

第2章 XML 语 法

本章我们将细致地学习 XML 的基本语法，通过一个基本文档的例子了解它的实质内容。在最后，大家将能够编写满足自身需要的基本 XML 文档。作为本章的一部分，我们将开始使用 XML 开发书籍目录应用程序的基础部分，对与书籍的内容描述和动态数据库有关的数据进行描述和建模，它们能够支持传统印刷目录的所有功能。对于那些只能在 Internet 上做到的电子商务功能，这些将成为其实现的基础。

你可以在 <http://www.w3.org/XML> 找到 XML 1.0 推荐标准（和其他信息）。在 <http://www.xml.com/axml/axml.html> 还有一个该标准的带注释的版本。

2.1 标记语法

XML 标记负责提供和描述一个 XML 文件或数据包（也就是大家所熟知的 XML 实体）的内容结构。它们由界定内容的不同部分的标记（tag）组成，负责提供到特殊符号和文本宏的引用，或者将特殊指令传递给应用软件，以及把注释传递给文档编辑器。

相信你们对 HTML 元素的标记已经很熟悉了（参见图 2-1）。

XML 元素的结构与 HTML 基本相同，XML 也同样使用尖括号来界定标记——以小于号（<）起始、大于号（>）结尾。但二者的相同点也就仅此而已。

与 HTML 不同，几乎所有的 XML 标记都是大小写敏感的，其中包括元素的标记名和属性值；也就是说：

Book ≠ BOOK ≠ book ≠ bOoK

之所以大小写敏感，主要是满足 XML 国际化的设计目标和简化处理过程的需要。大多数非英语语言并不把字母表分成若干种写法，（即使是罗马字符）许多字母可能也没有对应的大写或小写。例如，在法语中“ç”就不一定是“ç”（也可以是“C”）。希腊字母“西格马”只有一个大写形式，但却有两个小写形式；阿拉伯语则对每一个字母使用多种形式的写法；等等。合并写法会存在许多缺陷，尤其是对于非 ASCII 码更是如此，而 XML 的设计者们大多选择避免这些问题。

下面让我们看一看 XML 是如何满足国际化的需要的。

2.1.1 字符

由于 XML 是要在全球范围内使用的，所以不能局限于 7 位的 ASCII 码字符集。XML 指定的字

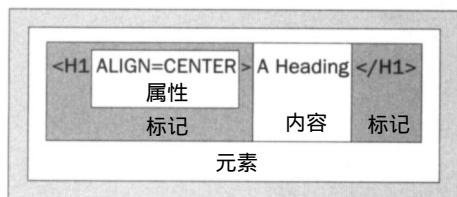


图 2-1

符均在16位的Unicode 2.1字符集（参见<http://www.unicode.org>——它目前与ISO/IEC 10646是一致的——参见<http://www.iso.ch>）中定义。这些都是相对较新的标准，而且当今世界还有许多文字没有编入统一码当中。但是，由于它被设计为大多数现存字符编码的超集，所以遗留的内容向统一码的转换也是简单直观的。例如，把ASCII码转换成统一码只需要把16位字符的前8位填充为0（而保留后8位）即可。

计算机字符史上还有一段关于连字符的小插曲（例如与从各类排版系统继承而来的“fi”或“ff”字符组合），此外还有早期对日文处理的尝试所采用的“半宽片假名”。虽然这些奇谈怪论都包含进了统一码标准，但并不鼓励大家使用——例如连字符就不是真正的字符，它们只是某种形式的印刷样式，能够以最好的方式处理文本显示，但并不嵌入到文本数据中。

访问由Jukka Korpela制作的页面<http://www.hut.fi/u/jkorpela/chars.html>，可以看到关于字符编码问题的一个很好的介绍。

合法的XML字符包括三个ASCII C0控制符，所有普通的ASCII可显示字符，以及所有其他统一代码字符值（用十六进制表示），可参见表2-1。

表 2-1

字符值（十六进制）	描 述
09	水平制表符（HT）
0A	换行（LF）
0D	回车（CR）
20 .. 7E	ASCII显示字符
80 .. D7FF	统一代码字符（包括“Latin-1”）
E000 .. F8FF	“私有区域”
F900 .. FFFD	CJK（中日韩）兼容的象形文字
10000 .. 10FFFF	待用集和“高度私有区域”

统一代码中包含一个数量超过137 000的字符集合用于应用程序特定字符，它被称作“保留区域”。当然，使用这些私有字符进行任何XML数据的交换都需要就这些字符的解释单独达成一致，所以统一代码的这个部分不应该在XML数据对象中使用，因为它们需要在相当广的范围内进行交换。

2.1.2 命名

在XML中使用的结构几乎总是被命名的。所有XML命名都必需以字母、下划线（_）或冒号（:）开头，后面跟着的是有效命名字符。有效命名字符除了前面的内容，还包括数字、连字符（-）、句点（.）。在实际应用当中不应该使用冒号，除非是用作命名空间的分隔符（参见第7章）。记住字母并非局限于ASCII码是非常重要的，因为不说英语的人们可以把自己的语言用在标记当中。

XML规范还定义了一种名叫命名令牌（name token，通常缩写为nmtoken）的相关观念，它可以是命名字符的任意组合而没有词首字母的任何限制。除了它能够使用在属性值

(后面将介绍)中这个问题,我们不想在本章过多地讨论命名令牌,但是当我们在下一章开始讨论有效XML文档时,它们就变得很重要了。

在命名方面另一个限制是它们不能由字符串“xml”、“XML”或任何以此顺序排列的这三个字母的各类组合(例如“xML”或“Xml”)开头。W3C保留对以这三个字母开头的命名的使用权。

下面就是一些合法的命名:

```
Book
BOOK
Wrox:Book_Catalog
ΑΓΔ
Conseil_Européen_pour_la_Recherche_Nucléaire_(CERN)
```

注意前两个命名并不等同——这一点我们前面已经讨论过,XML的命名是大小写敏感的。第三个是使用建议的命名空间分隔符(冒号)的典型例子——在第7章你可以了解到关于命名空间的更多信息。最后两个例子提醒大家注意希腊语和法语同英语一样,都可以用作XML的命名。

下面是一些非法的命名:

```
-Book
42book
AmountIn$
E=mc2
XmlData
XML_on_NeXt_machines
```

头两个例子开头使用的字母(“-”和“4”)虽然是合法的命名字符,但作为首字母却是非法的。第三个和第四个例子的字符根本就是非法字符(“\$”和上标“2”)。最后两个例子违反了“xml是保留字符”的限制(当然,如果它们是由W3C定义的则是另外一回事了)。在这种情况下或前两个例子中,如果开头字母是下划线(例如,“_42book”、“_xml”或“_XML”),这些命名就成为合法的了。

现在我们知道了如何按照XML语法正确地命名,下面来看一看如何使用它们。

2.2 文档部分

一个格式正规的XML文档由三个部分组成(参见图2-2):

- 一个可选的序言(prolog)。
- 文档的主体(body),由一个或多个元素组成,其形式为一个可能也包含字符数据(character data)的层次树。
- 可选的“鱼龙混杂”的尾声(epilog),其内容包括注释、处理指令(processing instruction, PI)和/或紧跟元素树后面的空白。

详细信息我们很快就会谈到。

既然数据对象即使没有序言和/或尾声也可以是格式正规的XML,我们暂时不考虑这些部分的细节,直到我们已经介绍完所有重要的中间部分——元素和字符数据。

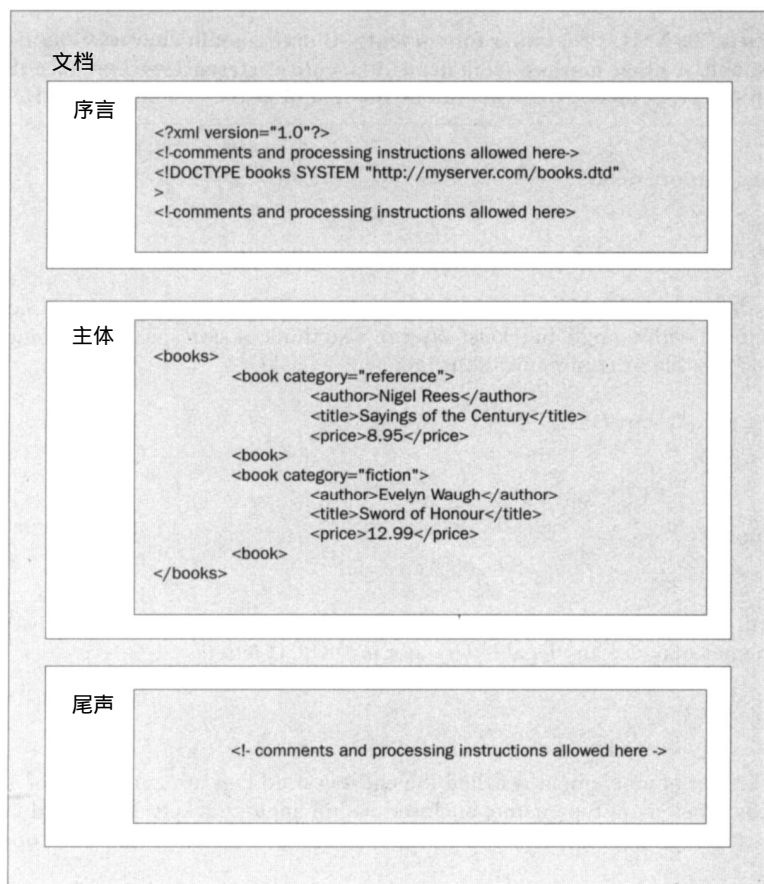


图 2-2

2.3 元素

元素是XML标记的基本组成部分。它们可以包含其他的元素、字符数据、字符引用、实体引用、PI、注释和/或CDATA部分——这些合在一起被称作元素内容（element content）。（不用担心这些术语都是什么意思，接下来我们会进行解释。要注意这些元素都是容器。）所有的XML数据（除了注释、PI和空白）都必须包容在其他元素中。

