

第3章 文档类型定义

上一章介绍了如何编写格式正规的 XML 文档。然而，当你开发符合 XML 1.0 的文档结构时，出现了这样一个有趣的问题：你如何与其他人交流你设计的结构？主流的浏览器已经支持或者正在准备支持 XML，但是这仅限于显示 XML 的内容。如果你开发的程序不仅用到 XML，而且创建了新的 XML 词汇表，即：将你的设计意图隐藏在代码中。那么，为了使 XML 1.0 的其他用户能够理解符合你创建的词汇表的文档的结构，作为 XML 词汇表的设计者，你必须通过某种通用的方式说明词汇表的语法规则。为此，XML 1.0 提供了一种机制——文档类型定义（Document Type Definition，DTD），并将其作为规范的一部分。DTD 使用正式的语法定义 XML 文档的结构和允许值。你在上一章看到的 XML 是格式正规的 XML。它符合 XML 的基本语法规则，并且没有其他任何语法约定。在本章，我们将创建有效的 XML：它不仅遵循 XML 的语法规则，而且受到你所创建的词汇表规则的约束。

DTD 将带来以下优越性。首先，通过创建 DTD，能够正式而精确地定义词汇表。所有词汇表规则都包含在 DTD 中。凡是未在 DTD 中出现的规则都不属于词汇表的一部分。许多解析器可以利用 DTD 验证文档实例的有效性。只要在文档实例中写入一条简单的声明语句，解析器就能够获取 DTD，并将其中的内容与文档实例进行比较。另外，XML 创作工具也可以通过类似的方式使用 DTD。一旦选择了 DTD，创作工具就能够实施 DTD 中的规则，它根据 DTD 中说明的结构，仅允许用户在文档中添加 DTD 允许的元素或属性。

XML 1.0 推荐标准专门描述了如何构建 DTD，以及如何将它与根据其中规则编写的文档相关联。它还定义了解析器应该对 DTD 执行的处理。在本章中，我们将讨论使用 DTD 的原因。除此之外，我们还将介绍 XML 1.0 DTD 的语法规则，以及如何在文档实例与 DTD 之间建立关联。利用以上知识，我们将为有关图书目录的例子创建 DTD。

3.1 为何需要正式的结构

当你编写的代码要对以特殊词汇表为依据的文档进行操作时，实际上你是在创建文件资料。你的源代码中溶入了词汇表规则。代码必须遵循某种结构；当结构改变时，必须修改代码。通常情况下，这是可以接受的。设计者可以将他的设计意图传达给一小组程序员，应用程序中的所有代码都将遵照这些假设进行编写。毕竟，编写完全数据驱动的代码是非常困难的。

然而，如果没有显式的文件资料，就无法可靠地捕获文档中的错误。唯一的错误检测机制就是运行代码。如果你的代码执行通过，或者文档以一种出乎意料的方式偏离了设计者的初衷，就很难检测出错误。最终，你的应用程序将无法实现预期的目标。

为了解决上述问题，需要依靠清晰、准确的语法规则文档，它应该包含词汇表允许的所有规则。如果配备了这样的文档，程序员就不必为了确认对词汇表的理解程度与词汇表的设计者进行面对面的交流。如果文档本身也是用一种正式的（具有严格精确的格式）语法书写的，解

析器就能够阅读这些规则。由此形成了一种可靠错误检测机制。解析器能够指出任何检测到的词汇表错误，你可以先修改这些错误，然后再着眼于应用程序的逻辑。

3.1.1 文档域

XML文档可以看作是程序中数据结构的快照。它们用于程序之间的信息交流。这些信息都属于某个应用领域——你要解决的问题空间。如果你的 XML 词汇表所构建的模型非常适合于要解决的问题，就能够简化应用程序的编写和维护。为了设计出有效的 XML 词汇表，你必须深入分析应用程序要解决的问题。如果你的 XML 仅仅符合格式正规约束，可能很难明确地反映出商业过程。你不能理所当然地认为你的 XML 例子能够覆盖每种可能出现的情况。即使真的如你所愿，它们也不能以有效的方式传递你掌握的知识。

相反，DTD 能够通过定义记录词汇表中的所有信息。你在设计词汇表时考虑到的所有问题都必须写入 DTD。从而，其他人可以通过 DTD 了解你对问题的理解（或者至少可以知道你针对这个问题所记录下来的内容）。DTD 具有以下两个作用：将你掌握的知识提供给程序，同时获得了文件资料。

3.1.2 验证文档的有效性

如果格式正规的文档是遵循一些隐式规则编写的，解析器无法根据这些规则检查其中的错误。整个系统的完整性取决于创建和使用 XML 的应用程序的完整性。代码中的错误可能很难被发现。它们还可能引起其他程序的中断，或者导致错误的数据进入系统。然而，XML 1.0 推荐标准规定了验证有效性的解析器应该具有的功能。如果某个 XML 文档引用了 DTD，验证有效性的解析器应该读取 DTD，并确保文档符合 DTD 中描述的语法。如果你需要完善的错误检测机制，只需使用 DTD 和验证有效性的解析器。文档语法、词汇表以及指定值中的任何错误都逃不过解析器的眼睛。如果文档顺利通过解析器的有效性验证，你就可以放心大胆地考虑程序逻辑，不必再纠缠于语法问题。当然，有效性验证并不能避免应用程序逻辑方面的失误，但是它能够过滤出代码中的无效数据。

对于 Internet 应用程序来说，这一点尤为重要。你不能假设你要处理的应用程序经历了与你的代码同样严格的质量控制。为另一个企业服务的编程小组可能针对特定的业务或领域实现了公共的 XML 词汇表。他们对词汇表的解释可能与你的想法不同。他们的测试自然也与你的不尽相同。利用 DTD 和验证有效性的解析器，就能够立即对文档的完整性进行可靠的检查。当然，有效性检查的程度取决于 DTD。了解了以上概念，下面我们开始介绍如何编写有效的 DTD。

3.2 编写 DTD：通用原则

简单来说，XML 文档由元素和相应的属性组成。虽然我们还可以定义其他项，但元素和属性是文档支持的两个主要概念。此外，元素的内容是通过其他元素或 XML 标准中规定的基本类型进行定义的。DTD 必须能够定义文档中的所有元素，元素可以设置的属性，以及元素之间的关系。

3.2.1 将DTD与XML文档相关联

DTD是与文档相关的。通常，文档中包含一条用于与 DTD建立关联的指令，当验证有效性的解析器读到该指令时，它获取 DTD，并根据其中定义的规则对文档进行检验。下面我们将讨论如何在DTD与文档实例之间建立关联。

1. DOCTYPE标记

我们在第2章曾经简要讨论过这个标记。为了将 DTD声明与文档实例相关联，XML 1.0提供了特殊的DOCTYPE声明。DOCTYPE声明必须位于XML声明之后，且在任何文档元素之前。但是，XML声明和DOCTYPE声明之间可以插入注释和处理指令。

DOCTYPE声明包含关键字 DOCTYPE、文档根元素的名称，以及内容声明结构。在详细阐述多少有些晦涩的语句之前，我们先通过一个例子看一下 DOCTYPE声明在文档实例中的位置。以下是某个XML文档的前三行：

```
<?xml version="1.0"?>
<!DOCTYPE Catalog ...>
<Catalog>...
```

第一行的XML声明表示该文档符合XML 1.0的语法，第二行说明该文档使用Catalog词汇表——文档类型“Catalog”。更确切地说，文档的第一个元素或称根元素最好是Catalog，否则解析器会产生错误。在本例中，根元素恰好是Catalog。

程序段中的省略号隐藏了 DOCTYPE声明的其余部分。真正的声明到底在哪里呢？XML规范定义了两种提供声明的方法。你可以在独立的 DTD文件中提供外部子集声明，或者在 DOCTYPE声明体中包含内部子集，或者同时采用上述两种方式。在上例中（内部 DTD与外部DTD相混合的情况），内部DTD可以添加新的声明，或者覆盖外部 DTD中的声明。（根据XML规范的定义，解析器首先读取内部子集，其中的声明具有较高的优先权。）

在我们讨论如何提供声明之前，还有一个问题要考虑。正如我们在第2章所看到的，XML声明可以有 standalone属性。该属性可以取以下两个值：yes或no。如果属性值为yes，说明文档实例没有会影响到传递给应用程序的文档信息的外部声明。如果属性值为 no，说明文档有外部声明，且声明中包含的值是正确定义文档内容所必需的——例如，特殊的缺省值。

下面的代码是前一例子的变种，它表示我们需要的所有声明都包含在文档中：

```
<?xml version="1.0" standalone="yes" ?>
<!DOCTYPE Catalog ...>
```

在实际应用中，可选的standalone属性很少出现。属性值yes并不能保证文档没有任何类型的外部依赖，而仅仅意味着即使在处理过程中不考虑外部声明，在作为接收方的应用程序关注的范围内，文档不会产生错误（即使文档可能是格式正规的XML）。因此，它的主要用途是作为解析器和其他应用程序的标志，表示是否需要获取外部内容。

现在，让我们清除前两个程序段中的省略号。DOCTYPE声明由以下部分组成：关键字、文档的根元素名称（在本例中是Catalog）、可选的外部标识符，以及可选的标记声明块。外部标识符用于外部DTD（外部子集）的命名和定位，标记声明块是由标记声明（内部子集）构成的。首先让我们来讨论标记声明块。

2. 内部DTD子集

如果必要的话，我们可以将所需的全部声明都包含在内部子集中，正如你在上一章所看到的例子。DOCTYPE标记中的标记声明块由以下几部分构成：左方括号、声明列表和右方括号。下面是一个简单的例子：

```
<!DOCTYPE Catalog [ ...internal subset declarations here... ]>
```

内部DTD非常有用。你永远也不会为找不到 DTD 而头疼。然而，即便是最简单的 XML 词汇表，内部 DTD 也会令文档的长度剧增。另外，无论文档是由人编写的，还是由程序生成的，每个文档实例中都必须包含相同的内部 DTD。即使文档的用户不打算验证文档的有效性，这些声明也必须随文档一起传输。我们不推荐频繁地使用内部 DTD，但是对于简单的词汇表——特别是测试标记原型时，它们还是值得考虑的。

