XML Schema

W3C XML 架构设计模式：避免复杂性

Dare Obasanjo

Microsoft Corporation

2003年1月

适用于：

   W3C XML 架构

**摘要：**对 Kohsuke Kawaguchi 关于使用 W3C XML 架构的指导性文章进行评论，分析他最初提出的各种指导方针，并对每个主要观点提出了相同或不同的看法，或者对某些观点进行了阐述。还对在使用 W3C XML 架构时应该做什么、不应该做什么提供了其他的一些指导方针。（本文包含一些指向英文站点的链接。）

**目录**

* [简介](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic01)
* [指导方针](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic02)
  + [为何应该使用全局和局部元素声明](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic03)
  + [为何应该使用全局和局部属性声明](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic04)
  + [为何应该了解 XML 命名空间对 W3C XML 架构的影响](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic05)
  + [为何应该始终将 elementFormDefault 设置为“qualified”](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic06)
  + [为何应该使用属性组](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic07)
  + [为何应该使用模型组](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic08)
  + [为何应该使用内置的简单类型](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic09)
  + [为何应该使用复杂类型](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic10)
  + [为何不应该使用表示法声明](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic11)
  + [为何应该慎重使用替换组](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic12)
  + [为何应该使用 key/keyref/unique 而不是使用 ID/IDREF 进行标识约束](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic13)
  + [为何应该慎重使用可变架构](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic14)
  + [为何不应该使用默认值和固定值，尤其是对于 xs:QName 类型](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic15)
  + [为何应该使用简单类型的限制和扩展](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic16)
  + [为何应该使用复杂类型的扩展](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic17)
  + [为何应该慎重使用复杂类型的限制](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic18)
  + [为何应该慎重使用抽象类型](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic19)
  + [应该使用通配符以提高可扩展性](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic20)
  + [不要使用组和类型重定义](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic21)
* [小结](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic22)
* [致谢](http://www.microsoft.com/china/MSDN/library/archives/library/dnXML/html/XMLschemachange.asp#xmlscmavdcmplx_topic23)

## **简介**

在过去的一年里，许多新老架构作者一直在争论 W3C XML 架构语言的各个方面。由于 W3C XML 架构建议（[第一部分](http://www.w3.org/TR/xmlschema-1/)和[第二部分](http://www.w3.org/TR/xmlschema-2/)）内容繁多且相当复杂，对许多架构作者来说，理解并使用 W3C XML 架构提供的部分功能而不是试图理解该语言的全部细节，会得到更好的效果。

已经有一些文档试图从一般使用的角度来定义 W3C XML 架构的有效子集，最有影响力的是 [Kohsuke Kawaguchi](http://www.xml.com/pub/au/105) 提出的 [W3C XML Schema Made Simple](http://www.xml.com/pub/a/2001/06/06/schemasimple.html) 和 [公认标准委员会 (ASC) X12](http://www.x12.org/) 提出的 [X12 Reference Model for XML Design](http://www.x12.org/x12org/comments/X12Reference_Model_For_XML_Design.pdf)。但是，这两篇文章都极其保守，它们都建议不要使用 W3C XML 架构语言的强大功能，而对不使用这些功能所付出的代价没有进行充分的论述。

## **指导方针**

下面是 Kohsuke 最初提出的一些指导方针，以及一些补充建议：

* **应该**使用元素声明、属性组、模型组和简单类型。
* **应该**尽可能使用 XML 命名空间，了解使用它们的正确方法。
* **不要**试图成为 XML 架构的高手。这可能需要花几个月时间。
* **应该**（Kohsuke 最初的指导方针中建议不要）使用复杂类型和属性声明。
* **不要**使用表示法。
* **应该**（Kohsuke 最初的指导方针中建议不要）使用局部声明。
* **应该**慎重（Kohsuke 最初的指导方针中建议不要）使用替换组。
* **应该**慎重（Kohsuke 最初的指导方针中建议不要）使用没有 **targetNamespace** 属性的架构（即可变架构）。

在补充的指导方针中，对建议不要使用复杂类型、属性声明和局部声明提出了一些不同意见，并对使用可变架构和替换组两种指导方针进行了阐述，解释了它们的优缺点。

最后，除了对 Kohsuke 最初的指导方针进行补充以外，还提出了以下建议：

* **应该**使用 key/keyref/unique 而不是使用 ID/IDREF 进行标识约束。
* **不要**使用默认值或固定值，尤其是对于 **xs:QName** 的类型。
* **不要**使用类型或组重定义。
* **应该**使用简单类型的限制和扩展。
* **应该**使用复杂类型的扩展。
* **应该**慎重使用复杂类型的限制。
* **应该**慎重使用抽象类型。
* **应该**将 **elementFormDefault** 设置为 qualified，将 **attributeFormDefault** 设置为 unqualified。
* **应该**使用通配符以提高可扩展性。

如果您是初学者，除非待解决的问题必须使用标有“慎重”字眼的指导方针，否则最好避免使用这些指导方针。下面对上述建议的合理性进行解释。

### **为何应该使用全局和局部元素声明**

[元素声明](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-element)用于指定元素的结构、类型、表现形式和值约束。元素声明在架构文档中是最重要、最常用的构造块。

作为 [xs:schema](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-schema) 元素的子元素出现的元素声明被认为是“全局元素”。全局元素可以被重复使用，既可以被同一架构的其他部分引用，也可以被其他架构文档引用。全局元素的另一重要特性是：它们可以是[替换组](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/#subsGroups)的成员。由于 W3C XML 架构建议未提供指定有关被验证文档的根元素的机制，因此可以将任意全局元素用作有效文档的根元素。

