### [Java6.0新特性之StAX--全面解析Java XML分析技术](http://zangweiren.iteye.com/blog/647334)

Java 6.0对XML支持的新特性有许多方面。比如StAX、针对XML-Web服务的Java架构（JAX-WS）2.0、针对XML绑定的API（JAXB）2.0、XML数字签名API，甚至还支持SQL:2003 'XML'数据类型。在这一篇文章中我们将要介绍的是StAX技术，因为它在我们的开发中将被使用地更加频繁。   
  
StAX是Streaming API for XML的缩写，是一种针对XML的流式拉分析API。关于对XML进行分析（或解析）的技术，大家一定都不陌生了。在Java 6.0之前，就已经有四种：

1. DOM：Document Object Model
2. SAX：Simple API for XML
3. JDOM：Java-based Document Object Model
4. DOM4J：Document Object Model for Java

关于它们的解析原理，以及性能和优缺点，我会在本文的结尾做一个简要的介绍。这篇文章中，我们主要说说StAX这种新的解析方式。   
  
首先我们来搞清楚两个概念：****推分析****和****拉分析****。   
  
在程序中访问和操作XML文件一般有两种模型：DOM（文档对象模型）和流模型。它们的优缺点如下：

**引用**

DOM优点：允许编辑和更新XML文档，可以随机访问文档中的数据，可以使用XPath（XML Path Language，是一种从XML文档中搜索节点的查询语言）查询。   
DOM缺点：需要一次性加载整个文档到内存中，对于大型文档，会造成性能问题。

**引用**

流模型优点：对XML文件的访问采用流的概念，在任何时候内存中只有当前节点，解决了DOM的性能问题。   
流模型缺点：是只读的，并且只能向前，不能在文档中执行向后导航操作。

关于什么是DOM，文章结尾处会有介绍。这里我们简单说一下流：它是一个连续的字节序列，可以理解为不停地从源头向目标搬运着字节的特殊对象。   
  
让我们回到主题。流模型每次迭代XML文档中的一个节点，适合于处理较大的文档，所耗内存空间小。它有两种变体--“推”模型和“拉”模型。

**引用**

推模型：就是我们常说的SAX，它是一种靠事件驱动的模型。当它每发现一个节点就引发一个事件，而我们需要编写这些事件的处理程序。这样的做法很麻烦，且不灵活。

**引用**

拉模型：在遍历文档时，会把感兴趣的部分从读取器中拉出，不需要引发事件，允许我们选择性地处理节点。这大大提高了灵活性，以及整体效率。

到此，我们就弄明白了“推分析”和“拉分析”的概念：

**引用**

基于流模型中推模型的分析方式称为推分析；基于流模型中拉模型的分析方式就称为拉分析。

StAX就是一种拉分析式的XML解析技术。它也支持对XML文件的生成操作，但是这篇文章里我们只介绍有关解析的知识。   
  
从一开始，JAXP（Java API for XML Processing）就提供了两种方法来处理XML：DOM和SAX。StAX是一种面向流的新方法，最终版本于2004年3月发布，并成为JAXP 1.4（包含在Java 6.0中）的一部分。StAX的实现使用了JWSDP（Java Web Services Development Pack）1.6，并结合了SJSXP（Sun Java System XML Streaming Parser，位于javax.xml.stream.\*包中）。   
  
JWSDP是用来开发Web Services、Web应用程序以及Java应用（主要是XML处理）的开发包。它包含的Java API有：

* JAXP：Java API for XML Processing
* JAXB：Java Architecture for XML Binding
* JAX-RPC：Java API for XML-based Remote Procedure Calls
* JAX-WS：Java API for XML Web Services
* SAAJ：SOAP with Attachments API for Java
* JAXR：Java API for XML Registries
* Web Services Registry

JWSDP的早期版本中还包括：

* Java Servlet
* JSP：JavaServer Pages
* JSF：JavaServer Faces

现在，JWSDP已经被GlassFish所替代。   
  
StAX包括两套处理XML的API，分别提供了不同程度的抽象。它们是：基于指针的API和基于迭代器的API。   
  
我们先来了解基于指针的API。它把XML作为一个标记（或事件）流来处理，应用程序可以检查解析器的状态，获得解析的上一个标记的信息，然后再处理下一个标记，依次类推。   
  
在开始API探索之前，我们首先创建一个名为users.xml的XML文档用于测试，它的内容如下：

**Xml代码**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<company>**
3. **<depart** title="Develop Group"**>**
4. **<user** name="Tom" age="28" gender="male" **>**Manager**</user>**
5. **<user** name="Lily" age="26" gender="female" **/>**
6. **</depart>**
7. **<depart** title="Test Group"**>**
8. **<user** name="Frank" age="32" gender="male" **>**Team Leader**</user>**
9. **<user** name="Bob" age="45" gender="male" **/>**
10. **<user** name="Kate" age="25" gender="female" **/>**
11. **</depart>**
12. **</company>**

