Web Service 描述语言 WSDL 详解

为什么使用 WSDL?

像 Internet 协议之类的标准有没有为权威所利用，或者人们这样看待它是因为顺之所获的好处远远超出了代价？曾经有许多试图建立的标准都流产了。有时候，那些还没有普遍使用的标准甚至由法令或政府规定强行推出：Ada 语言就是一例。

我相信正是跟随标准所带来的好处使它广泛接受。例如，对于铁路服务来说，真正重要的是，不同公司所铺设的铁路结合到一起，或者是来自好几个公司的产品协调的工作在一起。几家大的企业合力建立了 SOAP 标准。Web Service 描述语言(WSDL)向这种 Web Service 的提供商和用户推出了方便的协调工作的方法，使我们能更容易的获得 SOAP 的种种好处。几家公司的铁道并在一起不算什么难事，他们所需遵循的只是两轨间的标准距离。对 Web Service 来说，这要复杂得多。我们必须先制定出指定接口的标准格式。

曾经有人说 SOAP 并不真需要什么接口描述语言。如果 SOAP 是交流纯内容的标准，那就需要一种语言来描述内容。SOAP 消息确实带有某些类型信息，因此 SOAP 允许动态的决定类型。但不知道一个函数的函数名、参数的个数和各自类型，怎么可能去调用这个函数呢？没有 WSDL，我可以从必备文档中确定调用语法，或者检查消息。随便何种方法，都必须有人参与，这个过程可能会有错。而使用了 WSDL，我就可以通过这种跨平台和跨语言的方法使 Web Service 代理的产生自动化。就像 COM 和 CORBA 的 IDL 文件，WSDL 文件由客户和服务器约定。

注意由于 WSDL 设计成可以绑定除 SOAP 以外的其他协议，这里我们主要关注 WSDL 在 HTTP 上和 SOAP 的关系。同样，由于 SOAP 目前主要用来调用远程的过程和函数，WSDL 支持 SOAP 传输的文档规范。WSDL 1.1 已经作为记录递交给 W3C（见 http://www.w3.org/TR/wsdl.html）

WSDL 文档结构

若要理解 XML 文档，将之看作块状图表非常有用。下图以 XML 的文档形式说明了 WSDL 的结构，它揭示了 WSDL 文档五个栏之间的关系。

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，截然不同的网站都可以实现。随网站而异的东西如序列化便归入底部分，因为它包含具体的定义。

I 抽象定义

Types

独立与机器和语言的类型定义

Messages

包括函数参数（输入与输出分开）或文档描述

PortTypes

引用消息部分中消息定义来描述函数签名（操作名、输入参数、输出参数）

2 具体定义

Bindings

PortTypes 部分的每一操作在此绑定实现

Services

确定每一绑定的端口地址

下面的图中，箭头连接符代表文档不同栏之间的关系。点和箭头代表了引用或使用关系。双箭头代表"修改"关系。3-D 的箭头代表了包含关系。这样，各 Messages 栏使用 Types 栏的定义，PortTypes 栏使用 Messages 栏的定义；Bindings 栏引用了 PortTypes 栏，Services 栏引用 Bindings 栏，PortTypes 和 Bindings 栏包含了 operation 元素，而 Services 栏包含了 port 元素。PortTypes 栏里的 operation 元素由 Bindings 栏里的 operation 元素进一步修改或描述。

