**Java 处理 XML 的三种主流技术及介绍**

阅读次数: 692次 发布时间: 2012-08-16 10:31:20发布人: jvuentuslm

来源: 本站原创

**简介：** XML (eXtensible Markup Language) 意为可扩展标记语言，它已经是软件开发行业中大多数程序员和厂商用以选择作为数据传输的载体。本文作者对于 Java 处理 XML 的几种主流技术进行一些总结和介绍，希望帮助那些有不同需求的开发人员对于 XML 处理技术的作出最优的选择。

最初，XML 语言仅仅是意图用来作为 HTML 语言的替代品而出现的，但是随着该语言的不断发展和完善，人们越来越发现它所具有的优点：例如标记语言可扩展，严格的语法规定，可使用有意义的标记，内容 存储和表现分离等等优势注定了该语言从诞生之日起就会走向辉煌。 XML 语言在成为 W3C 标准之后进入到了一个快速发展的时期，当然它本身所具有的一系列优点和优势也注定了各大技术厂商对它的偏爱，Java 作为软件行业的一种开发技术也迅速作出了反应，出现了多种对 XML 支持的工具，本文将会从这个角度对 Java 处理 XML 的几种主流技术进行介绍，希望能对您有所帮助。在这篇文章中，您将会得到以下信息：

1、Java 提供了哪些优秀的类库及工具便于程序员对 XML 进行处理 ?

2、有了 DOM 了，其它工具类库还有必要么 ?

3、几个小例程带你快速了解这三种解析方式

Java 有哪些优秀的类库及工具便于程序员对 XML 进行处理 ?

●大名鼎鼎的 DOM

●绿色环保的 SAX

●默默无闻的 Digester

**XML 三种解析方式简介**

*大名鼎鼎的 DOM*

说它大名鼎鼎可是一点不为过，DOM 是 W3C 处理 XML 的标准 API，它是许多其它与 XML 处理相关的标准的基础，不仅是 Java，其它诸如 Javascript，PHP，MS .NET 等等语言都实现了该标准， 成为了应用最为广泛的 XML 处理方式。当然，为了能提供更多更加强大的功能，Java 对于 DOM 直接扩展工具类有很多，比如很多 Java 程序员耳熟能详的 JDOM，DOM4J 等等， 它们基本上属于对 DOM 接口功能的扩充，保留了很多 DOM API 的特性，许多原本的 DOM 程序员甚至都没有任何障碍就熟练掌握了另外两者的使用，直观、易于操作的方式使它深受广大 Java 程序员的喜爱。

*绿色环保的 SAX*

SAX 的应运而生有它特殊的需要，为什么说它绿色环保呢，这是因为 SAX 使用了最少的系统资源和最快速的解析方式对 XML 处理提供了支持。 但随之而来繁琐的查找方式也给广大程序员带来许多困扰，常常令人头痛不已，同时它对 XPath 查询功能的支持，令人们对它又爱又恨。

*默默无闻的 Digester：XML 的 JavaBean 化*

Digester 是 apache 基金组织下的一个开源项目，笔者对它的了解源于对 Struts 框架的研究，是否有很多程序员想要一解各大开源框架的设计甚至想要自己写一个功能强大的框架时会碰到这样一个难题： 这些形形色色的用 XML 语言标记的框架配置文件，框架底层是用什么技术来解析呢？ DOM 解析耗费时间，SAX 解析又过于繁琐，况且每次解析系统开销也会过大， 于是，大家想到需要用与 XML 结构相对应的 JavaBean 来装载这些信息，由此 Digester 应运而生。它的出现为 XML 转换为 JavaBean 对象的需求带来了方便的操作接口，使得更多的类似需求得到了比较完美的

解决方法， 不再需要程序员自己实现此类繁琐的解析程序了。与此同时 SUN 也推出了 XML 和 JavaBean 转换工具类 JAXB，有兴趣的读者可以自行了解。

**三种解析方式比较**

***DOM***

优缺点：实现 W3C标准，有多种编程语言支持这种解析方式，并且这 种方法本身操作上简单快捷，十分易于初学者掌握。其处理方式是将 XML 整个作为类似树结构的方式读入内存中以便操作及解析，因此支持应用程序对 XML 数据的内容和结构进行修改，但是同时由于其需要在处理开始时将整个 XML 文件读入到内存中去进行分析，因此其在解析大数据量的 XML 文件时会遇到类似于内存泄露以及程序崩溃的风险，请对这点多加注意。

**适用范围：**小型 XML 文件解析、需要全解析或者大部分解析 XML、需要修改 XML 树内容以生成自己的对象模型

***SAX***

SAX 从根本上解决了 DOM 在解析 XML 文档时产生的占用大量资源的问题。其实现是通过类似于流解析的技术，通读整个 XML 文档树，通过事件处理器来响应程序员对于 XML 数据解析的需求。由于其不需要将整个 XML 文档读入内存当中，它对系统资源的节省是十分显而易见的，它在一些需要处理大型 XML 文档以及性能要求较高的场合有起了十分重要的作用。**支持 XPath 查询**的 SAX 使得开发人员更加灵活，处理起 XML 来更加的得心应手。但是同时，其仍然有一些不足之处也困扰广大的开发人员：首先是它十分复杂的 API 接口令人望而生畏，其次由于其是属于类似流解析的文件扫描方式，因此不支持应用程序对于 XML 树内容结构等的修改，可能会有不便之处。