未引用全局元素的[复杂类型](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-complexType)或[模型组](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-group)定义中出现的元素声明被认为是“局部声明”。与全局元素不同，只要不是在同一级中声明，在一个架构中可以有多个名称相同但类型不同的局部元素。[W3C XML Schema Primer 的 3.3 节](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/#GlobalvsLocal)给出了以下示例：

只能声明一个名为“title”的全局元素，并且该元素仅限于一种类型（例如 xs:string 或 PersonTitle）。但是，可以声明一个名为“title”的局部元素，其类型为字符串，并且是“book”的子元素。在同一架构（目标命名空间）内，还可以声明另一个也叫做“title”的元素，它具有“Mr Mrs Ms”枚举值。

当需要从目标架构或从其他架构文档重复利用某些元素时，如果元素及其相关类型可以绑定在一起以便广泛使用，则应该将这些元素声明为全局元素。而对于那些仅在声明类型的上下文中有意义且不需要重复使用的元素来说，声明为局部元素更合适一些。

默认情况下，全局元素的命名空间名称与架构的目标命名空间名称相同，而局部元素没有命名空间名称。这就意味着在默认情况下，当根据全局元素声明来验证 XML 文档中的元素时，这些元素应该具有与全局元素架构的目标命名空间相同的命名空间名称，而当根据局部元素进行验证时，应该没有命名空间名称。以下示例说明了这一点。

test.xsd  
<?xml version="1.0" encoding="UTF-8" ?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"   
 targetNamespace="http://www.example.com"  
 xmlns="http://www.example.com">  
  
 <!-- 全局元素声明验证来自 http://www.example.com 命名空间的   
 <language> 元素 -->  
 <xs:element name="language" type="xs:string" />  
 <xs:element name="Root" type="sequenceOfLanguages" />  
 <xs:element name="Root2" type="sequenceOfLanguages2" />  
   
 <!-- 带有局部元素声明的复杂类型验证不带命名空间名称的   
 <language> 元素 -->  
 <xs:complexType name="sequenceOfLanguages" >   
 <xs:sequence>  
 <xs:element name="language" type="xs:NMTOKEN" maxOccurs="unbounded" />  
 </xs:sequence>  
 </xs:complexType>  
  
 <!-- 引用了全局元素声明的复杂类型 -->  
 <xs:complexType name="sequenceOfLanguages2" >   
 <xs:sequence>  
 <xs:element ref="language" maxOccurs="10" />  
 </xs:sequence>  
 </xs:complexType>  
</xs:schema>  
  
test.xml  
<?xml version="1.0"?>  
<ex:Root xmlns:ex="http://www.example.com">  
 <language>EN</language>   
</ex:Root>   
  
test2.xml  
<?xml version="1.0"?>  
<ex:Root2 xmlns:ex="http://www.example.com">  
 <ex:language>English</ex:language>   
 <ex:language>Klingon</ex:language>   
</ex:Root2>

### **为何应该使用全局和局部属性声明**

[属性声明](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-attribute)用于指定属性的类型、可选性和默认信息。

作为 [xs:schema](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-schema) 元素的子元素出现的属性声明被认为是“全局属性”。全局属性可以被重复使用，既可以被同一架构的其他部分引用，也可以被其他架构文档引用。未引用全局属性的[复杂类型定义](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-complexType)中出现的属性声明被认为是“局部属性”。

对于那些需要从目标架构和其他架构文档重复使用的类型，应该使用全局属性声明。而对于那些仅在声明类型的上下文中有意义且不需要重复使用的属性来说，声明为局部属性更合适一些。因为属性通常紧密耦合到其父元素中，所以架构作者一般倾向于使用局部属性声明。但是，在某些场合，适用于多个命名空间中的大量元素的全局属性是非常有用的，例如 [xsi:type](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#xsi_type) 和 [xsi:schemaLocation](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#xsi_schemaLocation)。

**注意：**默认情况下，全局属性的命名空间名称与架构的目标命名空间名称相同，而局部属性没有命名空间名称。这就意味着在默认情况下，当根据全局属性声明来验证 XML 文档中的属性时，这些属性应该具有与全局属性架构的目标命名空间相同的命名空间名称，而当根据局部属性进行验证时，应该没有命名空间名称。以下示例说明了这一点。

test.xsd  
<?xml version="1.0" encoding="UTF-8" ?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"   
 targetNamespace="http://www.example.com"   
 xmlns="http://www.example.com">  
  
 <!-- 全局属性声明验证来自 http://www.example.com 命名空间的  
 language 属性 -->   
 <xs:attribute name="language" type="xs:string" />  
 <xs:element name="Root" type="sequenceOfNotes" />  
 <xs:element name="Root2" type="sequenceOfNotes2" />  
  
 <!-- 带有局部属性声明的复杂类型验证不带命名空间名称的  
 language 属性 -->  
 <xs:complexType name="sequenceOfNotes" >   
 <xs:sequence>  
 <xs:element name="Note" type="xs:string" />  
 </xs:sequence>  
 <xs:attribute name="language" type="xs:NMTOKEN" />   
 </xs:complexType>  
  
 <!-- 引用了全局属性声明的复杂类型 -->  
 <xs:complexType name="sequenceOfNotes2" >   
 <xs:sequence>  
 <xs:element name="Note" type="xs:string" />  
 </xs:sequence>  
 <xs:attribute ref="language" />  
 </xs:complexType>  
</xs:schema>  
  
