実用的な Groovy: XML を作成し、構文解析し、容易に扱う

XML を苦労なく操作する

Scott Davis

2009年 5月 19日

Groovy を使うと XML をどれほど容易に扱えるようになるかを学びましょう。今回の「実用的な Groovy」では、MarkupBuilder や StreamingMarkupBuilder を使って XML を作成する場合であれ、あるいは XmlParser や XmlSlurper を使って XML を構文解析する場合であれ、Groovy がこの XML という汎用データ・フォーマットを処理するための強力なツール・セットになることを、著者の Scott Davis が説明します。

このシリーズの他の記事を見る

XML は相当以前から存在しているように感じられます。実際、XML が誕生して 2008年で 10 年になりました (「参考文献」を参照)。Java™ 言語が XML よりもほんの 2、3 年前に生まれたことを考えると、Java 開発者にとっては XML は相当以前から存在していると言うことができます。

このシリーズについて

Groovy は Java プラットフォーム上で実行される最新のプログラミング言語の 1 つです。Groovy は既存の Java コードとシームレスに統合できる一方、クロージャーやメタプログラミングなどの強力な新機能も導入することができます。簡単に言えば Groovy とは、21 世紀に Java 言語が作成されていたら Groovy のようになっていたであろう、そういった言語なのです。

開発ツールキットの一部として新しいツールを採用する際に重要なことは、どういう場合にそのツールを使い、どういう場合には使わずにおくかを理解することです。Groovy は非常に強力ですが、適切な方法で、適切な状況の中で使用した場合にのみ強力なツールとなるのです。そのため「実用的な Groovy」シリーズでは、どういう状況で、どのようにして Groovy を使えば効果的であるかを学べるように、Groovy の実用的な使い方を解説します。

Java 言語を開発した Sun Microsystems は、XML が誕生したときから XML を強力にサポートしてきました。結局のところ、プラットフォームからの独立を約束している XML は Java 言語の「1 度作成すれば、どこででも実行できる」というメッセージと見事に適合します。この 2 つの技術がともに持つ実用性を考えると、Java 言語と XML は現状よりも協調できてよいと思うかもしれませんが、実際には、Java 言語での XML の構文解析や生成は不思議なほどに面倒で複雑です。

幸いなことに、Groovy には、XML の作成や処理のための新しい優れた方法が導入されています。この記事では、いくつかの例を示しながら (すべての例は「ダウンロード」セクションから入手することができます)、Groovy によって XML の作成や構文解析が驚くほど単純になることを説明します。

XML の構文解析に関する Java と Groovy との比較

私はこのシリーズの「each を活用する」の最後で、リスト 1 に示す簡単な XML 文書を紹介しました。(今回は `type` 属性を追加し、少しだけ興味深いものにしました。)

リスト 1. 私が知っている言語を一覧表示する XML 文書

```
<langs type="current">
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
```

この簡単な XML 文書の構文解析を Java 言語で行おうとすると、決して簡単ではありません。これはリスト 2 を見るとわかります。5 行の XML ファイルを構文解析するために 30 行のコードが必要です。

リスト 2. Java で XML ファイルを構文解析する

```
import org.xml.sax.SAXException;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.IOException;

public class ParseXml {
  public static void main(String[] args) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.parse("src/languages.xml");

      //print the "type" attribute
      Element langs = doc.getDocumentElement();
      System.out.println("type = " + langs.getAttribute("type"));

      //print the "language" elements
      NodeList list = langs.getElementsByTagName("language");
      for(int i = 0 ; i < list.getLength();i++) {
        Element language = (Element) list.item(i);
        System.out.println(language.getTextContent());
      }
    }catch(ParserConfigurationException pce) {
      pce.printStackTrace();
    }catch(SAXException se) {
      se.printStackTrace();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

リスト 2 の Java コードを、これに対応する Groovy コードと比較してみてください (リスト 3)。

リスト 3. Groovy で XML ファイルを構文解析する