在某些情况下，设计人员可能希望同时使用内部 DTD 和外部 DTD。内部 DTD 用于添加声明。当内部 DTD 与外部 DTD 声明的项目重复时，内部声明将取代外部声明。这一特征使得设计者能够根据特殊文档的需求调整声明，但是使用该特征时应该谨慎。如果我们过度频繁地覆盖外部 DTD，就不太合适了——这充分证明了初始设计的失败。

3. 外部DTD

从某种角度讲，外部 DTD 更加灵活。上一章曾经介绍过如何引用 DTD；现在我们将进行更加详细的阐述。在本例中，DOCTYPE 声明不仅包含常规的关键字和根元素名称，而且包含指示外部 DTD 源的关键字和 DTD 的位置。XML 规范定义了几种声明源的方法。声明中可以使用关键字 SYSTEM 或 PUBLIC。如果使用 SYSTEM 关键字，解析器将仅根据给出的 URL 寻找 DTD——DTD 通过 URL 显式地直接定位。在我们的例子中，位于“SYSTEM”关键字之后的是用于命名 DTD 文件的 URL。

用于定位 DTD 的 URL 不应该包含段标识符（字符#加名称）。XML 1.0 建议指出，如果 URL 中包含该标识符，解析器将产生错误指示。

下面是两个例子：

```
<!DOCTYPE Catalog SYSTEM "http://myserver/decs/PubCatalog.dtd">
```

和：

```
<!DOCTYPE Catalog SYSTEM  
"http://www.universallibrary.org/publishing/PubCatalog.dtd">
```

在第一个例子中，对于包含的 DOCTYPE 声明的文档，验证其有效性所需的所有声明都位于文件 PubCatalog.dtd 中。在第二个例子中，DTD 文件位于虚构的世界图书馆组织的 Web 服务器上。在以上两例中，PubCatalog.dtd 文件应该包含 Catalog 元素的声明。

然而，如果 DTD 源关键字为 PUBLIC，情况就略微复杂一些。PUBLIC 关键字用于声明众所周知的词汇表。例如，假设出版界已经对图书种类 DTD 达成了大量共识。需要根据该词汇表解析文档的应用程序可能会通过某种策略来定位 DTD。如果这个 DTD 非常普及，应用程序或许有本地拷贝。直接使用本地拷贝可能比从 Web 服务器上远程下载更可取。DTD 可能存放在数据库中，或者可以通过其他与应用程序相关的技术获得。如果使用 PUBLIC 关键字和 URI，应用程序就有机会利用自己的算法定位 DTD。

统一资源标识符（URI）可以是URL，也可以是一个单独的名字。

例如：

```
<!DOCTYPE Catalog PUBLIC "universal/Publishing/Book">
```

如果URI “universal/Publishing/Book”对于处理这类文档的应用程序来说是已知的，应用程序可以通过某种有效的方式自行寻找 DTD。或许我们恰好有适合该领域的解析器。它可能有 DTD 的本地拷贝，或者它可以访问由本地数据库服务器维护的 DTD。最关键的是，寻找 DTD 的方法主要是由负责处理 DOCTYPE 声明的应用程序确定的。

当然，“众所周知”通常是相对的。因此，XML 1.0 允许 PUBLIC 声明同时有公共 URI 和系统标识符。如果使用文档的应用程序或解析器不能从 PUBLIC 关键字提供的 URI 定位 DTD，它必须使用系统标识符。

```
<!DOCTYPE Catalog PUBLIC "universal/Publishing/Book"  
    "http://www.universallibrary.org/publishing/PubCatalog.dtd">
```

在本例中，文档的设计者允许作为接收方的应用程序根据公共的 URI 自行寻找 DTD。如果该过程失败，对我们的出版领域不熟悉的通用解析器一般属于这种情况，应用程序可以从地址为 www.universallibrary.org 的 Web 服务器请求指定的文件。

3.2.2 基本标记声明

DTD 通过四种标记声明定义 XML 文档中允许出现的内容。表 3-1 显示了与这些声明相关的关键字及其含义。前两个声明与 XML 文档中的信息有关——元素和属性。

表 3-1

DTD 关键字	含 义
ELEMENT	XML 元素类型声明
ATTLIST	特定元素类型可设置的属性及这些属性的允许值声明
ENTITY	可重用的内容声明
NOTATION	不需要解析的外部内容（例如：二进制数据）的格式声明，以及用于处理这些内容的外部应用程序

后两种声明起辅助作用。特别是实体（ENTITY）用于简化 XML 词汇表的设计。它所包含的内容通常会在 DTD 或文档中反复出现，因此需要创建特殊的声明。该声明的作用类似于 C/C++ 中的 include 语句，它以特定的名称作为内容的替代符。

表示法（NOTATION）用于处理非 XML 内容。表示法用于声明特殊的数据类，并将之与外部程序相关联。这个外部程序就成为所声明的数据类的处理器。举例来说，如果你的文档与 JPEG 图像有关，你可能需要相关的显示程序来接收和展示 JPEG 二进制数据。当然，你的文档依赖于接收系统能够提供的处理器。有些设计者为了获得可移植性，宁愿放弃处理器引用。在那种情况下，表示法将退化为一种输入机制。

我们将在下一节详细讨论表示法和实体。

3.3 正式的 DTD 结构

现在，你已经对 DTD 有了一定程度的了解，并学会了如何将它与文档相关联。我们希望前

面介绍的例子能够激发起你进一步学习如何声明文档结构的兴趣。除了前面提到的四个标记声明，DTD还将用到其他结构。然而，我们首先来关注一下实体。

下面将要介绍的所有语法都在XML 1.0推荐标准（<http://www.w3.org/TR/REC-xml/>）中有明确的定义。有时，建议中的内容可能会令你感到迷惑，那么不妨看看由Tim Bray编写的XML规范解读（Annotated XML Specification），Tim Bray是XML推荐标准的作者之一。该文档位于<http://www.xml.com/axml/testaxml.htm>。它是用XML创建的，因此也不失为一个有价值的XML应用实例。

3.3.1 实体

XML提供了声明内容块的方法，你可以根据需要多次引用这些内容块，它不仅能够节省空间，而且能够减少文档创作者的代码输入量。为了在 DTD中声明实体，需要定义实体的名称及它引用的内容。当你需要使用它时，采用特殊的语法通过名称进行引用，这种特殊的语法能够说明你所提供的名称是实体引用。它类似于 C/C++中的define指令，或其他形式的可替换的样板文本或内容。文档内容中使用的实体称为通用实体（general entity）。我们可以根据是否解析实体的内容将定义进一步细化。解析实体（parsed entity）是XML内容。实体的值称为置换文本。相反，未解析实体（unparsed entity）可以是非文本内容。即使它是文本，并不一定要求是XML。这就是“未解析”一词的来历。如果你知道用于替换的内容不是XML，或者甚至不是文本，那么解析器就没有必要对它进行处理。另一方面，解析实体是要粘贴到文档内容中的XML，因此，解析器就必须将它传递到文档中。

下面我们将详细地讨论实体的分类，再次重申并扩展上一章介绍的内容。

1. 预定义实体

XML必须保留某些字符用于本身格式的定义，例如：尖括号。另外，有些字符是不可打印的。鉴于此，XML提供了一些预定义的实体，用户可以利用这些实体在文档中使用上述字符，并保证不产生冲突。因此在元素的文本内容中，可以用实体表示一些特殊字符，以免它们在解析时与文档的标记混淆。

表 3-2

字 符	实体引用
<	<
>	>
&	&
'（单引号）	'
"（双引号）	"

任何字符都可以表示为数字引用。具体方法是在符号“&#”之后加上字符的数字值和分号（它们之间没有空格）。例如，大于号可以表示为>。对于使用频率极高的字符，XML提供了预定义的实体（参见表3-2）。

例如：

```
<some_math>We have Arthur&apos;s variable x, where 10 &lt; x &gt; 46 </some_math>
```

2. 通用实体

通用实体是最简单的实体形式。它能够声明与某个名称相关联的可解析的文本块，我们将通过该名称引用相应的文本。这类实体声明包含关键字 ENTITY、实体名称和替换值。例如：

```
<!ENTITY copyright "© MegaTrouble Toys, Inc., 1999">
```

利用这个声明，我们只需引用名称“copyright”，就能够在文档内容的任何位置插入版权信息。当然，进行实体引用时，我们需要通过某种方式告诉解析器这是实体引用，以免它将实体名称与标记文本混淆。为此，我们在名称之前增加符号&，在其后增加分号。名称与定界符（&和;）之间不含空格。例如：

```
&copyright;
```

值得注意的是，由于字符&是XML的保留字符，因此如果我们需要在文档中使用它，必须借助（前面介绍的）预定义实体。

当实体引用在解析过程中被替换为置换文本时，其结果必须是格式正规的XML。

通用实体也有外部形式，即：将置换文本存放在外部文件中。其声明形式如下：

```
<!ENTITY myEntity SYSTEM "http://www.wrox.com/boilerplate/copyrighttext.txt" >
```

关键字SYSTEM用于指示外部源，后面的URL表示文件的位置。你也可以使用PUBLIC关键字、URI标识符和后备的URL组合。对于文档中的外部实体，XML建议有一定的限制：属性值中不能引用可解析的外部实体，以避免实体的字符编码与主文档的编码形式不同。

XML推荐标准不要求不验证有效性的解析器读取并插入外部实体的内容。

最后，实体不能包含直接或间接的对自身的引用。因此以下声明是不合法的：

```
<!ENTITY badSelfRef "Dancing with my&badSelfRef;" >  
<!ENTITY referToOther "Talking about &referBack;" >  
<!ENTITY referBack "Talking about &referToOther;" >
```

现在，让我们来讨论另一种实体：参数实体。

3. 参数实体

仅仅在DTD中使用的解析实体称为参数实体。它使我们能够简便地引用或修改 DTD中常用的结构，我们只需维护一处代码。与逐一修改 DTD中出现每个结构相比，这种方法简单得多，但是当我们打算扩展结构时，仍然需要编辑 DTD。参数实体声明由以下几部分组成：ENTITY关键字、百分号、名称和替换值。例如：

```
<!ENTITY % peopleParameters "age CDATA #IMPLIED weight CDATA #IMPLIED height  
CDATA #REQUIRED">
```

