# Introduction t JAXP

Java API for XML Processing (JAXP) 是使用以Java编程语言编写的应用程序处理XML数据。JAXP使用解析器标准Simple API for XML Parsing（SAX）和文档对象模型（DOM），以便你可以选择以事件流的或者构建表示对象的方式来解析数据。JAXP还支持可扩展样式表语言转换（XSLT）标准，使您能够控制数据的呈现，并使您能够将数据转换为其他XML文档或其他格式，例如HTML。JAXP还提供了命名空间支持，允许您使用可能具有命名冲突的DTD。最后，从版本1.4开始，JAXP实现了Streaming API for XML（StAX）标准。

为了设计上的最大灵活性，JAXP允许您在应用程序中使用任何兼容XML的解析器。这是通过被称作一种可插拔层来实现的，可以让您插入SAX或DOM API的实现。可插拔层还允许您插入XSL处理器，让您控制XML数据的显示方式。

## Overview of the Packages

SAX和DOM API分别由XML-DEV组织和W3C定义。定义这些API的库如下所示：

* javax.xml.parsers: JAXP API，为不同厂商的SAX和DOM解析器提供了一个通用接口。
* org.w3c.dom: 定义Document类（一个DOM）以及DOM的所有组件的类。
* org.xml.sax: 定义基本的SAX API。
* javax.xml.transform: 定义可将XML转换为其他表单的XSLT API。
* javax.xml.stream: 提供StAX指定转换API。

Simple API for XML(SAX)是一个事件驱动，序列访问的机制，它以一个元素接一个元素的方式进行处理。该级别的API将XML读取并写入数据存储库或Web中。对于服务器端和高性能应用程序，您需要充分了解此级别。但是对于大多数应用来说，简单的理解就足够了。

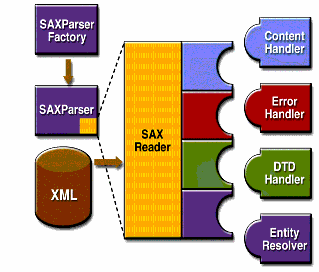
DOM API是一个更容易使用的API。它提供了一个熟悉的对象树结构。您可以使用DOM API来操纵其封装的应用程序对象的层次结构。DOM API是交互式应用程序的理想选择，因为整个对象模型都存在于内存中，这样用户可以访问和操作DOM对象。

另一方面，构建DOM需要读取整个XML结构并将对象树保存在内存中，因此它是CPU和内存密集型的。基于这个原因，SAX API倾向用于服务器端应用程序和数据过滤器，它不需要在内存中表示数据。

在javax.xml.transform中定义的XSLT API可以将XML数据写入文件或将其转换为其他形式。如本教程的XSLT部分所示，您甚至可以将其与SAX API结合使用，将旧数据转换为XML。

最后，javax.xml.stream中定义的StAX API提供了一个基于Java技术的流形式，事件驱动，pull式解析的API，用来读取和写入XML文档。**StAX提供比SAX更简单的编程模型和比DOM更有效的内存管理。**

## Simple API for XML APIs

AX解析API的基本概要如图1-1所示。要想启动处理，需要一个SAXParserFactory类的实例用于生成解析器的实例。

解析器包装一个SAXReader对象。当解析器的parse()方法被调用时，reader调用应用程序中实现的几个回调方法中的一个。这些回调方法由接口ContentHandler，ErrorHandler，DTDHandler和EntityResolver定义。

以下是SAX API的关键摘要：

SAXParserFactory

SAXParserFactory对象创建解析器的实例，它是由系统属性javax.xml.parsers.SAXParserFactory决定的。

SAXParser

SAXParser接口定义了几种parse()方法。 通常，需要传递给解析器一个XML数据源和一个DefaultHandler对象，用来处理XML并在handler对象中调用适当的方法。

SAXReader

SAXParser包装一个了SAXReader。通常情况下，您并不需要关注这一点，但是当你需要获取它的时候，使用SAXParser的getXMLReader()，这样您可以对它进行配置。实际上是SAXReader负责与你定义的SAX事件handler进行对话。

DefaultHandler

图中未显示，DefaultHandler实现了ContentHandler，ErrorHandler，DTDHandler和EntityResolver接口（实现都是空方法），这样你可以只需要重写你需要的方法。

