

# XML 使用、命令空间、约束、解析

## 1 XML初识

---

### 1.1 概念

XML(Extensible Markup Language) , 可拓展标记语言

- 标记, 指的是标记语言, 也称标签语言, 可以用一系列的标签来对数据进行描述。

例如, `<name>tom</name>`

- 拓展, 指的是用户可以自定义标签

### 1.2 作用

1. XML可以作为数据传输的标准 ( **重要** )

作为数据的一种传输标准, 要考虑可读性、可扩展性、可维护性, 并且最好是和语言无关。XML作为传输数据的标准就比较适合, 在上述方面表现都很好。

2. XML可以作为配置文件 ( **重要** )

很多软件和框架, 都会提供XML文件配置的方式, 以便可以方便快捷的修改软件或框架的功能

3. XML可以持久化数据

可以将数据存到xml文件中, 把xml当做一个临时的“数据库”, 当然, 重要的数据还是要存到真的数据库中

4. XML 简化平台变更

在系统更换平台的时候, 普通的数据会存在不兼容的问题, 但是XML 数据以文本格式存储, 这使得 XML 在不损失数据的情况下, 更容易扩展和升级。

## 2 XML历史

---

- 1969年 GML 通用标记语言

用于计算机之间的通信, 通信就会传输数据, 那么就需要一种数据的规范

- 1985年 SGML 标准通用标记语言

对GML进行完善

- 1993年 HTML 超文本的标记语言(HyperText Markup Language)

随着万维网的推广, 在SGML的基础上, 又出现了HTML语言, 用于万维网上的页面展示

- 1998年 XML 可扩展的标记语言(Extensible Markup Language)

HTML有不少的缺陷, HTML语言的标记不能自定义, HTML语言的标记本身不能用来描述数据等。

W3C组织在1998年推出了可扩展标记语言XML, 最初的本意是用来替代HTML语言的, 但是两种语言还有一定差异的, 所以中间出现了一种过渡的语言: XHTML

但实际上XML语言已经很难替代HTML语言了, 因为HTML语言的使用在整个万维网上使用太广泛了。

---

## XML 和 HTML对比

1. XML 主要是用来**描述**数据的
2. HTML 主要是用来**展现**数据的

## 3 XML语法

---

### 3.1 文档声明

XML 声明文件的可选部分，如果存在需要放在文档的第一行，如下所示：

```
1  <?xml version="1.0" encoding="utf-8"?>
```

这里描述了XML的版本，以及XML文档所使用的编码

### 3.2 元素

**元素**，指的就是XML中的标记，这种标记也被称为标签、节点。

一个XML元素可以包含字母、数字以及其它一些可见字符，但必须遵守下面的一些规范:

- 不能以数字或部分标点符号开头
- 不能包含空格和特定的几个符号
- 标签必须成对出现，不允许缺省结束标签
- 根元素有且只有一个
- 大小写敏感
- 允许多层嵌套但是不允许交叉嵌套

例如，

```
1  <element>元素测试</element>
```

注意，XML中的第一行，文档声明部分，是固定写法，不需要关闭标签。

#### XML 必须包含根元素

根元素有且只能有一个，它是所有其他元素的父元素。

例如，

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <child>
4          <subchild>.....</subchild>
5      </child>
6  </root>
```

这里的 **<root>** 标签就是根元素

**元素是可以包含标签体的**，也就是在开始标签和结束标签之间写一些内容，例如

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <root>
3   <a>www.baidu.com</a>
4 </root>
```

**元素也可以不包含标签体**，也就是在开始标签和结束标签中间什么也不写，例如

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <root>
3   <a></a>
4 </root>
```

对于XML标签中出现的空格和换行，在XML解析程序都会当作标签内容进行处理。

例如，以下两个 `<a>` 标签中的内容是有区别的

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <root>
3   <a>www.baidu.com</a>
4 </root>
```

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <root>
3   <a>
4     www.baidu.com
5   </a>
6 </root>
7
```

注意，在一些第三方封装好的解析工具类中，是会自动忽略掉这些空格和换行的

## XML 标签对大小写敏感

标签 `<Letter>` 与标签 `<letter>` 是不同的。必须使用相同的大小写来编写开始标签和关闭标签。

例如，

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <root>
3   <message>这是正确的</message>
4 </root>
```

例如，

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <root>
3   <Message>这是错误的</message>
4 </root>
```

## XML标签允许 嵌套 但是不允许 交叉

例如，

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <message>
4          <a>这是正确的</a>
5      </message>
6  </root>

```

例如，

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <message>
4          <a>这是错误的 </message>
5      </a>
6  </root>

```

### 3.3 属性

XML属性 ( Attribute ) 提供了元素相关的一些额外信息

XML 属性写在开始标签中，并且属性值必须加引号，例如，可以是双引用

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <person sex="female"></person>
4  </root>

```

或者也可以是单引用

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <person sex='female'></person>
4  </root>

```

一个元素可以有多个属性，它的基本格式为：<元素名 属性名="属性值" 属性名="属性值">

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <person sex="female" age='18' email='briup@briup.com'></person>
4  </root>

```

### 3.4 示例

大多数情况下，使用子标签或者属性，来描述数据都是可以的，这里并没有强制的规则。

例如，使用date属性来描述时间

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <note date="10/01/2008">
3      <to>tom</to>
4      <from>mary</from>
5      <msg>Love~~~</msg>
6  </note>

```

例如，使用date元素来描述时间

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <note>
3      <date>10/01/2008</date>
4      <to>tom</to>
5      <from>mary</from>
6      <msg>Love~~~</msg>
7  </note>

```

例如，使用date元素和其扩展元素来描述数据

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <note>
3      <date>
4          <day>10</day>
5          <month>01</month>
6          <year>2008</year>
7      </date>
8      <to>tom</to>
9      <from>mary</from>
10     <msg>Love~~~</msg>
11 </note>

```

如果有多个note标签的时候，可以使用id进行区别：

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <messages>
3      <note id="1">
4          <date>
5              <day>10</day>
6              <month>01</month>
7              <year>2008</year>
8          </date>
9          <to>tom</to>
10         <from>mary</from>
11         <msg>Together~~~</msg>
12     </note>
13     <note id="2">
14         <date>
15             <day>10</day>
16             <month>02</month>
17             <year>2008</year>
18         </date>
19         <to>tom</to>
20         <from>mary</from>
21         <msg>Together~~~</msg>
22     </note>
23 </messages>

```

### 3.5 实体

在 XML 中，一些字符拥有特殊的意义，在XML文档中，是不能直接使用的。

例如，如果把字符 "<" 放在 XML 元素中，会发生错误，这是因为解析器会把它当作新元素的开始。

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <message>if salary < 1000 then</message>
```

浏览器中的错误信息：

**This page contains the following errors:**

error on line 4 at column 21: StartTag: invalid element name

**Below is a rendering of the page up to the first error.**

这时候，可以使用XML中预定义的实体，来代替这个特殊符号，例如，

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <message>if salary &lt; 1000 then</message>
```

XML中实体的格式：**&实体名;**

XML中预定义实体有，任何用到这些字符的地方，都可以使用实体来代替

实体	字符	简介
&lt;	<	Less than
&gt;	>	Greather than
&amp;	&	mpersand
&apos;	'	Apostrophe
&quot;	"	Quotation mark

当然这些预先定义的实体，还支持自定义实体。

定义格式：

```
1 <!DOCTYPE 根元素名称[
2     <!ENTITY 实体名 实体内容>
3 ]>
```

例如，

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE root[
3      <!ENTITY company "杰普软件科技有限公司">
4  ]>
5  <root>
6      <name>&company;</name>
7  </root>

```

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```

▼ <root>
  <name>杰普软件科技有限公司</name>
</root>

```

在XML中定义实体，然后使用实体，就像java程序中，定义变量，然后使用变量一样。

**观察一个实体定义出来的的过程：**

```

1  <>
2
3  <!--
4
5  <!DOCTYPE>
6
7  <!DOCTYPE 根元素>
8
9  <!DOCTYPE 根元素[]>
10
11 <!DOCTYPE 根元素[
12
13 ]>
14
15 <!DOCTYPE 根元素[
16     <!ENTITY>
17 ]>
18
19 <!DOCTYPE 根元素[
20     <!ENTITY 实体名 "实体值">
21 ]>

```

## 3.6 注释

XML中的注释，和java程序中的注释一样，都是对当前程序作出解释说明的。

格式：

```

1  <!-- 这是一个注释 -->

```

```
1  <>
2
3  <!-->
4
5  <!-->
6
7  <!-- 注释 -->
```

使用注释的时候，需要注意以下几点：

1. 注释内容中不要出现 --
2. 不要把注释放在标签中间
3. 注释不能嵌套

例如，注释中不能出现 --

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <name>briup</name>
4  </root>
5  <!-- -- -->
```

浏览器解析结果：

**This page contains the following errors:**  
error on line 5 at column 6: Double hyphen within comment: <!--  
**Below is a rendering of the page up to the first error.**

例如，不要把注释放在标签中间

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <name <!-- 这是说明名字的 --> >briup</name>
4  </root>
5
```

浏览器解析结果：

**This page contains the following errors:**  
error on line 3 at column 8: error parsing attribute name  
**Below is a rendering of the page up to the first error.**

例如，注释不能嵌套

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <!-- 外层注释<!-- 内部注释--> -->
4      <name>briup</name>
5  </root>
6
```



浏览器解析结果：

This page contains the following errors:

error on line 3 at column 15: Comment must not contain '--' (double-hyphen)

Below is a rendering of the page up to the first error.

### 3.7 CDATA

XML 解析器，会解析 XML 文档中所有的文本。

当某个 XML 元素被解析时，其标签之间的文本也会被解析。

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <tag>
4          <name>&amp;</name>
5          <entity>&amp;</entity>
6      </tag>
7  </root>
```

浏览器解析结果：

```
▼<root>
  ▼<tag>
    <name>&</name>
    <entity>&</entity>
  </tag>
</root>
```

在xml中

- 解析器进行解析的内容，称为PCDATA (Parsed CDATA)
- 解析器不会解析的内容，称为CDATA, (Character Data)

因为在一些情况下，我们在xml中编写的特殊内容，例如特殊字符、代码等，并不希望解析器去解析，而是希望把这些内容按字符串原样输出即可，不需要做额外任何解析处理。

这时候，我们就可以把这些内容，写在指定的CDATA区域内即可。

CDATA区域的格式要求：

```
1  <![CDATA[需要原样输出的字符串]]>
```

```
1  <>
2
3  <!--
4
5  <![ ]>
6
7  <![CDATA]>
8
9  <![CDATA[ ]>
10
11 <![CDATA[
12
13 ]>
```

例如，

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <root>
3      <tag>
4          <name>&lt;/name>
5          <entity><![CDATA[&lt;]]></entity>
6      </tag>
7
8  </root>
9
```

浏览器解析结果：

---

```
-<root>
-  <tag>
    <name>&</name>
    <entity>&lt;/entity>
  </tag>
</root>
```

注意，XML 文档中的所有文本均会被解析器解析。只有 CDATA 区段中的文本会被解析器忽略

注意，在不同浏览器中，对XML解析显示的结果稍有差异

## 4 XML和CSS

XML文件中的内容，可以配合CSS进行内容的样式渲染，例如控制字体大小、颜色等

## 4.1 CSS

CSS ( Cascading Style Sheets ) 层叠样式表，通常用来给HTML中内容、或者XML中内容进行样式的渲染，这里需要先简单的使用一下，配合XML来渲染需要显示的内容。

例如，给name标签和age标签的内容添加指定样式

```
1  name{
2      font-size:30px;
3      font-weight:bold;
4      color:red;
5  }
6  age{
7      font-size:30px;
8      font-weight:bold;
9      color:green;
10 }
11
```

## 4.2 指令

处理指令，简称PI ( processing instruction )，可以用来指定解析器如何解析XML文档内容。

例如，在XML文档中可以使用xml-stylesheet指令，通知XML解析引擎，使用test.css文件来渲染xml内容。

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="test.css" type="text/css"?>
3  <class>
4      <student id="001">
5          <name>张三</name>
6          <age>20</age>
7      </student>
8      <student id="002">
9          <name>李四</name>
10         <age>20</age>
11     </student>
12 </class>
```

最后内容被渲染的效果：

---

**张三 20 李四 20**

## 5 XML命名空间

---

因为XML文档中的标签，是用户可以自定义的，所以可能在两个不同的XML文档中，出现相同名字的标签元素。

例如，

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <table>
3      <tr>
4          <td>hello</td>
5          <td>world</td>
6      </tr>
7  </table>
```

这个xml中的table标签，表示的是一个表格

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <table>
3      <width>80</width>
4      <length>120</length>
5  </table>
```

这个xml中的table标签，表示的是一张桌子

这种情况下，可以使用**前缀**，来避免这样的冲突

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <x:table>
3      <tr>
4          <td>hello</td>
5          <td>world</td>
6      </tr>
7  </x:table>
```

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <y:table>
3      <width>80</width>
4      <length>120</length>
5  </y:table>
```

还可以进一步，声明这个前缀是属于哪个**命名空间**：

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <x:table xmlns:x="http://www.briup.com/XML/ns/x/">
3      <tr>
4          <td>hello</td>
5          <td>world</td>
6      </tr>
7  </x:table>
```

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <y:table xmlns:y="http://www.briup.com/XML/ns/y/">
3     <width>80</width>
4     <length>120</length>
5 </y:table>
```

也可以不要前缀，直接定义默认的命名空间：

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <table xmlns="http://www.briup.com/XML/ns/">
3     <tr>
4         <td>hello</td>
5         <td>world</td>
6     </tr>
7 </table>
```

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <table xmlns="http://www.briup.com/XML/ns/">
3     <width>80</width>
4     <length>120</length>
5 </table>
```

其实，这些做法的目的，就是为了让这个自定义的标签具有**唯一性**，不会和别人定义的冲突。

思考，java中自定义的一个类，怎么保证不会和别人定义的类有相同名字的冲突？

## 6 XML约束

XML是可扩展的标记语言，用户在使用过程中，可以自定义任何一个标签元素。

但是，在很多情况下，需要让用户按照我们的要求去编写XML文件内容，并不能真的让他自己随意的自定义标签。

这时候，就需要对XML文件内容作出约束，约束的目的，就是为了让用户按照我们提前设置好的要求，去编写XML内容。

可以使用dtd文件或者Schema文件对XML文件内容作出约束，dtd文件和schema文件中，只是语法不同，但是最终的效果都是一样的。

## 6.1 良构和有效

一个XML文件，如果里面的内容，满足XML的基本语法要求，那么这个XML文件就可以说是良构的。

在XML文件是良构的基础上，如果这个XML文件还通过dtd文件或者Schema文件的约束验证，那么这个XML文件就可以说是有效的。

所以，总结如下：

- 良构的XML文件不一定是有效的
- 有效的XML文件一定是良构的

## 6.2 DTD

DTD(Document Type Define)，dtd文件中描述并规定了元素、属性和其他内容在xml文档中的使用规则。

DTD文件的后缀名为 .dtd。

### 6.2.1 语法

注意，DTD的语法，作为了解内容即可，不要求编写，但是要能看懂和知道如何使用DTD

描述**元素**的语法格式：

```
1  <!ELEMENT 元素名 (内容模式)>
```

1. EMPTY：元素不能包含子元素和文本（空元素）
2. (#PCDATA)：可以包含任何字符数据，但是不能在其中包含任何子元素
3. ANY：元素内容为任意的，主要是使用在元素内容不确定的情况下
4. 修饰符：()|+\*?,
  - () 用来给元素分用组
  - | 在列出的元素中选择一个
  - +表示该元素最少出现一次，可以出现多次(1或n次)
  - \*表示该元素允许出现零次到任意多次(0到n次)
  - ?表示该元素可以出现，但只能出现一次(0到1次)
  - ,对象必须按指定的顺序出现

