

第10章 XML和数据库

XML的未来与数据库技术是密不可分的。从存储在不同介质上的数据中自动生成 XML文档的能力，从不同的数据存储交换信息的能力，将成为未来面向信息的 Internet的主要特点。动态 XML文档将会增加，变得与 XML一样，被用于装载任何东西，从天气报告到货物定单，电影回放放到体育结果，从图像到声音。

这一章的目的是看一下数据库与 XML的关系。我们将从两方面来做这件事情。首先我们将介绍如何使用数据库存储类似 XML的数据，然后我们将描述如何将 XML作为一种不同数据库之间数据交换的方法。对这两个方法的界定不是勾勒得很清楚，但是第一个与存储 XML数据的新系统的关系更加密切，第二个与遗留的系统关系更紧密——现有的数据库需要与遗留的系统交换数据。最后我们将介绍一个例子。所以，这一章可以分成三个部分：

- 存储XML。
- 将XML用于交换。
- 图书目录的例子。

在存储的部分，我们将看几个用于存储 XML的方法，并且考查一下为什么一个简单的文件系统对我们想要建立精细和复杂系统不是很适合。我们将看到文件系统的局限是什么，为什么我们需要超越XML文档，并且哪一种类型的软件包将是最适合存储 XML。

在我们考虑用来保存 XML的理想软件时，可能意识到不是每件事情都是必需的（要看你想要做些什么），但是我们至少可以建立起一套需求和想法，这样在评测产品时做到心中有数。

在交换部分，我们将看一下数据是如何在不同的存储服务器间通信的，通过使用 XML作为一种中间格式。尽管使用 XML作为一种从一个数据库取出数据放入另一个数据库的方法是意义重大的一步——例如，从关系型数据库管理系统到一个面向对象数据库管理系统，或者从 Oracle到Sybase——我们会看到对于XML的结构仍然要加以考虑。

了解了存储XML的理想方法，以及从现有数据提取信息作为 XML的最好方法之后，我们将通过图书目录例子举例说明我们的一些想法。

那么，让我们首先看看围绕 XML存储的一些问题。

10.1 存储XML

让我们从XML的存储开始，看看有哪些可用的方法。我们首先从总体上讨论存储的有关问题，然后继续阐述为什么要使用数据库而不使用磁盘文件。

10.1.1 持续性问题

为了保证到目前为止你在本书中遇到的 XML文档能够被再次使用，它们应该保存在某种存

储介质上——通常是文件系统。将信息保存起来留作以后使用称为持续性，而且它也是指在一个程序结束运行后使信息仍然可用的操作。例如，你可能很高兴地用一个字处理软件来写一封信，但是如果你关掉计算机或退出程序，就会丢掉信。持续化信件意味着将其保存为以后使用。

到目前为止，在这本书中我们对 XML 的经验是可以使用普通的文件系统中的文档保持 XML 的持续性。保存这些文档的方式与存储字处理文档或电子表格的方法相同——都保存为文件。因为它们是 XML，你可以使用众多专门的 XML 编辑工具中的某一个来编辑它们。然而，不像 Microsoft Word 文档或声音文件，XML 也是普通的文本文件，以你也可以使用文本编辑器来编辑它们。

这种技术有许多合适的应用。仍然以我们的图书目录为例，可以看到如果只有少量的出版商，而且每个出版商仅拥有几本书和几位作者，那么文档不会很大。同样，如果文档中的信息只被一、两个人使用——例如 CD 中收集的 XML 文件，我相信你可能已经将它们扔在你的 PC 中的某个地方了——那么，单个文件已经就足够了。

1. 文件系统的限制

在某些方面，基于文件的方法很好。然而，对于关键性的应用，这个文件系统的方法就不是太好了。我们将在这里小节一下这种方法的一些局限并且建立一个我们想实现的列表。

(1) 大小

第一个局限是文档大小。如果在我们的这本书的目录列表中有 20 个出版商，每一个出版商有 200 本书和 50 个作者会怎么样？我们所传递的 XML 文件将变得非常不实用。不仅仅是因为它太大了，而且如果你想维护文档的不同部分也变得难于操纵。

- 我们想处理巨大的文档，并且想检查同其他部分分离的部分文档。

(2) 并发性

正如我们可能想在文档不同部分间快速和简单地移动一样，我们也可能想让不同的人在不同的时间更新不同的部分。也许图书的编辑负责增加书名和作者，经理考虑出版商。在一个文件系统中只有一个单一文档，在一个时间只能有一个人可以处理信息。如果一个编辑需要增加新的书目，同时会计想更新作者的信息，一个人不得不等着其他人完成。的确，也可能两个人试图在同一时刻编辑同一文件，并且其中一个人所作的改变会完全丢失。

- 我们想允许许多人在同一时刻处理同一文档。

(3) 做这项工作的合适工具

也可能是这种情况，一个 XML 编辑器可能不是处理一个文档不同部分的合适工具。我们可能让会计部门来维护作者信息，而不被文档的其他部分所牵扯，对于他们来说所选择的工具可能是一个简单的表格。这个表格可能提供其他特性，如查找电话号码或电话拨号。另一方面，维护书的列表工具可能需要包含书自身的草稿文档。

- 我们想使用最适合处理数据的工具来维护文档的各个部分。

(4) 版本

一个经常考虑的重要问题是控制同一文档的不同版本。

- 我们想能够记录一个文档不同版本的轨迹。

(5) 安全

使用不同的工具处理文档的不同部分，并且允许不同的用户在同一时刻处理文档的不同部分引发出安全问题。

- 我们想控制一个文档的某一部分只有某人可以查看或修改。

(6) 综合性：集中和重复

也存在对数据的集中和重复的问题。会计部门可能已经有一个作者的数据库，用于处理皇室的支付。我们需要重复所有这些信息吗，当需要在目录中生成作者列表时，或者我们应该能够产生包含许多源的数据的XML文档吗？

- 我们想在文档中无缝地集成其他的外部数据。

2. 解决问题

用普通的文件系统来解决这些问题是非常困难的。尽管大部分的操作系统允许许多用户同时打开一个文件，目前大部分的文本和XML编辑器不能很好地控制这些文件，让用户编辑同一文档的不同部分，并且不会影响他人的工作。另外，操作系统所应用的安全机制通常是在文件级的。这就意味着不同的人可被赋予不同的处理权限处理不同的文档，但是不能处理同一文档的不同部分。

为了解决上面所提出的问题，需要理解关于XML文档的不同观点。实际上，我们需要从我们基本的XML单元是一个文档的想法中脱离出来，而把它考虑成可操作的节点集。

文档和节点

对于那些刚接触XML的人来说，经常困惑的是假设存储XML文档的基本单元必须是在一个文件中的普通文本。因为一个文档必需是人可读的，所以假设就是文件必需被存储为一个文本文件，或者在一个文件系统内，或者也许在一个数据库的一个字段里。这个实际上是对XML 1.0规范的一种错误理解。

一个应用程序——你或我可能写的一个计算机程序——不是想要直接处理XML文件，而是作为一个解析器运作，或XML解析器。XML 1.0规范的介绍（第一部分）中说：

称为XML处理器的软件模块是用于读取XML文档和提供对他们的内容和结构的处理。一个XML处理器的工作基于另一个称为应用的模块的行为。这个规范[XML 1.0]描述了一个XML处理器应具备的行为，根据它应该如何读XML数据，并且信息是如何提供给应用的。

换句话说，尽管我们花费了大量时间来看XML文档和羡慕它们的结构，现实是我们应该考虑的文档应根据文档所能够表示的节点结构。DOM的角色——是否是用Java或VB写的；是否是IBM或Sun写的——都向我们隐藏了分级结构和让我们操作一棵树的节点。当然，当我们在讨论数据或在打印页面中表现它们时，就如同在本书中，做到这一点的最容易的方法是使用XML标记语法；然而，我们应该记住这种分离（参见图10-1）。

这些没有一个对你来说是新的——定位这些节点的方法是DOM的任务，并且已经在第5章中讨论过——那么你可能会问为什么我要重复它。嗯，这意味着我们对XML文档的存储的理解有些改变。比起简单地寻找处理文本的机制——比方说，文件系统或数据库的文本字段——我们需要理解不同的工具和产品是如何处理分级节点集的。XML作为文本，仅仅变成一种在它们的节点上从一个系统到另一个系统转载信息的便捷的方法（参见图10-2）。

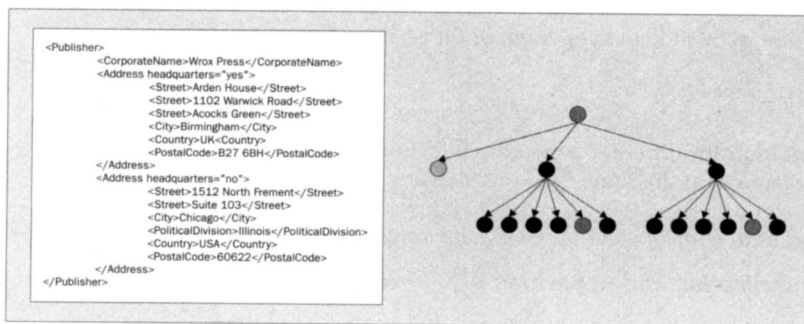


图 10-1

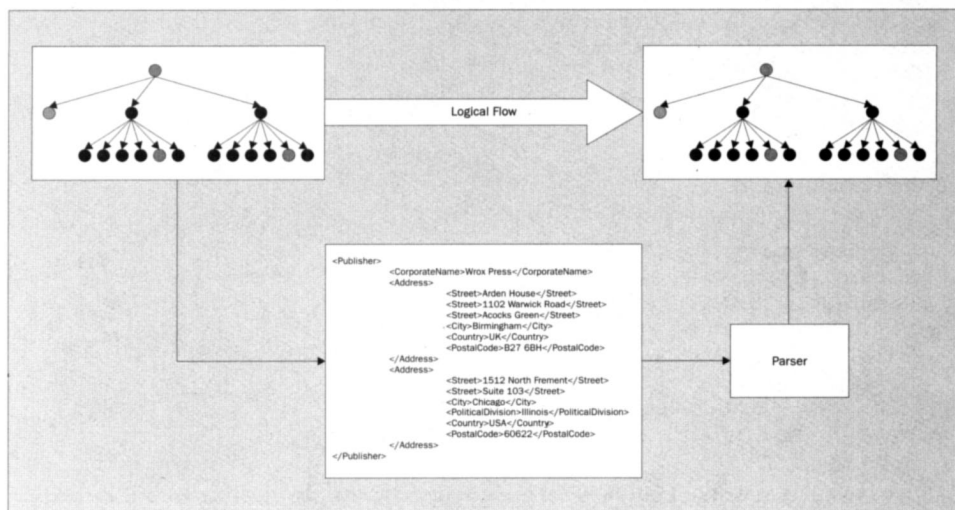


图 10-2

然而，传送节点的方法仍然是 XML 文档；然而，对于解析器来说，一个 XML 文档不过是一个输入单元。以上图中的文档为例：

程序清单 10-1

```

<Publisher>
  <CorporateName>Wrox Press</CorporateName>
  <Address headquarters="yes">
    <Street>Arden House</Street>
    <Street>1102 Warwick Road</Street>
    <Street>Acocks Green</Street>
    <City>Birmingham</City>
    <Country>UK</Country>
    <PostalCode>B27 6BH</PostalCode>
  </Address>
  <Address headquarters="no">
    <Street>1512 North Fremont</Street>
    <Street>Suite 103</Street>
    <City>Chicago</City>

```

```
<PoliticalDivision>Illinois</PoliticalDivision>
<Country>USA</Country>
<PostalCode>60622</PostalCode>
</Address>
</Publisher>
```

没有理由说，为什么我们不能取出这个文档的地址信息，并且创建两个更完美的可接受的XML文档。这就是在讨论XML时必然想到的。例如，我可能说 Wrox Press的UK地址是：

程序清单 10-2

```
<Address headquarters="yes">
  <Street>Arden House</Street>
  <Street>1102 Warwick Road</Street>
  <Street>Acocks Green</Street>
  <City>Birmingham</City>
  <Country>UK</Country>
  <PostalCode>B27 6BH</PostalCode>
</Address>
```

并且US地址是：

程序清单 10-3

```
<Address headquarters="no">
  <Street>1512 North Frement</Street>
  <Street>Suite 103</Street>
  <City>Chicago</City>
  <PoliticalDivision>Illinois</PoliticalDivision>
  <Country>USA</Country>
  <PostalCode>60622</PostalCode>
</Address>
```

尽管我们没有设计出将如何做到这样，通过处理节点，至少我们已经创建了为两个人对同一个XML文档进行工作的可能性——通过允许他们处理分离的节点。倘若我们可以建立一个系统，它允许独立的节点控制，我们就可以创建一个系统，它看上去允许处理一个XML文档的不同部分。不同的用户可以指定不同的节点集进行处理——以一种有效的XML文档的形式。继续考虑我们的两个地址，将包括如下部分（见图10-3）。

一个服务器，按这种方法被设计成保证节点的有效——或者XML文档，它可能是某个大的XML文档的一部分——我们将它称为XML服务器。把这些概念记住是很重要的，在评估不同的产品时，因为很多的应用程序声称能够处理XML，但是可能不允许像我们所期待的控制方法。我们将在后面讨论一下这些应用程序。

某些人争辨说，这个“节点模型”没有什么名气，因为节点本身与它们所表示的元素无关。例如，为了处理<Address>元素的headquarters属性，我们应该能够使用：

```
var hq = n.headquarters;
```

在使用DOM的地方我们必需使用：

```
var hq = n.attributes.getNamedItem("headquarters").value;
```

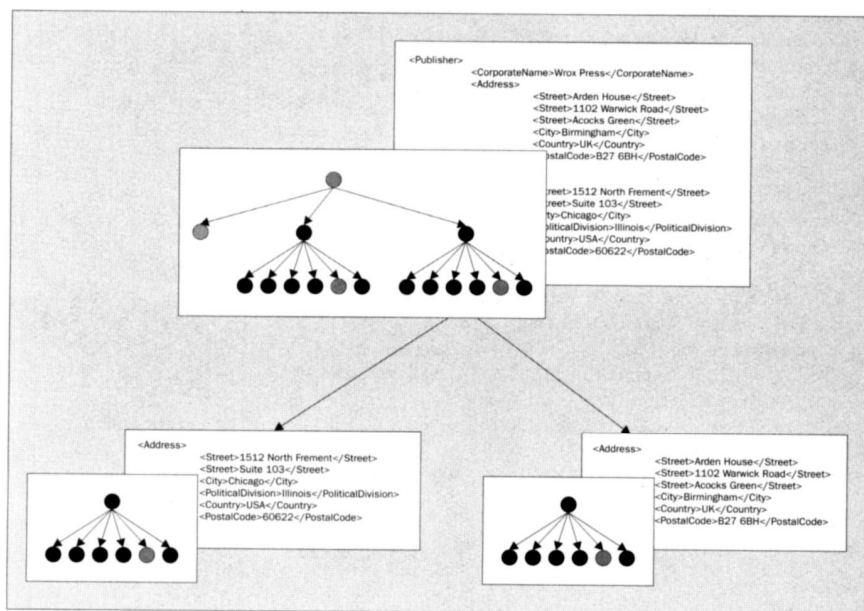



图 10-3

每个节点从本质上说是一个有着自己的属性和方法的对象。这个方法叫做树林范式。参考 Robin Cover 的站点得到更多的细节，在 <http://www.oasis-open/cove/topics.html#groves>。同时这个方法产生一些重要的问题，被激烈地讨论，但不太可能会很快出现在你的系统里。

3. 小结

为了真正地取得对 XML 文档的控制，我们需要解决在文件系统的局限性中所遇到的问题：

- 我们想处理极大的文档，并且想检查同其他部分分离的部分文档。
- 我们想允许许多人在同一时刻处理同一文档。
- 我们想使用最适合处理数据的工具来维护文档的各个部分。
- 我们想能够记录一个文档不同版本的轨迹。
- 我们想控制一个文档的某一部分只有某人可以查看或修改。
- 我们想在文档中无缝地集成其他的外部数据。

为解决这些问题，我们必须有在文档中处理任何节点的能力。

10.1.2 文档管理

在了解节点的操作之前，有必要花些时间在中间技术上——在数据库中存储文档。我们已经确定，使用文件系统来处理文档不是一个用来解决安全性和用户数量的有效方法。我们也看到，在节点级进行操作——比起对文档来——可能需要某种方法来解决。在解决这些问题之前，我将简要地谈到“在中间（in between）”的方法，用于改善处理文档的方法。这个方法要比文件系统复杂得多。

可能正好是这种情况，在你的项目中你可能将 XML 作为一系列的文档。也许你只是需要某个比文本文件或文件系统稍好一些的系统，但是对独立的节点加锁和安全控制上更精细一些。我们将很快地从整体上浏览一些处理 XML 文档的存储和提取的产品。其中的某些产品可以满足我们关于团队开发和版本管理的需求，然而，因为它们将文档作为处理的基本单元，所以不可能满足我们的所有需要。通过看到这些局限，应该更清楚为什么我们需要某个更复杂的解决方案。

1. DBMS 解决方案

把 XML 文档作为关系型数据库管理系统的一个文本字段进行存储是很直接的，并且对于任何数据库产品都是可能的。然而，一旦存储完成，文件就不再是一个文本字符串——XML 的存储与你在数据库中保存图像或文本文档采用相同的方法。Oracle 在它的 8i 产品中增加了一个特性，它允许 XML 文档被存在一个字段中，或按 XML 进行查询。这是通过扩展普通的 SELECT 语句来完成的。这对于一个强大的文档存储技术有利；一个表中的一条记录可以有一个字段来保存 XML 文档，并且其他的字段可以用来保存文档的信息，像是否被加锁，谁最后修改过它，等等。

与一般的开发项目相比这些特性最可能被某人用于建立一个自己的全 XML 服务器。虽然从本质上说是两种数据模型的混合体——关系型数据和层次型结构化数据——它将可能建立或者是一个 XML 文档服务器或甚至 XML 节点服务器。在像 Oracle 这样的 RDBMS 的上面构造这样的应用明显要比在一个不支持 XML 的 RDBMS 上做这样的事容易得多。

你可以通过访问 http://technet.oracle.com/tech/xml/xsql_servlet/ 获得更进一步的信息。一个 servlet 也可以将原来的 SQL 查询转换为 XML。Microsoft 也发布了一项技术，它能够将数据库查询自动地转变为 XML。更进一步的信息请看 <http://msdn.microsoft.com/workshop/xml/articles/xmlsql/default.asp>。

2. OODBMS 解决方案

面向对象的数据库管理系统（OODBMS）已经出台很长时间了，并且在我们想看到的理想的 XML 服务器方面提供了许多的特色。然而，大部分商家中的一少部分已经开始把他们的数据库产品用于非常强大的 XML 文档仓库。让我们看一下其中一些产品：

(1) Inso Corporation DynaBase 3.1

DynaBase 基于 ObjectStore 4.0，但是看上去在处理 XML 文档上并不聪明。它们按照一个图片或电子表格那样存储，尽管使用所有的面向对象的优点来管理它们。更多的细节可以在 <http://www.inso.com/> 和 <http://www.ebt.com/dynabase/> 上找到。