ContentHandler

当识别XML标签时，会调用诸如startDocument，endDocument，startElement和endElement之类的方法。该接口还定义了character()和processingInstruction()方法，当解析遇到XML元素中的文本或内联处理指令时会调用这两个方法。

ErrorHandler

当解析遇到各种错误时，会调用error()，fatalError(),和warning()方法。默认的错误handler会为发生的致命错误抛出异常，但是忽略其他错误（包括验证错误）。这是您需要了解有关SAX解析器的一个原因，即使您正在使用DOM。有时，应用程序可能能够从验证错误中恢复。其他时候，它可能需要产生一个异常。为了确保正确的处理，您需要将自己实现的错误处理handler提供给解析器。

DTDHandler

你通常不会被要求使用它定义的方法。这时用于处理DTD以识别和对未解析实体的声明采取行动。

EntityResolver

当解析器必须辨别由URI标识的数据时，会调用resolveEntity方法。在大多数情况下，URI只是一个URL，用于指定文档的位置，但在某些情况下，文档可能由网络空间中唯一的URN（公共标识符或名称）标识。除了URL之外，还可以指定公共标识符。然后，EntityResolver可以使用公共标识符而不是URL来查找文档，例如，如果文档存在，则访问文档的本地副本。

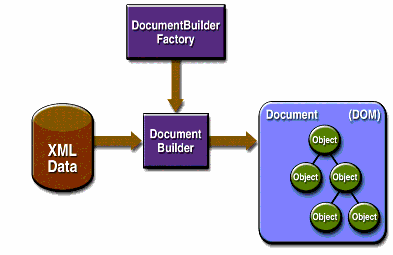
一个典型的应用程序至少实现了大部分ContentHandler方法。因为这个接口的默认实现DefaultHandler会忽略除致命错误之外的所有输入，所以一个健壮的实现也可能需要实现ErrorHandler方法。

### SAX Packages

SAX解析器在下表中列出的包中定义。

|  |  |
| --- | --- |
| **Packages** | **Description** |
| org.xml.sax | 定义了SAX接口。org.xml是由定义了SAX API的组织确定的包前缀名称。 |
| org.xml.sax.ext | 定义用于执行更复杂的SAX处理的SAX扩展，例如处理文档类型定义（DTD）或查看文件的详细语法。 |
| org.xml.sax.helpers | 包含一些使其更容易使用SAX帮助类。比如，通过定义了一个实现所有接口并且都是空方法的默认handler，以便您只需要重写那些实际要实现的方法。 |
| javax.xml.parsers | 定义SAXParserFactory类，返回SAXParser。还定义了用于报告错误的异常类。 |

## Document Object Model APIs

下图显示了正在运行的DOM API。 图DOM API

您可以使用javax.xml.parsers.DocumentBuilderFactory类获取DocumentBuilder实例，并使用该实例来生成符合DOM规范的Document对象。事实上，您获得的builder由系统属性javax.xml.parsers.DocumentBuilderFactory决定，该属性选择用于生成builder的工厂类实现（java平台的默认值可以用命令行覆盖重写）。

您还可以使用DocumentBuilder newDocument（）方法来创建一个实现org.w3c.dom.Document接口的空Document 对象。或者，您可以使用其中一个builder的解析方法从现有XML数据创建一个Document对象。方法会返回一个如上图所示的DOM树。

注意-虽然它们被称为对象，但DOM树中的实体实际上是相当低级的数据结构。例如，考虑这个结构：<color> blue </ color>。这是一个color标签的元素节点，并且在它里面有一个包含数据blue的文本节点。这个问题将在本教程的DOM课程中详细探讨，但是开发人员期望返回的对象，会惊讶的发现当在元素节点上调用getNodeValue（）方法却不会返回任何内容。对于一个真正面向对象的树，请参阅http://www.jdom.org上的JDOM API。

