

## 第6章 SAX 1.0 : XML简易API

在第5章里考虑了如何使用文档对象模型 (DOM) 编写应用程序。本章将着眼于处理 XML 文档的另一种方式：SAX 接口。我们先论述一下为什么要选择使用 SAX 接口而不是文档对象模型，然后通过编写一些简单的应用程序来探究 SAX 接口技术。本章也将讨论一些设计模式，它们在创建更复杂的 SAX 应用程序时会有所帮助。最后本章描述了 SAX 未来的发展前景。

SAX 的接口风格完全不同于文档对象模型。文档对象模型应用程序通过遵循内存中的对象参照来要求文档中的内容；使用 SAX 解析器通过向应用程序报告解析事件流来告知应用程序文档的内容。

SAX 即 XML 简易应用程序编程接口，全称是扩展标记语言简易应用程序编程接口。

从名称上可以看出，SAX 接口可以用来编写应用程序以读取 XML 文档中存有的数据。SAX 在本质上是一种 Java 接口，本章所有给出的例子也是用 Java 编写的。（因为本章没有足够的篇幅讲解 Java 技术，为此我们假设读者已具有了 Java 的相关知识。参见 Wrox 出版社出版的《Beginning Java2》，ISBN 1861002238；也可以在 <http://www.java.sun.com> 查找到更多的信息。）

SAX 接口事实上被所有的 Java XML 解析器支持，而且兼容性非常好。参看 <http://www.xmlsoftware.com> 或 David Megginson 的站点 <http://www.megginson.com/SAX/> 列出的一些实例。

在用 Java 编写 SAX 应用程序前，你需要安装 SAX 类（当然要先安装 Java JDK）。大多数情况下你会发现 XML 解析器自动帮你安装了 SAX 类（稍后我们将告知如何快捷地获取这些解析器）。检查在你的类路径中是否包含如 `org.xml.sax.Parser` 的类，如果没有，你可以从 <http://www.megginson.com/SAX/> 安装这些类。我们随后将介绍一下关于 SAX 的起源与发展趋势。但是现在我们将仅提及它最显著的一个特性：SAX 不属于任何标准组织或团体，也不属于任何公司或个人；它是供任何人实现与使用的一种计算机技术。尤其与大多数 XML 标准族的不同之处在于，SAX 和 W3C 组织没有任何关系。

SAX 的开发工作由 David Megginson 负责协调，SAX 的规范可在 David Megginson 的站点查找（<http://www.megginson.com/SAX/>）。这个规范中有些不重要的编辑方面的修改，对本书附录 C 中的相关约定进行了再版。

### 6.1 事件驱动接口

从程序中读取 XML 文档基本上有三种方式：

- 把 XML 只当作一个文件读取，然后自己挑选出其中的标签。这是黑客们的方法，我们不推荐这种方式。你很快会发现处理所有的特殊情况（包括不同的字符编码，例外约定，内部和外部实体，缺省属性等）比想象的困难得多；你可能不能够正确地处理所有的特殊情况，

这样你的程序会接收到一个非常规范的 XML 文档，却不能正确地处理它。要避免这种想法：XML 解析器似乎并不昂贵（大多数是免费的）。

- 可以用解析器分析文档并在内存里创建对文档内容树状的表达方式：解析器将输出传递给文档对象模型，即 DOM。这样程序可以从树的顶部开始遍历，按照从一个树单元到另一个单元的引用，从而找到需要的信息。
- 也可以用解析器读取文档，当解析器发现标签时告知程序它发现的标签。例如它会告知它何时发现了一个开始标签，何时发现了一些特征数据，以及何时发现了一个结束标签。这叫做事件驱动接口，因为解析器告知应用程序它遇到的有含义的事件。如果这正是你需要的那种接口，可以使用 SAX。

让我们更加详细地看一下事件驱动解析过程。

你可能已经在用户接口编程中遇到过“事件驱动”这个术语，用来编写应用程序以响应发生的如鼠标点击等事件。事件驱动解析器和其类似：特别是你必须习惯于应用程序不是你所控制这一概念。一旦事情要开始发生，你不需要调用解析器，而是解析器调用程序。开始看起来有些奇怪，但是一旦你习惯了，这就不是个问题。实际上，它比用户接口编程更容易，因为不需要忙于处理随时发生的鼠标事件，XML 要解析的事件按照相对可预见的顺序出现。XML 元素必须完全成对出现，所以你可以知道每个已经打开的元素随后肯定会被关闭。

下面是一个简单的 XML 文件：

程序清单 6-1

```
<?xml version="1.0"?>
<books>
  <book>Professional XML</book>
</books>
```

当解析器进行处理时，它会调用一连串方法，如下所示（我们将在后面描述实际的方法命名与参数，这里仅为例证说明）：

程序清单 6-2

```
startDocument()
startElement( "books" )
startElement( "book" )
characters( "Professional XML" )
endElement( "book" )
endElement( "books" )
endDocument()
```

你的应用程序提供当 startElement 和 endElement 等事件发生时需要调用的方法。

为什么使用事件驱动接口

假如可以选择的话，那么知道什么情况下使用 SAX 这样的事件驱动接口最好，或者在什么时候使用 DOM 这样基于树结构的接口更好一些是很重要的。

这两种接口都已经很好地标准化并得到广泛的支持，因此不管使用哪一种接口，都有很多性能良好的解析器可供选择，而且大部分是免费的。实际上很多解析器同时支持两种接口。

### 1. SAX的优点

下面的部分简述了SAX接口最显著的一些优点。

#### (1) 可以解析任意大小的文件

因为SAX不需要把整个文件加载到内存，所以对内存的占用一般比 DOM小得多，而且不随着文件大小的增加而增加。当然 DOM使用的实际内存数量要视解析器而定，但在大多数情况下一个100Kb的文档至少要占用1Mb的内存。

但是有一点要注意：如果SAX应用程序自身在内存中创建文档的表达，它会占用和允许解析器创建空间一样大小的内存。

#### (2) 适合创建自己的数据结构

应用程序可能会想用如书、作者以及出版者这样的高级对象而不是一些低级元素、属性和处理指令来创建数据结构。这些“交易对象”可能只是和XML文件内容有一点关系；例如它们可能只是组合XML文件和其他数据源的数据。在这种情况下，如果想在内存中创建面向应用的数据结构，首先创建一个低级DOM结构然后毁坏它是很不合算的。可以仅在每个事件发生时处理它，这样保证商务对象模型合理地增加变动。

#### (3) 适合小信息子集

如果仅对计算本周图书馆购进的书籍数量或确定它们的平均价格感兴趣，那么把不需要的全部数据和需要的少量数据一起读入内存是非常低效和不必要的。SAX 一个非常好的特点就是可以非常容易地忽略不感兴趣的数据信息。

#### (4) 简单

如题所示，SAX非常易于使用。

#### (5) 快速

如果可以从经由文档的简单序列中获取你需要的信息，SAX几乎一定是最快的方法。

### 2. SAX的缺点

在论述完其优点之后应该指出使用SAX时可能遇到的不足之处。

#### (1) 不能对文档做随机存取

因为文档并不加载到内存，所以必须按照数据提交的次序进行处理。对于文档中包含许多内部交叉引用如使用ID和IDREF属性的情况，SAX使用起来会困难一些。

#### (2) 难以实现复杂的查询

复杂的查询对程序而言是非常凌乱的，因为必须自己维护含有你所需要保留信息的数据结构，如当前元素祖先的属性。

#### (3) 不能使用文档类型定义 ( DTD )

SAX1.0不会告知DTD的任何内容。实际上DOM也不会告知太多内容什么，尽管有些提供商已经扩展了DOM接口以支持这种功能。这对大多数应用程序来说并不是个问题：DTD主要是解析器感兴趣的问题；而且在本章末尾可以看到这个问题在SAX 2.0中得到了解决。

#### (4) 不可获取词法信息

SAX的设计原理是它不提供词法信息。SAX设法告知文档作者想要说明什么，而不是让你忙于研究他们说明方式的细节。例如：

- 你不能查明原始文档中是否包含“&#xa;”或“&#10”或它是否包含一个换行字符：所有这三种情况以相同的方式报告给应用程序。
- 你不会被告知文档中的注释说明：SAX假设注释是为方便作者而不是读者设计的。
- 你不会得知属性书写的顺序：这被认为是无关紧要的。

只有当你考虑到以后可能会需要编辑文档，想按照文档原先书写的方式重新创建它时，这些限制才会导致一些问题。例如你需要编写这样一个应用程序，它用来在完整无缺地保留文档原有内容的基础上，从另外一个文档添加某些额外信息到原文档中。这样如果你随意改变了属性的顺序或丢弃了所有的注释，原文档作者会不满意的。实际上，大多数限制和 DOM是一样的，尽管DOM的确在某些方面提供了稍多一些的信息：例如它保留了注释。此外，很多这样的限制在SAX 2.0里得到改进；尽管没有完全解决，例如属性的顺序和分割符（单引号或双引号）的选取一样仍然是个难题。

#### (5) SAX是只读的

DOM可以从XML原文件中读取文档，也可以创建和修改内存中的文档。相比较而言，SAX是用来读取XML文档而不是书写文档。

实际结果是SAX接口可以很容易地和读取XML文档一样书写文档。稍后可以看到，解析器读取XML文档时发送给应用程序的事件流同样可以被应用程序发送到文档生成器以创建文档。

#### (6) 当前的浏览器不支持SAX

尽管有许多支持SAX接口的XML解析器。在编写本书时还没有一个主流的Web浏览器内置XML解析器以支持SAX接口。你当然可以把兼容SAX的解析器合成到一个Java applet程序中去，但是从服务器下载applet的开销会使低速接入Internet的用户失去耐心。实际上客户端XML编程可选的接口是相当有限的。

## 6.2 SAX的由来

较少文档提及SAX的历史，因为所有讨论是通过XML-DEV公共邮件列表实现的，邮件列表的文本可以从<http://www.list.ic.ac.uk/hypermail/xml-dev/>获取。David Megginson也在<http://www.megginson.com/SAX/history.html>中概述了SAX的历史。

整个过程始于1997年底，作为像Peter Murray-Rust这样一些开发XML应用程序并致力于解决不同解析器间不能无缝兼容问题的XML用户施加压力的结果。早期XML解析器的提供者，包括Tim Bray, David Megginson和James Clark对讨论也做出了贡献，许多其他邮件列表成员对不同的草稿也提出一些见解。David Megginson发明了一种讨论方法，当然是依照Internet初始的“征求意见稿”精神，由此意见和建议可以被迅速而公平地处理，最后他于1998年5月11日发布了规范终稿。

SAX成功的一个主要原因是初始的规范，Megginson为许多流行的XML解析器——包括他自己的AElfred, Tim Bray的Lark和Microsoft的MSXML提供了前端驱动。一旦SAX以这种方式建立，其他解析器提供商如IBM, Sun和Oracle很快在它们的解析器中集成了最早的SAX接口，这样它们的产品就可以和现有的应用程序一起运行。

最终的SAX规范是根据Java接口书写的。它已经被改写成其他语言，尽管我们只知道用

Python语言编写的接口得到积极支持，Python是Lars Marius Garshol创立的（参见<http://www.stud.ifi.uio.no/~larsga/download/python/xml/saxlib.html>）。当然，Java接口可以在其他可以和Java交互操作的语言中使用，例如通过使用Microsoft的Java虚拟机提供的从Java到COM的接口。不过在本章里将只使用初始的Java。

### SAX的结构

SAX是由许多Java 接口构建而成的。了解接口和类之间的区别是很重要的：

- 接口表示它们是什么方法和它们需要的参数种类。接口完全是一个规范；当方法被调用时它并不提供任何执行代码。然而接口是具体的规范，而不仅仅是文件片段，Java编译器保证需要实现接口的类正确地操作。
- 类提供可执行的方法，包括可以被其他类中代码调用的公共方法。
- 类可以实现一个或多个接口。很多情况下SAX指定的一些接口理论上可以被各不相同的类实现，但实际上经常是和某单个类结合实现的。为了实现一个接口，类必须提供接口中定义的每个方法的代码。
- 几个类可以实现同一个接口。当然这正是SAX需要重点做的——有许多SAX解析器接口的实现可以选择，因为它们实现相同的接口，应用程序无须关心使用的是哪一个实现。

一些SAX接口是解析器中的类实现的，而有些SAX接口必须由应用程序中的类实现。SAX自身提供了一些类，尽管可以不使用它们。有些类是解析器必须提供的（如错误处理类），但是应用程序可以根据需要忽略它们。