(2) Chrystal Astoria 3.0

Astoria 允许组件的存储，它可以是任何类型的文档。基本上是一个使用 XML/SGML 的团队开发产品，Astoria 可以同其他的批准工具一起控制团队处理组件。它支持搜索、授权、编辑、回顾和修定、翻译和多媒体发布。更多的信息，参见 <http://www.chrystal.com/>。

(3) DataChannel Rio 3.2

Rio 在文件夹中存储 XML 文档，这样用户可以有多种层次来处理。文档可以是 Microsoft 的 Office 文件，并且可以自动进行转化。Rio 产品主要是涉及到保证用户在企业内部网上保证与最新文件同步。然而，最新的发行版使用了 X-Machine，来自于 Software AG（参见后面的 Tamino）。

同样，注意DataChannel站点称它为XStore，但是没有在Software AG的站点上提到XStore。然而，X-Machine的描述填在了DataChannel叫做XStore的海报上。) Data Channel的站点是：<http://datachannel.com/>。

(4) Vignette StoryServer

生成动态文档内容，并且能够发布文档到其他的系统中，包括XML。他们最近宣布SyndicationServer允许存储在他们的系统中，可以按ICE (Information & Content Exchange——关于ICE在第12章有更多的介绍) 格式输出，该格式遵守XML的语法。另外一个考虑使用XML作为一种发布环境的是Vignette同Quark的合作，它是到处存在的QuarkXPress产品的开发者。请看<http://www.vignette.com>得到更多的细节。

3. 结论

文档管理技术的出现是为了克服文件系统中存储XML的局限。RDBMS和OODBMS产品将允许体积巨大的信息被存储、加锁，等等。这些可能对某些项目来说是足够了，但是这些产品更可以被用来作为发现XML节点服务器使用的基础。

10.1.3 XML存储和数据库

让我们回到用文件系统存储XML文档的局限上来，看一下数据库的一些特性，可能会帮助我们找到解决办法：

- 大小——数据库通常可以处理海量的数据信息。因为这个信息可以以一种很小的单位来访问——我们可以指出一个作者或两本书——那么就很容易导航。
- 并发性——数据库被设计为允许多个用户同时处理信息。例如，大部分的产品将允许一个用户处理作者，同时另一个处理书目。他们也允许一个用户编辑一个作者，同时另一个用户编辑另一个作者。
- 完成这项工作的合适工具——数据库通常是后端的产品，用于很多不同类型的应用。就如同所有的应用可以从文件系统中读和写一样，也可以处理数据库。一个数据库的界面包括Microsoft的Access表格，一个Web表格，或者是另外的处理信息的数据库。
- 版本——关系数据库一般没有对版本的本地支持，但是许多面向对象的数据库则有。
- 安全——正如数据库允许一个良好的控制，用于控制存在其中的数据的处理，那么，它们通常也允许不同级别的处理。一个用户可以看到作者，但是不能编辑它们，另一个可能增加新的书目，但是不能删除它们。
- 集成性——数据库对于可重用信息的共享非常适合。从会计部门来的作者数据可以在图书目录中所共享。

看上去数据库技术的确很适合我们的需要。关系数据库和面向对象数据库两个都允许控制小数量的信息。RDBMS和OODBMS都允许人们编辑同一个区域的不同片段，就像我们在上面讨论过的两个地址的例子一样。对象数据库将更进一步地允许将安全设在独立的对象上（关系型数据库通常只允许控制某类的所有对象）。

在这一部分，我们将看一下对象和关系型数据库技术，并且看一看它们各自的好处，关于模型化和存储分级的节点，正如我们在本节的第一部分所讨论的。

1. 面向对象数据库

近几年来，以对象的形式模拟真实世界的数据库已经可用了。尽管处理信息的速度比不上关系型数据库，但它的好处是，在使用了面向对象的编程技术时，可以让信息更易于管理。从我们当前讨论的观点中的主要好处是他们非常直观地模拟我们的节点分级。

在讨论为什么面向对象在建立 XML 文档上可能是一个有用的方法之前，我们先粗略地看一下OO的原则。

(1) 面向对象的程序设计

面向对象的程序设计——即OOP——提供一个非常有效和可靠的机制，用于编写结构良好的程序。一个对象可以是程序员希望任何东西，从切实的，就像汽车或一个订单，到更抽象的，就像一个列表。一个类定义是说每一个对象应该是什么：有什么属性；是否还包含其他类；并且类中有什么属性。

例如，对于作者的类定义可能详细说明了，一个作者有名和姓，他写的书的列表。然而，一个作者的对象类将拥有实际的值，就像“Stephen”和“Mohr”。当一个对象从它的类定义中被创建，这个类就称为被实例化了。

用OO方法来产生可靠代码的一个好处是与一个对象有关的信息可以被封装。这个可以有效地用于在可以被使用的层次上展现信息。通过允许一个类从另一个类继承，封装的类可以被用于建立在经过使用和测试的代码上。例如，假设我们有一个 Java 类 Person，定义如下：

程序清单 10-4

```
public class Person
{
    public String FirstName;
    public String MI;
    public String LastName;

    public Person(String FirstName, String MI, String LastName)
    {
        this.FirstName = FirstName;
        this.MI = MI;
        this.LastName = LastName;
    }

    public String fullName ()
    {
        return this.FirstName + " " + this.MI + ". " + this.LastName;
    }
}
```

并且有一个fullName()的例程，它已经被彻底地测试过，并且发现非常稳定。我们理所当然想从中得到好处。当我们在Wrox这个程序中开始定义Author类，可以通过继承建立在Person类上：

程序清单 10-5

```
public class Author extends Person
{
    public int authorCiteID;
    public DCollection books;
```

```

public Author(int authorCiteID, String FirstName, String MI,
               String LastName)
{
    super(FirstName, MI, LastName);
    this.authorCiteID = authorCiteID;
}

public void printBooks()
{
    Enumeration elements;
    System.out.println("Books written by " + this.fullName() + ":");
    for (elements = books.elements(); books.hasMoreElements();)
    {
        System.out.println(elements.nextElement());
    }
}
}

```

我们现在可以将精力集中在实现 Author 类上,建立在 Person 类的设计者已经完成的工作之上。不需要我们写一行额外的代码,任何 Author 已经自动地继承了 Person 的特性。所以可以参照下面的代码显示一个作者的名字:

```

Author a = new Author(1, "Stephen", "", "Mohr")
System.out.println(a.fullName());

```

(2) OO和XML

可能并不需要总是关注在前面部分中对象与 XML 元素之间的密切关系。回忆一下第 7 章中的模式,一个作者可能看到以下内容:

程序清单 10-6

```

<Author authorCiteID="4">
  <FirstName>Frank</FirstName>
  <MI />
  <LastName>Boumphrey</LastName>
  <Biographical>
    Frank Boumphrey currently works for Cormorant Consulting, a firm that
    specializes in medical and legal documentation. His main objective at
    the present is to help XML to become the language of choice in web
    documents.
  </Biographical>
  <Portrait
    picLink="http://webdev.wrox.co.uk/resources/authors/boumphreyf.gif" />
</Author>

```

为了使用 DOM 创建这样一个元素将要求大量调用 createElement() 和 setAttribute(), 这样可能会有很多的错误。使用刚刚介绍的基于面向对象的技术,我们看到创建作者的对象更稳定。例如,下面的 JavaScript 创建刚列出的作者文档中的第一部分:

程序清单 10-7

```

oNode = oParser.createElement("Author");
oNode.setAttribute("authorCiteID", "4");

oTemp = oParser.createElement("FirstName");

```

```
oTemp.text = "Frank";  
oNode.appendChild(oTemp);  
  
oTemp = oParser.createElement("LastName");  
oTemp.text = "Boumphrey";  
oNode.appendChild(oTemp);
```

当然可以通过使用函数来让以上部分小些，但是面向对象方法的好处是处理数据的模型，而不只是节点。在Java例子中，可以说插入一个作者，而不是一个名为‘author’的节点。

(3) 持续性

OO数据库填补了Java等应用程序对持续性对象的需求。虽然也可以在一个磁盘文件或一个关系数据库中表示Author对象，但是前者的效率很低，后者要求程序员经常在两个数据模型中切换——关系型和层次型。的确存在很多工具，用于映射在像Java和C++一类语言中的类与关系数据库，但是对于在一个关系数据库中模拟分级数据要求对那些表做很多的连接（我们将在下一部分看到连接和关系数据库）。如果对象树很深，这就可能用很多表来表示，结果也会相当的慢。

(4) 好处

OODBMS好处是在当我们真的想处理关系和深度复杂的对象时候。然而，因为它们不像一般的RDBMS那样快，如果复杂度不大的话，后者仍然可以用来模拟对象结构。我们将看一下在两个使用对象技术的产品上演示前面所讨论的大部分的高级技术。

(5) 面向对象数据库产品

让我们看一下两个面向对象数据库：

(6) Object Design——eXcelon 1.1

eXcelon产品是受到广泛赞誉的OO数据库，最近加入了XML的支持。虽然一般的ObjectStore 4.0——存在于eXcelon下的数据库——的行为是用于存储文件，在对象的分层中将其作为一个单元（看一下前面的文档管理部分），eXcelon确实分析XML输来产生新的对象。换句话说，比起每个对象在数据库中是一个全XML文档，它用的只是文档中的节点。这些节点可以通过使用XQL查询（参见第8章）而得到（参见图10-4）。

由于eXcelon能够分析节点，并将它们保存为唯一的对象，因此eXcelon领先于其他产品。ObjectStore底层的OO技术完全能够操纵这些节点。Web站点<http://www.odi.com/excelon/>上列出了更多的信息。

(7) POET——Content Management System 2.0

CMS是基于Object Server Suite(OSS) 6.0，POET是自己所有的面向对象数据库。同将输入的XML转化为eXcelon对象的方式有着相似的行为，CMS允许节点在任何一级被作为“组件”进行描绘。这些组件相互之间是独立地用于注销和版本控制，等等，这就意味着两个人可以在同一时刻编辑同一文档的不同节点。

在结构中搜索可以有两种方法来执行，或者作为标准的文本进行广泛的搜索，或者在标记内。然而，后一种方式并不区分是哪一级别的标记，所以如果叫<Title>元素出现在我们的不同文档中，它们将被找到。所以它们不能像我们在后面要说的XPath一样复杂。

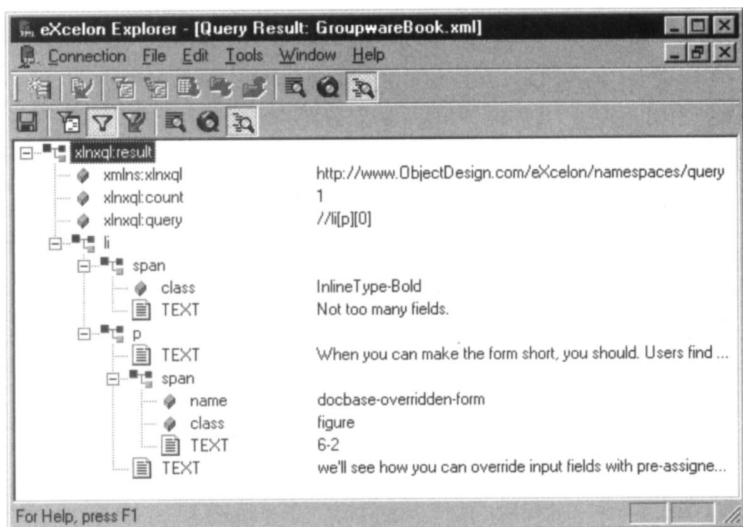


图 10-4

然而，一个非常有力的特点是重用内容的能力。一个组件可以从其他的地方引用，但是只需要维护一次。组件的方式仿佛确实是在它所引用的位置上（参见图 10-5）。

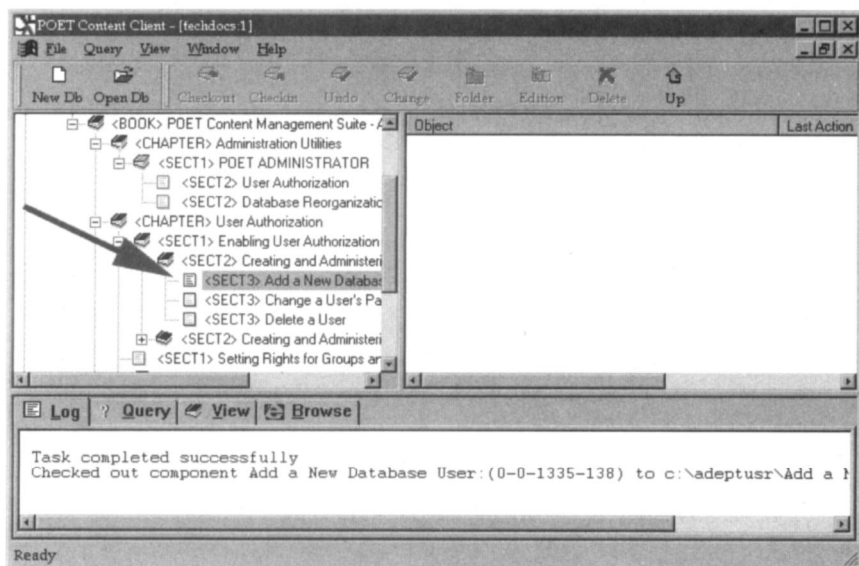


图 10-5

像其他的一些产品一样，CMS具有组件登录（check-in）和注销（check-out），版本控制和样式转换等功能。更多的信息请访问 <http://www.poet.com/>。

(8) 小结

OO数据库的本地能力使它们成为操纵 XML文档的最有可能的后选对象。独立操纵节点的能

力——是否加锁、安全或版本控制——给了它们满足前面所提到的高级需求的潜力。这里提出的两个应用程序的每一个都提供一个可靠的基础，用在一个基于需要先进的存储和获取处理的XML的项目中。

如果每件事都很好，你可能会问自己收获是什么。在许多方面，文化比技术更重要。OO数据库软件安装的库只是关系数据库的一小部分，这一点似乎也不会很快发生改变。当然像这里所介绍的产品可能会取得一些进展，原因就是它们可以很好地管理XML，但是用OO技术的产品很少。

这不是说不存在技术问题。听上去很明显，但OO数据库比对象数据更好。在模拟其他类型的结构时，它们不像关系数据库一样有效，然而在分级的情况下是非常快的——比方说几个对象，每一个都包含几万个其他的对象。

所以，尽管我们找到了一个用OO数据库的很好的解决方案，但必需继续探索，因为大部分你要处理的是关系型存储介质。

2. 关系数据库

关系数据库管理系统——或RDBMS——使用熟悉的行和列的方法来存储数据（参见图10-6）。

authorCiteID	FirstName	MI	LastName	Biographical	Portrait
1	Stephen	<NULL>	Mohr	Stephen began pro	http://webdev.wro
2	Kathie	<NULL>	Kingsley-Hughes	Kathie is the MD of	<NULL>
3	Alex	<NULL>	Homer	Alex is a software c	http://webdev.wro
4	Frank	<NULL>	Boumphey	Frank Boumphey c	http://webdev.wro

图 10-6

这个模式显示了一张表，它将包含作者。每一个作者用一行表示，一个作者的每个属性用一列表示。

抛开OO数据库的众多优势，到目前为止RDBMS仍然是最流行的，因为它们可以表示太多的现实世界的问题，并且对很多这样的问题给出更快的响应时间。例如，货物订单。这些例子都非常适用于这个模型，作为一个二维数组的集合，像联系管理系统，股票控制软件包，等等（参见图10-7）。

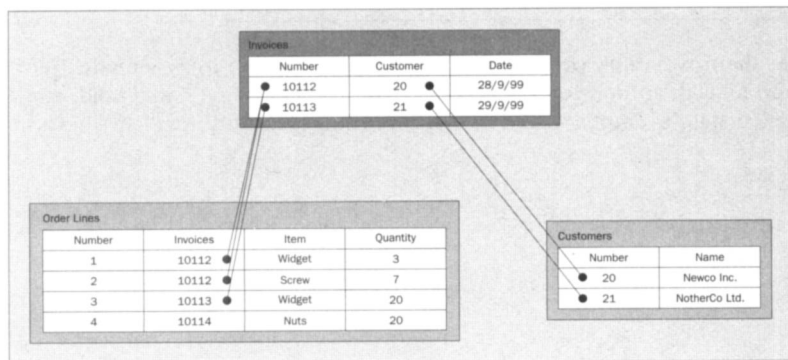


图 10-7

基本的关系数据库概念是：

- 表
- 查询
- 连接

下面的几个部分介绍了这些概念，如果你对使用关系数据库很有经验，可以忽略这些概念。在学习过这些基础知识后，将看一下如何用关系数据库来模拟节点的分级，然后是如何自动完成某些转换。你将需要熟悉这些概念，并了解这个模拟是如何做的。

(1) 表

一个简单的用来存储作者的书目的数据表可能包含如表 10-1所示的内容：

表 10-1

作者城市编号	名	MI	姓	简 历
1	Stephen		Mohr	Stephen高中起即开始编程，现在是Omicron Consulting的资深软件系统工程师。他使用C++、Java、JavaScript、COM和各种互联标准和协议设计和开发系统
2	Kathie		Kingsley Hughes	Kathie是Kingsley-Hughes开发公司的MD，这是一家专注于Web开发和可视化编程语言的培训和咨询公司
3	Frank		Boumphrey	Frank Boumphrey现在在Cormorant Consulting供职，这是一家专门为医药和法律文档提供服务的公司。它主要负责将XML应用在Web文档中

在表中每一个作者有着他们自己的入口或行。每一行是由许多字段组成的，对每一个作者是通用的，例如，每个作者都有一个姓的字段和名的字段。即使一个特别的字段为空，像每一个作者的MI字段，但在每一行中仍然会存在。

表头对应于表定义中的列。这些可以被定义如下形式（参见表 10-2）：

表 10-2

列名	数据类型	是否允许空值
AuthorCiteID	Integer	否
FirstName	String	是
MI	String	是
LastName	String	否
Biography	String	是

用这个定义，我们已经指明了要存储的每个作者的属性，并且指出哪些字段必须存在，哪些是可选的。因为很少有作者没有一个唯一的ID和姓，我们已经制订了一条记录的最小需求。

在决定了如何在表中存储数据之后，我们需要一种机制来取出它。

(2) 查询

处理存储在关系数据库中数据的通用办法是使用结构化查询语言或SQL。尽管在这几年中对于SQL中的可用的特性有了改进和增强，它的核心仍是SELECT语句。使用这个语句，请求RDBMS服务器返回给我们所有在数据库中的作者的请求应该是如下形式：

```
SELECT * FROM Author;
```

对那些新手来说的一个常见的错误是将星号读成“请检索所有的记录”。这个理解是将星号的作用等同于在目录中查找文件时*的作用，但是在SQL中，它的意思是“请检索所有的列”。

在前面我们定义的作者表的例子中，将意味着返回五列数据。一个只取出指定列的查询，应该是这样：

```
SELECT FirstName, LastName FROM Author;
```

一个SELECT语句的结果形成一个结果集——一个查询结果的列表，可以被单步遍历，取出每一行的数据——在这个例子中将包含如下的结果（参见表 10-3）：

表 10-3

名	姓
Stephen	Mohr
Kathie	Kingsley-Hughes
Frank	Boumphrey

如你所见，即使没有星号我们也从数据库中取回了所有行，因为这是SELECT的缺省行为。一个限制返回记录的方法是使用一个WHERE子句。例如，如果我们需要Stephen Mohr的传记，我们应该书写下面的查询：

```
SELECT Biographical FROM Author WHERE LastName = 'Mohr';
```

