# XML概述

eXtensible Markup Language 的缩写，即可扩展的标记语言。

主要作用：配置文件、数据交换。

XML没有固定的标记，任何人都可以定义它自己的标记。XML只是规定了一个XML文档应该如何编写。

# XML文档语法

一个XML文档通常包含：文档声明、元素定义、属性定义、注释等。

## 文档声明

在XML头部声明的内容：

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

## 所有 XML 元素都须有关闭标签

在 HTML，经常会看到没有关闭标签的元素：

<p>This is a paragraph

<p>This is another paragraph

在 XML 中，省略关闭标签是非法的。所有元素都**必须**有关闭标签：

<p>This is a paragraph</p>

<p>This is another paragraph</p>

**注释：**您也许已经注意到 XML 声明没有关闭标签。这不是错误。声明不属于XML本身的组成部分。它不是 XML 元素，也不需要关闭标签。

## XML 标签对大小写敏感

XML 元素使用 XML 标签进行定义。

XML 标签对大小写敏感。在 XML 中，标签 <Letter> 与标签 <letter> 是不同的。

必须使用相同的大小写来编写打开标签和关闭标签：

<Message>这是错误的。</message>

<message>这是正确的。</message>

**注释：**打开标签和关闭标签通常被称为开始标签和结束标签。不论您喜欢哪种术语，它们的概念都是相同的。

## XML 必须正确地嵌套

在 HTML 中，常会看到没有正确嵌套的元素：

<b><i>This text is bold and italic</b></i>

在 XML 中，所有元素都**必须**彼此正确地嵌套：

<b><i>This text is bold and italic</i></b>

在上例中，正确嵌套的意思是：由于 <i> 元素是在 <b> 元素内打开的，那么它必须在 <b> 元素内关闭。

## XML 文档必须有根元素

XML 文档必须有一个元素是所有其他元素的**父元素**。该元素称为**根元素**。

<root>

<child>

<subchild>.....</subchild>

</child>

</root>

## XML 的属性值须加引号

与 HTML 类似，XML 也可拥有属性（名称/值的对）。

在 XML 中，XML 的属性值须加引号。请研究下面的两个 XML 文档。第一个是错误的，第二个是正确的：

<note date=08/08/2008>

<to>George</to>

<from>John</from>

</note>

<note date="08/08/2008">

<to>George</to>

<from>John</from>

</note>

在第一个文档中的错误是，note 元素中的 date 属性没有加引号。

## 实体引用

在 XML 中，一些字符拥有特殊的意义。

如果你把字符 "<" 放在 XML 元素中，会发生错误，这是因为解析器会把它当作新元素的开始。

这样会产生 XML 错误：

<message>if salary < 1000 then</message>

为了避免这个错误，请用一个**实体引用**来代替 "<" 字符：

<message>if salary &lt; 1000 then</message>

在 XML 中，有 5 个预定义的实体引用：

|  |  |  |
| --- | --- | --- |
| &lt; | < | 小于 |
| &gt; | > | 大于 |
| &amp; | & | 和号 |
| &apos; | ' | 单引号 |
| &quot; | " | 引号 |