关键字CDATA代表字符数据；我们将在属性一节详细讨论它。

上面的置换文本是属性列表声明的一部分，它包含三个普通属性。当解析器处理上述语句时，它会用这三个属性取代参数实体名称。如果我们需要在 DTD输入该属性集合，只需引用实体peopleParameters即可。

在DTD中，所有参数实体必须在引用之前进行声明。这意味着 DTD内部子集不能引用在外部子集中声明的参数实体，因为解析器首先读取内部子集——所以会导致引用出现在声明之前。

引用参数实体时，需要在实体名称之前增加百分号，在其后增加分号。定界符与名称之间没有空格。下面的代码显示了如何引用上面定义的参数实体：

```
<!ATTLIST InsuredPerson  
    %peopleParameters;  
    carrier CDATA #REQUIRED >
```

以上代码声明 InsuredPerson 元素包含 4 个属性：其中 carrier 是显式声明的，其余三个参数（age、weight 和 height）包含在参数实体中，当解析器用置换文本取代实体引用时，会出现元素的完整列表。因此，上面的例子等价于以下代码段：

程序清单 3-1

```
<!ATTLIST InsuredPerson
  age CDATA #IMPLIED
  weight CDATA #IMPLIED
  height CDATA #REQUIRED
  carrier CDATA #REQUIRED >
```

这种类型的替换形式——在声明中进行替换——只能用于 DTD 外部子集。在内部子集中，参数实体引用只能位于其他声明之间；因此这类实体引用的置换文本必须是一个完整的声明，否则将影响 DTD 格式的正规性。

一个格式正规的文档应该遵循的规则都可以应用于参数实体。用置换文本取代实体引用后，仍然必须保证文档格式的正规性。当你构建参数实体时，一定要谨记这条规则。通常，在参数实体的置换文本中使用标记时要格外谨慎。下面的例子就破坏了格式正规约束：

```
<!ENTITY % myParm "<!ENTITY genEnt 'some replacement text here' ">
%myParm;
```

%myParm; 的置换文本是不完整的声明，它缺少结束标记 >，因此当解析器替换 %myParm; 时，DTD 就不再是格式正规的了。

与通用实体类似，参数实体的置换文本也可以位于外部文件中。例如：

```
<!ENTITY % myParam SYSTEM "http://www.wrox.com/declarationsets/Wroxdecls1A.ent">
```

从以上讨论可以看出，对于定义 XML 文档词汇表来说，实体是一种非常有价值的工具。下面让我们看看如何定义词汇表中的元素类型。

3.3.2 元素

元素是 XML 的核心与灵魂。在 DTD 中，元素类型是通过 ELEMENT 标记声明的。除了关键字，标记还提供所声明类型的名称和内容规范。正如第 2 章所述，元素类型名要遵守 XML 对名称的限制。名称可以是字母、数字，也可以使用标点符号，如：冒号（:）、下划线（_）、连字符（-）和句点（.）。然而，名称不能以数字开头。它的第一个字符只能是字母、下划线或冒号。

虽然名称中可以使用冒号，但是在第 7 章介绍名称空间时，你会看到有关冒号的保留用法。

鉴于这方面的原因，最好避免在元素名称中使用冒号。

元素内容可以分为以下四种类型：空、元素、复合及任意。空元素中既不包含文本，也不含子元素。但是它可以有属性。它用关键字 EMPTY 来表示。元素（更确切地说是纯元素）内容是指元素中只包含子元素，而不含文本。顾名思义，复合内容是元素和可解析字符数据（#PCDATA）或文本的组合。对于两种类型，我们可以通过结构表达所需的内容。复合内容和元素内容是采用内容模型（content model）表示的。内容模型是一种规范，它定义了元素内容的内部结构。如果你希望元素具有任意形式的内容，同时不破坏 XML 的格式正规语法，应该使

用关键字 ANY 进行声明。

```
<!ELEMENT SomeData EMPTY>
<!ELEMENT AnyOldThing ANY>
```

元素类型 SomeData 不含任何内容。下面是该类型的实例：

```
<SomeData/>
```

通常，在以下情况你可能会使用空元素。在文档中写入元素本身足以起到标识的作用。例如，HTML 中的
 元素。你可以利用这种方法通知应用程序改变处理模式。如果你希望在文档中插入一组相关的参数，而专门为它们建立结构又不太值得，此时你可以使用空元素，利用它来表达参数之间的关系。稍后讨论属性时，你会对此有进一步认识，XML 定义的某些属性类型可以用来表示一对一或一对多关系。如果你要说明的仅仅是关系本身，空元素是最适合不过的。

AnyOldThing 声明为 ANY 内容，因此我们可以使用元素和文本的任意组合。一般而言，使用 ANY 内容模型时要格外谨慎，因为解析器基本上不能提供有效性验证。

表 3-3

顺序运算符	含 义
,	(逗号) 表示严格顺序
	(管道符号) 表示选择

内容模型即元素结构的声明。它是由圆括号包含的若干子元素名称、运算符和 #PCDATA 关键字的组合。运算符用于说明元素包含的元组，以及元素和字符数据之间的组合方式（参见表 3-3）。

以逗号分隔的列表表示顺序排列的元素。下面的代码声明了 PersonName 元素：

```
<!ELEMENT PersonName (First, Middle, Last)>
```

在元素实例中，First、Middle 和 Last 必须按指定的顺序出现。如果你希望为文档的创作者提供选择的余地，可以参考下面的 FruitBasket 元素类型声明，它可以包含 Apple 或 Orange，但是两者不能同时出现：

```
<!ELEMENT FruitBasket (Apple | Orange)>
```

内容模型可以嵌套。下面的例子是修改后的 FruitBasket，它包含的第一个参数是 Cherry，第二个参数可以从 Apple 或 Orange 中选择其一：

```
<!ELEMENT FruitBasket (Cherry, (Apple | Orange))>
```

根据以上声明产生的实例必须包含两个元素：Cherry 以及 Apple 或 Orange，且它们必须按照指定的顺序出现。根据声明，元素实例只可能有以下两种形式：

程序清单 3-2

```
<FruitBasket>
  <Cherry>...</Cherry>
  <Orange>...</Orange>
</FruitBasket>
```

程序清单 3-3

```
<FruitBasket>
  <Cherry>...</Cherry>
  <Apple>...</Apple>
</FruitBasket>
```

除了我们前面介绍的顺序运算符，还有一种非常重要的运算符——元组运算符。特定的元素类型允许多少实例？表 3-4 列出了元组运算符。

如果没有元组运算符，说明元组数为一。元组运算符可以用于元素或内容模型，它能够产生许多非常复杂的结构。让我们进一步修改 FruitBasket 元素类型声明：

```
<!ELEMENT FruitBasket (Cherry+, (Apple | Orange)*)>
```

以上内容模型组表示 FruitBasket 可以有一个或多个元素类型 Cherry 的实例，以及零个或多个 Apple 或 Orange 的实例。而且所有 Cherry 元素必须连续出现。下面是一个正确的 FruitBasket 实例：

程序清单 3-4

```
<FruitBasket>
  <Cherry>...</Cherry>
  <Cherry>...</Cherry>
  <Apple>...</Apple>
  <Orange>...</Orange>
  <Orange>...</Orange>
</FruitBasket>
```

如果你希望表示复合内容，需要在内容模型中包含 #PCDATA。内容模型中的元素必须以运算符分隔，而且整个组声明为可出现“零次或多次”：

```
<!ELEMENT MixedBag (#PCDATA | ItemA | ItemB)*>
```

根据 XML 1.0 推荐标准中规定的语法，使用复合内容模型时，#PCDATA 关键字必须是模型中的第一个选项。

以上代码表示可以从 ItemA、ItemB 和 #PCDATA 中选择零个或多个选项。它可以有以下实例：

程序清单 3-5

```
<MixedBag>
  <ItemA>...</ItemA>
  Here is some text I wanted to include as pcdta
  <ItemA>...</ItemA>
  <ItemB>...</ItemB>
</MixedBag>
```

考虑以下内容模型，看看它们各自表达什么含义：

```
<!ELEMENT foo (A, (B | C))>
```

元素 foo 包含两个子元素，第一个永远是元素 A。第二个是 B 或 C。

```
<!ELEMENT foo (A, B?, C)>
```

在上例中，foo 包含两个或三个按顺序排列的子元素。其中 B 是可选的。

```
<!ELEMENT foo (A?, ((B, C) | D), E?)>
```

表 3-4

元组运算符	含 义
?	可选的；可有可无
*	零个或多个
+	一个或多个

现在，foo元素变得更加复杂了。它的第一个子元素可以是 A、B或D。根据选择不同，它可以有一至四个子元素。其中A是可选的，然后是B和C或D，E也是可选的。

```
<!ELEMENT foo ((A, B) | (C | D))>
```

在上例中，元素foo可以有一个或两个子元素。可能是顺序排列的 A和B，也可能是C或者D。让我们再稍微修改一下这个模型：

```
<!ELEMENT foo ((A, B)+ | (C | D))>
```

在上例中，元素foo可以包含重复的 A、B对列表，或者一个单独的 C或D。借助内容模型，可以产生变化多样的子内容实例。例如：

```
<!ELEMENT foo (A, (B, C)*, D+)>
```

根据上述定义，元素foo可以包含一个A，零个或多个B、C对，以及至少一个D。

我们希望通过上述例子能够激发起你尝试更复杂模型的兴趣。内容模型的规则虽然简单，但是它能够产生灵活多样的结构。为了测试以上定义，你可以在 DTD中插入其中一段代码，然后编写符合定义的文档，并在能够验证有效性的解析器上运行。

Internet上提供了几个可以通过 Web页面访问的解析器。我常用的是 <http://www.stg.brown.edu/service/xmlvalid/>，我经常用它来检查DTD结构。

