Práce s XML

1

Jazyk XML slouží jako univerzální datový formát v celé řadě situací. Dá se použít od práce s textem, přes popis grafických objektů, až po uložení libovolných strukturovaných dat, což je asi jeho nejčastější použití, se kterým se můžeme setkat. Jazyk XML byl navržen tak, aby byl srozumitelný pro člověka a současně velice jednoduše strojově zpracovatelný. Díky tomu vznikl kolem jazyka XML velice košatý ekosystém. V tomto semináři si ukážeme základní práci s jazykem XML na platformě Java.

1 Velmi stručný úvod do formátu XML

Než se dostaneme k popisu jednotlivých rozhraní, které platforma Java pro práci s XML nabízí, uděláme krátkou exkurzi do formátu XML, ke které použijeme následující příklad.

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <document>
3     <title>Lorem Ipsum &amp; Image</title>
4     <content>
5        <text>
6           Lorem ipsum dolor sit amet, consectetur adipiscing
7           elit. In sagittis ullamcorper nisl, eget
8           ultricies enim ultrices.
9        </text>
10       <!-- komentar -->
11       <img file="https://example.com/image-1.jpg" title="sample image" />
12    </content>
13 </document>
```

Na prvním řádku máme deklaraci XML, která nám udává verzi jazyka XML, případně kódávání znaků. Dále následuje kořenový *element* `document`, který je tvořen *otevírající* a *uzavírající značkou* (tagem) `<document>` a `</document>`. Jednotlivé elementy mohou obsahovat další prvky jazyka XML, například opět elementy (viz 3. řádek a element `title`), další text (opět viz 3. řádek), komentáře (viz 10. řádek), případně další prvky, které jdou nad rámec tohoto semináře (např. instrukce pro zpracování, práci s textem). Protože některé znaky mají v jazyce XML speciální význam, např. `<`, `>` nebo `&`, jsou tyto symboly reprezentovány pomocí tzv. *entit*, jak ukazuje řádek č. 3 použití entity `&amp;` pro symbol `&`.

Za zmínku stojí i řádek č. 11, kde element `img` má dva *atributy*, jmenovitě `file` a `title` a současně zdánlivě nemá odpovídající uzavírací značku. Pokud element nemá žádný obsah, můžeme místo `<element></element>` použít kratší verzi `<element />`.

Ke zpracování XML dokumentů je nutné dodat, že narozdíl od uspořádání elementů, kde na pořádí záleží, uspořádání atributů může být libovolné, z čehož plyne, že dva elementy s různě uspořádanými atributy jsou identické. Podobně je nutné podotknout, že bílé znaky sloužící například k odsazení jsou interpretovány jako text. Proto dva dokumenty, které obsahují stejné elementy jen s jinak odsazenými značkami, mohou být chápany jako odlišné.

Představili jsme si velice malou, ale v praxi velmi často používanou podmnožinu jazyka XML. Avšak jazyk XML je docela komplexní datový formát, který má celou řadu dalších vlastností[1], proto p.t. čtenáře odkazujeme na standard jazyka, kde najde další podrobnosti.[2]

2 Simple API for XML (SAX)

Platforma Java nabízí hned několik rozhraní pro zpracování XML dokumentů. Tím nejjednodušším je Simple API for XML, které se většinou označuje jako SAX, a které k XML dokumentu přistupuje jako k posloupnosti událostí, které nastaly při zpracování dokumentu. To znamená, že *syntaktický analyzátor* (někdy též *parser*), který dokument zpracovává, postupně prochází jednotlivé prvky dokumentu jako jsou otevírací a uzavírací značky a pro každý takový prvek zavolá obslužnou metodu.

Při zpracování XML dokumentu postupujeme tak, že nejdříve vytvoříme parser:

```
SAXParserFactory parserFactory = SAXParserFactory.newInstance();
SAXParser parser = parserFactory.newSAXParser();
```

Samotné vytvoření parseru je rozděleno do dvou kroků, nejdříve vytvoříme továrnu, a tou vytvoříme samotný parser. Návrhový vzor továrna[3] zde umožňuje volit vhodnou implementaci parseru podle požadované konfigurace, například s podporou validace nebo jmenných prostorů.

Proces zpracování XML dokumentu spustíme zavoláním metody `SAXParser.parse`, které předáme dva argumenty: (i) první argument představuje XML soubor (ve formě objektu typu `InputStream`, `File` nebo `String`) a (ii) druhý argument je objekt dědící ze třídy `DefaultHandler`, který reaguje na jednotlivé události. Třída `DefaultHandler` má metody typu:

```
public void startDocument()
public void startElement(String uri, String localName, String qName, Attributes attributes)
public void endElement(String uri, String localName, String qName)
```

Překrytím jednotlivých metod definujeme jednotlivé reakce. Například následovně:

```
parser.parse(input, new DefaultHandler() {
    private int level = 0;
    @Override
    public void startDocument() throws SAXException {
```

[1] Namátkou třeba jmenné prostory, které umožňují kombinovat různé datové formáty v jednom dokumentu (např. XHTML pro text a SVG pro vektorovou grafiku).

[2] `https://www.w3.org/TR/xml/`

[3] `https://en.wikipedia.org/wiki/Abstract_factory_pattern`

```
        System.out.println("Zahajeno cteni dokumentu");
    }

    @Override
    public void startElement(String uri, String localName, String qName,
                             Attributes attributes) throws SAXException {
        System.out.println("Precten element: " + qName);
        level++;
    }
    @Override
    public void endElement(String uri, String localName, String qName)
           throws SAXException {
        System.out.println("Konec elementu: " + qName);
        level--;
    }
}
```

V tomto případě implementujeme reakce na tři typy událostí, (i) začátek zpracování dokumentu, (ii) nalezení uvozovací a (iii) ukončovací značky elementu. Pro jednoduchost jsme použili anonymní třídu, ale jde použít i běžná třída. Poznamenejme, že třída nemusí obsahovat jen metody, ale může obsahovat například i stav, který souvisí se zpracováním dokumentu. V našem případě se jedná o úroveň zanoření jednotlivých elementů (atribut `level`).

Poněkud netradičně vypadá zpracování textových dat, které je realizováno metodou

```
public void characters(char[] ch, int start, int length)
```

Tato metoda vrací odkaz do pole znaků, což sice umožňuje předat text z parseru s minimální režií, ale pro další práci to není pohodlný způsob reprezentace textu. Naštěstí má třída `String` odpovídající konstruktor a je možné z takto předaných hodnot vytvořit standardní řetězec, například následovně:

```
@Override
public void characters(char[] ch, int start, int length) throws SAXException {
    System.out.println("Text: " + new String(ch, start, length));
}
```

Výhodou rozhraní SAX je, že nevyžaduje, aby byl dokument načten do paměti celý. To se hodí například při zpracování dat velkého rozsahu (např. mnoha gigabytové soubory z OpenStreetMap) a zpracování dat tak může být velice rychlé.

3 Streaming API for XML (StAX)

Určitým neduhem rozhraní SAX může být způsob řízení výpočtu. V momentě, kdy zavoláme metodu `parse`, řízení přebírá parser a produkuje jednu událost za druhou, které jsou zpracovávány jedním ob-

jektem. Tento přístup je dostačující, pokud se pracuje s jednoduše strukturovanými daty, u komplexních formátů to často vede na komplikovanému zpracování dat a k nepřehlednému kódu.

Odpovědí na tento problém je Streaming API for XML (StAX), které přistupuje k XML dokumentu podobně jako SAX, tj. nahlíží na něj také jako na proud (posloupnost) prvků dokumentu s tím rozdílem, že je to volající kód (nikoliv parser), kdo z dokumentu „vytahuje" jednotlivé prvky.[4]

3.1 StAX: Čtení dokumentu

Klíčovým bodem rozhraní StAX je objekt typu `XMLStreamReader`, který zpřístupňuje jednotlivé prvky dokumentu. Objekt typu `XMLStreamReader` opět získáme ve dvou krocích, kdy nejdříve vytvoříme továrnu a až následně žádaný objekt, kterému předáme objekt obsahující samotný dokument, v našem případě se jedná o objekt typu `InputStream`.