例如，

```
1  <!ELEMENT students (stu*)>
2  <!ELEMENT stu (id|name|age)*>
```

说明，

- 根元素students里面可以出现stu子元素0到n次
- stu元素中的属性有id、name、age子元素
- id、name、age子元素出现的顺序没有要求
- id、name、age子元素都可以出现0~n次

描述**属性**的语法格式：

```

1  <!-- 元素名称
2      属性名称      属性类型      属性特点
3      属性名称      属性类型      属性特点
4  >

```

属性类型：

1. CDATA：属性值可以是任何字符（包括数字和中文）
2. ID：属性值必须唯一,属性值必须满足xml命名规则
3. IDREF：属性的值指向文档中其它地方声明的ID类型的值。
4. IDREFS:同IDREF，但是可以具有由空格分开的多个引用。
5. enumerated：(枚举值1|枚举值2|枚举值3...)，属性值必须在枚举值中

属性特点：

1. #REQUIRED：元素的所有示例都必须有该属性
2. #IMPLIED：属性可以不出现
3. default-value：属性可以不出现，但是会有默认值
4. #FIXED：属性可以不出现，但是如果出现的话必须是指定的属性值

例如，

```

1  <!-- 元素名称 (student+)>
2  <!-- 元素名称 (name,age?,score*)>
3  <!-- 元素名称 id CDATA #REQUIRED>
4  <!-- 元素名称 (#PCDATA)>
5  <!-- 元素名称 firstName CDATA #IMPLIED>
6  <!-- 元素名称 (#PCDATA)>
7  <!-- 元素名称 xuAge CDATA #FIXED "20">
8  <!-- 元素名称 (#PCDATA)>
9  <!-- 元素名称 sel (60|80|100) #REQUIRED>

```

## 6.2.2 引入

一个XML文件中，引入DTD文件，对其进行约束验证的方式有两种：

1. 内部的DTD，DTD和xml文档在同一个文件中。（不常用）

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- 内部DTD -->
3  <!DOCTYPE students[
4      <!-- 元素名称 (stu)>
5      <!-- 元素名称 (id,name,age)>
6      <!-- 元素名称 id (#PCDATA)>
7      <!-- 元素名称 name (#PCDATA)>
8      <!-- 元素名称 age (#PCDATA)>
9  ]>
10
11  <students>
12      <stu>
13          <id>1</id>
14          <name>tom</name>
15          <age>20</age>

```

```
16     </stu>
17 </students>
18
```

## 2. 外部的DTD，DTD和xml文档不在同一个文件中。（常用）

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- 外部DTD -->
3  <!DOCTYPE students SYSTEM "dtd/students.dtd">
4  <students>
5      <stu>
6          <id>1</id>
7          <name>tom</name>
8          <age>20</age>
9      </stu>
10 </students>
```

在XML中，引入外部的一个DTD文件的方式，有两种：

1. 本地DTD文件引入，这种方式一般都是自己编写DTD，然后自己使用。

```
1  <!DOCTYPE 根元素 SYSTEM "DTD文件的路径">
```

2. 公共DTD文件引入，这种方式可以让很多人使用共同的一个DTD文件。

```
1  <!DOCTYPE 根元素 PUBLIC "DTD名称" "DTD文档的URL">
```

思考，你是否可以想象到一些使用本地DTD和公共DTD的场景？

### 6.2.3 使用

DTD在使用的时候，分为两种情况

- 本地DTD
- 公共DTD

#### 1 如果是本地DTD的引入：

student.dtd

```
1  <!ELEMENT students (stu)>
2  <!ELEMENT stu (id,name,age)>
3  <!ELEMENT id (#PCDATA)>
4  <!ELEMENT name (#PCDATA)>
5  <!ELEMENT age (#PCDATA)>
```

students.xml



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE students SYSTEM "dtd/students.dtd">
3 <students>
4     <stu>
5         <id>1</id>
6         <name>tom</name>
7         <age>20</age>
8     </stu>
9     <test></test>
10 </students>

```

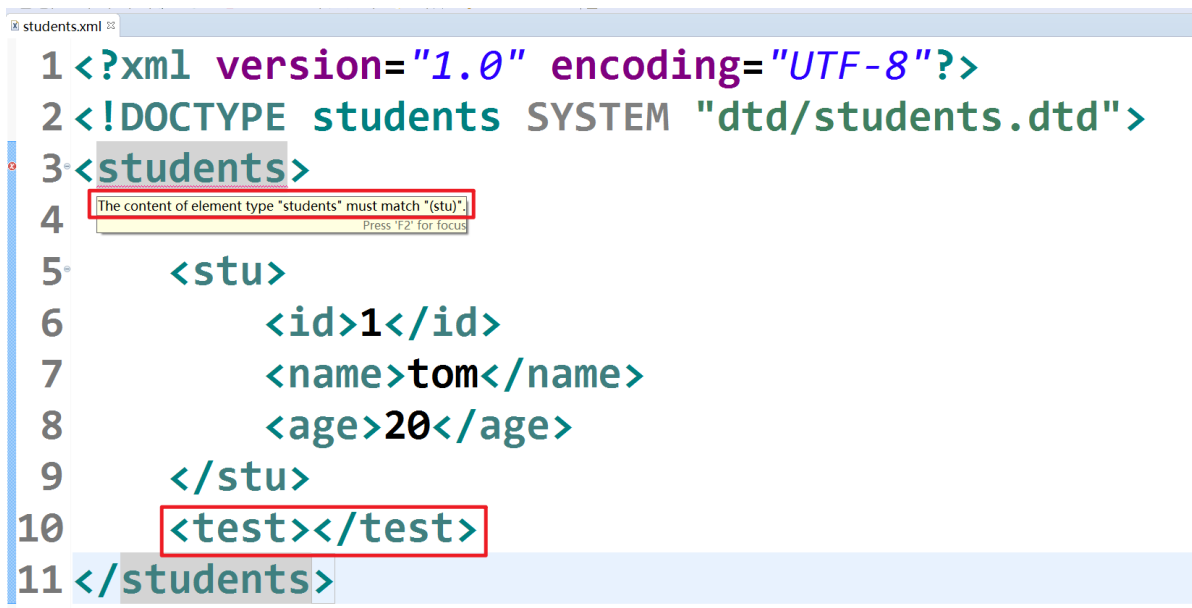
在Eclipse中，创建并编写以上两个文件，注意它们两个文件的相对位置：

```

v demo
  v dtd
    students.dtd
  students.xml

```

Eclipse会自动根据XML中引入的DTD位置，进行查找，找到DTD后，自动完整约束验证，验证失败会报错：



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE students SYSTEM "dtd/students.dtd">
3 <students>
4     <stu>
5         <id>1</id>
6         <name>tom</name>
7         <age>20</age>
8     </stu>
9     <test></test>
10 </students>

```

可以看出，DTD中，没有配置test标签元素，我们在XML中编写了test标签元素，那么就验证失败，然后报错！

注意，只有DTD验证通过了，XML文件才是**有效的**，Eclipse中就不会报错了。

思考，如果不引入DTD，或者引入的DTD没有找到，那么在Eclipse中，这个XML会报错么？

思考，把这个XML文件中的标签删掉，然后按Eclipse的自动提示快捷键（alt+/），观察是否有提示内容？

## 2 如果是公共DTD的引入：

一般在使用第三方框架的时候，官方会提供一个XML配置文件和一个DTD文件（也可能是Scheme文件），这时候都是这种引入方式，来对此XML文件做验证，目的就是让我们能按照官方要求的方式去配置XML文件内容！

这里使用hibernate框架的XML配置文件和DTD文件：

hibernate.cfg.xml

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <!DOCTYPE hibernate-configuration PUBLIC
3      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5
6  <hibernate-configuration>
7
8
9  </hibernate-configuration>
```

hibernate-configuration-3.0.dtd

```
1  <!-- Hibernate file-based configuration document.
2
3
4  <!DOCTYPE hibernate-configuration PUBLIC
5      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
6      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
7
8  An instance of this document contains property settings and references
9  to mapping files for a number of SessionFactory instances to be listed
10 in JNDI.
11
12 -->
13
14 <!ELEMENT hibernate-configuration (session-factory,security?)>
15
16 <!ELEMENT property (#PCDATA)>
17 <!ATTLIST property name CDATA #REQUIRED>
18
19 <!ELEMENT mapping EMPTY> <!-- reference to a mapping file -->
20 <!ATTLIST mapping resource CDATA #IMPLIED>
21 <!ATTLIST mapping file CDATA #IMPLIED>
22 <!ATTLIST mapping jar CDATA #IMPLIED>
23 <!ATTLIST mapping package CDATA #IMPLIED>
24 <!ATTLIST mapping class CDATA #IMPLIED>
25
26 <!ELEMENT class-cache EMPTY>
27 <!ATTLIST class-cache class CDATA #REQUIRED>
28 <!ATTLIST class-cache region CDATA #IMPLIED>
29 <!ATTLIST class-cache usage (read-only|read-write|nonstrict-read-
30 write|transactional) #REQUIRED>
31 <!ATTLIST class-cache include (all|non-lazy) "all">
32
33 <!ELEMENT collection-cache EMPTY>
```

```

33 <!--ATTLIST collection-cache collection CDATA #REQUIRED>
34 <!--ATTLIST collection-cache region CDATA #IMPLIED>
35 <!--ATTLIST collection-cache usage (read-only|read-write|nonstrict-read-
write|transactional) #REQUIRED>
36
37 <!--ELEMENT event (listener*)>
38 <!--ATTLIST event type (auto-flush|merge|create|create-onflush|delete|dirty-
check|evict|flush|flush-entity|load|load-collection|lock|refresh|replicate|save-
update|save|update|pre-load|pre-update|pre-insert|pre-delete|pre-collection-
recreate|pre-collection-remove|pre-collection-update|post-load|post-update|post-
insert|post-delete|post-collection-recreate|post-collection-remove|post-
collection-update|post-commit-update|post-commit-insert|post-commit-delete)
#REQUIRED>
39
40 <!--ELEMENT listener EMPTY>
41 <!--ATTLIST listener type (auto-flush|merge|create|create-onflush|delete|dirty-
check|evict|flush|flush-entity|load|load-collection|lock|refresh|replicate|save-
update|save|update|pre-load|pre-update|pre-insert|pre-delete|pre-collection-
recreate|pre-collection-remove|pre-collection-update|post-load|post-update|post-
insert|post-delete|post-collection-recreate|post-collection-remove|post-
collection-update|post-commit-update|post-commit-insert|post-commit-delete)
#IMPLIED>
42 <!--ATTLIST listener class CDATA #REQUIRED>
43
44 <!--ELEMENT session-factory (property*, mapping*, (class-cache|collection-cache)*,
event*, listener*)>
45 <!--ATTLIST session-factory name CDATA #IMPLIED> <!-- the JNDI name -->
46
47 <!--ELEMENT security (grant*)>
48 <!--ATTLIST security context CDATA #REQUIRED> <!--the JACC contextID-->
49
50 <!--ELEMENT grant EMPTY>
51 <!--ATTLIST grant role CDATA #REQUIRED>
52 <!--ATTLIST grant entity-name CDATA #REQUIRED>
53 <!--ATTLIST grant actions CDATA #REQUIRED>
54

```

其中，这里的重点是XML文件中的头部声明：