现在让我们来看一看属性。

3.3.3 属性

属性是对元素的补充和修饰，它能够将一些简单的特性与元素相关联。通过属性，我们可以给元素绑定大量信息。例如，在 HTML标记IMG中，SRC就是一个属性。属性在 XML DTD中是使用ATTLIST标记声明的。对于含属性的元素，至少要通过一个 ATTLIST标记声明其属性列表。ATTLIST声明由以下部分构成：ATTLIST关键字、属性修饰的元素名称，以及零个或多个属性定义。为了增强可读性，每个属性定义通常占据单独的一行。

属性定义包含属性名称、类型和缺省声明。

```
<!ATTLIST myElement AttributeName CDATA #REQUIRED >
```

在以上代码中，我们声明了一个名为 AttributeName的属性，它必须在myElement元素实例的起始标记中出现（#REQUIRED——这是缺省设置），属性的值是字符串（CDATA）。

属性声明可以有几种不同的缺省设置，它定义了属性在文档中出现的方式。在研究属性类型之前，我们先来看看属性声明的缺省设置。

1. 缺省值

属性声明可以有四种缺省设置，如表 3-5所示。

表 3-5

属性缺省设置	含 义
#REQUIRED	元素的每个实例必须包含该属性
#IMPLIED	元素实例可以选择是否包含该属性
#FIXED加上缺省值	属性的值永远固定为缺省值；如果元素中不包含该属性，解析器

(续)

属性缺省设置	含 义
只有缺省值	将缺省值作为属性值 如果元素中不包含该属性，解析器将缺省值作为属性值。否则，该属性可以有其他值

如果 ATTLIST 声明中设置了缺省的属性值，即使文档中的某些元素实例忽略了该属性，XML 解析器仍然会认为该属性已经被赋予了缺省值。因此，对于下面显示的属性声明，这两个元素实例是等价的：

程序清单 3-6

```
<!ATTLIST SomeElt color "blue">

<SomeElt color="blue">...</SomeElt>
<SomeElt>...</SomeElt>
```

从上例可以看出，color 属性声明有缺省值：blue。在第一个元素实例中，我们显式声明了这个属性，而在第二个实例中，我们省略了属性。对于解析器来说，这两个实例是相同的——都有值为 blue 的属性 color。

在下面的例子中，Book 元素包含一个名为 level 的属性。如果我们将 level 属性的缺省值设为 Professional，考虑一下会出现什么情况。

```
<!ATTLIST Book
  level CDATA "Professional"
>
```

如果文档中的 Book 元素不含 level 属性，任何处理该元素的应用程序都会认为元素设置了 level 属性，且值为 Professional。如果缺省值出现的几率非常高，不妨采用这种方式。在这种情况下，我们可以声明缺省值，当元素实例的属性值与缺省值相同时，可以省略该属性。

然而，这种技术也可能给应用程序带来麻烦。你必须确保所选择的缺省值对于应用程序的处理来说是可靠的。元素的属性很容易被遗忘。在这种情况下，应用程序将使用 DTD 中声明的缺省值。如果你编写的代码极其依赖于属性值的正确设置，应该使用 #REQUIRED 关键字（或枚举值，我们稍后会讨论有关内容），以确保属性值的显式设置。

下面显示了元素 Book 的属性列表，你不必对各部分的含义过于计较：

程序清单 3-7

```
<!ATTLIST Book
  ISBN ID #REQUIRED
  level CDATA #IMPLIED
  pubDate CDATA #REQUIRED
  pageCount CDATA #REQUIRED
  authors IDREFS #IMPLIED
  threads IDREFS #IMPLIED
  imprint IDREF #IMPLIED >
```

在属性列表中，首先要指定元素名称 Book，然后是属性名称、类型，以及元素是否必须包

含该属性。可选的属性用关键字 `#IMPLIED` 表示。用关键字 `#REQUIRED` 修饰的属性必须出现在每个 `Book` 元素实例中。

表3-6列出了XML定义的属性类型。

表 3-6

属性类型	含 义
CDATA	字符数据（字符串）
ID	特定文档中唯一的名称
IDREF	对某些具有ID属性的元素的引用，这些元素的ID属性值必须与IDREF属性的值相同
IDREFS	若干以空格分隔的IDREF
ENTITY	已定义的外部实体的名称
ENTITIES	若干以空格分隔的ENTITY名称
NMTOKEN	名称
NMTOKENS	若干以空格分隔的NMTOKEN
NOTATION	接受一个在DTD中声明为用于指示表示法类型的名称
[枚举值]	接受用户显式定义的属性可选值中的一个值

下面让我们依次讨论这些属性类型。

2. CDATA

所有的内容最终都会变成文本。当属性值为纯文本时，你可以将该属性声明为 `CDATA` 类型。

例如：

```
<!ATTLIST SomeElt someText CDATA #IMPLIED>
```

该属性的值可以是任意长度的字符串。唯一的限制是它不能包含标记。上述声明可以有以下实例：

```
<SomeElt someText="This is valid">...</SomeElt>
```

只要属性值是纯文本，解析器都会将它视作有效。

3. ID、IDREF、IDREFS：文档中的关系表示

毫无疑问，对于 `ID` 类型的属性，其值必然是具有唯一标识功能的名称。而且它们必须遵守XML名称定义的规则。特定元素的 `ID` 属性值在整个文档中必须是唯一的。它可以作为元素的唯一标识符。每个元素至多有一个 `ID` 类型的属性。最后需要说明的是，`ID` 类型的属性必须设置为 `#IMPLIED` 或 `#REQUIRED`，不能是 `#FIXED` 或缺省的。可想而知，为 `ID` 提供缺省值，特别是固定的缺省值是毫无意义的。这会破坏 `ID` 的唯一性。例如，在下面的声明中，社会保障号（`SSN`）作为个人的唯一标识符，并与文件中的个人信息相关联：

```
<!ATTLIST Person
    SSN ID #REQUIRED>
```

如何使 `ID` 类型的属性发挥作用呢？当然是通过引用。我们可以利用它在两个对象之间建立一对一的关系。`IDREF` 类型可以用于在文档中创建链接和交叉引用。`IDREF` 属性的值必须受到与 `ID` 类型同样的约束。它们必须与文档中的某个 `ID` 属性具有相同的值。`IDREF` 值不能指向文档中不存在的 `ID`（但是除此之外还有其它方法，我们将在第8章介绍）。在应用程序中，我们通过

ID和IDREF实现交叉引用，而不必多次重复整个元素。如果文档中包含上述声明，可以在 DTD 中写入以下声明：

```
<!ELEMENT Customer EMPTY>
<!ATTLIST Customer
    id IDREF #REQUIRED>
```

我们很容易想到，id属性是指Person中的SSN属性。则文档中可以写入以下代码：

程序清单 3-8

```
<Person SSN="111-22-3333">
    <Name>...</Name>
    ...
</Person>
...
<Customer id="111-22-3333"/>
```

通过交叉引用，Customer可以写作具有IDREF属性的空元素，而不必包含整个 Person元素。当我们需要Person信息时，由于Person元素的SSN属性与Customer的id属性具有相同的值，因此应用程序能够通过id找到Person元素。

有时，我们希望将一个元素与其他多个元素相关联。这就要依靠 IDREFS类型。它能够建立一对多的关系。这类属性的值是一系列以空格分隔的 ID值。其中每个ID必须满足对ID类型的约束，当然它们必须与文档中的ID属性值相匹配。

```
<!ELEMENT Team EMPTY>
<!ATTLIST Team
    members IDREFS #REQUIRED>
```

以上代码段声明了一个空元素，它定义了项目组与成员之间的一对多包含关系。Members属性通过引用Person元素列举出项目组成员的标识，Person元素是在DTD的其他位置声明的，它具有ID类型的属性。例如：

```
<Team members="111-22-3333 222-11-4444 123-45-6789"/>
```

以上代码表示的项目组由三个人构成，他们的社会保障号分别是：111-22-3333、222-11-4444和123-45-6789。

利用ID、IDREF和IDREFS，我们可以表示关系数据库中常见的关系。如果你将XML作为本地数据库与专用数据模式之间的转换工具，你会深刻体会到这几种类型的价值。

4. ENTITY、ENTITIES：可替换的内容

实体可以用于属性声明中，它能够重用公共的结构，提高代码效率。对于一个可能多次出现的结构，你可以声明代表该结构的实体，然后通过引用实体实现对结构的调用。另外，实体中可以包含未解析内容，并作为有效的属性值。通过这种方式，文档创作者可以引用各种类型的数据，而不仅仅是XML标记。如果你有一个图形文件，并希望将它作为图解，可以借助实体将它插入文档。为此，首先将属性类型声明为 ENTITY：

```
<!ATTLIST SalesResults
    month_graph ENTITY #IMPLIED>
```

在DTD中，还要声明实体：

```
<!ENTITY sales_chart SYSTEM "sales_chart.gif" NDATA gif>
```

NDATA（表示法数据）关键字说明实体的数据有相应的 notation 类型（参见后面对 NOTATION 类型的讨论）。

而后，在 XML 文档中，我们可以在属性中引用图像：

```
<SalesResults month_graph="sales_chart">...</SalesResults>
```

以上代码将 GIF 文件 sales_chart.gif 与 SalesResult 元素相关联。

对于经常要重用的实体，这种方法非常值得推崇。例如，在我们所举的例子中，每月只需修改 sales_chart.gif 文件，就可以重用它。但是，假如实体的值需要频繁修改，这种方法就不可取了。

为了将 ENTITY 作为属性类型，你需要执行四个步骤。前三个步骤都是在 DTD（外部 DTD 或内部子集）中进行声明。第四个步骤涉及特定的文档实例。我们将这四个步骤总结如下：

- 声明一个表示法（我们很快就会介绍有关内容）。
- 声明一个或多个实体，以便在属性中使用。
- 为元素声明类型为 ENTITY 的属性。
- 在文档中创建元素类型实例，将实体名称作为属性值。

正如我们能够将多个 IDREF 值作为单一的属性值（IDREFS），实体也可以有类似的特性。这就是 ENTITIES 类型，它与 IDREFS 具有类似的效果。属性值中的每个名称必须符合 ENTITY 类型的规则，实体名称之间以空格分隔。因此，我们有以下代码（我们暂时省略表示法声明）：