可以让我们使用基于指针的API的接口是javax.xml.stream.XMLStreamReader（很遗憾，你不能直接实例化它），要得到它的实例，我们需要借助于javax.xml.stream.XMLInputFactory类。根据JAXP的传统风格，这里使用了抽象工厂（Abstract Factory）模式。如果你对这个模式很熟悉的话，就能够在脑海中想象出我们将要编写的代码的大致框架了。   
  
首先，获得一个XMLInputFactory的实例。方法是：

**Java代码**

1. XMLInputFactory factory = XMLInputFactory.newInstance();

或者：

**Java代码**

1. XMLInputFactory factory = XMLInputFactory.newFactory();

这两个方法是等价的，它们都是创建了一个新的实例，甚至实例的类型都是完全一致的。因为它们的内部实现都是：

**Java代码**

1. {
2. **return** (XMLInputFactory) FactoryFinder.find("javax.xml.stream.XMLInputFactory", "com.sun.xml.internal.stream.XMLInputFactoryImpl");
3. }

接下来我们就可以创建XMLStreamReader实例了。我们有这样一组方法可以选择：

**Java代码**

1. XMLStreamReader createXMLStreamReader(java.io.Reader reader) **throws** XMLStreamException;
3. XMLStreamReader createXMLStreamReader(javax.xml.tranform.Source source) **throws** XMLStreamException;
5. XMLStreamReader createXMLStreamReader(java.io.InputStream stream) **throws** XMLStreamException;
7. XMLStreamReader createXMLStreamReader(java.io.InputStream stream, String encoding) **throws** XMLStreamException;
9. XMLStreamReader createXMLStreamReader(String systemId, java.io.InputStream stream) **throws** XMLStreamException;
11. XMLStreamReader createXMLStreamReader(String systemId, java.io.Reader reader) **throws** XMLStreamException;

这些方法都会根据给定的流创建一个XMLStreamReader实例，大家可以依据流的类型、是否需要指定解析XML的编码或者systemId来选择相应的方法。   
  
在这里，我们对systemId稍作说明，并简单解释一下它与publicId的区别。   
  
systemId和publicId是XML文档里DOCTYPE元素中经常出现的两个属性。它们都是对外部资源的引用，用以指明引用资源的地址。systemId是直接引用资源，publicId是间接定位外部资源。具体一点说是这样：

**引用**

systemId：外部资源（大多是DTD文件）的URI。比如本地文件file:///user/dtd/users.dtd或者网络某个地址的文件http://www.w3.org/dtd/users.dtd。

**引用**

publicId：相当于一个名字，这个名字代表了一个外部资源。比如，我们规定"W3C HTML 4.0.1"这个字符串对应"http://www.w3.org/dtd/users.dtd"这个资源。那么，publicId="W3C HTML 4.0.1"和systemId="http://www.w3.org/dtd/users.dtd"的作用就是一样的。

好了，我们接着用以上列出的第一个接口来创建一个XMLStreamReader实例：

**Java代码**

1. **try** {
2. XMLStreamReader reader = factory.createXMLStreamReader(**new** FileReader("users.xml"));
3. } **catch** (FileNotFoundException e) {
4. e.printStackTrace();
5. } **catch** (XMLStreamException e) {
6. e.printStackTrace();
7. }

要遍历XML文档，需要用到XMLStreamReader的下面几个方法：

**Java代码**

1. **int** getEventType();
3. **boolean** hasNext() **throws** XMLStreamException;
5. **int** next() **throws** XMLStreamException;

getEventType()方法返回XMLStreamConstants接口中定义的一个标记常量，表示当前指针所指向标记（或事件）的类型。根据当前事件类型的不同，应用程序可以做出不同的处理。标记常量的类型和含义如下：

1. START\_DOCUMENT：文档的开始
2. END\_DOCUMENT：文档的结尾
3. START\_ELEMENT：元素的开始
4. END\_ELEMENT：元素的结尾
5. PROCESSING\_INSTRUCTION：处理指令
6. CHARACTERS：字符（文本或空格）
7. COMMENT：注释
8. SPACE：可忽略的空格
9. ENTITY\_REFERENCE：实体的引用
10. ATTRIBUTE：元素的属性
11. DTD：DTD
12. CDATA：CDATA块
13. NAMESPACE：命名空间的声明
14. NOTATION\_DECLARATION：标记的声明
15. ENTITY\_DECLARATION：实体的声明

next()方法将指针移动到下一个标记，它同时返回这个标记（或事件）的类型。此时若接着调用getEventType()方法则返回相同的值。   
  
hasNext()用于判断是否还有下一个标记。只有当它返回true时才可以调用next()以及其它移动指针的方法。   
  
看了上面几个方法的介绍，大家就会发现使用XMLStreamReader遍历XML文档是非常容易的，因为它的用法和每个人都熟悉的Java迭代器（Iterator）是一样的。下面我们就用已经掌握的这几个方法对上文中给出的XML文档做一个测试。希望你还记得它的内容，如果忘记了，请翻回去重新浏览一下。   
  