在这个例子中，我们要求数据库返回字段 LastName被设成Mohr的所有记录，然后只返回给我们Biographical字段。你可能希望WHERE子句非常丰富，下面举出的例子将返回Kathie和Stephen两个人的传记。

```
SELECT Biographical FROM Author WHERE LastName > 'K';
```

(3) 连接

之所以称为关系数据库是因为它们可以处理关系。连接的主要目的是在一个数据中建立表之间的关系。尽管也可能把数据存储在一个巨大的表中，没有连接，它将很快在资源使用及响应时间上变得低效。例如，没有人阻止我们创建像这样的系统（参见表 10-4）：

表 10-4

FirstName	LastName	Biographical	Title	ISBN
Stephen	Mohr	XML Application	1-861001-52-5
Stephen	Mohr	Designing Distributed Applications with XML, ASP, IE5, LDAP And MSMQ	1-861002-27-0
Kathie	Kingsley Hughes	XML Application	1-861001-52-5
Frank	Boumphrey	Professional Style Sheets For HTML and XML	1-861001-65-7
Frank	Boumphrey	XML Application	1-861001-52-5

然而，很明显，不需要重复每一个作者的名字和传记的细节在每一本他们写的书上，同样

也没有地方显示参与每本书写作的每个作者的名字。不仅仅是因为低效，也是因为难于维护，当一个作者的信息改变时就需要将它拷贝到每一个出现的地方。

为了实现在书与作者之间更高效的关系，我们需要建立多对多的关系。这就要求一个特别的技术，稍后将进行介绍。在介绍之前，先看一下一对一和一对多的关系应如何建立。

(4) 一对一和一对多连接

你可能还记得我们的书的目录，一本书可以包括一个价格，每个价格可以有一个货币符号：

程序清单 10-8

```
<AttributeType name="currency" dt:type="enumeration" dt:values="USD GBF CD"
    required="yes" />

<ElementType name="Price" dt:type="fixed.14.4" content="textOnly">
    <attribute Type="currency" />
</ElementType>
```

如果我们采用刚才提到的方法——将所有数据都放在一张表中——那样它将像表 10-5 所示存在数据库中：

表 10-5

Title	ISBN	Currency	Price
XML Application	1-861001-52-5	USD	49.99
Designing Distributed Application	1-861002-27-0	USD	49.99
With XML, ASP, IE5, LDAP and MSMQ			
Professional Style Sheets for HTML And XML	1-861001-65-7	GBF	45.99

然而，模式可能会认为只有三种货币符号是可用的，每一个价格（指书）只能使用一种货币符号，尽管并不反对每个货币符号被使用多次。所以我们可以书的实体与一个新的实体，货币符号，之间建立一种一对多的关系。合理化数据的过程叫做标准化，将在下面更详细地讨论。将货币符号分离到另一个表中会给我们带来几个好处：

- 可以存储货币符号的附加信息，像用在下拉框和报告中的货币符号名称，也可能是一个汇率。
- 我们可以保证在列表中只有一个符号被输入到书的表中。

我们通过增加一个货币符号的新表来实现它（参见表 10-6）：

表 10-6

货币编号	缩写	
1	USD	美元
2	GBF	英镑
3	CD	加拿大元

CurrencyID列是表的主键。用于确定表中的每一行都是唯一的，也用来记录在其他表中的地址的方法。尽管在这个例子中我们可以使用 ShortName作为主键，但是有一个普遍的经验就是使用数字来连接表，因为它们是压缩的，并且可以比较得很快。

在建立起我们的货币符号表之后，修改书的表以便从货币符号表中指出它的货币符号（参见表10-7）：

表 10-7

Title	ISBN	CurrencyID	Price
XML Application	1-861001-52-5	1	49.99
Designing Distributed Application With XML, ASP, IE5, LDAP and MSMQ	1-861002-27-0	1	49.99
Professional Style Sheets for HTML And XML	1-861001-65-7	2	45.99

这里我们看到通过使用主键的值建立了一个引用。因为 CurrencyID在书表中看上去不像在货币符号表中是主键，我们说它是一个外键。在某些数据库系统，可能会加强这种主键与外键的关系，例如，我们不能把数字 10 000放到字段中，因为在货币符号表中不存在相应的记录。设置一个列具有一个外键的属性是用来保证数据一致性的重要方法。

现在来通过在 WHERE子句里使用变量生成书和它们的货币符号的记录集：

```
SELECT Title, ShortName, Price
FROM Book, Currency
WHERE Book.CurrencyID = Currency.CurrencyID;
```

像前面一样，我们想取出的列被列在 SELECT和FROM之间，但是这一次我们要求两个不同的表中的数据，书的表和货币符号表。然而，为了让每一行有意义我们要求书表中的每一行都要与货币符号表中的每一行相匹配。但是我们不是想要随便的一行，而是想要与在书表中的货币符号数字相同的行。实际上，在书表中的 CurrencyID列执行了一个查找，在货币符号表中获得它所需要的细节。结果集看上去将同表 10-8所示。

表 10-8

Title	ShortName	Price
XML Application	USD	49.99
Designing Distributed Application With XML, ASP, IE5, LDAP and MSMQ	USD	49.99
Professional Style Sheets for HTML And XML	GBP	45.99

这时，尽管已经将我们的数据分割成不同的表，但可能在任何时候通过连接重新得到它，就好像是一个表。

这个货币符号的例子模拟了一对多的关系——一个货币符号与多本书有关——但是同样的技术也可用来模拟一对一关系。例如，我们可能在一张表中保存价格和货币符号，并且从一本书中指出全部的价格。一本书应该只有一个价格。

(5) 多对多的连接

刚刚描述的方法，是在一个表中的一列中设置了对另一个表的一列的引用，对于一对一或一对多的关系很好。然而，它不允许我们表示在书到作者的假设中所需要的多对多的关系。提醒一下，我们说过一本书可以有多个作者，并且一个作者可能写过许多书。为了实现这一点，需要一个单独的表，除了我们试图建立的关系外什么都没有。

让我们假设有如下的作者表（参见表 10-9）：

表 10-9

作者编号	名	姓	简历
1	Stephen	Mohr
2	Kathie	Kingsley-Hughes
3	Frank	Boumphrey

注意每一个作者有一个唯一数字——或主键——用来标识他们，就像货币符号那样，但是在这个表中没有对书的引用。现在，证我们看一下书表（参见表 10-10），对于每一本书也有一个唯一的ID：

表 10-10

书籍编号	书 名	ISBN
1	XML Applications	1-861001-52-5
2	Designing Distributed Applications with XML, ASP,IE5, LDAP and MSMQ	1-861002-27-0
3	Professional Style Sheets for HTML and XML	1-861001-65-7

再次注意，就像在作者表中没有对书表的引用一样，这个表也没有对作者的引用。不像一对一和一对多关系那样使用表的额外字段来实现，我们需要建立第三张表，用来维护多对多的连接。这个额外表参见表 10-11。

这个表只包含了其他两个表的关系。这个技术可以被扩展，可以提供需要的更多表的关系。那么我们如何从数据库中提取信息呢？为了找到 Stephen Mohr 所写的所有书，我们将使用下面的 SQL 语句：

表 10-11

BookID	authorCiteID
1	1
1	2
1	3
2	1
3	3

程序清单 10-9

```
SELECT * FROM Author, Book, BookAuthor
WHERE BookAuthor.authorCiteID = Author.authorCiteID
AND Author.LastName = 'Mohr'
AND Book.BookID = BookAuthor.BookID;
```

并且，为了找到“XML Applications”这本书的作者，我们使用语句：

程序清单 10-10

```
SELECT * FROM Author, Book, BookAuthor
WHERE BookAuthor.BookID = Book.BookID
AND Book.Title = 'XML Applications'
AND Author.authorCiteID = BookAuthor.authorCiteID;
```

注意这些查询将这三个表进行了连接。第一个查询进行如下工作：

- 匹配 BookAuthor.authorCiteID 到 Author.authorCiteID 的 WHERE 子句部分用来取得在 BookAuthor 表提到的所有作者。

- LastName到‘Mohr’的比较部分用来将作者的列表缩小到 Stephen Mohr。
- 在两个BookID字段进行的比较查找 Stephen 所写的书。

使用这个连接，我们可以提出关于书和它们到作者的关系的问题。完全的表的配置看上去如图10-8所示。

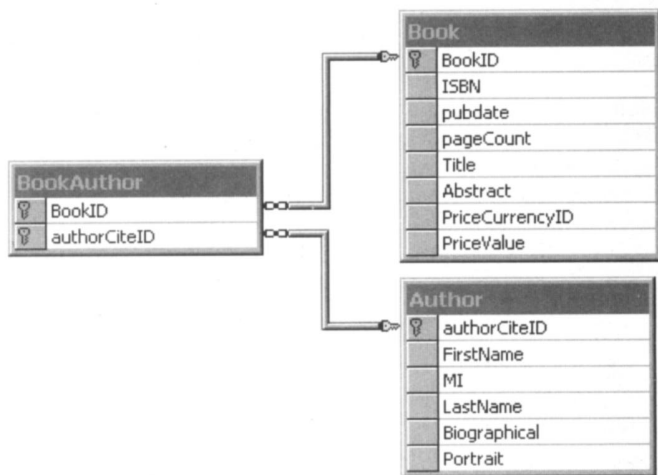


图 10-8

(6) RDBMS和XML

让我们回到这一部分的中心议题上来；对于关系数据库所给出的这些特性，它们能够很好地提供对XML文档操作的高级功能吗？为了解决这个问题，我们需要检查关系数据库是如何模拟节点信息的。

记起我们关于表的讨论，可以看到一行可以很容易地表示一个元素，用表的列来保存属性。例如，在前面表配置图中的 Author表很容易保存书目录的<Author>元素。

我们知道一些元素也包含文本属性，但是这个可以使用一个已命名的列进行模拟，比如PCDATA。因为它有一个我们知道的名字，这样当进行输出时，不应该以一个属性被取出而应该是作为元素。

书的目录中关于这一点的一个例子是<Imprint>元素，看上去如：

```
<Imprint shortImprintName="XMLAPP">XML Applications</Imprint>
```

在Imprint表中的数据如图10-9表示：

在一个表中将元素映射成元素相当直接，但是只是完成了一半；我们仍然需要表示节点间的关系。为了做到这一点，我们需要使用与在前面部分所看到的不同类型的连接。我们可以模拟父亲/孩子关系，简单地通过在表中创建一个外键，用它来表示孩子，在表中用主键表示父亲。例如，我们刚创建的Imprint表可能被扩展成（参见图10-10）：

其中fk_Imprints字段是一个引用——或外键——做为父表Imprints的主键。这个表被提出面向出版商为（参见图10-11）。

attr_id	shortImprintName	pdata
DDA		Designing Distributed Applications
XMLAPP		XML Applications
PROXML		Professional XML

图 10-9

pk_Imprint	fk_Imprints	attr_id	shortImprintName	pdata
1	1	DDA		Designing Distributed Applications
2	1	XMLAPP		XML Applications
3	1	PROXML		Professional XML

图 10-10

pk_Imprints	fk_Publisher
1	1

图 10-11

(7) RDBMS的局限

刚看上去好像相当有用；父亲和孩子节点的分级表示可以通过设置从子表到父表的一种引用来建立。然而，这里面有一个主要的问题，父 / 子关系太严格了。例如，用我们刚才提出的方法，就不可能表示程序清单 10-11 所示的结构：

程序清单 10-11

```

<A>
  <B>
    <C>
      <D />
    </C>
  </B>
  <C>
    <D />
  </C>
</A>

```

这是因为在类型 C 的元素与它的父亲之间的关系不是固定的；它可能有一个类型 A 的父亲也可能有一个类型 B 的父亲。如果我们使用刚介绍的简单的主 / 外键技术，这个 XML 结构将只能存在或者这个父亲或者那个父亲。换句话说，类型 C 的元素将有一个到 A 表或到 B 表的外键，而不是两个都有。

所以，尽管 RDBMS 用它的表结构在相当程度上模拟了节点，但它不能很好地处理这些节点的关系。尽管这样说，我们不应该放弃将 RDBMS 作为存储 XML 节点的机制。有很多的情况用这类的数据库将表现得非常充分。实际上，正像我们在后面要看到的，书的目录的例子工作得很好，因为这个模式不包括我们才提到的那种情况。

后面，我们将深入介绍如何将一个模式转化为一系列的表和关系。

(8) 产品

尽管RDBMS的安装库是很巨大的，但是在这一领域没有什么能与 POET或eXcelon相比。下面的产品可能是最先进的。我们列出了比较感兴趣的几个应用。

LivePage Corporation LivePage Enterprise 3.0

LivePage可以转化XML到关系数据库。它通过在关系数据库上设置一个软件层来实现的，它可以处理转入和转出。LivePage本身不是一个数据库，但是搭乘在下面其中一个产品的上面：

- IBM DB2 2.1或更新版
- Microsoft SQL Server 6.0或更新版
- Oracle SQL 7或更新版
- Sybase SQL Server 10和11
- Sybase SQL Anywhere 5.0或更新版

这个产品不是作为一个XML服务器来投入市场和销售的，因为它提供了存储和发布 XML文档以外的工具。然而，提到它很重要，因为对于我们在前面所提到的使用 RDBMS中的局限，可以被克服，从而产生一个独立的强大的产品。查看 <http://www.livepage.com/>得到更详细的信息。

(9) 实用工具

下面的可能更关心于在现有的数据库上进行 XML的项目，但是一般不用在大型的项目需求中。

(10) Cerium Component Software Incorporated——XMLDB和XML servlet

XMLDB根据XML文档和生成指令来创建要求的表并且插入数据。XML servlet是一个模板驱动的，使用一个基于 XML的语言，可以处理指定的 SQL查询和HTML表格。参见 <http://ceriumworks.com/tech.html>。

(11) IBM——DataCraft

用他们自己的话说：“一个应用生成工具应该针对于在Web商务应用中的上下文RDF/XML应用。DataCraft，是一个能够生成可视化框架和对于DB2的运行查询的工具，是一个出色工具，可以在使用XML的Web数据库应用生成方面。DataCraft提供聪明的工具，可以可视化地导航资源图，并且可以可视化地从基于XML和RDF的图中建立查询。DataCraft使用RDF和XML来描述集中的数据，并且用于在服务器和客户端交换资源图和查询。”参见 <http://www.alphaworks.ibm.com/formula/datacraft>。

(12) Intelligent Systems Research——ODBC2XML

一个用于转换ODBM数据库数据到XML文档的工具。SELECT语句被嵌在一个模板中作为处理指令。参见 <http://members.xoom.com/gvaughan/odbc2xml.htm>。

(13) Mey & Westphal RIPOSTE Software——XOSL

一个Microsoft Windows DLL，用于转换一个数据库到XML文档。这是一个模板驱动的，使用嵌入特殊XOSL元素的查询。参见 <http://www.riposte.com/xosl/>。

(14) Ronald Bourret——XML-DBMS

使用Java类在关系数据库与XML文档间进行转换，使用一个映射语言来决定哪一个列是属性，哪一个是元素。参见 <http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/xmldbms/xmldbms.htm>。

(15) Stonebroom——ASP2xml

一个OLE COM组件用于在一个XML文档和任何ODBC或OLE-DB数据源间转换数据。这个产品是模型驱动的，但是输入和输出XML都必需使用ASP2XML的特殊标记。参见<http://www.stonebroom.com/>。

(16) Volker Turau——DB2XML

使用Java类从一个关系数据库转换数据到一个XML文档，可能返回一个文件、流或DOM对象。参见<http://www.informatik.fh-wiesbaden.de/~turau/DB2XML/index.html>。

10.1.4 结论

此刻，OO数据库具有表示和维护XML文档的优势。它们的分级结构可以便利地表示包含XML文档的分级结构，并且在一个文档中的很多重要的独立节点可以在数据库中通过对象来表示。

然而，关系数据库和XML的流行表明战争没有结束。关系数据库厂商将继续加入日益增加的复杂的XML的特性，同时，第三方提供的产品将在这些数据库上建立XML服务器。有资料表明，许多公司已经抓住关系系统的解决方案，通过在这些信息上放置一个XML层——就像LivePage——将变得重要和有价值。

在继续之前，我想提一下下面的产品，它们不是很容易地适应我们上面的目录例子。它们或多或少的被注上“XML服务器”的牌子。

1. Bluestone——XML Suite

这是一个基于Java的产品，它允许XML文档从服务器发到服务器，并且在每一个服务器中一个文档的处理器可以基于文档类型被调用。这是一个功能非常强大的方法，特别关系到现有的数据。这个套件的XML服务器部分使用DSIM（数据源集成模块）来允许XML文档被取出或存进其他的系统中。下面是这个产品所带的：

- ODBC
- JDBC
- XML
- LDAP
- SMTP e-mail
- FTP

下面是可以单独获得的：

- SAP R/3
- PeopleSoft
- Tuxedo
- CICS
- MQSeries

想了解详情，参见<http://www.bluestone.com/xml/>。

2. Software AG——Tamino

Tamino像XML一样存储XML文档。一个很重要的特点是它在现有数据库系统上提供一个位于上面的层，可以用来将数据库映射到 XML。这个模块叫做 X-Node。参见 <http://www.softwareag.com/>。

3. UserLand Software——Frontier

Frontier可以从它的对象数据库驱动的内容管理系统中取出 web页面，对XML的输入和输出进行特殊的处理。参见 <http://frontier.userland.com/>。

10.2 XML的交换

因为现有的数据问题是非常的重要，我们将看一下这些数据如何从 XML提取出来并且用XML进行表示。在这一部分，我们将看一下XML是如何：

- 提供一个标准的方法在不同系统之间交换信息。
- 提供一个标准的方法在不同的系统中查询数据。
- 给客户机/服务器技术一个新的生命延续。

10.2.1 数据标准

正如我们在前面所讨论过的，许多人都有从 XML的文档视图到数据结构视图转变的困难。尽管XML经常被作为一种有力的标记文档的方法，也应该记住在 XML 1.0规范讨论“XML文档”时，它的意思是与一个字处理文档或电子表格不同的东西。

字处理和电子表格文档在你的硬盘上是清晰可辨的，它们保存在一个目录下的文件里。一个XML文档也可能以这种形式存在，并且许多正在开发的应用程序也使用这种方法来保存信息。然而，这些应用程序最终有局限，通常调用一些在文本两边用一些标记来标识出的行，以便它们能够以一种有趣的方法来表示莎士比亚的著作。

然而，一个XML文档不需要存在这种固定的形式。它可以在需要的地方被创建，通过一个Web服务器。然后，也可能对同样的“文档”有两个需求，可能返回不同的结果。例如，生成一个XML文档的页面，其中包含一列 Wrox的作者，可能今天返回一个列表，明天就又增加了新的作者了。

