糊性和灵活性，使得每个人对协议本身的理解并不完全一致。
* 不同Web服务实现平台对Web服务标准的支持程度不一致。

于是成立了WS-I(Web Services Interoperability Organization)组织，它致力于促进跨平台、跨操作系统和跨程序语言的网络服务互操作性。然而WS-I不是一个标准制订组织，它只是站在实践的 角度，为在不同的环境下如何选择和使用各标准组织提供的各类标准提出实践的建议，它是W3C的补充。

WS-I 的第一个基本概要(Basic Profile 1.0，简称BP 1.0)在阐明各个规范方面做得非常不错，但它并不完美。尤其对SOAP with Attachments (Sw/A)的支持仍然相当不明确。于是同一年WS-I发布了第二个基本概要(BP 1.1)用于描述SOAP 1.1、WSDL 1.1和带附件的MIME SOAP协议的互操作性。WS-I将附件从BP 1.1中分离出来，并对第一版中一些没有讨论的内容进行了补充。当时WS-I添加了两个互不包括的基本概要补充文档：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是简单SOAP绑定概要 (Simple SOAP Binding Profile)，描述并不支持Sw/A的Web服务引擎。WS-I 所提供的其他概要文件都是以这些基本概要文件为基础构建的。之后WS-I又发布了第三个基本概要(BP 1.2)和第四个基本概要(BP 2.0)，这两个基本概要都兼容BP 1.1。BP 1.2在BP 1.1的基础上添加了一些新的约束，而BP 2.0的最大不同之处就是它支持的SOAP版本是SOAP 1.2。

BP 1.1 对消息绑定样式进行了限制：禁止使用Encoded编码，只能使用XML Schema 1.0编码，名称空间为” http://www.w3.org/2001/XMLSchema”；只支持rpc-literal和document-literal（当然 document-literal-wrapped也是document-literal样式）两种绑定样式；RPC绑定必须使用 wsdl:part 中的type属性，而 Document 绑定必须同element属性一起使用；属性element引用 XML 模式元素，而属性type则指示XSD中的 simpleType 和 complexType。

简而言之，Document 样式的消息基于 XML Schema元素定义，而 RPC 样式的消息则使用 XML Schema类型定义。而且，只有全局级别的元素和类型能够在 WSDL 定义中定义，这些元素和类型是 XSD 中的 <schema> 的直接子项。所有非直接子项组件都是本地的，通常嵌套在另一个模式组件中，此处的组件将引用模式元素、complexType 或 simpleType。

### 总结

本文中涉及的SOAP规范和WSDL规范分别是SOAP 1.1版本和WSDL 1.1版本。WSDL 1.1和WSDL 2.0规范都可以绑定SOAP 1.1 和SOAP 1.2规范，因为这两个SOAP版本的差别并不大。WSDL 2.0版本已经消除了wsdl:message和wsdl:part元素，而是直接使用element属性引用XML Schema的元素定义，所以WSDL 2.0不存在文中所介绍的六种样式。因为目前WSDL 1.1规范仍然广泛使用，所以了解各种消息绑定样式和WS-I的BP规范还是非常重要的。BP1.1规范仅支持rpc-literal、 document-literal和document-literal-wrapped三种样式，虽然每种样式都有自己的用武之地，但是在大多数情况下， 最好的样式还是document-literal-wrapped样式。

## 如何开发web service

（1）开发阶段：

   实现一个 Web 服务，使这个 Web 服务能响应和接收 SOAP 消息(开发工具会自动帮助实现）；

定义好逻辑模块（这个 Web 服务总要干点事情吧，这个就是我们程序员要干的事情了）；

然后再撰写 WSDL 文件（这个呢，开发工具会自动生成的，不需要人工撰写了）；

（2）部署阶段：

     指定 Web 服务的传输协议，将 Web 服务注册到相应服务描述部署文件（这些也可以由工具来自动完成的）

（3）发布阶段：

      将 Web 服务的接口和调用的地址公开给客户端调用，常用的发布方式为基于 Web 提供的WSDL 的链接，当然 UDDI 也是一个选择。

总结，我们基本上主要精力就放在这个上面（红字部分），其他的工具（如cxf）都已会帮我们来完成。

# java webService

## Java Webservice规范

### JAX-WS

一种 Java 规范，名为 JAX-WS（JSR-224），全称 Java API for XML-Based Web Services，可以将规范理解为官方定义的一系列接口。