```
def langs = new XmlParser().parse("languages.xml")
println "type = ${langs.attribute("type")}"
langs.language.each{
  println it.text()
}

//output:
type = current
Java
Groovy
JavaScript
```

Groovy コードの最も良いところは、等価な Java コードよりも大幅に短いことではありません (もちろん、5 行の XML を 5 行で構文解析できる Groovy の方が優れていることは明確です)。私が Groovy コードで最も良いと思う点は、Groovy コードの方がはるかに表現力が豊かなことです。`langs.language.each` と書くと、私は XML を直接扱っているような気がします。Java バージョンでは XML を扱っている気がしません。

String 変数と XML

XML を Groovy で処理することによる利点は、XML をファイルに保存するのではなく `String` 変数に保存する際に一層明らかになります。Groovy の三重引用符 (他の言語では通常、ヒア・ドキュメント (`HereDoc`) と呼ばれます) を利用すると、何も苦労せずに XML を内部に保存することができます (リスト 4)。リスト 3 での Groovy の例との唯一の違いは、`XmlParser` メソッドから `parse()` (`File`、`InputStreams`、`Reader`、そして `URI` を処理します) を呼び出す代わりに `parseText()` を呼び出すようにしていることです。

リスト 4. Groovy で XML を内部に保存する

```
def xml = """
<langs type="current">
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println "type = ${langs.attribute("type")}"
langs.language.each{
  println it.text()
}
```

三重引用符によって複数の XML 文書を容易に操作できることに注目してください。`xml` 変数は、まさに昔ながらの単純な `java.lang.String` です (`println xml.class` を追加すれば、それを確認することができます)。また三重引用符は `type="current"` という内部引用も処理することができ、Java コードの場合のようにバックスラッシュ文字を使って内部引用を強制的にエスケープする必要がありません。

リスト 4 での単純でスマートな Groovy コードを、これに対応する Java コード (リスト 5) と比較してみてください。

リスト 5. Java コードで XML を内部に保存する

```
import org.xml.sax.SAXException;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;

public class ParseXmlFromString {
  public static void main(String[] args) {
    String xml = "<langs type=\"current\">\n" +
             "  <language>Java</language>\n" +
             "  <language>Groovy</language>\n" +
             "  <language>JavaScript</language>\n" +
             "</langs>";

    byte[] xmlBytes = xml.getBytes();
    InputStream is = new ByteArrayInputStream(xmlBytes);

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.parse(is);

      //print the "type" attribute
      Element langs = doc.getDocumentElement();
      System.out.println("type = " + langs.getAttribute("type"));

      //print the "language" elements
      NodeList list = langs.getElementsByTagName("language");
      for(int i = 0 ; i < list.getLength();i++) {
        Element language = (Element) list.item(i);
        System.out.println(language.getTextContent());
      }
    }catch(ParserConfigurationException pce) {
      pce.printStackTrace();
    }catch(SAXException se) {
      se.printStackTrace();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

内部引用のためのエスケープ文字と改行によって `xml` 変数がわかりにくくなっていることに注目してください。しかしもっと面倒なことは、`String` を `byte` 配列に変換してから再度 `ByteArrayInputStream` に変換する必要があり、そうしないと構文解析できないことです。奇妙なことに、`DocumentBuilder` には単純な `String` を XML として構文解析するための直接的な方法が用意されていません。

`MarkupBuilder` を使って XML を作成する

Groovy が Java 言語よりも優れていることが最も明確にわかるのは、コードで XML 文書を作成する場合です。リスト 6 のコードは、5 行の XML スニペットを作成するために 50 行の Java コードが必要なことを示しています。

リスト 6. Java コードで XML を作成する

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
```

```
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;

public class CreateXml {
  public static void main(String[] args) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.newDocument();

      Element langs = doc.createElement("langs");
      langs.setAttribute("type", "current");
      doc.appendChild(langs);

      Element language1 = doc.createElement("language");
      Text text1 = doc.createTextNode("Java");
      language1.appendChild(text1);
      langs.appendChild(language1);