```
InputStream input = /* ... */;
XMLInputFactory xmlInputFactory = XMLInputFactory.newInstance();
XMLStreamReader reader = xmlInputFactory.createXMLStreamReader(input);
```

Princip, na kterém funguje objekt `XMLStreamReader` je velice podobný návrhovému vzoru iterátor[5]. Objekt třídy `XMLStreamReader` ukazuje na nějaký prvek dokumentu, jehož typ můžeme získat metodou `getEventType()`. Jedná-li se o uvozovací nebo ukončovací značku, můžeme získat jméno elementu metodou `getName()` apod., jak ukazuje následující příklad.

```
switch (reader.getEventType()) {
case XMLStreamReader.START_ELEMENT:
    System.out.println("Precten element: " + reader.getName());
    break;
case XMLStreamReader.END_ELEMENT:
    System.out.println("Konec elementu: " + reader.getName());
    break;
case XMLStreamReader.CHARACTERS:
    System.out.println("Text: " + reader.getText());
    break;
default:
    break;
}
```

Pokud se chceme přesunout k dalšímu prvku dokumentu, zavoláme metodu `reader.next()`. Tato metoda se obvykle používá ve spojení s metodou `reader.hasNext()`, která indikuje, zda jsme narazili na konec dokumentu, nebo ne.

Výhodou tohoto přístupu je, že nekoncentruje zpracování dokumentu do jednoho objektu (instance třídy `DefaultHandler`), ale je možné kód rozdělit do logických celků podle toho, jakou část dokumentu zpracovávají.

[4] SAX se dá označit jako přístup typu *push*, StAX jako přístup *pull*, kdy z parseru vytahujeme jednotlivé prvky.

[5] `https://en.wikipedia.org/wiki/Iterator_pattern`

3.2 StAX: Tvorba dokumentu

Narozdíl od SAX nabízí rozhraní StAX prostředky i pro vytváření XML dokumentů. Toto rozhraní je velmi intuitivní, jak ukazuje následující příklad.

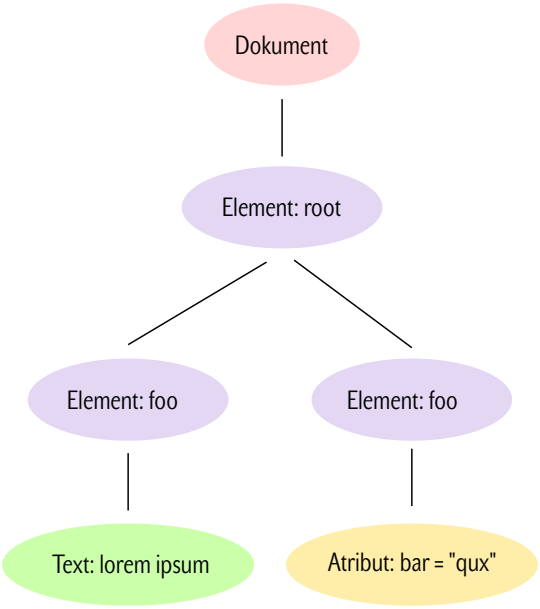
```
StringWriter buf = new StringWriter();

XMLOutputFactory xmlOutputFactory = XMLOutputFactory.newInstance();
XMLStreamWriter xmlWriter = xmlOutputFactory.createXMLStreamWriter(buf);
```

Nejdříve si vytvoříme cíl, kam budeme data zapisovat. V tomto případě jsme použili `StringWriter`, který funguje jako buffer, kam se bude výsledný dokument zapisovat. Můžeme ale použít libovolnou implementaci třídy `Writer` nebo `OutputStream`. Poté, pomocí `XMLOutputFactory` vytvoříme `XMLStreamWriter`, který představuje rozhraní, kterým do cíle (argument `buf`) bude zapisovat jednotlivé prvky XML dokumentu.

Toto rozhraní je velice přímočaré a jednotlivé jeho metody odpovídají jednotlivým prvkům jazyka XML, jak ukazuje následující příklad.