程序清单 3-9

```
<!ELEMENT AccidentScene (#PCDATA)>
<!ATTLIST AccidentScene
  photos ENTITIES #IMPLIED>
<!ENTITY site SYSTEM "http://someserver/photos/scene.jpg" NDATA JPEG>
<!ENTITY auto_pic SYSTEM "http://someserver/photos/auto145.jpg" NDATA JPEG>
<!ENTITY victims SYSTEM "http://someserver/photos/victims.jpg" NDATA JPEG>
...
<AccidentScene photos="site auto_pic victims">Accident Scene report...
</AccidentScene>
```

我们关于事故现场的报告有一个 AccidentScene 元素，其中包含现场、汽车和受害者的照片。

5. NMTOKEN、NMTOKENS：名称记号

某些情况下，你可能希望将属性值作为离散的记号，而不是文本。为此我们可以使用枚举类型（稍后即将讨论该类型），但是，假如我们希望值列表能够无限扩展呢？这就需要依靠 XML 中称为名称记号（name token）的类型。它在 DTD 中缩写为 NMTOKEN。NMTOKEN 类型必须遵守元素名称的命名规则，但是其中一项限制除外。它们只能包含字母、数字、冒号、句点和连字符。然而，与元素和属性名称不同的是，NMTOKEN 的第一个字符可以是任意字符。下面的代码说明了如何声明 NMTOKEN 属性：

程序清单 3-10

```
<!ATTLIST Employee
  security_level NMTOKEN #REQUIRED>

<Employee security_level="trusted">...
```

上述代码表示元素 Employee 有一个名为 security_level 的属性，其值符合 XML 名称记号的规则。我们可以用它来控制对机密文档的访问。由于定义属性列表时使用了 NMTOKEN，而不是枚举类型，文档创作者只需创建新的值，就能够适应新的安全级别要求，而不必每次都编辑 DTD。只要符合我们前面介绍的有效的 NMTOKEN 值应该遵守的规则，任何值都可以作为这种属性的值。

显然，NMTOKEN 类型使得应用程序必须承担验证值有效性的任务。然而对于枚举类型，解析器能够提供有效性检查。

与 IDREFS 和 ENTITIES 类似，你可以声明属性类型 NMTOKENS，它的值由多个名称记号构成。每个名称必须是有效的名称记号，它们之间以空格分隔：

程序清单 3-11

```
<!ATTLIST Employee
  security_compartments NMTOKENS #IMPLIED>

<Employee security_compartments="red green mega ultra">...
```

这个职员能够访问名为 red、green、mega 和 ultra 的安全区域。就类型而言，这些都是有效的 NMTOKEN 值。与枚举类型不同，解析器不检查这些值的有效性。文档的作者必须确保自己使用了适当的名称。

6. NOTATION：非XML数据

当我们讨论实体类型的属性时，曾经提及表示法。通过将实体名称作为属性值，可以将 GIF 和 JPEG 图形文件与元素相关联。然而，XML 解析器不能处理二进制格式。那么，解析器的作用是什么呢？我们可以使用表示法标识要链接到 XML 文档的外部数据项的格式。表示法声明能够说明格式的名称，以及相关的外部处理器。解析器可以根据声明将自己不能识别的数据交给外部处理器处理。处理器声明类似于用于定位 DTD 文件的 DOCTYPE 声明。它可以是 PUBLIC 或 SYSTEM 的，而且必须包含外部处理器的名称：

```
<!NOTATION jpg SYSTEM "jpgviewer.exe">
<!NOTATION gif SYSTEM "gifviewer.exe" >
```

现在我们知道，当 jpg 作为表示法名称时，与之相关的数据将发送给 jpgviewer.exe 处理。利用表示法，XML 文档可以容纳多种不同的数据类型。这对于报表、病历、法律文书、学术报告，以及任何丰富多彩的多媒体演示来说都是非常有用的。但是，XML 仅仅是一个最基本的工具集。为了提供正确的表示语义，应用程序还有大量工作要做。

通过使用关键字 NOTATION，可以将属性定义为表示法名称类型的。例如：

```
<!ATTLIST Image type NOTATION (gif|jpg) "gif" >
```

```
<Image type="jpg">...
```

在以上声明中，Image元素可以有一个名为type的属性，它是表示法类型的。该属性可选的值有gif和jpg。如果元素实例没有定义type属性，解析器会假设该属性设置为缺省值 gif。然而，在上述实例中，值jpg覆盖了缺省值。

7. 枚举类型：选择

名称记号的长度是不受限的。虽然 NMTOKEN 和 NMTOKENS 属性值的格式必须符合命名规则，但是它所允许的值是可以自由设置的。在许多情况下，我们只希望允许一小部分字符串值，例如：yes 和 no 是表示决策的枚举值；red、yellow 和 green 是信号灯的颜色，等等。在这些情况下，我们要采用枚举属性。

为了声明枚举属性，在通常出现类型关键字的位置应该放置一组值。这些可选值包含在圆括号中，并以管道符号（|）分隔。声明中的可选值不需要带引号，但是与 XML 中的名称一样，它是大小写敏感的。文档中的属性实例必须包含唯一的一个可选值，且这个值必须是在属性声明中列举的。与其他属性值类似，枚举值必须包含在引号中。下面是两个简单的例子：

程序清单 3-12

```
<![ATTLIST Employee
    manager (yes | no) #REQUIRED>

<![ATTLIST ClassifiedDoc
    security_level (unclassified | secret | Top_Secret) #REQUIRED>
```

在第一个例子中，属性值只能是 yes 或 no；YES、NO 和 maybe 都是无效的。设置属性值时，不仅必须使用枚举类型声明中提供的值，而且要注意大小写。如果你构建的枚举类型的值可能由用户手工输入，应该考虑因大小写产生的各种变体。

现在让我们讨论 DTD 中使用的另一种技术。

3.3.4 条件部分

许多程序员都习惯于在程序中指定要解析的信息，仅当满足特定的条件时，编译器才解析指定的内容。DTD 提供了类似的功能，虽然它比通常的编程语言有更多的限制——运行时不能执行条件表达式。DTD 可以包含条件部分，它用于向解析器说明包含或忽略声明部分。它们能够用来控制 DTD 中的相关声明块。然而，DTD 内部子集不支持条件部分。

条件部分包括：惊叹号、左方括号、关键字，以及由方括号包含的声明块。如果关键字为 INCLUDE，其中的声明被认为是 DTD 的一部分。如果关键字为 IGNORE，处理器虽然读取其中的声明，但是在处理时忽略它：

程序清单 3-13

```
<![INCLUDE
    [<![ELEMENT AuditEntry (#PCDATA)>
    <![ATTLIST AuditEntry
        timestamp CDATA #REQUIRED
        userID IDREF #REQUIRED>
    ]]>
```

```
<![IGNORE
  [<![ELEMENT DebugEntry (#PCDATA)>
    <![ATTLIST DebugEntry
      serial ID #REQUIRED
      page CDATA #IMPLIED>
  ]]>
```

在上例中，AuditEntry及其属性将成为DTD的一部分，DebugEntry及其属性则不会对DTD产生任何影响。根据该DTD创建的文档可以使用AuditEntry，但是文档中的DebugEntry元素会被认为是无效的。

这一特征乍看起来没有什么价值。如果你不需要某些声明，为什么还要将它们添加到 DTD 中呢？如果声明包含在DTD中，为什么要使用INCLUDE呢？实际上，条件部分应该与参数实体配合使用。让我们更改一下以上实例，并就条件部分的用法作进一步说明。假设创建文档时，我们在文档实例的DOCTYPE声明中写入以下实体声明：

程序清单 3-14

```
<![DOCTYPE MixedBag SYSTEM "testCONDSECT.dtd" [
<![ELEMENT MixedBag (foo)>
<![ELEMENT foo (A, B?, stuff)>
...
<![ELEMENT stuff (#PCDATA| AuditEntry|DebugEntry)*>
<![ENTITY % accountsDept "INCLUDE">
<![ENTITY % codeDept "IGNORE">
]>
```

假设文档在创建时需要包含调试信息或计费信息。如果 DTD外部子集包含以下代码：

程序清单 3-15

```
<?xml encoding="UTF-8" ?>
<![%accountsDept;
  [<![ELEMENT AuditEntry (#PCDATA)>
    <![ATTLIST AuditEntry
      timestamp CDATA #REQUIRED
      userID IDREF #REQUIRED>
  ]]>

<![%codeDept;
  [<![ELEMENT DebugEntry (#PCDATA)>
    <![ATTLIST DebugEntry
      serial ID #REQUIRED
      page CDATA #IMPLIED>
  ]]>
```

然后，假设文档实例是要交给财务部门的：

程序清单 3-16

```
...
<stuff>We have Arthur's variable x, where 10 < x < 46
<AuditEntry timestamp="1999-11-29" userID="ref222">blahblah taxes
blahblah</AuditEntry>
</stuff>
...
```


这样，文档就能够正确地验证有效性。

由于内部DTD子集中的声明是先读的，因此实际上，参数实体声明是出现在外部 DTD子集中的参数实体引用之前的。

在以上例子中，我们需要的是在生产环境中用于审核的元素，但是在某些情况，出于测试或错误检测的需要，我们可能希望在文档中包含用于调试的元素。与 AuditEntry相关的声明将包含在DTD中，而与 DebugEntry相关的声明将被忽略。为了将 DebugEntry作为文档的有效元素，只需交换参数实体的关键字，这一操作是相当简单的。如果使用恰当，条件部分能够提供大量功能，并改善代码的可重用性。

3.4 DTD的缺点

DTD能够有效地推动XML的发展。然而，它也受到一些因素的限制。首先，它使用自己的一套语法，与文档实例的语法截然不同。更重要的是，如果 XML解析器能够使应用程序简便地访问它们所处理的 DTD中的声明，就会使 DTD成为一种非常有用的工具。遗憾的是，目前几乎没有解析器能够做到这一点。这一现状妨碍了我们利用 DTD验证文档的有效性，以及将相应领域的信息传达给编程人员。我们的应用程序无法了解 DTD中的声明及其结构。