      Element language2 = doc.createElement("language");
      Text text2 = doc.createTextNode("Groovy");
      language2.appendChild(text2);
      langs.appendChild(language2);

      Element language3 = doc.createElement("language");
      Text text3 = doc.createTextNode("JavaScript");
      language3.appendChild(text3);
      langs.appendChild(language3);

      // Output the XML
      TransformerFactory tf = TransformerFactory.newInstance();
      Transformer transformer = tf.newTransformer();
      transformer.setOutputProperty(OutputKeys.INDENT, "yes");
      StringWriter sw = new StringWriter();
      StreamResult sr = new StreamResult(sw);
      DOMSource source = new DOMSource(doc);
      transformer.transform(source, sr);
      String xmlString = sw.toString();
      System.out.println(xmlString);
    }catch(ParserConfigurationException pce) {
      pce.printStackTrace();
    } catch (TransformerConfigurationException e) {
      e.printStackTrace();
    } catch (TransformerException e) {
      e.printStackTrace();
    }
  }
}
```

読者のなかには、これを見て「これはやり過ぎだ」と叫んでいる方がいるかもしれません。このコードをもっと単純にできるサードパーティーのライブラリーがたくさんあります (よく使われるものとして JDOM と dom4j の 2 つがあります)。しかしどの Java ライブラリーも、Groovy の `MarkupBuilder` を使った場合の単純さにはかないません (リスト 7)。

リスト 7. Groovy で XML を作成する

```
def xml = new groovy.xml.MarkupBuilder()
xml.langs(type:"current"){
  language("Java")
  language("Groovy")
  language("JavaScript")
}
```

こうすると、コードと XML の比がほとんど 1:1 になることに注目してください。もっと重要なことは、こうすることで XML が見えるようになることです。もちろん、不等号括弧が中括弧のクロージャーに置き換えられ、また属性には等号の代わりにコロン (Groovy の `HashMap` 表記) が使われていますが、Groovy の場合も XML の場合も、基本的な構造を読み取ることができます。これはまるで XML を作成するための DSL (ドメイン特化言語) のように思えないでしょうか。

Groovy がこの `Builder` マジックを実現できるのは、Groovy が動的な言語であるためです。一方 Java 言語は静的です。Java コンパイラーはメソッドを呼び出す前に、それらのメソッドがすべて存在することを確認します。(存在しないメソッドを呼び出そうとすると、Java コードをコンパイルすることすらできず、もちろん実行させることもできません。) しかし Groovy の `Builder` を見ると、ある言語のバグが別の言語の特徴になることがわかります。`MarkupBuilder` の API ドキュメントを見ると、`langs()` メソッドも `language()` メソッドもなく、他の要素名もありません。幸いなことに、Groovy は存在しないメソッドへのそうした呼び出しを捉え、それらを使って少し生産的なことをします。例えば `MarkupBuilder` の場合には、存在しないメソッドへの呼び出しを捉え、それを基に整形式の XML を生成します。

リスト 8 は上で示した簡単な `MarkupBuilder` の例を拡張したものです。`String` 変数の中に XML 出力を取り込みたい場合には、`MarkupBuilder` のコンストラクターに `StringWriter` を渡します。`langs` に属性を追加したい場合には、追加したい属性を単純にカンマで区切って渡します。`language` 要素の本体が値であり、前に名前がないことに注目してください。同じカンマ区切りのリストの中に属性と本体を追加することができるのです。

リスト 8. `MarkupBuilder` を拡張した例

```
def sw = new StringWriter()
def xml = new groovy.xml.MarkupBuilder(sw)
xml.langs(type:"current", count:3, mainstream:true){
  language(flavor:"static", version:"1.5", "Java")
  language(flavor:"dynamic", version:"1.6.0", "Groovy")
  language(flavor:"dynamic", version:"1.9", "JavaScript")
}
println sw