元素使用标记（tag）进行分隔——由一对尖括号（“<>”）围住元素类型名（一个字符串）。每一个元素都必须由一个起始标记和一个结束标记分隔开，这与要求比较松的HTML不同，后者的结束标记可以省略。这项规则唯一的例外是没有任何内容的元素，即空元素（empty element），它既可以使用起始标记/结束标记对，也可以使用短小精悍的混合形式——空元素标记。在下面的章节里，我们会看到许多标记的例子。

元素是XML对内容的容器——它可以包容字符数据、其他元素以及/或其他标记（注释、PI、实体引用等等）。既然元素代表的是一些离散的对象，我们可以把它们看作是XML语

言中的名词。

下面我们看一看这些标记的细节信息。

2.3.1 起始标记

一个元素开始的分隔符被称作起始标记。起始标记是一个包含在尖括号里的元素类型名。我们也可以把起始标记看作是“打开”了一个元素，就像我们打开一个文件或通信链路一样。

下面是一些合法的起始标记：

```
<Book>
<BOOK>
<Wrox:Book_Catalog>
<ΑΓΔ>
```

再次强调，由于 XML 是大小写敏感的，所以前两个例子不是等价的标记；而且，元素类型名可以使用任何合法字母，而不一定是 ASCII 码字符。

2.3.2 结束标记

一个元素最后的分隔符被称作结束标记。结束标记由一个反斜杠和元素类型名组成，被围在一对尖括号中。每一个结束标记都必须与一个起始标记相匹配，我们可以把结束标记理解为关闭了一个由起始标记打开的元素。

下面是一些合法的结束标记，它们与前面列举的起始标记相对应：

```
</Book>
</BOOK>
</Wrox:Book_Catalog>
</ΑΓΔ>
```

所以，带有完整的起始、结束标记的元素应该是如下形式：

<某个标记> 包含的内容 </某个标记>

下面我们简单地了解一下没有内容的元素。

2.3.3 空元素标记

空元素可能不包含任何内容。比如说想准确地指明文档中的某些特定位置（下一节将看到这样的例子）。我们可以只加入起始标记和结束标记而不在其中包含任何内容：

```
<point1></point1>
```

当然，如果你只是想指定一个点，而不是提供一个包容器，节省些空间可能会更好。所以，XML 指定空元素可以用缩略形式表示，它是起始和结束标记的混合体。它既短小精悍，而且还能明确指出该元素既不会有内容，也不允许有内容。

空元素标记由一个元素类型名称紧跟一个反斜杠组成，并围在一对尖括号中。

```
<point1/>
```

XML 数据对象可能只包含单个文档元素和一些空元素（可能有属性）！这样的文件可以用

于包含程序的配置信息或者 C++ 对象模板。

2.3.4 标记：一个简单的例子

任何简单的 ASCII 文本文件都是非常奇妙的容器（文件），其中有一系列更小的容器（文本行），而其中又顺序排列着字母。从另一种意义上说，文件的物理存在也包容在它的父文件系统中。但是，在没有一个明确表示“文件开始”的分隔符的同时，文件结束位置经常用一个特殊控制字母标记出来（例如“Ctrl-Z”或它的十六进制值“1A”）。文本行隐含的开始位置是它的父文件的物理起始处，但每一个文本行都有一个用回车和 / 或换行符表示的“行结尾”的分隔符。

例如，下面是一个基本的 ASCII 码文件（注意：行号并不是文件内容的一部分）：

程序清单 2-1

```
1:  A Simple Example
2:    by Yours Truly
3:  This is the 3rd line of a simple 5-line text file.
4:  ...the 4th line...
5:  And lastly, a final line of text.
```

当同样的文本用 XML 文档表示时，原本含糊不清的数据结构现在变得清晰明确了（再次说明，这里的行号并不是文档的一部分）：

程序清单 2-2

```
1:  <textfile>
2:    <line>A Simple Example</line>
3:    <line>  by Yours Truly</line>
4:    <line>This is the 3rd line of a simple 5-line text file.</line>
5:    <line>...the 4th line...</line>
6:    <line>And lastly, a final line of text.</line>
7:    <EOF/>
8:  </textfile>
```

在这个例子中，我们显式地表示出整个文件内容的开始和结尾（第 1 行和第 8 行），里面每一个文本行的起始和结束，并包含一个文件尾的标记（第 7 行）。这是一个表述清晰、可验证的结构，它由 7 个元素（其中一个包含其他 6 个）组成，这些元素用三个不同的元素类型表示（<textfile>、<line>和<EOF/>）。

现在，我们已经知道了元素类型名称的要求以及如何将标记应用在元素中，下面让我们先暂时停止对元素的讨论，看一看 XML 文档的结构。

2.3.5 文档元素

格式正规的 XML 文档的定义形式是一个简单的层次树，每个文档都有一个，而且只有一个根节点，它被称作文档实体（document entity）或文档根（document root）。这个节点可能包含 PI 和/或注释，而且总是包含子元素树，它们的根被称作文档元素（document element）。这个元

素是这个树中其他所有元素的父元素，而且它可能不包含在其他任何元素当中。由于文档根和文档元素并不是一回事，所以最好不要把文档元素看作是“根元素”（即使它是子元素树的根）。

图2-3显示了任何XML数据对象中最大的文档树的结构。

图2-4显示了上一节关于“textfile”的例子所隐含的文档树结构。

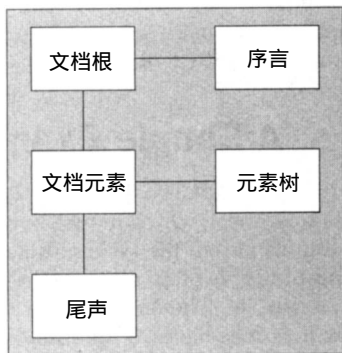


图 2-3

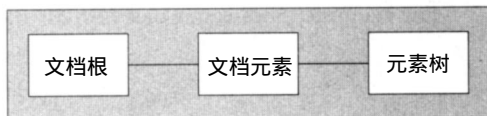


图 2-4

每个XML文档的文档根也是使用 DTD或模式定义的文档描述的附属品（你将在第 3章和第 7章了解到更多信息）。

任何格式正规的XML文档都必须由形成一个简单的层次树的元素组成，其中有一个被称作“文档根”的单个根节点。它包含第二层的元素树，这个树也存在一个被称作“文档元素”的根节点。

下面我们看一看文档主体中的元素是如何安排的。

2.3.6 子元素

XML文档中其他所有元素都是文档元素的后代（“孩子”）。在前面的文本文件例子中，文档元素是<textfile>元素，而<line>和<EOF>元素都是它的子元素。

图2-5所示的就是前一个“textfile”例子中隐含的元素树：

元素树和其中的父-子关系是XML的一个非常重要的特性。

任何元素类型只能包含四种内容中的一种。如果元素类型只允许包含其他元素或标记，而不能包含字符数据，那就是说它包含元素内容。可能包含字符数据和其他元素的元素类型被认为是可以包含混合内容。混合内容的一种子集就是只包含字符数据的元素，我们习惯地称它为“字符内容”。最后，“空元素”就是不包含任何内容的元素，虽然空元素标记可能包含属性（这一点稍后我们将提到）。

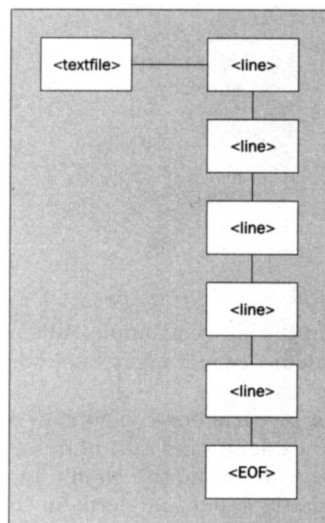


图 2-5

2.3.7 元素嵌套

XML对元素有一种非常重要的要求——它们必须正确地嵌套。对现实世界的对象的分析会有助于解释“正确嵌套”的含义。实际上，我们甚至可以说 XML元素是任何必需遵守它们的现实来源的规则的单词。

让我们来看一看本书传递到读者手中的整个过程。完成印刷后，本书会和其他 23本书打包到一个盒子中。两个盒子会被封装到一个纸箱中，许多纸箱会被装入一辆卡车然后运送到书店中。整个过程可以用以下 XML元素表示：

程序清单 2-3

```
<truck>
  <carton>
    <box>
      <book>...</book>
      <book>...</book>
      <book>...</book>
      ...
      <book>...</book>
    </box>
    <box>
      <book>...</book>
      ...
      <book>...</book>
    </box>
  </carton>
  <carton>
    ...
  </carton>
  ...
  <carton>
    ...
  </carton>
</truck>
```