```

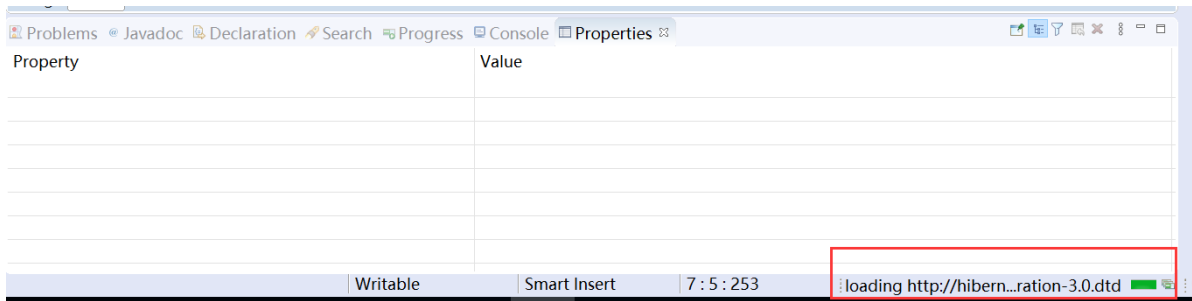
1 <!DOCTYPE hibernate-configuration PUBLIC
2     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

```

说明，

- Eclipse默认会根据第一个双引号中的内容，查找对应的DTD所在位置
- 如果有DTD文件，那么就读取并验证，同时可以根据DTD的内容给出自动提示的信息（Alt+/）
- 如果没有DTD文件，那么就根据第二个双引号中的内容（URL地址），去网络中下载并读取、验证

例如，



注意，如果之前你的Eclipse中引入过这个DTD，也就是之前下载过，那么就看到右下角这个下载状态了！

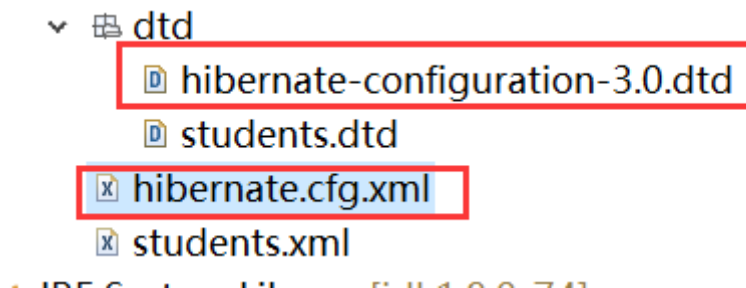
其实，第二个双引号中的URL，就是这个XML对应DTD的下载地址，我们也可以手动下载：



注意，如果网络出现问题，那么Eclipse就无法从这个URL获取指定的DTD文件。  
即使网络没有问题，但是也会可能由于URL地址本身问题，Eclipse也无法下载获取其内容。

那么，在这种情况下，我们还是需要自己**手动**在Eclipse中进行**配置**，配置的目的，就是将这个XML，和其对应的DTD文件进行关联，以便让Eclipse可以通过第一个双引号中的内容，找到它关联的DTD文件，然后读取并验证！

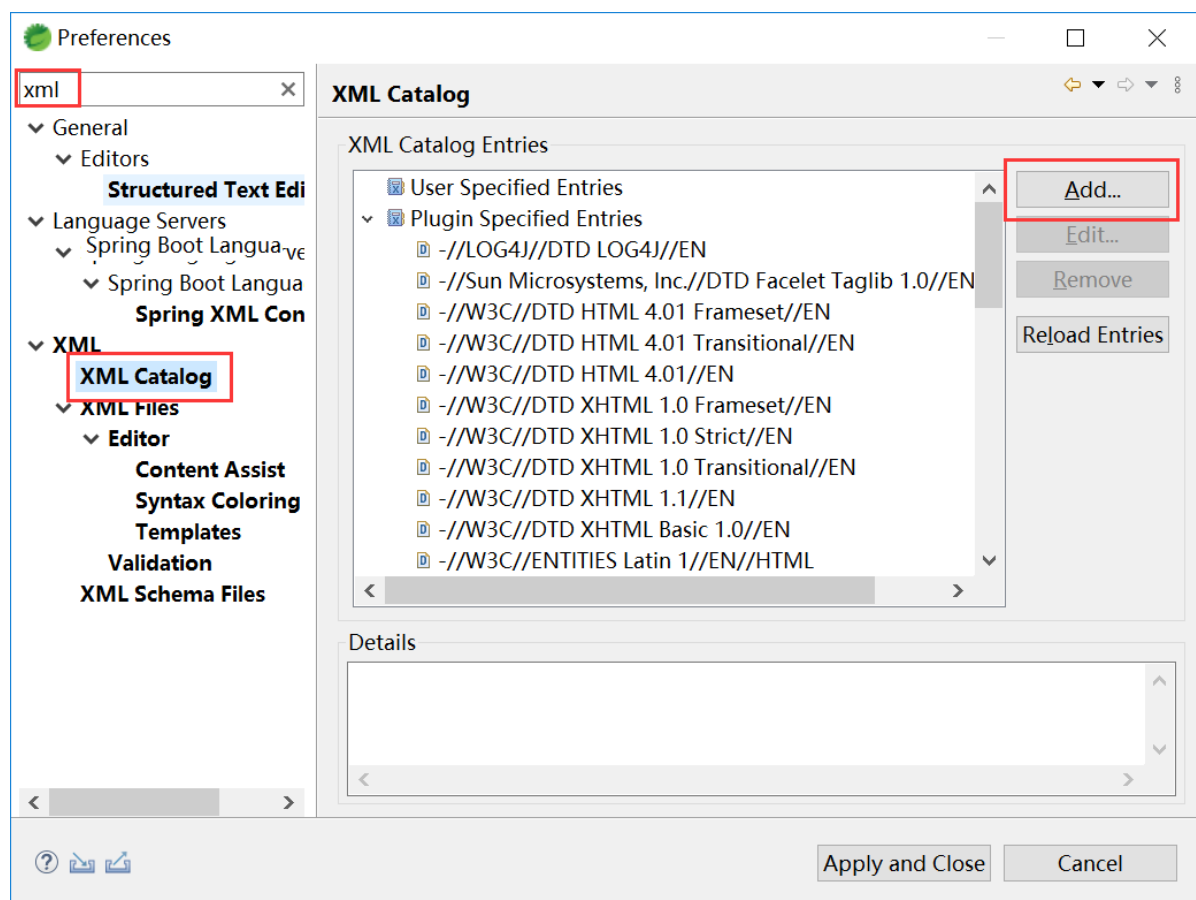
先保存DTD文件在本地：



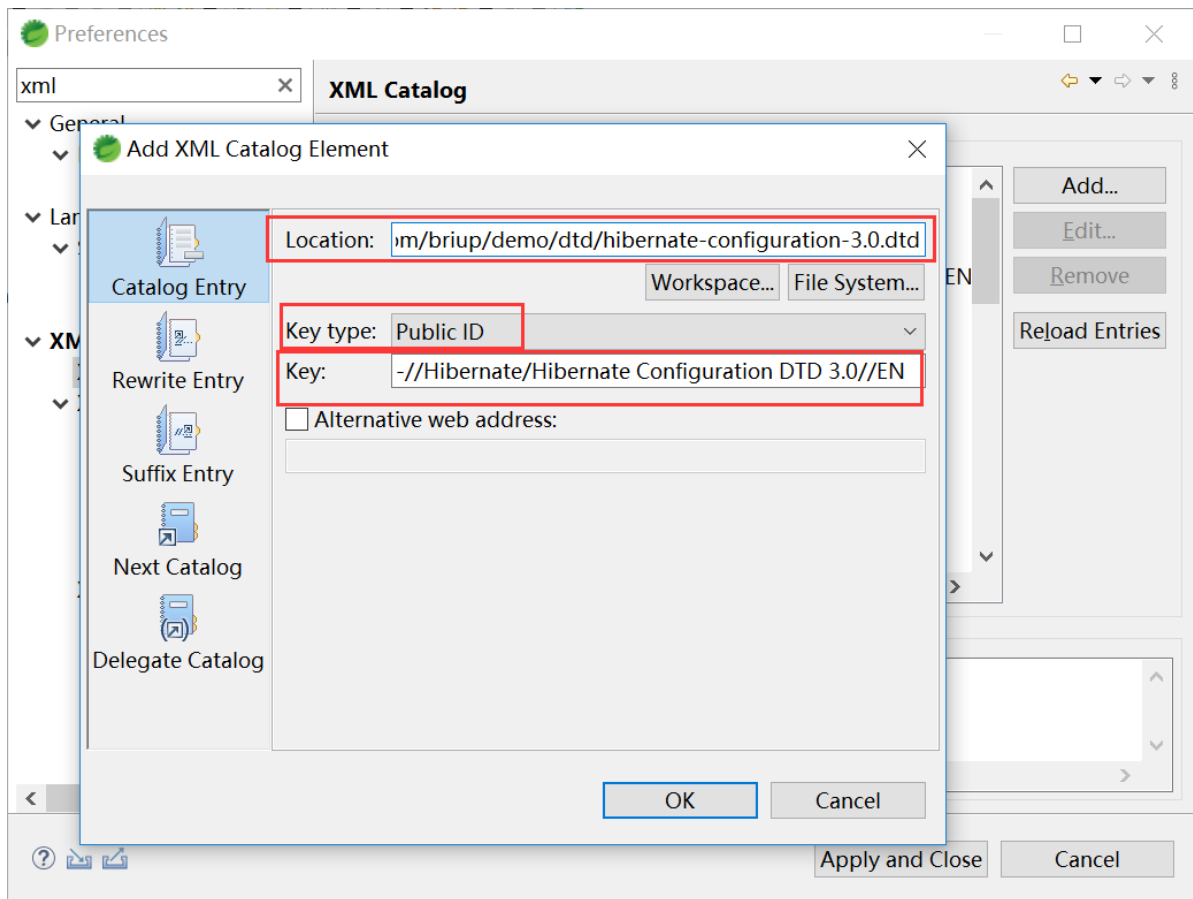
赋值XML头部声明中第一个双引号中间的内容，这个内容作为key，要去配置的value就是DTD的存放位置：

```
1 -//Hibernate/Hibernate Configuration DTD 3.0//EN
```

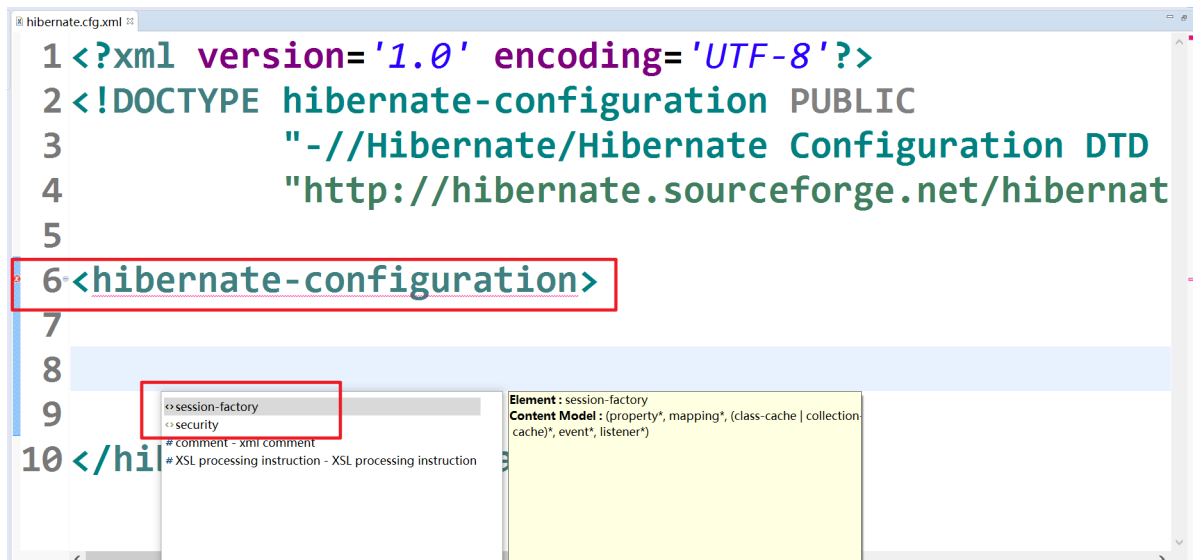
然后再Window-》Preferences中，搜索xml，并选择XML Catalog：



点击Add按钮，将上面复制的内容粘贴到key的位置，key type选择public ID，然后选择到DTD存放的位置：



最后，保存关闭即可。这时候，在XML中，按提示快捷键（alt+/），就可以看到Eclipse给我们提示的内容了，这个内容就是hibernate框架官方要求我们在配置文件中编写的内容！



可以看出，这时候已经有了提示信息了，根元素处报错，是因为我们还没有按照DTD的要求对该文件进行配置

注意，上一步配置完成之后，需要把XML文件关闭，再重新打开，以便让刚刚的配置生效！

## 6.3 Schema

Schema文件也可以用来对XML文件内容进行约束、验证，Eclipse也可以根据schema验证进行自动提示

Schema本身单词的含义就是：提要，纲要

Schema和DTD区别，

- DTD文件中是DTD的专用语法，Schema文件中就是正常的XML语法，所以Schema文件其实也是个XML文件
- DTD文件的后缀是.dtd，Schema文件的后缀是.xsd，xsd的意思为XML Schema Definition
- DTD文件中的约束条件相对简单，Schema文件中的约束条件更加详细、复杂
- Schema的出现是为了替代DTD文件，但是平时的使用中，两者都会见到

很多第三方的框架提供的配置文件中，除了DTD之外，也会使用Schema，那么我们就需要知道，在Eclipse中，如何将XML和对应的Schema文件进行关联，以便Eclipse可以使用Schema对XML进行约束、验证，并支持自动提示。

这里使用spring框架的XML配置文件和Schema文件：

bean.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:context="http://www.springframework.org/schema/context"
4       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans
6                           http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
7                           http://www.springframework.org/schema/context
8                           http://www.springframework.org/schema/context/spring-context-4.3.xsd">
9
10
11 </beans>
```

spring-beans-4.3.xsd

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2
3 <xsd:schema xmlns="http://www.springframework.org/schema/beans"
4             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5             targetNamespace="http://www.springframework.org/schema/beans">
6
7     <xsd:import namespace="http://www.w3.org/XML/1998/namespace"/>
8
9     <xsd:annotation>
10         <xsd:documentation><![CDATA[
11             Spring XML Beans Schema, version 4.3
12             Authors: Juergen Hoeller, Rob Harrop, Mark Fisher, Chris Beams
13
14             This defines a simple and consistent way of creating a namespace
15             of JavaBeans objects, managed by a Spring BeanFactory, read by
16             XmlBeanDefinitionReader (with DefaultBeanDefinitionDocumentReader).
```

```

17
18     This document type is used by most Spring functionality, including
19     web application contexts, which are based on bean factories.
20
21     Each "bean" element in this document defines a JavaBean.
22     Typically the bean class is specified, along with JavaBean properties
23     and/or constructor arguments.
24
25     A bean instance can be a "singleton" (shared instance) or a "prototype"
26     (independent instance). Further scopes can be provided by extended
27     bean factories, for example in a web environment.
28
29     References among beans are supported, that is, setting a JavaBean property
30     or a constructor argument to refer to another bean in the same factory
31     (or an ancestor factory).
32
33     As alternative to bean references, "inner bean definitions" can be used.
34     Such inner beans do not have an independent lifecycle; they are typically
35     anonymous nested objects that share the scope of their containing bean.
36
37     There is also support for lists, sets, maps, and java.util.Properties
38     as bean property types or constructor argument types.
39     ]]></xsd:documentation>
40 </xsd:annotation>
41
42 <!-- base types -->
43 <xsd:complexType name="identifiedType" abstract="true">
44     <xsd:annotation>
45         <xsd:documentation><![CDATA[
46             The unique identifier for a bean. The scope of the identifier
47             is the enclosing bean factory.
48             ]]></xsd:documentation>
49         </xsd:annotation>
50         <xsd:attribute name="id" type="xsd:string">
51             <xsd:annotation>
52                 <xsd:documentation><![CDATA[
53                     The unique identifier for a bean. A bean id may not be used more than once
54                     within the same <beans> element.
55                     ]]></xsd:documentation>
56                 </xsd:annotation>
57             </xsd:attribute>
58         </xsd:complexType>
59
60     //....
61     //共1000多行，这里给出部分代码示例
62
63 </xsd:schema>
64

```

### spring-context-4.3.xsd

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <xsd:schema xmlns="http://www.springframework.org/schema/context"
4             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5             xmlns:beans="http://www.springframework.org/schema/beans"