我们的测试代码如下：

**Java代码**

1. /\*\*
2. \* 列出所有用户
3. \*
4. \* @author zangweiren 2010-4-17
5. \*
6. \*/
7. **public** **class** ListUsers {
8. // 获得解析器
9. **public** **static** XMLStreamReader getStreamReader() {
10. String xmlFile = ListUsers.**class**.getResource("/").getFile()
11. + "users.xml";
12. XMLInputFactory factory = XMLInputFactory.newFactory();
13. **try** {
14. XMLStreamReader reader = factory
15. .createXMLStreamReader(**new** FileReader(xmlFile));
16. **return** reader;
17. } **catch** (FileNotFoundException e) {
18. e.printStackTrace();
19. } **catch** (XMLStreamException e) {
20. e.printStackTrace();
21. }
22. **return** **null**;
23. }
25. // 列出所有用户名称
26. **public** **static** **void** listNames() {
27. XMLStreamReader reader = ListUsers.getStreamReader();
28. // 遍历XML文档
29. **try** {
30. **while** (reader.hasNext()) {
31. **int** event = reader.next();
32. // 如果是元素的开始
33. **if** (event == XMLStreamConstants.START\_ELEMENT) {
34. // 列出所有用户名称
35. **if** ("user".equalsIgnoreCase(reader.getLocalName())) {
36. System.out.println("Name:"
37. + reader.getAttributeValue(**null**, "name"));
38. }
39. }
40. }
41. reader.close();
42. } **catch** (XMLStreamException e) {
43. e.printStackTrace();
44. }
45. }
47. **public** **static** **void** main(String[] args) {
48. ListUsers.listNames();
49. }
50. }

运行结果：

**引用**

Name:Tom   
Name:Lily   
Name:Frank   
Name:Bob   
Name:Kate

在上面的示例代码中，我们用到了XMLStreamReader的两个新方法：

**Java代码**

1. String getLocalName();
3. String getAttributeValue(String namespaceURI, String localName);

与此相关的还有一个方法：

**Java代码**

1. QName getName();

这三个方法牵扯到XML的namespace（命名空间）、localName（本地名称）、QName（Qualified Name，限定名称）三个概念，我们顺便解释一下：   
  
命名空间是为了支持相同名称不同含义的XML标签而产生的，它可以这么定义：

**Xml代码**

1. **<com:company** xmlns:com="http://www.zangweiren.com/company"**>**
2. <!-- here is other tags -->
3. **</com:company>**

其中，com是命名空间的前缀，company是命名空间的标签，http://www.zangweiren.com/company是命名空间的标识，相同的标识被认为是同一个命名空间。标识又叫URI，是唯一的，有URL（统一资源定位器）和URN（统一资源名称）两种。前缀是命名空间的简写，目的是为了使用方便。命名空间被声明后就可以被使用：

**Xml代码**

1. **<com:company** xmlns:com="http://www.zangweiren.com/company"**>**
2. **<com:depart** name="Develop Group" **/>**
3. **</com:company>**

在上例的<com:depart />标签中，前缀com是命名空间，depart是localName，这两个合起来就是QName。   
  
在明白了这三个XML基本概念之后，也就明白了getLocalName()和getAttributeValue(String namespaceURI, String localName)方法的含义。   
  
现在，我们已经学会了使用XMLStreamReader遍历XML文档，并对特定标签进行解析了。   
  
我们再来看看下面两个方法：

**Java代码**

1. String getElementText() **throws** XMLStreamException;
3. **int** nextTag() **throws** XMLStreamException;

getElementText()方法返回元素的开始标签（START\_ELEMENT）和关闭标签（END\_ELEMENT）之间的所有文本内容，若遇到嵌套的元素就会抛出异常。   
  
nextTag()方法将跳过所有空白、注释或处理指令，直到遇到START\_ELEMENT或END\_ELEMENT。它在解析只含元素内容的XML文档时很有用。否则，在发现标记之前遇到非空白文本（不包括注释和处理指令），就会抛出异常。   
  
比如我们修改上一个测试程序，增加一个新方法：

**Java代码**

1. // 列出所有用户的名称和年龄
2. **public** **static** **void** listNamesAndAges() {
3. XMLStreamReader reader = ListUsers.getStreamReader();
4. **try** {
5. **while** (reader.hasNext()) {
6. // 跳过所有空白、注释或处理指令，到下一个START\_ELEMENT
7. **int** event = reader.nextTag();
8. **if** (event == XMLStreamConstants.START\_ELEMENT) {
9. **if** ("user".equalsIgnoreCase(reader.getLocalName())) {
10. System.out.println("Name:"
11. + reader.getAttributeValue(**null**, "name")
12. + ";Age:"
13. + reader.getAttributeValue(**null**, "age"));
14. }
15. }
16. }
17. reader.close();
18. } **catch** (XMLStreamException e) {
19. e.printStackTrace();
20. }
21. }

然后把它添加到主方法中：

**Java代码**

1. **public** **static** **void** main(String[] args) {
2. ListUsers.listNames();
3. ListUsers.listNamesAndAges();
4. }