### JAX-RS

一种java规范，名为 JAX-RS（JSR-339），全称 Java API for RESTful Web Services，可以将规范理解为官方定义的一系列接口。 为了让 WS 的开发与使用变得更加简单、更加轻量级，于是出现了这种风格的 WS。 它也有若干实现，cxf是其中比较著名的一种。

## Java webService框架--cxf 和axis

常见的java webService框架有cxf和axis。两者都是apache的开源项目。

CXF = Celtix + XFire，开始命名为 Apache CeltiXfire，后更名为 Apache CXF，现在版本是3.x

Axis2是从Axis1.x系列发展而来。

1>.CXF支持 WS-Addressing，WS-Policy， WS-RM， WS-Security和WS-I Basic Profile。Axis2不支持WS-Policy，但是承诺在下面的版本支持。

2>. CXF可以很好支持Spring。Axis2不能

3>. AXIS2支持更广泛的数据并对，如XMLBeans，JiBX，JaxMe和JaxBRI和它自定义的数据绑定ADB。注意JaxME和JaxBRI都还是试验性的。CXF只支持JAXB和Aegis。在CXF2.1

4>. Axis2支持多语言-除了Java,他还支持C/C++版本。

比较这两个框架的Web Service开发方法与比较它们的特性同样重要。 从开发者的角度，两个框架的特性相当的不同。 Axis2的开发方式类似一个小型的应用服务器，Axis2的开发包要以WAR的形式部署到Servlet容器中，比如Tomcat，通过这些容器可以对工作中的Web Service进行很好的监控和管理。Axis2 的Web administrion模块可以让我们动态的配置Axis2.一个新的服务可以上载，激活，使之失效，修改web服务的参数。管理UI也可以管理一个或者多个处于运行状态的服务。这种界面化管理方式的一个弊端是所有在运行时修改的参数没有办法保存，因为在重启动之后，你所做的修改就会全部失效。

Axis2允许自己作为独立的应用来发布Web Service，并提供了大量的功能和一个很好的模型，这个模型可以通过它本身的架构（modular architecture）不断添加新的功能。有些开发人员认为这种方式对于他们的需求太过于繁琐。这些开发人员会更喜欢CXF。

CXF更注重开发人员的工效（ergonomics）和嵌入能力（embeddability）。大多数配置都可以API来完成，替代了比较繁琐的XML配置文件， Spring的集成性经常的被提及，CXF支持Spring2.0和CXF's API和Spring的配置文件可以非常好的对应。CXF强调代码优先的设计方式（code-first design)，使用了简单的API使得从现有的应用开发服务变得方便。