```



```

6      xmlns:tool="http://www.springframework.org/schema/tool"
7      targetNamespace="http://www.springframework.org/schema/context"
8      elementFormDefault="qualified"
9      attributeFormDefault="unqualified">
10
11      <xsd:import namespace="http://www.springframework.org/schema/beans"
12      schemaLocation="http://www.springframework.org/schema/beans/spring-beans-
13      4.3.xsd"/>
14      <xsd:import namespace="http://www.springframework.org/schema/tool"
15      schemaLocation="http://www.springframework.org/schema/tool/spring-tool-4.3.xsd"/>
16
17      <xsd:annotation>
18          <xsd:documentation><![CDATA[
19              Defines the configuration elements for the Spring Framework's application
20              context support. Effects the activation of various configuration styles
21              for the containing Spring ApplicationContext.
22          ]]></xsd:documentation>
23      </xsd:annotation>
24
25      <xsd:complexType name="propertyLoading">
26          <xsd:attribute name="location" type="xsd:string">
27              <xsd:annotation>
28                  <xsd:documentation><![CDATA[
29                      The location of the properties file to resolve placeholders against, as a
30                      Spring
31                      resource location: a URL, a "classpath:" pseudo URL, or a relative file path.
32                      Multiple locations may be specified, separated by commas. If neither location
33                      nor
34                      properties-ref is specified, placeholders will be resolved against system
35                      properties.
36                  ]]></xsd:documentation>
37              </xsd:annotation>
38          </xsd:attribute>
39          <xsd:attribute name="properties-ref" type="xsd:string">
40              <xsd:annotation>
41                  <xsd:documentation source="java:java.util.Properties"><![CDATA[
42                      The bean name of a Properties object that will be used for property
43                      substitution.
44                      If neither location nor properties-ref is specified, placeholders will be
45                      resolved
46                      against system properties.
47                  ]]></xsd:documentation>
48              </xsd:annotation>
49          </xsd:attribute>
50          <xsd:attribute name="file-encoding" type="xsd:string">
51              <xsd:annotation>
52                  <xsd:documentation><![CDATA[
53                      Specifies the encoding to use for parsing properties files. Default is none,
54                      using the java.util.Properties default encoding. Only applies to classic
55                      properties files, not to XML files.
56                  ]]></xsd:documentation>
57              </xsd:annotation>
58          </xsd:attribute>
59          <xsd:attribute name="order" type="xsd:token">
60              <xsd:annotation>
61                  <xsd:documentation><![CDATA[
62                      Specifies the order for this placeholder configurer. If more than one is
63                      present

```

```

55     in a context, the order can be important since the first one to be match a
56     placeholder will win.
57         ]]></xsd:documentation>
58     </xsd:annotation>
59 </xsd:attribute>
60 <xsd:attribute name="ignore-resource-not-found" type="xsd:boolean"
default="false">
61     <xsd:annotation>
62         <xsd:documentation><![CDATA[
63             Specifies if failure to find the property resource location should be
ignored.
64             Default is "false", meaning that if there is no file in the location
specified
65             an exception will be raised at runtime.
66         ]]></xsd:documentation>
67     </xsd:annotation>
68 </xsd:attribute>
69 <xsd:attribute name="ignore-unresolvable" type="xsd:boolean"
default="false">
70     <xsd:annotation>
71         <xsd:documentation><![CDATA[
72             Specifies if failure to find the property value to replace a key should be
ignored.
73             Default is "false", meaning that this placeholder configurer will raise an
exception
74             if it cannot resolve a key. Set to "true" to allow the configurer to pass on
the key
75             to any others in the context that have not yet visited the key in question.
76         ]]></xsd:documentation>
77     </xsd:annotation>
78 </xsd:attribute>
79 <xsd:attribute name="local-override" type="xsd:boolean" default="false">
80     <xsd:annotation>
81         <xsd:documentation><![CDATA[
82             Specifies whether local properties override properties from files.
83             Default is "false": Properties from files override local defaults.
84         ]]></xsd:documentation>
85     </xsd:annotation>
86 </xsd:attribute>
87 </xsd:complexType>
88
89 //....
90 //共500多行，这里给出部分代码示例
91
92 </xsd:schema>

```

其中，这里的重点是XML文件中的头部声明：

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2     xmlns:context="http://www.springframework.org/schema/context"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5         http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
6         http://www.springframework.org/schema/context
7         http://www.springframework.org/schema/context/spring-context-4.3.xsd">
```

说明，

- xmlns，指定了当前XML中，标签所属的默认命名空间地址
- xmlns:context，指定了当前XML中，context作为前缀开头的标签所属的命名空间地址
- xmlns:xsi，指定了当前XML中，所使用的schema实例的命名空间地址，xsi ( XMLSchema-instance )
- xsi:schemaLocation，指定了当前XML中，所使用的schema文件都有哪些，已经分别对应的URL地址

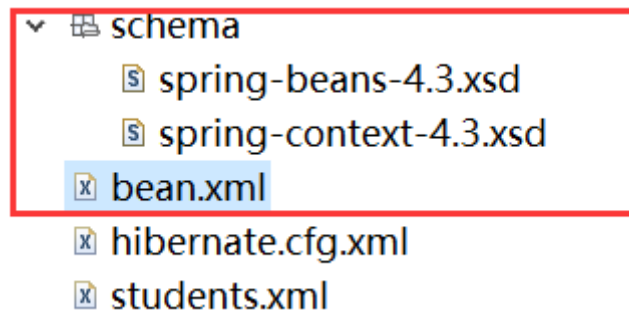
和DTD类似，这里已经有了schema文件具体的URL地址，在联网的情况下，Eclipse也会默认去下载、读取、验证的。但如果由于其他未知原因，下载失败了，我们仍然需要自己在Eclipse中进行手动配置。

例如，把XML和spring-beans-4.3.xsd进行关联

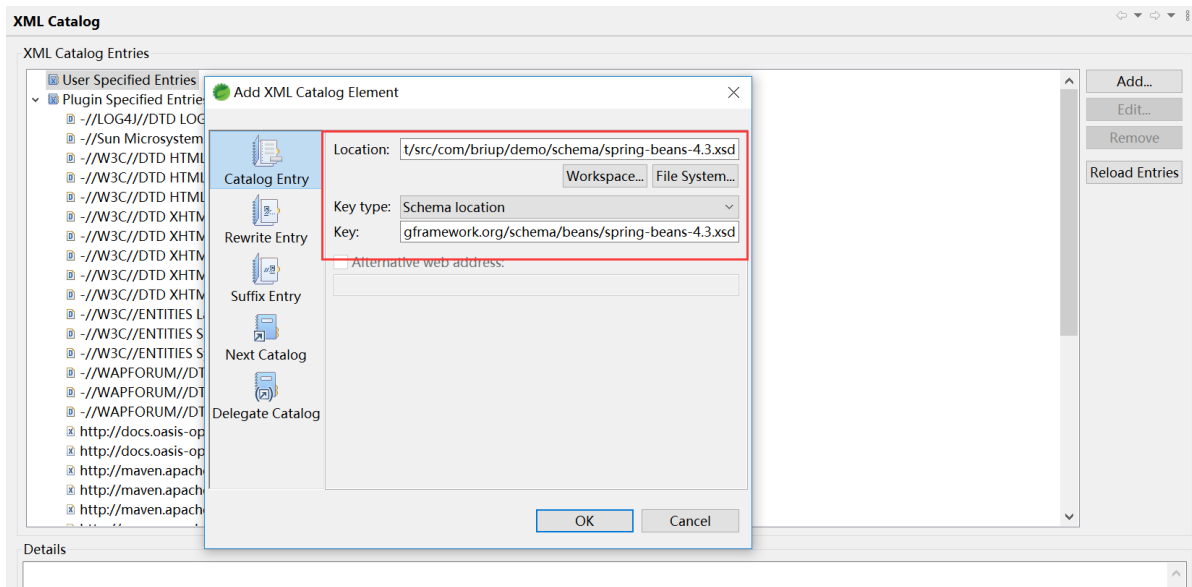
先复制该schema文件的URL地址：

```
1 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
```

把此地址作为key，在Eclipse中关联对应的value，这个value就是文件所在的本地位置：



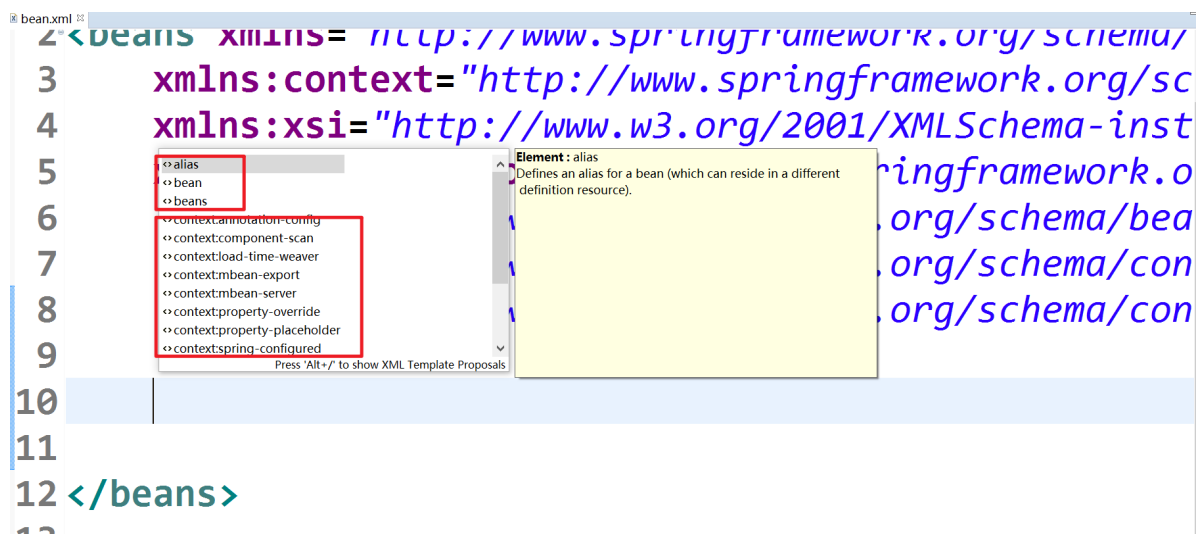
Window-》 Preferences-》 XML Catalog-》 Add：



注意，这里需要**先**填写key（提前复制好的），**然后**再选择Location具体文件位置，**最后**选择key type为Schema location。

再使用上面同样的方式，把spring-context-4.3.xsd也配置进来即可。

最后保存并关闭窗口，别忘了把XML关闭重新打开，再进行测试！



可以看出，XML中，已经可以进行自动提示了（Alt+/），并且还有context前缀的标签元素的提示！

**一定要掌握，在Eclipse中如何把XML和DTD、Schema配置关联，并且进行约束、验证以及自动提示！**

## 7 XML解析

## 7.1 解析方式

XML解析的方式一般分为两种：

- DOM解析
- SAX解析

**DOM**，（ Document Object Model ）文档对象模型，是 W3C 组织推荐的处理 XML 的一种方式。

使用DOM方式解析，要求解析器把整个XML文档装载到一个Document对象中。Document对象包含文档元素，即根元素，根元素包含N个子元素。

根据DOM的定义，XML 文档中的每个元素都是一个**节点（Node）**：

- XML文档只有一个根节点
- XML中每个元素都是一个元素节点
- XML中每个文本都是一个文本节点
- XML中每个属性都是一个属性节点
- XML中每个注释都是一个注释节点

例如， `book.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <bookstore>
3      <book category="cooking">
4          <title lang="en">Everyday Italian</title>
5          <author>Giada De Laurentiis</author>
6          <year>2005</year>
7          <price>30.00</price>
8      </book>
9      <book category="children">
10         <title lang="en">Harry Potter</title>
11         <author>J K. Rowling</author>
12         <year>2005</year>
13         <price>29.99</price>
14     </book>
15     <book category="web">
16         <title lang="en">XQuery Kick Start</title>
17         <author>James McGovern</author>
18         <year>2003</year>
19         <price>49.99</price>
20     </book>
21     <book category="web" cover="paperback">
22         <title lang="en">Learning XML</title>
23         <author>Erik T. Ray</author>
24         <year>2003</year>
25         <price>39.95</price>
26     </book>
27 </bookstore>
```

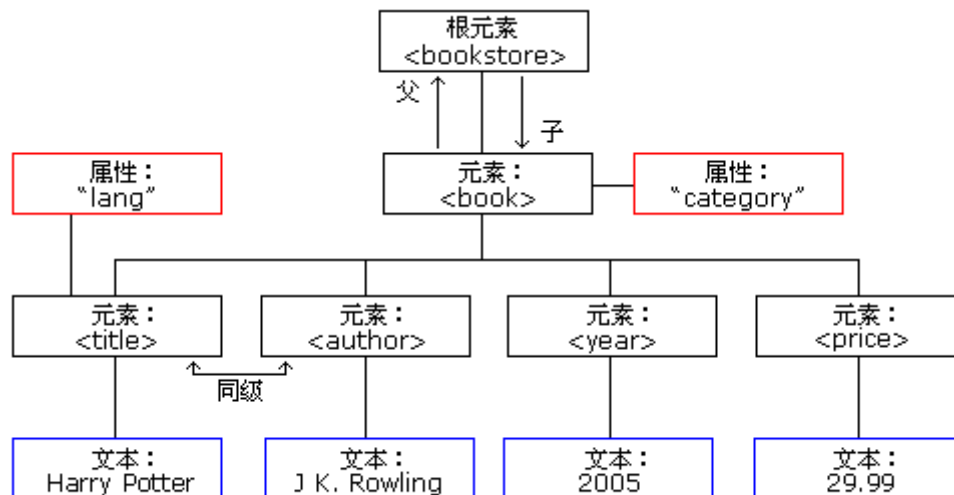
说明，

- 整个XML解析后，是一个Document对象
- 根节点是 `<bookstore>`
- 根节点下有四个子节点， `<book>` 、 `<book>` 、 `<book>` 、 `<book>`

- 每个 `<book>` 节点中，又包含四个子节点，`<title>`、`<author>`、`<year>`、`<price>`
- 这四个子节点中，每个都有文本节点
- `<book>` 节点中和 `<title>` 节点中，还有属性节点

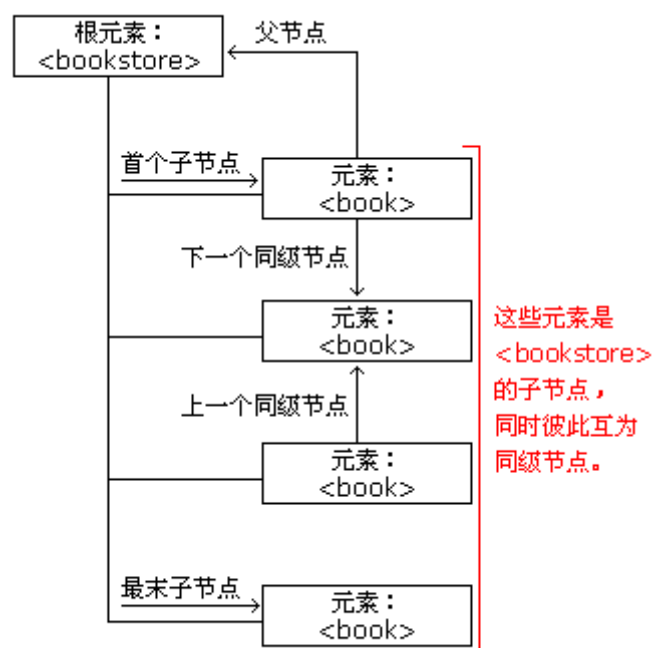
DOM 把 XML 文档解析为一种树结构。这种树结构被称为**节点树**，通过这棵树可以访问任何一个节点，也可以修改或删除它们的内容，或者创建新的元素。