运行它试试看，在解析到<user name="Tom" age="28" gender="male" >Manager</user>的时候会报错，因此你会得到一个类似这样的错误信息：   
  
javax.xml.stream.XMLStreamException: ParseError at [row,col]:[4,53]   
Message: found: CHARACTERS, expected START\_ELEMENT or END\_ELEMENT   
  
对于基于指针的XMLStreamReader来说，虽然API文档说的是“事件”，但是我们把它看成“标记”更易于理解，而且不会与另一套基于事件的API相混淆。   
  
XMLStreamReader的某些方法，无论当前标记（或事件）是什么类型的，都可以被调用。它们的定义和作用如下：

* String getVersion();//获得XML文档中的版本信息
* String getEncoding();//获得XML文档中的指定编码
* javax.xml.namespace.NamespaceContext getNamespaceContext();//获得当前有效的命名空间上下文，包含前缀、URI等信息
* String getNamespaceURI();//获得当前有效的命名空间的URI
* javax.xml.stream.Location getLocation();//获得当前标记的位置信息，包含行号、列号等
* boolean hasName();//判断当前标记是否有名称，比如元素或属性
* boolean hasText();//判断当前标记是否有文本，比如注释、字符或CDATA
* boolean isStartElement();//判断当前标记是否是标签开始
* boolean isEndElement();//判断当前标记是否是标签结尾
* boolean isCharacters();//判断当前标记是否是字符
* boolean isWhiteSpace();//判断当前标记是否是空白

对于以上方法都很容易理解和记忆，我们不再编写代码展示它们的效果。   
  
让我们看看有关属性操作方法。还是首先熟悉一下它们的定义：

**Java代码**

1. **int** getAttributeCount();
3. String getAttributeLocalName(**int** index);
5. QName getAttributeName(**int** index);
7. String getAttributeNamespace(**int** index);
9. String getAttributePrefix(**int** index);
11. String getAttributeType(**int** index);
13. String getAttributeValue(**int** index);
15. String getAttributeValue(String namespaceURI, String localName);

这些方法都十分容易理解，基本上看方法的名称和参数就知道它的用途了。而且最后一个方法在上面的示例中我们已经用过了。让我们再用一个简单的示例程序进一步加深对这些方法的认识。

**Java代码**

1. // 列出所有用户的名称和年龄
2. **public** **static** **void** listNamesAndAges() {
3. XMLStreamReader reader = ListUsers.getStreamReader();
4. **try** {
5. **while** (reader.hasNext()) {
6. // 跳过所有空白、注释或处理指令，到下一个START\_ELEMENT
7. **int** event = reader.nextTag();
8. **if** (event == XMLStreamConstants.START\_ELEMENT) {
9. **if** ("user".equalsIgnoreCase(reader.getLocalName())) {
10. System.out.println("Name:"
11. + reader.getAttributeValue(**null**, "name")
12. + ";Age:"
13. + reader.getAttributeValue(**null**, "age"));
14. }
15. }
16. }
17. reader.close();
18. } **catch** (XMLStreamException e) {
19. e.printStackTrace();
20. }
21. }

把它加入到主方法中：

**Java代码**

1. **public** **static** **void** main(String[] args) {
2. ListUsers.listNames();
3. // ListUsers.listNamesAndAges();
4. ListUsers.listAllAttrs();
5. }

运行结果：

**引用**

1.name=Tom;age=28;gender=male;   
2.name=Lily;age=26;gender=female;   
3.name=Frank;age=32;gender=male;   
4.name=Bob;age=45;gender=male;   
5.name=Kate;age=25;gender=female;

相信你看到这里，已经可以顺利地使用XMLStreamReader来完成XML文档的解析了。   
  
上面我们介绍了基于指针的StAX API。这种方式尽管效率高，但是没有提供XML结构的抽象，因此是一种低层API。   
  
较为高级的基于迭代器的API允许应用程序把XML作为一系列事件对象来处理，每个对象和应用程序交换XML结构的一部分。应用程序只需要确定解析事件的类型，将其转换成对应的具体类型，然后利用其方法获得属于该事件对象的信息。   
  
StAX中基于迭代器的API是一种面向对象的方式，这也是它与基于指针的API的最大区别。它通过将事件转变为对象，让应用程序可以用面向对象的方式处理它们，这有利于模块化和不同组件之间的代码重用。   
  
事件迭代器API的主要接口是javax.xml.stream.XMLEventReader和javax.xml.stream.events.XMLEvent。XMLEventReader和XMLStreamReader相比要简单的多，这是因为关于解析事件的所有信息都封装在了事件对象（XMLEvent）中。   
  