为了举例说明它，让我们创建一个基本的 ASP脚本来输出一个 Wrox作者的列表作为简化的XML：

程序清单 10-12

```
<%@ Language=VBScript %>
<%
    Response.ContentType = "text/xml"
    Dim dbConn
    Set dbConn = CreateObject("ADODB.Connection")
    dbConn.Open "DSN=XMLPRO;"

    Dim rs
    Set rs = Server.CreateObject("ADODB.Recordset")
    rs.Open "Select * from Author", dbConn, 0, 3
%>
```



```

<?xml version="1.0" encoding="UTF-8"?>
<Authors>
<%
    Do While Not rs.EOF
%>
    <Author authorCiteID="<%=rs("authorCiteID")%>">
        <FirstName><%=rs("FirstName")%></FirstName>
        <MI><%=rs("MI")%></MI>
        <LastName><%=rs("LastName")%></LastName>
        <Biographical><%=rs("Biographical")%></Biographical>
        <Portrait picLink="<%=rs("Portrait")%>" />
    </Author><%=vbCrLf%>
<%
    rs.MoveNext
Loop
%>
</Authors>
<%
    rs.Close
    dbConn.Close
%>

```

XML的结果文档可能看上去像：

程序清单 10-13

```

<?xml version="1.0" encoding="UTF-8"?>
<Authors>
  <Author authorCiteID="1">
    <FirstName>Stephen</FirstName>
    <MI></MI>
    <LastName>Mohr</LastName>
    <Biographical>
      Stephen began programming in high school and is now a senior software
      systems architect with Omicron Consulting, he designs and develops
      systems using C++, Java, JavaScript, COM, and various internetworking
      standards and protocols.
    </Biographical>
    <Portrait picLink =
      "http://webdev.wrox.co.uk/resources/authors/mohrs.gif" />
  </Author>
  <Author authorCiteID="2">
    <FirstName>Kathie</FirstName>
    <MI></MI>
    <LastName>Kingsley-Hughes</LastName>
    <Biographical>
      Kathie is the MD of Kingsley-Hughes Development Ltd, a Training and
      Consultancy firm specialising in Web Development and visual
      programming languages, first going into CDF channels with The Dragon
      Channel.
    </Biographical>
    <Portrait picLink="" />
  </Author>
  <Author authorCiteID="3">
    <FirstName>Alex</FirstName>
    <MI></MI>
    <LastName>Homer</LastName>

```

```

<Biographical>
  Alex is a software consultant and developer whose company, Stonebroom
  Software, specialises in office integration and Internet-related
  development. He works regularly with Wrox Press on a range of
  projects.
</Biographical>
<Portrait picLink =
  "http://webdev.wrox.co.uk/resources/authors/homera.gif" />
</Author>
<Author authorCiteID="4">
  <FirstName>Frank</FirstName>
  <MI></MI>
  <LastName>Boumphrey</LastName>
  <Biographical>
    Frank Boumphrey currently works for Cormorant Consulting, a firm that
    specializes in medical and legal documentation. His main objective at
    the present is to help XML to become the language of choice in web
    documents.
  </Biographical>
  <Portrait picLink =
    "http://webdev.wrox.co.uk/resources/authors/boumphreyf.gif" />
</Author>
</Authors>

```

我们已经将相关的数据表示成一系列的节点，通过使用 XML，意味着现在可以发布这个数据到任何感兴趣的系统中，倘若系统可以首先转化 XML。例如，我们可以输出一个存在 Oracle 中的作者列表，输入到 POET 数据库中，不需要写任何的 Oracle 到 POET 的转换程序。实际上，甚至没有人要编写一个关系到对象的转换程序。我们所要做的就是确保两边都可以读或写普通的 XML 语法。应该记住，ODBC 是被设计用来在不同的数据库系统上建起一座桥梁，然而我们拥有 XML 意味着更加先进。

一个应用程序，它可以像 Microsoft 的 Access 表格或一个 Java 应用程序一样直接处理数据库中的信息——如果发生同 Internet 的通信，它可以使用如图 10-12 所示的结构。

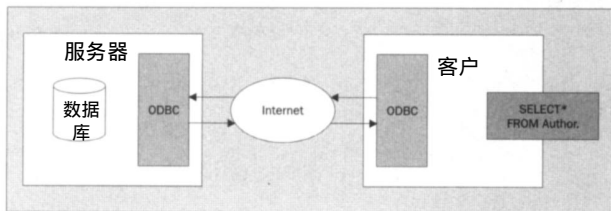


图 10-12

然而在这个方法中有几个问题：

- 仅有理解 ODBC 的系统可以接收这个信息。
- 很多防火墙不允许 ODBC 的交易。
- ODBC 易被黑客攻击。这些黑客可能发送一些需要被服务器验证的未经授权的交易。

通过在通信管道的两端加上 XML 接口，我们去除了客户端对 ODBC 的依赖（参见图 10-13）。同样，如果我们以某种方法封装 XML，并且通过 80 端口（正常的 HTTP 端口）传送它，可以

解决防火墙的阻挡问题。实际上，Microsoft有一个新的技术，SOAP（简单对象访问协议，Simple Object Access Protocol），它就是这么做的。想了解关于SOAP的更多信息，参见第11章。

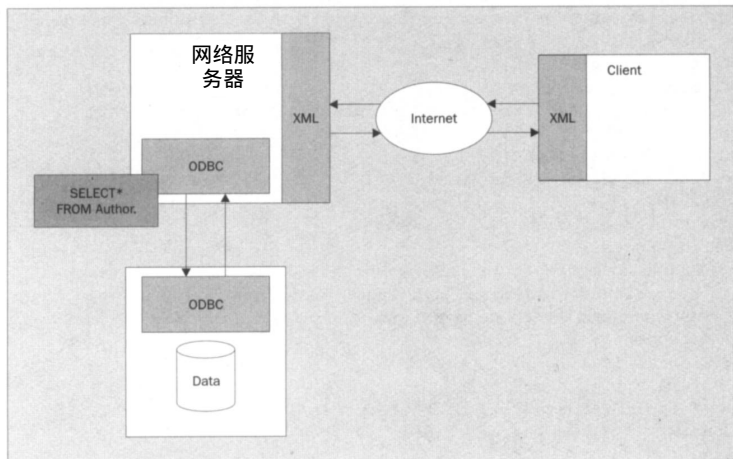


图 10-13

10.2.2 查询标准

如前所述，我们刚写的用于返回一个包括作者信息的 XML文档的ASP脚本，将返回所有在数据库中的作者。显然，这个对我们很有用，我们想通过查询这个数据的子集减少网络流量和响应时间。

为了开始，让我们只是简单地修改一下作者到 XML脚本，以便它可以在 WHERE子句中接收一个参数。大部分的代码仍然保留，所以我只显示出新的行：

程序清单 10-14

```
Set rs = Server.CreateObject("ADODB.Recordset")
Dim sQuery
Dim sWhere
sQuery = "SELECT * FROM Author"
sWhere = Request.QueryString("WHERE")
If Not IsEmpty(sWhere) Then
    sQuery = sQuery & " WHERE " & sWhere
End If
rs.Open sQuery, dbConn, 0, 3
%>
```

这个允许我们简单地通过在 URL中指定一个参数而取回不同作者的 XML表示。例如，我们要求返回的XML被放在HTML页面中的数据岛（Data Island）中，格式为：

程序清单 10-15

```
<H1>Wrox Authors</H1>
<HR>
<XML ID="authors" SRC="xmlAuthor.asp?WHERE=LastName='Mohr'"></XML>
```

```
<TABLE BORDER=1 DATASRC="#authors">
<THEAD>
<TH>authorCiteID</TH>
```

目前只有 Microsoft 实现了数据岛，但是对这个例子没有什么影响。我将介绍如何用 URL 来要求 XML 返回单个的作者。同样的脚本也可以用来取回全部或几个作者。

1. XML 查询与数据库查询

然而，现在我们遇到了一个有趣的问题，我们是以相关数据库的特殊格式来运行查询，但是得到的结果却是 XML。这个现象非常重要，我们表现的输出数据独立于后面的数据存储的格式，如果也能够使用一种独立于后端的数据库的方式查询数据是不是更好呢？例如，我们用对象数据库来替换关系数据库。刚才使用的查询方法——`LastName='Mohr'`——可能在一个对象环境下就没用了。如果我们用来取回数据的查询不需要改变就会很理想了。

如前面所述，尽管我们已经在 XML 的数据库上创建了一个层，但仍需要按照处理下面的数据库的方式进行查询。实际上，应用结构如图 10-14 所示。

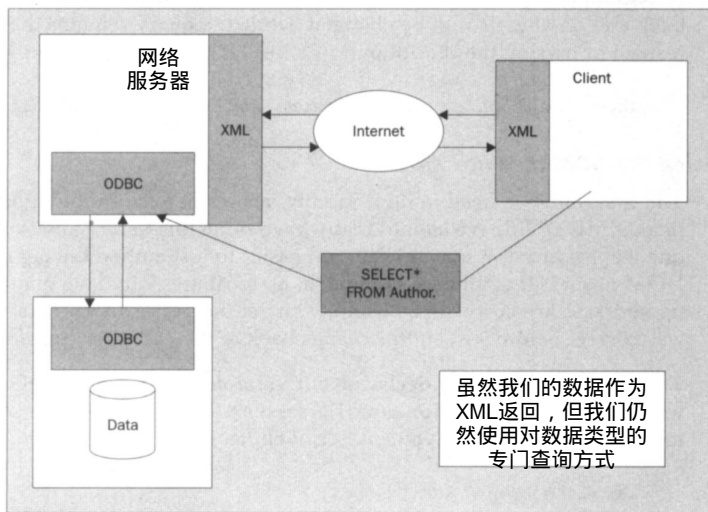


图 10-14

Microsoft 已经明确表态；一个层位于 SQL Server 7 上返回指定的 SELECT 语句查询结果为 XML。

2. XML 查询语言

我们已经建立了一个简单的方法从任意的关系数据库取回数据并且以 XML 来表现它们。换句话说，我们已经将关系数据映射到 XML 的节点结构。至于外部的世界关系到我们的数据库，可能也包含 XML，用我们现在的实现方法查询数据，外部世界将需要知道数据是以什么格式存储在这个例子中的关系数据库。相反我们需要能够查询节点。

在写这本书的时候，有许多关于分析 XML 语法查询的工作已经完成——这个我们在第 8 章已经讨论过。因为关于这方面不存在什么协议，专门为现在这个例子的目的，我们将把数据库看成一个大的 XML 文档，并且使用 XSL 语法来满足要求。

关于XSL的查询语法已经从XPath中分离出来。可以在<http://www.w3c.org/TR/xpath>上看到一些介绍。

我宁愿使用XPath而不是其他的建议来查询XML，因为它允许一个查询被表述成一个URL。这是一个有关以数据为中心的Internet的一个重要的考虑。

所以，假使我们想使用XPath来进行查询，我们应如何正确地进行呢？最好和最有效的解决方案是，实际地编写一些将XPath的查询转化为针对下面的数据库的正确形式的查询。实现的细节将依赖于不同的数据库系统，但是我们将举例说明一些后面例子中的一些观点。

现在，我们将使用一个相当简单的方法，用这个方法装入作者列表到XML DOM（文档数据模型，Document Object Model）中去，并且接着使用XSL来选择节点。然后我们会得到可以改变后端数据库的好处，同时查询保持不变。让我们修改一下脚本，替换SELECT查询中所传递的WHERE部分，我们像这样传递一个XPath语法的查询：

```
http://www.wrox.com/xmlAuthors.asp?xsl=//Author[LastName='Mohr']
```

(1) 使用DOM来写XML

我们需要做的第一件事就是修改脚本来存储从数据库取回的信息到一个XML DOM。我们应该真正地在做其他事之前完成它，但是对于某些文档，像我们所产生的那个，非常快和容易，只需要用脚本嵌入一些标记的名字。然而，使用DOM则要确保所有的标记是匹配的，所有的属性都用双引号括起来，并且所有的命名空间都是正确的。我们甚至可以用一个解析器来验证节点树，在我们将结果发回给调用者或传输结果之前，来确认结果是正确的。

在声明完变量之后，是创建一个DOM，用来保存结果（注意IE5将需要安装IIS来运行这个特殊的脚本，尽管你可以容易地使用所选的解析器来修改脚本）：

程序清单 10-16

```
<%@ Language=VBScript %>
<%
    Response.ContentType = "text/xml"
    Dim dbConn
    Set dbConn = CreateObject("ADODB.Connection")
    dbConn.Open "DSN=WroxBookCatalog;"

    Dim oParser
    Set oParser = Server.CreateObject("Microsoft.XMLDOM")
```

这个对象提供给我们处理所有DOM的特性（查看第5章，关于可用特性的全部讨论）。接着，我们在数据库上运行查询用来检索所有的作者：

程序清单 10-17

```
Dim rsAuthors
Dim sQuery
sQuery = "SELECT * FROM Author"
Set rsAuthors = Server.CreateObject("ADODB.Recordset")
rsAuthors.Open sQuery, dbConn, 0, 3
```

这段代码同样能够正确地建立起结果集。然而在能够写出标记之前，这一次将使用DOM。

第一步是创建一个叫做<Authors>的元素，它将用来保存所有的作者：

```
Dim oAuthors
Set oAuthors = oParser.createElement("Authors")
```

如你所见，创建这个元素比起在开始和结尾写出 <Authors>和</Authors>还要容易，因为DOM保证了这个层次将被正确地维护。现在准备开始循环处理数据。每一次，从数据库中得到一条记录，我们需要在DOM中创建一条新的元素，这一次叫<Author>：

```
Do While Not rsAuthors.EOF
    Dim oAuthor
    Set oAuthor = oParser.createElement("Author")
```

请注意，对于使用DOM的新手常犯的一个错误是假设刚才所创建的节点与某个东西相连；不是这样的！尽管你不得不通过解析器对象来调用元素的生成函数，节点是完全自由地浮动的。当它已经被完全配置好了在循环的末尾，我们将把它附接给<Author>节点。

处理<Author>元素的第一件事就是设置它的authorCiteID属性。注意，如果在数据库中的值为NULL，setAttribute()函数将会失败。所以，一般情况下我们将首先检查 NULL。然而，在这种情况下，我们知道因为它是表的主键，它的值将总是存在的：

```
oAuthor.setAttribute "authorCiteID", rsAuthors("authorCiteID")
```

在记录中接着的四个字段被在XML文档中的元素所表示。为了向<Author>节点加入元素容易些，我们增加了一个叫做FieldToElement()的函数，细节如下：

程序清单 10-18

```
FieldToElement oAuthor, rsAuthors, "FirstName"
FieldToElement oAuthor, rsAuthors, "MI"
FieldToElement oAuthor, rsAuthors, "LastName"
FieldToElement oAuthor, rsAuthors, "Biographical"
```

在数据库中的Portrait字段对应一个叫做picLink的属性，在一个叫<Portrait>的空元素中。不像authorCiteID属性，我们需要检查数据库的值是否是 NULL，因为如果是，它将跳出循环。如果是NULL，我们将不厌其烦地创建<Portrait>元素：

程序清单 10-19

```
If Not IsNull(rsAuthors("Portrait")) Then
    Dim oNode

    Set oNode = oParser.createElement("Portrait")
    oNode.setAttribute "picLink", rsAuthors("Portrait")
    oAuthor.appendChild(oNode)
    Set oNode = Nothing
End If
```

在完成循环进到下一条记录之前，我们将<Author>节点追加到<Authors>节点上：

程序清单 10-20

```
oAuthors.appendChild(oAuthor)
rsAuthors.MoveNext
Set oAuthor = Nothing
Loop
```

一旦完成增加新的节点到 <Authors> 节点，可以拷贝所建立的节点列表到我们在前面创建的解析器对象的 XML 文档容器中。我们需要这么做，因为在这个地方 oAuthors 对象只是一个包含一组节点的节点，不是一个完整的 XML 文档：

```
rsAuthors.Close
dbConn.Close
Set oParser.documentElement = oAuthors
```

一旦我们创建了 DOM，需要把它发送给浏览器。用必要的信息放在它的前面来显示它是一个 XML 文档，然后使用 DOM 的 xml 属性来编写它的内部结构的文本化的信息。DOM 考虑了开和关标记，将属性用引号引起来等等：

程序清单 10-21

```
%>
<?xml version="1.0" encoding="UTF-8"?>
<%
    Response.Write oParser.xml
```

最后是一个在前面我们提到过的一个函数，它是用来简化从记录集的字段中创建节点，和把它们加入到另一个节点中。同时在使用 picLink 属性时，不用担心当数据库中的值是 NULL 时会创建任何东西。如果数据集很大，这件事就很值得做，因为可能会存在许多空元素，它们占用空间。然而，应该只有在知道文档的 DTD 允许当元素为空可以不存在时我们才这样做。在这种情况下它将是：

程序清单 10-22

```
Sub FieldToElement(oTargetNode, rs, sField)
    If Not IsNull(rs(sField)) Then
        Dim oTempNode : Set oTempNode = oParser.createElement(sField)
        oTempNode.text = rs(sField)
        oTargetNode.appendChild(oTempNode)
        Set oTempNode = Nothing
    End If
End Sub
```

(2) 增加查询

我们刚写的代码只是简单地以 XML 的形式输出所有的作者，但是使用 DOM 要好于写自己的标记。换句话说，我们不应该增加额外的功能到已经有的东西上。然而，既然数据在 DOM 中，可以用它来做任何我们想处理 XML 的事情。特别是，可以增加自己的 XSL 查询语句。

附加的代码被突出显示，而且它位于将数据库中的数据转换为 DOM 的代码之后。只有那时我们才可以使用 XSL 语句：

程序清单 10-23

```
<?xml version="1.0" encoding="UTF-8"?>
<%
    Dim sXSL
    sXSL = Request.QueryString("xsl")
    If IsEmpty(sXSL) Then
```

```

    Response.Write oParser.xml
Else
    Set oNodeList = oParser.documentElement.selectNodes(sXSL)

```

如果没有查询被传给调用者，则返回整个结果集。然而，如果存在一个查询，我们使用 selectNodes 语句来进行过滤。selectNodes 语句可以用在任意节点上，所以我们可以这么写：

```
Set oNodeList = oAuthors.selectNodes(sXSL)
```

然而，因为所建立的 <Authors> 节点不是一个正确的 XML 文档，它没有根节点，并且查询开始用一个 /——意味着从根开始工作——结束时不会返回任何东西。所以查询事先创建的 DOM 对象的文档元素，我们将 <Authors> 节点保存在里面。

(3) 输出结果

用 XSL 选择的节点的结果不必是一个有效的 XML 文档。例如，XSL 查询：

```
/Authors/Author/LastName
```

将返回下面的节点：

```

<LastName>Mohr</LastName>
<LastName>Kingsley-Hughes</LastName>
<LastName>Boumphrey</LastName>

```

作为一个 XML 文档它是无效的，因为没有根节点——或一个以上的根节点，看你是怎么看了！

然而，我们可以像以前一样简单地用一个 <Authors> 节点包装每件东西，因为结果不一定是作者。以前的查询用 <Authors> 包装后看上去将是：

程序清单 10-24

```

<Authors>
  <LastName>Mohr</LastName>
  <LastName>Kingsley-Hughes</LastName>
  <LastName>Boumphrey</LastName>
</Authors>

```

它将打乱我们的 DTD 或模式，因为 <LastName> 只能作为 <Author> 的子节点而不是 <Authors> 的。当然可以说结果只能返回作者，并且只能将结果放在一个 <Authors> 元素里，但是这将极大地降低了脚本的灵活性。例如，如果有人想列出在数据库中所有图片的引用列表，那样他们就可以创建一个肖像长廊，他们可能使用如下的查询：

```
/Authors/Author/Portrait[@picLink]
```