**注释：**在 XML 中，只有字符 "<" 和 "&" 确实是非法的。大于号是合法的，但是用实体引用来代替它是一个好习惯。

## XML 中的注释

在 XML 中编写注释的语法与 HTML 的语法很相似：

<!-- This is a comment -->

## 总结：格式良好的XML文档必须遵循如下规则

* XML 文档必须有根元素
* XML 文档必须有关闭标签
* XML 标签对大小写敏感
* XML 元素必须被正确的嵌套
* XML 属性必须加引号

# 简单理解命名空间的基本概念

**XML 命名空间可提供避免元素命名冲突的方法。**

## 命名冲突

在 XML 中，元素名称是由开发者定义的，当两个不同的文档使用相同的元素名时，就会发生命名冲突。

这个 XML 文档携带着某个表格中的信息：

<table>

<tr>

<td>Apples</td>

<td>Bananas</td>

</tr>

</table>

这个 XML 文档携带有关桌子的信息（一件家具）：

<table>

<name>African Coffee Table</name>

<width>80</width>

<length>120</length>

</table>

假如这两个 XML 文档被一起使用，由于两个文档都包含带有不同内容和定义的 <table> 元素，就会发生命名冲突。

XML 解析器无法确定如何处理这类冲突。

## 使用前缀来避免命名冲突

此文档带有某个表格中的信息：

<h:table>

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

此 XML 文档携带着有关一件家具的信息：

<f:table>

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

现在，命名冲突不存在了，这是由于两个文档都使用了不同的名称来命名它们的 <table> 元素 (<h:table> 和 <f:table>)。

通过使用前缀，我们创建了两种不同类型的 <table> 元素。

## 使用命名空间（Namespaces）

这个 XML 文档携带着某个表格中的信息：

<h:table xmlns:h="http://www.w3.org/TR/html4/">

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

此 XML 文档携带着有关一件家具的信息：

<f:table xmlns:f="http://www.w3school.com.cn/furniture">

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

与仅仅使用前缀不同，我们为 <table> 标签添加了一个 xmlns 属性，这样就为前缀赋予了一个与某个命名空间相关联的限定名称。

# XML约束和验证

即约束一个XML文档中要编写的内容。最常见的两种约束模式语言是： XML DTD和XML Schema。

## XML DTD

XML文档通过使用 DOCTYPE 声明语句（文档类型定义语句）来指明它所遵循的DTD文件，DOCTYPE 声明语句紧跟在XML文档声明语句后面，有两种格式：

（1）<!DOCTYPE 文档类型名称 SYSTEM "DTD文件的URL">

（2）<!DOCTYPE 文档类型名称 PUBLIC "DTD名称" "DTD文件的URL">

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE web-app PUBLIC  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  "http://java.sun.com/dtd/web-app\_2\_3.dtd">  <web-app>  <welcome-file-list>  <welcome-file>index.jsp</welcome-file>  </welcome-file-list>  </web-app |

或：

|  |
| --- |
| <!DOCTYPE 书单 SYSTEM "books.dtd"> |

## XML Schema

XML Schema被设计用来代替DTD。因为它的功能更加强大。

利用XML Schema，可以：

* 定义可出现在文档中的元素
* 定义可出现在文档中的属性
* 定义哪个元素是子元素
* 定义子元素的次序
* 定义子元素的数目
* 定义元素是否为空，或者是否可包含文本
* 定义元素和属性的数据类型
* 定义元素和属性的默认值以及固定值

如何在XML文档中引用XML Schema？

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"> |

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-3.0.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.0.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"> |

# XML解释

各种语言，都支持对XML文档的解释。

XML文档解释有两种方式：SAX和DOM。

对XML文档的解释框架有很多：JAXP、DOM4J、JDOM等等，其中，JAXP是JDK自带的功能，而DOM4J是最常用的XML解释框架。

## SAX

SAX是Simple API for XML的缩写。它适合处理大文件。SAX是基于事件流的解析。即从头读一篇XML文档，每碰到一些内容，就要求进行处理。在一次处理过程中，只能从头读到尾。