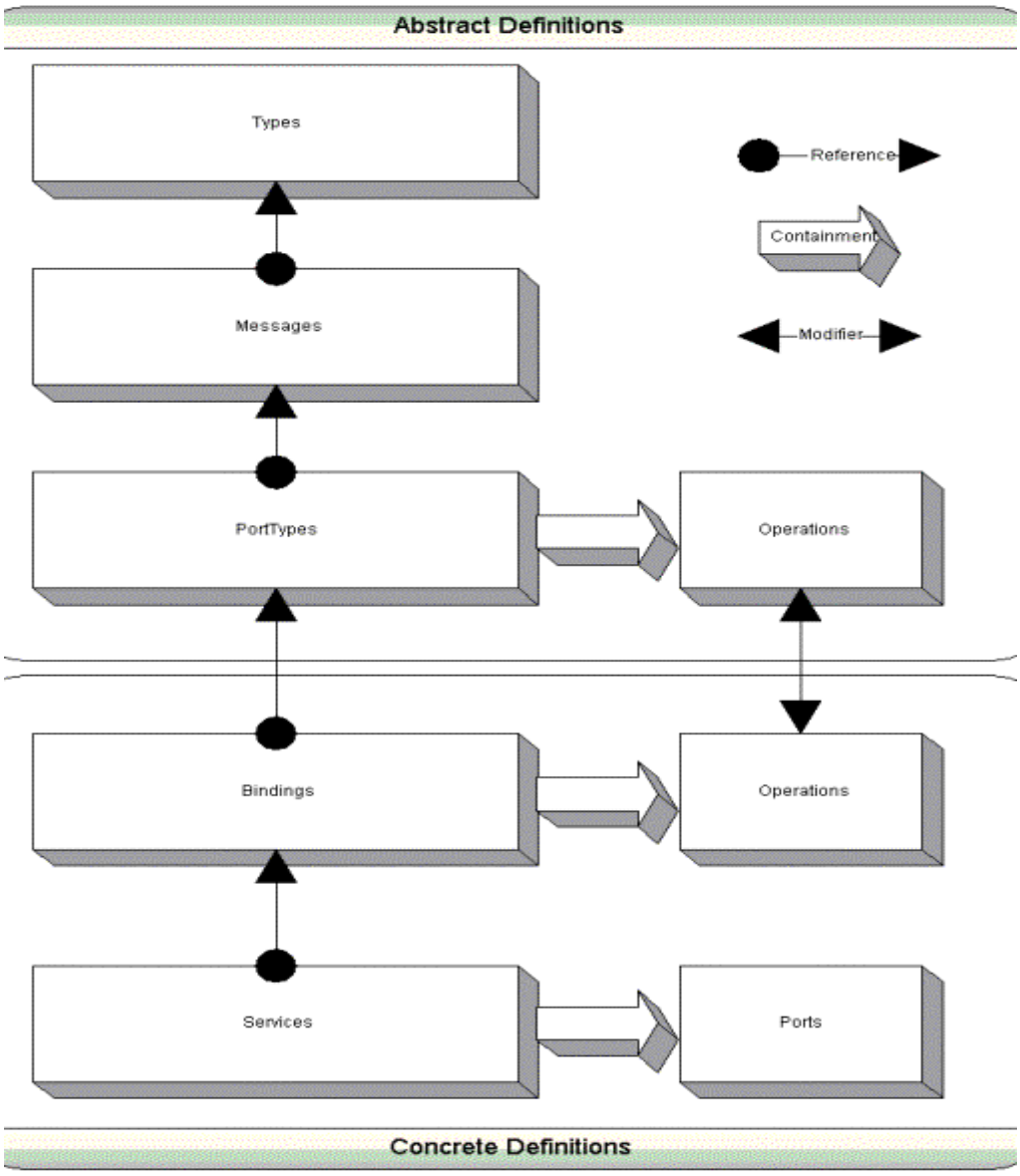
在此背景中，我将使用标准的 XML 术语来描述 WSDL 文档。Element 是指 XML 的元素，而"attribute"指元素的属性。于是：

```
＜element attribute="attribute-value"＞contents＜/element＞
```

内容也可能由一个或多个元素以递归的方式组成。根元素是所有元素之中最高级的

元素。子元素总是从属于另一个元素，父元素。

注意，文档之中可能只有一个 Types 栏，或根本没有。所有其他的栏可以只有零元素、单元素或是多元素。WSDL 的列表要求所有的栏以固定的顺序出现：import, types, message, portType, binding, service。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。



图一：抽象定义和具体定义

WSDL 文件示例

让我们来研究一下 WSDL 文件，看看它的结构，以及如何工作。请注意这是一个非常简单的 WSDL 文档实例。我们的意图只是说明它最显著的特征。以下的内容中包括更加详细的讨论。

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="FooSample"
  targetNamespace="http://tempuri.org/wsdl/"
  xmlns:wsdlns="http://tempuri.org/wsdl/"
  xmlns:typens="http://tempuri.org/xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:stk="http://schemas.microsoft.com/soap-toolkit/wsdl-extension"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
<schema targetNamespace="http://tempuri.org/xsd"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      elementFormDefault="qualified"  >
</schema>
</types>

<message name="Simple.foo">
  <part name="arg" type="xsd:int"/>
</message>

<message name="Simple.fooResponse">
  <part name="result" type="xsd:int"/>
</message>

<portType name="SimplePortType">
  <operation name="foo" parameterOrder="arg"  >
    <input message="wsdlns:Simple.foo"/>
    <output message="wsdlns:Simple.fooResponse"/>
  </operation>
</portType>

<binding name="SimpleBinding" type="wsdlns:SimplePortType">
  <stk:binding preferredEncoding="UTF-8" />
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="foo">
    <soap:operation soapAction="http://tempuri.org/action/Simple.foo"/>
    <input>
      <soap:body use="encoded" namespace="http://tempuri.org/message/"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output>
      <soap:body use="encoded" namespace="http://tempuri.org/message/"
```

```
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
</binding>

<service name="FOOSAMPLEService">
    <port name="SimplePort" binding="wsdlns:SimpleBinding">
      <soap:address location="http://carlos:8080/FooSample/FooSample.asp"/>
    </port>
</service>
</definitions>
```

以下是该实例文档的总述：稍后我将详细讨论每一部分的细节。

第一行申明该文档是 XML。尽管这并不是必需的，但它有助于 XML 解析器决定是否解析 WSDL 文件或只是报错。第二行是 WSDL 文档的根元素：<definitions>。一些属性附属于根元素，就像<schema>子元素对于<types>元素。

<types>元素包含了 Types 栏。如果没有需要声明的数据类型，这栏可以缺省。在 WSDL 范例中，没有应用程序特定的 types 声明，但我仍然使用了 Types 栏，只是为了声明 schema namespaces。

<message>元素包含了 Messages 栏。如果我们把操作看作函数，<message>元素定义了那个函数的参数。<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，就像以前所用的"fooResponse"。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。

用于交换文档时，WSDL 允许使用<message>元素来描述交换的文档。

<part>元素的类型可以是 XSD 基类型，也可以是 SOAP 定义类型(soapenc)、WSDL 定义类型(wsdl)或是 Types 栏定义的类型。

一个 PortTypes 栏中，可以有零个、单个或多个<portType>元素。由于抽象 PortType 定义可以放置在分开的文件中，在某个 WSDL 文件中没有<portType>元素是可能的。上面的例子里只是用了一个<portType>元素。而一个<portType>元素可在<operation>元素中定义一个或是多个操作。示例仅使用了一个名为"foo"的<operation>元素。这和某个函数名相同。<operation>元素可以有一个、两个、三个子元素：<input>, <output> 和<fault>元素。每个<input>和<output>元素中的消息都引用 Message 栏中的相关的<message>元素。这样，示例中的整个<portType>元素就和以下的 C 函数等效：