在上面的例子中，缩排只是为了突出这些嵌套元素的层次结构，为了简单起见也省略了许多对书和纸箱的描述。

现实世界中的盒子能够包容整本书，但不可能出现书的某些部分在盒子中，而其他部分在外面的情况。同样，一本书也只能放在一个盒子中，不可能一部分在一个盒子，其他部分在另一个盒子（当然，我们要恳求大家不要把书撕成两半）。此外，盒子必需放在纸箱中，而纸箱必需顺序摆放在卡车里（请不要打开门把它们扔到大街上）。当然，XML元素也必须遵守这些现实世界包容关系的基本法则。

本例中隐含的层次树参见图 2-6。

不正确的嵌套

在元素结构的问题上，HTML和许多字处理格式几乎都没有 XML那么严格的要求。下面是最常见的HTML标记重叠的例子，它可以在大多数浏览器中使用，但在 XML中却是非法的：

```
<B>...some bold text, with <I>bold-italic</B> followed by plain italic text...</I>
```



```
name --- {name ∩ address} --- address
```

图 2.3.7 重叠

由于标记没有正确地嵌套，所以根本形成不了简单的元素层次树。中间的元素类型并没有被定义而且并没有真正存在，它只是一个指明两个被正确定义的元素类型的交叉点的占位符。虽然集合理论和非层次树肯定允许我们使用这类结构，但对于真正的编程实现来说这实在是一场噩梦。

由于没有办法区分明确的或模糊的重叠，而且对由这类重叠生成的树操作起来也比较复杂。所以XML只是简单地禁止任何重叠标记。在第一次遇到没有被正确定义的嵌套标记时，XML解析器必需报告一个“not well-formed（非格式正规的）”错误，而且通常情况下会退出处理并报告一个“致命”错误（在本章后面关于解析器的部分可以了解更多关于致命错误的信息）。

HTML/SGML:

HTML浏览器通常甚至正式接受不完整的标记，而SGML工具即使遇到错误也经常试图继续处理文档。但从设计的角度看，XML不允许这样的行为存在。

在我们开始讨论元素类型的属性前，我们先要说一说XML中的字符串。

2.3.8 字符串

字符串（string literal）主要用在属性值、内部实体和外部标识符中。XML都使用单引号（'）或双引号（"）作为一对分隔符将其中的字符串包围起来。对于这些字符串的一个限制是用于分隔符的字符不能够出现在字符串中——如果字符串中包含单引号，分隔符就必须使用双引号，反之亦然。如果两个字符都必须出现在字符串中，用在字符串中（同时也用作分隔符）的字符必需用适当的实体引用顶替（'或者"，二者我们都会在本章后面的2.6节“字符引用和实体引用”中讨论）。

下面是一些合法的字符串表述：

```
"string"  
'string'  
"...his friend&apos;s cow said "moo"
```

下面则是一些不合法的字符串表述：

```
"string"  
'...his friend's cow said "moo"'
```

从技术的角度讲，根据XML规范，字符串分隔符之间的文本是文档字符数据的一部分，在讨论属性之前，我们先看一看它所包含的意义。

2.4 字符数据

字符数据就是任何不是标记的文本，它是元素或属性值的文本内容。

小于号和&符号是标记分隔符，因此它们绝不能以字符串的形式出现在字符数据中（CDATA部分除外，这一点我们将在后面提到）。如果这些字符是字符数据所必需的，它们必需使用实体引用&或<来代替。这两个替代物是XML规范定义的5个类似字符串中的一部分，而且在所有兼容XML的解析器中都得到实现（在后面的2.6节“字符引用和实体引用”中我们将

了解到更多的信息)。

这里,我们需要再次提醒大家,由于XML的目的是在全球使用,所以文本是指统一代码,而不仅仅是ASCII码(参见本章前面的关于字符的部分)。

现在,我们来讨论属性的问题。

2.5 属性

如果说元素是XML中名词,那么属性就是这种语言的形容词。

在很多情况下,我们会希望将某些信息附着在元素上,它们与元素本身包含的信息内容有所不同。我们利用属性(attribute)来做到这一点,它们都包括一个名称-值组合,使用的格式有如下两种形式:

```
attribute_name="attribute_value"
attribute_name='attribute_value'
```

属性值必需是分隔开的字符串(字符串规则的要求),其中可能包含实体引用、字符引用(本章稍候将解释),以及/或文本字符。但是,正如我们刚才解释的那样,任何一个受保护的标记字符(<和&)都不能简单地在属性值中当作字符使用——它们必须用<或&实体引用来替代。

HTML允许数字化的属性,例如;或者不分隔的属性,比如<P ALIGN=LEFT>——但这两种情况在XML中都不允许存在。

在起始标记或空标记中属性只允许有一个实例存在。例如,下面的例子在XML当中就是非法的,因为src在一个标记中出现了两次:

```

```

这种限制极大地简化了XML解析器对属性的处理。

正如我们在前面暗示的,起始标记和空标记可能在标记中包含属性。例如,回到我们前面提到的关于书本、盒子、纸箱和卡车的例子,如果我们希望给每个运送书本的纸箱编上一个号码的话,可以使用如下属性:

程序清单 2-4

```
<carton number="0-666-42-1">
...
</carton>
<carton number='0-666-42-2'>
...
</carton>
```

在这个例子中,属性名称是“number”,相应元素起始标记中的值为“0-666-42-1”和“0-666-42-2”。注意两个合法的字符串分隔符(“和”)在本例中都被使用了。

同样,我们在前面提到的<textfile>例子中使用的“<EOF/>”空元素标记也可能包含原始文本文件文件尾字符的十六进制值:

```
<EOF char="1A"/>
```

在本例中，属性的名称为“char”而值为“1A”(即MS-DOS时代的Ctrl-Z)。除了你可以自行定义的属性外，还有两个在XML中起到特定作用的特殊属性。

2.5.1 特殊属性

在XML 1.0推荐标准中定义了两个特殊的属性：`xml:space`和`xml:lang`。

XML文档的作者可以使用这些属性向XML应用程序(例如，浏览器)传递某种信号。这两种属性都使用XML命名空间的语法，即一个命名空间的前缀(“xml”)紧跟一个冒号(:)，然后是属性名称(“space”和“lang”)。在第7章我们将了解到关于命名空间的更多信息。

1. `xml:space`属性

这个属性之所以存在，是因为HTML的`<pre>`标记被广泛使用以保持文本格式，其中包括任何嵌入的空白字符。但是，缺省情况下，使用XML的应用程序在特定元素中可能不保留空白部分，这取决于它们的目的。

XML文档的作者不会依赖于应用程序的默认行为，而是使用`xml:space`属性来告知应用程序它应该保留空白部分(虽然应用程序对于此信号的实际响应是由应用程序本身定义的——针对这一问题XML推荐标准中没有任何要求)。

该属性的值将被应用到元素及其所有子元素中——而不仅仅是带有`xml:space`属性的元素。这一点与XML对属性的通常处理方式不太一致，但这一例外确实还是必要的。

如果使用的是带校验的解析器，`xml:space`属性被限制为只能使用两个特定值：“preserve”和“default”——其他任何值都无效(在第3章中讨论DTD时我们会得到更多信息)。

下面让我们看一看另一个特殊属性。

2. `xml:lang`属性

这个属性之所以存在，是出于XML国际化设计目标的需要。统一代码的使用只是为其他人类语言使用的字符提供了一种标准的编码方法——统一代码可以说是在为文本的显示做着无声无息的贡献(在其中包含一些特殊的显示字符和用于显示带有二义性的犹太语文本的“BIDI”算法，以及用于组成亚洲字符的途径)。此外，它还包含了其他一些针对语言问题的考虑：字符和符号的排序；如何分隔字符以便于全文本索引；连字处理；特定性别的代名词或短语。

如果使用的是带验证的解析器，`xml:lang`属性(和其他属性一样)都必须在DTD(在第3章可以了解到更多的有关信息)中做出声明。此外，在这种情况下，该属性只能取如下几种值：

三类语言代码中的一种：

- ISO 639——http://sunsite.berkeley.edu/amher/iso_639.html
 - IETF的RFC1766——<http://www.ietf.org/rfc/rfc1766.txt>
 - 用户定义的语言代码
- ISO 3166国家代码——http://sunsite.berkeley.edu/amher/iso_3166.html

以下几种基本格式之一：

- 两个字母的ISO 639语言代码；例如，法语或日语可以表示为

```
fr
ja
```

- 两个字母的国家代码紧跟一个或多个子代码。如果存在第一个子代码而且包含两个字母，那么这些字符必须是 ISO 3166 国家码中的一员（见下面的前两个例子）。另外，子代码可以用来描述语言的书写符号、语调、地方变体等等。（见下面的后五个例子）；例如，像英语（“en”）和挪威语（“no”）这样的语言变量，或者那些用于阿塞拜疆语的书写字母变体：

程序清单 2-5

```
en-US
en-GB
en-cockney
no-bokmaal
no-nynorsk
az-arabic
az-cyrillic
```

- 字符串“ I- ”或“ i- ”，紧跟3个或8个字母、注册过的IANA语言代码（<http://www.isi.edu/in-notes/iana/assignments/languages>）；例如，美国本地的方言，Cherokee：

```
I-cherokee
```

- 字符串“ X- ”或“ x- ”，紧跟一个用户定义的语言代码；例如，由 Star Trek发明的语言 Klingon可以表示为：

```
X-klingon
```