//output:
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
```

こうした、`MarkupBuilder` によるいくつかの手法を使えるようになると、興味深いことができます。例えば、整形式の HTML 文書を素早く作成し、それをファイルに書き出すことができます。リスト 9 はそのコードを示しています。

リスト 9. `MarkupBuilder` を使って HTML を作成する

```
def sw = new StringWriter()
def html = new groovy.xml.MarkupBuilder(sw)
html.html{
  head{
    title("Links")
  }
  body{
```

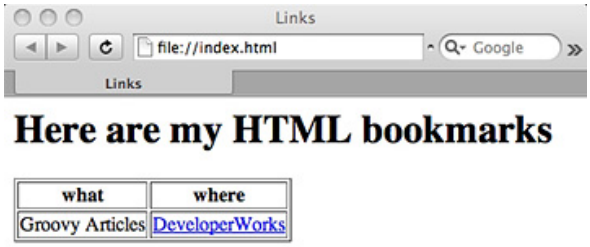
```
    h1("Here are my HTML bookmarks")
    table(border:1){
      tr{
        th("what")
        th("where")
      }
      tr{
        td("Groovy Articles")
        td{
          a(href:"http://ibm.com/developerworks", "DeveloperWorks")
        }
      }
    }
  }
}

def f = new File("index.html")
f.write(sw.toString())

//output:
<html>
  <head>
    <title>Links</title>
  </head>
  <body>
    <h1>Here are my HTML bookmarks</h1>
    <table border='1'>
      <tr>
        <th>what</th>
        <th>where</th>
      </tr>
      <tr>
        <td>Groovy Articles</td>
        <td>
          <a href='http://ibm.com/developerworks'>DeveloperWorks</a>
        </td>
      </tr>
    </table>
  </body>
</html>
```

図 1 はリスト 9 で作成した HTML をブラウザーで表示したものを示しています。

図 1. 表示された HTML



`StreamingMarkupBuilder` を使って XML を作成する

`MarkupBuilder` は単純な XML 文書を同時に作成する場合に非常に有効です。もっと高度な XML を作成する場合のために、Groovy には `StreamingMarkupBuilder` が用意されていま

す。StreamingMarkupBuilder を利用すると、XML に関連するあらゆるものを mkp ヘルパー・オブジェクトを使って追加することができます (処理命令、名前空間、そして (CDATA ブロックに最適な) エスケープされないテキストなど)。リスト 10 では StreamingMarkupBuilder のさまざまな興味深い機能を使っています。

リスト 10. StreamingMarkupBuilder を使って XML を作成する

```
def comment = "<![CDATA[<!-- address is new to this release -->]]>"
def builder = new groovy.xml.StreamingMarkupBuilder()
builder.encoding = "UTF-8"
def person = {
  mkp.xmlDeclaration()
  mkp.pi("xml-stylesheet": "type='text/xsl' href='myfile.xslt'" )
  mkp.declareNamespace('':'http://myDefaultNamespace')
  mkp.declareNamespace('location':'http://someOtherNamespace')
  person(id:100){
    firstname("Jane")
    lastname("Doe")
    mkp.yieldUnescaped(comment)
    location.address("123 Main")
  }
}
def writer = new FileWriter("person.xml")
writer << builder.bind(person)

//output:
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type='text/xsl' href='myfile.xslt'?>
<person id='100'
        xmlns='http://myDefaultNamespace'
        xmlns:location='http://someOtherNamespace'>
  <firstname>Jane</firstname>
  <lastname>Doe</lastname>
  <![CDATA[<!-- address is new to this release -->]]>
  <location:address>123 Main</location:address>
</person>
```

ここで、bind() メソッドが呼び出されてマークアップやさまざまな命令を含むクロージャーが渡されるまで、StreamingMarkupBuilder が最終的な XML を作成しないことに注目してください。こうすることで、XML 文書のさまざまな部分を非同期に作成することができ、それらの部分をすべて同時に出力することができます (詳細は「参考文献」を参照)。

XmlParser を理解する

Groovy には XML を作成する方法として、MarkupBuilder と StreamingMarkupBuilder が用意されており、それぞれ異なる機能を持っています。同様に、XML を構文解析する方法としても異なる機能を持った、XmlParser または XmlSlurper を使用することができます。

XmlParser を利用すると、プログラマー寄りの見方で XML 文書を構文解析することができます。Element の List や Attribute の Map の観点で XML 文書について考えることに慣れている場合には、XmlParser を使いやすいと感じるはずです。リスト 11 は XmlParser を少し分解したものです。

リスト 11. XmlParser を詳細に調べる