### DOM Packages

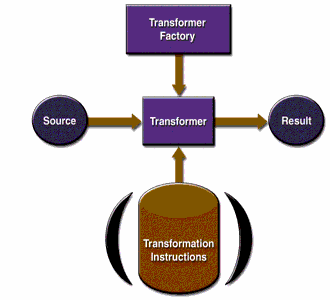
文档对象模型实现在下表中列出的包中定义。

|  |  |
| --- | --- |
| **Package** | **Description** |
| org.w3c.dom | 定义W3C指定的XML（和可选的HTML）文档的DOM编程接口。 |
| javax.xml.parsers | 定义DocumentBuilderFactory类和DocumentBuilder类，它返回一个实现W3C Document接口的对象。用于创建builder的factroy由javax.xml.parsers系统属性决定，该属性可以在调用新的实例方法时从命令行设置或覆盖。此包还定义了用于报告错误的ParserConfigurationException类。 |

## Extensible Stylesheet Language Transformations APIs

下图显示了XSLT API的操作

TransformerFactory对象被实例化并用于创建Transformer。Source对象是转换过程的输入。Source对象可以由SAX reader，DOM或从输入流创建。



类似地，Result对象是转换过程的结果。该对象可以是SAX事件处理程序，DOM或输出流。

当创建transformer时，可以从一组转换指令中来创建transformer，在这种情况下，执行指定的转换。如果不是由任何特定的指令来创建，那么transformer对象只是简单的把source复制到result中。

### XSLT Packages

|  |  |
| --- | --- |
| **Package** | **Description** |
| javax.xml.transform | 定义TransformerFactory和Transformer类，您可以使用它来获取能够进行转换的对象。创建transformer 对象后，调用其transform（）方法，并且为其提供一个输入（源）和一个输出（结果）。 |
| javax.xml.transform.dom | 从DOM创建输入（源）和输出（结果）对象的类。 |
| javax.xml.transform.sax | 从SAX解析器创建输入（源）对象并从SAX事件handler创建输出（结果）对象的类。 |
| javax.xml.transform.stream | 从I / O流创建输入（源）对象和输出（结果）对象的类。 |

## Streaming API for XML APIs

StAX是JAXP系列中最新的API，为希望能进行高性能流式过滤，处理和修改的开发人员提供一个SAX，DOM，TrAX和DOM的替代方案，特别是对于低内存和有限的可扩展性要求。

总之，StAX为流式XML处理提供了一个标准的双向pull式解析器接口，提供比SAX更简单的编程模型和比DOM更有效的内存管理。StAX使开发人员以事件形式解析和修改XML流，并扩展XML信息模型以允许特定于应用程序的添加。

### StAX Packages

|  |  |
| --- | --- |
| **Package** | **Description** |
| javax.xml.stream | 定义XMLStreamReader接口，用于迭代XML文档的元素。 XMLStreamWriter接口指定如何写入XML。 |
| javax.xml.transform.stax | 提供特定于StAX的转换API。 |

## Where Do You Go From Here?

* 如果数据结构已经确定，并且正在编写需要快速处理的服务器应用程序或XML过滤器，使用Simple API for XML.
* 如果您需要从XML数据构建一个对象树，以便您可以在应用程序中进行操作，或将内存中的对象树转换为XML，使用Document Object Model.
* 如果您需要将XML标签转换为其他形式，如果要生成XML输出，或者（与SAX API结合使用）想要将旧数据结构转换为XML，使用Extensible Stylesheet Language Transformations.
* 如果您想要一种基于Java技术的流形式的，基于事件驱动的pull式解析API来读取和写入XML文档，或者想要创建快速，相对易于编程和轻量级内存占用的双向XML解析器，使用Streaming API for XML.

# Simple API for XML

本课程重点介绍用于访问XML文档的Simple API for XML（SAX），一种事件驱动的序列访问机制。该协议常被需要传输和接收XML文档的servlet和面向网络的程序所使用，因为它是目前可用于处理XML文档的最快和最少的内存占用的机制，而不是Streaming API for XML（StAX）。

简而言之，SAX面向状态独立处理，其中处理元素不依赖于以前的元素。而StAX则是面向状态依赖的处理。有关更详细的比较，请参阅何时使用SAX。

建立一个使用SAX的程序比建立一个使用文档对象模型（DOM）的程序需要更多的工作。SAX是一个事件驱动模型（你提供回调方法，并且解析器在读取XML数据时调用它们），并且更加难以可视化。最后，您不能“备份”到文档的较早部分，或重新排列它，不能备份序列数据流或重新排列您从该流中读取的字符。