```
int foo(int arg);
```

这个例子足见 XML 和 C 相比要冗长的多。（包括＜message＞元素，XML 在示例中共使用了 12 行代码来表达相同的单行函数声明。）

Bindings 栏可以有零个、一个或者多个＜binding＞元素。它的意图是制定每个＜operation＞通过网络调用和回应。Services 栏同样可以有零个、一个、多个＜service＞元素。它还包含了＜port＞元素，每个＜port＞元素引用一个 Bindings 栏里的＜binding＞元素。Bindings 和 Services 栏都包含 WSDL 文档。

Namespace

＜definitions＞和子节点＜schema＞都是 namespace 属性：

```
＜definitions name="FooSample"
  targetNamespace="http://tempuri.org/wsdl/"
  xmlns:wsdlns="http://tempuri.org/wsdl/"
  xmlns:typens="http://tempuri.org/xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:stk="http://schemas.microsoft.com/soap-toolkit/wsdl-extension"
  xmlns="http://schemas.xmlsoap.org/wsdl/"＞

＜types＞
  ＜schema targetNamespace="http://tempuri.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    elementFormDefault="qualified" ＞
  ＜/schema＞
＜/types＞
```

每个 namespace 属性都声明了一个缩略语，用在文档中。例如"xmlns:xsd"就为 http://www.w3.org/2001/XMLSchema 定义了一个缩略语（xsd）。这就允许对该 namespace 的引用只需简单的在名字前加上前缀就可以了，如："xsd:int"中的"xsd"就是合法的类型名。普通范围规则可运用于缩略前缀。也就是说，前缀所定义的元素只在元素中有效。

Namespace 派什么用？namespace 的作用是要避免命名冲突。如果我建立一项 Web Service，其中的 WSDL 文件包含一个名为"foo"的元素，而你想要使用我的服务与另一项服务连接作为补充，这样的话另一项服务的 WSDL 文件就不能包含名为"foo"的元素。两个服务器程序只有在它们在两个事例中表示完全相同的东西时，才可以取相同的名字。如果有了表示区别的 namespace，我的网络服务里的"foo"就可以表示完全不同于另一个网络服务里"foo"的含义。在你的客户端里，你只要加以限制就可以引用我的"foo"。

见下例：http://www.infotects.com/fooService#foo 就是完全限制的名字，相当于

"carlos:foo"，如果我声明了 carlos 作为 http://www.infotects.com/fooService 的快捷方式。请注意 namespace 中的 URL 是用来确定它们的唯一性的，同时也便于定位。URL 所指向的地方不必是实际存在的网络地址，也可以使用 GUID 来代替或补充 URL。例如，GUID"335DB901-D44A-11D4-A96E-0080AD76435D"就是一个合法的 namespace 指派。

targetNamespace 属性声明了一个 namespace，元素中所有的声明的名字都列于其内。在 WSDL 示例中，＜definitions＞的 targetNamespace 是 http://tempuri.org/wsdl。这意味着所有在 WSDL 文档中声明的名字都属于这个 namespace。＜schema＞元素有自己的 targetNamespace 属性，其值为 http://tempuri.org/xsd ，在＜schma＞元素中定义的所有名字都属于这个 namespace 而不是 main 的 target namespace。

＜schema＞元素的以下这行声明了默认的 namespace。Schema 中所有有效的名字都属于这个 namespace。

```
xmlns="http://www.w3.org/2001/XMLSchema"
```

SOAP 消息

对于使用 WSDL 的客户机和服务机来说，研究 WSDL 文件的一种方法就是决定什么来接受所发送的信息。尽管 SOAP 使用底层协议，如 IP 和 HTTP 等，但应用程序决定了服务器与客户机之间交互的高级协议。也就是说，进行一项操作，比如"echoint"把输入的整数送回，参数的数目、每个参数的类型、以及参数如何传送等因素决定了应用程序特定的协议。有很多方法可以确定此类协议，但我相信最好的方法就是使用 WSDL。如果我们用这种视角来看待它，WSDL 不只是一种接口协议，而且是一种协议特定的语言。它就是我们超越"固定"协议（IP、HTTP 等）所需要的应用程序特定协议。

WSDL 可以确定 SOAP 消息是否遵从 RPC 或文档风格。RPC 风格的消息（就是示例中所用的）看起来像是函数调用。而文档风格的消息则更普通，嵌套层次更小。下面的 XML 消息就是示例 WSDL 文件解析后的发送/接受效果，解析使用的是 MS SOAP Toolkit 2.0（MSTK2）中的 SoapClient 对象。

从客户端调用"foo(5131953)"函数：