  
  
test.xml  
<?xml version="1.0"?>  
<ex:Root xmlns:ex="http://www.example.com" language="EN" >  
 <Note>此处没有任何内容</Note>   
</ex:Root>   
  
test2.xml  
<?xml version="1.0"?>  
<ex:Root2 xmlns:ex="http://www.example.com" ex:language="英语  
 语言">  
 <Note>此处没有任何内容</Note>   
</ex:Root2>

### **为何应该了解 XML 命名空间对 W3C XML 架构的影响**

对 [XML 命名空间](http://www.w3.org/TR/REC-xml-names/)的支持已被紧密融合到 W3C XML 架构建议中。许多场合都需要使用命名空间，例如：

* 引用全局元素、属性或类型时
* 使用 XPath 表达式进行[标识约束](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/)时
* 确定架构中声明可以验证的元素和属性时
* 导入并包括其他架构文档时

基于这个原因，建议架构作者熟悉命名空间的工作方式以及对 W3C XML 架构的特殊影响。MSDN 文章 [XML Namespaces and How They Affect XPath and XSLT](http://www.microsoft.com/msdn/library/en-us/dnexxml/html/xml05202002.asp) 详细介绍了 XML 命名空间的细节，文章 [Working with Namespaces in XML Schema](http://www.microsoft.com/msdn/library/en-us/dnexxml/html/xml08192002.asp) 解释了命名空间对 W3C XML 架构产生的主要影响。

### **为何应该始终将 elementFormDefault 设置为“qualified”**

在前一节中，曾提到默认情况下，全局声明用于验证具有命名空间名称的元素或属性，而局部声明用于验证不具有命名空间名称的元素或属性。用来描述具有命名空间名称的元素或属性的术语是“命名空间限定”。根据局部声明是否验证命名空间限定的元素或属性，默认方式可能会被替代。[xs:schema](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/) 元素具有 **elementFormDefault** 和 **attributeFormDefault** 属性，分别用来指定架构中的局部声明是否应该验证命名空间限定的元素或属性。这两个属性的有效值是 **qualified** 和 **unqualified**，默认值都是 unqualified。

同样，局部元素和属性声明的 form 属性可用于替代 [xs:schema](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-schema) 元素上指定的 **elementFormDefault** 和 **attributeFormDefault** 属性值。这样就可以更精确地控制是根据全局声明还是根据局部声明来验证实例文档中的元素和属性。

以下示例取自 Kohsuke 文章中的“[Why You Should Avoid Local Declarations](http://www.xml.com/pub/a/2001/06/06/schemasimple.html?page=2#avoid_local)”一节，它深入分析了这些属性对验证结果的巨大影响。

架构：

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
 targetNamespace="http://example.com">  
 <xs:element name="person">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="familyName" type="xs:string" />  
 <xs:element name="firstName" type="xs:string" />  
 <xs:sequence>  
 <xs:complexType>  
 <xs:element>  
<xs:schema>

验证以下文档：

<foo:person xmlns:foo="http://example.com">  
 <familyName> KAWAGUCHI <familyName>  
 <firstName> Kohsuke <firstName>  
<foo:person>

...架构作者通常并不希望这样做，而是在无意中引入。将架构更改为：

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
 targetNamespace="http://example.com"   
 **elementFormDefault="qualified">**  
 <xs:element name="person">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="familyName" type="xs:string" />  
 <xs:element name="firstName" type="xs:string" />  
 <xs:sequence>  
 <xs:complexType>  
 <xs:element>  
<xs:schema>

...允许它验证以下文档：

<person xmlns="http://example.com">  
 <familyName> KAWAGUCHI <familyName>  
 <firstName> Kohsuke <firstName>  
<person>

—或—

<foo:person xmlns:foo="http://example.com">  
 <foo:familyName> KAWAGUCHI <foo:familyName>  
 <foo:firstName> Kohsuke <foo:firstName>  
<foo:person>

让 **attributeFormDefault** 属性取默认值 **unqualified** 是合适的，因为大多数架构作者不希望通过添加前缀来明确地对所有属性进行命名空间限定。

### **为何应该使用属性组**

[属性组定义](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-attributeGroup)是一种创建由[属性声明](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-attribute)和[属性通配符](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-anyAttribute)组成的命名集合的方法。属性组有助于架构的模块化，允许您在一处声明常用的属性集合，然后从一个或多个架构中引用。

### **注意：**Koshuke 的文章中提到属性组可以取代全局属性，这可能会给读者造成两者互相排斥的错误印象。全局声明的属性是独立的，可以重复使用。而属性组是模块化的属性集合，属性组中的属性声明可以是局部声明，也可以是对全局声明的引用。因此，Koshuke 的文章中将属性组描述为可以取代全局属性声明并不是十分准确。

**为何应该使用模型组**

[模型组定义](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#declare-namedModelGroup)是一种使用[所有](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-all)、[选项](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-choice)或[顺序](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-sequence)复合方式来创建命名元素组的机制。模型组适用于那些希望避免复杂类型（特别是类型派生）而需要重复使用某些元素的场合。但是，模型组不能取代复杂类型，因为它们不能包含属性声明，也不能指定元素声明的类型。此外，与复杂类型的派生相比，模型组的派生功能有很多限制。