由于这个原因，一个开发人员如果编写一个面向用户的用来展示XML文档并且可能会修改文档的应用程序，可能会更倾向使用在[Document Object Model](http://docs.oracle.com/javase/tutorial/jaxp/dom/index.html).中描述的DOM机制。

但是，即使您打算专门构建DOM应用程序，有几个可以让您熟悉SAX模型的重要的原因：

* **Same Error Handling**: SAX和DOM API生成相同种类的异常，因此错误处理代码几乎相同。
* **Handling Validation Errors**: 默认情况下，规范要求忽略验证错误。如果你想在验证错误的事件下抛出一个异常（你可能你会这么做），那么你需要了解SAX错误处理的工作原理。
* **Converting Existing Data**: 正如您将在Document Object Model中看到的，您可以使用一种将现有数据集转换为XML的机制，然而，利用该机制需要了解SAX模型。

## When to Use SAX

当您要将现有数据转换为XML时，了解SAX事件模型是有帮助的。转换过程的关键是修改现有应用程序以在读取数据时传递SAX事件。

SAX是快速和高效的，但是它的事件模型使其对于不依赖状态的过滤最有用。例如，当遇到元素标签时，SAX解析器调用应用程序中的一个方法，并在遇到文本时调用另一个不同的方法。如果你正在做的处理是与状态无关的（这意味着它不依赖于之前的元素），则SAX可以很好的工作。

另一方面，对于状态依赖的处理，程序需要在元素A下的数据进行操作，但是对在元素B下数据进行不同的操作，这时一个pull式解析比如Streaming API for XML (StAX)可能是一个更好的选择。使用一个pull式解析器，无论发生什么，在代码的任何地方你都可以获得下一个节点。因此，您可以轻松地改变处理文本的方式（例如），因为您可以在程序中多个位置处理它（更多详细信息，请参阅[Further Information](http://docs.oracle.com/javase/tutorial/jaxp/sax/info.html)）。

SAX需要比DOM少得多的内存，因为SAX不像DOM那样构建XML数据的内部表示（树结构）。相反，SAX只是简单的在读取时将数据发送给程序。然后，您的应用程序可以对其所接收的数据做任何事情。

Pull式解析器和SAX API像序列I / O流一样工作。您可以看到数据流入，但您不能返回到较早的位置或向前跳跃到不同的位置。一般来说，当您只想读取数据并使应用程序对其进行操作时，这样的解析器就会很好地运行。

但是当您需要修改XML结构时，尤其是当您需要以交互方式进行修改时，一个在内存中的结构则更有意义。DOM就是这样的一个模型，然而，尽管DOM为大型文档（如书籍和文章）提供了许多强大的功能，但它依旧需要大量复杂的代码。该过程的详细信息在下一课中[When to Use DOM](http://docs.oracle.com/javase/tutorial/jaxp/dom/when.html)突出显示。

对于更简单的应用程序，这种复杂代码可能是不必要的。为了更快的开发和更简单的应用程序，面向对象的XML编程标准之一，如JDOM（http://www.jdom.org）和DOM4J（http://www.dom4j.org/）可能会更有意义。

## Parsing an XML File Using SAX

在实际应用程序中，您将需要使用SAX解析器来处理XML数据并对其进行有用的处理。本节将查看你一个示例JAXP项目SAXLocalNameCount，它统计在一个XML文档中只使用localName组件元素的个数。为了简单起见，命名空间名称则会被省略。这个示例同时展示如何使用一个SAX ErrorHandler。

从JAXP下载区域下载并安装了JAXP API的源代码后，本示例程序可以在目录install-dir / jaxp-1\_4\_2-release-date / samples / sax中找到。它与之交互的XML文件位于install-dir / jaxp-1\_4\_2-release-date / samples / data中。

### Creating the Skeleton

SAXLocalNameCount程序在名为SAXLocalNameCount.java的文件中创建。

public class SAXLocalNameCount {

static public void main(String[] args) {

// ...

}

}

因为你将独立运行，所以需要一个main（）方法。并且您需要命令行参数，以便您可以告诉应用程序要处理哪个文件。

### Importing Classes

应用程序将使用的类的import语句如下。

package sax;

import javax.xml.parsers.\*;

import org.xml.sax.\*;

import org.xml.sax.helpers.\*;

import java.util.\*;

import java.io.\*;

public class SAXLocalNameCount {

// ...