```
＜?xml version="1.0" encoding="UTF-8" standalone="no"?＞
  ＜SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"＞
  ＜SOAP-ENV:Body＞
    ＜m:foo xmlns:m="http://tempuri.org/message/"＞
      ＜arg＞5131953＜/arg＞
    ＜/m:foo＞
  ＜/SOAP-ENV:Body＞
   ＜/SOAP-ENV:Envelope＞
  从服务器接受的信息：
    ＜?xml version="1.0" encoding="UTF-8" standalone="no"?＞
```

```
＜SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"＞
＜SOAP-ENV:Body＞
＜SOAPSDK1:fooResponse xmlns:SOAPSDK1="http://tempuri.org/message/"＞
＜result＞5131953＜/result＞
＜/SOAPSDK1:fooResponse＞
＜/SOAP-ENV:Body＞
＜/SOAP-ENV:Envelope＞
```

两函数都调用了消息，其回应是有效的 XML。SOAP 消息由几部分组成，首先是＜Envelop＞元素，包含一个可选的＜Header＞元素以及至少一个＜body＞元素。Rpc 函数所调用的消息体有一个根据操作"foo"命名的元素，而回应信息体有一个"fooResponse"元素。Foo 元素有一个部分＜arg＞，就和 WSDL 中描述的一样，是单参数的。fooResponse 也相应的有一个＜result＞的部分。注意 encodingStyle、envelope 和 message 的 namespace 和 WSDL Bindings 栏中的预定义的一致，重复如下：

```
＜binding name="SimpleBinding" type="wsdlns:SimplePortType"＞
＜stk:binding preferredEncoding="UTF-8" /＞
＜soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/＞
＜operation name="foo"＞
＜soap:operation
soapAction="http://tempuri.org/action/Simple.foo"/＞
＜input＞
＜soap:body use="encoded"
namespace="http://tempuri.org/message/"
encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/" /＞
＜/input＞
＜output＞
＜soap:body use="encoded"
namespace="http://tempuri.org/message/"
encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/" /＞
＜/output＞
＜/operation＞
＜/binding＞
```

WSDL 的 Types 栏和 Messages 栏中的 XML Schema

WSDL 数据类型是基于"XML Schema: Datatypes"(XSD)的，现在已经被 W3C 推荐。这一文档共有三个版本（1999，2000/10，2001），因此必须在 namespace 属性的＜definitions＞元素中指明所使用的是哪一个版本。

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

在本文中，我将只考虑 2001 版本。WSDL 标准的推荐者强烈建议使用 2001 版。

在本栏和以后各部分，需使用以下简缩或前缀

前缀	代表的 Namespace	描述
Soapenc	http://schemas.xmlsoap.org/soap/encoding	SOAP 1.1 encoding
Wsdl	http://schemas.xmlsoap.org/wsdl/soap	WSDL 1.1
Xsd	http://www.w3.org/2001/XMLSchema	XML Schema

XSD 基类型

下表是直接从 MSTK2 文档中取出的，列举了 MSTK2 所支持的所有 XSD 基类型。它也告诉在客户端或服务器端的 WSDL 读取程序如何把 XSD 类型映射到在 VB、C++ 和 IDL 中相应的类型。

XSD (Soap)类型	变量类型	VB	C++	IDL	Comments
anyURI	VT_BSTR	String	BSTR	BSTR	
base64Binary	VT_ARRAY | VT_UI1	Byte()	SAFEARRAY	SAFEARRAY(unsigned char)	
Boolean	VT_BOOL	Boolean	VARIANT_BOOL	VARIANT_BOOL	
Byte	VT_I2	Integer	short	short	转换时验证范围有效性
Date	VT_DATE	Date	DATE	DATE	时间设为 oo:oo:oo
DateTime	VT_DATE	Date	DATE	DATE	
Double	VT_R8	Double	double	double	
Duration	VT_BSTR	String	BSTR	BSTR	不转换和生效
ENTITIES	VT_BSTR	String	BSTR	BSTR	不转换和生效
ENTITY	VT_BSTR	String	BSTR	BSTR	不转换和生效
Float	VT_R4	Single	float	float	
GDay	VT_BSTR	String	BSTR	BSTR	不转换和生效
GMonth	VT_BSTR	String	BSTR	BSTR	不转换和

					生效
GMonthDay	VT_BSTR	String	BSTR	BSTR	不转换和生效
GYear	VT_BSTR	String	BSTR	BSTR	不转换和生效
GYearMonth	VT_BSTR	String	BSTR	BSTR	不转换和生效
ID	VT_BSTR	String	BSTR	BSTR	不转换和生效
IDREF	VT_BSTR	String	BSTR	BSTR	不转换和生效
IDREFS	VT_BSTR	String	BSTR	BSTR	不转换和生效
Int	VT_I4	Long	long	long	
Integer	VT_DECIMAL	Variant	DECIMAL	DECIMAL	转换时范围生效
Language	VT_BSTR	String	BSTR	BSTR	不转换和生效
Long	VT_DECIMAL	Variant	DECIMAL	DECIMAL	转换时范围生效
Name	VT_BSTR	String	BSTR	BSTR	不转换和生效
NCName	VT_BSTR	String	BSTR	BSTR	不转换和生效
negativeInteger	VT_DECIMAL	Variant	DECIMAL	DECIMAL	转换时范围生效
NMTOKEN	VT_BSTR	String	BSTR	BSTR	不转换和生效
NMTOKENS	VT_BSTR	String	BSTR	BSTR	不转换和生效
nonNegativeIntege	VT_DECIMAL	Variant	DECIMAL	DECIMAL	转换时范围生效
nonPositiveInteger	VT_DECIMAL	Variant	DECIMA	DECIMAL	转换时范围生效
normalizedString	VT_BSTR	String	BSTR	BSTR	
NOTATION	VT_BSTR	String	BSTR	BSTR	不转换和生效
Number	VT_DECIMAL	Variant	DECIMAL	DECIMAL	
positiveInteger	VT_DECIMAL	Variant	DECIMAL	DECIMAL	转换时范围生效

Qname	VT_BSTR	String	BSTR	BSTR	不转换和生效
Short	VT_I2	Integer	short	short	
String	VT_BSTR	String	BSTR	BSTR	
Time	VT_DATE	Date	DATE	DATE	日设为 1899 年 12 月 30 日
Token	VT_BSTR	String	BSTR	BSTR	不转换和生效
unsignedByte	VT_UI1	Byte	unsigned char	unsigned char	
UnsignedInt	VT_DECIMAL	Variant	DECIMAL	DECIMAL	转换时范围生效
unsignedLong	VT_DECIMAL	Variant	DECIMAL	DECIMAL	转换时范围生效
unsignedShort	VT_UI4	Long	Long	Long	转换时范围生效

XSD 定义了两套内建的数据类型：原始的和派生的。在下文中查阅内建数据类型的层次十分有益：