#### 1. 基本结构

一个简单的SAX应用程序的构成如图6-1所示。

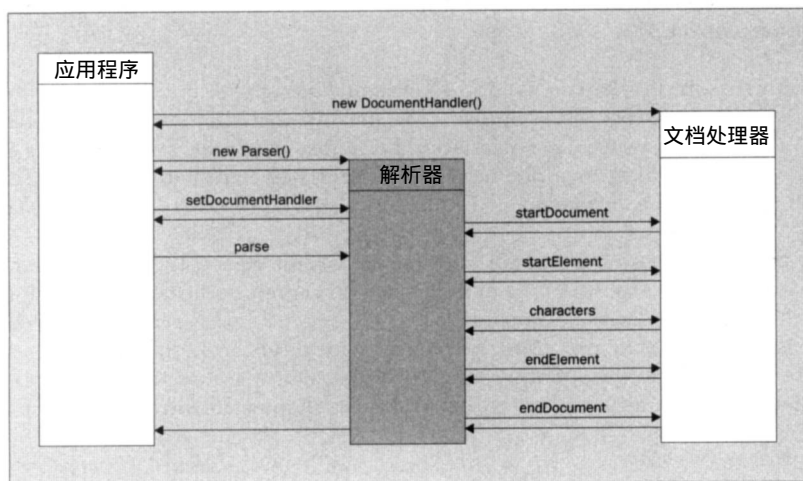


图 6-1

在图中：

- 应用程序是主程序：你编写的用来开始整个处理过程的代码。
- 文档处理器是你编写的用来处理文档内容的代码。



• 解析器是符合SAX标准的XML解析器。

应用程序的工作就是创建一个解析器（更准确地说是初始化一个实现 `org.xml.sax.Parser`接口的类）；创建一个文档处理器（通过初始化一个实现 `org.xml.sax.DocumentHandler`接口的类）；告知解析器使用哪个文档处理器（通过调用解析器的 `setDocumentHandler()`方法）；告知解析器开始处理一个特定的输入文档（通过调用解析器的 `parse()`方法）。

解析器的作用是通报文档处理器它在文档中找到的所有感兴趣的内容，例如元素的起始和结束标签。

文档处理器的作用是处理这些通报以获取应用程序需要的所有内容。

## 2. 一个简单的SAX例子

下面是一个非常简单的应用程序：程序仅仅对输入 XML文件中<book>元素的数量进行统计（XML文件清单附后）。

在本例中，通过使用同一个类实现主程序与文档处理器以简化上面图中描述的 SAX应用程序结构。这是因为一个Java类可以实现几个接口，所以它可以同时做不同工作。

应用程序必须首先创建一个解析器：

程序清单 6-3

```
import org.xml.sax.*;

...

Parser p = new com.jclark.xml.sax.Driver();
```

你只需要在这里说明要使用哪种特定的 SAX解析器。我们选用了James Clark开发的xp解析器，它可以从<http://www.jclark.com> 获取。当然就像你使用的其他Java类一样，它必须加到Java类路径中。

选用的解析器必须实现 SAX解析器接口 `org.xml.sax.Parser`（否则Java处理起来会很困难），所以可以把一个 `Parser`类型的变量赋值给它。由于开始的 `import`语句，`Parser`实际上代表 `org.xml.sax.Parser`。

所以你需要知道选用解析器的相关类的名称。奇怪的是很多可用的 SAX解析器并不特别明确地公布它们的类名。所以表 6-1列出了一些更流行的解析器和创建实例时需要使用的类名（但是注意这些名称可能随着产品后来的版本而改变）。

表 6-1

产 品	详 细 描 述
AElfred	地址: <a href="http://www.microstar.com/aelfred.html">http://www.microstar.com/aelfred.html</a> 解析器类: <code>com.microstar.xml.SAXDriver</code>
Datachannel DXP	地址: <a href="http://www.datachannel.com/products/xjparser.html">http://www.datachannel.com/products/xjparser.html</a> 解析器类: <code>com.datachannel.xml.sax.SAXDriver</code>
IBM xml4j	地址: <a href="http://alphaworks.ibm.com/tech/xml4j">http://alphaworks.ibm.com/tech/xml4j</a> 解析器类（非确认型）: <code>com.ibm.xml.parsers.SAXParser</code> 解析器类（确认型）: <code>com.ibm.xml.parsers.ValidatingSAXParser</code>
Oracle	地址: <a href="http://www.oracle.com">http://www.oracle.com</a> （需要TechNet注册）

(续)

产 品	详 细 描 述
Sun Project X	解析器类: oracle.xml.parser.v2.SAXParser 地址: <a href="http://java.sun.com/products/xml/">http://java.sun.com/products/xml/</a>
	解析器类 (非确认型): com.sun.xml.parser.Parser
	解析器类 (确认型): com.sun.xml.parser.ValidatingParser
xp	地址: <a href="http://www.jclark.com/xp">http://www.jclark.com/xp</a> 解析器类: com.jclark.xml.sax.Driver

这样你就已经创建了一个解析器。现在可以指定解析器要进行的操作了。

首先需要告诉解析器当事件发生时调用哪个文档处理器。它可以是任何实现 org.xml.sax.DocumentHandlerSAX 接口的类。最简单也是最常用的方式是让你的应用程序本身作为文档处理器。

DocumentHandler 本身是 SAX 定义的一个接口。可以让应用程序直接实现接口, 这样的话就必须为接口所需要的各种不同方法提供代码实现。但是在本例中, 我们希望忽略大多数事件, 那么定义许多什么都不做的方法是相当繁琐的。好在 SAX 提供了对空操作 DocumentHandler 的实现——HandlerBase, 我们的程序可以在此基础上进行扩展, 这样它会继承所有“空”方法。如下所示:

程序清单 6-4

```
import org.xml.sax.*;

...

public class BookCounter extends HandlerBase
{
    public void countBooks()
    {
        Parser p = new com.jclark.xml.sax.Driver();
        p.setDocumentHandler(this);
    }
}
```

setDocumentHandler() 调用告诉解析器“this”类(应用程序)要接收事件的告示。这个类是 org.xml.sax.DocumentHandler 的实现, 因为它继承了 org.xml.sax.HandlerBase, 而其又实现了 DocumentHandler。

解析器现在基本上可以运行了; 它只需一个要解析的文档和 Java main() 方法使它成为一个独立的程序。下面首先给出要解析的文件:

程序清单 6-5

```
import org.xml.sax.*;

...

public class BookCounter extends HandlerBase
{
```

```
public void countBooks() throws Exception
{
    Parser p = new com.jclark.xml.sax.Driver();
    p.setDocumentHandler(this);
    p.parse("file:///C:/data/books.xml");
}
```

注意 `parse()` 的参数是一个 URL 字符串。稍后会给出如何用 一个文件名代替 URL。因为程序涉及到数据的输入和输出，所以必须添加 “ `throws Exception` ” 到 `countBooks()` 方法以使发生错误时发出警告。

还需要添加 Java `main()` 方法以使程序成为一个独立的应用程序。在 `main` 方法中用 `new BookCounter()` 创建类的一个实例，然后调用对象的 `countBooks()` 方法；同时对新建对象进行全程跟踪。代码如下所示：

#### 程序清单 6-6

```
import org.xml.sax.*;
...

public class BookCounter extends HandlerBase
{
    public static void main (String args[]) throws Exception
    {
        (new BookCounter()).countBooks();
    }

    public void countBooks() throws Exception
    {
        Parser p = new com.jclark.xml.sax.Driver();
        p.setDocumentHandler(this);
        p.parse("file:///C:/data/books.xml");
    }
}
```

现在程序可以运行了：解析文档直至运行结束（当然假设存在要被解析的文档）。

唯一的不足是程序现在没有产生输出。为使程序更有用，需要添加一个方法对出现的 `<book>` 开始标签进行计数，另外添加方法在文档末尾输出书的统计数目。这些方法使用了全局变量 `count`。

应用程序的最后版本如下所示。可以在我们的站点 <http://www.wrox.com/> 上关于本书的专栏里找到本章中的代码。

#### 程序清单 6-7

```
import org.xml.sax.*;

public class BookCounter extends HandlerBase
{
    private int count = 0;
```



```
public static void main (String args[]) throws Exception
{
    (new BookCounter()).countBooks();
}

public void countBooks() throws Exception
{
    Parser p = new com.jclark.xml.sax.Driver();
    p.setDocumentHandler(this);
    p.parse("file:///c:/data/books.xml");
}

public void startElement(String name, AttributeList atts) throws SAXException
{
    if (name.equals("book"))
        count++;
}

public void endDocument() throws SAXException
{
    System.out.println("There are " + count + " books");
}
}
```

现在可以通过下列形式的命令从命令行运行应用程序：

```
java BookCounter
```

程序会输出给出的XML文件中<book>元素的数目。假设c:\data\books.xml文件中包含下列文件（可以从<http://www.wrox.com>下载获取本章中的代码）：

程序清单 6-8

```
<?xml version="1.0"?>
<books>
  <book category="reference">
    <author>Nigel Rees</author>
    <title>Sayings of the Century</title>
    <price>8.95</price>
  </book>
  <book category="fiction">
    <author>Evelyn Waugh</author>
    <title>Sword of Honour</title>
    <price>12.99</price>
  </book>
  <book category="fiction">
    <author>Herman Melville</author>
    <title>Moby Dick</title>
    <price>8.99</price>
  </book>
</books>
```

这样在终端会显示下列输出：

```
>java BookCounter
There are 3 books
```

### 3. DocumentHandler接口

如上面的例子所示，一个 SAX 应用程序的主要工作是在实现 DocumentHandler 接口的类里完成的。通常我们会对比上面简单例子包含的更多的事件感兴趣，因此让我们看一下构成接口的其他方法。

#### (1) 文档事件

首先，有一对方法标记文档处理的开始与结束：

- startDocument()
- endDocument()

这两个方法没有参数和返回值。实际上，你通常可以不使用它们，因为程序开始进行的操作一般可以在调用 parse() 之前完成，而程序结束时要做的操作可以在 parse() 返回值时完成。但是在一个更复杂的应用程序中，你可能希望调用 parse() 的应用程序是从 DocumentHandler 派生的一不同的类，在此种情况下，这两个方法有助于初始化变量和最后回收变量资源。

注意一个 SAX 解析器（Parser 类的一个简单实例）应该同时只能解析一个 XML 文档。一旦它完成了解析，可以再使用它解析其他文档。如果你想同时解析几个文档，就需要为每个文档创建一个 Parser 类的实例。你肯定想把这种文档实例一一对应的规则应用到 DocumentHandler，因为在告知事件来源于哪个文档的事件信息中也没有进行什么其他操作。

#### (2) 元素事件

与文档事件一样，有一对可调用的方法用来标记文档中每个元素的开始和结束标签：

- startElement(String name, AttributeList attList)
- endElement(String name)

name 是指在元素开始和结束标签中出现的名称。

如果文档对一个空元素使用简短语法表示（即 “<tag/>”），解析器将完全按你书写 “<tag></tag>” 的方式通报开始和结束标签。这是因为 XML 定义这两个结构是同等的，所以应用程序不需要知道使用的是哪一个。

在开始标签里出现的属性捆绑在一起成为 AttributeList 对象，同时立刻被应用程序处理。基于事件的模式即始于此，你可能希望每个属性在出现时都发出通报。AttributeList 是 SAX 定义的另一个接口。类似于解析器，它定义了一个实现接口的类：应用程序需要知道的是它为了获取各个属性详细信息时可以调用的方法。其中最有用的一个是：

- getValue(String name)

该方法在属性存在时以字符串形式返回命名属性的值，否则返回空。

关于 AttributeList 需要注意的是它仅在 startElement() 方法的生命周期中有效。一旦该方法将控制权返回给解析器，它可能（并且经常会）用不同的信息覆盖 AttributeList。如果想保留属性信息以备后用，你需要做一个副本。一种简便的方式是使用 SAX “helper” 类 AttributeListImpl：可以让你创建另一个 AttributeList 作为已给出 AttributeList 的私有副本。

#### (3) 字符数据

XML 文档中出现的字符数据一般通过下面的方法报告给应用程序：

- characters(char[] chars, int start, int len)

定义这种接口更多的是为了效率而不是方便。如果想把字符数据作为 String 处理，你可以简单地构造一个字符串：

```
String s = new String(chars, start, len);
```

