



实战 XML 第二版

原作者: **William Psrdi**

译者: 关欣怡

出版日期: 2000/6/01

书号: 957-0312-48-3

前言

HTML 已经成为全球信息网的基础。它提供标准化的方法将信息格式化并经由因特网传送给全世界的使用者。HTML 为人们的传送和接受信息带来了革命性的变化。但是 HTML 主要是被设计为资料显示之用，因此，HTML 的焦点几乎完全放在信息应如何显示上，而不是信息的内容及它的结构。这便是我们需要 XML 的原因。

XML 是一个开放的，以文字为基础的卷标语言，它可以提供结构的以及与语意有关的信息给数据。这些「关于数据的数据」或中继数据（**metadata**）提供附加的意义和目录给使用那些数据的应用程序，而且也将以网络为基础的信息管理和操作提升到一个新的水平。XML 是 SGML 的一个子集合，并且为了要在网络上应用而作了最佳化的设计。这使 XML 成为 HTML 一个强而有力且标准化的互补，作为在网络上传递信息的一个重要的功能。

XML 的目的是同时供网页的创作者和程序设计人员使用。既然 XML 是以文字为基础的，所以它可以很容易地让非技术性人员了解。但是，它所提供的组织、描述和结构化数据的能力，使它也可作为技术性的应用程序所使用。因此，XML 所建立的结构化数据便可同时供数据处理和显示之用。

虽然 XML 也被称为一种语言，但是严格来说它应该是中继语言—**metalanguage**。这表示 XML 能用来创造其它的语言。藉此，语汇一是 XML 的语言应用，便可因应特定问题而创建或解决特定的问题。XML 语汇的一些例子如下：

- **XSL**：一种以样式表为基础的格式化语言，能应用 XML 数据，并产生不同应用范围的输出结果。
- **XLL**：一个延伸的连结规格，可将 HTML 的超级链接机制带到新的水平。
- **SMIL**：一个标准化的多媒体制作语言，允许复杂的多媒体应用程序在网页上显示。
- **XSL Patterns**：XML 的查询语言，它提供 XML 数据的进阶搜寻能力。

关于本书

作为一个 XML 发展的观察者和实作人员而言，笔者很讶异于整个软件工业是如此热切地期盼来支持和使用这个技术。很难不在一本计算机杂志或任何应用软件制造厂商的网站看到有关于 XML 的消息。XML 的开发人员都了解，要使 XML 广泛地被采用，它不只要容易了解，它还必须有许多的支持资源可供使用。以笔者的观点而言，已经看到业界对 XML 的需求。因此，希望这本书会成为有效的工具，以便帮助开发人员和文件作者成功地使用 XML。

本书的重点

在软件业界已有许多有关 XML 的实际应用及许多 XML 处理器。本书的重点在于微软公司 IE5 中 XML 的实作和微软公司的 XML 处理器—Msxml。IE5.0 中有些特有的功能是 Msxml 处理器尚未支持的，这些特别的功能会在 [第 11 章](#) 中讨论。所以，在本书中笔者都尝试直接讨论 XML 语言本身和 Msxml 处理器。虽然，本书里的所有例子及随书光盘上的范例都可以在 IE5，或较早的版本中执行，它们也可以在任何其它使用 Msxml 应用程序中执行，像是 Microsoft Visual Basic 或 C++，唯一的例外是 [第 7 章](#) 中讨论的 XLink 和 XPointer（它们尚未纳入 IE5 里 Msxml 的支持范围）。然而，因为它是 XML 的一个重要的部分而且最后将会被许多 XML 应用程序所采用，所以本书中还是会有关于这个主题的讨论。

您应该知道的

这本书的目标读者是中级的计算机使用者。笔者希望读者至少都有关于 HTML 及全球信息网的基本知识或经验。然而，像 SGML 等卷标语言的知识是有帮助的但不是必需的。程序设计的能力也非必需的，但是懂一些 Script 对了解本书的内容会有所帮助的。如果您面临下列几种情况，本书绝对会让您受益无穷：

- 想要将 XML 纳入您现有的 WEB 开发计划中
- 想要了解如何应用 XML 语言到整个企业环境中
- 想要知道如何开发 XML 的应用程序
- 想要学习如何使用 XML 来管理及建立结构化的数据
- 只想了解有关 XML 的基本知识

范例

程序代码的范例如下所示：

```
<?xml version="1.0"?>
<MEMO>
  <TO>Jodie</TO>
  <FROM>Bill</FROM>
  <CC>Philip</CC>
  <SUBJECT>My first XML document</SUBJECT>
  <Body>Hello, World!</Body>
</MEMO>
```

有灰色背景的程序代码都可以在随书光盘中找到，JScript 为本书中主要的 Script 程序语言。

大小写

所有的卷标都是以大写字母，如：<TITLE></TITLE>表示，对象的属性、方法和事件则是大小写有别的。但 XML 的宣告应该永远是以小写出现的，如：<?xml version=" 1.0 "?>。虽然 XML 规格中并未硬性规定大小写的差别，在程序中维持大小写的一致性则是一个好习惯。

网络资源

大多数的章节都包含相关的网络资源地址。这本书在编写时，这些网址都正确的。

[附录 C](#) 列出在 SBN(Microsoft Site Builder Network)和 MSDN(Microsoft Developer Network)上相关的主题，这些主题都能在随书光盘或因特网上找到。为了要使用 CD 上的相关主题，打开 Workshop XML Reference.htm 便为 SBN 的内容目录，或到 <http://msdn.microsoft.com/resource/pardixml.htm>，依循连结找寻相关的主题。

在命令列使用 Msxml 解析器

从随书光盘的 Tools\Command Line XML 档案夹中，将 Msxml.exe 复制到你的计算机上，在「MS-DOS 模式」中执行它，执行时指定一个 XML 档案的名称。这个工具会告诉您这个 XML 文件是否有效，并且也能用来转换 XML 成不同的编码(encoding)。请参阅 Command Line XML_readme.htm 档案中使用这个工具的说明。

随书光盘中包含：

- 书中所有的程序代码都已放在 **sample** 目录下以章命名的子目录中（您可以双击 **Setup.exe** 来安装这些范例）。
- 在 Ebook 目录中有 HTML 格式的电子书。
- SBN Workshop 网站的纲要节录。
- 在 Tools 目录中有测试及建立 XML 档案的工具。
- 在 XML1Spec 目录中有一份 HTML 格式的 XML 1.0 规格书。

译者序

随着因特网的持续发烧到现在的 B2C、B2B 的热潮，相信有许多人正赶着搭上这班列车，只是对于常听到的像是 DNA 2000、DNS、BizTalk、Business Internet、WAP、SOAP...等专有名词常常令人感到一阵晕眩、不知所措，而 XML (eXtensible Markup Language) 更是其中一个令大家好奇却又不知如何下手了解的技术。常常会听到有人说：「听说 XML 会取代 HTML 成为新的标准？」这样的说法造成许多 HTML 的程序设计师感到恐慌，以前辛苦学的 HTML 难道要被淘汰了吗？我还要再学另一种语言吗？这些问题都一一地在心里浮现。

其实这种说法只说对了一部分，HTML 和 XML 是彼此独立的技术，可以整合使用，也可以单独运用。XML 其实是一种关于数据的数据，以前 HTML 的程序设计师可能只能用 W3C 定义好的卷标来定义数据显示格式，例如：了解 XML，稍微对 HTML 熟悉的朋友一看，可能知道它是粗体的「了解 XML」，但是还是不知道这个「了解 XML」真正有什么意义。如果能将它改成<Chapter>了解 XML</Chapter>，那我们就能很容易的了解它指的是章节的标题。所以「了解 XML」是数据，而<Chapter>则代表数据的意义，所以我说 XML 是关于数据的数据（意义）。当然卷标是不能任意定的，定义标签必须符合 XML 规定的规则，这在第 3、4 章有详细的说明。

其实 XML 真正的用意是要将数据与格式分开，不再像以前撰写 HTML 那样，将要显示的数据与格式写在一起，您可以轻易地将数据（XML）与接口样板（XSL）整合在一起而产生显示 XML 资料的网页（这在第 8、9 章会详细介绍），这样对资料的维护有很大的帮助，这在 B2B 的数据交换上是个很重要的关键技术。

本书主要是引领读者认识 XML，许多更深入的实作探讨，读者可以参考 Michael J. Young 所著的《XML Step by Step》一书，在不久的将来此书也将译成中文，造福更多读者。

最后，本书要感谢许建志先生在技术上所提供的支持与协助，以及林智权先生严格的要求与耐心的等待才能使本书顺利完成。

官欣怡

2000.6 台北

Part 1 XML 简介

1. 了解卷标语言

- . 卷标语言简史
- . 标签是如何运作的
- . 特定与通用卷标语言
- . 卷标语言的概观

2. 进入主题—XML

- . 什么是 XML?
- . XML 适用于何处?
- . XML 的目标
- . XML、建议书，与标准

1. 了解卷标语言

读这本书是因为您想要学习如何使用 XML（可延伸卷标语言）。若是您跟我一样，您可是会想在拿起计算机书后马上就开始动手撰写程序。您可能已经听到很多关于 XML 如何改变 Web，以及整合各种不同型态数字讯息的传言。或者您也确信 XML 一定是必须学习的您，而且非常渴望赶紧跳入这个领域并开始撰写这些程序。假如您是这种人，您挑对书了！我们将会在 [第2章](#) 结束前学会一些 XML 程序的开发。但是您的目标应该放在真正深入地了解 XML，您将可以从它过去的发展背景历程得到一些认知。除了深入探讨相关的程序外，前几章也为其它章节提供了一个完整的架构。您应该可以在读完这几章后，对 XML 有更深一层的了解。了解整个 XML 完整的架构是很重要的，理由如下：

- 您可能对卷标语言所使用的一些观念不很熟悉。前几章将会让您很快地了解这些语言的基本概念以及它们是如何运作的。
- 您可能曾经使用过 HTML（Hypertext Markup Language）或 SGML（Standard Gen-eralized Markup Language），而应该了解 XML 与这两种语言之间的差异性，以及为何 XML 能变成如此强有力的另一项选择。

从很多观点来看，XML 代表的是信息在 Web 上传递方式的改变。XML 也许不像其它 Web 技术让人感觉到那样的「艳丽」，但是它却有改变 Web 传递信息的潜力，就如同 HTML 数年前所产

生的影响一样。在本章中，您将可以了解为什么我们需要 XML 这种具有可扩充性的语言，我们将讨论文字卷标的发展背景与运作方式，也将检视一些较常见的卷标语言，并比较它们之间的不同处。

卷标语言简史

卷标语言根源于传统印刷。单以「标签」（markup）这个字来看，即是由英文 mark up 两字相连而来，意指在稿件或文章上加上各种记号以便付印。不仅如此，英文里「markup」一词更特别是指针对电子文件加以做标记。主要是因为以下两个目的：第一，修改文件的格式以及外观；第二，建立文件的架构与意义以便传输到其它媒介，如打印机或 World Wide Web。

若您曾经使用过微软的 FrontPage HTML 编辑器或者是 Word 文字处理器，应该稍微了解文件中变换文字格式的观念，不过您可能不知道这些编辑程序一样使用了标示的观念来完成更改文件格式，稍后我们将来看看这是如何运作的。

除了将文字格式化外，卷标也能定义本文中各元素的意义与整体结构。例如，标签可以建立仅包含姓名、生日，以及年龄等元素（element）的文件；甚至可进一步声明，除非文件中包含有姓名元素，否则文件中将不包含生日与年龄元素。而卷标更可指定姓名元素必须为文字型态，生日元素必须为日期型态，而年龄元素必须为数字型态。以这种方式，标签便可以建立起文件的结构并明确定义出元素的语义。稍后几章将涵盖这个主题并做完整的探讨。

旧式的文件处理方式

在电子出版业尚未出现之前，需先拷贝手稿，其方式是用手抄或打字，再对完成后的副本以人工手动方式注记与编辑。草稿可能来回校订修改多次，有时可能要重打文件，或将文件上的标记符号涂涂改改好多层才能底定。对于文件中不同部分的风格以及文件本身的样式，这些类型列出一份详细的说明，并把它当作手写注释的一部分。当文件送到打字处时，最后的样式将会确立样版并展示，然后完成的文件将送去印刷。

进入电子印刷业

电子文件的前置准备工作省去大量的手工需求。在印刷之前，整个文件的处理过程中，改变元素变得容易多了。传统排版方式中，本文格式的选择如字型、段落起始点、边界，以及对齐功能等，都是由排版人员建立的。排版人员使用文件上有标示或脚注的铅字样版，执行完整的拷贝计算以确定具可读性（代表排版无误），于是完成此页的制版。

在电子文件的世界中，也有相同的处理程序。我们必须加「码」（**code**）到文字中。如此，才可以让输出装置知道文件的结构以及文件应显现什么样的外观。这个「码」即是电子式的卷标。

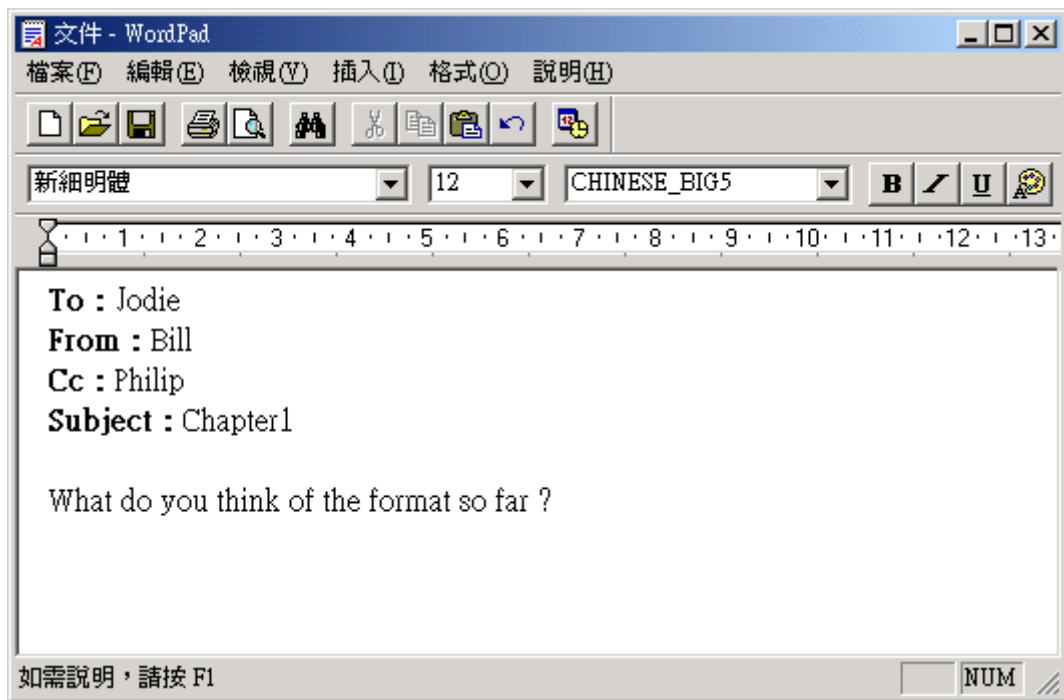
标签是如何运作的

标签基本上是由一些码（**code**）或标签（**tag**）所组成的，这些卷标会被附加在文字中，以改变被标注文字的意义与外观。文件中被标注的文字，通常称为「原始码」（**source code**）或简称为「码」（**code**）。大部分的文字处理器、桌上出版系统，甚至简单的文字编辑器都可以使用一些卷标语言来产生格式化的文件。例如，本书即是使用微软 **Word** 软件所编写，其支持的卷标语言为 **RTF**（**Rich Text Format**）格式。

卷标常被用来改变文字的外观，只要藉由格式化标签的加入，像粗体、斜体、字号，及文字缩排等即可改变面貌。标签传统上运作的方式是在需要时启动上述的属性，而在不需要时关闭它们即可。现在让我们来看一个例子。

RTF 标签

如先前所提，**Word** 支持 **RTF** 卷标语言，而随附于 **Microsoft Windows** 的 **WordPad** 也是支援 **RTF** 的。以下是由 **WordPad** 编辑的一小段备忘录文件：



这段文字与您常使用的文字处理器或桌上出版程序，甚至在 **Web** 上观看网页的文字段落很像。

但如前所述，这段文字实际上已被 **RTF** 卷标格式化且储存成 **RTF** 檔。以下是这份文件实际加
码后的样子：

```
{\rtf1\ansi\ansicpg1252\deff0\deflang1033\deflang1033\pard\plain\fs20\b To: \plain\fs20 Jodie  
  
\f0\fs20\b From: \plain\fs20 Bill  
  
\f0\fs20\b Cc: \plain\fs20 philip  
  
What do you think of the format so far ?
```

```
\par \plain\f2\fs20\b Subject: \plain\f2\fs20 Chapter 1

\par

\par What do you think of the format so far?

\par}
```

被标注过的文字与显示出来的文字相当的不同。这段文字码告诉处理这份文件的应用程序（称为「处理器」：**processor**）有关此段文字的信息。通常整份文件中的卷标会告知应用程序所有的信息，从使用的卷标语言开始（开头可见`\rtf1` 卷标）至文字的颜色（`\colorb1\red0\green0\blue0`），甚至于每一行的开头（`\par`）皆是。在屏幕上显示的文字有一个明显的特征，即有些文字是粗体字而有些不是，这在文字码中也同样地被标注出来。请注意文字码第六行的位置，卷标**b**只出现在 **To:**之前，这卷标启动了那行文字的粗体属性，在 **To:**之后您可以看到卷标`\plain`，这时卷标便停止粗体属性，并改用一般属性，就如同没有做任何格式化的设定一般。

如您所见，在稍微了解了标签的定义之后，**RTF** 程序代码立即变得相当易懂易读。然而，对其他卷标语言来说，并非尽皆如此。如果您尝试以 **Word** 编辑并储存文件，再看看那份文件，您会发现内容是相当不同的。右图是一段以 **Word** 格式储存，并以纯文字格式观看的文件。

语言本身是封闭的，它主要的目的是确保某些文件内容的文字码是无法被读取的，就如同今日的 Word 文件一样。

您可能觉得很奇怪，我们是如何产生上面所看到 RTF 文件的程序代码？以下是简单的说明：先在 WordPad 中新增一个文件并储存它，然后用纯文字编辑软件的记事本（Notepad）将此文件打开。因为记事本是不能解译 RTF 标签的，所以只能将文件的所有数据与文字码显现出来，而不能对文字套用任何显示格式，于是我们便可以很容易地读取文件中所有的文字码啰！

上面这段说明描述了一个重要的观念：为了要让标签适当地运作，处理器（processor）必须能够读懂标示的文字码，同时也必须能解读卷标是如何影响文字进而显示结果的。WordPad 为一个可读懂 RTF 格式的处理程序，但它却不能处理其它像 HTML 的卷标语言。所以如果用 WordPad 开启 HTML 文件，您可能看到的是文字以及 HTML 卷标，而不是已经被处理过的文件内容。

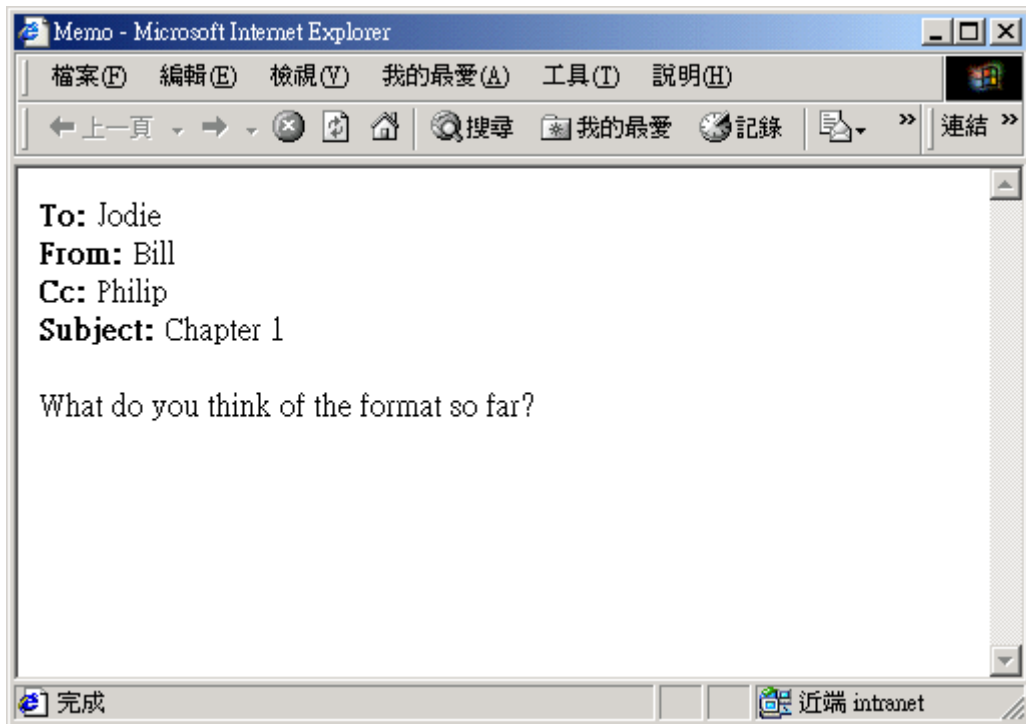
文件架构

若您仔细观察「RTF 卷标」一节 RTF 文字码的例子，您应该会注意到：想要从文字码中推导出文件的架构的确是不大可能的事。在卷标内没有任何讯息告诉您有关整合整份文件的规则。文件的作者能在文件中放置任何的文字与样式，并且可以随意地排列顺序。当然这种自由性似乎很讨人喜欢，但它同时也会产生一大堆的问题。例如：让读者去解译一份复杂文件的标签程序代码是相当困难的。就算您能解读某一部分的标签，您可能还是无法了解为何它会被放置在那里，或它与文件的其它部分是否有关连。这样松散的架构，除了作者以外，其它人似乎不可能产生相同类型的

文件。例如，若有人想要制作一段与我们一样的备忘录，他便必须对所有相关的文字码做一份完整的拷贝，并在新的备忘录中更换需要改变的部分。对小型文件而言，这是做得到的，但是处理又长又复杂的文件就变得很困难了。另一个问题是，特定的文件不易移植到其它的平台或装置上。因为那些平台或装置上并不存在文件架构的规格，所以对其他人来说，要开发一个能精确解译文件的处理器是相当困难的。如此，当其它文件已经针对既有的规格产生文字码后，我们便无法将它再做延伸了。从另一个角度来看，若是缺少了对文件建立规格的能力，就不可能从原始文件的架构创造出其它种类的新架构。

HTML 标示

您可能会认为以上所描述的并不是什么新鲜的问题，但是在解决这些问题时，您会发现问题一个比一个复杂。**HTML** 是广为接受的解决方式之一，现在让我们用 **HTML** 来讨论先前所示范的文件。



如您所见，在外观上，HTML 文件与 RTF 文件几乎相同，虽然所使用的卷标语言全然不同。以

下为 HTML 的文字码：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 FINAL//EN">

<HTML>

    <HEAD>

        <TITLE>Memo</TITLE>

    </HEAD>

    <BODY>

        <FONT FACE="Times New Roman" SIZE="2">

            <P>
```

To:

Jodie

From:

Bill

Cc:

Philip

Subject:

Chapter 1

</P>

<P>

What do you think of the format so far?

</P>

</BODY>

</HTML>

若仔细地检视这份文件的标注，您会看见已有不明示出的架构应用于此文件之上：请注意

<HTML>标签出现在开头，</HTML>出现在结尾，此文件内的所有东西皆包含在这些标签之间。

您也会注意到 **Head** 元素、**Title** 元素，以及 **Body** 元素。在文件中像这些卷标所形成的文字码有它特定的位置与目的。现在让我们看看用来作为格式化的标签！如同 RTF 一般，HTML 语言也包含粗体的格式，但请注意它使用了两个粗体标签****开启粗体格式，****结束粗体格式。

HTML 的架构并不是任意设计的。所有的 HTML 文件都必须遵守一组规则，而此规则明确地规范了文件应是如何地被整合起来。这些规则告知处理器在 HTML 卷标语言中可读取哪些元素，并辨别哪些元素可以包含在哪些元素当中，以及识别何种类型的外部档案可以放置在文件中。甚至这组规则还建立了连结至其它文件或档案的规则，这样的动作我们称之为「超级链接」

（hyperlinking）。所有的这些规则都包含在「文件类型定义」DTD（Document Type Definition）中。您可能也注意到在 HTML 文字码的开头处有个 DTD 的字样。这一行的目的即为文件类型宣告，告知处理器应使用哪种 DTD。

我们将会在 [第4章](#) 中详细地讨论 DTD。若您对 DTD 不太了解，目前您只要知道 DTD 内定义了文件的结构，就像建筑物的蓝图一样即可。

虽然每一个 HTML 文件都需要符合特定的 DTD。但实际使用 HTML 文件的应用程序，如 HTML 处理器（大部分为 Web 浏览器）并不依照 DTD 的定义检视文件，甚至不读取 DTD。因此大部分的浏览器允许 HTML 写作者不必严格遵守这些规则。例如先前显示的 Memo.htm 档案将 Title 元素放于 Head 元素之外，或将 Body 元素放于 HTML 元素之外。（其实并不建议您如此做，但若是真的这样做，大部分的浏览器仍可读取此档）当了解 XML 后，您将发现以此方式撰写 XML 文字码，将是无法运作的。

Note

在许多卷标语言中，一组开始与结束的标签中所包含的内容即是元素，例如，`<TITLE>`与`</TITLE>`都是标签。若它们被放在一起，像`<TITLE></TITLE>`，会产生「标题元素」（Title element）。大部分的元素在开始与结束卷标间包含一些内容，如`<TITLE> Memo </TITLE>`。但并不是所有的卷标语言都要求有开始与结束卷标才算是有效元素，在某些例子中，单一标签即可。

特定与通用卷标语言

目前大多使用两种卷标语言：「特定」卷标语言与「通用」卷标语言。特定卷标语言针对特定应用程序与设备，产生其文字码，这些卷标语言是为了满足特定的需求而建立的。通用卷标语言则描述文件中文字的意义与结构，但并不定义如何使用文字，这类卷标语言并不是为了满足任何特定应用程序而设计，而是为了让许多不同的、一般性的程序使用。通用卷标语言所描述的文件比起特定卷标语言描述的文件，有较佳的移植性，现在就让我们仔细检视这些观念。

特定卷标语言（Specific and Generalized Markup Language）

本章到目前为止所有的例子都在示范特定卷标语言-HTML 与 RTF 语言，它们都是为特定目的而被发展出来的。HTML 的目的为 Web 文件的格式化，而 RTF 亦有相似的目的-文字的格式化。

先前所看到的例子中：即使两段文字码为相似的目的而产生，但 HTML 与 RTF 的文字码看起来却不尽相同。您可能已经猜到，两种语言不能相互替代，能了解 RTF 的处理器并不能了解 HTML，反之亦然。

Note

即使卷标语言不能交替使用，应用程序仍可读取及显示不同种类的卷标语言。例如：若有安装正确的附加程序，Word 也可以读取及显示 RTF、HTML、纯文字、WordPerfect、Microsoft Work，与其它种类的文件。请注意：现今要求不同软件能读取与显示每一种卷标语言。当然，在一份文件内标签程序代码仍不能交替使用。例如，RTF 格式文件里应只包含 RTF 标签，否则显示出来的文件可能不如预期所料。

许多卷标语言对 Web 或打印处理文件格式化来说非常适用，然而对描述数据以及对数据提供上下文的一些讯息来说便不尽理想。让我们再看看 RTF 格式的备忘录文件例子。

```
{\rtf1\ansi\ansicpg1252\deff0\defstab720\{\fonttbl
{\f0\fswiss MS Sans Serif;}
```



```

{\f1\froman\fcharset2 Symbol;}

{\f2\froman Times New Roman;}}

{\colortb1\red0\green0\blue0;}

\deflang1033\pard\plain\f2\fs20\b To: \plain\f2\fs20 Jodie

\par \plain\f2\fs20\b From: \plain\f2\fs20 Bill

\par \plain\f2\fs20\b Cc: \plain\f2\fs20 philip

\par \plain\f2\fs20\b Subject: \plain\f2\fs20 Chapter 1

\par

\par What do you think of the format so far?

\par}

```

请注意：每个卷标描述出文字是如何被格式化的，但却没有告诉我们包含在文件内的文字数据种类为何。我们可以很容易地更换所有的文字并舍弃这些文字原是备忘录文件的事实。我们能这样做是因为——许多卷标语言是为了要做到描述文件格式以及文档版式设计的特殊目的而产生的，而不是为了其它目的，如定义某种数据结构或提供替换不兼容数据格式的方法。这样的特性，如同特定卷标语言一般，会造成许多的限制：

- 文件作者受限于特定的标签组。若此组标签不能符合需要，文件作者就必须找到变通的方法，不然只好顺应这样的限制了。

- 文件可能无法移植到其它应用程序而被读取。因为数据不具备自我描述的能力，

所以文件不能因应任何其它的目的而被使用，就算文件尝试做这样的改变。

- 语言具有不兼容于其它卷标语言文件的特性。这样对文件作者来说，需要使用许

多的语言以适应不同的应用，这将会造成困惑与许多额外的工作。

Note

自我描述文件（**Self-describing document** 在稍后几章将会介绍）基本上提供有关文件中数据的讯息，如此，文件中的数据可以跳脱描述文件如何被显示的格式问题。例如，文件中可能包含数字形式的讯息，自我描述文件可能识别数字为年龄——为树的年龄，而树为重新造林计划的一部分，如此等等。

当电子文件在讯息传递上的影响越来越大时，很明显地这些许多限制将造成问题，并让人退却，这也助长了一般化卷标语言的使用。

一般化卷标语言（**Generalized Markup Languages**）

1970 年代 C.F Goldfarb 博士（原为一位律师，后来到 IBM 工作）与他的两位同事提出一种不限于某一种应用或设备的文字描述方法。此方法有两个基本的部分：

- 标签应该能描述文件的架构，而不是只有描述它的格式与类型特征。
- 应严格规定卷标的语法，如此文字码才能被应用程序或人们清楚的了解。

基于这些建议，IBM 发展出「一般化卷标语言」（Document Composition Facility Generalized Markup Language），简称 DCF GML 或 GML。GML 为「标准一般化卷标语言」（Standard Generalized Markup Language，SGML）的前身，而 SGML 于 1986 被国际标准组织协会（International Organization for Standardization，ISO）所采纳成为一个标准。

Note

ISO 于 1970 年成立，包含约 130 个会员国。ISO 存在的目的主要在「以促进国际货物与服务观点上，提升全世界标准化与相关活动，并促进智慧、科学、科技，以及经济各方面活动的合作。」ISO 因此产生一组通行全世界的标准，其标准化影响的范围从通讯、农业到服务业。您常看到一些有关标准数据都参照着 ISO 号码（例如 ISO 8879 代表 SGML 标准）。有关 ISO 的信息请参照 <http://www.iso.ch/welcome.html>。

SGML 标准对文件标示带来了重要的改变：除了提供规划文件架构的方法外，SGML 附加了：

- 识别文件中使用到的字符。藉由允许文件指定使用何组字符集（例如，ISO 646 或 ISO 8859）来确保处理器能了解文件内所有的内容。
- 提供辨识文件中对象的方法。当片段文字或其它数据出现在文件多处时，这些对象（称之为「实体（entity）」）便派上用场。在文件某处宣告实体后，任何对此宣告的改变将反应至整个文件中有此一实体的地方。（实体将在第 3、4 章讨论）
- 提供将外部数据整合到文件内的方法。这允许数据可以不是文件内使用的文字。

现在让我们来看看，我们的备忘录文件如何以 SGML 文件显现。

```
<!DOCTYPE MEMO PUBLIC "-//BJP//DTD MEMO//EN">

<MEMO>

  <TO>Jodie

  <FROM>Bill

  <CC>Philip

  <SUBJECT>Chapter 1
```

```
<BODY>What do you think of the format so far?  
</MEMO>
```

请仔细检视这项文字码，您会发现有些元素与我们先前在标示中所提过的很相像——当然，您也会看到一些不同点。首先先来看看相似的地方。

这份文件跟<HTML 标签>一节中 HTML 版本的备忘录文件很相似，若您回头看看那个 HTML 版本，您会看到在文件顶端有类似的 DTD 宣告，而且也有相似的标签格式。例如，备忘录元素包含开始卷标与结束卷标。文件的大部分内容（介于卷标之间的文字）与 HTML 文件的相同。

其理由为：主要是因为 HTML 为 SGML 的一项应用，所以才会存在这些相似点，也就是说 HTML 是使用 SGML 标准而产生。也是因为如此，许多 SGML 细部资料也被带到 HTML 之内，但并不是所有细部都包括到。现在让我们看看两个版本的不同处。

首先请注意：许多 SGML 元素不包括结束卷标。这些结束标签随您选择，而且可以很容易地被包含到任何元素当中。例如，我们可以加个结束卷标到 BODY 元素后而不改变 SGML 文字码的意义。

```
<!DOCTYPE MEMO PUBLIC "-//BJP//DTD MEMO//EN">  
  
<MEMO>  
  
  <TO>Jodie  
  
  <FROM>Bill  
  
  <CC>Philip
```

```
<SUBJECT>Chapter 1

<BODY>What do you think of the format so far?

</MEMO>
```

HTML 也提供「最小化技术」（minimization technique）。现在对您来说也许还不是很重要，但是稍后当我们在讨论 XML 时，您将发现它的重要性。

SGML 与 HTML 之间最大的不同点是：在 SGML 文件中，没有讯息指出数据应该看起来长怎样，但标签却可以辨别出文件的架构。请注意：某些内容已被移除，特别是地址讯息（To:、 From: 等等）。

由于地址讯息现在已变成文件结构的一部分，所以这样做是安全的。事实上，DTD 描绘了什么类型的元素可存在于什么类型的文件中、它们能出现在何处，以及它们能包含何种数据...这些规则架构。处理器基于这些架构与文章内容来读取文件，并以适当的方式输出数据。

卷标语言的概观

如果这些观念对您来说是全新的，那么以上的讨论可能已对您造成一些困惑。但是至少您要了解，SGML 版本备忘录内的标签是特别为备忘录文件而设计的。如此做是可行的，因为 SGML 具扩充性，这意味着 SGML 允许文件作者藉由定义适合的标签，而自行定义出特定的结构（就如同备忘录文件架构）。您应该知道 HTML 是 SGML 的一项应用，也就是说有一群人为了在 World Wide Web 上应用，而利用 SGML 提出一个全新的语言。

HTML 语言不具扩充性，也就是说不能以 HTML 为基础，发展出另一种拥有自己规则与目标的语言。而因为 SGML 允许产生其它的语言，所以称 SGML 为卷标语言有点不恰当，我们应该称之为「中继语言」（metalanguage）。

现在，您可能会怀疑：SGML 与本书要讨论的主题 XML 有什么关系。XML 是直接由 SGML 衍生出来的。但它不像 HTML 语言，XML 并不是一种新的语言，它是 SGML 的一部分集合，一样可以作为中继语言。事实上，XML 能做的，SGML 都能做到，甚至 SGML 能做的更多。但如果这是事实，那我们为什么需要 XML 呢？在下一章中，我们将继续讨论这个问题。

2. 进入主题—XML

在 [第一章](#) 中，我们已经讨论了有关卷标语言的基本知识。本章我们将讨论为何要发展 XML，而且您将学会如何把 XML 与 SGML、HTML 整合在一起。本章也会讨论到 XML 语言的目标，并检视由这些标准团体所制定完成的内容。我保证：在本章结束之前，我们会实际运用到 XML 文字码。

什么是 XML？

如 [第一章](#) 所提到的，XML 是从 SGML 推演出来的，但是不像 HTML。XML 并不是 SGML 的一种应用，而是它的子集。XML 也是一种中继语言，并且在许多方面都与 SGML 雷同。也就是说，在 XML 中能发展出其它的语言与语汇（[第 5 章](#) 在语汇方面将多做介绍）。如 [第一章](#) 所提到的，任何用 XML 完成的工作同样也可以用 SGML 来完成。若是如此，为何还需要 XML 呢？

XML 实例

因为 XML 能有效地在 World Wide Web 上应用，所以 XML 带来了在 SGML 中所没有的优点。

XML 能与 HTML 整合来描述与显示数据，所以对于在 Web 上传递数据，XML 提供了几个超越

SGML 的优点：

- XML 的规模比 SGML 小。XML 的设计者尝试在 SGML 中去除所有在 Web 上传递数据所不需要的部分，结果产生了 XML 这个较简单与缩减的语言。（规格书的厚度证明了这一点：基本的 SGML 规格书约有 155 页之多，然而 XML 的规格书却只有 35 页。）
- XML 包含超级链接结构的规格，它被描述为一种独立的语言，称为可延伸连结语言（Extensible Linking Language，简称 XLL）。XML 不仅支持在 HTML 中的基本超级链接，并且更进一步应用延伸连结的概念（第 8 章将详细介绍延伸连结）。虽然，SGML 允许定义超级链接机制，但它并没有将超级链接纳入初版的规格书中。
- XML 包含样式表语言的规格，此一样式表语言称为可延伸样式表语言（Extensible Stylesheet Language，简称 XSL）。XSL 提供支持样式表机制，并提供一些在 SGML 中所没有的支持。样式表允许文件作者建立各种样式（如：粗体及斜体等等）的样版，或者结合各种样式并将它们应用到文件的元素中。

对 XML 家族来说，XLL 与 XSL 是两项功能强大的附加语言。我们会在 [第 7 章](#) 详细讨论 XLL，而 XSL 将于 [第 8 章](#) 讨论。

简单来说，XML 拥有 SGML 百分之八十的功能，但其复杂度却只有 SGML 的百分之二十。

XML 与数据的关系

如果说 HTML 是用来显示信息的，那 XML 即是关于信息的描述。XML 是一种标准的语言，它被用来描述数据并将之结构化，且可让不同的应用程序了解这些数据的内容。XML 的功能在于能够将数据与使用者接口分离。现在，让我们改写第一章备忘录范例的文字码，看它是如何运作的。这份新的 XML 文字码显示如下：

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

<MEMO>

  <TO>Jodie</TO>

  <FROM>Bill</FROM>

  <CC>Philip</CC>

  <SUBJECT>Chapter 2</SUBJECT>

  <BODY>This is where we start getting into some XML code!</BODY>

</MEMO>
```

在上述的文字码中，除了每一个元素多了结束卷标外，您应该注意到这段文字码类似于第一章中 SGML 版本的文字码。请注意，没有任何信息指出数据应该如何显示。换句话说，这里没有格式化的信息（如粗体、斜体字型、内缩与字号等）在里面，然而，大部分文件的文字码在描述数据是什么。普通读者很容易看的懂这些文字码，并了解文件的内容及架构。

XML 文件被认为具有自我描述的能力，也就是说，每个文件包含一组规则，文件中的数据都必须遵从这些规则。因为任何一组规则皆可在其它文件中重复使用，如果需要的话，其它的开发者可以很容易地创造出相同的文件类别。

Note

我们将在 [第四章](#) 中讨论文件类别，类别的观念来自对象导向程序设计。在对象导向程序设计中，每个类别被用来描述一群拥有一组共同特征的对象。文件类别是一种有效率的文件分类方法，根据文件的内容作为分类的依据。

其它使用 XML 与数据整合的方式如下：

- 使用 XML 来当作一种数据转换格式。许多使用一段时间的系统，我们称之为「legacy system」，它包含不同格式的数据，而且需要开发人员投入大量心力，利用因特网来连结这些系统。他们所面临的挑战之一：即是要能转换系统间原本

不兼容的数据。**XML** 可以解决这些问题。自从 **XML** 的文字格式成为标准格式后（意味许多应用程序都能了解这种文字格式），数据可以转换成 **XML**，然后让它系统或应用程序容易了解。

- 使用 **XML** 处理 **Web** 数据。假设有一个 **HTML** 网页，且其中不含任何内容。现在取而代之，将内容储存于 **XML** 档案中，而 **HTML** 网页的目的只为了简单地格式化与显示。**XML** 内容可被修改与转译成另一种语言，倘若不是文件作者本人来修改，就可能需要接触到 **HTML** 文字码。
- 使用 **XML** 来为信息建立共同的数据储存方式，让数据可以用不同的方式被取得，**XML** 创造出一种共同的数据储存方式。假设您正为杂志社写一篇文章，出版商也想将这篇文章放到网站上，并将它放到一本书或期刊中。假若原本的文章以作者自己的格式创作，如 **RTF** 格式，为了要在 **Web** 上刊登，这篇文章需要重新处理，并且又为了因应书本与期刊的出版，亦需再做处理。假若文件以 **XML** 格式写作，他可同时在三种不同的环境中处理，因为文章数据与如何被显示是没有关系的。数据的格式化、输出等是视使用数据的应用程序而定，而不是属于内容本身。更进一步来说，显示数据的应用程序只需重新撰写一次，即可显示任何文章。

您将会发现：对于编写与储存数据来说，XML 是一个非常强而有效的方式，它不仅能应用在 Web 上，同时也能有效地应用在其它方面。

XML 适用于何处？

您已经知道 XML 为 SGML 的子集，并且也知道 XML 并不是要取代 HTML，而是补足它。在这一节，我们将仔细看看，在这两种相近的语言间，到底 XML 卷标语言在哪方面具有相辅相成的效果。

XML 与 SGML 之间的关系

SGML 应用在出版业已经很多年了。尽管 HTML 为 SGML 的一种应用，而且也成为网络出版业的标准，但 SGML 本身却从未得到网络发展族群的重视，主要是由于 SGML 的复杂性造成它不易被使用的结果。

1996 年 8 月，一个来自 World Wide Web Consortium (W3C) 里的组织，命名为 SGML Editorial Review Board (SGML ERB)，但这组织最后变成 XML 工作小组。这个组织的主要目标是：建立一个通用的卷标语言，且其必须容易在 Web 上执行。SGML ERB 与 SGML 工作小组合作，后来变成 XML Special Interest Group。他们决定了 SGML 的哪些部分可用来建立 XML 卷标语言。

XML 工作小组的重点在于：将网络出版所需要使用的 SGML 纳入 XML 中。结果，XML 的确去除了 SGML 中许多复杂的部分，以及所有在 SGML 中证实是多余的部分。如此，XML 保留了 SGML 的主要的优点：

- XML 为通用的卷标语言。开发者能定义自己的标签集（tag set）。
- 文件具有自我描述能力。一份有效文件包含特定文件类别的所有规则。
- 文件的有效性可被确认。藉由使用文件类型定义（DTD）中所发现的规则，处理器可验证文件是根据规则所开发的。

Note

您可能会怀疑为什么移除 SGML 中所有选择性的功能（optional features）是有益处的。一般来说，使用选择性的功能将会增加文件与应用程序不兼容的可能性。也就是说，使用某一组功能选项来设计处理 SGML 文件的应用程序，将不能处理使用其它功能选项的 SGML 文件。XML 的制定者希望减少或消除因选项所产生的不兼容问题。

再次强调 XML 为 SGML 的子集合是很重要的，任何有效的 XML 文件亦是有效的 SGML 文件，您可以把 XML 看作是成功减肥后的 SGML。

XML 与 HTML 的关系

许多人认为 XML 为 HTML 的替代品。这种说法可能只有少部分是真实的，其实这两种语言是彼此互补而不是彼此竞争的。事实上，从处理数据的角度来看，这两种语言操作数据的层级是不同的。例如，XML 是用来在 Web 上结构化与描述数据用的；HTML 则比较像是用来格式化数据的。因为大部分 HTML 网页的文字码不但储存数据，也同时将数据格式化，任何对 HTML 的置换都将在数据储存区发生。在许多情况下，XML 可以接管资料储存的工作（还有资料描述的工作），如此 HTML 只被用来作数据的格式化与脚本。

检视 XML 程序代码

我曾经承诺过在这章结束前讨论一些 XML 文字码的实作，现在就让我们开始吧！这个实作用来帮助您清楚地了解 XML 与 HTML 如何一同运作，让我们来看看如何使用它们在 Web 上显示资料。这个例子引用两份文件：数据的 XML 文件以及数据格式化与显示的 HTML 文件，而藉由修改本章先前备忘录文件的文字码开始。

如同所有的程序设计书一样，我们的第一个例子将会显示「Hello World!」。此份 XML 文件可在随书光盘 Chap02\Lst2_1.xml 档案中找到，现在就来看看文字码 2-1：


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

<MEMO>

    <TO>Jodie</TO>

    <FROM>Bill</FROM>

    <CC>Philip</CC>

    <SUBJECT>My first XML document</SUBJECT>

    <BODY>Hello, World!</BODY>

</MEMO>
```

文字码 2-1

接下来我们将制作一个 **HTML** 网页来显示备忘录内的资料。如此，我们需要将 **XML** 处理器加到

网页中并撰写一些 **Script** 来读取资料并且以我们想要的格式显示。您可以在随书光盘

Chap\Lst2_2.htm 档案中找到这段 **HTML** 文字码，以下是文字码 2-2：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

    <HEAD>

        <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

            loadDoc ( ) ;

        </SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">

var rootElem;

var xmlDoc=new ActiveXObject ("microsoft.xmlDOM") ;

xmlDoc.load ("Lst2_1.xml")


function loadDoc ()

{

if (xmlDoc.readyState == "4")

start () ;

else

window.setTimeout ("loadDoc () ", 4000)

}

function start ()

{

rootElem = xmlDoc.documentElement;

todata.innerText = rootElem.childNodes.item (0) .text;

fromdata.innerText = rootElem.childNodes.item (1) .text;

ccdata.innerText = rootElem.childNodes.item (2) .text;

subjectdata.innerText = rootElem.childNodes.item (3) .text;

bodydata.innerText = rootElem.childNodes.item (4) .text;
```

```
    }

</SCRIPT>

<TITLE>Code Listing 2-2</TITLE>

</HEAD>

<BODY>

  <DIV ID = "to" STYLE = "font-weight:bold;font-size:16">

    To:

    <SPAN ID ="todata" STYLE ="font-weight:normal"></SPAN>

  </DIV>

  <BR>

  <DIV ID = "from" STYLE = "font-weight:bold;font-size:16">

    From:

    <SPAN ID ="fromdata" STYLE ="font-weight:normal"></SPAN>

  </DIV>

  <BR>

  <DIV ID = "cc" STYLE = "font-weight:bold;font-size:16">

    Cc:

    <SPAN ID ="ccdata" STYLE ="font-weight:normal"></SPAN>

  </DIV>
```

```
<BR>

<DIV ID = "subject" STYLE = "font-weight:bold;font-size:16">

    Subject:

    <SPAN ID ="subjectdata" STYLE ="font-weight:normal"></SPAN>

</DIV>

<BR>

<HR>

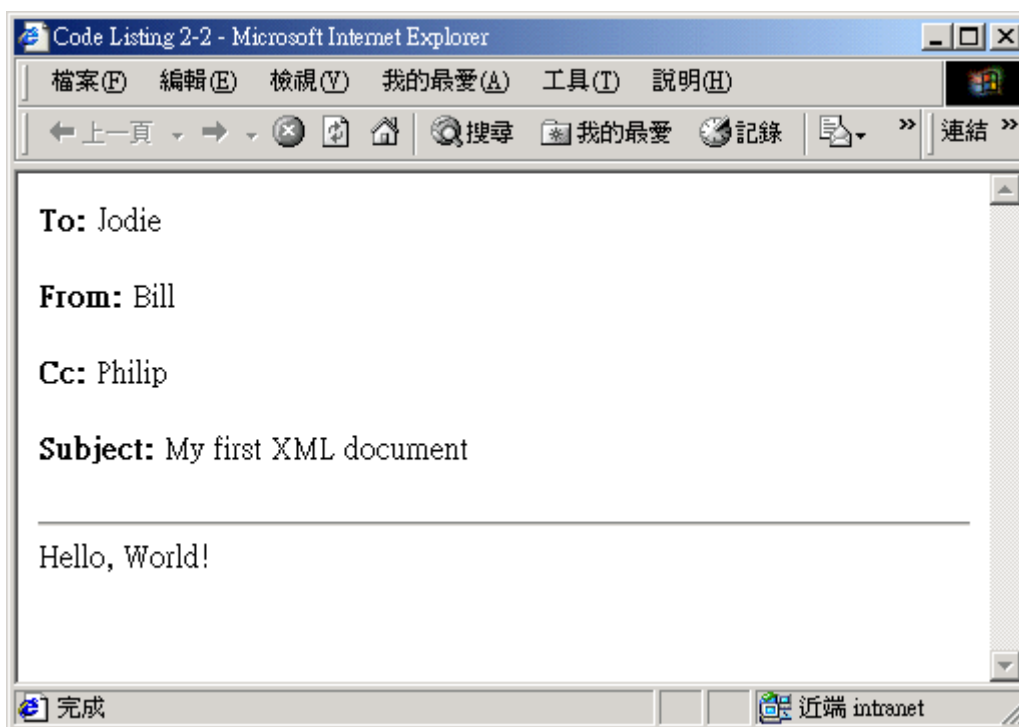
<SPAN ID = "bodydata" STYLE ="font-weight:normal"></SPAN>

</BODY>

</HTML>
```

文字码 2-2

当您执行 **Lst2_2.htm** 的程序后，执行结果应如下图所示：



保持数据与格式化的独立

您可能认为只为了展示这小小的备忘录竟要花那么多文字码！虽然在这个例子中使用 XML 并不太适合，但这个例子主要在告诉您：如何将内容或数据从格式化中分离出来。所以假若您想改变备忘录的内容，便只需要去改变 XML 文字码，而不需要在 HTML 文字码中寻找您想要改变的内容。我们将不深入探讨这段文字码，因为稍后几章将会有更深入的说明。您可以很容易地看到为何 XML 会成为一个强而有力的数据储存机制，您甚至不需了解任何 HTML 文字码。要改变备忘录，只需遵循下列几个步骤：

1. 利用文字编辑器，如 Notepad，打开 Lst2_1.xml 档案，并且改变 XML 元素的内容，小心不要改到卷标，例如，TO 元素可改成<TO>Yoko</TO>。可以随您高兴改变元素的内容。
2. 储存改变后的 XML 文件并关闭文件。
3. 开启 Lst2_2.htm 档案。

Note

如果您没有执行 Setup 来安装范例，而是直接从光盘中拷贝 Lst2_1.xml 与 Lst2_2.htm 档案，则您必须将 XML 文件的只读属性取消，如此才能改变文件内容。于 [档案管理员](#) 中对 Lst2_1.xml 档案按鼠标右键并且从功能选单中选择 [属性](#)。在 Lst2_1.xml 的 [属性](#) 对话框中，取消 [只读](#) 复选框。

您所作的改变将在 HTML 文件的显示中反应出来。请注意即使数据不同，格式仍与原本的 XML 文件相同。您甚至可以使用不同语言来输入 XML 的内容，如德文，而不需要改变任何 HTML 文字码，它仍然可以依您想要的格式显示数据。这是一个教您如何使用 XML 架构与储存数据的简单例子，稍后几章，您将会看到更多的例子，这只是牛刀小试罢了。

XML 与 HTML 相互配合

如前面所述之例，HTML 与 XML 可以整合的很好。因为所有的备忘录内容与 HTML 文字码分隔开来，内容可以很容易地被修改与使用。HTML 文件可以当作输出模板，用来制作不同数据的备忘录文件。有一点要注意的是：因为 HTML 文字码不须再储存任何文件的资料，所以 XML 可以取代部分的 HTML 文字码。不过，XML 文字码并不能替代所有的 HTML 文字码。

Note

稍后几章中，将看到更多 XML 与 HTML 合作的例子。您甚至可以将 XML 数据片段嵌入 HTML 文件内，当作「XML 数据岛（XML data islands）」。一般来说，格式化由 HTML 负责处理，不过数据必须确保为严谨的架构及为 XML 的处理规则。

很明显地，XML 的目的并不在取代 HTML，所以让我们再来看看，XML 的制定者发展此语言时，所期望的目标是什么。

XML 的目标

XML 是为了能有效地在 Web 中运作而设计的。此目标虽为设计的主因，但 XML 仍能在 Web 以外的环境中运作，包括出版业，数据交换所以及商业应用当中。为了能让 XML 广泛地在不同的环境中应用，XML 的设计人员知道 XML 必须要简单、强而有力并且容易让不同类型使用者制作文件。

XML 的目标

为了更了解 XML 尝试做到哪些功能，让我们来看看 XML 制定人员为这个语言订定了哪些目标。

目标一：XML 要能在因特网中直接使用。首先，XML 必须能在因特网中有效的运作，而且必须考虑到在分布式网络环境中执行程序的需要。这并不意味着 XML 必须立即加入目前 Web 应用程序当中，最重要的应是能在因特网中完善的运作。

另一要点是关于「直接」这两个字。SGML 的使用对某些开发者来说太过复杂；对于要让客户端在网络的工作环境中有效率的执行来说，SGML 的架构又太过复杂繁琐。因为 XML 去除 SGML 中多余的部分，只留下必要的，所以大部分在 SGML 中会造成问题的复杂性与额外的负担都被移除了。请注意：这个目标并不是要限制 XML 于因特网上的应用，而是要带领我们到另一个目标。

目标二：XML 应广泛支持不同种类的应用程序。这个目标规定了 XML 必须广泛地被应用于任何领域，如文件制作工具（authoring tools）、内容显示引擎（content display engines）、翻译

工具甚至数据库应用。XML 制定人员了解：若要 XML 快速地被采纳，有赖于软件应用程序能有效地使用 XML。

Note

现今许多软件产品都支持 XML，请查看 http://www.gca.org/conf/xml/xml_what.htm#xml-soft。

最终，大部分广受欢迎的文字处理器或其它数据软件将支持 XML。

目标三：XML 应与 SGML 相容。在 XML 的设计中，这是个极困难的目标。但是，这也是属于众多麻烦目标中的一项而已。这个想法是说—任何有效的 XML 文件亦为有效的 SGML 文件。建立这样的目标使得现存的 SGML 使用工具能处理并解析 XML 文字码。请注意：有效的 XML 相对地也是有效的 SGML 文件，但反之则不一定成立。请记得：XML 是由去除 SGML 中不重要的部分而建立的卷标语言，所以使用 XML 处理器解析 SGML 会失败而产生错误。

Note

什么是解析（parsing）？要解析 XML 或 SGML 文件的语意，处理器必须将文件分割成基本构成部分，并且了解文件架构与基本构成部分的关系。XML（以及 SGML）应是可解析的文件，

因为它们被要求遵循严谨的规则（通常存在于 DTD 中），并允许任何了解这些规则的处理器能正确地解译文件。即使 HTML 是 SGML 的一项应用，但它通常并无法真正地可被解析。因为大部分的处理器并不能严格的实行这些规定，而且大部分文件作者亦不能严谨地遵循这样的规则。

目标四：处理 XML 文件的程序应该很容易撰写。隐藏在此目标背后的想法是：语言被采用的程度将与工具的可用性成正比。何谓「容易」当然是相对的定义。其实原本的构想为具备计算机科学学位的开发者能在一至二个星期内，撰写出基本的 XML 处理器。这两个星期的基准目标已不再被采用了，主要是因为 XML 工具数目激增，而且大部分为免费软件，由此我们可以印证此目标确实已达到了！

目标五：XML 中选择性的功能数量应保持在最少数目，但理想情况应为零。此目标来自存在于 SGML 中的问题：SGML 规格书中包含许多功能选项（很多功能选项从未使用过），这些增加了 SGML 处理器额外的负担，并且增加文件与处理器间兼容的困难度。例如，应用程序被设计用来读取与处理指定选项的 SGML 文件，如此应用程序便不能正确地解析使用其它功能选项的文件。XML 藉由减少选项的数量至零来避免不兼容的潜在问题。这意味着，任何 XML 处理器应能分析任何 XML 文件，不论文件本身包含哪些数据或架构。

目标六：XML 文件必须易读而且相当清楚。此目标包含了理想与实际的目的。自从 XML 使用纯文字描述数据与数据间的关系后，它比使用二进制格式更易于处理与读取。

自从文字码使用简单易懂的方式格式化后，使得机器与人都能很容易地了解 XML。从实际的立场来看，如果 XML 是如此容易了解与使用，您就不需要复杂的工具与程序来执行 XML。所以，无须很复杂的工具，任何文件作者只要利用简单文字编辑器即可撰写 XML 文字码。

目标七：XML 文件应可被快速地制作。正如前面已经讨论过的，因为 Web 需要一个具有延伸性的语言，所以有了 XML 的构想。而这一个目标也就自然地被纳入了。若是 XML 不能快速而有效地延伸 HTML，其它的组织则可能提出更适当的解决方法。SGML ERB 相信延伸性的解决方法需要从 SGML 中寻求。制定人员亦确定应开放正确解决方案的架构与扩充能力，并且不能只让单一软件业者所专有。

目标八：XML 的制定应力求严谨与简明。此目标的重点在于 XML 规格书。这个目标是想藉由规格书中正式的语法，使它们尽可能地简单明了。为了要达成这个目标，规格书使用了 Extended Backus-Naur Form (EBNF)，为一个用来描述程序语言的标准格式，包含宣告。为了避免单调，在整个使用 EBNF 的过程中，尽可能保持严谨与简单明了。假若 XML 语言容易了解与使用，那此目标与第四个目标即代表此语言将更容易被接受。

目标九：XML 文件应易于产生。正如第六项目标所陈述的，XML 文件应该让读者更容易阅读与了解。尽管 XML 文件能用某些简单的纯文字编辑器创造出来，但实际上，复杂的文件有时证明太过笨重而无法在这样的环境下执行。这将有赖于市场来决定这样的目标是否符合，不过很多建立与使用 XML 的工具（商业的或免费的软件）已是很容易取得了，这表示 W3C 离达成这个目标不远了。

目标十：XML 标签的简洁对最小化是重要的。此目标是要弥补 SGML 文件中的问题。SGML 支持最小化技术（Minimization Techniques），简单说即是：SGML 允许一些快捷方式，以减少文件作者打字的数量。众所皆知的，SGML 最小化同样也是 HTML 的一部分，亦即是忽略许多元素的结束卷标。这样的语言，在下一个开始标签出现时，能够发出信号告知前面一个元素应该结束。虽然这可降低文件作者的工作量，但却造成读者困惑。在 XML 中，清楚应是优先于简洁的。

目标的实践

上述十项目标驱策了 XML 规格书的发展。我们可以看见规格书风格与 XML 语言的许多特性都是基于这些目标所制定的。您将会发现：XML 是一个简单、强而有力，且极具有弹性的卷标语言。

XML、推荐书，与标准

本节将讨论目前 XML 规格书的状态，以及其它由 W3C 正在检视与发展中的 XML 相关的规格书。

XML 1.0 版

XML1.0 规格书是 W3C 目前公认的推荐书。这意味着 W3C 成员已达成共识及规格书将可稳定、广泛地被使用。然而，这并不代表此规格书将被采纳成为标准。「建立推荐书 ...此程序是 W3C 会员所同意的标准可选择程序，而不是取代程序或修正程序。在这种情况下，修正推荐书是必要的，而相同的程序也将发生在再版推荐书上。」尽管 W3C 推荐书可送交给像是 ISO 的标准组织审核，但这个程序不是必要的。

Note

更多有关 W3C 活动的讯息，请查阅 <http://www.w3.org/Consortium/Process/>。此网页描述整个 W3C 活动程序的大纲，从提案到完成正式的推荐书。

XML 规格书并不是 W3C 在 XML 方面唯一完成的工作。许多其它初步的工作，如 XML 的应用与语言本身的加强工作，都在其工作范围内。

数学卷标语言（Mathematical Markup Language）

当作者在撰写本书的同时，数学卷标语言（MathML）1.0 已经由 W3C 推荐发行。数学卷标语言（MathML）是一种 XML 的应用，被设计来使 XML 更容易在 Web 上使用数学公式与科学数据。此一中继语言将可应用于科学、数学、工程与医学领域。更多相关的信息，可参阅 MathML1.0

规格网站 <http://www.w3.org/TR/REC-MathML/> 以及 W3C 算术活动网

页 <http://www.w3.org/Math/Activity.html> 。

资源描述架构（Resource Description Framework）

资源描述架构（RDF）为数据模块化架构，它是处理中继数据（**metadata**）的基础。资源描述

架构（RDF）使用 XML 做为它的编码语法，虽然还有其它方式来描绘 RDF 模型。RDF 模型与

语法规格仍处于 W3C 推荐的草案阶段，若需要更多的讯息请参

阅 <http://www.w3.org/TR/WD-rdf-syntax/>；对 RDF 的描述，请参阅 W3C 网站对 RDF 的介

绍 <http://www.w3.org/RDF>。

XML 连结语言（XML Linking Language）

XML 连结语言（简称 XLink）为 XML 的一种应用，它定义一种在 XML 文件中强而有力的连结

机制。除了提供 HTML 中惯用的及简单的连结使用方法外，XLink 包含一种可延伸的连结机制：

- 能连结到更多资源。
- 能让文件中不能包含传统行内连结的文字也能有连结能力。

- 帮助「smart」连结能动态地被应用，并以内容与上下文为基础过滤连结。
- 能让原本不支持连结的数据格式中也能有连结能力，例如：影像。更多的讯息请参阅 XLink 工作草稿 <http://www.w3.org/TR/WD-xlink>。

同步多媒体整合语言（Synchronized Multimedia Integration Language）

同步多媒体整合语言（简称 SMIL）为一种 XML 应用。SMIL 试着能够在 Web 上使用如同电视般（TV-like）的多媒体。SMIL 的好处是允许文件作者不需要转换内容的格式（如：视频影像），而且不需要使用复杂的 Script 语言，来创造花俏、同步的多媒体。在 [第八章](#) 将对 SMIL 做深入的探讨。当笔者写到这里，SMIL 1.0 才刚由 W3C 推荐书发行。您在 <http://www.w3.org/TR/REC-smil/> 可以看到目前 SMIL 的规格。

可延伸的样式表语言（Extensible Stylesheet Language）

可延伸样式表语言（简称 XSL）为一种文字格式化（text-formatting）语言，具有弹性与延伸性。XSL 拾取了串接样式表（Cascading Style Sheet，简称 CSS）所遗漏的部分，并加入这些特性，

来允许包含在样式表内的文字码能执行复杂的格式化工作。目前 W3C 的 XSL 规格仅为草稿。

若想获得更多的信息，请参阅 W3C XSL 网页：<http://www.w3.org/TR/NOTE-XSL.html>。

Part 2 XML 基础入门

3. XML 的结构及基本语法

- . XML 文件的结构
- . XML 语法
- . 有效的 (Valid) 与格式正确的 (Well-Formed) XML

4. DTD 的规则

- . 文件类别 (Document Classes)
- . 文件类型定义 (Document Type Definition, DTD)

3. XML 的结构及基本语法

正如您在上一章中所学到的，XML 的制定人员在制定这个语言时，心中已有特定的目标。为了要达成这些目标，他们制定了相当严格的结构和语法规则。虽然刚开始时，这些规则会使 XML 看起来似乎比 HTML 更琐碎，但稍后您将会发现，XML 所具有的可延伸性和协同合作的能力都是 HTML 所无法比拟的。

Note

可延伸性计算机语言允许开发者延伸或修改它的语法及语意。协同合作的能力则是指应用程序间可互相分享数据。

在本章中，我们将要检视 XML 文件的逻辑、实体结构，以及 XML 语言是如何建立起这些结构。同时我们也要学习 XML 语言的基本语法，并且将它与 HTML 的语法做一个比较。当然我们也将建立与讨论一个简单的 XML 文件来展示这些原则。

XML 文件的结构

XML 最佳的优点之一是：它能提供一个文件的架构。每一个 XML 文件都包含了逻辑结构和实体结构。逻辑结构就像一个样本，告诉您在这个文件中包含那些元素与其顺序。而实体结构则包含

文件中使用的实际数据，这些数据可能是储存在您计算机内存中的文字，也可能是在 World Wide Web 上的一个图形档案等等。想要更清楚的了解 XML 的结构，请参照右图的文件模型。

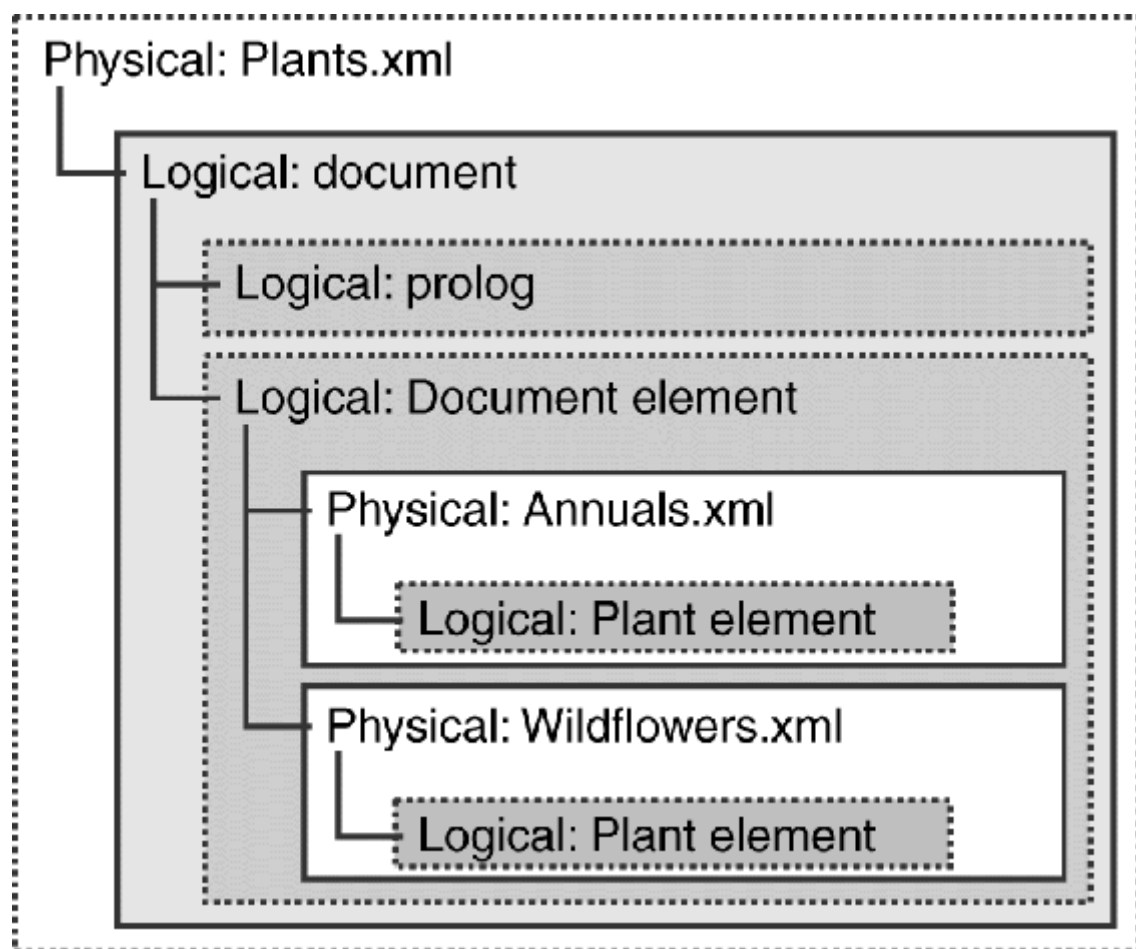


图 3-1: XML 文件的逻辑结构及实体结构

XML 中的逻辑结构

如果您很熟悉 HTML，您可能对逻辑结构的概念已有基本的了解。逻辑结构是文件系统中各个不同部分的组织结构。相对于文件的内容是什么，逻辑结构则代表文件是如何被建立起来的。而 HTML 文件的逻辑结构，代表被标记在文件中的元素，如下列的文字码所示：

```
<HTML>

  <HEAD>

    <TITLE>Title content goes here</TITLE>

    Other Head content goes here

  </HEAD>

  <BODY>

    Body content goes here

  </BODY>

</HTML>
```

这段文字码中的 HEAD、TITLE，及 BODY 等元素都包含在 HTML 元素中；而 TITLE 元素则在 HEAD 元素内。这个 HTML3.2 文件类型定义（DTD）提供了一个如何建立一份 HTML 文件的完整规则参考，您可在下列网址中找到相关参考文件 <http://www.w3.org/TR/REC-html32.html#dtd>。

Note

一份 XML 文件是由宣告 (Declaration)、元素 (element)、处理指令 (processing instructions), 以及批注 (comments) 所组成的。其中一些组件是选择性的, 而另一些则是必要的。在本节中要由建立的范例文件来检视 XML 文件的基本宣告及元素, 在下一节中还要继续讨论其它的组件。

PROLOG 元素

在 XML 文件中, 第一个结构性的元素是选择性的前言 (prolog), 前言由两个基本组件构成:

XML 宣告与文件类型宣告, 它们也都是选择性的组件。

XML 宣告

XML 宣告中定义了文件遵守的 XML 规格的版本。虽然它是选择性的元素, 但您还是应该在您的

XML 文件中将它包含进去。这个范例文件就是由基本的 XML 宣告开始的。

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

Note

上面的这行文字码必须使用小写字母。

XML 宣告中同时也包含了编码宣告和单一文件的宣告。这个编码宣告是用来辨识字符编码系统，像是 UTF-8 或 EUC-JP。不同的编码系统会对应到不同的字符格式或语言。例如预设的 UTF-8 包含了绝大部分英文的字符。XML 解析器被要求必须要支持 Unicode 编码标准，才能够支持大部分的语言。

单一文件宣告是用来辨识 XML 文件是否有使用外部的标签宣告（稍后会加以讨论）。单一文件宣告可以有一个 yes 或 no 的值。

Note

若是想对编码宣告或单一文件宣告有更进一步的了解,请参阅随书光盘中<< XML1.0 >>规格书的 2.8、2.9, 和 4.3.3 等节。

如果读者会使用到中文的内容，则必须在后面加上中文编码的宣告：

```
<?xml version = "1.0" encoding = "Big5"?>
```

文件类型宣告

文件类型宣告由标签码组成，对特定的文件类型指示文法规则或文件类型宣告（DTD）等。文件类型宣告也可以指向一个包含所有或部分 DTD 外部的档案。文件类型宣告必须在 XML 宣告之后与其它文件元素之前出现，下面这段文字码示范如何在一个 XML 文件中加入文件类型宣告：

```
<?xml version = "1.0"?>  
  
<!DOCTYPE Wildflowers system "Wldflr.dtd">
```

上面这段叙述告诉 XML 处理器，这份文件中有一个 **Wildflowers** 类别，并且遵守在命名为 **Wldflr.dtd** 的 DTD 档案中定义的规则。（[第 4 章](#) 将讨论文件类别和 DTD 的细节）。

跟在 prolog 之后的是文件元素—XML 文件的核心及文件内容所在的地方。

文件（Document）元素

对您来说，这也许有点奇怪，一个单一的文件元素包含整个 XML 文件里的所有数据。然而，这个单一元素可以包含任何数目的巢状阶层元素和外部实体（**entities**）。这有点类似您计算机上的 C 磁盘驱动器。所有计算机上的数据都存在一个单一的磁盘驱动器中。但是仍然还有许多不

定数目的活页夹与次阶层的活页夹保存各个不同的数据在逻辑与易于管理的结构中。下面这段文字码示范如何把文件元素（在这个例子中加入的是 **PLANT** 元素）加到模板文件中：

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

<!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">


<PLANT>

    <COMMON>Columbine</COMMON>

    <BOTANICAL>Aquilegia canadensis</BOTANICAL>

</PLANT>
```

巢状结构

所谓的巢状结构就是将一个对象包含在另一个对象之中，举例来说：**XML** 文件可以包含巢状元素甚至是其它的文件。

元素的巢状结构建立了它们之间的父/子关系，每个子元素（不是指文件元素）是完全地包含在它的父元素里面。

下面的文字码说明了这样的关系：

```
<DOCUMENT>
```

```
<PARENT1>

  <CHILD1></CHILD1>

  <CHILD2></CHILD2>

</PARENT1>

</DOCUMENT>
```

然而，下列的文字码中，是不适当巢状元素的示范：

```
<DOCUMENT>

  <PARENT1>

    <CHILD1></CHILD1>

    <CHILD2></CHILD2>

</DOCUMENT>

</PARENT1>
```

XML 文件的实体结构

XML 文件的实体结构是由文件中所有内容所组成。如果您把逻辑结构想成是一个停车场的蓝图，那您可以把停车场的空间想成是文件的实体结构。

这些停车空间或保管单位称为实体，它们可以是这文件的一部分或是外部文件（就像是飞机场的外部停车场一样）。每个实体都有一个独一无二的名字及属于它自己的内容，包括从文件中的一

个单一字符到文件外的一个大型档案。以 XML 文件的逻辑结构而言，实体是在前言中被宣告的，而且在文件元素中被参照。

实体宣告在告诉 XML 文件的处理器，要把什么样的内容填入那个「停车空间」。只要在 DTD 中宣告过，这个实体就可以在文件中的任何地方被使用。一个实体的参照告诉处理器去取得实体的内容，并且在文件中使用它。

可解析（parsed）与不可解析（unparsed）的实体

实体可分为可解析与不可解析的。可解析的实体，也叫作文字实体（text entity），这种实体的内容被处理后就成为 XML 文件的一部份。而不可解析的实体可说是一个容器，它的内容可能是文字，也可能不是。如果是文字的话，其内容仍旧是无法解析的 XML。

可解析的实体

可解析实体的目的是要让 XML 处理器解读的，所以它的内容会被处理器摘录出来。在被摘录之后，被解析实体的内容会在实体所参照到的文件位置出现，变成文件文字的一部分。举例来说，在我们的 Wildflowers 文件中，一个 light requirement（LR1）实体可以被宣告成：

```
<!ENTITY LR1 "light requirement: mostly shade">
```

这宣告的含意为：「我宣告一个名为 **LR1** 的实体，它的内容为 **light requirement:mostly shade**」。

此后，每当这个实体在文件中被参照时，它就会被它的内容替换。现在，您已开始看到使用实体的好处，也就是说，如果您想要改变这个实体的内容时，只需要在宣告它的地方改变它的内容即可，而这项改变将会反映到文件中任何用到这个实体的地方。

实体参照

如上面所提到的，每一个实体的内容会被加到文件中每个实体参照到的位置。实体参照如同是内容创作的「容器」，而 **XML** 处理器会将真正的内容在每个实体参照的地方置换。要将实体参照加入文件中，得先插入「&」符号，然后输入实体的名字，跟着是分号「;」。以上面的 **LR1** 为例，我们将插入「&LR1;」，内容会如下文所示：

```
<TERM>Wild Ginger has the following &LR1; </TERM>
```

当处理器处理协同合作的能力到这行时，实体「&LR1;」将会以实体的内容取代。所以，这行将被解析为「Wild Ginger has the following light requirement: mostly shade.」。

参数实体参照

另外一种实体参照是参数实体参照。参数实体参照使用「%」代替「&」，除此之外，它与任何其它的实体参照相同。「%CDF;」便是参数实体参照的一个例子，我们将会在下一章详细地讨论参数实体。

不可解析实体

不可解析实体有时被称为二进制（**binary**）实体，因为它的内容通常是二进制的档案（例如影像）等不能直接由 **XML** 处理器来编译的。虽然如此，不可解析实体也能包含单纯的文字，所以「二进制」这个词有点误导了它真正的意涵。不可解析实体需要跟可解析实体不同的信息：它需要一个可以用来识别实体来源格式或类型的标记。先让我们看一个例子：

```
<!ENTITY MyImage SYSTEM "Image001.gif" NDATA GIF>
```

这个实体宣告表示「在 **GIF** 标记中名为 **MyImage** 的实体是一个二进制档案」，简单地说就是「这是一个 **GIF** 影像」。要让这些实体宣告是有效的，这些标记必须先宣告好。标记宣告帮助 **XML** 应用程序来处理这些外部，或是二进制的档案。现在我们已经用过 **GIF** 标记了，另一个可用的标记宣告如下：

```
<!NOTATION GIF SYSTEM "/Utils/Gifview.exe">
```

这段宣告在告诉 XML 处理器，每当它遇到 GIF 类型的实体时，它应该使用 Gifview.exe 来处理它。正如其它的宣告一样，标记一经宣告，便能在整个文件中使用。我们在下一章将会比较详细地讨论这个主题。

Note

实体参照中不应包含不可解析实体的名字。不可解析实体应该只在 ENTITY 或 ENTITIES 的属性（ATTRIBUTE）值中被提到。稍后在本章「标签的开始和结束」中会讨论属性与属性值。另外，请参考 [第 4 章](#) 中有关属性类型的说明。

预先定义的实体

在 XML 中，某些字符有着特殊的作用，以下例来说，如「<>」及正斜线「/」等，处理器会把它们解译成文件的标签而不是真实的字符数据。

```
<PLANT>Bloodroot</PLANT>
```

也就是说，这些和其它字符是被保留作为文件的标签，而不能被当作内容使用。如果您要将这些字符当做数据显示，它们一定会被当作标签处理而无法显示，您必须使用实体参照将字符插入文件中。举例来说，如果您要插入「<PLANT>」这个字到文件中，您应该使用下列的叙述：

```
&lt;PLANT&gt;
```

在这个例子中，「<」是「<」的实体参照，而「>」则为「>」的实体参照。

下列是所有预先定义实体的实体参照：

实体参照	代表字符
<	<
>	>
&	&
'	'
"	"

Note

根据 W3C 的文件，所有 XML 处理器必需要能辨认预先定义的实体参照，即使这些实体不曾在文件中宣告。虽然如此，这些实体仍被要求要在 DTD 中宣告，如此才是一个效的 XML 文件。

内部和外部的实体

先前的例子已经示范过内部和外部实体间的不同。内部实体并没有个别的储存单位存在，实体的内容就在它的宣告叙述中，如下列所示：

```
<!ENTITY LR1 "light requirement : mostly shade">
```

然而，外部实体在它的宣告中参考到一个储存单位，藉由使用 **system** 或 **public** 辨识字符串来参照实际内容的储存体位置。例如：**system** 辨识字符串提供一个指针来指示一个实体的内容可以在哪被参照，像是 URI（Uniform Resource Identifier），如下列所示：

```
<!ENTITY MyImage  
  
    SYSTEM http://www.wildflowers.com/Images/Image001.gif  
  
    NDATA GIF>
```

在这个叙述中，XML 处理器必须读取 Image001.gif 档案来取得这个实体的内容。

除了 **system** 辨识字符串外，一个实体可以包含 **public** 辨识字符串。**public** 辨识字符串提供一个额外、另一种方法给 XML 处理器去读取实体的内容。**Public** 可以用来让应用程序连接到一个公有的文件函式库。例如：如果处理器无法经由 **public** 辨识字符串产生应用程序所参照的位置，它便会检查 **system** 辨识字符串所参照的 URI。

Note

当本书讨论 XML 的时候，URI 时常被用来代替大家比较熟悉的 URL（Uniform Resource Locator）。在 XML 中，URI 可能是 URN（Uniform Resource Name）或 URL。URI 通常用来作为 Web 资源描述，而且包含在 XML 规格书中。更多有关 URI 的信息，请参阅 <http://www.w3.org/Addressing>。

下列的文字码显示 public 标识字符串的用法：

```
<!ENTITY MyImage
    PUBLIC "-//wildflowers//TEXT Standard images//EN"
    "http://www.wildflowers.com/images/image001.gif"
    NDATA GIF>
```

当所要处理的对象为有效的公有实体时，public 标识字符串便很有用。XML 处理器会在资源列表中检视 public 标识字符串，确认资源是否被连接上，以及决定不需要去取得新的实体，因为它已经有效地存在本地端了。然而，必须直到一个公有信息储存机制更广泛地被使用，system 标识字符串才会被普遍被利用。

下面是我们已经讨论过的四种不同实体类型的摘要：

- 内部实体：实体在它的宣告里面被定义，并且在前言中被宣告。（内部实体必是文字）。
- 外部实体：参照到外部储存单位的实体，例如二进制的档案。（外部实体可能是，也可能不是文字）。
- 可解析实体：实体是由可解析的文字所组成。（一经解析后，文字就变成 XML 文件的一部分）。
- 不可解析实体：不能够被 XML 处理器所解析的实体。（不可解析实体可能是，也可能不是文字。如果是文字，它也是不能解析的文字）。

从上面的四种实体，我们又可以得到四种可能的组合：

- 内部，可解析实体：由可解析文字所组成的内部实体。
- 内部，不可解析实体：由不可解析的文字所组成的内部实体。

- 外部，可解析实体：指向可解析文字的外部实体参照。（一经解析，文字就变成 XML 文件的一部分）。

- 外部，不可解析实体：参照到二进制档案或不可解析文字的外部实体。

XML 语法

XML 的结构规则反映在它的语言规则或语法上，在本节中，我们将要检视一些结构规则，并看看它们在撰写语言组件时扮演的角色。与 SGML 比较起来，大多数的读者应对 HTML 比较熟悉，所以，在本节中我们将用 HTML 来参照 XML。正如您所知，HTML 和 XML 都是 SGML 的应用。也正因为 HTML 和 XML 有相同的父语言，两者的语法也就很类似，但也并不是完全十分相似。

开始及结束标签

在 HTML 文字码中，一个元素通常包含开始和结束标签。XML 不像 HTML，它要求每个元素都要有结束卷标。举例来说，HTML 的 Paragraph 元素通常会包括一个开始标签、一些内容，和一个结束标签，如下所示：

```
<P>This is an HTML Paragraph element.</P>
```

如果您写过很多 HTML 的文字码，您可能正在想：「等一下，我从来不曾使用过 Paragraph 的结束标签!」您甚至可能不知道 HTML 有 Paragraph 的结束标签。因为 HTML（和它的父语言 SGML）允许卷标的省略。也就是说，您可以省略结束卷标而文字码仍然正确。

HTML 是以预先定义结构为基础，这样的结构允许处理器假定某一标签应该位于文件的何处。更明白的说，因为在 HTML 中的 Paragraph 是不能在另一个 Paragraph 标签里面，所以当处理器能读到一个开始的 Paragraph 标签时，便假定前一个段落的结束。这种最小化的技巧在 XML 中是不被允许的，而这正是二种语言语法上最明显的不同。

空白元素卷标

即使 XML 必须使用结束标签，但它却支持空白元素的呼叫。空白元素卷标有效地将没有内容元素的开始和结束卷标结合起来，使用特殊的格式：<TAGNAME/>。请注意：跟随标签名称后的是正斜线 (/) —HTML 并不支持这样的标签。假如，我们建立一个标签叫 <GENUS>，若这个 GENUS 元素没有包含任何数据，我们可能要用开始和结束标签，如下所示：

```
<GENUS></GENUS>
```

或者，我们也可以用空白元素卷标：

```
<GENUS/>
```

属性

属性提供一个方法给元素指定一个值，但却不使属性本身成为元素内容的一部分。让我们看看一

般 HTML 元素以及它如何使用属性：

```
<A HREF = "http://www.microsoft.com">Microsoft Home Page</A>
```

在这里，以<A>卷标所代表的 **Anchor** 元素包含一个名为 **HREF** 的属性。属性的值是

http://www.microsoft.com。虽然，这个属性的值永远不会显示给使用者，但它却包含元素重要

的讯息，并且为 **Anchor** 元素提供了目的地。在 **XML** 中，**name/value** 的格式说明属性使用的方法。

下面这个例子为我们范例文件的其中一种组件加入了属性：

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

<!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">

<PLANT ZONE=3>

    <COMMON>Columbine</COMMON>

    <BOTANICAL>Aquilegia canadensis</BOTANICAL>

</PLANT>
```

请注意：在<PLANT>开始卷标中的 ZONE 属性是跟着 name/value 格式的。

Note

在上面的例子中，并未提到属性值的另一个重要特质是：他们能包含任何 ASCII 字符，包括那些通常保留作为卷标用的字符。因此，在当初设计时，属性的值是无法被 XML 处理器所解析的，也就是说，无法检查属性值的有效性（**validate**）。因此，处理器将只检查属性名称及值，看看它们是否与 DTD 中宣告的类型相符，但并不在乎值的内容是什么。（更多有关属性的宣告信息，请参阅 [第 4 章](#)）

有效的（Valid）与格式正确的（Well-Formed）XML

XML 最重要的二个特征是：它能为文件提供良好的结构以及让数据具有自我描述的能力。但如果不能强制执行那些结构和文法的规则，上述的特征将不会有很大的意义。如果您曾经建立过 SGML 文件，您应该对有效的（Valid）文件有初步了解。

如果您熟悉 HTML，您也应该了解格式正确（Well-Formed）文件的概念。下面二节将讨论这些名词的意义。

有效的文件

如本章先前所讨论的，在前言中所指定的 DTD 规定了文件的所有规则。一个有效的 XML 文件必须很严谨地遵从所有的规则。（下一章会详细地讨论 DTD ）。此外，有效的文件也必须遵守 XML 规格书所规定的有效文件的限制（**constraint**）。

下面的范例是从 XML 规格书第 3.3.2 节中节录出来，关于属性预设的有效性限制：

```
Validity Constraint: Required Attribute
```

如果预设宣告是 **#REQUIRED** 关键词，那么所有元素的属性型态都必须在属性列表宣告中被列出。

处理器必须了解 XML 规格书中的有效性限制，同时检查文件中任何可能违反限制的情况。如果处理器发现了任何的错误，它必须将这些错误传送给 XML 的应用程序。处理器同时也必须读取 DTD 档案，并以它所订的规则来检查 XML 文件的有效性，同样地将这些错误传送给 XML 的应用程序。因为这些处理与检查要花很多的时间（更不要说频宽了），而且这些检查不一定是必需的，因此 XML 支持格式正确（**well-formed**）文件的概念。

格式正确的（**well-formed**）文件

虽然「格式正确的（**well-formed**）」表示有一些规则必须遵守，但这并不像那些有效性限制般严格。在 XML 规格书中说明了格式正确（**well-formed**）文件不是有效文件的概念。所幸，只有 XML 处理器必须与那些「格式正确的」规则打交道。如果您是一位 XML 文件的作者，当您没有遵从那些规则时，处理器会让您知道!