```
http://www.w3.org/TR/2001/PR-xmlschema-2-20010330
```

complex 类型

XML schema 允许 complex 类型的定义，就像 C 里是 struct。例如，为了定义类似如下的 C 的 struct 类型：

```
typedef struct {
   string firstName;
   string lastName;
   long ageInYears;
   float weightInLbs;
   float heightInInches;
} PERSON;
```

我们可以写 XML schema：

```
＜xsd:complexType name="PERSON"＞
＜xsd:sequence＞
   ＜xsd:element name="firstName" type="xsd:string"/＞
   ＜xsd:element name="lastName" type="xsd:string"/＞
   ＜xsd:element name="ageInYears" type="xsd:int"/＞
   ＜xsd:element name="weightInLbs" type="xsd:float"/＞
   ＜xsd:element name="heightInInches" type="xsd:float"/＞
```

```
</xsd:sequence>
</xsd:complexType>
```

不过，complex 类型可以表达比 struct 更多的信息。除了<sequence>以外，它还可以有其他的子元素，比如<all>

```
<xsd:complexType name="PERSON">
<xsd:all>
  <xsd:element name="firstName" type="xsd:string"/>
  <xsd:element name="lastName" type="xsd:string"/>
  <xsd:element name="ageInYears" type="xsd:int"/>
  <xsd:element name="weightInLbs" type="xsd:float"/>
  <xsd:element name="heightInInches" type="xsd:float"/>
</xsd:all>
</xsd:complexType>
```

这意味着<element>的成员变量可以以任何顺序排列，每一个都是可选的。这和 C 中的 struct 类型不太一样。

注意内建数据类型 string, int, float。C 的 string 也是 XML 的 string，float 也类似。但 C 中的 long 类型在 XML 中是 int（上表中）。

在 WSDL 文件中，像上面的 complex 类型可以在 Types 栏声明。例如，我可以用以下方式声明 PERSON 类型并用在 Messages 栏。

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions … >
<types>
<schema targetNamespace="someNamespace"
xmlns:typens="someNamespace" >
<xsd:complexType name="PERSON">
<xsd:sequence>
  <xsd:element name="firstName" type="xsd:string"/>
  <xsd:element name="lastName" type="xsd:string"/>
  <xsd:element name="ageInYears" type="xsd:int"/>
  <xsd:element name="weightInLbs" type="xsd:float"/>
  <xsd:element name="heightInInches" type="xsd:float"/>
</xsd:sequence>
</xsd:complexType>
</schema>
</types>

<message name="addPerson">
  <part name="person" type="typens:PERSON"/>
```

```
</message>

<message name="addPersonResponse">
  <part name="result" type="xsd:int"/>
</message>

</definitions>
```

上例中第一个消息由"adperson"，并且有一个<part>，其类型为"PERSON"。PERSON 类型是在 Types 栏声明的。

如果我们使用完整的 WSDL 文件包含以上的部分，并以之初始化 MSTK2 SoapClient，它将成功的解析该文件。当然，它不会去调用<addPerson>。这是因为 SoapClient 本身并不知道如何处理 complex 类型，它需要定制类型映射来处理 complex 类型。MSTK2 文档中有包含定制类型映射的示例。

还有另一种方法可以把<part>元素联系到类型声明。这就是使用元素。下例中我将 Types 栏中声明两个元素("Person"和"Gendr")，然后我将在"addPerson"<message>中使用元素属性来引用它们。

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions … >
<types>
<schema targetNamespace="someNamespace"
  xmlns:typens="someNamespace" >
<element name="Person">
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="firstName" type="xsd:string"/>
    <xsd:element name="lastName" type="xsd:string"/>
    <xsd:element name="ageInYears" type="xsd:int"/>
    <xsd:element name="weightInLbs" type="xsd:float"/>
    <xsd:element name="heightInInches" type="xsd:float"/>
  </xsd:sequence>
</xsd:complexType>
</element>
<element name="Gender">
<xsd:simpleType>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Male" />
    <xsd:enumeration value="Female" />
  </xsd:restriction>
</xsd:simpleType>
</element>
```