这个查询要求我们的服务器查找所有的 <Portrait> 元素，元素要有 picLink 的属性，它被赋了一些值，那就是，它是非空的。另外，每个返回的 <Portrait> 元素必须是 <Author> 元素的子元素，<Author> 元素必须是 <Authors> 元素的子元素。那个查询可能返回下面的结果：

程序清单 10-25

```

<Portrait picLink =
  "http://webdev.wrox.co.uk/resources/authors/mohrs.gif"/>
<Portrait picLink =
  "http://webdev.wrox.co.uk/resources/authors/boumphreyf.gif"/>

```

如果不将它包装在 `<Authors>` 里，并且不能仅仅创建像希望中的新的如 `<LastName>` 和 `<Portraits>` 这样的包含元素（因为它们也会校验失败），然后可能会考虑用一般的元素来进行包装。例如，可以设计自己的容器，并且也许在它的里面包括对到达的信息应如何返回的信息：

程序清单 10-26

```
<wr:ResultsWrapper wr:query="/Authors/Author/LastName"
  xmlns:wr="wroxresultsnamespace">
  <LastName>Mohr</LastName>
  <LastName>Kingsley-Hughes</LastName>
  <LastName>Boumphrey</LastName>
</wr:ResultsWrapper>
```

甚至可以更进一步，加入时间戳和其他的信息，用来辅助处理查询结果。在下一部分，我们将简要地看一下关于一些包装数据的方法。现在只能返回所要求的数据——例如，可能有多于一个的元素在文档的根。让我们完成脚本，把 XSL 查询结果写出来：

程序清单 10-27

```
Else
  Set oNodeList = oParser.documentElement.selectNodes(sXSL)
  Dim ix
  For ix = 0 To oNodeList.length - 1
    Response.Write oNodeList.item(ix).xml
  Next
End if

Sub FieldToElement(oTargetNode, rs, sField)
```

注意这次使用了循环。因为可能有一些顶级的节点，需要在结果节点列表中循环，为每一个节点输出 XML，要好于简单地把整棵树输出成为 XML。

3. 优化

在前面所勾勒出的技术相当低效。在发觉只想要其中的一个之前，它要求我们读出所有的作者。在我们这个简单的例子中，它不是一个很大的处理，但是在一个有着成千上万记录的数据库中可能会很慢。

为了提高效率，应该利用其下的数据库的查询能力，并且使用 XSLT 和 XPath 的能力来合并它们。假设想主要考查关系数据库，我们将快速地看一下 XPath 语句是如何被映射成 SQL 语句。例如：

Author[@MI]

要求所有叫 MI 的作者。这个可以直接被映射为：

```
SELECT * FROM Author WHERE MI <> "";
```

选择一个单个的作者应该是：

Author[LastName = 'Mohr']

将被映射成：

```
SELECT * FROM Author WHERE LastName = 'Mohr';
```

XPath 也允许根据树中其他部分的节点的值来选择节点。例如，使用第 7 章的书目录模式，

如果你想选择所有的书，这些书与一个特别的出版商相关联，则需要下面的 XPath 查询：

```
/Catalog/Book[imprint/@ID=/Catalog/Publisher[CorporateName="Wrox Press Ltd."]/Imprints/Imprint/@shortImprintName]
```

这个意思是说，我们需要所有的 <Book> 元素，把它的 imprint/@ID 设成与 <Publisher> 的 <CorporateName> 为 “Wrox Press Ltd.” 下的 <Imprints> 元素下的 <Imprint> 元素的 shortImprintName 属性一样的值。

在 SQL 中模拟分级的语句是很直接的。我们可以使用简单地连接语句来得到父 / 子关系，所以 /Catalog/Book 将变成：

```
SELECT *  
FROM Catalog, Book  
WHERE pk_Catalog = fk_Catalog
```

请求的指定出版商的查询部分为：

```
/Catalog/Publisher[CorporateName="Wrox Press Ltd."]
```

也不是很困难：

程序清单 10-28

```
SELECT *  
FROM Catalog, Publisher  
WHERE pk_Catalog = fk_Catalog  
AND CorporateName = "Wrox Press Ltd."
```

（假设我们知道文本元素 <CorporateName> 已经被存在一列而不是表中了。我们可以在后面看到。）

在 Catalog/Book 上的过滤很困难。为了做到这一点，我们需要介绍关于 SELECT 语句的另一个方面——子查询。子查询允许查询结果被用作值的列表用于检查另一个查询。在我们的例子中，想创建一个所有 shortImprintName 值的列表，并且查看哪一个与 imprint 元素的 ID 这个值相匹配，在使用那个列表查找引用 imprint 的书之前。我们可以如下实现：

程序清单 10-29

```
SELECT *  
FROM Book  
WHERE imprint IN (  
    SELECT *  
    FROM Imprint  
    WHERE ID IN (  
        SELECT shortImprintName  
        FROM Imprint  
    )  
)
```

请注意，我已经方便地优化了在以前建立的用来只选择一个出版商的其他过滤器。然而，我确信你可以看出，用 SQL 查询来模拟 XPath 查询是可能的。但是存在两个问题。第一个就是 XPath 仍然不是一个标准，所以现在你所写的可能会改变。第二个就是你需要分析 XPath 语句，并且能够理解整个语法。这些问题如何被完全解决还不是很清楚，但是我们可以看到 XPath 解析

器的出现是位于特别数据库层的顶端，就像 ODBC 站在不同低层数据存储格式的顶端一样。

10.2.3 结论

XML 提供了一个相当强大的在不同数据间交换数据的方法。这个对于数据库特别有用，因为在不同类型数据间发生的数据相互交换是相当困难的。

尽管数据可以以一种普通的格式被表示，并没有什么用于取回的可以接受的标准。尽管不像我们希望地那样灵活，用 XPath 指定查询说明了使用标准查询语法的潜力。

10.2.4 你可能需要的标准

我们已经讨论了用于选择 XML 数据的不同可能的查询语法。对于查询标准的问题，我们也讨论了如何返回数据结果的论点。提出几个标准指示应该如何去做。

1. 片段数据交换

最简单和可能最有效的是片段数据交换建议，在 <http://www.w3.org/TR/WD-xml-fragment>。这个提议提供了一种机制，通过对它的松散节点给出一些上下文。在第 8 章我们已经看到了这一点，但是现在拿来在这里使用，并且看一个例子：取出一个作者的传记。我们在前面所写的代码允许使用下面的语法：

```
http://www.wrox.com/xmlAuthors.asp?xsl=//Author[LastName='Mohr']/Biographical  
来取回：
```

程序清单 10-30

```
<Biographical>  
  Stephen began programming in high school and is now a senior software  
  systems architect with Omicron Consulting, he designs and develops systems  
  using C++, Java, JavaScript, COM, and various internetworking standards and  
  protocols.  
</Biographical>
```

片段数据交换的提议建议，应创建一个提供上下文的分离的文档：

程序清单 10-31

```
<f:fcs xmlns:f="http://www.w3.org/XML/Fragment/1.0"  
  xmlns="http://www.wrox.com/authors.xdr">  
  <Authors>  
    <Author authorCiteID="1">  
      <FirstName />  
      <MI />  
      <LastName />  
      <f:fragbody fragbodyref="xmlAuthors.asp?xsl=//Author  
        [LastName='Mohr']/Biographical"/>  
      <Portrait />  
    </Author>  
    <Author authorCiteID="2" />  
    <Author authorCiteID="3" />  
  </Authors>  
</f:fcs>
```


注意，这个片段数据交换文档显示在同一层次的元素为需要上下文的元素，但是它不会为这些元素的内容来浪费空间。所以你可以看到，所有在<Author>元素中的元素都被命名了，但是没有内容。同样，所有其他在数据库中的表示其他作者的<Author>元素也没有内容，尽管属性仍然被传递给了每个元素，而不管它在什么位置上。查询结果数据通过在URI的<fragbody>处被引用。为了让这个机制工作，你需要创建一个脚本，用来输出象上面格式的XML，并且按以下方式进行引用：

```
http://www.wrox.com/fcsAuthors.asp?xsl=//Author[LastName='Mohr']/Biographical
```

这个片段参考了前面的<xmlAuthors>脚本。

前面关于片段数据交换的提议的草稿允许片段和它的内容包装到一个XML文档中。尽管可以实现，但是这个提议不坚持把它标准化。相反它宣布一个包装工作组将解决这个问题。

2. SOAP和XML-RPC

我们可以返回结果的另外一种方法是通过SOAP或XML-RPC，它们将在第11章被深入讨论。这两个协议都提供了包装对一个服务器调用结果的方法，尽管它们不包含上下文信息。增加上下文信息到返回文档是一件简单的事情。

10.3 图书目录的例子

理论已经足够了。我们已经了解数据库可以保存XML并且可以通过XML在不同的数据库之间交换数据。是到了作为一个数据库实现图书目录的时候了。

在下面的例子中，假设Wrox Press已经有了一个关于作者信息的关系数据库，但是我们需要创建新的数据库用于出版商和图书信息。在后面，将使用模式来定义一个关系数据库表结构。

10.3.1 处理存在的数据和应用

假设我们拥有一个数据库，其中包含作者的名字、地址和他们的一些记录。出于这个演示的目的，我已经简单地创建了一个图书数据库，使用了Microsoft Access创建数据库向导。这个例子的数据入口表单有两页，如图10-15所示。

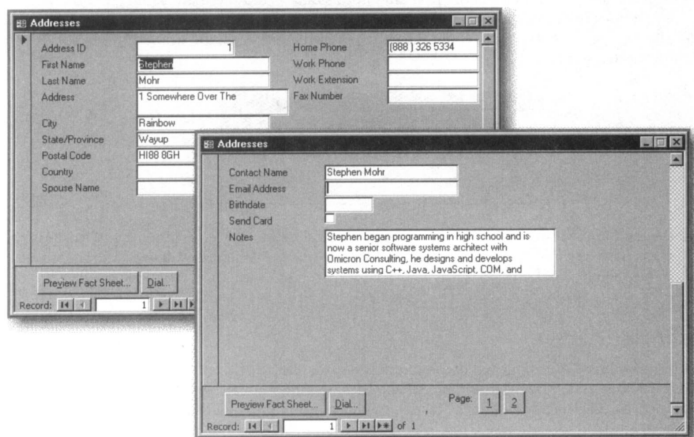


图 10-15

关于这个例子有几点要弄明白。第一个就是数据已经存在了。在计划一个可能基于 XML 的项目时把它提到台面是很重要的。试图让公司的每一个人把在他们存储介质上的现有的数据库应用程序移到 XML 上应非常慎重。同样因为这些应用程序，所以使用 XML 处理可能不是最好的。在这种情况下，我们需要简单的数据录入表格，用它可以很容易地增加新记录和修改记录。用户应该被要求这样编辑吗？

程序清单 10-32

```
<authors>
  <Author authorCiteID="1">
    <FirstName>Stephen</FirstName>
    <LastName>Mohr</LastName>
    <Address>1 Somewhere Over The</Address>
    <City>Rainbow</City>
    <State>WayUp</State>
    <PostalCode>HI88 8GH</PostalCode>
    <HomePhone>(888) 326 5334</HomePhone>
    <Notes>
      Stephen began programming in high school and is now a senior software
      systems architect with Omicron Consulting, he designs and develops
      systems using C++, Java, JavaScript, COM, and various internetworking
      standards and protocols.
    </Notes>
  </Author>
  <Author authorCiteID="2">
    <FirstName>Kathie</FirstName>
    <LastName>Kingsley-Hughes</LastName>
    ...
    <Notes>Kathie is the MD of Kingsley-Hughes Development Ltd, a...</Notes>
  </Author>
</authors>
```

即使提供给用户 XML 的编辑器来进行数据维护——在市场上有一些很不错的产品——它仍然在某些地方不同具备简单的 Microsoft Access 表单所具有的功能。例如，当光标停留在家庭电话号码域上时，点击拨号按钮，将产生如图 10-16 所示的对话框：

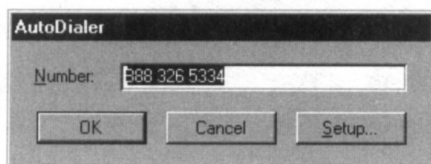


图 10-16

转换数据到 XML 意味着弱化我们的应用，而不是增强它，简单地说，我们已经有了一个相当适合的应用和数据存储介质。让我们离开那，但是看一下是否我们可以按 XML 抽取这个数据。

1. XML 记录集

我们需要确定的第一件事是当我们以 XML 输出现有的数据时，应该采用什么样的格式。在前面我们看到，很多数据库的产品已经被扩展了，允许数据作为一系列相关的记录输出。数据仍然是一个 XML 文档，但是所使用的 XML 的语法反映了数据存储的关系数据库中。这里有一个

例子，显示了我们的作者地址信息是如何以这种结构类型被输出的：

程序清单 10-33

```
<RecordSet name="authors">
  <Record name="Author">
    <Column name="authorCiteID" value="1" />
    <Column name="FirstName" value="Stephen" />
    <Column name="LastName" value="Mohr" />
    <Column name="Address" value="1 Somewhere Over The" />
    <Column name="City" value="Rainbow" />
    <Column name="State" value="WayUp" />
    <Column name="PostalCode" value="HI88 8GH" />
    <Column name="HomePhone" value="(888) 326 5334" />
    <Column name="Notes" value="Stephen began..." />
  </Record>
  <Record name="Author">
    <Column name="authorCiteID" value="2" />
    <Column name="FirstName" value="Kathie" />
    <Column name="LastName" value="Kingsley-Hughes" />
    ...
    <Column name="Notes" value="Kathie is..." />
  </Record>
</RecordSet>
```

为了与使用结构输出XML相区分，我们把从数据库中输出的XML叫做XML记录集。它对于我们有用——主要是因为它用的是XML。一旦我们用XML来记录数据，显然可以使用所有以前在这本书中的操作XML部分中学到的工具；可以使用XSLT来将信息转换成为一系列的使用HTML的表，或者可以改变列的名字，这样数据就可以插入到别的数据库中了。

但是让我们诚实一点，使用这个方法有一个主要的缺点。最明显的就是，唯一能够执行的验证是保证一个<RecordSet>元素只包含Record元素，并且从顺序上它们只包含<Column>元素。如果我们希望转换这个信息到另一个服务器上去，数据将被存放在那里，服务器没有一个简单的方法来知道被突出显示的列不应该显示在这里：

程序清单 10-34

```
<Record name="Author">
  <Column name="authorCiteID" value="1" />
  <Column name="FirstName" value="Stephen" />
  <Column name="LastName" value="Mohr" />
  <Column name="Address" value="1 Somewhere Over The" />
  <Column name="City" value="Rainbow" />
  <Column name="State" value="WayUp" />
  <Column name="PostalCode" value="HI88 8GH" />
  <Column name="HomePhone" value="(888) 326 5334" />
  <Column name="InvoiceNumber" value="1223" />
  <Column name="Notes" value="Stephen began..." />
</Record>
```

当然，我们能够包括带有数据的信息，这些数据是用来说明什么列是所要求的。Microsoft的活动数据对象（ADO，Active Data Objects）采用这种方法。下面显示了存在我们的Access数据库的数据将如何被输出，如果ADO的新特性被用于持续一个记录集：

Microsoft的ADO现在版本为2.5。更多的信息可以看<http://www.microsoft.com/data/ado>。

程序清单 10-35

```
<xml xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882'
      xmlns:dt='uuid:C2F41010-65B3-11d1-A29F-00AA00C14882'
      xmlns:rs='urn:schemas-microsoft-com:rowset'
      xmlns:z='#RowsetSchema'>
<s:Schema id='RowsetSchema'>
  <s:ElementType name='row' content='eltOnly'>
    <s:attribute type='authorCiteID' />
    <s:attribute type='FirstName' />
    <s:attribute type='LastName' />
    <s:attribute type='Address' />
    <s:attribute type='City' />
    <s:attribute type='State' />
    <s:attribute type='PostalCode' />
    <s:attribute type='HomePhone' />
    <s:attribute type='Notes' />
    <s:extends type='rs:rowbase' />
  </s:ElementType>
  <s:AttributeType name='authorCiteID' rs:number='1' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='FirstName' rs:number='2' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='LastName' rs:number='3' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='Address' rs:number='4' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='City' rs:number='5' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='State' rs:number='6' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='PostalCode' rs:number='7' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='HomePhone' rs:number='8' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
  <s:AttributeType name='Notes' rs:number='9' rs:nullable='true'
    rs:write='true'>
    <s:datatype dt:type='string' dt:maxLength='255' />
  </s:AttributeType>
```

```

</s:Schema>
<rs:data>
  <z:row authorCiteID="1" FirstName="Stephen" LastName="Mohr"
    Address="1 Somewhere Over The" City="Rainbow" State="WayUp"
    PostalCode="HI88 8GH" HomePhone="(888) 326 5334"
    Notes="Stephen began..." />
</rs:data>
</xml>

```

但是即使使用了这个额外信息，仍然有一个普通的数据版本在数据库中；我们已经按记录集表示数据，要好于使用一些结构。那就是说，在数据需要被转换成其他的关系数据库的情况下，这个技术将可能被大量地应用。因为所有的 RDBMS 可以按照表、行、列来转换，所有的都可能识别该格式。使用模式元素提供的附加信息来表示列的定义，这种方法是一个相当有力的技术。下面的代码使用 ADO 来生成输出结果也是简单的（下面用的是 Visual Basic）：

程序清单 10-36

```

Set con = New ADODB.Connection
con.ConnectionString = "DSN=XMLPRO;"
con.Open
Set rs = con.Execute("Select * FROM Addresses")

rs.save "AuthorsRS", adPersistXML

```

这种简单性所带来的好处可能就是你的应用所需要的。

2. XML 数据：使用转换

然而，假设我们需要在单调的信息上增加一些结构，最简单的方式是将结构放在将要转换输出的数据上。下面是一个 XSLT 样式表的例子，这个例子可以将这种单调的图书格式转变为图书目录例子使用的格式（如果我们想将一个单调结构融合到结构化的目录中）：

程序清单 10-37

```

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:z='#RowsetSchema'>
  <xsl:template match="z:row">
    <xsl:element name="Author">
      <xsl:attribute name="authorCiteID">
        <xsl:value-of select="@authorCiteID" />
      </xsl:attribute>
      <xsl:element name="FirstName">
        <xsl:value-of select="@FirstName" />
      </xsl:element>
      <xsl:element name="LastName">
        <xsl:value-of select="@LastName" />
      </xsl:element>
      <xsl:element name="Biographical">
        <xsl:value-of select="@Notes" />
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>

```

这个样式表简单地从<z:row>中取出每个属性，并且把它们输出成元素，尽管使用了不同的名字。转换的结果为下面的简单的XML文件：

程序清单 10-38

```
<Author authorCiteID="1">
  <FirstName>Stephen</FirstName>
  <LastName>Mohr</LastName>
  <Biographical>Stephen began...</Biographical>
</Author>
```

在小型系统中可能很少会在这种技术或一种不需要中介质的XML记录集的数据而直接编写为不同目标的不同脚本的技术中间进行选择。然而在一个大一些的系统，它可能需要为许多的用户提供很多种格式的信息，这种技术可能更有用。例如，如果我们想输出信息，以这样一种方式，它能够被一个完全不同的系统所理解，或者我们以单调格式输让目录系统来处理它，或者我们可能生成另一种样式表用它再次地转换单调格式。

因为这些映射类型不总是一对一的，甚至可能设计出一种文档结构，用来说明如果将记录和字段映射到要求的模式格式。像上面的样式表可能被自动生成，甚至可能提供信息让低层的数据库对其进行修改。