类似地，我们不能使用解析器动态创建 DTD。这似乎不是一个重要的限制。毕竟，DTD被认为是恒定不变的定义，而不像动态建立的文档那样需要进行有效性验证。即使如此，仍然有一些需要根据条件做出选择的情况，例如：根据某些值改变词汇表规则。我们可以从数据库读取当前值，并据此构建 DTD。该 DTD可以用来创建一系列文档，这些文档在当前状态下是有效的。然后，DTD将随文档一起传送，以便接收者根据文档创建时的环境状态验证文档的有效性。DTD提供的条件选择结构不能实现上述功能，因此不得不动态创建 DTD。如果缺乏解析器的支持，我们只能手工创建 DTD。

DTD是一种封闭的结构。XML词汇表的规则完全包含在 DTD中。如果你不需要从其他 DTD借用声明或结构，或许感受不到这方面的局限性。由于实体处在一个很低的层次，因此试图改善 DTD扩展性的工作往往徒劳无功。事实上，没有简明的方法能够改善 DTD的扩展性。对于可扩展的标记工具，DTD也显得极不合作。我们无法根据概念和对象的相关性将声明分为若干段。假如能够做到这一点，就可以创建许多描述各个商业领域的 DTD，然后通过引用将它们粘合在一起，以满足真正的应用需求。当我们在第 7章讨论命名空间和模式时，会分析各种借用信息的可能性。它涉及的内容超出了 DTD的范围。

DTD在数据类型信息方面也存在一定的缺陷。它所提供的唯一工具就是表示法。我们无法根据现有的类型定义自己的新类型。表示法能够为一种未解析文本标记一个名称，但是这与强大的类型定义机制有着本质区别。我们希望将某些值表示为简单类型，例如：数字，而不是文本，并对这些值执行恰当的操作。

在 W3C提出弥补 DTD不足的解决方案之前，已经出现了其他一些模式机制，我们将在第 7章介绍有关内容。然而，暂时不考虑 DTD的缺陷，DTD是目前唯一一种声明 XML词汇表结构和内容的正式方法。DTD是非常基础性的概念，它有助于理解其他模式，以及如何利用 XML通过标准的方式交换文档。

3.5 用于图书目录问题的DTD

现在，让我们利用本章所学的内容为图书出版领域定义一个 XML 词汇表。更明确地说，在下面的例子中，我们将要定义描述图书目录的语法。从最通用的角度讲，我们的词汇表将涉及一类图书。从最精确的角度讲，我们可以利用词汇表从整体上描述图书。我们将允许文档创作者包含图书本身的目录。这似乎超出了我们的讨论范围。如果需要的话，我们能够提供某种形式的图书文档链接，但是在进行图书分类时，确实没必要包含其目录。

3.5.1 图书目录问题的正式定义

我们首先来描述一下要建模的业务流程。目前，你还不必担心 DTD。一旦理解了要处理的业务流程，你就能够建立漂亮的模型，急于创建规则往往会影响你对问题本身的理解。我们要做的第一件事情是设计问题空间的主要对象及其相互关系。

1. 问题模型

首先必须明确的是，我们要讨论的是一个目录。我们要描述的所有内容都包含在这个目录中。我们不会涉及各个目录之间的关系，而仅考虑目录本身的内部关系。这有助于我们集中注意力，我相信这是编写优秀的 XML 词汇表的关键。

目录中包含一本或多本图书——这就是整个词汇表。如果我们停留在这样的认识上，会遗漏什么问题吗？图书是由出版社出版的，因此我们应该在目录中包含出版社。一个出版社就够了吗？当然。这就意味着一家出版社至少需要一个 Book Catalog 文档。虽然它也可以有多个文档。出版社有时会涉猎多个领域。这对我们的模型有影响吗？它们是如何组织的？通常，目录是根据主题组织的。而主题是出版社出版的内容。如果一家出版社拥有大量书籍，它会根据某些主题或知识领域划分出更细的目录。

就问题本身而言，并不需要包含多家出版社。但是既然谈到这个因素，你可能会考虑谁还有可能使用该模型。图书馆或图书收集者会有同样的需求，他们需要根据主题来描述图书。图书销售商也有类似的需求。这三类用户需要在他们的类别中包含多个出版社。通过这方面的修改，为该模型创造了更多的应用机会。这种修改通常意味着你遇到了问题。你可能转移了问题的方向。然而，在本例中却不是这种情况。我们仍然能够满足单一出版社的需求。无论有多少出版社，在出版社信息之后，文档包含的依然是一系列图书摘要。从“有且仅有一个”出版社变为“一个或多个”出版社使得我们的模型更加灵活，同时并不影响我们要描述的内容。

主题是什么？它是用于组织知识或讨论的线索。它类似于新闻组中的线索。主题或线索通常可以从书名或书的目录获得，但是我不希望采取这种方式。线索应该有自己的描述性信息。到目前为止，我们得到了图 3-1 所示的组织图。

在我们的模型中，一个目录包含一家或多家出版社、零个或多个线索，以及一本或多本图书。现在，我要讨论两个问题。它们都是关于我对问题空间所做的假设。第一个问题是元组的数目。为什么我会认为图书和出版社可以是“一个或多个”，而线索甚至可以不出现？不包含任何书籍的图书目录是一种极端情况，只有数学家会对此感兴趣。我们基本上不会对空集有兴趣——如果目录为空，有什么必要讨论它吗？这是个具有实际用途的词汇表，而不是理论意义上

的。然而，词汇表中可以没有用于组织图书的出版社线索。

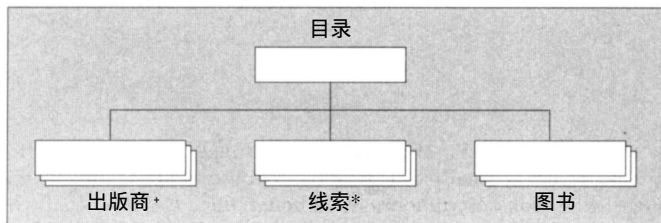


图 3-1

谈到出版社，又出现了一些不太明确的问题。有些情况下，我们只对目录中的书感兴趣。为此，我们是否应该将出版社的元组数设置为“零个或多个”？你的想法也许与我的不同——对于建立DTD来说，没有永恒的规则——但是每本书确实与一家出版社有关系。即使相当简单的情况——只有一家出版社——我们也希望锁定这种关系。我们可以忽略出版社信息，但是电子商务应用中需要该信息。因此，在我们的例子中，我们将出版社的数目设定为“一个或多个”。

另一个问题是如何将线索与图书相关联。刚才我们曾经将图书线索与新闻组线索进行比较，鉴于此，我们可以将图书作为线索的子元素。然而，这会产生一定的限制。根据目录中线索的特征以及图书涉及的范围，一本书往往会与多个线索相关联。有些用户可能对于按照线索组织信息不感兴趣。许多程序都属于这种情况。例如一个简单的清单程序。它注重的是按照字母顺序或者根据ISBN排列所有图书。如果根据线索组织图书，这些程序不得不通过线索寻找图书。在本例中，我们会将线索作为一个独立的结构。当然，为了在图书与线索之间建立联系，必须通过某种机制定义它们之间的链接。

在正式编写DTD之前，首先让我们看一下有关对象的结构。由于这是一个关于图书的目录——因此出版社和线索的重要性仅仅体现在它们与图书的关系——我们将以此为出发点。

2. 图书

在此，我并打算提供元素组成图，我只希望通过图 3-2说明元素的包含关系。我们将确定图书元素的子元素。（从图书项的角度考虑，）图书元素应该包含哪些内容？

当然，书总是要有书名的。我还加入了摘要。这是从学术论文得到的启发，它是图书内容的简短描述，通常只有一段。零售目录也有图书简介，虽然其内容一般比学术摘要少。无论如何，简短的描述是非常有用的。实际上，由于它是图书内容的总结，因此可以作为主要的搜索目标。当你根据关键字进行搜索时，相信你宁愿将查找的范围限定在摘要中。因为即使书的正文中包含你要找的关键字，书的内容也有可能与你所关心的问题毫不相干。然而，摘要体现的是书的主题。无论出于什么目的，图书的摘要都

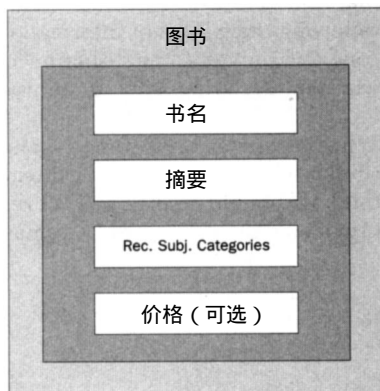


图 3-2

是目录的主要组成部分。

许多出版的商业图书的背面都有三个推荐的主题域。它们是由出版社提供的，目的是帮助图书经销商给图书进行适当的分类。作为商业用途的目录文档也应该包含该信息。

事实上，目录上出现的每本书都是为了销售的。然而也有一些例外。当我们讨论出版社的数目时，曾经提到该 DTD 也可以供图书收集者使用，而他们对于图书的价格并不感兴趣。另外，博物馆或图书馆也不关心书的价格；书籍本身是有价值的，但是给它们做目录并不是为了销售。而且出版社在给图书确定零售价之前可能已经将它归入某个目录。鉴于以上原因，我们将价格作为可选的。

至此，我们已经有了四个子项。记住，我们早就将书的目录排除在词汇表之外。当你需要更深入地了解书的内容时，你会需要它的目录。书的目录由章节标题构成。现在让我们暂停对图书对象的讨论，转向另一个重要对象——出版社。

3. 出版商

图3-3是出版商对象的包含关系图。

出版商是一个法人实体。它是负责图书编译、出版和发行的公司。我们的出版商模型将反映它的商业特征。显然，公司的名称是最基本的信息。由于这是一家公司，它常常会有许多分公司。例如，一家大型出版商可能在每个洲都有分公司。Internet或许会改变这种情况，但是许多出版商都需要提供位置列表——即：若干地址。出版社通常有多个印记。它类似于目录中的线索，或者说市场中的品牌。印记通常由名称和徽标构成。描述出版社必须列出其所有印记。