```
</schema>
</types>

<message name="addPerson">
  <part name="who" element="typens:Person"/>
  <part name="sex" element="typens:Gender"/>
</message>

<message name="addPersonResponse">
  <part name="result" type="xsd:int"/>
</message>
</definitions>
```

Types 栏中的 Gender<element>里嵌入了枚举类型，其枚举值为"Male""Female"。然后我又在"addPerson"<message>中通过元素属性而不是类型属性来引用它。

"元素属性"和"类型属性"在把某特定类型关联到<part>时有什么不同呢？使用元素属性，我们可以描述一个部分，它可以假定几个类型（就像变量一样），而是用类型属性我们就无法这样做。下例说明了这一点。

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions … >
<types>
<schema targetNamespace="someNamespace"
xmlns:typens="someNamespace">
<xsd:complexType name="PERSON">
  <xsd:sequence>
    <xsd:element name="firstName" type="xsd:string"/>
    <xsd:element name="lastName" type="xsd:string"/>
    <xsd:element name="ageInYears" type="xsd:int"/>
    <xsd:element name="weightInLbs" type="xsd:float"/>
    <xsd:element name="heightInInches" type="xsd:float"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="femalePerson">
<xsd:complexContent>
  <xsd:extension base="typens:PERSON" >
  <xsd:element name="favoriteLipstick" type="xsd:string" />
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="malePerson">
<xsd:complexContent>
```

```
<xsd:extension base="typens:PERSON" >
<xsd:element name="favoriteShavingLotion" type="xsd:string" />
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="maleOrFemalePerson">
<xsd:choice>
  <xsd:element name="fArg" type="typens:femalePerson" >
  <xsd:element name="mArg" type="typens:malePerson" />
</xsd:choice>
</xsd:complexType>
</schema>
</types>

<message name="addPerson">
  <part name="person" type="typens:maleOrFemalePerson"/>
</message>

<message name="addPersonResponse">
  <part name="result" type="xsd:int"/>
</message>

</definitions>
```

上例也告诉我们 extension 的派生。"femailPerson"和"malePerson"都是从"PERSON"派生出来的。它们各有一些额外的元素："femalePerson"有"favoriteLipstick"元素，"malePerson"有"favoriteShavingLotion"元素。两派生类型都归入一个 complex 类型"maleOrFemalePerson"，使用的是<choice>构造。最后，在"adperson"<message>中，新类型有"person"<part>引用。这样，参数或<part>就可以是"femalePerson"或"malePerson"了。

数组

XSD 提供<list>结构来声明一个数组，元素之间有空格界定。不过 SOAP 不是使用 XSD 来编码数组的，它定义了自己的数组类型--"SOAP-ENC: Array"。下列的例子揭示了从这一类型派生出一位整数数组的方法：

```
<xsd:complexType name="ArrayOfInt">
<xsd:complexContent>
  <xsd:restriction base="soapenc:Array">
    <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]"/>
  </xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
```

新的 complex 类型从 soapenc:array 限制派生。然后又声明了 complex 类型的一个属性。引用"soapenc:arrayType"实际上是这样完成的：

```
<xsd:attribute name="arrayType" type="xsd:string"/>
```

wsdl:arrayType 属性值决定了数组每个成员的类型。数组的成员也可以是 Complex 类型。：

```
<xsd:complexType name="ArrayOfPERSON">
<xsd:complexContent>
<xsd:restriction base="soapenc:Array">
<attribute ref="soapenc:arrayType"
wsdl:arrayType="typens:PERSON[]"/>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
```

WSDL 要求数组的类型由"ArrayOf"和每个数组元素的类型串联而成。很显然，顾名思义，"ArrayOfPERSON"是 PERSON 结构的数组。下面我将使用 ArrayOfPERSON 来声明一个<message>，并加入不止一个 PERSON：

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions … >
<types>
<schema targetNamespace="someNamespace"
xmlns:typens="someNamespace" >
<xsd:complexType name="PERSON">
  <xsd:sequence>
    <xsd:element name="firstName" type="xsd:string"/>
    <xsd:element name="lastName" type="xsd:string"/>
    <xsd:element name="ageInYears" type="xsd:int"/>
    <xsd:element name="weightInLbs" type="xsd:float"/>
    <xsd:element name="heightInInches" type="xsd:float"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ArrayOfPERSON">
<xsd:complexContent>
<xsd:restriction base="soapenc:Array">
<attribute ref="soapenc:arrayType"
wsdl:arrayType="typens:PERSON[]"/>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
```

```
</schema>
</types>

<message name="addPersons">
  <part name="person" type="typens:ArrayOfPERSON"/>
</message>

<message name="addPersonResponse">
  <part name="result" type="xsd:int"/>
</message>