### **为何应该使用内置的简单类型**

W3C XML 架构比 XML 1.0 中的 DTD 优越，主要表现在它的数据类型方面。在 W3C XML 架构中，可以将元素值或属性值指定为字符串、日期或数值数据，这使得架构作者能够以互操作和独立于平台的方式来指定并验证 XML 数据的内容。由于存在[许多内置数据类型](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#built-in-datatypes)（约 44 种），如果架构作者对内置类型的子集进行标准化以避免信息重载，这将是十分明智的。

大多数情况下，用户不需要使用 [xs:string](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#string)（如 [xs:ENTITY](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#ENTITY) 或 [xs:language](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#language)）的子类型、[xs:integer](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#integer)（如 [xs:short](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#short) 或 [xs:unsignedByte](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#unsignedByte)）的子类型以及 Gregorian 日期类型（如 [xs:gMonthDay](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#gMonthDay) 或 [xs:gYearMonth](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#gYearMonth)）。去掉上面提到的这些类型，可以将要跟踪的类型减少到可以接受的数量。

### **为何应该使用复杂类型**

[复杂类型定义](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-complexType)用于指定由元素和属性组成的内容模型。元素声明可以通过引用某个命名或匿名的复杂类型来指定其内容模型。对于命名的复杂类型来说，可以从定义它的架构或外部架构文档中通过名称对其进行引用，而匿名的复杂类型必须在使用该类型的元素声明中定义。此外，可以使用 W3C XML 架构继承机制对命名复杂类型的内容模型进行扩展或限制。

复杂类型与[模型组定义](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#declare-namedModelGroup)类似，但有两个主要区别：复杂类型定义可以在其定义的内容模型中包括属性，并可以使用类型派生，而命名模型组则不能。在 Kohsuke 的文章中，他主张[将匿名复杂类型、模型组定义和属性组结合起来使用，而不要使用命名复杂类型来指定元素的内容模型](http://www.xml.com/pub/a/2001/06/06/schemasimple.html)，试图避免处理命名复杂类型的“复杂性”。但有人提出反对意见，认为在指定元素的内容模型时，使用三种机制与使用一种机制相比实际上更容易引起混乱。因此，命名复杂类型除了允许重复使用内容模型以外，还是指定元素内容模型的最直观方法。

只有**不**需要从元素声明外部引用类型且不需要类型派生时，才应该使用匿名复杂类型。**有一点值得注意，从匿名复杂类型不能派生出新类型。**一般来说，大量使用匿名类型的架构在统一性和一致性方面很可能会出问题。

### **为何不应该使用表示法声明**

Kohsuke 曾建议[避免使用表示法声明](http://www.xml.com/pub/a/2001/06/06/schemasimple.html?page=2#avoid_notations)，这是完全正确的。它们的存在仅仅是为了与 DTD 保持向后兼容，但不能与 DTD 表示法向后兼容。最好假定它们根本就不存在。

### **为何应该慎重使用替换组**

[替换组](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/#SubsGroups)为 XML 元素提供了一种类似编程语言中的子类型多态的机制。将一个或多个元素标记为可以替代全局元素（也称为头元素），意味着替换组中的成员可以与内容模型中的头元素互换。例如，对于包含成员 USAddress 和 UKAddress 的 Address 替换组来说，可以在内容模型中使用通用元素 **Address**，也可以用 USAddress 或 UKAddress 来取代它。唯一的要求是替换组成员的类型必须相同，或者与头元素属于同一类型层次结构。

下面给出了一个示例架构及其验证的实例：

example.xsd:  
 <xs:schema   
 xmlns:xs="http://www.w3.org/2001/XMLSchema"  
 targetNamespace="http://www.example.com"  
 xmlns:ex="http://www.example.com"  
 elementFormDefault="qualified">  
  
 <xs:element name="book" type="xs:string" />  
  
 <xs:element name="magazine" type="xs:string" substitutionGroup="ex:book" />  
  
 <xs:element name="library">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element ref="ex:book" maxOccurs="unbounded"/>  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
  
  
</xs:schema>  
example.xml:  
<library xmlns="http://www.example.com">  
 <magazine>MSDN Magazine</magazine>  
 <book>专业的 XML 数据库</book>  
</library>

在上面的例子中，library 元素的内容模型允许包含一个或多个 book 元素。由于 magazine 元素包含在 book 替换组中，因此 magazine 元素可以出现在需要 book 元素的实例 XML 中。

替换组使得内容模型更加灵活，其可扩展性大大超出架构作者的想象。灵活性是一把双刃剑，虽然它提供了更好的可扩展性，但是处理基于这种架构的文档将更加困难。例如，在上面的示例中，处理 library 元素的代码不仅要处理它的 book 子元素，还要处理 magazine 元素。如果实例文档还通过 [xsi:schemaLocation](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#xsi_schemaLocation) 属性指定了更多架构，那么应用程序就必须将更多的 book 替换组成员作为 library 元素的子元素进行处理。

更复杂的是，替换组中的成员可以是从替换组的头类型派生出来的类型。通过编写代码来正确处理派生类型，这通常都会比较困难，尤其是当存在两种相对的派生方法时：restriction，限制内容模型的范围或值；extension，向内容模型添加元素或属性。通过使用元素声明中的某些属性，架构作者可以更好地控制实例文档中的元素替换，并降低 XML 实例文档中出现意外替换的可能性。**block** 属性可用于指定元素（其类型使用某种派生方法）是否可以替换实例文档中的元素，而 **final** 属性可用于指定元素（其类型使用某种派生方法）是否可以将自己声明为目标元素替换组的一部分。架构中所有元素声明的 **block** 和 **final** 属性的默认值都可以通过根 **xs:schema** 元素的 **blockDefault** 和 **finalDefault** 属性来指定。默认情况下，允许执行所有替换，没有任何限制。