## java webService注解

注解api 位于j2se rt.jar中。

以下注解的说明，均是在cxf-3.07下进行验证。

### @WebService

|  |
| --- |
| \*两种方式开发webserbice：接口+接口实现-->以下称之为A；类-->以下称之为B；  \*  \* param：name-->used as of <wsdl:portType name="">;  \* A情况下-->都不指定(接口+接口实现)， <wsdl:portType name="">将使用“接口的非限定名”  \* A情况下-->接口指定、接口实现不指定， <wsdl:portType name="">将使用“接口的指定”  \* A情况下-->接口不指定、接口实现指定， <wsdl:portType name="">将使用“接口的非限定名”  \* A情况下-->接口指定、接口实现指定， <wsdl:portType name="">将使用“接口的指定”  \* B情况下-->类不指定， <wsdl:portType name="">将使用“类的非限定名”  \* B情况下-->类指定， <wsdl:portType name="">将使用“类的指定”  \*  \* 总结：<wsdl:portType name=""> 由接口说了算，接口不存在接口实现才说了算；  \*  \* portType究竟是什么？直译“端口类型”，在java世界中就是“接口类型”的意思。  \*  \* param：portName-->used as the name of <wsdl:port name="">;  \* A情况下-->都不指定(接口+接口实现)， <wsdl:port name="">将使用“接口实现的非限定名+prot”  \* A情况下-->接口指定、接口实现不指定， <wsdl:port name="">将使用“接口实现的非限定名+prot”  \* A情况下-->接口不指定、接口实现指定， <wsdl:port name="">将使用“接口实现的指定”  \* A情况下-->接口指定、接口实现指定， <wsdl:port name="">将使用“接口实现的指定”  \* B情况下-->类不指定， <wsdl:porte name="">将使用“类的非限定名+prot”  \* B情况下-->类指定， <wsdl:port name="">将使用“类的指定”  \*  \* 总结：<wsdl:port name=""> 由接口实现说了算，根接口没关系；  \* 另，A情况或B情况-->当接口实现或类指定了name属性，但没指定portName属性，则<wsdl:port name="">将使用“接口实现或类的  \* name属性值+port”  \*  \* port究竟是什么？直译“端口”，在java世界中就是“入口”的意思，“入口是什么？当然是实现喽”  \*  \*  \* param：serviceName-->used as the name of <wsdl:service name="">;  \* A情况下-->都不指定(接口+接口实现)， <wsdl:service name="">将使用“接口实现的非限定名+Service”  \* A情况下-->接口指定、接口实现不指定， <wsdl:service name="">将使用“接口实现的非限定名+Service”  \* A情况下-->接口不指定、接口实现指定， <wsdl:service name="">将使用“接口实现的指定”  \* A情况下-->接口指定、接口实现指定， <wsdl:service name="">将使用“接口实现的指定”  \* B情况下-->类不指定， <wsdl:service name="">将使用“类的非限定名+Service”  \* B情况下-->类指定， <wsdl:service name="">将使用“类的指定”  \*  \* 总结：<wsdl:service name=""> 由接口实现或类说了算，根接口没关系；  \* 另，A情况或B情况-->当接口实现或类指定了name属性，但没指定serviceName属性，则<wsdl:service name="">依旧使用“接口实现  \* 或类的非限定名+Service”  \*  \* serviceName究竟是什么？直译“服务名称”，在java世界中就是“服务名称”的意思？当然是实现喽  \*  \* param：wsdlLocation-->正常情况下，cxf会为我们自动生成wsdl文件，但是如果程序员自己编写了wsdl文件，那么在这里可以指定程序员编写wsdl文件  \* 的访问地址，请注意一旦指定cxf就不会为我们在生成了。  \* |

### @WebMethod、@Oneway、@WebParam、@WebResult

|  |
| --- |
| /\*\*  \* 没有@WebMethod注解，则所有方法全部皆为webservice  \*  \* 以下为@WebMethod注解属性说明：  \* param:operationName->指定与此方法相匹配的wsdl:operation 的名称;缺省值为 Java 方法的名称。  \* <wsdl:operation ...>是<wsdl:binding ...>子标签  \* 总结：此属性值以接口为准，当接口不存在时则已类为准。  \*  \* param:action-->指定<soap:operation soapAction=""/> soapAction 的值;缺省值为 “”。  \* <soap:operation soapAction=""/>是 <wsdl:operation ...>的子标签  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \* param:exclude->指定是否从 Web Service 中排除某一方法。缺省值为 false。  \* 总结：此属性值只能配置在接口实现或类中，不能配置到接口中。  \*  \* 注解@Oneway:只有输入消息而没有输出消息的 Web Service 单向操作。  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \*  \* 注解：WebParam  \* param:name->如果操作是远程过程调用（RPC）类型并且未指定partName 属性，那么这是用于表示参数的 wsdl:part 属性的名称。  \* 如果操作是文档类型或者参数映射至某个头，那么 是用于表示该参数的 XML 元素的局部名称。  \* <xs:complexType name="hello">  \* <xs:sequence>  \* <xs:element minOccurs="0" name="name0" type="xs:string"/>  \* </xs:sequence>  \* </xs:complexType>  \* 如果操作是文档类型、参数类型为 BARE 并且方式为 OUT 或 INOUT，那么必须指定此属性。  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \* param:partName->定义用于表示此参数的 wsdl:part属性的名称。  \* 仅当操作类型为 RPC 或者操作是文档类型并且参数类型为BARE 时才使用此参数。（字符串）  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \* param:mode->此值表示此方法的参数流的方向。有效值为 IN、INOUT 和 OUT。（字符串）  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \* param:header-> 指定参数是在消息头还是消息体中。缺省值为 false。（布尔值）  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \* 注解：WebResult  \* param:name->当返回值列示在 WSDL 文件中并且在连接上的消息中找到该返回值时，指定该返回值的名称。  \* 对于 RPC 绑定，这是用于表示返回值的 wsdl:part属性的名称。  \* 对于文档绑定，-name参数是用于表示返回值的 XML 元素的局部名。  \* 对于 RPC 和 DOCUMENT/WRAPPED 绑定，缺省值为 return。  \* 对于 DOCUMENT/BARE 绑定，缺省值为方法名 + Response。（字符串）  \*  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \* param:partName->指定 RPC 或 DOCUMENT/BARE 操作的结果的部件名称。缺省值为@WebResult.name。  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \* param:header-> 指定头中是否附带结果。缺省值为false。（布尔值）  \* 总结：此属性值以接口为准，当接口不存在时则以类为准。  \*  \*  \*/ |