解析器可能已经创建了 String 结构，但是在 Java 中创建新的对象是需要占用大量资源的，所以它只给出指向字符存放位置的指针。

使用 Java 处理 XML 的一个优点是 Java 和 XML 都是采用统一码标准字符集。不管开始的源文档中使用哪种字符编码方式，以 chars 数组形式传递的字符始终是本地化的 Java 统一码字符。

重要的一点是解析器可以任意分解字符数据，也可以同时只传送一部分。这就是说如果你想在文档中查找 “gold”，那么下面的代码就有错误：

程序清单 6-9

```
public void characters(char[] chars, int start, int len) throws SAXException
{
    String s = new String(chars, start, len);
    if (s.indexOf("gold") >= 0) ...
}
```

为什么呢？因为可能文档中出现了 “gold” 字符串，但是是通过两次或更多次 characters() 方法调用来告知应用程序。理论上，可能会有四个调用，分别对应 “g”, “o” 和 “ld”。

问题的最坏情况是在测试中你可能没有发现这种问题，因为实际上解析器很少用这种方式分解文本。例如，它们可能仅当文本恰好处于一个 4096 字节边界时才分解文本（同时如果内存因为某些原因受这种限制），这种情况可能直到程序成功运行几个月后都不会发生。注意这种问题。

有一种情况解析器必须分解文本，就是当使用外部实体的时候。SAX 规范明确规定一个 character() 调用不能包含来自两个不同外部实体的文本。

如果想直接对字符数据操作而不是简单地无条件地复制它到输出文件，你也许会对其所属哪个元素感兴趣。然而 SAX 接口不会直接提供这种信息。如果需要这种上下文信息，你的应用程序必须维护一个数据结构，它保留以前事件的一些内存映像。最常见的是堆栈。在下一节会说明如何使用一些简单的数据结构，既能组合解析器提供的字符数据片段，又可以确定它所属的元素。

这是第二种报告字符数据的方法，即：

```
• ignorableWhitespace(char[] chars, int start, int len)
```

这个接口可以告知 SAX 规范泛指 “可忽略空白”。如果 DTD 用 “元素内容” 定义了一个元素（也就是说元素可以有子元素，但不能包含 PCDATA），那么即使 “真正” 的字符数据是不允许的，XML 也可以用空格、制表符和换行分割开子元素。这种空白很可能是无关紧要的，所以一个 SAX 应用程序几乎总是忽略它：你可以仅仅通过使用一个空操作的 ignorableWhitespace() 方法实现。只有当你的应用程序把输入数据不加改变地复制到输出时，你才有可能想做一些其他操作。

然而 XML 规范允许一个解析器忽略外部 DTD 中的信息。非确认解析器不需要辨别包含元素

内容的元素和包含混合内容的元素。在这种情况下，可忽略空白可以通过一般的 `characters()` 接口被告知。

不幸的是无法在一个 SAX 应用程序中判断解析器是否是确认型的，所以一个可移植的应用程序必须考虑到这两种情况。这也是 SAX 的一个局限性，但是在 SAX 2.0 里得到了修正。

#### (4) 处理指令

解析器还报告一种被称为处理指令的元素。你很可能不会经常地遇到处理指令：它们可能出现在 XML 文档中 “`<? ” “ ?>`” 之间任何位置的指令。一个处理指令有一个名称（即目标）和任意特征数据（目标应用程序有关的指令）。

处理指令通过下面的方法通知给 `DocumentHandler`：

- `processingInstruction(String name, String data)`

按照约定，应该忽略所有的处理指令（或者不加改变地复制它），除非可以识别处理指令的名称。

在文档开始的 XML 声明可能像是一个处理指令，但是它不是一个真正的处理指令，并且不会通过上述接口告知应用程序——实际上是什么也不告知。

处理指令经常被写成像元素开始标签的形式，包含一连串 `keyword="value"` 属性。然而这种语法对一个应用程序约定是完备的，而不在 XML 标准中定义。所以 SAX 不能识别它；处理指令数据的内容被当作乱七八糟的东西被忽略掉。

#### 4. 错误处理

到目前为止我们掩饰了对错误的处理，但是在一个真正的生产性应用程序中，错误处理总是需要认真考虑的。

主要有三种错误可能出现：

- 不能打开 XML 输入文件或者它引用的另一个文件，例如 DTD 或另一个外部实体。在这种情况下，解析器就会产生一个 `IOException`（输入/输出异常）而由应用程序处理它。
- 解析器检测到的 XML 错误，包括规范化错误和合法性错误。可以通过调用应用程序提供的错误处理器进行错误处理，如下面所述。
- 应用程序检测到的错误：例如属性中不完整的日期或数字。可以通过在检测到错误的 `DocumentHandler` 方法中产生一个例外进行错误处理。

##### (1) 处理 XML 错误

根据 XML 标准自身用到的术语，SAX 规范对错误严重程度定义了三个级别（参见表 6-2）。

表 6-2

错 误	描 述
致命错误	通常表示 XML 格式不正确。如果存在注册的错误处理器，解析器就会调用它；反之解析器产生一个 <code>SAXParseException</code> 。大多数情况下，解析器发现第一个致命错误后就会终止运行
错误	通常表示 XML 格式正确但不合法有效。如果存在注册的错误处理器，解析器就会调用它；反之解析器将忽略错误

(续)

错 误	描 述
警告	表示XML是正确的，但是解析器认为报告一些情况是有用的。例如可能是一个“交互操作”规则的冲突：输入是正确的XML但不是正确的 SGML。如果存在注册的错误处理器，解析器就会调用它；反之解析器将忽略错误

应用程序可以通过解析器的 `setErrorHandler()` 方法注册一个错误处理器。错误处理器包含三个方法，`fatalError()`，`error()`和`warning()`，分别对应三种级别的错误。如果不想定义全部三种方法，你可以创建一个从 `HandlerBase` 继承而来的错误处理器：它包含所有三种方法的样式，但所做的操作和没有注册错误处理器相同。

在所有三种情况下，错误处理器方法的参数是一个 `SAXParseException` 对象。你可能认为当错误发生时产生并处理的是 Java 例外；但是实际上例外是一个正规的 Java 对象，可以像其他 Java 对象一样作为一个参数传递给方法：也可能根本不会被产生。`SAXParseException` 包含错误的有关信息，如它在 XML 源文件中何处出现。错误处理器方法最常见的操作是抽取这些信息以生成一个错误消息，错误消息可以被写到适当的目标文件：例如一个 Web 服务器日志文件。

错误处理器方法还可以产生异常：异常经常被当做参数由解析器提供，但不是必须这样。如果异常产生了，解析一般会被终止，并且高层的应用程序会看到由 `parse()` 方法产生的相同的异常。这是另一个输出诊断消息的时机。是从错误处理器里生成一个致命错误消息还是让高层应用程序捕获异常，这完全由你决定。

### (2) 应用程序检测到的错误

当应用程序在 `DocumentHandler` 方法中检测到一个错误（例如一个错误格式的日期），`DocumentHandler` 方法会产生一个 `SAXException`，它包含对问题进行解释的适当消息。此后，解析器就像它自己检测到错误一样进行恰当地处理。一般来说，它不会去捕获这个例外，而是相同的例外立即从 `parse()` 方法中退出，这样高层应用程序就可以捕获例外。

### (3) 确定错误发生的位置

当解析器检测到一个 XML 语法错误，它将在 `SAXParseException` 对象里提供错误的详细信息。这个对象包括错误所在的 URL、行、列（行号本身没有多大用处，因为错误可能是在一些外部实例而不是主文档里）。当在应用程序里捕获到 `SAXParseException`，你就可以抽取这些信息展示给用户以定位错误。

如果 XML 文件相关的错误是在应用程序级被检测到（例如一个无效的日期），那么告知用户错误发生的位置也同等重要，这时你就不能依赖 `SAXParseException` 去定位错误，而是由 SAX 定义一个 `Locator` 接口。SAX 规范并不要求解析器提供一个 `Locator`，但是大多数解析器提供。

在文档处理器里必须实现的一个方法是 `setLocator()` 方法。如果解析器持有定位信息，它将调用 `setLocator()` 方法告知文档处理器 `Locator` 对象的位置。在文档处理器处理时间后的任何时候它都可以查询 `Locator` 对象以得到源文档中当前坐标的详细信息。总共有三种坐标：

- 文档或当前处理的外部实体的 URL。
- URL 里的行号。

- 行里的列号。

所有相同的信息你当然也可以从 SAXParseException 对象里获取，实际上当应用程序检测到错误时可以很容易做到的一件事就是产生一个 SAXParseException，由 SAXParseException 直接从 Locator 对象里得到坐标信息——只要写下列语句：

```
if ( [data is not valid] )
{
    throw new SAXParseException("Invalid data", locator);
}
```

为什么不简单地把定位信息包含于如 startElement() 等要传递给文档处理器的事件里呢？原因是效率：大多数应用程序只想得到出错时的定位信息，这样当不需要有关信息时可以有最小的开销。从解析器到文档处理器的每个调用都提供定位信息是不必要的过多开销。

#### 5. 另一个使用字符数据和属性的例子

了解完错误处理过程后，让我们开发一个稍微复杂一些的 SAX 应用程序实例。

现在应用程序的任务是输出目录中科幻书籍的平均价格。我们采用和前面实例相同的数据文件（books.xml）。

我们只关心那些含有 category="fiction" 属性的 <book> 元素，而且对这些元素只需要知道它 <price> 子元素的内容。先累加价格，然后清点书的数量，最后用总价格除以书的总数。

下面是应用程序的最初版本：

#### 程序清单 6-10

```
import org.xml.sax.*;

public class AveragePrice extends HandlerBase
{
    private int count = 0;
    private boolean isFiction = false;
    private double totalPrice = 0.0;
    private StringBuffer content = new StringBuffer();

    public void determineAveragePrice() throws Exception
    {
        Parser p = new com.jclark.xml.sax.Driver();
        p.setDocumentHandler(this);
        p.parse("file:///c:/data/books.xml");
    }

    public void startElement(String name, AttributeList atts) throws SAXException
    {
        if (name.equals("book"))
        {
            String category = atts.getValue("category");
            isFiction = (category != null && category.equals("fiction"));
            if (isFiction) count++;
        }
        content.setLength(0);
    }
}
```



```
public void characters(char[] chars, int start, int len) throws SAXException
{
    content.append(chars, start, len);
}

public void endElement(String name) throws SAXException
{
    if (name.equals("price") && isFiction)
    {
        try
        {
            double price = new Double(content.toString()).doubleValue();
            totalPrice += price;
        }
        catch (java.lang.NumberFormatException err)
        {
            throw new SAXException("Price is not numeric");
        }
    }
    content.setLength(0);
}

public void endDocument() throws SAXException
{
    System.out.println("The average price of fiction books is " +
        totalPrice / count);
}

public static void main (String args[]) throws java.lang.Exception
{
    try
    {
        (new AveragePrice()).determineAveragePrice();
    }
    catch (SAXException err)
    {
        System.err.println("Parsing failed: " + err.getMessage());
    }
}
```

在这段代码中主要有三个要点：

- 应用程序需要保持一些上下文信息，即当前书目是否是科幻书籍。使用一个实体变量记录有关情况，当遇到科幻书目的开始标签时就设置 isFiction 为真，而遇到非科幻书目的开始标签时就设置 isFiction 为假。
- 注意字符内容如何在 Java StringBuffer 里累加，直到 endElement() 事件通知时才被真正处理。这可以一举两得：解决了单个元素内容可能会被分解而分别通知的问题；同时意味着当处理数据时，我们可以知道正在处理哪个元素。任何读到开始和结束标签时 StringBuffer 置为空，即当应用程序遇到 PCDATA 元素（只包含字符数据的元素）的结束标签时，缓冲区里将包含该元素的字符数据。
- 当书目的价格不是一个有效数字时应用程序需要做一些智能判断。（除非 XML 纲要被标准化，不能指望让解析器做这些确认工作：DTD 没有提供在元素中限制字符数据的数据类型

的方法。) 通过将字符串转换为数值的 Java 构造器 Double (String s) 报告一个例外可以发现这种情况。有关代码捕获这个例外, 发布一个描述错误的 SAXException。然后用适当的错误消息终止解析过程。

- 对 XML 实例文件运行代码, 产生下列输出:

```
>java AveragePrice
The average price of fiction books is 10.99
```