例如，



节点树中的每个节点之间都有层级关系，父节点、子节点、同级节点这些都是用来描述节点之间的关系

节点树中，部分节点之间的关系如下：



一个XML文档使用DOM方式解析后，会得到一个Document对象，由于元素与元素之间还保存着结构关系，所以解析期间可以方便的针对元素对象进行各种操作。

- 使用DOM解析方式，XML文档的结构关系，在内存的对象（document）中依然存在
- 但如果XML文档过大，那么需要把整个XML文档装载进内存，这时候会对内存空间要求比较高

**SAX**，（Simple API for XML）它不是W3C标准，但它是XML社区事实上的标准，因为使用率也比较高，几乎所有的XML解析器都支持它。

使用SAX方式解析，每当读取一个开始标签、结束标签或者文本内容的时候，都会调用我们重写的一个指定方法，该方法中编写当前需要完成的解析操作。直到XML文档读取结束，在整个过程中，SAX解析方法不会在内存中保存节点的信息和关系。

- 使用SAX解析方式，不会占用大量内存来保存XML文档数据和关系，效率高。
- 但是在解析过程中，不会保存节点信息和关系，并且只能从前往后，顺序读取、解析。

注意，DOM和SAX都是解析XML文档的一种模型/标准/理论，是需要编写具体的代码去实现的，而实现了这些解析方式的代码，被称之为XML解析器。

## 7.2 解析器

基本上现在主流的解析器，都提供了对DOM和SAX这两种解析方式的支持。

不同的公司、组织、团队都可以推出自己的编写代码实现的解析器，其中就包含了对这两种解析方式的支持

- Crimson解析器，早期SUN公司推出，JDK1.4之前使用的解析器，性能较差
- Xerces解析器，IBM公司推出的基于DOM和SAX的解析器，现在已经由Apache基金会维护，是当前流行的解析器之一，在JDK1.5之后，已经添加到JDK之中
- Aelfred2解析器，DOM4J团队推出的解析器，也是DOM4J工具中默认的解析器

可以看出，其实我们并不需要编写解析器，我们可以直接调用已经实现的解析器中提供的API，就可以完成对XML文档的解析，并且一般都会支持DOM和SAX两种解析方式。

## 7.3 JAXP

JAXP（Java API for XML Processing），是JavaSE-API的一部分，它由javax.xml、org.w3c.dom、org.xml.sax包及其子包组成

也就是说，不借助于第三方jar包，使用JDK自带的API，就可以完成对XML文档的解析。

### 7.3.1 DOM

JAXP中的DOM解析的步骤：

- 调用 `DocumentBuilderFactory.newInstance()` 方法得到创建DOM解析器的工厂
- 调用工厂对象的 `newDocumentBuilder` 方法得到DOM解析器对象
- 调用DOM解析器对象的 `parse()` 方法解析XML文档

- `parse()` 方法会返回解析得到的 Document 对象
- 然后根据文档的结构关系和特点，调用相应的API，对其进行解析

例如，需要解析的class.xml文件

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <students>
3      <stu id="2022001">
4          <name>王五</name>
5          <age>23</age>
6      </stu>
7      <stu id="2022002">
8          <name>李四</name>
9          <age>20</age>
10     </stu>
11 </students>
12
```

例如，使用JAXP中的DOM解析

```
1  package com.briup.demo;
2
3  import org.w3c.dom.*;
4
5  import javax.xml.parsers.DocumentBuilder;
6  import javax.xml.parsers.DocumentBuilderFactory;
7
8  public class Test {
9      public static void main(String[] args) {
10
11          Test test = new Test();
12          String filePath = "src/com/briup/demo/class.xml";
13          test.domParse(filePath);
14
15      }
16
17      public void domParse(String filePath) {
18
19          //创建DocumentBuilderFactory类的对象
20          DocumentBuilderFactory builderFactory =
21              DocumentBuilderFactory.newInstance();
22
23          try {
24              //通过工厂对象创建出DocumentBuilder对象
25              DocumentBuilder documentBuilder =
26                  builderFactory.newDocumentBuilder();
27
28              //通过DocumentBuilder对象，解析xml文件，并且获取document
29              //document就是dom中的文档对象，用来表示被解析的xml文件
30              Document document = documentBuilder.parse(filePath);
31
32              //获取根元素
33              Element root = document.getDocumentElement();
34
35          }
```



```

36         //获取根元素下面所有的子节点(儿子节点)
37         NodeList childNodes = root.getChildNodes();
38
39         for(int i=0;i<childNodes.getLength();i++) {
40             //获取当前处理的子节点
41             Node node = childNodes.item(i);
42
43             //判断节点的类型，是文本节点还是元素节点
44             switch (node.getNodeType()) {
45                 case Node.TEXT_NODE:
46                     System.out.println("文本节点内容: "+node.getTextContent());
47                     break;
48                 case Node.ELEMENT_NODE:
49                     System.out.println("元素节点名字: "+node.getNodeName());
50                     //获取当前元素节点中的所有属性
51                     NamedNodeMap attributes = node.getAttributes();
52                     //循环变量拿到每一个属性
53                     for(int j=0;j<attributes.getLength();j++) {
54                         //强制转换为Attr类型的对象，表示属性
55                         Attr attr = (Attr)attributes.item(j);
56                         //可以通过方法获取到属性名和属性值
57                         System.out.println(attr.getName()+" =
"+attr.getValue());
58                     }
59
60                     //获取当前节点下面的子节点
61                     NodeList nodeList = node.getChildNodes();
62                     for(int k=0;k<nodeList.getLength();k++) {
63                         Node item = nodeList.item(k);
64                         //如果这个节点是元素节点的话
65                         if(item.getNodeType()==Node.ELEMENT_NODE) {
66                             //拿出元素节点的名字和它的文本值
67                             System.out.println(item.getNodeName()+" =
"+item.getTextContent());
68                         }
69                     }
70
71                     break;
72                 default:
73                     throw new RuntimeException("解析到意外的节点类型:"+node);
74             }
75
76         }
77
78     } catch (Exception e) {
79         e.printStackTrace();
80     }
81
82 }
83
84 }

```

如果XML中带有DTD的引入：

```
class.xml
```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE students SYSTEM "class.dtd">
3 <students>
4     <stu id="2022001">
5         <name>王五</name>
6         <age>23</age>
7     </stu>
8     <stu id="2022002">
9         <name>李四</name>
10        <age>20</age>
11    </stu>
12 </students>
13

```

class.dtd

```

1 <!ELEMENT students (stu*)>
2 <!ELEMENT stu ANY>
3 <!ATTLIST stu
4     id      CDATA #REQUIRED
5     sname   CDATA "TOM"
6 >
7

```

解析的代码不变，再次运行该代码，在输出的结果中，可以看到sname属性的值：

```

1 id = 2019001
2 sname = TOM
3 name = 王五
4 age = 23

```

因为XML文件中，虽然没有添加sname属性，但是DTD文件中，指定了sname属性的默认值为TOM，所以解析的时候可以直接读到这个默认的属性值

当然，我们也可以在程序中，跳过DTD文件的读取，只需要在程序中添加如下代码：

```

1 documentBuilder.setEntityResolver((publicId, systemId) -> {
2     String str = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";
3     InputStream in = new ByteArrayInputStream(str.getBytes());
4     return new InputSource(in);
5     //本来是需要读取dtd文件所在路径的，例如下面代码
6     //但是我想跳过dtd的解析，所以就重写为上面代码
7     //return new InputSource(systemId);
8 });

```

例如，

```

1 package com.briup.demo;
2
3 import org.w3c.dom.*;
4 import org.xml.sax.InputSource;
5
6 import javax.xml.parsers.DocumentBuilder;

```

```

7  import javax.xml.parsers.DocumentBuilderFactory;
8  import java.io.ByteArrayInputStream;
9  import java.io.InputStream;
10
11  public class Test {
12      public static void main(String[] args) {
13
14          Test dom = new Test();
15          String filePath = "src/com/briup/demo/class.xml";
16          dom.parse(filePath);
17
18      }
19
20
21      public void parse(String filePath) {
22
23          //创建DocumentBuilderFactory类的对象
24          DocumentBuilderFactory builderFactory =
25              DocumentBuilderFactory.newInstance();
26
27          try {
28              //通过工厂对象创建出DocumentBuilder对象
29              DocumentBuilder documentBuilder =
30                  builderFactory.newDocumentBuilder();
31
32              /*====其他地方不变,添加此处代码====*/
33              documentBuilder.setEntityResolver((publicId, systemId) -> {
34                  String str = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";
35                  InputStream in = new ByteArrayInputStream(str.getBytes());
36                  return new InputSource(in);
37                  //本来是需要读取dtd文件所在路径的,例如下面代码
38                  //但是我想跳过dtd的解析,所有就重写为上面代码
39                  //return new InputSource(systemId);
40              });
41              /*====其他地方不变,添加此处代码====*/
42
43              //通过DocumentBuilder对象,解析xml文件,并且获取document
44              //document就是dom中的文档对象,用来表示被解析的xml文件
45              Document document = documentBuilder.parse(filePath);
46
47              //获取根元素
48              Element root = document.getDocumentElement();
49
50              //获取根元素下面所有的子节点(儿子节点)
51              NodeList childNodes = root.getChildNodes();
52
53              for(int i=0;i<childNodes.getLength();i++) {
54                  //获取当前处理的子节点
55                  Node node = childNodes.item(i);
56
57                  //判断节点的类型,是文本节点还是元素节点
58                  switch (node.getNodeType()) {
59                      case Node.TEXT_NODE:
60                          System.out.println("文本节点内容: "+node.getTextContent());
61                          break;
62                      case Node.ELEMENT_NODE:
63                          System.out.println("元素节点名字: "+node.getNodeName());
64                          //获取当前元素节点中的所有属性

```

```

65         NamedNodeMap attributes = node.getAttributes();
66         //循环变量拿到每一个属性
67         for(int j=0;j<attributes.getLength();j++) {
68             //强制转换为Attr类型的对象，表示属性
69             Attr attr = (Attr)attributes.item(j);
70             //可以通过方法获取到属性名和属性值
71             System.out.println(attr.getName()+" =
"+attr.getValue());
72         }
73
74         //获取当前节点下面的子节点
75         NodeList nodeList = node.getChildNodes();
76         for(int k=0;k<nodeList.getLength();k++) {
77             Node item = nodeList.item(k);
78             //如果这个节点是元素节点的话
79             if(item.getNodeType()==Node.ELEMENT_NODE) {
80                 //拿出元素节点的名字和它的文本值
81                 System.out.println(item.getNodeName()+" =
"+item.getTextContent());
82             }
83         }
84
85         break;
86     default:
87         throw new RuntimeException("解析到意外的节点类型:"+node);
88     }
89 }
90 }
91
92 } catch (Exception e) {
93     e.printStackTrace();
94 }
95
96 }
97
98 }
99

```

此时运行代码，就不会解析出sname属性的值，因为我们在代码中跳过了DTD的读取

除了解析XML文档内容之外，还可以在内存中先创建出document对象，以及文档中节点之间的关系，然后将document对象写出为一个指定路径的XML文档：

例如，

```

1     package com.briup.demo;
2
3     import org.w3c.dom.Document;
4     import org.w3c.dom.Element;
5     import org.w3c.dom.Comment;
6
7     import javax.xml.parsers.DocumentBuilder;
8     import javax.xml.parsers.DocumentBuilderFactory;
9     import javax.xml.parsers.ParserConfigurationException;

```

```
10 import javax.xml.transform.OutputKeys;
11 import javax.xml.transform.Transformer;
12 import javax.xml.transform.TransformerFactory;
13 import javax.xml.transform.dom.DOMSource;
14 import javax.xml.transform.stream.StreamResult;
15
16 public class Test {
17     public static void main(String[] args) {
18
19         Test dom = new Test();
20         String filePath = "src/com/briup/demo/my.xml";
21
22         dom.createDocument(filePath);
23
24     }
25
26     //创建Document文档对象，并写出去成为xml
27     public void createDocument(String filePath) {
28
29         //创建DocumentBuilderFactory类的对象
30         DocumentBuilderFactory builderFactory =
31         DocumentBuilderFactory.newInstance();
32
33         try {
34             //通过工厂对象创建出DocumentBuilder对象
35             DocumentBuilder documentBuilder =
36             builderFactory.newDocumentBuilder();
37
38             //通过DocumentBuilder对象，创建出一个新的document对象
39             Document document = documentBuilder.newDocument();
40
41             //创建一个根节点
42             Element root =
43             document.createElement("root");
44
45             //创建一个元素节点student
46             Element studentNode =
47             document.createElement("student");
48
49             //设置这个元素节点的属性
50             studentNode.setAttribute("id", "2022001");
51
52             //创建一个元素节点name
53             Element nameNode =
54             document.createElement("name");
55
56             //创建一个注释节点comment
57             Comment comment = document.createComment("这里是注释信息");
58
59             //创建name节点中的文本内容
60             nameNode.setTextContent("tom");
61
62             //将根节点添加到document中
63             document.appendChild(root);
64
65             //将注释节点添加到根节点中
66             root.appendChild(comment);
67         } catch (Exception e) {
68             e.printStackTrace();
69         }
70     }
71 }
```

```

66         //将student节点添加到根节点中
67         root.appendChild(studentNode);
68
69         //将name节点添加到student节点中
70         studentNode.appendChild(nameNode);
71
72         //调用自定义的私有方法，将document对象写出为指定路径的xml文件
73         createXMLByDocument(document, filePath);
74
75     } catch (ParserConfigurationException e) {
76         e.printStackTrace();
77     }
78
79 }
80
81 /**
82  * 通过document对象创建出对应的xml文件
83  * @param document 内存中构建好的document对象
84  * @param filePath 生成xml文件的位置
85  */
86 private void createXMLByDocument(Document document, String filePath) {
87
88     //获取TransformerFactory工厂类对象
89     TransformerFactory transformerFactory =
90         TransformerFactory.newInstance();
91
92     try {
93         //通过工厂类对象，获得转换器对象
94         //负责把document对象转换为xml文件
95         Transformer transformer =
96             transformerFactory.newTransformer();
97
98
99         //设置xml中的编码
100        transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
101        //设置xml中的内容换行，否则全部写到一行中
102        transformer.setOutputProperty(OutputKeys.INDENT, "yes");
103
104        //把document对象转为xml文件
105
106        transformer.transform(new DOMSource(document), new
StreamResult(filePath));
107
108
109    } catch (Exception e) {
110        e.printStackTrace();
111    }
112
113 }
114
115 }