</definitions>
```

<portType>和<operation>元素

PortType 定义了一些抽象的操作。PortType 中的 operation 元素定义了调用 PortType 中所有方法的语法，每一个 operation 元素声明了方法的名称、参数（使用<message>元素）和各自的类型（<part>元素要在所有<message>中声明）。

在一篇 WSDL 文档中可以有几个<PortType>元素，每一个都和一些相关操作放在一起，就和 COM 和一组操作的接口相似。

在<operation>元素中，可能会有至多一个<input>元素，一个<output>元素，以及一个<fault>元素。三个元素各有一个名字和一个消息属性。

<input>，<output>，<fault>元素属性的名字有何含义呢？它们可以用来区别两个同名操作（重载）。例如，看下面两个 C 函数：

```
void foo(int arg);
void foo(string arg);
```

这种重载在 WSDL 中可以这样表示：

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="fooDescription"
  targetNamespace="http://tempuri.org/wsdl/"
  xmlns:wsdlns="http://tempuri.org/wsdl/"
  xmlns:typens="http://tempuri.org/xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:stk="http://schemas.microsoft.com/soap-toolkit/wsdl-
  extension"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
<schema targetNamespace="http://tempuri.org/xsd"
```

```
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  elementFormDefault="qualified"  >
</schema>
</types>

<message name="foo1">
  <part name="arg" type="xsd:int"/>
</message>

<message name="foo2">
  <part name="arg" type="xsd:string"/>
</message>

<portType name="fooSamplePortType">
<operation name="foo" parameterOrder="arg "  >
  <input name="foo1" message="wsdlns:foo1"/>
</operation>
<operation name="foo" parameterOrder="arg "  >
  <input name="foo2" message="wsdlns:foo2"/>
</operation>
</portType>

<binding name="fooSampleBinding" type="wsdlns:fooSamplePortType">
<stk:binding preferredEncoding="UTF-8" />
<soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="foo">
<soap:operation soapAction="http://tempuri.org/action/foo1"/>
  <input name="foo1">
    <soap:body use="encoded" namespace="http://tempuri.org/message/"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  </input>
</operation>
<operation name="foo">
<soap:operation soapAction="http://tempuri.org/action/foo2"/>
<input name="foo2">
<soap:body use="encoded"
      namespace="http://tempuri.org/message/"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
/>
</input>
</operation>
```

```
</binding>

<service name="FOOService">
<port name="fooSamplePort" binding="fooSampleBinding">
<soap:address
    location="http://carlos:8080/fooService/foo.asp"/>
</port>
</service>
</definitions>
```

到目前为止，还没有一种 SOAP 的实现支持重载。这对基于 JAVA 的客户端十分重要，因为 JAVA 服务器使用的接口用到 JAVA 的重载特性。而对基于 COM 的客户端，就不那么重要，因为 COM 是不支持重载的。

<binding>和<operation>元素

Binding 栏是完整描述协议、序列化和编码的地方，Types, Messages 和 PortType 栏处理抽象的数据内容，而 Binding 栏是处理数据传输的物理实现。Binding 栏把前三部分的抽象定义具体化。

把相关的数据制定和消息声明分开，这意味着同一类型服务的提供者可以把一系列的操作标准化。每个提供者可以提供定制的 binding 来互相区分。WSDL 也有一个重要的结构，使抽象定义可以放在分离的文件中，而不是和 Bindings 和 Services 在一起，这样可在不同的服务提供者之间提供标准化的抽象定义，这很有帮助。例如，银行可以用 WSDL 文档来标准化一些银行的操作。每个银行仍然可以自由的订制下层的协议、串行优化，及编码。

下面是重载的 WSDL 示例 的 Binding 栏，重复在此以便讨论：

```
<binding name="fooSampleBinding" type="wsdlns:fooSamplePortType">
<stk:binding preferredEncoding="UTF-8" />
<soap:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="foo">
  <soap:operation soapAction="http://tempuri.org/action/foo1"/>
    <input name="foo1">
      <soap:body use="encoded" namespace="http://tempuri.org/message/"
                   encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
  </operation>
<operation name="foo">
<soap:operation soapAction="http://tempuri.org/action/foo2"/>
  <input name="foo2">
    <soap:body use="encoded"
                    namespace="http://tempuri.org/message/"
```

```
                encodingStyle=
                    "http://schemas.xmlsoap.org/soap/encoding/" />
  </input>
</operation>
</binding>
```

<binding>元素已经取了一个名字（本例中"fooSampleBinding"），这样就可以被 Services 栏的<port>元素引用了。它有一个"type"的属性引用<portType>，本例中就是"wsdlns:fooSamplePortType"。第二行是 MSTK2 的扩展元素<stk:binding>，它指定了 preferredEncoding 属性为"UTF-8"。

<soap:binding>元素指定了所使用的风格（"rpc"）和传输方式。Transport 属性应用了一个 namespace，正是这个 namespace 指明使用 HTTP SOAP 协议。

有两个同以"foo"命名的<operation>元素。唯一不同的是它们各自的<input>名字，分别为"foo1"和"foo2"。两个<operation>元素中的<soap:operation>元素有同样的"soapAction"属性，是 URI。soapAction 属性是 SOAP 特定的 URI，它只是简单的使用于 SOAP 消息。所产生的 SOAP 消息有一个 SOAPAction 头，而 URI 也仅在<soap:operation>元素里才起作用。soapAction 属性在 HTTP 的 binding 中是必需的，但在其他非 HTTP binding 中却不要提供。目前它的使用并不清楚，但它似乎有助于本例中的两个"foo"操作。SOAP 1.1 指明 soapAction 用来确定消息的"意图"。似乎服务器可以在不解析整个消息的情况下就能使用这一属性来发送消息。实际上，它的使用多种多样。<soap:operation>元素也可以包含另一属性，即"style"属性，在有必要冲突<soap:binding>元素指定的风格时可以使用。