### **为何应该使用 key/keyref/unique 而不是使用 ID/IDREF 进行标识约束**

DTD 提供了一种将某种属性指定为类型 [ID](http://www.w3.org/TR/REC-xml#id)（即属性值在文档中是唯一的）的机制，这种机制与 XML 1.0 中的[名称生成](http://www.w3.org/TR/REC-xml#NT-Name)机制相对应。XML 1.0 中的 ID 也可以被类型 [IDREF](http://www.w3.org/TR/REC-xml#idref) 或 [IDREFS](http://www.w3.org/TR/REC-xml#idrefs) 的属性引用。为了与 DTD 兼容，W3C XML 架构提供了 [xs:ID](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#ID)、[xs:IDREF](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#IDREF) 和 [xs:IDREFS](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#IDREFS) 类型。

W3C XML 架构[标识约束](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#cIdentity-constraint_Definitions)可用于通过使用在元素声明范围内定义的 XPath 表达式来指定唯一值、主键以及对主键的引用。从功能方面来看，标识约束机制比 ID/IDREF 机制的功能多。首先，它对可以作为标识约束使用的值或类型没有限制，而 ID 只能是规定的值之一（例如，7 就不是有效的 ID）。架构标识约束优于 ID/IDREF 的另一个重要方面在于：在文档中，后者必须是唯一的，而前者没有此项限制。换句话说，唯一 ID 的符号空间是整个文档，而唯一主键的符号空间是 Xpath 的目标范围。如果同一 XML 文档的两个不同范围有重叠的值空间，并且这些值又要求唯一，主键的这种特性就会非常有用。包含某个酒店的房间号和餐桌号的 XML 文档就是一个例子。很有可能存在号码重叠的情况（例如 18 号房间和 18 号餐桌），但是它们在自己的值空间中不应该重叠。

### **注意：**W3C XML 架构的 ID 类型系列不完全与 DTD ID 类型兼容。首先，**xs:ID**、**xs:IDREF** 和 **xs:IDREFS** 类型在 W3C XML 架构中既可用于元素，也可用于属性，但在 DTD 中却只能用于属性。其次，元素中可以出现的类型 **xs:ID** 的属性数量没有限制，而 DTD 中的 ID 属性则有限制。

**为何应该慎重使用可变架构**

架构文档的目标命名空间用于标识可与架构进行验证的元素和属性的命名空间名称。没有目标命名空间的架构通常只能验证无命名空间名称的元素和属性。但是，如果一个没有目标命名空间的架构包含在另一个有目标命名空间的架构中，那么无目标命名空间的架构就把包含它的架构的目标命名空间作为自己的目标命名空间。这种功能通常称为[可变架构设计模式](http://www.xfront.com/ZeroOneOrManyNamespaces.html#mixed)。

在 Kohsuke 的文章中，[他声称可变架构方案行不通](http://www.xml.com/pub/a/2001/06/06/schemasimple.html?page=2#avoid_chameleon)，这种说法是不正确的。Michael Leditschke 在关于 XML-DEV 的文章中[反驳](http://lists.xml.org/archives/xml-dev/200209/msg00027.html)了 Kohsuke 的观点，说明这种设计方案是可行的，并指出它对创建需要重复使用的类型定义和声明模块是非常有用的。

将可变架构与[标识约束](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#cIdentity-constraint_Definitions)结合使用时会有一些问题。问题在于，在可变架构中，QName 对类型定义和声明的引用被强制应用包含架构的命名空间，但 **xs:key**、**xs:keyref** 和 **xs:unique** 标识约束所使用的 XPath 表达式却没有被强制应用包含架构的命名空间。请考虑以下架构：

<xs:schema  
 xmlns:xs="http://www.w3.org/2001/XMLSchema"  
 elementFormDefault="qualified">  
  
 <xs:element name="Root">  
  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="person" type="PersonType" maxOccurs="unbounded" />  
 </xs:sequence>  
 </xs:complexType>  
  
 <xs:key name="PersonKey">  
 <xs:selector xpath="person"/>  
 <xs:field xpath="@name"/>  
 </xs:key>  
  
 <xs:keyref name="BestFriendKey" refer="PersonKey">  
 <xs:selector xpath="person"/>  
 <xs:field xpath="@best-friend"/>  
 </xs:keyref>  
  
 </xs:element>  
  
 <xs:complexType name="PersonType">  
 <xs:simpleContent>  
 <xs:extension base="xs:string">  
 <xs:attribute name="best-friend" type="xs:string" />  
 <xs:attribute name="name" type="xs:string" />  
 </xs:extension>  
 </xs:simpleContent>  
 </xs:complexType>  
  
</xs:schema>

如果上面的架构被包含到另一个具有目标命名空间的架构中，那么 key 和 keyref 中的 XPath 表达式都会失败。在这个示例中，person 元素在可变架构中不具有命名空间，但是一旦被其他架构包含，它就会使用包含它的架构的目标命名空间。与没有命名空间的 person 相匹配的 XPath 表达式不能正常工作**并不**表示它们永远不能正常工作，因为处理器无需保证标识约束中的 path 表达式一定要返回结果。