```

运行后，打开生成的my.xml文档：

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <root>
3 <!--这里是注释信息-->
4 <student id="2022001">
5 <name>tom</name>
6 </student>
7 </root>
8
```

### 7.3.2 SAX

SAX解析允许在读取文档的时候，立即对文档进行处理，而不必等到整个文档加载完才对文档进行操作

解析器在使用SAX方式解析XML文档的时候，如果发现了XML文档中的内容（开始标签、结束标签、文本值等），就会去调用我们重写之后的方法，也就是说这时候该如何处理读到的内容，由我们自己决定。

在基于SAX解析的程序中，有五个最常用SAX事件：

1. `startDocument()`，解析器发现了文档的开始标签，会自动调用该方法
2. `endDocument()`，解析器发现了文档结束标签，会自动调用该方法
3. `startElement()`，解析器发现了一个起始标签，会自动调用该方法
4. `character()`，解析器发现了标签里面的文本值，会自动调用该方法
5. `endElement()`，解析器发现了一个结束标签，会自动调用该方法

每当解析中，触发了某个事件，那么就会自动调用我们重写的指定方法。

JAXP中的SAX解析的步骤：

- 获取SAXParserFactory工厂类对象，`SAXParserFactory.newInstance()`
- 使用工厂对象，创建出SAX解析器，`saxParserFactory.newSAXParser()`
- 调用解析器的 `parse()` 解析xml文件，然后重写DefaultHandler类中的方法，进行事件处理  
`saxParser.parse(filePath, new DefaultHandler(){...})`

例如，需要解析的class.xml文件

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <students>
3      <stu id="2022001">
4          <name>王五</name>
5          <age>23</age>
6      </stu>
7      <stu id="2022002">
8          <name>李四</name>
9          <age>20</age>
10     </stu>
11 </students>
12

```

例如，使用JAXP中的SAX解析

```

1  package com.briup.demo;
2
3  import org.xml.sax.Attributes;
4  import org.xml.sax.SAXException;
5  import org.xml.sax.helpers.DefaultHandler;
6
7  import javax.xml.parsers.SAXParser;
8  import javax.xml.parsers.SAXParserFactory;
9
10 public class Test {
11     public static void main(String[] args) {
12
13         Test dom = new Test();
14         String filePath = "src/com/briup/demo/class.xml";
15
16         dom.saxParse(filePath);
17     }
18
19     public void saxParse(String filePath) {
20
21         //获取SAXParserFactory工厂类对象
22         SAXParserFactory saxParserFactory =
23             SAXParserFactory.newInstance();
24
25         try {
26             //使用工厂对象，创建出SAX解析器
27             SAXParser saxParser =
28                 saxParserFactory.newSAXParser();
29
30             //解析xml文件，重写DefaultHandler类中的方法，进行事件处理
31             saxParser.parse(filePath, new DefaultHandler() {
32
33                 //当解析器解析到xml文档开头的时候，调用该方法
34                 @Override
35                 public void startDocument() throws SAXException {
36                     System.out.println("开始解析xml文档");
37                 }
38
39                 //当解析器解析到xml文档结束的时候，调用该方法
40                 @Override
41

```



```

42         public void endDocument() throws SAXException {
43             System.out.println();
44             System.out.println("结束解析xml文件");
45         }
46
47         /**
48          * 当解析器解析到一个开始标签的时候，调用该方法
49          * @param uri 命名空间
50          * @param localName 本地名称（不带前缀）
51          * @param qName 限定的名称（带有前缀）
52          * @param attributes 该节点上的属性列表
53          * @throws SAXException
54          */
55         @Override
56         public void startElement(String uri, String localName, String
qName, Attributes attributes)
57             throws SAXException {
58
59             System.out.println("节点: "+qName+"解析开始");
60             if(attributes.getLength()>0) {
61                 System.out.print(" 属性列表: ");
62                 for(int i=0;i<attributes.getLength();i++) {
63                     String attrName = attributes.getQName(i);
64                     String attrValue = attributes.getValue(i);
65                     System.out.print(attrName+"="+attrValue+" ");
66                 }
67                 System.out.println();
68             }
69
70         }
71
72
73         /**
74          * 当解析器解析到一个结束标签的时候，调用该方法
75          * @param uri 命名空间
76          * @param localName 本地名称（不带前缀）
77          * @param qName 限定的名称（带有前缀）
78          * @throws SAXException
79          */
80         @Override
81         public void endElement(String uri, String localName, String
qName) throws SAXException {
82             System.out.println("节点: "+qName+"解析结束");
83         }
84
85         //当解析器解析到文件值的时候，调用该方法
86         @Override
87         public void characters(char[] ch, int start, int length) throws
SAXException {
88             String str = new String(ch,start,length);
89             if(!"".equals(str.trim())) {
90                 System.out.println("文本节点: "+str);
91             }
92         }
93
94     });
95
96     } catch (Exception e) {

```

```

97         e.printStackTrace();
98     }
99
100 }
101
102
103 }

```

**案例，基于JAXP的SAX解析方式，完成对class.xml文档的解析并输出打印**

```

1  package com.briup.demo;
2
3  import org.xml.sax.Attributes;
4  import org.xml.sax.SAXException;
5  import org.xml.sax.helpers.DefaultHandler;
6
7  import javax.xml.parsers.SAXParser;
8  import javax.xml.parsers.SAXParserFactory;
9
10 public class Test {
11     public static void main(String[] args) {
12
13         Test dom = new Test();
14         String filePath = "src/com/briup/demo/class.xml";
15
16         dom.saxParse(filePath);
17
18     }
19
20     public void saxParse(String filePath) {
21
22         //获取SAXParserFactory工厂类对象
23         SAXParserFactory saxParserFactory =
24             SAXParserFactory.newInstance();
25
26
27         try {
28             //使用工厂对象，创建出SAX解析器
29             SAXParser saxParser =
30                 saxParserFactory.newSAXParser();
31
32             //解析xml文件，重写DefaultHandler类中的方法，进行事件处理
33             saxParser.parse(filePath, new DefaultHandler() {
34
35                 @Override
36                 public void startDocument() throws SAXException {
37                     System.out.println("<?xml version='1.0' encoding='utf-8'>");
38                 }
39
40                 @Override
41                 public void startElement(String uri, String localName, String
qName,
42                                         Attributes atts) throws SAXException {
43                     StringBuilder sb = new StringBuilder();

```

```

44         sb.append("<").append(qName);
45         for(int i = 0; i < atts.getLength(); i++) {
46             sb.append(" ");
47             sb.append(atts.getQName(i));
48             sb.append("=");
49             sb.append("'");
50             sb.append(atts.getValue(i));
51             sb.append("'");
52         }
53         sb.append(">");
54         System.out.print(sb.toString());
55     }
56
57     @Override
58     public void endElement(String uri, String localName, String
qName)
59         throws SAXException {
60         System.out.print("</" + qName + ">");
61     }
62
63     @Override
64     public void characters(char[] ch, int start, int length)
65         throws SAXException {
66         System.out.print(new String(ch, start, length));
67     }
68
69     });
70
71     } catch (Exception e) {
72         e.printStackTrace();
73     }
74
75 }
76
77
78 }

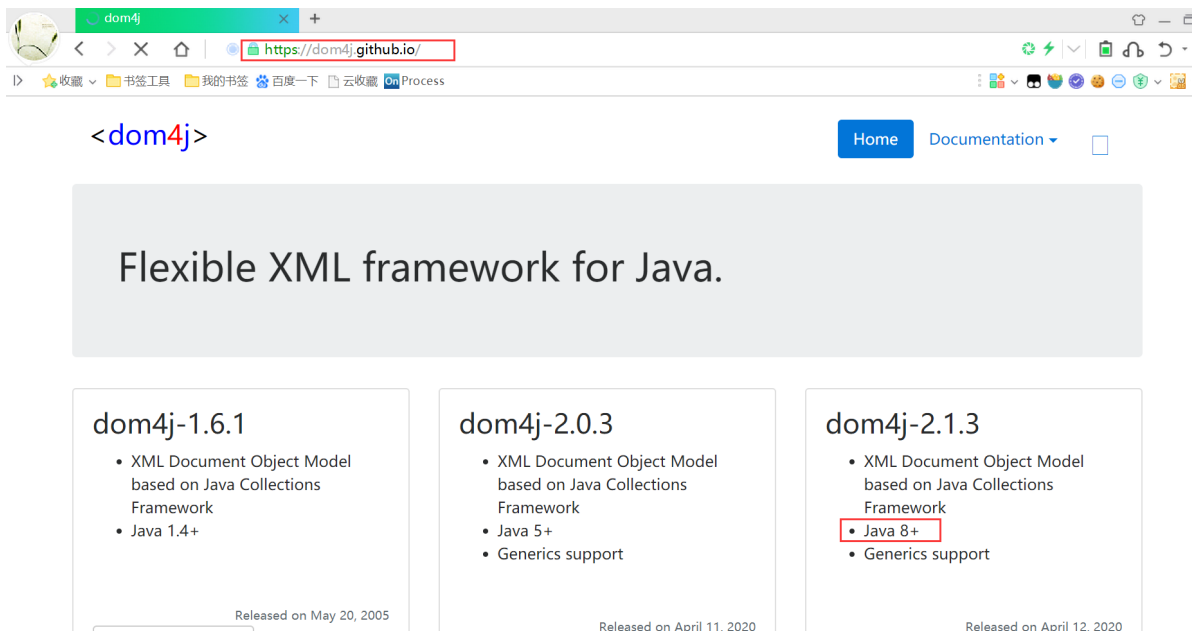
```

## 7.4 DOM4J

### 7.4.1 概述

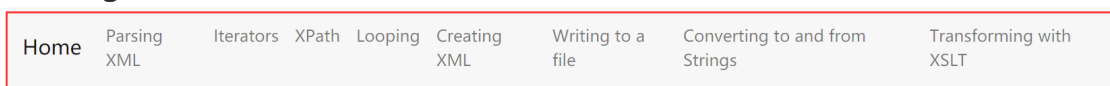
Dom4j 是一个Java的XML API，用来读写XML文件的，它是一个十分优秀的Java XML API，具有性能优异、功能强大和极其易使用的特点，且开源免费。

[DOM4J官网地址](#) [下载地址](#)



官网提供的API使用说明和实例：

## Getting started



### Parsing XML

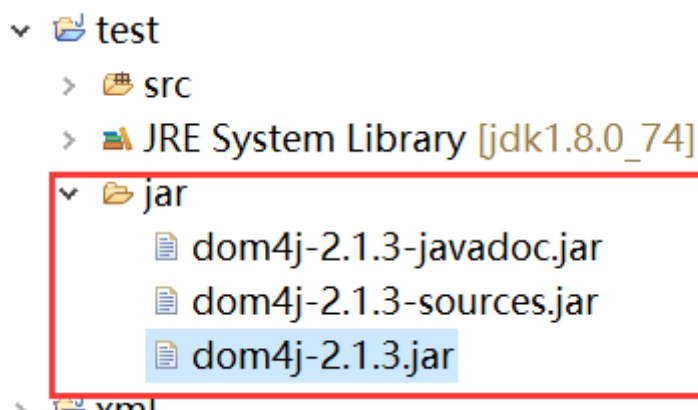
One of the first things you'll probably want to do is to parse an XML document of some kind. This is easy to do in `<dom4j>`. The following code demonstrates how to do this.

```
1 import java.net.URL;
2
3 import org.dom4j.Document;
4 import org.dom4j.DocumentException;
5 import org.dom4j.io.SAXReader;
6
7 public class Foo {
8
9     public Document parse(URL url) throws DocumentException {
10         SAXReader reader = new SAXReader();
11         Document document = reader.read(url);
12         return document;
13     }
14 }
```