Note

虽然格式正确（**well-formed**）文件并没有被要求严格地遵从有效性限制，但是一份有效的文件必须符合所有的规则和有效性限制。

为什么 XML 允许文件作者简单的遵从语法规则，且不用担心 DTD 而来建立内容?虽然这很可能造成混乱，但这并不是它的用意。还记得第 2 章所提到的，XML 的目标之一是要使 XML 文件易于建立。而「格式正确（**well-formed**）」的概念，可以在没有建立 DTD 的情况下，而仍能达到文件易于建立的目标。下列是「格式正确（**well-formed**）」所提供的好处：

- 「格式正确（**well-formed**）」能减少客户端所需做的工作。举例来说，如果服务器已经检查过文件的有效性，客户就不需要再作一次确认。这样，会因为客户端不需要下载 DTD 而节省下载时间，且因为 XML 处理器不需要处理 DTD，也节省

了处理的时间。

- 在许多情况下，制作 DTD 或确认文件的有效性并不是必要的。举例来说，某人想在一个小规模公司中使用 XML 提供部门网站的架构，但在这个网站中，确认有效性的功能并不需要。
- 除了使用 DTD 之外，仍有许多其它方法提供文件的规则，我们在第 10 章中将会讨论到。在那些情况下，DTD 并非必要的。

根据 XML 规格书，格式正确（**well-formed**）的文件必须符合下列各项标准：

1. 它必须符合文件的定义（如下面所描述）。
2. 它必须遵守 XML 规格书中定义格式正确（**well-formed**）文件的限制。
3. 在文件中所有被参照的可解析实体都是「格式正确（**well-formed**）」。

它必须符合文件的定义 它必须符合文件的定义，也就是说：

1. 它至少包含一个元素。
2. 它包含一个根元素，也称为文件元素，而其它的元素都有适当的巢状架构。

现在让我们再看看稍早所制作的范例文件：

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

<!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">


<PLANT>

    <COMMON>Columbine</COMMON>

    <BOTANICAL>Aquilegia canadensis</BOTANICAL>

</PLANT>
```

这份文件包含 **PLANT** 元素作为单一文件元素,而且 **COMMON** 和 **BOTANICAL** 元素是以巢状结构置于文件元素中。为了要举例说明这项观念,下面的例子不是一个「格式正确的(**well-formed**)」

XML, 因为它在根部 (**ROOT**) 包含二个元素:

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

<!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">


<COMMON>Columbine</COMMON>

<BOTANICAL>Aquilegia canadensis</BOTANICAL>
```

COMMON 和 **BOTANICAL** 元素都位于文件根部层级 (**root level**), 也就是说, 这二个完整的元素紧跟在前言之后。每个元素都有开始和结束标签, 但没有一个元素是在另一个元素里面。

它必须遵守 XML 规格书中定义格式正确（well-formed）文件的限制 XML 规格书中为「格式

正确（well-formed）」规定了某些必须遵守的限制。任何人想要开发一个 XML 处理器都必须

了解这些限制而且严格的执行他们。下面是 XML 规格书中的限制范例：

Well-Formedness Constraint: Legal Character

使用字符参照的字符必须根据非结束字符（nonterminal Char）的规则。

如果字符是以「&#x」为开头，后面的数字和字母一直到遇到「;」符号，代表这个字符的值在

ISO/IEC 10646 里为十六进制。如果它是以「&#」开始，随后的数字直到「;」符号之前，代

表字符的值为十进制。

在文件中所有被参照的可解析实体都是标准的 当可解析实体被 XML 处理器解析后，它就变成

文件的一部分，它们也必须将文件格式化来符合标准的限制。如果您会使用其它人所建立的外部

实体，这一点要特别注意。如果那些外部实体不符合标准的限制，它就可能您的文件中造成错

误。

4. DTD 的规则

在 [第 3 章](#) 中，我们讨论了有关 XML 文件的实体和逻辑结构，包括 XML 宣告的前言与文件类型宣告。本章的重点在前言的文件类型宣告，文件类型宣告代表要被处理的特定文件类型和规范整个文件的规则。这些规则称为文件类型定义（Document Type Definition），或是 DTD——它包含绝大多数的文件类型宣告。

在讨论建立 DTD 的细节之前，让我们先来看看文件类型宣告。

文件类别（Document Classes）

藉由宣告一份文件的特定类型，文件作者可以逐渐理解透过类型或类别所建立的文件。这种类别的观念是从对象导向程序设计引用而来的。在讨论如何将这个概念应用在 XML 之前，您应该先熟悉对象导向设计的一些观念和特性。

Note

您不需要非常熟悉对象导向程序设计语言才能建立 XML 文件，这一节只是要为设计 XML 提供一些背景介绍。

物件：可以重复使用的文字码

可以重复使用的对象能独立地使用内部项目，但通常都是与其它的对象或程序一起使用。大多数的对象是为了特定的目的而设计的；它们有自己的属性，并且会执行特定的动作（称为方法）。对象也有一些共通的特征，这使它们成为强有力且具弹性的程序设计方法。如果您熟悉对象导向程序设计，您应该听说过像：继承（inheritance）、多形（polymorphism），和封装（encapsulation）等术语。对于 XML 文件类别来说，继承和多形的观念特别重要。

以继承来维护「关系」

在程序设计中，继承的观念很类似于家庭组织中的遗传关系：经由家庭的基因遗传，一个人会带有一些相同的家庭特色，对象导向程序设计也以相似的方式运作。程序设计人员建立包含所有「父」对象的特性及属性。新的「子」对象则继承来自父对象的特性。然后程序设计人员能修正新对象的特性，或者加入新对象来建立一个拥有独特特性并且完全独立的对象。

这就是重点！现在让我们来看看类别的观念。我们称「父」对象为「基底类别（base class）」——以此为建立其它对象的基础。由此延伸的「子」对象称为「子类别（subclass）」。程序设计人员可以建立任何数目的子类别，甚至也可以建立子类别的子类别！这样的好处是允许程序设计人员可以使用对象已存在的功能与特性，而且可以修改它们来符合特别的需要。使用对象类别不

但节省程序设计人员重复撰写程序的时间，而且使程序更一致并省下许多测试时间，因为基底类别从不须改变。

现在让我们来看一个范例，假设您写了一个名为 **Book** 的对象。这个对象有 **NumberOfChapters** 和 **CoverColor** 两个属性。**Book** 对象代表一个基底类别，您将以此基底类别为基础而建立几个子类别。现在就来建立二个子类别：**CookBook** 和 **TextBook** 吧！在新的 **CookBook** 对象中，您将 **NumberOfChapters** 属性设为 10，而新的 **TextBook** 对象中，将 **NumberOfChapters** 属性设为 21。**CookBook** 的 **CoverColor** 属性是 Red，而 **TextBook** 的 **CoverColor** 属性是 Blue。接下来您在 **TextBook** 中加了一个 **Glossary** 的属性，并在 **CookBook** 中加了一个 **Recipt** 的属性。

现在您从一个单一的基底类别延伸出二个对象，而且每个外加的属性内容都用来符合特定的目的。**NumberOfChapters** 和 **CoverColor** 属性是直接继承自基底类别 **Book**，它们同时也可以供所有的子类别使用。这个范例的类别阶层如图 4-1 所示：

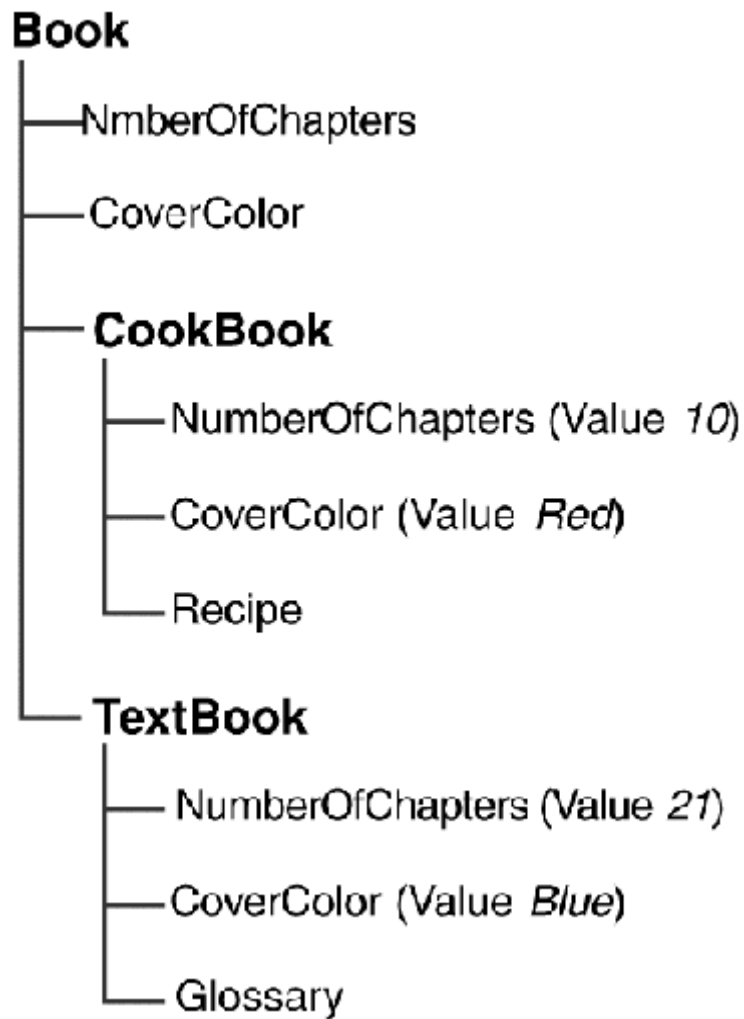


图 4-1: Book 基底类别和它的子类别

Note

您可能已经猜到，您可以更进一步地应用子类别来延伸对象（子类别的子类别）。例如：您能够建立以 **CookBook** 类别为基础的 **VegetableCookBook** 对象，这个对象包含 **CookBook** 类别所有的属性，也包括继承自基底类别的 **Book** 属性。

多形（polymorphism）——让好的物件更好

透过延伸对象多形的特性，经过修改后能执行不同的功能，也就是重写（overriding）基底类别的特性。以 **Book** 对象为例，假设先建立一个名为 **ArtBook** 的子类别。您想取代原本只接受颜色的 **CoverColor** 属性成为可以指定颜色样式名称，也就是想要 **ArtBook** 对象的 **CoverColor** 属性值等于 **PolkaDots**。在这个例子中，基底类别的 **CoverColor** 属性便被重写，与已修改过的子类别 **CoverColor** 属性拥有相同的属性名称。这便是多形的基本观念。

用 XML 来建立文件类别

继承和多形继续在 XML 世界中存在着一但这只是一个概念罢了。检视上一章的 XML 文字码：

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

<!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">

<PLANT>

    <COMMON>Columbine</COMMON>

    <BOTANICAL>Aquilegia canadensis</BOTANICAL>

</PLANT>
```

这段文字码宣告了一个 **Wildflowers** 的文件类型，同时告诉 **XML** 处理器这份文件应该遵从在 **Wldflr.dtd** 档案中所建立的规则。**XML** 的程序设计就如同对象导向的程序设计一样，您可以建立一个「子文件」来继承「父文件」的规则，即使这份「子文件」可能包含完全不同的内容。您也可以修改 **XML**「子文件」的规则以适应特别的需要，这便符合了多形的观念，这是文件类型宣告第二部分的 **DTD** 让这些变成可行的。

文件类型定义（Document Type Definition, DTD）

本章其余的部分把重点放在 **DTD** 上，该如何建构一份 **DTD** 文件以及该如何为您的文件建立 **DTD**。正如先前所提到的，**DTD** 的作用好像一本规则手册，它允许文件作者建立相同类型的新文件，并且拥有与基底文件相同的特性。例如：一份由医学单位所制作的 **DTD** 文件包含病人的名字、药物治疗、诊疗记录等项目，任何使用以 **XML** 为基础文件系统的医学机构，都可以很容易地读取这份文件上的信息。这个系统不但提供一个标准化的文件格式给所有的医疗机构，同时也提供一个格式给单一机构的各个部门使用。相同的文件格式能被医生、护士、管理人员、药剂师、医学专家，和其它相关人员使用。**DTD** 的另一个优点是：能被修改以符合特别应用的需要。

DTD 的结构

DTD 可以由二个部分组成：外部的 **DTD** 子集以及内部的 **DTD** 子集。外部的 **DTD** 子集是存在文件内容以外的 **DTD**，通常这是一份通用的 **DTD**，就像上面的例子；内部的 **DTD** 子集是包含在

XML 文件中的。文件能包含两者或其中之一的子集类型。如果文件中包含上述两种子集，内部子集会优先被处理。当文件作者已使用了外部的 DTD，但为了特殊的应用想要自订 DTD 某些部分，这个功能就很有用了。我们会在<DTD 的类别>一节中详加讨论。

如果您要纳入内部的 DTD 子集到您的文件中，您只需要简单直接地将它写到文件类型宣告区段中即可。然而，外部的 DTD 子集必须经由 DTD 参照来告诉处理器如何透过指定的 DTD 文件名称找到这个外部的 DTD 子集。DTD 参照也包含 DTD 作者的信息、DTD 的目的，和所使用的语言。请看下面的宣告范例：

```
<!DOCTYPE catalog PUBLIC "-//flowers//DTD Standard //EN"
"http://www.wildflowers.com/dtd/wldflr.dtd">
```

建立简单的 DTD

在讨论更多的细节之前，让我们用简单的 DTD 来建立一份文件。我们将会修改在第二章中所建立的备忘录文件，因为它是使用内在 DTD 子集的 Email 文件。这份新文件可以在随书光盘

Chap04\Lst4_1.xml 中找到，如下所示：

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

<!DOCTYPE EMAIL [

    <!ELEMENT EMAIL (TO , FROM , CC , SUBJECT , BODY) >
```

```
<!ELEMENT TO (#PCDATA) >

<!ELEMENT FROM (#PCDATA) >

<!ELEMENT CC (#PCDATA) >

<!ELEMENT SUBJECT (#PCDATA) >

<!ELEMENT BODY (#PCDATA) >

]>

<EMAIL>

  <TO>Jodie@msn.com</TO>

  <FROM>Bill@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <SUBJECT>My First DTD</SUBJECT>

  <BODY>Hello , World ! </BODY>

</EMAIL>
```

文字码 4-1

请注意在文件类型宣告中包含外加的信息。这是一个内部的 DTD 子集，它同时定义了文件中可用的元素及可包含的数据类型。透过 Chap04\Lst4_1.htm 来执行这份文件，并按一下「START」按钮，您会发现这份文件与第 2 章的文件（没有 DTD）看起来很相似，如图 4-2 所示。

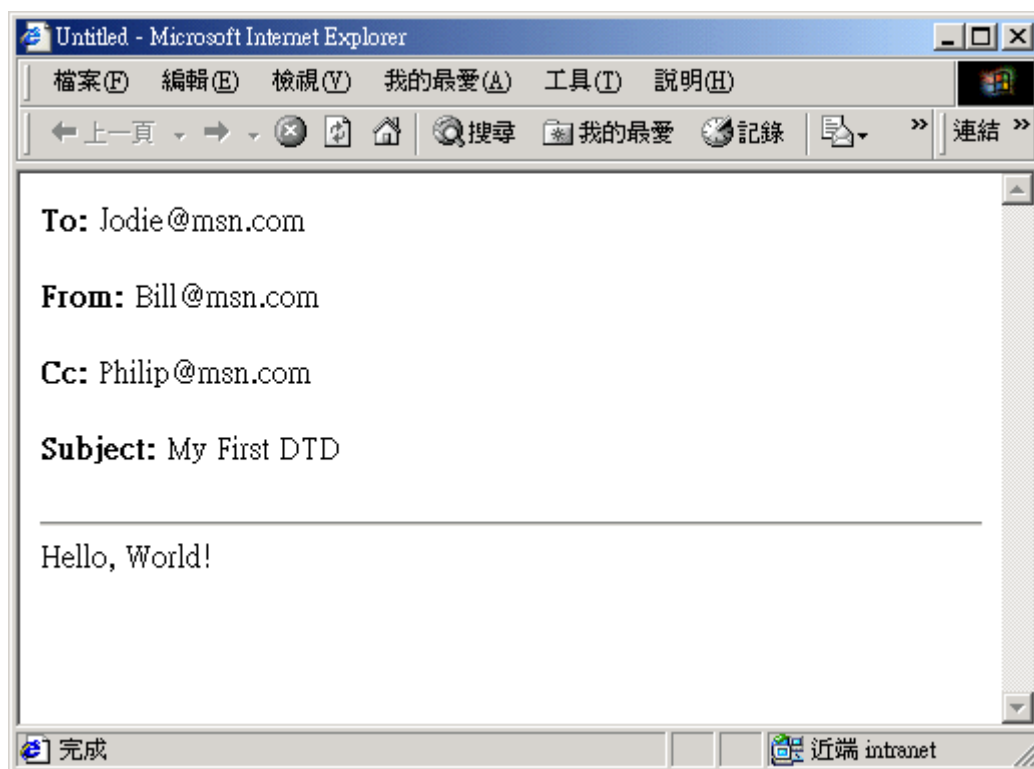


图 4-2：使用内部 DTD 子集的 Email 文件

Note

用来显示 XML 文件的 HTML 网页并没有被详细地讨论,但它与我们在 [第 2 章](#) 中所建立的 HTML 网页相似。您可以用 `Lst4_1.htm` 网页来观看本章中所有的 XML 文件范例。您只要将 `Lst4_1.htm` 中的 `xmlDoc.load` 叙述里的档名改为欲显示的 XML 文件的档名即可。

这份文件不同于 [第 2 章](#) 所建立的文件,因为在这个范例中,XML 处理器用来确认文件的有效性,看是否有违背 DTD 的规则。换句话说,显示文件的 HTML 是使用可确认有效性的处理器来解析

文件。也就是说，处理器会以 **DTD** 来检查文件的有效性，以确定文件中所用的文字码都是被允许的。

Note

并非所有的 **XML** 处理器都是能确认有效性的处理器。不能确认有效性的处理器不需要用 **DTD** 来确认文件的有效性，如此可增进执行效率。

为了要看看确认有效性的工作是如何达成的，让我们加入一个不属于 **DTD** 的元素到文件中。我们要在文件末端加入一个 **Signature** 元素，如下所示。

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

<!DOCTYPE EMAIL [

    <!ELEMENT EMAIL (TO , FROM , CC , SUBJECT , BODY) >

    <!ELEMENT TO (#PCDATA) >

    <!ELEMENT FROM (#PCDATA) >

    <!ELEMENT CC (#PCDATA) >

    <!ELEMENT SUBJECT (#PCDATA) >

    <!ELEMENT BODY (#PCDATA) >
```

```
]>
```

```
<EMAIL>
```

```
  <TO>Jodie@msn.com</TO>
```

```
  <FROM>Bill@msn.com</FROM>
```

```
  <CC>Philip@msn.com</CC>
```

```
  <SUBJECT>My First DTD</SUBJECT>
```

```
  <BODY>Hello , World ! </BODY>
```

```
  <SIGNATURE>Bill</SIGNATURE>
```

```
</EMAIL>
```

因为处理器在 DTD 中并没有发现 **Signature** 元素的宣告，所以当您试着执行这份文件时，将会看见如下的错误讯息：

```
Element content is invalid according to the DTD/Schema.
```

Note

除了经由 **HTML** 来处理 **XML** 的文件之外，您也可以透过能确认有效性的解析器执行 **XML** 文字码来检视执行后的结果。在这个例子中，确认有效性的解析器可从命令列来执行。想要知道如何从命令列来执行解析器，请参阅本书前言 [<有关如何使用 Msxml>](#) 的介绍。

现在回到最初的 XML 档案。这次，我们将对 DTD 作一些改变。请注意第一个和 **Email** 元素有关的宣告：

```
<!ELEMENT EMAIL (TO , FROM , CC , SUBJECT , BODY) >
```

在括号中的是这份文件所能包含的其它元素。这样的清单称为内容 (**content**) 模型，它定义了 **Email** 元素必须包含的子元素以及它们的顺序（更多有关内容模型的细节，请参阅下一节的说明）。如果您将 **Subject** 元素从内容 (**content**) 模型中移除，便如下所示：

```
<!DOCTYPE EMAIL [  
  
    <!ELEMENT EMAIL (TO , FROM , CC , BODY) >  
  
    <!ELEMENT TO (#PCDATA) >  
  
    <!ELEMENT FROM (#PCDATA) >  
  
    <!ELEMENT CC (#PCDATA) >  
  
    <!ELEMENT SUBJECT (#PCDATA) >  
  
    <!ELEMENT BODY (#PCDATA) >  
  
]>
```

正如您所预期的，当执行这份文件时，会引起处理器产生错误，因为它并没有遵循内容 (**content**) 模型的定义。让我们回到最初文件中改变 **From** 元素和 **Cc** 元素的顺序，如下所示：


```
<EMAIL>

  <TO>Jodie@msn.com</TO>

  <CC>Philip@msn.com</CC>

  <FROM>Bill@msn.com</FROM>

  <SUBJECT>My First DTD</SUBJECT>

  <BODY>Hello , World ! </BODY>

  <SIGNATURE>Bill</SIGNATURE>

</EMAIL>
```

同样地，当您执行这份文件时，处理器产生了错误，因为它不能得到预期中的元素。

现在，您应该比较清楚 DTD 就如同是 XML 文件严格的规则手册。也正因为那些规则很严谨，因此谨慎地规划您的 DTD 是很重要的。在本章中的 DTD 都是很简单的。接下来的部分，我们要看看可以加到 DTD 中的其它部分，它可以使 DTD 更稳定及更具弹性。

元素宣告

每个元素宣告中都包含元素的名字和数据型态，称为元素的「内容（content）规格」，是由下列其中一类型所组成：

- 其它元素的列表，称为内容（content）模型

- 关键词 **EMPTY**
- 关键词 **ANY**
- 混合的内容 (**Mixed Content**)

更详细地讨论内容 (**Content**) 模型

在前一个例子中的 DTD 以元素宣告作为开始，在宣告中包含了文件的内容 (**content**) 模型，如下面括号中所示：

```
<!ELEMENT EMAIL (TO , FROM , CC , SUBJECT , BODY) >
```

Email 元素中只包含了次元素或子元素。对于内容 (**content**) 模型中的每一个元素，必须在接下来的 DTD 中出现相关元素的宣告。

EMPTY 元素宣告

您可以使用关键词 **EMPTY** 来宣告一个不能包含任何内容的元素，如下面的例子：

```
<!ELEMENT TEST EMPTY>
```

文件中的 **Test** 元素依照上述的宣告将无法包含任何内容，而且被要求必须是 **Empty** 元素，例如

`<Test/>`。一个不能有任何内容的元素看起来没有什么用，但是它却能包含有内容的属性

(**Attribute**)，或是在文件里提供特定功能。例如 **HTML** 里的`
`标签，它虽然是空元素卷标，

但它却能告诉 **HTML** 处理器—文件在此换行，但是这个元素不包含任何内容。

Any 元素的宣告

与 **Empty** 相反的元素是 **ANY**。如果元素宣告使用 **ANY** 关键词来定义它的内容，那么这个元素

的型态可包含任何 **DTD** 允许的内容，并且以任何的顺序出现。**ANY** 元素宣告看起来如下所示：

```
<!ELEMENT TEST ANY>
```

混合的内容 (Mixed Content)

宣告一个元素的内容时，您也可以将它宣告为一组可以选择的内容，这些可选择的内容互相以「|」

符号分开。例如：

```
<!ELEMENT EXAMPLE (#PCDATA|x|y|z) *>
```

有关`#PCDATA`、字符数据（如 **x**，**y** 和 **z**）、「|」及「*」的用法，会在接下来的各节中详细地

讨论。

数据类型

在 XML 中包含数据类型的运用是很简单的，但仍有一些值得探讨。在文件内容中，XML 允许可解析的字符数据（以关键词 **#PCDATA** 宣告）和字符数据（以关键词 **CDATA** 宣告）。可解析的字符数据包含卷标。而字符数据则是普通的文字，它能包含那些通常被保留作为卷标的字符。

XML 处理器会假设 XML 档案预设的内容是可解析字符。（例外的情况像是属性数据，它只是普通的字符数据。稍后会详细地讨论它）。

可解析的字符数据通常用来作为 XML 文件的内容，当文件作者不想要资料被解析时便可以使用字符数据。检视下面使用字符数据的文件范例：

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

<LESSON>

  <TITLE>Working with XML Markup</TITLE>

  <EXAMPLE>

    <![CDATA[<ELEMENT>A sample element</ELEMENT>]]>

  </EXAMPLE>

</LESSON>
```

在 **Example** 元素中的数据会被显示为<ELEMENT>A sample element</ELEMENT>，其中的标签并不会被解析。如上所示，宣告一个字符数据的区段，您必须以「<![CDATA[」作为区段的开始，同时用「]]」来结束。任何在这范围内的资料都会不经解析而直接显示。

结构符号

XML 使用一组符号来定义元素宣告的结构。您已经看过的符号，如「|」及「,」。表 4-1 列出每一个有效的符号与其目的，以及如何使用和每个符号的意义。

表 4-1 元素宣告符号

符号	目的	例子	意义
()	括号内包含元素群组或一组可选择的内容	(内容 1, 内容 2)	元素必须包含一连串的内容，如内容 1 及内容 2
,	将一连串项目分开并确定显示顺序	(内容 1, 内容 2, 内容 3)	元素必须依指定的顺序包含内容 1、内容 2 及内容 3
	将可选择的项目以群组分开	(内容 1 内容 2 内容 3)	元素必须包含内容 1、内容 2 及内容 3 其中一项
?	表示项目必须出现一次或完全不出现	内容 1?	元素可能包含内容 1。如果内容 1 真的出现，它也只能出现一次

*	表示项目可出现任意次数	内容 1*	元素可以包含内容 1。如果内容 1 真的出现，它可以出现任意次数
+	表示项目必须出现一次或一次以上	内容 1+	元素必须包含内容 1。它可以出现一次或任意次数
没有任何符号	表示只有一个项目会出现	内容 1+	元素必须包含内容 1

现在就让我们来看看如何在我们的范例文件中加入内容模型。

```
<!ELEMENT EMAIL (TO+ , FROM , CC* , SUBJECT? , BODY?) >
```

这个宣告包含：

- **To** 元素是必要的，而且至少出现一次。
- **From** 元素只能出现一次。
- **Cc** 元素是选择性的，但它可以出现一次或一次以上。
- **Subject** 元素是选择性的，如果出现在文件内时，只能出现一次。

- **Body** 元素是选择性的，如果出现在文件内时，只能出现一次。

属性

除了定义元素结构及其内容的种类之外，您也可以将属性与元素联想在一起。属性提供关于元素或元素内容额外的信息，如果您曾制作过 **HTML** 文件，应该对属性很熟悉。现在就以下的 **HTML** 文字码来作说明：

```
<HTML>

<HEAD>

  <TITLE>Database Web Site</TITLE>

</HEAD>

<BODY>

  <A HREF="http://mspress.microsoft.com">

    Click here for a Web link

  </A>

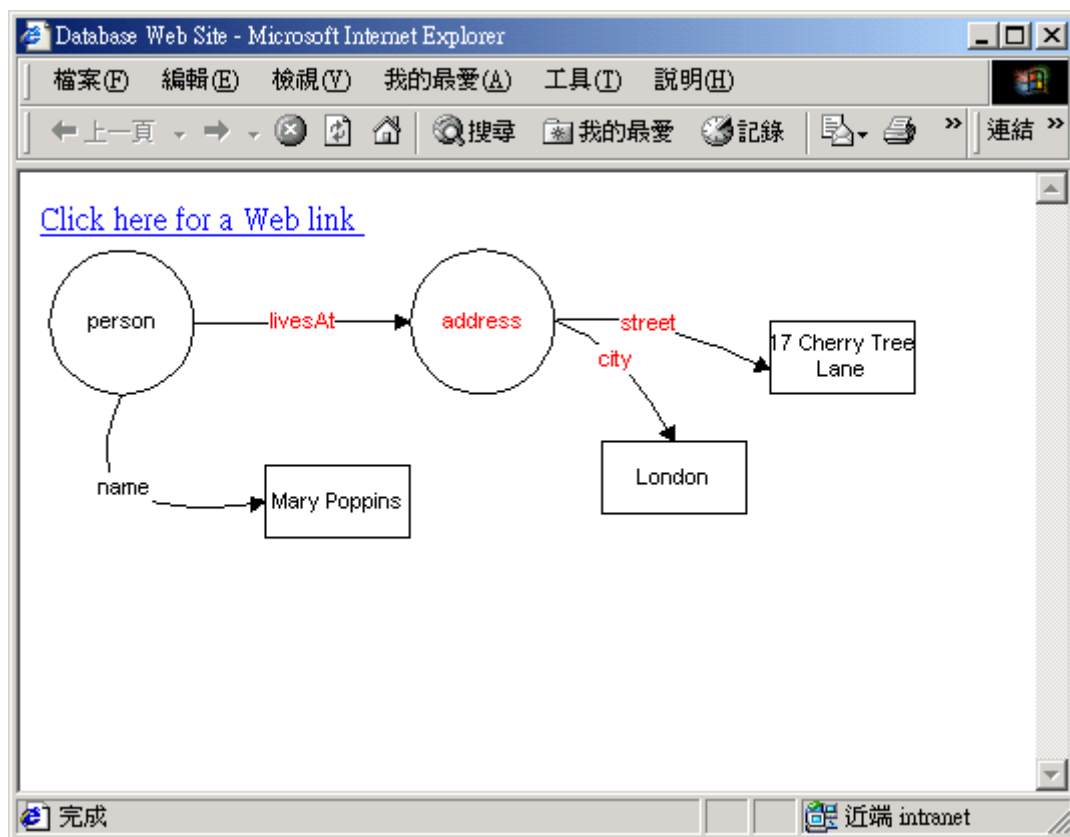
  <BR>

  <IMG SRC="Schemas2.gif" BORDER=0 ALT="A Schema Map">

</BODY>
```

</HTML>

执行结果如下图所示：



Anchor 元素（表示为<A>卷标）和 Image 元素（标示为卷标）都包含属性：Anchor 元素包含了 HREF 属性，而 Image 元素包含了 SRC、BORDER，及 ALT 属性。这些属性为浏览器提供了额外的信息。请注意：Anchor 元素包含的内容为「Click here for Web Link」，但是 Image 元素却显示一个空元素—它包含不可见的內容。然而，事实上这并不完全是真的，当这个元素不包含内容时，它的 SRC 属性是一个文件名称，用来告诉处理器要显示哪一个档案。所以在这个

例子中，一个图形档案被显示在网页上，这个范例呈现属性一个重要的观念，就是属性通常包含了重要的讯息，它不属于元素内容的一部分。这表示，虽然属性的结果通常是看得见，但属性本身对 XML 处理器的意义比那些人们可看见的内容要重要的多。

属性宣告

XML 需使用下列语法在 DTD 中宣告属性：

```
<!ATTLIST ElementName AttributeName Type Default>
```

<!ATTLIST>这个卷标是用来定义属性宣告的卷标；**ElementName** 是元素的名字；**AttributeName** 则是属性的名字；**Type** 指的是属性之前宣告的类型；**Default** 则是属性的默认值。

Note

属性宣告可以出现在 DTD 的任何位置。但是将属性宣告置放于接近其所对应的元素宣告处，会使 DTD 比较容易为人所了解，您也可以在单一元素中使用多个属性宣告。在这个例子中，处理器会将所有的宣告结合成一个清单。如果处理器遇到相同的属性有超过一个以上的宣告时，则只有第一个才有效。

表 4-2 中列出 XML 中有用的属性类型。

表 4-2 XML 中的属性类型

属性型态	用法
CDATA	在属性中只有字符数据可被使用 。
ENTITY	属性值必须参照到在 DTD 中宣告的外部二进制的实体。
ENTITIES	和 ENTITY 一样，但是允许多个数值，可由空格键分隔开来。
ID	属性值应是唯一的。如果一个文件包含的 ID 属性有相同的属性值，则处理器应该会产生错误。
IDREF	其值应参照到文件中别地方宣告的 ID。如果属性没有符合参照到的 ID 值，则处理器应该会产生错误。
IDREFS	和 IDREF 一样，但是允许多个数值，可由空格键分隔开来。
NMTOKEN	属性值是字符名称的组合，这些字符应为字母、数字、虚线、冒号或底线。
NMTOKENS	和 NMTOKEN 一样，但是允许多个数值，可由空格键分隔开来。
NOTATION	属性值必须参照到 DTD 中其它地方宣告的记号。宣告也可以是记号列表，而这个值必须是记号列表中的一个记号，每个记号必须在 DTD 中都有它自己的宣告。
Enumerated	属性值必须符合列举值之一。举例来说： <!ATTLIST MyAttribute (content1 content2) >。

属性宣告的最后一个部分是它的属性默认值。属性默认值可以为下表中四个类型之一。表 4-3 列出可用的属性默认值。

表 4-3 属性默认值

默认值	用法
#REQUIRED	每一个包含这个属性的元素都必须为此元素指定一个值，没有指定值给它会产生错误得结果。
#IMPLIED	这个属性是选择性的。如果处理器没有读到指定的值，便会忽略此属性。
#FIXED fixedvalue	此属性必须有一个 fixedvalue 值。若此属性未包含在元素中，处理器假设值为 fixedvalue。
default	指定属性的默认值，若此属性未包含于元素中，处理器假设值为指定的默认值。

现在就让我们看一看，在 DTD 范例文件中加入属性宣告，其属性该如使用：

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

<!DOCTYPE EMAIL[

    <!ELEMENT EMAIL (TO+ , FROM , CC* , BCC* , SUBJECT? , BODY?) >

    <!ATTLIST EMAIL

        LANGUAGE (Western|Greek|Latin|Universal) "Western"
```

```
ENCRYPTED CDATA #IMPLIED

PRIORITY (NORMAL|LOW|HIGH) "NORMAL">

<!ELEMENT TO (#PCDATA) >

<!ELEMENT FROM (#PCDATA) >

<!ELEMENT CC (#PCDATA) >

<!ELEMENT BCC (#PCDATA) >

<!ATTLIST BCC

HIDDEN CDATA #FIXED "TRUE">

<!ELEMENT SUBJECT (#PCDATA) >

<!ELEMENT BODY (#PCDATA) >

]>
```

在这个例子中，**Email** 和 **Bcc** 元素中都新增了属性。**Email** 元素中首先加入的是 **Language** 属性，**Language** 属性可以包含几种可能选项中的一个。如果没有指定其它值，这个属性将使用默认值 **Western**。**Email** 元素的下一个属性是 **ENCRYPTED**。这个元素必须包含字符数据，而且自从它的默认值设为 **#IMPLIED** 后，如果没有指定其它的值，处理器会忽略这个属性。**Email** 元素的最

后一个属性是 **PRIORITY**，**PRIORITY** 属性有三个可能的值：**LOW**、**HIGH**，以及它的默认值 **NORMAL**。

Bcc 元素中包含 **HIDDEN** 属性。**HIDDEN** 属性的类型是 **CDATA**，而且自从它的默认值设为 **#FIXED** 后，所以属性的默认值被指定紧跟在关键词 **#FIXED** 之后。这个属性必须在 **DTD** 中永远指定一个值给它，在这个范例中指定的值为 **TRUE**。

Note

即使这个属性名称是 **HIDDEN**，同时它的值为 **TRUE**，实际上 **XML** 处理器并不知道它所表示的意义。换句话说，「**HIDDEN**」这个字在 **XML** 中并没有特别的意义；它只是被当成属性名称在使用罢了，而全看应用程序要如何来处理这个属性以及它的值。

属性在 XML 文件中的运作

现在让我们将这份 **DTD** 与其它的文件整合起来，看看属性如何在文件中运作。如下面文字码 4-2

所示：（您可以在随书光盘 **Chap04\Lst4_2.xml** 中找到这段文字码。）

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

```

<!DOCTYPE EMAIL[

  <!ELEMENT EMAIL (TO+ , FROM , CC* , BCC* , SUBJECT? , BODY?) >

  <!ATTLIST EMAIL

    LANGUAGE (Western|Greek|Latin|Universal) "Western"

    ENCRYPTED CDATA #IMPLIED

    PRIORITY (NORMAL|LOW|HIGH) "NORMAL">

  <!ELEMENT TO (#PCDATA) >

  <!ELEMENT FROM (#PCDATA) >

  <!ELEMENT CC (#PCDATA) >

  <!ELEMENT BCC (#PCDATA) >

  <!ATTLIST BCC

    HIDDEN CDATA #FIXED "TRUE">

  <!ELEMENT SUBJECT (#PCDATA) >

  <!ELEMENT BODY (#PCDATA) >

]>

<EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

  <TO>Jodie@msn.com</TO>

  <FROM>Bill@msn.com</FROM>

  <CC>Philip@msn.com</CC>

```

```
<BCC>Naomi@msn.com</BCC>

<SUBJECT>My First DTD</SUBJECT>

<BODY>Hello , World ! </BODY>

</EMAIL>
```

文字码 4-2

请注意：在文字码中，列出所有 **Email** 元素包含的属性与指定的值。在这段文字码中，**Bcc** 元素没有包含任何属性。也因为 **HIDDEN** 属性有预设的**#FIXED** 值，所以处理器会假定这个值是来自 **DTD**。因此当文件被显示出来时，它会如图 4-3 所示：

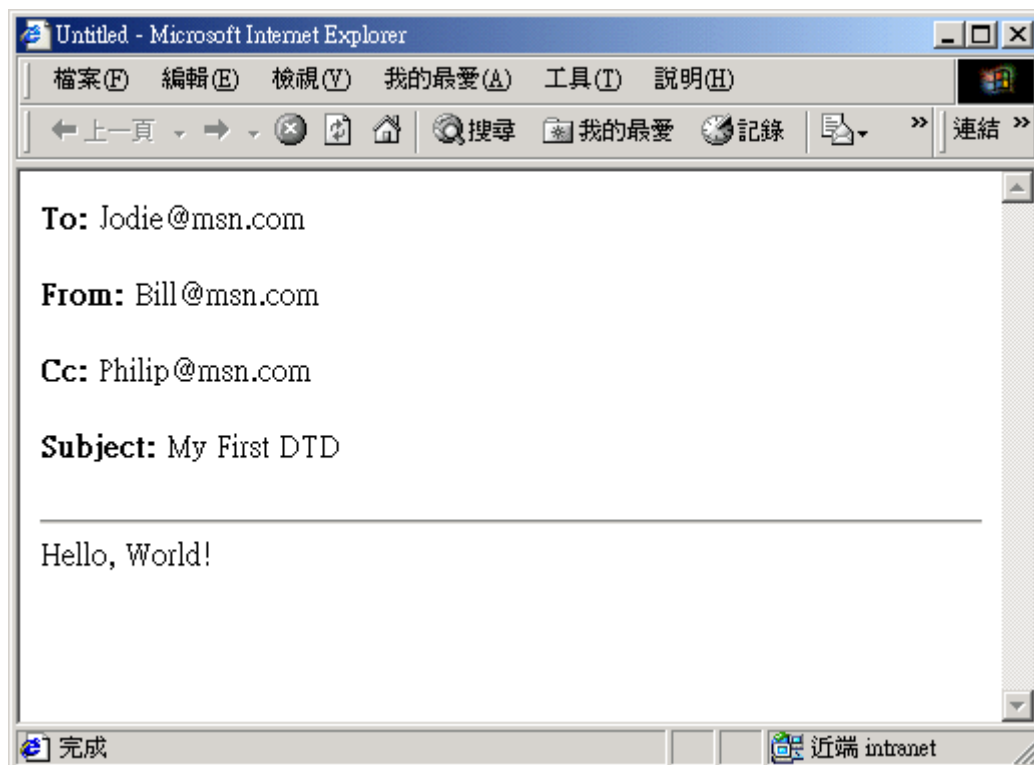


图 4-3: 一份有属性的 XML 文件

您也许已经注意到这份文件与没有属性的文件看起来都一样，别怀疑您是否已经使用了属性！上面这个图说明了属性提供给处理器与应用程序的信息远比提供给使用者的信息多。虽然这份文件中的属性可以影响内容的显示方式（例如插入不同的值给 **LANGUAGE** 属性），但这决定于应用程序要如何使用这些文件提供的信息。因为在这个范例中，并没有使用任何属性来改变内容的显示，所以看起来并没有什么改变。

实体 (Entity)

试着回想一下 [第 3 章](#) 中有关实体结构 (Physical structure) 和实体 (Entity) 的观念。除了有关一般实体 (general entity) 的讨论外，还存在另外一个实体称为参数实体 (parameter entity)。在这一节中我们会详细讨论与说明这两种类型的实体，同时看看应该如何在 DTD 里宣告实体。首先让我们先温习一下一般实体。

一般实体的回顾

您已经知道可以把实体看成是容器的内容，而这些内容可以属于 XML 文件（一个内部的实体），或者是一个文件外部的档案（一个外部的实体）。大多数的实体都必须在 DTD 中宣告。实体宣告遵照与其它宣告相同的基本语法：


```
<!ENTITY EntityName EntityDefinition>
```

在 DTD 中的实体可以被解析或不被解析。可解析的实体或称文字实体，包含的文字会成为 XML 文件的一部分。不可解析实体或称二进制实体，通常会参照到一个外部的二进制档案。不可解析的实体也可能是不可解析的文字，所以不要把不可解析实体想成是 XML 文件的一部分。

内部的实体

内部的实体在 DTD 中被宣告，并且包含文件会用到的内容。下面这行文字码加入一个内部的实体称为 SIGNATURE 到 XML 文件范例中。

```
<!ENTITY SIGNATURE "Bill">
```

这个实体会被加到 DTD 中，而且（您将会在本节稍后看到有关实体参照的说明）每当这个实体在文件中被参照时，它会置换成实体的内容—Bill。

外部的实体：关键词 SYSTEM 和 PUBLIC

这里有一个外部的实体宣告，您可以将它加到 DTD 中。这个实体参照到一个外部的 GIF 档案，而且会在 XML 文件中出现：

```
<!ENTITY IMAGE SYSTEM "Xmlquot.gif" NDATA GIF>
```

请注意：这个外部实体的宣告是不同于内部实体宣告的：它在实体名称后，使用 **SYSTEM** 这个关键词。

Note

外部的实体也可以参照到其它的 XML 档案。举例来说，如果您想把一本书整合成一份 XML 文件，这本书主要的文件便可以使用实体参照到章节。一个实体参照可能是 `<!ENTITY CHAP01 SYSTEM "Chapter1.xml">`。使用这样的参照将会大大地减少主要文件的大小，并且允许个别章节的档案各自独立。

外部的实体宣告可以包括 **SYSTEM** 或 **PUBLIC** 关键词。许多 DTD 是在本地端开发的，也就是说，他们为特定的组织、企业或网站而开发的。在这种情况下，就应该使用 **SYSTEM** 关键词。**SYSTEM** 关键词是紧接在 URI（Uniform Resource Identifier）之后的，用来告诉处理器在何处可以找到宣告中被参照的对象。在上述的例子中，是使用文件名称，因为这个文字码只在本地端使用而已。在下列的宣告中，URI 是 Web 地址，用来指出参照档案的位置：

```
<!ENTITY IMAGE1 SYSTEM  
"http://www.XMLCo.com/Images/Xmlquot.gif" NDATA GIF>
```

对于广大的使用者而言，一些 DTD 被确立为标准是有效的。这时，就应该使用 PUBLIC 关键词，处理器会在 PUBLIC 关键词后找寻一个有效的标准函数库。跟在 PUBLIC 之后的也是 URI，很类似前面例子里 SYSTEM 关键词所用的 URI。使用 PUBLIC 关键词的宣告如下所示：

```
<!ENTITY IMAGE1 PUBLIC "-//XMLCo//TEXT Standard images//EN"
"http://www.XMLCo.com/Images/Xmlquot.gif" NDATA GIF>
```

外部的实体：Notations 及 Notation 宣告

现在，再来看看下面的实体宣告：

```
<!ENTITY IMAGE1 SYSTEM "Xmlquot.gif" NDATA GIF>
```

Notation (NDATA GIF) 会出现在宣告的最后。这个 Notation 告诉处理器什么类型的对象需要被参照。这是个重点，如果您只是简单地将实体宣告加到 DTD 中，并且透过处理器执行它，您会得到如下所示的错误讯息：

```
Declaration 'IMAGE1' contains reference to undefined notation 'GIF'
```

发生这个错误是因为实体宣告参考到一个二进制的档案，而处理器尚未被告知应如何处理这个二进制的档案。请记住这是一个处理器无法「了解」的不可解析实体。此时，notation 必须被宣告成 notation 宣告。notation 宣告会告诉处理器该如何处理某一个特定型态的二进制档案。

Note

虽然在 **notation** 宣告中的信息通常会定义一个「**helper**」应用程序，但在 **XML** 规格书中并不要求这样做。处理器把 **notation** 宣告中的信息传给处理的应用程序；它不关心应用程序是否能了解。这些在宣告中的信息也能被应用在其它目的中：例如提供讯息给应用程序也会显示给使用者。

notation 宣告应该遵从下面的格式：

```
<!NOTATION GIF SYSTEM "Iexplore.exe">
```

上面这个宣告告诉处理器每当它在 **DTD** 中遇到 **GIF** 档案时，它应该使用 **Iexplore.exe** 程序来处理这种类型的档案。

我们要在 **DTD** 范例文件中加入简单的实体宣告：

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

<!DOCTYPE EMAIL [

    <!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?)>

    <!ATTLIST EMAIL

        LANGUAGE (Western|Greek|Latin|Universal) "Western"
```

```
    ENCRYPTED CDATA #IMPLIED

    PRIORITY (NORMAL|LOW|HIGH) "NORMAL">

<!ELEMENT TO (#PCDATA) >

<!ELEMENT FROM (#PCDATA) >

<!ELEMENT CC (#PCDATA) >


<!ELEMENT BCC (#PCDATA) >

<!ATTLIST BCC

    HIDDEN CDATA #FIXED "TRUE">


<!ELEMENT SUBJECT (#PCDATA) >

<!ELEMENT BODY (#PCDATA) >


<!ENTITY SIGNATURE "Bill">

]>
```

现在 DTD 包括了实体宣告。但是与其它的宣告一样，它无法发挥任何作用，直到我们在一份真实的 XML 文件中使用它或参照到它。

实体参考

您现在回想到 [第3章](#)，在文件中使用特定的语法来作实体参照：**&EntityName**。当处理器在文

件中遇到实体参照时，这个参照会告诉处理器应该用这个实体宣告中的内容来置换它。在文字码

4-3 中，我们改变了 XML 范例文件并加入了实体参照，如下所示：

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

<!DOCTYPE EMAIL [

    <!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?)>

    <!ATTLIST EMAIL

        LANGUAGE (Western|Greek|Latin|Universal) "Western"

        ENCRYPTED CDATA #IMPLIED

        PRIORITY (NORMAL|LOW|HIGH) "NORMAL">

    <!ELEMENT TO (#PCDATA)>

    <!ELEMENT FROM (#PCDATA)>

    <!ELEMENT CC (#PCDATA)>

    <!ELEMENT BCC (#PCDATA)>

    <!ATTLIST BCC

        HIDDEN CDATA #FIXED "TRUE">
```

```
<!ELEMENT SUBJECT (#PCDATA) >

<!ELEMENT BODY (#PCDATA) >

<!ENTITY SIGNATURE "Bill">

]>

<EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

  <TO>Jodie@msn.com</TO>

  <FROM>&SIGNATURE;@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <BCC>Naomi@msn.com</BCC>

  <SUBJECT>Sample Document with Entity References</SUBJECT>

  <BODY>

    Hello, this is &SIGNATURE;.

    Take care, -&SIGNATURE;

  </BODY>

</EMAIL>
```

请注意在这段文字码中，任何一个参照到实体 **SIGNATURE** 的地方将以 **Bill** 显示，图 4-4 显示文件被处理过后的结果。

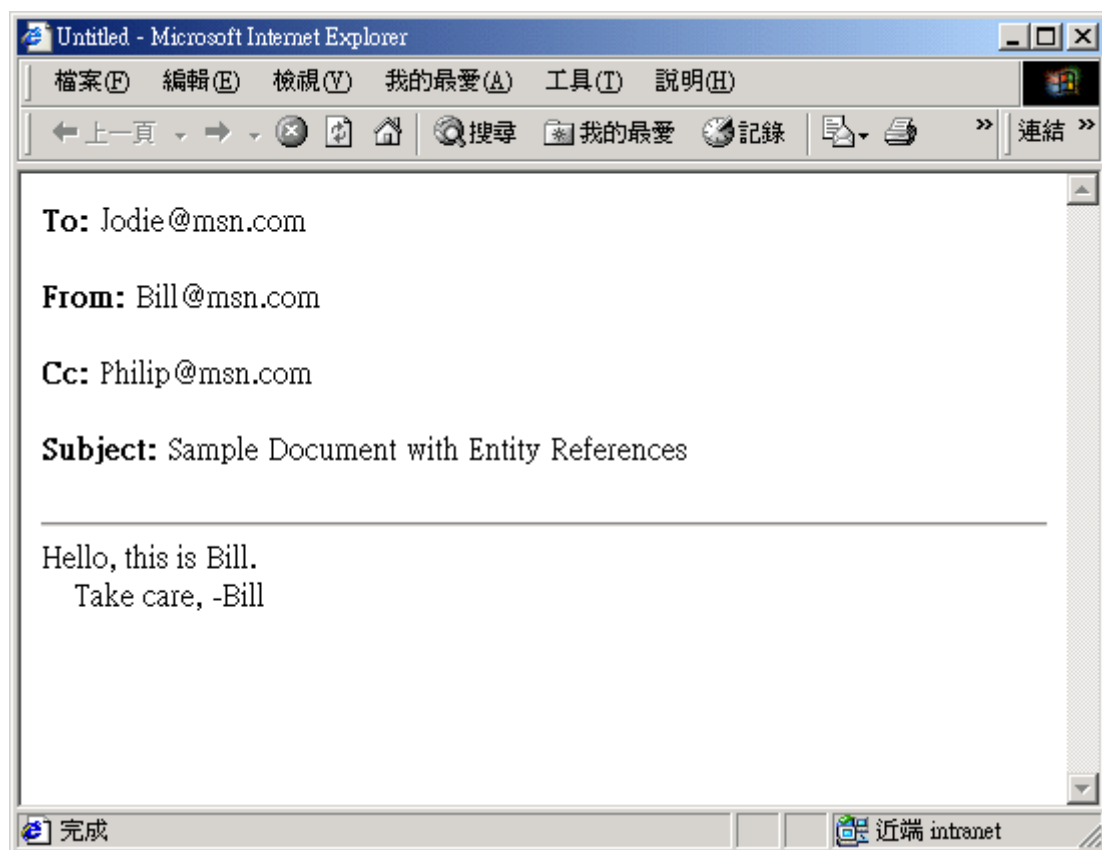


图 4-4：使用实体的 XML 文件

为了要很快地示范实体的功能，让我们将 **SIGNATURE** 实体宣告换成下列所示：

```
<!ENTITY SIGNATURE "Colleen">
```

当文件被处理后，在每个参照 **SIGNATURE** 实体的位置，显示的内容已经改变，如图 4-5 所示：

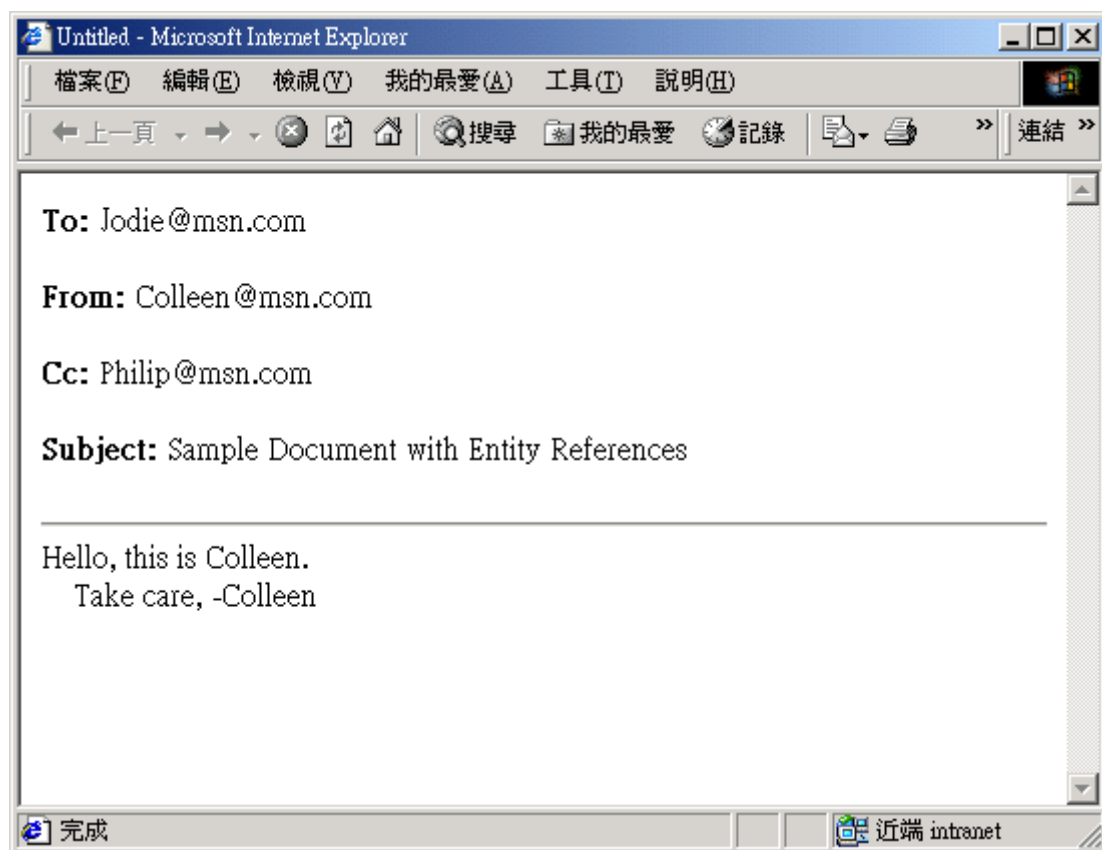


图 4-5：文件所显示的内容会随着实体宣告中的内容而改变

参数实体（Parameter Entities）

虽然参数实体与一般实体工作的方式相同，但是他们在语意上有个重要的不同点。参数实体在宣告和参照中都使用百分比符号「%」。而在实体宣告中，百分比符号「%」是跟在关键词!ENTITY之后、实体名称之前。如下面的例子所示（请注意在%前后的空白是必须的）：

```
<!ENTITY % ENCRYPTION "40bit CDATA #IMPLIED 128bit CDATA #IMPLIED">
```

现在这个实体便可以在 **DTD** 文件其它的地方被参照，举例来说：

```
<!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?)>
```

```
<!ATTLIST EMAIL
```

```
    LANGUAGE (Western|Greek|Latin|Universal) "Weatern"
```

```
    ENCRYPTED %ENCRYPTION;
```

```
    PRIORITY (NORMAL|LOW|HIGH) "NOMAL">
```

请注意除了用%代替&之外，参数实体参照（%ENCRYPTION;）使用相同基本格式来产生实体参照。而在实体参照的%记号后面也不需要加空白。

Note

参数实体仅限于在 **DTD** 中使用。您不能在 **XML** 文件元素中参照参数实体。

就像您所看到的，参数实体是一种在 **DTD** 中建立快捷方式的方式，并且可让它变的比较简洁和更为组织化。因为他们可能会使文件变得复杂和难于管理，所以您必须小心地使用这些实体。举

例来说，您可以在单一参数实体宣告中参照好几个其它的参数实体。身为文件的作者，您必须确定那些参照实际上存在并且其内容是有效的。

IGNORE 及 INCLUDE 关键词

IGNORE 及 INCLUDE 关键词可以让文件作者将 DTD 的某部分「关闭」或「开启」。IGNORE 及 INCLUDE 被用来在 DTD 中建立条件式，使文件适用于各种不同的目的。举例来说，使用 IGNORE 及 INCLUDE 允许文件作者在追踪各种变化时可测试各种不同的结构。IGNORE 及 INCLUDE 与 CDATA 的使用方式大致相同：

```
<![IGNORE [DTD section]]>  
  
<![INCLUDE [DTD section]]>
```

这两个关键词都出现在宣告中，而且每个 DTD 区域中必须包含一个完整的宣告或一组宣告、批注和空白。下面是这两个关键词如何使用的例子：

```
<![IGNORE[<!ELEMENT BCC (#PCDATA) >  
  
<!ATTLIST BCC  
  
    HIDDEN CDATA #FIXED "TRUE">]]>  
  
<![INCLUDE[<!ELEMENT SUBJECT (#PCDATA) >]]>
```

这段程序告诉处理器忽略 **Bcc** 元素及属性清单，同时将 **Subject** 元素纳入处理。当您检视这段文字码时，您可能会认为 **IGNORE** 关键词似乎较 **INCLUDE** 关键词来得有用。因为我们即使不用 **INCLUDE** 关键词也可以完成相同的效果。然而，**INCLUDE** 可以在任何时候很快地改变文件中已经包括或忽略的项目。检视下面的文字码，改变这两个关键词成为参数实体，以及在适当的地方置换其内容：

```
<!ENTITY % SECURE "IGNORE">

<!ENTITY % UNSECURE "INCLUDE">

<![%SECURE; [any number of declarations go here]]>

<![%UNSECURE; [any number of declarations go here]]>
```

在这个范例中，许多不同的宣告能经由改变实体宣告而轻易地开启或关闭。

处理指令（Processing instructions）

处理指令（PI）提供指令给正在处理这份文件的应用程序。**PI** 通常出现在文件的前言中，但是他们能被放置在 **XML** 文件中的任何地方。在我们的 **XML** 范例文件顶端的 **XML** 宣告便是一种常见的 **PI**：

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

PI 是由「<?」开始，接着是 PI 的名称、值或指令，最后以「?>」结束。PI 的名字，或者称为

PI 目标定义了 PI 要使用的应用程序。有关 PI 的详细介绍请参阅<XML 1.0>规格书中的第 2.6 节。

Note

XML 保留 x, m 和 l 这三个字符给他自己使用。除此限制外，处理指令（PI）被用来传送指令给处理文件的应用程序。

下面是使用 PI 的范例：

```
<?AVI CODEC="VIDE01" COLORS="256"?>
```

```
<?WAV COMPRESSOR="ADPCM" BITS="8" RESOLUTION="16"?>
```

批注

批注是属于 DTD 的「其它」部分。虽然批注并不是必要的，但是它们却广泛地被用来批注文件，使文件更易于了解。您可以用批注来解释 DTD 某区段的目的或定义某个参照的意义等等。很明显地，批注可以用来提醒当初制作这份文件的目的，如此您便可以很容易地回到 DTD 中来修改

文件，或让其它人也可以很轻易地达成这个目的。批注并不仅限于在 DTD 中使用，它也可以在 XML 文件中使用。因为这些批注只是为了便于让人们了解而已，所以 XML 处理器将会忽略他们。批注可以出现在批注标签（<!--）与（-->）之间，而且可以是任何文字、卷标，和符号的组合，除了那些会使批注卷标无效的符号以外。（有关批注的细节请参阅<XML1.0>规格书中的第 2.5 节）。下列的黑体字展示批注如何在一份文件中被使用：

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

<!DOCTYPE EMAIL [

<!-- This document could be used as an email template. -->

  <!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?) >

  <!ATTLIST EMAIL

    LANGUAGE (Western|Greek|Latin|Universal) "Western"

    ENCRYPTED CDATA #IMPLIED

    PRIORITY (NORMAL|LOW|HIGH) "NORMAL">

  <!ELEMENT TO (#PCDATA) >
```

Note

为 DTD 及 XML 文件撰写完整的批注，是一种良好的程序撰写习惯。虽然如此，本书的范例文件考虑空间的因素并未使用许多的批注。

外部的 DTD

您或许已经注意到我们的范例文件（在〈实体参考〉一节的文字码 4-3）已经变得相当大了。您或许也已经注意到文件中大部分为 DTD。您可以将 DTD 和文件分开使它们更容易整合在一起。在建立一份独立的 DTD 后，您可以在文件中参照它。

要将我们的范例 XML 文件的 DTD 部分分开，您只要将 DTD 部分剪下，再将它贴入新的文字文件内即可，并在新的档名后加上「dtd」为扩展名。

文字码 4-4 即为一份独立的 DTD 档案，文件名称为 Lst4_4.dtd:

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

<!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?)>

<!ATTLIST EMAIL

    LANGUAGE (Western|Greek|Latin|Universal) "Western"

    ENCRYPTED CDATA #IMPLIED
```

```
PRIORITY (NORMAL|LOW|HIGH) "NORMAL">
```

```
<!ELEMENT TO (#PCDATA) >
```

```
<!ELEMENT FROM (#PCDATA) >
```

```
<!ELEMENT CC (#PCDATA) >
```

```
<!ELEMENT BCC (#PCDATA) >
```

```
<!ATTLIST BCC
```

```
HIDDEN CDATA #FIXED "TRUE">
```

```
<!ELEMENT SUBJECT (#PCDATA) >
```

```
<!ELEMENT BODY (#PCDATA) >
```

```
<!ENTITY SIGNATURE "Bill">
```

文字码 4-4

如果您将先前例子里的内部 DTD 与这份新建立的外部 DTD 作一个比较，您将会发现它们完全相同。然而，要使用这份 DTD，您必须在范例 XML 文件中加入它的参照，如文字码 4-5 所示（可在随书光盘 Chap04\Lst4_5.xml 中找到）。参照到新的外部 DTD 档案是以黑体字型所显示：

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

```
<!DOCTYPE EMAIL SYSTEM "Lst4_4.dtd">
```



```
<EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

  <TO>Jodie@msn.com</TO>

  <FROM>&SIGNATURE;@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <BCC>Naomi@msn.com</BCC>

  <SUBJECT>Sample Document with External DTD</SUBJECT>

<BODY>

  Hello, this is &SIGNATURE;.

  Take care, -&SIGNATURE;

</BODY>

</EMAIL>
```

文字码 4-5

将 DTD 与 XML 文件分开可大幅缩减 XML 档案的大小，并且提供了一些其它的好处。既然 DTD 是独立的档案，它可以被任何其它的文件所参照。此外，其它文件作者可以使用它来建立相同结构但内容完全不同的文件。同时也因为这份新的文件是遵循 DTD 的规则，所以它能被任何知道该如何处理 DTD 的应用程序读取。

这一点将我们带回到本章一开始所谈的文件对象的概念。

类别 DTD

现在您应该很清楚—使用 **DTD** 允许您用相同的基本属性针对不同目的建立文件。这是否让您想到继承的观念呢?为了要建立一份可被许多文件共享的 **DTD**，您必须建立一份基底类别 **DTD**。这基底类别 **DTD** 订定了所有使用它的文件都应遵守的基本规则。更有甚者，藉由结合内部与外部 **DTD** 的使用，文件作者能延伸一份文件子类别并更改一些属性。文字码 **4-6** 示范子类别。（可在随书光盘 **Chap04\Lst4_6.xml** 中找到。）

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

<!DOCTYPE EMAIL SYSTEM "Lst4_4.dtd"[

    <!ENTITY SIGNATURE "Joe">

]>

<EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

    <TO>Jodie@msn.com</TO>

    <FROM>&SIGNATURE;@msn.com</FROM>

    <CC>Philip @msn.com</CC>

    <BCC>Naomi@msn.com</BCC>

    <SUBJECT>Sample Document with External DTD</SUBJECT>
```

```
<BODY>

  Hello, this is &SIGNATURE;.

  Take care, -&SIGNATURE;

</BODY>

</EMAIL>
```

文字码 4-6

这份文件重写（**overrides**）一份外部的，或称基本类别的 DTD，成为内部的 DTD 子集合。如果 XML 处理器同时遇上内部的 DTD 和外部的 DTD，它会使用内部 DTD 中第一个读到的宣告。藉由以 **SIGNATURE** 为名称来宣告一个实体，并以 **Joe** 为它的值，上面的文件便会在类别 DTD 中重写相同名称的实体。从此之后，只要在文件中参照到 **SIGNATURE** 实体，它的值 **Joe** 便会取代最初 DTD 中所定义的 **Bill**。

必要的标签宣告

如同我们在 [第 3 章](#) 中所讨论过的，一份格式正确（**well-formed**）的文件不需要读取或处理 DTD。

这对许多情形都适用，但在某些特殊情况下却会引起问题。举例来说，就算在格式正确

（**well-formed**）的文件中，每个外部的实体仍须被宣告。此时，处理器可能不需要处理外部的 DTD，但是它仍可能需要处理内部的 DTD，因为如此必要的实体宣告才会被适当地读取和处理。

仍然会有必须处理所有的 DTD，以便可以适当地解译一份文件的情况存在。处理这样的情况时，必须在 XML 宣告中纳入必要的标签宣告，称为 RMD。RMD 会告诉处理器它应该如何处理 DTD。

RMD 可以为下列三个值之一：

- **NONE**：这表示处理处不必读取不论是内部的或外部的 DTD 处理文件。
- **INTERNAL**：如果内部的 DTD 为有效的，则处理器必须处理内部的 DTD。
- **ALL**：处理器必须读取与处理任何有效的内部和外部的 DTD。

下面是如何使用 RMD 的例子。在这个例子中，处理器不需要参考任何 DTD：

```
<?xml version="1.0" RMD="NONE"?>
```

如果 RMD 没有被宣告，处理器会预设其值为 ALL。

语汇（Vocabularies）

语汇相当于是本章中主要涵盖的主题，XML 语汇是一组实际的元素与特殊文件类型的结构。在 DTD 中定义的语汇是给规则书使用的。目前，语汇已被应用在因特网及某些组织和企业中。其

中最广为人知的一种可能是频道定义格式（**Channel Definition Format: CDF**）。它是用来定义被设计来自动传送给使用者，或称为「推播（pushed）」的网页。

语汇适合给各种层次的应用程序，以及像是被用来开发特定企业中数据交换的系统，像是图文电视、医药，和法律相关团体等。语汇也适用于同阶层的应用程序，如上面所述的「推播」应用程序中。当本书正在撰写时，已经有好几种语汇完成或是正在开发之中。下面针对其中一些语汇可能的使用作简单的描述。

频道定义格式（**Channel Definition Format,CDF**）

CDF 用来描述推播网页（PUSH）的行为。CDF 现在被微软公司的 IE 所使用，它主要用来描述如下载时间表、频通更新，网页用量和频率等。

开放的金融信息交换（**Open Financial Exchange, OFX**）

OFX 是 SGML 目前的应用。在金融机构中它被用来透过封包通讯，而 OFX 也会很快地成为 XML 的基础。

开放的软件描述（**Open Software Description,OSD**）

OSD 是一种可经由因特网更新和安装软件的数据格式。当软件更新版本时，可用来通知使用者软件版本的更新，而且提供使用者透过因特网获得程序的机制。

电子数据交换（EDI）

EDI 目前被用来支持全世界的数据交换和交易。然而，在目前的应用实作上，只有那些已经安装了兼容系统的组织能安装它来作数据的交换。XML 能够延伸 EDI 的应用，并且让更多的组织都能使用它。目前正在努力让 EDI 成为以 XML 为基础的格式。

5. Script XML

- . Script 的回顾
- . XML 处理器 (Processor)
- . XML: 父/子关系
- . 由 XML 到 HTML
- . 更多的 Script 技术

6. 用 XML 作为资料来源

- . XML 的数据型态
- . XML 的命名空间
- . 使用 XML 数据来源对象 (Data Source Object, DSO)
- . 整合所有的观念

7. 以 XML 连结

- . HTML 的简易连结方法
- . Xlink: XML 的连结机制
- . XPointer 探讨

8. XSL: 具样式的 XML

- . 样式表的基本概念
- . XSL 的概观
- . XSL 样式表 (XSL Style Sheets)

5. Script XML

所有 XML 的支持者都会传达一个强烈的讯息：「XML 是与资料有关的」。而对 XML 感兴趣的人，最常提出的问题则是：「我要如何在我的网页中运用 XML 数据？」。本章将协助您找到这个问题的解答。我们将会讨论 XML 处理器是如何运作的，以及它所产生的数据类型。我们将从 XML 的「对象模型」（object model）开始，XML 对象模型提供一个可存取 XML 数据的应用程序接口给文件作者；同时我们也会讨论对象模型和 XML 数据的关系。最重要的是，我们将会看到 Script 在 HTML 网页里安置和操纵 XML 资料的能力。我们会将第 4 章的示范文件加以修改，而在本章中使用。为了简化讨论起见，用来确认文件的 DTD 并不包含在内。

Script 的回顾

由于本章并没有提供关于 Script 的完整介绍，我们只回顾了一些在网页编辑中必要的认知，以便您对散落在本书各处的例子能有所了解。您应该要熟悉诸如微软的 JScript 或 VBScript 等的 Script 语言（script language），并可以使用 Dynamic HTML 做一些实作运用。

本书所有的 **Script** 范例都将使用微软的 **JScript**。

HTML 和 Script

正因为 **HTML** 元素需要使用卷标，将 **Script** 插入 **HTML** 档案也需要一种特别的标签，如下所示：

```
<SCRIPT LANGUAGE="JavaScript">  
  
<!--  
  
.  
  
.  
  
.  
  
-->  
  
</SCRIPT>
```

这个开始的`<SCRIPT>`卷标可以包含 **LANGUAGE** 属性，使得文件作者能指定所使用的 **Script**

语言 是 **JScript**、**VBScript**，或是其它的语言。

JScript 和 JavaScript

JScript 类似于 Netscape 浏览器的 JavaScript。因此，在您的 Script 程序中应该指定使用哪个语言，如果您使用的是微软的 Internet Explorer，无论您在<SCRIPT>卷标的 LANGUAGE 属性设定为 JScript 或是 JavaScript，JScript 的解译引擎都将被启动。在另一方面，其它的浏览器可能无法辨认 JScript 语言，使用它的结果自然会导致错误。既然 JavaScript 广泛地被许多浏览器接受，所以如果将<SCRIPT>卷标的 LANGUAGE 属性设定为 JavaScript，就不太容易产生错误。

尽管 Script 语言可以出现在 HTML 文件内的任何地方，但通常我们建议放在档案的 HEAD 部分。既然 Script 语言需要和网页上的其它元素交互作用，而且 HTML 文件的下载和运作是异步进行的，把 Script 语言放置在最前面的部分，将可确定它们是先被下载，而在需要时，确定是可以读取到的。

Note

如果您也使用其它的 Script 语言，可能会有兴趣了解其实 IE 是可以作为一个「Script host」。这表示它可以加载其它的 Script 语言，而不是只有在产品出版时所附的 Script 语言而已。这也

使得 Script 开发者可以使用他们更熟悉的语言，例如 REXX 和 PERL。想知道更多 Script host 的信息，可以参考下列网

站： <http://msdn.microsoft.com/scripting/default.htm?scripting/hosting/hosting.htm> .

Dynamic HTML 对象模型

IE 4.0 （或之后的版本）提供 HTML 元素一个更有效和更灵活的对象模型。这个对象模型提供开发者可以存取网页里每个元素的方法，并同时具有完整的属性（每个对象的信息）和方法（在每个对象上动作的方式）。举例来说，对象模型所提供的属性可能包含颜色、文字内容、元素位置 and 元素的 HTML 属性等。举例来说，对象模型有可能显露它的颜色、主题、元素位置和元素属性的 properties。

Note

本节将简短地讨论 Dynamic HTML 对象模型，稍后，您将会学到 XML 的对象模型。当本书正在撰写时，W3C 刚刚发行一个独立于程序语言「文件对象模型」（Document object Model，DOM）

第一级（Level 1）的规格，这可以提供一种使 HTML 和 XML 交互运作的机制。微软计划使用

XML 对象模型来实作 DOM Level 1，并打算持续跟随未来 W3C 在 DOM 上发展的脚步。

文字码 5-1 中显示 Dynamic HTML 对象模型实际运作的方式。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

  <HEAD>

    <SCRIPT LANGUAGE="JavaScript"

      FOR="window" EVENT="onload">

        alert ("Background color is " +

          document.bgColor + ".")

      </SCRIPT>

    <SCRIPT LANGUAGE="JavaScript">

      function changeColor ()
```

```
{

document.bgColor = "red";

alert ("Background color is " +

document.bgColor + ".")