```
def xml = """
```

```
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println langs.getClass()
// class groovy.util.Node

println langs
/*
langs[attributes={type=current, count=3, mainstream=true};
      value=[language[attributes={flavor=static, version=1.5};
                      value=[Java]],
             language[attributes={flavor=dynamic, version=1.6.0};
                      value=[Groovy]],
             language[attributes={flavor=dynamic, version=1.9};
                      value=[JavaScript]]
            ]
]
*/
```

`XmlParser.parseText()` メソッドが `groovy.util.Node` (この場合は XML 文書のルート `Node`) を返すことに注目してください。`println langs` を呼び出すと、`println langs` は `Node.toString()` メソッドを呼び出し、デバッグ出力を返します。実際のデータに到達するためには、`Node.attribute()` または `Node.text()` を呼び出します。

`XmlParser` を使って属性を取得する

先ほど見たように、`Node.attribute("key")` を呼び出すことで個々の属性を取得することができます。`Node.attributes()` を呼び出すと、その `Node` のすべての属性を含む `HashMap` が返されます。「each を活用する」で学んだ `each` クロージャーを利用すると、各属性をウォークスルーすることは難しくありません。リスト 12 はこの一例を示しています。(`groovy.util.Node` の API ドキュメントに関しては「参考文献」を参照)。

リスト 12. 属性を HashMap として扱う `XmlParser`

```
def langs = new XmlParser().parseText(xml)

println langs.attribute("count")
// 3

langs.attributes().each{k,v->
  println "-" * 15
  println k
  println v
}

//output:
---------------
type
current
---------------
count
3
---------------
mainstream
true
```

XmlParser は属性を手軽に扱えるだけではなく、要素もスマートな方法で扱うことができます。

XmlParser を使って要素を取得する

XmlParser には、要素を照会するための、GPath という直感的な方法が用意されています。(GPath は XPath と似ていますが、単に Groovy で実装されているにすぎません。) 例えばリスト 13 は、先ほど使用した langs.language 構成体が、クエリーの結果を含むa groovy.util.NodeList を返すことを示しています。NodeList は java.util.ArrayList を継承するため、NodeList は基本的に、GPath の強大な力を与えられた List です。

リスト 13. GPath と XmlParser を使ったクエリー

```
def langs = new XmlParser().parseText(xml)

// shortcut query syntax
// on an anonymous NodeList
langs.language.each{
  println it.text()
}

// separating the query
// and the each closure
// into distinct parts
def list = langs.language
list.each{
  println it.text()
}

println list.getClass()
// groovy.util.NodeList
```

もちろん、GPath は MarkupBuilder を補完するものです。GPath は存在しないメソッドを呼び出す先ほどの場合と同じ手法を使いますが、この場合は既存の XML にクエリーを実行するために使われ、即時動作で XML を生成するために使われるわけではありません。

GPath によるクエリーの結果が List であることを知っていると、さらにコードを簡潔にすることができます。Groovy には展開ドット演算子 (*.) が用意されています。GPath は基本的に、1 行のコードの中でリストに対して繰り返し処理を行い、各項目のメソッドの呼び出しを実行します。その結果は List として返されます。例えば、クエリー結果の中にある各項目の Node.text() メソッドを呼び出したいだけであれば、リスト 14 のように 1 行のコードでそれを行うことができます。

リスト 14. 展開ドット演算子と GPath とを組み合わせる