下面这个例子是利用JAXP来解释XML文档的示例。

|  |
| --- |
| **package** cn.com.leadfar.oa.service;  **import** java.util.ArrayList;  **import** java.util.List;  **import** javax.xml.parsers.SAXParser;  **import** javax.xml.parsers.SAXParserFactory;  **import** org.xml.sax.Attributes;  **import** org.xml.sax.SAXException;  **import** org.xml.sax.helpers.DefaultHandler;  **public** **class** SaxDemo **extends** DefaultHandler {  **private** List<String> tagName;  **private** List<String> tagValue;  **private** **int** step;  // 开始解析XML文件  **public** **void** startDocument() **throws** SAXException {  tagName = **new** ArrayList<String>();  tagValue = **new** ArrayList<String>();  step = 0;  }  // 结束解析XML文件  **public** **void** endDocument() **throws** SAXException {  **for** (**int** i = 0; i < tagName.size(); i++) {  **if** (!tagName.get(i).equals("") || tagName.get(i) != **null**) {  System.*out*.println("节点名称：" + tagName.get(i));  System.*out*.println("节点值：" + tagValue.get(i));  }  }  }  /\*\*  \* 在解释到一个开始元素时会调用此方法.但是当元素有重复时可以自己写算法来区分 这些重复的元素.qName是什么? <name:page ll=""></name:page>这样写就会抛出SAXException错误  \* 通常情况下qName等于localName  \*/  **public** **void** startElement(String uri, String localName, String qName,  Attributes attributes) **throws** SAXException {  // 节点名称  tagName.add(qName);  // 循环输出属性  **for** (**int** i = 0; i < attributes.getLength(); i++) {  // 获取属性名称  System.*out*.println("属性名称：" + attributes.getQName(i));  // 获取属性值  System.*out*.println("属性值："  + attributes.getValue(attributes.getQName(i)));  }  }  /\*\*  \* 在遇到结束标签时调用此方法  \*/  **public** **void** endElement(String uri, String localName, String qName)  **throws** SAXException {  step = step + 1;  }  /\*\*  \* 读取标签里的值,ch用来存放某行的xml的字符数据,包括标签,初始大小是2048, 每解释到新的字符会把它添加到char[]里。 \*  \* 注意,这个char字符会自己管理存储的字符, 并不是每一行就会刷新一次char,start,length是由xml的元素数据确定的,  \* 暂时找不到规律,以后看源代码.  \*  \* 这里一个正标签，反标签都会被执行一次characters，所以在反标签时不用获得其中的值  \*/  **public** **void** characters(**char** ch[], **int** start, **int** length)  **throws** SAXException {  // 只要当前的标签组的长度一至，值就不赋，则反标签不被计划在内  **if** (tagName.size() - 1 == tagValue.size()) {  tagValue.add(**new** String(ch, start, length));  }  }  **public** **static** **void** main(String[] args) {  String filename = "SaxDemo.xml";  SAXParserFactory spf = SAXParserFactory.*newInstance*();  **try** {  SAXParser saxParser = spf.newSAXParser();  saxParser.parse(SaxDemo.**class**.getResourceAsStream(filename), **new** SaxDemo());  } **catch** (Exception e) {  e.printStackTrace();  }  }  **public** List getTagName() {  **return** tagName;  }  **public** **void** setTagName(List tagName) {  **this**.tagName = tagName;  }  **public** List getTagValue() {  **return** tagValue;  }  **public** **void** setTagValue(List tagValue) {  **this**.tagValue = tagValue;  }  } |

相应的映射文件如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <people>  <person personid="e01" enable="true">  <name>张三</name>  <tel>44565654566</tel>  <email>zangsan@sina.com</email>  </person>    <person personid="e02" enable="false">  <name>李四</name>  <tel>324243444</tel>  <email>lisi@sina.com</email>  </person>    <person personid="e03" enable="true">  <name>王五</name>  <tel>3453553</tel>  <email>wangwu@188.net</email>  </person>    </people> |

## DOM

Document Object Model的缩写。DOM解释的方式更加常用。即把文档读到内存，然后通过一套API，随便访问文档中的任意内容。

下面是一个通过DOM4J来解释XML文档的示例：

|  |
| --- |
| **public** **class** Dom4jDemo **extends** TestCase {  **public** **void** testReadXml() **throws** Exception{  Document document = **new** SAXReader().read(**this**.getClass().getResourceAsStream("demo.xml"));  List nodes = document.selectNodes("//name");  **for** (Iterator iterator = nodes.iterator(); iterator.hasNext();) {  Element e = (Element) iterator.next();  System.*out*.println(e.getText());  }  }  } |