用户定义的语言代码必需以“ x- ”或“ X- ”开头，以避免可能与注册的 IANA代码冲突。

在Internet用户群中，有一种固定的习惯，就是用小写字母表示语言代码，用大写字母表示国家代码。即使这些属性值并非大小写敏感（这一点与大多数 XML命名不同），坚持这一习惯还是相当必要的（记住XML的设计目的是简化已有的Internet协议的使用）。

下面的小段程序演示几种可能出现的情况：

程序清单 2-6

```
<example>
  <song xml:lang="de">Sagt mir wo die Blumen sind</song>
  <question xml:lang="en-GB">
    What is your favourite colour of flower?
  </question>
  <question xml:lang="en-US">
    What is your favorite color of flower?
  </question>
  <question xml:lang="X-INVERSE">
    What flower is your color?
  </question>
</example>
```

一个应用程序（或样式表）可能使用 xml:lang属性根据一些用户的语言配置设置决定显示哪

一个<question>标记。

和xml:space属性一样，xml:lang属性的值不仅应用于包含该属性的元素，而且也应用于它所有的子元素和其他属性。此外，应用程序没有义务去关心xml:lang属性的情况。类似于xml:space属性的一点是它可能对XML数据的样式化非常有帮助，xml:lang属性可能是几乎所有国际化文档所必需的功能。

这里我们还需要提一提对该功能的其他描述方法：大多数处理多语言文本的用户喜欢把这种对内容的处理叫做“语言标记”。当然，这会与XML术语相冲突：因为xml:lang属性并不是一个XML标记，所以，当我们讨论多语言XML文本时应当注意过滤和转化这种概念。

现在我们已经覆盖了属性和元素的语法，即XML的基本组成，你应该已经能够建立自己的简单的XML文档。但是在我们深入讨论另一个关键主题——实体之前，让我们先简单地了解两个问题，在你用自己的应用程序处理文档之前，这是必需掌握的内容：空白和行尾处理。

2.5.2 空白

在讨论xml:space属性时，我们使用了还没有真正介绍的术语“空白”。但不管是对于人类语言还是计算机语言来说，空白确实一个非常重要的语言概念。在XML数据中，只有4个字符可以作为空白使用（参见表2-2）。

无论如何，制表位占用的位置都只不会超过一个字符，所以它们中的每一个都可以简单地看作是一个字符。同样，任何由LF和/或CR隐含的格式也是交给应用程序和/或样式单处理。

同时，统一代码定义了许多不同种类的空格，但其中没有一个能够成为XML中的空白。

XML规范要求XML解析器将所有的字符，包括所有的空白字符，传递给应用程序。如果使用的是需要验证的解析器，当空白字符出现在元素内容中时（也就是说这些空白字符是元素字符数据的一部分），它会要求通知应用程序。所以，负责处理空白的总是应用程序。

XML处理空白的规则非常简单：解析器会保留内容中所有的空白字符并不加修改地传递给应用程序，但元素标记和属性值中的空白会被删除。

在书写SGML和HTML文档时，标记的缩进是非常常见的做法，但在处理文档时，HTML浏览器只会留下单词和文本中其他可识别单元之间的一个空白字符，即便是在文档内容中也是如此。这就意味着在书写文档时，作者可以根据自己的需要添加任意多的空白以便使文档更易于阅读并突出文档的结构，但是在文档处理时这一切都将失效。此外，不同的浏览器对于空白的删除办法有不同的缺省规则，许多HTML作者必须使用<pre>标记、 实体和/或表格标记来确定文本的空白部分。XML的设计目的就是简化HTML的处理办法以及它在空白处理方面自相矛盾的做法，同时避免SMGL针对空白的复杂规则。

表 2-2

字符值（十六进制）	描 述
09	水平制表（HT）
0A	换行（LF）
0D	回车（CR）
20	ASCII码中的空格字符

在HTML中，标记间的空白通常会被HTML浏览器所忽略。SGML在决定是否保留“由标记产生的”空白时有一大堆复杂的规则—但无论如何，这些规则从未非常清晰或简明。

现在，让我们看一看XML是如何处理文档中的行尾的。

2.5.3 行尾的处理

XML数据对象经常存储在离散的计算机文件当中，它们被分割为若干个文本“行”。在四个XML空白字符当中有两个是标准的ASCII码行尾控制字符。正如我们前面提到的，在用来表示行尾时，有这两个字符的三种常见组合：CR-LF，只有LF，以及只有CR。

为了简化XML应用程序的编码，XML解析器需要将所有的行尾字符串转换为单个LF（换行）字符。自然，这会让Unix编程者感到非常高兴，而让许多MS-Windows的开发人员怨声载道（MacOS用户已经适应了处理多种行尾字符串）。Tim Bray曾经提出过一些折衷办法（主要是考虑到MS-Windows的市场份额），但结果是XML仍然要求使用Unix风格的行尾字符。

在了解了如何处理空白和行尾之后，让我们着手正确解决字符和实体引用的问题。

2.6 字符引用和实体引用

与之前的SGML和HTML一样，XML为显示非ASCII码字符集中的字符提供了两种方法：

- 字符引用
- 实体引用

让我们先来了解一下字符引用。

2.6.1 字符引用

在XML中，字符引用是一个字符文字形式的替代品，当对该字符的文字形式直接处理会导致违反XML对格式正规的要求时（参见本章后面的2.12节“格式正规的文档”），它会起到非常重要的作用。

字符引用用来表示一个可显示的字符，它由十进制或十六进制的数字前面加上“&#”或“&#x”，后面紧跟分号（;）组成：

```
&#NNNNN;  
&#xXXXX;
```

上面的字符串“NNNNN”和“XXXXX”可能是一个或多个数字，它们对应着任何XML允许的统一代码字符值。虽然在HTML中十进制数字更加通用，但XML还是偏向于使用十六进制的形式，因为统一代码就使用十六进制进行编码。

例如，©或©（在HTML浏览器中）会被显示为(c)，而®或­会显示为®。下面，我们把它们同实体引用比较一下。

2.6.2 实体引用

实体引用允许在元素内容或属性值中插入任何字符串，这就为字符引用提供了一种助记的

替代方式。

实体引用是一种合法的XML名字，前面带有一个符号“&”，后面跟着一个分号(;)：

&name;

有五个实体被定义为XML的固有部分，它们通常用作XML标记分隔符号的转义序列（参见表2-3）。

表 2-3

实 体	用 途
&	通常用来替换字符&（除了在CDATA部分中——本章稍后将详细介绍）
<	通常用来替换字符小于号(<)（除了在CDATA部分中）
>	可能用来替换字符大于号(>)——在CDATA部分中，如果>紧跟着字符串“]]”就必须使用该实体
'	可用来替换字符串中的单引号(')
"	可用来替换字符串中的字符双引号(")

除了上述五个实体，所有实体都必须在文档使用前予以定义（就像传统编程中宏的定义和使用一样）。实体在文档的DTD中定义，DTD可以是一个被称作“外部子集”的文档外的独立对象（参见第3章）；也可以是一个在文档本身中使用<!DOCTYPE...>声明的“内部子集”（参见本章稍后的“文档类型声明”部分）。如果XML解析器发现一个未定义的实体引用，就会按照XML规范定义的那样报告一个致命错误（在2.12节“格式正规的文档”部分你也会看到关于这项内容的细节）。

例如：AT&T在支持XML的浏览器中会显示为AT&T，"Jack's Tracks"则显示为“Jack's Tracks”

实体引用还可以用作普通的文本宏（样本文件）。例如下面的文本包含了一对实体引用：

NOTE: &Disclaimer; [per &WROX;]

当引用被替换成它们所代表的值时，它可能显示为：

NOTE: This information is not to be used for navigation! [per Wrox Press, Ltd.]

当然，此时我们假设这些实体已经经过定义。

如果实体的替换文本在声明时包含另一个实体引用，该引用会顺序展开，直到所有嵌套的引用全部解析完毕。但是，嵌套的“名称”不能够包含对自己的递归引用，不管是直接的还是间接的。（在后面“文档类型声明”部分我们将知道对实体的声明是如何处理的。）

现在，我们来看一看处理指令。

2.7 处理指令

既然XML像以前的SGML一样是一种描述性的标记语言，它并不预先假设元素或者其内容的处理办法。这是一种非常强大的优势，因为它提供了显示的灵活性，以及针对应用程序和操作系统的独立性。但是，我们会发现经常需要把某些线索通过文档传递给应用程序。处理指令（Processing Instruction，PI）正是XML为此目的提供的一种机制。

PI使用的是XML元素语法的一种变形：

```
<?target ...instruction... ?>
```

处理指令target是必须的部分，而且必须是有效的XML名称，它用来指明哪个应用程序（或者其他对象）需要PI的帮助。PI的... instruction ...部分只不过是一个字符串表示，它可能包含任何有效的字符，除了“?”（因为这是PI的结束符）。此外在XML 1.0推荐标准中就没有更多关于PI语法的定义了。

另一个几乎随处可见的PI的用途就是将一个样式单和XML数据对象关联起来：

```
<?xml-stylesheet ... ?>
```

这个PI并没有出现在XML 1.0推荐规范当中，但在W3C推荐标准中则不同，你可以在<http://www.w3.org/TR/xml-stylesheet>找到1999年6月29提出的“Associating Style Sheets with XML Documents, Version 1.0”。在第13章中你会了解到更多关于这个处理指令的使用的内容。