3. XML数据：使用脚本

把信息看成一种结构化的XML的方法是生成一个作者的列表，作为在Web服务器上的XML页面。在前面我们已经看到是如何做的，所以在这里我只显示主要的循环部分，并做了一点修改，允许现有的作者数据库能够使用：

程序清单 10-39

```
<Authors>
<%
  Do While Not rs.EOF
%>
  <Author authorCiteID="<%=rs("authorCiteID")%>">
    <FirstName><%=rs("FirstName")%></FirstName>
    <MI><%=rs("MI")%></MI>
    <LastName><%=rs("LastName")%></LastName>
    <Biographical><%=rs("Notes")%></Biographical>
    <Portrait picLink="<%=rs("Portrait")%>" />
  </Author>
<%
  rs.MoveNext
Loop
%>
</Authors>
```

4. 结论

从现有系统提取数据带给我们很多问题。尽管最完美的解决方案是以一种普通的格式取回数据——关系结构的XML表示——然后转换成希望的格式，这可能很充分地表现出性能问题。如果对于数据所要完成的目标不多，那么特定的脚本可能就是答案。如果对于数据有很多的目的格式，那么将数据的中间XML表示进行缓冲可能是最好的，然后可以按照要求进行转换。

10.3.2 图书目录模式

现在已经确定如何使用已经存在的作者数据，但是仍然需要在一个数据库中表示图书目录的其他部分。在这一部分，我们将研究图书目录模式定义的一些关键部分，并且随着发展，我们将勾勒出一些创建关系数据库的关键点。在数据库中，将保存数据并且与模式相一致。在做这个的过程中，我们将建立一些规则，可以在后面使用它们来使从一个模式创建一个关系数据库的处理自动化。最后，我们将讨论用来进行自动转化代码的主要特点。代码将用来执行自动转换，关于源代码的全部有效的清单，从 <http://www.wrox.com> 下载。

1. 定义目录数据库

在这个讨论中，我们将经常转换每一个模式定义为一个节点结构。我将通过下面的简单转换来表示节点结构（参见图 10-17）。

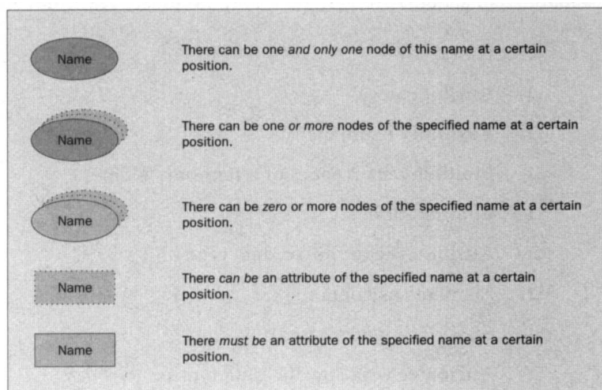


图 10-17

结构通过节点的位置被指出。例如，在图 10-18中指出一个 A 类型的节点有一个请求的属性叫做 B，一叫 C 类型的节点和零个或多个 D 类型的节点：

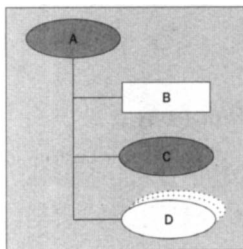


图 10-18

满足这个节点结构的 XML 文档的例子包含如下结构：

程序清单 10-40

```
<A B="xyz">
  <C />
</A>
```

```
<A B="xyz">
  <C />
  <D />
</A>
```

```
<A B="xyz">
  <C />
  <D />
  <D />
  <D />
  <D />
  <D />
  <D />
</A>
```

让我们开始转换模式定义为关系数据库，通过例举出一般的模式的一些关键特性——明显的XML文档——它们是：

- 分层。
- 属性。
- 纯文本元素。
- 纯文本元素的多发性。
- 枚举。
- 拥有id数据类型的属性。
- 作为容器的元素。
- minOccurs属性。
- 拥有指定数据类型的属性。
- 拥有idref数据类型的属性。
- 拥有idrefs数据类型的属性。

(1) 分层

第7章的图书目录模式开始部分如下所示：

程序清单 10-41

```
<?xml version="1.0"?>
<Schema name="PubCatalog.xdr"
  xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes"
  xmlns:athr="x-schema:authors.xml">

  <ElementType name="Catalog" content="eltOnly" order="seq">
    <element type="Publisher" minOccurs="1" maxOccurs="*" />
    <element type="Thread" minOccurs="0" maxOccurs="*" />
    <element type="Book" minOccurs="1" maxOccurs="*" />
  </ElementType>
```

其中描述了一个<Catalog>元素，它只能包含其他元素。这就意味着不允许有其他的文本，它包括在<Catalog>元素中但是没有包括在<Catalog>所包括的元素中（你可能还记得第7章的这个意思，<Catalog>不能包含混合的内容）。这个特性极大地简化了数据库的实现。

<Catalog>元素可以包含<Publisher>，<Thread>和<Book>这类的元素。所以这些元素可以出

现任意多次，但是至少有一个 <Publisher>，并且至少一个 <Book>。由这种模式定义所表示的XML文档的例子包括：

程序清单 10-42

```
<Catalog>
  <Publisher />
  <Book />
</Catalog>

<Catalog>
  <Publisher />
  <Publisher />
  <Thread />
  <Thread />
  <Book />
  <Book />
</Catalog>

<Catalog>
  <Publisher />
  <Thread />
  <Thread />
  <Thread />
  <Book />
  <Book />
  <Book />
  <Book />
  <Book />
</Catalog>
```

节点结构如图10-19所示。

你可能会回忆起对关系数据库的特性的讨论，这个模式可以容易地用四个表来表示。你可能会问“为什么不是树？”，因为这里只有一个 <Catalog>，并且我们可以隐藏其他的表与 <Catalog> 之间的关系。然而，因为我们想产生一个可以被容易生成的解决方案——也可能该数据库在将来会扩展——我们将创建一个Catalog表。

这些表之间的层次关系可以通过在低层次的表与包含它的表之间创建连接来建立。例如，为了产生一条 Publisher表中的Publisher记录与一条Catalog记录（在这种情况下只有一条）的连接，我们将需要如图10-20所示的结构。

为了找到 <Catalog> 的所有 <Book> 子节点，我们将简单地查询Book表得到所有fk_Catalog列被设置成了与我们正在搜索的 <Catalog> 值一样的值。

现在我们能够定义最初的两个规则（规则1仅用来节省在后面的规则中重复自身使用）：

- 规则1——无论何时我们创建一个新表，创建一个与表名相同的主键，但是使用一个 pk_ 的前缀。这一列将是一个 automatically-incremented（自动增加）的整数。
- 规则2——对每一个元素节点类型，创建一个与元素相同名字的表，然后：

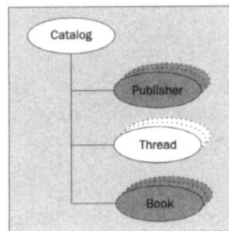


图 10-19

- 如果元素节点是一个子节点，创建一个与父元素节点同名的列，但是使用fk_的前缀。
- 创建一个在刚创建的列和与其父元素节点同名的表中的列的外键关系，列的名字是父元素的名字加一个前缀pk_。

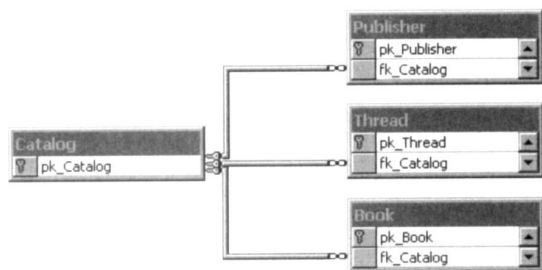


图 10-20

(2) 属性

我们需要的三个表中的第一个是 Publisher表。模式定义告诉我们需要表的的信息：

程序清单 10-43

```
<AttributeType name="isbn" required="yes"/>
<ElementType name="Publisher" content="eltOnly" order="seq">
  <description>Publisher section</description>
  <attribute type="isbn"/>
  <element type="CorporateName"/>
  <element type="Address" minOccurs="1" maxOccurs="*" />
  <element type="Imprints"/>
  <element type="athr:Author" minOccurs="0" maxOccurs="*" />
</ElementType>
```

就像<Catalog>元素，一个<Publisher>元素只能包含其他元素——它能包含非混合的内容。元素是<CorporateName>，<Address>，<Imprints>和<Author>，如图10-21所示。

应用分层规则我们建立了四张表，并且建立了它们之间的关系。然而，<Publisher>也有一个叫做isbn的属性节点。它可以非常容易被表示为Publisher表的一个字段，并且我们将给它同节点一样的名字。注意，节点结构表明需要一个属性节点，所以应该通过在表中增加要求字段而实现它。因为不允许NULL值，所以如图10-22所示定义这个列。

现在可以定义另一个规则：

- 规则3——对于每个属性节点。
- 创建一个与属性节点同名的列。
- 如果要求属性，那么列不允许为NULL值。

(3) 纯文本元素

<Publisher>元素可以包含的下一个元素是<CorporateName>，定义如下：

```
<ElementType name="CorporateName" content="textOnly"/>
```

像第一个规则要求的那样，我们将创建一个名为 CorporateName的表。在根据规则设置好键值之后，一个在Publisher表和CorporateName表之间的一对一的关系就出现了，如图10-23所示。

以一个数据库的观点，用于 corporateName的额外的表是不必要的。这就是说，数据被放入到另外一个表，正像 Publisher表的一列。而从 XML角度，额外的表也是不必要的。

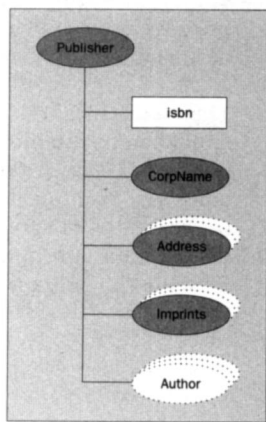


图 10-21

<CorporateName>元素的内容仅是文本——它不能包含任何子节点——所以可以被容易地存储为 Publisher表中的一列。因此，Publisher表更有效率的形式如图 10-24 所示。

Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value
isbn	varchar	255	0	0	<input type="checkbox"/>	

图 10-22

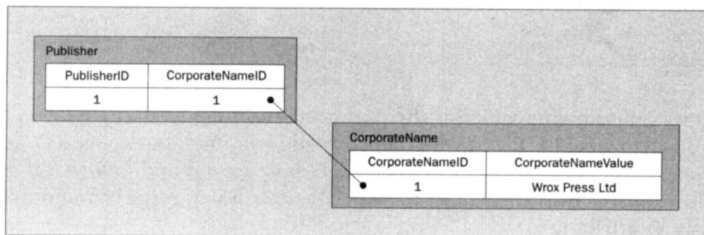


图 10-23

Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value	Identity	Identity Seed	Identity Increment
pk_Publisher	int	4	10	0	<input type="checkbox"/>		<input checked="" type="checkbox"/>	1	1
fk_Catalog	int	4	10	0	<input type="checkbox"/>		<input type="checkbox"/>		
isbn	varchar	255	0	0	<input type="checkbox"/>		<input type="checkbox"/>		
CorporateName	varchar	255	0	0	<input checked="" type="checkbox"/>		<input type="checkbox"/>		

图 10-24

比起每一个元素都有一个表，它提高了性能，因为它不需要创建许多的连接来取得数据。为了取得<Publisher>信息，只需要下面的查询：

```
SELECT isbn, CorporateName FROM Publisher;
```

然而使用一个独立的公司名称表，需要这样做：

```
SELECT p.isbn AS isbn, cn.CorporateName AS CorporateName
FROM Publisher AS p, CorporateName AS cn
WHERE p.CorporateNameID = cn.CorporateNameID;
```

尽管看上去好像没有什么多余的工作，只用了一个查询来检索出版商地址，但当我们看到后面的地址模式时，将会发现这是多么的复杂——并且可能很慢。

在实现一个用于 XML 数据的关系存储时，应决定哪一个字段应该是属性，哪一个应该是纯文本元素，这是件很容易的事。建好以后，刚刚在 Publisher 表中描述的数据可以被输出为元素：

程序清单 10-44

```
<Publisher>
  <isbn>1-86100</isbn>
  <CorporateName>Wrox Press Ltd.</CorporateName>
</Publisher>
```

或者输出为元素的属性：

```
<Publisher isbn="1-86100"
  CorporateName="Wrox Press Ltd."/>
```

我们没有一个简单的方法知道哪一个更好的输出，那么问题是用什么方法来表示数据库

是最佳的，但我们应能容易重新生成 XML。一种可以考虑的技术是在输出之前，把模式当作数据。另一种方法是在取得数据后，对它进行转换，然后每一个字段作为属性重新返回。

这两种方法可能适用于不同的情况，被用在某些在前面关系数据库部分所描述的工具中。在这里我们使用了一个简单方法，它是在数据库中将列命名，通过这种方法就很容易重新取回丢失的信息。如果属性用 attr_ 作为前缀，纯文本元素使用 elem_，那么应该能够很容易区分它们的不同。我们需要修改规则 3，以包括前缀 attr_：

- 规则3——对于每个属性节点。
- 创建一个与属性节点同名的列，使用前缀 attr_。
- 如果要求属性，那么列不应该允许为 NULL 值。

我们也可以包含下一个规则：

- 规则4——如果一个元素的节点只包含文本，那么在它的父元素表中创建一列，列的名字与节点一样，使用前缀 elem_。

(4) 纯文本元素的多发性

尽管把一个纯文本元素作为一个字段是一个有用的优化，也存在不能用的情况。当元素可以显示一次或多次的时候就是这种情况。如果我们继续处理模式，可以看到 <Street> 元素就只包含文本，但是它可以在地址的街道部分表示为多行：

程序清单 10-45

```
<ElementType name="Address" content="eltOnly" order="seq">
  <attribute type="headquarters"/>
  <element type="Street" minOccurs="1" maxOccurs="*" />
  <element type="City"/>
  <element type="PoliticalDivision"/>
  <element type="Country"/>
  <element type="PostalCode"/>
</ElementType>

<AttributeType name="headquarters"
  dt:type="enumeration"
  dt:values="yes no" />
<ElementType name="Street" content="textOnly"/>
```

尽管 <Street> 没有子元素，但是由于它具有多个值，所以不能在关系数据库用一列来实现。所以我们需要增加一个表来保存 <Street> 的多个值。首先让我们改一下规则 4：

- 规则4——如果一个元素的节点只包含文本，并且最大发生次数为 1 次，那么在它的父元素表中创建一列，列的名字与节点一样，使用前缀 elem_。

这就意味着我们已经拥有了所有要完成 Address 表的规则——没有 Street 列——看上去将如图 10-25 所示。

然而，在创建 Street 表之前还有一个问题。到现在为止，我们创建的用来保存元素信息的所有表都简单地与它的父元素相关联，并可能拥有属性。然而 <Street> 元素没有属性但是包含自己的数据。所以需要在在一个表中用一列来保存它。但是因为这个数据不是一个属性或是一个纯文本的子元素，所以我们无法命名。在一个关系数据库中，每一列必须有一个名字，所以把它叫

做pcdata。

Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value	Identity	Identity Seed	Identity Increment
pk_Address	int	4	10	0	<input type="checkbox"/>		<input checked="" type="checkbox"/>	1	1
fk_Publisher	int	4	10	0	<input type="checkbox"/>		<input type="checkbox"/>		
attr_headquarters	varchar	255	0	0	<input checked="" type="checkbox"/>		<input type="checkbox"/>		
elem_City	varchar	255	0	0	<input checked="" type="checkbox"/>		<input type="checkbox"/>		
elem_PoliticalDivision	varchar	255	0	0	<input checked="" type="checkbox"/>		<input type="checkbox"/>		
elem_Country	varchar	255	0	0	<input checked="" type="checkbox"/>		<input type="checkbox"/>		
elem_PostalCode	varchar	255	0	0	<input checked="" type="checkbox"/>		<input type="checkbox"/>		

图 10-25

现在Street表将如图10-26所示。

Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value	Identity	Identity Seed	Identity Increment
pk_Street	int	4	10	0	<input type="checkbox"/>		<input checked="" type="checkbox"/>	1	1
fk_Address	int	4	10	0	<input type="checkbox"/>		<input type="checkbox"/>		
pcdata	varchar	255	0	0	<input type="checkbox"/>		<input type="checkbox"/>		

图 10-26

使用这个表，地址中街道部分的数据可能如图10-27所示。

	pk_Street	fk_Address	pcdata
1	1		Arden House
2	1		1102 Warwick Road
3	1		Acocks Green
4	2		1512 North Fremont
5	2		Suite 103

图 10-27

在这个例子中有两个地址，第一个使用了街道记录为1，2和3，第二个使用4和5。现在需要修改规则2来处理元素拥有自己文本内容的情况：

- 规则2——对每一个元素节点类型，创建一个与元素相同名字的表，然后：
- 如果元素节点是一个子节点，创建一个与父元素节点同名的列，但是使用前缀fk_。
- 创建一个在刚创建的列和与其父元素节点同名的表中的列的外键关系，列的名字是父元素的名字加一个前缀pk_。
- 如果元素节点的内容只是本文，那么创建一个名字为pcdata的列。

一个元素的多发性是在<RecSubjCategories>里定义的：

程序清单 10-46

```
<ElementType name="RecSubjCategories" content="eltOnly" order="seq">
  <element type="Category"/>
  <element type="Category"/>
  <element type="Category"/>
</ElementType>
<ElementType name="Category" content="textOnly"/>
```

<RecSubjCategories>元素本身容易模拟；只要创建一个新表。可能有一些冗余——因为在

这种情况下一个<Category>可能只关联了一个<Book>——但是我们不得不保留看上去冗余的表，以便在需要时重建正确的XML结构（在后面将进一步讨论）。

比较难于模拟的部分是<Category>元素出现了三次。这意味着我们不能使用通常的规则，因为既不能创建三个名为 Category 的表——如果 Category 元素不只是文本——也不能在 RecSubjCategories 表中创建三个名叫 Category 的表——因为<Category>只是文本。

我们有一个规则说明，如果一个元素发生一次以上，它必需放在自己的表中，所以我们将创建一个名为 Category 的表。然而，我们会看到当开始写元素处理代码时，需要：

- 在当前的分层中计算元素的处理会发生的次数。
- 不要在一个元素再次发生时创建一个新表。

如果按照下面的定义，结果表将没有什么不同：

程序清单 10-47

```
<ElementType name="RecSubjCategories" content="eltOnly" order="seq">
  <element type="Category" minOccurs="1" maxOccurs="*" />
</ElementType>
<ElementType name="Category" content="textOnly"/>
```

实际上一个触发器只能触发 3 次。

注意这个规则假设后续的分类不重要。如果很重要，可以在 SQL DDL 中加入位置指示器用来指示元素显示的顺序。

(5) 枚举

在我们查看关于地址的模式时，你可能已经认识到 headquarters 属性值只能是 yes 或 no。一种可以实现的方法是把 headquarters 创建为一个布尔字段，但这在后面可能会产生问题，例如再增加一个 'Used To Be' 的选项。这样的话，我们希望建立对枚举的一般的需求，以便可以用在任何环境下。