**适用范围：**大型 XML 文件解析、只需要部分解析或者只想取得部分 XML 树内容、有 XPath 查询需求、有自己生成特定 XML 树对象模型的需求

***Digester/JAXB***

优缺点 : 由于其是在上述两者的基础上衍生出来的工具类，为的是满足将 XML 转换为 JavaBean 的特殊需求，故而没有什么特别明显的优缺点。作为大名鼎鼎的开源框架 Struts 的 XML 解析工具 Digester，为我们带来了将 XML 转换为 JavaBean 的可靠方法。

**适用范围 :** 有将 XML 文档直接转换为 JavaBean 需求。

应用示例

下面给出一段用于解析的 XML 片段：

清单 1. XML 片段

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <?xml version="1.0" encoding="UTF-8"?>  <books>    <book id="001">       <title>Harry Potter</title>       <author>J K. Rowling</author>    </book>    <book id="002">       <title>Learning XML</title>       <author>Erik T. Ray</author>    </book>  </books> |

**DOM 解析 XML**

Java 中的 DOM 接口简介： JDK 中的 DOM API 遵循 W3C DOM 规范，其中 org.w3c.dom 包提供了 Document、DocumentType、Node、

NodeList、Element 等接口， 这些接口均是访问 DOM 文档所必须的。我们可以利用这些接口创建、遍历、修改 DOM 文档。

javax.xml.parsers 包中的 DoumentBuilder 和 DocumentBuilderFactory 用于解析 XML 文档生成对应的 DOM Document 对象。

javax.xml.transform.dom 和 javax.xml.transform.stream 包中 DOMSource 类和 StreamSource 类，用于将更新后的 DOM 文档写入 XML 文件。

下面给出一个运用 DOM 解析 XML 的例子：

清单 2. DOM 解析 XML

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | import java.io.File;   import java.io.IOException;   import javax.xml.parsers.DocumentBuilder;   import javax.xml.parsers.DocumentBuilderFactory;   import javax.xml.parsers.ParserConfigurationException;   import org.w3c.dom.Document;   import org.w3c.dom.Element;   import org.w3c.dom.Node;   import org.w3c.dom.NodeList;   import org.xml.sax.SAXException;  public class DOMParser {   DocumentBuilderFactory builderFactory = DocumentBuilderFactory.newInstance();   //Load and parse XML file into DOM   public Document parse(String filePath) {   Document document = null;   try {   //DOM parser instance   DocumentBuilder builder = builderFactory.newDocumentBuilder();   //parse an XML file into a DOM tree   document = builder.parse(new File(filePath));   } catch (ParserConfigurationException e) {   e.printStackTrace();   } catch (SAXException e) {   e.printStackTrace();   } catch (IOException e) {   e.printStackTrace();   }   return document;   }  public static void main(String[] args) {   DOMParser parser = new DOMParser();   Document document = parser.parse("books.xml");   //get root element   Element rootElement = document.getDocumentElement();  //traverse child elements   NodeList nodes = rootElement.getChildNodes();   for (int i=0; i < nodes.getLength(); i++)   {   Node node = nodes.item(i);   if (node.getNodeType() == Node.ELEMENT\_NODE) {   Element child = (Element) node;   //process child element   }   }  NodeList nodeList = rootElement.getElementsByTagName("book");   if(nodeList != null)   {   for (int i = 0 ; i < nodeList.getLength(); i++)   {   Element element = (Element)nodeList.item(i);   String id = element.getAttribute("id");   }   }   }   } |

在上面的例子中，DOMParser 的 Parse() 方法负责解析 XML 文件并生成对应的 DOM Document 对象。其中 DocumentBuilderFactory 用于生成 DOM 文档解析器以便解析 XML 文档。 在获取了 XML 文件对应的 Document 对象之后，我们可以调用一系列的 API 方便的对文档对象模型中的元素进行访问和处理。 需要注意的是调用 Element 对象的 getChildNodes() 方法时将返回其下所有的子节点，其中包括空白节点，因此需要在处理子 Element 之前对节点类型加以判断。

可以看出 DOM 解析 XML 易于开发，只需要通过解析器建立起 XML 对应的 DOM 树型结构后便可以方便的使用 API 对节点进行访问和处理，支持节点的删除和修改等。 但是 DOM 解析 XML 文件时会将整个 XML 文件的内容解析成树型结构存放在内存中，因此不适合用 DOM 解析很大的 XML 文件。

**SAX 解析 XML**

与 DOM 建立树形结构的方式不同，SAX 采用事件模型来解析 XML 文档，是解析 XML 文档的一种更快速、更轻量的方法。 利用 SAX 可以对 XML 文档进行有选择的解析和访问，而不必像 DOM 那样加载整个文档，因此它对内存的要求较低。 但 SAX 对 XML 文档的解析为一次性读取，不创建任何文档对象，很难同时访问文档中的多处数据。