但是程序还不是完美无缺的。

首先, 如果输入文档不是期望的结构, 程序很容易运行失败。例如如果 `<price>` 元素在 `<book>` 之外出现, 或者 `<book>` 中没有 `<price>` 元素, 或者 `<price>` 元素有自己的子元素, 程序都会给出错误的答案。这些情况都可能会出现, 因为没有 DTD, 或者是因为使用了不检查 DTD 的非确认解析器, 或者是因为文档被用非期望的其他 DTD 提交, 或者是因为在程序写完之后 DTD 已经被增强了。

其次, 检测到错误时的诊断信息很不友好。用户将被告知某个书目价格不是数值, 但是在列表中可能有成千上百的书目: 明确是哪个书目会更好一些。在一次程序运行中报告所有的错误甚至会更有帮助, 这样用户就不需要为每发现和纠正一个错误去运行一次程序。(实际上大多数 XML 解析器在一次运行中只报告一个语法错误, 这限制了我们目前只能如此)。

在下一节里会看到如何保持更多关于元素上下文的信息, 如果要做更全面的有效性确认, 这些信息是必要的。在此之前, 先改进一下错误处理过程。使用 Locator 对象确定在源文档中错误发生的位置并据此报告。为了清楚地说明处理过程, 我们用 IBM Alphaworks 的 xml4j 解析器代替 James Clark 的 xp 解析器, 它可以提供更清晰的信息。下面是修改过的程序:

#### 程序清单 6-11

```
import org.xml.sax.*;

public class AveragePrice extends HandlerBase
{
    private int count = 0;
    private boolean isFiction = false;
    private double totalPrice = 0.0;
    private StringBuffer content = new StringBuffer();
    private Locator locator;

    public void determineAveragePrice() throws Exception
    {
        Parser p = new com.ibm.xml.parsers.SAXParser();
        p.setDocumentHandler(this);
        p.parse("file:///c:/data/books.xml");
    }

    public void setDocumentLocator(Locator loc)
    {
        locator = loc;
    }

    public void startElement(String name, AttributeList atts) throws SAXException
```

```
{
    if (name.equals("book"))
    {
        String category = atts.getValue("category");
        isFiction = (category!=null && category.equals("fiction"));
        if (isFiction) count++;
    }
    content.setLength(0);
}

public void characters(char[] chars, int start, int len) throws SAXException
{
    content.append(chars, start, len);
}

public void endElement(String name) throws SAXException
{
    if (name.equals("price") && isFiction)
    {
        try
        {
            double price = new Double(content.toString()).doubleValue();
            totalPrice += price;
        }
        catch (java.lang.NumberFormatException err)
        {
            if (locator!=null)
            {
                System.err.println("Error in " + locator.getSystemId() +
                    " at line " + locator.getLineNumber() +
                    " column " + locator.getColumnNumber());
            }
            throw new SAXException("Price is not numeric", err);
        }
    }
    content.setLength(0);
}

public void endDocument() throws SAXException
{
    System.out.println("The average price of fiction books is " +
        totalPrice / count);
}

public static void main (String args[]) throws java.lang.Exception
{
    try
    {
        (new AveragePrice()).determineAveragePrice();
    }
    catch (SAXException err)
    {
        System.err.println("Parsing failed: " + err.getMessage());
    }
}
}
```

这个版本通过一点额外操作改进了诊断信息。修改后的应用程序做了三件事情：

- 程序保存解析器提供的 Locator 对象的注解。
- 当错误发生时，程序在生成 SAXException 之前使用 Locator 对象输出错误定位的有关信息。注意应用程序必须支持没有 Locator 的情况，因为并不要求解析器必须提供 Locator。
- 程序包括最初的“初因”异常的细节信息（NumberFormatException），异常被封装在 SAXParseException 里，也可以写一些更精确的诊断信息。
- 把 Moby Dick 的价格“8.99”改为“A.99”，下面是用 xml4j 解析器得到的输出：

```
>java AveragePrice
Error in file:///c:/data/books.xml at line 16 column 22
Parsing failed: Price is not numeric
```

本例中应用程序在产生异常之前生成了一条包含位置信息的信息，然后当异常在高层被捕获时生成真正的错误消息。另一种方法是把位置信息作为异常的一部分来传递，可以通过产生 SAXParseException 而不是通常的 SAXException 做到这一点。然而应用程序必须处理没有 Locator 的情况，这时产生 SAXParseException 就不是很方便。一个替代的方法是当解析器不提供 Locator 时让应用程序创建自己缺省的 Locator（不包含什么有用信息）。

#### 6. 保持上下文信息

迄今为止从两个例子中可以看出在解析处理过程中 DocumentHandler 一般需要保持一些上下文信息。第一个例子是对元素累加计数；第二个例子是由 DocumentHandler 明确当前是否是在一个含有 category="fiction" 的 <book> 元素里。

几乎所有实际的 SAX 应用程序都需要保持一些此类的上下文信息。它经常适于明确当前元素的嵌套情况，很多情况下也用于了解当前处理数据所有祖先元素的属性。

明显的数据结构是堆栈，因为堆栈很自然地在遇到元素开始标签时添加元素信息，而在遇到元素结束标签时删除元素信息。当然堆栈需要比存储整个文档少得多的内存，因为堆栈中需要的最大条目数就是元素的最大嵌套层数。即使在一个庞大复杂的文档里，嵌套层数也很少超过 10 个左右。

如果更改一下对前面程序例子的要求，我们会看到堆栈是如何发挥作用的。这次允许书籍目录包含多卷书目，每卷有一个价格并且整个集合有一个价格。在计算平均价格时，我们想知道整个集合的价格而不是个别卷的价格。

源文档如下所示（也可以在 <http://www.wrox.com> 站点得到）：

程序清单 6-12

```
<?xml version="1.0"?>
<books>
  <book category="reference">
    <author>Nigel Rees</author>
    <title>Sayings of the Century</title>
    <price>8.95</price>
  </book>
  <book category="fiction">
    <author>Evelyn Waugh</author>
    <title>Sword of Honour</title>
    <price>12.99</price>
```

```

</book>
<book category="fiction">
  <author>Herman Melville</author>
  <title>Moby Dick</title>
  <price>8.99</price>
</book>
<book category="fiction">
  <author>J. R. R. Tolkien</author>
  <title>The Lord of the Rings</title>
  <price>22.99</price>
  <volume number="1">
    <title>The Fellowship of the Ring</title>
    <price>8.95</price>
  </volume>
  <volume number="2">
    <title>The Two Towers</title>
    <price>8.95</price>
  </volume>
  <volume number="3">
    <title>The Return of the King</title>
    <price>8.95</price>
  </volume>
</book>
</books>

```

处理这种情况的方法是在程序里引入另一个标记，当遇到 `<volume>` 开始标签时设置标记，当遇到 `</volume>` 结束标签时取消标记；如果标记被设置了，程序将忽略 `<price>` 元素。但是这种编程风格很快就导致了标记的增多和 if-then-else 条件判断的复杂嵌套。更好的方法是把关于当前打开元素的所有信息放到一个堆栈里，这样就可以按需查询。

下面是新的程序版本：

程序清单 6-13

```

import org.xml.sax.*;
import org.xml.sax.helpers.AttributeListImpl;
import java.util.Stack;

public class AveragePrice1 extends HandlerBase
{
    private int count = 0;
    private double totalPrice = 0.0;
    private StringBuffer content = new StringBuffer();
    private Locator locator;
    private Stack context = new Stack();

    public void determineAveragePrice() throws Exception
    {
        Parser p = new com.jclark.xml.sax.Driver();
        p.setDocumentHandler(this);
        p.parse("file:///c:/data/books1.xml");
    }

    public void setDocumentLocator(Locator loc)
    {

```

```

        locator = loc;
    }

    public void startElement(String name, AttributeList atts) throws SAXException
    {
        ElementDetails details = new ElementDetails(name, atts);
        context.push(details);
        if (name.equals("book"))
        {
            if (isFiction()) count++;
        }
        content.setLength(0);
    }

    public void characters(char[] chars, int start, int len) throws SAXException
    {
        content.append(chars, start, len);
    }

    public void endElement(String name) throws SAXException
    {
        if (name.equals("price") && isFiction() && !isVolume())
        {
            try
            {
                double price = new Double(content.toString()).doubleValue();
                totalPrice += price;
            }
            catch (java.lang.NumberFormatException err)
            {
                if (locator!=null)
                {
                    System.err.println("Error in " + locator.getSystemId() +
                                         " at line " + locator.getLineNumber() +
                                         " column " + locator.getColumnNumber());
                }
                throw new SAXException("Price is not numeric", err);
            }
        }
        content.setLength(0);
        context.pop();
    }

    public void endDocument() throws SAXException
    {
        System.out.println("The average price of fiction books is " +
                           totalPrice / count );
    }

    public static void main (String args[]) throws java.lang.Exception
    {
        (new AveragePrice1()).determineAveragePrice();
    }

    private boolean isFiction()
    {
        boolean test = false;
        for (int p=context.size()-1; p>=0; p--) {

```



```

        ElementDetails elem = (ElementDetails)context.elementAt(p);
        if (elem.name.equals("book") &&
            elem.attributes.getValue("category")!=null &&
            elem.attributes.getValue("category").equals("fiction"))
        {
            return true;
        }
    }
    return false;
}

private boolean isVolume()
{
    boolean test = false;
    for (int p=context.size()-1; p>=0; p--) {
        ElementDetails elem = (ElementDetails)context.elementAt(p);
        if (elem.name.equals("volume"))
        {
            return true;
        }
    }
    return false;
}

private class ElementDetails
{
    public String name;
    public AttributeList attributes;
    public ElementDetails(String name, AttributeList atts)
    {
        this.name = name;
        this.attributes = new AttributeListImpl(atts); // make a copy
    }
}
}

```

下面是期望的输出：

```

>java AveragePrice1
The average price of fiction books is 14.99

```

看起来好像是我们费了好大力气维护这个堆栈而并没有取得多少效果。然而这是值得的。所有实际的应用程序变得越来越复杂，拥有一个支持这种逻辑而不破坏程序结构的数据结构是有益的。注意条件判断，如 `isFiction()` 和 `isVolume()`，现在成为应用于上下文数据结构的方法而不是应用于当事件发生时保持的标记。随着需要判断条件数量的增多，可以编写更多这样的方法而不增加 `startElement()` 和 `endElement()` 方法的复杂度。

### 6.3 SAX的高级特性

目前涉及到的SAX特性对90%的SAX应用程序来说已经足够了。但是了解一些其他的特性是有所帮助的，以备需要时使用。本章节给出了这些特性及其设计目标的概观。

### 6.3.1 可选择的源输入

目前在我们的例子中，被解析的文档是以 URL 的形式描述的。一般情况下，就 URL 可以描述的资源的范围而言，这已经足够了。它允许文档存于本地或远程文件中，或由 Web 服务器动态生成。

#### 1. 从字节流或字符流中获取输入

有时想把由另一个程序生成的 XML 流而不是存放于文件中的 XML 提交给解析器。例如，XML 可能是存放于关系数据库中，或者是一个 EDI 消息翻译程序的输出，或者是嵌入在一个非 XML 格式文件或消息中的 XML 片段。你不想仅仅为了让解析器可以读取你的文档而不得不去编写 XML 文件（或安装 Web 服务器）。

为了处理这种情况，SAX 允许以字符流或字节流的形式提交 XML 输入。SAX 提供了 `InputSource` 类以生成所有可能的输入源。

例如，假设程序想解析存放于一个字符串里的 XML，它刚被通过 JDBC 从关系数据库中读取。下面的代码完成上述工作：

程序清单 6-14

```
public void parseString(String s) throws SAXException, IOException
{
    StringReader reader = new StringReader(s);
    InputSource source = new InputSource(reader);
    parser.parse(source);
}
```

`InputSource` 是 SAX 发行版提供的一个类（而不是一个接口）。应用程序可以设置源输入的多详细信息，这些信息是相互排斥的。包括提供 URL，Reader，InputStream，编码名称或一个“公共标识”。（在 SAX 中的公共标识就像和在 XML 自身规范中一样令人费解：没有什么线索可以确定解析器应该对公共标识真正做些什么。但是后面会看到，应用程序可以使用公共标识。）

为什么 SAX 需要提供两种可选的内存数据，InputStream 和 Reader？

InputStream 是一个字节流。XML 标准对字节流如何转换为统一码字符流提供了许多规则，例如包括编码属性（它是文档内容开始 `<?xml` 声明的一部分）。为把字节转换成字符，完全由标准 Java 库实现这种转换是不够好的，因为它们不理解这些规则，当然也不能指望它们读取编码属性。如果 XML 从二进制源获得，在处理完编码属性后，希望把字节流传递给解析器直接解释。