## cxf

### cxf 依赖

#### 服务端依赖

|  |
| --- |
| <!-- CXF Dependencies -->  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-rt-frontend-jaxws</artifactId>  <version>${cxf.version}</version>  </dependency>  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-rt-transports-http</artifactId>  <version>${cxf.version}</version>  </dependency>  <!-- Jetty is needed if you're are not using the CXFServlet -->  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-rt-transports-http-jetty</artifactId>  <version>${cxf.version}</version>  </dependency>  <!-- End of CXF Dependencies --> |

#### 客户端依赖

|  |
| --- |
| \*cxf 客户端调用需要的最少jar（3.0.7）  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/cxf-core-3.0.7.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/cxf-rt-databinding-jaxb-3.0.7.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/cxf-rt-bindings-soap-3.0.7.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/cxf-rt-frontend-jaxws-3.0.7.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/cxf-rt-frontend-simple-3.0.7.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/cxf-rt-transports-http-3.0.7.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/cxf-rt-wsdl-3.0.7.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/xmlschema-core-2.2.1.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/wsdl4j-1.6.3.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/stax2-api-3.1.4.jar"/>  <classpathentry kind="lib" path="/home/smart/soft/cxf/apache-cxf-3.0.7/lib/woodstox-core-asl-4.4.1.jar"/>  \*/ |

### cxf服务端发布

#### jax-ws 方式/endpoint方式

|  |
| --- |
| // 1、jax-ws 方式/endpoint方式 ，与cxf无关  String url = "http://192.168.5.143:8080/smart/ws/hello";  Endpoint.*publish*(url, **new** HelloWorldServiceImpl());  System.***out***.println("soap ws is published"); |

#### cxf 方式来发布

|  |
| --- |
| JaxWsServerFactoryBean factory = **new** JaxWsServerFactoryBean();  factory.setAddress("http://192.168.5.143:8080/smart/ws/hello");  factory.setServiceClass(HelloWorldService.**class**);  factory.setServiceBean(**new** HelloWorldServiceImpl());  factory.create();  System.***out***.println("soap ws is published"); |

#### CXF Simple server 方式来发布

|  |
| --- |
| //该方式无需在类中指定相关协议，也就是说不需要在类中指定相关webservice注解，一个普通类就行  ServerFactoryBean factory = **new** ServerFactoryBean();  factory.setAddress("http://192.168.5.143:8080/smart/ws/hello");  factory.setServiceClass(HelloWorldService.**class**);  factory.setServiceBean(**new** HelloWorldServiceImpl());  factory.create();  System.***out***.println("soap ws is published"); |

#### 借助容器发布

Web.xml配置

|  |
| --- |
| <servlet>  <servlet-name>cxf</servlet-name>  <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>cxf</servlet-name>  <url-pattern>/ws/\*</url-pattern>  </servlet-mapping> |