如果要使用XPATH，请加入相关的依赖包。

## 用getResource()和getResourceAsStream()读取配置文件

JAVA里面对于类进行调用配置资源的文件数据，以this.getClass().getResourceAsStream()来读取比较合适。

路径采用相对路径直接可以从工程的path路径去找。

主要问题是如果类中采用的是静态块的话，则该this.getClass()报错，因为读静态块时，可能该对象并未构造，所以用this来指向当前对象不行。

ClassLoader提供了两个方法用于从装载的类路径中取得资源：

public URL getResource(String name);

public InputStream getResourceAsStream(String name);

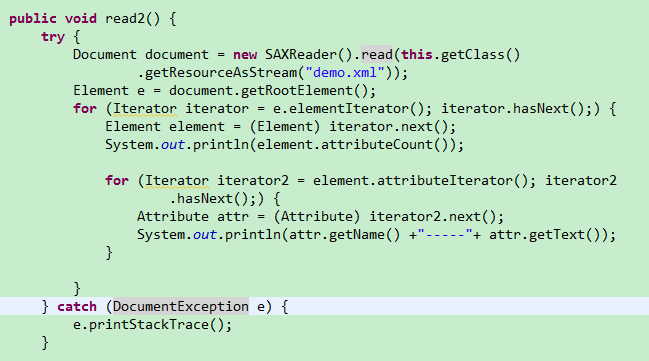
这里name是资源的类路径，它是相对与“/”根路径下的位置。getResource得到的是一个URL对象来定位资源，而getResourceAsStream取得该资源输入流的引用保证程序可以从正确的位置抽取数据。

然而，程序中调用的通常并不是ClassLoader的这两个方法，而是Class的getResource和getResourceAsStream方法，因为Class对象可以从你的类得到（如YourClass.class或YourClass.getClass()），而ClassLoader则需要再调用一次YourClass.getClassLoader()方法，但根据JDK文档的说法，Class对象的这两个方法其实是“委托”（delegate）给装载它的ClassLoader来做的，所以只需要使用Class对象的这两个方法就可以了。

# xpath

获取在xml文件中特定节点的path

List nodes = document.selectNodes("//name");



freemarker velocity

xpath是类似于sql语句的语句，专门用来查找xml文件中节点的

具体示例可参考网址：

http://www.zvon.org/xxl/XPathTutorial/General/examples.html



## 概述

\*

现节点下所有元素

\*/Elem

现节点下所有节点的字节点中为“Elem”的节点

@Prop

属性值

@\*

所有属于现节点的属性

.

现节点

..

现节点的上级

Elem[i]

现节点下第i个叫做Elem的元素(从1开始）

Elem[position() = 1]

同上

Elem/[@prop=“somevalue”]

现节点下，名字为Elem，具有prop的属性，并且属性值为somevalue的，那个元素

Elem1|elem2

现节点下，名字为Elem1或elem2的元素

.//elem

现节点下，可以跨越级别，所有的名字叫做elem的元素

Elem1//elem2

现节点下，可以跨越级别，所有的名字叫做elem2，且elem2的上级中有人叫Elem1，且Elem1是现节点的子元素，的元素

=

!=

&lt;

<=

&gt;

>=

text()

现节点的子元素中所有的文字节点

count()

count(PERSON[name=‘tom’])

number()

select=“number(book/price)”

substring(value,start,length)

select=“substring(name,1,3)”

sum()

select=“sum(//price)”

## XPath数据类型

XPath可分为四种数据类型：

* 节点集(node-set)   
  节点集是通过路径匹配返回的符合条件的一组节点的集合。其它类型的数据不能 转换为节点集。
* 布尔值(boolean)   
  由函数或布尔表达式返回的条件匹配值，与一般语言中的布尔值相同，有true和 false两个值。布尔值可以和数值类型、字符串类型相互转换。
* 字符串(string)   
  字符串即包含一系列字符的集合，XPath中提供了一系列的字符串函数。字符串可 与数值类型、布尔值类型的数据相互转换。
* 数值(number)   
  在XPath中数值为浮点数，可以是双精度64位浮点数。另外包括一些数值的特殊描 述，如非数值NaN（Not-a-Number）、正无穷大infinity、负无穷大-infinity、 正负0等等。number的整数值可以通过函数取得，另外，数值也可以和布尔类型、 字符串类型相互转换。