```
// the long way of gathering the results
def results = []
langs.language.each{
  results << it.text()
}

// the short way using the spread-dot operator
def values = langs.language*.text()
// [Java, Groovy, JavaScript]

// quickly gathering up all of the version attributes
def versions = langs.language*.attribute("version")
// [1.5, 1.6.0, 1.9]
```

XmlParser はスマートで強力ですが、XmlSlurper を使うと、さらに上のレベルの XML 処理を行うことができます。

XmlSlurper を使った XML の構文解析

先ほどのリスト 2 で、Groovy を使うと、まるで XML を直接扱っているような気持ちになると言いました。確かに XmlParser は非常に強力ですが、XmlParser を使った場合も相変わらずプログラムで XML を処理する必要があります。Node の List や Attribute の HashMap を処理する必要があり、相変わらず Node.attribute() や Node.text() などを呼び出してコア・データを取得する必要があります。しかし XmlSlurper を利用すると、メソッド呼び出しに関する最後の痕跡を消し去ることができ、XML を直接扱うかのような気持ちのよい幻想にひたることができます。

技術的に言えば、XmlParser は Node と NodeList を返し、XmlSlurper は groovy.util.slurpersupport.GPathResult を返します。しかしそれを理解できたら、私が XmlSlurper の実装の詳細について触れたことを忘れてください。背後で何が起きているかを考えようとしなければ、XmlSlurper によるマジック・ショーを心から楽しむことができます。

リスト 15 は XmlParser と XmlSlurper を並べて比較しています。

リスト 15. XmlParser と XmlSlurper

```
def xml = """
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println langs.attribute("count")
langs.language.each{
  println it.text()
}

langs = new XmlSlurper().parseText(xml)
println langs.@count
langs.language.each{
  println it
}
```

XmlSlurper にはメソッド呼び出しの概念がまったくないことに注目してください。langs.attribute("count") を呼び出す代わりに、langs.@count を呼び出します。@ 記号は XPath から借用したものですが、この記号を使うことによって、attribute() メソッドを呼び出す代わりに、まるで属性を直接操作するかのように処理を行うことができます。it.text() を呼び出すのではなく、単純に it を呼び出します。これは要素の内容を直接操作したい、ということが前提です。

実際の XmlSlurper

langs や language にとどまらず、実際に XmlSlurper が使われている例を見てみましょう。Yahoo! は現在の気象状況を郵便番号ごとに RSS フィードで提供しています。もちろん、RSS は XML の特

別な方言です。Web ブラウザーに `http://weather.yahooapis.com/forecastrss?p=80020` と入力してみてください。この郵便番号はコロラド州 Broomfield のものですが、これを皆さんの住所の郵便番号に変えてみてください。その結果の RSS フィードを単純化したものがリスト 16 です。

リスト 16. 現在の気象状況を示す Yahoo! の RSS フィード

```
<rss version="2.0"
     xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0"
     xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
  <channel>
    <title>Yahoo! Weather - Broomfield, CO</title>
    <yweather:location city="Broomfield" region="CO"   country="US"/>
    <yweather:astronomy sunrise="6:36 am"   sunset="5:50 pm"/>

    <item>
      <title>Conditions for Broomfield, CO at 7:47 am MST</title>
      <pubDate>Fri, 27 Feb 2009 7:47 am MST</pubDate>
      <yweather:condition text="Partly Cloudy"
                          code="30"  temp="25"
                          date="Fri, 27 Feb 2009 7:47 am MST" />
    </item>
  </channel>
</rss>
```

まず、この RSS をプログラムで処理する必要があります。そのためには weather.groovy という名前のファイルを作成し、リスト 17 のコードを追加します。

リスト 17. プログラムで RSS を取得する