请注意样式单关联的处理指令的名字以字符串“xml”开头；这对于任何非W3C定义的PI都属于非法，因为它们都在W3C规范中保留使用。

XML开发者群体一直在争论PI是否真正有用，以及特殊语法是否会妨碍XML的普遍性（因为在现有的浏览器中缺少对PI的支持，而且非标准化的目标名称可能会造成标记的不兼容性）。另一个针对PI的争论是许多通过PI传递的信息最好仅用在外部样式单中。

另一方面，PI也可能有几个好处：它可以作为脚本或服务端包含文件的挂钩（避免类似HTML语法中“<!-- -->”注释语法的泛滥成灾）；可以作为扩展模式的机制（否则它们就不能被修改）；它也是一种无需改变DTD认证就可以扩展文档的方法；此外，它也可以作为一种传递嵌入在文档当中的文档显示信息的途径，而且不会影响文档的结构。

既然我们已经谈到了HTML当中的注释语法。下面让我们来看一看XML的类似机制。

2.8 注释

这种机制对于在文档当中插入提示，或者叫注释（comment）来说是相当有帮助的。这些注释可能提供修订记录、历史信息或者其他类型的可能对创建者或者文档编辑者来说有着特殊意义但又不是真正的文档内容的元数据。注释可能出现在文档中除其他标记部分以外的任何地方。

XML注释的基本语法是：

```
<!--...comment text...-->
```

其中“comment text”部分可以是任何不包含“--”的字符串（这主要是为了保证对SGML的兼容性）。此外，“...”部分不能以连字号（“-”）结尾，因为这可能造成结束分隔符的混乱。

注释并不是文档的字符数据的组成部分！在注释部分当中，实体不可能展开，任何标记也不会被解释。

XML 1.0规范允许，但并不要求XML处理器为应用程序提供一种方法来获取注释的文本。因此，XML应用程序永远不能依靠使用注释来传输特殊指令（而这却是一个相当流行的HTML技巧）。

让我们来看一些例子：

程序清单 2-7

```
<tag> ...some content... </tag> <!-- this is a legal comment -->
<!--===== Beginning of some more comments =====>
<!--
    this is a comment containing unexpanded <tag> and &entity_reference;'s...
    that is continued on another line..
-->
<!--===== End of comments =====>
```

下面是一些不合法的注释（它们不能出现在一个元素标记中；“--”除非是注释分隔符的一部分，否则不能使用）：

程序清单 2-8

```
<tag> ...some content... </tag> <!-- this is an illegal comment --> >
<!-- this is an illegal comment -- because of the double hyphen within -->
```

下面，我们看一看避开大块文本的办法。

2.9 CDATA部分

CDATA部分是一种用来包含文本的方法，其对象是那些其中的字符如果不如此处理就会被识别为标记的文本。这项特性对于希望在自己的文档中包含 XML 标记的使用举例的作者来说是最有用的，就像本书中的举例。但这可能是在文档中包含 CDATA 部分的唯一说得过去的理由，因为在使用这些部分时 XML 几乎所有的优势都丧失殆尽。

CDATA部分并不是在 XML 文件中包含二进制数据的好办法！这些数据永远不能包含三字节的序列“5D 5D 3E”（“]]>”的十六进制表示），因为这个序列可能被解释为 CDATA 部分的结尾。二进制数据可以用 Base64 或其他什么技术编码，只要它能够保证被编码的数据没有包含大于符号（>）。但是，如果用了这种方法，用 Base64 编码的二进制数据就可以包含在任何元素的内容当中，因此 CDATA 部分也就不必要了。

只要有字符数据出现的地方就可能出现 CDATA 部分，但它们不能够嵌套。在 CDATA 部分中唯一能够被识别的标记字符串就是它的结束分隔符（“]]>”）——小于符号和 & 符号可能以字符形式出现；它们不必（也不能）被忽略。

CDATA 部分的基本语法如下：

```
<![CDATA[...]]>
```

在这里，“...”部分可以是任何字符串，只要不包含字符串“]]>”。

如果你希望包含一个连续的标记块，其中有 XML 文档中的实体引用（无需 XML 解析器展开实体或解释元素标记），可以使用下面两种方法的任意一种：

程序清单 2-9

```
<![CDATA[
<Catalog>
    <Legalese>&nbsp;&copy; 1999 Wrox Press, Ltd. &reg;</Legalese>
</Catalog>
]]>
```

或者：

程序清单 2-10

```
<Catalog>
  <Legalese>&nbsp;&copy; 1999 Wrox Press, Ltd.
  &reg;</Legalese>
</Catalog>
```

从解析器输出到应用程序的字符串在两种情况下是一样的：

```
"<Catalog><Legalese> © 1999 Wrox Press, Ltd. ® </Legalese></Catalog>"
```

第一种方法明显要易于读写一些，而且还有一个额外的好处，就是允许在任何地方都可以直接剪切粘贴XML代码。后一种方法仅仅是用正确的实体引用来代替两个标记字符，这样在解析时它们就不会被错误地解释为元素标记或实体引用（如果希望详细了解解析和解析器，参见后面的章节）。

2.10 文档结构

根据你可能希望完成的操作，我们已经零敲碎打地了解到可能在XML文档中使用的许多语法，下面让我们来看一看文档的整体逻辑结构。

2.10.1 序言

XML文档是以序言开头的。它用来表示XML数据的开始，描述字符的编码方法，为XML解析器和应用程序提供其他一些配置线索。

序言的组成包括：一个可选的XML声明（下面就会介绍），可能紧跟着几个（或者没有）注释、处理指令、空白字符，其后可能有一个可选的文档类型声明（再带着几个可选的注释、处理指令和空白字符）。由于这些内容都是可选择的，所以就意味着序言可以被省略，而文档仍然是格式正规的（参见XML 1.0推荐标准的2.8节）。

我们先来看一看序言的第一个组成部分。

1. XML声明

所有的XML文档可能（也应该！）由一个XML声明（XML Declaration）开始。虽然文档声明使用的是同指令处理类似的语法，但从技术上讲，根据XML推荐标准它们并不是一回事，因为声明是XML中的保留部分。

```
<?xml version="1.0" ?>
```

如果包括XML声明，它必须处在文档最前面——前面不允许有任何空白或注释。严格地讲，在XML当中这种声明并不是必须的，但我们后面会看到，当处理文档时，它确实会起到一些优化的作用。

XML早期的草案并没有要求名称大小写敏感，所以许多早期实现者，包括微软在内，用的都是声明的大写版本（“<?XML ... ?>”）。但是，最终的W3C推荐标准提出了大小写敏感的要求，并将“xml”规定为小写。这样一来，某些所谓的XML文档就不再是合法的XML 1.0数据。

下面的例子是一个XML声明的完整语法（包括可选的属性 encoding 和 standalone）：

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

这些属性已经在XML 1.0规范中做出了定义：

- version——不能省略；值必须为“1.0”；该属性用来保证对XML未来版本的支持。
- encoding——可选；值必须是一种合法的字符编码，例如“UTF-8”、“UTF-16”或者“ISO-8859-1”（即Latin-1字符编码）。所有XML解析器都要求至少支持UTF-8和UTF-16。如果没有包含这个属性，就假设是“UTF-8”或“UTF-16”编码，这取决于开始的“<?xml”字符串的格式。参见本章后面的字符编码部分。
- standalone——可选；值必须是“yes”或“no”；如果是“yes”就意味着所有必须的实体声明都包含在文档中，如果是“no”就意味着需要外部的DTD。（参见第3章关于DTD的部分。）

与大多数XML属性不同，这些必须按上面的顺序依次出现。另一方面，也是与大多数XML属性不同，encoding属性值不是大小写敏感的。这种不一致主要是因为XML对现有ISO和IANA标准关于字符编码命名的依赖。

针对XML的、支持最广泛的字符编码包括：

- 统一代码：“UTF-8”，“UTF-16”
- ISO 10646：“ISO-10646-UCS-2”，“ISO-10646-UCS-4”
- ISO 8859：“ISO-8859-n”（其中的n代表数字1到9）
- JIS X-0208-1997：“ISO-2022-JP”，“Shift-JIS”，“EUC-JP”

虽然HTTP为服务器联络客户端（浏览器）提供了一种方法，编码的作用也正在于此，但有时候并没有服务器的存在（比如在我们察看本地文件系统上的文件时）。使用encoding属性也比试图依靠自动检测字符编码要可靠得多，后者已经很常见，但无法区分UTF-8和ISO-8859-1，也不能可靠地检测出UTF-7编码。

下面，我们再细致地了解一下编码的意义。

(1) 字符编码

统一代码原本是设计成一个简单的16位的标量值集合，因为它要包含任何现代语言必需的大多数特殊符号或字符。尽管这65 536个字符并没有全部用上，但很明显需要的字符会越来越多，而统一代码也增加了代理块（surrogate block）机制来增加1 048 576个额外字符（稍后我们将详细阐述）。

虽然统一代码使用标量值识别字符，但存储这些数字还是有不同的办法。XML需要在网络中广泛使用的设计目的要求有各种各样的编码方法。

UTF-8能够正常地处理7位的ASCII码，但对于其他任何内容都需要2到8个字节（每个的变化范围都是从80到FF）——对于占统治地位的ASCII文本来说相当不错，但对于除此之外的事物来说则不是什么好消息。由于UTF-8对非ASCII数据支持不够，而且Java是构建在统一代码字符集之上，所以对于国际化的应用来说，UTF-8并不是一个好的选择。“UTF”意味着“统一代码传输格式”（在统一代码文档中是这样描述的），或者“USC传输格式”（在ISO 10646中是这样描述的）。