}

</SCRIPT>

<TITLE>Code Listing 5-1</TITLE>

</HEAD>

<BODY BGCOLOR="white" ONCLICK="changeColor () ">

</BODY>

</HTML>
```

文字码 5-1

当使用者在网页上按下鼠标按钮时，范例中的网页（在随书光盘中的 Chap05\Lst5_1.htm 档案）

会利用 **Script**，透过 **document.bgColor** 属性的改变来更换网页的背景颜色。类似的 **Script** 技巧，

在本章稍后（以及本书稍后的章节中）使用 **XML** 对象模型时还会被用到。

事件处理程序（Event Handler）

Script 语言常需要去处理发生在网页上的一些事件。在程序 5-1 例中，当使用者点选网页时，文

件（**Document**）对象就会产生一个事件。这个事件会在网页上触发一个事件，而系统会帮您呼

叫 **Script** 函式以启动「事件处理程序」（**Event Handler**）。通常，一个事件处理程序是在等待

特殊事件的发生，并在它发生时执行函式里面的文字码。当文件被加载时，一个常用的网页事件

处理程序会被呼叫，如下所示（见文字码 5-1）：

```
<SCRIPT LANGUAGE="JavaScript"
FOR="window" EVENT="onload">

  alert ("Background color is " +
    document.bgColor + ".") CODE
</SCRIPT>
```

另一种事件处理程序的使用方式是行内（inline）事件处理程序，同样也可在文字码 5-1 找到：

```
<BODY BGCOLOR="white" ONCLICK="changeColor ( ) ">
```

这种形式的处理程序是被置入到产生这个事件的卷标或对象中。

使用对象的属性

所谓「属性」就是对象的特性。举例来说，HTML 的 `img` 对象具有 `SRC` 属性（`attribute`），这是一种用来辨别影像文件名称来源的属性，而 `WIDTH` 和 `HEIGHT` 属性则是用来辨别影像尺寸的属性。大部分的对象属性在对象（包含 `document` 对象）被使用时都可以被读取和改变，就像程序 5-1 中所示范的。在撰写有效的 `Script` 程序时，运用对象属性是必须的步骤，而当使用 XML 对象模型来撰写 `Script` 程序时，也需要对象的属性。

对象名称

对象的命名是遵从 `dot`（点）记号格式的，也就是由对象名字、子对象、属性、事件或方法的名字组合而成，并用（`.`）分离。在运用 `Script` 语言的对象时，点记号是一种很便利的命名原则。

在点记号之内，您可以由左而右了解一个对象的阶层组织，举例来说：

`window.document.image1.height` 指定了 `height` 是 `image1` 对象的属性，`image1` 是 `document`

对象的子对象，而 `document` 对象是 `window` 的子对象。在某些情况下，命名是可以被缩短的。

举例来说，既然 `image1` 对象与 `document` 对象存在于同样的窗口中，而 `Script` 语言是在 `document`

中运作，因此属性名称可以被缩短为 `image1.height`。在这例子中，`window` 和 `document` 被假设

在同样的窗口中，而 `image1` 对象则存在于 `document` 中。

文字码 5-2 中（在随书光盘的 `Chap05\Lst5_2.htm` 档案中）示范了上述所讨论的 `Script` 原则。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

```
<HEAD>
```

```
<SCRIPT LANGUAGE="JavaScript">
```

```
function showStats ()
```

```
{
```

```
widthData.innerText =
```

```
    "Image width is " + image1.width + ".";
```

```
heightData.innerText =
```

```
        "Image height is " + image1.height + "."
    }

function smaller ()

{

    image1.width = image1.width - 25;

    image1.height = image1.height - 25;

    showStats ()

}


function bigger ()

{

    image1.width = image1.width + 25;

    image1.height = image1.height + 25;

    showStats ()

}

</SCRIPT>
```

```
<TITLE>Code Listing 5-2</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<INPUT TYPE="Button" NAME="smaller" VALUE="Smaller"
```

```
onclick="smaller () ">
```

```
<INPUT TYPE="Button" NAME="bigger" VALUE="Bigger"
```

```
onclick="bigger () ">
```

```
<P>
```

```
<IMG ID="image1" SRC="star.gif" WIDTH=172 HEIGHT=152
```

```
onclick="showStats () ">
```

```
<P>
```

```
<SPAN ID="widthData"></SPAN>
```

```
<BR>
```

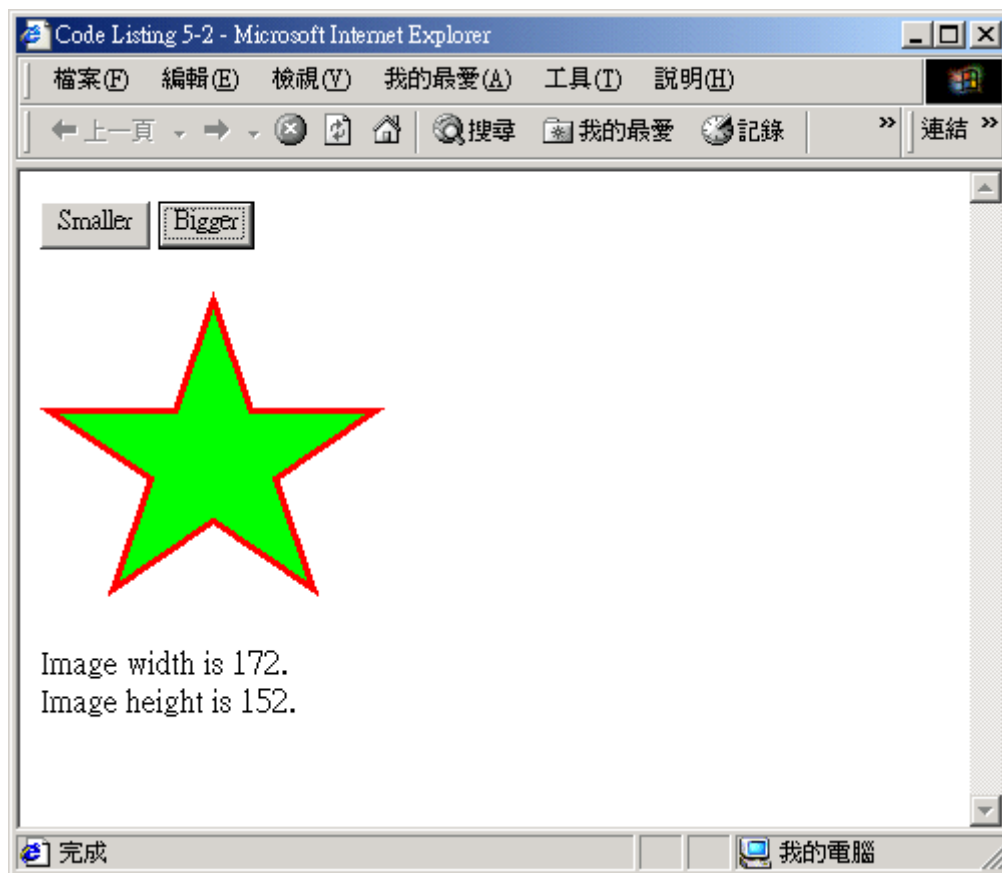
```
<SPAN ID="heightData"></SPAN>
```

```
</BODY>  
  
</HTML>
```

文字码 5-2

当使用者点选某一按钮后，此网页中的 **Script** 会存取 **image1** 对象的宽和高属性以改变网页里影像的大小。每个按钮的事件处理程序皆可触发并执行适当的 **Script** 函式。最后，两个 **Span** 元素的 **innerText** 属性则被使用来显示影像的大小。

执行 **Lst5_2.htm** 后结果如下图所示：



将 XML 与 Script 语言混合运用

现在我们先对 HTML 中的 Script 语言使用暂时告一段落。在本章稍后，我们会讨论许多使用在 HTML 中的方法，也会使用在 XML 的 Script 程序上。Script 语言让您可以使用 XML 来开发具威力的、动态的、数据感知（data aware）的网页。

XML 处理器（Processor）

前面 1 到 3 章中的焦点是放在欲加载进 XML 处理器的数据上，所以您现在应该已经熟悉被送到

处理器 XML 数据的规则 and 标准。现在就让我们来看看处理器是如何运作的。

由于 XML 只描述数据的结构，而不包含处理器应如何展示数据的指令，为了适当的显示数据，

这便是 XML 处理器存在的原因。在前面的例子中很明显的可以看出，处理器主要的工作是解析

(parse) 和验证 XML 数据，但是处理器还必须将数据转变成为可被应用程序使用的格式。既然

资料在没被显示前是无用的，这便是处理器的另一个重要功能，因此处理器的角色可被认为是数

据和显示机制的中介者。

微软的 XML 处理器

如果您使用的是微软的 IE 浏览器，它便已经有个内建的 XML 处理器了。这个处理器 (Msxml)

被设计为窗口组件，而且可被当成 ActiveX 控件在网页、Visual Basic 或 C++ 应用程序中使用。

微软同时也和 DataChannel 公司合作发展以 Java 为基础的 XML 处理器。想知道更多有关 Java

处理器的信息，可以参考 DataChannel 网页，位于：<http://www.datachannel.com>。

选择正确的 XML 处理器

在市面上（或网络上）已有一些商业的、共享的或免费的 XML 处理器，其中有些是「验证有效性」（**validating**）的处理器，其它的则是不验证有效性的（**nonvalidating**）。「不验证有效性」的处理器有执行效率上的优点，因为它不需要读取与解析 DTD。在很多情况下，格式正确（**well-formed**）的 XML 文字码已经够好了，因此它不需要再经由处理器来确认文件的有效性。

许多的处理器都是用 **Java** 程序语言写的。虽然这看起来也许不是很重要，但事实上处理器是由哪种语言所开发的将导致执行效率和可用性的不同。举例来说，尽管 **Java** 应用程序（包含 **applets**）在执行时比 **C++** 的应用程序慢，但 **Java** 可以在多种作业平台上运作。**C++** 应用程序一般而言会比较小和比较快。一般来说，如果您需要最佳的执行效率，并且验证有效性对您来说不是很重要，使用不具验证有效性的 **C++** 处理器是一个较佳的选择。但如果您需要验证您的文件，同时还希望在不同的作业平台上运作，那么您应该选择具验证有效性的 **Java** 处理器。

XML：父/子关系

正如前一章中所提及的，XML 文件具有高度结构性，并且必须遵从语法而是「格式正确的」

（**well-formed**）或是「有效的」（**valid**）。这个结构利用了 XML 元素特殊的阶层次序，因此您

会看到所有 XML 文件都是以父/子元素的方式，组织成为一个「树状」的结构。

回归到基本

还记得 XML 文件只可以含有一个根（**root**）元素，此外每个元素必须套入巢状结构中，也就是

说，子元素的结束卷标必须在父元素结束卷标之前。接下来的程序显示一个正确的树状结构和卷

标及元素间正确的父/子关系。

```
<ROOT>
```

```
  <C1-ROOT.Child>
```

```
    <Ca-C1.Child></Ca-C1.Child>
```

```
    <Cb-C1.Child></Cb-C1.Child>
```

```
    <Cc-C1.Child></Cc-C1.Child>
```

```
  </C1-ROOT.Child>
```

```
  <C2-ROOT.Child>
```



```
<Ca-C2.Child></Ca-C2.Child>

<Cb-C2.Child></Cb-C2.Child>

<Cc-C2.Child></Cc-C2.Child>

</C2-ROOT.Child>

</ROOT>
```

这个树状结构显示一个次级元素都是上一级元素的子元素，而根元素是所有元素的父元素。现在，

如果我们用线条来连结这些分支，您便可以清楚地看出文件的树状结构。

```

<ROOT>
  <C1-ROOT.Child>
    -<Ca-C1.Child></Ca-C1.Child>
    -<Cb-C1.Child></Cb-C1.Child>
    -<Cc-C1.Child></Cc-C1.Child>
  </C1-ROOT.Child>

  <C2-ROOT.Child>
    -<Ca-C2.Child></Ca-C2.Child>
    -<Cb-C2.Child></Cb-C2.Child>
    -<Cc-C2.Child></Cc-C2.Child>
  </C2-ROOT.Child>
</ROOT>

```

尽管文件实际的大小和复杂性的差异可能非常大，所有的 **XML** 文件都是以类似的方法制作。为

了更进一步举例说明这个概念，让我们来看一个错误的巢状结构的例子：

```

<ROOT>

  <C1-ROOT.Child>

    <Ca-C1.Child>

    <Cb-C1.Child>

    </Ca-C1.Child>

    </Cb-C1.Child>

```

```
<Cc-C1.Child></Cc-C1.Child>

</ROOT>

</C1-ROOT.Child>
```

在第五列中，**Ca** 元素在他的子元素 **Cb** 的内部结束。同样的，**ROOT** 元素在 **Cl** 元素前结束（如同第 8 行所示）。这个结构在 **XML** 处理器解析文件时将导致错误的产生。

上面的例子就举例说明的目的而言已经足够了，但是，一个实际的 **XML** 文件可能更能够帮助您了解阶层结构实际运作的情形。文字码 5-3 中（您在随书光盘的 **Chap05\Lst5_3.xml** 中可以找到），显示了一个简单的 **XML** 文件：

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

<EMAIL>

  <TO>Jodie@msn.com</TO>

  <FROM>Bill@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <SUBJECT>My document is a tree</SUBJECT>

  <BODY>This is an example of a tree structure</BODY>

</EMAIL>
```

文字码 5-3

现在，让我们经由命令列将这个程序传送给处理器，并且以「树状模式」输出结果。结果显示在

文字码 5-4 中（您可以在随书光盘的 Chap05\Lst5_4.txt 中找到）。

```
DOCUMENT

|---XMLDECL

|   |---ATTRIBUTE version "1.0"

+---ELEMENT EMAIL

    |---ELEMENT TO

        +---PCDATA "Jodie@msn.com"

    |---ELEMENT FROM

        +---PCDATA "Bill@msn.com"

    |---ELEMENT CC

        +---PCDATA "Philip@msn.com"

    |---ELEMENT SUBJECT

        +---PCDATA "My document is a tree"

+---ELEMENT BODY
```

```
+---PCDATA "This is an example of a tree structure"
```

文字码 5-4

Note

在本书「前言」中讨论过如何使用命令列处理器。

就像您在文字码中所看到的，这个合乎语法的文件遵循了结构的规则，而处理的结果便是正确的

树状格式。您必须由 XML 文件中元素的父/子关系，来了解如何从 XML 文件中读取数据。

XML 的对象模型

您刚看到，当 XML 处理器解析一份文件时，它会为文件中所有的元素建立起类似的树状结构。

请记得这个树状结构仅存在于计算机的内存中，直到应用程序执行完毕。由于资料只是为显示而

被输出，为了让资料真正有用，开发者必须有一个一致的方法来存取数据。因此，开发者必须知

道数据是如何被放置于文件的结构。

XML 对象模型便是要提供这个需要。它提供了一个让开发者可以存取 XML 数据的接口。这个对象模型会将对象中的属性、方法和实际的内容（数据）显露出来。由于 XML 数据的结构是树状的形式，您应该会预期这个对象模型是可以让您取得枝节上的数据的，我们称其为这个树状结构的「节点」（NODE）。没错，这个对象模型确实可让开发者存取整份文件中的数据，无论是在树状结构的根节点或是支节部分（本章只是大略介绍 XML 对象模型，在稍后的章节中将会有更详细的讨论）。

由 XML 到 HTML

让我们先由简单的 HTML 文件开始，接着将会增加读取及处理 XML 文件所必须的机制，最后会用 Script 语言将 XML 数据读入 HTML 网页。

建立基本的 HTML 文件

文字码 5-5（在随书光盘中的 Chap05\Lst5_5.htm）显示一个基本的 HTML 网页。我们将会以此为基础，建构所有取得和显示 XML 资料所需要的一切。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>

<HEAD>

  <TITLE>Code Listing 5-5</TITLE>

</HEAD>

<BODY>

</BODY>

</HTML>
```

文字码 5-5

正如稍早所讨论过的，XML 处理器是介于 XML 文件和 HTML 网页（或任何其它应用程序）之间的中介者。我们需要产生一个 XML 处理器的执行实体（**instance**），让它成为网页上的对象，如此我们就可以经由 **Script** 语言和网页互动。至于要如何建立处理器的实体，完全视您选用哪种处理器而定。

Note

建立一个处理器实体，意味着欲启动这个处理器应用程序（比如说：**Msxml**），并且长驻于内存中以供使用。

假设您使用的是微软的处理器，您便可以使用 C++ 中的 **ActiveX** 控件或是以 **Java** 为基础的 **applet**。如果您决定使用 **Java applet**，在您的文件本体中，需要包含一个 **Applet** 元素。这个元素需要看起来和下列的程序很像：

```
<BODY>

  <APPLET CODE=com.ms.xml.dso.XMLDSO.class

    WIDTH=100% HEIGHT=0 ID=xmlldso MAYSCRIPT=true>

  <!-- In next line, replace filename with XML document name. -->

  <PARAM NAME="URL" VALUE="filename">

  </APPLET>

</BODY>
```


加入 Script

除非您使用微软 IE5 浏览器，否则，XML 数据不会在网页下载时也跟着自动下载。当网页下载

时，您必须使用 Script 语言来下载 XML 数据。我们将增加一个简单的 Script 事件

`document.onload`，这个事件会呼叫 `loadDoc` 函式。而这个函式包含了所有 XML 数据的解析程

序。

```
<SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

    loadDoc ( ) ;

</SCRIPT>
```

接下来的 Script 建立一个处理器的实体，并以它来读取 XML 文件。

```
<SCRIPT LANGUAGE="JavaScript">

    var xmlDoc = new ActiveXObject ("microsoft.xmldom") ;

    xmlDoc.load ("Lst5_3.xml") ;

    function loadDoc ( )

    {
```

```
if (xmlDoc.readyState == "4")

    start ()

else

    window.setTimeout ("loadDoc () ", 4000) ;

}

function start ()

{

    var rootElem = xmlDoc.documentElement;

    var roVar = rootElem.childNodes.item (0) .text;

}

</SCRIPT>
```

现在，让我们花点时间仔细地来看看这段 **Script**。这段 **Script** 的第一部分产生 **XML** 处理器的实体，并指定所欲加载的文件：

```
var xmlDoc = new ActiveXObject ("microsoft.xmldom") ;

xmlDoc.load ("Lst5_3.xml") ;
```

这段程序使用一个名为 `microsoft.xmlDOM` 的 **ActiveX** 对象，并将这个对象的实体指派给变量

`xmlDoc`。然后 XML 的檔名 (`Lst5_3.xml`) 会透过 `load` 函式被传到 **ActiveX control** 的实体。

接下来的 **Script** 中定义了 `loadDoc` 方法，它是用来检查 XML 处理器的 `readyState` 属性。当处

理器读到 `readyState` 的值为 **4** 时（表示文件已经完全被下载），`start` 方法便接着会被呼叫。

Note

这个 `readyState` 会有几个可能的值。请参看附录 A 中的 `readyState` 说明文件。

这段 **Script** 的最后一部分定义了 `start` 方法。这个方法能浏览 XML 文件的树状结构并存取其数据。

而这正是我们进入 XML 对象模型的开端。

让我们再看一次这段程序：

```
function start ()  
  
    {  
  
        var rootElem = xmlDoc.documentElement;
```

```
var toVar = rootElem.childNodes.item (0) .text;

}
```

这个方法中的第一行设定 **rootElem** 变量为 XML 文件的根节点。这个根节点在文件中是唯一的，

正如我们稍早时曾讨论过的。下面便是在文件中对应的程序：

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

<EMAIL>
```

也可以下列结构表示：

```
DOCUMENT

|---XMLDECL

|   |---ATTRIBUTE version "1.0"

+---ELEMENT EMAIL
```

对应到 **Script** 程序中则为

```
var rootElem = xmlDoc.documentElement;
```

上述方法中的第二列程序利用 `childNodes.item (0) .text` 这个属性来读取根元素中第一个子元素的文字内容（第一个子元素是 **To** 元素），并且将结果指派给变量 `toVar`。

这里是文件中对应的文字码所在：

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

<EMAIL>

<TO>Jodie@msn.com</TO>
```

也可以下列树状结构表示：

```
DOCUMENT
|
|---XMLDECL
|
|   |---ATTRIBUTE version "1.0"
|
+---ELEMENT EMAIL
|
|   |---ELEMENT TO
|
|       +---PCDATA "Jodie@msn.com"
```

对应到 **Script** 程序中则为：

```
var toVar = rootElem.childNodes.item (0) .text;
```

文字码 5-6 把所有程序放置到 HTML 文件中。整个网页可在随书光盘中的 Chap05\Lst5_6.htm

中找到。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

<HEAD>

    <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

        loadDoc ()

    </SCRIPT>

    <SCRIPT LANGUAGE="JavaScript">

        var xmlDoc = new ActiveXObject ("microsoft.xmlDOM") ;

        xmlDoc.load ("Lst5_3.xml") ;

        function loadDoc ()

        {
```

```
if (xmlDoc.readyState == "4")

    start ()

else

    window.setTimeout ("loadDoc () ", 4000) ;

}
```

```
function start ()

{

    var rootElem = xmlDoc.documentElement;

    var toVar = rootElem.childNodes.item (0) .text;

}
```

</SCRIPT>

<TITLE>Code Listing 5-6</TITLE>

</HEAD>

<BODY>

```
</BODY>
```

```
</HTML>
```

文字码 5-6

如果您了解这个程序，您便已经了解如何产生 XML 处理器的执行实体、下载一个 XML 文件、浏览文件树状结构及从树状结构中存取数据。也许您不相信，但是最困难的部分已经结束了！以上所示范的便是处理所有 XML 文件的基本动作。

现在我们来看看它在网页中显示的情况。结果显示在图 5-1 中。

Note

如果您是自己制作这个 HTML 网页，必须由随书光盘中拷贝文件名为 Chap05\Lst5_3.xml 的 XML 文件来使网页能运作。这个 XML 文件应该被放置在与 HTML 相同的活页夹中。

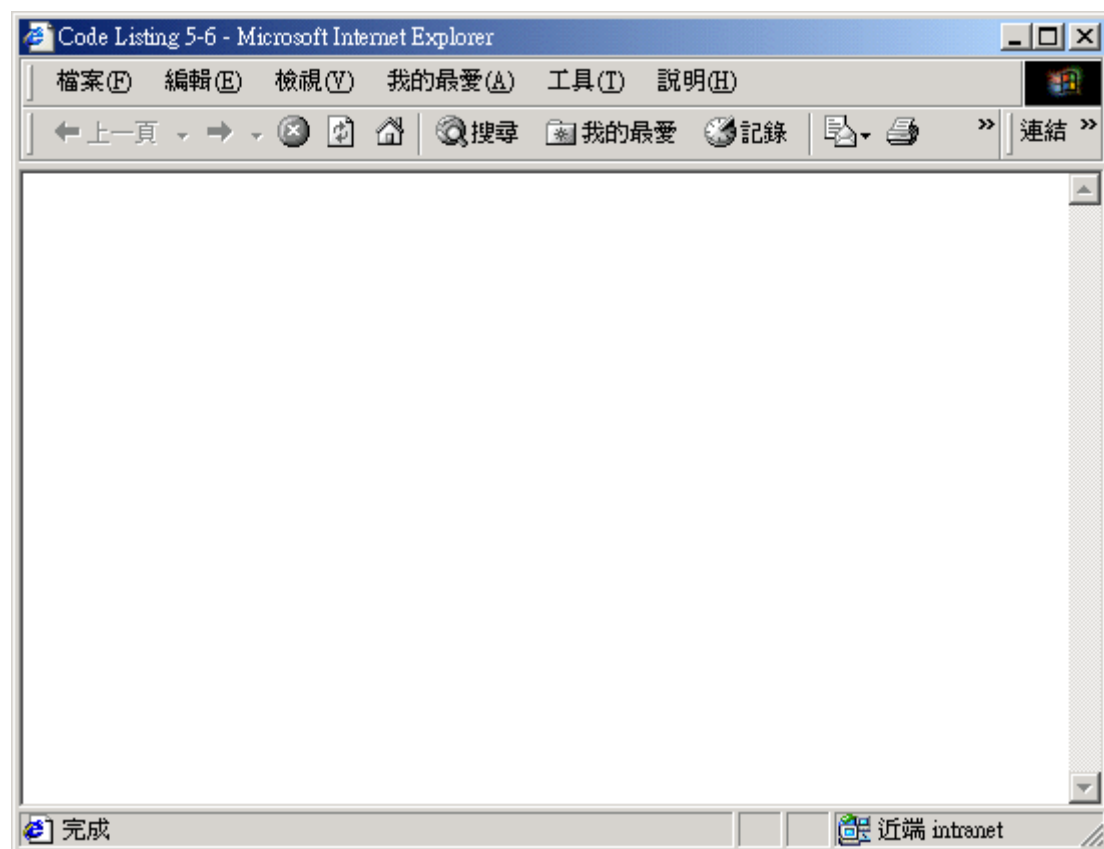


图 5-1

[显示资料](#)

如果您的网页看起来如图 5-1 所显示，表示这个网页算是运作得很完美！这只是个简单的例子，因为我们还没有对从 XML 文件所取得的数据做任何处理。仔细地检视这个网页，您将会发现在网页上没有指定资料应该如何显示或它是否应该出现在网页上。

要让数据显示在网页上，需要做更多的工作，在这之前，让我们先确定程序是否适当地运作。为了要判定我们是否已从 XML 文件中取得想要的资料，我们可以显示对话框，由它来显示从 XML 中传回的值。首先我们得要改变 **start** 方法中的文字码，如下所示：

```
function start ()  
  
    {  
  
        var rootElem = xmlDoc.documentElement;  
  
        var toVar = rootElem.childNodes.item (0) .text;  
  
        alert (toVar) ;  
  
    }
```

您可以看到一个如下所示的对话框：



这个对话框显示这个变量包含正确的值。现在我们要在网页上显示资料。我们需要一个附加的

HTML 元素来包含这个数值内容，及附加的 Script 程序将这数值内容显示在 HTML 网页上。让

我们先从增加 HTML 元素开始。在 HTML 文件的本体中，我们须增加下列的元素：

```
<DIV ID= "to" STYLE="font-weight:bold;font-size:16">  
  
  To:  
  
  <SPAN ID="todata" STYLE="font-weight:normal"></SPAN>  
  
</DIV>
```

这个 Div 元素提供数据的格式和配置的方式。此元素包含一个 Span 元素，它将会显示由 XML

中所取得的资料。您应该已注意到 Div 元素也同时包含一些显式数据的格式化信息。同样地，注

意 Span 元素有 todata 这个 ID，因此我们可以在我们的 Script 程序中参照它。

我们现在要来加入将 XML 数据插入 Span 元素中的 Script 程序。我们先要返回 start 方法中，替

换呼叫 alert 方法的文字码如下：

```
todata.innerText = toVar;
```

这段程序告诉文件用 **toVar** 的值来置换 **todata** 对象中的文字。您应该还记得稍早时我们将 **XML**

中的数据指定给 **toVar**，因此它包含从 **XML** 文件读取的数据。这个程序您可以在随书光盘的

Chap05\Lst5_7.htm 中找到，它应该如文字码 **5-7** 所示：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

    <HEAD>

        <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

            loadDoc ()

        </SCRIPT>

        <SCRIPT LANGUAGE="JavaScript">

            var xmlDoc = new ActiveXObject ("microsoft.xmlDOM") ;

            xmlDoc.load ("Lst5_3.xml") ;

            function loadDoc ()
```

```
{

if (xmlDoc.readyState == "4")

    start ()

else

    window.setTimeout ("loadDoc () ", 4000) ;

}


function start ()

{

var rootElem = xmlDoc.documentElement;

var toVar = rootElem.childNodes.item (0) .text;

todata.innerText = toVar;

}

</SCRIPT>

<TITLE>Code Listing 5-7</TITLE>

</HEAD>
```

```
<BODY>

<DIV ID="to" STYLE="font-weight:bold;font-size:16">

    To:

    <SPAN ID="todata" STYLE="font-weight:normal"></SPAN>

</DIV>

</BODY>

</HTML>
```

文字码 5-7

现在，当网页显示时，资料应该是以 HTML 的型式来显示，如图 5-2:

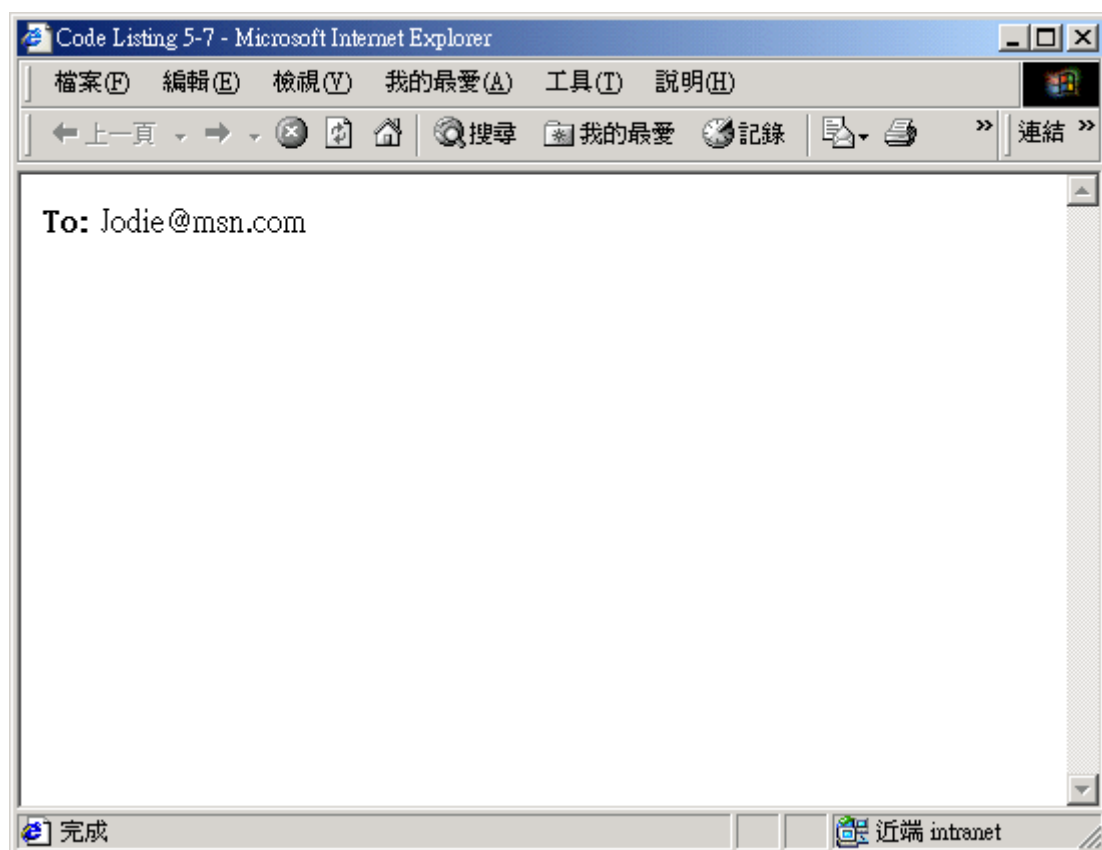


图 5-2

这个文件已经差不多要完成了，但在这之前，我们应该要为这个程序作最佳化，如此可减少程序的复杂性并缩减程序的大小。在目前的程序中，我们将 XML 数据的值指定给一个变量，再以这个变量的值来取代 **Span** 元素中的文字。现在，我们把 XML 数据直接指定到 **Span** 元素中。因此这个程序：

```
var toVar = rootElem.childNodes.item (0) .text;  
  
todata.innerText = toVar;
```

会变成：

```
todata.innerText = rootElem.childNodes.item (0) .text;
```

这虽是一个很小的改变，但是像这样的最佳化动作确实可以在大型的文件中造成一定程度的影响。

现在，我们要增加用来显示 XML 文件中其余元素所需的 HTML 和 Script 程序，而我们所要做的只是重复上述讨论的步骤罢了。

Note

在参考接下来或随书光盘上的程序之前，先看看您自己能否做到。请记得，您需要 Script 程序和可包含数据的 HTML 元素来将数据显示在 HTML 网页上。

文字码 5-8（在随书光盘 Chap05\Lst5_8.htm）包含了完整网页所需的 HTML 和 Script 程序。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```



```
<HTML>
```

```
<HEAD>
```

```
<SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>
```

```
    loadDoc ()
```

```
</SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">
```

```
    var xmlDoc = new ActiveXObject ("microsoft.xmlDOM") ;
```

```
    xmlDoc.load ("Lst5_3.xml") ;
```

```
function loadDoc ()
```

```
{
```

```
    if (xmlDoc.readyState == "4")
```

```
        start ()
```

```
    else
```

```
        window.setTimeout ("loadDoc () ", 4000) ;
```

```
    }

    function start ()

    {

var rootElem = xmlDoc.documentElement;

todata.innerText = rootElem.childNodes.item (0) .text;

fromdata.innerText = rootElem.childNodes.item (1) .text;

ccdata.innerText = rootElem.childNodes.item (2) .text;

subjectdata.innerText = rootElem.childNodes.item (3) .text;

bodydata.innerText = rootElem.childNodes.item (4) .text;

    }

</SCRIPT>

<TITLE>Code Listing 5-8</TITLE>

</HEAD>

<BODY>

<DIV ID="to" STYLE="font-weight:bold;font-size:16">
```

To:

</DIV>

<DIV ID="from" STYLE="font-weight:bold;font-size:16">

From:

</DIV>

<DIV ID="cc" STYLE="font-weight:bold;font-size:16">

Cc:

</DIV>


```
<DIV ID="subject" STYLE="font-weight:bold;font-size:16">  
  
Subject:  
  
<SPAN ID="subjectdata" STYLE="font-weight:normal"></SPAN>  
  
</DIV>  
  
<BR>  
  
  
  
<HR>  
  
<SPAN ID="bodydata" STYLE="font-weight:normal"></SPAN>  
  
</BODY>  
  
</HTML>
```

文字码 5-8

完成后的网页如图 5-3 所示。

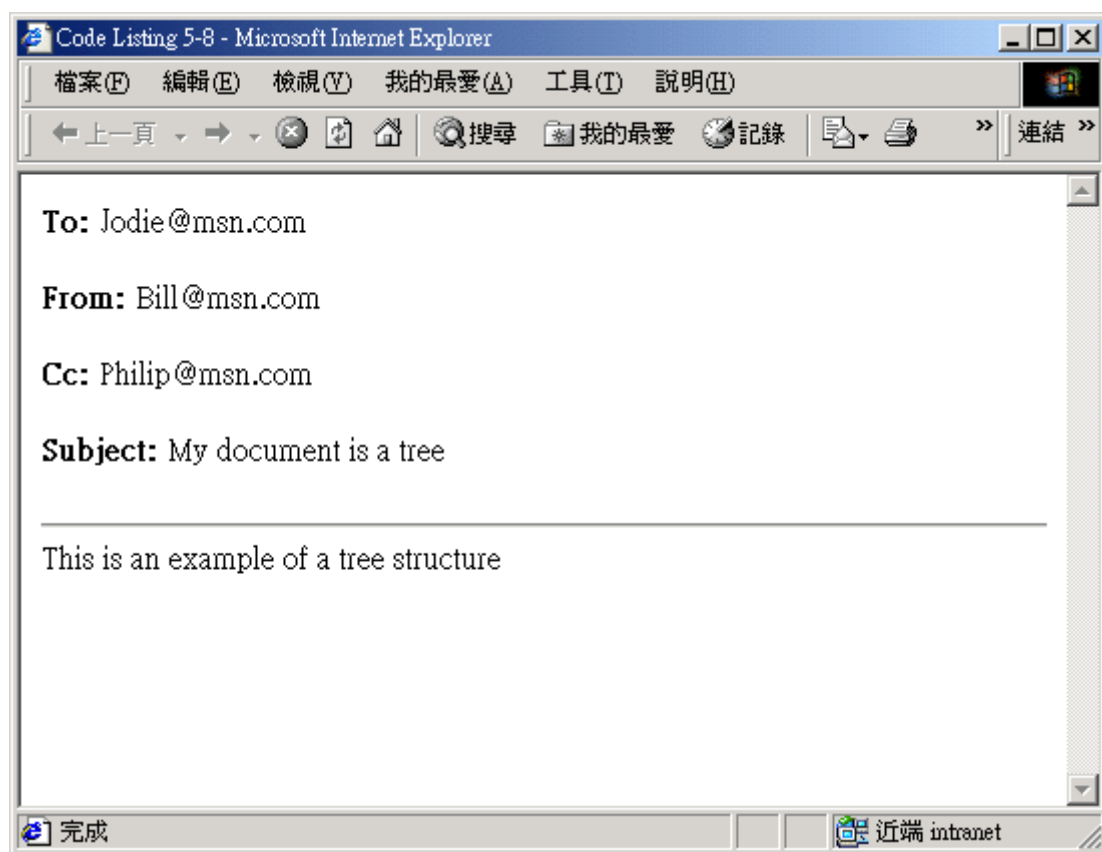


图 5-3

评估成本与优点

在更进一步讨论之前，让我们先来讨论一些对于 XML 可能的潜在质疑，在看了 HTML 网页上显示 XML 数据所有相关的工作之后，您可能会认为直接把数据交给 HTML 显示应该会容易得多，因为那将不需要 Script 程序或独立的 XML 文件。

虽然很明显的，即使制作一份如此简单的文件却需要那么多的程序牵涉在其中，但是下面列举了

使用 XML 所能带来的主要优点：

- 数据和网页是独立的：本章中所制作的 HTML 网页对任何 XML 内容而言就像个样板。尽管制作 HTML 网页需要一些额外的工作，但这文件对不同内容仍能一再被重复使用。如果要针对相同的文件制作不同的版本时，便可以很简单地将新的 XML 档案插入 HTML 样板中，而不是从头制作一份新的 HTML 档案。
- 简化数据建立过程：上述优点的另一面是，不管在任何方式下修正或置换数据内容不表示您必须在 HTML 中工作。举例来说，将内容转换成为不同的语言，只有 XML 档案需要被改变。虽然这可能对小的文件来说没什么用，它对大的或复杂的文件来说却很有帮助。下一点会更扩大这个概念。
- 当复杂性增加时，它的成效越彰显：用 XML 发展简单的文件可能不甚合理。但是当文件越复杂时，使用 XML 便是非常合乎常理的。想象一个高复杂的网页或网站，如果需要定期地改变或更新数据，在完成 HTML 文件后，只需使用 XML 来提供数据，便不需时常编辑或更新 HTML 档案。除此之外，网站的管理者也不

需要编辑 HTML 程序。同样的好处也适用于其它使用 XML 作为数据来源的复杂

系统，就算它们并不用 HTML 文件。

- 数据的格式标准化：由于资料与显示是被分隔开来的，XML 在两个不一致的数据

来源间可以作为一个交换机制。在这个前提下，XML 并不关心数据是如何被使

用的。如果每个系统都可以读写 XML，资料就可以在它们之间被交换。

上面所提的只是 XML 让数据的更新维护更加方便。尽管在运用 XML 之前，确实需要花费额外的时间，但在大部分的应用方面，它所带来的利益远超过所花费的。另外要注意的是，本章中许多例子的目的是让读者了解 XML 文件的结构化以及处理器如何从文件中读取资料。就像您在稍后的章节中会看到的，还有其它方式来将 XML 数据整合在 Web 网页或应用程序中，如 XML Data Source 对象（[第 6 章](#)）、XSL（Extensible Stylesheet Language，[第 8 章](#)），和 XSL Pattern（[第 9 章](#)）。除此之外，还有微软 IE5 增加 XML「数据岛」（Data Islands）的支持（[第 11 章](#)），这些都可以让您直接在 HTML 网页中显示 XML 资料，而不需要分开的 XML 文件。这些方法对于使用 XML 来建构和储存数据可以提供更多的好处。

更多的 Script 技术

本章中简单的电子邮件档案范例示范了如何使用 XML 对象模型，以及其数据。仔细地检视文件和 Script 程序后，您会发现：简单是有代价的。虽然上述的 Script 程序很简单，但是要在大多数的情况下运作的很好，就必须应用一些技术。以下是应考量的因素：

- 从这些 XML 文件树状结构中读取数据的 Script 程序，我们知道它不只表示出元素存在的数量而且包括元素的型态，而这些文字码里所描述的元素型态，并不一定适用于所有的应用程序。
- Script 程序是分别地描述每个元素。这对较小型的档案而言，可以运作的很好，但在大型的或很复杂的文件中，就会变得冗长乏味和具有潜在的错误倾向。
- HTML 文件的数据并非完全的独立。HTML 标签是用来对内容作格式化的（就像 To:和 From:）。如果数据需要被转换为其它语言，它会要求 XML 文件和 HTML 文件都被重新编辑。

我们现在来看一些进阶的 **Script** 技术，这可以帮助我们避免一些刚才所描述的陷阱。

浏览 XML 文件的树状结构

尽管在前面的例子中作者需要知道 **XML** 文件中元素的数目和型态，使用其它的 **XML** 对象模型

的属性和方式可以轻易地避免这个陷阱。首先，来看文字码 **5-8** 和利用下列程序置换 **start** 方法：

```
function start ()  
  
    {  
  
        var newHTML = "";  
  
        rootElem = xmlDoc.documentElement;  
  
        for (el=0; el<rootElem.childNodes.length; el++)  
  
            {  
  
                alert (rootElem.childNodes.item (el) .text) ;  
  
            }  
  
    }
```

现在，下载 HTML 档案时，应该会产生有 XML 文件中元素内容的五个对话框，就像右图所显示的对话框。



上面程序中所作的改变是使用对象模型（`rootElem.childNodes.length` 这个属性）来询问 XML 文件根元素中包含多少子元素。而后，Script 程序再进入循环并浏览根元素中所有的子元素，读取每个元素所包含的内文直到最后一个为止。现在不需要知道有多少元素或子元素以及它们的型态为何。同时，因为我们的文件结构很简单，只需要使用一个循环来读取所有的内容；在较复杂的文件中，要取得所有的元素和子元素时，您也可以使用循环来处理。您甚至可以利用一个元素的 `childNodes` 这个属性，找出一个元素是否有子元素。如果子元素存在，这个属性会返回一个值，如果不存在则传回零。（在附录 A 中有完整的讨论。）在这个例子中，既然我们没有办法知道哪个 HTML 卷标应该接收哪份数据，每个元素的计算结果就都显示在对话框中。请记

得，我们不知道 XML 文件中的元素和 HTML 元素间一对一的关系。我们可以藉由 **Script** 来建立

一个 HTML 元素解决这个问题。我们要把 HTML 文件的本体改成下列所示：

```
<BODY>

  <DIV ID="content">

    </DIV>

</BODY>
```

相当的简单！这是因为我们不再需要一个写定的元素来包含内容。现在要增加一段 **Script** 程序，

可以在 XML 文件读出数据后实时建立一个 HTML 元素，藉由改变 **start** 方法来达成这个目标：

```
function start ()

{

  var newHTML = "";

  rootElem = xmlDoc.documentElement;

  for (el=0; el<rootElem.childNodes.length;el++)

  {

    if (el != rootElem.childNodes.length-1)
```

```

{

newHTML = newHTML +

    "<SPAN STYLE=' font-weight:bold;font-size:16'>" +

    rootElem.childNodes.item (el) .nodeName +

    ": </SPAN><SPAN STYLE=' font-weight:normal'>" +

    rootElem.childNodes.item (el) .text + "</SPAN><P>";

}

else

{

newHTML = newHTML +

    "<HR><SPAN STYLE=' font-weight:normal'>" +

    rootElem.childNodes.item (el) .text + "</SPAN><P>";

}

}

```

```
Content.innerHTML = newHTML;
```

```
}
```

这个 **Script** 程序完成了几件事情。每当 **Script** 程序读取一个新的 **XML** 元素，它便会建立一个新的 **HTML** 元素，用来包含这个元素所有的格式化信息。然后将这个元素加到 **newHTML** 变量中。

这个 **Script** 也使用一个 **if** 叙述来测试文件中的最后一个元素。当最后一个元素被读取时，便应用不同的格式化方式。最后，**Script** 将 **newHTML** 的内容指派给名为 **content** 的 **Div** 元素。既然

newHTML 中包含了有效的 **HTML** 程序，它便会显示我们所要的。文字码 5-9（在随书光盘的 **Chap05\Lst5_9.htm** 档案中）显示这个经过修改的网页。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

    <HEAD>

        <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

            loadDoc ()

        </SCRIPT>

        <SCRIPT LANGUAGE="JavaScript">
```

```
var xmlDoc = new ActiveXObject ("microsoft.xmlDOM") ;
```

```
xmlDoc.load ("Lst5_3.xml") ;
```

```
function loadDoc ()
```

```
{
```

```
if (xmlDoc.readyState == "4")
```

```
start ()
```

```
else
```

```
window.setTimeout ("loadDoc () ", 4000) ;
```

```
}
```

```
function start ()
```

```
{
```

```
var newHTML = "";
```

```
rootElem = xmlDoc.documentElement;
```

```
for (el=0; el<rootElem.childNodes.length; el++)
```

```

{

if (el != rootElem.childNodes.length-1)

{

newHTML = newHTML +

    "<SPAN STYLE='font-weight:bold;font-size:16'>" +

    rootElem.childNodes.item (el) .nodeName +

    ": </SPAN><SPAN STYLE='font-weight:normal'>" +

    rootElem.childNodes.item (el) .text + "</SPAN><P>";

}

else

{

newHTML = newHTML +

    "<HR><SPAN STYLE='font-weight:normal'>" +

    rootElem.childNodes.item (el) .text + "</SPAN><P>";

}

}

```

```
        content.innerHTML = newHTML;

    }

</SCRIPT>

<TITLE>Code Listing 5-9</TITLE>

</HEAD>

<BODY>

    <DIV ID="content">

    </DIV>

</BODY>

</HTML>
```

文字码 5-9

当这个网页显示时，它看起来会像图 5-4。

在图 5-4 中，您应会注意到文件看起来很像图 5-3 中的文件。两者不同的是：这份 HTML 文件是完全独立于数据的。所有的资料都是由 XML 文件中读取的，如此，我们可以经由更灵活和更简洁的程序来达成同样的效果。

为了做更进一步的示范，让我们增加一个 Bcc 元素到 XML 文件 Lst5_3.xml 上，如下面所示。

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

<EMAIL>

  <TO>Jodie@msn.com</TO>

  <FROM>Bill@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <BCC>Naomi@microsoft.com</BCC>

  <SUBJECT>My document is a tree</SUBJECT>

  <BODY>This is an example of a tree structure</BODY>

</EMAIL>
```

现在，当示范的 HTML 网页开启时，Bcc 元素会包含在文件中，尽管 HTML 程序并没有做什么改变。这结果显示在图 5-5。

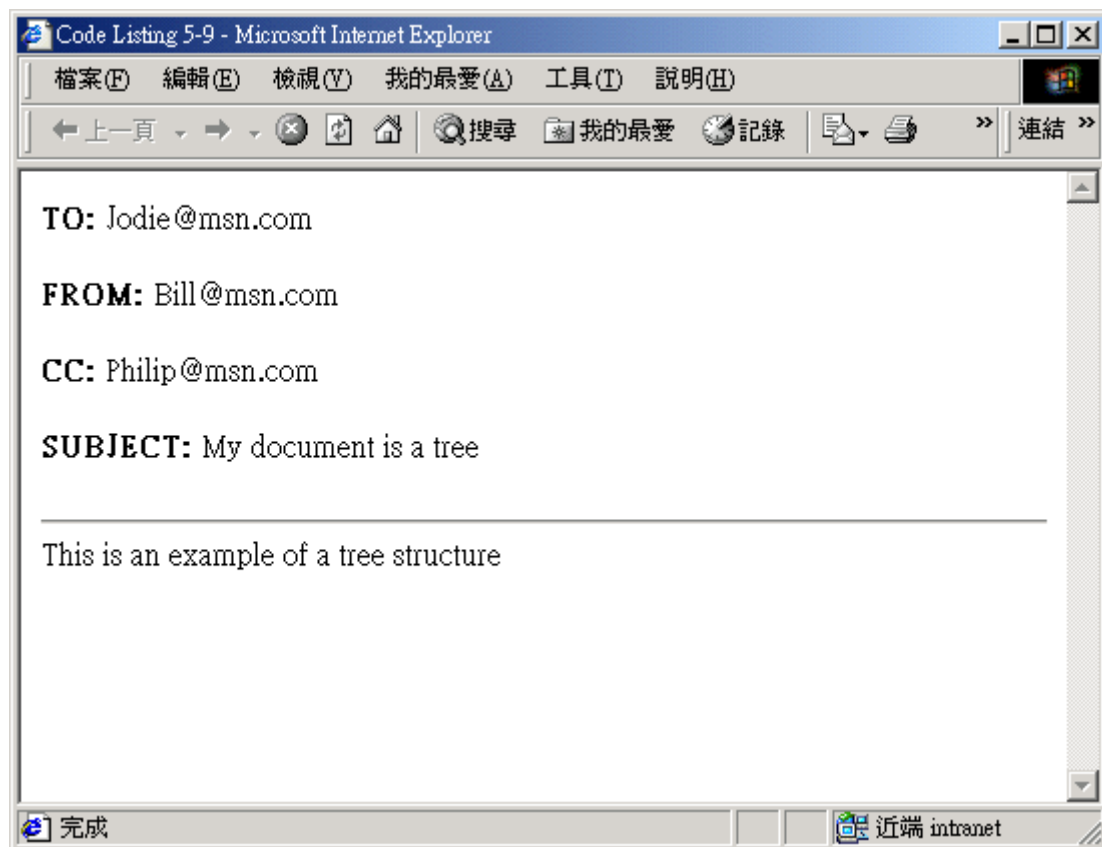


图 5-4：显示 XML 文件的内容

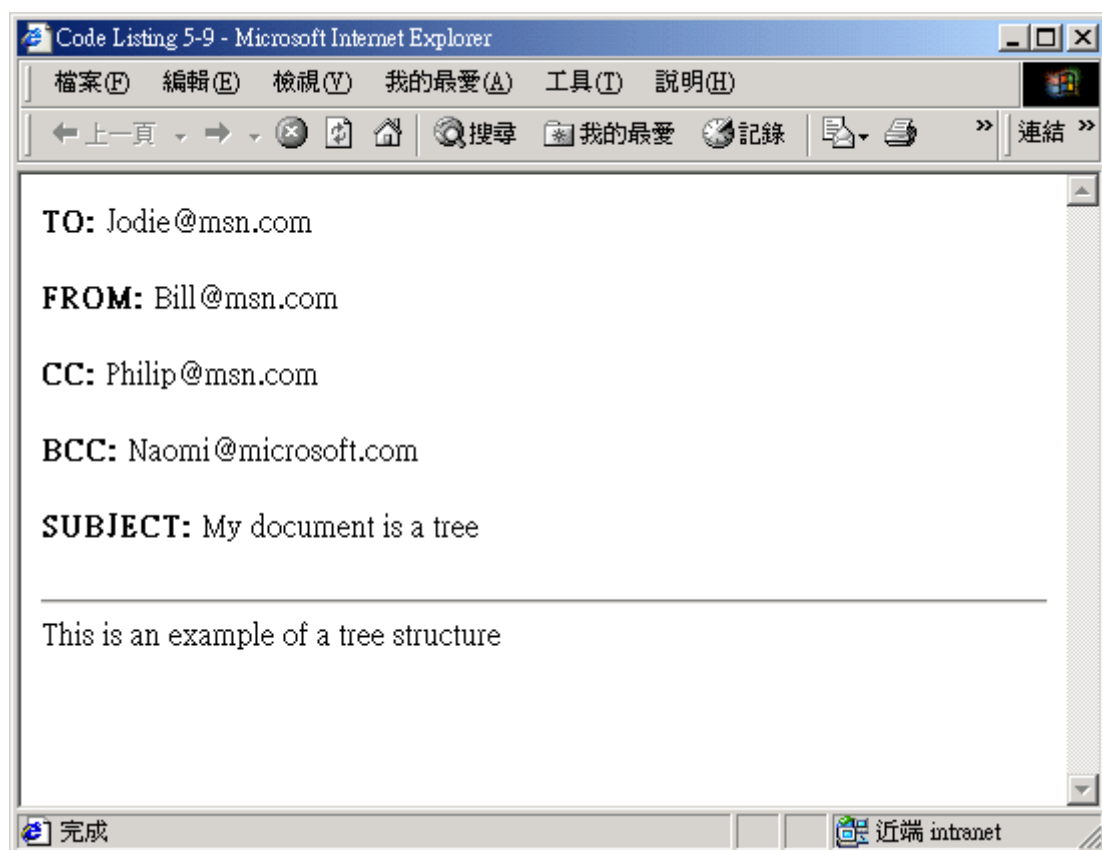


图 5-5: 在 XML 文件中加数据并不需要改变 HTML 档案

这个网页现在完全是数据导向的。尽管如此，我们仍然需要很多有关 XML 文件数据的信息，否则我们仍然不知道如何去应用格式化。举例来说，程序会测试 XML 文件的最后一个元素，因为我们知道最后一个元素是 **BODY** 而且它应该和其它的元素有不同的格式。

请注意，尽管使用这些技术可以使 HTML 文件更灵活和有力，但这个文件仍然只是特殊类别 XML 文件的一个样板。这使 HTML 和 XML 能互补对方的不足。

错误的处理

当一切运作正常时，**Script** 是完美的语言，但当问题发生时，对使用者而言，它也可能是令人懊恼的。**XML** 对象模型提供了一些让开发者处理由 **XML** 文件所导致错误的方法。其中之一是文件对象的 **parseError** 这个属性，它提供有关在 **XML** 文件所发生的错误信息，由此开发者可以决定要如何处理它们。

这 **parseError** 的属性对每个发生的错误提供一个代码。作者可以使用这些代码来：

- 排除 **XML** 文件或 **Script** 程序的错误。
- 告知使用者问题所在，并且提供相关解决方式的建议
- 捕捉某些错误并在发生处修正它们

下列的示范显示 **parseError** 如何被使用来告知使用者问题。

```
var xmlDoc = new ActiveXObject ("microsoft.xmlDOM") ;
```

```
xmlDoc.load ("Lst5_3.xml") ;

if (xmlDoc.parseError.reason == "")

    alert ("Document Toaded successfully");

else

    alert ("The following error occurred: " +

        xmlDoc.parseError.errorCode)
```

这个技术可被用来排除错误，以及避免使用者被错误所苦。**Document** 对象和它的许多属性在 [附录A](#)

[附录A](#) 中有更详细的讨论。

6. 用 XML 作为资料来源

前面几章是把焦点放在 XML：即 XML 档案如何建立、管理结构的规则，以及如何透过 Script 语言存取 XML 对象模型的属性及方法。本章中，我们要将注意力移转到它的内容或资料上。之后，讨论的 XML 都与资料相关。因此我们要试着将 XML 当作数据来源使用，就如同使用微软的 Access 及 SQL 数据库一样。本章涵盖了 XML 数据型态，以及在 XML 中建立数据结构。另外要来学习使用 XML 数据来源对象（Data Source Object, DSO），并测试如何使用 DSO 来读写数据。

XML 的数据型态

如果您熟悉数据库或程序设计，您应该了解数据的不同型态，如字符串（string）、数字（number）、日期（date），货币（currency）或其它的数据型态。到目前为止，我们已讨论过的 XML 数据都只有字符串，或是文字数据而已。然而，XML 数据型态允许文件作者指定元素数据为一个对象，并可以被解译成不同的型态。

由数据型态来辨别元素型态是很重要的。在第 4 章已经看过如何在 DTD 中定义元素型态，您应该还记得一个元素可以包含数据，数据可以是被解析的字符数据（#PCDATA）或是字符数据

(CDATA)。DTD 可以更进一步的藉由它的内容或在结构中的位置来定义元素。这个元素型态的定义方式，表达了元素的语义（就像年龄表示一个人有多老），但却没有描述元素所包含的数据型态（就像年龄包含一个数字）。被解析的字符数据与字符数据都是字符串数据型态。您可以回顾一下第 4 章中还有其它的型态可以指定给 XML 属性。这些型态在下面重新叙述一遍：

属性型态	用法
CDATA	在属性中只有字符数据可被使用。
ENTITY	属性值必须参照到在 DTD 中宣告的外部二进制的实体。
ENTITIES	和 ENTITY 一样，但是允许多个数值，可由空格键分隔开来。
ID	属性值应是唯一的。如果一个文件包含的 ID 属性有相同的属性值，则处理器应该会产生错误。
IDREF	其值应参照到文件中别的地方宣告的 ID。如果属性没有符合参照到的 ID 值，则处理器应该会产生错误。
IDREFS	和 IDREF 一样，但是允许多个数值，可由空格键分隔开来。
NMTOKEN	属性值是字符名称的组合，这些字符应为字母、数字、虚线、冒号或底线。
NMTOKENS	和 NMTOKEN 一样，但是允许多个数值，可由空格键分隔开来。

NOTATION	属性值必须参照到 DTD 中其它地方宣告的记号。宣告也可以是记号列表，而这个值必须是记号列表中的一个记号，每个记号必须在 DTD 中都有它自己的宣告。
Enumerated	属性值必须符合列举值之一。举例来说：<!ATTLIST MyAttribute (content1 content2) >。

在上述表中所列出的属性型态都是我们已经知道的基本数据型态，微软 XML 处理器（Msxml）

也支持 rich 数据型态，像是传统的程序语言和数据库系统中所包含的数据型态种类。本章所讨

论的数据型态，焦点放在 rich 数据型态，以及指定解析类别（parser class）用来解译数据。

Note

上述表格内的属性型态，可在<XML 1.0> 规格书的 3.3.1 节中（在随书所附的光盘片上）加以确认。

严格的型态以及松散的型态

资数据类型在两种基本关系中逐渐被了解：严格的型态（**Strong typing**）和松散的型态（**weak typing**）。就严格的型态而言，一个元素必须包含单一数据类型，元素的内容必须符合关于其型态严谨的规则。举例而言，您可能有一个被命名为 **Part** 的元素，而这个元素必须包含一个整数（**integer**），它不能包含一个字符串、日期，甚至是小数。在数据库的 API 中，数据通常是严格的型态，譬如 ODBC（Open Database Connectivity，开放式数据库连结）或 JDBC（Java Database Connectivity，JAVA 数据库连结）。

松散的型态，正如您所猜想的，允许多种数据类型存在于单一元素中。因此，如果我们的 **Part** 元素被指定为松散的型态，则它可能包含一个整数、一个字符串、一个名字、一个日期或一些数据类型组合。

指定数据类型

在 XML 文件中，你可以使用元素的 **dt:dt** 属性来指定数据类型。语法是 **dt:dt="datatype"**，其中 **datatype** 表示一个 XML 支持的数据类型。在下面的例子中，**Id** 元素是数字型态。

```
<?xml version="1.0"?>  
  
<PRODUCT xmlns:dt="urn:schemas-microsoft-com:datatypes">
```

```
<PART>

  <ID dt:dt="number">4535645.234</ID>

  <NAME>widget</NAME>

</PART>

</PRODUCT>
```

Note

dt: dt 的语法并不常见，因为数据类型是由命名空间所定义的。您可能也注意到，上面文字码第二行中的命名空间宣告了 **datatypes** 的命名空间。命名空间使用特殊的前缀来定义独特的元素和属性，这可以在本章稍后的 [<XML 命名空间>](#) 一节中看到。

下面的表格里定义了一些普通的数据型态。完整的数据型态请参阅本书 [附录 B 的<数据类型>](#)。

数据类型 态	范例	说明
boolean	0; 1 （0 为伪，1 为真）	0 或 1。

char	a	字符串（只有一个字符）。
float	34234.376; 477	可带正负号的数字或分数，基本上没有数字位数的限制，可以包含指数。
int	345	不带正负号的整数，不含指数。
number	-23; 567556; 443.34; 67E12	可带正负号的数字或分数，基本上没有数字位数的限制，可以包含指数。
String	This is a string.	#PCDATA
uri	http://mspress.microsoft.com	URI (Universal Resource Identifier)

每当一个节点被指派一个型态时，不管数据最初的格式为何，节点的数据都会遵从指定的型态。

举例而言，检视这个 **Number** 元素：

```
<NUMBER dt:dt="int">-255</NUMBER>
```

这个元素会以被指定的数据型态来处理：整数 **255**，而不会以 **-255** 被处理。当您使用数据型态

时，了解型态转换是很重要的，因为一不小心，非预期的结果就会出现。

在 Script 中使用数据型态

XML 对象模型允许 Script 存取数据型态。就像对象模型中其它的对象一样，Script 程序也可以使用数据型态对象。这个元素节点的 `dataType` 及 `nodeTypedValue` 属性允许程序设计人员可以存取内容树状结构中任何节点的数据型态。让我们看看文字码 6-1 的 XML 文件中，这些属性如何在 Script 程序中运作。

Note

在本章中，我们将以一个 XML 文件作范例，包含 **wildflower** 植物清单列表的信息。这个目录文件在随书光盘 `Chap06\Lst6_1.xml` 的档案中。在下列文字码 6-1 中将会显示其中的一部分。

```
<CATALOG xmlns:dt="urn:schemas-microsoft-com:datatypes">

  <PLANT>

    <COMMON>Bloodroot</COMMON>

    <BOTANICAL>Sanguinaria canadensis</BOTANICAL>
```

<ZONE>4</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE dt:dt="fixed.14.4">2.44</PRICE>

<AVAILABILITY dt:dt="dateTime">1999-03-15</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Columbine</COMMON>

<BOTANICAL>Aquilegia canadensis</BOTANICAL>

<ZONE>3</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE dt:dt="fixed.14.4">9.37</PRICE>

<AVAILABILITY dt:dt="dateTime">1999-03-06</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Marsh Marigold</COMMON>

```
<BOTANICAL>Caltha palustris</BOTANICAL>

<ZONE>4</ZONE>

<LIGHT>Mostly Sunny</LIGHT>

<PRICE dt:dt="fixed.14.4">6.81</PRICE>

<AVAILABILITY dt:dt="dateTime">1999-05-17</AVAILABILITY>

</PLANT>

</CATALOG>
```

文字码 6-1

接下来，我们要建立一个 HTML 网页来和 XML 文件中的数据配合。文字码 6-2（在随书光盘中的 Chap06\Lst6_2.xml）使用 XML 对象模型来浏览文件树状结构中的元素，并撷取我们所需要的数据。

Note

要更详细地了解 XML 对象模型，可以参阅附录 A 的<XML 对象模型>。

dataType 属性

在下列文字码 6-2 中，**dataType** 属性被用来取得部分节点的数据型态。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

<HEAD>

    <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

        loadDoc ( ) ;

    </SCRIPT>

    <SCRIPT LANGUAGE="JavaScript">

        var xmlDoc = new ActiveXObject ( "microsoft.xmlDOM" ) ;

        xmlDoc.load ( "Lst6_1.xml" ) ;

        function loadDoc ( )

        {
```



```
if (xmlDoc.readyState == "4")

    start () ;

else

    window.setTimeout ("loadDoc () ", 4000) ;

}


function start ()

{

    var rootElem = xmlDoc.documentElement;

    var plantNode = rootElem.childNodes.item (0) ;

    var plantLength = plantNode.childNodes.length;

    for (cl=0; cl<plantLength; cl++)

    {

        currNode = plantNode.childNodes.item (cl) ;

        switch (currNode.nodeName)

        {

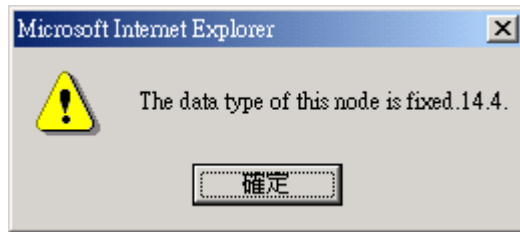
            case "PRICE":
```

```
        alert ("The data type of this node is " +  
            currNode.dataType + ".") ;  
    }  
    break;  
}  
}  
}  
  
</SCRIPT>  
  
<TITLE>Code Listing 6-2</TITLE>  
  
</HEAD>  
  
<BODY>  
  
</BODY>  
  
</HTML>
```

文字码 6-2

当执行文字码 6-2 时，**start** 函式显示出 **fixed.14.4** 这个值（如下图所示），因为 **fixed.14.4** 是

XML 文件所指定的数据型态。



NodeTypedValue 属性

这个属性是节点的型态值，当它在文件中被格式化时，可能会有不同的值。文字码 6-3（在随书光盘的 Chap06\Lst6_3.htm 档案中）使用 `nodeTypedValue` 属性来显示型态值。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

<HEAD>

    <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

        loadDoc ( ) ;

    </SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">

    var xmlDoc = new ActiveXObject ("microsoft.xmlDOM") ;

    xmlDoc.load ("Lst6_1.xml") ;


function loadDoc ()

{

    if (xmlDoc.readyState == "4")

        start () ;

    else window.setTimeout ("loadDoc () ", 4000) ;

}


function start ()

{

    var rootElem = xmlDoc.documentElement;

    var plantNode = rootElem.childNodes.item (0) ;

    var plantLength = plantNode.childNodes.length;

    for (cl=0;cl<plantLength;cl++)
```

```
{

    currNode = plantNode.childNodes.item (c1) ;

    switch (currNode.nodeName)

    {

        case "AVAILABILITY":

            alert ("The typed value of this node is " +

                currNode.nodeTypeValue + ".") ;

            break;

        }

    }

}

</SCRIPT>

<TITLE>Code Listing 6-3</TITLE>

</HEAD>

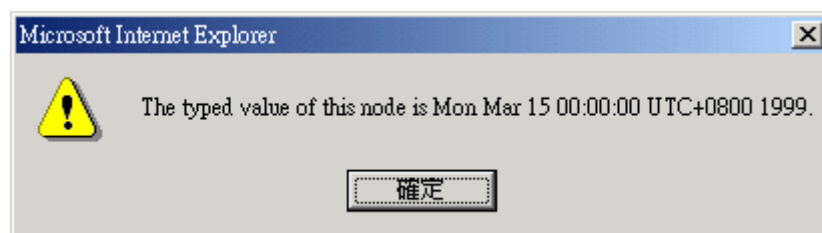
<BODY>

</BODY>
```

</HTML>

文字码 6-3

这个程序显示值为 **Mon Mar 15 00:00:00 UTC+0800 1999** 而非 **1999-03-15**（如下图所示），因为被指定的型态是 **dateTime**。虽然当 XML 文件被显示时，资料可以用日期显示（没有时间），但节点的 **nodeTypedValue** 属性是它的型态值，而不是只有日期。



改变数据型态

您可以经由处理称为数据型态的 **coercion** 或 **casting** 来改变一个元素或属性的数据型态。然而，您只能将基本数据型态转换成 **rich** 数据型态。XML 并不支持两个不同 **rich** 数据型态间的转换。

要改变一个元素或属性的数据型态，您只要设定它的 **dataType** 属性为不同的 **rich** 数据型态即可。

您也可以使用 **dataType** 属性来恢复一个元素或属性目前的 **rich** 数据类型态。注意 **dataType** 属性

只能被用来恢复一个 **rich** 数据类型态。如果一个元素或属性是基本数据类型态，它的 **dataType** 属性

是 **null** 值。文字码 6-4（随书光盘中的 Chap06\Lst6_4.htm 档案）是这个例子的文字码。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

    <HEAD>

        <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

            loadDoc ( ) ;

        </SCRIPT>

        <SCRIPT LANGUAGE="JavaScript">

            var xmlDoc = new ActiveXObject ( "microsoft.xmlDOM" ) ;

            xmlDoc.load ( "Lst6_1.xml" ) ;

            function loadDoc ( )
```

```

    {

        if (xmlDoc.readyState == "4")

            start () ;

        else

            window.setTimeout ("loadDoc () ", 4000) ;

    }


    function start ()

    {

var rootElem = xmlDoc.documentElement;

var plantNode = rootElem.childNodes.item (0) ;

var plantLength = plantNode.childNodes.length;

for (cl=0; cl<plantLength; cl++)

    {

        currNode = plantNode.childNodes.item (cl) ;

        switch (currNode.nodeName)

            {

```



```
        case "ZONE":

            alert ("The data type of this node is " +

                currNode.dataType + ".") ;

            currNode.dataType = "number";

            alert ("The new data type of this node is " +

                currNode.dataType + ".") ;

            break;

        }

    }

}

</SCRIPT>

<TITLE>Code Listing 6-4</TITLE>

</HEAD>

<BODY>

</BODY>
```

</HTML>

文字码 6-4

这个例子改变了 **Zone** 元素的数据型态，由它原本预设的基本数据型态（**#PCDATA**）变成数字（如右图所示）。处理器会视这个元素的值为数字值来取代文字。



下表列出 **Msxml** 必须支持的 **rich** 数据型态。请注意，其中有些型态可能不被其它的处理器支持。

rich 数据型态 名称	说明
bbin. base64	MIME-styleBase64 编码二进制区段。
bin. hex	以十六进制数字表现八进位。
Boolean	0（假） 或 1（真）。

char	一个字节字符串。
date	ISO 8601 格式的子集合日期格式，不含时间的日期，例如：1998-11-02。
dateTime	ISO 8601 格式的子集合日期格式，可包含时间的日期但不含时区，计算到秒数 字数，例如：1988-07-09T18:39:09。
dateTime.tz	ISO 8601 格式的子集合日期格式，可包含时间及时区，计算到秒数 字数，例如：1988-07-09T18:39:09-08:00。
fixed.14.4	与数字相同，但是支持小数点左边 14 位数，小数点右边则最多只能有 4 位数。
float	浮点数主要是由整数字数、小数字数和指数三部分组成，所能表示的范围从 1.7976931348623157E+308 到 2.2250738585072014E-308。
int	为带符号位（表正负数）的数字，不含分数与指数。
number	无表示范围限制，可包含前置符号、小数，与指数。
time	ISO 8601 格式的子集合时间格式，不包含日期和时区，例如：06:18:35。
time.tz	ISO 8601 格式的子集合时间格式，不包括日期，例如：03:15:25-04:00。
i1	可带正负号的数字（1byte），但不含小数及指数，例如：1，34，-165。
i2	可带正负号的数字（2byte），但不含小数及指数，例如：1，244，-56344。

i4	可带正负号的数字（4byte），但不含小数及指数，例如：1, 556, -34234, 156645, -2005000700。
i8	可带正负号的数字（8byte），但不含小数及指数，例如：1, 646, -65333, 2666345433454, -2007000800090090。
r4	与 number 的数据型态相似，包含前置符号、小数，及指数，可表示的范围从 3.40282347E+38F 到 1.17549435E-38F。
r8	与 float 的数据型态相同。
ui1	不含正负号、小数，及指数的整数（1byte），例如：1, 255。
ui2	不含正负号、小数，及指数的整数（2byte），例如：1, 255, 65535。
ui4	不含正负号、小数，及指数的整数（4byte），例如：1, 660, 2005000000。
ui8	不含正负号、小数，及指数的整数（8byte），例如：1582437474934。
uri	URI（Universal Resource Identifier），例如： urn:schemasflowers-com:wildflowers。
uuid	以十六进制数字表现八进位，并使用连接符号（-），例如： 333C7BC4-460F-11D0-BC04-0080C7055A83。

XML 命名空间 (Namespaces)

若要更进一步了解 XML 并将它作为数据来源，必须先了解命名空间 (Namespaces) 这个概念。

就像您已经看到的：XML 文件的数据 (内容) 是透过检视文件中单一节点取得。这是因为 XML

文件的阶层结构、有效的规则，以及标准的规范，因此保证 XML 文件内的每个节点都是独一无

二的。反过来说，任何一个节点都只有一个参照。然而当您在多重文件的环境下使用 XML 文件

时，潜在的问题发生了。举例来说，两个 (或更多) 文件可能包含同样名字的元素但是不同语义。

(并且文件可能使用同样的方法建构。) 如果在单一环境下使用这两个文件，重迭的元素将会产

生混淆。请看下列的 XML 文字码：

```
<AUTOMOBILE>
```

```
  <ID>232-HDF</ID>
```

```
<?AUTOMOBILE>
```

```
<DOG>
```

```
  <ID>Rover</ID>
```

```
</DOG>
```

这里的 **Automobile** 元素和 **Dog** 元素都包含一个 **Id** 元素，但是这个 **Id** 元素在不同情况下有不同的意义。这些元素来自不同的来源，但都被结合到一个单一文件中，如下面的文字码所示，这 **Id** 元素将会失去它的意义。

```
<FAMILY>
```

```
  <MOM>
```

```
    </MOM>
```

```
  <DAD>
```

```
    </DAD>
```

```
  <KIDS>
```

```
    </KIDS>
```

```
  <AUTOMOBILE>
```

```
    <ID>232-HDF</ID>
```

```
  </AUTOMOBILE>
```

```
<DOG>

  <ID>Rover</ID>

</DOG>

</FAMILY>
```

这是一个非常实际的问题，而且如果 XML 在 Web 和组织中被广泛地使用，这样的问题会变得更普遍。解决方案就是 XML 命名空间（Namespaces），它提供了一个建立名称的方法，无论元素在哪里被使用，都可以保持独一无二的名字。

经由 XML 命名空间来命名

命名空间在传统程序设计中的说法是表示名字的集合，而在这集合之中不会有重复的情形发生。

因为 XML 允许文件作者定义他们自己的标签集，这很可能在 XML 文件中导致名字的重复，在

XML 中的命名空间增加了一些内涵。在 XML 中的命名空间是一种方法论，使用独一无二的外部

资源来确认元素名称，然后在 XML 文件中建立独一无二的名称。因此，在 XML 中的命名空间

是一个名字的集合，并且是透过 Uniform Resource Identifier（URI）来确认的。一个命名空间

可以是有效的或是无效的。

有效的名称

在 **XML** 中，一个有效的名称由两个部分组成：命名空间的名称和本地端的部分。命名空间的名称是一个 **URI**；而本地端的部分则是本地文件元素或属性名字。因为 **URI** 是独一无二的，命名空间的名称和本地端部分的组合也建立了一个独一无二的元素名称。为了要在 **XML** 文件中使用命名空间，必须在文件的前言中包含命名空间宣告。命名空间的前缀也可以包含在宣告中，然后，您可以利用冒号将前缀和本地端的部分连结起来成为命名空间。

让我们来看一个范例。在下列的文件中，我们以前缀宣告了两个命名空间，然后在文件中使用命名空间。

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

<?xml:namespace ns="http://inventory/schema/ns" prefix="inv"?>

<?xml:namespace ns="http://wildflowers/schema/ns" prefix="wf"?>

<PRODUCT>

    <PNAME>Test1</PNAME>

    <inv:quantity>1</inv:quantity>

    <wf:price>323</wf:price>
```


<DATE>6/1</DATE>

<PRODUCT>

在上面的文字码中，前缀被用来识别所选命名空间中的元素。这个结果不仅名称是独一无二的，而且语义的保存也十分良好。**inv:quantity** 和 **wf:price** 元素现在包含完全有效的名称，无论使用在什么地方，它们都将是独一无二的。前缀是元素名称的一部分，而且包含象征命名空间的元素。

Note

本章中所有的命名空间，及稍早所讨论过 **datatypes** 的命名空间，都只是范例而已。**Datatypes** 的命名空间实际存在并且可以使用在<指定数据类型>中所讨论的方式。

无效的名称

无效的名称是没有命名空间的名称组合。典型的 XML 元素名称都是无效的，因为它们都没有指定一个命名空间。举例来说，在下列 XML 文字码中，所有的元素名称都是无效的，而且不是独一无二的。

```
<PRODUCT>

  <NAME>Bloodroot</NAME>

  <QUANTITY>10</QUANTITY>

</PRODUCT>
```

命名空间的范围

前言并不是命名空间宣告唯一的地方。相反的，您可以直接在元素中纳入命名空间宣告，那是命名空间的一部分。当元素第一次使用时，简单地纳入宣告即可，如下列所示：

```
<PRODUCT>

  <PNAME>Test1</PNAME>

  <inv:quantity>1</inv:quantity>

  <wf:price
```

```
xmlns:wf="urn:schemas-wildflowers-com:xml-prices">

    323

</wf:price>

<DATE>6/1</DATE>

</PRODUCT>
```

在元素的内容中命名空间已为有效的，换句话说，该元素和它所有的子元素都能使用这个命名空间。如果命名空间在文件元素中被宣告，那命名空间就能在整个文件中使用。

预设的命名空间

透过没有指定前缀的宣告，您可以预设命名空间。在这个例子中，假设在元素内容中的命名空间已经被宣告过，如下面文字码所示:

```
<CATALOG>

    <INDEX>

        <ITEM>Trees</ITEM>

        <ITEM>Wildflowers</ITEM>
```

```
</INDEX>

<PRODUCT xmlns:wf="urn:schemas-wildflowers-com">

  <NAME>Bloodroot</NAME>

  <QUANTITY>10</QUANTITY>

  <PRICE>$2.44</PRICE>

</PRODUCT>

</CATALOG>
```

在上述文字码中的 **Product** 元素，包含一个没有前缀的命名空间宣告，如此，这个命名空间就被假设给 **Product** 元素和它所有的子元素，但不是给任何 **Product** 元素以外的元素。唯一的例外发生在：当子元素包含另一个命名空间宣告时，就会重写预设的命名空间宣告。下面的文字码显示一个范例：

```
<CATALOG>

  <INDEX>

    <ITEM>Trees</ITEM>

    <ITEM>Wildflowers</ITEM>
```

```
</INDEX>

<PRODUCT xmlns:wf="urn:schemas-wildflowers-com">

  <NAME>Bloodroot</NAME>

  <inv:quantity

    xmlns:wf="urn:schemas-wildflowers-com:xml-inventory">

    10

  </inv:quantity>

  <PRICE>$2.44</PRICE>

</PRODUCT>

</CATALOG>
```

在这里，预设的命名空间重写成为 `inv:quantity` 元素。

宣告命名空间为 **URL 或 URN**

由于所有的 **URL** 都是独一无二的，因此可以用来提供独一无二的命名空间名称，如果一个命名空间对应（**mapped**）到一个 **URL**，整个内容中凡是用到命名空间的地方，这个命名空间都会是独一无二的。

另一种情况是：可能有一个命名空间结构（**schema**）已经存在，用来确认命名空间中的名称与它们该如何被建构。不过，这并不是 **XML** 命名空间的名称提供取得结构（**schema**）机制的目的。

Note

结构（**Schema**）是一种文件的定义，类似 **DTD**，但使用一种特别的 **XML** 语汇称为 **XML-Data**。[第 10 章](#) 会详细讨论结构（**Schema**）和 **XML-Data**。对于命名空间而言，结构（**Schema**）可以用来定义所有包含在命名空间内的名称。举例来说，一个结构（**Schema**）可能定义 **dt** 为前置前缀，并对应到命名空间的名称，同时定义 **string** 为命名空间内的名称。在这个例子中，元素名称 **dt:string** 就能在 **XML** 文件中使用。

URN 提供了一种机制，用来设置与取得定义特殊命名空间的结构（Schema）档案。有点像是透过 URL 提供功能，对于这个目标来说，URN 是更健全且更易于管理的，因为 URN 可以指向多个 URL。

Note

更多 URN 的相关信息，请参

阅 <http://www.ncsa.uiuc.edu/InformationServers/Horizon/URN/urn.html>

下列的文字码示范如何在 XML 的命名空间中使用 URN：

```
<CATALOG>
```

```
  <INDEX>
```

```
    <ITEM>Trees</ITEM>
```

```
    <ITEM>Wildflowers</ITEM>
```

```
  </INDEX>
```

```
<wf:product xmlns:wf="urn:schemas-wildflowers-com">

  <wf:name>Bloodroot</wf:name>

  <QUANTITY>10</QUANTITY>

  <PRICE>$2.44</PRICE>

</wf:product>

</CATALOG>
```

这里有一个命名空间的结构（**Schema**），可能在 **URN** 所指定的地址中找到，而且处理的应用程序便会知道该如何取得结构（**Schema**），这个结构（**Schema**）会详细地定义命名空间中可能在文件里使用的元素细节。

属性的命名空间

命名空间可以应用在属性和元素中，并且让它变得更简单，两者的命名机制是相同的，如下面所示：

```
<wf:product TYPE="plant" class:kingdom="plantae"

  xmlns:wf="urn:wildflowers:schemas:product"
```



```
xmlns:class="urn:bio:botany:classification">

<PNAME>Test1</PNAME>

<QUANTITY>1</QUANTITY>

<PRICE>323</PRICE>

<DATE>6/1</DATE>

</wf:product>
```

在这个例子中，`wf:product` 元素和 `class:kingdom` 属性名称，在命名空间宣告中被组合，这个属

性命名空间使用了与元素命名空间相同的方式来命名。

随着新的语汇和以 XML 为基础的技术逐渐发展，命名空间的重要性将与日遽增，已经有一些技

术仰赖命名空间，譬如 XML-Data、XML 数据型态，和 SMIL (Synchronized Multimedia Integration

Language: 同步多媒体整合语言)。 [第 10 章](#) 包括了如何藉由 XML-Data 来使用 XML 命名空

间的实际例子。

使用 XML 数据来源对象 (Data Source Object, DSO)

到本章为止，我们已经使用了 **XML** 文件作为资料来源，但我们是将文件载到处理器中，循行文件树状的路径，并且一次一个节点地运用数据作业。这种技术已经证明了：您运用跟处理数据库记录的相同方式来运作 **XML** 文件。然而，还有其它的方式可以运用 **XML** 资料，如使用 **XML DSO**，您可以将数据链路到 **HTML** 网页上的控件中，这会让您一次一个节点地运用数据作业，如果您希望的话。但您也可以采取区块（**chunk**）的方式来处理资料，或者不遵循文件树状一次多个节点来处理数据。

Note

微软的 **IE4.0** 或以上的版本才支持 **XML DSO**。

一次使用一笔记录

使用 **XML DSO** 和对象模型，您也可以继续循着 **XML** 文件树状路径，用与以往相同的方式来处理文件，不过，差别在于：数据会结合到网页上特定的控件，而这种控件会自动地撷取来自 **DSO** 的数据。让我们看看它如何在文字码 **6-5** 中运作（见随书光盘 **Chap06\Lst6_5.htm**）。

Note

文字码 6-5 使用的是 `Lst6_1.xml` 修正过的版本，称为 `Lst6_1a.xml`，这里没有包含任何来自于数据类型命名空间的数据型态。直到撰写本文时（使用 `Msxml` 的 `beta` 版本），`DSO` 还没有支持命名空间，我们预期在正式版中将会支持命名空间。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

  <HEAD>

    <SCRIPT FOR="window" EVENT="onload">

      var xmlDso = xmlDso.XMLDocument;

      xmlDso.load ("Lst6_1a.xml") ;

    </SCRIPT>

    <TITLE>Code Listing 6_5</TITLE>
```

</HEAD>

<BODY>

<OBJECT WIDTH="0" HEIGHT="0"

CLASSID="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"

ID="xmldso">

</OBJECT>

<DIV ID="catalog">

Common Name:

<SPAN ID="common" DATASRC=#xmldso

DATAFLD="COMMON" STYLE="color:blue">

Botanical Name:

<SPAN ID="botan" DATASRC=#xmldso

DATAFLD="BOTANICAL" STYLE="color:blue">

Zone:

<SPAN ID="zone" DATASRC=#xmldso

DATAFLD="ZONE" STYLE="color:blue">

Light Needs:

<SPAN ID="light" DATASRC=#xmldso

DATAFLD="LIGHT" STYLE="color:blue">

Price:

<SPAN ID="price" DATASRC=#xmldso

```

        DATAFLD="PRICE" STYLE="color:blue">

</SPAN>

<BR>

</DIV>

<P>

<INPUT TYPE=button VALUE="Previous Plant"

ONCLICK="xmldso.recordset.moveprevious () ">

<INPUT TYPE=button VALUE="Next Plant"

ONCLICK="xmldso.recordset.movenext () ">

</BODY>

</HTML>

```

文字码 6-5

文字码 6-5 将 Lst6_1a.xml 文件加载到 XML DSO，并且将各种不同的数据元素连结到 HTML 网页中的 Span 元素。请注意，Span 元素显示出我们想要展示的每一个元素，并对应到 XML 文件

中的元素，举例而言，下面的文字码将源自 DSO 的 Price 元素连结到 HTML 网页中称为 price 的 Span 元素。

```
<SPAN ID="price" DATASRC=#xmldso  
  
    DATAFLD="PRICE" STYLE="color:blue">  
  
</SPAN>
```