}

javax.xml.parsers包包含的SAXParserFactory类，用于创建使用的解析器实例。如果它不能产生与指定的选项配置相匹配的解析器，它会抛出ParserConfigurationException异常。 （稍后，您将看到有关配置选项的更多信息）。javax.xml.parsers包还包含SAXParser类，这是工厂类返回用来解析的类。org.xml.sax包定义了用于SAX解析器的所有接口。org.xml.sax.helpers包包含DefaultHandler，它是定义了将处理解析器生成的SAX事件的类。java.util和java.io中的类用来来提供哈希表和输出。

### Setting Up I/O

业务的第一步是处理命令行参数，在这个阶段，这个参数只能用来获取要处理的文件的名称。main方法中的以下代码告诉应用程序SAXLocalNameCountMethod要处理什么文件。

static public void main(String[] args) throws Exception {

String filename = null;

for (int i = 0; i < args.length; i++) {

filename = args[i];

if (i != args.length - 1) {

usage();

}

}

if (filename == null) {

usage();

}

}

以上代码设置当main方法遇到任何问题的时候抛出一个Exception，并定义了告诉应用程序需要处理的XML文件的名称所需的命令行选项。这部分代码的其他命令行参数会在稍后查看，当我们开始研究验证的时候。

您运行应用程序时给出的filename字符串将通过内部方法convertToFileURL（）转换为java.io.File URL。这通过SAXLocalNameCountMethod中的以下代码完成。

public class SAXLocalNameCount {

private static String convertToFileURL(String filename) {

String path = new File(filename).getAbsolutePath();

if (File.separatorChar != '/') {

path = path.replace(File.separatorChar, '/');

}

if (!path.startsWith("/")) {

path = "/" + path;

}

return "file:" + path;

}

// ...

}

如果运行程序时指定了不正确的命令行参数，则调用SAXLocalNameCount应用程序的usage（）方法，以在屏幕上打印出正确的选项。

private static void usage() {

System.err.println("Usage: SAXLocalNameCount <file.xml>");

System.err.println(" -usage or -help = this message");

System.exit(1);

}

在本节后更多的usage（）选项会被查看，当验证被解决的时候。

### Implementing the ContentHandler Interface

SAXLocalNameCount中最重要的接口是ContentHandler。此接口需要一些由SAX调用用来响应各种解析事件的方法。主要的事件处理方法是: startDocument, endDocument, startElement, 和 endElement.

实现此接口的最简单方法是继承在org.xml.sax.helpers包中定义的DefaultHandler类。该类为所有ContentHandler事件提供了无操作的方法。示例程序继承了该类。

public class SAXLocalNameCount extends DefaultHandler {

// ...

}

DefaultHandler还定义了DTDHandler，EntityResolver和ErrorHandler接口中定义的其他主要事件的空操作方法。您将在本课中稍后详细了解这些方法。

### Document Events

以下代码处理start-document和end-document事件：

public class SAXLocalNameCount extends DefaultHandler {

private Hashtable tags;

public void startDocument() throws SAXException {

tags = new Hashtable();

}

public void endDocument() throws SAXException {

Enumeration e = tags.keys();

while (e.hasMoreElements()) {

String tag = (String)e.nextElement();

int count = ((Integer)tags.get(tag)).intValue();

System.out.println("Local Name \"" + tag + "\" occurs "

+ count + " times");

}

}

private static String convertToFileURL(String filename) {

// ...

}

// ...

}

该代码定义了当解析器遇到要解析的文档的开始和结束点时应用程序的功能。ContentHandler接口的startDocument（）方法创建一个java.util.Hashtable实例，它将在元素事件中填充解析器在文档中找到的XML元素。当解析器到达文档的末尾时，调用endDocument（）方法，以获取哈希表中包含的元素的名称和统计次数，并在屏幕上打印出一条消息，告诉用户每个元素的发生次数有多少找到。

这两个ContentHandler方法都抛出一个SAXExceptions。您将在Setting up Error Handling.中了解有关SAX异常的更多信息。

### Element Events

如Document Events所述，由startDocument方法创建的哈希表需要使用解析器在文档中找到的各种元素来填充。以下代码处理start-element和end-element事件：

public void startDocument() throws SAXException {

tags = new Hashtable();

}

public void startElement(String namespaceURI,

String localName,

String qName,

Attributes atts)

throws SAXException {

String key = localName;

Object value = tags.get(key);

if (value == null) {

tags.put(key, new Integer(1));

}

else {

int count = ((Integer)value).intValue();

count++;

tags.put(key, new Integer(count));

}

}

public void endDocument() throws SAXException {

// ...