**因此，建议在可变架构中不要使用标识约束。**

### **为何不应该使用默认值或固定值，尤其是对于 xs:Qname 类型**

使用默认值和固定值的主要问题是：它们会导致在验证后将新数据插入到源 XML 中，从而改变数据。这意味着，如果某个文档中的架构使用了未经验证的默认值，该文档就是不完整的。尝试对 XML 文档的实际内容进行验证是不明智的，因为有时可能没有架构。假设文档的用户始终执行验证也是不明智的。

由于窗体不规范而需要验证时，[xs:QName](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#QName) 类型还会出现一些其他问题。考虑以下架构和 XML 实例：

<xs:schema  
 xmlns:xs="http://www.w3.org/2001/XMLSchema"  
 targetNamespace="http://www.example.com"  
 xmlns:ex="http://www.example.com"  
 xmlns:ex2="ftp://ftp.example.com"  
 elementFormDefault="qualified">  
  
 <xs:element name="Root">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="Node" type="xs:QName" default="ex2:FtpSite" />  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
  
</xs:schema>  
  
<Root xmlns="http://www.example.com"   
 xmlns:ex2="smtp://smtp.example.org"   
 xmlns:foo="ftp://ftp.example.com">  
 <Node />  
</Root>

在上面的例子中，应当将什么值插入到 **Node** 元素中以便进行验证？如果 ex2 前缀被映射到实例文档而不是架构中的不同命名空间，那么应该是 **ex2:FtpSite** 吗？如果“foo”前缀与 ex2 被映射到架构中相同的命名空间，那么应该是“foo:FtpSite”吗？那么，如果 ftp://ftp.example.com 命名空间没有相应的 XML 命名空间声明，将会发生什么？必须向 XML 中插入命名空间声明吗？如果不违反有关正确操作的某些观点，这些问题都得不到满意的回答。最好避免使用 **xs:QName** 默认值，因为在上面的例子中，不同的实现在语义上不太可能一致。

### **为何应该使用简单类型的限制和扩展**

如前所述，W3C XML 架构中存在两种形式的类型派生：限制和扩展。对简单类型进行限制包括约束类型的各个[方面](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#facets)，从而减少类型的允许值。这种限制包括指定字符串值的最大长度、指定日期范围或枚举允许值的列表。架构作者通常以这种方式来使用类型约束，这些类型在 W3C XML 架构中占了类型派生的绝大多数。元素和属性的类型定义中都可以使用这些类型。

对简单类型进行扩展允许您使用具有属性的简单内容来创建复杂类型（即元素内容模型）。典型的扩展例子是元素声明将简单类型作为自己的内容，并且具有一种或多种属性。由于 XML 文档中普遍存在这种元素内容模型，因此通过扩展进行派生也是架构作者常用的功能。

值得注意的是，与复杂类型一样，简单类型也有命名和匿名两种。对于命名的简单类型来说，可以从定义它的架构或外部架构文档中通过名称对其进行引用，而匿名的简单类型必须在使用该类型的元素声明或属性声明中定义。同样，与复杂类型一样，只有命名类型才能执行类型派生。

常见的错误概念是，结构相同的匿名类型就是相同的类型：

<-- 片断 A -->  
  
<xs:element name="quantity">  
 <xs:simpleType>  
 <xs:restriction base="xs:positiveInteger">  
 <xs:maxExclusive value="100"/>  
 </xs:restriction>  
 </xs:simpleType>  
</xs:element>  
  
<xs:element name="size">  
 <xs:simpleType>  
 <xs:restriction base="xs:positiveInteger">  
 <xs:maxExclusive value="100"/>  
 </xs:restriction>  
 </xs:simpleType>  
</xs:element>

相当于：

<-- 片断 B -->  
  
<xs:simpleType name="underHundred">  
 <xs:restriction base="xs:positiveInteger">  
 <xs:maxExclusive value="100"/>  
 </xs:restriction>  
</xs:simpleType>  
  
<xs:element name="size" type="underHundred"/>   
  
<xs:element name="quantity" type="underHundred"/>

从根本上说，认为片段 B 中的两个元素声明类型相同是错误的。即使两种类型具有相同的结构，也不能认为它们相同，除非它们已命名，名称相同，并且来自于相同的目标命名空间。W3C XML 架构的各个方面（例如替换组、指定 key/keyref 对以及类型派生）都可能要求元素声明具有相同的类型。例如，keyref 必须与 key 的类型相同。但是，W3C XML 架构的大多数功能要求片段 A 中的元素声明具有*不同的*类型，而片段 B 中的元素声明具有相同的类型。

### **为何应该使用复杂类型的扩展**

对复杂类型进行扩展包括向派生类型的内容模型添加更多的属性或元素。通过扩展添加的元素按顺序附加到基本类型的内容模型中。对于提取一组复杂类型的共有特征，然后通过扩展基本类型定义来重复使用这些特征，该技术非常有用。以下架构片段来自 W3C XML Schema Primer 中的[对复杂类型扩展的讨论](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/#DerivExt)和[示例](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/#address.xsd)，显示了如何通过扩展来重复使用邮件地址的通用功能。

<xs:complexType name="Address">  
 <xs:sequence>  
 <xs:element name="name" type="xs:string"/>  
 <xs:element name="street" type="xs:string"/>  
 <xs:element name="city" type="xs:string"/>  
 </xs:sequence>  
 </xs:complexType>  
  
 <xs:complexType name="USAddress">  
 <xs:complexContent>  
 <xs:extension base="Address">  
 <xs:sequence>  
 <xs:element name="state" type="USState"/>  
 <xs:element name="zip" type="xs:positiveInteger"/>  
 </xs:sequence>  
 </xs:extension>  
 </xs:complexContent>  
 </xs:complexType>  
  