DATASRC 属性指定了 DSO 的使用（接在#后的数据来源对象名称），而 DATAFLD 属性指定提供数据的元素。经由这种方法，数据就结合到一些控件或 HTML 元素中，结果如图 6-1 所示。

单一值元素

在我们的范例中，都只使用一个记录集（recordset），或元素数据集（element data set），这是由于 HTML 元素的特性所导致的。大部分的元素像是 Span 元素，都是单一值元素，也就是说，在任何时刻仅有一部分数据能够结合到这个元素。因为一次仅能使用一部分数据，在使用中的数据就被认为是目前的记录集。您可能也从文字码 6-5 中注意到，在记录集中，每个值均需要一个独立的 Span 元素。

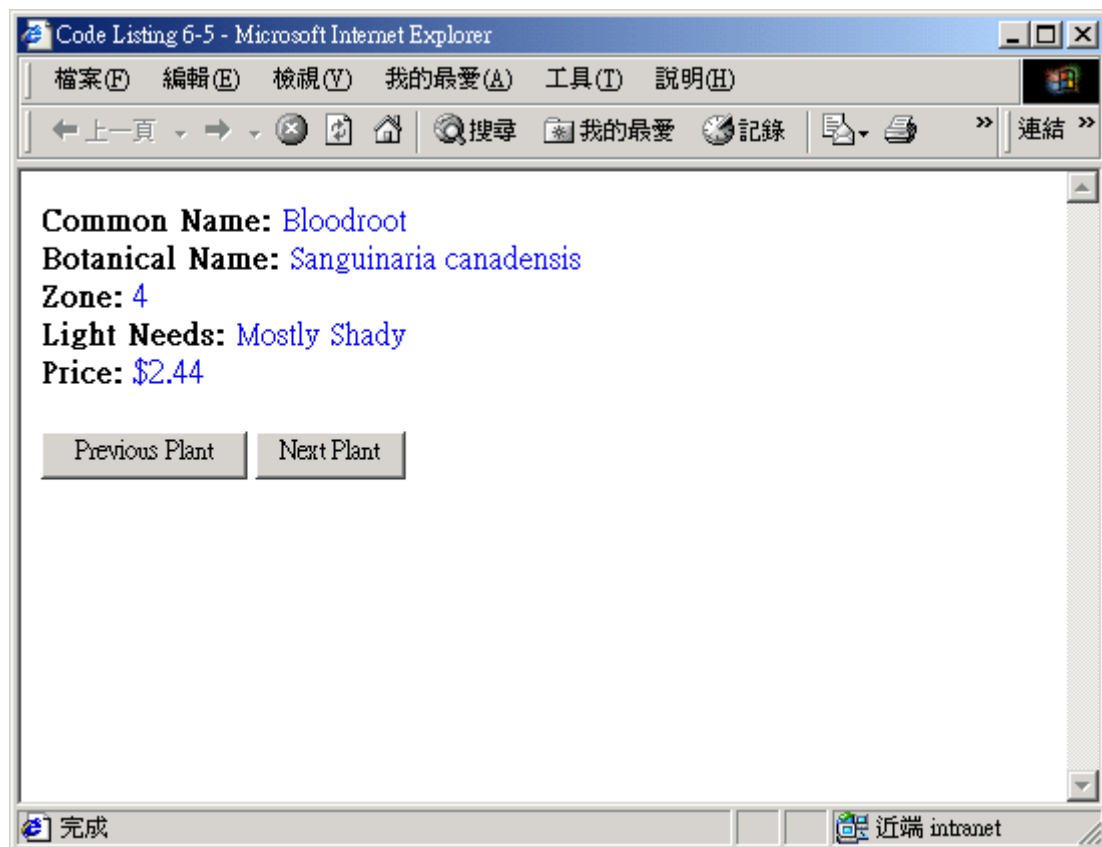


图 6-1：使用 XML DSO 在 HTML 网页上显示 XML 资料

透过记录集移动

正如您在图 6-1 所看到的，HTML 网页中含有两个按钮，让使用者可以按下这两个按钮来实际操作文件。这些按钮结合 DSO 中的方法，在数据间向前或向后移动：`recordset.movePrevious` 方

法是在记录集中移动到目前记录的前一笔数据，而 `recordset.moveNext` 方法则是在记录集中移动到当前记录的最后一笔数据。您可能注意到，使用 DSO 让我们只要写少量的 Script 文字码，即能获得相当多的功能。

在数据来源文件检视所有的数据

您可以使用 XML DSO 来检视数据来源文件里所有的数据，而不需要循行文件树状路径来撷取数据。您可以使用方才讨论过的相同资料结合技术，不过这一次，数据是结合到含有多重值的元素，而非数个单一值的元素群。就目前而言，HTML 唯一可含有多重值元素的是表格（Table）元素。文字码 6-6（见随书光盘 Chap06\Lst6_6.htm）显示数据如何能结合到一个表格中，来检视 XML 文件中全部的内容。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

  <HEAD>

    <SCRIPT FOR="window" EVENT="onload">

      var xmlDso = xmlDso.XMLDocument;
```

```
xmlDso.load ("Lst6_1a.xml") ;
```

```
</SCRIPT>
```

```
<TITLE>Code Listing 6_6</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<OBJECT WIDTH="0" HEIGHT="0"
```

```
CLASSID="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"
```

```
ID="xmldso">
```

```
</OBJECT>
```

```
<TABLE DATASRC=#xmldso BORDER="1"
```

```
CELLSPACING="5" CELLPADDING="2">
```

```
<THEAD>
```

```
<TH>Common Name</TH>
```

<TH>Botanical Name</TH>

<TH>Zone</TH>

<TH>Light</TH>

<TH>Price</TH>

<TH>Availability</TH>

</THEAD>

<TR ALIGN="center">

<TD><DIV DATAFLD="COMMON"></TD>

<TD><DIV DATAFLD="BOTANICAL"></TD>

<TD><DIV DATAFLD="ZONE"></TD>

<TD><DIV DATAFLD="LIGHT"></TD>

<TD><DIV DATAFLD="PRICE"></TD>

<TD><DIV DATAFLD="AVAILABILITY"></TD>

</TR>

</TABLE>

```
</BODY>

</HTML>
```

文字码 6-6

在这个例子中，数据结合到表格的字段中。因为表格元素是一个多重值元素，所以表格是基于可取得数据数量多寡而实时建立的。结果产生一个包含文件中所有数据的表格，如图 6-2 所示。

Note

基于执行效率的理由，表格是以异步方式建立的。若资料集（**dataset**）的数量很大，在显示表格之前，会在内存中建立一个完整的表格，此时可能会导致一段长时间的延迟。



Common Name	Botanical Name	Zone	Light	Price	Availability
Bloodroot	Sanguinaria canadensis	4	Mostly Shady	\$2.44	031599
Columbine	Aquilegia canadensis	3	Mostly Shady	\$9.37	030699
Marsh Marigold	Caltha palustris	4	Mostly Sunny	\$6.81	051799
Cowslip	Caltha palustris	4	Mostly Shady	\$9.90	030699
Dutchman's-Breeches	Diecentra cucullaria	3	Mostly Shady	\$6.44	012099

图 6-2：使用 XML DSO 和 HTML 表格元素来显示 XML 数据。

在 XML 中使用阶层数据

使用 XML 当作数据来源时，其中一个最大的好处是：XML 允许您使用阶层数据，而不像许多以网络为基础的标准数据来源。因此，除了能运用栏和列作业外，您也可以运用完整的 XML 文件树状结构。为了证明这一点，我们将重新组织 **wildflower** 文件，如文字码 6-7 所示（见随书光盘 Chap06\Lst6_7.xml）。

<CATALOG>

<REGION>

<ZONE>3</ZONE>

<PLANT>

<COMMON>Columbine</COMMON>

<BOTANICAL>Aquilegia canadensis</BOTANICAL>

<LIGHT>Mostly Shady</LIGHT>

</PLANT>

<PLANT>

<COMMON>Dutchman' s-Breeches</COMMON>

<BOTANICAL>Diecentra cucullaria</BOTANICAL>

<LIGHT>Mostly Shady</LIGHT>

</PLANT>

<PLANT>

<COMMON>Ginger, Wild</COMMON>

<BOTANICAL>Asarum canadense</BOTANICAL>

<LIGHT>Mostly Shady</LIGHT>

</PLANT>

</REGION>

<REGION>

<ZONE>4</ZONE>

<PLANT>

<COMMON>Bloodroot</COMMON>

<BOTANICAL>Sanguinaria canadensis</BOTANICAL>

<LIGHT>Mostly Shady</LIGHT>

</PLANT>

<PLANT>

<COMMON>Marsh Marigold</COMMON>

<BOTANICAL>Caltha palustris</BOTANICAL>

```

    <LIGHT>Mostly Sunny</LIGHT>

  </PLANT>

  <PLANT>

    <COMMON>Cowslip</COMMON>

    <BOTANICAL>Caltha palustris</BOTANICAL>

    <LIGHT>Mostly Shady</LIGHT>

  </PLANT>

</REGION>

</CATALOG>

```

文字码 6-7

要注意在这个文字码中，**plant** 是以区块的方式来组织，因此加入另一个「阶层」到文件树状结构上。**XML DSO** 允许我们从树状结构的任一个阶层来结合数据，这使得检视数据时，产生高度的复杂性。为了显示文字码 6-7 中的资料，我们将制作一个 **HTML** 网页，这个网页包含了三个巢状表格，每一个表格连结到文件树状结构中的不同阶层，这个 **HTML** 的文字码如下所示（见随书光盘 Chap06\Lst 6_8.htm）。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```



```
<HTML>
```

```
<HEAD>
```

```
<SCRIPT FOR="window" EVENT="onload">
```

```
var xmlDso = xmldso.XMLDocument;
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

```
<HEAD>
```

```
<SCRIPT FOR="window" EVENT="onload">
```

```
var xmlDso = xmldso.XMLDocument;
```

```
xmlDso.load ("Lst6_7.xml") ;
```

```
</SCRIPT>
```

```
<TITLE>Code Listing 6-8</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

<OBJECT WIDTH="0" HEIGHT="0"

CLASSID="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"

ID="xmldso">

</OBJECT>

<TABLE CELSPACING="6" ID="catalog">

<TR>

<TD>

<TABLE BORDER="2" DATASRC=#xmldso>

<TR>

<TD STYLE="font-size:18; font-weight:bold">

Zone:

</TD>

</TR>

<TR>

<TD>

<TABLE DATASRC=#xmldso DATAFLD="PLANT">

<THEAD ALIGN="left">

<TH>Common Name</TH>

<TH>Botanical Name</TH>

<TH>Light Requirement</TH>

</THEAD>

<TR>

<TD><DIV DATAFLD="COMMON"></TD>

<TD><DIV DATAFLD="BOTANICAL"></TD>

<TD><DIV DATAFLD="LIGHT"></TD>

</TR>

</TABLE>

</TD>

</TR>

</TABLE>

</TD>

```
</TR>

</TABLE>

</BODY>

</HTML>
```

文字码 6-8

要注意的是：对于我们想要循行的阶层，我们都插入一个 **DATASRC** 属性。当我们想要撷取数据时，我们使用 **DATAFLD** 属性，结果产生一组自动展示阶层组织数据的巢状表格，如图 6-3 所显示的。

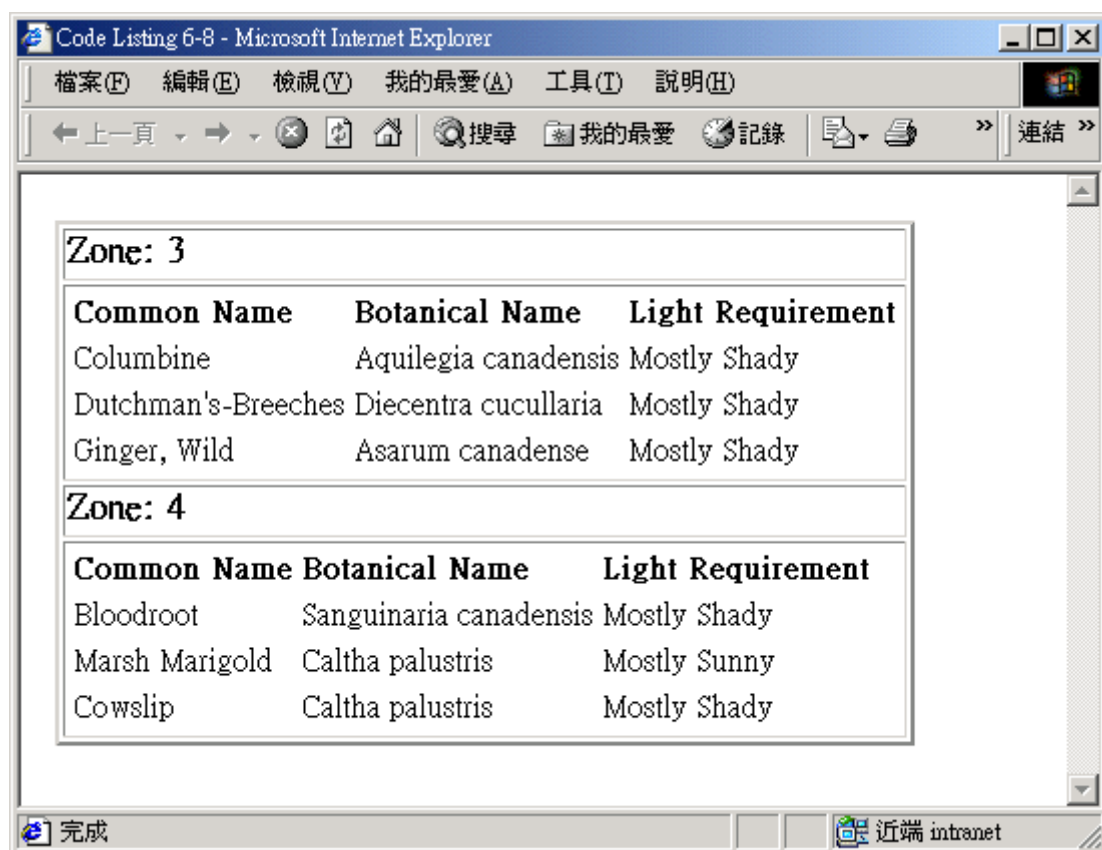


图 6-3: 一组结合到 XML 数据的巢状表格

整合所有的观念

在本节中，我们将把我们已讨论过的数据来源观念结合在一起，来建立应用于 Web 上且以 XML 为基础的应用程序，即一般所熟知的「Web 应用程序」（weblication）。我们的 Web 应用程序将应用 wildflower 文件的数据到一个在线商店，这个在线商店允许使用者将他们的名字加到数据储存区、选择植物，将植物加入购物单中，并可检视目前的购物单。所有的植物数据都将来自于

Lst6_1a.xml 档案，同时所有的使用者数据将储存在独立的 XML 档案中，这个档案仅包含使用者的信息。我们的在线商店将透过 HTML 文件来操作，此文件使用 XML 对象模型来读取和写入数据。在这个范例中，我们将同时使用 XML DSO 和标准 XML 解析器。

首先，让我们来看看完成这个例子所需要的文件，并透过文字码来了解整个程序是如何运作的。

您已经知道我们需要的第一个文件，此文件中包含 **wildflower** 所有的植物信息。还记得在文字码

6-5 中使用的 Lst6_1a.xml 档案吗？

下一个部分，我们需要的是 XML 文件，其架构视使用者、顾客及信息而定。这份文件如文字码

6-9 所示（于随书光盘 Chap06\Lst 6_9.xml 档案中）。

```
<CUSTOMERS>

  <CUSTOMER>

    <CNAME></CNAME>

    <PNAME></PNAME>

    <QUANTITY></QUANTITY>

    <PRICE></PRICE>

    <DATE></DATE>

  </CUSTOMER>
```

```
</CUSTOMERS>
```

文字码 6-9

请注意：这个文件尚未包含任何数据，当我们开始营业时，我们会把数据加到这个文件结构中。

每一位顾客的数据都被有系统地放在 **Customer** 元素中，包含顾客的名称（**Cname** 元素），每

一个 **Customer** 元素也提供一个产品信息，包括产品的名称（**Pname** 元素）、数量（**Quantity**

元素）、价格（**Price** 元素）和购买日期（**Date** 元素）。整个网络商店的最后一个部分是 **HTML**

文件，此文件包含所有执行在线商店和展示数据会用到的逻辑。这个 **HTML** 如文字码 6-10 所示

（在随书光盘 Chap06\Lst 6_10.htm 中）。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

```
<HEAD>
```

```
<STYLE>
```

```
.category {font-weight:bold; font-size:14}
```

```
.fdata {font-size:16; color:#000099}
```

```
.pdata {font-size:14; font-weight:bold}
```

```
.cdata {font-size:16; color:#993300}

</STYLE>


<SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="onload">

var xmlDso = xmldso.XMLDocument;

xmlDso.load ("Lst6_1a.xml") ;

var xmlDso2 = xmldso2.XMLDocument;

xmlDso2.load ("Lst6_9.xml") ;

</SCRIPT>

<SCRIPT SRC=ShowXML.js>

</SCRIPT>


<SCRIPT LANGUAGE="JavaScript">

function doMenu ()

{

    if (fmenu.style.display == "none")

        fmenu.style.display = "";
```



```
else

    fmenu.style.display = "none";

}


function goRecord (indexNum)

{

var row = window.event.srcElement;

var c = row.recordNumber - 1;

xmldso.recordset.MoveFirst () ;

while (c > 0)

{

    xmldso.recordset.MoveNext () ;

    c = c - 1;

}

doMenu () ;

}
```

```
function mouseHover (state)

{

    var row = window.event.srcElement;

    var colorChange = ( (state == "over") ? "#ffff00" : "" ) ;

    row.style.backgroundColor = colorChange;

}
```

```
function updateList ()

{

    var custVal = custName.value;

    if (custVal == "")

        alert ("You must enter a customer name.")

    else

        addToList () ;

}
```

```
function addToList ()
```

```

        {

var pmatch = 0;

var cmatch = 0;

var rootElem = xmldso2.XMLDocument.documentElement;

var rootChild = rootElem.childNodes;

var childNum = rootChild.length;

var currDate = new Date ( ) ;

fullDate = (currDate.getMonth ( ) + 1) + "/";

fullDate += currDate.getDate ( ) + "/";

fullDate += currDate.getFullYear ( ) ;


for (i = 0; i < childNum; i++)

    {

        var currNode = rootChild.item (i) ;

        if (currNode.nodeName == "CUSTOMER")

            {

var custChild = currNode.childNodes;

```

```
var cnameNode = custChild.item (0) ;

var pnameNode = custChild.item (1) ;

if (cnameNode.text == custName.value)

    {

        cmatch = 1;

    }

else

    {

        cmatch = 0;

    }

if (pnameNode.text == common.innerText)

    {

        pmatch = 1;

    }

else

    {

        pmatch = 0;
```

```

    }

}

if (cmatch == 1 && pmatch == 1)

{

    xmldso2.recordset.moveFirst ( ) ;

    for (ds = 0; ds <= i; ds++)

    {

        if (ds != i)

        {

            xmldso2.recordset.moveNext ( ) ;

        }

    }

    break;

}

}

```

```

if (cmatch != 1 && pmatch != 1)

```

```

        {

xmlldso2.recordset.AddNew ( ) ;

xmlldso2.recordset ("CNAME") = custName.value;

xmlldso2.recordset ("PNAME") = common.innerText;

xmlldso2.recordset ("QUANTITY") ="1";

xmlldso2.recordset ("PRICE") = price.innerText;

xmlldso2.recordset ("DATE") = fullDate;

        }

else

        {

            if (cmatch == 1 && pmatch == 1)

                {

                    xmlldso2.recordset ("QUANTITY") =

                        parseInt (xmlldso2.recordset ("QUANTITY") ) + 1;

                }

            else

                {

```

```
if (cmatch != 1)

{

xmldso2.recordset.AddNew ( ) ;

xmldso2.recordset ("CNAME") = custName.value;

xmldso2.recordset ("PNAME") = common.innerText;

xmldso2.recordset ("QUANTITY") ="1";

xmldso2.recordset ("PRICE") = price.innerText;

xmldso2.recordset ("DATE") = fullDate;

}

else

{

if (cmatch == 1)

{

xmldso2.recordset.AddNew ( ) ;

xmldso2.recordset ("CNAME") = custName.value;

xmldso2.recordset ("PNAME") = common.innerText;

xmldso2.recordset ("QUANTITY") ="1";
```

```

        xmldso2.recordset ("PRICE") = price.innerText;

        xmldso2.recordset ("DATE") = fullDate;

    }

}

}

}

showList ();

}

function showList ()

{

var hold = 0;

var rootElem = xmldso2.XMLDocument.documentElement;

var rootChild = rootElem.childNodes;

var childNum = rootChild.length;

var purTable = "<TABLE CELSPACING='5'><THEAD ALIGN='center'>" +

    "<TH>Name</TH><TH>Quantity</TH><TH>Date</TH></THEAD>"

```



```
for (i = 0; i < childNum; i++)

{

var currNode = rootChild.item (i) ;

if (currNode.nodeName == "CUSTOMER")

{

var custChild = currNode.childNodes;

var custNum = custChild.length;

for (ci = 0; ci < custNum; ci++)

{

var currCustNode = custChild.item (ci) ;

if (currCustNode.nodeName == "CNAME")

{

var currCustName = custName.value;

if (currCustNode.text == currCustName)

{

hold = 1;

}

}
```

```
        else

            {

                hold = 0;

            }

        }

    if (currCustNode.nodeName == "PNAME")

        {

            if (hold == 1)

                {

                    purTable = purTable + "<TR ALIGN='center'><TD>" +

                        currCustNode.text + "</TD>";

                }

        }

    if (currCustNode.nodeName == "QUANTITY")

        {

            if (hold == 1)

                {
```

```

        purTable = purTable + "<TD>" + currCustNode.text
+ "</TR>";

    }

}

}

if (currCustNode.nodeName == "DATE")

{

    if (hold == 1)

    {

        purTable = purTable + "<TD>" + currCustNode.text
+ "</TR>";

    }

}

}

}

}

purTable = purTable + "</TABLE>"

PTData.innerHTML = purTable

```

```
}
```

```
</SCRIPT>
```

```
<TITLE>Code Listing 6-10</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<OBJECT WIDTH="0" HEIGHT="0"
```

```
CLASSID="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"
```

```
ID="xml_dso">
```

```
</OBJECT>
```

```
<OBJECT WIDTH="0" HEIGHT="0"
```

```
CLASSID="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"
```

```
ID="xml_dso2">
```

```
</OBJECT>
```

```
<TABLE STYLE="position:absolute; left:10; top:10"
```

```
CELLSPACING="6" ID="catalog">
```

<TR>

<TD ALIGN="right" CLASS="category">Common Name:</TD>

<TD><DIV CLASS="fdata" ID="common"

DATASRC=#xml dso DATAFLD="COMMON">

</TD>

</TR>

<TR>

<TD ALIGN="right" CLASS="category">Botanical Name:</TD>

<TD><DIV CLASS="fdata" ID="botan"

DATASRC=#xml dso DATAFLD="BOTANICAL">

</TD>

</TR>

<TR>

<TD ALIGN="right" CLASS="category">Zone:</TD>

<TD><DIV CLASS="fdata" ID="zone"

DATASRC=#xmldso DATAFLD="ZONE">

</TD>

</TR>

<TR>

<TD ALIGN="right" CLASS="category">Light Needs:</TD>

<TD><DIV CLASS="fdata" ID="light"

DATASRC=#xmldso DATAFLD="LIGHT">

</TD>

</TR>

<TR>

<TD ALIGN="right" CLASS="category">Price:</TD>

<TD><DIV CLASS="fdata" ID="price"

DATASRC=#xmldso DATAFLD="PRICE">

</TD>

</TR>

</TABLE>

<DIV STYLE="position:absolute; left:300; top:20">

Customer Name:

<INPUT TYPE="Text" NAME="custName">

<INPUT TYPE="Button" NAME="SL" VALUE="Buy It!"

onClick="updateList () ">

<INPUT TYPE="Button" NAME="Show" VALUE="Show XML Data"

onClick="ShowXML (xmldso2.XMLDocument) ;">

<DIV ID=PTData></DIV>

</DIV>

<INPUT TYPE="Button" NAME="Flower" VALUE="Wildflowers"

STYLE="position:absolute; left:10; top:170"

onClick="doMenu () ">

```

<TABLE ID="fmenu"

STYLE="position:absolute; left:10; top:192;

display:none; cursor:hand"

DATASRC=#xmldso

CELLSPACING="0" CELLPADDING="0" BORDER="1"

onMouseOver = "mouseHover ( ' over' ) "

onMouseOut = "mouseHover ( ' out' ) "

onClick = "goRecord ( ) ">

<TR>

<TD><DIV DATAFLD=COMMON></TD>

</TR>

</TABLE>

</BODY>

</HTML>

```


当 HTML 网页在执行时，在线商店便已开始营业并等待使用者的光临，如图 6-4 所示。文件中第一种植物的信息—罂粟科植物，是预设的显示资料。将名称输入到 Customer Name 字段之后，顾客可以按一下 Wildflowers 的按钮，由植物名称清单中选择其中一种植物。按一下 Buy It 按钮便可以将所选择的植物放入购物单中，如下图 6-5 所示。

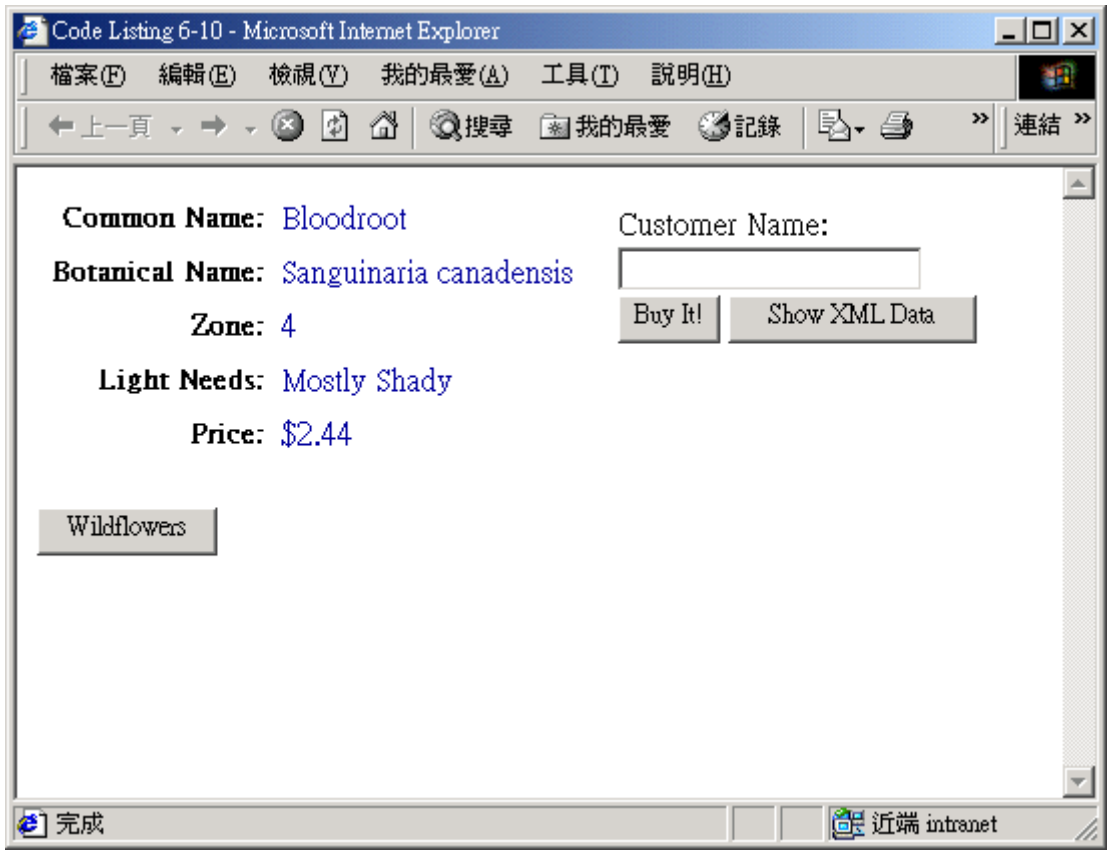


图 6-4：XML-BASED 的在线商店。

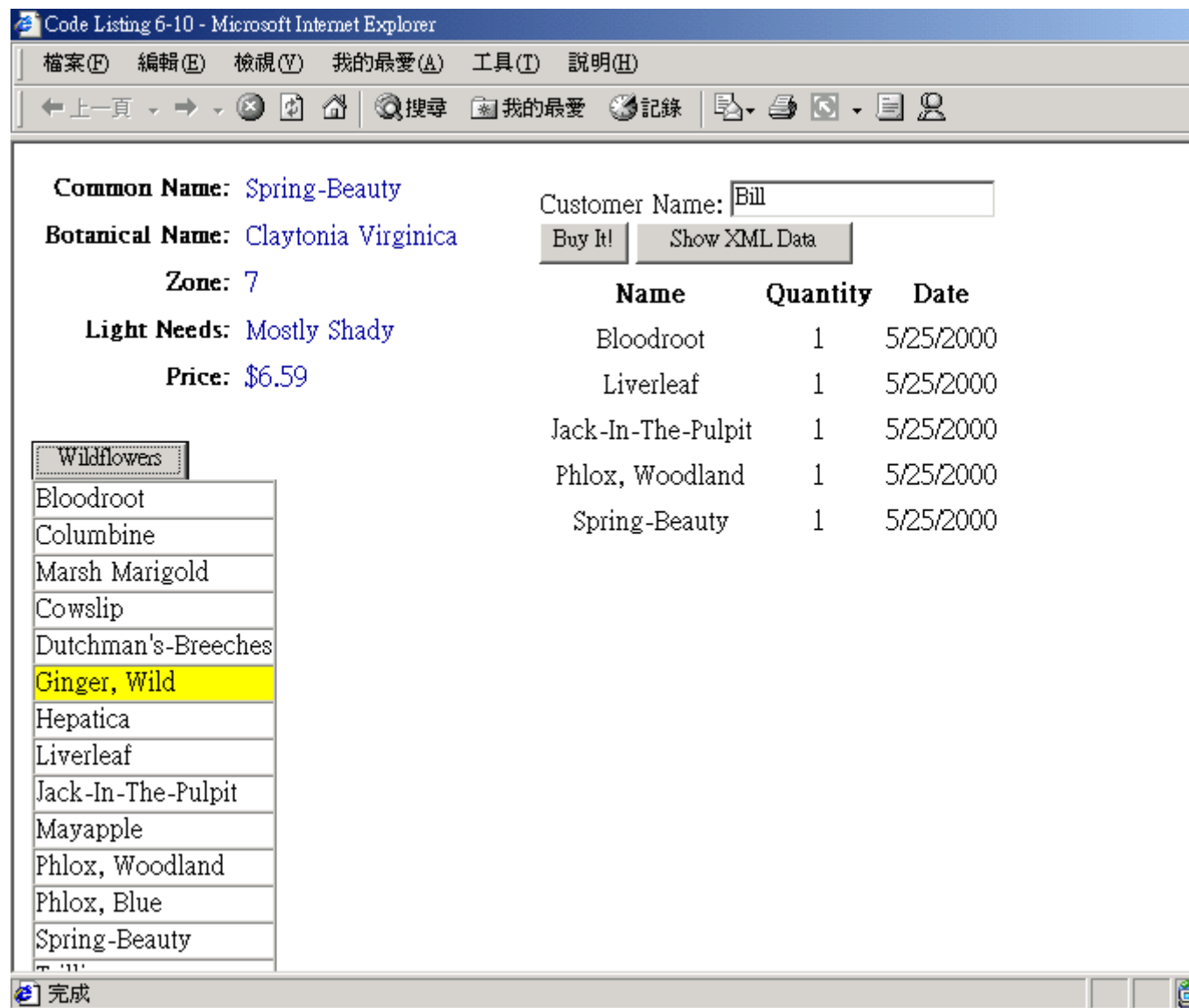


图 6-5: Buy It 按钮利用独立的 XML 文件追踪使用者的购物单。

上面的文字码也许看起来有点复杂，所以让我们花点时间来讨论一下，如此您将会了解它是如何运作的。

样式表

在文字码 **6-10** 开始的第一个部分，如下所示：

```
<STYLE>

    .category {font-weight:bold; font-size:14}

    .fdata {font-size:16; color:#000099}

    .pdata {font-size:14; font-weight:bold}

    .cdata {font-size:16; color:#993300}

</STYLE>
```

这设定了我们稍后会用到的一些文字字型。

onload Script

这个 **Script** 下载适当的 **XML** 文件到 **XML DSO** 对象中。这发生在网页一开始下载时，如此文件在它们一被需要时便即可运作。

Script 程序的连结

您会注意到在分离的 **Script** 元素中有一个被连结的 **Script** 称为 **ShowXML.js**。这是用来开启一个新的窗口，并为 **xmlIdso2** 对象建立一个 **XML** 树状结构。在此我不讨论这个 **Script** 的细节部分，但您可以藉由点选 **Show XML Data** 按钮来看看它是如何运作的。我已经把这个按钮放在网页上，如此当您与网页上的元素互动时，将可以察看 **XML** 数据改变的情形。

其余的 Script

接下来的 **Script** 元素包含所有网页上会使用到的函式。

doMenu 函式

当 **Wildflowers** 按钮被点选时，**doMenu** 函式会显示或隐藏植物清单。因为我们希望这清单的数据是来自我们的 **XML** 数据来源，因此我们必须建立自己的清单来取代 **HTML** 的窗体。大部分「内建的」**HTML** 窗体会要求您预先定义您的清单项目，而且没有我们想要的卷动功能。

goRecord 函式

goRecord 函式以在 **Wildflowers** 清单中被选取的项目为基础，来改变 **XML DSO** 的记录。这个清单与 **XML** 文件中的数据维持一对一的关系，因为这个清单与它的数据是结合在一起的，因此当数据改变时，清单会被更新来反应这些改变。

mouseHover 函式

当使用者在菜单项目上移动鼠标时，**mouseHover** 函式会改变项目的颜色。这让使用者很容易就知道哪个项目被选择过。

updateList 函式

`updateList` 函式做了下面几件事。当使用者点选 **Buy It !** 时，`updateList` 会检查使用者是否存在于顾客名单。如果没有，它显示一个讯息告诉使用者先加入顾客名单。如果他的名字已在顾客名单内，`updateList` 会呼叫 `addToList` 函式。

addToList 函式

这是 **Script** 中最复杂的一个函式。其中一个原因是它被设计成一个智能型的函式。举例来说，如果使用者已经加入一种植物到购物单中，我们便不希望再次增加这个植物，相反的，我们只希望增加数量罢了。同样的，这个函式需要去检查新的顾客名字并加到数据来源中。为了完成这些任务，这个函式会循行整个 **XML** 文件（**Lst6_9.xml**），来决定这个顾客名字是否已经存在于资料来源中。如果是的话，它会检查目前的植物是否已经加到这个顾客的购物单中。如果不在的话，这个函式会将这个新的顾客加到 **XML** 文件中，并且在购物单中增加这个植物的数据。如果植物已经找到但不属于这个顾客，它会建立一个新的顾客并将这个植物增加到他的购物单中。如果它同时找到植物和顾客名字，它只会增加这个植物的购买数量。当所有的工作完成时，这个函式会呼叫 `showList` 函式来显示。

showList 函式

`showList` 函式会建立一个表格显示目前顾客购买的植物名称及数量。这个表是以 **XML** 来源中的数据为基础而实时建立的。

控件部分

文字码接下来的部分是有关控件的部分，有两个 **XML DSO** 控件被加到网页中。因为我们需要同时使用两份 **XML** 文件，而每份文件需要一个控件。

```
<OBJECT WIDTH="0" HEIGHT="0"

    CLASSID="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"

    ID="xmldso">

</OBJECT>

<OBJECT WIDTH="0" HEIGHT="0"

    CLASSID="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"

    ID="xmldso2">

</OBJECT>
```

请注意这两个控件的大小都设为零。因为我们只希望使用控件提供的数据，而不需要在网页上显示它们。

Wildflower 信息表格

在这一节中，我们要建立表格来显示数据来源中每一笔植物「记录」的信息。请注意：所谓「每一笔」的基础字段包含 **Div** 元素，它是结合到 **XML** 数据来源的元素。

```
<TABLE STYLE="position:absolute; left:10; top:10"
  CELLSPACING="6" ID="catalog">

  <TR>

    <TD ALIGN="right" CLASS="category">Common Name:</TD>

    <TD><DIV CLASS="fdata" ID="common"

      DATASRC=#xmldso DATAFLD="COMMON">

    </TD>

  </TR>

  <TR>
```


<TD ALIGN="right" CLASS="category">Botanical Name:</TD>

<TD><DIV CLASS="fdata" ID="botan"

DATASRC=#xmldso DATAFLD="BOTANICAL">

</TD>

</TR>

<TR>

<TD ALIGN="right"> CLASS="category">Zone:</TD>

<TD><DIV CLASS="fdata" ID="zone"

DATASRC=#xmldso DATAFLD="ZONE">

</TD>

</TR>

<TR>

<TD ALIGN="right" CLASS="category">Light Needs:</TD>

<TD><DIV CLASS="fdata" ID="light"

DATASRC=#xmldso DATAFLD="LIGHT">

```
</TD>

</TR>

<TR>

  <TD ALIGN="right" CLASS="category">Price:</TD>

  <TD><DIV CLASS="fdata" ID="price"

    DATASRC=#xmldso DATAFLD="PRICE">

  </TD>

</TR>

</TABLE>
```

每当使用者选择 **Wildflowers** 清单上的项目时，DSO 便会被更新，而与数据链路的表格也会自动地更新。记住，表格是多重值的元素。正常的情况下我们会设定 **DATASRC** 属性到 **Table** 元素中。但在这里如果我们这么做，它会一次显示所有的资料！因为我们一次只需要一个节点或一笔记录，我们便将数据来源与表格上每个独立的字段结合在一起。

顾客区域和功能按钮

接下来要讨论的是供顾客输入名字的文字盒，它会显示 **Buy It !** 按钮，以及 **Show XML Data** 按钮。

（**Buy It !** 按钮用来显示顾客的购物单，而 **Show XML Data** 按钮则用来显示顾客资料来源。）

这个部分也包含一个空的 **Div** 元素，当 **Buy It !** 按钮被点选时，它会显示 **showList** 函式中的资料。

最后一个按钮是 **Wildflowers** 按钮，它会显示 **Wildflowers** 的列表。既然它在网页上显示的位置不同，因此它在主要的 **Div** 元素之外。

```
<DIV STYLE="position:absolute; left:300; top:20">

  <SPAN>Customer Name:</SPAN>

  <INPUT TYPE="Text" NAME="custName">

  <BR>

  <INPUT TYPE="Button" NAME="SL" VALUE="Buy It!"

    onClick="updateList ( )" >

  <INPUT TYPE="Button" NAME="Show" VALUE="Show XML Data"

    onClick="ShowXML (xmlldso2.XMLDocument) ;">

  <DIV ID=PTData></DIV>

</DIV>

<INPUT TYPE="Button" NAME="Flowers" VALUE="Wildflowers"
```

```
STYLE="position:absolute; left:10; top:170"
```

```
onClick="doMenu ( ) ">
```

清单表格

这个表格是用来显示 **Wildflowers** 清单的表格。

```
<TABLE ID="fmenu"
```

```
STYLE="position:absolute; left:10; top:192;
```

```
display:none; cursor:hand"
```

```
DATASRC=#xmldso
```

```
CELLSPACING="0" CELLPADDING="0" BORDER="1"
```

```
onMouseOver = "mouseHover ( ' over' ) "
```

```
onMouseOut = "mouseHover ( ' out' ) "
```

```
onClick = "goRecord ( ) ">
```

```
<TR>
```

```
<TD><DIV DATAFLD=COMMON></TD>
```

```
</TR>
```

```
</TABLE>
```

请注意：我们在此将整个表格与 **DSO** 结合在一起，来取代前面将个别字段与 **DSO** 的数据域位结合。这样允许表格显示整个 **wildflower** 名称清单，而非只是一笔记录。通常这个清单是隐藏的，直到 **doMenu** 函式启动它为止。

使用 **XML** 作为应用程序数据

此例显示 **XML** 如何被应用在真实的世界中。这个使用 **XML** 的方式，跟预期中的有点不同，因为它不只作为数据来源，而且也像一个数据储存机构。这描述了 **XML** 就是数据的外貌，此外，作为一种文件语义和结构的语言，**XML** 也提供了数据的储存和撷取的机制。您在稍后的章节将可以看到，如 **XSL** 或 **XSL Pattern** 的技术，提供更有力的方法来操纵、搜寻和显示 **XML** 资料。

7. 以 XML 连结

近几年来全球信息网（WWW）的成长真可说是进步神速，而造成这股旋风最主要的原因乃在于它提供了无远弗届的连结功能。更确切地说，Web 这个字代表了具有文件与文件间连结沟通的能力；而驱动 Web 连结能力的力量就是 HTML 机制。Web 上数以百万计的文件都是藉由这种简单却强大的连结系统，来连结部分信息到其它使用 Anchor（<A>）标签的地方。

在本章中，您将学习如何以 HTML 来完成文件间的连结，我们也会比较 HTML 与 XML 两种连结的异同，并且检视 XML 如何透过 XML 应用程序呼叫 XLink 的方式，在 Web 上提供更强而有力的方法来连结信息。最后，我们会先浏览一下 XPointer，这个机制允许可以连结到 XML 文件内部的架构。

Note

当笔者在写本书时，XLink 的规格还在草拟阶段，因此，本章所提供的信息皆以目前的版本为准，您可以在 <http://www.w3.org/TR/WD-xlink> 找到详细的信息。而且，在此时也无法取得支持 XLink 的处理软件，所以本章提供的文字码范例，都是以目前的版本做基础。由于规格未来可能会改变，因此这些范例在往后厂商开发支持 XLink 的软件中可能无法顺利执行。

HTML 的简易连结方法

如果您曾经使用过 WWW 或者在工作上接触过 HTML，您应该非常熟悉 HTML 连结文件的方式。

一个文字连结通常透过改变文字的格式（比如改变文字的颜色或在该文字下加底线），来表示具有连结的能力，而当鼠标移动到文字连结上面时，大部分网页浏览器也会改变鼠标光标的形状。

当您按下一个连结时，通常若不是开启了一份新文件（该连结的目的地）来取代目前的文件，不然就是会再开启一个浏览器窗口来显示新文件，即使新文件和前一份文件只有些许不同，仍无法避免必须执行这些动作（前一份文件就这样「跳过去」，我们会连到新的地址，通常是另一份文件）。这种连结就是众所皆知的简易连结，因为它只是单向的连结。HTML 提供了两种基本的文件连结方法：锚元素（Anchor element）与连结元素（Link element）。

Note

当您在浏览器按下 [上一页](#) 按钮时，可能会误以为自己使用的是双向连结，但这并不是真的双向连结。[上一页](#) 按钮没有使用 HTML 文字码来回到前一份文件，而是我们使用的浏览器记下了文件的位置，当按下 [上一页](#) 时重新连结到前一份文件罢了！所以 [上一页](#) 这个功能是浏览器所提供的，和 HTML 一点关系都没有。

HTML 以锚（Anchor）元素连结

在 HTML 文件中，锚元素是经常被用来连结的方法，它支持多种属性，但最广泛使用的属性就是随处可见的 HREF 属性。HREF 属性使用 URL（URL 代表的是连结目的地的地址）或片段指针（fragment identifier）当作属性值，所谓片段指针即提供相同文件或另一份文件一个预定的目的地。当指定连结到相同文件的位置时，片段指针可以单独使用，或者它可以属于一个 URL，用来表示连结到分离的文件位置。文字码 7-1（随书光盘中的 Chap07\Lst7_1.htm）说明了如何使用片段指标产生连结。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

    <HEAD>

        <TITLE>Code Listing 7-1</TITLE>

    </HEAD>

    <BODY>

        This is a

        <A HREF="Lst7_2.htm#jumplocation">simple link</A>

        in HTML that uses the Anchor element.
```

```
</BODY>
```

```
</HTML>
```

文字码 7-1

文字码 7-1 中，我们在 URL 后面加入了一个片段指标（fragment identifier），所以当打开目的文件（Lst7_2.htm）时，便会跳到片段指针所指的位置，也就是 jumplocation。文字码 7-2（随书光盘中的 Chap07\Lst7_2.htm）显示目的文件包含一个内部的连结。

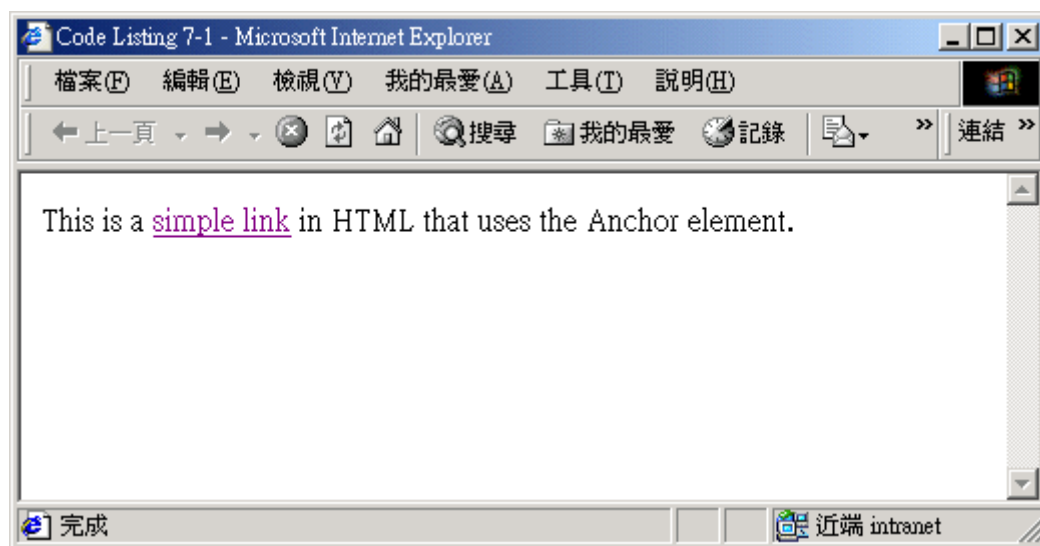


图 7-1：包含一个连结来源的 HTML 文件。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

<HEAD>

<TITLE>Code Listing 7-2</TITLE>

</HEAD>

<BODY>

Here is the new document!

This is the location for the
jump from the previous document.

Clicking

here

will take you to another part of this document
by using a fragment identifier.

Here is the location for the jump in this document.

Click here

to go to the top of this document.

</BODY>

</HTML>

文字码 7-2

Note

文字码 7-2 包含了大量的换行标签
，这样做是用来模拟一份大型的文件。换行标签在行与行之间建立了很多空白，藉此可以证明在一般情况下，连结是不需要换行标签的。

当使用者按下文字码 7-1 中的连结时，文字码 7-2 中的文件（Lst7_2.htm）会被开启并依据片段指标跳到该连结指向的位置，也就是 **jumplocation**。请注意文字码 7-2 包含了一个只有 **NAME** 属性的锚元素（Anchor element），而 **NAME** 属性的值就是 **jumplocation**，属性值也告诉了文字码 7-1 该跳往何处。如果目的文件中没有名为 **jumplocation** 的元素，文字码 7-2 中描述的文件还是会被打开，但是无法跳到所指定的位置。

在文字码 7-2 中，跟在锚元素之后称为 **jumplocation** 的是另一个连结，这一次指向同一份文件的位置。虽然这个内部的连结看起来和外部的连结一样（图 7-2），但我们知道这个连结指向目前文件内部的位置，因为该连结的目的地是一个片段指标（**fragment identifier**），并不包含 **URL**。但无论如何，锚（**Anchor element**）元素包含的目的名称必须存在，才能顺利的连到目的位置。

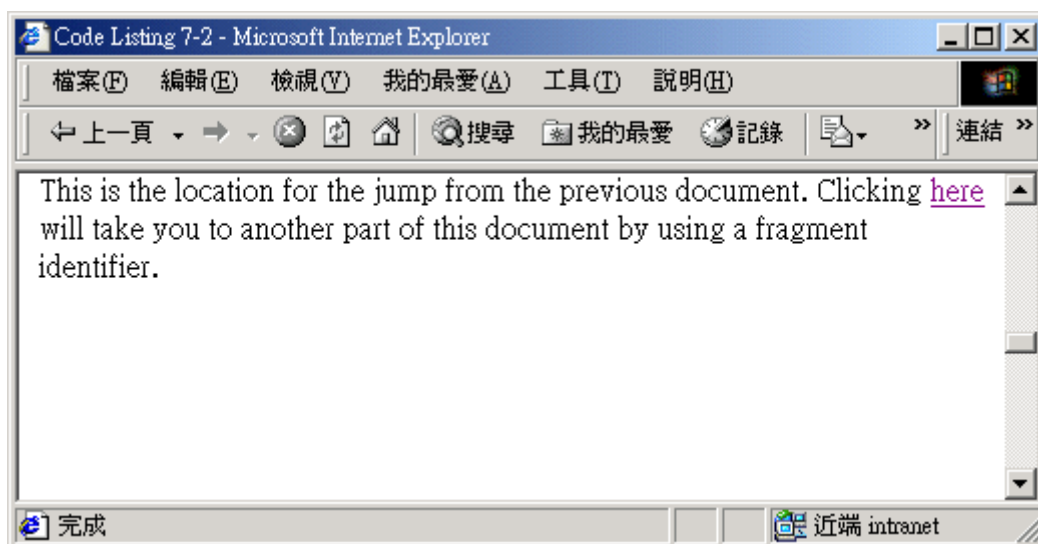


图 7-2：使用片段指标（fragment identifier）的内部连结。

以 Link 元素（Link Element）连结到外部文件

Link 元素只能在文件的标头区域被使用，而且不被用来作文件的跳跃，取而代之，这个元素是用来建立目前文件和其它文件或对象的关联性。文字码 7-3（随书光盘中的 Chap07\Lst7_3.htm），Link 元素被用来连结到文件的样式表（style sheet）。虽然这个样式表属于原始文件的一部分，但经由 Link 元素便可以使用。Link 元素用的这个方法其实很像 Image 元素，Image 元素以 SRC 属性连结到外部对象，使这个对象变成目前文件的一部分。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

```
<HEAD>
```

```
<LINK REL="stylesheet" HREF="Styles.css">

<TITLE>Code Listing 7-3</TITLE>

</HEAD>


<BODY>

  <SPAN CLASS=style1>This text uses style1</SPAN>

  <BR>

  <SPAN CLASS=style2>This text uses style2</SPAN>

</BODY>

</HTML>
```

文字码 7-3

虽然在文字码 7-3 中的 Link 元素包含了 HREF 属性,但该连结并不会跳到 URL 所描述的文件位置,而是将目的文件 (target document) 连回原始文件 (source document)。

Note

Link 元素中的 REL 属性可以告诉处理的程序该元素是属于何种型态的连结，它说明为何所连结的资源与目前文件有关。虽然 REL 属性可以包含任何值，但它也有一些默认值，诸如 **stylesheet**、**home**、**help** 等等，而 REL 属性也并不是所有的浏览器都认得。

我们在前面提到的连结机制，特别是锚元素，已经被数以万计的人以及数以千万的网页使用了好些年，所以这就是为何 XML 不以自己的连结规格来改变现有的连结方式。这个好消息是说 XML 没有改变 HTML 所用的简单连结机制，像锚元素（Anchor element）与连结元素（Link element）等等。相反的，XML 的连结规格是以这些简单的连结机制为出发点，并延伸原来连结的概念，成为更强而有力的新方式来处理连结的信息。

XLink: XML 的连结机制

XML 的规格并没有包含内建的连结方法，而是预计透过 XML 语汇的 XLink 来实现 XML 中的连结。XLink 使用 XML 来定义所有必要的部分，方便在 XML 文件中建立连结，XLink 定义两种基本的连结类型：简易连结（simple links）与延伸连结（extended links）。XLink 是设计来维护 HTML 内的简易连结，但当您需要的时候，它却可以提供更强而有力与可延伸的能力。为了实现这样的目标，XLink 的制定者在发展 XLink 规格时，是根据几个设计目标来发展的。

Note

简易连结与延伸连结最主要也是最大的不同在于：延伸连结可以连接到任何数目的资源，不像 HTML 的简易连结，只能连接一个近端或远程的资源。详细内容请参阅本章<XML 的延伸连结>一节的说明。

目标一：XLink 必须可以在因特网上简易而方便的使用。 这个目标最主要在于：使目前所用的简易连结机制也可以在 XML 中使用。这个目标同样也希望 XLink 可以很容易地应用在因特网中，尽管增加任何新的特色也要将此目标考虑进去。

目标二：XLink 要能在广泛的连结使用领域和连结应用软件中有效地使用。 此目标指出 XLink 不希望只是在 Web 应用软件中使用，而是具有足够的弹性来满足各种应用软件的使用者。

目标三：描述 XLink 的语言就是 XML。 XLink 是 XML 的应用，此目标并不是要重新创造一种新语言，只是建立 XML 的词汇。当我们使用 XML 语言时，XLink 则是一个建置好的架构和格式，并不需要文件作者或软件设计人员再去学习一种新的语言。

目标四：XLink 的设计必须尽快确立。 这一点在 XML 是确定的，XLink 的制定者深知如果 XML 团体不尽快确立 XML 中的连结机制，其它的组织与商业团体可能会自己建立在 XML 中专有的连结方法，而这将会违背 XML 语言所希望开放性与标准化的基本原则。

目标五: XLink 的制定必须要合乎格式而且简单明了。这个目标也是 XML 目标中的一个, XLink 的制定者知道, 如果想要使企业采用此种规格, 它必须要容易了解与使用。此目标帮助制定人员专注于规格用语的格式, 以及继续让规格尽可能越小越好。

目标六: XLink 必须让人们易于阅读。因为 XLink 是以 XML 建立的, 它必须像 XML 一样容易了解, 并让人们容易阅读。而不需要由计算机来编译程序代码中所做的事情。

目标七: XLink 可以存在于文件外部。这个目标其实就是 XLink 语言所强调的延伸性。您将会看见, XLink 提供指向多个资源的连结, 超越了 HTML 简易的单一连结, 这正是 XLink 制定者心中想要的。

目标八: XLink 应该要呈现连结的抽象架构与意义。此目标以 XLink 语言之后的一些哲学为基础。制定人员决定 XLink 不应仅仅在 XML 中提供连结的机制, 也应该要让 XLink 建立的连结能描述连结的语意, 以及分辨连结与对象或资源与连结者的关系。这样便可以提供使用者信息, 诸如连结的角色或者该连结所使用的资源。

目标九: XLink 必须可以实作。就像 XLink 所强调的易学易读, 连结亦应让程序设计人员易于实作。制定者也知道假如这种语言要能被广泛地接受, 它就必须容易实作。但从目前书面数据来看, 就算有, 也只有非常少数的软件使用 XLink, 所以这个目标是否达成, 还需要进一步的观察。

XML 内的简易连结

虽然 XML 可以使用 HTML 的方式来做文件的连结，但我们应该要开始把 XML 内的连结想象成资源连结，也就是说，XLink 可以经由连结元素的定位器（locator）连接到任何资源。这并不是 XLink 特有的概念，而是 WWW 上本来就存在的资源概念，其中最大的不同在于，XLink 提供了一种更先进的语法来定义连结元素与行为，像是允许发生多方向的连结等等。XLink 也变得更具有弹性，允许文件可以连结到其它网站上取得任何形式的资源。

XLink 中的简易连结可以有相同的功能，像是 HTML 中的锚元素（Anchor element），简易连结就是 HTML 中的行内连结（inline link），也就是说，连结是属于元素的一部分，而且只能做单向的连结。

简易连结仅有一个资源指标或定位器，所谓定位器就像是锚元素的 HREF 属性，包含了连结的数据。在 XLink 中，所有的定位器信息都包含在连结元素内，也就是说，处理的软件不需要去寻找关于定位器的其它信息。在文字码 7-1 和 7-2 中，如果原始文件与目的文件都属于 XML 文件，则目的文件就不需要使用包含片段指针名字的元素，因为所有必要的定位器信息都已经包含在连结元素内了。既然 HTML 没有提供这个阶层的功能，就让我们看看如何在 XML 中建立连结。

我们在第一个范例连结里建立一个 XML 简易连结。即使是在 XML 中建立一个简易连结元素，也比单单建立一个锚元素和指定一个 HREF 属性值复杂多了。请记住，XML 是中介语言（meta language），所以要在文件中建立一个可以使用的新元素，就必须宣告该元素和所有属性。

还好 XML 和 XLink 规格提供我们很多功能强大的选项来建立连结。

我们现在就来建立一个叫做 **SLINK** 的简易连结元素，底下是可能会出现在 **DTD** 中的元素宣告和属性列表。

Note

SLINK 这个元素并没有什么特别的地方。事实上，您甚至不需要建立只有连结才需要的元素（像是 **HTML** 中的锚元素），而可以把任何元素都当作连结元素，也就是说，您在 **XML** 文件中的连结信息可以有很多种选择。

```
<!ELEMENT SLINK ANY>

<!ATTLIST SLINK

    XML:LINK CDATA #FIXED "simple"

    HREF CDATA #REQUIRED

    INLINE (true|false) "true"

    ROLE CDATA #IMPLIED

    TITLE CDATA #IMPLIED

    SHOW (replace|new|embed) #IMPLIED

    ACTUATE (auto|user) #IMPLIED

    BEHAVIOR CDATA #IMPLIED
```

```
CONTENT-ROLE CDATA #IMPLIED
```

```
CONTENT-TITLE CDATA #IMPLIED
```

```
>
```

您可能已经注意到，我们所熟悉的 **HREF** 属性出现在 **SLINK** 这个元素中，但其它的属性都是第一次见到。底下就让我们进一步来探讨它们的用途吧！

XML: LINK

XLink 的制定者知道他们必须提出一个可靠的方法，以便让处理它的软件都能识别连结元素，他们是透过保留标签名称（比如 **HTML** 里的 **<A>**），保留属性名称，或者将控制权交给应用软件等方法。最后，他们决定保留属性名称，相信这样是在「允许文件作者定义他们的元素」与「将连结元素维持为元素架构的一部分」之间提供一个平衡点，这样的结果是被指派的属性 **XML:**

LINK 可以有 **simple** 或 **extended** 值。读者如果不健忘的话，前面的例子中，我们就是指定 **simple** 这个值来建立简易连结的（本章稍后会介绍延伸连结），而在使用上，**Slink** 元素将会在文件中以下述方式出现：

```
<SLINK XML:LINK="simple" HREF="mspress.microsoft.com">
```

```
Microsoft Press Home Page
```

```
</SLINK>
```

XML: LINK 属性包含在上述 **Slink** 元素中，不过假如属性在元素宣告中已经预先宣告一个值给它，则元素本身即不需要包含 **XML: LINK** 属性来运作。

HREF

每一个在 **XML** 中的连结元素都必须要有个来源定位器（**resource locator**），它用来确认连结将要连接的资源，而 **XLink** 中的定位器属性就是 **HREF**，它工作的方式和在 **HTML** 内一样。定位器也包含了一些 **HTML** 没有提供的功能，底下就是一些相关的讨论。

定位器与片段指标（**Fragment Identifiers**）

现在就让我们瞧瞧一些在 **XLink** 中定位器的延伸功能。在 **HTML** 中，定位器的 **HREF** 属性可能会包含一个片段指标，如下所示：

```
This is a <A HREF="Lst7_2.htm#jumplocation">simple link</A>
```

片段指标会提供一个连结到预定的地方，可能是在同一份文件或另一份文件中，在 **XML** 中，定位器会提供信息，允许应用程序跟着一个以文件架构为基准的连结，一个元素识别码或者甚至是一个元素的内容。这一部分可以透过 **XML XPointer** 寻址机制来完成，它的语法如下所示：

```
HREF="uri#Xpointer"
```

```
href="uri|xpointer"
```

我们会在本章后面详细介绍 **XPointer**，它允许文件作者可以跟着连结进入 **XML** 文件中任何部分，并且使用各式各样的方法说明需要的资源。既然定位器机制提供了一个富有弹性的浏览方法（**traversal method**），那么指标便可以只传回目的文件的一小部分，来节省频宽和客户端的运算资源。

INLINE

XLink 连结元素如果不是行内（**inline**）就是行外（**out-of-line**）（更多关于这些连结类型的信息，请参阅本章的 [<行内延伸连结>](#) 与 [<行外延伸连结>](#)）。一个行内连结可以视为它所拥有资源中的一个，也就是说，连结元素包含的内容与连结目的一样。**HTML** 的锚元素（**Anchor element**）便是这类连结型态最好的范例，锚元素也包含了与连结目的相同的内容。在我们的例子中，因为 **INLINE** 属性宣告默认值为 **true**，所以并不需要特别指出该元素的属性。

ROLE

ROLE 是 **XLink** 连结元素的一大特色，因为 **ROLE** 属性可以让应用软件确认连结的意义或重要性。请注意，这个属性是用来提供与连结相关的完整信息，而不仅仅只是提供该连结的远程资源。此外，这些信息是用来给应用程序了解的，而不是给我们看的，也就是说使用 **ROLE** 时，您可以提供更多与连结相关的详细信息给应用程序，而不是简单的说明连结到哪去。举例来说，某些连结可能会连到专业术语，或是连到某个主题的信息背景，也可能连到某个资源的属性信息（像是版本信息）等等，此时应用程序可以藉由连结直接取得这类信息并做出适当的响应。

Note

本节许多属性的描述都是使用资源语意（**resource semantics**），也就是说，它们提供关于意义的信息或资源内容，而这些属性当中有些是用在远程的资源语意，有些则是使用在近端的资源语意，因此，必须注意其适用性。

TITLE

TITLE 属性和 **HTML** 中的 **<ALT>** 标签属于同一种类型，它包含了一个能够显示的卷标（**label**）或文字（**text**），用来提供使用者一些额外的信息。**TITLE** 是一种远程资源属性：它并不完全与

连结有关，但却能够提供信息给使用者，用来知道资源与连结的相关性，其实就像 **ROLE** 属性是用来提供信息给机器，**TITLE** 属性只不过是用来提供信息给使用者而已。

SHOW

SHOW 属性是 XML 远程资源语意的一部分，而且它可能是改善 HTML 连结方式最有名的方法之一。**SHOW** 属性接受的属性值有 **replace**、**new** 与 **embed**，这些属性用来描述连结如何被浏览。属性值 **replace** 表示以远程资源替代近端资源，这是我们在 HTML 中最常见的技巧，实际上，当使用者按下一个连结时，浏览器便会跳到一个新的网页。属性值 **new** 则表示目的资源（**target resource**）会以新的内容开启，其功能就像 HTML 中锚元素（**Anchor element**）的 **TARGET** 属性一样，通常该连结会开启一个新的浏览器窗口来浏览新的目的内容，如果没有找到相同名称的内容，一个新的内容会被建立（通常是开启一个新的浏览器窗口）。

属性值 **embed** 是一种浏览连结的新技巧，**embed** 属性值一旦被设定，连结目标的内容便会内嵌于原始连结的内容中。举例而言，若有一份文件包含一个连结，连结到特殊标题的背景信息，当使用者按下以 **embed** 为属性值的连结时，信息便会显示在来源文件右边的内容中。

ACTUATE

ACTUATE 属性也是属于远程资源语意的一部分,它用来描述要以什么方式浏览连结。**ACTUATE** 属性可以接受的属性值有 **auto** 或 **user**, 属性值 **auto** 表示当应用程序处理连结时, 能够自动的浏览连结; 而属性值 **user** 则表示该连结必须透过一个外部机制来浏览, 比如说按下鼠标。

Note

您可能会开始联想, 经由组合特定的属性便可以新增一个全新的方法来连结信息。比如说, 组合「**SHOW=embed**」属性和「**ACTUATE=auto**」属性, 文件作者便可以建立一份包含许多内嵌连结的文件。当使用者开启这份文件, 所有的连结都会自动被浏览, 并直接内嵌于文件中。这种复合文件, 包含了从多种来源收集到的信息, 而这些使用者是看不到的。

BEHAVIOR

直到现在, 前面所提的大部分属性都是用来描述连结的: 它们为连结下定义, 并描述连结组成的部分等等。**BEHAVIOR** 属性也是属于远程资源语意的一部分, 它提供连结的作者一个空间, 用来完整描述当连结被驱动时应该做些什么。举例来说, 假设一个连结有 **before** 和 **after** 两种状态, 状态 **before** 表示连结作用前的外观, 可能包含字型、颜色和其它格式 (请注意 **XLink** 本身并没有提供控制连结格式的方法, 而是把这一部分的处理交给应用程序); 状态 **after** 则发生在连结

作用后。我们以 **SHOW** 和 **ACTUATE** 属性来表示基本的行为，而 **BEHAVIOR** 属性则不受约束，意思是包含任何形式的指令，被用来与处理连结的应用程序沟通。**BEHAVIOR** 是用来提供 **SHOW** 和 **ACTUATE** 属性所无法提供的额外行为信息，但是 **XLink** 并未指出 **BEHAVIOR** 属性可以包含的特殊属性值。

CONTENT-ROLE

CONTENT-ROLE 这个属性的工作方式有许多与 **ROLE** 属性的非常相似，只不过它是属于近端资源，而它是用来让软件知道属于连结一部分的近端资源所要达成的目的。与 **ROLE** 属性一样，是提供信息给软件，而不是提供信息给人们。

CONTENT-TITLE

CONTENT-TITLE 是另一个近端资源语意的属性，它提供使用者关于连结近端部分的信息。如同 **TITLE** 属性的功能，**CONTENT-TITLE** 是提供信息给人们的。

以 XML 模拟 HTML 连结

现在您已经了解建立简单连结的程序，现在就让我们看看如何使用 **XLink** 的概念，重新建立 **HTML** 中锚元素（**Anchor element**）所提供的功能。本例可用来说明 **XML** 的连结方法是如何仿

真 HTML 连结的功能，而且也可以看出 XML 与 HTML 不同的地方。就让我们从宣告一个基本元素开始吧！

```
<!ELEMENT ANCHOR ANY>

<!ATTLIST ANCHOR

    XML:LINK CDATA #FIXED "simple"

    HREF CDATA #REQUIRED

    TITLE CDATA #IMPLIED

    INLINE (true|false) "true"

    SHOW (replace|new|embed) "replace"

    ACTUATE (auto|user) "user"

>
```

这是一个好的开始，上述文字码包含了大部分 HTML 中的锚元素经常使用的属性，而底下这行文字码则允许我们在 XML 中写一个符合 HTML 连结的简易锚元素（Anchor element）：

```
<A HREF="http://www.microsoft.com">Microsoft Home Page</A>
```

要真正完成锚元素的复制，我们必须增加更多 HTML 锚元素所支持的属性，诸如：REV、REL、NAME 和 ARGET 等等，而 REV 和 REL 则需要重新对应到 XML 的属性，才能使它们正常使用（底下会有更详细的说明）。

属性对应 (Attribute Mapping)

有时候我们必须对应已经存在的 **XLink** 属性名称到其它的属性名称。比如说，在一份存在的 **XML** 文件所包含元素的属性名称与 **XLink** 的属性名称互相冲突时，如果要使用这些元素当作连结元素的话，便会产生问题。而您可能也会想要对应存在的 **XLink** 属性名称到 **HTML** 属性名称，这样会比较接近我们熟悉的 **HTML** 语法，而且我们也可能会使用已存在的 **HTML** 连结。

XLink 提供一个属性对应机制以便完成这类工作，亦即 **XML: ATTRIBUTES** 属性，它的基本语法是在关键词 **XML: ATTRIBUTES** 之后跟着一对以空白分隔的属性名称。每一对属性名称的第一个名称代表的是 **XLink** 的属性名称，另一个属性名称是第一个属性名称所扮演的角色名称。

让我们瞧瞧底下的例子。虽然 **HTML** 的 **REV** 和 **REL** 属性与 **XLink** 中的 **ROLE** 和 **CONTENT-ROLE** 相等，但我们将在 **XML** 版本的锚元素中重新对应 **XML** 的属性，就像下面的文字码：

```
XML:ATTRIBUTES CDATA #FIXED "ROLE REV CONTENT-ROLE REL"
```

上面的文字码可以用来告诉处理的应用程序，在这个元素中遇到 **REV** 属性时，必须对应到 **XLink** 的 **ROLE** 属性，而遇到 **REL** 属性时，则必须对应到 **XLink** 的 **CONTENT-ROLE** 属性。

现在就让我们加入属性对应到我们的元素宣告中，宣告中目前只有 **REV** 与 **REL** 属性：

```
<!ELEMENT ANCHOR ANY>
```

```
<!ATTLIST ANCHOR

    XML:LINK CDATA #FIXED "simple"

    XML:ATTRIBUTES CDATA #FIXED "ROLE REV CONTENT-ROLE REL"

    HREF CDATA #REQUIRED

    TITLE CDATA #IMPLIED

    INLINE (true|false) "true"

    SHOW (replace|new|embed) "replace"

    ACTUATE (auto|user) "user"

    REV CDATA #IMPLIED

    REL CDATA #IMPLIED

>
```

最后，我们可以加上 **NAME** 属性与 **TARGET** 属性的宣告。

```
<!ELEMENT ANCHOR ANY>

<!ATTLIST ANCHOR

    XML:LINK CDATA #FIXED "simple"

    XML:ATTRIBUTES CDATA #FIXED "ROLE REV CONTENT-ROLE REL"

    HREF CDATA #REQUIRED

    TITLE CDATA #IMPLIED

    INLINE (true|false) "true"
```

```
SHOW (replace|new|embed) "replace"

ACTUATE (auto|user) "user"

REV CDATA #IMPLIED

REL CDATA #IMPLIED

TARGET CDATA #IMPLIED

NAMEID #IMPLIED

>
```

现在我们建立了一个 XML 锚元素（Anchor element），它的行为和 HTML 锚元素一样，而且支持相同的属性。由此我们可以知道：XLink 不仅可以完全复制 HTML 中的简易连结，而且能超越它。下面我们就来看看，XLink 的延伸连结如何将连结信息的概念带到另一个新的境界。

XML 的延伸连结（Extended Links）

如同前面提到的，简易连结与延伸连结最大的不同在于：延伸连结可以连接到任何数量的资源，而不只是像 HTML 一样，只能连接一个近端或远程资源。使用延伸连结，文件作者可以从单一的来源定义可能的目的群组。比如说，我们想要从单一类别中提供连结连接到很多的体育新闻相关报导，使用延伸连结取代特殊的控件或长串的程序代码来提供这些连结所需的信息。

此外，延伸连结也提供了其它强大的功能，在此我们仅扼要地说明，至于其它细节将在下一节提到。

- 能够在只读资源中连结。一份文件可以包含向外的连结，即使它无法被修改来包含行内连结。
- 能够过滤相关连结。当使用者执行及时过滤连结资源的机制时，过滤可以及时完成。

行内延伸连结（Inline Extended Links）

延伸连结因为提供了强大的功能，所以使用上也比较复杂。下列是基本的行内延伸连结元素宣告：

Note

就像简易连结（**simple link**）一样，延伸连结可以宣告任何元素名称。下列文字码以 **ELINK** 这个名称为例子。

```
<!ELEMENT ELINK ANY>
```

```
<!ATTLIST ELINK
```

```
XML:LINK CDATA #FIXED "extended"
```



```
INLINE (true|false) "true"

ROLE CDATA #IMPLIED

TITLE CDATA #IMPLIED

SHOW (replace|new|embed) #IMPLIED

ACTUATE (auto|user) #IMPLIED

BEHAVIOR CDATA #IMPLIED

CONTENT-ROLE CDATA #IMPLIED

CONTENT-TITLE CDATA #IMPLIED

>
```

您可能会注意到：这些文字码看起来和先前建立的简易连结宣告极为相似，但它们有两点不同。

第一，**XML: LINK** 属性值以 **extended** 取代 **simple**；第二，我们并未宣告 **HREF** 属性。这是因为使用延伸连结时，**locators** 必须被包含在分开的元素集合里，而这些元素就是 **locator** 元素。

接下来我们便试着使用这种方法来定义 **locator**，允许文件作者指定多个 **locator** 给一个连结元素，

当使用其它元素时，**locator** 元素则必须被宣告。接下来让我们看看 **locator** 元素宣告。

```
<!ELEMENT ELOCATOR ANY>

<!ATTLIST ELOCATOR

    XML:LINK CDATA #FIXED "locator"

    HREF CDATA #REQUIRED

    ROLE CDATA #IMPLIED
```

```
TITLE CDATA #IMPLIED
```

```
SHOW (replace|new|embed) #IMPLIED
```

```
ACTUATE (auto|user) #IMPLIED
```

```
BEHAVIOR CDATA #IMPLIED
```

>

请注意 **XML: LINK** 的属性值为 **locator**，用来指出这个元素是一个 **XLink locator**。此外，我们也使用了 **HREF** 属性。

现在我们就来看看在 **XML** 文件中，这样的架构是如何运作的。以前面的宣告为基础，连结元素看起来可能会像底下的文字码：

```
<ELINK XML:LINK="extended">minivan review
```

```
  <ELOCATOR TITLE="Chrysler Town and Country" HREF="Chrysler. htm"/>
```

```
  <ELOCATOR TITLE="Ford Windstar" HREF="Ford. htm"/>
```

```
  <ELOCATOR TITLE="Chevrolet Venture" HREF="Chevy. htm"/>
```

```
  <ELOCATOR TITLE="Honda Odyssey" HREF="Honda. htm"/>
```

```
  <ELOCATOR TITLE="Nissan Quest" HREF="Nissan. htm"/>
```

```
  <ELOCATOR TITLE="Toyota Sienna" HREF="Toyota. htm"/>
```

```
</ELINK>
```

此种格式提供处理器所有关于延伸连结的信息，而应用程序也可以根据这些信息来决定下一步的动作。比方说，这个应用程序可以用呈现其它 Web 连结的方式，来呈现这个连结。如下图 7-3 所示。

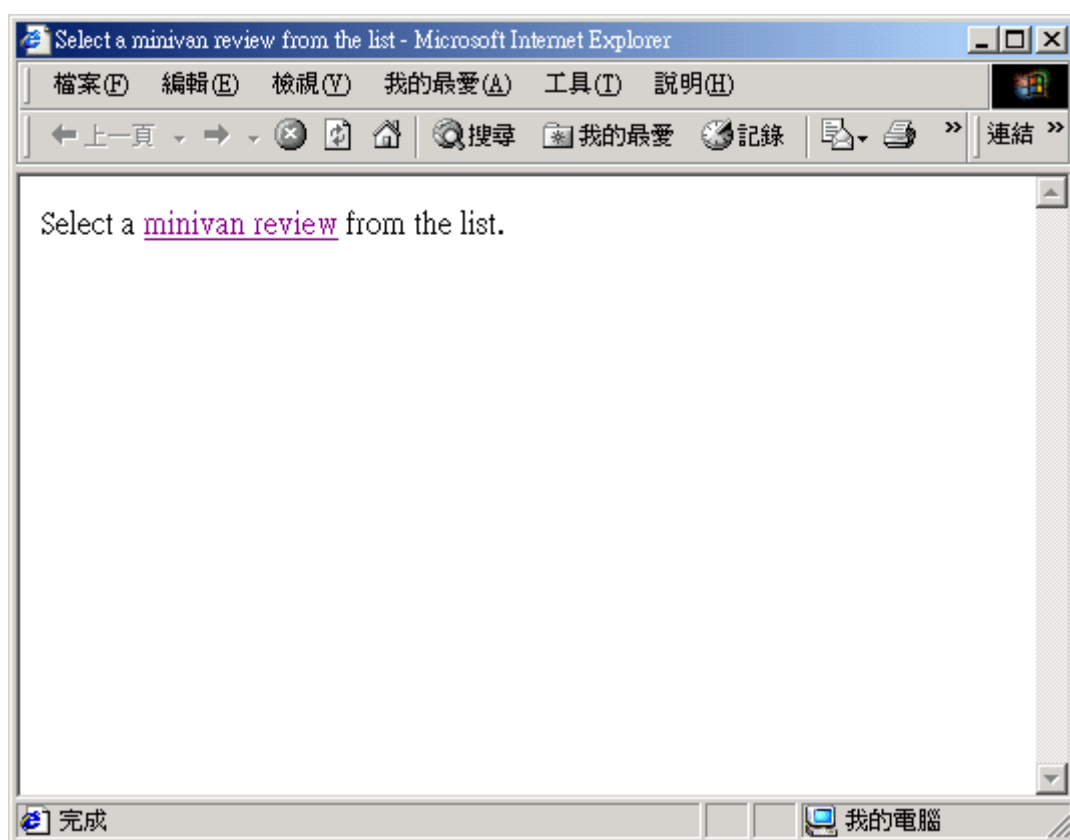


图 7-3：行内延伸连结以其它 Web 连结的方式呈现。

但当使用者按下连结时，应用程序可以执行程序设计人员设计好的动作。比如说，依据 locator 清单的内容，应用程序可以呈现可能的区域使用者列表，如图 7-4 所示。

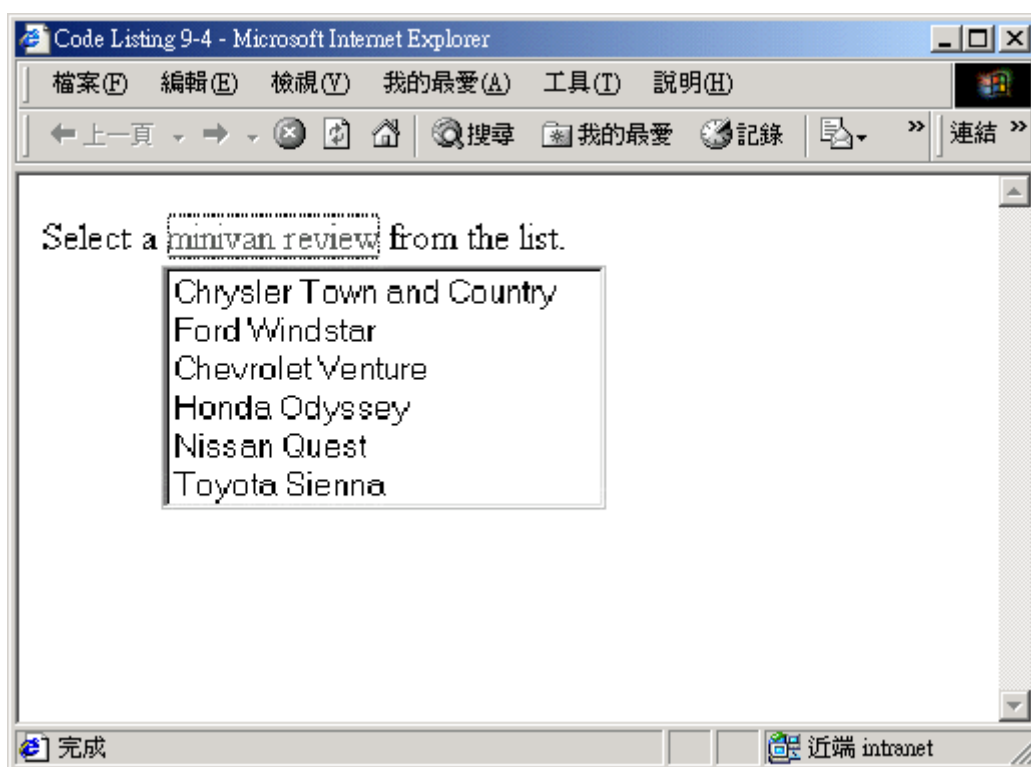


图 7-4: 显示按下行内延伸连结的结果。

这种技术最大的好处，不在于能够立刻显示有用的文件到清单中，而是在所有文件之间建立了相对应的连结。假使将元素宣告与 **locator** 元素放在分开的 XML 文件里头，而这些 XML 文件亦可以从 **locator** 清单中的所有文件取得。下面就让我们看看使用另一种延伸连结的范例，也就是行外延伸连结。

行外延伸连结（Out-of-Line Extended Links）

我们之前提到的例子采用行内连结概念，并一再强调它的重要性，但近端资源与连结文字本身仍旧属于整个连结的一部分。换句话说，行内连结需要连结文字来完成连结的动作。延伸连结允许