其中后三种数据类型与其它编程语言中相应的数据类型差不多，只是第一种数据类型是XML文档树的特有产物。

## XPath节点类型

另外，由于XPath包含的是对文档结构树的一系列操作，因此搞清楚XPath节点类型 也是很必要的。回忆一下第二章中讲到的XML文档的逻辑结构，一个XML文件可以包 含元素、CDATA、注释、处理指令等逻辑要素，其中元素还可以包含属性，并可以利 用属性来定义命名空间。相应地，在XPath中，将节点划分为七种节点类型：

1. 根节点（Root Node）   
   根节点是一棵树的最上层，根节点是唯一的。树上其它所有元素节点都是它的子节点 或后代节点。对根节点的处理机制与其它节点相同。在XSLT中对树的匹配总是先从根 节点开始。
2. 元素节点（Element Nodes）   
   元素节点对应于文档中的每一个元素，一个元素节点的子节点可以是元素节点、注释 节点、处理指令节点和文本节点。可以为元素节点定义一个唯一的标识id。 元素节点都可以有扩展名，它是由两部分组成的：一部分是命名空间URI，另一部分是 本地的命名。
3. 文本节点（Text Nodes）   
   文本节点包含了一组字符数据，即CDATA中包含的字符。任何一个文本节点都不会有紧 邻的兄弟文本节点，而且文本节点没有扩展名。
4. 属性节点（Attribute Nodes）   
   每一个元素节点有一个相关联的属性节点集合，元素是每个属性节点的父节点，但属 性节点却不是其父元素的子节点。这就是说，通过查找元素的子节点可以匹配出元素 的属性节点，但反过来不成立，只是单向的。再有，元素的属性节点没有共享性，也 就是说不同的元素节点不共有同一个属性节点。   
   对缺省属性的处理等同于定义了的属性。如果一个属性是在DTD声明的，但声明为 #IMPLIED，而该属性没有在元素中定义，则该元素的属性节点集中不包含该属性。   
   此外，与属性相对应的属性节点都没有命名空间的声明。命名空间属性对应着另一 种类型的节点。
5. 命名空间节点（Namespace Nodes）   
   每一个元素节点都有一个相关的命名空间节点集。在XML文档中，命名空间是通过保留 属性声明的，因此，在XPath中，该类节点与属性节点极为相似，它们与父元素之间的 关系是单向的，并且不具有共享性。
6. 处理指令节点（Processing Instruction Nodes）   
   处理指令节点对应于XML文档中的每一条处理指令。它也有扩展名，扩展名的本地命 名指向处理对象，而命名空间部分为空。
7. 注释节点（Comment Nodes）   
   注释节点对应于文档中的注释。

## 一个XML文档树

我们来构造一棵XML文档树，作为后面举例的依托：

|  |
| --- |
| <A id="a1">  <B id="b1">  <C id="c1">  <B name="b"/>  <D id="d1"/>  <E id="e1"/>  <E id="e2"/>  </C>  </B>  <B id="b2"/>  <C id="c2">  <B/>  <D id="d2"/>  <F/>  </C>  <E/>  </A> |

以下将要介绍一些XPath中节点匹配的基本方法。

## 路径匹配

路径匹配与文件路径的表示相仿，比较好理解。有以下几个符号：

（1）用“/”指示节点路径   
如“/A/C/D” 表示节点"A"的子节点"C"的子节点"D"，即id值为d2的D节点， “/”表示根节点。

（2）用“//” 表示所有路径以"//"后指定的子路径结尾的元素   
如“//E” 表示所有E元素，结果是所有三个E元素， 如“//C/E”表示所有父节点为C的E元素，结果是id值为e1和e2的两个E元素 。