而 Reader 是一个统一码字符流。如果已经有了字符形式的数据，我们不想仅仅为了解析器能够对它重新解码，而不得不先把它作为字节流进行编码（如 UTF-8 编码所要求）。最好由解析器直接处理字节流。（实际上关于是否希望在 SAX 里提供这种选择还有一些争论。但是它显然很有用处，它不完全符合 XML 规范的精神，XML 规范严格定义了 XML 文档为字节序列。也许最好是不把输入字符流看作是一个 XML 文档，而是当作一个经过预处理的 XML 文档，该文档已经完成了第一步处理，即字节的解码。）

对于是使用字节流还是字符流，有一个值得注意的障碍：解析器无法解析源文档中出现的相对 URL。假设源文档包含下面这样一行：

```
<!DOCTYPE books SYSTEM "books.dtd">
```

到哪里去找books.dtd? XML规范说明(事实上)应该在和源文档相同的目录下,但是当然并没有源文档目录,因为当开始解析时它是在内存中。

SAX通过允许用字节流或字符流提交系统标识(即 URL)避免了这种情况。这个 URL不是用于读取源文档,而只是作为一个用以解析源文档中出现的相对 URL的基础。

## 2. 指定文件名而不是URL

另一个常见的输入源是文件名:例如,命令行接口一般使用文件名而不是 URL作为输入参数,而且你可能想在应用程序接口中也使用这种形式的参数。

SAX InputSource类不直接支持为输入指定一个文件名;你必须把文件名转换成 URL以使解析器能够处理它。如果使用 Java 2就简单了:Java File类有一个相配的方法。要解析文件 c:\sample.xml,你可以写这样的语句:

```
parser.parse((new File("c:\sample.xml")).toURL().toString());
```

(注意parse()方法要求URL是一个字符串而不是一个Java URL对象,因此需要调用toString()实现这种转换。)

对于Java 1.1,如果想让代码能够同时在Windows和UNIX上运行,那么把文件名转换成URL就要比想象中的稍微困难一点,因为有很多种文件名格式。下面的方法能够处理绝大多数格式,尽管错误处理还不完善:

程序清单 6-15

```
public String CreateURL(File file)
{
    String path = file.getAbsolutePath();
    try
    {
        return (new URL(path)).toString();
    }
    catch (MalformedURLException ex)
    {
        String fs = System.getProperty("file.separator");
        char sep = fs.charAt(0);
        if (sep != '/') path = path.replace(sep, '/');
        if (path.charAt(0) != '/') path = '/' + path;
        return "file://" + path;
    }
}
```

## 3. 非XML输入源

SAX使用的更令人惊奇的一种输入方式是把根本不是 XML形式的数据提交给应用程序。只要数据是分层格式并可以被合理地映射到 XML数据模型,你就可以编写一个各种操作类似 XML解析器的驱动程序。驱动程序发送如 startElement()和endElement()的事件给应用程序的 DocumentHandler,就像数据是来源于一个XML文档,实际上并不存在一个要解析的XML文档。

为什么要这样做?它可以利用为接收 XML数据而编写的程序,而不经事先以 XML格式编写数据然后再进行解析这种笨拙的处理过程。例如,如果有一个应用程序,程序是为处理电

子商务的输入XML-EDI消息而设计的，你也许就想编写一个转换程序，能够把比较老的专有格式的消息提交给应用程序处理。转换程序的一种实现方法是创建一个XML文件然后把文件提交给应用程序。但是如果目标应用程序是为使用SAX编写的，有一个巧妙的简便方法，就是转换程序假装是一个XML解析器去直接调用应用程序。

下面关于SAX过滤器的章节将讨论使用这种方法的可能性。

### 6.3.2 处理外部实体

人们经常把在文档文本中出现的像 `&aumlaut;` 形式的标志看作XML实体。这并不完全准确：`&aumlaut;` 严格来说不是一个实体，而是一个实体参照。实体是 `&aumlaut;` 所应用的对象，这就是DTD中的定义，此定义把“`aumlaut`”和它的扩展文本“`ä`”联系起来。

在XML中有许多不同种类的实体，要非常谨慎地明确我们讨论的是哪种实体。如第3章中所提到的，包括表6-3所列各项。

表 6-3

实 体	描 述
字符引用 (Character references)	以数值编码（十进制或十六进制）指定的字符，如 <code>&amp;#xa;</code> 或 <code>&amp;#10;</code> ；（这些并不是严格意义上的实体，为了完备性，在此处包含这种情况）
预定义实体 (Predefined entities)	XML标准中定义的特殊的实体参照，如 <code>&amp;lt;</code> 和 <code>&amp;amp;</code> ；只是唯一一种不需要在DTD中有相应定义（内部或外部）而使用的实体参照
内部实体 (Internal entities)	其扩展文本在DTD中定义（不作为对一些外部存储对象的引用）
外部解析实体 (External parsed entities)	其扩展文本是在一个独立文件中定义的完备的XML形式，该文件是从主XML文档通过系统标识或URL引用的
非解析实体 (Unparsed entities)	包括非XML数据（例如二进制编码图像）：总是外部的。真正的格式可以是以符号形式标识
参数实体 (Parameter entities)	包括DTD而不是文档主体的组成部分
文档实体 (Document entity)	主源XML文档本身就是一个实体
外部文件类型定义 (External DTD)	如果文档应用了一个外部DTD，那么DTD也是一个实体

SAX中处理实体的功能关注于解析对外部实体的引用，外部实体即存放于不同“文件”中的数据——更严格地说，是存放于被系统标识或公共标识识别的容器中。内部实体、字符引用和预定义实体被解析器直接处理，应用程序不能干预它们扩展的方式。

XML中的外部实体总是由一个系统标识来识别（一个URL或更实际的类似于URL的东西），或者也可以是由一个公共标识来识别。公共标识为SGML预先考虑：尽管基于已经建立的SGML惯例有一些约定，XML标准（SAX同样如此）并不真正说明公共标识用来做什么和应该如何使用。

很多情况下，通过解释系统标识或URL来解析外部实体引用的标准规则实际上是不够的。这些情况包括：

- 当实体存放在数据库中（或任何其他不能被URL寻址的地方，例如字处理系统中的短

语库)。

- 当同一个实体引用根据上下文被分别解析。例如，`&currentUser`；实体引用可能被扩展为当前登录用户的名字。
- 当使用多版本系统，同一实体具有多个版本，以及决定在给定情形使用哪个版本的规则。
- 当存在一系列标准实体的许多副本，而且系统考虑性能因素，想找到最近的副本。
- 当实体被公共标识符而不是 URL 引用时。公共标识在 SGML 领域越来越流行而且许多出版商也希望能够继续通过 XML 使用它们。在 SGML 中，公共标识符一般通过被称为目录的查找表映射到实际文件。XML 中没有定义这样的机制，但是 SAX 允许应用程序使用这种机制。

当外部实体不能仅仅通过 URL 被找到，SAX 应用程序应该提供一个实体分解器 (EntityResolver)：即一个实现 `org.xml.sax.EntityResolver` 接口的类。应用程序可以通过调用解析器的 `setEntityResolver()` 方法来为解析器注册实体分解器。

实体分解器只需要实现一个方法：`resolveEntity()`。它被解析器通过两个参数调用：系统标识符（即 URL）和公共标识符。如果在实体声明中没有指定公共标识符，公共标识符设为空值。`resolveEntity()` 方法的任务是返回一个 `InputSource` 对象，解析器将使用这个对象读取外部实体的内容。

在附录 C SAX 规范中有一个实体分解器的简单实例。

#### 非解析实体与注释

SAX 一般来说不会提供给应用程序任何关于 DTD 内容的信息。在 SAX 定义过程中，人们认为绝大多数应用程序都不需要这类信息，因此它被搁置了。（我们将看到 SAX 2.0 在这方面扩展一些可用功能。）

然而，完全禁止对 DTD 内容的访问将使 SAX 应用程序不能处理一些文档，这些文档包含对非解析实体与注释的引用。果真这样的话，有一些很少使用但不能断言如此的 XML 特性，它们仍然为一些人所拥戴。非解析实体允许一个 XML 文档包含对非 XML 对象如二进制图像或声音的引用；非解析注释可以注册和准确识别这些对象的格式。当遇到一个非解析实体时，解析器（按照定义）不会做任何处理，而由应用程序进行解释。但是应用程序只有在能够识别外部实体和注释时才能处理它，因此它需要访问 DTD 中的有关声明。

因此 SAX 接口 `DTDHandler` 实际上只提供关于非解析实体与注释很少而且非常特殊的信息，尽管其名字暗示它可以提供对 DTD 中各种感兴趣对象的访问。如果需要这些信息，你可以像使用其他事件处理接口一样使用 `DTDHandler`：编写一个实现 `org.xml.sax.DTDHandler` 的类，并使用 `DTDHandler()` 方法为解析器注册它。随后解析器会告知 DTD 对非解析实体和注释的声明中使用的系统标识符和公共标识符，然后当在文档主体中遇到对这些对象（以 `ENTITY`，`ENTITIES` 和 `NOTATION` 类型属性的形式）的引用时可以使用这些信息。

但是不要介意 `DTDHandler` 只提供了比其名字所许诺的少的信息！

### 6.3.3 选择解析器

就这个标题，可以分别考虑两个问题：

- 作为设计者，如何决定使用哪个产品？
- 作为程序员，如何把应用程序配置成可以在运行时选择解析器？

第一个问题实际上超出了本书的范围。我们已经列出了一些可用的 SAX 解析器，凭心而论，它们之间的差别很小。它们都是免费的，尽管彼此的许可条件书不同：可以尝试所有产品然后选择你喜欢的。

解析器一般分为两类，由个人开发的和由组织机构开发的。这两类产品一样可靠。由组织机构开发的解析器可能有更好的文档资料和支持，它们也会包括许多辅助特性（如支持中文字符编码或有 COBOL/CICS 接口模块）。如果你恰好需要这些特性，那么它们是很好的选择；如果不需要，它们将浪费硬盘空间和下载时间。

如果想要一个只进行 SAX 解析的解析器，对其他如速度、可靠性和与标准一致性等都不关心，而且你不需要技术支持，那么还有几个比从 <http://www.jclark.com/xp> 获取的 James Clark 的 xp 解析器更好的产品。AElfred (<http://www.microstar.com/aelfred.html>) 是一个小巧的解析器，可以选择它嵌入到自己的应用程序中，特别是当下载时间有重要影响作用的 applet 中。Sun 和 IBM 的解析器可以对不正确的 XML 文件生成更有帮助的诊断信息，所以它们在 XML 编辑环境中比较有用。对于其他解析器，主要考虑它们的运行环境：例如 Oracle 解析器在大量用到 Oracle 产品的应用程序中是显而易见的选择。

实际上保留可选择性是一个不错的主意：你不知道解析器将来会怎么样，而且也不知道应用程序的潜在购买者是否也有如“拒绝无支持的软件”或“拒绝没有法语错误消息的软件”的策略。这就意味着你希望避免用决定性陈述的方式编写应用程序，这把你和你的客户限定于某个特定的产品。例如：

```
Parser p = new com.jclark.xml.sax.Driver();
```

如果是在一个如 CORBA（通用对象请求代理体系结构——参看 <http://www.omg.org>）的分布式对象环境中运行，这种问题正确的结构方法是应用程序授权 Trader 完成寻找解析器的任务，Trader 可以使用各种规则以发现满足运行要求的解析器。可以理解 SAX 的设计者希望避免依赖于这样一个运行时环境，而是给你一些选择：

- 可以使用 SAX 发行版包含的简单帮助类 ParserFactory。应用程序调用静态方法 ParserFactory.makeParser()。它通过读取 org.xml.sax.parser 的系统属性并将其作为一个类名解释。可以通过使用 Java 命令行的 -D 选项设置系统属性，因此可以编写一个命令行脚本从环境变量中设置系统属性。
- 可以实现自己的 Parser 类初始化机制，类名在运行时确定。可以在一个配置文件或 Windows 注册表中保留类名。假设可以把名字当作字符串读取，你就可以使用如下的 Java 语句创建一个 Parser 实例。实际应用中，你需要添加一些错误处理以捕获可能产生的各种例外。

```
String parserName = [*** read name of parser ***];  
Parser p = (Parser) (Class.forName(parserName).newInstance());
```