文件作者建立一个不属于连结部分的连结到近端资源中，也就是说，我们可以把元素或内容的部分当作连结，即使它原本并不提供这样的使用方式。这种连结就叫做行外连结，它在连结信息时提供更强大、更弹性的方法。在建立行外连结时，我们使用与建立行内延伸连结时，相同的延伸连结元素宣告与 **locator** 元素宣告，当元素被插入到一份文件时，必须作一些更动。在元素卷标中，我们必须设定 **INLINE** 属性的值为 **false** 来指出该元素为行外连结。

```
<ELINK XML:LINK="extended" INLINE="false">

  <ELOCATOR TITLE="Chrysler Town and Country" HREF="#Chrysler"/>

  <ELOCATOR TITLE="Ford Windstar" HREF="#Ford"/>

  <ELOCATOR TITLE="Chevrolet Venture" HREF="#Chevy"/>

  <ELOCATOR TITLE="Honda Odyssey" HREF="#Honda"/>

  <ELOCATOR TITLE="Nissan Quest" HREF="#Nissan"/>

  <ELOCATOR TITLE="Toyota Sienna" HREF="#Toyota"/>

</ELINK>

<REVIEW ID="Chrysler">

  <TITLE="Chrysler Town and Country">/TITLE>

  <!-- This is the Chrysler section -->

</REVIEW>

<REVIEW ID="Ford">

  <TITLE="Ford Windstar">/TITLE>
```

```
<!-- This is the Ford section -->

</REVIEW>

<REVIEW ID="Chevy">

  <TITLE="Chevrolet Venture">/TITLE>

  <!-- This is the Chevrolet section -->

</REVIEW>

<REVIEW ID="Honda">

  <TITLE="Honda Odyssey">/TITLE>

  <!-- This is the Honda section -->

</REVIEW>

<REVIEW ID="Nissan">

  <TITLE="Nissan Quest">/TITLE>

  <!-- This is the Nissan section -->

</REVIEW>

<REVIEW ID="Toyota">

  <TITLE="Toyota Sienna">/TITLE>

  <!-- This is the Toyota section -->

</REVIEW>
```

在这个范例中，**locator** 被保留在它们专属的区域，而且没有联系到任何近端的连结资源。**locator** 包含了提供连结到文件适当区域所有必要的信息，但使用何种方式显示连结给使用者，则交给应用程序决定。举例来说，提供从内容分割的清单允许使用者在任何时间都能连接到这些连结。这样的方法设定了真正的多方位连结，并且改变使用者目前在 **Web** 上惯用的「上一页与下一页」搜寻方式。

更进一步来说，假设一份 **XML** 文件包含在自己的 **DTD** 中，而 **Elink** 与 **Elocator** 元素沿用我们之前的宣告，并假设它也包含在自己的文件元素中，下面列出 **locator** 元素：

```
<ELINK XML:LINK="extended" INLINE="false">

  <ELOCATOR TITLE="Chrysler Town and Country" HREF="Chrysler.htm"/>

  <ELOCATOR TITLE="Ford Windstar" HREF="Ford.htm"/>

  <ELOCATOR TITLE="Chevrolet Venture" HREF="Chevy.htm"/>

  <ELOCATOR TITLE="Honda Odyssey" HREF="Honda.htm"/>

  <ELOCATOR TITLE="Nissan Quest" HREF="Nissan.htm"/>

  <ELOCATOR TITLE="Toyota Sienna" HREF="Toyota.htm"/>

</ELINK>
```

现在任何使用这份文件的应用程序都可以取得这些连结，因为 **locator** 已经包含在它们自己的文件中。这样便建立了一个多方位连结架构，因为我们可以从任何文件中取得连结到任何文件的连结。至于另一个好处，因为 **locator** 清单属于自己的文件，所以连结清单可以从文件中分开来管

理。此外，这样的架构可以用来增加连结到文件中，却无法编辑来包含自己拥有的行内连结。最后，即使在显示连结前，应用程序也会确认是否能够取得 **locator** 中的文件，也就是说不会有断掉的连结。

您可能开始看到，**XLink** 提供了一些非常强大的机制，来建立在以 **XML** 为基础的信息资源间的连结，但您也会意识到要有能力为连结设定多个目的、行外延伸连结与多方位连结等等，您可能会停止去尝试管理这些令人头昏眼花的连结。事实上，**XLink** 提供我们一个解决的方法：延伸连结群组（**extended link groups**）。

延伸连结群组（Extended Link Groups）

延伸连结群组可以让您经由设定包含相关文件清单的元素，来管理相关的连结信息，我们就延伸上一节的例子来作说明。假使您替消费者杂志工作，而且每个月必须发表新式汽车的报告，您一定不会每个月把每个目录中的车辆清单都摆在一份文件上，而是将清单分类以便管理。使用延伸连结群组有两个必要的元素，其中一个用来定义群组，而另一个则用来指定群组中的文件。下面文字码列出您可能看到的宣告方式：

```
<!ELEMENT GROUP (DOCUMENT*) >

<!ATTLIST GROUP

    XML:LINK CDATA #FIXED "group"

    STEPS CDATA #IMPLIED
```



```
>

<!ELEMENT DOCUMENT EMPTY>

<!ATTLIST DOCUMENT

    XML:LINK CDATA #FIXED "document"

    HREF CDATA #REQUIRED

>
```

不知到您有没有注意到,这些宣告大体上都比前面的例子短,此处唯一使用到的新属性为 **STEPS** 属性,它是用来告诉应用程序在结束搜寻前有多少层的文件需要浏览,当群组包含文件,而文件中又包含其它群组,然后其它群组又包含其它文件,这种关系一直延伸下去时可以使用 **STEPS** 这个属性。在元素宣告的后面,我们需要加入 **Group** 与 **Document** 元素来建立我们的连结群组。

```
<GROUP STEP=2>

    <DOCUMENT HREF="minivans.htm"/>

    <DOCUMENT HREF="passengercars.htm"/>

    <DOCUMENT HREF="pickups.htm"/>

    <DOCUMENT HREF="sportscars.htm"/>

</GROUP>
```

如果有一份叫做 **Reviews.xml** 的文件包含了上述的元素群组,应用程序便会处理所有的文件,以及检验返回 **Review.xml** 文件的连结。如果连结存在,应用程序便会建立 **Reviews.xml** 可以取

得的连结清单，并且开放这些连结给使用者使用。延伸连结群组提供一种透过建立中央核心文件来集中管理连结的方法，而这份核心文件包含所有其它的连结。如下图 7-5 的说明，如果无法取得指向某个资源的连结，则该资源本身如同连结一样，亦是无法使用的。

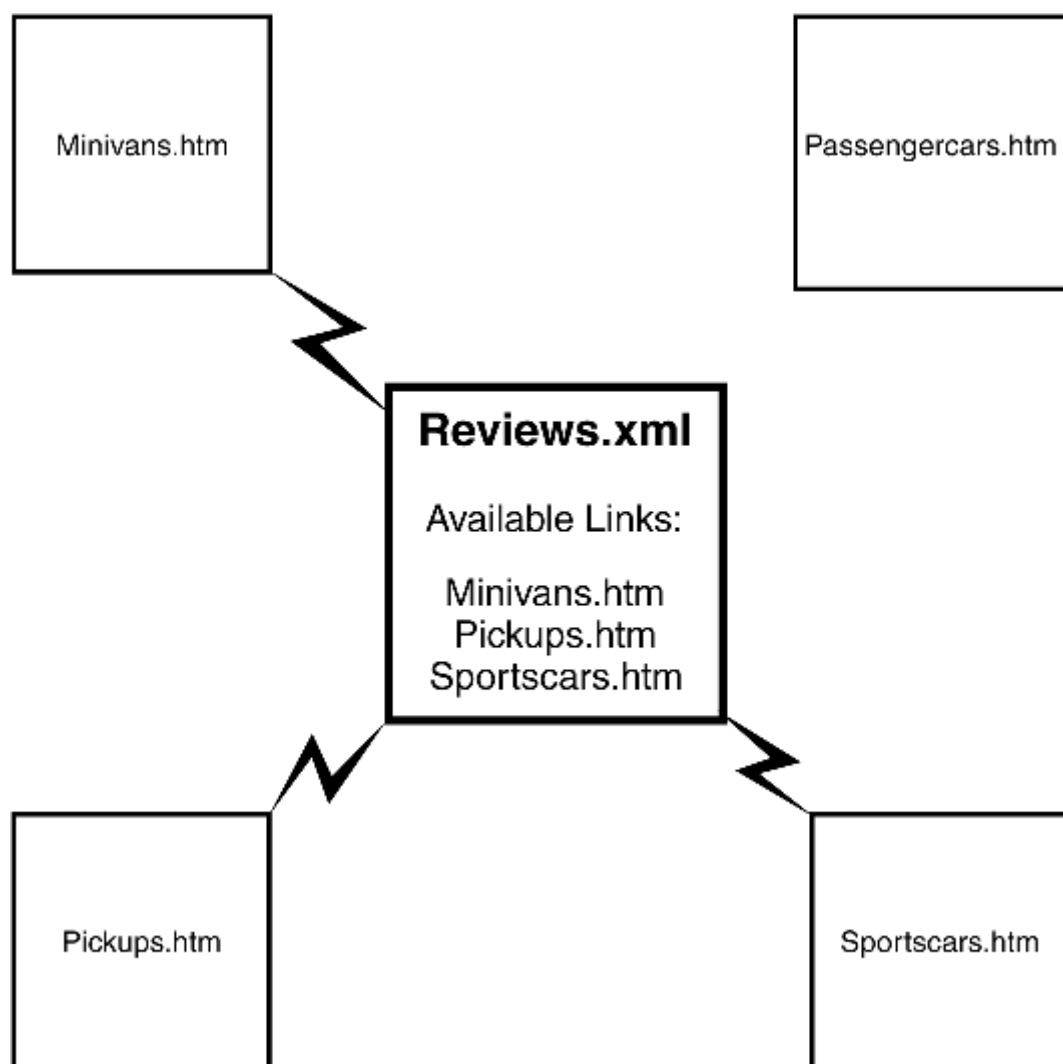


图 7-5：使用连结群组管理连结。

到目前为止，我们所看到的连结都是连接到外部内容或文件章节，但这里有一种更严谨的方式来连结到内部的 XML 文件。这可以由其它以 XML 为基础的语言来完成，我们称为 Xpointer。

XPointer 探讨

XPointer 最主要的目的在提供 XML 文件一个处理内部架构的方法。在前面的例子中，如果我们想要处理文件中指定的部分，我们都是使用片段指针与元素的 ID 属性来处理。而 XPointer 被设计用来处理文件的内部架构，不管是不是包含 ID 属性。XPointer 是一种独立的语言，可能需要一整个章节来说明它，本节将简单地介绍 XPointer 与一些其所具备的功能。请注意：第 8 章提到的 XSL（Extensible Stylesheet Language）不仅提供了相似的功能，而且在微软的 Msxml 处理器中已经被使用。

Note

本书撰写时 XPointer 仍只在草拟阶段，而且也未经过 W3C 正式的推荐，本节中的信息是以下列网址中的规格为准：<http://www.w3.org/TR/1998/WD-xptr-19980303>。

XPointer 的基本概念

XPointer 应用在元素与架构间，它用来组成 **XML** 文件的树状架构。**XPointer** 包含一连串的地址术语，用来指出文件树状中的位置。虽然，**XPointers** 比 **HTML** 中的片段指标复杂，但它的用法与使用片段指标则是大同小异。地址术语可以使用下列其中之一的语法：

```
HREF="uri#Xpointer"
```

```
HREF="uri|Xpointer"
```

如果使用「**#**」当作分隔字符，则表示客户端应该要处理该连结；如果使用「**|**」当作分隔字符，则连接机制是左边开启的。这样提供一个连结机会可由伺服器端处理连结，同时亦可节省频宽。

一个地址术语需要一个地址来源，如此便可以告诉 **XPointer** 应该从文件架构的何处开始。每一个地址术语使用一个关键词，并且可以包含参数。关键词用来指出地址来源，像是 **Root**、**Id** 等等，这些都可以告诉 **XPointer** 该从哪儿开始，而参数则提供有关 **XPointer** 在地址来源中应该到哪儿的进一步信息。举例而言，一个地址术语可能如下所示：

```
Child (2 , PRODUCT)
```

上述文字码表示「第二个 **Product** 元素为目前元素的子元素之一」。

地址术语可以对应到绝对地址、相对地址、跨越（**spanning**）地址、属性地址或是字符串地址。

现在就让我们看看一些不同型态的地址术语，以便了解它们是如何运作的。

绝对地址术语（**Absolute Location Terms**）

一个绝对地址术语指向文件架构中指定的地方，它并不需要地址来源。一个绝对地址术语可以用来确定一个地址来源，或者也可以服务包含于自身的 **XPointer**。绝对地址术语支援下列关键词：

- **ROOT**: **Root** 关键词用来指出地址来源是所包含资源的根元素。假如没有指定其它的关键词，默认值便是 **ROOT**。因为地址术语的目的就是用来指到文件中，所以这个关键词很少被使用。
- **Origin**: 假如 **XPointer** 因为一个浏览的要求（比方说一个连结）而被处理，这个关键词便会产生有用的地址资源。如果 **XPointer** 以 **Origin** 关键词为出发点，则该地址来源便是从浏览发生的地方产生的资源。
- **Id**: **Id** 关键词使用较常见的文件浏览技巧，**ID** 和 **NAME** 为一对。也就是说浏览开始于元素的 **ID** 与指定的 **Name** 相符合。
- **Html**: 此关键词模拟片段指针在 **HTML** 文件中的运作方式，它包含了 **NAMEVALUE** 属性。地址来源为第一个锚元素，它包含了 **NAME** 属性，其属性值必须符合地址术语中的 **NAMEVALUE** 属性。

下面是一个绝对地址术语的例子：

相对地址术语 (Relative Location Terms)

相对地址术语的关键词取决于现存的地址来源。假使一个地址来源不存在，该 **XPointer** 便会认定资源的 **root** 元素包含该术语。下列为相对地址术语所支持的关键词：

- **Child**：如果使用了 **Child** 关键词，它会确认地址来源的子节点，该关键词会挑出地址来源所有的子项目。
- **Descendant**：表示文件节点存在于地址来源内容的任何地方，而不用管它的层级到底有几层。
- **Ancestor**：表示节点包含地址来源或者它的父元素。
- **Preceding**：表示节点在地址来源之前出现在文件树中。
- **Psibling**：表示兄弟元素出现在地址来源之前。（兄弟元素有同样的父节点）。

- **Following:** 表示节点在地址来源之后出现在文件树中。
- **Fsibling:** 表示兄弟元素出现在地址来源之后。

下面为相对地址术语的范例：

```
Child (2 , SECTION)
```

跨越地址术语（**Spanning Location Terms**）

跨越地址术语使用两个参数的数据指向一个次资源，而这两个参数与跨越地址术语的地址来源有关，例如：

```
Id (PRODUCT) .Span (Child (1) , Child (3) )
```

属性地址术语（**Attribute Location Terms**）

属性地址术语使用属性名称找到该属性，然后将属性值传回。底下是它的用法：

```
Attr (COLOR)
```

字符串地址术语（**String Location Terms**）

字符串地址术语在地址来源中选择一个、多个字符串或位置，字符串地址术语支持下列关键词：

- **InstanceOrAll:** 确认指定字符串的发生顺序。如果该数目为正数，**XPointer** 从地址来源的起始点往前计算；如果该数目为负数，**XPointer** 便从地址来源的结束点往后计算。假如使用 **All** 这个值，则所有字符串都会被用到。
- **SkipLit:** 表示该字符串存在于地址来源中。
- **Position:** 表示一个字符串开始的字符或符合的最后字符串的开始。
- **Length:** 表示字符串中被选取的字符数目。

下面是字符串地址术语的例子：

```
Id (PRODUCT).String (3, "widget")
```

这些 **XPointer** 简短的介绍并不是要提供您如何在应用程序中实作的细节，而是要告诉您 **XPointer** 种类的细节并加到您的连结中，也许这样可以激起您想要在自己的 **XML** 文件中使用连结元素时，使您更想去深入了解 **XPointers**。

8. XSL：具样式的 XML

到目前为止，我们讨论的都是 XML 中和资料相关的部分，而 XML 为文件中数据所提供的架构与前后文关系，确实在文件的意义描述上助益良多，但是我们并未在 XML 文件中看到有关数据显示的方法。截至目前为止，本书都是使用 HTML 来提供显示的信息，也就是说我们浏览了 XML 文件树状架构，并且将每一个 XML 元素中的数据插到一个 HTML 元素中。虽然直接使用 HTML 是格式化 XML 文件较有效的方法之一，但我们也可以使用另一种方法：XSL（Extensible Stylesheet Language）。XSL 是一种特别针对 XML 文件所建立的格式化语言，也可以说是 XML 的一种应用，所以 XSL 的架构和语法都和 XML 极为相似。

本章中，我们将会检视 XSL 的功能，并以格式化的规则做基础，实际以 XSL 来建立具延伸能力的样式表。最后我们将进一步探讨一些 XSL 较新进的功能，比方说以 XSL 来使用 Script 并重新制作 XML 的输出画面。

样式表的基本概念

XSL 是建立在「样式表的机制」上。样式表通常为整份文件提供一致的样式或格式化信息，而在 Web 上最常用的格式就是以「串接样式表」（CSS，Cascading Style Sheets）规格为基础。

CSS 可以让设计人员定义适用于整份 HTML 文件的样式类别 (classes)。让我们看看一些 CSS 的范例，如此往后使用 XSL 时更能有较清楚的概念。

CSS 的概观

CSS 的运作包含两个部分：一为包含所有样式定义或规则的样式表，另一为一份应用这些样式的文件。首先，让我们建立包含几个简单样式规则的样式表。文字码 8-1（随书光盘中的 Chap08\Lst8_1.css）显示两个样式规则：

```
H1 {font-style:italic; font-size:24}

.bold16 {font-weight:bold; font-size:16}
```

文字码 8-1

再来，我们必须建立一份使用此样式的 HTML 文件，如文字码 8-2 所示。（随书光盘中的 Chap08\Lst8_2.htm）请注意：该 HTML 文件连结到前面所展示的样式表，而<LINK>卷标告诉应用程序要在该文件指定的部分使用该样式表定义的样式。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>


    <HEAD>


        <TITLE>Code Listing 8-2</TITLE>
```

```
<LINK HREF="Lst8_1.css" REF=STYLESHEET TYPE="text/css">

</HEAD>

<BODY>

    <H1>An H1 paragraph</H1>

    <BR>

    <SPAN CLASS=bold16>

        A Span element with the bold16 style rule applied

    </SPAN>

</BODY>

</HTML>
```

文字码 8-2

文字码 8-1 的样式表用了两种不同方法来定义出两个样式规则。第一个规则（H1）直接对应到 HTML 的 H1 元素；换句话说，HTML 文件中的每个 H1 元素都将使用样式表中该样式的定义，而非该元素的预设样式。第二个样式规则（bold16）为一个样式规则类别（**class**，以.表示的皆是），所以并非指定任何特殊的 HTML 元素。两个样式规则都包含自订的格式（换句话说，我们可以决定该样式规则的显示定义），但是 bold16 样式规则也包含了自订的名称，即 bold16；也就是说，因为 bold16 是一个样式规则类别，我们便可以替它取名字。要在上述 HTML 文件中使用该样式规则类别，必须使用 **CLASS** 属性，明确地指定该规则给使用的 **Span** 元素，如此便把该样式规则定义以相同的名称指定给目前的元素。

当显示该页时，看起来应该如图 8-1 所示。

现在我们再使用样式表的一些技巧，以便您能全盘了解 CSS 的功用。因为我们知道 H1 样式规则对应到 HTML 的 H1 元素，所以该样式适用于包含在文件中的每一个 H1 元素。但我们也可以为一个特定的 H1 元素指定不同的样式类别，以从样式表中将新的样式重写（override）到该元素。举例而言，您可以用 bold16 的样式类别来建立一个 H1 元素，就像文字码 8-3 所示（随书光盘中的 Chap08\Lst8_3.htm）。

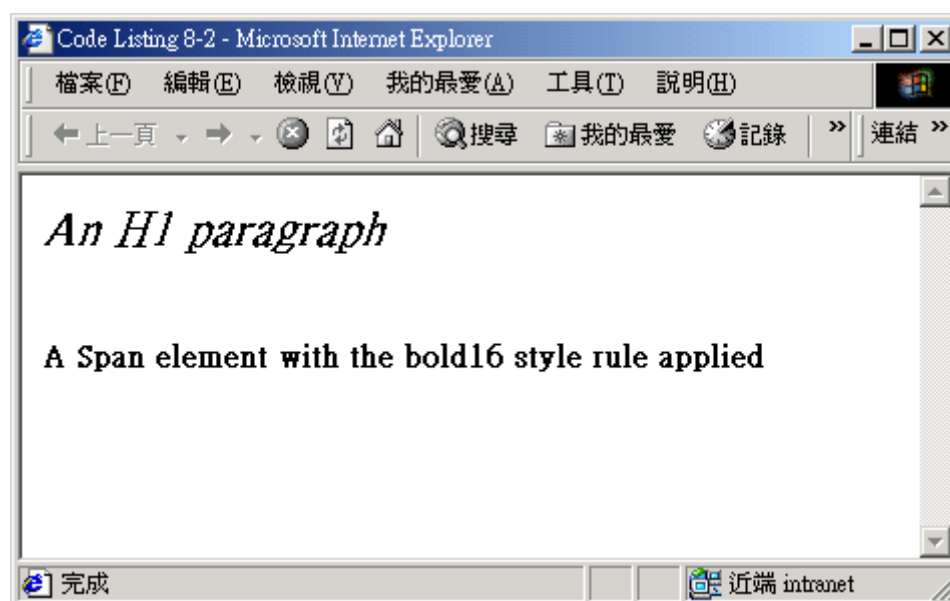


图 8-1：此 HTML 文件使用一个 CSS 样式表来定义两个元素的样式。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

```
<HEAD>

    <TITLE>Code Listing 8-3</TITLE>

    <LINK HREF="Lst8_1.css" REL=STYLESHEET TYPE="text/css">

</HEAD>

<BODY>

    <H1>An H1 paragraph</H1>

    <BR>

    <SPAN CLASS=bold16>

        A Span element with the bold16 style rule applied

    </SPAN>

    <P>

    <H1 CLASS=bold16>

        An H1 paragraph with an overriding class style

    </H1>

    <BR>

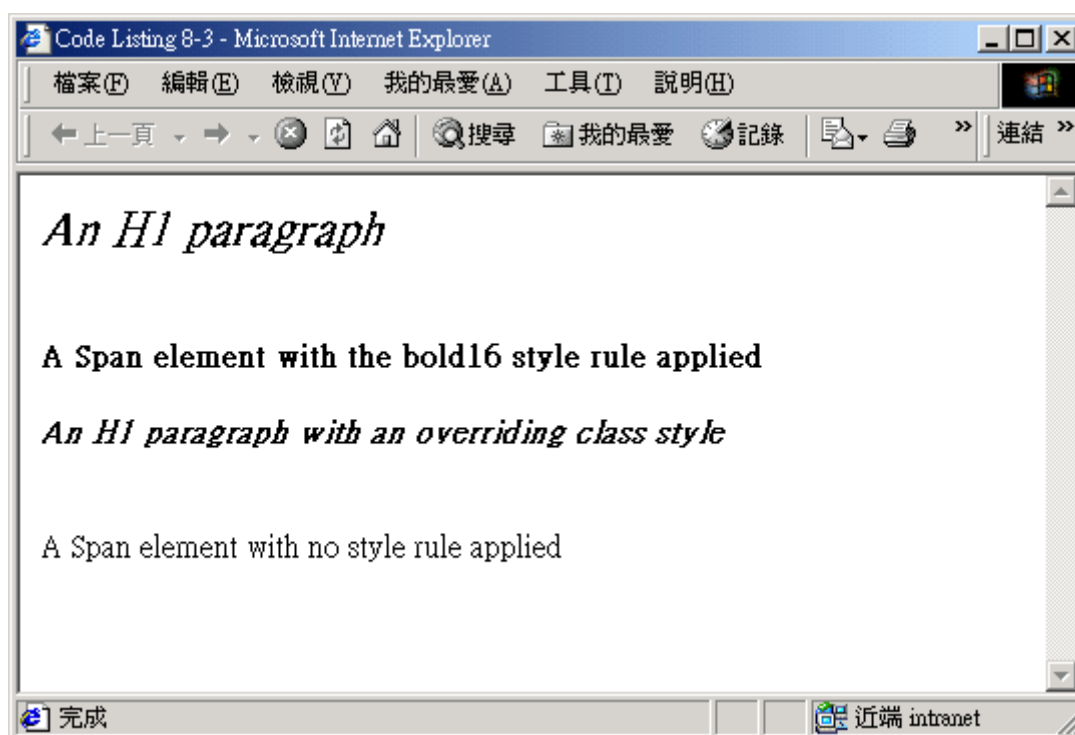
    <SPAN>A Span element with no style rule applied</SPAN>

</BODY>

</HTML>
```

文字码 8-3

当显示此页时，第二个 H1 元素的样式融合了 **bold16** 类别和原先 H1 的样式，而不仅仅是样式表中定义的 H1 而已。请注意：重载过的样式表增加了新的属性（本例中为加深的黑体字），而且只是改变相同名称的属性值（本例中为字号），至于斜体字的字型样式属性仍是由原先 H1 的样式而来。



瞧瞧第二个 Span 元素，它没有包含任何的样式信息，所以显示时使用预设的一些设定，亦即没有样式。

XSL 的概观

XSL 也使用样式表，它可以提供与 CSS 一样的格式化能力与弹性（甚至更多），但是它使用的是不同的处理方式。XSL 建立在样板（**template**）的想法上，有点像是 CSS 的样式规则。XSL 样板在符合所叙述的条件之下，会提供格式化的信息给对应数据。

XSL 的组成成分

在更详细探讨之前，我们必须讨论 XSL 的组成成分，这有助于我们更了解 XSL，以便更有效地利用它。XSL 语言由两部分组成：一为 XSL「转换格式语言」（**transformation language**），一为「格式化对象规格」（**formatting object specification**）。这两部分分开的原因会在下面说明，但它们可以一起提供文件显示强大的格式化功能。XSL 转换语言和格式化对象规格被实作成 XML Namespaces（请参考 [第 6 章关于 Namespaces 的讨论](#)）。

Note

W3C 组织经常使用「**formatting vocabulary**」来描述前面提到的「格式化对象规格」，请不要为此感到困惑。就像使用其它的词汇一样，**formatting vocabulary** 就是一种规格，所以规格（**specification**）和语汇（**vocabulary**）是一样的（请参考 [第 4 章关于语汇的细节](#)）。

XSL 转换语言（XSL Transformation Language）

XSL 转换语言（**xsl namespace**）是一个告知处理器如何把一份 XML 文件从某个结构转换成另一种结构的叙述语言。这种转换动作可以将一份文件的树形图转换成另一份文件的树形图。最可能也最明显的用途（至少最初）便是将一份描述语意的结构转换成显示的结构，比如说将一份 XML 文件转换成一份 HTML 文件。但事实上并不一定要如此应用，因为转换的动作和最后的输出是独立的，所以未来有着很大的扩展性，也就是说 XSL 可以将文件转换成现在还未想到的文件结构。

格式化对象规格（Formatting Object Specification）

格式化对象规格（**fo namespace**）可提供新的格式化语意，而可以发展成一种新的 XML 语汇。因此一个显示引擎要可以直接处理 **fo namespace** 中的格式化信息（与 **xsl namespace** 中的信息不同），甚至处理器还能更进一步的转换该格式化信息为另一种格式化信息的结构，比如 HTML 的文字码。这个方法和 **xsl namespace** 方法最大的不同在于：**fo namespace** 中的方法是针对语意的格式化，以针对特定的应用（比如多媒体）发展语汇；而 **xsl namespace** 的功能主要是针对文件对象模型（**document object model**）做转换的动作，这和语意的格式化是独立的。

Note

XSL 在微软 IE5.0Msxml 中的实作，几乎把重心都放在 HTML 文字码的输出上，所以 Msxml 是以 xsl namespace 的方式发展成的。因此，本章剩下的部分会把重点放在使用 xsl namespace 上。

XSL 样式表（XSL Style Sheets）

如同前面所说的，XSL 有能力将一份 XML 文件转换为其它的输出结构，这是由样式表完成的。

XSL 样式表描述的是应该建立何种输出架构的程序。

使用样板（Templates）

一份样式表通常由一个或多个包含了「模块样式」（patterns）的样板组成。如同其名，样板提供的是输出文件的架构，而输出的元素则可以是任何东西。XSL 样板甚至也不一定需要包含任何所参照到的 XML 数据。例如下面所列的样板：

```
<xsl:template xmlns:xsl="uri:xsl">

  <HTML>

    <HEAD>
```

```
<TITLE>XSL Test</TITLE>

</HEAD>

<BODY>

    <B>This is a test of an XSL template.</B>

</BODY>

</HTML>

</xsl:template>
```

该样板只以 **HTML** 元素当作它的输出元素。如果看得更仔细些，您将发现此样式表并不实用，因为它只能输出静态的 **HTML** 文字码，而没有参照到任何的 **XML** 资料；但是它的确说明了样板是如何描述输出结构的。其实样式表真正的功用在于将 **XML** 文件转换输出成新文件的能力，而 **XSL** 样板是经由模块样式（**pattern**）来参照 **XML** 资料的。

模块样式（**Patterns**）

XSL 使用 **pattern** 来指定 **XSL** 样板所要套用的 **XML** 元素。这种 **pattern** 的比对方式，使 **XSL** 成为一种宣告式的语言（**declarative language**），而非程序式的语言。也就是说，**pattern** 透过在文件树中阶层结构的指定，来指定在文件树中所要比对的特定分支。举例来说，「**ROOT/NODE1**」**pattern** 便是指出「**Node1** 元素是存在于 **Root** 元素中」。我们来看看一份简易的样式表，以便让您了解样板的架构。首先必须建立一份要使用该样式表的 **XML** 文件。

```

<?xml version="1.0" ?>

<?xml-stylesheet type="text/xsl" href="sample.xsl" ?>

<CATALOG>

    <PLANT>

        <COMMON>Bloodroot</COMMON>

        <BOTANICAL>Sanguinaria canadensis</BOTANICAL>

        <ZONE>4</ZONE>

        <LIGHT>Mostly Shady</LIGHT>

        <PRICE>$7.05</PRICE>

        <AVAILABILITY USONLY="true">02/01/99</AVAILABILITY>

    </PLANT>

</CATALOG>

```

再来，我们以单一样板为 **Common** 元素建立一份 **sample.xsl** 的 XSL 样式表：

```

<?xml version="1.0"?>

<xsl:template xmlns:xsl="uri:xsl">

    <HTML>

        <BODY>

            <xsl:for-each select="CATALOG/PLANT">

                <DIV>

```

```

                                <SPAN STYLE="font-weight:bold;
font-size:20">

                                <xsl:value-of select="COMMON"/>

                                </SPAN>

                                </DIV>

                                </xsl:for-each>

                                </BODY>

                                </HTML>

</xsl:template>

```

当这份 XML 文件透过 `<?xml-stylesheet type="text/xsl" href="sample.xsl" ?>` 的叙述使用 XSL 样式表时，处理器会输出下列 HTML 文字码：

```

<HTML>

<BODY>

<DIV>

<SPAN STYLE="font-weight:bold; font-size:20">

Bloodroot

</SPAN>

</DIV>

</BODY>

```

```
</HTML>
```

Note

本章所提到的例题都是输出 **HTML** 文字码，但请记住 **XSL** 规格支持其它的输出结构，而这些结构与 **XSL** 语言都是互相独立的。

现在请看看我们在上面样式表定义的样板，并检视当文件进行时会发生什么情况。

解析样板

每个样板会包含一个或多个模块样式（**pattern**）。而上面的例子便包含了两个 **pattern**，底下便是包含 **pattern** 的部分：

```
<xsl:for-each select="CATALOG/PLANT">
<DIV>
    <SPAN STYLE="font-weight:bold; font-size:20">
        <xsl:value-of select="COMMON"/>
    </SPAN>
</DIV>
```

```
</xsl:for-each>
```

第一个 **pattern** 指定了要使用 **Catalog** 元素的任意 **Plant** 子元素（您可以先忽略 **for-each** 的叙述，部分我们将在后面说明）。第二个 **pattern** 则指定 **Common** 元素。这个规则把在 **pattern** 指定元素中所发现的数据放入 **Span** 元素中，并在 **Span** 元素中使用「**font-weight:bold; font-size:20**」的显示样式。所以假使我们要以日常用语来描述这个样板（**template**），我们便可以说：针对 **Catalog** 元素的每个 **Plant** 子元素部分，取出其 **Common** 元素的值，并将该值放到具「加深黑体字」和「字号为 20」显示方式的 **Span** 元素中。

样板可以套用到文件中每个符合样板中 **pattern** 或已定义过 **pattern** 的元素。为了进一步说明这情形，让我们加入一些 **Plant** 元素到范例中：

```
<CATALOG>

<PLANT>

    <COMMON>Bloodroot</COMMON>

    <BOTANICAL>Sanguinaria canadensis</BOTANICAL>

    <ZONE>4</ZONE>

    <LIGHT>Mostly Shady</LIGHT>

    <PRICE>$7.05</PRICE>

    <AVAILABILITY USONLY="true">02/01/99</AVAILABILITY>

</PLANT>
```

<PLANT>

<COMMON>Columbine</COMMON>

<BOTANICAL>Aquilegia canadensis</BOTANICAL>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$3. 20</PRICE>

<AVAILABILITY>04/08/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Marsh Marigold</COMMON>

<BOTANICAL>Caltha palustris</BOTANICAL>

<ZONE>4</ZONE>

<LIGHT>Mostly Sunny</LIGHT>

<PRICE>\$2. 90</PRICE>

<AVAILABILITY>01/09/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Cowslip</COMMON>

```
<LIGHT>Mostly Shady</LIGHT>

<PRICE>$3.83</PRICE>

<AVAILABILITY USONLY="true">05/19/99</AVAILABILITY>

</PLANT>

</CATALOG>
```

假使我们在这份文件中使用与之前范例中相同的 XSL 样式表，便会输出下列结果：

```
<HTML>

<BODY>

<DIV>

<SPAN STYLE="font-weight:bold; font-size:20">

Bloodroot

</SPAN>

</DIV>

<DIV>

<SPAN STYLE="font-weight:bold; font-size:20">

Columbine

</SPAN>

</DIV>

<DIV>
```



```
<SPAN STYLE="font-weight:bold; font-size:20">
```

Marsh Marigold

```
</SPAN>
```

```
</DIV>
```

```
<DIV>
```

```
<SPAN STYLE="font-weight:bold; font-size:20">
```

Cowslip

```
</SPAN>
```

```
</DIV>
```

```
</BODY>
```

```
</HTML>
```

请注意，这个样板仅会套用到 **pattern** 指定的元素，其它元素则会被忽略，这也显示了某些 **XSL** 的强大功能。当我们使用 **XSL pattern** 的比对功能时，您可以完全地重新安排 **XML** 数据以符合特定的目的。假如在输出结果中不要包含某些元素，只要不包含对应元素的 **pattern** 即可；若只想使用 **XML** 文件特定一部分的单一元素，您也可以建立一个更详细的 **pattern** 来指定所欲取出的部分。

单一样板结构

请注意我们所建立的样板是以<xsl:template xmlns:xsl="uri:xsl"> 标签开始，并以

</xsl:template>标签结束的。使用这一对标签（称为「容器」，**container**）可以让我们在使用多

个样板时，更容易指定不同部分样式表的部分内容，我们稍后便会看到。

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

<xsl:document xmlns:xsl="uri:xsl">

  <HTML>

    <BODY>

      <xsl:for-each select="CATALOG/PLANT">

        <DIV>

          <SPAN STYLE="font-weight:bold; font-size:20">

            <xsl:value-of select="COMMON"/>

          </SPAN>

        </DIV>

      </xsl:for-each>

    </BODY>

  </HTML>

</xsl:document>
```

上面的样式表是「单一样板结构」（**simple template structure**）的范例。也就是说，每一个样式表皆是由单一样板所组成的。其实样式表也可以拥有「多重样板结构」（**multiple template structure**），即包含了多个可以独立套用的样板。

多重样板结构

多重样板样式表使用的容器卷标为<xsl:stylesheet></xsl:stylesheet>，而在这一对标签里可以包含多对的<xsl:template></xsl:template> 卷标，同时每一对标签都可以独立地套用。让我们看看底下的例子，文字码 8-4 显示本范例中我们要使用的 XML 档案。

```
<CATALOG>

  <PLANT BESTSELLER="no">

    <NAME>

      <COMMON>Bloodroot</COMMON>

      <BOTAN>Sanguinaria canadensis</BOTAN>

    </NAME>

    <GROWTH>

      <ZONE>4</ZONE>

      <LIGHT>Mostly Shady</LIGHT>

    </GROWTH>
```

<SALESINFO>

<PRICE>\$3.00</PRICE>

<AVAILABILITY>4/21/99</AVAILABILITY>

</SALESINFO>

</PLANT>

<PLANT BESTSELLER="yes">

<NAME>

<COMMON>Columbine</COMMON>

<BOTAN>Aquilegia canadensis</BOTAN>

</NAME>

<GROWTH>

<ZONE>3</ZONE>

<LIGHT>Mostly Shady</LIGHT>

</GROWTH>

<SALESINFO>

<PRICE>\$9.00</PRICE>

<AVAILABILITY>4/10/99</AVAILABILITY>

</SALESINFO>

</PLANT>

<PLANT BESTSELLER="no">

```
<NAME>

  <COMMON>Marsh Marigold</COMMON>

  <BOTAN>Caltha palustris</BOTAN>

</NAME>

<GROWTH>

  <ZONE>4</ZONE>

  <LIGHT>Mostly Sunny</LIGHT>

</GROWTH>

<SALESINFO>

  <PRICE>$9.00</PRICE>

  <AVAILABILITY>4/19/99</AVAILABILITY>

</SALESINFO>

</PLANT>

</CATALOG>
```

文字码 8-4

Note

随书光盘中，档案 Chap08\Lst8_4.xml 为文字码 8-4 的完整版本。因为该文件包含了很多重复的情形，所以本书内文中便以精简形式列出以节省空间。

文字码 8-5 显示我们要使用的样式表，该样式表包含多个样板，而您也将了解到这些样板是如何独立套用到 XML 文件的不同部分。（您可以在随书光盘中找到该样式表，档案为 Chap08\Lst8_5.xsl）。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <TABLE BORDER="1">

          <TR STYLE="font-weight:bold">

            <TD>Common Name</TD>

            <TD>Botanical Name</TD>

            <TD>Zone</TD>

            <TD>Light</TD>

            <TD>Price</TD>

            <TD>Availability</TD>

          </TR>

          <xsl:for-each select="CATALOG/PLANT">
```

```

        <TR>

            <xsl:apply-templates/>

        </TR>

    </xsl:for-each>

</TABLE>

</BODY>

</HTML>
</xsl:template>

<xsl:template match="NAME">

    <TD><xsl:value-of select="COMMON"/></TD>

    <TD><xsl:value-of select="BOTAN"/></TD>

</xsl:template>

<xsl:template match="GROWTH">

    <TD><xsl:value-of select="ZONE"/></TD>

    <TD><xsl:value-of select="LIGHT"/></TD>

</xsl:template>

<xsl:template match="SALESINFO">

```

```
<TD><xsl:value-of select="PRICE"/></TD>

<TD><xsl:value-of select="AVAILABILITY"/></TD>

</xsl:template>

</xsl:stylesheet>
```

文字码 8-5

请注意：样式表以<xsl:stylesheet>标签作为开始，而以</xsl:stylesheet>标签作为结束。因为一份 XML 文件只能有一个根（root）元素，而且 XSL 是 XML 的一种应用。在样式表中，单一的 xsl:stylesheet 元素允许包含多个 xsl:template 元素。

请注意：模块样式（pattern）的第一个样板简化为「/」，这表示是 XML 文件的根（root）元素。

也请注意在这个样板中的 xsl:for-each 元素，这个元素代表另一种 pattern，CATALOG/PLANT

描述了输出结构，这些结构会在符合 Pattern 的地方被套用。在该输出结构中的是

xsl:apply-templates 元素，它告诉处理器在样式表中寻找其它符合的样板并套用它们，这就是多

样板被引入的地方。您可能会注意到在样式表中的其它三个样板，每一个都拥有属于自己的

pattern，因为 xsl:apply 样板元素存在于 CATALOG/PLANT pattern 中，其它样板只有在该元

素中发现符合的地方才套用。比如说，NAME 样板只套用在 CATALOG/PLANT/NAME 阶层中

所存在的元素。

在此，先让我们回顾一下之前所讨论过的，XSL 转换 XML 到输出元素，这些输出元素透过套用一些格式化的分类到 XML 资料上来准备显示的资料，但这并不是唯一必要的目的。该转换发生的顺序如下：

1. 一份样式表指定在 XML 文件中找到符合数据的 pattern。这些 pattern 是独立样板的一部分并包含输出结构。
2. 处理器找到符合每个 pattern 的数据，并将数据转换到输出结构。
3. 当整个样式表被执行后，一个新的数据结构会以存在于计算机内存样式表的输出为基础。

现在我们已经准备好，马上可以动手去处理数据啰！

显示输出元素

现在介绍 XSL 的另一个部分，也就是显示的部分。在我们的范例中，XSL 处理器的输出为 HTML 文字码，所以我们必须让 HTML 文字码可以被显示。其中的一个方法就是，该应用程序将 HTML 写到另一个独立的档案中，如此便可以建立一份新的 HTML 文件；另一个方法（这个方法稍后在本书中会使用到）就是简单地将 XSL 样式表产生的 HTML 文字码插到一份 HTML 文件中，这样做会允许处理与显示发生在同一份文件中，而使用者也不用为了看到结果而再做其它的处理。

文字码 8-6（在随书光盘中的 Chap08\Lst8_6.htm）是我们要用来显示范例的 HTML 网页。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

```
<HEAD>
```

```
<SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="onload">
```

```
var source = new ActiveXObject ("Microsoft.xmlDOM") ;
```

```
source.load ("Lst8_4.xml") ;
```

```
var style = new ActiveXObject ("Microsoft.xmlDOM") ;
```

```
style.load ("Lst8_5.xsl") ;
```

```
document.all.item ("xslContainer").innerHTML =
```

```
source.transformNode (style.documentElement) ;
```

```
</SCRIPT>
```

```
<TITLE>Code Listing 8-6</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<DIV ID="xslContainer"></DIV>
```

```
</BODY>
```

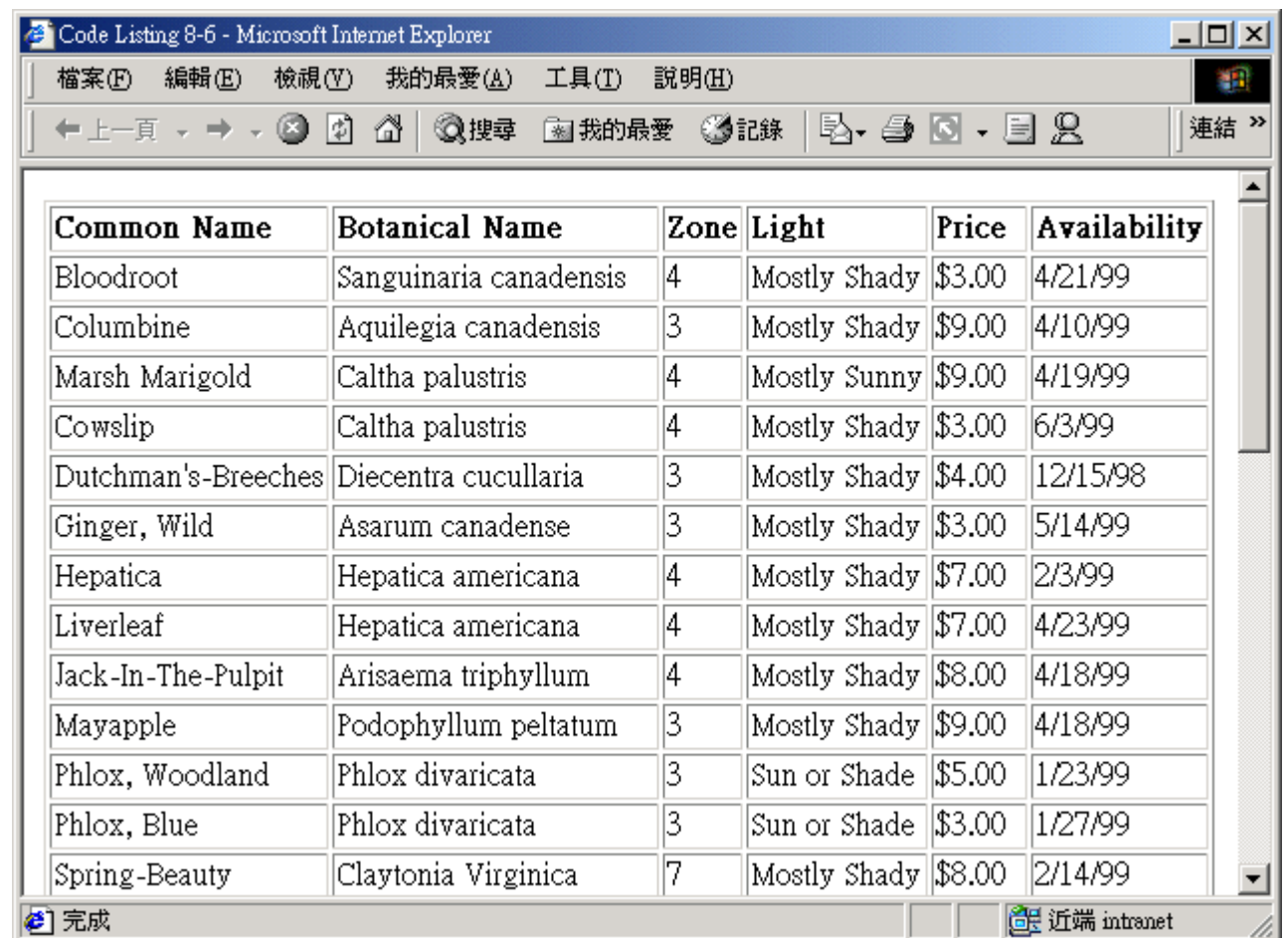
</HTML>

文字码 8-6

这个 HTML 网页使用一个 Div 元素作为 HTML 输出元素的容器，而此 HTML 输出元素是由 XSL

处理器产生的。已有两个 XML 对象被建立：一个包含 XML 文件，而另一个则包含 XSL 样式表。

该样式表对象使用 transformNode 方法来套用 XML 对象，图 8-2 显示了结果。



The screenshot shows a Microsoft Internet Explorer window titled "Code Listing 8-6 - Microsoft Internet Explorer". The address bar is empty. The menu bar includes "档案(F)", "編輯(E)", "檢視(V)", "我的最愛(A)", "工具(T)", and "說明(H)". The toolbar contains buttons for "上一頁", "後一頁", "停止", "刷新", "主页", "搜尋", "我的最愛", "記錄", "打印", "复制", "粘贴", "地址", "连接", and "后退". The main content area displays a table with the following data:

Common Name	Botanical Name	Zone	Light	Price	Availability
Bloodroot	Sanguinaria canadensis	4	Mostly Shady	\$3.00	4/21/99
Columbine	Aquilegia canadensis	3	Mostly Shady	\$9.00	4/10/99
Marsh Marigold	Caltha palustris	4	Mostly Sunny	\$9.00	4/19/99
Cowslip	Caltha palustris	4	Mostly Shady	\$3.00	6/3/99
Dutchman's-Breeches	Diecentra cucullaria	3	Mostly Shady	\$4.00	12/15/98
Ginger, Wild	Asarum canadense	3	Mostly Shady	\$3.00	5/14/99
Hepatica	Hepatica americana	4	Mostly Shady	\$7.00	2/3/99
Liverleaf	Hepatica americana	4	Mostly Shady	\$7.00	4/23/99
Jack-In-The-Pulpit	Arisaema triphyllum	4	Mostly Shady	\$8.00	4/18/99
Mayapple	Podophyllum peltatum	3	Mostly Shady	\$9.00	4/18/99
Phlox, Woodland	Phlox divaricata	3	Sun or Shade	\$5.00	1/23/99
Phlox, Blue	Phlox divaricata	3	Sun or Shade	\$3.00	1/27/99
Spring-Beauty	Claytonia Virginica	7	Mostly Shady	\$8.00	2/14/99

The status bar at the bottom shows "完成" (Done) and "近端 intranet" (Local intranet).

图 8-2：使用 HTML 网页来显示 XSL 输出。

从 XML 取得资料

正如前面例子所示范的，多种 XSL 元素可以用来从 XML 文件中取得数据，XSL 元素与 XSL 属性的组合提供一个强大的机制来从 XML 文件中准确地取得所需的数据。

直接取得

取得数据到样板中最直接的方法就是使用 `xsl:value-of` 元素与 `select` 属性。`xsl:value-of` 元素取得 `select` 属性所指定的元素值，然后该值会以文字形态插到样板中。请参考文字码 8-7（在随书光盘中的 Chap08\Lst8_7.xml）的示范。

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

<xsl:template xmlns:xsl="uri:xsl">

  <H1><xsl:value-of select="CATALOG/PLANT/NAME/COMMON"/></H1>

  <H2><xsl:value-of select="CATALOG/PLANT/NAME/BOTAN"/></H2>

</xsl:template>
```

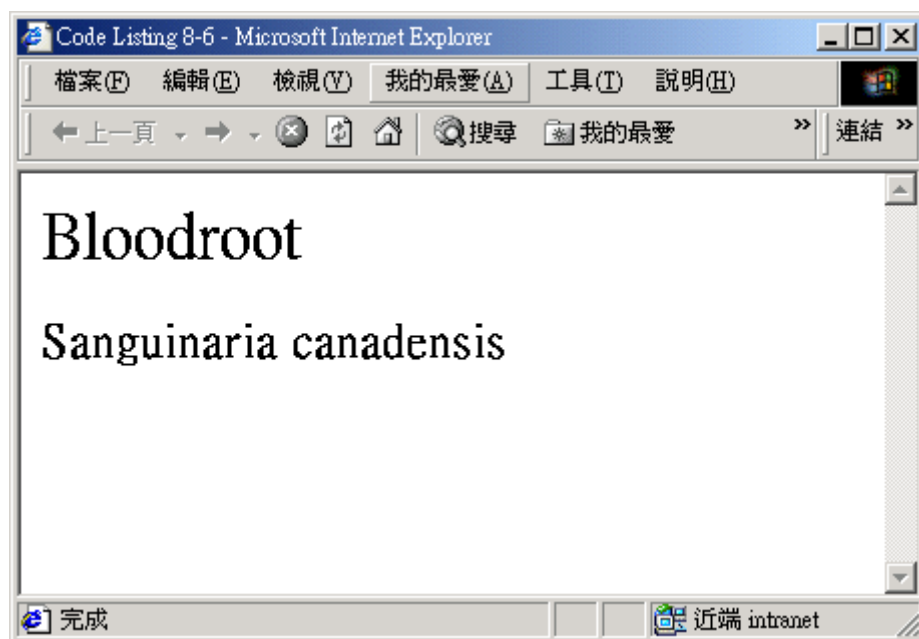
文字码 8-7

要使用文字码 8-7 的样式表，请修改 Lst8_6.htm 档案，并使用 Lst8_7.xsl 作为 XSL 档案，如下

所示：

```
var style=new ActiveXObject ("Microsoft.xmlDOM") ;  
  
style.load ("Lst8_7.xsl") ;
```

当该样式表被执行后，结果会如下图所示：



Note

除非有其它的批注，本章中剩余的范例将使用 Lst8_4.xml 的 XML 档案与 Lst8_6.htm 的 HTML 网页，您也可以试着在 HTML 网页中改变 XSL 档案的名称并观看结果。

`xsl:value-of` 元素有一个相等的属性。也就是说，您可以使用「{}」来包含 XML 元素的值，以便插入到指定 HTML 元素的属性中。举例来说，下面的 Div 元素跟着（您想在 XSL 文件中所包含的）插到 Div 元素 ID 属性中的 CATALOG/PLANT/NAME/COMMON 元素值：

```
<DIV ID={CATALOG/PLANT/NAME/COMMON}></DIV>
```

Note

XSL Pattern 会在 [第 9 章](#) 更深入地讨论，因为它是一种一般性目的查询与寻址语言，可以在文件或跨文件中用来执行查询。我们将审视本章中 XSL 会用到的 XSL Pattern。

使用多重元素

您可能会注意到在前面的例子中，每个 `xsl:value-of` 元素只传回一个元素，即使 XML 文件包含许多元素，而这些元素符合每个元素的标准，这是因为 `xsl:value-of` 元素只传回第一个符合的元

素。如果要传回每个文件中符合的元素，您可以使用 **xsl:for-each** 元素。文字码 8-8（在随书光盘中的 Chap08\Lst8_8.xml）显示此种方法。

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

<xsl:template xmlns:xsl="uri:xsl">

  <xsl:for-each select="CATALOG/PLANT/NAME">

    <SPAN STYLE="font-weight:bold; font-size:20">

      <xsl:value-of select="COMMON"/>

    </SPAN>

    <SPAN STYLE="font-weight:bold">

      (<xsl:value-of select="BOTAN"/>)

    </SPAN>

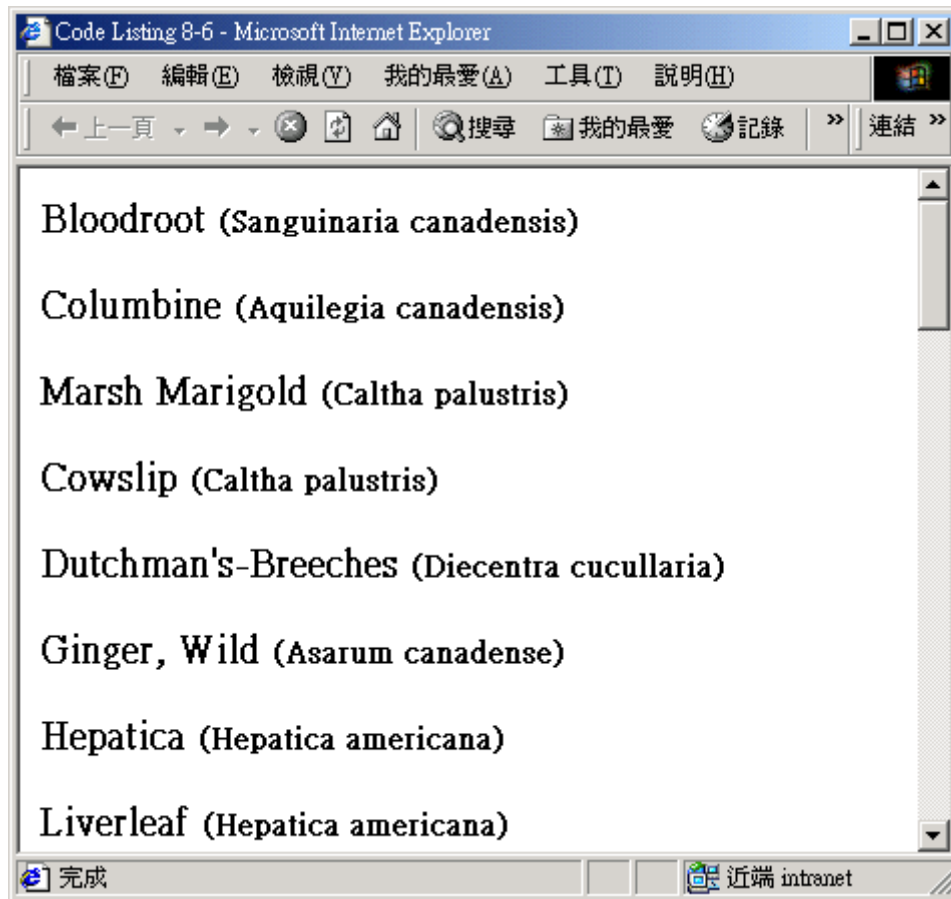
    <P></P>

  </xsl:for-each>

</xsl:template>
```

文字码 8-8

xsl:for-each 元素找到每个符合 **select** 属性值的元素，并将输出对象插入到样板输出，其结果就是一组符合所有元素的值，如下图所示：



请注意：先前样板中的 **Pattern** 指定了重复的 **CATALOG/PLANT/NAME** 元素，以及欲取得元素值的元素。当然使用前后一致或预期的结构对处理数据或 XML 文件是有益的，但是对于不具有前后一致结构的文件而言，就另当别论了。我们前面提到的多重样板样式表提供了最好的机制，来适当地取得以数据结构为基础的数据，现在就让我们更进一步的来看看。

使用多重样板

多重模板机制允许 XSL 处理器指出特定资料的 **Pattern** 需要使用什么样板。比如

xsl:apply-templates 元素告诉处理器，以查询结果为基础去寻找一个 **xsl:template** 元素，而

xsl:template 元素使用 **match** 属性来判断该元素是否符合。在 XML 文件中，每个 **Plant** 元素包

含了数个子元素，而文字码 8-9 中（在随书光盘中的 Chap08\Lst8_9.xml）的 **xsl:template** 元素，

则是用来过滤所需的信息，以便组成一份价目表。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Price List</H1>

        <TABLE CELLSPACING="4" CELLPADDING="2">

          <TR STYLE="font-weight:bold; font-size:18">

            <TD>Name</TD>

            <TD>Price</TD>

            <TD>Available</TD>

          </TR>

          <xsl:for-each select="CATALOG/PLANT">

            <TR>
```

```

        <xsl:apply-templates/>

    </TR>

</xsl:for-each>

</TABLE>

</BODY>

</HTML>

</xsl:template>

<xsl:template match="NAME">

    <TD STYLE="font-style:italic; font-size:20">

        <xsl:value-of select="COMMON"/>

    </TD>

</xsl:template>

<xsl:template match="SALESINFO">

    <TD><xsl:value-of select="PRICE"/></TD>

    <TD><xsl:value-of select="AVAILABILITY"/></TD>

</xsl:template>

</xsl:stylesheet>

```

上述结果就是一份根据每种植物名称、价格，以及有效日期的植物清单，如图 8-3 所示。

这个档案中的其它资料都被忽略掉了，因为样式表可以透过 **Pattern** 来比对资料，只选择相关的数据。请注意 **xsl:apply-templates** 元素寻找所有符合 **Plant** 元素子元素的 **xsl:template** 元素。而在每个样板中，**select** 属性选择了一个特定子元素，并将该值插入到样板中。为了更进一步说明，让我们使用一组相同的 **XML** 数据，但输出完全不同的结果。文字码 8-10（在随书光盘中的 Chap08\Lst8_10.xsl）仅取得关于成长方面的信息。

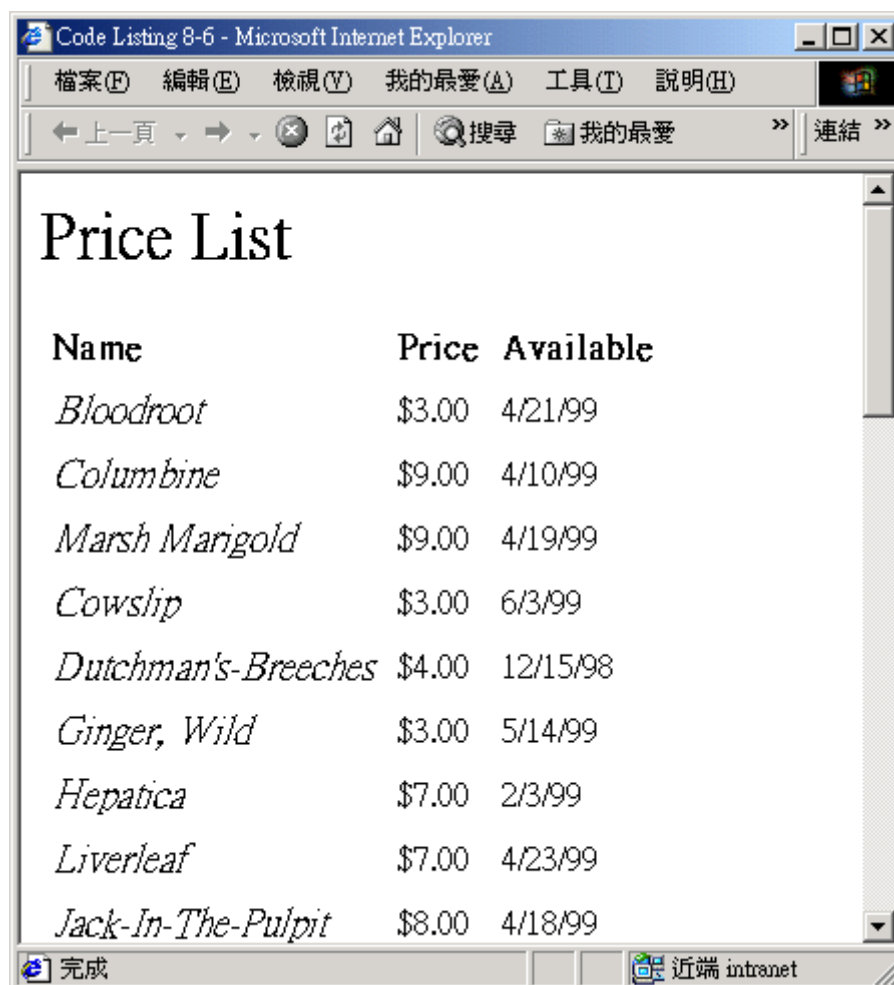


图 8-3: XSL 样式表仅取得关于价格清单的数据。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Growing Information</H1>

        <TABLE CELLSPACING="4" CELLPADDING="2">

          <TR STYLE="font-weight:bold; font-size:18">

            <TD>Botanical Name</TD>

            <TD>Zone</TD>

            <TD>Light Requirement</TD>

          </TR>

          <xsl:for-each select="CATALOG/PLANT">

            <TR>

              <xsl:apply-templates/>

            </TR>

          </xsl:for-each>

        </TABLE>

      </BODY>

    </template>

  </stylesheet>

</xml>
```

```

        </HTML>

</xsl:template>

<xsl:template match="NAME">

        <TD STYLE="font-style:italic; font-size:20">

                <xsl:value-of select="BOTAN"/>

        </TD>

</xsl:template>

<xsl:template match="GROWTH">

        <TD><xsl:value-of select="ZONE"/></TD>

        <TD><xsl:value-of select="LIGHT"></TD>

</xsl:template>

</xsl:stylesheet>

```

文字码 8-10

在此我们并没有修改 XML 文件，而只更改了样式表，结果显示在图 8-4。

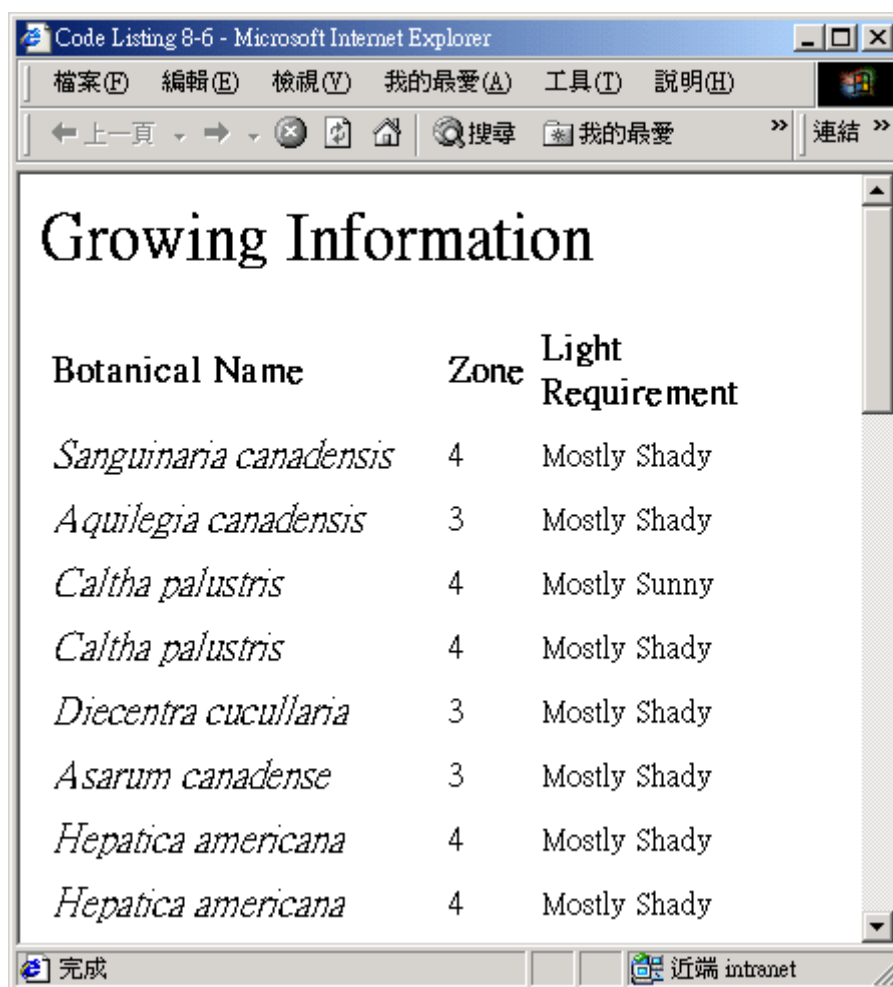


图 8-4: XSL 样式表取得了成长信息。

您可以从这些例子中看出，XSL 不仅允许文件作者能从 XML 数据来源显示精确的数据，也允许以建立的样式表为基础，完全地更新 XML 数据的结构。截至目前为止，范例已经示范了用来提供 Pattern 给样板的基本查询。如同前面所提到的，XSLPattern 在第 9 章会有更深入的解说。但既然 XSL 使用查询，我们就简短地示范如何建立可以在 XSL 样式表中使用的查询，作为您应用 XSL Pattern 的入门，底下介绍 XSL 特别重要的功能。

在 XSL 中建立查询 (Queries)

查询是 XSL 隐藏于背后的能力，虽然「查询」这个词听起来像是具高度技巧性的程序设计名词，但它并不会吓到任何 XSL 的新手，就像我们已经看到的，查询为样板提供一个简单的方法来取得正确数据。查询提供了一个透过指定 **Pattern**，然后符合该 **Pattern** 的方法来寻址数据，查询也在 XSL 中提供了一个排序的机制。

XSL 中的排序

要在 XSL 中排序，您必须在 `xsl:apply-templates` 或 `xsl:for-each` 元素中包含 `order-by` 属性，此属性用来指定数据排序的标准。文字码 8-11（在随书光盘中的 `Chap08\Lst8_11.xml`）示范如何显示以植物名称字母作为排序的列表。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Growing Information</H1>

        <TABLE CELLSPACING="4" CELLPADDING="2">

          <TR STYLE="font-weight:bold; font-size:18">

            <TD>Name (Sorted Alphabetically) </TD>
```

```

        </TR>

        <xsl:for-each select="CATAOG/PLANT" order-by="+
NAME/

        COMMON">

        <TR>

            <xsl:apply-templates/>

        </TR>

    </xsl:for-each>

</TABLE>

</BODY>

</HTML>

<xsl:template>

<xsl:template match="NAME">

    <TD STYLE="font-style:italic; font-size:20">

        <xsl:value-of select="COMMON"/>

    </TD>

</xsl:template>

</xsl:stylesheet>

```


请注意 **order-by** 标准的语法在该元素名称前包含一个正号 (+)，这也表示排序该如何发生：使

用正号 (+) 为升序排序，而使用减号 (-) 为降序排序。图 8-5 显示了排序的结果。



图 8-5：使用 XSL 来排序 XML 数据。

在查询中使用元素值

除了使用 XSL 排序资料外，您可以透过所包含的数据来过滤 XML 元素，以便以要求的数据为基础产生更精确的输出。可以使用标准操作数来建立元素值查询，其标准操作数如下所示：

- 相等 (\$EQ\$ 或 =)
- 不相等 (\$NE\$ 或 !=)
- 小于 (<&)
- 大于 (>&)

这个型态的查询显示在文字码 8-12（在随书光盘中的 Chap08\Lst8_12.xsl）中，这里的样式表样板包含一个以 **Name** 元素为过滤的查询，且 **Common** 元素包含 **Bloodroot** 值。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Plant Selection</H1>
```

```

<TABLE CELLSPACING="4" CELLPADDING="2">

  <TR STYLE="font-weight:bold; font-size:18">

    <TD>Name</TD>

  </TR>

  <xsl:for-each select="CATALOG/PLANT" order-by="+ NAME/
    COMMON">

    <TR>

      <xsl:apply-templates/>

    </TR>

  </xsl:for-each>

</TABLE>

</BODY>

</HTML>

</xsl:template>

<xsl:template match="NAME[COMMON='Bloodroot']">

  <TD STYLE="font-style:italic; font-size:20">

    <xsl:value-of select="COMMON"/>

  </TD>

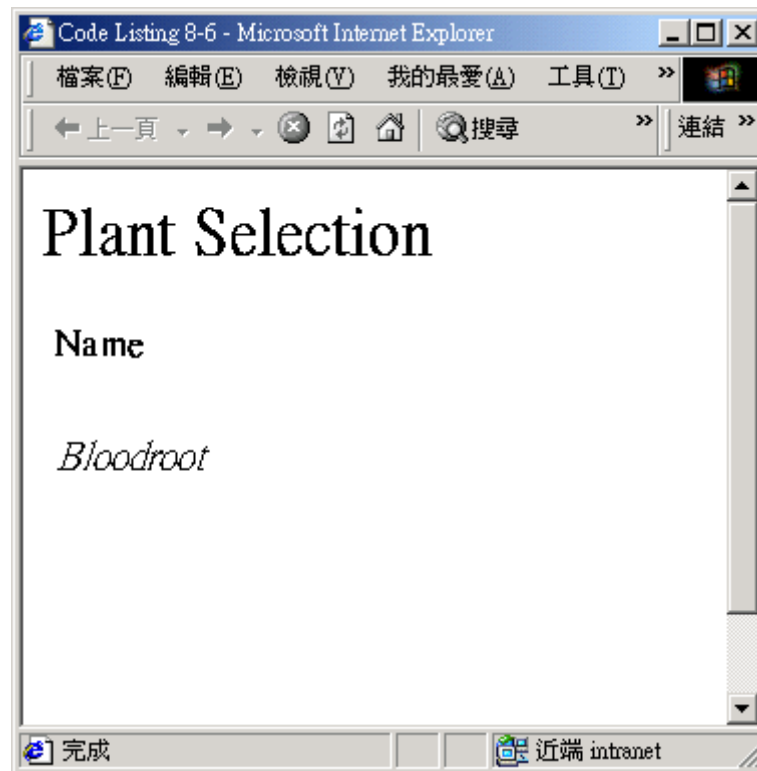
</xsl:template>

```

```
</xsl:stylesheet>
```

文字码 8-12

当该样式表被处理后，只有符合该查询的数据才会被输出，结果如下所示。



在查询中使用属性值

除了元素值外，您也可以使用属性值作为查询的标准，属性值和元素值使用的方式相同，除了属性名称必须在前面加上「@」符号。文字码 8-13（在随书光盘中的 Chap08\Lst8_13.xml）为对 **Plant** 元素的查询，该元素包含 **BESTSELLER** 属性，而其属性值为 **yes**。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Best Sellers</H1>

        <TABLE CELSPACING="4" CELLPADDING="2">

          <TR STYLE="font-weight:bold; font-size:18">

            <TD>Name (Sorted Alphabetically) </TD>

          </TR>

          <xsl:for-each select="CATALOG/PLANT[@BESTSELLER='yes']"

            order-by="+ NAME/COMMON">

            <TR>

              <xsl:apply-templates/>

            </TR>

          </xsl:for-each>

        </TABLE>

      </BODY>

    </HTML>

  </template>

</stylesheet>
```

```

        </TABLE>

    </BODY>

</HTML>

</xsl:template>

<xsl:template match="NAME">

    <TD STYLE="font-style:italic; font-size:20">

        <xsl:value-of select="COMMON"/>

    </TD>

</xsl:template>

</xsl:stylesheet>

```

文字码 8-13

此文字码中不管任何元素数据，只有符合属性标准的元素才会被输出，而此文字码也可以用来核对某存在属性并忽视任何相关的值。比方说，我们试着把上述的查询改变成：

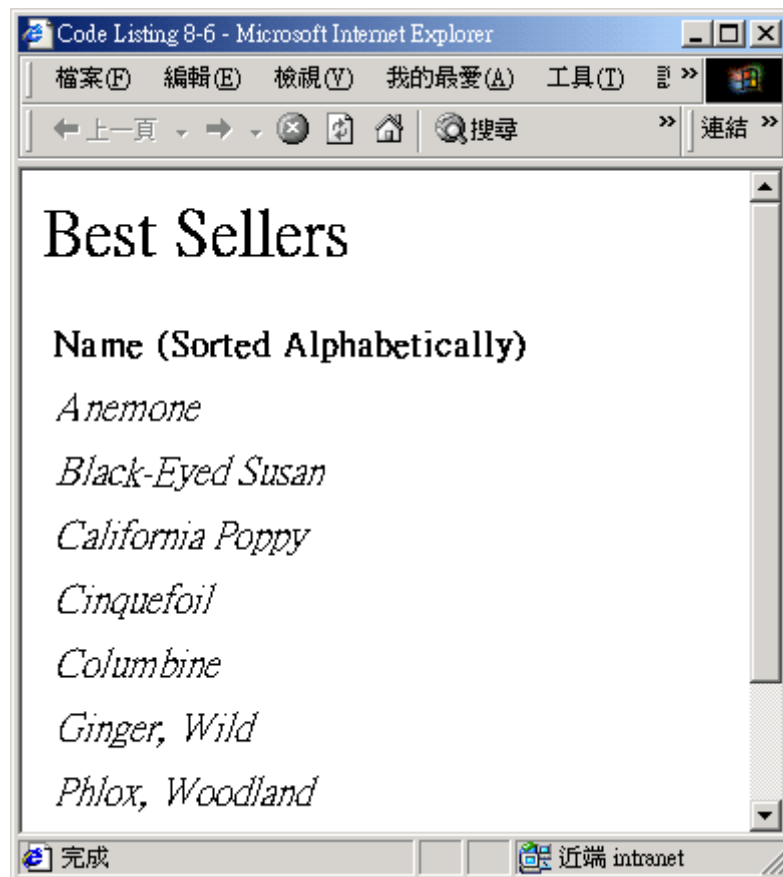
```

<xsl:for-each select="CATALOG/PLANT[@BESTSELLER]"
order-by="+ NAME/COMMON">

```

上述执行的结果会输出较长的清单列表，因为该查询传回任何包含 **BESTSELLER** 属性的元素。

文字码 8-13 的结果如右图所示：



使用条件叙述查询

除了在 XML 中排序与过滤资料，您也可以使用 `xsl:if`、`xsl:choose`、`xsl:when` 与 `xsl:otherwise`

元素来执行条件式测试。

使用 `xsl:if` 元素

xsl:if 元素与 **match** 属性组合在一起，允许使用条件式测试与子模块样式（**subpatterns**）。**match** 属性接受 **XSL pattern** 查询作为它的属性值。为了说明起见，我们将更改文字码 8-13 来测试存在的 **BESTSELLER** 属性，新的样式表显示于文字码 8-14（在随书光盘中的 Chap08\Lst8_14.xsl）。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Bestsellers</H1>

        <TABLE CELLSPACING="4" CELLPADDING="2">

          <TR STYLE="font-weight:bold; font-size:18">

            <TD>Name (Sorted Alphabetically) </TD>

          </TR>

          <xsl:for-each select="CATALOG/PLANT" order-by="+
NAME/COMMON">

            <xsl:if match=".[@BESTSELLER]">

              <TR>

                <xsl:apply-templates/>


```



```

        </TR>

    </xsl:if>

</xsl:for-each>

</TABLE>

</BODY>

</HTML>

</xsl:template>

<xsl:template match="NAME">

    <TD STYLE="font-style:italic; font-size:20">

        <xsl:value-of select="COMMON"/>

    </TD>

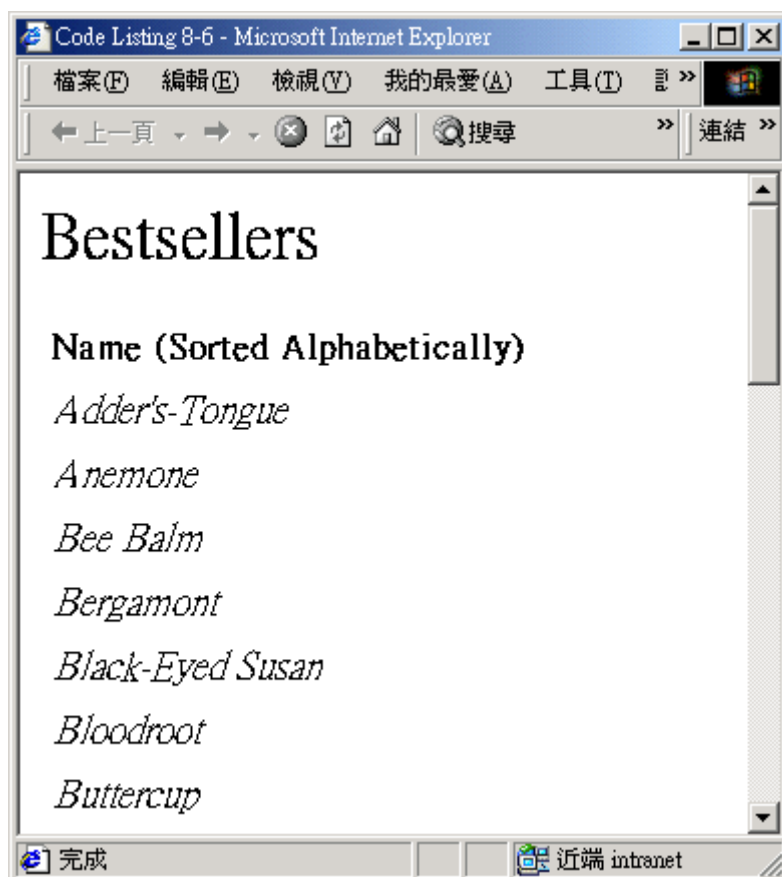
</xsl:template>

</xsl:stylesheet>

```

文字码 8-14

文字码 8-14 显示结果如下图所示：



使用 `xsl:choose` 元素

您可以使用 `xsl:choose` 元素做多个条件的测试，并以结果为基础控制输出，您也可以组合这个元素与 `xsl:when` 及 `xsl:otherwise` 元素来设定条件分支，就如同一些程序设计语言的 `if-then-else` 叙述。这样的筛选条件允许文件作者设定复杂、数据导向（**data-driven**）样式表，并提供以处理的资料为基础的输出。

文字码 8-15（在随书光盘中的 `Chap08\Lst8_15.xsl`）示范当 `BESTSELLER` 属性值为 `yes` 时，使用 `xsl:choose` 来更改输出元素。

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

```
<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Bestsellers</H1>

        <TABLE CELLSPACING="4" CELLPADDING="2">

          <TR STYLE="font-weight:bold; font-size:18">

            <TD>Name (Sorted Alphabetically) </TD>

          </TR>

          <xsl:for-each select="CATALOG/PLANT" order-by="+
NAME/COMMON">

            <xsl:choose>

              <xsl:when match=". [@BESTSELLER=' yes' ]">

                <TR BGCOLOR="#CC0000"><xsl:apply-templates/>

                  </TR>

              </xsl:when>

              <xsl:otherwise>

                <TR><xsl:apply-templates/></TR>

              </xsl:otherwise>

            </xsl:choose>

          </xsl:for-each>

        </TABLE>

      </BODY>

    </HTML>

  </template>

</stylesheet>
```

```
        </xsl:choose>

        </xsl:for-each>

    </TABLE>

</BODY>

</HTML>

</xsl:template>

<xsl:template match="NAME">

    <TD STYLE="font-style:italic; font-size:20">

        <xsl:value-of select="COMMON"/>

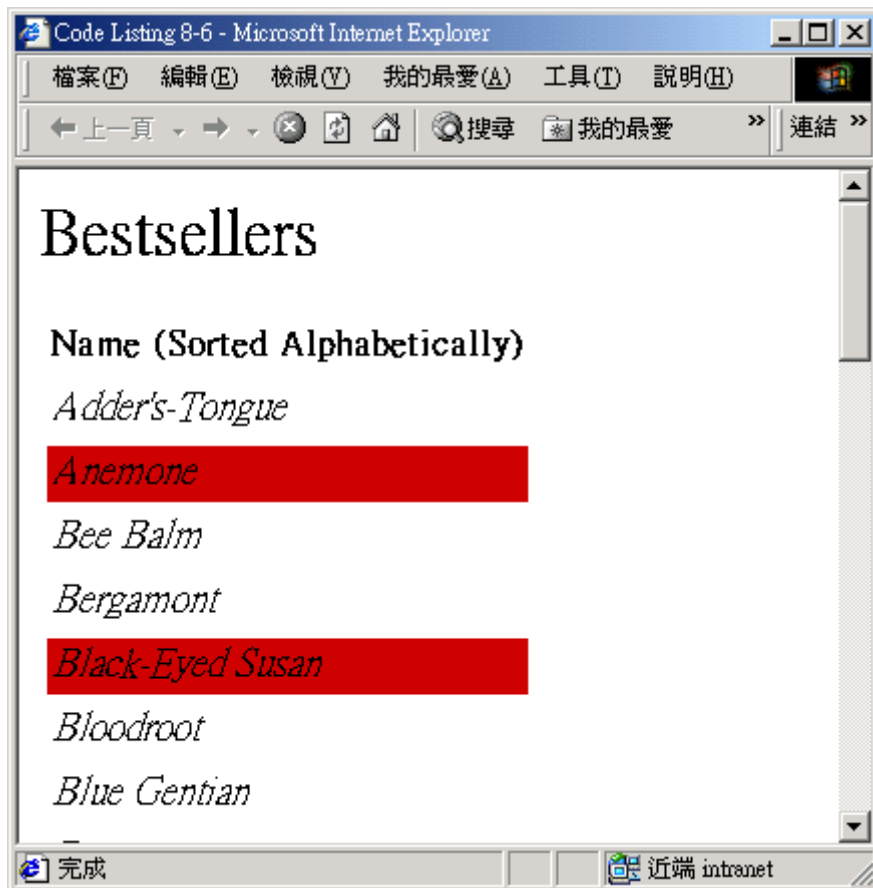
    </TD>

</xsl:template>

</xsl:stylesheet>
```

文字码 8-15

程序 8-15 显示结果如下图所示:



在 XSL 中使用 Script

就像其它的 XML 技术, XSL 支持 Script 语言的使用, 来允许文件作者可以运用既有的功能。XSL

没有限定使用任何特定的 Script 语言, 它原本就是设计来使用 Script 语言的, 也就是说使用者

只要指定 Script 语言的名称给 language 属性, 类似在 HTML 中指定使用的 Script 语言一样。

XSL 中的几个元素支持 language 属性, 包括 xsl:stylesheet、xsl:script、xsl:template 及 xsl:eval,

假如没有指定值给 language 属性, 该属性的默认值是 JavaScript。

文字码 8-16 (在随书光盘中的 Chap 08\Lst8_16.xsl) 包含了一个 Script 函数, 用来产生 1 到 10

的随机数, 该函数在 xsl:eval 元素的样板中被呼叫, 而输出的结果如右图所示:



图 8-12: 在 XSL 中使用 script。

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

<xsl:stylesheet xmlns:xsl="uri:xsl">

  <xsl:template match="/">

    <HTML>

      <BODY>

        <H1>Inventory</H1>
```

```

<TABLE CELLSPACING="4" CELLPADDING="2">

  <TR STYLE="font-weight:bold; font-size:18">

    <TD>Name, Quantity</TD>

  </TR>

  <xsl:for-each select="CATALOG/PLANT">

    <TR>

      <xsl:apply-templates/>

    </TR>

  </xsl:for-each>

</TABLE>

</BODY>

</HTML>

</xsl:template>

<xsl:script language="JavaScript">

  function numCalc ()

  {

    rndNum = (Math.round (Math.random () * 9) + 1) ;

    return rndNum;

  }

```

```
</xsl:script>

<xsl:template match="NAME">

  <TD STYLE="font-style:italic; font-size:20">

    <xsl:value-of select="COMMON"/>,

    quantity:

    <xsl:eval language="JavaScript">

      numCalc ( ) ;

    </xsl:eval>

  </TD>

</xsl:template>

</xsl:stylesheet>
```

文字码 8-16

Note

Script 语言一般允许全域变量（global variable），Script 可以透过全域变量更改这些变量的值或寻址对象。由于 XSL 使用样式表与提供数据的本能，若全域变量的使用与其它架构在使用上会产生副作用的话，XSL 将会限制它们的使用。

在 **xsl:eval** 元素中也可以直接包含 **Script** 文字码，相对于经由函数呼叫存取。例如，下列样板就和我们之前使用的一样。

```
<xsl:template match="NAME">

  <TD STYLE="font-style:italic; font-size:20">

    <xsl:value-of select="COMMON"/>,

    quantity:

    <xsl:eval language="JavaScript">

      rndNum= (Math.round (Math.random () * 9) + 1) ;

    </xsl:eval>

  </TD>

</xsl:template>
```

Script 文字码可以在其它 **Web** 环境中使用，比如 **HTML**。但既然 **XSL** 本身就是一种语言，您也可以选择一种较熟悉的语言来工作。

XSL 元素

本章中，我们已经使用了好几个 **XSL** 元素，**XSL** 元素就像是命令，告诉 **XSL** 处理器如何处理数据。底下完整列出目前所支持的 **XSL** 元素清单，并描述它们的功能。

XSL 元素	说明
xsl:apply-templates	以指定的 Pattern 为基础,指引 XSL 处理器去寻找正确的样板来套用。
xsl:attribute	产生一个属性节点,并将它套用到输出元素。
xsl:cdata	在输出中输出一个 CDATA 区段。
xsl:choose	允许条件式测试。此元素与 xsl:otherwise 和 xsl:when 元素一起使用。
xsl:comment	在输出结构中建立一个批注。
xsl:copy	从来源复制一个目的节点,以便包含到输出中。
xsl:define-template-set	在指定层级的范围定义一组样板。
xsl:element	在输出中产生一个指定名称的元素。
xsl:entity-ref	在输出中产生一个指定名称的实体参照。
xsl:eval	评估一个字符串的文字,通常是 Script 文字码。
xsl:for-each	套用相同的样板到多重文件节点中。
xsl:if	允许在样板中条件式测试。
xsl:node-name	插入目前节点的名称到输出以作为文字字符串。
xsl:otherwise	为条件式测试而提供的,这个元素必须与 xsl:choose 和 xsl:when 元素一起使用。
xsl:pi	在输出中产生一个处理指令。

xsl:script	定义全域变量宣告和函数。
xsl:stylesheet	定义一组样板套用到来源文件树中，以便产生输出文件。
xsl:template	以指定的 Pattern 为基础，为输出定义一个样板。
xsl:value-of	评估在 select 属性中指定的 XSL Pattern，并以文字格式传回要插入到样板中指定节点的值。
xsl:when	为条件式测试而提供的，这个元素必须与 xsl:choose 和 xsl:otherwise 元素一起使用。

XSL 的方法

除了上述的 XSL 元素以外，XSL 也提供了一些内建（built-in）的方法，这些方法可以从 **xsl:eval**

元素来呼叫，就如同从标准的 **Script** 文字码中呼叫一样。例如，**formatIndex** 方法可以这样使用：

```
<xsl:template match="NAME">

  <TD STYLE="font-style:italic; font-size:20">

    <xsl:value-of select="COMMON"/>,

    item number:

    <xsl:eval>

      formatIndex (childNumber (this) ,"1")

    </xsl:eval>
```

```
</TD>

</xsl:template>
```

下表说明目前支持内建的方法，这些或其它方法将于第 9 章进一步说明。

XSL 方法	说明
absoluteChildNumber	传回指定节点中与所有相关的兄弟节点数目。
ancestorChildNumber	传回指定名称节点最接近的上层节点数目。
childNumber	传回相同名称的兄弟节点数目。
depth	对所指定的节点，传回文件树中的阶层深度。
elementIndexList	传回指定节点与所有父节点的子节点数目数组，其循环取决于 root 节点。
formatDate	使用指定的格式化选项来做日期的格式化。
formatIndex	提供指定的数字系统来做整数的格式化。
formatNumber	提供指定的格式来做数字的格式化。
formatTime	提供指定的格式化选项来做时间的格式化。
uniqueID	传回指定节点的单一指标。

内建样板

在 XSL 中，每个元素必须有一个样板，以便文件可以正确地处理。实际上，如果处理器试着处理一个不包含样板的元素，不仅会发生错误，而且剩下的文件处理将会停止。在我们的例子中，我们并没有为每个元素清楚地指定样板，虽然文件看起来仍旧可以被处理，但那是因为有些功能继续在背后执行，这些功能便是内建的样板。基本上，内建样板为我们的文字码建立了一层安全网，也就是它们确保在样板遗失或空着时，文件仍然可以被处理。

当处理器辨识元素没有使用样板时，它便会让该元素使用内建的预设样板，而预设样板只是简单取出问题中的元素并不为它指定格式。底下为预设的元素样板：

```
<xsl:template>

    <xsl:apply-templates/>

</xsl:template>
```

冲突处理

您可能已经注意到我们可以建立一个冲突状况，就是同样的数据却使用两个样板。XSL 有一个内建的冲突处理机制，那就是特征测试^[m1]（**specificity test**）。冲突处理所使用的特征包含测试冲突 **Pattern**，并使用较具体的 **Pattern**。例如，试着检视底下的两个样板：

```
<xsl:template match="NAME">

    <TD STYLE="font-style:italic; font-size:10">

        <xsl:value-of select="COMMON"/>
```

```
</TD>

</xsl:template>

<xsl:template match="PLANT/NAME">

  <TD style="font-weight:bold; font-style:italic; font-size:20">

    <xsl:value-of select="COMMON"/>

  </TD>

</xsl:template>
```

在本例中，第二个模板将会被使用，那是因为它在决定目的元素上比较具体，也就是第二个模板所使用的 **PLANT/NAME** 比第一个模板仅使用 **NAME** 具体多了。因为 **XSL** 处理器会自动地做这个工作，所以知道冲突处理的运作是非常有用的，它可以用来防止样式表做出超乎预期的动作。

Part 4 进阶技巧

9. 以 XSL Pattern 来表现资料

- . 什么是 XSL Patterns?
- . **XSL Patterns** 语法
- . **XSL Patterns** 的对象模型
- . **XSL Patterns** 的其它讯息

10. XML-Data

- . 结构语言 (Schema Language) 的需求
- . **XML-Data** 结构语言
- . 结构 (Schema) 实例
- . 进阶的议题
- . 这只是个开始

11. XML 的现在与未来

- . **XML** 对于资深程序设计师的进阶使用
- . **XML** 资料岛 (Data Islands)
- . 多媒体描述语言 (Multimedia Description Languages)
- . 用 XML 来向量化影像 (Vector Images with XML)
- . 文件对象模型 (Document Object Model)
- . 文件内容描述 (Document Content Description)
- . 跨平台的 XML
- . 结论

9. 以 XSL Pattern 来表现资料

在应用程序中使用 XML 作为数据来源的价值，完全视应用程序的能力，以及使用者存取数据而定。目前为止，我们已经看过三种在 XML 文件中存取数据的方法。第一种是浏览 XML 的文件树，并在我们浏览所寻找的数据时取得数据的信息。这个方法普通且直接了当，但若要求相当复杂的程序代码，或对于较复杂的文件而言，可能会降低执行效能。

第二个方法是使用 XML 的数据来源对象（Data Source Object, DSO），将 XML 数据放到表格中。这个方法提供较多的弹性，它可以是一种全有或全无（all-or-nothing）的方法，并可以包含比预期还多的数据，同样地，DSO 在格式化数据时也无法提供高度的弹性。

我们也看过使用 XSL 将规则应用到文件上，藉以描述资料该如何呈现，既然文件可以用 XSL 来筛选与重新排列，我们只要稍微描述 XSL 如何被用在一份或多份文件中指出特定资料的外观即可。

但这三种方法都有一个缺点，因为它们都没有能力使用特定的基准来表现数据，比如像是结构化查询语言（Structured Query Language, SQL）中用来表现数据的方式。XSL Patterns（一种 XSL 的延伸）以特定的语法来解决这些问题，就是在 XML 文件中需要表现或筛选的数据时的语法。本章我们将学习到什么是 XSL Patterns，并告诉您为什么要建立 XSL Patterns，以及了解 XSL Patterns 基本的语法和架构。我们也会以范例说明如何在 XML 文件中使用 XSL Patterns，而最后将介绍这些语法比较特殊的地方。

Note

XSL Patterns 为可延伸查询语言（Extensible Query Language, XQL）的一个版本。本书撰写时，XQL 还在 W3C 的计划阶段，而 XSL Patterns 是 XQL 运用在微软 IE5 的方式。因为如此，XQL 最后的规格可能会有些地方和本章所描述的 XSL Patterns 版本有些不同。

什么是 XSL Patterns?