下面是一个 SAX 解析 XML 的例子：

清单 3. SAX 解析 XML

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | import org.xml.sax.Attributes;   import org.xml.sax.SAXException;   import org.xml.sax.XMLReader;   import org.xml.sax.helpers.DefaultHandler;   import org.xml.sax.helpers.XMLReaderFactory;  public class SAXParser {  class BookHandler extends DefaultHandler {   private List nameList;   private boolean title = false;  public ListgetNameList() {   return nameList;   }   // Called at start of an XML document   @Override   public void startDocument() throws SAXException {   System.out.println("Start parsing document...");   nameList = new ArrayList();   }   // Called at end of an XML document   @Override   public void endDocument() throws SAXException {   System.out.println("End");   }  /\*\*   \* Start processing of an element.   \* @param namespaceURI Namespace URI   \* @param localName The local name, without prefix   \* @param qName The qualified name, with prefix   \* @param atts The attributes of the element   \*/   @Override   public void startElement(String uri, String localName, String qName,   Attributes atts) throws SAXException {   // Using qualified name because we are not using xmlns prefixes here.   if (qName.equals("title")) {   title = true;   }   }  @Override   public void endElement(String namespaceURI, String localName, String qName)   throws SAXException {   // End of processing current element   if (title) {   title = false;   }   }  @Override   public void characters(char[] ch, int start, int length) {   // Processing character data inside an element   if (title) {   String bookTitle = new String(ch, start, length);   System.out.println("Book title: " + bookTitle);   nameList.add(bookTitle);   }   }  }  public static void main(String[] args) throws SAXException, IOException {   XMLReader parser = XMLReaderFactory.createXMLReader();   BookHandler bookHandler = (new SAXParser()).new BookHandler();   parser.setContentHandler(bookHandler);   parser.parse("books.xml");   System.out.println(bookHandler.getNameList());   }   } |

SAX 解析器接口和事件处理器接口定义在 org.xml.sax 包中。主要的接口包括 ContentHandler、DTDHandler、EntityResolver 及 ErrorHandler。 其中 ContentHandler 是主要的处理器接口，用于处理基本的文档解析事件；DTDHandler 和 EntityResolver 接口用于处理与 DTD 验证和实体解析相关的事件； ErrorHandler 是基本的错误处理接口。DefaultHandler 类实现了上述四个事件处理接口。上面的例子中 BookHandler 继承了 DefaultHandler 类， 并覆盖了其中的五个回调方法 startDocument()、endDocument()、startElement()、endElement() 及 characters() 以加入自己的事件处理逻辑。

**Digester 解析 XML**

为了满足将 XML 转换为 JavaBean 的特殊需求，Apache 旗下的一个名为 Digester 的工具为我们提供了这么一个选择。由于最终是将 XML 转化为 JavaBean 存储在内存当中， 故而解析性能等方面其实与使用者并没有多大关系。解析的关键在于用以匹配 XML 的模式以及规则等，由于该工具较为复杂，限于篇幅，作者只能给予简单的介绍。

下面是一个 Digester 解析 XML 的例子片段：

清单 4. Digester 解析 XML

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // 定义要解析的 XML 的路径，并初始化工具类   File input = new File("books.xml");   Digester digester = new Digester();  // 如果碰到了 这个标签，应该初始化 test.myBean.Books 这个 JavaBean 并填装相关内容   digester.addObjectCreate("books", "test.myBean.Books");   digester.addSetProperties("books");   // 如果碰到了 这个标签，同上初始化 test.myBean.Book 这个 JavaBean   digester.addObjectCreate("books/book", "test.myBean.Book");   digester.addSetProperties("books/book");   // 通过调用上面已经初始化过的 JavaBean 的 addBook() 方法来把多个 加到一个集合中   digester.addSetNext("books/book", "addBook", "test.myBean.Book");  // 定义好了上面的解析规则后，就可以开始进行解析工作了   Books books = (Books) digester.parse(input); |

上述代码简单的向读者展示了 Digester 处理 XML 的一些要点，主要是说明了一些模式以及规则的匹配。 简言之，Digester 就是一种用来把一个 XML 转化为一个与该 XML 结构类似的 JavaBean。你可以把 XML 根元素想象成一个 JavaBean， 该根元素的 attribute 就是这个 JavaBean 的各种 Field，当该根元素有其他子 tag 时，又要把这个子 tag 想象成一个个新的 XML，将其视为一个新的 JavaBean， 并作为一个 Field 加入到父 Bean 当中，然后以此类推，通过循环的方式将整个 XML 进行解析。

**结束语**

本文介绍了 Java 解析 XML 的三种常用技术，其中 DOM 易于上手，程序易于理解，但缺点在于占用内存大，不适合于解析较大的 XML 文件； SAX 基于事件模型占用系统资源少，能够胜任较大的 XML 文件解析，但解析过程较为繁琐查找元素不方便； Digester/JAXB 基于上述两种技术衍生而来。文中的实例向读者展示了三种 API 的基本使用方法， 在实际开发过程中使用那种技术解析 XML 更好要依据各自的优缺点视具体情况而定。