 <xs:complexType name="UKAddress">  
 <xs:complexContent>  
 <xs:extension base="Address">  
 <xs:sequence>  
 <xs:element name="postcode" type="UKPostcode"/>  
 </xs:sequence>  
 <xs:attribute name="exportCode" type="xs:positiveInteger" fixed="1"/>  
 </xs:extension>  
 </xs:complexContent>  
 </xs:complexType>

在上面的架构中，**Address** 类型定义了一般地址共有的信息，而它的两个派生类型分别添加了美国和英国的特殊地址信息。通过扩展来重复使用和建立内容模型是 W3C XML 架构具有的一项强大且有用的功能，它促进了相似内容的模块化和一致性。

使用处理器来处理通过扩展而派生出的类型时需要注意一个问题。此问题与类型相关处理器以及通过扩展而添加到内容模型中的元素有关。将来，类型相关语言（例如 [XQuery/](http://www.w3.org/TR/xquery) 或 [XLST 2.0](http://www.w3.org/TR/xslt20/)）有可能能够以多种形式来处理 XML 元素和属性。例如，应用程序可以决定处理类型 **Address** 的所有元素或处理以 Address 为基本类型的所有元素，选择处理所有类型的共有信息。但是，对于下面的查询：

//\*[. instance of Address]/city

如果使用以下方法处理扩展了内容模型的派生类型，它将返回意外的结果：

<xs:complexType name="BadAddress">  
 <xs:complexContent>  
 <xs:extension base="Address">  
 <xs:sequence>  
 <-- 地址格式中有两个 city 项，一个用于临近地区，  
 另一个用于实际的城市 -->  
 <xs:element name="city" type="xs:string"/>  
 <xs:element name="state" type="xs:string"/>  
 <xs:element name="country" type="xs:string"/>  
 </xs:sequence>  
 <xs:attribute name="exportCode" type="positiveInteger" fixed="1"/>  
 </xs:extension>  
 </xs:complexContent>  
 </xs:complexType>

尽管上面的例子是专门设计的且似乎是不可能的，但它说明了确实存在危险。有关这个潜在问题的详细说明，请参阅 [Paul Prescod on XML-Dev](http://lists.xml.org/archives/xml-dev/200209/msg00032.html)。

### **为何应该慎重使用复杂类型的限制**

对复杂类型的限制包括创建派生的复杂类型（其内容模型是基本类型的子集）。

首先，给您一些忠告。W3C XML 架构建议中描述的通过限制复杂类型来进行派生（[第 3.4.6 节](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#derivation-ok-restriction)和[第 3.9.6 节](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#cos-particle-restrict)）通常被认为是文档最复杂的部分，一般较难理解。实现时的大部分问题出在如何正确支持此功能方面，因此，当讨论通过限制复杂类型来进行派生的各种细节时，经常可以看到实现者们争论得面红耳赤。另一个问题是，通过限制复杂类型而实现的派生不能很好地映射到面向对象编程或关系数据库理论中的概念，而面向对象编程和关系数据库理论却是 XML 数据的主要来源和使用者。这与通过扩展复杂类型来进行派生的情形完全相反。

通过限制复杂类型来进行派生的另一问题是由声明限制的方法导致的：如果某个给定的复杂类型是通过限制另一个复杂类型而派生得到的，则必须复制和重定义它的内容模型。定义的副本会复制定义，这可能要经过一个很长的派生链，因此，对初始类型进行任何修改都必须手动传遍派生树。而且，这种复制不能越过命名空间边界 - 如果 ns2:SlowCar 具有子元素 ns2:MaxSpeed，则可能无法从 ns1:Car 派生 ns2:SlowCar，因为无法正确从 ns1:Car 的子元素 ns1:MaxSpeed 派生出 ns2:MaxSpeed。

以下架构通过限制派生方式来限制一个复杂类型，该复杂类型将 XML-DEV 邮件列表的一个用户描述为说明 Dare Obasanjo 的类型。符合 DareObasanjo 类型的任何元素也可以作为 XML-Deviant 类型的实例进行验证。

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema>  
  
 <!-- 基本类型 -->  
 <xs:complexType name="XML-Deviant">  
 <xs:sequence>  
 <xs:element name="numPosts" type="xs:integer" minOccurs="0"  
maxOccurs="1" />   
 <xs:element name="signature" type="xs:string" nillable="true" />  
 </xs:sequence>  
 <xs:attribute name="firstSubscribed" type="xs:date" use="optional" />  
 <xs:attribute name="mailReader" type="xs:string"/>  
 </xs:complexType>  
  
 <!-- 派生类型 -->   
 <xs:complexType name="DareObasanjo">  
 <xs:complexContent>  
 <xs:restriction base="XML-Deviant">  
 <xs:sequence>  
 <xs:element name="numPosts" type="xs:integer" minOccurs="1" />   
 <xs:element name="signature" type="xs:string" nillable="false" />  
 </xs:sequence>  
 <xs:attribute name="firstSubscribed" type="xs:date" use="required" />  
 <xs:attribute name="mailReader" type="xs:string" fixed="Microsoft Outlook" />  
 </xs:restriction>  
 </xs:complexContent>  
 </xs:complexType>   
  
</xs:schema>

通过限制复杂类型而进行的派生是一种具有多面性的功能，当第二种类型需要符合通用的主类型时它非常有用，但却为自己添加了比主类型更多的约束。但是，因为它太复杂，所以只有完全了解 W3C XML 架构功能的人才可以使用。