创建XMLEvent对象前同样需要一个XMLInputFactory实例。它有如下这些创建XMLEvent实例的方法：

**Java代码**

1. XMLEventReader createXMLEventReader(java.io.InputStream stream) **throws** XMLStreamException;
3. XMLEventReader createXMLEventReader(java.io.InputStream stream, String encoding) **throws** XMLStreamException;
5. XMLEventReader createXMLEventReader(java.io.Reader reader) **throws** XMLStreamException;
7. XMLEventReader createXMLEventReader(String systemId, java.io.InputStream stream) **throws** XMLStreamException;
9. XMLEventReader createXMLEventReader(String systemId, java.io.Reader reader) **throws** XMLStreamException;
11. XMLEventReader createXMLEventReader(Source source) **throws** XMLStreamException;
13. XMLEventReader createXMLEventReader(XMLStreamReader reader) **throws** XMLStreamException;

最后一个方法不同与其它的，它是将一个XMLStreamReader对象转换成一个XMLEventReader对象。值得注意的是，XMLInputFactory没有提供将XMLEventreader对象转换成XMLStreamreader对象的方法。我想，在我们的开发过程中，应该不会出现这种需要将高层API转换成低层API来使用的情况。   
  
XMLEventReader接口扩展了java.util.Iterator接口，它定义了以下几个方法：

**Java代码**

1. String getElementText() **throws** XMLStreamException;
3. **boolean** hasNext();
5. XMLEvent nextEvent() **throws** XMLStreamException;
7. XMLEvent nextTag() **throws** XMLStreamException;
9. XMLEvent peek() **throws** XMLStreamException;

其中，getElementText()、hasNext()、nextTag()三个方法的含义及用法类似于XMLStreamReader，而nextEvent()方法类似于XMLStreamReader的next()方法。所以，这里只对peed()方法做一下说明。   
  
调用peek()方法，你将得到下一个事件对象。它与nextEvent()方法的不同是，当你连续两次或两次以上调用它时，你得到的都是同一个事件对象。   
  
我们再看看XMLEvent接口中定义的方法。这些方法大体可以分为三种类别。第一类是用于事件类型判断的：

* boolean isAttribute();//判断该事件对象是否是元素的属性
* boolean isCharacters();//判断该事件对象是否是字符
* boolean isStartDocument();//判断该事件对象是否是文档开始
* boolean isEndDocument();//判断该事件对象是否是文档结尾
* boolean isStartElement();//判断该事件对象是否是元素开始
* boolean isEndElement();//判断该事件对象是否是元素结尾
* boolean isEntityReference();//判断该事件对象是否是实体的引用
* boolean isNamespace();//判断该事件对象是否是命名空间
* boolean isProcessingInstruction();//判断该事件对象是否是处理指令

第二类是将XMLEvent转换为具体的子类对象的：

* Characters asCharacters();//转换为字符事件对象
* StartElement asStartElement();//转换为标签开始事件对象
* EndElement asEndElement();//转换为标签结尾事件对象

第三类是获取事件对象通用信息的：

* javax.xml.stream.Location getLocation();//获得事件对象的位置信息，类似于XMLStreamReader的getLocation()方法
* int getEventType();//获得事件对象的类型，类似于XMLStreamReader的getEventType()方法

其中，getEventType()方法的返回值也是XMLStreamConstants中定义的常量，其类型和含义与XMLStreamReader的getEventType()方法的返回值完全相同。   
  
下面让我们用一段示例代码来熟悉基于迭代器的StAX API的使用方法，进而引出XMLEvent接口的子接口类型。我们仍然使用users.xml作为测试文件：

**Java代码**

1. // 列出所有信息
2. @SuppressWarnings("unchecked")
3. **public** **static** **void** listAllByXMLEventReader() {
4. String xmlFile = ListUsers.**class**.getResource("/").getFile()
5. + "users.xml";
6. XMLInputFactory factory = XMLInputFactory.newInstance();
7. **try** {
8. // 创建基于迭代器的事件读取器对象
9. XMLEventReader reader = factory
10. .createXMLEventReader(**new** FileReader(xmlFile));
11. // 遍历XML文档
12. **while** (reader.hasNext()) {
13. XMLEvent event = reader.nextEvent();
14. // 如果事件对象是元素的开始
15. **if** (event.isStartElement()) {
16. // 转换成开始元素事件对象
17. StartElement start = event.asStartElement();
18. // 打印元素标签的本地名称
19. System.out.print(start.getName().getLocalPart());
20. // 取得所有属性
21. Iterator attrs = start.getAttributes();
22. **while** (attrs.hasNext()) {
23. // 打印所有属性信息
24. Attribute attr = (Attribute) attrs.next();
25. System.out.print(":" + attr.getName().getLocalPart()
26. + "=" + attr.getValue());
27. }
28. System.out.println();
29. }
30. }
31. reader.close();
32. } **catch** (FileNotFoundException e) {
33. e.printStackTrace();
34. } **catch** (XMLStreamException e) {
35. e.printStackTrace();
36. }
37. }

把它加到主程序中：

**Java代码**

1. **public** **static** **void** main(String[] args) {
2. ListUsers.listNames();
3. // ListUsers.listNamesAndAges();
4. ListUsers.listAllAttrs();
5. ListUsers.listAllByXMLEventReader();
6. }