UTF-16编码最为简单，它用两种传统方式（大尾数法和小尾数法）中的一种来存储 16位的字符（这取决于计算机处理器的体系结构）。

其他字符可能使用代理块（参见下节）编码。对 UTF-16的关注主要来自东亚，在那里有时需要用两个编码单元来表示一个字符。但是，亚洲字符当中的绝大部分都只需要一个编码单元。根据统一代码 FAQ，代理对的使用要比所有文本存储的 1%少许多。所以，大多数构造于统一代码之上的软件使用的都是 UTF-16编码。

UTF-32编码（几乎等同于 ISO 10646 UCS-4编码）可以简化编程工作——所有的统一代码字符（以及大约 40 亿个附加字符）能够使用这种方法编码。虽然对于某些语言 / 平台来说能够简化编程，但这并不是一种理想的选择，因为用这种编码每个字符还是需要双倍的存储空间（也就是 ASCII 字符存储空间的四倍）。

ISO 10 646和统一代码之间的关系在<http://consult.cern.ch/cnl/215/node47.html>有详细描述，ISO 8859也在<http://ppewwww.ph.gla.ac.uk/~flavell/iso8859/iso8859-pointers.html>有相关的讨论。

(2) 代理块

很明显，65 536个字符可能足够支持大多数现代语言，满足学者和历史学家的需要，具备表现语言充满活力的天性，但同时它也需要更大的空间。例如，统一代码添加了大约 14 500个复合字符以兼容现有的字符集；此外还有许许多多数学符号、亚洲象形文字、历史字符没有包含在统一代码当中。所以，统一代码增加了代理块（surrogate block）。

代理块是统一代码用来扩展超出现有 16位数字的字符空间的办法。它保留了两个 1 024个字符的区域（D800到DBFF和DC00到DFFF）以容纳 1 048 576个附加字符。第一个区域内的 16位值是一个 32位数字的高端部分，而第二个则是低端部分。这就允许并不理解这些组合字符的程序能够简单地显示一个或两个占位符。这些扩充字符能够用于现代象形文字语言（例如汉语和日语），古语（例如埃及的象形文字），甚至新发明的语言（Klingon、Tolkien的Dwarvish码）。

但无论是好是坏，代理块字符对并不是合法的 XML 字符。

(3) 字节顺序掩码（Byte-Order Mark、BOM）

统一代码数据利用 BOM 作为信号来指定所使用的编码方法（参见表 2-4）。

表 2-4

字 节	编码形式
00 00 FE FF	UTF-32/UCS-4，大尾数法
FF FE 00 00	UTF-32/UCS-4，小尾数法
FE FF 00 ##	UTF-16，大尾数法
FF FE ## 00	UTF-16，小尾数法
EF BB BF	UTF-8

虽然这些值在 XML 数据中并不是合法的，但它们能够把有效的 XML 数据加入到被传输的数据对象当中，并为 XML 解析器和 / 或应用程序提供有价值（虽然通常是冗余的）的编码信息。XML 解析器必需在 XML 数据之前处理 BOM，因为需要同时支持 UTF-8 和 UTF-16 编码。

下面，让我们看一看序言的第二种可选的组件。

2. 文档类型声明

我们不能把它同文档类型定义（也就是通常所说的 DTD）相混淆的。而且，文档类型声明

中包含着文档类型定义的内部子集和/或到外部子集的引用。

所有有效的XML文档必需包含这个声明，但简单的格式正规的文档并不需要这样的包含关系，而且它们也不包含（除了5个标准XML实体）任何实体引用。如果使用了附加的字符实体，在将用非验证型的解析器解析的、格式正规但并不有效的文档中，它们可能在一个独立的XML数据对象中的内部子集中声明，而不需要使用外部DTD或模式。

(1) 外部子集引用

```
<!DOCTYPE root_element_name SYSTEM "system_identifier">  
<!DOCTYPE root_element_name PUBLIC "public_identifier" "system_identifier">
```

有两种形式的引用DTD的外部子集的文档类型声明：

root_element_name是DTD的附着点。XML文档的元素树中的所有子元素都继承这个根节点的DTD声明。

SYSTEM（上面例子中的“system_identifier”）和PUBLIC（上面的“public_identifier”）标识符的值都是URI（Uniform Resource Identifier，统一资源标识符）。在本文撰写时，在大多数实践中，URI都可以和URL等同看待；虽然它的形式是“http://www.wrox.com/myfile.dtd”——但实际上可以是任何能够被负责处理的应用程序识别出来的唯一名称。

PUBLIC标识符是用来引用某些DTD的，而它们是使用某些在XML中没有定义的方法来分类和检索的，但这只能是一个结构中的某种标准，或者是交换XML数据的团队之间的协议。这种不会像URL那样成为一种盲目的引用，但它的缺点在于其适用范围不能超出相关机构的范围。

XML解析器会试图使用PUBLIC标识符生成URI。但如果做不到这一点，就会使用SYSTEM标识符的URI。这种形式的缺点在于URI可能失去作用（域名和文件路径可能改变，或者被废弃）而DTD也找不到了。

由于通常使用的分段标识符（字符#）并不真正是URI的组成部分，所以如果在SYSTEM标识符中出现分段标识符时可能会提示错误。任何包含非ASCII字符的URI都必须用UTF-8中的字符表示（可能用一个或多个字节），然后使用标准的URI“%HH”转义序列来处理它们当中的每一个字节（这里“HH”是十六进制值）。

下面是两个引用外部DTD的实例：

程序清单 2-11

```
<?xml version="1.0" standalone="no"?>  
<!DOCTYPE Catalog SYSTEM "http://www.wrox.com/DTDs/PubCatalog.dtd">  
<Catalog>  
  ...  
</Catalog>
```

或：

程序清单 2-12

```
<?xml version="1.0" standalone="no"?>  
<!DOCTYPE Catalog PUBLIC "-//PubCatalog"  
  "http://www.wrox.com/DTDs/PubCatalog.dtd">  
<Catalog>  
  ...  
</Catalog>
```

“标识符”是XML从SGML继承过来的另一小部分内容。

SGML的“公共标识符”提供了一种方法，允许文档引用一个本地目录（或者使用其他更好的办法）来查找DTD。这当然是一个非常有用的技术，但是在设计XML时，并没有就解决公共标识符的通用办法达成一致。结果是，XML允许使用公共标识符，但需要系统标识符作为备份引用。

(2) 内部子集声明 如果没有外部DTD可用，XML数据对象可以仍然使用实体引用，只要它们在DTD的内部子集中声明过，后者是通过使用 `<!DOCTYPE...>` 的扩充形式来实现的。DTD语法是下一章的主题，所以届时将揭示出更多的细节信息。作为基本常识，我们应该知道内部子集的声明是在 `<!DOCTYPE...>` 使用中括号 [...] 分隔的，而 `<!ENTITY...>` 声明则用来定义实体引用的扩充部分。