<operation>属性可以包含<input>，<output> 和<fault>的元素，它们都对应于 PortType 栏中的相同元素。只有<input>元素在上例中提供。这三个元素中的每一个可有一个可选的"name"属性，在本例中，我们用这种方法来区分同名操作。在本例的<input>元素中有一个<soap:body>元素，它指定了哪些信息被写进 SOAP 消息的信息体中。该元素有以下属性：

Use

用于制定数据是"encoded"还是"literal"。"Literal"指结果 SOAP 消息包含以抽象定义（Types, Messages, 和 PortTypes）指定格式存在的数据。"Encoded"指"encodingStyle"属性决定了编码方式。

Namespace

每个 SOAP 消息体可以有其自己的 namespace 来防止命名冲突。这一属性制定的 URI 在结果 SOAP 消息中逐字使用。

EncodingStyle

对 SOAP 编码，它应该有以下 URI 值：

"http://schemas.xmlsoap.org/soap/encoding"

文档风格实现

在前几栏中，＜soap:binding＞元素有一个类型属性，设为"rpc"。此属性设为"document"时会改变传输时消息的串行化。不同于函数签名，现在的消息是文档传输的。在这类 binding 中，＜message＞元素定义文档格式，而不是函数签名。作为例子，考虑以下 WSDL 片段：

```
＜definitions
xmlns:stns="(SchemaTNS)"
xmlns:wtns="(WsdlTNS)"
targetNamespace="(WsdlTNS)"＞

＜schema targetNamespace="(SchemaTNS)"
elementFormDefault="qualified"＞
＜element name="SimpleElement" type="xsd:int"/＞
＜element name="CompositElement" type="stns:CompositeType"/＞
＜complexType name="CompositeType"＞
＜all＞
  ＜element name='a' type="xsd:int"/＞
  ＜element name='b' type="xsd:string"/＞
＜/all＞
＜/complexType＞
＜/schema＞

＜message...＞
  ＜part name='p1' type="stns:CompositeType"/＞
  ＜part name='p2' type="xsd:int"/＞
  ＜part name='p3' element="stns:SimpleElement"/＞
  ＜part name='p4' element="stns:CompositeElement"/＞
＜/message＞
…
＜/definitions＞
```

schema 有两个元素：SimpleElement 和 CompositeElement，还有一个类型声明（CompositeType）。唯一声明的＜message＞元素有四个部分：p1：Composite 型；p2：int 型；p3：SimpleElement 型；p4：CompositeElement 型。以下有一个表，对四种类型的 use/type 决定的 binding 作一比较：rpc/literal, document/literal, rpc/encoded, 以及 document/encoded。表指明了每种 binding 的表现。

＜service＞和＜port＞元素

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。这样，我就可以方便的使用另一个＜service＞来重定向我的股市查询申请。我的客户端程序仍然工作，因为这种根据协议归类的服务不随服务而变化。多个＜service＞元素的另一个作用是根据特定的协议划分端口。例如，我可以把所有的 HTTP 端口放在同一个＜service＞中，所有的 SMTP 端口放在另一个＜service＞里。我的客户可以搜索与它可以处理的协议相匹配的＜service＞。

```
＜service name="FOOService"＞
＜port name="fooSamplePort" binding="fooSampleBinding"＞
  ＜soap:address
      location="http://carlos:8080/fooService/foo.asp"/＞
＜/port＞
＜/service＞
```

在一个 WSDL 文档中，＜service＞的 name 属性用来区分不同的 service。因为同一个 service 中可以有多个端口，它们也有"name"属性。

总结

本文中我描述了 WSDL 文档关于 SOAP 方面的最显著的特点。不过应该说明的是 WSDL 并不仅限于 HTTP 上的 SOAP。WSDL 用来描述 HTTP-POST、HTTP-GET、SMTP 及其他协议时非常清晰。使用了 WSDL，SOAP 更加容易处理了，无论是开发者还是使用者。我相信 WSDL 和 SOAP 一起将会开创网络应用程序世界的新时代。

WSDL 的 namespace 里有一系列的 XML 元素。下表概述了那些元素、它们的属性和内容。

元素	属性	内容（子元素）
＜definitions＞	name targetNamespace xmlns (other namespaces)	＜types＞ ＜message＞ ＜portType＞ ＜binding＞ ＜service＞
＜types＞	(none)	＜xsd:schema＞
＜message＞	Name	＜part＞
＜portType＞	Name	＜operation＞
＜binding＞	name type	＜operation＞
＜service＞	name	＜port＞
＜part＞	name type	(empty)
＜operation＞	name	＜input＞

	parameterOrder	＜output＞ ＜fault＞
＜input＞	name message	(empty)
＜output＞	name message	(empty)
＜fault＞	name message	(empty)
＜port＞	name binding	＜soap:address＞

资源：

1. WSDL 1.1
2. SOAP 1.1
3. XML Schema Primer
4. MS SOAP Toolkit Download Site
5. A tool for translating IDL to WSDL
6. Free Web Services resources including a WSDL to VB proxy generator
7. PocketSOAP: SOAP related components, tools & source code