Spring.xml配置

|  |
| --- |
| <import resource=*"classpath:META-INF/cxf/cxf.xml"*/>  <import resource=*"classpath:META-INF/cxf/cxf-servlet.xml"*/>  <!-- cxf 方式  <jaxws:server id="helloService" address="/soap/hello">  <jaxws:serviceBean>  <ref bean="helloServiceImpl"/>  </jaxws:serviceBean>  </jaxws:server>  -->  <!-- CXF Simple FrontEnd 方式  <simple:server id="helloService" serviceClass="#helloService" address="/soap/hello">  <simple:serviceBean>  <ref bean="#helloServiceImpl"/>  </simple:serviceBean>  </simple:server>  -->    <!-- endpoint 方式发布 -->  <jaxws:endpoint id=*"helloWorldService"* implementor=*"#helloWorldServiceImpl"* address=*"/hello"*/> |

### cxf客户端调用

|  |
| --- |
| //1、标准的JAX-WS的API完成客户端调用---需调用借助wsdl2java工具生成的部分java文件，  //注：此处http://smart.wdlt.com/ws/demo是服务的命名空间，可查看wsdl文件中的targetNamespace  // QName qName = new QName("http://smart.wdlt.com/ws/demo","HelloWorldServiceImplService");  // HelloWorldServiceImplService ImplService =new HelloWorldServiceImplService(new URL("http://192.168.5.143:8080/smart/ws/hello?wsdl"),qName);  // HelloWorldService port=(HelloWorldService)ImplService.getPort(HelloWorldService.class);  // System.out.println(port.hello("张三"));    //2、javax.xml.ws.Service-----需调用wsdl2java工具生成的部分java文件，  // String endPointAddress = "http://192.168.5.143:8080/smart/ws/hello";//服务实地址 非wsdl地址  // //qname：service name （参见HelloWorldService）  // javax.xml.ws.Service service = javax.xml.ws.Service.create(new QName("http://smart.wdlt.com/ws/demo","HelloWorldServiceService1"));  // //qname：port name(这里即为HelloWorldServicePort)  // service.addPort(new QName("http://smart.wdlt.com/ws/demo","HelloWorldServicePort"),javax.xml.ws.soap.SOAPBinding.SOAP11HTTP\_BINDING, endPointAddress);  // HelloWorldService port =service.getPort(HelloWorldService.class);  // System.out.println(port.hello("张三"));      //3、使用CXF中JaxWsProxyFactoryBean客户端代理工厂调用web服务-----需调用wsdl2java工具生成的部分java文件，  // JaxWsProxyFactoryBean clientFactory = new JaxWsProxyFactoryBean();  // clientFactory.setAddress("http://192.168.5.143:8080/smart/ws/hello");  // clientFactory.setServiceClass(HelloWorldService.class);  // Object o = clientFactory.create();  // HelloWorldService service = (HelloWorldService)o;  // System.out.println(service.hello("张三"));    // 4、使用CXF的JaxWsDynamicClientFactory动态代理工厂---无需写服务端接口，无需配置文件    JaxWsDynamicClientFactory clientFactory =  JaxWsDynamicClientFactory.*newInstance*();  //clientFactory  Client client =clientFactory.createClient("http://192.168.5.28:8080/smart/ws/score?wsdl");  //client.getOutFaultInterceptors().add(e);//增加安全验证  Object result[] = client.invoke("submitIn","889","order");  System.***out***.println(result[0]);    // 5、spring 配置文件 调用---需写接口，需配置文件  // ApplicationContext ctx = new ClassPathXmlApplicationContext("spring/spring-cxf\_client.xml");  // HelloWorldService hello = (HelloWorldService) ctx.getBean("helloClient");  // String str = hello.hello("张三");  // System.err.println(str);  // String str1 = hello.hello1("张三");  // System.err.println(str1); |

### cxf工具

|  |
| --- |
| /\*\*  \* cxf wsdl2java  \* cd ～/soft/cxf  \* ./apache-cxf-3.0.7/bin/wsdl2java -p com.wdlt.smart.ws -d ./wsdl2java/ -all <http://192.168.5.143:8080/smart/ws/hello>? wsdl  \*/ |