- 也可以在应用程序中创建已知解析器的列表，然后试着轮流加载它们直至找到一个可以成功加载的解析器。这样就允许你的用户安装类路径上的任何一个解析器，当然不允许使用



一个你所不知道的解析器。

第二种技术的一个例子可以在 `ParserManager` 类中找到，该类可以从 Michael Kay 的 SAXON 包得到（参见 <http://users.iclway.co.uk/mhkay/saxon/>）。该类根据一个叫 `ParserManager.properties`（SAXON 包提供）的配置文件里的信息初始化一个解析器。为了用一个不同的解析器运行应用程序，只需简单编辑配置文件（文件中写明了指令）。`ParserManager` 是可以不依赖于 SAXON 包中其他组件使用的独立类，而且是自由发布的。一旦安装了 `ParserManager`，并且在类路径中包含属性文件，你就可以创建一个 SAX Parser，只需写：

```
Parser = ParserManager.makeParser();
```

将会在后面的例子中实现这一操作。

## 6.4 一些SAX设计模式

上述 SAX 应用程序实例仅关注于处理一两个不同的元素，而且这种处理是很简单的。在实际的应用程序中，需要处理很多个不同的元素类型，这样程序的风格将会迅速地变得非常不结构化了。主要有两个原因：第一，处理相同全局上下文数据的不同事件的交互操作是很难解决的；第二，每个事件处理方法都需要做一些完全独立的操作。

所以为了避免这种情况，需要仔细考虑 SAX 应用程序的设计。本节提供了一些可能的方法。下面看看以下两种常用的模式：筛选模式和基于规则模式。

### 筛选设计模式

筛选设计模式，有时也称为管道模式，其中处理过程的每一步都可以表示为管道的一段：数据流经管道，每段管道筛选流经的数据。如图 6-2 所示。

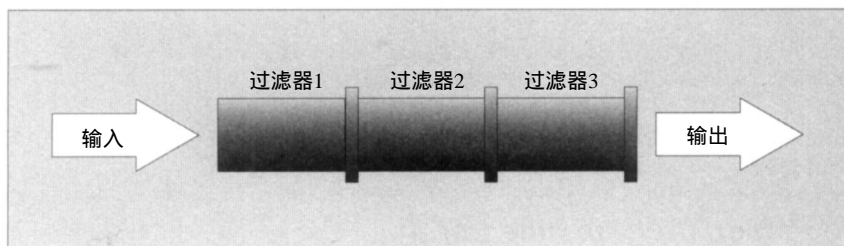


图 6-2

一个筛选器可以做许多不同的事情，例如：

- 除去源文档中不需要的元素。
- 修改标签或属性名。
- 进行确认验证。
- 规格化数据值（如日期）。

筛选模式的重要特性就是每个筛选器都有输入和输出，它们遵循相同的接口。筛选器一端实现了接口，另一端作为同一个接口的客户端。所以任何相邻的筛选器对，左边的相当于解析器 `Parser`，右边的相当于文档处理器 `DocumentHandler`。实际上，这种结构中的筛选器一般同时

实现 SAX 的 Parser 和 DocumentHandler 接口。（在这里说“Parser”，当然可能不太恰当。SAX Parser 的特性不是它可以理解 XML 的文法和语法规则，而是它告知文档处理器发生的事件。任何发出这种通知的程序都可以实现 SAX Parser 接口，即使它可能不做任何实际的解析操作。）

筛选器也可能有多个输出，通知多个接收者发生的事件；相对少见的情况是筛选器有多个输入，从多个数据源汇合事件。

筛选器设计模式的优势是筛选器可以被高度复用，因为就像实际的管道装置一样，相同标准的筛选器可以以很多不同的方式组装到一起。

### 1. ParserFilter 类

有很多工具可以用来创建上述形式的管道。最简单的是 John Cowan 的 ParserFilter 类，可以从 <http://www.ccil.org/~cowan/XML/> 获取。这是一个抽象类：定义了所有筛选器需要的操作，而且允许你为自己管道中每个特定的筛选器定义子类。

正像你所期望的，ParserFilter 同时实现了 SAX Parser 和 DocumentHandler 接口；实际上，它也额外地实现了其他 SAX 事件处理接口（DTDHandler，ErrorHandler 和 EntityResolver）。该类中的所有事件处理方法所做的就是把事件传送给管道中的下一个筛选器；定义的子类可以忽略任何需要加工的方法。

ParserFilter 类有一个构造器可以把 Parser 作为它的参数：结果是创建一段管道并且连接它到其左边的另一段管道。为创建上图里的三段管道，可以编写下列代码：

程序清单 6-16

```
ParserFilter pipeline = new Filter3(  
    new Filter2 (  
        new Filter1 (  
            new com.jclark.xml.sax.Driver())));  
pipeline.setDocumentHandler(outputHandler);
```

管道的初始化输入当然是一个 SAX Parser，而最终输出是一个 SAX DocumentHandler。

### 2. ParserFilter 实例：一个缩进器

下面是一个完全实现 ParserFilter 的实例 Indenter。该筛选器接收 SAX 事件流，通过在开始和结束标签前添加空白格消息数据，以使文档的嵌套结构可视化。然后传递消息化的数据给下一个 DocumentHandler（当然可能是另一个筛选器）。

程序代码应该是自己注释的。注意它是如何依赖于超类中的方法以真正地发送事件到 DocumentHandler：

程序清单 6-17

```
import java.util.*;  
import org.xml.sax.*;  
import org.ccil.cowan.sax.ParserFilter;  
  
/**  
 * Indenter: This ParserFilter indents elements, by adding white space where  
 appropriate.  
 * The string used for indentation is fixed at four spaces.  
 */  
public class Indenter extends ParserFilter {
```

```

private final static String indentChars = "    "; //indent by four spaces
private int level = 0; // current indentation level
private boolean sameline = false; // true if no newlines in
//element
private StringBuffer buffer = new StringBuffer(); // buffer to hold character
//data

/**
 * Constructor: supply the underlying parser used to feed input to this filter
 */

public Indenter(Parser p) {
    super(p);
}

/**
 * Output an element start tag.
 */

public void startElement(String tag, AttributeList atts) throws SAXException
{
    flush(); // clear out pending character data
    indent(); // output white space to achieve indentation
    super.startElement(tag, atts); // output the start tag and attributes
    level++; // we're now one level deeper
    sameline = true; // assume a single line of content
}

/**
 * Output element end tag
 */

public void endElement(String tag) throws SAXException
{
    flush(); // clear out pending character data
    level--; // we've come out by one level
    if (!sameline) indent(); // output indentation if a new line was found
    super.endElement(tag); // output the end tag
    sameline = false; // next tag will be on a new line
}

/**
 * Output a processing instruction
 */

public void processingInstruction(String target, String data) throws
    SAXException
{
    flush(); // clear out pending character data
    indent(); // output white space for indentation
    super.processingInstruction( // output the processing instruction
        target, data);
}

/**
 * Output character data
 */

```

```

public void characters(char[] chars, int start, int len) throws SAXException
{
    buffer.append(chars,          // add the character data to a buffer for now
        start, len);
}

/**
 * Output ignorable white space
 */

public void ignorableWhitespace(char[] ch, int start, int len) throws
                                SAXException
{
    // ignore it
}

/**
 * Output white space to reflect the current indentation level
 */

private void indent() throws SAXException
{
    // construct an array holding a newline
    //character
    // and the correct number of spaces
    int len = indentChars.length();
    char[] array = new char[level*len + 1];
    array[0] = '\n';
    for (int i=0; i<level; i++)
    {
        indentChars.getChars(0, len, array, len*i + 1);
    }
    // output this array as character data
    super.characters(array, 0, level*len+1);
}

/**
 * Flush the buffer containing accumulated character data.
 * White space adjacent to markup is trimmed.
 */

public void flush() throws SAXException
{
    // copy the buffer into a character array
    int end = buffer.length();
    if (end==0) return;
    char[] array = new char[end];
    buffer.getChars(0, end, array, 0);
    // trim white space from the start and end
    int start=0;

    while (start<end && Character.isWhitespace(array[start])) start++;
    while (start<end && Character.isWhitespace(array[end-1])) end--;
    // test to see if there is a newline in the buffer
    for (int i=start; i<end; i++)
    {
        if (array[i]=='\n') {
            sameline = false;

```

```
        break;
    }
}

// output the remaining character data
super.characters(array, start, end-start);
// clear the contents of the buffer
buffer.setLength(0);
}
}
```

为了实际运行这个实例，需要一个输出 XML文档的 `DocumentHandler`；假设存在一个叫做 `XMLOutputter` 的文档处理器（下一节里将展示 `XMLOutputter` 是如何编写的）。然后可以编写如下的主程序：

程序清单 6-18

```
public static void main(String[] args) throws Exception
{
    Indenter app = new Indenter(ParserManager.makeParser());
    app.setDocumentHandler(new XMLOutputter());
    app.parse(args[0]);
}
```

还必须在文件的首部添加对 `ParserManager` 类的导入声明：

程序清单 6-19

```
import java.util.*;
import org.xml.sax.*;
import com.icl.saxon.ParserManager;
import org.ccil.cowan.sax.ParserFilter;
```

为使程序更加实用可行，通过把输入文件作为一个可以在命令行指定的参数（从 `args[0]` 获取），并且通过使用前面介绍的 `ParserManager` 类创建下面的 SAX Parser。它仍然不是一个生产性程序，例如如果没有输入参数，它就会调用失败，但是已经接近一个实用的生产性程序了。一旦设置了类路径（记住使用了 `ParserManager`，`ParserManager.properties` 文件必须也包含在类路径中），就从命令行运行程序，如：

```
java Indenter file:///c:/data/books.xml
```

可以看到输出排版整齐地缩进。因为参数是一个 URL，所以可以格式化任何 Web 上的 XML 文件。

### 3. 管道的终点：生成 XML

如前例所示，经常管道最终的输出会是一个新的 XML 文档。所以常常需要一个 `DocumentHandler`，使用来自管道的事件流生成 XML 文档：一种反向的解析。

奇怪的是不能在 Web 上找到实现这种功能的 `DocumentHandler`，所以在这里我们编写了一个这样的 `DocumentHandler`。

该类如下所示。它相当简明，除了为特殊字符生成实体和字符参照的代码，它使用了一些

Java比较少用到的处理字符串和数组的方法：

程序清单 6-20

```
import org.xml.sax.*;
import java.io.*;

/**
 * XMLOutputter is a DocumentHandler that uses the notified events to
 * reconstruct the XML document on the standard output
 */

public class XMLOutputter implements DocumentHandler
{
    private Writer writer = null;

    /**
     * Set Document Locator. Provided merely to satisfy the interface.
     */

    public void setDocumentLocator(Locator locator) {}

    /**
     * Start of the document. Make the writer and write the XML declaration.
     */

    public void startDocument () throws SAXException
    {
        try
        {
            writer = new BufferedWriter(new PrintWriter(System.out));
            writer.write("<?xml version='1.0' ?>\n");
        }
        catch (java.io.IOException err)
        {
            throw new SAXException(err);
        }
    }

    /**
     * End of the document. Close the output stream.
     */

    public void endDocument () throws SAXException
    {
        try
        {
            writer.close();
        }
        catch (java.io.IOException err)
        {
            throw new SAXException(err);
        }
    }
}
```



```

* Start of an element. Output the start tag, escaping special characters.
*/

public void startElement (String name, AttributeList attributes)
                        throws SAXException
{
    try
    {
        writer.write("<");
        writer.write(name);

        // output the attributes

        for (int i=0; i<attributes.getLength(); i++)
        {
            writer.write(" ");
            writeAttribute(attributes.getName(i), attributes.getValue(i));
        }
        writer.write(">");
    }
    catch (java.io.IOException err)
    {
        throw new SAXException(err);
    }
}

/**
 * Write attribute name=value pair
 */

protected void writeAttribute(String attname, String value) throws
                        SAXException
{
    try
    {
        writer.write(attname);
        writer.write("=");
        char[] attval = value.toCharArray();
        char[] attesc = new char[value.length()*8]; // worst case scenario
        int newlen = escape(attval, 0, value.length(), attesc);
        writer.write(attesc, 0, newlen);
        writer.write("");
    }
    catch (java.io.IOException err)
    {
        throw new SAXException(err);
    }
}

/**
 * End of an element. Output the end tag.
 */

public void endElement (String name) throws SAXException
{
    try
    {