因为 XML 是和数据相关的，而数据只有在可以取得时才有用，XSL Patterns 被建立来使 XML 文件数据更容易取得。XSL Patterns 是一种为一般性应用而设计的语言，所以适用于很多不同的应用程序，而且可用来解决各种问题。例如：

- 在 XML 文件中执行查询。
- 在集合文件中执行查询。

- 在文件中处理数据片段。
- 在 XSL 样式表中执行查询。

XSL Patterns 的目标

为了解决我们上面所提到的问题，XSL Patterns 的制定者有一个相当长的目标清单，因而驱策了语言的发展。虽然此处并未列出所有的目标，但一些比较需要注意的目标将会在下面介绍，以便让您知道这个语言发展的过程：

- XSL Patterns 应该易于解析。
- XSL Patterns 应该能以字符串表示，而字符串原本就适用于 URL。
- XSL Patterns 应该能指定存在于 XML 文件中的任何路径，以及该路径中节点的任一组条件。
- XSL Patterns 应该能在 XML 文件中确认节点是否为独一无二的。

- **XSL Patterns** 的查询是陈述的而不是程序的，它们指出该找什么，而不是如何找。

（这一点是非常重要的，因为最佳化的查询必须能够自由使用索引，或使用其它架构有效率地找到结果。）

- **XSL Patterns** 的查询语法不是语言相依的，而是独立于任何程序语言的，而且并不表示在查询引擎中使用一种特别的实作语言。

- **XSL Patterns** 的查询条件可以在文件的任何层级被运算，而且并不预期可以从文件的 **root** 开始。（因为透过需求的方向来查询会使得查询没有效率。）

- **XSL Patterns** 的查询条件可以是文件的任何部分，也可以组合查询条件来标示阶层或参照。

- **XSL Patterns** 查询会在文件中传回不含重复节点排序的结果。

其中一个值得特别注意的目标是：**XSL Patterns** 应该可以用字符串表示，而字符串本来就适用于 **URL**。因此这个目标在建立 **XSL Patterns** 语言的实际语法时是一个非常重要的因素。您即将看到，一个查询看起来就像是我们要指出一个档案的路径一样，事实上，查询应该在文件中指定

一个路径。因为 **XSL Patterns** 表达的方式是照着 **URL** 语法的模式，所以对于使用过档案路径或 **WWW** 网址的使用者而言，应该是为大家所熟悉的。

是什么，而非如何

就像 **XSL** 一样，**XSL Patterns** 是一种描述的而非程序的语言；也就是说，**XSL Patterns** 的查询指出在 **XML** 文件中要找什么，而不是如何去找，因此提供了应用程序更大的弹性，以决定较具效率的方法来寻找数据。

Note

您可能会注意到 **XSL Patterns** 描述的本质与 **XML** 本身的原理是相同的，**XML** 也是一种描述的语言，它指出该数据为何，而不是指出在文件中该具何种外观。**XSL Patterns** 就像 **XML** 一样，是一种一般性的语言，可以在不同的方法中使用，视应用的需要而定。

让我们看看一个 **XSL Patterns** 查询的简短例子。假如，我们想要寻找所有在 **Book** 元素中的 **Chapter** 元素，**XSL Patterns** 的查询应该看起来像这样：

```
BOOK/CHAPTER
```

这就是这个查询的全部！请注意该查询的格式，除了非常容易外，它看起来就像一个档案路径或网络地址，而地址元素以一个斜线「/」分隔开来。您可能也注意到该查询简单的指出要寻找什么，而并没有指出要如何找到资料。事实上，它是由处理的应用程序而决定，会以查询的参数为基础指定最好的方法来寻找数据。

传回什么？

下一个您对此查询会问的逻辑问题可能是：我会得到什么样的结果？**XSL Patterns** 的查询结果将是 XML 文件的一组节点，或者是查询执行后的文件，而不仅仅是传回 XML 文件的节点而已。但是，这些节点之间的关系将完整的存在。这一点非常重要，您必须知道：取代传回未经处理的数据，**XSL Patterns** 传回整个 XML 树状数据，以便您与其它的 XML 数据处理。

使用 **XSL Patterns** 的好处

在进一步检视 **XSL Patterns** 如何运作之前，让我们实际看看为何 **XSL Patterns** 对于取得 XML 数据是一个较好的解决方式。假如您一步步地跟着本书的范例，您可能会注意到我们已经使用过小型的 XML 文件了，并且使用它们示范了特定的功能。当然，那是由于书本格式的大小及其它方面的限制，但在真实的世界里，您的文件不可能总是如此地小。让我们看看一些之前使用的范例文字码，来了解所学与现实有时是会冲突的，也顺便瞧瞧 **XSL Patterns** 如何帮助我们解决这

些问题。请重新叫出第 4 章的范例，就是我们使用一个 HTML 档案来显示一个 XML 电子邮件档

案的范例，这两个档案的文字码如下：

文字码 9-1 为 Lst4_1.xml 档案（在随书光盘中的 Chap09\Lst9_1.xml）：

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

<!DOCTYPE EMAIL [

    <!ELEMENT EMAIL (TO, FROM, CC, SUBJECT, BODY)>

    <!ELEMENT TO (#PCDATA)>

    <!ELEMENT FROM (#PCDATA)>

    <!ELEMENT CC (#PCDATA)>

    <!ELEMENT SUBJECT (#PCDATA)>

    <!ELEMENT BODY (#PCDATA)>

]>

<EMAIL>

    <TO>Jodie@msn.com</TO>

    <FROM>Bill@msn.com</FROM>

    <CC>Philip@msn.com</CC>

    <SUBJECT>My First DTD</SUBJECT>

    <BODY>Hello, World!</BODY>
```

</EMAIL>

文字码 9-1

底下则是 **Lst4_1.htm** 档案:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

    <HEAD>

        <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

            loadDoc();

        </SCRIPT>

        <SCRIPT LANGUAGE="JavaScript">

            var xmlDoc = new ActiveXObject("microsoft.xmlDOM");

            xmlDoc.load("Lst4_1.xml");

            function loadDoc()

            {

                if (xmlDoc.readyState == "4")

                    start();
```



```
else

    window.setTimeout("loadDoc()", 4000);

}

function start()

{

    var rootElem = xmlDoc.documentElement;

    var rootLength = rootElem.childNodes.length;

    for (cl=0; cl<rootLength; cl++)

    {

        currNode = rootElem.childNodes.item(cl);

        switch (currNode.nodeName)

        {

            case "T0":

                todata.innerText=currNode.text;

                break;

            case "FROM":

                fromdata.innerText=currNode.text;

                break;

            case "CC":
```

```
        ccdata.innerText=currNode.text;

        break;

    case "SUBJECT":

        subjectdata.innerText=currNode.text;

        break;

    case "BODY":

        bodydata.innerText=currNode.text;

        break;

    }

}

}

</SCRIPT>

<TITLE>Untitled</TITLE>

</HEAD>

<BODY>

    <DIV ID="to" STYLE="font-weight:bold;font-size:16">

        To:

        <SPAN ID="todata" STYLE="font-weight:normal"></SPAN>

    </DIV>

    <BR>
```

<DIV ID="from" STYLE="font-weight:bold;font-size:16">

From:

</DIV>

<DIV ID="cc" STYLE="font-weight:bold;font-size:16">

Cc:

</DIV>

<DIV ID="subject" STYLE="font-weight:bold;font-size:16">

Subject:

</DIV>

<HR>


```
<P>  
  
</BODY>  
  
</HTML>
```

要让此范例可以运作，我们必须先了解这个 **XML** 档案，并假设该放些什么数据。现在就让我们更进一步瞧瞧那些文字码，并检视这些假设。

Start 这个函式从 **XML** 文件的节点中收集数据，并且将这些数据放到 **HTML** 文件中指定的元素。

为了完成这个动作，该函式一个接一个拜访树状中的每一个节点，并确认所拜访的节点是否含有需要的数据。基本上，该文字码走过文件树状结构中的每个节点，并询问这些节点：您有我想要的的数据吗？当节点回答：是，便撷取这些数据并将它储存到 **HTML** 元素中。要让文字码可以拜访整个文件树状结构，并撷取我们想要的的数据，我们必须知道下面几件事情：

- 每个节点所包含的数据型式。
- 那些节点的名称，这样才能直接存取。
- 那些数据是不是真的能应用（或不能应用）于我们的应用程序。

本分文件的文件树状只包含了一个层级，虽然这个方法好像只适用于范例这样的小型文件，大型文件则必须使用大量复杂的文字码来处理。

假设我们只告诉 XML 处理器我们需要些什么，它就会传回适当的数据吗？虽然我们仍需要了解该文件及文件架构的相关信息，但我们并不需要浏览整个树状结构来寻找数据。使用 XSL Patterns 便可以做到，而这个能力也是 XSL Patterns 其中一个主要的优点。为了显示 XSL Patterns 如何戏剧化地改变整个程序，让我们重写使用 XSL Patterns 查询来从树状结构中取得数据的文字码，而 XML 档案则不需做改变。新的 HTML 网页将如文字码 9-2（在随书光盘中的 Chap09\Lst9_2.htm）显示。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

  <HEAD>

    <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

      loadDoc();

    </SCRIPT>

    <SCRIPT LANGUAGE="JavaScript">

      var xmlDoc = new ActiveXObject("microsoft.xmlDOM");

      xmlDoc.load("Lst9_1.xml");
```

```
function loadDoc()

{

    if (xmlDoc.readyState == "4")

        start();

    else

        window.setTimeout("loadDoc()", 4000);

}
```

```
function start()

{

    todata.innerText =

        xmlDoc.selectSingleNode("EMAIL/TO").text;

    fromdata.innerText =

        xmlDoc.selectSingleNode("EMAIL/FROM").text;

    ccdata.innerText =

        xmlDoc.selectSingleNode("EMAIL/CC").text;

    subjectdata.innerText =

        xmlDoc.selectSingleNode("EMAIL/SUBJECT").text;

    bodydata.innerText =
```

```
        xmlDoc.selectSingleNode("EMAIL/BODY").text;

    }

</SCRIPT>

<TITLE>The First XSL Patterns Example</TITLE>

</HEAD>

<BODY>

    <DIV ID="to" STYLE="font-weight:bold;font-size:16">

        To:

        <SPAN ID="todata" STYLE="font-weight:normal"></SPAN>

    </DIV>

    <BR>

    <DIV ID="from" STYLE="font-weight:bold;font-size:16">

        From:

        <SPAN ID="fromdata" STYLE="font-weight:normal"></SPAN>

    </DIV>

    <BR>

    <DIV ID="cc" STYLE="font-weight:bold;font-size:16">

        Cc:

        <SPAN ID="ccdata" STYLE="font-weight:normal"></SPAN>
```

```
</DIV>

<BR>

<DIV ID="from" STYLE="font-weight:bold;font-size:16">

    Subject:

    <SPAN ID="subjectdata" STYLE="font-weight:normal"></SPAN>

</DIV>

<BR>

<HR>

<SPAN ID="bodydata" STYLE="font-weight:normal"></SPAN><P>

</BODY>

</HTML>
```

文字码 9-2

上面修改过的文字码中，**start** 函式在文件架构中指定节点路径的方式来处理想要的节点，也就是说：找到符合型态的节点，并从中取得数据。亦即它并不需要拜访整个文件树状结构来寻找想要的节点。这个简单的范例表现一些 **XSL Patterns** 的能力，现在就让我们更进一步的了解这个语言，以及它如何被使用来建立更复杂的查询。

XSL Patterns 语法

上面最后一个范例中，**XSL Patterns** 在一份 XML 文件中以指定的型态比对数据，然后传回与 **XSL Patterns** 对象模型（**XSL Patterns object model**，稍后将会提到）符合的样板结果。在文字码 9-2 中，每一笔结果都是实际的资料，但您将会看到，传回的结果不只是单一节点的数据。事实上，传回的结果可以是一个包含数据的复杂节点集合。

提供 Context

使用 **XSL Patterns** 查询时，您必须提供它将运作的目录（**context**），目录就是查询的节点或节点的排列。请记得一份 XML 文件的树状架构，而 **XSL Patterns** 可以在 **root** 层级或任何树状结构的分支层级运作。很明显地，查询所使用的目录可以大大地改变结果。

让我们回去看看文字码 9-2 中所使用的一个查询型态。

```
EMAIL/TO
```

此查询指明了处理器应该在名为 **EMAIL** 的节点下（本例中为 **root** 节点），寻找名为 **TO** 的节点，XML 文件 **Lst9_1.xml** 只包含了一个名为 **TO** 的元素，所以结果只传回一个节点。

但是,假如该文件中有很多节点拥有相同的名称呢? 让我们使用一个较复杂的文件来建立另一个范例。文字码 9-3 是一份 **Wildflower** 植物目录, 看起来像是前面章节所使用的范例, 这份文件包含了 很多拥有相同名称的节点, 所以我们可以看看, 在这个情况下查询会有什么反应。

```
<CATALOG>
```

```
  <PLANT>
```

```
    <COMMON>Bloodroot</COMMON>
```

```
    <BOTANICAL>Sanguinaria canadensis</BOTANICAL>
```

```
    <ZONE>4</ZONE>
```

```
    <LIGHT>Mostly Shady</LIGHT>
```

```
    <PRICE>$7.05</PRICE>
```

```
    <AVAILABILITY USONLY=1Vtrue1I>02/01/99</AVAILABILITY>
```

```
  </PLANT>
```

```
  <PLANT>
```

```
    <COMMON>Columbine</COMMON>
```

```
    <BOTANICAL>Aquilegia canadensis</BOTANICAL>
```

```
    <LIGHT>Mostly Shady</LIGHT>
```

```
    <PRICE>$3.20</PRICE>
```

```
    <AVAILABILITY>04/08/99</AVAILABILITY>
```

</PLANT>

<PLANT>

<COMMON>Marsh Marigold</COMMON>

<BOTANICAL>Caltha palustris</BOTANICAL>

<ZONE>4</ZONE>

<LIGHT>Mostly Sunny</LIGHT>

<PRICE>\$2.90</PRICE>

<AVAILABILITY>01/09/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Cowslip</COMMON>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$3.83</PRICE>

<AVAILABILITY USONLY="true">05/19/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Dutchman's-Breeches</COMMON>

<BOTANICAL>Diecentra cucullaria</BOTANICAL>

<ZONE>3</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$7. 63</PRICE>

<AVAILABILITY>01/09/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Ginger, Wild</COMMON>

<BOTANICAL>Asarum canadense</BOTANICAL>

<ZONE>3</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$8. 90</PRICE>

<AVAILABILITY>03/01/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Hepatica</COMMON>

<BOTANICAL>Hepatica americana</BOTANICAL>

<ZONE>4</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$7.75</PRICE>

<AVAILABILITY>12/16/98</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Liverleaf</COMMON>

<BOTANICAL>Hepatica americana</BOTANICAL>

<ZONE>4</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$3.64</PRICE>

<AVAILABILITY>12/29/98</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Jack-In-The-Pulpit</COMMON>

<BOTANICAL>Arisaema triphyllum</BOTANICAL>

<ZONE>4</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$2.87</PRICE>

<AVAILABILITY>02/12/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Mayapple</COMMON>

<BOTANICAL>Podophyllum peltatum</BOTANICAL>

<ZONE>3</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$3.99</PRICE>

<AVAILABILITY>02/04/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Phlox, Woodland</COMMON>

<BOTANICAL>Phlox divaricata</BOTANICAL>

<ZONE>3</ZONE>

<LIGHT>Sun or Shade</LIGHT>

<PRICE>\$8.82</PRICE>

<AVAILABILITY>05/21/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Phlox, Blue</COMMON>

<BOTANICAL>Phlox divaricata</BOTANICAL>

<ZONE>3</ZONE>

<LIGHT>Sun or Shade</LIGHT>

<PRICE>\$9.65</PRICE>

<AVAILABILITY>02/11/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Spring-Beauty</COMMON>

<BOTANICAL>Claytonia Virginica</BOTANICAL>

<ZONE>7</ZONE>

<LIGHT>Mostly Shady</LIGHT>

<PRICE>\$3.44</PRICE>

<AVAILABILITY>03/11/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Trillium</COMMON>

<BOTANICAL>Trillium grandiflorum</BOTANICAL>

<ZONE>5</ZONE>

<LIGHT>Sun or Shade</LIGHT>

<PRICE>\$8. 97</PRICE>

<AVAILABILITY>05/22/99</AVAILABILITY>

</PLANT>

<PLANT>

<COMMON>Wake Robin</COMMON>

<BOTANICAL>Trillium grandiflorum</BOTANICAL>

<ZONE>5</ZONE>

<LIGHT>Sun or Shade</LIGHT>

<PRICE>\$6. 76</PRICE>

<AVAILABILITY>03/14/99</AVAILABILITY>

</PLANT>

</CATALOG>

Note

为了节省空间，我们已经简略了文字码 9-3，如果要取得整份文件，请参考随书光盘中的

Chap09\Lst9_3.xml。

接下来，我们需要一份 HTML 网页来显示结果，文字码 9-4（在随书光盘中的 Chap09\

Lst9_4.htm）加载 XML 文件，并在资料上执行一个查询，然后显示结果。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

  <HEAD>

    <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

      loadDoc();

    </SCRIPT>

    <SCRIPT LANGUAGE="JavaScript" SRC="Lst9_5.js">

    </SCRIPT>

    <SCRIPT LANGUAGE="JavaScript">
```

```
var rootElem;

var xmlDoc = new ActiveXObject("microsoft.xmlDOM");

xmlDoc.load("Lst9_3.xml");


function loadDoc()

{

    if (xmlDoc.readyState == 4)

        start();

    else

        window.setTimeout("loadDoc()", 250);

}


function start()

{

    var qry = xmlDoc.selectNodes("CATALOG");

    var bt;

    var hTank = "";

    for (bt = qry.nextNode(); bt != null; bt = qry.nextNode())

    {
```

```

        hTank += buildTree(bt);

    }

    document.body.innerHTML = hTank

}

</SCRIPT>

<TITLE>Code Listing 9-4</TITLE>

</HEAD>

<BODY>

</BODY>

</HTML>

```

文字码 9-4

文字码 9-4 的输出结果显示在图 9-1。XSL Patterns 对象模型并没有传回整列的 XML 文字码，文字码 9-4 是使用一个被连结的 Script 来从传回的数据建立 XML 文字码。这个 Script 如文字码 9-5 所示，您也可以在随书光盘中的 Chap09\Lst9_5.js 找到。

```

function buildTree(qNode, i)

{

    var output = "<DL CLASS=xml><DD>";

    if (qNode != null)

```

```
{

type = qNode.nodeType;

if (type == 6)

{

    output += "<SPAN>" + qNode.nodeValue + "</SPAN></DD></DL>";

    return output;

}

output += "<SPAN CLASS=tag>< " + qNode.nodeName + "</SPAN>";

var hasChildren = qNode.childNodes.length > 0;

if (!hasChildren)

    output += "<SPAN CLASS=tag>/></SPAN>";

else

    output += "<SPAN CLASS=tag>></SPAN>";

if (hasChildren > 0)

{

    if (isMixed(qNode) > 0)

    {

        output += qNode.text;
```

```

    }

else

    {

        var child;

        var children = qNode.childNodes;

        for (child = children.firstChild();

            child != null;

            child = children.firstChild())

        {

            output += "\n";

            output += buildTree(child, i + 1);

        }

    }

    output += "<SPAN CLASS=tag></ " + qNode.nodeName +

        "></SPAN>\n";

}

}

output += "</DD></DL>"

return output;

```

```

    }

function isMixed(qNode, num)

{

    var child;

    var children = qNode.childNodes;

    for (child = children.firstChild();

        child != null;

        child = children.nextSibling())

    {

        var type = child.nodeType;

        if (type == 3 || type == 4 || type == 5)

        {

            return 1;

        }

    }

    return 0;

}

```

文字码 9-4 中最主要的一行是 **start** 函式一开始的查询：

```
var qry=xmlDoc.selectNodes ("CATALOG") ;
```

为了示范这个查询是如何运作的，我们将改变查询的目录，然后观察结果如何改变。请注意：文

字码 9-4 的结果包含了整个 XML 文件，因为该查询的目录为 **CATALOG**，也就是根（**root**）节

点。让我们更改该查询，搜寻到树状结构中深度到 **PLANT** 这个层级：

```
var qry=xmlDoc.selectNodes ("CATALOG/PLANT") ;
```

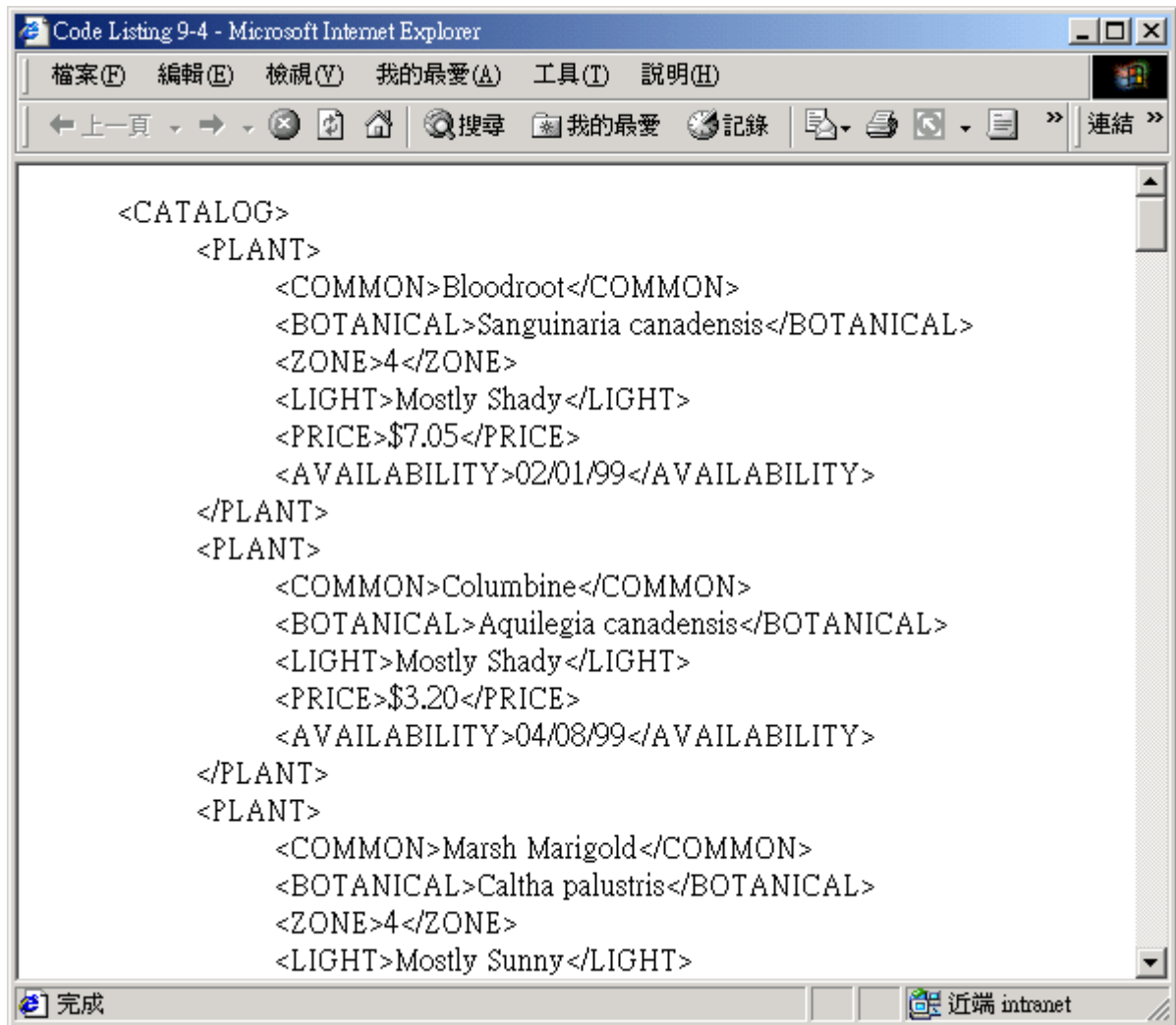


图 9-1: XSL Patterns 查询的结果。

此查询现在表示：当 PLANT 节点包含在 CATALOG 节点中时，给我所有 PLANT 节点包含的数据。结果显示在图 9-2。

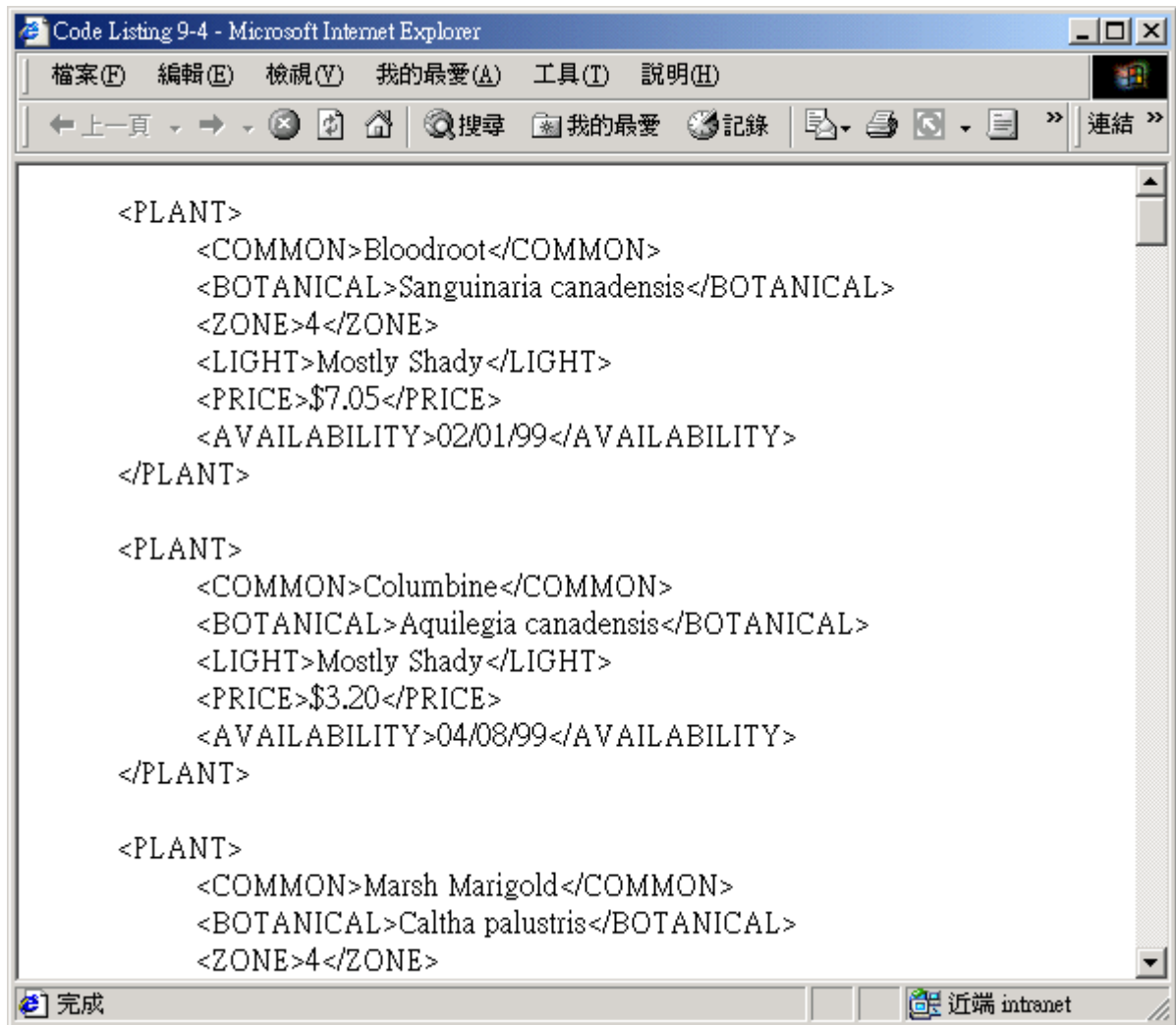


图 9-2: XSL Patterns 查询有限制目录 (context) 的结果。

现在让我们看看再深一层的查询及其结果。虽然这个层级有很多不同类型的节点，我们只要随便选择一个就行了。

```
var qry=xmlDoc.selectNodes ("CATALOG/PLANT/LIGHT");
```

这个查询只要求符合指定型态（**pattern**）的节点，其结果显示于图 9-3，是名为 CATALOG 节

点中 PLANT 节点下所有名为 LIGHT 节点的集合。

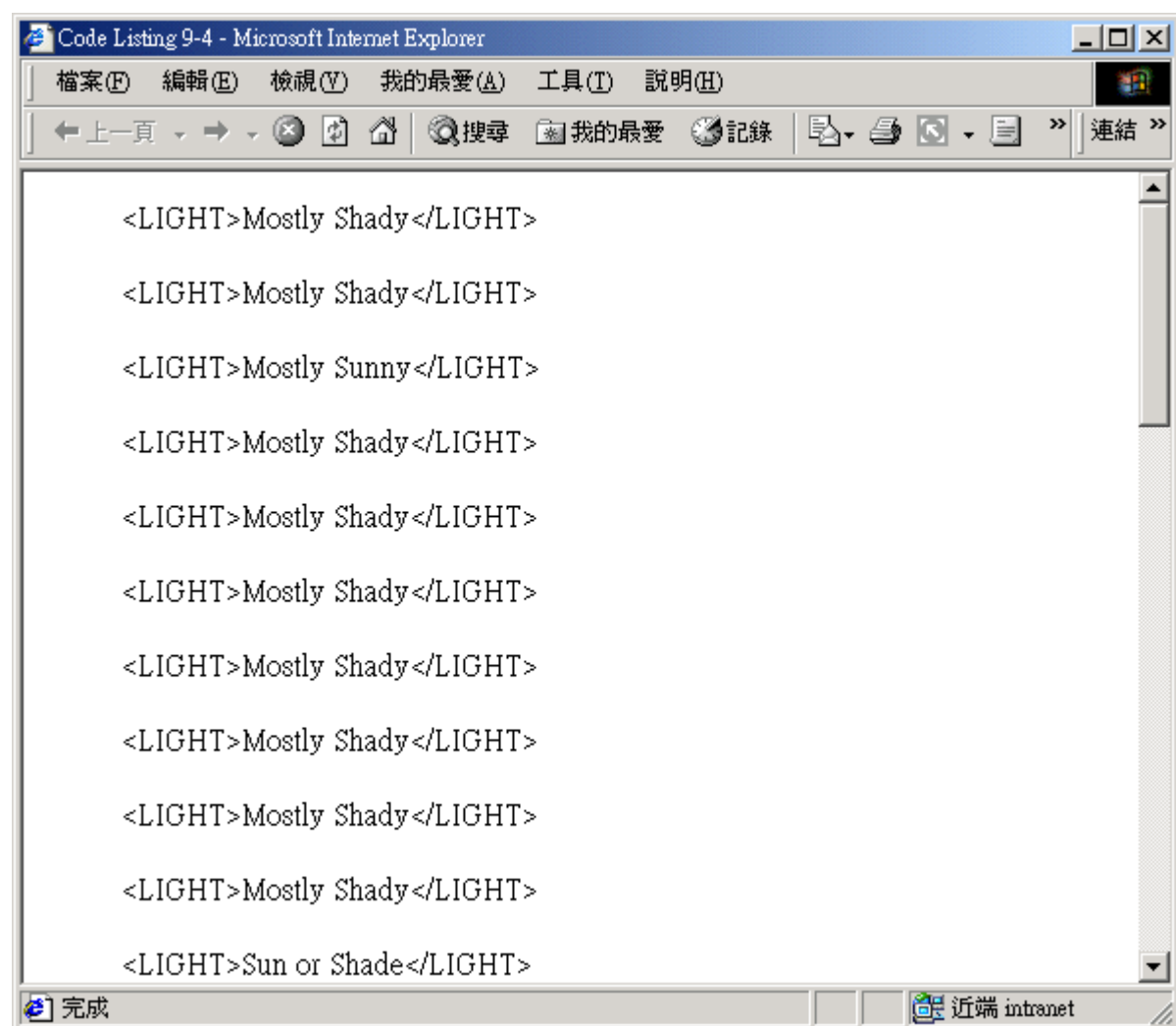


图 9-3：更严格限定目录的 XSL Patterns 查询结果

集合（Collections）

您可能已经注意到，该查询的结果是一组符合查询型态的节点，这就叫做集合（**collection**），而一个集合（**collection**）可以透过给予一个元素名称来参照，所以上述最后一个例子中，该查询传回一个名为 **LIGHT** 的集合，而 **LIGHT** 集合为 **PLANT** 集合的一部分，以此类推。您可以开始试着改变查询型态，来看看查询结果的改变。

到目前为止，我们只有看到最简单的查询类型，也就是节点路径。但是 **XSL Patterns** 提供了很多不同的目录操作数，可以让我们建立更进阶的查询。

选择（Selection）操作数

您已经看过基本选择（**Selection**）操作数在上述例子中的运作，该前置的斜线「/」指出，在型态（**pattern**）中一个结点与另一个结点间的阶层关系，而结点可以被确认是一个父节点或是子节点，完全取决于它们出现在斜线的哪一方。比方说，在型态 **CATALOG/PLANT** 中，斜线左边的项目 **CATALOG** 为父节点，而斜线右边的项目 **PLANT** 为 **CATALOG** 的子节点。所以使用此型态的查询，若 **PLANT** 节点与 **CATALOG** 节点关系正确的话，将传回 **PLANT** 节点。

您可以在其它方面使用这个 **Selection** 操作数，来指定查询不同的目录。比方说，我们可以在一个查询之前使用单一的前置斜线（**/**），来指定目录与文件的 **root** 层级是相关的（假如我们未加上前置斜线，则会预设所表示的是 **root** 层级，所以不一定总是需要斜线）。这个情形如下所示：

```
/CATALOG/PLANT/LIGHT
```

这告诉处理器从文件的 **root** 层级开始寻找 **CATALOG/PLANT/LIGHT** 型态，既然 **root** 层级可以不需加上前置斜线 (**/**)，同样的查询方式可能为：

```
CATALOG/PLANT/LIGHT
```

另一个选择 (**Selection**) 操作数为两条前置斜线 (**//**)，此操作数表示递归下降；也就是说，该查询要求每一个型态的实例 (**instance**)，这个型态显示在操作数的右边，并且位于显示在操作数左边型态的下面。比方说，查询 **CATALOG//LIGHT** 表示寻找所有在 **Catalog** 节点之下的 **Light** 节点，而查询 **//LIGHT** 表示寻找文件中所有的 **Light** 节点。

为了测试这一点，请改变文字码 **9-4** 中的查询成为下述的样子：

```
var qry=xmlDoc.selectNodes ("//LIGHT") ;
```

虽然搜寻型态不同，但结果（如图 **9-4**）看起来就像前面的图 **9-3**，这是因为 **LIGHT** 节点在 **XML** 文件中仅存在一个目录，在这个范例中，我们只是使用不同的方法来达成相同结果的查询。如果 **LIGHT** 节点出现在阶层的其它层级中，新的查询将会传回所有的 **LIGHT** 节点，而不只是在指定目录中的 **LIGHT** 节点。

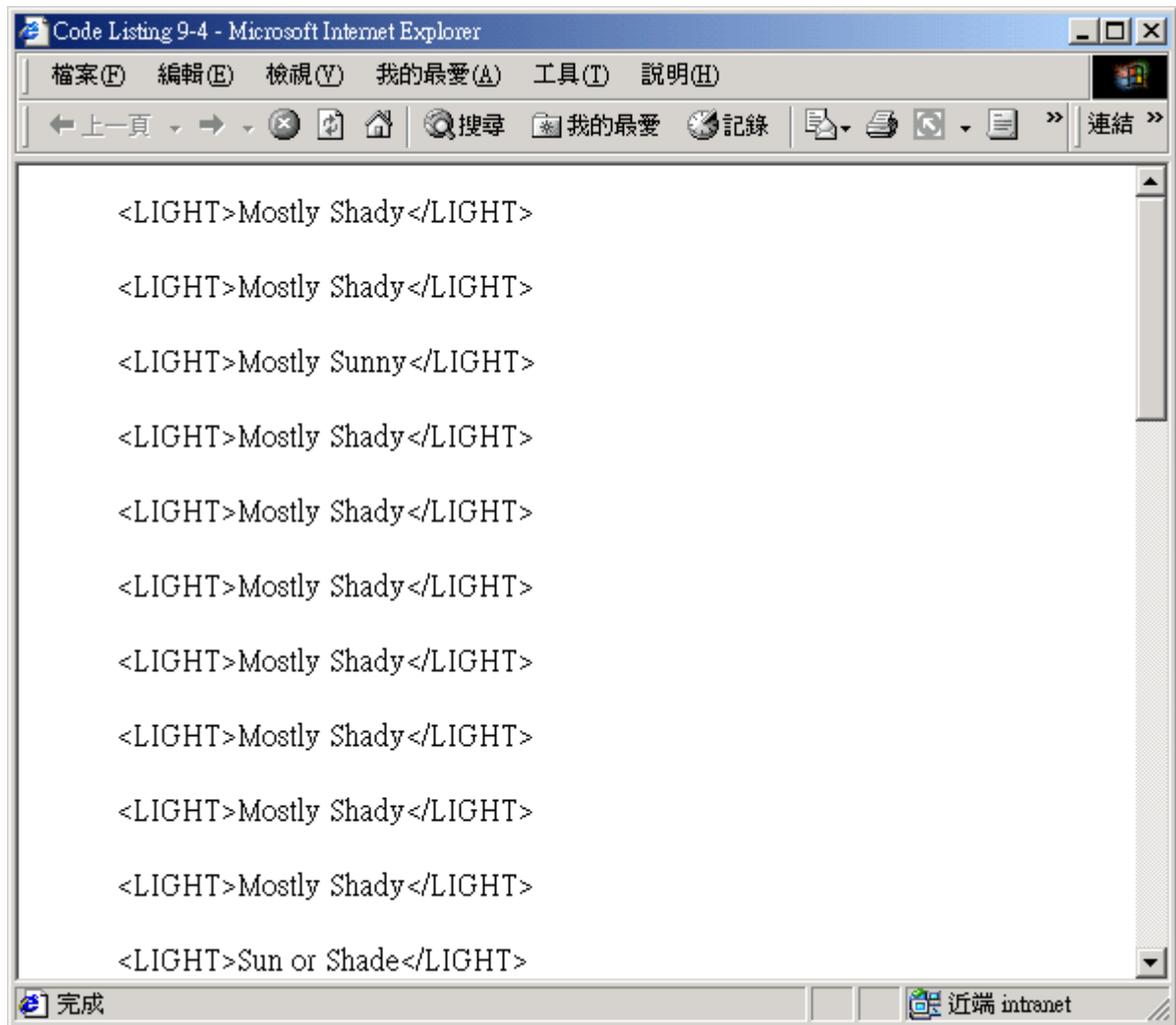


图 9-4：不同的查询可能产生相同结果。

另两个选择（**Selection**）操作数为前置点斜线（./）与前置点双斜线（.//）。前置点斜线（./）操作数表示：右方的型态应该在目前的目录中被找到。而前置点双斜线（.//）操作数表示：右方的型态应该从目前的目录中递归下降寻找。既然这两个操作数从未用过，我们将在后续的使用上多加介绍。

所有（All）操作数

所有操作数（*）在给予的目录中寻找所有的子节点，如下面的查询范例：

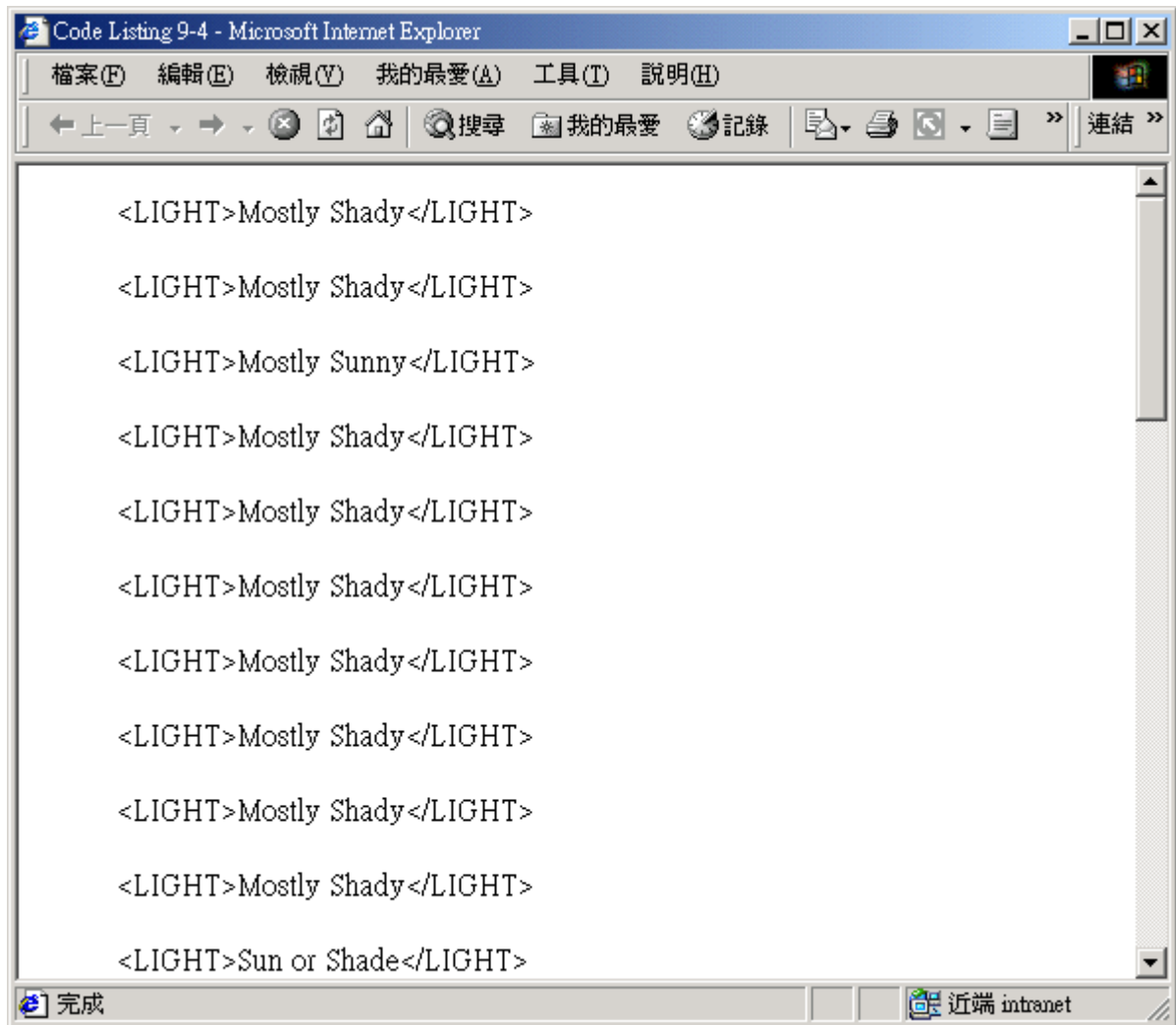
```
var qry=xmlDoc.selectNodes ("CATALOG/PLANT/*") ;
```

传回的结果如下：



而另一个例子，型态 CATALOG/*/LIGHT 寻找所有 LIGHT 节点，为 CATALOG 的孙节点

（grandchildren nodes），结果如下图所示：



属性（Attribute）操作数

A 属性操作数（@）表示指定元素的一个属性，属性操作数透过要求的属性名称前面放置@符号

来运作。例如，下列的查询寻找所有包含在 USONLY 属性的 Availability 元素：

```
var qry=xmlDoc.selectNodes ("CATALOG/PLANT/AVAILABILITY/@USONLY") ;
```

群组与优先权

有时一般查询语法不够清楚来允许型态比对，为了处理这个问题，**XSL Patterns** 允许您使用拥有内建优先权的群组与次查询操作数。群组操作数为括号()，而次查询操作数为中括号[]。

优先权的顺序从高至低如下：

1. 群组操作数：()
2. 次查询操作数：[]
3. 选择操作数：/、//、./、.//

次查询（Subqueries）

一个次查询提供一个机制来限制查询，我们可以透过插入型态到次查询操作数中，来加入一个条件判断到查询型态中。而产生的结果就是符合次查询标准的所有元素集合。比方说，查询型态 **CATALOG/PLANT[ZONE]**表示：寻找所有 **Catalog** 元素的子元素 **Plant**，而 **Plant** 元素中至少包含一个 **Zone** 元素。

Note

次查询的子句有点像是 SQL 语法的 WHERE 子句，所以使用这样的说法，则将上面的查询看成 SQL 语法，就是：寻找 ANY Catalog 元素的子元素 Plant，WHERE Plant 元素至少要存在一个 Zone 元素为 TRUE。

任何数目的次查询可以存在型态的任意层级中，而且次查询可以使用巢状结构，但是却不允许空的次查询。

数据比对

大多数功能强大的数据库程序设计语言都提供您一组数据与另一组数据比对的能力，XSL Patterns 也提供了类似的功能，允许文件作者建立复杂的查询表达式。

布尔表达式（AND/OR/NOT）

布尔表达式可以在次查询中使用，并且使用「\$操作数\$」格式，而我们可以使用下列这三个布尔操作数：

- \$and\$

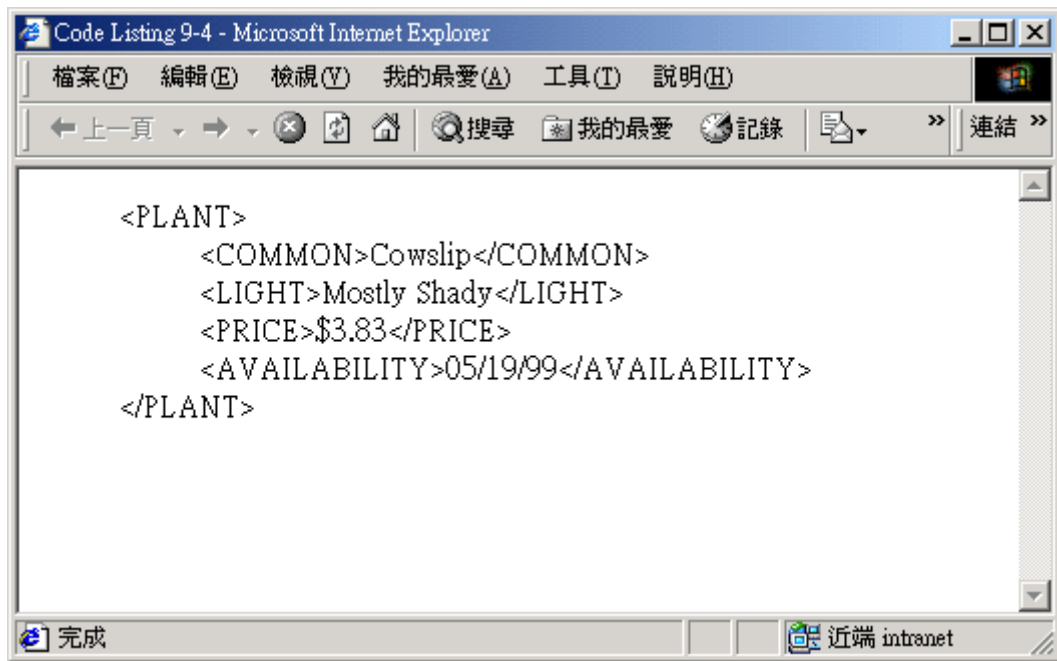
- \$or\$

- \$not\$

现在我们就使用这些操作数来建立查询，并且以群组操作数组合它们，来瞧瞧如何在 XSL 型态中建立一些相关的复杂逻辑表达式。

```
var qry=xmlDoc.selectNodes  
  
("CATALOG/PLANT[$not$ (ZONE $or$ BOTANICAL) $and$ LIGHT]");
```

这个查询表示：寻找所有 **Catalog** 元素的子元素 **Plant**，而且 **Plant** 元素中至少要包含一个 **Light** 元素，但不能包含 **Zone** 或 **Botanical** 元素。这个查询结果为 XML 文件中符合型态的单一节点，如右所示：



相等

有些人可能一直想要知道这一部分，您可能也已经注意到，截至目前为止我们都没有测试任何元素的值，而只是测试来确定元素存不存在罢了！在很多情况下，查询一个文件架构的能力远比只能搜寻文件内容来的有价值，而相等表达式允许我们以特定的值来测试元素的内容。

相等性运算有下列四种格式：

- 相等 (\$eq\$ 或 =)
- 不分大小写相等 (\$ieq\$)

- 不相等 (\$ne\$ 或 !=)
- 不分大小写不相等 (\$ine\$)

让我们看看这些表达式的运作。下面的查询寻找所有文件中包含一个 **Light** 元素的 **Plant** 元素，

而且 **Light** 元素的值为 **Mostly Shady**。

```
var qry=xmlDoc.selectNodes ("//PLANT[LIGHT='Mostly Shady']");
```

该表达式也可以写成这样：

```
var qry=xmlDoc.selectNodes ("//PLANT[LIGHT $eq$ 'Mostly Shady']");
```

Note

字符串值必须包含于单引号内。

其它比较操作数 (**ALL/ANY**)

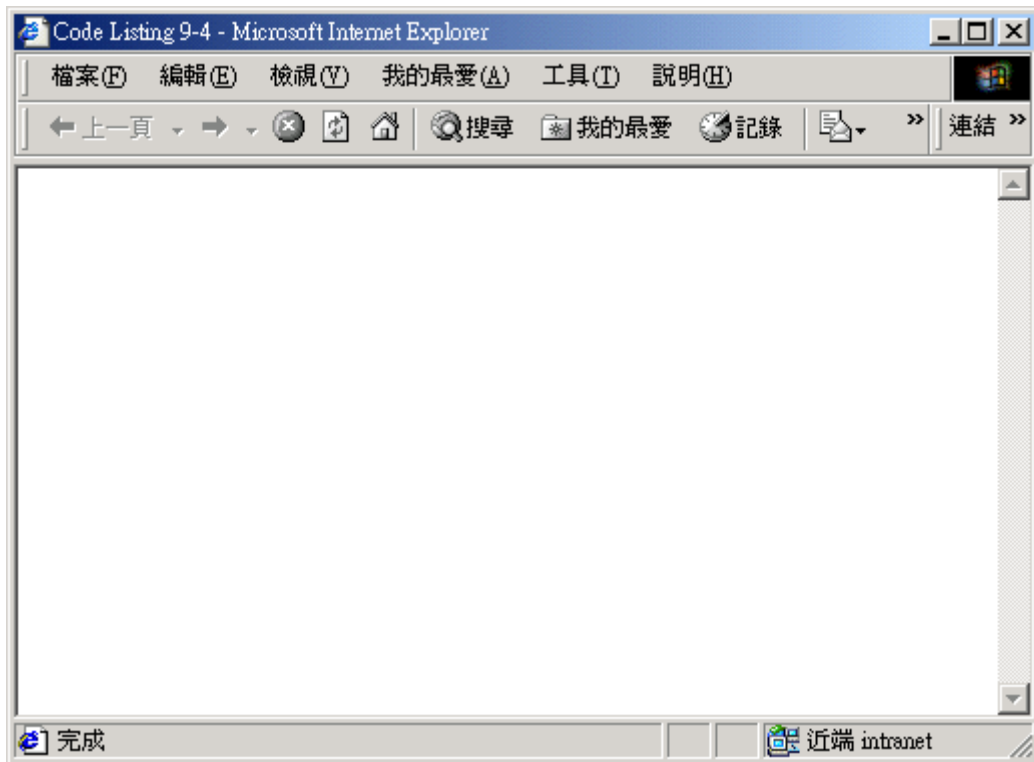
all 以及 **any** 比较操作数允许我们在一个集合中搜寻，来指定集合中的每个元素应该如何运算求值，换句话说，根据使用的操作数来搜寻参数，可以适用于集合中的所有元素或任何一个元素。

all 操作数表示将整个集合当成结果传回，只要查询状态符合集合中的所有元素即可，如果没有找到符合的数据，就会传回空集合。**All** 操作数的写法为：**\$all\$**。

下面的查询寻找 **CATALOG** 节点中，所有包含一个 **ZONE** 节点，而且 **ZONE** 节点值为 **4** 的 **PLANT** 节点。

```
var qry=xmlDoc.selectNodes  
  
（“//CATALOG[$all$ PLANT/ZONE='4']”）；
```

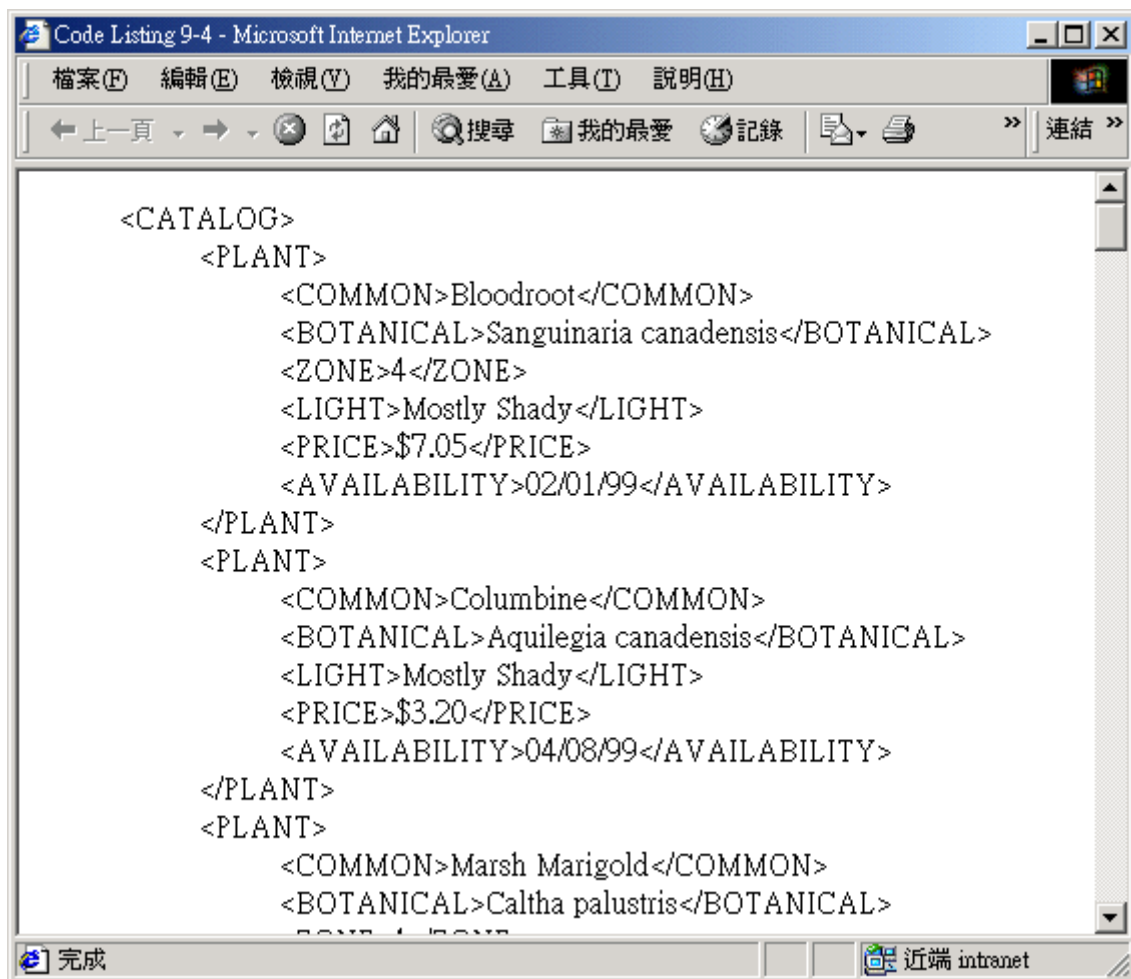
这个查询结果为一个空集合，因为并非每个 **PLANT** 节点都包含一个值为 **4** 的 **ZONE** 子节点，如下所示：



any 操作数则是用来寻找一个符合的型态，也就是找看看是否有任何元素符合该查询的条件，换句话说，只要有一个集合中的元素符合查询型态，整个集合就会被当成查询结果而被传回。**any** 操作数写法为：**\$any\$**，底下是它的用法：

```
var qry=xmlDoc.selectNodes  
  
（“//CATALOG[$any$ PLANT/ZONE='4']”）；
```

在这个范例中，如果任何的 **PLANT** 节点含有值为 **4** 的 **ZONE** 子节点，整个集合将会被当成结果传回，如下所示：



数字比较

当所有 XML 文件中的元素和属性值都是字符串型态时, XSL Patterns 支持数字的比较, 比如 2 <

45, 底下的清单列出 XSL Patterns 支持的数字操作数。

操作数	缩写	说明
\$1t\$	<	小于

\$ilt\$		区分大小写（Case-insensitive）的小于
\$le\$	< =	小于等于
\$ile\$		区分大小写的小于等于
\$gt\$	>	大于
\$igt\$		区分大小写的大于
\$ge\$	>=	大于等于
\$ige\$		区分大小写的大于等于

为了比较数值，XSL Patterns 会将文字值（text values）转换为适当的数字型态来比较，而各种比对的应用，取决于所比对值的原始型态，也就是该表达式右边的值。

比对值的型态	比对类型
string（字符串）	用于字符串与字符串间的比对。
integer（整数）	比对值将以长整数来计算。
real（实数）	比对值将以倍精度数字来计算。

使用方法（Methods）

除了提供操作数来建立表达式（**expressions**），**XSL Patterns** 也提供函式（**functions**）或方法（**methods**），用来处理集合或取得节点及元素的信息，方法的格式为 **method(arguments)**，而且它大小写的意义不同（**case-sensitive**）。下列文字码中，我们在 **context** 之后附加一个惊叹号及 **method**，如下所示：

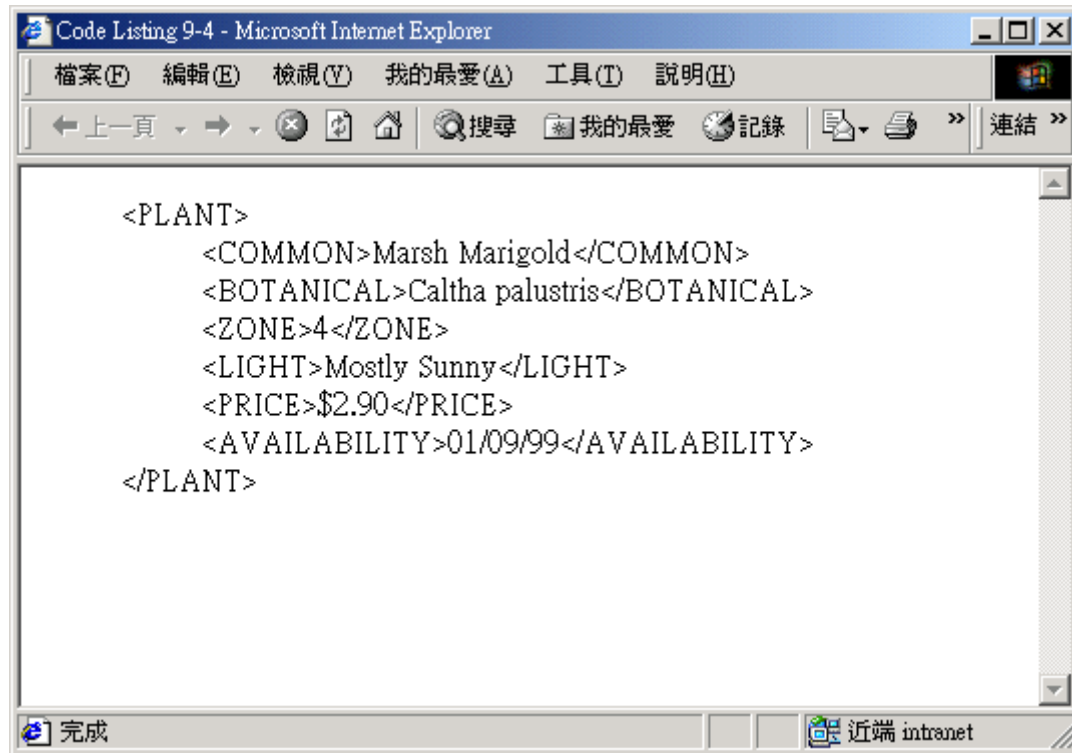
```
context!method (arguments)
```

text 方法

以下将使用 **text** 方法来传回节点的文字，而不包含文件架构。请注意，该方法传回文字给查询，然后使用这些文字来当作查询型态的一部分，而查询的结果仍是一个节点或集合。例如，下面的查询寻找每个 **PLANT** 节点下，且文字值为 **Mostly Sunny** 的 **LIGHT** 节点：

```
var qry=xmlDoc.selectNodes  
  
（“//PLANT[LIGHT!text（）='Mostly Sunny' ]”）；
```

该查询的结果如下所示，只有文件中的 **PLANT** 节点符合标准：



Note

长整数（long integer）是一个整数，支持的范围介于-2,147,483,648 到 2,147,483,647，而精度数字为一个小数数字。

NodeType 方法

NodeType 方法会传回一个数值，该数值可以指出节点的类型，下表列出每一种不同的节点类型以及对应的值。

节点型态	nodeType 传回的值
element	1
attribute	2
text	3
cdata	4
entity reference	5
entity	6
processing instruction	7
comment	8
document	9
document type	10
document fragment	11
notation	12

下面的范例传回一个节点的型态属性集合：

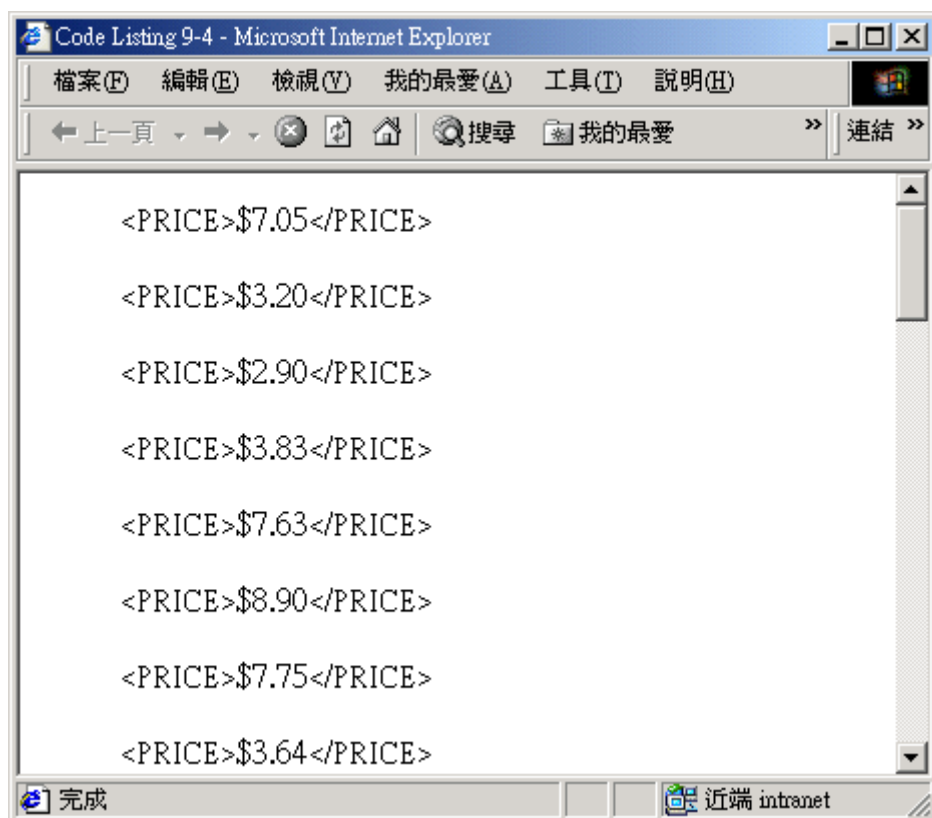
```
var qry=xmlDoc.selectNodes
( "//PLANT/AVAILABILITY/@USONLY[nodeType ( ) ='2']" ) ;
```

nodeName 方法

nodeName 方法传回在目录中所指定的节点名称，比如下面的查询范例，传回一个节点名称为

PRICE 的节点集合，执行结果如下图所示：

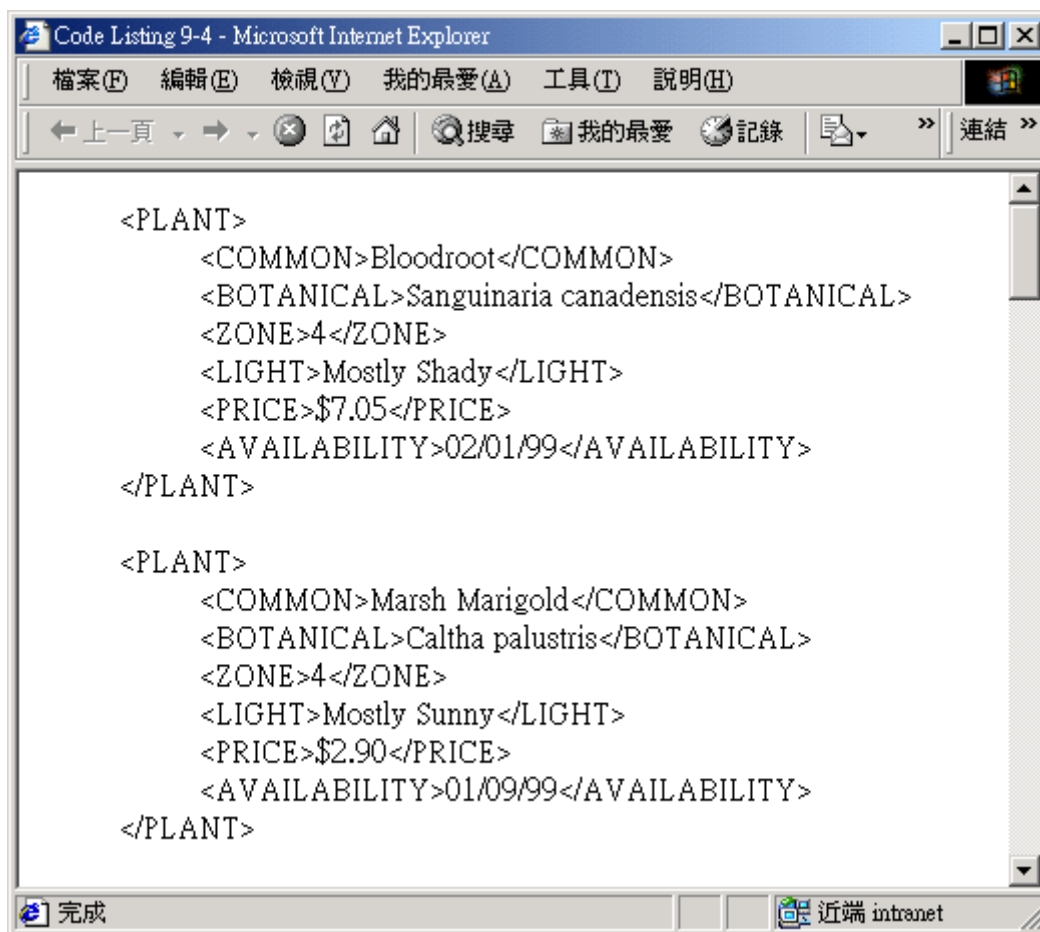
```
var qry=xmlDoc.selectNodes ("//PLANT /*[nodeName () ='PRICE']") ;
```



value 方法

使用 **value** 方法来取得在目录中指定的节点值，而在多数情况下，该值会与使用 **text** 方法传回的文字相同，但是如果支持转换（**cast**）的话，则 **value** 方法将会传回型态转换的值。底下的查询传回 **Zone** 元素值为 **4** 的节点集合，执行结果如下图所示：

```
var qry=xmlDoc.selectNodes ("//PLANT[ZONE!value () =4]");
```

