

XML et Programmation Internet

Cours 7

kn@lri.fr

Moteur XPath en java

L'API JAXP contient un moteur XPath 1.0 complet. Outre les classes nécessaires au chargement de fichier et à la manipulation du DOM (voir cours 6), il faut charger les éléments du package `javax.xml.xpath`. Comme pour le reste de JAXP, on passe par un `XPathFactory` pour créer une nouvelle instance du moteur XPath.

Plan

- 1 Introduction, UTF-8 et XML ✓
- 2 XPath ✓
- 3 XPath (suite) ✓
- 4 XSLT ✓
- 5 XSLT (suite) ✓
- 6 DOM ✓
- 7 XPath et XSLT en Java
 - 7.1 Requêtes XPath en Java
 - 7.2 XSLT
 - 7.3 Streaming avec SAX

Exemple : packages

```
//Pour les documents et DOM
import org.w3c.dom.*;
import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

//Pour le moteur XPath
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;

public class TestXPath {
//Deux attributs pour contenir le moteur XPath et le document builder
    XPath xp_ = null;
    DocumentBuilder db_ = null;
}
```

Exemple : constructeur

```
public TestXPath () {
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        db_ = factory.newDocumentBuilder();

        XPathFactory xf = XPathFactory.newInstance();
        xp_ = xf.newXPath();

    } catch (Exception e) {
        //Peuvent être levées en cas d'erreur de création d'objet XPath
        //DocumentBuilder, par exemple si des options passées sont
        //non-supportées.
    }
}
```

Exemple : méthode d'évaluation

```
NodeList eval(String fichier, String chemin) throws Exception {

    //Création d'un DOM pour le fichier source
    Document doc = db_.parse(fichier);

    NodeList nl = (NodeList) xp_.evaluate(chemin,
                                           doc,
                                           XPathConstants.NODESET);

}
```

Méthode XPath.evaluate()

La méthode `XPath.evaluate(xpath, n, type)` permet d'évaluer une l'expression **xpath** (donnée sous-forme de chaîne de caractères), à partir du nœud contexte **n** (qui doit implémenter l'interface **Node**). Le résultat est de type **typ**. La fonction renvoie un résultat de type **Object**. L'argument **typ** peut avoir 5 valeurs possibles, définies dans la class `XPathConstants` :

- `XPathConstants.BOOLEAN`: le résultat est de type java Boolean
- `XPathConstants.NUMBER`: le résultat est de type java Double
- `XPathConstants.STRING`: le résultat est de type java String
- `XPathConstants.NODE`: le résultat est de type java Node
- `XPathConstants.NODESET`: le résultat est de type java NodeList

En effet, une expression XPath peut avoir comme valeur un booléen, un ensemble de nœuds ou une chaîne dépendant du **contexte** où elle est utilisée. On peut demander à Jaxp d'évaluer la requête XPath pour un certain contexte.

Exemple

```
Document doc = ...
String chemin = "///descendant::year[position () = 1]";

//Crée une NodeList à un élément
NodeList nl = (NodeList) xp_.evaluate(chemin, doc, XPathConstants.NODESET);

//Renvoie le nœud correspondant ou null
Node n = (Node) xp_.evaluate(chemin, doc, XPathConstants.NODE);

//Renvoie le double java correspondant à la valeur
Double d = (Double) xp_.evaluate(chemin, doc, XPathConstants.NUMBER);

//Renvoie la chaîne java correspondant au texte
String s = (String) xp_.evaluate(chemin, doc, XPathConstants.STRING);

//Renvoie la valeur de vérité correspondant au chemin
Boolean b = (Boolean) xp_.evaluate(chemin, doc, XPathConstants.BOOLEAN);
```

La classe XPathExpression

Cette classe est similaire à l'utilisation de **PreparedStatement** en JDBC.
Utilité ? compiler la requête XPath une fois pour toute et donc éviter de re-parser la chaîne de caractère à chaque appel.

Exemple :

```
XPathExpression ex = xp_.compile("//movie/title");

NodeList nl1 = (NodeList) ex.evaluate(doc1, XPathConstants.NODESET);
NodeList nl2 = (NodeList) ex.evaluate(doc2, XPathConstants.NODESET);
NodeList nl3 = (NodeList) ex.evaluate(doc3, XPathConstants.NODESET);
...
```

Applications de transformations XSLT

Appliquer une transformation XSLT est une opération complexe à cause des différentes combinaisons possibles :

- Le fichier source (ex: `movie.xml`) peut être soit déjà chargé comme un DOM, soit sous forme de fichier, soit sous forme de chaîne de caractères,...
- Le fichier destination (ex: `resultat.xhtml`) est représenté par une DOM qui doit être éventuellement *sérialisé* (i.e. retransformé en fichier XML).
- La transformation elle-même (ex: `style.xsl`) peut être sous forme diverse (fichier, URL, DOM, ...)

On a donc une série de classes d'encapsulation (Source, ...), de factory, ...