}

这些代码处理元素标签，包括在起始标签中定义的任何属性，以获取命名空间统一资源标识符（URI），该元素的本地名称和限定名称。然后，startElement（）方法为每个类型的元素，将其本地名称和其统计次数填充在由startDocument（）创建的哈系表中。请注意，当调用startElement（）方法时，如果未启用命名空间处理，则元素和属性的本地名称可能会变为空字符串。每当简单名称为空字符串时，代码将使用限定名称来处理这种情况。

### Character Events

JAXP SAX API还允许您使用ContentHandler.characters（）方法处理解析器提传递给应用程序的字符。

字符事件在SAXLocalNameCount示例中没有演示，但是为了完整，本节将简要描述。

解析器不需要一次返回任何特定数量的字符。一个解析器可以一次返回一个字符上至几千个，这仍然是一个符合标准的实现。因此，如果您的应用程序需要处理它看到的字符时，明智之举就是让characters()方法在一个java.lang.StringBuffer中累加字符，并且只有当您确定已经找到所有字符时，才会对它们进行操作。

当元素结束时，您将完成解析文本，因此您通常会在此处执行字符处理。但是，您可能还需要在元素开始时处理文本。这对于可以包含与文本混合的XML元素的文档样式数据是必需的。例如，考虑这个文档片段：

<para>This paragraph contains <bold>important</bold> ideas.</para>

这个初始文本” This paragraph contains “被<bold>元素的开始标签打断。”inportant”文本又被结束标签</bold>中断,并且最终的文本”idea“被结束标签</para>终止。

要想解析严格准确，字符处理程序应扫描符号字符（＆）和左角括号（<），并用字符串＆amp;或&lt;来代替,这将在下一节中进行说明。

#### Handling Special Characters

在XML中，实体是具有名称的XML结构（或纯文本）。按名称引用实体将导致将其插入到文档中而不是实体引用。要创建实体引用，您可以使用＆符号和分号来围绕实体名称：

&entityName;

当您处理包含许多特殊字符的大块XML或HTML时，可以使用CDATA区。CDATA区部分功能如HTML中的<code>…</code>一样更重要的是，所有CDATA区内的空格都是有意义的，并且其中的字符不会被解释为XML。CDATA部分以<！[[CDATA [以结尾]]>开头。

从示例XML文件install-dir / jaxp-1\_4\_2-release-date / samples / data / REC-xml-19980210.xml获取的CDATA部分的示例如下所示。

<p><termdef id="dt-cdsection" term="CDATA Section"<<term>CDATA sections</term> may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "<code>&lt;![CDATA[</code>" and end with the string "<code>]]&gt;</code>"

一旦解析，此文本将显示如下：

CDATA sections may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "<![CDATA[" and end with the string "]]>".

CDATA的存在使得XML的适当回应有点棘手。如果要输出的文本不在CDATA部分，则文本中的任何尖括号，＆符号和其他特殊字符都应替换为相应的实体引用（更换左尖括号和＆号码是最重要的，其他字符将被正确解释，而不会误导解析器。）。但是，如果输出文本在CDATA部分，则不应该进行替换，其结果类似于前面示例中的文本。在一个简单的程序，如我们的SAXLocalNameCount应用程序，这不是特别重要。但是许多XML过滤应用程序将要跟踪文本是否出现在CDATA部分，以便它们可以正确处理特殊字符。

### Setting up the Parser

以下代码设置解析器并启动它：

static public void main(String[] args) throws Exception {

// Code to parse command-line arguments

//(shown above)

// ...