首先，需要设置一个表，用来保存枚举值（如图 10-28）。把它叫做 enum_headquarters 用来减少与其他名字冲突的可能性。

Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value	Identity	Identity Seed	Identity Increment
pk_enum_headquarters	varchar	50	0	0	<input type="checkbox"/>		<input type="checkbox"/>		

图 10-28

这个表仅有一列，因为它只需要包含 headquarters 所可能有的值。因为它们是唯一的，所以可以被设置为主键。在表中的数据如图 10-29 所示。

这就给出枚举的规则：

- 规则5——对于每一个具有枚举类型的属性，用与属性节点值相同的名字创建一个表，使用前缀 enum_。包含一个与属性节点同名的可变长字符串列，使用前缀 pk_enum_ 和 variable-length string 数据类型。把枚举的值填在表中。

我们需要再次修改规则 3，以保证不会丢失被枚举的属性的信息：

- 规则3——对于每个属性节点：

pk_enum_headquarters
yes
no

图 10-29

- 创建一个与属性节点同名的列：
 - 如果属性是一个正常的属性使用前缀 attr_。
 - 如果属性是一个枚举类型使用前缀 attr_enum_。
- 如果要求属性，那么列不应该允许为 NULL 值。

实际上我们仍然没有完成规则 3！在后面会看到更多……

(6) 拥有 id 数据类型的属性

出版商的定义的下一部分是它的名称和地址的列表：

程序清单 10-48

```
<ElementType name="Imprints" content="eltOnly" order="seq">
  <element type="Imprint" minOccurs="1" maxOccurs="*" />
</ElementType>
<ElementType name="Imprint" content="textOnly">
  <AttributeType name="shortImprintName" dt:type="id" />
  <attribute type="shortImprintName" />
</ElementType>
```

每一个<Publisher>都有一个叫做<Imprints>的元素，按顺序包含名为<Imprint>的元素。每一个<Imprint>元素都有一些文本，像名称和地址的标题和一个叫 shortImprintName 的属性，它是 id 类型数据。

使用规则 2 我们创建了两个新表，Imprints 和 Imprint，每一个都指向它们的父元素。第一个表看上去如图 10-30 所示。

	Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value	Identity	Identity Seed	Identity Increment
?	pk_Imprints	int	4	10	0	<input type="checkbox"/>		<input checked="" type="checkbox"/>	1	1
	fk_Publisher	int	4	10	0	<input type="checkbox"/>		<input type="checkbox"/>		

图 10-30

第二个表——Imprint——包含文本内容，所以有一个 pcdat 列。这也是按规则 2 处理的。仅有一件事我们没有注意到，就是 shortImprintName 属性。这个属性为 id 类型，意思就是它将用于 XML 文档的其他部分，用来引用一个元素。实际上，这个模式已经设置了这样一种方式就是<Thread>元素——稍后我们就会看到——可以通过引用它的简称来指向一个名称地址。因为使用了枚举信息，我们不想丢掉这个信息，所以我们通过在列的前面加上 attr_id_ 来保留它。所以修改规则 3：

- 规则 3——对于每个属性节点：
- 创建一个与属性节点同名的列：
 - 如果属性是一个正常的属性使用前缀 attr_。
 - 如果属性是一个枚举类型使用前缀 attr_enum_。
 - 如果属性是 id 类型使用前缀 attr_id_。
- 如果要求属性，那么列不应该允许为 NULL 值。

所以 Imprint 表看上去如图 10-31 所示。

Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value	Identity	Identity Seed	Identity Increment
pk_Imprint	int	4	10	0	<input type="checkbox"/>		<input checked="" type="checkbox"/>	1	1
fk_Imprints	int	4	10	0	<input type="checkbox"/>		<input type="checkbox"/>		
attr_id_shortimprintName	varchar	255	0	0	<input type="checkbox"/>		<input type="checkbox"/>		
pcdata	varchar	255	0	0	<input type="checkbox"/>		<input type="checkbox"/>		

图 10-31

(7) 作为容器的元素

注意在这里我们介绍的一些冗余的东西是不可避免的。如果我们在设计这个数据库的时候没有想过将数据输出为 XML，那么可能不需要 Imprints 表，因为它只允许 Imprint 表中的名称地址能够同 Publisher 表相连接。像这样的一个表结构通常是用在当想让每一个出版商有许多名称地址的时候，但是在我们的模式中，每一个出版商仅有一个名称地址。如果我们从节点的角度来表现这个数据库结构，那么下面如图 10-32 所示的两个结构没有什么不同。

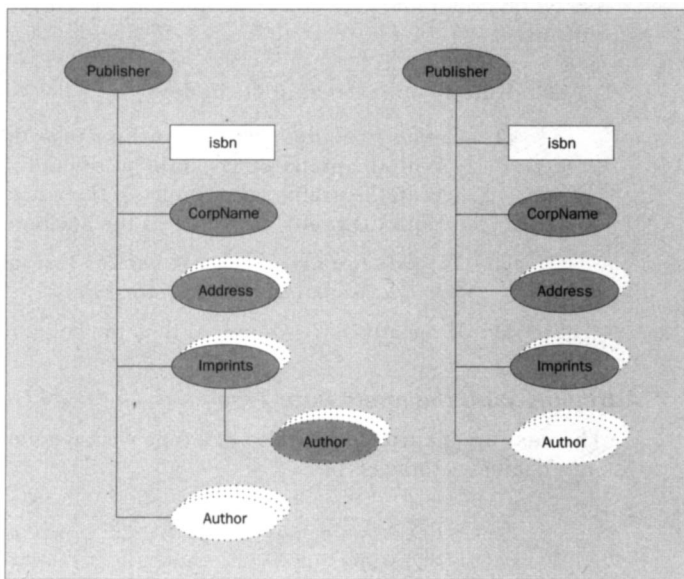


图 10-32

这是因为，Imprints 没有自己的数据——它只是其他数据的一个简单容器。然而，尽管我们想使关系数据库尽可能的高效，但也不得不保留这个表，与没必要的多个的连接在一起。否则当输出 XML 的时候，将没有办法知道在 <Publisher> 与每个 <Imprint> 之间有一个包含元素。

(8) minOccurs 属性

在 <Author> 元素中的 MI 元素可以为 NULL。这是因为 minOccurs 在模式中被设成了 0，意味着如果需要可以完全忽略那个元素。不能被忽略的元素有一个给定的空串作为缺省值，它与 NULL 不同。在输出数据的时候，我们可以检查是否它为 NULL，如果是就不会输元素。否则，尽管字段为空，它也得输出。为此我们增加一条规则：

- 规则 6——如果一个元素有一个为 0 的 minOccurs 值，设置允许列为空。

(9) 拥有指定数据类型的属性

<Catalog> 中的最后的项级元素是 <Book> 元素。定义的开始使用属性类型，我们在以前已经

看过，除了一点：

```
<AttributeType name="pageCount" dt:type="int" required="yes"/>
```

属性类型被指明了，要好于把它当作一般的文本。另一个例子是在 <Price>元素中：

```
<ElementType name="Price" dt:type="fixed.14.4" content="textOnly">
  <attribute type="currency"/>
</ElementType>
```

因此我们需要创建一个元素的正确类型的列。这就要求修改规则 3：

- 规则3——对于每个属性节点：
- 创建一个与属性节点同名的列：
 - 如果属性是一个正常的属性使用前缀 attr_。
 - 如果属性是一个枚举类型使用前缀 attr_enum_。
 - 如果属性是id类型使用前缀attr_id_。
- 数据类型应该是一个长度为 255的variable-length字符串，除非用dt:type属性指出数据类型。
- 如果要求属性，那么列不应该允许为 NULL值。

(10) 拥有idrefs数据类型的属性

下面这个属性集也是我们以前没有遇到的，但是这些不能被简单地模拟为数据库存储类型：

```
<AttributeType name="authors" dt:type="idrefs"/>
<AttributeType name="threads" dt:type="idrefs"/>
```

这两个属性的目的是指供一个用空白进行分隔引用了 <Author>和<Thread>元素的列表。没有什么可以简单地在用数据库的列对它进行维护，所以我们将通过创建与属性同名的表来模拟这种关系，使用外键来指向 Author和Thread表。我们将以创建元素表的方法来创建这个表，所以 Author将如图 10-33所示。

	Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Default Value	Identity	Identity Seed	Identity Increment
PK	pk_authors	int	4	10	0	<input type="checkbox"/>		<input checked="" type="checkbox"/>	1	1
FK	fk_Book	int	4	10	0	<input type="checkbox"/>		<input type="checkbox"/>		
	attr_idref_Author	varchar	255	0	0	<input type="checkbox"/>		<input type="checkbox"/>		

图 10-33

并且threads将同图10-34所示。

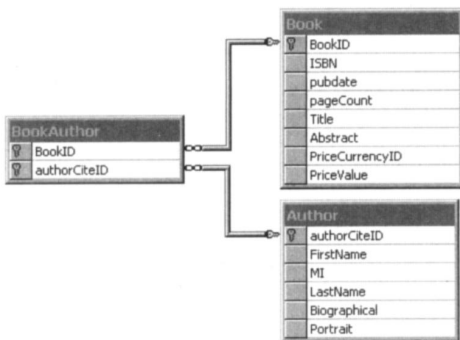


图 10-34

我们需要定义的规则如下所示：

- 规则7——如果一个属性是idrefs类型那么：
- 创建一个与属性同名的表。
- 在其中创建一个列，名字与元素名字首字母大写的单数形式一样，加上前缀 attr_idref_。
- 创建一个与首字母大写的单数形式同名的表的外键，对于要处理的表使用 attr_id_列。

因为它有一点复杂，让我们用 authors属性来进行一步步地检查。首先，创建一个名为 authors的表，拥有一般的特性。然后，在表中增加一列，名字为 authors的单数形式，且第一个字母为大写——Author——并且加上一个前缀 attr_idref_。最后，创建一个外键，将这个列与在目标表中的 attr_id_列连接（在这个例子中是 Author表的 attr_id_authorCiteID）。注意这个规则假设idrefs属性拥有一个元素名字的复数形式，它的 ID被引用——在设计XML模式时一个普通的习惯。

一个迄今为止我们已经看到的允许的模式设计数据可能是：

程序清单 10-49

```
<Book ISBN="1-861001-52-5"
      Level="Experienced"
      pubdate="1/10/99"
      pageCount="458"
      authors="1 2 4"
      threads="1 3 6"/>
```

你可以看到 authors和threads属性是在文档中任何地方出现的 ids的简单列表。

注意如果我们创建的表结构对你来说是很熟悉，那么你走对路了；我们将正确地建立同样的表和关系，用来生成下面的XML：

程序清单 10-50

```
<Book ISBN="1-861001-52-5"
      Level="Experienced"
      pubdate="1/10/99"
      pageCount="458">
  <authors>
    <Author authorCiteID="1">
      <FirstName>Stephen</FirstName>
      <LastName>Mohr</LastName>
      <Biographical>
        Stephen began programming in high school and is now a senior
        software systems architect with Omicron Consulting, he designs and
        develops systems using C++, Java, JavaScript, COM, and various
        internetworking standards and protocols.
      </Biographical>
      <Portrait picLink="http://webdev.wrox.co.uk/resources/
        authors/mohrs.gif" />
    </Author>
    <Author authorCiteID="2">
      <FirstName>Kathie</FirstName>
      <LastName>Kingsley-Hughes</LastName>
      <Biographical>
```



```

        Kathie is the MD of Kingsley-Hughes Development Ltd, a Training
        and Consultancy firm specialising in Web Development and visual
        programming languages, first going into CDF channels with The
        Dragon Channel.
    </Biographical>
    <Portrait picLink="" />
</Author>
</authors>
<threads>
    <Thread threadID="1">Internet</Thread>
    <Thread threadID="3">Programming</Thread>
    <Thread threadID="6">XML</Thread>
</threads>
</Book>

```

然而，请注意这种方法，因为每一本书都包含它的作者，所以会重复很多次——如果一个作者写了两本书，那么就会重复两次。这个 idrefs 技术允许元素包含其他的元素，而没有不必要的重复。

(11) 拥有 idref 数据类型的属性

另一个我们还没有给它定义规则的属性是 idref。一个使用这个类型的例子是当引用一本书的地址名称时：

```
<AttributeType name="imprint" dt:type="idref"/>
```

不像 idrefs，在这个属性中只能有一个引用可以出现，所以我们不需要使用一个额外的表。所需要做的是用一个合适的名字创建一个列，然后创建一个与 Imprint 表的这个列相连的外键。我们的规则将是：

- 规则8——如果一个属性是 idref 类型，那么：
- 创建一个与属性同名的列，加上前缀 attr_idref_。
- 创建一个在这个列与对应的属性同名表的 attr_id_column 间的外键关系。

在这个例子中，我们在 Book 表的 attr_idref_imprint 列与表 Imprint 表的 attr_id_shortImprintName 间的外键关系。注意这个规则假设 idref 属性有着与 id 所指的元素相同的名字——再一次指出，这是一个在设计 XML 模式时的普通的习惯。

(12) 结构

图10-35是将使用我们的规则创建 Publisher 分层信息的表和关系的表示。

图10-36是将使用我们的规则创建 Book 分层信息的表和关系的表示。

(13) 小结

使用 XML 通过 XML 模式来定义我们的 XML 文档的好处表明我们能够自动地完成许多任务。在这个例子中，我们已经创建了一系列的规则，这些规则将允许从模式中创建一个关系数据库。

在查看代码之前，将总结一下迄今为止所创建的规则。注意我已经增加了一些子句用来明确地显示规则的优先级（后面的规则在某种情况下要更先一些）：

- 规则1——无论何时创建一个新表，创建一个与表名相同的主键，但是使用一个前缀 pk_。这一列将是一个 automatically-incremented（自动增加）的整数。
- 规则2——对每一个与规则4不匹配的元素节点类型，创建一个与元素相同名字的表，然后：

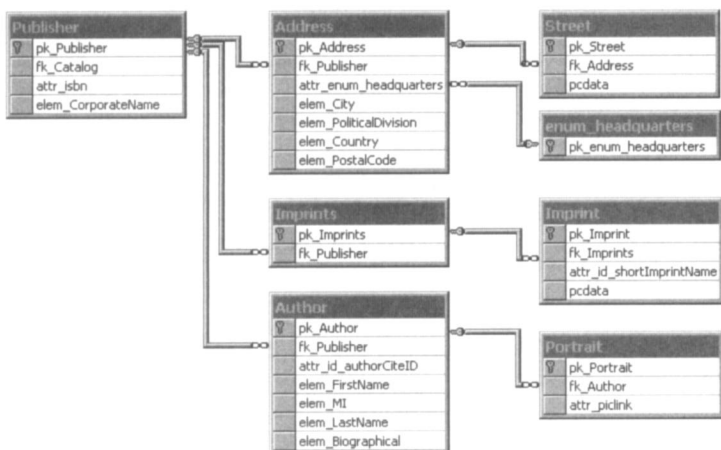


图 10-35

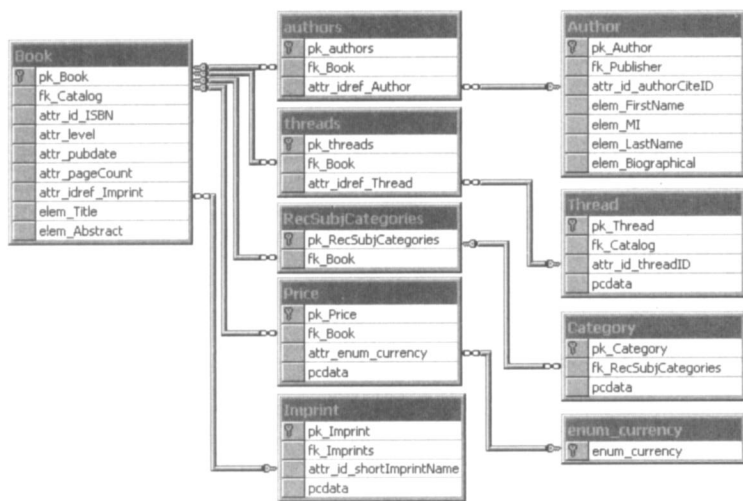


图 10-36

- 如果元素节点是一个子节点，创建一个与父元素节点同名的列，但是使用前缀 `fk_`。
- 创建一个在刚创建的列和与其父元素节点同名的表中的列的外键关系，列的名字是父元素的名字加一个前缀 `pk_`。
- 如果元素节点的内容只是本文，那么创建一个名字为 `pcdata` 的列。
- 规则3——对于每个与规则5、7或8不匹配的属性节点：
- 创建一个与属性节点同名的列：
 - 如果属性是一个正常的属性使用前缀 `attr_`。
 - 如果属性是一个枚举类型使用前缀 `attr_enum_`。
 - 如果属性是 `id` 类型使用前缀 `attr_id_`。

- 数据类型应该是一个长度为255的variable-length字符串，除非用dt:type属性指出数据类型。
- 如果要求属性，那么列不应该允许为NULL值。
- 规则4——如果一个元素的节点只包含文本，并且最大发生次数为1次，那么在它的父元素表中创建一列，列的名字与节点一样，使用前缀elem_。
- 规则5——对于每一个具有枚举类型的属性，用与属性节点值相同的名字创建一个表，使用前缀enum_。包含一个与属性节点同名的可变长度字符串列，使用前缀pk_enum_和variable-length string数据类型。把枚举的值填在表中。
- 规则6——如果一个元素有一个为0的minOccurs值，设置允许列为空。
- 规则7——如果一个属性是idrefs类型那么：
 - 创建一个与属性同名的表。
 - 在其中，创建一个列，名字与元素名字首字母大写的单数形式一样，加上前缀attr_idref_。
 - 创建一个与首字母大写的单数形式同名的表的外键，对于要处理的表使用attr_id_列。
- 规则8——如果一个属性是idref类型，那么：
 - 创建一个与属性同名的列，加上前缀attr_idref_。
 - 创建一个在这个列与对应的属性同名表的attr_id_column间的外键关系。

2. 自动创建数据库

下面的脚本出了我们前面章节设计和创建树的一系列指令的规则。我们已经讲过了，因为同等的XSLT样式表转换成XML-DR模式是非常复杂的，所以，如果转换过程太复杂，就应该使用脚本来实现。在关系型数据库的两个表之间创建关系，主要的问题是不仅需要这些表存在，而且在目标表上的主键要存在。为了做到这一点，我们总是在创建表和键值之后创建关系。这样做的最安全的办法是，在我们能够写入的任何层次点，进程的任何阶段构造表示这些结构的节点树。

使用节点结构构造命令的另外一个优点是，可以有机会优化命令。例如，为每一个元素建立一个命令创建表之后，我们能够删除这些命令，这些元素能够容易地以列的形式存储。

脚本(schematordb.asp)首先创建表单定义模式，然后，两个DIV拥有处理结果。第一步提供查看将被创建的表和列的快捷方式，第二步给出了将被产生的活动SQL命令：

程序清单 10-51

```
<HTML>
<HEAD>
<TITLE>Schema to RDBMS Converter</TITLE>
</HEAD>
<BODY>
<TABLE>
  <TR>
    <TD><FONT face=Verdana>Schema File: </FONT></TD>
    <TD>
      <INPUT id="loadfile"
        name="loadfile"
        style="HEIGHT: 22px; WIDTH: 230px"
        value="file:c:/PubCatalog.xml">
    </TD>
  </TR>
</TABLE>
```

```

<TD>
  <FONT face=Verdana>
    <INPUT id="GenerateBtn"
      name="GenerateBtn"
      type="button"
      value="Generate"
      OnClick="OnGenerateClick()">
  </FONT>
</TD>
</TR>
</TABLE>
<HR>
<TABLE>
  <TR>
    <TD valign="TOP"><DIV id="infoResult"></DIV></TD>
    <TD valign="TOP"><XMP id="sqlResult"></XMP></TD>
  </TD>
</TABLE>