```

```

        writer.write("</" + name + ">");
    }
    catch (java.io.IOException err)
    {
        throw new SAXException(err);
    }
}

/**
 * Character data.
 */

public void characters (char[] ch, int start, int length) throws SAXException
{
    try
    {
        char[] dest = new char[length*8];
        int newlen = escape(ch, start, length, dest);
        writer.write(dest, 0, newlen);
    }
    catch (java.io.IOException err)
    {
        throw new SAXException(err);
    }
}

/**
 * Ignorable white space: treat it as characters
 */

public void ignorableWhitespace(char[] ch, int start, int length)
throws SAXException
{
    characters(ch, start, length);
}

/**
 * Handle a processing instruction.
 */

public void processingInstruction (String target, String data)
                                throws SAXException
{
    try
    {
        writer.write("<?" + target + ' ' + data + ">");
    }
    catch (java.io.IOException err)
    {
        throw new SAXException(err);
    }
}

/**
 * Escape special characters for display.
 * @param ch The character array containing the string
 * @param start The start position of the input string within the character
 * array

```

```

* @param length The length of the input string within the character array
* @param out Character array to receive the output. In the worst case,
* this should be
* 8 times the length of the input array.
* @return The number of characters used in the output array
*/

```

```

private int escape(char ch[], int start, int length, char[] out)
{

```

```

    int o = 0;
    for (int i = start; i < start+length; i++)
    {
        if (ch[i]=='<')
        {
           ("<").getChars(0, 4, out, o); o+=4;
        }
        else if (ch[i]=='>')
        {
           (">").getChars(0, 4, out, o); o+=4;
        }
        else if (ch[i]=='&')
        {
           ("&").getChars(0, 5, out, o); o+=5;
        }
        else if (ch[i]=='"')
        {
           ("&#34;").getChars(0, 5, out, o); o+=5;
        }
        else if (ch[i]=='\''')
        {
           ("&#39;").getChars(0, 5, out, o); o+=5;
        }
        else if (ch[i]<127)
        {
            out[o++] = ch[i];
        }
        else
        {
            // output character reference
            out[o++] = '&';
            out[o++] = '#';
            String code = Integer.toString(ch[i]);
            int len = code.length();
            code.getChars(0, len, out, o); o+=len;
            out[o++] = ';';
        }
    }

```

```

    return o;
}

```

```

}

```

现在可以看到SAX是如何像读取XML文档一样编写XML文档。实际上，可以接连运行SAX：代替Parser这种由其他人编写的标准软件，而 DocumentHandler是特定的应用代码，可以编写 org.xml.sax.Parser的实现，其中包含你生成XML的应用逻辑，使其和非标准的 DocumentHandler

配合编写XML文档输出。

#### 4. 其他ParserFilter

下面是一些其他有用的ParserFilter。

##### (1) NamespaceFilter

该ParserFilter实现在第7章中描述的XML命名空间建议稿。可以从John Cowan的<http://www.ccil.org/~cowan/XML/>站点获取此建议稿。

SAX在XML命名空间建议稿发布之前就定义了，因此没有考虑到命名空间。如果一个元素名在源文档里以<html:table>形式书写，那么传递给startDocument()方法的元素名将是“html:table”。应用程序很难确定“html”引用的是哪个命名空间。

NamespaceFilter解决了这个问题。NamespaceFilter跟踪文档中所有的命名空间的声明（即“xmlns:xxx”属性），当带前缀的元素或属性名称被SAX解析器通告，NamespaceFilter在把它沿管道继续传递之前，用整个命名空间URL代替元素前缀。例如，如果元素开始标签是<html:table xmlns:html=“http://www.w3.org/TR/REC-html40”>，那么传递给下一个DocumentHandler的元素名将是“http://www.w3.org/TR/REC-html40^table”。“^”用来把命名空间URL和元素的本地部分分开，因为“^”不能出现在URL或XML名中。

有时应用程序不仅需要知道命名空间URL，同样也希望知道前缀信息（例如，用在错误消息中）。NamespaceFilter不提供此类信息，但是它很容易扩展以提供此类信息。

##### (2) InheritanceFilter

InheritanceFilter也可以从John Cowan的站点<http://www.ccil.org/~cowan/XML/>获取。

很多XML文档设计使用可继承属性的概念。这种思想是如果元素的特定属性没有出现，则其值从其包容元素的同一属性中获取。XML标准本身采用了这种思想，对特殊的属性xml:lang和xml:space，这种思想在一些其他标准如XSL格式化对象建议中也被采用。

InheritanceFilter是一个ParserFilter，它通过包含那些元素中没有真正出现但是从父元素中继承的属性，扩展了属性列表。扩展了的属性列表传递给startElement()方法进行处理。InheritanceFilter需要用被作为继承属性处理的属性名列表进行修剪。

##### (3) XLinkFilter

XLinkFilter提供对XLink规范草稿关于在XML文档间创建超链接的支持。Simon St.Laurent在<http://www.simonstl.com/projects/xlinkfilter/>站点发布了XLinkFilter。

不像大多数的ParserFilter，XLinkFilter不加改变地传递所有的事件。然而在此之中，XLinkFilter创建了一个数据结构以反映文档中遇到的XLink属性。在管道的后续步骤中可以查询该数据结构。

XLink规范中定义的一种链接是所谓的“包含”链接，链接文本被设计用于在主文档中按行显示——很像C中的预处理#include命令。相应的XLink语法是show=“parsed”。这很类似于一个外部实体参照，除了应用程序可以控制决策是否和何时包含链接文本：例如，用户可能在显示文档的长短形式上做选择。当然，可以实现一个直接扩展这种链接的筛选器，提交包含文档给管道的后续步骤，就像包含文档是物理地嵌入到源文档中一样。

##### (4) 共享上下文的管道

有关管道的一个潜在的问题是其中的每个筛选器必须自己处理其他筛选器已经识别的东西；一个常见的例子是当前元素的父元素。如果一个筛选器已经维持了一个确定此类信息的元素堆栈，那么其他的筛选器重复同样的工作是一种浪费。

可以通过允许筛选器访问以前筛选器创建的数据结构，或者直接通过公共方法，解决这个问题。但是这需要管道中的筛选器彼此之间了解比单纯管道模型更多的信息，这降低了任意组装筛选器的可能性。可以论证，当处理达到这种复杂程度时，最好完全不使用基于事件的处理方法，而是使用DOM（采用导向性设计模式）。

#### 5. 基于规则的设计模式

创建SAX应用程序的另一种方式是基于规则的方法，它的目标也是功能分割，结构模块化和简单化。

一般情况下，基于规则的程序使用一种“事件-条件-操作”的模型：程序包含形如“如果事件在此条件下发生，则执行操作”的规则集合。基于规则编程因此可以被看作是基于事件编程的自然扩展。

XSL 处理模型（第9章中将讨论）可以被看作是一个基于规则编程的例子。所有 XSL模板制定一个规则：事件处理源文档中的节点；条件控制激活哪个模板的模式；操作是模板的主体部分。可以使用SAX应用程序中相同的概念。

图6-3说明了一个基于规则 SAX应用程序的结构。从XML解析器得到的输入传到—个开关，开关根据已经定义的条件分析事件，决定激活哪个动作。然后操作被传递给设计用于实现特定任务的处理模块。

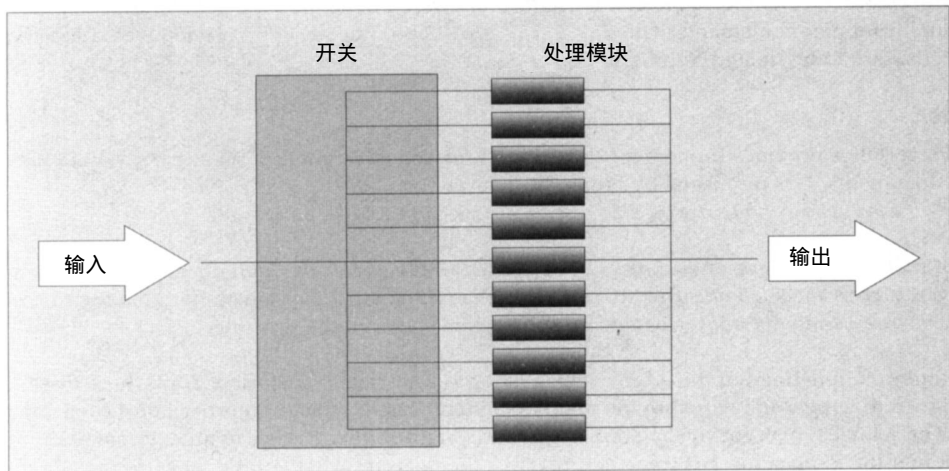


图 6-3

可以实现各种类型的条件和操作，但是下面描述一个非常简单的实现，它的条件只基于一个元素类型。

首先，需要编写 DocumentHandler。因为它的工作是把处理操作切换到处理特定元素类型的一段代码，所以可以称它为开关器（Switcher）。

开关器所做的是维护一组哈希表 (Hashtable) 形式的规则。规则集合按照元素类型索引。应用程序可以指定一个叫做 `ElementHandler` 的类处理特定的元素类型。当解析器报告一个元素开始标签, 相应的 `ElementHandler` 放在规则集合里并被调用以处理这个开始标签。同时, `ElementHandler` 在堆栈里记录, 这样相同的 `ElementHandler` 可以用于处理元素的结束标签和元素中直接出现的字符数据。

整个开关器的代码如下:

程序清单 6-21

```
import org.xml.sax.*;
import java.util.*;

/**
 * Switcher is a DocumentHandler that directs events to an appropriate element
 * handler based on the element type.
 */
public class Switcher extends HandlerBase
{
    private Hashtable rules = new Hashtable();
    private Stack stack = new Stack();

    /**
     * Define processing for an element type.
     */
    public void setElementHandler(String name, ElementHandler handler)
    {
        rules.put(name, handler);
    }

    /**
     * Start of an element. Decide what handler to use, and call it.
     */
    public void startElement (String name, AttributeList atts) throws
        SAXException
    {
        ElementHandler handler = (ElementHandler)rules.get(name);
        stack.push(handler);
        if (handler!=null)
        {
            handler.startElement(name, atts);
        }
    }

    /**
     * End of an element.
     */
    public void endElement (String name) throws SAXException
    {
        ElementHandler handler = (ElementHandler)stack.pop();
        if (handler!=null)
```



```

        {
            handler.endElement(name);
        }
    }

    /**
     * Character data.
     */

    public void characters (char[] ch, int start, int length) throws SAXException
    {
        ElementHandler handler = (ElementHandler)stack.peek();
        if (handler!=null)
        {
            handler.characters(ch, start, length);
        }
    }
}

```

ElementHandler有点类似于DocumentHandler，但是它只是处理了事件的子集：元素的开始和结尾，以及字符数据。所以尽管在这里可以使用DocumentHandler，我们仍定义了一个特定类。这可以作为接口的定义使用，也可以作为实际元素处理器的超集使用：良好的Java编程实践可能建议使用一个独立的接口类，但是现在可以这样做：

程序清单 6-22

```

import org.xml.sax.*;

/**
 * ElementHandler is a class that process the start and end tags and
 * character data
 * for one element type. This class itself does nothing; the
 * real processing should
 * be defined in a subclass
 */

public class ElementHandler {

    /**
     * Start of an element
     */

    public void startElement (String name, AttributeList atts) throws
                                SAXException {}

    /**
     * End of an element
     */

    public void endElement (String name) throws SAXException {}

    /**
     * Character data
     */
}

```

```

        public void characters (char[] ch, int start, int length) throws
                                SAXException {}
    }

```

到此为止，这是一个完整的通用模型。可以对各种类型的文档使用 `Switcher` 和 `ElementHandler` 类进行各种处理操作。现在在一个实际的应用程序中使用它们：我们想要生成一个 HTML 页面显示从书目列表选择的数据。

下面是相应的应用程序。我们将从主控制结构开始。应用程序创建了一个 `Switcher` 并注册了几个 `ElementHandler` 类以处理在输入 XML 文档中的特定元素。然后它创建一个 `Parser`，指定 `Switcher` 为 `DocumentHandler`，并进行解析：