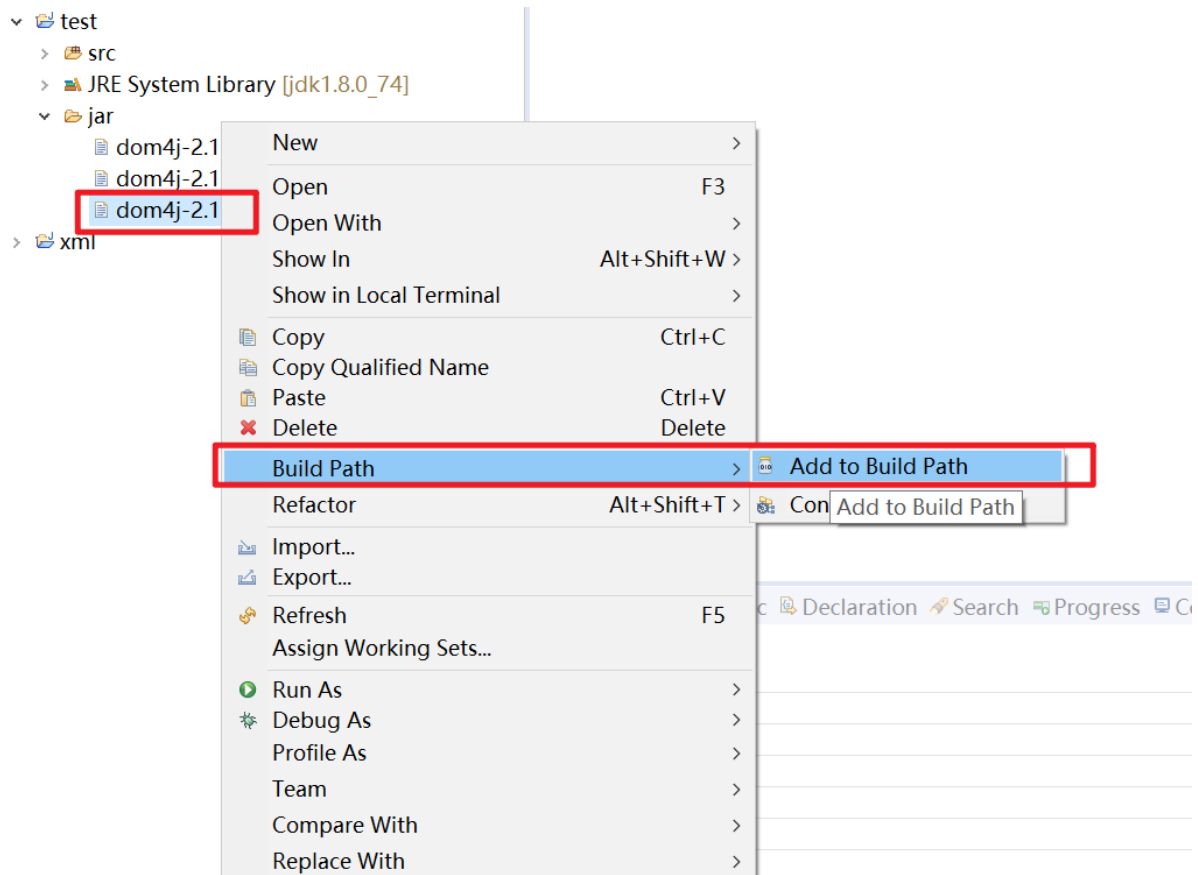
## 7.4.2 引入

如果是普通的java项目：

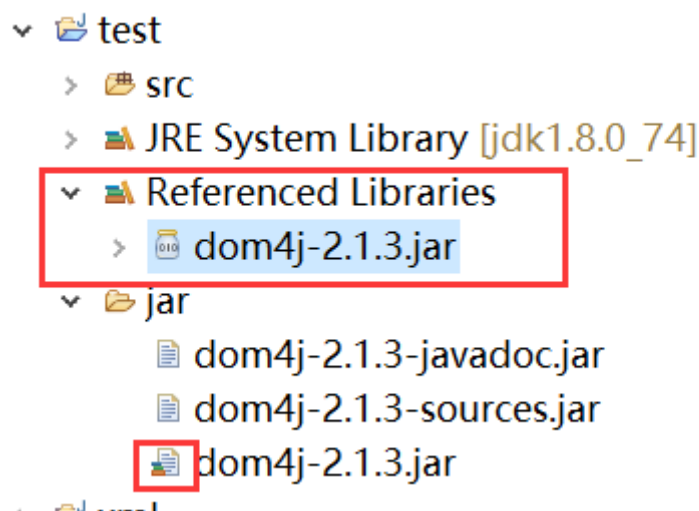
项目中新建文件夹，然后把DOM4J的jar放入该文件夹



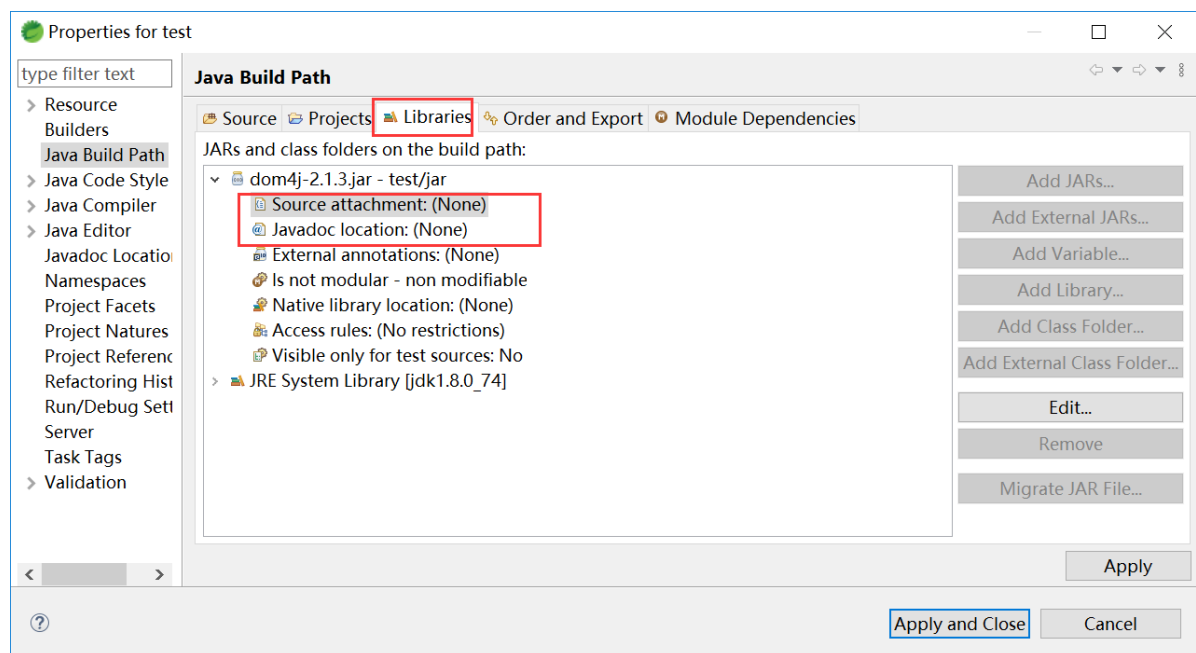
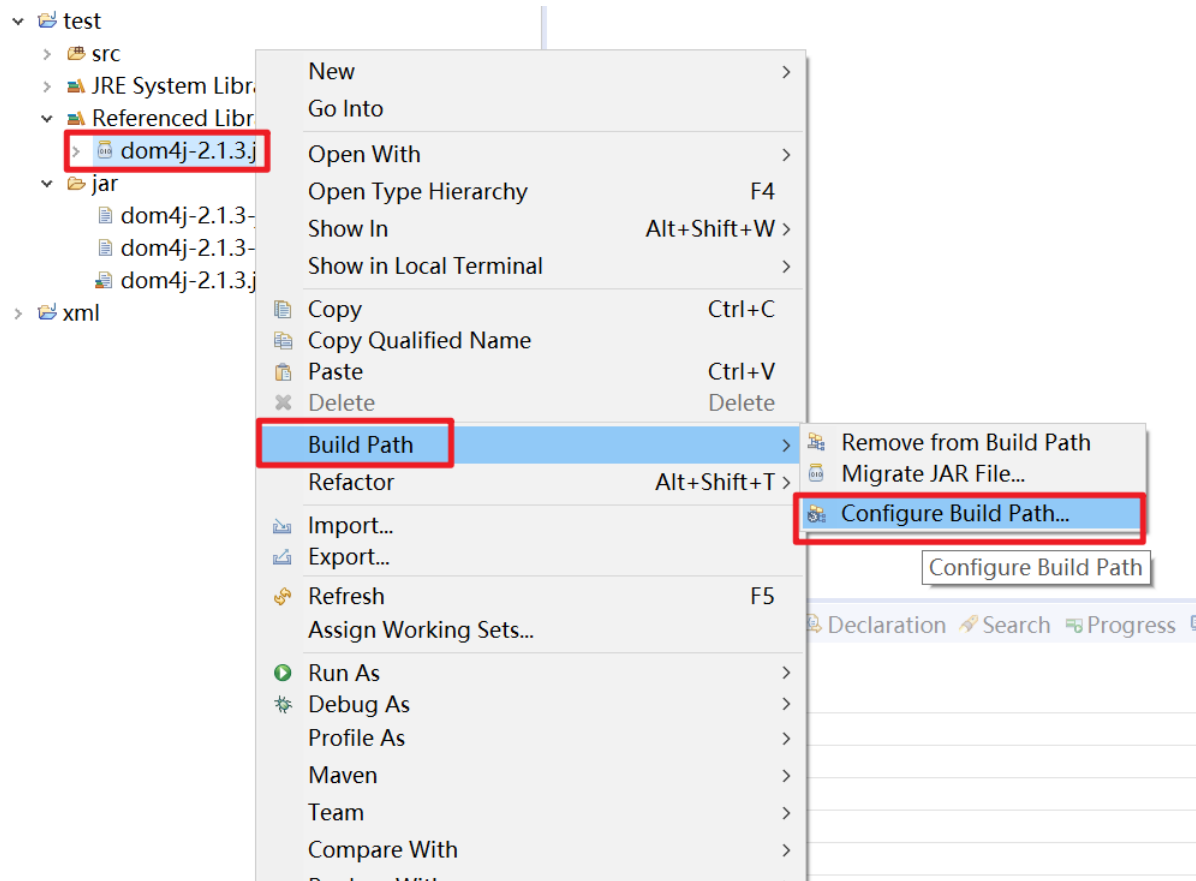
然后把DOM4J的jar包添加到项目的classpath中：

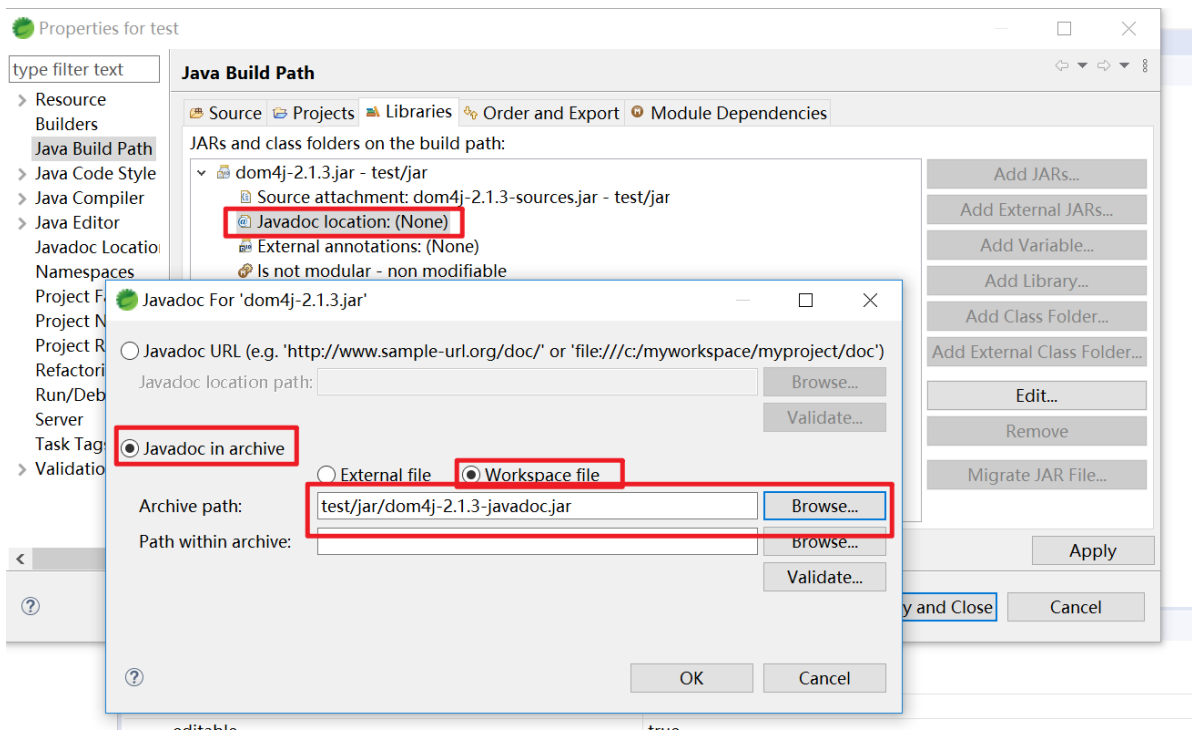
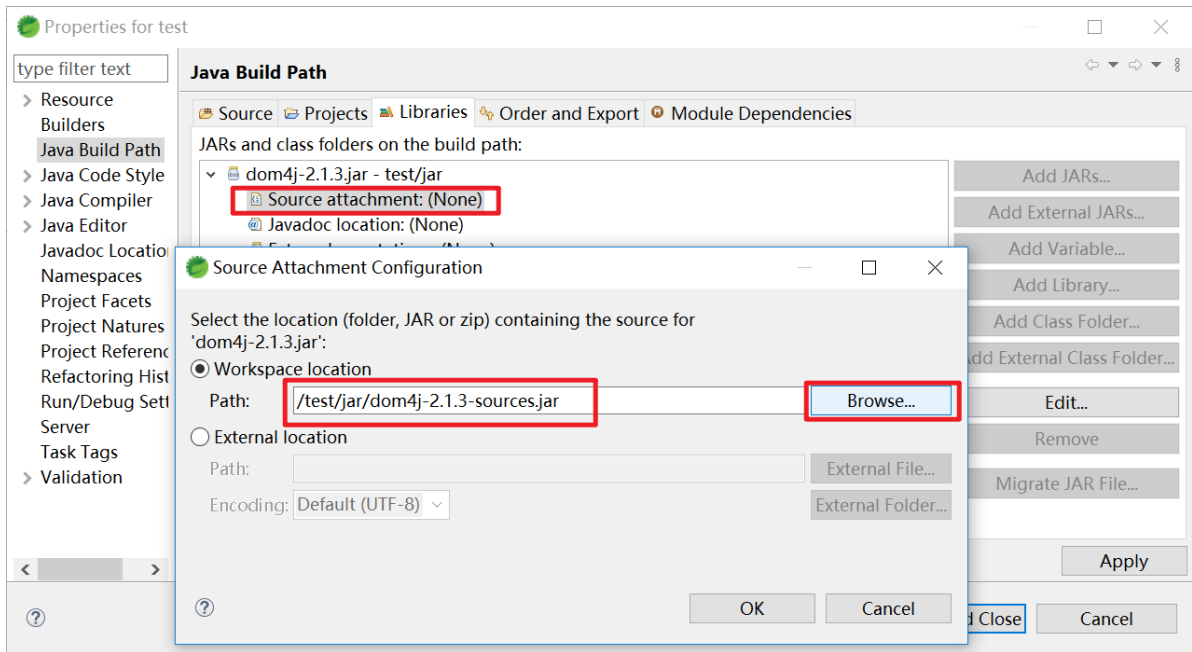