SAXParserFactory spf = SAXParserFactory.newInstance();

spf.setNamespaceAware(true);

SAXParser saxParser = spf.newSAXParser();

}

这些代码行创建一个SAXParserFactory实例，由javax.xml.parsers.SAXParserFactory系统属性的设置确定。创建的工厂类通过方法setNamespaceAware设置为true，被配置支持XML命名空间，然后通过调用其newSAXParser（）方法从工厂类获取SAXParser实例。

javax.xml.parsers.SAXParser类是定义一些方便方法的包装器。它包装（有些不太友善）的org.xml.sax.Parser对象。如果需要，您可以使用SAXParser类的getParser（）方法获取该解析器。

您现在需要实现所有解析器必须实现的XMLReader。应用程序使用XMLReader来告诉SAX解析器对所讨论的文档执行什么处理。XMLReader由main方法中的以下代码实现。

// ...

SAXParser saxParser = spf.newSAXParser();

XMLReader xmlReader = saxParser.getXMLReader();

xmlReader.setContentHandler(new SAXLocalNameCount());

xmlReader.parse(convertToFileURL(filename));

在这里，通过调用SAXParser实例的getXMLReader（）方法，为解析器获取一个XMLReader实例。然后，XMLReader将SAXLocalNameCount类注册为其content handler，以便解析器执行的操作将是Handling Content Events中显示的startDocument（），startElement（）和endDocument（）方法的操作。最后，XMLReader通过传递XML文件的位置来告诉解析器那个文档需要解析，文件位置是以由Setting Up I/O.中定义的convertToFileURL（）方法生成的文件URL的形式来传递。

### Setting up Error Handling

您现在可以开始使用您的解析器，但实现一些error handler会更安全。解析器可以生成三种错误：致命错误，错误和警告。当发生致命错误时，解析器无法继续。因此，如果应用程序不生成一个异常，而是由默认的错误事件handler生成一个异常。但是对于非致命错误和警告，默认error handler从不生成异常，并且不显示任何消息。

如Document Events所示，应用程序的事件处理方法抛出SAXException。例如，ContentHandler接口中的startDocument（）方法的签名被定义为返回一个SAXException。

public void startDocument() throws SAXException { /\* ... \*/ }

一个SAXException可以由一个消息或另一个异常来构建，或者两个都使用。

因为默认解析器只会为致命错误生成一个异常，并且由于默认解析器提供的有关错误的信息有一些限制，所以SAXLocalNameCount程序通过MyErrorHandler类来定义自己的错误处理。

xmlReader.setErrorHandler(new MyErrorHandler(System.err));

// ...

private static class MyErrorHandler implements ErrorHandler {

private PrintStream out;

MyErrorHandler(PrintStream out) {

this.out = out;

}

private String getParseExceptionInfo(SAXParseException spe) {

String systemId = spe.getSystemId();

if (systemId == null) {

systemId = "null";

}

String info = "URI=" + systemId + " Line="

+ spe.getLineNumber() + ": " + spe.getMessage();

return info;

}

public void warning(SAXParseException spe) throws SAXException {

out.println("Warning: " + getParseExceptionInfo(spe));

}

public void error(SAXParseException spe) throws SAXException {

String message = "Error: " + getParseExceptionInfo(spe);

throw new SAXException(message);

}

public void fatalError(SAXParseException spe) throws SAXException {

String message = "Fatal Error: " + getParseExceptionInfo(spe);

throw new SAXException(message);

}

}

以与Setting up the Parser中相同的方式，其中XMLReader指向正确的content handler，这里XMLReader通过调用其setErrorHandler（）方法指向新的error handler。

MyErrorHandler类实现标准的org.xml.sax.ErrorHandler接口，并定义了一种方法来获取由解析器生成的任何SAXParseException实例提供的异常信息。getParseExceptionInfo（）这个方法只是简单的通过调用标准的SAXParseException方法getLineNumber() 和 getSystemId()，来获取在XML文档中错误发生的行号以及该解析器运行在的系统标识号。然后，将此异常信息提供给基本SAX错误处理方法error（），warning（）和fatalError（）的实现中，这些更新将发送关于文档中错误的性质和位置的适当消息。

#### Handling NonFatal Errors

当XML文档有效性约束失败时，会发生非致命错误。如果解析器发现文档无效，就会生成一个错误事件。给文档一个类型定义（DTD）或者shcema，当文档由一个无效标签，或者找到一个不允许的标签，或者（在schema情况下）元素包含无效的数据，验证解析器就会抛出这样的错误。