程序清单 6-23

```

import org.xml.sax.*;
import com.icl.saxon.ParserManager;

public class DisplayBookList
{
    public static void main (String args[]) throws Exception
    {
        (new DisplayBookList()).go(args[0]);
    }

    public void go(String input) throws Exception
    {
        Switcher s = new Switcher();
        s.setElementHandler("books", new BooklistHandler());
        s.setElementHandler("book", new BookHandler());
        s.setElementHandler("author", new AuthorHandler());
        s.setElementHandler("title", new TitleHandler());
        s.setElementHandler("price", new PriceHandler());
        s.setElementHandler("volume", new VolumeHandler());
        Parser p = ParserManager.makeParser();
        p.setDocumentHandler(s);
        p.parse(input);
    }

    //...rest of code goes in here...
}

```

实际的元素处理器可以被定义为 `DisplayBookList` 类中的一个内部类：这有助于它们共享对数据的访问。

处理最外面的元素 “book” 的 `ElementHandler` 创建一个 HTML 页面框架：

程序清单 6-24

```

private class BooklistHandler extends ElementHandler
{
    public void startElement(String name, AttributeList atts)
    {

```

```

        System.out.println("<html>");
        System.out.println("<head><title>Book List</title></head>");
        System.out.println("<body><h1>A List of Books</h1>");
        System.out.println("<table>");
        System.out.println("<tr><th>Author</th>");
        System.out.println("<th>Title</th><th>Price</th></tr>");
    }

    public void endElement(String name)
    {
        System.out.println("</table></body></html>");
    }
}

```

处理多个“book”元素的ElementHandler开始并结束生成的HTML表中的行，并初始化一些变量以存放数据：

程序清单 6-25

```

private String author;
private String title;
private String price;
private boolean inVolume;

private class BookHandler extends ElementHandler
{
    public void startElement(String name, AttributeList atts)
    {
        author = "";
        title = "";
        price = "";
        inVolume = false;
    }

    public void endElement(String name)
    {
        System.out.println("<tr><td>" + author + "</td>");
        System.out.println("<td>" + title + "</td>");
        System.out.println("<td>" + price + "</td></tr>");
    }
}

```

最后，处理<book>元素中字段的元素处理器更新存放数据的本地变量。为了使程序结构清晰，在这里没有考虑性能问题——使用StringBuffers变量程序的性能会好于使用Strings变量。

程序清单 6-26

```

private class AuthorHandler extends ElementHandler
{
    public void characters (char[] chars, int start, int len)
    {
        author = author + new String(chars, start, len);
    }
}

```

```

}

private class TitleHandler extends ElementHandler
{
    public void characters (char[] chars, int start, int len)
    {
        if (!inVolume)
        {
            title = title + new String(chars, start, len);
        }
    }
}

private class PriceHandler extends ElementHandler
{
    public void characters (char[] chars, int start, int len)
    {
        if (!inVolume)
        {
            price = price + new String(chars, start, len);
        }
    }
}

private class VolumeHandler extends ElementHandler
{
    public void startElement(String name, AttributeList atts)
    {
        inVolume = true;
    }

    public void endElement(String name)
    {
        inVolume = false;
    }
}

```

inVolume标志用来跟踪当前元素是否包含于一个 <volume>包容元素中，在这种情况下，它是可以忽略的。一旦把所有的代码汇合在一起（完整的代码可以在 <http://www.wrox.com>找到），可以用下面的命令运行程序处理一个 XML实例文件：

```
>java DisplayBookList file:///c:/data/books2.xml
```

输出将如下所示：

程序清单 6-27

```

<html>
<head><title>Book List</title></head>
<body><h1>A List of Books</h1>
<table>
<tr><th>Author</th><th>Title</th><th>Price</th></tr>
<tr><td>Nigel Rees</td>
<td>Sayings of the Century</td>
<td>8.95</td></tr>

```

```
<tr><td>Evelyn Waugh</td>
<td>Sword of Honour</td>
<td>12.99</td></tr>
<tr><td>Herman Melville</td>
<td>Moby Dick</td>
<td>8.99</td></tr>
<tr><td>J. R. R. Tolkien</td>
<td>The Lord of the Rings</td>
<td>22.99</td></tr>
</table></body></html>
```

可以按你所需加工细化该设计模式。可以在以下方面进行优化提高：

- 允许元素处理器访问包含它们上下文细节信息的堆栈。
- 根据条件而不仅仅根据元素名称选择元素处理器。
- 通过允许元素处理器把事件传送给另一个 DocumentHandler，把事件处理器当作管道的一部分使用。

该设计模式的优点是避免了许多 if-then-else 程序语句。在每引入一个新的元素类型时，它不需要更改 DocumentHandler 以添加条件逻辑。取而代之的是只需注册另外一个元素处理器。

## 6.5 SAX 2.0

SAX 1.0 已经被非常普遍地实现，并且几乎从 1998 年 1 月 12 日第一个草稿发布那天起，即比 XML 1.0 的最终建议稿早一个月，它已经被广泛地使用。它很好地满足了用户的需求，尽管也有一些批评意见，其中有些在本章提到了。

所以后续版本 SAX 2.0 的开发就是自然而然了，它的开发相对来说不是那么紧迫。在 1999 年的前几个月，XML-DEV 邮件列表讨论了有关的需求情况，而且 David Megginson 在 1999 年 6 月 1 日发布修改规范的测试版本（尽管没有广为宣传通告）。基本上取得了一致意见，看起来 SAX 2.0 的最终规范接近于它现在的形式，可以在 <http://www.megginson.com/SAX/SAX2/> 站点找到。

该规范是否能被广泛地支持是另一个问题。时间会证明一切。

对原 SAX 接口的扩展方法本身是很有意思的。已经定义了一个标准机制，允许应用程序要求解析器支持特定的特性或设置特定的属性；解析器在任何情况下都可以拒绝该要求。特性和属性的集合特性和属性可以在任何时候被任何人提交产生。为了达到这一点，特性和属性是用 URL 来标识，很类似于 XML 命名空间的标识方式。

### 6.5.1 可配置的接口

SAX2 中主要的新接口是 Configurable 接口。一个 SAX2 解析器必须像实现 `org.xml.sax.Parser` 接口一样实现 `org.xml.sax.Configurable`。Configurable 接口包含四个方法（参见表 6-4）。

在各种情况下，如果解析器不能识别特性或属性名，它就必须产生一个 `SAXNotRecognizedException`。这通常意味着应用程序不能确认解析器是否支持该特性。如果解析器识别了特性或属性名，但不能把它设置为指定值，它就必须产生一个 `SAXNotSupportedException`。

更具体地说，考虑一个新的称为 <http://xml.org/sax/features/validation> 的核心特性。该特性用来修复 SAX 1.0 在应用程序不能发现或控制解析器是否有效时的问题。在 SAX2.0 中，如果特性

是设置为打开的，解析器必须验证 XML 文档；如果是关闭的，解析器必须不做验证（换句话说，只要文档是规范正确的，解析器必须能够正常工作）。

表 6-4

方 法	描 述
<code>getFeature(featureName)</code>	允许应用程序查询解析器它是否支持一个特定的特性
<code>setFeature(featureName,boolean)</code>	允许应用程序要求解析器开关一个特定的特性
<code>getProperty(featureName)</code>	允许应用程序获取一些特定属性的当前值
<code>setProperty(featureName,object)</code>	允许应用程序用提供的值设置一些特定的属性

一个明确地要求解析器进行验证的应用程序可以进行下列调用：

```
parser.setFeature("http://xml.org/sax/features/validation", true);
```

这是一个核心属性，所以所有的 SAX2 解析器应该能够识别该属性名。可以进行验证的解析器将正确地返回值，而不能进行验证的解析器将产生一个 `SAXNotSupportedException`。

同样，一个明确地要求解析器不进行验证的应用程序可以进行下列调用：

```
parser.setFeature("http://xml.org/sax/features/validation", false);
```

这时，坚持进行验证的解析器必须用一个 `SAXNotSupportedException` 响应这个请求。

另一方面，仅仅想知道解析器是否进行验证的应用程序可以进行下列调用：

```
if (parser.getFeature("http://xml.org/sax/features/validation")) ...
```

## 6.5.2 核心的特性和属性

表6-5是在SAX2中定义的核心特性和属性。特性只是 Boolean 值属性的简称。

表 6-5

名称(前缀为http://xml.org/sax)	值	含 义
<code>/features/validation</code>	boolean	进行验证
<code>/features/external-general-entities</code>	boolean	扩展一般的（解析过的）外部实体
<code>/features/external-parameter-entities</code>	boolean	扩展外部 DTD 子集和外部参数实体
<code>/features/namespace</code>	boolean	处理命名空间声明。带前缀的元素名和属性名将使用命名空间的 URI 代替前缀
<code>/features/normalize-text</code>	boolean	通过确保字符数据的所有连续片段在 <code>characters()</code> 方法的一个调用中被传递，以规格化字符数据
<code>/features/use-locator</code>	boolean	通过调用 <code>setDocumentLocator()</code> 方法提供给应用程序一个 <code>Locator</code> 对象
<code>/properties/namespace-sep</code>	String	当命名空间特性被设置时，被用于 URI 和名称本地部分之间的分隔符
<code>/properties/dom-node</code>	<code>org.w3c.dom.Node</code>	只读属性：如果源文档的 DOM 存在于内存中，该属性识别和当前事件有关的 DOM 节点



(续)

名称(前缀为http://xml.org/sax)	值	含 义
/properties/xml-string	String	只读属性：对当前事件进行XML方式表达的字符串
/handlers/DeclHandler	org.xml.sax.misc. DeclHandler	设置一个处理器处理DTD中出现的元素和属性声明
/handlers/LexicalHandler	org.xml.sax.misc. LexicalHandler	设置一个处理器处理词法事件，包括CDATA项，实体和注释
/handlers/NamespaceHandler	org.xml.sax.misc. NamespaceHandler	设置一个处理器处理命名空间声明

SAX2中的核心属性包括三个新的事件处理接口：特性、属性和处理器。（然而，记住“核心”仅仅指每个解析器必须识别这些特性的请求；解析器仍然可以拒绝请求。）

声明处理器，DeclHandler，满足了访问DTD中结构化定义的需求。作为应用程序必须解析的一个字符串，DeclHandler提供对元素声明的访问，而且这种访问是可能的最简单的方式。

词法处理器，LexicalHandler，满足了另一种访问需求，这种访问是针对SAX 1.0中禁止的信息，因为这些信息被认为是应用程序不感兴趣的。这包括内部实体的界限，CDATA项的界限和注释的存在信息。因为允许应用程序尽可能减少在文档被复制时对文档的更改，很多应用程序编写者需要这些特性信息。同样，基于其他一些原因也需要注释信息：例如，XSLT建议稿允许一个样式指示应该对源文档中注释的操作，所以用SAX接口编写的XSLT解释器需要访问这些信息。

命名空间处理器，NamespaceHandler，满足了比命名空间特性更高级的命名空间处理的需求。命名空间特性仅仅使用目前有效的命名空间定义扩展元素和属性前缀，相对而言，命名空间处理器允许命名空间定义本身以它们自己的名义作为事件进行处理。在下面一些情况中，这会有所帮助：

- 应用程序在上下文中而不是元素名和属性名中使用前缀（例如，可能在属性值中使用前缀）。
- 应用程序需要知道使用的前缀（例如，用于错误消息中，或试图复制部分源文档时）。

如前面说明的，SAX 2.0规范还不能认为是稳定的规范，所以即使找到一个支持SAX 2.0的解析器，也要小心使用该解析器。

## 6.6 小结

本章提供了关于SAX接口起源的一些信息，SAX接口被大多数解析器实现支持。

基于事件是SAX的特性和有别于DOM接口的地方。本章讨论了一些导致你使用一个基于事件的接口而不是DOM的因素。

本章讲述了一个简单的SAX应用程序的结构，以及三个主要类之间的关系：应用程序、解析器和文档处理器。本章给出了几个程序例子，说明如何使用这些类编写SAX应用程序。

本章给出了SAX应用程序的一些重要的设计模式，特别是筛选器、管道模式和基于规则

模式。

最后，本章还介绍了有望在稳定的 SAX 2.0 规范中出现的特性。

最后应该提醒一句。本章给出的所有程序例子也可以用 XSLT 更加容易地编写，我们将在第 9 章里讲述 XSLT。当然这并不意味着不需要 SAX 了：Java 应用程序可以做很多 XSL 不能做的工作——例如，加载数据到关系数据库；而且 Java 应用程序一般要快得多。但是在决定使用 SAX 之前，仔细地考虑你要解决的问题是很有必要的，因为很多情况下，使用 XSL 方法，或用 XSL 做预处理的混合方法可能会更好一些。