（3）用“\*” 表示路径的通配符   
如“/A/B/C/\*”表示 A元素→B元素→C元素下的所有子元素，即name值为b的B元素、 id值为d1的D元素和id值为e1和e2的两个E元素   
“/\*/\*/D”表示上面有两级节点的D元素，匹配结果是id值为d2的D元素 ， 如“//\*”表示所有的元素。

## 位置匹配

对于每一个元素，它的各个子元素是有序的。

如：/A/B/C[1]表示A元素→B元素→C元素的第一个子元素，得到name值为b的B元素

/A/B/C[last()]表示A元素→B元素→C元素的最后一个子元素，得到id值为e2的E元素

/A/B/C[position()>1]表示A元素→B元素→C元素之下的位置号大于1的元素，得到id值为d1的D元素和两个具有id值的E元素

## 属性及属性值

在XPath中可以利用属性及属性值来匹配元素，要注意的是，元素的属性名前 要有"@"前缀。例如：

//B[@id]表示所有具有属性id的B元素，结果为id值为b1和b2的两个B元素

//B[@\*]表示所有具有属性的B元素，结果为两个具有id属性的B元素和一个具有name属性B元素

//B[not(@\*)]表示所有不具有属性的B元素，结果为A元素→C元素下的B元素

//B[@id="b1"] id值为b1的B元素，结果为A元素下的B元素

## 亲属关系匹配

XML文档可归结为树型结构，因此任何一个节点都不是孤立的。通常我们把节点之 间的归属关系归结为一种亲属关系，如父亲、孩子、祖先、后代、兄弟等等。在对 元素进行匹配时，同样可以用到这些概念。例如：

//E/parent::\* 表示所有E节点的父节点元素，结果为id值为a1的A元素和id值为c1的C元素

//F/ancestor::\* 表示所有F元素的祖先节点元素，结果为id值为a1的A元素和id值为c2的C元素

/A/child::\* 表示A的子元素，结果为id值为b1、b2的B元素，id值为c2的C元素，以及没有任何属性的E元素

/A/descendant::\* 表示A的所有后代元素，结果为除A元素以外的所有其它元素

//F/self::\* 表示所有F的自身元素，结果为F元素本身

//F/ancestor-or-self::\* 表示所有F元素及它的祖先节点元素，结果为F元素、F元素的父节点C元素和A元素

/A/C/descendant-or-self::\* 表示所有A元素→C元素及它们的后代元素，结果为id值为c2的C元素、该元素的子元素B、D、F元素

/A/C/following-sibling::\* 表示A元素→C元素的紧邻的后序所有兄弟节点元素，结果为没有任何属性的E元素

/A/C/preceding-sibling::\* 表示A元素→C元素的紧邻的前面所有兄弟节点元素，结果为id值为b1和b2的两个B元素

/A/B/C/following::\* 表示A元素→B元素→C元素的后序的所有元素，结果为id 为b2的B元素、无属性的C元素、无属性的B元素、id为d2的D元素、无属性的F元素、\无属性的E元素。

/A/C/preceding::\* 表示A元素→C元素的前面的所有元素，结果为id为b2的B元素、id为e2的E元素、id为e1的E元素、id为d1的D元素、name为 b的B元素、id为c1的C元素、id为b1的B元素

## 条件匹配

条件匹配就是利用一些函数的运算结果的布尔值来匹配符合条件的节点。常用于条件 匹配的函数有四大类：节点函数、字符串函数、数值函数、布尔函数。 例如last()、position()等等，这里我们就不再赘述。

以上这些匹配方法中，用得最多的还要数路径匹配。在上一章样式表的例子中， 无论是在语句<xsl:template match="学生花名册">中，还是在语句 <xsl:value-of select="名字"/>中，都是依靠给出相对于当前路径的子路 径来定位节点的。