Linguagem de Definição de Dados

Análise e Desenvolvimento de Sistemas

Paulo Maurício Gonçalves Júnior

Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco

23 de agosto de 2023

Parte IX

DOM – Document Object Model

Introdução

- DOM provê uma API que lhe permite criar, modificar, remover e rearranjar nós.
- Todos os dados do documento XML ficam armazenados em memória.
- Vantagem
  - Pode revisitar conteúdo.
  - Pode ser usado para criar documentos.
- Desvantagens
  - Alto consumo de memória.

Tipos de Nós I

- Todo item constituinte do XML como elementos, atributos, texto, etc. é chamado de nó.
  - *Document Node*: nó no topo da árvore. Não é um elemento, representando todo o documento.
  - *DocumentFragment Node*: usado para armazenar uma parte do documento. Não precisa estar bem formado.
  - *Element Node*: representa um elemento.
  - *Attribute Node*: representa um único atributo.
  - *Text Node*: conteúdo textual.
  - *DocumentType*, *CDATASection*, *Notation*, *Entity*, *Comment Nodes*: usos mais avançados.
- Cada nó possui três propriedades cujos valores variam de acordo com o nó. São eles: nome, tipo e valor.
- Vejamos uma tabela com os tipos mais comuns:

Tipos de Nós II

Nó	**Tipo**	**Nome**	**Valor**
Elemento	1	Nome da tag	null
Atributo	2	Nome do atributo	Valor do atributo
Texto	3	#text	Conteúdo textual
Comentário	8	#comment	Conteúdo do comentário
Documento	9	#document	null

Lendo um documento XML

- Podemos criar uma representação em memória de um arquivo XML:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(new File(filename));
```

- A partir do objeto `Document` podemos ter acesso a todos os nós do arquivo XML.

Percorrendo nós

- A partir de um nó, podemos percorrer nós próximos ao atual. Exemplos: `getFirstChild()`, `getLastChild()`, `getNextSibling()`, `getPreviousSibling()`, e `getParentNode()`.
- Podemos ter acesso às informações de um nó, como vistos anteriormente: `getNodeName()`, `getNodeType()` e `getNodeValue()`.
- Podemos pesquisar por elementos:
  - `getElementById()`: retorna o único elemento com um identificador específico. Disponível em `Document`.
  - `getChildNodes()`: retorna todos os nós filhos do nó atual.
  - `getElementsByTagName()`: retorna todos os elementos cujo nome é igual ao parâmetro passado. Disponível em `Document` e elementos. Para iterar nos elementos retornados:

```
NodeList nList = doc.getElementsByTagName("year");
for (int i = 0; i < nList.getLength(); i++) {
    Element ele = (Element) nList.item(i);
    list.add(ele.getElementsByTagName("MonthId").item(0).
    getTextContent());
}
```

XPath I

- Os métodos básicos de DOM são muito restritos na pesquisa de elementos.
- Podemos obter qualquer tipo de nó usando XPath.
- Passaremos uma expressão XPath e poderemos processar os resultados.

```
XPathFactory xpathfactory = XPathFactory.newInstance();
XPath xpath = xpathfactory.newXPath();
XPathExpression expr = xpath.compile("//book[@year>2001]/title/text()")
    ;
```

- Após criar uma expressão, informaremos o documento no qual a expressão deve ser avaliada.
- É importante passarmos o tipo de resultado esperado, presentes na classe `XPathConstants`. Os valores possíveis são
  - BOOLEAN
  - NODE

XPath II

- NODESET
- NUMBER
- STRING

```
Object result = expr.evaluate(doc, XPathConstants.NODESET);
NodeList nodes = (NodeList) result;
for (int i = 0; i < nodes.getLength(); i++) {
  System.out.println(nodes.item(i).getNodeValue());
}

expr = xpath.compile("count(//book/title)");
result = expr.evaluate(doc, XPathConstants.NUMBER);
Double count = (Double) result;
System.out.println(count.intValue());
```

- Para usarmos XPath 2.0 em diante, precisaremos utilizar a biblioteca Saxon.
- As classes estão no pacote `net.sf.saxon.s9api`.

XPath III