```

脚本开始时创建 XML DOM，处理 XML 计划输入，产生 SQL DDL 输出，产生样式表的输出。在这个过程中包括两个样式表。第一个把创建的命令列表转换成每一个列的属性摘要，按照将要显示的列的次序排序。主键和外键将能够看到。

第二个样式表把相同的创建节点转换成正确的 SQL 语句，创建反映模式的数据库。

首先，我们创建一些变量存储输出的各个部分：

程序清单 10-52

```

<SCRIPT language="Javascript">
<!--
var COMPLETED = 4;

var parser = new ActiveXObject("microsoft.xmlDOM");
var oParsOut = new ActiveXObject("microsoft.xmlDOM");
var oStyle = new ActiveXObject("microsoft.xmlDOM");

var oDB, oDBT, oDBPK, oDBID, oDBFK, oDBDAT;

```

当按下按钮时，函数被执行：

程序清单 10-53

```

function OnGenerateClick()
{
  parser.async = "false";
  oParsOut.async = "false";
  oStyle.async = "false";

```

首先装入模式文档，然后检验它是否真正包含模式节点：

程序清单 10-54

```

parser.load(document.all("loadfile").value);
if (parser.readyState == COMPLETED && parser.parseError == 0)
{
  oDE = parser.documentElement;

```

```
// Minimal check for XML-DR validity
if (oDE.nodeName == "Schema")
{
```

下一步，创建所有存储计划表的分析结果的元素：

程序清单 10-55

```
oDB = oParsOut.createElement("DB");
oDBT = oParsOut.createElement("Tables");
oDBPK = oParsOut.createElement("PrimaryKeys");
oDBID = oParsOut.createElement("IDs");
oDBFK = oParsOut.createElement("ForeignKeys");
oDBDAT = oParsOut.createElement("Data");
```

我们实现模式的第一种途径是创建需要的所有的表。这是一个包含每一种元素类型的表，拥有idrefs类型的任何属性类型的表和拥有枚举类型的每一种类型的表。这两种情况的表在元素和属性类型信息存储之后被命名：

程序清单 10-56

```
// First we create a list of all the tables we might need.
// We will optimise this later, so some may get deleted.
// We need a table for:
// - each ElementType
// - each AttributeType which has a type of 'idrefs'
oNL = oDE.selectNodes(
    "//ElementType | //AttributeType[@dt:type='idrefs']"
);
for (var i = 0; i < oNL.length; i++)
{
    sTable = oNL(i).attributes.getNamedItem("name").value;
    CreateTABLE(sTable, true);
}
```

枚举的表被加以前缀enum_to确保不和别的名字冲突。我们不得不创建必要的结构，把列举的值加入到列举的表中：

程序清单 10-57

```
// - each AttributeType which has a type of 'enumeration'
oNL = oDE.selectNodes("//AttributeType[@dt:type='enumeration']");
for (var i = 0; i < oNL.length; i++)
{
    sTable =
        "enum_" + oNL(i).attributes.getNamedItem("name").value;
    CreateTABLE(sTable, false);
    CreateDATA(
        sTable,
        oNL(i).attributes.getNamedItem("dt:values").value
    );
}
```

虽然我们为每一个元素类型创建了表，但对元素之间存在的层次关系一无所知。因此，下

一阶段通过每一个元素和在匹配元素类型中定位它的定义的方式进行：

程序清单 10-58

```
// Next step through all the elements:
// - elements that are not text only, get attached to their parent
// - elements that are text only, but can occur more than once or
//   have attributes also get attached to their parent
// - other elements become columns
oNL = oDE.selectNodes("//element");
for (i = 0; i < oNL.length; i++)
{
    sType = getAttr(oNL(i), "type");
    bMulti = (getAttr(oNL(i), "maxOccurs") == "*");
    // Get the definition of the element (the ElementType)
    oET = oDE.selectSingleNode(
        "//ElementType[@name='" + sType + "']"
    );
    if (oET != null)
    {
        // We leave the table intact and link to the parent if the
        // element is NOT text only
        // OR there is more than one occurrence of the element
        bHasAttr = (oET.selectNodes("attribute").length != 0);
```

如果元素不仅仅是纯文本，或者不仅仅是元素的一个事件，或者是一些属性，我们将把它留在表中，并创建连接。然而，如果元素是纯文本，或者是一个事件，或者它没有属性，它能在父表中安全地当作一个列：

程序清单 10-59

```
if ((getAttr(oET, "content") != "textOnly") || bMulti ||
    bHasAttr)
{
    CreateLookup(sType, getAttr(oNL(i).parentNode, "name"));
}
// Otherwise mark the table for deletion from our create
// table list, and add the
// element as a column instead
else
{
    oTemp = oDBT.selectSingleNode("Table[@name='" + sType
                                   + "']");
    oTemp.setAttribute("Delete", "yes");
    NewColumnDEF(getAttr(oNL(i).parentNode, "name"),
                  "elem_" + sType);
}
}
```

介绍了所有的元素后，我们现在了解所有的属性：

程序清单 10-60

```
// Now put the attributes on. Those that refer to enumerations have
```



```
// special syntax
oNL = oDE.selectNodes("//attribute");
for (i = 0; i < oNL.length; i++)
{
```

对每一种属性，取得它们的定义：

程序清单 10-61

```
sType = getAttr(oNL(i), "type");
// Get the definition of the attribute (the AttributeType)
oAT = oDE.selectSingleNode("//AttributeType[@name='" + sType
                           + "']")
if (oAT != null)
{
```

不论是什么属性类型，需要对父表做一些事情，现在取得它的名称：

```
// Get information about the table that contains the attribute
sContainer = getAttr(oNL(i).parentNode, "name");
```

严格上来说，对属性做的任何事情依赖于它的类型：

```
switch (getAttr(oAT, "dt:type"))
{
```

枚举是依靠创建在属性元素表上的列实现的。枚举表在初期就已经被创建了。

```
case "enumeration":
    CreateLookup(sContainer, "enum_" + sType);
    break;
```

IDREF也是加到父节点的一列：

```
case "idref":
    CreateIDREF(sContainer, CapFirst(sType));
    break;
```

IDREFS需要建立一个连接：

```
case "idrefs":
    CreateIDREFS(sContainer, sType);
    break;
```

ID意味着加入一列到父节点中：

```
case "id":
    CreateID(sContainer, sType);
    break;
```

指定数据类型意味着创建指定数据类型的列：

程序清单 10-62

```
case "fixed.14.4":
    NewColumnMONEY(sContainer, "attr_" + sType);
    break;

case "int":
    NewColumnINT(sContainer, "attr_" + sType);
    break;
```

```

        default:
            NewColumnDEF(sContainer, "attr_" + sType);
            break;
    }
}
}

```

现在我们在对列表的循环中创建或者删除某一个表，来达到优化的目的。注意，如果删除命令创建表，则不得不删除相应的结构创建在表中的主键：

程序清单 10-63

```

// Finally, delete the tables we no longer need
oNL = oDBT.selectNodes("Table[@Delete='yes']");
for (i = 0; i < oNL.length; i++)
{
    oTemp = oDBT.removeChild(oNL(i));
    // If we delete a table we should delete its primary key too
    oCL = oDBPK.selectNodes("PK[@table='" + getAttr(oTemp, "name") +
        "' ]");
    for (j = 0; j < oCL.length; j++)
    {
        oTemp = oDBPK.removeChild(oCL(j));
    }
}
}

```

我们已经创建了列表，因此把它加入到包含节点中。

```
oDB.appendChild(oDBT);
```

把所有的限制、主键、外键和其他的内容，放进另一个容器，把它们放入包含节点中：

程序清单 10-64

```

oNode = oParsOut.createElement("Constraints");
oNode.appendChild(oDBPK);
oNode.appendChild(oDBID);
oNode.appendChild(oDBFK);
oDB.appendChild(oNode);

```

加入指令列表，用数据组装一些表：

```
oDB.appendChild(oDBDAT);
```

最后，为转换存储完整的文档，装入需要的样式表（DBCreateView.xsl，带有这本书的可下载的代码）。你可能需要为这个样式表编辑 URL 以便在自己的系统上定位它。如果装入成功，文档解析正确，转换它，并且把结果放在网页上正确的地方：

程序清单 10-65

```

oParsOut.documentElement = oDB;

oStyle.load("http://server/DBCreateView.xsl");
if (oStyle.parseError.errorCode != 0)
{
    sResult = reportParseError(oStyle.parseError);
}

```

```

    }
    else
    {
        try
        {
            sResult = oParsOut.transformNode(oStyle);
        }
        catch (exception)
        {
            sResult = reportRuntimeError(exception);
        }
    }
    infoResult.innerHTML = sResult;

```

第一次转换创建了将在数据库中被创建的摘要，列出了所有的表和列。下一次转换将自动发送一系列指令给 SQL 数据库，创建数据库的元素。（为了定位 DBCreateSQL.xml——带有可下载的代码，你需要改变系统显示的 URL）：

程序清单 10-66

```

oStyle.load("http://server/DBCreateSQL.xml");
if (oStyle.parseError.errorCode != 0)
{
    sResult = reportParseError(oStyle.parseError);
}
else
{
    try
    {
        sResult = oParsOut.transformNode(oStyle);
    }
    catch (exception)
    {
        sResult = reportRuntimeError(exception);
    }
    sqlResult.innerText = sResult;
}
else
    alert("The URL doesn't designate a schema file under XML-DR rules.");
}
else
    alert("Parser detects error: " + parser.parseError.reason);
}

```

剩下的代码提供需要建立命令指令的所有函数。第一个创建了一个轮流需要主键的新表：

程序清单 10-67

```

function CreateTable(s, bAuto)
{
    var oTemp = oParsOut.createElement("Table");
    oTemp.setAttribute("name", s);
    oDBT.appendChild(oTemp);
    CreatePK(s, bAuto);
}

```

下一个函数创建一个在表中的 ID 列：

程序清单 10-68

```
function CreateID(s, c)
{
    var oTemp = oParsOut.createElement("ID");
    c = "attr_id_" + c;
    oTemp.setAttribute("table", s);
    oTemp.setAttribute("name", c);
    oDBID.appendChild(oTemp);

    // Add the column to the create table node
    NewColumnDEF(s, c);
}
```

创建数据需要我们在一个已经存在的表上执行插入动作：

程序清单 10-69

```
function CreatedATA(sT, s)
{
    var ar = s.split(" ");
    for (var i = 0; i < ar.length; i++)
    {
        var oTemp = oParsOut.createElement("Insert");
        oTemp.setAttribute("table", sT);
        oTemp.setAttribute("value", ar[i]);
        oDBDAT.appendChild(oTemp);
    }
}
```

主键和这个函数一起被创建。注意，如果键值的主要目的是作为终点来参考，那么，它可能有 bAuto 参数设置。它设置了自动增加的整型的主键：

程序清单 10-70

```
function CreatePK(s, bAuto)
{
    var oTemp = oParsOut.createElement("PK");
    oTemp.setAttribute("table", s);
    oTemp.setAttribute("name", "pk_" + s);
    oDBPK.appendChild(oTemp);
    if (bAuto)
    {
        NewColumnAUTOINC(s, "pk_" + s);
    }
    else
    {
        NewColumnDEF(s, "pk_" + s);
    }
}
```

在一个表上设置外键可进行查找，指向另外一个表的主键：

程序清单 10-71

```
function CreateLookup(sT1, sT2)
{
    // Creating a lookup between one table and another requires the primary key
    // of the second table
    var oTemp = oDBPK.selectSingleNode("PK[@table='" + sT2 + "']");
    CreateRel(sT1, "fk_" + sT2, sT2, getAttr(oTemp, "name"));
}
```

一个IDREF列能在两个表之间建立联系，但没有必要对指定的表的主键建立联系：

程序清单 10-72

```
function CreateIDREF(sT1, sT2)
{
    // Creating an ID lookup between one table and another requires the
    // attribute that has a property of 'id' from the second table. We've
    // already created these in the 'unique values' list, so just look it up
    var oTemp = oDBID.selectSingleNode("ID[@table='" + sT2 + "']");
    CreateRel(sT1, "fk_attr_idref_" + sT2, sT2, getAttr(oTemp, "name"));
}
```

一个IDREFS属性已经成为了一个媒介，或者连接，被创建的表。我们需要找到这个表，从两个表中找到它，通过这个媒介自动连接：

程序清单 10-73

```
function CreateIDREFS(sT1, sT2)
{
    // Creating an IDREFS lookup involves an intermediate table.
    //The instruction to
    // create this will already be in our list, so find that first
    var oTemp = oDBT.selectSingleNode("Table[@name='" + sT2 + "']");
    CreateLookup(sT2, sT1);
    sT1 = sT2;
    sT2 = CapFirstSingular(sT2);
    CreateIDREF(sT1, sT2);
}
```

通过对需要创建的外键的列表增加一个指令来建立一个联系：

程序清单 10-74

```
function CreateRel(sT1, sFK, sT2, sPK)
{
    // Create an instruction to generate a foreign key
    var oTemp = oParsOut.createElement("FK");
    oTemp.setAttribute("table1", sT1);
    oTemp.setAttribute("src", sFK);
    oTemp.setAttribute("table2", sT2);
    oTemp.setAttribute("dest", sPK);
    oDBFK.appendChild(oTemp);

    NewColumnDEF(sT1, sFK);
}
```

这个函数允许我们找到一个已经存在的指令去创建表：

```
function getCreateTable(s)
{
    return oDBT.selectSingleNode("Table[@name='" + s + "']");
}
```

这两个函数允许我们转换单词，像 author到Author (CapFirst)，authors到Author (CapFirstSingular) 一样，在属性有IDREF，IDREFS的值时，创建被引用的表的时候进行转换：

程序清单 10-75

```
function CapFirst(s)
{
    return s.substr(0, 1).toUpperCase() + s.substr(1);
}

function CapFirstSingular(s)
{
    return CapFirst(s).substr(0, s.length-1);
}
```

在这个函数中，一个详细数据类型的新的列被增加了：

程序清单 10-76

```
function NewColumn(sT, s, sType, bNull)
{
    var oTemp = oParsOut.createElement("Column");
    oTemp.setAttribute("name", s);
    oTemp.setAttribute("type", sType);
    oTemp.setAttribute("null", bNull ? "yes" : "no");
    getCreateTable(sT).appendChild(oTemp);
}
```

下面描述了更多的指定函数的调用：

程序清单 10-77

```
function NewColumnDEF(sT, s)
{
    NewColumn(sT, s, "varchar(255)", false);
}

function NewColumnINT(sT, s)
{
    NewColumn(sT, s, "int", true);
}

function NewColumnAUTOINC(sT, s)
{
    NewColumn(sT, s, "int identity(1, 1)", false);
}

function NewColumnMONEY(sT, s)
{
    NewColumn(sT, s, "money", true);
}
```


这个函数允许我们从一个节点中检索属性的值，如果这个属性不存在，返回一个空串：

程序清单 10-78

```
function getAttr(o, s)
{
    var sRet = "";
    if (o != null)
    {
        var n = o.attributes.getNamedItem(s);
        if (n != null)
        {
            sRet = n.value;
        }
    }
    return sRet;
}
```

最后，是我们的错误输出函数。第一个使用在装入另外的 XML文件的时候，报告了一些解析错误。第二个使用在运行错误发生的时候：

程序清单 10-79

```
// Parse error formatting function
function reportParseError(error)
{
    var s = "";
    for (var i=1; i<error.linepos; i++)
    {
        s += " ";
    }
    r = "<font face=Verdana size=2><font size=4>XML Error loading '" +
        error.url + "'</font>" +
        "<P><B>" + error.reason +
        "</B></P></font>";
    if (error.line > 0)
        r += "<font size=3><XMP>" +
            "at line " + error.line + ", character " + error.linepos +
            "\n" + error.srcText +
            "\n" + s + "^" +
            "</XMP></font>";
    return r;
}

// Runtime error formatting function
function reportRuntimeError(exception)
{
    return "<font face=Verdana size=2><font size=4>XSL Runtime Error</font>" +
        "<P><B>" + exception.description + "</B></P></font>";
}
//-->
</SCRIPT>

</BODY>
</HTML>
```

命令的列表必须在创建指定数据库之前先进行准备，因为它允许我们优化和排序这些命令。例如，创建了需要的表的列表后，可以检查和删除纯文本的表。可以保证在两个表之间建立关系之前，所有的表都已经准备好了。

一旦我们有命令列表，样式表能够被应用，转换命令为 SQL 的描述。两个阶段的进程是比较有利的，因为，它把输入和输出分离开来。如果创建表的命令或插入键值与另外类型的数据库不兼容，转换样式表能够被改变。如果使用了不同的语法，源程序代码能够被改变。

图 10-37 是运行书中目录模式脚本的结果：

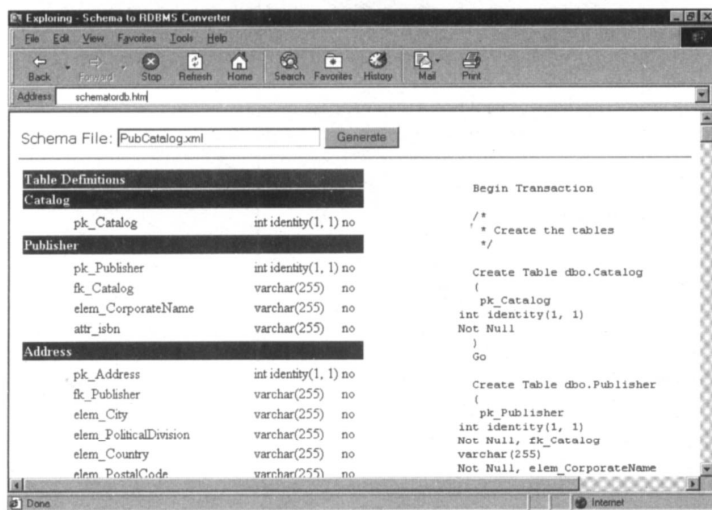


图 10-37

左边是需要被执行的指令的列表，右边是一系列 SQL 语句指令。

10.4 结论

XML 和数据库之间仍然有很大的距离。我们已经建立了大量的方法去完成任务——有时候是太多的方法。然而，数据库厂商的执行官曾经声明，承诺给数据库加入 XML 功能，把数据库的功能带给 XML，并且将加速发展。

XML 将日益成为数据交换、显示、索引等的基础，它将进入我们当前处理的每一种应用和设备。从移动电话，到家庭娱乐系统，到人造卫星系统，基于 XML 基础的应用将会更简洁和拥有更高的效率。

同时，通信语言逐渐使用 XML，诸如到处存在的信息存储单元、文档、图像、传真、音频、视频、电子表格，将发现它们本身的文件格式将成为 XML 的格式。Macromedia 和 Quark 公司已经宣称它们的知名产品将使用 XML 格式。Microsoft 也声明即将发行的 Office 2000 套餐中将附带 XML 的语句。

随着 XML 的大量应用，程序员和系统设计师设计交付动态 XML 文档变得越来越重要。为了做到这一点，他们当然需要了解数据库技术。