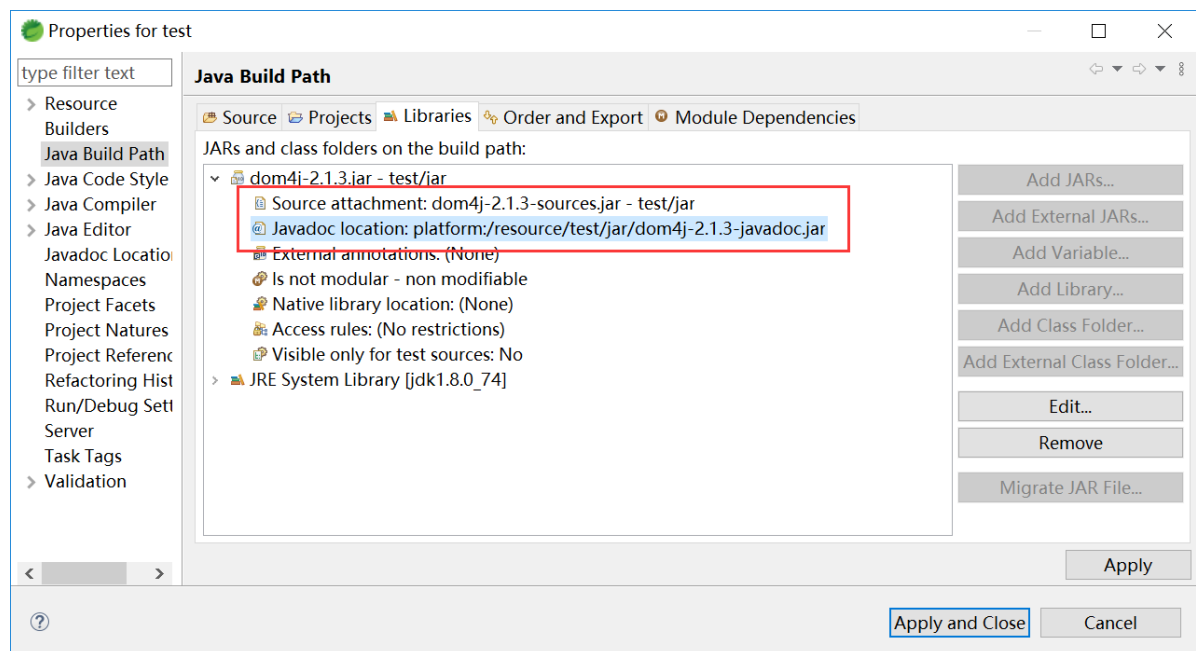
添加完成后的效果如下：



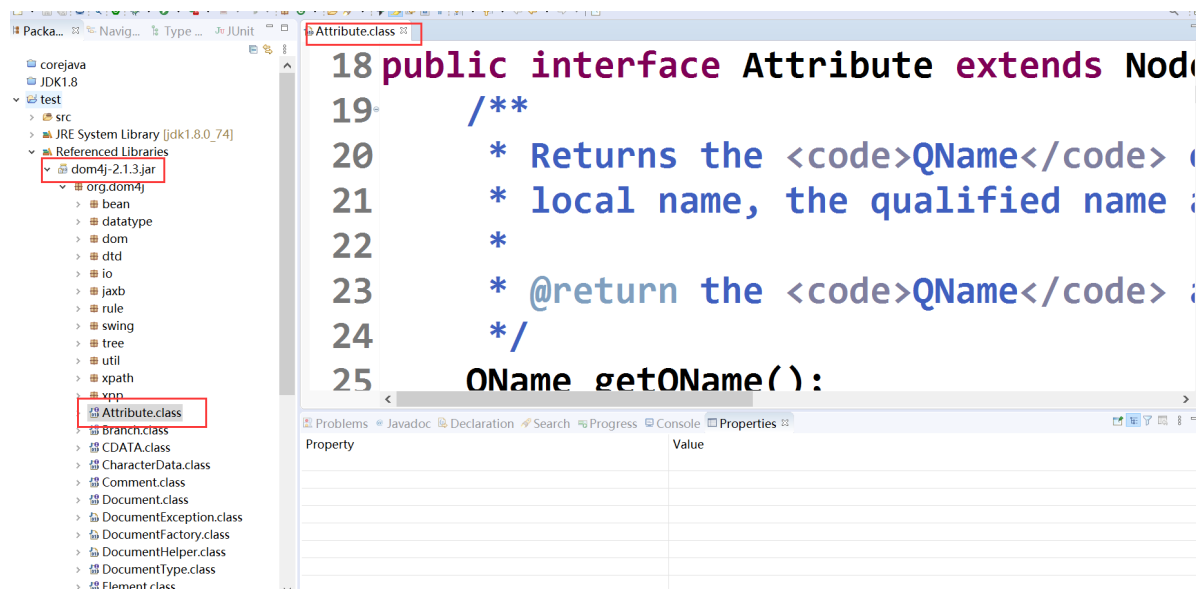
然后再配置这个jar和sources、javadoc相关联：



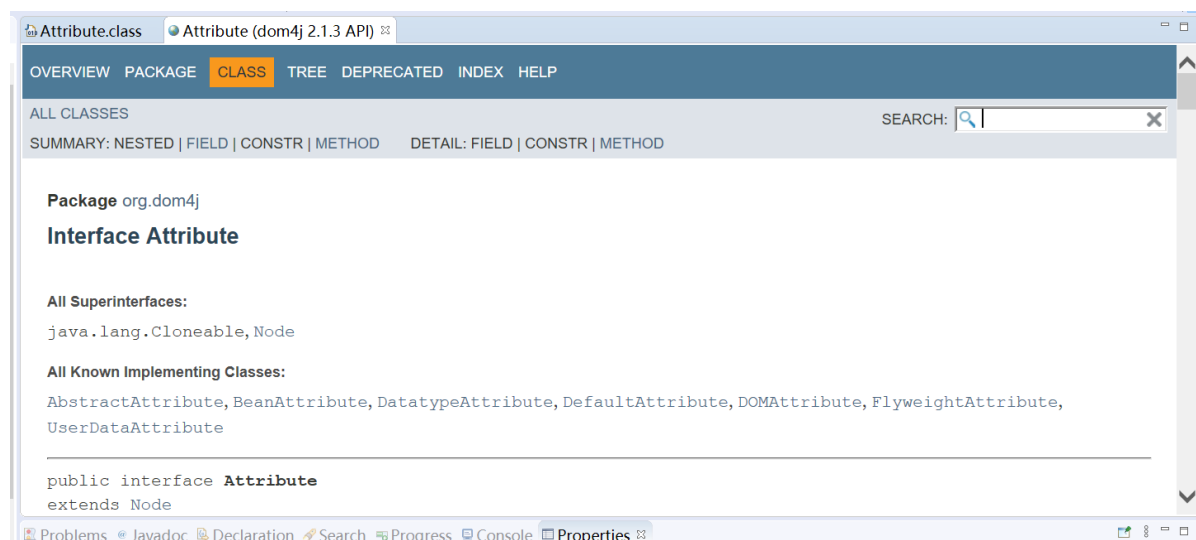




配置完成后，双击DOM4J中的class文件，自动打开关联的源代码：



双击选择这个类型，按快捷键shift+f2，打开此类型在javadoc中的说明：





如果是maven项目：

引入在pom.xml文件中，添加DOM4J在maven仓库中的坐标即可

```
1 <dependency>
2     <groupId>org.dom4j</groupId>
3     <artifactId>dom4j</artifactId>
4     <version>2.1.3</version>
5 </dependency>
```

我们之后在了解maven相关的内容

### 7.4.3 使用

class.xml，需要解析的XML文件如下

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <students sid="1" sname="briup">
3     <stu id="2022001">
4         <name>王五</name>
5         <age>23</age>
6         <score course="数学">90</score>
7     </stu>
8     <stu id="2022002">
9         <name>李四</name>
10        <age>21</age>
11    </stu>
12    <teacher></teacher>
13 </students>
14
```

获取 org.dom4j.Document 文档对象

```
1 public class Test {
2     public static void main(String[] args) {
3
4         Test test = new Test();
5         String filePath = "src/com/briup/demo/class.xml";
6
7         try {
8             Document document = test.parse(new File(filePath));
9
10        } catch (Exception e) {
11            e.printStackTrace();
12        }
13
14    }
15
16    public Document parse(File file) throws DocumentException {
17        SAXReader reader = new SAXReader();
18        Document document = reader.read(file);
```

```
19         return document;
20     }
21 }
```

获取一个元素的所有属性、所有子节点、指定名字的子节点：

```
1  public class Test {
2      public static void main(String[] args) {
3
4          Test test = new Test();
5          String filePath = "src/com/briup/demo/class.xml";
6
7          try {
8              Document document = test.parse(new File(filePath));
9
10             test.handler1(document);
11
12         } catch (Exception e) {
13             e.printStackTrace();
14         }
15     }
16
17     public Document parse(File file) throws DocumentException {
18         SAXReader reader = new SAXReader();
19         Document document = reader.read(file);
20         return document;
21     }
22
23     public void handler1(Document document) {
24
25         Element root = document.getRootElement();
26
27         System.out.println("-----获取根节点下面所有的子节点（儿子节点）-----");
28
29         // 获取根节点下面所有的子节点（儿子节点）
30         for (Iterator<Element> it = root.elementIterator(); it.hasNext();) {
31             Element element = it.next();
32             System.out.println(element.getName());
33         }
34
35         System.out.println("-----获取根节点下面指定名称的子节点（儿子节点）-----");
36
37         // 获取根节点下面指定名称的子节点（儿子节点）
38         for (Iterator<Element> it = root.elementIterator("stu"); it.hasNext();) {
39             Element element = it.next();
40             System.out.println(element.getName());
41         }
42
43         System.out.println("-----获取根节点中的所有属性-----");
44
45         // 获取根节点中的所有属性
46         for (Iterator<Attribute> it = root.attributeIterator(); it.hasNext();) {
47             Attribute attribute = it.next();
48         }
```

```

49         System.out.println(attribute.getName()+"="+attribute.getValue());
50     }
51
52     }
53 }
54
55 //运行结果:
56 -----获取根节点下面所有的子节点（儿子节点）-----
57 stu
58 stu
59 teacher
60 -----获取根节点下面指定名称的子节点（儿子节点）-----
61 stu
62 stu
63 -----获取根节点中的所有属性-----
64 sid=1
65 sname=briup

```

DOM4J中任何一个元素对象 `org.dom4j.Element`，都可以使用以上操作

使用XPath（路径表达式），获取XML中的内容信息：

XPath 是xml的路径语言，使用路径表达式来操作xml文档，使用XPath操作xml文档更加便捷。（**了解即可**）

注意，使用XPATh，需要再引入一个额外的jar包：jaxen-1.2.0.jar

1. dom4j提供了两个方法支持XPath搜索

```
List selectNodes(String expr);
```

```
Node selectSingleNode(String expr);
```

2. Xpath表达式的几种写法

第一种形式：

/AAA/BBB/CCC:表示层级结构，表示AAA下面BBB下面的所有CCC

第二种形式：

//BBB：选择文档中所有的BBB元素

第三种形式：

/AAA/BBB/\*：选择目录下的所有元素

//\*：选择所有的元素

//AAA[1]/BBB：选择第一个AAA下的BBB元素

第四种形式：

//AAA/BBB[1]：选择所有AAA的第一个BBB元素

//AAA/BBB[last()]：选择所有AAA的最后一个BBB元素

第五种形式：

//@id：选择所有的id属性

//BBB[@id]：选择具有id属性的BBB元素

第六种形式：

//BBB[@id='b1'] : 选择含有属性id并且其值为b1的BBB元素

例如，

```
1  public class Test {
2      public static void main(String[] args) {
3
4          Test test = new Test();
5          String filePath = "src/com/briup/demo/class.xml";
6
7          try {
8              Document document = test.parse(new File(filePath));
9
10             test.handler2(document);
11
12         } catch (Exception e) {
13             e.printStackTrace();
14         }
15     }
16
17     public Document parse(File file) throws DocumentException {
18         SAXReader reader = new SAXReader();
19         Document document = reader.read(file);
20         return document;
21     }
22
23     public void handler2(Document document) {
24
25         List<Node> list = document.selectNodes("//age");
26         list.forEach(n->System.out.println(n.getName()+"="+n.getText()));
27
28         System.out.println("-----");
29
30         Node node = document.selectSingleNode("//stu/score");
31         System.out.println(node.getName()+" = "+node.getText());
32
33         System.out.println("-----");
34
35         String course = node.valueOf("@course");
36         System.out.println(course);
37
38     }
39 }
40
41 //运行结果:
42 age=23
43 age=21
44 -----
45 score = 90
46 -----
47 数学
48
```

递归方式遍历出XML中的每个元素及其文本内容：

```
1  public class Test {
```

```

2     public static void main(String[] args) {
3
4         Test test = new Test();
5         String filePath = "src/com/briup/demo/class.xml";
6
7         try {
8             Document document = test.parse(new File(filePath));
9
10            test.treeWalk(document.getRootElement());
11
12        } catch (Exception e) {
13            e.printStackTrace();
14        }
15
16    }
17
18    public Document parse(File file) throws DocumentException {
19        SAXReader reader = new SAXReader();
20        Document document = reader.read(file);
21        return document;
22    }
23
24    public void treeWalk(Element element) {
25        for (int i = 0, size = element.nodeCount(); i < size; i++) {
26            Node node = element.node(i);
27            //如果这是一个元素节点
28            if (node instanceof Element) {
29                treeWalk((Element) node);
30            }
31            //如果这是一个文本节点
32            else {
33                //如果是空格、回车换行之类的就不输出了
34                if (!"".equals(node.getText().trim())) {
35                    System.out.println(node.getText());
36                }
37            }
38        }
39    }
40 }
41 //运行结果:
42 王五
43 23
44 90
45 李四
46 21

```

使用DOM4J创建一个xml文档：

```

1     public class Test {
2         public static void main(String[] args) {
3
4             Test test = new Test();
5
6             try {
7
8                 test.createXML(test.createDocument());

```

```

9
10     } catch (Exception e) {
11         e.printStackTrace();
12     }
13
14 }
15
16 public Document parse(File file) throws DocumentException {
17     SAXReader reader = new SAXReader();
18     Document document = reader.read(file);
19     return document;
20 }
21
22 @SuppressWarnings("all")
23 public Document createDocument() {
24     Document document = DocumentHelper.createDocument();
25     Element root = document.addElement("root");
26
27     Element author1 = root.addElement("author")
28         .addAttribute("name", "briup1")
29         .addAttribute("location", "上海")
30         .addText("hello");
31
32     Element author2 = root.addElement("author")
33         .addAttribute("name", "briup2")
34         .addAttribute("location", "北京")
35         .addText("world");
36
37     return document;
38 }
39
40 public void createXML(Document document) {
41     try {
42         FileWriter out = new FileWriter("src/com/briup/demo/my.xml");
43         document.write(out);
44         out.close();
45     } catch (IOException e) {
46         e.printStackTrace();
47     }
48 }
49 }

```

运行后得到的XML文件如下：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <root><author name="briup1" location="上海">hello</author><author name="briup2"
   location="北京">world</author></root>

```

使用DOM4J创建一个 "漂亮格式" 的xml文档：

```

1 public class Test {
2     public static void main(String[] args) {
3
4         Test test = new Test();
5
6         try {

```

```

7
8         test.createPrettyXML(test.createDocument());
9
10    } catch (Exception e) {
11        e.printStackTrace();
12    }
13
14 }
15
16 public Document parse(File file) throws DocumentException {
17     SAXReader reader = new SAXReader();
18     Document document = reader.read(file);
19     return document;
20 }
21
22 @SuppressWarnings("all")
23 public Document createDocument() {
24     Document document = DocumentHelper.createDocument();
25     Element root = document.addElement("root");
26
27     Element author1 = root.addElement("author")
28         .addAttribute("name", "briup1")
29         .addAttribute("location", "上海")
30         .addText("hello");
31
32     Element author2 = root.addElement("author")
33         .addAttribute("name", "briup2")
34         .addAttribute("location", "北京")
35         .addText("world");
36
37     return document;
38 }
39
40 public void createPrettyXML(Document document) {
41     try {
42         FileWriter out = new FileWriter("src/com/briup/demo/myPretty.xml");
43         //指定格式: pretty
44         OutputFormat format = OutputFormat.createPrettyPrint();
45         XMLWriter writer = new XMLWriter(out, format);
46         writer.write(document);
47         writer.close();
48     } catch (IOException e) {
49         e.printStackTrace();
50     }
51 }
52 }

```

运行后得到的XML文件如下：

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <root>
4      <author name="briup1" location="上海">hello</author>
5      <author name="briup2" location="北京">world</author>
6  </root>
7

```