```
Processor p = new Processor(false);
DocumentBuilder dbu = p.newDocumentBuilder();
XdmNode node = dbu.build(new File("input.xml"));
XPathCompiler xpath2 = p.newXPathCompiler();
XPathExecutable exec = xpath2.compile("avg(//price)");
XPathSelector selector = exec.load();
selector.setContextItem(node);
System.out.println(selector.evaluateSingle());
```

Criação de Documento I

- Também podemos criar um documento vazio ou modificar um documento existente.
- Podemos criar, remover, alterar nós e posteriormente criar um arquivo XML baseado na árvore gerada.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.newDocument();
```

- Os métodos mais comuns para criação de nós são:
  - `createElement()`: cria um elemento.
  - `createTextNode()`: cria um nó de texto.
  - `setAttribute()`: cria um atributo com um valor. Disponível dentro do objeto `Element`.
  - `setTextContent()`: modifica o nó de texto dentro de um elemento.
- Após criar nós, devemos informar onde eles serão acrescentados.
  - `appendChild()`: acrescenta um nó como último filho de um elemento.

Criação de Documento II

- `insertBefore()`: insere um nó antes de outro nó já existente.

```
Element element = doc.createElement("root");
doc.appendChild(element);

Comment comment = doc.createComment("This is a comment");
doc.insertBefore(comment, element);

Element itemElement = doc.createElement("item");
element.appendChild(itemElement);
itemElement.setAttribute("myattr", "attrvalue");
itemElement.insertBefore(doc.createTextNode("text"), itemElement.
    getLastChild());
```

XSLT I

- Também podemos realizar a transformação programática de um documento XML em outro formato usando XSLT.
- Inicialmente criamos um objeto `TransformerFactory`, que cria transformadores de XML para outro formato.

```
TransformerFactory factory = TransformerFactory.newInstance();
```

- Caso queiramos carregar os arquivos, usaremos o `StreamSource`:

```
Source xslt = new StreamSource(new File("transform.xslt"));
Source text = new StreamSource(new File("input.xml"));
```

- Caso os documentos já estejam em memória, através de objetos `document`, podemos usar o `DOMSource` ao invés do `StreamSource`:

```
Source xslt = new DOMSource(xsltDocument);
Source text = new DOMSource(xmlDocument);
```

XSLT II

- Finalmente, criamos o objeto que realizará a transformação, passamos o objeto que representa o arquivo XML e informamos onde armazenar o resultado. Se quisermos criar um arquivo:

```
Transformer transformer = factory.newTransformer(xslt);
transformer.transform(text, new StreamResult(new File("output.xml")));
```

- Também podemos criar um objeto `Document` como resultado:

```
Result out = new DOMResult();
transformer.transform(text, out);
```

- Para usarmos XSLT 2.0 em diante, precisaremos utilizar a biblioteca Saxon.
- As classes estão no pacote `net.sf.saxon.s9api`.

XSLT III

```
Processor p = new Processor(false);
XsltCompiler compiler = p.newXsltCompiler();
XsltExecutable exec = compiler.compile(xslt);
Serializer out = p.newSerializer(System.out);
out.setOutputProperty(Serializer.Property.METHOD, "html");
out.setOutputProperty(Serializer.Property.INDENT, "yes");
Xslt30Transformer transformer2 = exec.load30();
transformer2.transform(text, out);
```

Serializando um documento

- Podemos criar um documento XML a partir de sua representação em memória.
- Uma forma é realizar uma transformação sem passar um XSLT. Nenhuma transformação é realizada e o próprio arquivo XML é escrito na saída.

```
TransformerFactory transformerFactory = TransformerFactory.newInstance
    ();
Transformer transformer = transformerFactory.newTransformer();
DOMSource source = new DOMSource(doc);
StreamResult result = new StreamResult(new File("cars.xml"));
transformer.transform(source, result);
```

- Para escrever na saída padrão, basta modificar o objeto que representa o resultado:

```
StreamResult consoleResult = new StreamResult(System.out);
transformer.transform(source, consoleResult);
```