了解非致命错误的最重要原则是默认情况下它们会被忽略。但如果文档中出现验证错误，你可能不想继续处理文档。你可能想把这些错误当作致命错误一样看待。

为了接管错误处理，您可以重写DefaultHandler中处理致命错误，非致命错误和警告的方法，这些方法是ErrorHandler接口的一部分。如上一节中的代码摘录所示，SAX解析器为每个这些方法传递SAXParseException，因此在发生错误时生成异常就像重新抛回它一样简单。

查看org.xml.sax.helpers.DefaultHandler中定义的错误处理方法是有益的。你会发现error（）和warning（）方法中什么都没做，而fatalError（）抛出一个了异常。当然，您也可以重写fatalError（）方法来抛出异常。但是，如果您的代码在发生致命错误时不抛出异常，则SAX解析器将会抛出。 这时XML的规范要求。

#### Handling Warnings

警告在默认情况下也会被忽略。警告类型很丰富，并且只能在有一个DTD或者Schema的情况下才能生成。例如，如果一个元素在DTD中定义了两次，就会生成警告。这不是非法的，它不会造成问题，但是你可能想知道的是因为它可能不是故意的。针对DTD验证一个XML文档会在下面章节展示。

## Running the SAX Parser Example without Validation

如本课开始所述，在从JAXP源下载区下载并安装了JAXP API的源代码之后，可以在以下位置找到需要运行的示例程序和相关文件。

* 该示例的不同Java归档（JAR）文件位于目录install-dir / jaxp-1\_4\_2-release-date / lib中。.
* SAXLocalNameCount.java文件位于install-dir / jaxp-1\_4\_2-release-date / samples / sax中。.
* SAXLocalNameCount与之交互的XML文件位于install-dir / jaxp-1\_4\_2-release-date / samples / data中。

### To Run the SAXLocalNameCount Example without Validation

1. 移动到示例文件夹。% cd install-dir/jaxp-1\_4\_2-release-date/samples.
2. 编译示例类。% javac sax/\*
3. 运行 SAXLocalNameCount program 在一个XML 文件上.

选择data目录中的一个XML文件，并在其上运行SAXLocalNameCount程序。在这里，我们选择在文件rich\_iii.xml上运行该程序。

% java sax/SAXLocalNameCount data/rich\_iii.xml

XML文件rich\_iii.xml包含William版本的莎士比亚剧本Richard III的XML版本。当您运行SAXLocalNameCount时，您应该看到以下输出。

Local Name "STAGEDIR" occurs 230 times

Local Name "PERSONA" occurs 39 times

Local Name "SPEECH" occurs 1089 times

Local Name "SCENE" occurs 25 times

Local Name "ACT" occurs 5 times

Local Name "PGROUP" occurs 4 times

Local Name "PLAY" occurs 1 times

Local Name "PLAYSUBT" occurs 1 times

Local Name "FM" occurs 1 times

Local Name "SPEAKER" occurs 1091 times

Local Name "TITLE" occurs 32 times

Local Name "GRPDESCR" occurs 4 times

Local Name "P" occurs 4 times

Local Name "SCNDESCR" occurs 1 times

Local Name "PERSONAE" occurs 1 times

Local Name "LINE" occurs 3696 times

SAXLocalNameCount程序解析XML文件，并提供其包含的每种类型的XML标签的实例数量的计数。

1. 在文本编辑器中打开文件data/ rich\_iii.xml。

要检查错误处理是否正常，请从XML文件中的条目中删除结束标签，例如第30行的结束标记</ PERSONA>，如下所示。

30 <PERSONA>EDWARD, Prince of Wales, afterwards King Edward V.</PERSONA>

1. 再次运行SAXLocalNameCount。

这一次，您应该看到以下致命错误消息。

% java sax/SAXLocalNameCount data/rich\_iii.xml Exception in thread "main" org.xml.sax.SAXException: Fatal Error: URI=file:*install-dir* /JAXP\_sources/jaxp-1\_4\_2-*release-date*/samples/data/rich\_iii.xml Line=30: The element type "PERSONA" must be terminated by the matching end-tag "</PERSONA>".

如您所见，当遇到错误时，解析器生成一个SAXParseException，一个SAXException的子类，用于标识文件和发生错误的位置。