### cxf安全性

目前流行的做法有两种：

|  |
| --- |
| 1. 将用户名和密码做为入参; 2. 利用拦截器封装用户名和密码； |

第1种做法，应该使用范围更广，原因是对客户端的要求，仅需增加两个入参；第二种做法，不仅服务端需要增加拦截器配置，客户端也是需要的；

如何保证密码不泄漏，我们知道，如果明文传输密码，可能在传输过程中被截获一致泄漏，为了增加密码的安全性，我们可以将密码进行加密，例如使用md5等（当然这仅是一定程度上的）。

#### 入参实现安全性

这没什么好说的，此处略。

#### 拦截器实现安全性

CXF中的拦截器分为in拦截器和out拦截器，又有客户端拦截器和服务端拦截器。

拦截器使用流程：客户端（out）-> 服务端（in）->处理业务->服务端（out）->客户端（in），并不是每一步都需要拦截器。在这里我们用到的是客户端Out拦截 器和服务端in拦截器。服务端in拦截器检查用户级权限，客户端out浏览器发送用户信息给服务端。

创建服务端验证：

JaxWsServerFactoryBean或Endpoint都可以通过getInInterceptors方法，向WebService服务添加拦截器。

##### 服务端解析帐号/密码进行验证

|  |
| --- |
| public class AuthInterceptor extends AbstractPhaseInterceptor<SoapMessage> {    * //在调用之前拦截 * public AuthInterceptor() { * super(Phase.PRE\_INVOKE); * } * /\*\* * \* 自定义拦截器需要实现handleMessage方法，该方法抛出Fault异常，可以自定义异常集成自Fault， * \* 也可以new Fault(new Throwable()) * \*/ * public void handleMessage(SoapMessage soap) throws Fault { * System.out.println("开始验证用户信息"); * List<Header> headers = soap.getHeaders(); * //检查headers是否存在 * if(headers == null | headers.size()<1){ * throw new Fault(new IllegalArgumentException("找不到Header，无法验证用户信息")); * } * Header header = headers.get(0); * Element el = (Element)header.getObject(); * NodeList users = el.getElementsByTagName("username"); * NodeList passwords = el.getElementsByTagName("password"); * //检查是否有用户名和密码元素 * if(users.getLength()<1){ * throw new Fault(new IllegalArgumentException("找不到用户信息")); * } * String username = users.item(0).getTextContent().trim(); * if(passwords.getLength()<1){ * throw new Fault(new IllegalArgumentException("找不到密码信息")); * } * String password = passwords.item(0).getTextContent(); * //检查用户名和密码是否正确 * if(!"admin".equals(username) || !"admin".equals(password)){ * throw new Fault(new IllegalArgumentException("用户名或密码不正确")); * }else{ * System.out.println("用户名密码正确允许访问"); * } * } * } |

##### 客户端追加帐号/密码入soap message中

|  |
| --- |
| public class ClientLoginInterceptor extends AbstractPhaseInterceptor<SoapMessage> {    * private String username; * private String password; * public void setUsername(String username) { * this.username = username; * } * public void setPassword(String password) { * this.password = password; * } * /\*\* * \* 创建一个新的实例 ClientLoginInterceptor. * \* * \* @param username * \* @param password * \*/ * public ClientLoginInterceptor(String username, String password) { * super(Phase.PREPARE\_SEND); * this.username = username; * this.password = password; * } * /\* (non-Javadoc) * \* @see org.apache.cxf.interceptor.Interceptor#handleMessage(org.apache.cxf.message.Message) * \*/ * public void handleMessage(SoapMessage soap) throws Fault { * // TODO Auto-generated method stub * List<Header> headers = soap.getHeaders(); * Document doc = DOMUtils.createDocument(); * Element auth = doc.createElement("authrity"); * Element username = doc.createElement("username"); * Element password = doc.createElement("password"); * username.setTextContent(this.username); * password.setTextContent(this.password); * auth.appendChild(username); * auth.appendChild(password); * //doc.appendChild(auth); * headers.add(0, new Header(new QName("tiamaes"),auth)); * } * } |

# 事物

1 如果webservice是查询，和你的提交回滚没有关系  
  
2 如果webservice是增删改事务操作，先做你的数据操作，根据异常与否决定是否调用webservice  
  
3 如果webservice既是增删改事务又必须等待它的返回数据，才能执行你的数据操作，那么强制把你的操作数据记录入日志，即使一次数据操作失败，也要在后面把操作补上。