JAXB I

- Em Java, podemos converter um arquivo XML em um objeto Java e vice-versa.
- Para isso, será feita uma conversão do conteúdo dos elementos ou atributos para atributos em uma ou mais classes Java.
- Vejamos como fazer a conversão de um objeto em Java para XML:
- Temos uma classe POJO:

```
public class Pessoa {
    private String nome;
    private double salario;
    private int idade;

    // Construtores
    // Getters e setters
}
```

- A conversão é feita assim:

JAXB II

```
Pessoa p = new Pessoa("Fulano", 1234.56, 23);
JAXBContext context = JAXBContext.newInstance(Pessoa.class);
Marshaller m = context.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(p, System.out);
```

- Podemos verificar que o arquivo XML possui elementos com o mesmo nome dos atributos da classe.
- Caso tenhamos a mesma classe e um arquivo XML como o abaixo, como criar um objeto com os dados do XML?

pessoa.xml

```
<pessoa>
    <idade>23</idade>
    <nome>Fulano</nome>
    <salario>1234.56</salario>
</pessoa>
```

JAXB III

- A conversão é feita assim:

```
JAXBContext context = JAXBContext.newInstance(Pessoa.class);
Unmarshaller um = context.createUnmarshaller();
Pessoa p = (Pessoa) um.unmarshal(new StreamSource("pessoa.xml"));
```

- Podemos acrescentar anotações para ter mais controle sobre a conversão:
  - Permitir que o nome da classe/atributos não precisem ser iguais ao nome dos elementos.
  - Permitir que atributos em XML sejam ligados a atributos da classe.
  - Documentos complexos com descendentes.
  - Simplificar o código de conversão.
- A seguir veremos as mais comuns.

Anotações

@XmlRootElement

- Define a raiz de um documento.

```
@XmlRootElement
public class Pessoa {
    private String nome;
    private double salario;
    private int idade;

    // Construtores
    // Getters e setters
}
```

- Se quisermos que o nome do elemento no XML não seja igual ao nome da classe, basta passar um parâmetro para a anotação:

```
@XmlRootElement(name = "person")
```

Anotações

@XmlAcessorType

- Define quais atributos das classe Java serão exportadas para o XML.
  - FIELD: Todo atributo não estático.
  - NONE: Nenhum atributo.
  - PROPERTY: Todo par de método get/set.
  - PUBLIC_MEMBER: Todo par de método get/set e todo atributo público (padrão).

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Pessoa {
    private String nome;
    private double salario;
    private int idade;
}
```

Anotações

@XmlElement

- Mapeia um atributo de uma classe Java para um elemento XML.

```
@XmlRootElement
public class Pessoa {
    @XmlElement
    private String nome;
    @XmlElement(name="salary")
    private double salario;
    private int idade;
}
```

Anotações

@XmlAttribute

- Mapeia um atributo de uma classe Java para um atributo XML.

```
@XmlRootElement
public class Pessoa {
    @XmlAttribute
    private Integer id;
    private String nome;
    private double salario;
    private int idade;
}
```

```
<pessoa id="1">
    <idade>23</idade>
    <nome>Fulano</nome>
    <salario>1234.56</salario>
</pessoa>
```

Anotações

@XmlType

- Mapeia uma classe para um XML Schema. Define o tipo, espaço de nomes e ordem dos elementos.

```
@XmlRootElement
@XmlType(propOrder = {"salario", "idade", "nome"})
public class Pessoa {
    private String nome;
    private double salario;
    private int idade;
}
```

```
<pessoa id="1">
    <salario>1234.56</salario>
    <idade>23</idade>
    <nome>Fulano</nome>
</pessoa>
```

Anotações

@XmlTransient

- Evita que um atributo de uma classe Java seja representado no arquivo XML.

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Pessoa {
    @XmlTransient
    private int id;
    private String nome;
    private double salario;
    private int idade;
}
```

```
<pessoa>
    <nome>Fulano</nome>
    <salario>1234.56</salario>
    <idade>23</idade>
</pessoa>
```