图 3-3

我最后要提到的问题可能有些争议。书是由作者编写的。我们会在书中包含作者的信息。然而，一位作者常常会写多本书。在图书之外单独描述作者，并在两者之间建立联系似乎是一种更恰当的方式。这意味着需要某种链接机制。当我们讨论将图书置于线索之外时，曾经提到过这种需求。我们在此最好加以强调。作者是否不属于出版商的范围？虽然某些作者会从一家出版商跳到另一家出版商，但这只是一种例外，而不是常规。作者群体是出版商的重要智力资产，因此我认为应该在出版商对象中包含作者信息。

4. 线索

现在我们来讨论线索。它是一个有些难以捉摸的概念。在现实世界中找不到合适的类比。但是，我们确实需要它，这一点在前面已经有过论述。实际上，很难为线索定义一种适用于所有目录的正式结构。我相信简短的文本内容——#PCDATA——就足以描述线索。

Wrox 出版社的Web站点 (<http://www.wrox.com>) 是根据八条线索组织的。其他出版社也有类似的概念，有时称为系列，或者根据目标读者划分类别。在计算机图书领域，图书通常是根据特定的技术、语言或产品族来组织的，因此线索的概念是相关的。

5. 出版目录 DTD

既然已经了解了要描述的内容，让我们开始定义图书目录文档用到的标记。在 XML 中，对象是通过元素模型化的。它们所包含的信息通常可以转化为子元素，某些简单的特性也可以通

过属性来描述。

(1) 目录

显然，目录元素应该是目录文档的根。根据前面的组织结构图，我们可以将目录描述为：

```
<!ELEMENT Catalog (Publisher+ , Thread* , Book+ )>
```

内容模型相当简单。Catalog元素包含顺序排列的子元素：出版社、线索和图书。每个子元素的元组数目如下：一个或多个 Publisher，零个或多个 Thread，一个或多个 Book。这说明 Thread虽然很有用，但并不是十分重要。由于线索的数量定义，出版社列表后面有可能紧跟着图书列表。

目录元素不包含任何属性。它在文档中只有一个，因此没有必要提供标识符。我们将对目录的讨论分散到它所包含的其他对象中。

(2) 出版社

现在，让我们来看看Publisher（出版商）元素的定义：

```
<!ELEMENT Publisher (CorporateName , Address+ , Imprints , Author* )>
```

上述定义对应于我们在前面看到的示意图。Publisher元素包含一个公司名称，一个或多个地址，若干印记，以及零个或多个作者。有些出版社为了简单起见有时宁愿不要作者，但是在我们的目录中，无作者代表该书是匿名作者编写的，或者作者过多，例如：会议的专题文集。

有一点需要特别注意，Imprints没有设定元组数。这并不是疏忽。其他属性都是在同一级列出的，对于Imprints，我希望改变一下风格，通过元素集合来说明。Publisher元素将包含由Imprint元素集构成的子元素。这种方式并不会影响元素的含义，它有利于应用程序的程序员理解DTD。如果你希望忽略该元素，只需直接跳至下一元素。如果你对该元素感兴趣，你可以利用解析器检查属于Imprints元素的子元素的数量，由此得到印记数。

另外，Publisher元素还有一个属性isbn：

```
<!ATTLIST Publisher isbn CDATA #REQUIRED >
```

Publisher元素需要一个唯一的指示符，而公司名称不能作为指示符。图书的ISBN包含分配给出版商的唯一数字。例如，Wrox出版社的所有书的ISBN都包含序号1861。我希望文档的创作者将整个ISBN中的这个片断作为该属性的值。

下面我们来介绍构成Publisher元素的子元素：

程序清单 3-17

```
<!ELEMENT CorporateName (#PCDATA )>
<!ELEMENT Address (Street+ , City , PoliticalDivision , Country , PostalCode )>
<!ELEMENT Imprints (Imprint+ )>
<!ELEMENT Imprint (#PCDATA )>
<!ELEMENT Author (FirstName , MI? , LastName , Biographical , Portrait )>
```

以上代码应该与你想象的差不多。大多数元素都是 #PCDATA类型的。

本节最后给出了图书目录DTD的完整代码。

Imprints元素是由一个或多个 Imprint元素构成的，如前所述，Imprint元素是 #PCDATA类型的。Author元素看上去略微混乱一些，但是在分析其内容模型之前，我们还要进一步说明一下

Address和Imprint元素的属性：

```
<!ATTLIST Address headquarters (yes | no ) #IMPLIED >
<!ATTLIST Imprint shortImprintName ID #IMPLIED >
```

由于出版社可能有多处办公地点，因此要将总部和分公司区分开。在此，我使用简单的枚举类型，而且该属性是可选的。关键字 #IMPLIED表示文档的作者可以忽略该属性，同时不会影响文档的有效性。如果元素包含该属性，它的值必须是 yes或no。

现在让我们来研究 Imprint元素的属性列表。由于该元素的内容为 #PCDATA，因此 Imprint的名称的长度不受限制。我需要将图书与印记相关联，所以我希望有个较短的关键字。这就是 shortImprintName属性的作用。此后，Our Incredibly Wonderful Children's Book Division可以缩写为IncredChild。我们同样没有限制该属性的长度。实际上，DTD语法并未提供控制字符串长度的设置。

与字符串长度问题相比，如何使用 ID类型的属性更加重要。如果文档中出现该属性，它就作为唯一的关键字，可以被文档中的其他元素引用。在我们的例子中，我们知道该属性将被 Book元素引用，但是DTD并不提供显式的声明机制。

让我们回到 Author元素，以下是 Author及其子元素的声明：

程序清单 3-18

```
<!ELEMENT Author (FirstName , MI? , LastName , Biographical , Portrait )>
<!ATTLIST Author authorCiteID ID #REQUIRED>

<!ELEMENT FirstName (#PCDATA )>
<!ELEMENT MI (#PCDATA )>
<!ELEMENT LastName (#PCDATA )>
<!ELEMENT Biographical (#PCDATA )>
<!ELEMENT Portrait EMPTY>
<!ATTLIST Portrait picLink CDATA #IMPLIED >
```

Author的子元素相对来说比较通俗，它包含的文本信息你可想而知。我们提供了名称和个人简历信息。该元素的关键在于它的属性。Author元素的authorCiteID属性是元素的简短引用，它的作用与Imprint元素的shortImprintName属性类似。它是一个ID类型的属性，用于将作者与一本或多本书相关联。

Portrait元素用于将作者的照片与作者的文字信息相关联。由于XML标记中不能包含二进制数据，因此需要借助链接机制。我们不能使用实体取代链接，因为在一份很长的目录清单中，会出现许多作者和相应的图像文件，当创建文档时它们大多数都是动态命名的。XML委员会针对链接提出了许多方法，但是都未形成正式的建议规范，而且几乎未得到解析器厂商的支持。为了简单起见，我们还是暂时不提XLink或XPointer的语法为好，我将图像文件的URL作为picLink属性的值。然而，我们的应用程序无论如何都需要支持某种链接机制。如果读取目录文档的应用程序使用浏览器，我们可以为浏览器生成包含IMG标记的HTML文件。在这类应用程序中，picLink的值将成为IMG标记的SRC属性的值。如果使用文档的应用程序不具备可视化的界面，它只需忽略该属性。

你是否曾经考虑过使用表示法？由于我不希望限制应用程序使用 DTD 的行为，因此放弃了这种方法。如果没有提供针对特殊类型的处理程序，表示法只是给数据类型定义了一个名称。如果我指定了处理程序，就必须保证任何使用目录文档的应用程序有特定的行为，例如：显示链接。而我的设计目标是无论应用程序是否有可视化的界面，DTD 都应该发挥同样的作用。

(3) 线索

我们已经完整而详细地分析了 Publisher 元素的内容。在 Catalog 内容模型中，下一个对象是 Thread。以下是该元素的声明：

```
<!ELEMENT Thread (#PCDATA )>
<!ATTLIST Thread threadID ID #IMPLIED >
```

元素的内容是用于描述线索的文本。它可能是关键字列表（例如：对于 Wrox 出版社来说，元素的内容可能是 ASP、Database、XML、Scripting 等），或者更长的描述。属性 threadID 用于在图书与线索之间建立关联。

(4) 图书

下面我们将讨论最后一个元素——Book。它是 DTD 的核心。我们介绍问题模型时曾经给出了图书对象的示意图，元素的声明完全是按照该图产生的：

```
<!ELEMENT Book (Title , Abstract , RecSubjCategories , Price? )>
```

Title 元素和 Abstract 元素是 #PCDATA 类型的。RecSubjCategories 元素的声明如下：

```
<!ELEMENT RecSubjCategories (Category , Category , Category )>
```

Category 是 #PCDATA 类型的。我们曾经说过，每本书都有三个推荐的主题域。实际上，每本书有且仅有三个主题域。最简单的表示方法是利用 DTD 提供的元组工具，将内容模型说明为三个 Category 元素的实例。

XML 模式能够提供更精确的控制，我们将在第 7 章介绍有关内容。我们将使用模式语法说明 Category 至少出现三次，且至多出现三次。

Price 元素有属性 currency，它用于说明图书价格的货币单位。在该 DTD 中，我们规定只能选择美元、加元或英镑。

```
<!ATTLIST Price currency (USD | GBP | CD ) #REQUIRED>
```

我们很容易增加货币单位的种类。这就是为何我们将 Price 设计为元素，而不是属性。通常情况下，我们可以将一个简单的原子特性设计为属性。由于我们要处理国际贸易，因此必须指定 Price 内容的货币单位，以便满足应用程序的需求。我希望说明货币单位是应用于 Price 的。所以，我没有将价格和货币单位都作为 Book 的属性。通过将 Price 说明为有 #PCDATA 内容的元素，我可以将 currency 作为属性，并显式指定它的内容。

下面的程序清单显示了 Book 元素的属性：

程序清单 3-19

```
<!ATTLIST Book ISBN ID #REQUIRED
level CDATA #IMPLIED
pubDate CDATA #REQUIRED
```

```

pageCount CDATA #REQUIRED
authors IDREFS #IMPLIED
threads IDREFS #IMPLIED
imprint IDREF #IMPLIED >