```
xmlWriter.writeStartDocument();

// korenovy element
xmlWriter.writeStartElement("root");

xmlWriter.writeStartElement("foo");
xmlWriter.writeCharacters("Lorem ipsum");
xmlWriter.writeEndElement();

xmlWriter.writeStartElement("foo");
xmlWriter.writeAttribute("bar", "qux");
xmlWriter.writeEndElement();

// konec korenoveho element
xmlWriter.writeEndElement();

xmlWriter.writeEndDocument();
```

Mohlo by se zdát, že takové rozhraní je pro svou jednoduchost zbytečné a jeho funkci dokáže zastupit standardní `StringBuilder`. Ďáběl je ukryt v detailu. Vzhledem k tomu, že některé znaky mají v XML speciální význam, je nutné je konvertovat na odpovídající entity, o což se rozhraní StAX postará. V případě tvorby XML dokumentu pomocí `StringBuilderu` je dost dobře možné, že programátor zapomene převést speciální znaky na entity a výsledný program tak bude vytvářet nevalidní dokumenty. Proto rozhraní StAX dává smysl i u jednoduchých XML dokumentů.



Obrázek 1: Ilustrace DOM

4 Document Object Model (DOM)

Rozhraní SAX i StAX dokáží být velice rychlé a nenáročně při zpracování dokumentů, protože potřebují držet v paměti jen velmi malou (aktuálně zpracovávanou) část dokumentu. Tato rychlost a efektivita je však vykoupena nízkou mírou abstrakce, kterou toto rozhraní poskytuje. Všimněme si, že v obou případech je přirozeně strukturovaný dokument redukován na jednorozměrný proud prvků, a potřebujeme-li získat zpět tuto strukturu, musí se o to programátor postarat sám.

Tento problém řeší Document Object Model (zkráceně DOM), který nahlíží na XML dokument jako na strom objektů, kde jednotlivé uzly představují prvky jazyka jako jsou elementy, textové řetězce nebo atributy. Tuto reprezantaci ilustruje Obrázek 1.

4.1 DOM: Čtení dokumentu

Načtení dokumentu je realizováno ve třech krocích:

```
DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();
Document doc = documentBuilder.parse(input);
```

Nejdříve pomocí `DocumentBuilderFactory` získáme objekt typu `DocumentBuilder`, který slouží k sestavení dokumentu a následně zavoláním metody `DocumentBuilder.parse(InputStream)`[6], získáme XML dokument reprezentovaný objektem typu `Document`.

S takto zpracovaným dokumentem se dá již pohodlně manipulovat. Obvykle tak, že získáme kořenový element, případně jeho části, a strom procházíme do hloubky, například, jak ukazuje následující kód.

[6] Všimně me si, že metoda `parse` může skončit výjimkou `SAXParserException`, což ukazuje, že k `DocumentBuilder` vnitřně využívá SAX.

```
Element root = doc.getDocumentElement();
System.out.println("Korenovy element:" + root.getNodeName());

for (int i = 0; i < root.getChildNodes().getLength(); i++) {
    Node node = root.getChildNodes().item(i);
    if (node.getTextContent().length() > 0) {
        System.out.println("Potomek c. " + i + " obsahuje text: " + node.getTextContent());
    }
    if (node.hasAttributes()) {
         // atribut je taky uzel
         Node attr = node.getAttributes().getNamedItem("bar");
         String value = attr.getTextContent();
         System.out.println("Potomek c. " + i + " ma attribut bar='" + value + "'");
     }
}
```

Na tomto rozhraní je patrné, že úplně nezapadá do konvencí jazyka Java. Je to dáno tím, že toto rozhraní vzniklo pod hlavičkou W3C primárně potřeby webových prohlížečů[7].

4.2 DOM: Tvorba dokumentu

Tvorba XML dokumentu s DOM spočívá v sestavení stromu objektů, které odpovídájí částem dokumentu.

K vytvoření nového XML dokumentu potřebujeme objekt třídy `DocumentBuilder`, jehož vytvoření jsme si ukázali v předchozí kapitole. S jeho metodou `newDocument()` vytvoříme nový dokument.