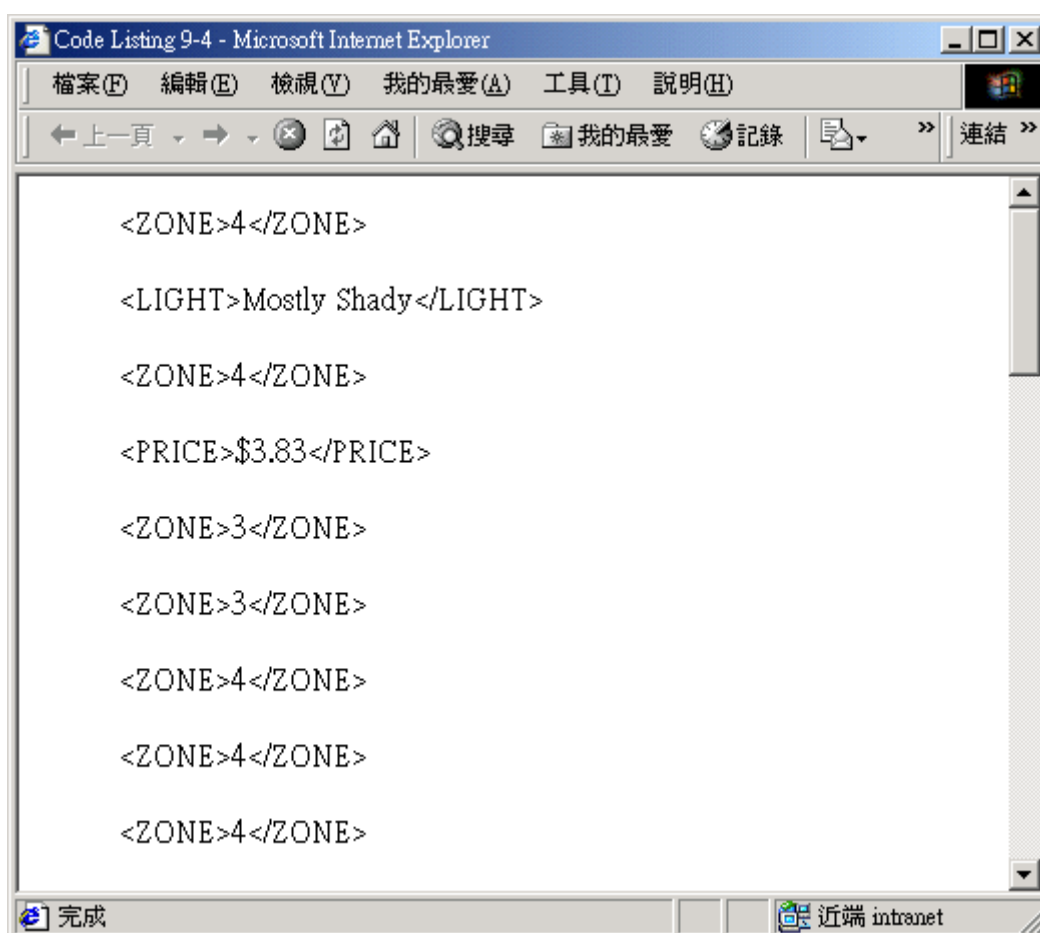


index 方法

`index` 方法传回节点与其父节点间关系的索引值，请注意索引值是以 0 为基准，所以第一个子节点的索引值为 0。底下的查询范例在 **PLANT** 节点下寻找索引值为 2 的子节点：

```
var qry=xmlDoc.selectNodes ("//PLANT/*[index () =2]") ;
```

查询结果显示如下：



底下的查询在整个集合中寻找第三节点（索引值为 2），为 **PLANT** 的子节点：

```
var qry=xmlDoc.selectNodes (" (//PLANT/*) [index () =2]");
```

最后，这个查询例子在第四 **PLANT** 节点下寻找第三子节点：

```
var qry=xmlDoc.selectNodes ("//PLANT[index () =3]/*[index () =2]");
```

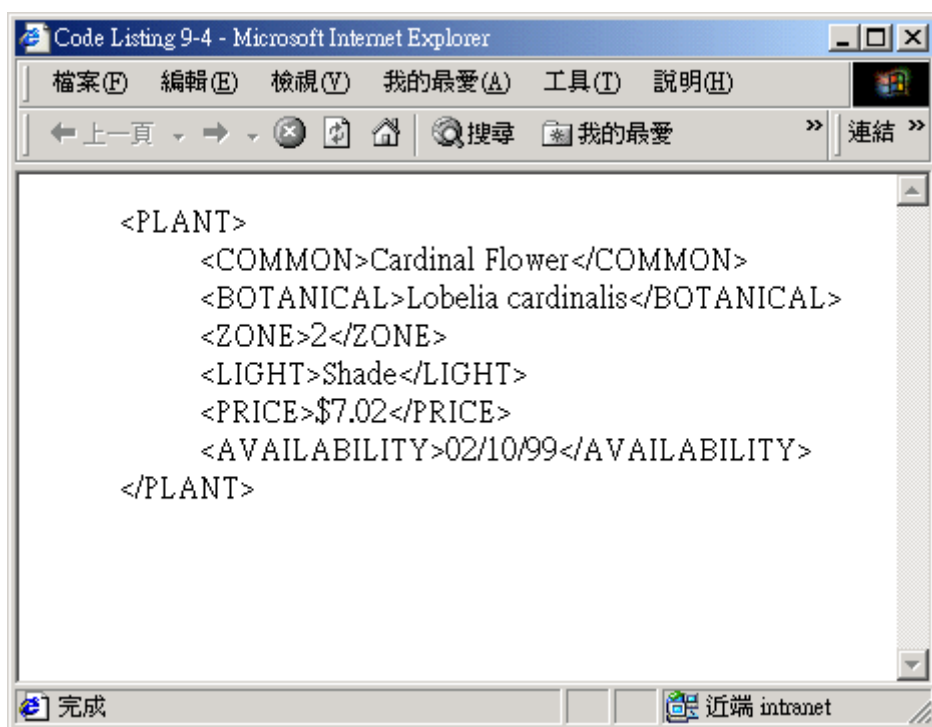
end 方法

若某元素为集合中的最后一个元素，**end** 方法会传回 **true**，其它状况则传回 **false**。底下的查询

寻找文件中所有 **PLANT** 节点集合中的最后一个 **PLANT** 节点：

```
var qry=xmlDoc.selectNodes ("//PLANT[end () ]");
```

该查询结果为最后一个 **PLANT** 节点，如下所示：



除了上面所讨论的方法之外，XSL Patterns 也支持其它方法，包括第 8 章中提到内建的 XSL 方法，这些 XSL Patterns 额外的方法可以归类为信息（information）方法及集合（collection）方法，底下的三个列表为 XSL Patterns 所提供的方法分类说明与介绍。

XSL 方法名称	说明
absoluteChildNumber	传回所有与指定节点相关的兄弟节点数目。
ancestorChildNumbe	传回与指定名称节点最接近的上层节点数目。
childNumber	传回与兄弟节点相同名称的相关节点数目。
depth	传回指定节点在文件树状结构中的阶层深度。
ElementIndexList	传回指定节点及其所有父节点的子节点数目数组，其循环取决于 root 节点。

formatDate	提供特定的格式化选项来做日期的格式化。
formatIndex	提供特定的数字系统来做整数的格式化。
formatNumber	提供特定的格式来做数字的格式化。
formatTime	提供特定的格式化选项来做时间的格式化。
uniqueID	传回指定节点的单一指标。
Information 方法	说明
date	转换数值成为日期格式。
end	若某元素为集合中的最后一个元素，end 方法会传回 true，其它状况则传回 false。
index	传回某个节点与其父节点关系的索引值。
nodeName	指出节点的标签名称，包括前置的命名空间。
nodeType	传回一个可以指出元素类型的数值。
text	指出元素中所含的文字。
value	传回一个元素型态转换版本的值。
Collection 方法	说明
ancestor	寻找最接近且符合型态的上层节点，传回单一元素或 null 值。

attribute	传回所有属性的集合。若提供选择性的文字参数，便只传回符合该特定文字的属性。
comment	传回批注节点的集合。
element	传回所有元素节点的集合。若提供选择性的文字参数，便只传回符合该特定文字的子元素。
node	传回所有不具属性的节点集合。
pi	为文件传回所有处理器指令节点的集合。
textnode	为目前文件传回所有文字节点的集合。

XSL Patterns 的对象模型

XSL Patterns 能在 Script 程序代码中使用，就必须包含一个可以从外部取得的对象模型。很幸运的，XSL Patterns 对象模型非常简单，事实上，它只包含一个对象及两个方法。XSL Patterns 对象是一种 nodeList 对象，XSL Patterns 提供两个方法可以使用在 nodeList 对象上来执行查询，这两个方法为 selectNodes 方法和 selectSingleNode 方法。

NodeList 物件

XSL Patterns 对象模型之所以简单，是因为查询总是传回 XML 的 **nodeList** 对象，因为传回的节点清单是个普通的 XML 对象，所以标准的 XML 对象模块才能接管。节点清单可以像任何的 XML 数据一样被处理，此时所有 XML 对象模块的属性及方法都是有用的。

SelectNodes 方法

假如您跟着本章的范例实作，现在应该可以熟悉 **selectNodes** 方法，**selectNodes** 方法传回一个 **nodeList** 对象，以便可以在其内容上执行。所有本章中的范例都已经执行过，所以我们现在可以来看看该查询结果的内容，文字码 9-6（随书光盘中的 Chap09\Lst9_6.htm）显示一个警告讯息，如图 9-5 所示。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>


<HEAD>

    <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

        loadDoc();

    </SCRIPT>


    <SCRIPT LANGUAGE="JavaScript" SRC="Lst9_5.js">
```

```
</SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">
```

```
var rootElem;
```

```
var xmlDoc = new ActiveXObject("microsoft.xmlDOM");
```

```
xmlDoc.load("Lst9_3.xml");
```

```
function loadDoc()
```

```
{
```

```
if (xmlDoc.readyState == 4)
```

```
start();
```

```
else
```

```
window.setTimeout("loadDoc()", 250);
```

```
}
```

```
function start()
```

```
{
```

```
var qry = xmlDoc.selectNodes("CATALOG/PLANT");
```

```
alert(qry);
```

```
}
```

```
</SCRIPT>

<TITLE>Code Listing 9-6</TITLE>

</HEAD>

<BODY>

</BODY>

</HTML>
```

文字码 9-6



图 9-5: 从 XSL Patterns 查询传回一个 nodeList 对象。

SelectSingleNode 方法

在本章开头处，我们已经简单地介绍过这个方法， 相对于 **selectNodes** 方法传回所有符合指定型态的节点， **selectSingleNode** 方法仅传回符合型态的第一个节点。假如我们更改文字码 9-4 一开始的 **start** 函式，改使用 **selectSingleNode** 方法，则结果将只会传回第一个符合的节点，而不是所有符合型态节点的集合，执行结果如下图所示：

```
function start()

{

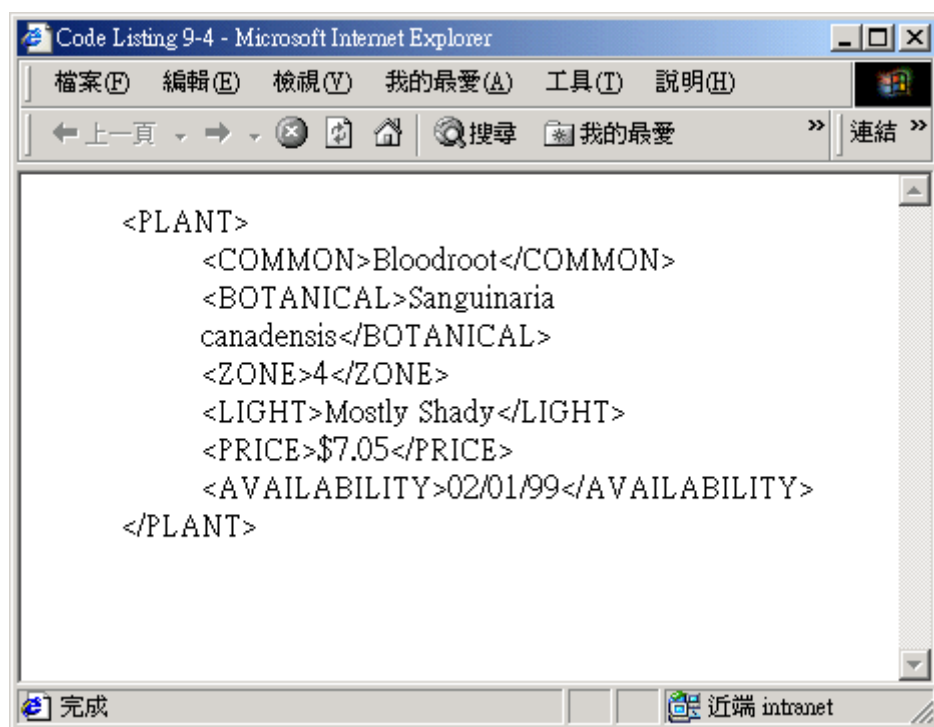
    var qry=xmlDoc.selectSingleNode("CATALOG/PLANT");

    var hTank="";

    hTank +=buildTree(qry);

    document.body.innerHTML=hTank;

}
```



XSL Patterns 的其它讯息

本章介绍了使用 XSL Patterns 的基本概念，想要取得更多关于 XQL 的信息，请参阅 W3C 的网站

<http://www.w3.org/Style/XSL/Group/1998/09/XQL-proposal.html>

请注意：在本书撰写期间，该网站仅供 W3C 成员参阅，但不久的将来势必会开放给大家。

10. XML-Data

您也许注意到这章的标题与 [第 6 章](#) 的标题<使用 XML 作为数据来源（XML As Data）>很相似，

事实用来建立结构（Schema）的语言，这个结构可用来验证架构以及限制特殊的 XML 文件。

在本章中，您将学到什么是 XML-Data，以及它为什么会被创造出来。您也会学到使用 XML-Data

结构来建立一份文件的基础。

Note

本章重点将放在微软 IE5.0 上的 XML-Data 规格。此规格是根据目前送往 W3CXML-Data 规格

的子项目所拟定的。若想知道完整的规格书，请参

考 <http://www.w3.org/TR/1998/NOTE-XML-data/>。

结构语言（Schema Language）的需求

请回想一下 [第 4 章](#) 所讨论的 DTD，它是一份 XML 文件结构的描述，也是所包含数据的规则叙

述。您应该还记得 DTD 机制是从 SGML 被引进到 XML 中的。许多应用程序开发的企业，包括

微软公司，相信 DTD 的语言对目前和未来 XML 的应用需求来说并不足够。因此，某些企业和

教育团体如微软、DataChannel 和爱丁堡大学，提议一个新的机制来达成 DTD 所能达成的一些基本工作，它还提供更强的功能及更多的弹性。

XML-Data 的目标

如同本书中谈到许多以 XML 为基础的语言有其目标，XML-Data 也订定了几个目标。如同上面所提到的，XML-Data 是被发展来避免在使用 DTD 时所引发的限制。下面指出 XML-Data 所要达成的目标，当您使用特定的语言时，便会了解 XML-Data 是如何达到这些目的。

- 目标 1：不应该要求特殊的工具来维护结构文件。

DTD 是使用特殊语法所写成的。因此，XML 文件要能使用 DTD，不只是 DTD 的文件作者要能了解它的语法，每个制作 XML 的应用程序也要有特殊的工具（可擦写 DTD）才行。XML-Data 的第一个目标便是避免对特殊工具的需求，只要 XML-Data 使用符合 XML 语法的语言，这目标便可轻易地完成。

- 目标 2：XML 和 HTML 的作者不需要学新的语法来叙述结构的信息。

这个目标和目标 1 很接近，因为结构语言不应该是一种新的语言或语法。同样地，只要使用 XML 语法就可以实现。

- 目标 3: 结构应该是可延展的。

结构应具有足够的弹性，来适用特定的应用程序而没有要求特定的条件限制。换句话说，应用程序不应该被要求提供可延展性。此弹性透过一个开放的结构定义语法来实现，因此，结构可藉由文件作者增加元素或属性而被特殊化。有关如何达到特殊化的方法，稍后本章将会讨论更多的细节。

- 目标 4: 结构语言应该尽量简单，以便应用在所有的 XML 处理器中。

再一次强调这个主题：一个标准一定要能被广泛地使用。这个目标被纳入来确保 XML-Data 语言并不会复杂到无法被大部分的 XML 程序开发者所接受。这个目标藉由使用 XML 语法来定义这个结构语言而实现，如此则可以将实作过程单纯化，并排除了要特殊处理器来解析结构的需求。

- 目标 5: XML-Data 应该符合以 Web 为发展基础的应用程序需求，而这些应用程序要求额外的数据确认可以在现今的 XML DTD 中被描述（诸如电子商务应用程序）。

正如 DTD 可以定义文件结构和规则，当定义数据型态和执行数据确认时，这个机制被限制着。XML-Data 语言定义原始的数据型态以及可以描述数据数值的范围，举例来说，藉由允许文件作者定义最小和最大值。这些典型的特性可以在关联性

数据库语言中发现，例如，结构查询语言（Structured Query Language: SQL）

以及大部分的现代程序语言。

- 目标 6: **XML-Data** 应该符合以 **Web** 为发展基础的应用程序需求，要求标准的编码以助于文件的交换。

数据的形式称为它的编码结构。**XML** 的特征之一是针对不同的数据格式执行一种

交换格式来完成数据交换。为了完成此功能，**XML-Data** 对其支持的数据型态，

定义了标准的编码结构。举例来说，在 **XML-Data** 中的浮点数使用相同标准形式，

这个标准形式可以在程序或数据库语言，如微软的 **Visual Basic** 或 **SQL** 中使用。

- 目标 7: 结构语言必须具有支持特殊文件的能力，用来组合几个来源中定义的部分。

请回想在 [第 4 章](#) 所提到的部分，**XML** 文件可以藉由使用 **DTD** 中的指针组合其它

文件或位于不同地方的文件片段。这个目标说明了 **XML-Data** 必须提供相似的功能，

藉由使用 **XML-Data** 结构中支持的命名空间来实现。

- 目标 8: 重复使用内容模型（Content Model）定义应该比使用参数实体来得简单。

请回想一下 [第 4 章](#) 所提到的，参数实体提供一个「快捷方式」将 **DTD** 组织化，

且帮助它变得更简洁。然而，参数实体的问题是：它们很快就变得复杂且难以管理。

XML-Data 的目标之一在将这些问题减到最小。藉由支持以对象为基础的内

容模型继承来实现。稍后您将会看到，内容模型继承允许文件作者重复使用内容模型定义，这种方式比参数实体更可减少麻烦且容易管理。

- 目标 9: XML-Data 必须与 XML 1.0 兼容。

这个目标是为了将「技术风暴（technology churn）」的问题减至最低程度。「技术风暴」指的是：一旦新技术出现，较旧的技术立即被更新。如目标 4 所期望的，XML-Data 要能被广泛的使用，就是让 XML-Data 使用 XML 语法。这个目标对于要求 XML-Data 必须兼容于 XML 1.0 的想法，加入了更深更广的定义。

XML-Data 结构语言

请注意：任何 XML-Data 结构必须是格式正确的（well-formed）XML 文件，同时，可能也是有效的 XML 文件。XML-Data 语言以 XML-Data 的 DTD 为基础，一份文件若是参照到 DTD，它便是有效的，但若只遵照 XML-Data 规格将成为格式正确的（well-formed）XML，即使在前言（prolog）中并没有参照到任何 DTD。

Note

如同这边所写的，还没有任何标准的单位决定 XML-Data 的 DTD 应在文件中的何处出现。但这并不会阻止特殊文件被确认为结构有效性，即使结构本身不是被确认有效的。

结构文件架构

在 [第3章](#) 中，我们讨论了基本的 XML 文件是由前言和文件元素所组成的，一个 XML-Data 结构文件也包含了这些组件。而它们两者间主要的差异在于结构文件并不包含 DTD。也就是说，文件架构在 **Schema** 元素内被定义，而 **Schema** 元素是在 **Schema** 定义中的文件或 **root** 元素。

其基本用法如下：

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

<Schema name="schemaname"

xmlns="urn:schemas-microsoft-com:xml-data">

    <!-- Declarations go here -->

    <ElementType name="elementname" content="contenttype"/>

</Schema>
```

Note

Schema 元素一定是源于 XML-Data 的命名空间 `urn: schemas-microsoft-com:xml-data`。在

Schema 的前言中宣告命名空间是不必要的。

通常会优先将 **XML-Data** 的命名空间设为预设的命名空间，但这并不是必要的。然而，藉由设定 **XML-Data** 的命名空间为默认值，您可以避免在所有结构（**Schema**）文件宣告中加上前导符（**prefix**）。举例来说，在下面的结构（**Schema**）文件中，并不假设 **XML-Data** 的命名空间为默认值。在命名空间宣告中使用前导符（**prefix**）上，下面的结构（**Schema**）文件和先前的结构（**Schema**）范例是一样。

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

<?xml:namespace ns="urn:schemas-microsoft-com:xml-data" prefix="s"?>

<s:Schema name="schemaname">

    <!-- Declarations go here -->

    <s:ElementType name="elementname" content="contenttype"/>

</s:Schema>
```

在这一章中，我们假设 **XML-Data** 的命名空间是预设的命名空间。然而，必须谨记在心的是：结构（**Schema**）宣告可以有不同的范围。也就是说，它们可能来自全域（**top-level**）宣告和区域（**local-level**）宣告的形式。宣告的范围可辨认出宣告项目在结构（**Schema**）文件中可在何处及如何被使用。

全域（**Top-Level**）宣告

全域宣告包括任何在 **Schema** 元素中宣告的元素型态或属性型态。在全域范围中，宣告的元素或属性型态可以在同一个结构中被其它元素型态的内容宣告参照。举例来说，在以下的结构中，元素型态名称在全域中被宣告，且在 **plant** 元素型态宣告中被参照。

```
<Schema name="wildflowers"
  xmlns="urn:schemas-microsoft-com:xml-data">

  <ElementType name="name" content="textonly"/>

  <ElementType name="plant">

    <element type="name"/>

  </ElementType>

</Schema>
```

区域 (Local-Level) 宣告

不是在全域宣告中出现的宣告，皆被视为区域范围。一个区域宣告只在被宣告的宣告中可以被参照。举例来说，让我们加入一个区域属性宣告到 **plant** 元素型态宣告中。

```
<Schema name="wildflowers"
  xmlns="urn:schemas-microsoft-com:xml-data">

  <ElementType name="name" content="textonly"/>
```



```
<ElementType name="plant">

  <element type="name"/>

  <attribute name="bestseller" values="yes no"/>

</ElementType>

</Schema>
```

在这个范例中，**bestseller** 属性只能在 **plant** 元素型态定义中使用。

Note

您可能会发现上面的范例中有许多尚未谈到的新观念，请不用担心，下面将会对要讲的主题做些简单的介绍。

元素型态宣告

元素型态是在 **ElementType** 元素中被宣告的，每一个元素型态宣告必须包括 **name** 属性以辨识

元素型态，例如，下面的元素型态宣告宣告了一个名为 **plant** 的元素型态：

```
<ElementType name="plant"/>
```

可以使用 **content** 属性宣告元素所包含的内容型态，用来限制元素内容只能为指定的型态。

内容型态

每一种元素型态可以包含下面四种内容分类中的一种：**empty**、**text only**、**subelements only** 或

是 **text** 及 **subelements** 的混合。这些分类在元素型态宣告时用来表示 **content** 属性的值。以下

就是可以使用的值：

- **empty** --- 不能包含任何内容。
- **textOnly** --- 只能包含文字（**text**）内容。
- **eltOnly** --- 只能包含子元素（**subelement**）。
- **mixed** --- 可包含文字及子元素的混合值。

在下面的范例中，**plant** 元素型态只能包含文字内容：

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

除了内容的限制外，元素型态的宣告也能指定该元素可出现的样式（**pattern**），这可透过 **order** 属性达成。

内容顺序（**content order**）

order 属性在元素型态宣告中限制元素型态（**type**）显示的样式，以下是 **order** 属性可能的内容值：

- **seq** --- 元素出现的顺序必须和在元素型态宣告时元素被参照的顺序一样，这是 **eltOnly** 内容的预设型态。
- **one** --- 在元素型态宣告中必须宣告一个子元素，而且这个子元素必须出现在父元素（**parent element**）中。
- **all** --- 所有元素的型态必须在元素型态宣告中被宣告，而且必须出现一个子元素，但这个子元素可以是任何顺序。
- **many** --- 任何在元素型态宣告中被宣告的子元素，可以用任何顺序出现，这是 **mixed** 内容的默认值。

Note

在 **XML-Data** 中的顺序限制不可套用在元素属性上，属性可以以任何顺序出现在元素中，而且每个属性在一个元素中只能出现一次，这跟 **XML 1.0** 规格的属性定义一致。

下面的范例是 **plant** 元素型态宣告，包含的子元素必须以一定的顺序出现。

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

<ElementType name="growth" content="mixed"/>

<ElementType name="salesinfo" content="mixed"/>

<ElementType name="plant" content="eltOnly" order="seq">

    <element type="name"/>

    <element type="growth"/>

    <element type="salesinfo"/>

</ElementType>
```

请注意：**plant** 元素的内容型态被限制为 **eltOnly**，我们不必多此一举地去宣告 **order** 属性的值，

因为 **eltOnly** 内容的预设顺序即为 **seq**，在这里包含 **order** 属性只是为了方便示范而已。

元素内容可以透过 **Group** 元素将元素参照分组，来更进一步地限制。**Group** 元素支持和

ElementType 元素相同的 **Order** 属性值。现在让我们用 **Group** 元素来产生一个新的内容模型：

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

<ElementType name="zone" content="textOnly"/>

<ElementType name="light" content="textOnly"/>

<ElementType name="price" content="textOnly"/>


<ElementType name="plant" content="eltOnly" order="seq">

  <element type="name"/>

  <group order="one">

    <element type="zone"/>

    <element type="light"/>

    <element type="price"/>

  </group>

</ElementType>
```

在这里，**plant** 元素型态必须包含一个名称，然后在名称后面跟着 **Zone**、**Light** 或 **Price** 元素中的一个。要注意的是：如果没有替 **group** 宣告顺序限制，则默认值将为 **order** 中定义的值。所以在上面的范例中，如果把 **group** 里面的 **order** 宣告拿掉，**order** 就会预设为 **seq**。

元素及群组数量

如果是 XML 的 DTD 的话，限制可以被替换为一个元素或群组在文件中可能出现或是可能不出现的次数，您可以在 Element 及 Group 元素中指定 minOccurs 及 maxOccurs 属性的内容。

minOccurs 属性表示元素最少可出现的次数；maxOccurs 属性则表示元素最多可以出现的次数，

表 10-1 列出 minOccurs 及 maxOccurs 属性可能的内容值组合及其意义。

表 10-1. minOccurs 及 maxOccurs 各种组合的意义

minOccurs	MaxOccurs	元素或群组可以出现的次数
1 或不指定	1 或不指定	1（必要的）
0	1 或不指定	0 或 1（选择性）
大于 1	大于 n	至少 minOccurs 次，不超过 maxOccurs 次
大于 1	小于 1	0
0	"*"	任何次数
1	"*"	至少一次
大于 0	"*"	至少 minOccurs 次
任何值	0	0

minOccurs 及 **maxOccurs** 的默认值都是 1；也就是说，除非特别指定，否则在任何给予的元素型态中，元素都必需正好出现一次。现在就来示范一个例子，看看它到底该如何使用（在这边，**group** 至少要出现一次，但可以出现一次以上）。

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

<ElementType name="zone" content="textOnly"/>

<ElementType name="light" content="textOnly"/>

<ElementType name="price" content="textOnly"/>


<ElementType name="plant" content="eltOnly" order="seq">

  <element type="name"/>

  <group minOccurs="1" maxOccurs="*" order="one">

    <element type="zone"/>

    <element type="light"/>

    <element type="price"/>

  </group>

</ElementType>
```

属性型态宣告

属性型态在 **AttributeType** 元素中被宣告。**XML-Data** 支持和 **XML 1.0 DTD** 相同的属性型态，同时也支持程序或数据库语言中常看到的其它数据型态，如下所示。

AttributeType 元素

如同 **ElementType** 元素，每个 **AttributeType** 元素都必须指定一个名称，下面的范例即宣告一个名为 **bestseller** 的属性型态：

```
<AttributeType name="bestseller"/>
```

属性型态在全域中宣告，与元素型态宣告各自独立，然后就可以在任何元素型态宣告中被参照。

在下面的范例中，我们将在 **plant** 元素型态宣告中使用 **bestseller** 属性：

```
<AttributeType name="bestseller"/>

<ElementType name="plant">

    <attribute type="bestseller"/>

</ElementType>
```

在这个范例中，该属性（**bestseller**）可视为 **plant** 元素出现在文件中的一部分。

默认值

属性型态宣告或参照也可以包括一个预设属性，也就是预设的属性值。例如下面的 **schema** 中，

在属性型态宣告里包含一个预设的属性：

```
<AttributeType name="bestseller" default="yes"/>

<ElementType name="plant">

  <attribute type="bestseller"/>

</ElementType>
```

如此，预设属性的值将会影响任何使用到该属性型态的元素，我们改变了上面的范例如下：

```
<AttributeType name="bestseller"/>

<ElementType name="plant">

  <attribute type="bestseller" default="no"/>

</ElementType>
```

当在 **plant** 元素中使用 **bestseller** 时，预设属性的值只能影响到该属性。如果同时在两个地方设

定预设属性的值，如下所示：

```
<AttributeType name="bestseller" default="yes"/>

<ElementType name="plant">

  <attribute type="bestseller" default="no"/>

</ElementType>
```

指定在宣告阶层（**declaration level**）的预设属性会优先发生作用，所以如果使用上面的范例，

除了在 **plant** 元素中默认值 **no** 会发生作用外，其它使用 **bestseller** 属性的地方都是默认值 **yes**

的影响范围。

Required 属性

属性型态宣告或参照可以包含一个 **required** 属性，用来辨识该属性是否必须有值。

```
<ElementType name="plant">
  <attribute type="bestseller" default="no" required="yes"/>
</ElementType>
```

上例中，在 **Plant** 元素里的 **bestseller** 属性必须要有值，而且因为有预设属性，因此除非设定其

它的值，否则 **bestseller** 将会一直是该默认值。

数据型态

属性的值以及元素可以被限制为特定的数据型态实例。**XML-Data** 较 **XML 1.0** 支持更多的数据型

态，就如同结构是由 **XML-Data** 命名空间定义的，数据型态则由 **datatype** 的命名空间定义，如

下所示：

```
<Schema name="wildflowers" xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="uuid:C2F41010-65B3-11d1-A29F-00AA00C14882">

  <AttributeType name="dateordered"/>

  <ElementType name="plant">

    <attribute type="dateordered"/>

  </ElementType>

</Schema>
```

Note

要使用 **datatypes** 的命名空间，必须像上面的范例一样先行宣告。在我们的范例中将会使用 **dt:**

当做前缀，当然，如果您使用其它的前缀也可以得到相同的结果。

type 属性

透过使用 **datatypes** 命名空间中 **type** 属性的参照来指定数据类型，在下面的范例中，我们为

dateordered 属性型态指定了一个数据类型：

```
<AttributeType name="dateordered" dt:type="dateTime"/>
```

```
<ElementType name="plant">

  <attribute type="dateordered"/>

</ElementType>
```

DateTime 数据类型符合属性支持的数据型态之一。

属性数据类型

表 10-2 中列出了属性支持的数据型态

表 10-2 支持的属性数据类型

数据类型名 称	XML 的表示方 式	意义
id	ID	属性值必须是独一无二的，如果一份文件中包含一个以上的 id 属性具有相同的值，则处理器会产生错误讯息。
string	PCDATA	在属性中只能使用字符数据。
entity	ENTITY	此属性值必须参照到一个实体（entity）。
entities	ENTITIES	与 entity 一样，但是允许两个以上由空白隔开的值。
idref	IDREF	值必须参照到在文件其它地方宣告过的 id 属性。
idrefs	IDREFS	与 idref 一样，但是允许两个以上由空白隔开的值。

nmtoken	NMTOKEN	属性值为任何名称符记（name token）字符的混合，可能是字母、数字、句点、长画符号（dash）、冒号或是底线。
nmtokens	NMTOKENS	与 nmtoken 一样，但是允许两个以上由空白隔开的值。
enumeration	ENUMERATION	属性值必须符合包括值中的一个。
notation	NOTATION	属性值必须参照一个标记（notation）。

元素（element）数据类型

表 10-3 列出了较常支持的元素数据类型，XML-Data 支持的数据型态完整列表请参考 [附录 B](#)。

表 10-3 支持的元素数据类型

数据类型名称	意义
boolean	1 或 0
string	字符数据
float	为带正负符号、整数、分数的数字，基本上位数没有限制，可包含指数。
int	为不带正负符号的整数，不包含指数。
number	为带正负符号、整数、分数的数字，基本上位数没有限制，可包含指数。
Uri	Universal Resource Identifier。
Uuid	十六进制表示八进位，选择性的连字号（-）可被忽略。

数据类型限制

限制可被包含在数据类型值里面，利用这些限制可进一步鉴别元素或属性中可包含的数据性质。

- min 与 max

min 与 max 属性定义了元素或属性中数据的上下界。例如，下面的属性可包含介于最小值 0 到最大值 50 的值：

```
<attribute type="inventory" dt:type="int" dt:min="0"
dt:max="50"/>
```

- 列举

有时候可能需要将元素或属性的值列举出来，下列的范例将定义 enumeration 数据类型以及 value 属性：

```
<ElementType name="growinginfo">
```

```
<datatype dt:type="enumeration" dt:values="sun partialsun
shade"/>

</ElementType>
```

被列举的值可由空白加以间隔。

- 最大数据长度

maxLength 属性以字符数目指出数据的长度。在下面的例子中，**Name** 元素最多

只能包含长度为 **30** 个字符的数据。

```
<ElementType name="name" dt:type="string"
dt:maxLength="30"/>

<ElementType name="plant">

    <element type="name"/>

</ElementType>
```

结构（Schema）实例

到目前为止，我们已经讨论了许多主题，但是尚未把它们整合起来，为了符合本书书名的精神，现在就来实际运用结构（Schema）。XML-Data 的目标之一在提供 DTD 之外的一个替代方案，接下来，我们将把在第 4 章的 DTD 范例以 XML-Data 结构的方式重做一次，让您知道 DTD 与结构是如何搭配的。

Email 文件

回想一下，我们曾经使用的 XML 文件范例（文字码 4-5）是一个包含标头及内容的电子邮件，如下所示：

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

<!DOCTYPE EMAIL SYSTEM "Lst4_4.dtd">

<EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

  <TO>Jodie@msn.com</TO>

  <FROM>&SIGNATURE;@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <BCC>Naomi@msn.com</BCC>

  <SUBJECT>Sample Document with External DTD</SUBJECT>
```



```
<BODY>

  Hello, this is &SIGNATURE;.

  Take care, -&SIGNATURE;

</BODY>

</EMAIL>
```

请注意这份文件包含属性、内部实体以及一个在前言中的 DTD 参照（reference）。

DTD 文件

我们的工作之一就是要用一个结构（Schema）来取代 DTD。首先，请再看一次文字码 4-4：

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

<!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?) >

<!ATTLIST EMAIL

  LANGUAGE (Western|Greek|Latin|Universal) "Western"

  ENCRYPTED CDATA #IMPLIED

  PRIORITY (NORMAL|LOW|HIGH) "NORMAL">

<!ELEMENT TO (#PCDATA) >

<!ELEMENT FROM (#PCDATA) >

<!ELEMENT CC (#PCDATA) >
```

```
<!ELEMENT BCC (#PCDATA) >

<!ATTLIST BCC

    HIDDEN CDATA #FIXED "TRUE">

<!ELEMENT SUBJECT (#PCDATA) >

<!ELEMENT BODY (#PCDATA) >

<!ENTITY SIGNATURE "Bill">
```

您应该可以在 **DTD** 中看到一些重要的项目，它们将会被并到我们的结构（**Schema**）中，除了定义电子邮件的总体结构外，此 **DTD** 亦包含其它的特定项目：

- **To** 元素旁边带有一个加号（+），表示它必须出现一次或一次以上。
- **From** 元素不带任何符号，表示必须且只能出现一次。
- **Cc** 与 **Bcc** 元素都带有一个星号（*），表示其可有可无，且可出现一次以上。
- **Subject** 与 **Body** 元素都带有问号（?），表示其可有可无，且最多只能出现一次。

- Email 与 Bcc 元素有相对应的属性，且属性带有默认值。
- 大部分的元素都被宣告为文字元素（#PCDATA）。
- 实体 Signature 被宣告且包含一个值。

这些项目提供我们制作符合 DTD 标准的结构（Schema）文件时所需要的信息。最后一部分，

我们需要使用第 4 章的 HTML 网页当作样板来显示资料：

HTML 网页

以下便是用来显示 XML 内容的 HTML 网页文字码：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

  <HEAD>

    <SCRIPT LANGUAGE="JavaScript" FOR=window EVENT=onload>

      loadDoc ( ) ;

    </SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">

    var xmlDoc = new ActiveXObject ("microsoft.xmlDOM") ;

    xmlDoc.load ("Lst4_5.xml") ;


    function loadDoc ()

    {

        if (xmlDoc.readyState == "4")

            start () ;

        else

            window.setTimeout ("loadDoc () ", 4000) ;

    }


    function start ()

    {

        var rootElem = xmlDoc.documentElement;

        var rootLength = rootElem.childNodes.length;

        for (cl=0; cl<rootLength; cl++)

        {

            currNode = rootElem.childNodes.item (cl) ;
```

```
switch (currNode.nodeName)

{

case "TO":

    todata.innerText=currNode.text;

    break;

case "FROM":

    fromdata.innerText=currNode.text;

    break;

case "CC":

    ccdata.innerText=currNode.text;

    break;

case "SUBJECT":

    subjectdata.innerText=currNode.text;

    break;

case "BODY":

    bodydata.innerText=currNode.text;

    break;

}

}

}
```

</SCRIPT>

<TITLE>Untitled</TITLE>

</HEAD>

<BODY>

<DIV ID="to" STYLE="font-weight:bold;font-size:16">

To:

</DIV>

<DIV ID="from" STYLE="font-weight:bold;font-size:16">

From:

</DIV>

<DIV ID="cc" STYLE="font-weight:bold;font-size:16">

Cc:


```
</DIV>

<BR>

<DIV ID="from" STYLE="font-weight:bold;font-size:16">

    Subject:

    <SPAN ID="subjectdata" STYLE="font-weight:normal"></SPAN>

</DIV>

<BR>

<HR>

<SPAN ID="bodydata" STYLE="font-weight:normal"></SPAN>

<P>

</BODY>

</HTML>
```

结果显示在图 10-1:

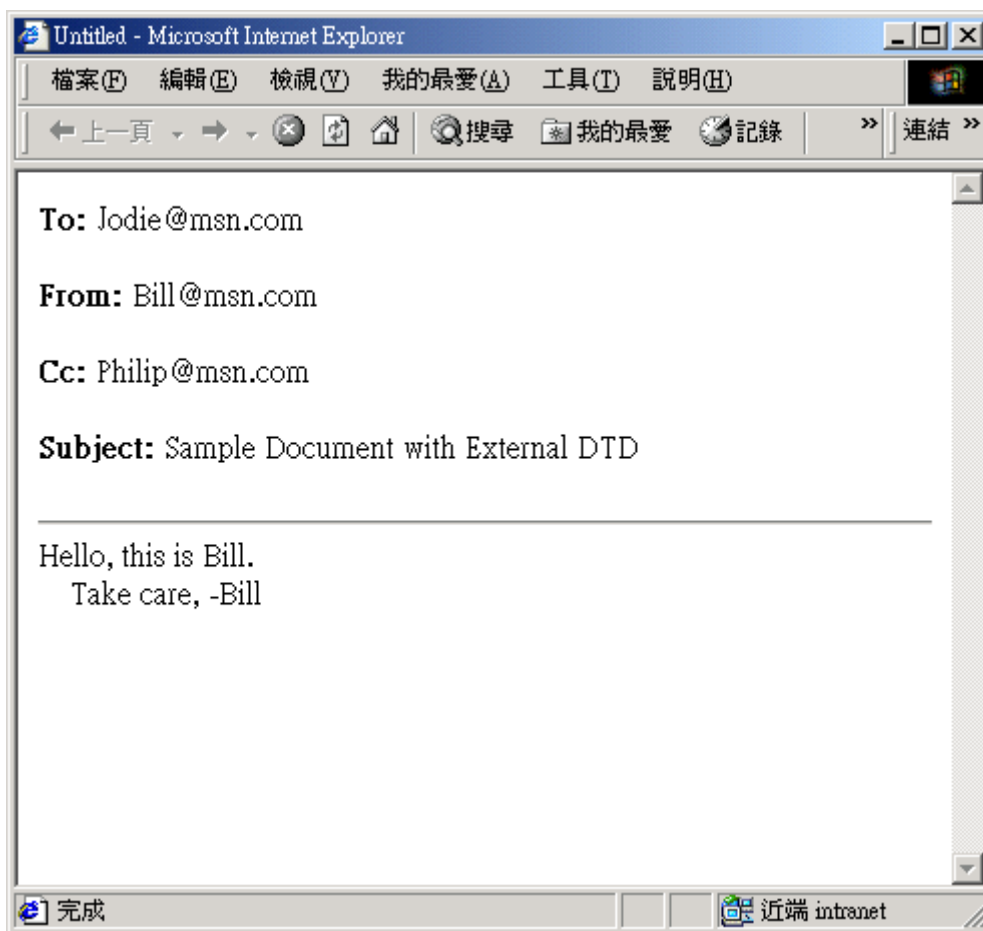


图 10-1：使用 DTD 与 HTML 网页的电子邮件。

结构（Schema）

现在就让我们来制作结构（Schema）的版本。首先要制作一份结构（Schema）文件，请记住，

我们必须提供与 DTD 一样的语意及结构规则，如下面文字码 10-1 所示（在随书光盘中的

chap10\Lst10_1.xml）：

Note

目前微软 IE5.0 中的 XML-Data 版本尚未支持实体，所以我们无法在结构（Schema）文件中复制实体的功能。此外，实体已列入向 W3C 所提报的规格书中，因此，未来 XML-Data 的版本应该会提供支持。

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

<Schema name="email" xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">

  <AttributeType name="language"

    dt:type="enumeration" dt:values="Western Greek Latin
    Universal"/>

  <AttributeType name="encrypted"/>

  <AttributeType name="priority"

    dt:type="enumeration" dt:values="NORMAL LOW HIGH"/>

  <AttributeType name="hidden" default="true"/>

  <ElementType name="to" content="textOnly"/>

  <ElementType name="from" content="textOnly"/>

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

```
<ElementType name="bcc" content="mixed">
```

```
  <attribute type="hidden" required="yes"/>
```

```
</ElementType>
```

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

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

```
<ElementType name="email" content="eltOnly">
```

```
  <attribute type="language" default="Western"/>
```

```
  <attribute type="encrypted"/>
```

```
  <attribute type="priority" default="NORMAL"/>
```

```
  <element type="to" minOccurs="1" maxOccurs="*/>
```

```
  <element type="from" minOccurs="1" maxOccurs="1"/>
```

```
  <element type="cc" minOccurs="0" maxOccurs="*/>
```

```
  <element type="bcc" minOccurs="0" maxOccurs="*/>
```

```
  <element type="subject" minOccurs="0" maxOccurs="1"/>
```

```
  <element type="body" minOccurs="0" maxOccurs="1"/>
```

```
</ElementType>
```

</Schema>

文字码 10-1

这份结构（Schema）看起来跟 DTD 有点不同，还好它们都是 XML 文字码，所以语法及结构看起来应该蛮熟悉的。现在让我们依照需求列表一一检查是否有任何遗漏。

- **To** 元素宣告为至少必须出现一次或一次以上。

请检查，并特别注意被参照的 **To** 元素使用属性 `minOccurs="1"` 及 `maxOccurs="**"`，则表示这个元素至少要出现一次，但是出现次数没有限制。

- **From** 元素宣告为必须出现一次。

请检查，并特别注意被参照的 **From** 元素使用属性 `minOccurs="1"` 及 `maxOccurs="1"`，则表示这个元素至少要出现一次。

- **Cc** 及 **Bcc** 元素宣告为选择性元素，而且能出现一次以上。

这两个元素皆包含属性 `minOccurs="0"` 及 `maxOccurs="**"`，即表示这两个元素为选择性且没有限制出现次数。

- **Subject** 及 **Body** 元素宣告为选择性元素，而且最多只能出现一次。

是的，`minOccurs="0"`及`maxOccurs="1"`属性表示这两个元素为选择性元素，而且最多只能出现一次。

- **Email** 及 **Bcc** 元素具有相关的属性，而且拥有适当的默认值。

请检查，您可以看到在文件的全域中已对属性型态（**types**）加以宣告，然后在对应的元素型态宣告中被参照。请注意，文件中的全域宣告属性型态并不是必要的，我们可以只将属性宣告放在全域中。但由于这是一份相当小的文件，为了要展示，请将宣告放在全域中（**top-level**）。

- 大部分的元素都被宣告为文字（**text**）元素。

是的，大部分的元素型态宣告都包括 `content="textOnly"` 属性，表示只能包含文字。唯一的例外是 **Email** 元素，它只能包含子元素（**subelement**）及 **Bcc** 元素（**Bcc** 元素包含一附加文字的属性）。

- **Signature** 实体被宣告且包含一个值。

在 **XML-Data** 结构（**schema**）中尚未支持实体，因此这边不需要加任何东西进来。

行文至此，好像前面的结构（**schema**）已经达到所有电子邮件文件型态的要求，现在让我们来修改 XML 文件，让它能跟我们的结构（**schema**）一起运作，文字码 10-2 如下（在随书光盘的 `chap10\Lst10_2.xml` 中）所示：

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

<em:email xmlns:em="x-schema:Lst10_1.xml"

  language="Western" encrypted="128" priority="HIGH">

  <em:to>Jodie@msn.com</em:to>

  <em:from>Bill@msn.com</em:from>

  <em:cc>Philip@msn.com</em:cc>

  <em:subject>My first schema</em:subject>

  <em:body>Hello, this is Bill. XML is cool! Take care, -Bill.</em:body>

</em:email>
```

文字码 10-2

您可能注意到 XML 文件的 DTD 版本中有两个主要的改变。首先，一开始我们在 **em: email** 元素中有一个结构（**schema**）的命名空间参照而非 DTD 参照；第二，所有在文件中的元素卷标都包含 **em** 前缀，代表它们是组合命名空间的一部分。除了上面这两个细小但重要的改变之外，基本上它跟第 4 章的范例是相同的文件，我们可以有另外一份结构规则相同的文件，因为我们的结构（**Schema**）在功能上符合先前产生的 DTD。

现在我们需要找一个方法来显示这份 XML 文件。只要透过第 4 章的 HTML 文件即可完成，但现在我们要做一些不同的事情，并使用一些没讨论过的显示方法。为了要格式化 XML 文件，我们将利用一个 XSL 样式表，如下所示之文字码 10-3（于随书光盘的 chap10\Lst10_3.xsl 中）：

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

```
<xsl:template xmlns:xsl="uri:xsl">
```

```
  <DIV STYLE="font-weight:bold;font-size:16">
```

```
    To:
```

```
    <SPAN STYLE="font-weight:normal">
```

```
      <xsl:value-of select="em:email/em:to"/>
```

```
    </SPAN>
```

```
  </DIV>
```

```
<BR></BR>
```

```
  <DIV STYLE="font-weight:bold;font-size:16">
```

```
    From:
```

```
    <SPAN STYLE="font-weight:normal">
```

```
      <xsl:value-of select="em:email/em:from"/>
```

```
    </SPAN>
```

```
  </DIV>
```

```
<BR></BR>
```

```
  <DIV STYLE="font-weight:bold;font-size:16">
```

```
Cc:

<SPAN STYLE="font-weight:normal">

    <xsl:value-of select="em:email/em:cc"/>

</SPAN>

</DIV>

<BR></BR>

<DIV STYLE="font-weight:bold;font-size:16">

    Subject:

    <SPAN STYLE="font-weight:normal">

        <xsl:value-of select="em:email/em:subject"/>

    </SPAN>

</DIV>

<HR></HR>

<SPAN STYLE="font-weight:normal">

    <xsl:value-of select="em:email/em:body"/>

</SPAN>

</xsl:template>
```

请记住，这个样式表是为了电子邮件文件型态而特别设计的，并不适用于其它的文件。好处在于它指定了一种明确的文件型态，甚至可以在不触碰到资料本身的情况下，修改与增加格式及输出的结构。此外，任何其它符合电字邮件结构（**Schema**）的文件也可套用此一样式表。

最后的部分需要一份 **HTML** 文件来执行整个工作。这份文件比第 4 章的简单很多，因为大部分负责显示的文字码都已包含在样式表中。**HTML** 文字码如下列文字码 10-4（于随书光盘的 chap10\Lst10_4.htm 中）所示：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

    <HEAD>

        <TITLE>Code Listing 10-4</TITLE>

    </HEAD>

    <SCRIPT LANGUAGE="JavaScript1" FOR="window" EVENT="onload">

        var source = new ActiveXObject ("Microsoft.XMLDOM") ;

        source.load ("Lst10_2.xml") ;

        var style = new ActiveXObject ("Microsoft.XMLDOM") ;

        style.load ("Lst10_3.xsl") ;
```



```
document.all.item ("dataContainer").innerHTML =  
    source.transformNode (style.documentElement) ;  
  
</SCRIPT>  
  
<BODY>  
    <DIV ID="dataContainer"></DIV>  
  
</BODY>  
  
</HTML>
```

文字码 10-4

执行结果如图 10-2 所示，您可看到即使文字码与原来的文件差很多，但是结果却是相同的。另外，我们得到一份有效的文件，其具备取代 DTD 结构（Schema）所附加的好处与用来格式化的 XSL 功能及弹性。

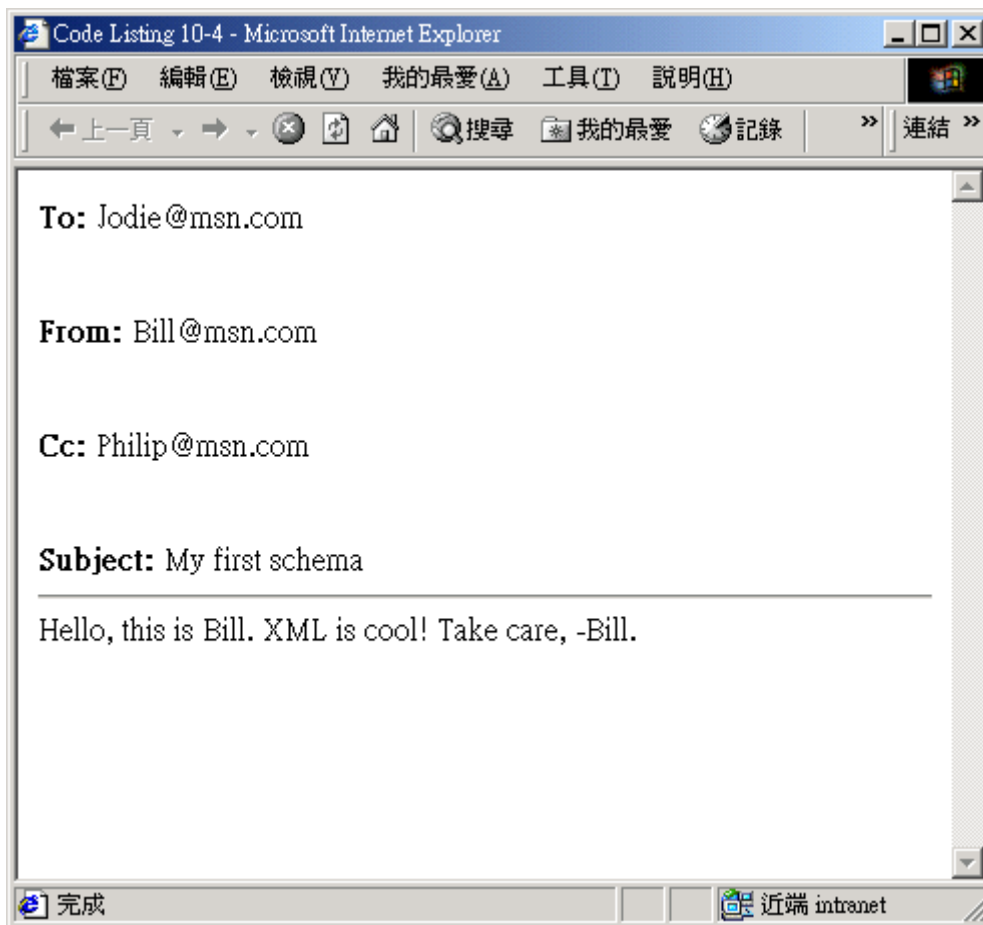


图 10-2：使用结构（schema）及 XSL 样式表的电子邮件文件。

进阶的议题

除了目前谈到的支持特点外，XML-Data 还支持 DTD 所不支持的其它特点。在这一节中，我们将会讨论一些主题，这应该会让您对进阶使用更感兴趣，例如，不同的内容模型选项、继承及 XML-Data 的子类别。

开放及封闭的内容模型（Open and Closed Content Models）

在 XML-Data 中，如果一个元素型态宣告设定内容属性（content attribute）为 empty 或 textOnly 以外的值，则该型态的元素便可包含宣告时尚未提到的属性及子元素，这就是所谓的开放的内容模型（open content model）。开放的内容模型提供更多的自由以便将未经宣告的内容加入文件中，但是如果许多人都将信息加到内容模型中，而这些都是未经宣告过的外部内容模型，它可能会变成非常不稳定的内容。

另外一种封闭的内容模型（closed content model）。就像您所猜想的，在一个封闭的内容模型中，只有在元素型态宣告中被提到的属性及子元素才可以出现在文件中，这是透过在元素型态宣告中使用模块属性而达成的，如下所示：

```
<ElementType name="plant" model="closed">

  <attribute type="dateordered"/>

  <attribute type="classification"/>

  <attribute type="inventory" dt:type="int"/>

</ElementType>
```

在这里，plant 元素只能包含三个被参照的属性。

Note

结构（Schema）元素的模块属性预设为 **open**，如此，它所有的子元素（**child elements**）也会使用相同的默认值，这可以透过改变结构元素中的模块属性值来达成，或者可以在本地端（**locally**）重写，如上面的例子所示。

元素的继承及子类别

一个元素型态可以被宣告给其它元素型态来重复使用内容模型，就是我们在第 4 章所谈到的对象导向设计，当一个内容模型被其它的元素型态重复使用时，则该新的元素型态会继承第一个内容模型所有的特性，这表示，在一个地方完成的工作可在别的地方继续发挥影响力，而且它在结构（Schema）中提供了较佳的一致性，因为同样的内容模型可在不同的地方使用。重复使用内容模型就是：将元素型态子类别化（**subclassing**），这个动作可透过 **Extends** 元素来达成。

Extends 元素

利用 **Extends** 元素不但可以重复使用一个元素型态的内容模型，而且可以在新的元素型态中详述它，如下面的范例所示：

```
<ElementType name="book">  
  
  <attribute type="language" required="yes"/>  
</ElementType>
```

```
<attribute type="coverColor"/>

<element type="title"/>

<element type="index"/>

</ElementType>

<ElementType name="textBook">

    <element type="glossary"/>

    <extends type="book"/>

</ElementType>

<ElementType name="cookBook">

    <element type="recipe"/>

    <extends type="book"/>

</ElementType>
```

这个范例中有一个称为 **book** 的基本元素型态。一个名为 **textBook** 的延伸元素型态，其中包含一个新的元素型态名为 **glossary**；还有一个名为 **cookBook** 的延伸元素型态，其中包含一个新的元素型态名为 **recipe**。因为两者都是由 **book** 延伸出来的，所以也都包含 **language**、**coverColor** 属性及 **Title**、**Index** 元素，任何时候一旦 **book** 改变，**textBook** 以及 **cookBook** 也会反应出该变化。让我们加入下面的范例码：

```
<ElementType name="scienceBook">

  <element type="diagram"/>

  <element type="formula"/>

  <extends type="textBook"/>

</ElementType>
```

scienceBook 元素型态包含 **textBook** 及 **book** 元素型态的所有部分，这是因为 **scienceBook** 为 **textBook** 的子类别，而且我们知道 **textBook** 为 **book** 的子类别，因此下面的元素是合法的：

```
<scienceBook language="English" coverColor="blue">

  <diagram>ch05d1.eps</diagram>

  <index>index.xml</index>

  <title>The Complete Book of Science</title>

</scienceBook>
```

事实上，延伸元素型态允许基本元素型态的内容模型，反之亦然。换句话说，任何允许基本元素型态的元素型态也会自动地允许延伸元素型态。在上面的范例中，意思是任何允许 **book** 的元素型态都会自动地允许 **textBook** 及 **scienceBook**，例如下面的元素型态宣告：

```
<ElementType name="library">

  <element type="book" minOccurs="1" maxOccurs="*" />

</ElementType>
```

会导致如下的结果：

```
<library language="multiple">

  <scienceBook language="Italian" coverColor="red"></scienceBook>

  <cookBook language="French"></cookBook>

</library>
```

基本规则

您可能会怀疑，所有的这些子类别及延伸元素型态会不会遇到整合不清楚的棘手情形。例如，您可能会试着延伸一个封闭内容模型的元素型态，但是却无法成功。因此，有下面几项规则来管理使用延伸元素的方式：

1. 不是基本元素型态必须有开启的内容模型，就是延伸元素型态不可以有任何内容。

这些元素型态的特性不是明确地加以定义就是从其它的基本元素型态继承而来。

2. **order** 属性的值必须一致。下面表格为允许的 **order** 属性值：

基本元素型态	延伸元素型态
seq	seq
all	all; seq

many	seq; one; all; many
------	---------------------

3.

4. **Note**

5. 如果基本元素型态的 **order** 值为 **one**，则无法发生延伸。

6.

7. **content** 属性的值必须一致。如下面表格所示：

基本元素型态	延伸元素型态
empty	empty
textOnly	textOnly; empty
eltOnly	eltOnly
mixed	mixed; textOnly; eltOnly

8. 属性与数据型态的限制可应用在所有合并的元素型态中，但如果合并的数据型态不能适当的结合会发生问题。

使用命名空间引入其它的结构（**Schema**）

在您的结构（**Schema**）中，您可以透过命名空间的使用来引入其它结构（**Schema**）的宣告。

例如，在下面的元素型态宣告中，一个命名空间的前缀被用来参照在其它结构（**schema**）中被宣告的元素型态。

```
<?xmlns:ann="uri:http://www.plants.com/schemas/annuals.xml"?>

<ElementType name="plant">

    <element type="ann:flower"/>

</ElementType>
```

如此则允许文件作者能在目前的文件中参照它，就可以存取一个个别的内容模型。

这只是个开始

在这一章中，作者只是粗略的说明可以用 **XML-Data** 来做些什么，您可以深入地研究，看看结构（**Schema**）还有哪些强大的功能。

11. XML 的现在与未来

生活在这个科技发达时代最大的好处是：新科技带来新的产品及服务，而这些产品及服务可以大大地增进我们的生活品质。当然，我们也可能变成新科技的白老鼠。本书写作时，正值 IE5.0 的开发阶段，在这段期间，XML 相关的技术（如本书中所提到的 XSL、XSL Pattern 和 XML-Data 等）仍继续不断地在发展。

本章将要讨论一些在发展中最有潜力的 XML 产品与技术，笔者希望这些讨论能点燃读者对 XML 的兴趣。

XML 对于资深程序设计师的进阶使用

其中一个执行的动机是利用 XML 来提供「永续」（**serialization**）功能，永续是一种较先进的程序设计技术，用来保持对象数据的持续性。如果建立一个代表对象结构且可以维护本身变量成员所拥有之值的「可永续」程序代码对象，并且包含在一个永续的储存机制中（比如说硬盘中的某个档案），这样对象便可以维护或者保留自己本身的状态。这个物件就可以从储存的地方透过读入重新建立对象本身，或者从储存的地方取消数据永续。在程序设计语言中，比方说 C++ 或 Java，永续是利用永续协议（**serialization protocol**）形式中的永续技术来达成，它允许对象类别（**object classes**）能够连载。当我们使用时，此协议会产生一个封闭的二进制档案（**closed binary file**），

来表示该永续对象；也就是说，它的内容只能透过特别的程序才能取得，更进一步来说，它具有语言相依性。举例来说，您无法很容易地在 **C++** 中永续一个对象，以及在 **Java** 中取消它的永续。这类工作比较复杂，而且通常是保留给较资深的程序设计师所使用。

对象的永续与 XML 的关系（Object Persistence with XML）

我们从把 **XML** 当作一种数据格式（**data format**）开始，来达成永续的目的—永续的目的是用来维护对象数据的持续性。这样做将使永续数据语言为独立语言，以便容易要求永续与取消永续的工作。而对象也可以在不同的对象架构中交换，比如微软的 **COM**（**Component Object Model**）以及对象管理群组的 **CORBA**（**Common Object Request Broker Architecture**）。底下的文字码显示一个简单的永续资料架构（**serialized data structure**）：

```
<OBJECT>

  <DATA>

    <STRING>Hello, World!</STRING>

  </DATA>

</OBJECT>
```

这几行文字码使用易懂的 **XML**，也许看起来并不令人印象深刻，但是这样单纯的动作却做了很重大的突破，因为既然简易的 **XML** 可以被用来描述复杂的对象数据（**object data**），那么也就

可以用来应用二进制数据。很明显的，如果我们要这样做的话，应用程序就必须懂得这个数据的架构是什么。而已经有一些公司与组织采用以 XML 为基础的永续技术，底下我们稍微说明一下。

Web 分布式数据交换 (Distributed Data Exchange)

XML 永续技术可以直接读取应用程序或对象（原本就具有 XML 格式），以及在两个不兼容格式交易层级中的函式，而 Web 分布式数据交换 (Web Distributed Data Exchange, 简称 WDDX) 预计完成后者。WDDX 由 Allaire 公司所发展，WDDX 是特地为 Web 应用程序所建立的技术，它由一个代表应用层级数据的建议标准 (proposed standard)，以及每个使用 WDDX 的语言或技术的永续 / 解除永续模型所组成。举例而言，在 JavaScript 中有一个解除永续工具 (deserializer)，它让 WDDX-data 能够在 JavaScript 环境中解除永续，像是 Web 浏览器。如果想知道更多的信息，请参阅 Allaire 网站，网址为 <http://www.allaire.com> 或 <http://www.codebits.com/wddx>。

Koala 对象卷标语言

Koala 对象卷标语言 (KOML) 是 Java 对象所使用的永续技术，永续 / 解除永续模型则建立在 Java 中，而且它们使用 XML 文件来当作永续性的储存方式。如果想知道更多的信息，请参阅 <http://www.inria.fr/koala/XML/serialization>。

XML 元数据对象永续 (Metadata Object Persistence)

XML 元数据对象永续 (XMOP) 是另一种以 XML 为基础的永续技术，它允许在不同的对象技术间 (比如 COM 及 Java) 相互操作。因为对象技术使用不同的永续方法，因此 XMOP 的其中一个目标便是提供一个共通的、跨技术的连续机制。如果知道更多的信息，请参阅 <http://jabr.ne.mediaone.net/documents/xmop.htm>。

COINS

Coins 是 JavaBeans 对 XML 永续化的一种选择，因为 Java 语言在各方面都被认为是一种适于开发组件的最佳程序语言，所以 JavaBeans 是一种以 Java 为基础的组件架构。这表示在组成连续化组件的 Java 模块中所做的任何改变，都可能会对该组件有负面的影响。Coins 以 XML 为基础的结构来取代 JavaBeans 的永续化，如果想要得到更多关于 Coins 的信息，您可以参阅下列网址 <http://www.jxml.com/coins/index.html>。

以 XML 为基础的永续化模型会提供一个全新的机制来处理应用程序间以及因特网上的数据交换，要达到这样的结果需要更多永续化结构的规格，这些标准化的基础已经在前面提到过。

XML 资料岛 (Data Islands)

在这本书中，我们将 XML 当作资料来源，而把 HTML 当作显示的机制。资料来源与显示网页是由分开的文件所合并的，为了撷取我们的 XML 数据到 HTML 网页中，该数据会透过使用 JavaScript、XML DSO、XSL 或 XSL Patterns 来解析，并将它插入到 HTML 文字码中。微软 IE5 的使用者还有另一个选择—XML 数据岛（data islands）。数据岛允许文件作者直接插入 XML 片段数据到一份 HTML 文件中，而这些片段数据可以是行内（inline）XML 或是被连结的 XML 文件。关于使用 XML 数据岛，有人赞成，也有人持反对立场，但它至少有下列的好处：

- 您不需要一份独立的（separate）XML 档案。
- 您不需要经由 Script 程序代码来加载 XML 数据。
- 您不需要为对象元素（Object element）明确说明 ActiveX 控件。

而其缺点则取决于如何使用数据岛，例如：

- XML 数据具 HTML 档案相依性。
- XML 数据对其他应用程序或 HTML 网页来说可携带性不高。

- 假如 XML 数据以 HTML 文字码散发，它可能变得更难管理。

使用行内 XML 数据岛（Inline XML Data Islands）

使用行内 XML 数据岛时，您只需简单地直接插入 XML 文字码到 HTML 网页即可，而 XML「文件」则被包含在一组<XML></XML> 标签中。为了说明起见，我们使用前面章节中熟悉的电子邮件文件，并将它嵌入 HTML 网页中。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

  <HEAD>

    <TITLE>Data Island</TITLE>

  </HEAD>

  <BODY>

    <XML ID="email">

      <EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

        <TO>Jodie@msn.com</TO>

        <FROM>Bill@msn.com</FROM>
```



```
<CC>Philip@msn.com</CC>

<SUBJECT>My First DTD</SUBJECT>

<BODY>Hello, World!</BODY>

</EMAIL>

</XML>

</BODY>

</HTML>
```

请注意，虽然该文件被 **Xml** 元素包住，但是该元素并不是根节点（**root**）或文件元素，因为浏览器认为包含在 **Xml** 元素中的是 **XML** 文字码。也请留意，**Xml** 元素有一个被赋予的 **ID**，如此它才能在 **HTML** 网页中的其它地方给 **Script** 程序代码建立实体参照。如果 **XML** 文字码被包含在一份分离的文件中，而且我们已经在 **HTML** 网页中建立了 **XML** 解析对象的实体，那么现在 **XML** 文字码便可以运作。文字码 11-1（随书光盘中的 **Chap11\Lst11_1.htm**）包含 **XML** 文件所有的显示机制以及 **XML** 文件的 **Script** 程序代码。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>

<HEAD>

<SCRIPT LANGUAGE="JavaScrip" FOR=window EVENT=onload>

    start ( ) ;
```

```
</SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">
```

```
function start ()
```

```
{
```

```
var rootElem = email.documentElement;
```

```
var rootLength = rootElem.childNodes.length;
```

```
for (cl=0; cl<rootLength; cl++)
```

```
{
```

```
currNode = rootElem.childNodes.item (cl)
```

```
switch (currNode.nodeName)
```

```
{
```

```
case "TO":
```

```
    todata.innerText=currNode.text;
```

```
    break;
```

```
case "FROM":
```

```
    fromdata.innerText=currNode.text;
```

```
    break;
```

```
case "CC":
```

```
    ccdata.innerText=currNode.text;
```

```
        break;

    case "SUBJECT":

        subjectdata.innerText=currNode.text;

        break;

    case "BODY":

        bodydata.innerText=currNode.text;

        break;

    }

}

}

</SCRIPT>

<TITLE>Untitled</TITLE>

</HEAD>

<BODY>

<XML ID="email">

    <EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

        <TO>Jodie@msn.com</TO>

        <FROM>Bill@msn.com</FROM>

        <CC>Philip@msn.com</CC>

        <SUBJECT>My first DTD</SUBJECT>
```

<BODY>Hello, World!</BODY>

</EMAIL>

</XML>

<DIV ID="to" STYLE="font-weight:bold;font-size:16">

To:

</DIV>

<DIV ID="from" STYLE="font-weight:bold;font-size:16">

From:

</DIV>

<DIV ID="cc" STYLE="font-weight:bold;font-size:16">

Cc:

</DIV>

```
<BR>

<DIV ID="from" STYLE="font-weight:bold;font-size:16">

    Subject:

    <SPAN ID="subjectdata" STYLE="font-weight:normal"></SPAN>

</DIV>

<BR>

<HR>

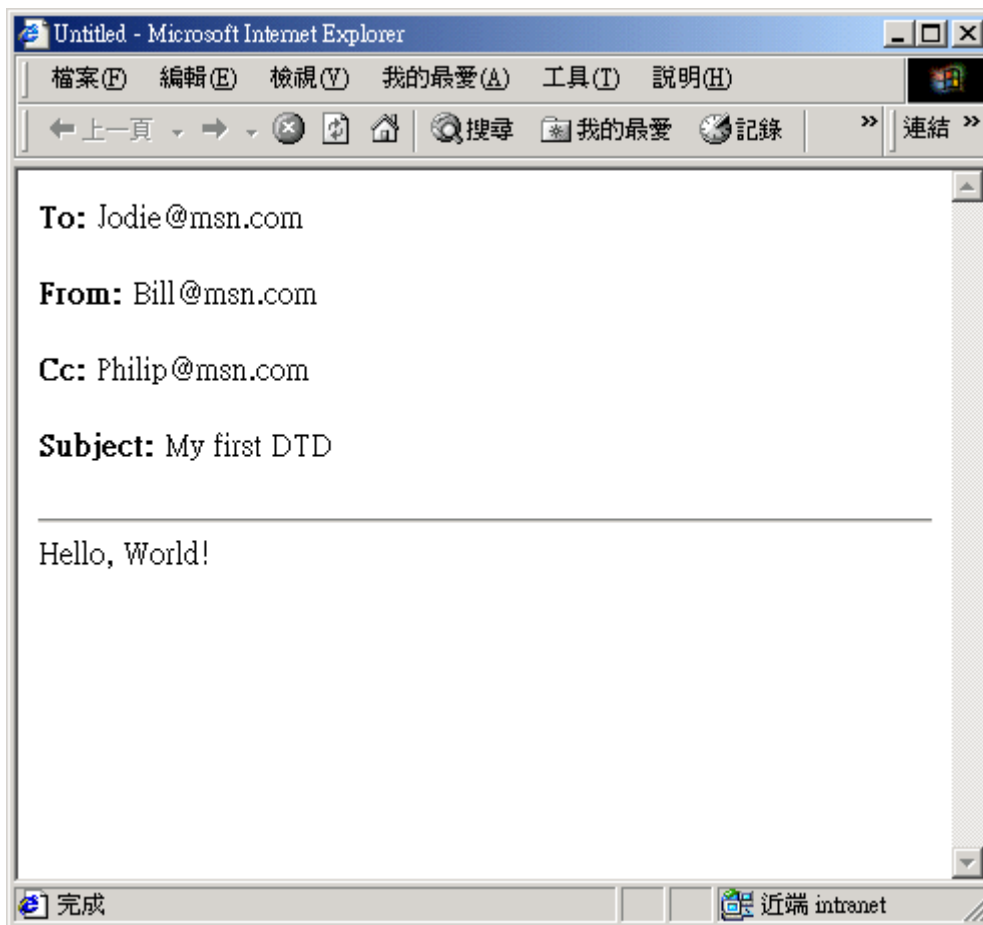
<SPAN ID="bodydata" STYLE="font-weight:normal"></SPAN><P>

</BODY>

</HTML>
```

文字码 11-1

执行结果如下图所示：



在这份文件中，XML 电子邮件文件是透过它的 ID 来参照它，当它被 ActiveX 控件呼叫时，我们也能执行该文件的所有相同函式。事实上，是同一个处理器被使用；IE5 能用另一个不同的方法使用它。除了行内数据岛（inline data island）外，还有另一个方法来使用 Xml 元素—透过连结它到一份外部文件。

使用被连结的 XML 数据岛（Linked XML Data Islands）

要连结一个 **Xml** 元素到一份外部文件时，第一件需要处理的事就是外部文件。底下的范例是由前述例子中剪下的 **XML** 程序代码，并将它放进自己所属的文件中，如下面文字码 11-2 所示（随书光盘中的 **Chap11\Lst11_2.xml**）。

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

<EMAIL LANGUAGE="Western" ENCRYPTED="128" PRIORITY="HIGH">

  <TO>Jodie@msn.com</TO>

  <FROM>Bill@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <SUBJECT>My first DTD</SUBJECT>

  <BODY>Hello, World!</BODY>

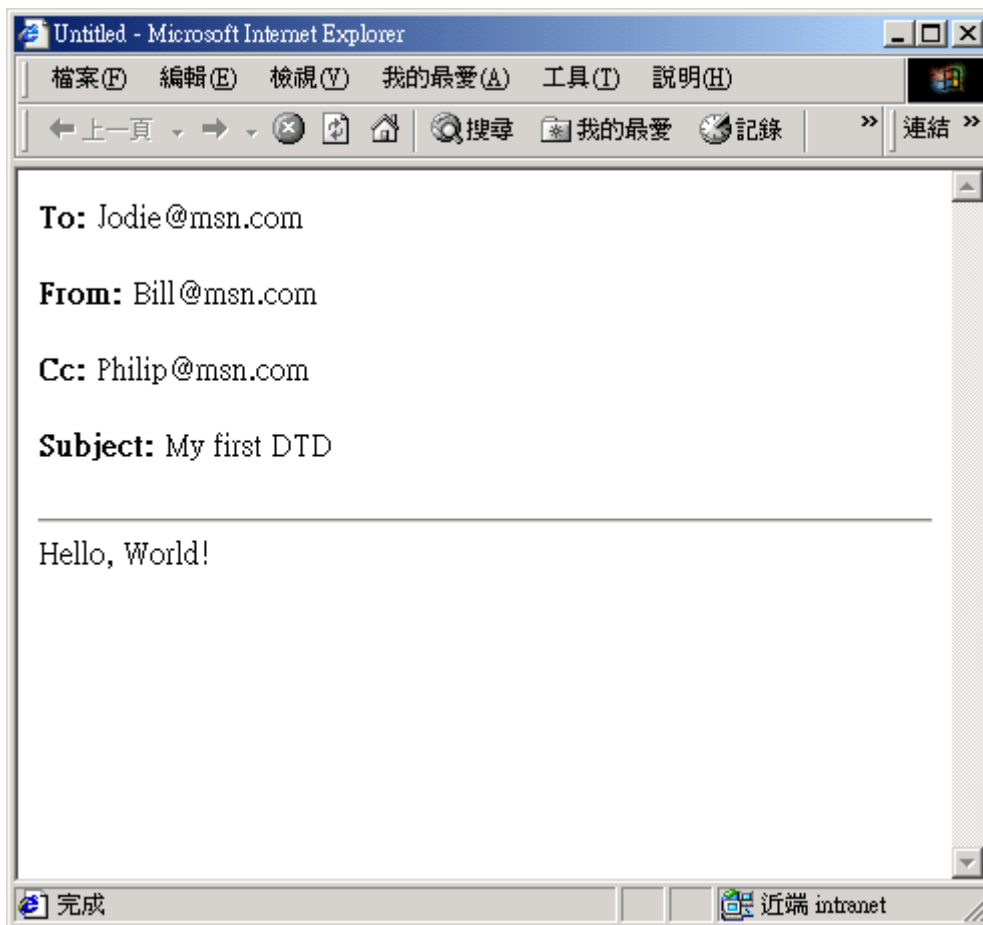
</EMAIL>
```

文字码 11-2

要连结 **Xml** 元素到这个外部文件时，只要将 **HTML** 网页中的 **Xml** 元素改变成下述的样子即可（完整的文字码可以在随书光盘的 **Chap11\Lst11_3.htm** 中找到）。

```
<XML ID="email" SRC="Lst11_2.xml"></XML>
```

执行结果如右图所示：



使用这个方法的好处是，该 XML 数据与 HTML 文件分别为独立文件。实际上，它运作的方式比较像是 ActiveX 控件方法，除了由浏览器担任控制与加载文件实体的任务比较不像外。

Xml 元素的属性

Xml 元素包含一些附加的属性（additional attributes），以便在文件被加载时提供更多的控制。

VALIDATEONPARSE 属性表示当文件被解析时，无论如何该文件都应该要确认有效。其用法如下：


```
<XML ID="xmlDoc" SRC="xmlDoc.xml" VALIDATEONPARSE="false"></XML>
```

假如元素未包含这个属性值，其默认值为 **true**。

ASYNC 属性表示无论如何文件都应该要异步下载。下面是这个元素的范例：

```
<XML ID="xmlDoc" SRC="xmlDoc.xml" ASYNC="false"></XML>
```

ASYNC 属性的默认值为 **true**。

Note

大部分本书所使用的范例都可以使用 **XML** 数据岛来运作，如果您使用 **IE5**，可能会想要回到前面去修改一些范例来使用 **XML** 数据岛（**data islands**）。

多媒体描述语言（Multimedia Description Languages）

以计算机为基础的多媒体在过去五至七年间有相当的发展。然而，其中一个让多媒体工业一直奋斗至今的问题就是：为多媒体内容的传送订定一套标准化的系统。目前市场上已经有很多的商业多媒体编辑系统（**Macromedia Director**、**Asymmetrix Toolbook**、以及 **Microsoft Liquid Motion**

等等），它们提供多媒体创作者一套完整的系统来建立以及传送多媒体内容。很多创作者使用 C++ 或 Java 来建立自订的多媒体引擎（multimedia engines），以便处理商业软件包无法处理的工作。所有这些多媒体递送系统都有一个共同点，它们使用专有的方法来传送内容。通常，在某一套系统上发展的内容经常无法在另一套系统上使用，至少，不具可携带性。由于 XML 的出现，相关单位目前已经着手进行发展多媒体描述语言（multimedia description language），以便提供文件作者一个标准的方法来描述多媒体内容，而这些内容都可以在懂得这种语言的任何系统上播放。目前较好的两种技术为 Synchronized Multimedia Integration Language（简称 SMIL，念作「smile」），以及 HTML Timed Interactive Multimedia Extensions（简称 HTML+TIME）。

SMIL

SMIL 1.0 为最近 W3C 推荐发表的语言，SMIL 以 XML 为基础，专为多媒体播送所设计的语言。它的目的在于提供一套标准化格式来描述多媒体，而且可以使用在各种编辑及显示系统上，比如说 Web 浏览器或是专门的编辑工具如 Macromedia Flash。在一般观念上，SMIL 依照以网页为基础的播送塑造自己。SMIL 文件在网页上定义出范围，允许创作者提供关于网页上对象的暂时信息，而且支持这些对象的超级链接。底下是 SMIL 文件的范例文字码：

```
<SMIL>
```

```
  <HEAD>
```

```
    <LAYOUT>
```

```
<ROOT-LAYOUT WIDTH="640" HEIGHT="480"

  BACKGROUND-COLOR="black"/>

  <!-- Logo -->

  <REGION ID="logo" LEFT="20" TOP="5" WIDTH="100" HEIGHT="50"

    FIT="fill"/>


  <!-- Video -->

  <REGION ID="vidbk" LEFT="200" TOP="50" WIDTH="150" HEIGHT="76"

    BACKGROUND-COLOR="#330033" Z-INDEX="1"/>

  <REGION ID="video" LEFT="210" TOP="55" WIDTH="100" HEIGHT="70"

    FIT="fill" BACKGROUND-COLOR="#000000" Z-INDEX="3" />

  <!-- Closed Caption -->

  <REGION ID="ccbk" LEFT="20" TOP="200" WIDTH="400" HEIGHT="30"

    BACKGROUND-COLOR="#666600" Z-INDEX="2"/>

  <REGION ID="ccscroll" LEFT="21" TOP="210" WIDTH="350"

    HEIGHT="25" FIT="fill" Z-INDEX="2" />


</LAYOUT>

</HEAD>
```

```
<BODY>

<PAR>

  <!-- Blocks below play one after the other (in parallel) -->

  <SEQ>

    <PAR>

      <IMG SRC="logo.gif" REGION="logo" FILL="freeze"/>

    </PAR>

    <PAR>

      <VIDEO SRC="video.asx" REGION="video" FILL="freeze" />

      <TEXT SRC="cctext.txt" REGION="ccscroll" FILL="freeze" />

    </PAR>

  </SEQ>

</PAR>

</BODY>

</SMIL>
```

Note

虽然微软的 IE 或网景的 Netscape 都没有计划要直接支持 SMIL，但是仍有几家公司提供支持

SMIL 的浏览器，这些功能可以经由使用独立的播放工具或 plug-in 的方式来完成，像是

RealNetworks 的 RealPlayer G2 便是这种播放工具的其中之一。

上述的例子只是要简单的显示 SMIL 文字码，并不是提供它使用的说明。如果想要知道 SMIL 更

详细的信息，以及目前可以用来编辑 SMIL 文件的工具，请参阅 W3C 的网站，网址

为 <http://www.w3.org/AudioVideo>，而目前 SMIL 的建议请参

阅 <http://www.w3.org/TR/REC-smil>。

SMIL 有一个缺点，就是虽然它是以 XML 为基础的语言，但却没有整合好目前以 Web 为基础的

多媒体解决方案，比如说 DHTML 以及 CSS。为了说明这个问题，微软联合 Compaq 计算机以

及 Macromedia，将 SMIL 加以延伸成一种以 XML 为基础的新多媒体语言，称为 HTML+TIME。

HTML+TIME

HTML+TIME 提供了两个解决方案来处理 SMIL 的缺点。第一，支持应用时间属性（time

attributes）到任何 HTML 元素的能力，所以在 Web 网页上的效果都可以排定时间。也就是说，

取代只是显示内容，例如，如果在网页上有一个 Div 元素，创作者便可以以播放的时间做基础，

指定它在何时应该出现。HTML+TIME 的支持也包括存取媒体元素暂时属性的能力，比如像是

streaming 媒体档案。也就是说，您可以取得某段激光视盘的数据。**HTML+TIME** 使用 **HTML** 来显示，并使用 **CSS** 作为网页的配置与样式依据。