```

这是目前为止我们见到的最复杂的属性列表。ISBN当然是图书唯一的ISBN号，我们将它作为ID类型以便于引用。Level属性代表出版社给图书设置的难度级别。你正在阅读的这本书设置为Professional。该属性的值是由各个出版社自行定义的。虽然大多数出版社都有类似的概念，但它并不是通用的，因此我将它指定为可选的属性。然而，如果元素包含该属性，它可以作为排序或过滤的关键字。pubDate和pageCount属性分别代表图书的出版日期和页数。

DTD语法不允许定义日期和数字类型，虽然我们在此非常需要这两种类型。应用程序可以通过简单的算法计算图书面世的时间，以及一套书的总页数。DTD不允许传递用于这类计算的信息。它们必须隐含在对词汇表的认识中。XML模式将解决这类问题。

我们已经在DTD中定义了几个ID类型的属性。现在终于到使用它们的时候了。属性 authors的值是指向代表特定作者的ID的authorCiteID值的一个或多个引用。例如：

```
<Book authors="smohr McKay mbirbeck" ...>
```

以上代码表示书的作者是 authorCiteID值分别为 smohr、McKay和mbirbeck的作者。属性 threads与属性authors类似，它将一本图书与一个或多个知识线索相关联：

```
<Book threads="COM database XML" ...>
```

由于线索可以引用多个ID，因此能够从多条途径分析目录。用户可以查阅一个目录文档，并根据他所感兴趣的一条或多条线索进行过滤。出版社能够以目录文档为基础，根据线索和级别选取图书，产生适当的阅读列表。

最后，imprint属性是对出版社印记的引用。一本书只能有一个印记，因此该属性是IDREF类型的。Book元素中为什么没有关于图书的出版社的属性？书的ISBN包含出版社的编号。这个编号位于ISBN中的固定位置，所以我们能够从ISBN中提取出版社编号，并搜索具有相同isbn属性值的Publisher元素。为了进一步简化，我们也可以根据imprint的值进行搜索。它能够直接得到Publisher元素，而不必象ISBN那样需要从属性值中提取子串。实际上，上述两种方法都不是直截了当的，但是它们都很实用。

任何设计工作都会产生各种不同的结果。你完全可以根据自己的经验或需求考虑类似的图书出版或目录信息问题。看看你会设计出什么样的DTD；我是否忽略了你所需的某些概念。你也可以尝试着通过不同的方式表达同样的想法，或许对于你来说，它更加有效。到底应该将某个项目模型划为元素还是属性，人们常常会在这方面产生争论。你不妨修改一下我的设计方案，看看会产生怎样的后果。优秀的DTD设计来自于经验和试验。

(5) 完整的DTD

下面是PubCatalog.dtd的完整清单。本书的其他章节将用到它：

程序清单 3-20

```

<!ELEMENT Catalog (Publisher+ , Thread* , Book+ )>

<!-- Publisher section -->
<!ELEMENT Publisher (CorporateName , Address+ , Imprints , Author* )>

```

```

<!ATTLIST Publisher isbn CDATA #REQUIRED >
<!ELEMENT CorporateName (#PCDATA )>

<!ELEMENT Address (Street+ , City , PoliticalDivision , Country , PostalCode )>
<!ATTLIST Address headquarters (yes | no ) #IMPLIED >
<!ELEMENT Street (#PCDATA )>
<!ELEMENT City (#PCDATA )>

<!--State, province, canton, etc.-->
<!ELEMENT PoliticalDivision (#PCDATA )>
<!ELEMENT Country (#PCDATA )>
<!ELEMENT PostalCode (#PCDATA )>

<!ELEMENT Imprints (Imprint+ )>

<!ELEMENT Imprint (#PCDATA )>
<!ATTLIST Imprint shortImprintName ID #IMPLIED >

<!-- Author section -->

<!ELEMENT Author (FirstName , MI? , LastName , Biographical , Portrait )>
<!ATTLIST Author authorCiteID ID #REQUIRED>

<!ELEMENT FirstName (#PCDATA )>
<!ELEMENT MI (#PCDATA )>
<!ELEMENT LastName (#PCDATA )>

<!ELEMENT Biographical (#PCDATA )>

<!ELEMENT Portrait EMPTY>
<!ATTLIST Portrait picLink CDATA #IMPLIED >

<!-- Organization of the catalog -->
<!ELEMENT Thread (#PCDATA )>
<!ATTLIST Thread threadID ID #IMPLIED >

<!-- Book summary information (no content) -->

<!ELEMENT Book (Title , Abstract , RecSubjCategories , Price? )>
<!ATTLIST Book ISBN ID #REQUIRED
            level CDATA #IMPLIED
            pubDate CDATA #REQUIRED
            pageCount CDATA #REQUIRED
            authors IDREFS #IMPLIED
            threads IDREFS #IMPLIED
            imprint IDREF #IMPLIED >

<!ELEMENT Title (#PCDATA )>
<!ELEMENT Abstract (#PCDATA )>
<!ELEMENT RecSubjCategories (Category , Category , Category )>
<!ELEMENT Category (#PCDATA )>
<!ELEMENT Price (#PCDATA )>
<!ATTLIST Price currency (USD | GBP | CD ) #REQUIRED>

```

3.5.2 对象关系问题

请特别注意我们如何利用 ID、IDREF和IDREFS类型的属性说明模型中对象之间的关系。这

些类型虽然结构简单，但是能够解决复杂的问题。XML 1.0在这方面涉及的内容非常有限，它仅指出ID在文档中必须是唯一的。规范中并未强制定义任何编程语法。如果应用程序需要使用该特征，我们必须自己提供编程支持。我们不能期望符合XML 1.0规范的解析器提供链接功能。

目前，某些解析器准备支持XSL，它使我们能够在不进行过多编程的情况下简便地实现链接功能。然而，这是XSL的特性，而不是XML的特征，如今市场上的大多数解析器都不提供该功能。

尽管如此，ID和IDREF仍然是XML中的重要结构。它使我们能够在纯文本的文档中实现类似于关系数据库中的关系模型。如果你所设计的词汇表要被一些未知的应用程序使用，就会发现这两种结构非常重要。只要知道存在关系，就可以利用适当的属性列表声明捕获信息。应用程序能够充分利用该信息。

3.5.3 进一步讨论

DTD语法还存在着一些局限性。它主要体现在以下三个方面：

- 所有内容必须写入同一 DTD：没有分段，除非利用适当的声明和参数实体机制将一个 DTD 从逻辑上分为若干子 DTD。然而在解析时，验证有效性所需的声明不能从逻辑分隔的模式实体中提取出来。
- DTD与XML文档采用不同的语法（因此需要两种不同的解析机制）。
- 某些事物的表述存在缺陷，例如：类型和元组。

我们的图书目录 DTD 不太长，也不十分复杂。即使如此，我们涉及的两个概念——出版社和作者——也可以应用于其他领域。有些 DTD 声明能够跨领域重用，在新的 DTD 中，必须包含这些 DTD 组件。然而，如果问题空间比较复杂，可能会产生一些麻烦。如果你需要验证文档的有效性，就必须从逻辑上将 XML 词汇表涉及的所有内容都包含在同一 DTD 中，这使得我们很难动态构建跨领域的内容，XML 模式在这方面有所增强。

DTD 的语法与 XML 文档使用的语法截然不同。验证有效性的 XML 解析器能够处理 DTD，但是它没有义务将 DTD 展示给执行调用的应用程序。如果我们仅仅希望解析器利用 DTD 验证文档的有效性，那么解析器的该特征不会对我们构成任何影响。然而，如果我们希望查看 DTD，就需要能够展示 DTD 内容的定制的解析器。假设我需要了解 Catalog 元素的内容模型。为了编写能够根据 DTD 动态创建模板文档且允许作者填入空白的通用工具，这一功能是非常必要的。我们还可以利用该功能告知文档的作者，当他面对文档中无效的标记时有哪些可选的修改方案。我们曾经希望能够动态创建 DTD，即：根据某些运行时的情况修改内容，但是如果没有能够读写 DTD 的特殊 API，我们只能望洋兴叹。就目前的 XML 1.0 解析器而言，DTD 是只读的。实际上，这方面的需求的确在不断增长。在不久的将来，我们希望看到能够自动发现 DTD 的内容（或者通用的有效性模式）并可对之进行修改的自动化工具。

我们已经看到了有关 DTD 语法的实例。由于 DTD 元组声明的缺陷，RecSubjCategories 元素的内容模型不得不重复三次 Category 元素。我们曾经在本章的前面看到一些非常复杂的内容模式，但是它们需要依靠设计者的智慧。许多情况下，之所以采用复杂的内容模型，完全是为了摆脱

DTD元组工具的限制。在图书目录例子中，通过 `pubDate`和`pageCount`属性，我们发现 DTD需要更加强大的属性数据类型定义机制。`Price`元素本身本应该使用更强的数据类型定义。如果我们的应用程序需要对来自 XML文档的数据进行操作，而不是单纯的显示，这方面的机制是至关重要的。DTD在某些表示方面确实有待改进。

3.6 小结

文档类型定义为前一章介绍的格式正规的 XML提供了严格而精确的规则。通过几个简单的标记声明，我们能够定义 XML文档的结构以及它所允许使用的内容。DTD为XML应用程序提供了以下三项功能：

- 精确的商业问题模型的文档。
- 模型与XML解析器之间的标准通信方式。
- 验证有效性的解析器能够检测 XML文档中的错误。

第一点意味着我们将 DTD作为分析和设计工具。你需要依靠 DTD验证XML的有效性，DTD迫使设计者提供有关 XML词汇表的精确信息。XML应用程序之间的通信是以它对词汇表的理解为基础的，因此DTD向程序员和测试者指出应用程序能够描述的内容。

内部和外部 DTD为应用程序提供了获取模型的标准方法。如果应用程序要求执行有效性验证，解析器将寻找内部声明子集或者请求获取外部 DTD。

验证有效性的XML解析器可以看作是一种非常便利的错误检测机制。它能够发现基于 XML的应用程序中的结构和内容错误，为进一步的逻辑检查扫清了障碍。为此，你应该尽可能保证 DTD中的声明详细而准确。你在 DTD上花费的心血越多，应用程序出现错误的几率就越小。