下面是一个内部子集声明的简单例子，它涉及三个符号字符实体和两个“文本宏”或“样板”实体：

程序清单 2-13

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE Catalog [
  <!ENTITY copy "&#169;">
  <!ENTITY nbsp "&#160;">
  <!ENTITY reg "&#174;">
  <!ENTITY COPYDATE "1999">
  <!ENTITY WROX "Wrox Press, Ltd.">
]>
<Catalog>
  <Legalese>&nbsp;&copy; &COPYDATE; &WROX; &reg;</Legalese>
</Catalog>
```

上面的例子如果在IE5中显示的话（如果用户没有指定使用的样式单），会得到如图2-8所示的画面。

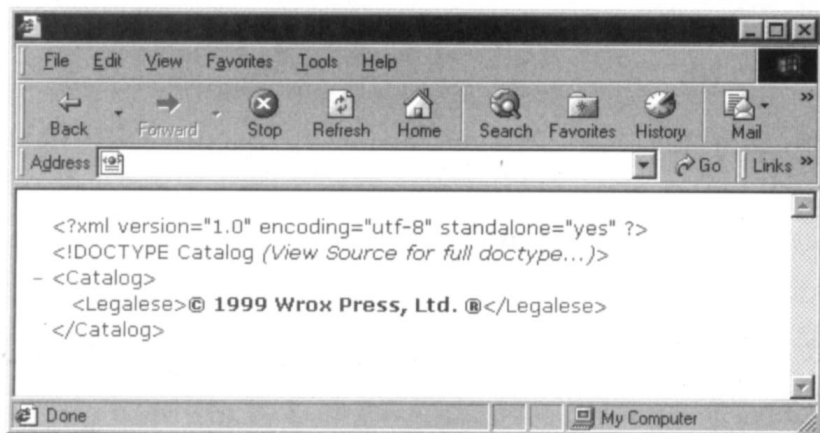


图 2-8

XML文档可能使用包含内部和/或外部子集的DTD。如果两种子集同时使用，而且有同一个元素类型或实体的两个声明，内部子集优先。

现在我们已经讲解完了语言的语法和格式正规的文档的核心语法，下面我们很快地浏览一下XML的第三个部分。

2.10.2 尾声

XML尾声可能包含注释、PI和/或空白。PI不一定非要应用到文档中尾声之前或接下来的数据中（如果有的话）的元素中。

这种方案可能本身就是自找麻烦，或者说它本身就存在着问题。因为 XML没有定义任何文档结束指示符号，所以大多数应用程序把文档元素的结束标记用于这种目的。这样一来，网络链路可能在遇到根元素的结束标记时就关闭了，而不再对尾声进行任何处理。而且，文档之间的处理指令在某种程度上也有些模糊不清。

尾声被Tim Bray（XML 1.0推荐标准的作者之一）认为是一个“真正的设计错误”——如果没有很好地利用，使用它可能是一种错误，我们应当注意到它不太可能具备与其他 XML应用程序的互操作性。

下面，我们应该好好回顾一下我们已经学到的关于 XML的语法了。

2.11 XML语法小结

XML标记的形式参见表2-5。

表 2-5

语 法	组 件
<tagname>	元素起始标记
<tagname attribute="value">	带有属性的起始标记
</tagname>	结束标记
<tagname/>	空元素标记
<tagname attr1="value1" attr2="value2"/>	带有两个属性的空元素标记

特殊的XML指令和声明参见表2-6。

表 2-6

语 法	组 件
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>	XML声明
<?name string ?>	用于XML应用程序的处理指令（PI）
<!-- string -->	注释，供撰写者或编辑者使用
<![CDATA[string . . .]]>	未解析的字符数据（CDATA）部分
<!DOCTYPE string . . .>	文档类型声明（不是DTD）

XML实体引用参见表2-7。

表 2-7

语 法	组 件
&#decimal; 例如: ®	使用十进制数字表示的字符编号的引用
&#xHEX; 例如: ૴	使用十六进制数字表示的字符编号的引用
&ref;	对预先定义的 XML 实体（如文本宏）的引用

现在我们了解了可能在格式正规的文档中使用的各个部分，下面让我们讨论一下“格式正规”到底意味着什么。

2.12 格式正规的文档

所有遵守 XML 语法规则的数据对象（文档）都是格式正规的 XML 文档。这类文档在使用时可以不使用 DTD 或模式来描述它们的结构，它们也被称作独立的（standalone）XML 文档。这些文档不能够依靠外部的声明，属性值只能是没有经过特殊处理的值或缺省值。

一个格式正规的 XML 文档包含一个或多个元素（用起始和结束标记分隔开），它们相互之间正确地嵌套。其中有一个元素，即文档元素，包含了文档中其他所有元素。所有的元素构成一个简单的层次树，所以元素和元素之间唯一的直接关系就是父子关系。兄弟关系经常能够通过 XML 应用程序内部的数据结构推断出来，但这些既不直接，也不可靠（因为元素可能被插入到某个元素和它的一个或多个子元素之间）。文档内容可能包括标记和/或字符数据。

数据对象如果满足下列条件就是格式正规的文档：

- 语法合乎 XML 规范。
- 元素构成一个层次树，只有一个根节点。
- 没有对外部实体的引用，除非提供了 DTD。

任何 XML 解析器如果发现在 XML 数据中存在并不是格式正规的结构，就必须向应用程序报告一个“致命”错误。致命错误不一定导致解析器终止操作——它可以继续处理，试图找出其他错误，但它不再会以正常的方式向应用程序传递字符数据和/或 XML 结构。

之所以采用这类错误处理方式，一是因为 XML 简洁的设计风格，二是因为 XML 更多的不是用于显示——因为这不那么容易使得 XML 数据对象做到格式正规。这种近乎残酷的错误处理会阻碍类似 Internet Explorer 和 Navigator 之类臃肿的软件的创造性（它们拥有各类特殊代码来解决如何处理意义含糊的 HTML）。

对于 HTML/SGML 来说，它们的工具都要比 XML 宽容许多。HTML 浏览器通常会显示出大多数支离破碎的 Web 页面，这为 HTML 的快速流行做出了巨大贡献。此外，真正的显示会因浏览器而异。同样，SGML 工具即使遇到错误，通常也会尽力继续处理文档。

格式正规的文档的存在使得可以使用 XML 数据而不必承担构建和引用外部描述的重任。术语“格式正规”与正式的数学逻辑有着相似之处——一个命题如果满足语法规则就是格式正规，而不在于它的正确与否。

2.13 解析器

除去指定XML的语法，W3C推荐标准还描述了XML客户端体系结构（XML处理器或解析器）中低层次的一些行为。目前，有两种类型的解析器：

- 非验证的（non-validating）——解析器仅仅保证数据对象是格式正规的XML
 - 验证的（Validating）——解析器使用DTD保证格式正规的数据对象的形式和内容的有效性。
- 一些解析器同时支持两种类型，其中有判定数据对象是否是是需要验证的文档的配置开关。

XML解析器的一些行为在定义时是为了减少应用程序处理XML数据的负担。例如，正如我们在前面描述的那样，用来界定文本记录结束位置的字符序列通常是因操作系统而异的。但是一个XML应用程序不必关心这个，因为XML解析器会把所有标准的文本记录分隔符格式化为一个换行符（十六进制值0A）。空白的处理是解析器严格要求的另一个领域——与HTML或SGML不同，所有的空白都必须从文档传递给应用程序。而且，正如外部或内部DTD子集所描述的那样，普通的（字符）实体会通过解析器加以扩充。

XML解析器在把属性值（即下面的AttValue）传递给XML应用程序之前需要对它们格式化。这意味着解析器将需要处理下面的引用和字符：

- 字符引用——将引用的字符添加到AttValue当中。
- 实体引用——递归地展开实体的替换文本，添加到AttValue当中。
- 空白字符——替换任何回车/换行字符对，它们可能是外部解析实体的组成部分，内部解析实体的字符串实体值，或者是空格字符（十六进制值20）代表的任何空白字符，然后在AttValue当中添加空格。
- 其他所有字符——将字符添加到AttValue当中。

接下来，AttValue会做进一步处理，删除掉前面和/或后面的空格，然后把多个空格序列转化成单个空格。这个规则的唯一例外就是如果属性值在DTD中被声明为CDATA，而且使用的是需验证的解析器（参见第3章关于DTD的描述和明确的声明）。

实现XML解析器有两种方法。虽然针对它们的争论非常之多，但是每种方法都有它们的优势。与许多其他现实世界中的问题一样，XML的处理的需求会有着极大的区别，所以不同的方法会适合不同的解决方案。

2.13.1 事件驱动的解析器

处理XML数据的一种方法是使用事件驱动的解析器，其模式对于现代GUI（图形用户界面，Graphical User Interface）和OS（操作系统，Operating System）的编程者来说实在是再熟悉不过了。在这种情况下，XML解析器为每类XML数据执行到应用程序的回调：元素（及其属性）字符数据、处理指令、符号或者注释。这等于是应用程序通过回调来处理XML数据——XML解析器在解析之后并不维护元素的树结构，或者任何数据。事件驱动的办法对系统资源的需求及其普通，即使对于非常大的文档也是如此；正是得益于这种简单的、对XML数据结构的低层次的访问，XML应用程序在处理数据时具有相当大的灵活性。

在这类解析器中，最早的、也是最为著名的一个一流程序就是James Clark的expat，它用

ANSI-C写成。它还有相应的C++ (expatpp) Perl (XML::Parser) 和Python (Pyexpat) 包装器。Clark是XML 1.0推荐标准的技术领袖, 他还曾经用Java编写了另一个解析器xp。

由David Megginson (Microstar的Aelfred解析器的作者) 领导的XML-DEV邮件列表的成员开发了一个基于事件的XML解析的标准接口, 即所谓的SAX 1.0: The Simple API for XML (SAX), 现在已经用Java、Perl和Python实现, 而且支持大量不同的XML解析器。SAX是Peter Murray-Rust将三个不同的XML解析器 (每个都有专用的API) 集成到JUMBO的结果——他计划为解析XML创建一个通用的、基于事件的Java API。他开始与Tim Bray和David Megginson进行了设计上的讨论, 而后者在1998年初用了近一个月的时间编写了最初的Java实现。此后, SAX通过许多人的努力得到了扩充, 成为了一种强大的XML数据处理办法 (大多数事件驱动的解析器现在都支持/使用这种API)。我们将在第6章详细讨论SAX。

2.13.2 基于树的解析器

软件工程中使用最为广泛的一种结构就是简单的层次树。所有格式正规的XML数据都被定义为类似的树, 这样一来, 不论是普通的还是成熟的算法都可以用来遍历XML文档的节点、搜索内容、编辑文档树。这些树的算法得益于数年的理论研究和商业开发。使用这种方法的XML解析器通常遵守W3C的文档对象模型 (Document Object Model, DOM)。DOM是一种独立于平台、语言的接口, 它能够对树结构的文档进行操作。另一方面, DOM树在文档能够被操作之前必需在内存中生成——所以高性能的虚拟内存支持对于大型文档来说是非常必要的。一旦树被生成, 应用程序就可以通过相关的API访问DOM。我们将在第5章详细讨论DOM。许多这类解析器事件都建立在事件驱动的XML解析器的基础之上, 它们提供了为构建DOM树需要的所有信息。

Microsoft曾经开发出一个被称作MSXML的基于Java的早期XML解析器。之后, Internet Explorer 5 (IE5) 包含了对XML的支持, 但使用的是另一套解析器。微软推荐将DataChannel (它吸收了IE5的解析器的功能特性) 提供的XJParser作为IE4的更新组件。此外, Microsoft的Office 2000还把XML当作自己的数据交换格式使用, 这极大地促进了XML的推广。同时, 要明确的是, 虽然XML的许多方面与推荐标准的最终要求还有一定差距, 但Microsoft正在走到这些标准的前面, 而且在Microsoft的实现形式与正式的W3C标准之间存在不少差异。

Netscape承诺在下一代浏览器中将完全支持XML 1.0和XSL, 这正在作为一个开放资源项目由Mozilla组织着手进行。该浏览器当前的代号为“SeaMonkey” (使用NGLayout/Gecko布局引擎)。目前, 它正处于开发当中 (尚未进入beta测试阶段), 而且最近的公司变动也给AOL/Netscape的浏览器的未来蒙上了阴影。

Fujitsu Laboratories曾经开发出一种高级SGML/XML浏览器HyBrick。它支持高级的链接和格式功能, 它使用的是DSSSL (ISO 10719) 显示法和XLink/XPointer引擎。该浏览器基于James Clark的SP和Jade, 能够处理有效的和格式正规的XML文档。大家可能想象得到, 这个浏览器对日文的处理和英文一样优秀。

2.13.3 解析器基准测试

两位开发者最近完成了XML解析器的基准测试, 它检验了Linux和Solaris系统上几种不同的解

析器。结果（可能正如大家预料的一样）表明用C语言编写的产品（尤其是James Clark的expat）仍然是最快的，然后是Java，最后是脚本语言（Perl和Python，两个实际上都是expat解析器的变种）。

Linux基准测试是由Clark Cooper执行和汇总的，他测试了6种解析器：

- Expat（James Clark用C语言编写）
- RXP（Richard Tobin用C语言编写）
- XP（James Clark用Java语言编写）
- XML4J（IBM用Java语言编写）
- XML::Parser（Clark Cooper用Perl编写）
- Pyexpat（Jack Jansen用Python编写）

Solaris基准测试是由Steven Marcus执行和汇总的，它测试了上面除去RXP的所有产品，再加上Sun的“xml-tr2”解析器（Javasoftware用Java写成）。这些可以通过以下地址得到：

- Linux测试——<http://www.xml.com/pub/Benchmark/exec.html>
- Solaris测试——<http://www.awaretechnologies.com/XML/xmlbench/solaris.html>

下面我们看一看针对特定问题的XML应用程序。

2.14 书籍目录应用程序

书籍目录应用程序是XML如何用于传统文档标志和更为普通（和强大）的数据建模的简单例子。这个XML词汇表包括典型的书籍目录元数据，也为向用户的搜索要求发送结构化的价格信息提供了基础，它实现了在WWW上进行书籍交易，能够为将书籍运送到商店生成包装清单和运送标签，还可以在出版商和书店之间交换财务和订单数据。我们将在下一章花费更多的时间来分析这个应用程序，最终设计出一个能够更加符合目录需要的DTD。而现在，我们能够以更简单的方式来解决这个问题。

让我们先从这个应用程序中的基本要素——书说起。根据本例的要求，我们忽略掉书本的实际内容，但当然文本标记仍是SGML和它的后继者（比如XML）的传统用法。书籍包含一些普通的目录元数据，例如作者、出版商、出版日期、版权等。

一旦我们开发出书籍的合理结构，我们就会在另一个层次——目录对其进行修改。这一层与其说是文本文档，到更不如说它像传统的数据库，因此也包括了XML数据的一些建模能力。

<Book>元素

首先，我们只使用简单的子元素来创建<Book>元素：

程序清单 2-14

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--===== The Wrox Press Book Catalog Application =====>
<Book>
  <Title>Professional XML</Title>
  <Author>Mark Birbeck, Steven Livingstone, Didier Martin,
    Stephen Mohr, Nikola Ozu, et al.
  </Author>
  <Publisher>Wrox Press, Ltd.</Publisher>
```

```

<PubDate>November 1999</PubDate>
<Abstract>XML 0-500kmh in 3 seconds</Abstract>
<Pages>750</Pages>
<ISBN>1-861003-11-0</ISBN>
<RecSubjCategories>
  <Category>Internet</Category>
  <Category>Web Publishing</Category>
  <Category>XML</Category>
</RecSubjCategories>
</Book>