Plan

- 1 Introduction, UTF-8 et XML ✓
- 2 XPath ✓
- 3 XPath (suite) ✓
- 4 XSLT ✓
- 5 XSLT (suite) ✓
- 6 DOM ✓
- 7 XPath et XSLT en Java
 - 7.1 Requêtes XPath en Java ✓
 - 7.2 XSLT
 - 7.3 Streaming avec SAX

Création d'une transformation

Pour créer une transformation XSLT, il faut les classes suivantes, du package: `javax.xml.transform`

```
//La classe permettant d'appliquer une transformation XSLT
//ainsi que sa factory
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;

//La classe permettant de charger des transformations ou des
//arguments de transformation sous forme de fichiers
import javax.xml.transform.stream.StreamSource;

//La classe permettant de charger des documents ou transformations
//sous forme de nœuds DOM
import javax.xml.transform.dom.DOMSource;
```

Exemple

```
TransformerFactory tf = TransformerFactory.newInstance();
//On crée un StreamSource à partir d'un nom de fichier contenant
//la feuille de style XSLT
Transformer tr = tf.newTransformer(new StreamSource("style.xml"));
```

Le code ci-dessus crée un objet de type `Transformer` représentant la transformation XSLT se trouvant dans le fichier `style.xml`.

Si on avait chargé le fichier sous forme d'un arbre DOM:

```
Document style_xml = ... ; //chargement de style.xml
```

```
TransformerFactory tf = TransformerFactory.newInstance();
Transformer tr = tf.newTransformer(new DOMSource(style_xml));
```

La méthode `Transformer.transform()`

```
transform(Source xmlSource, Result outputTarget)
```

Les interfaces `Source` et `Result` permettent d'abstraire le type de l'entrée et de la sortie. Ces dernières peuvent être :

- Des objets DOM : `DOMSource` et `DOMResult`
- Des objets d'entrée sortie de la bibliothèque java standard (`File`, `InputStream`, `OutputStream`, `Reader`, `Writer`), chaîne de caractère représentant un nom de fichier : `StreamSource` et `StreamResult`

Exemple

```
//On applique le transformer associé à style.xml
//sur le fichier movie et on écrit le résultat sur
//la sortie standard :
tr.transform(new StreamSource("movies.xml"), new StreamResult(System.out));
```

Sérialisation

La manière la plus simple de *sérialiser un document* est de créer une transformation XSLT vide (i.e qui fait l'identité) et de demander à ce que le résultat soit un fichier (ou la sortie standard)

```
Document doc = ...; //l'objet DOM que l'on veut sauver dans un fichier
Transformer tr = tf.newTransformer();
tr.transform(new DOMSource(doc),
    new StreamResult(new FileOutputStream("fichier.xml")));
```

Plan

- 1 Introduction, UTF-8 et XML ✓
- 2 XPath ✓
- 3 XPath (suite) ✓
- 4 XSLT ✓
- 5 XSLT (suite) ✓
- 6 DOM ✓
- 7 XPath et XSLT en Java
 - 7.1 Requêtes XPath en Java ✓
 - 7.2 XSLT ✓
 - 7.3 Streaming avec SAX

Streaming ?

Charger un document avec DOM permet d'accéder à l'arbre « en entier » mais peut être couteux en mémoire (chaque nœud possède au moins 4 pointeurs, 2 chaînes de caractères, ...). On veut pouvoir effectuer certains types d'opération à la volée :

- Faire des statistiques sur les documents (compter le nombre d'éléments, d'attributs, ...)
- Faire des transformations simples qui préservent la structure (par exemple mettre les balises en majuscules)
- Valider vis à vis d'une DTD

Programmation événementielle

Les parseurs SAX (Simple API for XML) reposent sur la programmation événementielle. Ils lisent le fichier d'entrée et génèrent un certain nombre d'évènements, auxquels on peut réagir avec du code. Les évènements sont :

- Début de document
- Fin de document
- Ouverture de balise (avec le nom et la liste des attributs)
- Fermeture de balise (avec le nom)
- Élément texte
- Commentaire
- ...

```
import javax.xml.parsers.*;  
import org.xml.sax.*;  
import org.xml.sax.helpers.*;
```

On doit étendre la classe par `DefaultHandler`

DefaultHandler

```
//le parseur a lu length caractères qui se trouvent dans
//ch à partir de la position start
void characters(char[] ch, int start, int length)

//le parseur a détecté la fin de document
void endDocument()

//le parseur a détecté la fin d'un élément
void endElement(String uri, String localName, String qName)

//le parseur a détecté du texte « ignorable »
void ignorableWhitespace(char[] ch, int start, int length)

//le parseur a détecté le début du document
void startDocument()
//le parseur a détecté le début d'un élément
void startElement(String uri, String localName, String qName,
                  Attributes attributes)
```

Attributes est une classe auxiliaire qui permet de connaître le nom, le nombre et les valeurs des attributs pour cette balise.

Handler personnalisé

On étend la classe DefaultHandler :

```
class MyHandler extends DefaultHandler {
    private int nb_elems;
    MyHandler() {
        nb_elems = 0;
    }
    void startElement(String uri, String localName, String qName,
                     Attributes attributes)
        throws SAXException {
        nb_elems++;
    }
    int getNbElems() { return nb_elems; };
}
```

Invocation du parseur

On utilise (encore) une factory :

```
public static void main(String[] args) {
    ...
    SAXParserFactory spf = SAXParserFactory.newInstance();
    spf.setNamespaceAware(true);
    SAXParser saxParser = spf.newSAXParser();

    MyHandler my = new MyHandler();
    XMLReader xmlReader = saxParser.getXMLReader();
    xmlReader.setContentHandler(my);
    xmlReader.parse(filename);
}
```