第二个解决方案 **HTML+TIME** 提供支持在播放中穿越所有元素使用一般属性，这个特性允许网页上的所有元素在播放时相互影响。底下文字码简单地示范 **HTML+TIME** 属性。

```
<DIV t:begin="1">

    This Div element appears after one second.

</DIV>

<IMG SRC="pic.gif" t:begin="2">

    This image appears after two seconds.

</IMG>

<P t:begin="3">

    This text appears after three seconds.

</P>
```

Note

HTML+TIME 并不是被发展来取代 SMIL 的，更确切的说，它的目的在填补 SMIL 中以 Web 为基础所发展出来多媒体的不足，事实上 W3C 置于网

址 <http://www.w3.org/TR/NOTE-HTMLplusTIME> 的注意事项有这样的标题「将 SMIL 延伸到

Web 浏览器中」，而 HTML+TIME 的建立者也了解 SMIL 在 Web 环境之外，是一种用来描述多媒体非常好的解决方案。

请注意：HTML+TIME 属性应用到标准的 HTML 元素，如同 SMIL 一样，HTML+TIME 也支持媒体元素（media elements）的编排，如下所示的范例文字码：

```
<DIV HEIGHT=200 WIDTH=300>

  <t:seq ID="presentation" t:beginEvent="none">

    <IMG SRC="image1.gif" t:dur="5" t:timeAction="display"

      onclick="this.endElement ()">

    <IMG SRC="image2.gif" t:dur="5" t:timeAction="display"

      onclick="this.endElement ()">

  </t:seq>

  <IMG SRC="showOver.gif" t:beginAfter="presentation"

    t:timeAction="display">
```

```
<P ALIGN=center onclick="presentation.beginElement () ">

    Click here to begin the slide show. It will play automatically.

</P>

<P ALIGN=center>

    Click an image to manually advance the presentation.

</P>

</DIV>
```

此时，HTML+TIME 还只是一个提议，而它在被大力推荐前还有很多需要更改的地方，但是一般的概念与功能应该还会维持不变才对。

HTML+TIME 使得在 Web 网页上整合以 XML 为基础的多媒体变得容易许多，既然它遵循熟悉的标准，我们主要的课题就只剩下学习新的属性。

用 XML 来向量化影像（Vector Images with XML）

Web 其中一个重要的功能即是在因特网上整合图形到文字网页中并显示，然而，其中一个仍旧困扰 Web 开发人员的问题是图形的大小。因为 Web 图形档案非常大，而且也必须花很长的时间来下载，所以这些问题成为使用者在 Web 上主要的瓶颈。在我们探讨 XML 对于图形格式以及下载问题的解决方案之前，请重新检视并比较目前的 raster 及向量格式（vector formats）。

Raster 影像格式（Raster Image Format）

Raster 影像（也称为二元映像图：bitmapped images）储存着影像中每一个像素的信息，比如颜色等等。例如，假设您有一个蓝色直线的 **raster** 影像，这条蓝色直线的长度为十个像素长，以文字来说明可能如下所示：

像素一 X 轴： 20， Y 轴： 20， rgb 值： 0,0,255

像素二 X 轴： 21， Y 轴： 20， rgb 值： 0,0,255

像素三 X 轴： 22， Y 轴： 20， rgb 值： 0,0,255

像素四 X 轴： 23， Y 轴： 20， rgb 值： 0,0,255

像素五 X 轴： 24， Y 轴： 20， rgb 值： 0,0,255

像素六 X 轴： 25， Y 轴： 20， rgb 值： 0,0,255

像素七 X 轴： 26， Y 轴： 20， rgb 值： 0,0,255

像素八 X 轴： 27， Y 轴： 20， rgb 值： 0,0,255

像素九 X 轴： 28， Y 轴： 20， rgb 值： 0,0,255

像素十 X 轴： 29， Y 轴： 20， rgb 值： 0,0,255

这种格式与其它档案格式比较起来之所以非常大的原因，在于它要求像素层级的信息。此外，因为 **raster** 影像为二元档案（**binary files**），所以总是需要一种了解这种格式的编辑工具，以便在做任何改变或更新之前可以制造好影像。为了要帮助减少 **raster** 影像所占的大小，以及让它们在 **Web** 上的下载能快一点，我们通常使用压缩的方法将影像压缩。而近几年来，大家所做的都是怎样改善影像的压缩方式，但是 **raster** 影像仍旧是占用了太多的下载时间，**Web** 上常用的影像格式，比如 **GIF** 与 **JPEG**，都属于 **raster** 格式。

向量影像格式（**Vector Image Format**）

另一种已经渐受欢迎的图形格式为向量格式，这种格式储存影像信息的方式和 **raster** 格式非常不同，它并不储存影像中每一个像素的信息，而是储存在影像中对象的数理描述。换句话说，因为一张影像是由直线、椭圆、弧、长方形以及其它形状所构成，所以向量格式可以确认并描述它们个别的几何学。而向量绘制引擎（**vector rendering engine**）则可以将每个对图形的描述直译出来，并由此重新画出整张图。因为这种格式不需要一个像素接着一个像素的画出直线或圆形，图形档案的大小就变得非常小。请注意：我们刚刚使用 **raster** 格式所描述的蓝色线条、十个像素长的直线在向量格式中只要三行就可以完整说明。

直线起点 X 轴：20，Y 轴：20

直线终点 X 轴：29，Y 轴：20

RGB：0,0,255

这个简单但有效的范例可以让我们明显地看出 **raster** 以及向量格式间的差别，除了大大减少档案的大小以外，向量格式也提供另一个 **raster** 格式图形所做不到的功能，那就是让影像可以按比例自由缩放。

Note

虽然我们可以按比例缩放 **raster** 影像，但大多都会扭曲变形，或者会在处理中像素化。这是因为显示程序必须要以原始的像素信息为基础来更改影像，而向量影像只要重新绘制就可以了，因为它使用新的向量信息，所以不会扭曲变形。

用 VML 向量化影像（Vector Images with VML）

因为向量格式具有如此简洁、自然的描述手法，因此几家公司包括 Autodesk、Hewlett-Packard、Macromedia、Microsoft，以及 Visio 便想到透过 XML 制造出一种语言作为向量解译方案的基础，该向量语言的建议名称为 Vector Markup Language（向量卷标语言，简称 VML）。虽然 VML 还不是一种标准语言，但是最近已经有人向 W3C 提出正式的建议，希望让 VML 成为标准的语言，更多关于这个提案的信息请参阅网址 <http://www.w3.org/TR/NOTE-VML>。

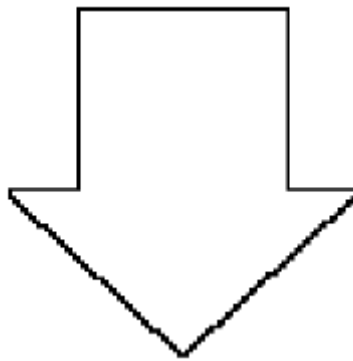
以 XML 为基础的向量格式的好处

如果您熟悉其它向量格式，比如 Microsoft Windows Metafile、Macromedia Flash、FreeHand，或其它的结构化图形（structured graphics），您可能会疑惑为何大家都想要使用 XML 来描述向量图形。那是因为 XML 对于处理这些向量图形的重大改善非常有价值。

- VML 以开放的标准 XML 为基础。这个简单且以文字为基础的语言可以用来描述小型、缩放，及可以被包括在 XML 应用程序中的图形。
- VML 也支持其它 W3C 的标准，如 CSS 及 DOM，这提供了 Web 网页上其它元素相互操作的能力。
- VML 影像很小而且比其它格式的影像下载快。
- 因为 VML 是以文字为基础的描述语言，所以 VML 影像与 XML 或 HTML 数据一起下载，并且完全整合于 Web 网页上的其它对象。对照于目前必须分开下载，而没有与其它网页对象整合的影像格式，您就知道 VML 的优点在哪了。

- 因为以标准为基础的设计方式，VML 应该在不同工具或平台上具备简化编辑以及使内部交换更容易的能力。

让我们实际看看 VML 提案里的一个 VML 程序代码例子，VML 允许文件作者描述一个基本的图形，比如说底下的箭头：



使用 VML 来描述该影像的程序代码如下所示：

```
<v:shapetype id="downArrow" coordsize="21600, 21600" adj="16200, 5400"  
  path="m0@0l@1@0@1, 0@2, 0@2@0, 21600@0, 10800, 21600xe">  
  <v:stroke joinstyle="miter"/>  
  <v:formulas>  
    <v:f eqn="sum #0 0 0"/>
```

```

<v:f eqn="sum #1 0 0"/>

<v:f eqn="sum height 0 #1"/>

<v:f eqn="sum 10800 0 #1"/>

<v:f eqn="sum width 0 #0"/>

<v:f eqn="prod @4 @3 10800"/>

<v:f eqn="sum width 0 @5"/>

</v:formulas>

<v:path textboxrect="@1, 0, @2, @6"/>

<v:handles>

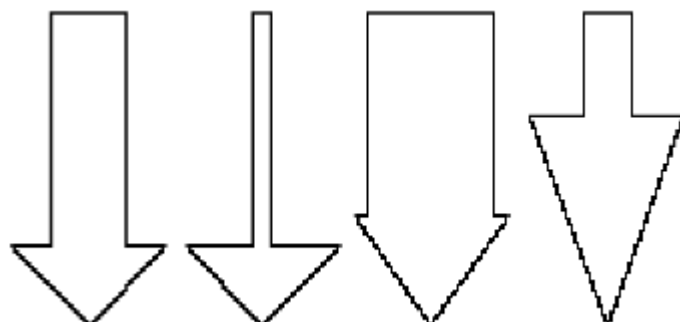
  <v:h position="#1, #0" xrange="0, 10800" yrange="0, 21600"/>

</v:handles>

</v:shapetype>

```

下面的箭头是经过改变的图形：



底下是 VML 程序代码：

```
<v:shape type="#downArrow"

    style='position: absolute; left: 77; top: 16;

    width: 64; height: 128' />

<v:shape type="#downArrow"

    style='position: absolute; left: 149; top: 16;

    width: 64; height: 128'

    adj=", 9450"/>

<v:shape type="#downArrow"

    style='position: absolute; left: 219; top: 16;

    width: 64; height: 128'

    adj="14175, 2025"/>

<v:shape type="#downArrow"

    style='position: absolute; left: 292; top: 16;

    width: 64; height: 128'

    adj="7088, 7425"/>
```

请注意：我们使用到原始图形的名称，但却可以产生不同的图形，那是因为应用 **CSS** 属性以及做了其它调整的原因。**VML** 组合其它的 **XML** 技术，比如 **SMIL** 或 **HTML+TIME**，可以产生小型的、整合过的、且以一般语法为基础的互动性的显示方式。

文件对象模型（Document Object Model）

就像本书先前所提过的，W3C 最近将通过 DOMLevel 1 的规格，即使您非常熟悉本书中所提过的 DOM，请再重新看看它能做到什么，还不能提供什么功能，以及将来会提供什么功能，对您将有莫大的帮助。

什么是 DOM，它从何处来？

文件对象模型（DOM）的发展是因为我们需要一个标准化的方法来做两件事。第一：定义一份文件的逻辑架构；第二：定义一份文件如何存取与使用。

文件专指 HTML 及 XML 文件，而 DOM 为这些文件提供了一个 API（application programming interface）。请试着回想，当 IE4 以及 Netscape4 出现时，它们都提供一种不同的实作技术，称为 Dynamic HTML（DHTML）。在它最基本的层级上，DHTML 提供文件作者可以存取各类不同 HTML 元素的能力，而且也经由属性、方法，及事件提供了处理这些元素的方法。这种技术最主要的问题在于：缺乏一个实作的标准，它会让 DHTML 的使用者产生跨浏览器兼容性的问题。而 DOM 就是被发展来解决这个兼容性问题的。请注意：虽然 DOM 受到 DHTML 的影响，但是它并不完全兼容 DHTML。比方说，事件（events）并没有包含于 Level 1 的规格中。DOM 之所以这样命名，是因为它提供文件对象一个架构以及行为模型，所以直觉上它是一个对象模型。

DOM 的文件架构

如上所述，DOM 提供了 HTML 或 XML 文件的架构模型，它指出一份文件逻辑的、树状的架构，但并未指出该架构如何被执行。让我们看看一份基本的 HTML 文件，并检视它的架构如何以 DOM 的方式呈现。

```
<HTML>

  <HEAD>

    <TITLE>DOM Test</TITLE>

  </HEAD>

  <BODY>

    <P>Some text</P>

    <DIV>

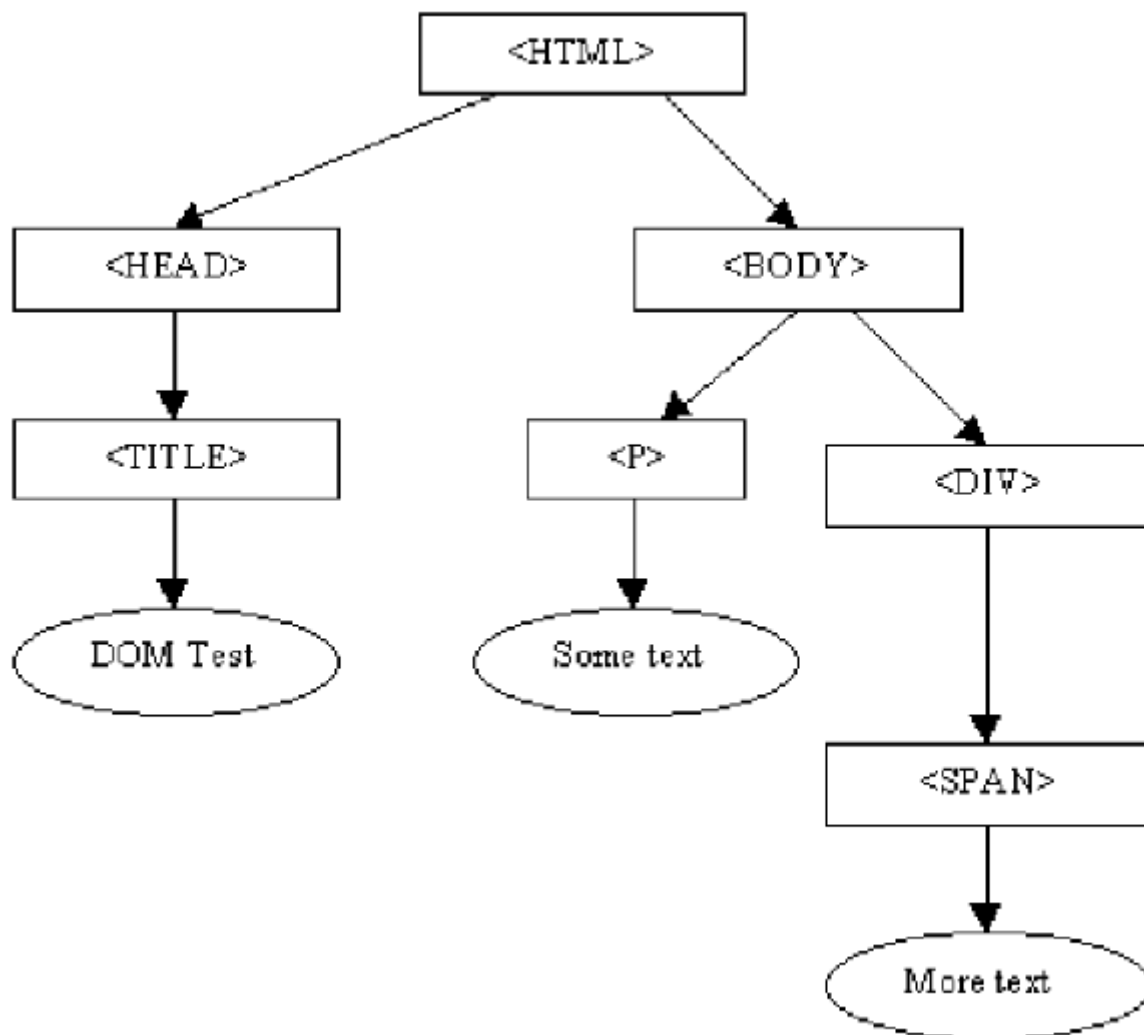
      <SPAN>More text</SPAN>

    </DIV>

  </BODY>

</HTML>
```

DOM 表示法图解如下：



请注意：该文件以阶层式架构呈现，假如文件中加入更多的元素，它们可以整合到相同的阶层里，而这个模型也可以应用到 XML 文件中。我们已经多次讨论到如何使用 XML 来显示资料，但以 DOM 的角度来说，一份文件并不是显示一个资料架构，从技术观点出发，DOM 视文件为个别对象的组成。

DOM 的文件行为 (Document Behavior)

就我们现在对 DOM 的了解而言，文件是由对象组成，而不仅仅是数据而已，也知道每个对象都有属性，比如一个 ID 以及方法。对于每个对象而言，DOM 代表：

- 操纵对象所使用的界面。
- 代表对象在文件其余部分中的关系，包括它的行为 (behavior) 及属性 (attributes)。
- 代表界面与对象间的合作

这种处理文件对象架构、语意以及行为的能力，提供了一个强大的机制来发展复杂且以文件为基础的应用程序。

DOM 无法做什么

要说清楚什么是 DOM，最好能指出 DOM 无法做什么以及与其它看起来相似的对象模型有何不同，相信对您的帮助是极大的。

1. 如上所述，DOM 在对象模型中代表对象的语意。然而我们必须了解——DOM 并没有定义 XML 语意或 HTML 文件，这些语意是透过那些语言的规格所定义的，而 DOM 只是简单的代表这些语意罢了。
2. DOM 并不是用来取代或与其它对象系统竞争的，例如 COM 或 CORBA。既然这些模型不是语言相依的，DOM 便是在 HTML 或 XML 中管理对象及界面的一个好方法。事实上，DOM 可以使用在以对象为基础的系统，如 COM 或 CORBA。
3. DOM 并没有定义一组数据架构，它只是定义对象与界面的一个模型。虽然一份文件可以数据元素的父子阶层关系来表示，但是 DOM 只是用来表示对象模型定义的逻辑关系罢了。请务必了解：一份文件是由 DOM 表示，而且可以由 HTML 或 XML 编辑。
4. 在本章前面，我们曾经讨论过如何在持续的储存机制中使用 XML 来架构数据。然而，这种持续并不是由 DOM 所提供的。与其说 DOM 定义对象可以由 HTML 或 XML 来决定如何显示，倒不如说 DOM 指定 HTML 与 XML 可以由对象显示，然后这些文件对象便可以在对象导向系统中使用。

DOM 的未来

就像我们在本节所讨论到的，DOM 是用来表示 HTML 或 XML 文件的对象及界面，这也是 DOM 在 Level 1 规格中的范围，但是 DOM 未来的版本计划包含了下列功能（请注意底下所提的大多数功能目前即使不用 DOM 也可以完成）：

- 将它当作一个标准的事件模型。目前我们是透过未标准化的界面来支持事件。
- 结构确认（**Schema validation**）。目前结构确认在处理器层级完成，而目前并没有任何规格来订定如何在 DOM 层级完成这项工作。
- 文件经由样式表（**style sheets**）翻译。目前使用的样式表是外部的 DOM。

如果想要得到更多 DOM 的信息，请参阅 W3C 的 DOM 网站，网址

为： <http://www.w3.org/TR/REC-DOM-Level-1>。

文件内容描述

[第 10 章](#) 详细说明了 XML 数据，其中一个用来描述 XML 文件架构的方法便是使用结构

（**Schema**）。事实上，XML 数据并不只是文件架构模型而已，目前涵盖的方面只是 XML 资料的一部分，而这些也是微软目前在 IE 上的实作。这个实作相当于在文件内容描述（**DCD**）规格中所描述的功能，DCD 是 XML 数据的一部分，它被设计来明确定义 XML 文件的架构与规则。

某些读者可能会有兴趣知道 DCD 描述的方法与资源描述结构（Resource Description

Framework, RDF）语法一致。RDF 在此处并没有提到，但您可以在下列网

址 <http://www.w3.org/TR/PR-rdf-syntax> 找到更多相关信息。

DCD 提供与传统 DTD 相同的功能，但也加入额外的功能，例如：支持基本的数据形态（**data types**）。您可能会注意到在下列的文字码中，DCD 的文字码看起来非常像我们在 [第 10 章](#) 建立的结构（**Schema**），即使语法有一点不同。

```
<DCD>

  <AttributeDef name="language" dt:type="enumeration"
    dt:values="Western Greek Latin Universal"/>

  <AttributeDef name="encrypted"/>

  <AttributeDef name="priority" dt:type="enumeration"
    dt:values="NORMAL LOW HIGH"/>

  <AttributeDef name="hidden" default="true"/>

  <ElementDef name="TO" content="textOnly"/>

  <ElementDef name="FROM" content="textOnly"/>

  <ElementDef name="CC" content="textOnly"/>

  <ElementDef name="BCC" content="mixed">
```

```
<attribute type="hidden" required="yes"/>

</ElementDef>

<ElementDef name="SUBJECT" content="textOnly"/>

<ElementDef name="BODY" content="textOnly"/>

</DCD>
```

目前 DCD 的规格已经在 W3C 的提案阶段，而本节的目的也不是要详述 DCD 的语汇

（**vocabulary**）及用法，因为在第 10 章已经讲过相似的功能。而 DCD 最终也会变成描述 XML

文件架构与内容的一种选择。想要取得更多关于 DCD 信息，请参

阅 <http://www.w3.org/TR/NOTE-dcd>。

跨平台的 XML

困扰 XML 开发人员的一般问题（尤其是以 Web 为基础（**Web-based**）的开发人员）就是 XML

提供了多少跨平台与跨浏览器的兼容性。假如您想要使用 XML 来寻求 Web 的解决方案，无拥

置疑地将面对浏览器兼容性的问题，就像 XML 一样，兼容性问题的解答有简单的也有复杂的。

简单的答案是：根据 W3C 公布的标准，跨平台的兼容性问题会发生在任何实作 XML 的平台上。

比如说，IE 中的 Msxml 兼容于 XML 1.0，如果 Netscape 处理器也是兼容于 XML 1.0，则在其

中一个浏览器上编辑的文件应该可以在另一个浏览器中解析出来。但其中的问题是：在 W3C 正

式提出建议方案之前，有太多以 XML 为基础（**XML-based**）的技术已经应用在目前的浏览器中。

这是因为软件公司经常跑在 W3C 之前。而另一个困难就如同科技的进步一样，总是与公布的规格不兼容。

当不同的软件公司站在软件立场来介绍特性与功能，却不考虑其它平台可否适用时，会让这样的情形变得更复杂。比如说，本章稍早提过 IE 的 XML 数据岛（data island）功能，它并不是 XML 规格的一部分，也不是 DOM 规格的一部分。让我们看看这代表什么意思，底下的 XML 文字码是一份格式正确的（well-formed）XML 1.0 文件：

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

<INVENTORY>

  <ITEM>

    <NAME>Brandenburg Concerto No. 2</NAME>

    <AUTHOR>Bach, J. S. </AUTHOR>

    <PRICE>17.95</PRICE>

  </ITEM>

  <ITEM>

    <NAME>Piano Sonata in A</NAME>

    <AUTHOR>Mozart, W. A. </AUTHOR>

    <PRICE>16.95</PRICE>
```



```
</ITEM>
```

```
</INVENTORY>
```

本文件可以被懂得 **XML 1.0** 文件的任何浏览器或软件解析。实际上，该文件如何被加载处理器中，以及在解析时对这些数据做什么，完全取决于应用程序，但是文件本身应该能跨平台。现在我们用相同的 **XML** 文件，以数据岛方式应用在 **HTML** 网页中，文字码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>Music Inventory</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<XML ID=music>
```

```
<INVENTORY>
```

```
<ITEM>
```

```
<NAME>Brandenburg Concerto No. 2</NAME>
```

```
<AUTHOR>Bach, J. S.</AUTHOR>
```

<PRICE>17.95</PRICE>

</ITEM>

<ITEM>

<NAME>Piano Sonata in A</NAME>

<AUTHOR>Mozart, W. A. </AUTHOR>

<PRICE>16.95</PRICE>

</ITEM>

</INVENTORY>

</XML>

<TABLE DATASRC=#music CELSPACING="10" CELLPADDING="2">

<THEAD STYLE="font-weight:bold">

<TD>Name</TD>

<TD>Author</TD>

<TD>Price</TD>

</THEAD>

<TR>

```
<TD><DIV DATAFLD="NAME"></TD>

<TD><DIV DATAFLD="AUTHOR"></TD>

<TD><DIV DATAFLD="PRICE"></TD>

</TR>

</TABLE>

</BODY>

</HTML>
```

目前，此网页只有在 IE5 中才看得到，这表示数据岛以及相似的技术对于 XML 以及使用者都不好吗？未必见得，它只不过表示软件开发人员经常走在这些将成为标准的解决方案前面，比如 XML。事实上，这些特殊技术经常发生，比如资料岛，已包含在正式建议方案下一个版本的格式中。

这样的讯息透露出当您建立以 XML 为基础（XML-based）的应用程序或 Web 网页时，明确地了解您使用的技术为何，以及谁是您想要的支持者是很重要的。

结论

XML 是所有 Web 问题的解决方案吗？或者只是夸大其词？我们希望读完本书后，您将了解两者都不是，因为没有一种软件解决方案可以解决所有的问题。然而，到目前为止观察使用 XML 所发展的解决方案，以及许多企业都支持这项技术的现象，很明显的可以知道 XML 不会只是一

时的流行。我们可以说 WWW 的现象是由 HTML 建立的，而要使 XML 像 HTML 一样成功，有几件事必须要整个信息产业来共襄盛举。

- 人们以及组织团体必须看到 XML 的价值。就如同 HTML 被接受是因为提供了标准格式，在因特网上显示丰富的内容，XML 应该被承认是一种用来结构化与传递数据很有价值的工具，不仅仅是在 Web 上，在其它的应用上，也可以运作的很好。
- 工具必须要能有效的允许人们存取这些技术。这些工具可以包括任何从 XML 编辑软件到软件开发工具（Software Development Kit，简称 SDK）以及类似本书的书籍。任何新的技术要被大家所接受，它必须要被介绍、教导以及支持。
- 标准化必须继续进行。选择 XML 而不选择其它数据格式和解决方案的原因之一，是因为它以 W3C 所发表的标准为基础，而不是由任何公司或组织所控制。标准化可以确保每个人，包括个别的公司团体可以存取该语言，并加速它的发展与实作。XML 要能成功，新的 XML 技术必须继续通过标准化的程序，而应用者也必须等待标准。

这本书的目的在于提供您发展以 XML 为基础的应用程序或网站解决方案所需的信息，而对于任何技术取得的最好方法就是像您我一样，直接使用它吧！

Appendix A XML 对象模型

[第 5 章](#) 谈论了如何使用 **Script** 在 **HTML** 网页中与组件互相整合，以及如何与 **XML** 对象模型合作。这个附录将详细地探究 **XML** 对象模型，包括组成模型的对象、它们的属性与方法，和它们如何被应用在应用程序中。

对象模型结构

正如同我们在 [第 3 章](#) 所讨论的，对象是一种树状结构的组织。树的主干便是 **Document** 对象，其它的对象便由此对象分支而来。

XML 对象模型由四种基本的对象所组成：

- **document** 对象—XML 数据来源
- **node** 对象—一个父节点或其中的一个子节点
- **nodeList** 对象—兄弟节点的清单
- **parseError** 对象—一个无内容的对象，用来接收解析错误的讯息

图 A-1 显示了这些对象间的关系。

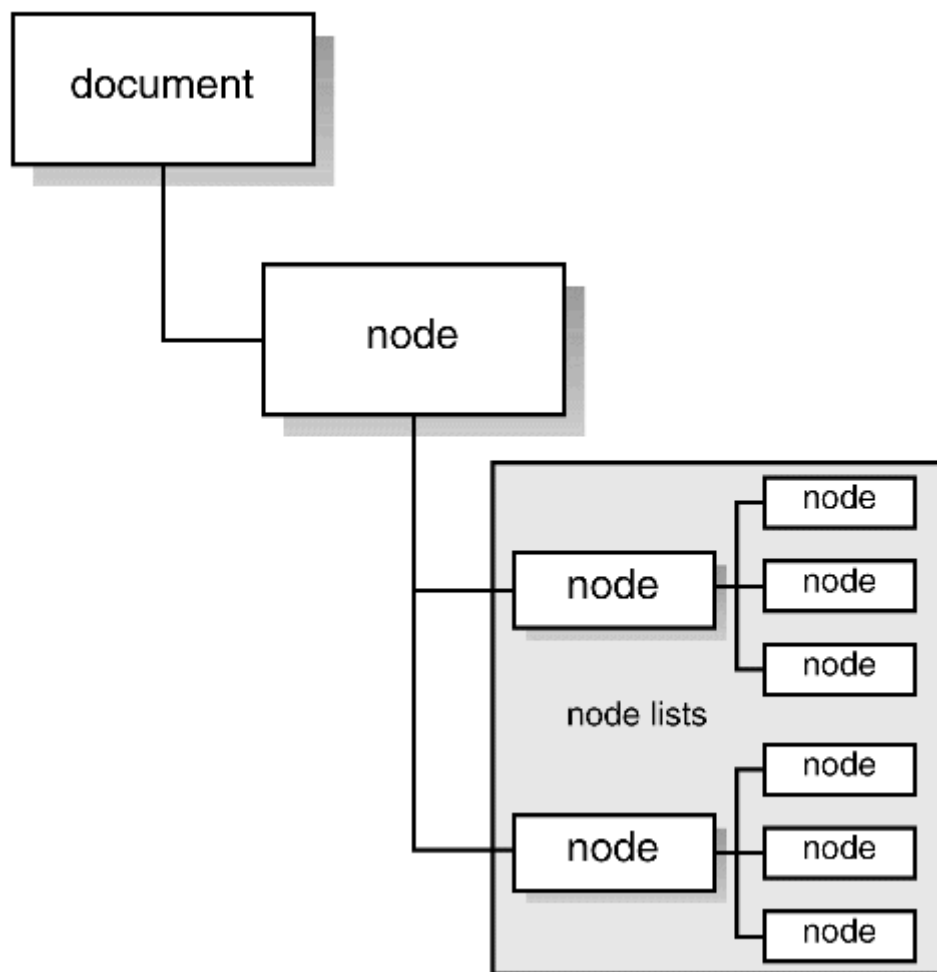


图 A-1: XML 对象模型

在对象模型中的每个对象拥有特定的属性或方法，或两者都有。若使用 **Script** 程序代码，内容作者可以直接地使用这些属性和方法来取得信息并运用 **XML** 数据。这个附录将完整地介绍 **XML** 对象模型，并检视这些对象和相关的属性与方法。

既然本书是讨论如何运用 **XML** 的，就必须深入了解对象模型的运作。稍后在这附录中，有一些如何在对象模型中使用对象的范例。要使用这些范例，您将需要列在 **A-1** 中的文字码，也可以在随书光盘中找到它（Appxa\LastA_1.xml）。

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

<!DOCTYPE EMAIL SYSTEM "LstA_2.dtd">

<EMAIL PRIORITY="HIGH">

  <TO>Jodie@msn.com</TO>

  <FROM>Bill@msn.com</FROM>

  <CC>Philip@msn.com</CC>

  <BCC>Naomi@msn.com</BCC>

  <SUBJECT>My document is a tree.</SUBJECT>

  <BODY>This is an example of a tree structure.</BODY>

</EMAIL>
```

文字码 A-1

您可以从文件类型宣告中看到，这份文件使用了 **DTD**。这份 **DTD** 文件如文字码 **A-2** 所示，您也可以在随书光盘中的 AppxA\LstA_2.dtd 找到它。

```
<!-- This is an XML document that could be used as an email template. -->
```

```
<!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?)>
```

```
<!ATTLIST EMAIL
```

```
    LANGUAGE (Western|Greek|Latin|Universal) "Western"
```

```
    ENCRYPTED CDATA #IMPLIED
```

```
    PRIORITY (NORMAL|LOW|HIGH) "NORMAL">
```

```
<!ELEMENT TO (#PCDATA)>
```

```
<!ELEMENT FROM (#PCDATA)>
```

```
<!ELEMENT CC (#PCDATA)>
```

```
<!ELEMENT BCC (#PCDATA)>
```

```
<!ATTLIST BCC
```

```
    HIDDEN CDATA #FIXED "TRUE">
```

```
<!ELEMENT SUBJECT (#PCDATA)>
```

```
<!ELEMENT BODY (#PCDATA)>
```

文字码 A-2

最后，XML 文件将透过 HTML 网页来显示，下面便是 HTML 的文字码（在随书光盘

AppxA\LstA_3.htm 中）。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<HTML>
```



```
<HEAD>
```

```
<SCRIPT LANGUAGE="JavaScrip" _ FOR=window EVENT=onload>
```

```
    showMe();
```

```
</SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">
```

```
    var xmlDoc = new ActiveXObject("microsoft.xmldom");
```

```
    xmlDoc.load("LstA_1.xml");
```

```
    function showMe()
```

```
    {
```

```
        // Insert sample code here.
```

```
    }
```

```
</SCRIPT>
```

```
<TITLE>Code Listing A-3</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
</BODY>
```

```
</HTML>
```

文字码 A-3

请注意在 A-3 文字码中，**showMe** 函式并没有包含程序代码。因为要插入什么程序代码是由您来决定的，这个程序片段只是要让您了解 **showMe** 函式的功用。

Note

使用附录中的范例时，您可以在 **LstA_3.htm** 的 **showMe** 函式中输入片断程序代码，也可以使用随书光盘中 **AppxA** 目录中的 **HTML** 档案。

Document 物件

如同前面所提到的，这 **Document** 对象代表这份文件的数据来源。这个对象模型允许文件作者使用任何支持 **XML** 处理器的浏览器或应用程序，透过 **script** 程序代码来加载 **XML** 文件。

举例来说，一个文件能藉由建立一个 **ActiveX control** 的实例，并且呼叫 **load** 方法（稍后讨论）来加载文件，如同范例中的 **HTML** 码：

```
var xmlDoc = new ActiveXObject("microsoft.xmldom");
```

```
xmlDoc.load("LstA_1.xml");
```

加载也可借着使用 **Java applet** 来完成，如下所示：

```
<SCRIPT>

    xmldso.load("LstA_1.xml");

</SCRIPT>

<BODY>

    <APPLET CODE=com.ms.xml.dso.XMLDSO.class

        WIDTH=100% HEIGHT=0 ID=xmldso MAYSCRIPT=true>

    </APPLET>

</BODY>
```

因为对象模型预期将会使用 **load** 方法，因此，程序代码与控件被加载后的程序代码是相同的。

重要的是，一旦数据来源被建立后，对象模型便会提供一致的方式来浏览及处理 **XML** 资料。

Note

Msxml 延伸基本的 **Document** 对象模型（DOM），来包含 **XML** 特定的界面。讨论 **DOM** 的概念

超出本书的范围。所以，下面的单元只把焦点集中在 **XML DOM** 的一般性使用。

Document 对象属性

以下所列是可用的 Document 对象属性：

- `async`
- `attributes`
- `childNodes`
- `doctype`
- `documentElement`
- `firstChild`
- `implementation`
- `lastChild`

- nextSibling
- nodeName
- nodeType
- nodeValue
- ondataavailable
- onreadystatechange
- ownerDocument
- parentNode
- parseError
- previousSibling

- readyState
- url
- validateOnParse
- xml

async 属性

async 属性表示是否允许异步的下载。

基本语法	<pre>boolValue = XMLDocument.async;XMLDocument.async = boolValue;</pre>
说明	布尔值是可擦写的（read/write），如果准许异步下载，值为 True；反之则为 False。

使用范例

以下的范例请参照随书光盘的 AppxA\LstA_4.htm:

```
xmlDoc.async = "false";  
  
alert(xmlDoc.async);
```

attribute 属性

传回目前节点的属性列表。

基本语法	<code>objAttributeList = xmlNode.attributes;</code>
说明	传回一个物件。如果此节点不能包含属性，则传回空值。

使用范例

以下的范例请参照随书光盘的 `AppxA\LstA_5.htm`：

```
objAttList = xmlDoc.documentElement.attributes;  
  
alert(objAttList);
```

Note

例子以[object]为结果传回。这是由于被传回的对象，若不使用其它的对象属性，就不能以文字来表示，在本附录中包含数个这样的例子。尽管很多对象属性本身似乎不是很有用，但您将会在附录中看到如何组合属性和方法得到您要的结果。

childNodes 属性

传回一个节点清单，包含该节点所有可用的子节点。

基本语法	objNodeList=node.childNodes;
说明	传回一个物件。假如这节点没有子节点，传回 null。

使用范例

以下的例子请参照随书光盘中的 AppxA\LstA_6.htm:

```
objNodeList = xmlDoc.childNodes;

alert(objNodeList);
```

doctype 属性

传回文件型态节点，包含目前文件的 DTD。这节点是一般的文件型态宣告，例如，节点

<!DOCTYPE EMAIL SYSTEM "LstA_2.dtd">，名为 EMAIL 的节点物件会被传回。

基本语法	objDocType=xmlDocument.doctype;
说明	传回一个对象，这个属性是只读的。假如这文件不包含 DTD，会传回 null。

使用范例

以下的例子请参照随书光盘 AppxA\LstA_7.htm:

```
objDocType = xmlDoc.doctype;

alert(objDocType.nodeName);
```

documentElement 属性

确认 XML 文件的根（Root）节点。

基本语法	objDoc=xmlDocument.documentElement;
说明	回一个在单一根文件元素中包含数据的对象。此属性可读/写，如果文件中不包含根节点，将传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_8.htm:

```
objDocRoot = xmlDoc.documentElement;  
  
alert(objDocRoot);
```

firstChild 属性

确认在目前节点中的第一个子元素。

基本语法	objFirstChild = xmlDocNode.firstChild ;
说明	此属性只读且会传回一对象，如果节点中没有包含第一个子元素，将传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_9.htm:

```
objFirstChild = xmlDoc.documentElement.firstChild;  
  
alert(objFirstChild);
```

implementation 属性

DOM 应用程序能使用其它实作中的对象。implementation 属性确认目前 XML 文件的 DOMImplementation 对象。

基本语法	objImplementation = xmlDocument.implementation;
说明	此属性只读且传回一个对象。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_10.htm:

```
objImp = xmlDoc.implementation;

alert(objImp);
```

lastChild 属性

确认目前节点中最后的子元素。

基本语法	objLastChild = xmlDocNode.lastChild;
说明	此属性只读且传回一个对象。如果节点中没有包含最后子元素，将传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_11.htm:

```
objLastChild = xmlDoc.documentElement.lastChild;

alert(objLastChild);
```

nextSibling 属性

在目前文件节点的子节点列表中传回下一个兄弟节点。

基本语法	<pre>objNextSibling = xmlDocNode.nextSibling;</pre>
说明	此属性是只读且传回一个对象。如果节点中没有包含其它的相关节点，会传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_12.htm:

```
objSibling = xmlDoc.documentElement.childNodes.item(1) .nextSibling;

alert(objSibling);
```

nodeName 属性

传回代表目前节点名称的字符串。

基本语法	strNodeName = xmlDocNode.nodeName ;
说明	传回一个字符串。这个属性是只读的，传回元素名称、属性或实体参照。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_13.htm:

```
strNodeName = xmlDoc.documentElement.nodeName;

alert(strNodeName);
```

nodeType 属性

辨识节点的 DOM 型态。

基本语法	numNodeType = xmlDocNode.nodeType ;
说明	此属性只读且传回一个数值。有效的数值符合以下的型别：
	1-ELEMENT

	2-ATTRIBUTE
	3-TEXT
	4-CDATA
	5-ENTITY REFERENCE
	6-ENTITY
	7-PI (processing instruction)
	8-COMMENT
	9-DOCUMENT
	10-DOCUMENT TYPE
	11-DOCUMENT FRAGMENT
	12-NOTATION

Note

上述的数据型态将在 [附录 B](#) 中有更深入的介绍。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_14.htm:

```
numNodeType = xmlDoc.documentElement.nodeType;

alert(numNodeType);
```

nodeValue 属性

传回指定节点相关的文字。这并非一个元素中数据的值，而是与一个节点相关且未解析的文字，
就像一个属性或者一个处理指令。

基 本 语 法	<pre>varNodeValue = xmlDocNode.nodeValue;</pre>
说 明	<p>传回的文字代表以节点的 nodeType 属性为主的型态值。（请参考附录中的 nodeType 属性。）</p> <p>因为节点型态可能是几种数据型态中的一种，传回值也因此有差异。传回 null 的节点型态有：DOCUMENT、ELEMENT、DOCUMENT TYPE、DOCUMENT FRAGMENT、ENTITY、ENTITY REFERENCE，和 NOTATION。此属性可擦写。</p>

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_15.htm:

```
varNodeValue = xmlDoc.documentElement.nodeValue;

alert(varNodeValue);
```

ondataavailable 属性

指定一个事件来处理 **ondataavailable** 事件。（更多关于 **ondataavailable** 事件的信息，请参阅附录中<Document 对象事件>的部分）。

基本语法	xmlDocNode.ondataavailable = value;
说明	此属性是唯写，允许文件作者一旦数据为可用，即可尽快的使用数据来运作。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_16.htm:

```
xmlDoc.ondataavailable = alert("Data is now available.");
```

onreadystatechange 属性

指定一个事件来处理 `onreadystatechange` 事件。这个事件能辨识 `readyState` 属性的改变。（更多有关 `onreadystatechange` 事件的信息，请参阅附录中 [<Document 对象事件>](#) 的部分。）

基本语法	<code>xmlDocNode.onreadystatechange = value;</code>
说明	此属性是唯写的，允许文件作者指定当 <code>readyState</code> 属性改变时呼叫事件。

使用范例

以下的范例请参照随书光盘中的 `AppxA\LstA_17.htm`：

```
xmlDoc.onreadystatechange =  
  
alert("The readyState property has changed.");
```

ownerDocument 属性

传回文件的根节点，包含目前节点。

基本语法	<code>objOwnerDoc = xmlDocDocument.ownerDocument;</code>
说明	此属性是只读的，传回一个包含文件根节点的对象，包含特定的节点。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_18.htm:

```
objOwnerDoc = xmlDoc.childNodes.item(2).ownerDocument;

alert(objOwnerDoc);
```

parentNode 属性

传回目前节点的父节点。只能应用在有父节点的节点中。

基本语法	<pre>objParentNode = xmlDocumentNode.parentNode;</pre>
说明	此属性是只读的，传回包含指定节点的父节点对象。如果此节点不存在于文件树中，将传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_19.htm:

```
objParentNode = xmlDoc.childNodes.item(1).parentNode;

alert(objParentNode);
```

parseError 属性

传回一个 DOM 解析错误对象，此对象描述最后解析错误的讯息。

基本语法	objParseErr = xmlDocument.parseError;
说明	此属性是只读的。如果没有错误发生，将传回 0。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_20.htm:

```
objParseErr = xmlDoc.parseError;

alert(objParseErr);
```

previousSibling 属性

传回目前节点之前的兄弟节点。

基本语法	objPrevSibling = xmlDocument.previousSibling;
说明	传回一个对象，这个属性是只读的。若该节点没有包含前面的兄弟节点，会传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_21.htm:

```
objPrevSibling =  
  
    xmlDoc.documentElement.childNodes.item(3).previousSibling  
  
alert(objPrevSibling);
```

readyState 属性

传回 XML 文件资料的目前状况。

基本语法	<pre>intState = xmlDoc.readyState;</pre>
说明	<div><div>这个属性是只读的，传回值有以下的可能：</div><div>0-UNINITIALIZED：XML 对象被产生，但没有任何文件被加载。</div><div>1-LOADING：加载程序进行中，但文件尚未开始解析。</div><div>2-LOADED：部分的文件已经加载且进行解析，但对象模型尚未生效。</div><div>3-INTERACTIVE：仅对已加载的部分文件有效，在此情况下，对象模型是有效但只读的。</div><div>4-COMPLETED：文件已完全加载，代表加载成功。</div></div>

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_22.htm:

```
alert("The readyState property is " + xmlDoc.readyState);
```

url 属性

传回最近一次加载 XML 文件的 URL。

基本 语法	<pre>strDocUrl = xmlDocument.url;</pre>
说明	这个属性是只读的，传回最近一次加载成功文件的 URL，若文件仅存在主存储器中（表示该文件并非由外部档案加载），则传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_23.htm:

```
alert(xmlDoc.url);
```

validateOnParse 属性

告诉解析器文件是否有效。

基本	<code>boolValidate = xmlDocument.validateOnParse;</code>
语法	<code>xmlDocument.validateOnParse = boolValidate;</code>
说明	此属性是可擦写的。如果传回值为 <code>true</code> ，表示文件被解析时被确认是有效的。如果传回 <code>false</code> ，表示文件是无效的，并被认为是标准格式的（well-formed）文件。

使用范例

以下的范例请参照随书光盘中的 `AppxA\LstA_24.htm`：

```
xmlDoc.validateOnParse = true;

alert(xmlDoc.validateOnParse);
```

xml 属性

传回指定节点的 XML 描述和所有的子节点。

基本语法	<code>xmlValue = xmlDocumentNode.xml;</code>
说明	此属性是只读的。

使用范例

以下的范例请参照随书光盘中的 `AppxA\LstA_25.htm`:

```
xmlValue = xmlDoc.documentElement.xml;  
  
alert(xmlValue);
```

Document 对象方法

以下列出关于 **Document** 对象可用的方法。

- `abort`
- `appendChild`
- `cloneNode`

- `createAttribute`
- `createCDATASection`
- `createComment`
- `createDocumentFragment`
- `createElement`
- `createEntityReference`
- `createNode`
- `createProcessingInstruction`
- `createTextNode`
- `getElementsByTagName`

- hasChildNodes
- insertBefore
- load
- loadXML
- nodeFromID
- parsed
- removeChild
- replaceChild
- selectNodes
- selectSingleNode

- transformNode

abort 方法

abort 方法取消一个进行中的异步下载。

基本语法	<code>xmlDocument.abort();</code>
说明	如果这个方法在异步下载时被呼叫，所有的解析动作会停止，而且在内存中的文件会被释放。

AppendChild 方法

加上一个节点当作指定节点最后的子节点。

基本语法	<code>xmlDocumentNode.appendChild(newChild);</code>
说明	<code>newChild</code> 是附加子节点的地址。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_26.htm:

```
docObj = xmlDoc.documentElement;

alert(docObj.xml);

objNewNode = docObj.appendChild(xmlDoc.documentElement. firstChild);

alert(docObj.xml);
```

cloneNode 方法

建立指定节点的复制。

基本 语法	<code>xmlDocumentNode.cloneNode(deep);</code>
说明	deep 是一个布尔值。如果为 true，此节点会复制以指定节点发展出去的所有节点。如果是 false，只有指定的节点和它的属性被复制。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_27.htm:

```
currNode = xmlDoc.documentElement.childNodes.item(1);
```

```
objClonedNode = currNode.cloneNode(1);  
  
alert(objClonedNode.xml);
```

createAttribute 方法

建立一个指定名称的属性。

基本语法	xmlDocument.createAttribute(name);
说明	name 是被建立属性的名称。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_28.htm:

```
objNewAtt = xmlDoc.createAttribute("encryption");  
  
alert(objNewAtt.xml);
```

createCDATASection 方法

建立一个包含特定数据的 CDATA。

基本语法	<code>xmlDocument.createCDATASection(data);</code>
说明	date 是一个字符串，且包含了被置放在 CDATA 的资料。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_29.htm:

```
objNewCDATA = xmlDoc.createCDATASection("This is a CDATA Section");
alert(objNewCDATA.xml);
```

createComment 方法

建立一个包含指定数据的批注。

基本语法	<code>xmlDocument.createComment(data);</code>
说明	data 是一个字符串，且包含了被置放在批注的资料。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_30.htm:

```
objNewComment = xmlDoc.createComment("This is a comment");  
  
alert(objNewComment.xml);
```

createDocumentFragment 方法

建立一个空的文件片断对象。

基本 语法	<code>xmlDocument.createDocumentFragment();</code>
说明	一个新的文件片断被建立，但没有加到文件树中。要加入片断到文件树中，必须使用插入方法，例如 <code>insertBefore</code> 、 <code>replaceChild</code> 或 <code>appendChild</code> 。

使用范例

以下的范例请参照随书光盘中的 `AppxA\LstA_31.htm`：

```
objNewFragment = xmlDoc.createDocumentFragment();  
  
alert(objNewFragment.xml);
```

createElement 方法

建立一个指定名称的元素。

基本语法	<code>xmlDocument.createElement(tagName);</code>
说法	tagName 是一个区分大小写的字符串来指定新元素名称。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_32.htm:

```
objNewElement = xmlDoc.createElement("T0");  
  
alert(objNewElement.xml);
```

createEntityReference 方法

建立一个参照到指定名称的实体。

基 本 语 法	<code>xmlDocument.createEntityReference(name);</code>
说	name 是一个区分大小写的字符串，来指定新实体参照的名称。一个新的实体参照被建立，

明	但是并没有被加到文件树中。若要将实体参照加到文件树中，必须使用一种插入方法，例如：insertBefore，replaceChild，或 appendChild。
---	---

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_33.htm:

```
objNewER = xmlDoc.createEntityReference("eRef");  
  
alert(objNewER.xml);
```

createNode 方法

建立一个指定型态、名称，及命名空间的新节点。

基 本 语 法	<code>xmlDocument.createNode(type, name, nameSpaceURI);</code>
说 明	type 用来确认要被建立的节点型态， name 是一个字符串来确认新节点的名称，命名空间的前缀则是选择性的。

	<p><code>nameSpaceURI</code> 是一个定义命名空间 <code>URI</code> 的字符串。如果前缀被包含在名称参数中，此节点会在 <code>nameSpaceURI</code> 的内文中以指定的前缀建立。 如果不包含前缀，指定的命名空间会被视为预设的命名空间。</p>
--	--

使用范例

以下的范例请参照随书光盘中的 `AppxA\LstA_34.htm`:

```
objNewNode = xmlDoc.createTextNode(1, "T0", "");  
  
alert(objNewNode.xml);
```

createProcessingInstruction 方法

建立一个新的处理指令，包含了指定的目标和数据。

基 本 语 法	<pre>xmlDocument.createProcessingInstruction(target, data);</pre>
说	<p><code>target</code> 是表示目标、名称或处理指令的字符串。<code>Data</code> 是表示处理指令的值。一个新的处理</p>

明	指令被建立，但是并没有加到文件树中。要把处理指令加到文件树中，必须使用插入方法，例如：insertBefore、replaceChild，或是 appendChild。
---	--

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_35.htm:

```
objNewPI =  
  
xmlDoc.createProcessingInstruction( 'XML' , 'version="1.0"' );  
  
alert (objNewPI.xml);
```

createTextNode 方法

建立一个新的 text 节点，并包含指定的数据。

基 本 语 法	<code>xmlDocument.createTextNode(data);</code>
说 明	data 是一个代表新 text 节点的字符串。一个新的 text 节点被建立，但是没有加到文件树中。若要将节点加到文件树中，必须使用插入方法，例如：insertBefore，replaceChild

	或 appendChild。
--	----------------

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_36.htm:

```
objNewTextNode = xmlDoc.createTextNode("This is a text node.");  
  
alert(objNewTextNode.xml);
```

getElementsByTagName 方法

传回指定名称的元素集合。

基本语法	<code>objNodeList = xmlDocument.getElementsByTagName(tagname);</code>
说明	tagname 是一个字符串，代表找到的元素卷标名称。使用 tagname "*" 传回文件中所有找到的元素。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_37.htm:

```
objNodeList = xmlDoc.getElementsByTagName("*");  
  
alert(objNodeList.item(1).xml);
```

hasChildNodes 方法

如果指定的节点有一个或更多子节点，传回值为 **true**。

基本语法	boolValue = xmlDocumentNode.hasChildNodes() ;
说明	如果此节点有子节点传回值为 true，否则传回 false 值。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_38.htm:

```
boolValue = xmlDoc.documentElement.hasChildNodes();  
  
alert(boolValue);
```

insertBefore 方法

在指定的节点前插入一个子节点。

基 本 语 法	<pre>objDocumentNode = xmlDocNode.insertBefore(newChild, refChild);</pre>
说 明	<p>newChild 是一个包含新子节点地址的对象，refChild 是参照节点的地址。新子节点被插到参照节点之前。如果 refChild 参数没有包含在内，新的子节点会被插到子节点列表的末端。</p>

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_39.htm:

```
objRefNode = xmlDoc.documentElement;

alert(xmlDoc.xml);

objNewNode = xmlDoc.createComment("This is a comment");

xmlDoc.insertBefore(objNewNode, objRefNode);

alert(xmlDoc.xml);
```

load 方法

表示从指定位置加载的文件。

基本语法	<pre>boolValue = xmlDoc.load(url);</pre>
说明	url 包含要被加载档案的 URL 的字符串。假如文件加载成功，传回值即为 true。若加载失败，传回值为 false。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_40.htm:

```
boolValue = xmlDoc.load("LstA_1.xml");  
  
alert(boolValue);
```

loadXML 方法

加载一个 XML 文件或字符串的片断。

基本语法	<pre>boolValue = xmlDoc.loadXML(xmlString);</pre>
说明	<p>xmlString 是包含 XML 文字码的字符串。</p> <p>这个字符串可以包含整个文件或者只是一个文件片断。如果文件加载成功，传回值为 true。</p>

	假如加载失败，传回值则是 false 。
--	-----------------------------

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_41.htm:

```
xmlString = "<GREETING><MESSAGE>Hello!</MESSAGE></GREETING>";  
  
boolValue = xmlDoc.loadXML(xmlString);  
  
alert(boolValue);
```

nodeFromID 方法

传回节点 ID 符合指定值的节点。

基本 语法	<code>xmlDocumentNode = xmlDocument.nodeFromID(idString);</code>
说明	idString 是一个包含 ID 值的字符串。符合的节点必定是 ID 型态。若符合，将传回一个对象；若操作失败，则传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_42.htm:

```
objDocumentNode = xmlDoc.nodeFromID("T0");  
  
alert(objDocumentNode);
```

parsed 方法

会验证该指定的节点(node)及其衍生的子节点(descendants)是否已被解析过。

基本语 法	<pre>boolValue = xmlDocNode.parsed();</pre>
说 明	如果全部的节点都被解析过了, 则传回值为 true; 如果有任何一个节点尚未被解析, 传回值则为 false。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_43.htm:

```
currNode = xmlDoc.documentElement.childNodes.item(0);  
  
boolValue = currNode.parsed();  
  
alert(boolValue);
```


removeChild 方法

会将指定的节点从节点清单中移除。

基本语法	<code>objDocumentNode = xmlDocumentNode.removeChild(oldChild);</code>
说 明	oldChild 为一个包含要被移除的节点对象。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_44.htm:

```
objRemoveNode = xmlDoc.documentElement.childNodes.item(3);  
  
alert(xmlDoc.xml);  
  
xmlDoc.documentElement.removeChild(objRemoveNode);  
  
alert(xmlDoc.xml);
```

replaceChild 方法

置换指定的旧子节点为提供的新子节点。

基本	<code>objDocumentNode = xmlDocumentNode.replaceChild(newChild,oldChild);</code>
----	---

语法	
说明	<p><code>newChild</code> 为包含新子节点的对象。如果此参数为 <code>null</code>，则此旧子节点会被移除而不会被取代。<code>oldChild</code> 为包含旧子节点的对象。</p>

使用范例

以下的范例请参照随书光盘中的 `AppxA\LstA_45.htm`:

```
objOldNode = xmlDoc.documentElement.childNodes.item(3);  
  
objNewNode = xmlDoc.createComment("I've replaced the BCC element.");  
  
alert(xmlDoc.xml);  
  
xmlDoc.documentElement.replaceChild(objNewNode, objOldNode);  
  
alert(xmlDoc.xml);
```

selectNodes 方法

传回所有符合提供样式(`pattern`)的节点。

基本语法	<pre>objDocumentNodeList = xmlDocNode.selectNodes(patternString);</pre>
说明	<p><code>patternString</code> 为一包含 XSL 样式的字符串。此方法会传回节点清单对象，包含符合样式</p>

明	的节点。如果没有符合的节点，则传回空的清单列表。
---	--------------------------

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_46.htm:

```
objNodeList=xmlDoc.selectNodes ("/") ;  
  
alert (objNodeList.item(0).xml) ;
```

selectSingleNode 方法

传回第一个符合样式的节点。

基本 语法	<code>objDocumentNode = xmlDocDocumentNode.selectSingleNode(patternString);</code>
说明	<code>patternString</code> 为一包含 XSL 样式的字符串。此方法会传回第一个符合的节点对象，如果没有符合的节点，则传回 <code>null</code> 。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_47.htm:

```
objNode = xmlDoc.selectSingleNode("EMAIL/BCC");  
  
alert(objNode.xml);
```

transformNode 方法

使用提供的样式表来处理该节点及其子节点。

基本 语法	<pre>strTransformedDocument = xmlDocNode.transformNode(stylesheet);</pre>
说 明	stylesheet 为一 XML 文件或是片断包含负责节点转换工作的 XSL 元素。此方法会传回一包含转换结果的字符串。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_48.htm:

```
var style = new ActiveXObject("Microsoft.XMLDOM");  
  
style.load("LstA_49.xsl");  
  
strTransform = xmlDoc.transformNode(style.documentElement);  
  
alert(strTransform);
```

Note

想更进一步了解有关 XSL 的细节，请参阅 [第 8 章](#) 及 [第 9 章](#)。

Document 对象的事件

下面列出来的是 Document 对象可用的事件：

- Ondataavailable

- Onreadystatechange

Ondataavailable 事件

此事件会在 XML 文件有效时被触发。

基本语法	<p>此一事件有下面三种处理方式</p> <ul style="list-style-type: none">• Inline: <code><element ondataavailable = handler>;</code>
------	--

	<ul style="list-style-type: none">• Event property: <code>object.ondataavailable = handler;</code>• Named script: <code><SCRIPT FOR = object EVENT = ondataavailable>;</code>
说明	<p>ondataavailable 事件只要一获得有效的数据就会被触发。这项技术并未说明在这个文件中有多少数据是有效的。</p>

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_50.htm:

```
xmlDoc.ondataavailable = alert('Data is now available.');
```

Onreadystatechange 事件

这个事件会在 `readyState` 属性内容改变时被触发。

基本 语法	<p>此一事件有如下三种处理方式:</p> <ul style="list-style-type: none">• Inline: <code><element onreadystatechange = handler>;</code>
----------	--

	<ul style="list-style-type: none"> • Event property: <code>object.onreadystatechange = handler;</code> • Named script: <code><SCRIPT FOR = object EVENT = onreadystatechange>;</code>
说明	<p><code>onreadystatechange</code> 事件在 <code>readyState</code> 属性内容改变时就会被触发，但这个事件并未说明「准备好」的状态是什么。必须使用 <code>readyState</code> 属性来取得现在的状态。</p>

使用范例

以下的范例请参照随书光盘中的 `AppxA\LstA_51.htm`:

```
xmlDoc.onreadystatechange =
alert("The readyState property is" + xmlDoc.readyState);
```

节点物件 (The node Object)

节点对象表示一文件树的个别分支。您将会发现节点对象含有许多和 **Document** 对象相同的属性、方法和事件，这是因为 **Document** 对象就很多方面来讲其实只是树中的另一个节点。因此，我们会列出节点对象的所有属性、方法、事件，但我们只会针对新的项目详细说明。

节点对象的属性

以下所列为节点对象可用的属性：

- attributes
- baseName
- childNodes
- dataType
- definition
- firstChild
- lastChild
- nameSpace

- nextSibling
- nodeName
- nodeStringType
- nodeType
- nodeTypedValue
- nodeValue
- ownerDocument
- parentNode
- prefix
- previousSibling

- specified
- text
- xml

baseName 属性

传回适当命名空间名称的基本名称。

基本语法	<pre>strValue = xmlDocumentNode.baseName;</pre>
说 明	<p>本属性为只读。一个适当的名称如 ns:base，名称右半边会被传回。在本例中，传回的值为 base。</p>

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_52.htm:

```
strBaseName = xmlDoc.documentElement.childNodes.item(1).baseName;
```

```
alert(strBaseName);
```

dataType 属性

取得或设定指定节点的数据型态。

基本	<code>objValue = xmlDocumentNode.dataType;</code>
语法	<code>xmlDocumentNode.dataType = objValue;</code>
说明	此一属性可供读写，但只针对 rich 数据型态做读取与设定（如果没有指定数据型态，则为 null）。关于 rich 数据型态请参阅 附录 B 。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_53.htm:

```
objNode = xmlDoc.documentElement.childNodes.item(1);  
  
objNode.dataType = "string";  
  
alert(objNode.dataType);
```

definition 属性

传回在 DTD 或结构（schema）中指定的节点定义。

基本语法	<code>objDefinition = xmlDocumentNode.definition;</code>
说明	此一属性只读且会传回一对象。只能应用在有符合实体参照的实体上，如果不属于上述情形则会传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_54.htm:

```
objNode = xmlDoc.documentElement.childNodes.item(1);  
  
alert(objNode.definition);
```

Note

因为这份文件不包含任何实体参照，因此传回 null。

nameSpace 属性

传回命名空间的 **URI**。

基本语法	<code>strURI = xmlDocumentNode.namespace;</code>
说 明	此属性只读且会传回一字符串，也就是 URI，但不会传回命名空间的名称。

使用范例

以下的范例请参照随书光盘中的 **AppxA\LstA_55.htm**：

```
objNode = xmlDoc.createNode(1, "bp:myNode", "bp/nodens");  
  
alert(objNode, namespace);
```

nodeStringType 属性

将节点型态以字符串传回。

基本语法	<code>strTypeValue = xmlDocumentNode.nodeType;</code>
说 明	此一属性只读且会传回一字符串。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_56.htm:

```
objNode = xmlDoc.documentElement.childNodes.item(0);  
  
alert(objNode.nodeType);
```

nodeTypedValue 属性

将节点的值以其定义的数据型态传回。

基本语法	<pre>objValue = xmlDocNode.nodeTypeValue; xmlDocNode.nodeTypeValue = objValue;</pre>
说 明	此一属性可供读写。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_57.htm:

```
objNode = xmlDoc.documentElement.childNodes.item(1);  
  
alert(objNode.nodeTypeValue);
```

prefix 属性

传回命名空间的前缀。

基本语法	<code>strPrefixValue = xmlDocumentNode.prefix;</code>
说 明	此一属性只读且会传回一字符串。只有命名空间的前缀会被传回。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_58.htm:

```
objNode = xmlDoc.createNode(1, "bp:myNode", "bp/nodens");  
  
alert(objNode.prefix);
```

specified 属性

说明一个节点的值是否在元素中被确切地指定，或者是否从 DTD 或结构（**schema**）中取得，此一属性通常被用来当做属性的值。

基本语法	<code>boolSpecified = xmlDocumentNode.specified;</code>
说 明	此一属性只读且会传回一布尔值。若该值在元素中被指定则传回值为 <code>true</code> ，但若是来自 DTD 或结构（ <code>schema</code> ）则传回值为 <code>false</code> 。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_59.htm:

```
objNode = xmlDoc.documentElement.childNodes.item(0);  
  
alert(objNode.specified);
```

text 属性

取得或设定节点的文字。

基本语法	strText = xmlDocoumentNode.text; xmlDocoumentNode.text = strText;
说 明	此属性为一可供读写的字符串值。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_60.htm:

```
objNode = xmlDoc.documentElement.childNodes.item(0);  
  
alert(objNode.text);
```


节点对象的方法

以下所列为节点对象可用的方法：

- `appendChild`
- `cloneNode`
- `hasChildNodes`
- `insertBefore`
- `parsed`
- `removeChild`
- `replaceChild`
- `selectNodes`

- selectSingleNode
- transformNode

hasChildNodes 方法

如果指定的节点有一个以上的子节点则传回 **true**。

基本语法	<code>boolValue = xmlDocumentNode.hasChildNodes();</code>
说 明	此一方法只读且会传回一布尔值。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_61.htm:

```
objNode = xmlDoc.documentElement;  
  
alert(objNode.hasChildNodes());
```

nodeList 物件

nodeList 对象为一在文件树中主动节点集合，「主动」的意思为只要 nodeList 对象有任何改变就会立即反应到集合中。

nodeList 对象的属性

length 属性是 nodeList 对象的唯一属性。

length 属性

传回集合中项目的个数。

基本语法	<code>intValue = xmlNodeList.length;</code>
说 明	此一属性只读且会传回一长整数值。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_62.htm:

```
objNodeList = xmlDoc.documentElement.childNodes;  
  
alert(objNodeList.length);
```

nodeList 对象的方法

下面列出 `nodeList` 对象可用的方法。

- `item`
- `nextNode`
- `reset`

item 方法

存取文件树中的单一节点。

基本语法	<code>objDocumentNode = xmlNodeList.item(index);</code>
说 明	<code>index</code> 为长整数指定子节点的 <code>index</code> (0-based)。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_63.htm:

```
objNode = xmlDoc.documentElement.childNodes.item(2);  
  
alert(objNode.xml);
```

nextNode 方法

存取集合中的下一个节点。

基本语法	objDocumentNode = xmlNodeList.nextNode();
说 明	传回包含下一个节点的对象。若无法取得下一个节点则传回 null。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_64.htm:

```
objNode = xmlDoc.documentElement.childNodes;  
  
objNextNode = objNode.nextNode();  
  
alert(objNextNode.xml);  
  
objNextNode = objNode.nextNode();  
  
alert(objNextNode.xml);
```

reset 方法

重新设定节点清单列表中的索引（**index**）指针。

基本语法	<code>objDocumentNode = xmlNodeList.reset();</code>
说 明	设定节点清单列表中的指针指向第一个节点的前面。

使用范例

以下的范例请参照随书光盘中的 **AppxA\LstA_65.htm**：

```
objNode = xmlDoc.documentElement.childNodes;

objNextNode = objNode.nextSibling();

alert(objNextNode.xml);

objNode.reset();

objNextNode = objNode.nextSibling();

alert(objNextNode.xml);
```

parseError 物件

`parseError` 对象传回最后一个解析错误的信息。要示范 `parseError` 对象如何运作，我们需要使用一份有错误的 XML 文件，如下文字码 A-1a 所示（在随书光盘中 AppxA\Lsta_1a.xml），是一份保有 `Cc`、`Bcc` 元素的电子邮件文件，根据 DTD 这是一篇错误的电子邮件文件。

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

<!DOCTYPE EMAIL SYSTEM "LstA_2.dtd">

<EMAIL PRIORITY="HIGH">

  <TO>Jodie@msn.com</TO>

  <FROM>Bill@msn.com</FROM>

  <BCC>Naomi@msn.com</BCC>

  <CC>Philip@msn.com</CC>

  <SUBJECT>My document is a tree.</SUBJECT>

  <BODY>This is an example of a tree structure.</BODY>

</EMAIL>
```

文字码 A-1a

`parseError` 对象的属性

以下所列为 `parseError` 对象可用的属性：

- errorCode
- filePos
- line
- linePos
- reason
- srcText
- url

errorCode 属性

传回最后一个解析错误的错误码。

基本语法	<code>intErrorValue = xmlDoc.parseError.errorCode;</code>
------	---

说 明	此一属性只读且会传回一长整数。
-----	-----------------

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_66.htm:

```
intParseValue = xmlDoc.parseError.errorCode;

alert(intParseValue);
```

filePos 属性

传回档案中错误发生的位置。

基本语法	intErrorValue = xmlDocument.parseError.filePos;
说 明	此一属性只读，且会传回表示绝对位置的长整数（以字符数表示）。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_67.htm:

```
intParseValue = xmlDoc.parseError.filePos;
```

```
alert(intParseValue);
```

line 属性

传回错误发生所在的行数。

基本语法	<code>intErrorValue = xmlDoc.parseError.line;</code>
说 明	此一属性只读，且会传回表示错误发生所在行数的长整数。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_68.htm:

```
intParseValue = xmlDoc.parseError.line;  
  
alert(intParseValue);
```

linePos 属性

传回错误发生在行中的哪个位置。

基本语法	<code>intErrorValue = xmlDoc.parseError.linePos;</code>
------	---

说 明	此一属性只读，且会传回一长整数表示错误发生在行中的哪一个字符位置。
-----	-----------------------------------

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_69.htm:

```
intParseValue = xmlDoc.parseError.linePos;  
  
alert(intParseValue);
```

reason 属性

传回最后一个错误发生的原因。

基本语法	strErrorReason = xmlDocument.parseError.reason;
说 明	此一属性只读，且会传回最后一个解析错误发生原因的叙述字符串。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_70.htm:

```
strErrorReason = xmlDoc.parseError.reason;
```

```
alert(strErrorReason);
```

srcText 属性

传回错误发生处该行的文字。

基本语法	<code>strSrcText = xmlDocument.parseError.srcText;</code>
说 明	此一属性只读。传回错误发生处该行完整的文字，而且包含空格。

使用范例

以下的范例请参照随书光盘中的 AppxA\LstA_71.htm:

```
strSrcText = xmlDoc.parseError.srcText;  
  
alert(strSrcText);
```

Appendix B XML 的数据型态

此附录包含三个说明 XML 数据型态的表格、两个说明 XML 内容的表格以及一个描述 XML DOM 的表格。

第一个表格包含基本数据型态；第二个包含支持的 rich 数据型态；第三个包含 DOM 节点型态。

基本数据型态在 XML 1.0 规格中被确认，这些基本数据型态被用来辨识 XML 文件中的不同成员，但在传统的程序语言或数据库管理系统（DBMS）中并不可见。例如，entity 型态可以辨识处理器的对象是否为实体，而且必须遵守规则。但是定义此型态时，并不需要指明该数据是文字(text)、数字(number)或是日期(date)，因为根据 XML1.0 规格，所有未标示(markup)的文字一律视为文件的字符数据，所以您可以将基本数据型态看作是字符数据另一种变化。更多关于基本数据型态的使用信息，请参阅 [第 4 章](#) 的说明。

rich 数据型态在 XML-Data 规格中被参照，而且被使用在数据型态的命名空间中，包括 int、char、date 等数据型态，其它相关数据请参阅 [第 6 章](#) 及 [第 10 章](#)。

DOM 提供支持文件树的节点型态，这些型态大多可以对应到基本的 XML 数据型态，但也包含其它的型态。DOM 节点型态可辨识节点型态，以及被包含在节点中的数据型态，例如，

NODE_COMMENT 表示某节点为一批注。DOM 透过数值来辨识每个节点的型态，更多 DOM 节点型态的数据请参阅 [第 9 章](#)。

基本型态(只适用于 Attribute)

数据类型名称	属性值范例	解析型态
entity	entity1	ENTITY
entities	entity1 entity2	ENTITIES
Enumeration	one	ENUMERATION
id	a	ID
idref	a	IDREF
idrefs	a b c	IDREFS
nmtoken	name1	NMTOKEN
Nmtokens	name1 name2	NMTOKENS
Notation	GIF	NOTATION
string	This is a string.	PCDATA

RICH 数据类型(适用于元素与属性)

数据类型名称	元素/属性值范例	解析型态
bin. base64		MIME-styleBase64 编码二进制数据流。
bin. hex		以十六进制数字表现八进位。

Boolean	0, 1 (0 =false, 1=true)	"0" 或 "1"。
char	x	字符串
date	1994-11-05	ISO 8601 子集的日期格式, 不含时间。
dateTime	1988-04-07T18:39:09	ISO 8601 子集的日期格式, 可包含时间, 但不含时区, 秒数准确度为十亿分之一秒。
dateTime.tz	1988-04-07T18:39:09-08:00	ISO 8601 子集的日期格式, 可包含时间及时区, 秒数准确度为十亿分之一秒。
fixed.14.4	12.0044	与 number 相同, 但小数点左边最多有 14 位数字, 小数点右边则最多只能有 4 位数字。
i1	1, 127, -128	可带正负号的数字, 但不含分数、指数。
i2	1, 703, -32768	可带正负号的数字, 但不含分数、指数。
i4	1, 703, -32768, 148343, -1000000000	可带正负号的数字, 但不含分数、指数。

i8	1, 703, -32768, 1483433434334, -1000000000000000	可带正负号的数字，但不含分数、指数。
int	1, 58502, -13	可带正负号的数字，但不含分数、指数。
Number	15, 3.14, -123.456E+10	基本上没有数字位数的限制，可包含前置符号、分数、指数，与美国英语标点方式相同。
r4	.3141592E+1	与 number 的解析型别相同，最小值为 1.17549435E-38F；最大值为 3.40282347E+38F。
r8	.314159265358979E+1	与 number 的解析型别相同，最小值为 2.2250738585072014E-308；最大值为 1.7976931348623157E+308。
String	This is a string.	PCDATA
Time	08:15:27	ISO 8601 子集的时间格式，不包含日期及时区。
time.tz	08:1527-05:00	ISO 8601 子集的时间格式，不包含日期但可包含时区。

ui1	1, 255	不含正负号、分数、指数的数字。
ui2	1, 255, 65535	不含正负号、分数、指数的数字。
ui4	1, 703, 3000000000	不含正负号、分数、指数的数字。
ui8	1483433434334	不含正负号、分数、指数的数字。
uri	urn:schemas-microsoft-com	URI (Universal Resource Identifier)
user-defined		VT_UNKNOWN
type		
Uuid	333C7BC4-460F-11D0-BC04-0080C7055A83	以十六进制数字表示八进位，可以忽略「-」连接符号。

DOM 节点型态

节点型态名称	值
NODE_ELEMENT	1
NODE_ATTRIBUTE	2
NODE_TEXT	3
NODE_CDATA_SECTION	4
NODE_ENTITY_REFERENCE	5

NODE_ENTITY	6
NODE_PROCESSING_INSTRUCTION	7
NODE_COMMENT	8
NODE_DOCUMENT	9
NODE_DOCUMENT_TYPE	10
NODE_DOCUMENT_FRAGMENT	11
NODE_NOTATION	12

Appendix C 相关主题

附录 C 中提供依章节排列的主题清单，包含额外 XML 相关的信息，这些信息为 Microsoft Site Builder Network (SBN) 与 Microsoft Developer NetWork (MSDN) 的一部分，这些主题可由随书光盘或因特网取得。要在随书光盘中取得这些主题，请打开档案 Workshop XML Reference.htm，就可随着文件上的连结浏览想要了解的主题。例如，要找寻「XML Tutorial Lesson1:Authoring XML Elements」主题，只要打开 Workshop XML Reference.htm 档案，在左边页框中展开「XML Tutorial」，然后按下「Authoring Elements」即可。

如果要浏览因特网在线版，请参阅 MSDN 在线资源网

页 <http://msdn.microsoft.com/resources/pardixml.htm>，或者打开随书光盘中的 MSDN_OL.htm 档案，然后选择「Authoring Elements」即可。

第 1 章 了解卷标语言

想更了解 XML 技术面的数据，请看主题 **XML: A Technical Perspective**，选择 **General Information**，然后选择 **Technical Perspective** 即可。

第 2 章 进入主题 XML

想更详细地了解 XML 的基础，请看下列主题：

- [XML: Enabling Next-Generation Web Applications](#)

选择 [General Information](#) ，再选择 [Overview](#) 即可。

- [Frequently Asked Questions About XML](#)

选择 [General Information](#) ，再选择 [FAQ](#) 即可。

第 3 章 XML 的结构及基本语法

想在因特网上读取 W3C XML 1.0 Recommendation，请至 [W3C XML 1.0 Recommendation](#) ，

然后选择 [XML Recommendation](#) 即可。

第 4 章 DTD 的规则

想进一步了解 DTD 的运作方式，可以透过至 [W3C XML 1.0 Recommendation](#) ，然后选择 [XML](#)

[Recommendation](#) ，即可在因特网上浏览 W3C XML 1.0 Recommendation ，或者参阅下面的

主题：

- [Frequently Asked Questions About XML](#)

选择 [General Information](#) ，然后选择 [FAQ](#) 。在左边页面中，选择下面两个问

题： [What are XML schemas?](#) 以及 [How are they different from DTDs?](#)

- [XML, Validation, and Extra Cheese](#)

选择 [Extreme XML Columns](#) ，然后再选择 [XML & DTD](#) 即可。

第 5 章 [Script XML](#)

想进一步了解 [Script](#) 及 [XML](#) 的对象模型，请看下面的主题：

- [XML Tutorial Lesson 5: Using the XML Object Model](#)

选择 [XML Tutorial](#) ，然后选择 [Using the Object Model](#) 。

- [XML DOM Reference](#)

在随书光盘中，选择 [XML Support in IE 5 Beta](#)，然后选择 [XML DOM](#)

[Reference](#)；若在因特网中，请选择 [XML Support in IE 5.0](#)，然后选择 [XML](#)

[DOM Reference](#)。

第 6 章 使用 XML 作为数据来源

更多关于如何将 XML 视为数据来源的数据，透过选择 [W3C XML Documentation](#)，然后选

择 [XML Namespaces Spec](#) 可在因特网上读取 [Namespaces in XML specification](#)，或者参

阅下面的主题：

- [XML Tutorial Lesson 11: Using the C++ XML DSO](#)

选择 [XML Tutorial](#)，然后选择 [Using the C++ XML DSO](#)。

- [XML Tutorial Lesson 6: Using Data Types Within XML Documents](#)

选择 [XML Tutorial](#) ，然后选择 [Using Data Types](#) 。

- [XML Tutorial Lesson 7: Accessing Typed XML Values](#)

选择 [XML Tutorial](#) ，然后选择 [Accessing Typed Values](#) 。

- [Get Your Data on Board: Creating XML Data Sources from Relational Databases](#)

选择 [Extreme XML Columns](#) ，然后选择 [Creating XML Data Sources](#) 。

- [XML 1.0 and Namespace Support](#)

在随书光盘中，请选择 **XML Support In IE 5 Beta**，然后选择 **XML 1.0 and**

Namespace Support；若在因特网，请选择 **XML Support In IE 5.0**，然后选

择 **XML 1.0 and Namespace Support**。

第 7 章 以 XML 连结

想进一步了解 XLink 与 XPointer，请参阅下面的网站。请注意，这些网站内容并不包含在随书光盘中，而且不是 SBN 或 MSDN 的一部份。

- **XLink Working Draft:** <http://www.w3.org/TR/WD-xlink>
- **XPointer Working Draft:** <http://www.w3.org/TR/WD-xptr>
- **XLink Design Principles:** <http://www.w3.org/TR/NOTE-xlink-principles>

第 8 章 XSL: 具样式的 XML

想进一步了解 XSL 的使用，透过选择 **W3C XML Documentation**，然后选择 **XSL Working**

Draft 即可在因特网上读取 **XSL working draft**，或者参阅下面的主题：

- [XML Tutorial Lesson 14: Handling XSL Errors](#)

选择 [XML Tutorial](#) ，然后选择 [Handling XSL Errors](#) 。

- [XML Tutorial Lesson 15: Using Script in XSL](#)

选择 [XML Tutorial](#) ，然后选择 [Using Script in XSL](#) 。

- [XML Tutorial Lesson 16: Handling Irregular in XSL](#)

选择 [XML Tutorial](#) ，然后选择 [Handling Irregular Data in XSL](#) 。

- [XSL Reference](#)

在随书光盘中，请选择 [XML Support In IE 5 Beta](#) ，然后选择 [XSL Reference](#) ；

如果在因特网上，请选择 [XML Support In IE 5.0](#) ，然后选择 [XSL Reference](#) 。

第 9 章 以 XSL Pattern 来表现资料

想进一步了解 XSL 样式(patterns)，请参阅下面的主题：

- [XML Tutorial Lesson 9: Using XSL Patterns to Retrieve Nodes](#)

选择 [XML Tutorial](#) ，然后选择 [Using XSL Patterns](#) 。

- [XML Tutorial Lesson 13: Using XSL to Generate XML](#)

选择 [XML Tutorial](#) ，然后选择 [Using XSL to Generate XML](#) 。

第 10 章 XML-Data

更多有关 XML-Data 规格的信息，可以透过选择 [W3C XML Documentation](#) ，然后选

择 [XML-Data Spec](#) ，即可在因特网上读取 [XML-Data specification](#) ，或者参阅下列主题：

- [XML Tutorial Lesson 8: Authoring XML Schemas](#)

选择 [XML Tutorial](#)，然后选择 [Authoring XML Schemas](#)。

- [XML Schema and Data Types Preview](#)

在随书光盘中，请选择 [XML Support In IE 5 Beta](#)，然后选择 [XML Schema and Data Types Preview](#)；若在因特网上，请选择 [XML Support In IE 5.0](#)，然后选择 [XML Schema and Data Types Preview](#)。

第 11 章 XML 的现在与未来

想进一步了解 XML 的发展，请参阅：

- [Java Parser](#)

选择 **General Information** ，然后选择 **Microsoft/DataChannel Parser**

Announcement 。

- **Applying XML to various scenarios**

选择 **Scenarios** ，然后选择任何一个有兴趣的主题即可。

- **Document Content Description for XML**

选择 **W3C XML Documentation** ，然后选择 **DCD for XML** 。

< 全书完 >