```
Document doc = documentBuilder.newDocument();
```

A následně můžeme vytvářet nové elementy, jak ukazuje následující kód.

```
Element rootElement = doc.createElement("root");
Element childNode1 = doc.createElement("foo");
Element childNode2 = doc.createElement("foo");
childNode2.setAttribute("bar", "qux");
Text textChildNode = doc.createTextNode("Lorem ipsum");
```

Všimněme si, že k vytvoření nových elementů je použit, poněkud neintuitivně, objekt `Document` a současně nově vytvořené elementy netvoří žádnou strukturu. Tu je nutné vytvořit propojením objektů do stromu například pomocí metody `Element.appendChild`, jak ukazuje následující příklad.

```
doc.appendChild(rootElement)
rootElement.appendChild(childNode1);
rootElement.appendChild(childNode2);
rootElement.appendChild(textChildNode);
```

[7] `https://www.w3.org/DOM/DOMTR`

Takto vytvořený XML dokument existuje jako strom objektů v paměti počítače, zbývá jej tedy převést do textové reprezentace. K tomu existují dva možné přístupy, z čehož ani jeden není úplně přímočarý.

4.2.1 Použití Load-And-Store implementace DOM

První způsob konverze DOM reprezentace XML dokumentu do textové podoby, který si ukážeme, je postavený na implementaci DOM s podporou pro ukládání a náčítání (Load/Store). Jelikož může existovat několik implementací DOM, které nepodporují kompletní sadu operací, je nejdříve nutné získat implementaci DOM s podporou načítání a ukládání, jak ukazuje následující příklad, viz argument `LS`.

```
DOMImplementationRegistry registry = DOMImplementationRegistry.newInstance();
DOMImplementationLS impl = (DOMImplementationLS) registry.getDOMImplementation("LS");
```

A z této implementace následně můžeme použít serializér, který dokument převede z objektů na text, jak ukazuje následující příklad.

```
LSSerializer serializer = impl.createLSSerializer();
String xml = serializer.writeToString(doc);
```

4.2.2 Využití nástrojů pro transformaci

Součástí ekosystému kolem jazyka XML jsou i nástroje pro transformaci XML dokumentů na jiné dokumenty[8]. A právě tyto mechanismy můžeme použít ke konverzi stromu objektů na text.

Nejdříve si vytvoříme továrnu na transformátory, pomocí níž vytvoříme transformátor, který převádí zdrojový dokument na cílový dokument bez jakékoliv změny, jak ukazuje následující kód.

```
TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
```

Stačí nám již jen zavolat metodu `transform` s vhodně obalenými argumenty:

```
DOMSource source = new DOMSource(doc);
StreamResult result = new StreamResult(System.out);
transformer.transform(source, result);
```

Vstupní dokument obalíme do objektu `DOMSource` a výstup v našem případě objekt typu `OutputStream` obalíme do objektu `StreamResult` tak, aby s nimi mohl transformátor pracovat.

Zajímavostí tohoto přístupu je, že umožňuje jednoduše specifikovat vlastnosti, toho jak má výsledný dokument vypadat. Například následujícím kódem můžeme určit, zda se má použít odsazení.

```
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
```

[8] Nejtypičtějším příkladem je jazyk XSLT.

5 Shrnutí

V tomto semináři jsme si představili tři přístupy ke zpracování XML dokumentů. Z těchto tří je, díky svému používání v webových prohlížečích, asi nejznámnější DOM. Vedle toho má DOM tu výhodu, že představuje pro programátora rozumnou abstrakci. Nevýhodou je, že tento přístup má nezanedbatelné nároky na paměť. Proto přístupy postavené na SAX nebo StAX davají smysl, obvzlášť v situacích, kdy pracujeme s velkými daty nebo je žádoucí data zpracovávat rychle. Kdy je vhodnější který přístup, záleží na konkrétní situaci.

Závěrem dodejme, že tento text obsahuje jen části zdrojových kódů. Úplné zdrojové kódy jsou k dispozici na webu semináře.