```

在上面的例子中有一个明显的错误：所有合作作者都被汇总到了一个元素中。让我们 来改进一下：

程序清单 2-15

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--===== The Wrox Press Book Catalog Application =====>
<Book>
  <Title>Professional XML</Title>
  <Authors>
    <Author>Mark Birbeck</Author>
    <Author>Peter Stark</Author>
    <Author>Steven Livingstone</Author>
    <Author>Didier Martin</Author>
    <Author>Stephen Mohr</Author>
    <Author>Michael Kay</Author>
    <Author>Nikola Ozu</Author>
  </Authors>
  <Publisher>Wrox Press, Ltd.</Publisher>
  <PubDate>November 1999</PubDate>
  <Abstract>XML 0-500kmh in 3 seconds</Abstract>
  <Pages>750</Pages>
  <ISBN>1-861003-11-0</ISBN>
  <RecSubjCategories>
    <Category>Internet</Category>
    <Category>Web Publishing</Category>
    <Category>XML</Category>
  </RecSubjCategories>
</Book>

```

现在，已经明确地把作者们分开了，所以让我们看一看目录元数据是如何显示的。例如，我们可能争论书名、ISBN等内容都是<book>元素的属性，而不是内容的一部分。

出于说明的需要，我们把这些元素改变成了属性（虽然最好的办法与此效果相当），并且把某些其他的<Book>子元素改为属性。我们还增加了一个<Price>元素以便更深入地说明属性的使用：

程序清单 2-16

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--===== The Wrox Press Book Catalog Application =====>
<Book title="Professional XML"
  publisher="Wrox Press, Ltd."
  pubdate="November 1999"
  pages="750"

```

```
isbn="1-861003-11-0">

<Authors>
  <!-- 'Author' elements deleted for brevity -->
</Authors>
<Price currency="USD">49.99</Price>
</Book>
```

很明显，在这个例子中有许多遗漏的地方。一个一直存在的不足就是 `publisher` 属性和 `<Author>` 元素的字符串常量的使用。我们将在下一章解决它和其他问题，此时书籍目录的概念已经得到很大的扩充。

前面的例子解释了元素和属性的处理原则：元素是代表对象的名词，而属性是描述这些对象的特点的形容词。

你可能注意到我们使用了一种代码风格的约定来分隔这两个对象：元素类型名使用适当的名词形式（第一个字母大写）；属性名都是小写。这有助于增强正在命名的内容，特别是当我们在注释或其他文本中讨论这些命名时尤其如此。

元素和属性的配合使用并不是件简单事情。在 XML-L 和 XML-DEV 新闻组中就这个主题曾经出现过许多讨论和争论。某些人认为根本就不应该使用属性——因为它们增加了不必要的处理复杂性，任何用属性代表的东西都可以更好地包含在子元素中。其他人则非常欣赏使用 DTD 验证属性值和设置缺省值所带来的好处。最近的试验表明，抛开表面现象，使用普通的数据压缩（例如 gzip、LZW 或者 zlib）都不具备数据存储和传输方面天生的优势（也就是说，压缩过的 XML 数据对象的大小没有什么变化）。

XML 1.0 推荐规范的两位编辑者和其他 SGML/XML 专家曾经就这个主题撰写过文章——通过以下链接大家可以看到这些内容（大多数都深藏在 Robin Cover 的 SGML/XML Web 页下）。

- Andrew Layman 的 “XML Syntax Recommendation for Serializing Graphs of Data” ——
<http://www.w3.org/TandS/QL/QL98/pp/Microsoft-serializing.html>
- Eliot Kimber 的 “Elements or attributes?” ——
<http://www.oasis-open.org/cover/attrKimber9711.html>
- Michael Sperberg-McQueen 的 “Element vs. Attributes” ——
<http://www.oasis-open.org/cover/attrSperberg92.html>
- Robin Cover 的 “SGML/XML: Using Elements and Attributes” ——
<http://www.oasis-open.org/cover/elementsAndAttrs.html>
- Tim Bray 的 “When is an attribute an attribute?” ——
<http://www.oasis-open.org/cover/brayAttr980409.html>
- G. Ken Holman 的 “When to use attributes as opposed to elements” ——
<http://www.oasis-open.org/cover/holmanElementsAttrs.html>

2.15 小结

在本章中，我们向大家演示了所有 XML 数据必需的基本语法。我们在没有其他任何知识或

工具的情况下就能够创建简单的、格式正规的文档，但这还没有开始充分利用 XML 的强大功能。

例如，在基本语法中并没有提供 HTML 中“HT”——超文本。在本书后面的部分（第 8 章）中讨论的一些待定的 XML 扩展中，改进的链接语法是非常重要的一个方面。

格式正规的文档对于一些应用程序来说已经足够，但这意味着任何数据的解释或验证都必须硬编码在应用程序中。一种更方便的办法就是提供第二个文档用来验证第一个。这个验证文档的形式可能是 DTD（参见第 3 章）也可能是模式（参见第 7 章）。

在简单的情况下，XML 是一种强大的数据交换介质。在使用 DTD 或模式、命名空间、链接和样式单扩充后，XML 将是更为强大的 Internet 时代的基础。再配合以 Java 或其他合适的语言，XML 将使得计算程序能够更为方便、使用范围更广。