```
def baseUrl = "http://weather.yahooapis.com/forecastrss"

if(args){
  def zip = args[0]
  def url = baseUrl + "?p=" + zip
  def xml = url.toURL().text
  println xml
}else{
  println "USAGE: weather zipcode"
}
```

コマンドラインで `groovy weather 80020` と入力し、RSS のソースを見られることを確認します。

このスクリプトで最も重要な部分は `url.toURL().text` です。`url` 変数は整形式の `String` です。すべての `String` には Groovy によって `toURL()` メソッドが追加されており、このメソッドによって `String` が `java.net.URL` に変換されます。すると、すべての `URL` には Groovy によって `getText()` メソッドが追加され、このメソッドが `HTTP` の `GET` リクエストを実行し、結果を `String` として返します。

これで `xml` 変数の中に RSS を保存できたので、`XmlSlurper` を少し混合し、興味深い部分のみをリスト 18 に示します。

リスト 18. `XmlSlurper` を使って RSS を構文解析する

```
def baseUrl = "http://weather.yahooapis.com/forecastrss"

if(args){
```

```
  def zip = args[0]
  def url = baseUrl + "?p=" + zip
  def xml = url.toURL().text

  def rss = new XmlSlurper().parseText(xml)
  println rss.channel.title
  println "Sunrise: ${rss.channel.astronomy.@sunrise}"
  println "Sunset: ${rss.channel.astronomy.@sunset}"
  println "Currently:"
  println "\t" + rss.channel.item.condition.@date
  println "\t" + rss.channel.item.condition.@temp
  println "\t" + rss.channel.item.condition.@text
}else{
  println "USAGE: weather zipcode"
}

//output:
Yahoo! Weather - Broomfield, CO
Sunrise: 6:36 am
Sunset: 5:50 pm
Currently:
   Fri, 27 Feb 2009 7:47 am MST
   25
   Partly Cloudy
```

`XmlSlurper` によって、いかに自然に XML を扱えるようになるかがわかると思います。`<title>` 要素を直接参照する (`rss.channel.title`) ことで `<title>` 要素を出力しています。単純な `rss.channel.item.condition.@temp` を使って `temp` 属性を抽出しています。これはプログラミングとは思えません。まるで XML を直接扱っているような感じがします。

`XmlSlurper` では名前空間も無視されることに気付いたでしょうか。コンストラクターの中で名前空間の認識機能をオンにすることもできますが、私が実際にそうすることは稀です。何も手を加えないままの `XmlSlurper` を使うことで、とても簡単に XML を扱うことができるのです。

まとめ

今日のような時代にソフトウェア開発者として成功するためには、XML を容易に扱うためのツール・セットが必要です。Groovy の `MarkupBuilder` と `StreamingMarkupBuilder` を使うと、いとも簡単に XML を作成することができます。`XmlParser` を使うと、`Element` の `List` や `Attribute` の `HashMap` を得ることができ、また `XmlSlurper` を使うとコードがほとんど不要となり、まるで XML を直接扱うような気持ちのよい錯覚に陥ります。

強力な XML 処理は Groovy の動的機能を利用しない限り実現することはできません。次回の記事では、Groovy の動的な性質を詳細に掘り下げます。Groovy でのメタプログラミングの実際について、標準的な JDK クラスに追加されたスマートなメソッド (`String.toURL()` や `List.each()` など) から、皆さん自身が追加するカスタム・メソッドに至るまで学びます。では次回まで、皆さんが Groovy の実用的な使い方をたくさん見つけられることを祈っています。

ダウンロード可能なリソース

内容	ファイル名	サイズ
Source code for examples in article	j-pg05199.zip	6KB

関連トピック

- XML 技術の誕生 10 周年に関するプレスリリース「W3C XML 生誕 10 周年 !」の中で、Tim Bray はこの技術の遍在性を詳述しています。
- Scott Davis による最新の書籍、『Groovy Recipes』(Pragmatic Programmers、2008年刊) を読み、Groovy と Grails について学んでください。
- developerWorks の Java technology ゾーンには Java プログラミングのあらゆる側面を網羅した記事が豊富に用意されています。