运行后得到如下结果：

**引用**

company   
depart:title=Develop Group   
user:age=28:name=Tom:gender=male   
user:age=26:name=Lily:gender=female   
depart:title=Test Group   
user:age=32:name=Frank:gender=male   
user:age=45:name=Bob:gender=male   
user:age=25:name=Kate:gender=female

这个例子中，我们利用基于迭代器的StAX API打印出了所有元素的本地名称以及它们的全部属性信息。大家可以看到，它的用法与基于指针的StAX API的用法十分相似。但是由于使用了面向对象的思想，更加容易理解。   
  
我们用到了两个新的接口：StartElement和Attribute。它们都是XMLEvent接口的子接口，且都在javax.xml.stream.events.\*包中。它们是更具体的事件对象类型。实际上在javax.xml.stream.events中，除了XMLEvent接口自身外，其余接口都是它的子接口。它们的名称和代表的具体事件对象类型如下：

1. Attribute：元素的属性
2. Characters：字符
3. Comment：注释
4. DTD：DTD
5. StartDocument：文档的开始
6. EndDocument：文档的结束
7. StartElement：元素的开始
8. EndElement：元素的结束
9. EntityDeclaration：实体声明
10. EntityReference：实体的引用
11. Namespace：命名空间声明
12. NotationDeclaration：标记的声明
13. ProcessingInstruction：处理指令

你可能觉得这些类看着很眼熟，因为它们在XMLStreamReader的getEventType()方法的返回值，也就是XMLStreamConstants中定义的常量中，都能找到一一的对应。唯独缺少了SAPCE（可忽略的空白）和CDATA（CDATA块）。也就是说，在基于指针的StAX API中定义事件类型，在基于迭代器的StAX API中都是以对象的形式提供给应用程序的，这就是为什么说后者是一种更具有面向对象思想的高层API的原因。   
  
这些事件对象接口不仅代表了一种事件类型，还包含对应事件对象的信息。至于它们所具有的方法大多是获取事件对象信息的访问器，其含义及具体用法，都很容易理解和使用，因此不再详细介绍。   
  
大家可能注意到，XMLEvent只提供了三个asXXX()形式的方法将它转换到具体的子类型，如果你想要处理的事件对象类型在这三种类型之外，直接使用强制类型转换就可以了。   
  
现在我们掌握了StAX的基于指针的拉分析API和基于迭代器的拉分析API的基本应用。我们再来看一种稍微高级的用法，它可以帮助我们更好地完成XML文档的解析工作。   
  
XMLInputFactory还有两个创建流读取器的方法：

**Java代码**

1. XMLStreamReader createFilteredReader(XMLStreamReader reader, StreamFilter filter) **throws** XMLStreamException;
3. XMLEventReader createFilteredReader(XMLEventReader reader, EventFilter filter) **throws** XMLStreamException;

它们分别为XMLStreamReader和XMLEventReader增加一个过滤器，过滤掉不需要解析的内容，只留下应用程序关心的信息用于解析。虽然我们可以在应用程序中做同样的过滤工作，就像之前示例程序中所写的那样，但是把过滤工作交给过滤器的好处是，让应用程序可以更加专注于解析工作，并且对于通用的过滤（比如注释），将它放到过滤器中可以实现过滤逻辑部分代码的重用。这符合软件设计原则。   
  
如果你编写过文件过滤器java.io.FileFilter的话，那么编写StreamFilter和EventFilter就更加容易。我们先来看看这两个接口的定义：

**Java代码**

1. **public** **interface** StreamFilter {
2. **public** **boolean** accept(XMLStreamReader reader);
3. }
5. **public** **interface** EventFilter {
6. **public** **boolean** accept(XMLEvent event);
7. }

我们就以StreamFilter为例来演示过滤器的用法。为此，我们使用users.xml为测试文档编写一段新的程序：

**Java代码**

1. /\*\*
2. \* StreamFilter示例程序
3. \*
4. \* @author zangweiren 2010-4-19
5. \*
6. \*/
7. **public** **class** TestStreamFilter **implements** StreamFilter {
9. **public** **static** **void** main(String[] args) {
10. TestStreamFilter t = **new** TestStreamFilter();
11. t.listUsers();
12. }
14. @Override
15. **public** **boolean** accept(XMLStreamReader reader) {
16. **try** {
17. **while** (reader.hasNext()) {
18. **int** event = reader.next();
19. // 只接受元素的开始
20. **if** (event == XMLStreamConstants.START\_ELEMENT) {
21. // 只保留user元素
22. **if** ("user".equalsIgnoreCase(reader.getLocalName())) {
23. **return** **true**;
24. }
25. }
26. **if** (event == XMLStreamConstants.END\_DOCUMENT) {
27. **return** **true**;
28. }
29. }
30. } **catch** (XMLStreamException e) {
31. e.printStackTrace();
32. }
33. **return** **false**;
34. }
36. **public** XMLStreamReader getFilteredReader() {
37. String xmlFile = TestStreamFilter.**class**.getResource("/").getFile()
38. + "users.xml";
39. XMLInputFactory factory = XMLInputFactory.newFactory();
40. XMLStreamReader reader;
41. **try** {
42. reader = factory.createXMLStreamReader(**new** FileReader(xmlFile));
43. // 创建带有过滤器的读取器实例
44. XMLStreamReader freader = factory
45. .createFilteredReader(reader, **this**);
46. **return** freader;
47. } **catch** (FileNotFoundException e) {
48. e.printStackTrace();
49. } **catch** (XMLStreamException e) {
50. e.printStackTrace();
51. }
52. **return** **null**;
53. }
55. **public** **void** listUsers() {
56. XMLStreamReader reader = getFilteredReader();
57. **try** {
58. // 列出所有用户的名称
59. **while** (reader.hasNext()) {
60. // 过滤工作已交由过滤器完成，这里不需要再做
61. System.out.println("Name="
62. + reader.getAttributeValue(**null**, "name"));
64. **if** (reader.getEventType() != XMLStreamConstants.END\_DOCUMENT) {
65. reader.next();
66. }
67. }
68. reader.close();
69. } **catch** (XMLStreamException e) {
70. e.printStackTrace();
71. }
72. }
74. }