### **为何应该慎重使用抽象类型**

通过借用面向对象编程语言如 C# 和 Java 中的一个概念，元素声明和复杂类型定义都可以变成抽象的。抽象元素声明指的是元素声明不能用于验证 XML 实例文档中的元素，只能通过替换出现在内容模型中。与此类似，抽象复杂类型定义也不能用于验证 XML 实例文档中的元素，但是可以用作元素派生类型的抽象父元素，或在使用 [xsi:type](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#xsi_type) 替代实例中的元素类型时使用。

抽象复杂类型和元素声明对于创建包含一组类型（例如 **Shape** 与 **Circle** 或 **Square**）的常见信息的通用基本类型很有用。但是只有在应用了其他派生（扩展或限制）后，定义才是“完整的”。尽管此功能易于使用，但其使用存在一些微妙和复杂之处，因此，使用抽象类型时应慎重。

### **应该使用通配符以提高可扩展性**

W3C XML 架构提供了通配符 **xs:any** 和 **xs:anyAttribute**，用来使指定命名空间中的元素和属性出现在内容模型中。使用通配符，架构作者既能使内容模型可扩展，同时又能保持对元素和属性的控制。有关使用通配符的好处，请参阅 [W3C XML 架构设计模式：处理改变](http://www.microsoft.com/msdn/library/en-us/dnxml/html/XMLSchemaChange.asp)中的讨论。

谨慎的架构作者关心由类型派生引起的问题，可能在复杂类型定义和元素声明中使用 **final** 属性（与 C# 中的 **sealed** 和 Java 中的 **final** 一样）来禁止类型派生，然后使用通配符对内容模型的某些部分实现扩展。这样，架构作者就可以更好地控制他们所定义的内容模型，从而减少复杂类型派生所出现的各种问题（尤其是通过扩展进行派生所引起的问题）。

值得注意的是，通配符有时会出现[不确定性](http://www.microsoft.com/msdn/library/en-us/cpguide/html/cpconnon-deterministicschema.asp)问题，如果使用不当，会与 [Unique Particle Attribution](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#cos-nonambig) 规则冲突。以下架构就出现了这样的问题，原因是：

<?xml version="1.0" encoding="utf-8" ?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"   
 targetNamespace="http://www.example.com/fruit/"  
 elementFormDefault="qualified">  
  
<xs:complexType name="myKitchen">  
 <xs:choice maxOccurs="unbounded">  
 <xs:any processContents="skip" />  
 <xs:element name="apple" type="xs:string"/>  
 <xs:element name="cherry" type="xs:string"/>   
 </xs:choice>  
</xs:complexType>  
  
</xs:schema>

...myKitchen 类型的内容模型可以包括一个或多个 apple、cherry 或任意其他元素。但是，在验证过程中，如果遇到一个 <apple> 元素，编译器不能辨别是根据通配符进行验证，还是根据苹果元素声明进行验证，于是导致不明确。

选择 **namepsace** 属性和 **processContents** 属性有一些既微妙又深奥的细节。过度限制值可能会妨碍可扩展性，而过度开放值又可能导致架构的误用。控制通配符支持的命名空间也容易使人不知所措，尤其是当可选命名空间集可能会发生改变时。

### **不要使用组或类型重定义**

[重定义](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/#Redefine)是 W3C XML 架构的一种功能，它允许您改变已包含的类型或组定义的含义。架构作者可以使用 [xs:redefine](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-redefine)，包括架构文档中的类型或组定义，然后以某种方式改变这些定义。重定义会同时影响具有包含关系的两个架构中的类型或组定义，所以其影响相当广泛。这样，所有对两个架构中原始类型或组的引用都将引用重定义的类型，同时原始定义被屏蔽。这样就导致了 [W3C XML 架构设计模式：处理改变](http://www.microsoft.com/msdn/library/en-us/dnxml/html/XMLSchemaChange.asp)中指出的问题：

这将在一定程度上削弱性能，因为重定义的类型可以逆向与派生类型结合，产生冲突。常见的冲突是当派生类型使用扩展向类型的内容模型添加元素或属性时，重定义也向内容模型添加类似的命名元素或属性。

类型重定义的主要问题是，与类型派生不同，不能使用 **block** 或 **final** 属性来禁止类型重定义。因此，所有架构均可以通过一般方法重定义自身的类型，从而完全改变它们的语义。由于可能会引起冲突，建议避免使用此功能。

许多架构作者试图使用类型重定义来加大[枚举类型](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#dt-enumeration)的值空间，但这种方式不能获得成功。用作基本类型的枚举类型可以接受的、唯一可以增加值的数量的方法是创建一个 [union](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#element-union)。但是，添加的这些值只能用于产生 union 类型的应用程序，而不能用于原始基本类型的应用程序。此外，还需要注意的是链式重定义（对重定义的类型再次重定义）可能会出现问题，导致意外的定义冲突。

## **小结**

W3C XML 架构建议是一个复杂的规范，因为它试图解决复杂的问题。利用架构语言中相对简单的部分，而放弃深究复杂的部分，了解这种架构时就会简单得多。总之，架构作者应该确保在多个架构处理器中验证他们的架构，以避免由于各种实现处理语言的方法不同而引起的所有常见问题，如“只能在自己机器的验证程序 X 中正常工作，而不能在接收端的验证程序 Y 中正常工作”。需要注意的是，架构的互操作性非常重要。架构作者注重特殊实现的细节，而不注意互操作性，这是不明智的。