测试结果：

**引用**

Name=Tom   
Name=Lily   
Name=Frank   
Name=Bob   
Name=Kate

大家可能已经发现，这里有一个与之前处理不同的地方，就是我们先打印了用户的信息，再调用next()方法；这与java.util.Iterator的先调用next()方法，再获取对象信息不同。而之前我们一直采用的是与Iterator一样的处理代码。这里，就有一个问题需要说明。   
  
对于XMLStreamReader的next()方法来说，第一次被调用的时候返回的是第二个标记（或事件）。要获得第一个标记，就需要在调用next()方法之前调用getEventType()方法。这是需要注意的地方。我们以上的代码之所以采用Java迭代器一样的处理方式，是因为第一个标记总是START\_DOCUMENT，而我们不需要对它进行操作，因此就采用了一种熟悉的编码方式，方便大家理解。XMLEventReader的nextEvent()方法就不存在这样的问题。   
  
EventFilter的用法与StreamFilter相同，不再举例说明。   
  
StAX还为我们提供了另外一种隔离标记或事件对象过滤逻辑的方法，那就是StreamReaderDelegate和EventReaderDelegate这两个类，它们都位于javax.xml.stream.util.\*包中。StAX API中大部分都是接口，这两个是确确实实的类。它们都做了同样的工作，就是分别包装了XMLStreamReader和XMLEventReader，并把所有的方法都委托（Delegate）给它们处理，既没有增加任何的方法或逻辑，也没有改变或删除任何方法，因此这里使用的是策略（Strategy）模式。我们可以采用装饰（Decorator）模式，给StreamReaderDelegate或EventReaderDelegate增加新的功能。请看下面的例子：

**Java代码**

1. /\*\*
2. \* 测试StreamReaderDelegate
3. \*
4. \* @author zangweiren 2010-4-19
5. \*
6. \*/
7. **public** **class** TestStreamDelegate {
9. **public** **static** **void** main(String[] args) {
10. TestStreamDelegate t = **new** TestStreamDelegate();
11. t.listUsers();
12. }
14. **public** XMLStreamReader getDelegateReader() {
15. String xmlFile = TestStreamFilter.**class**.getResource("/").getFile()
16. + "users.xml";
17. XMLInputFactory factory = XMLInputFactory.newFactory();
18. XMLStreamReader reader;
19. **try** {
20. reader = **new** StreamReaderDelegate(factory
21. .createXMLStreamReader(**new** FileReader(xmlFile))) {
22. // 重写（Override）next()方法，增加过滤逻辑
23. @Override
24. **public** **int** next() **throws** XMLStreamException {
25. **while** (**true**) {
26. **int** event = **super**.next();
27. // 保留用户元素的开始
28. **if** (event == XMLStreamConstants.START\_ELEMENT
29. && "user".equalsIgnoreCase(getLocalName())) {
30. **return** event;
31. } **else** **if** (event == XMLStreamConstants.END\_DOCUMENT) {
32. **return** event;
33. } **else** {
34. **continue**;
35. }
36. }
37. }
38. };
39. **return** reader;
40. } **catch** (FileNotFoundException e) {
41. e.printStackTrace();
42. } **catch** (XMLStreamException e) {
43. e.printStackTrace();
44. }
45. **return** **null**;
46. }
48. **public** **void** listUsers() {
49. XMLStreamReader reader = **this**.getDelegateReader();
50. **try** {
51. **while** (reader.hasNext()) {
52. reader.next();
53. **if** (reader.getEventType() != XMLStreamConstants.END\_DOCUMENT) {
54. // 列出用户的名称和年龄
55. System.out.println("Name="
56. + reader.getAttributeValue(**null**, "name") + ";age="
57. + reader.getAttributeValue(**null**, "age"));
58. }
59. }
60. reader.close();
61. } **catch** (XMLStreamException e) {
62. e.printStackTrace();
63. }
64. }
66. }

测试结果：

**引用**

Name=Tom;age=28   
Name=Lily;age=26   
Name=Frank;age=32   
Name=Bob;age=45   
Name=Kate;age=25

EventReaderDelegate的用法与StreamReaderDelegate相同。   
  
现在我们介绍完了StAX的两种解析XML文档的方式，大家也可能对它的使用有了自己的认识。我们最后总结一下：XMLStreamReader和XMLEventReader都允许应用程序迭代底层的XML流，区别在于它们如何对外提供解析后的XML信息片段。前者像个指针，指在刚刚解析过的XML标记的后面，并提供获得关于该标记更多信息的方法。因为不用创建新的对象，所以更节约内存。后者具有更多的面向对象特征，就是个标准的Java迭代器，解析器的当前状态反映在事件对象中，应用程序在处理事件对象的时候不需要访问解析器/读取器。   
  
关于各种XML解析技术的优劣   
  
除了我们刚刚介绍过的StAX这种Java 6.0新支持的XML文档解析技术之外，还有四种广为应用的解析方式，我们将对它们做一个简要介绍，并比较五种技术的优缺点以及性能表现，以供大家在开发中选择何种解析技术做参考。  
  
一、DOM（Document Object Model）   
  
文档对象模型分析方式。以层次结构（类似于树型）来组织节点和信息片段，映射XML文档的结构，允许获取和操作文档的任意部分。是W3C的官方标准。

**引用**

优点：   
1、允许应用程序对数据和结构做出更改。   
2、访问是双向的，可以在任何时候在树中上下导航，获取和操作任意部分的数据。

**引用**

缺点：   
1、通常需要加载整个XML文档来构造层次结构，消耗资源大。

二、SAX（Simple API for XML）   
  
流模型中的推模型分析方式。通过事件驱动，每发现一个节点就引发一个事件，通过回调方法完成解析工作，解析XML文档的逻辑需要应用程序完成。

**引用**

优点：   
1、不需要等待所有数据都被处理，分析就能立即开始。   
2、只在读取数据时检查数据，不需要保存在内存中。   
3、可以在某个条件得到满足时停止解析，不必解析整个文档。   
4、效率和性能较高，能解析大于系统内存的文档。

**引用**

缺点：   
1、需要应用程序自己负责TAG的处理逻辑（例如维护父/子关系等），使用麻烦。   
2、单向导航，很难同时访问同一文档的不同部分数据，不支持XPath。

三、JDOM（Java-based Document Object Model）   
  
Java特定的文档对象模型。自身不包含解析器，使用SAX。

**引用**

优点：   
1、使用具体类而不是接口，简化了DOM的API。   
2、大量使用了Java集合类，方便了Java开发人员。

**引用**

缺点：   
1、没有较好的灵活性。   
2、性能较差。

四、DOM4J（Document Object Model for Java）   
  
简单易用，采用Java集合框架，并完全支持DOM、SAX和JAXP。

**引用**

优点：   
1、大量使用了Java集合类，方便Java开发人员，同时提供一些提高性能的替代方法。   
2、支持XPath。   
3、有很好的性能。

**引用**

缺点：   
1、大量使用了接口，API较为复杂。

五、StAX（Streaming API for XML）   
  
流模型中的拉模型分析方式。提供基于指针和基于迭代器两种方式的支持。

**引用**

优点：   
1、接口简单，使用方便。   
2、采用流模型分析方式，有较好的性能。

**引用**

缺点：   
1、单向导航，不支持XPath，很难同时访问同一文档的不同部分。

为了比较这五种方式在解析XML文档时的性能表现，我们来创建三个不同大小的XML文档：smallusers.xml（100KB）、middleusers.xml（1MB）、bigusers.xml（10MB）。我们分别用以上五种解析方式对这三个XML进行解析，然后打印出所有的用户信息，并分别计算它们所用的时间。测试代码会在文章后面的附件中给出，这里只比较它们的耗时。   
  
单位：s（秒）

|  |  |  |  |
| --- | --- | --- | --- |
|  | 100KB | 1MB | 10MB |
| DOM | 0.146s | 0.469s | 5.876s |
| SAX | 0.110s | 0.328s | 3.547s |
| JDOM | 0.172s | 0.756s | 45.447s |
| DOM4J | 0.161s | 0.422s | 5.103s |
| StAX Stream | 0.093s | 0.334s | 3.553s |
| StAX Event | 0.131s | 0.359s | 3.641s |

由上面的测试结果可以看出，性能表现最好的是SAX，其次是StAX Stream和StAX Event，DOM和DOM4J也有着不错的表现。性能最差的是JDOM。   
  
所以，如果你的应用程序对性能的要求很高，SAX当然是首选。如果你需要访问和控制任意数据的功能，DOM是个很好的选择，而对Java开发人员来讲，DOM4J是更好的选择。   
  
如果只需要做XML文档解析的话，综合性能、易用性、面向对象特征等各方面来衡量，StAX Event无疑是最好的选择。