実用的な Groovy: each を活用する

「驚き最小」の繰り返し処理の手法を採用する

Scott Davis

2009年 4月 14日

概要: 今回の「実用的な Groovy」では、Scott Davis が、繰り返し処理の方法を次から次へと目まぐるしく紹介します。配列に対する繰り返しの処理はもちろん、リスト、ファイル、URL、等々に対しても繰り返しの処理を行います。最も目を引く点は、こうしたさまざまな集合をウォークスルーするために、Groovy には一貫したメカニズムが用意されているということです。

このシリーズの他の記事を見る

繰り返し処理はプログラミングの基本です。プログラマーは常に `List`、`File`、JDBC の `ResultSet` などを扱う必要があり、これらの中にある項目を 1 つずつウォークスルーする必要があります。ほとんどすべての場合において、Java 言語には必要なステップ操作のための方法が用意されています。しかし腹立たしい現実として、Java 言語には標準的な繰り返し処理の方法というものが用意されていません。Groovy での繰り返し処理の方法は極めて現実的であり、この点で Groovy プログラミングは Java プログラミングとは大きく異なっています。この記事では一連のコード・サンプル (すべて「ダウンロード」セクションで入手可能です) をとおして、Groovy に用意された多用途の `each()` メソッドを利用することで、対象ごとに異なる方法を使用する Java 言語での繰り返し処理から解放される方法を学びます。

Java での繰り返し処理の手法

例えばプログラミング言語の一覧、`java.util.List` があるとしましょう。リスト 1 は、「I know each of these (私はこれらの各言語を知っています)」という内容を Java 言語のプログラムで表現しています。

© Copyright IBM Corporation 2009

リスト 1. リストに対する Java での繰り返し処理

```
import java.util.*;

public class ListTest{
  public static void main(String[] args){
    List<String> list = new ArrayList<String>();
    list.add("Java");
    list.add("Groovy");
    list.add("JavaScript");

    for(Iterator<String> i = list.iterator(); i.hasNext();){
      String language = i.next();
      System.out.println("I know " + language);
    }
  }
}
```

ほとんどのコレクション・クラスが持っている `java.lang.Iterable` インターフェースのおかげで、`java.util.Set` に対しても `java.util.Queue` に対しても同じ方法で繰り返し処理をすることができます。

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。

今度は、言語の一覧が `java.util.Map` に保存されているとします。`Map` に対して `Iterator` を使おうとするとコンパイル時に失敗します。`Map` は `Iterable` インターフェースを実装していないからです。幸いなことに `map.keySet()` を呼び出すと `Set` が返され、作業を続けることができます。このわずかな違いのおかげで少し処理が遅くなりますが、処理ができないわけではありません。ただ、`List`、`Set`、`Queue`、`Map` はすべて同じ `java.util` パッケージの中にありますが、`List`、`Set`、`Queue` は `Iterable` を実装しており、`Map` は実装していないことを覚えておく必要があります。

今度は、言語の一覧が `String` 配列の中にあるとしましょう。配列はデータ構造であり、クラスではありません。`String` に対して `.iterator()` を呼び出すことはできないので、ほんの少しだけ異なる繰り返し処理の方法を使わざるを得ません。この場合も、面倒ではありますが、処理ができないわけではありません (リスト 2)。

リスト 2. 配列に対する Java での繰り返し処理

```
public class ArrayTest{
  public static void main(String[] args){
    String[] list = {"Java", "Groovy", "JavaScript"};

    for(int i = 0; i < list.length; i++){
      String language = list[i];
      System.out.println("I know " + language);
    }
  }
}
```

しかし待ってください。Java 5 で導入された for-each 構文はどうなのでしょう (「参考文献」を参照)？for-each 構文は `Iterable` を実装する任意のクラスに有効であり、配列にも有効です (リスト 3)。

リスト 3. Java 言語での for-each による繰り返し処理

```
import java.util.*;

public class MixedTest{
  public static void main(String[] args){
    List<String> list = new ArrayList<String>();
    list.add("Java");
    list.add("Groovy");
    list.add("JavaScript");

    for(String language: list){
      System.out.println("I know " + language);
    }

    String[] list2 = {"Java", "Groovy", "JavaScript"};
    for(String language: list2){
      System.out.println("I know " + language);
    }
  }
}
```

これで、配列と (`Map` を除く) コレクションに対して同じ方法で繰り返し処理をすることができます。しかし言語の一覧が `java.io.File` に保存されているとしたらどうでしょう。JDBC の `ResultSet` の場合はどうでしょう。XML 文書の場合はどうでしょう。`java.util.StringTokenizer` の場合はどうでしょう。それぞれの場合に対して、少しずつ異なる繰り返し処理の方法を使わざるを得ません。これは意図的な設計の結果ではなく、異なる API が異なる時期に異なる開発者によって開発されたためなのです。しかし実際には、プログラマーは 5、6 種類ある Java の繰り返し処理の手法を理解しておく必要があり、そしてもっと重要な点として、そうした方法を使用する特別なケースを理解しておかなければなりません。

Eric S. Raymond は彼の著書である『The Art of Unix Programming』(「参考文献」を参照) の中で、「The Rule of Least Surprise (驚き最小の原則)」について書いています。彼によれば、「使いやすいインターフェースを設計するためには、可能であれば、完全に新しいインターフェース・モデルを設計しないことがベストです。新しいものがあると敷居が高くなり、そのインターフェースを学ぶ負担をユーザーに負わせることになります。そのため、新しいものを最小限にとどめる必要があります。」Groovy は繰り返し処理に関して Raymond の助言に従っています。つまり Groovy では、ほとんどすべての構造のウォークスルーに関して、`each()` を覚えておくだけでよいのです。

Groovy での List に対する繰り返し処理

まず、リスト 3 の `List` の例を Groovy にリファクタリングします。`List` を `for` ループにしてしまうのではなく (そうするとあまりオブジェクト指向に見えなくなると思いませんか)、単純にリストに対して `each()` メソッドを直接呼び出し、それをクロージャーの中に渡します。

listTest.groovy という名前のファイルを作成し、リスト 4 のコードを追加します。

リスト 4. List に対する Groovy での繰り返し処理

```
def list = ["Java", "Groovy", "JavaScript"]
list.each{language->
  println language
}
```

リスト 4 の最初の行は `java.util.ArrayList` を作成するための Groovy のショートカット構文です。このスクリプトに `println list.class` を追加すると、このクラスが作成されているのを確認することができます。次に、単純にリストに対して `each()` を呼び出し、クロージャーの本体の中にある `language` 変数を出力します。そしてクロージャーの先頭で `language->` ステートメントを使って `language` 変数に名前を付けています。変数名を指定しないと、Groovy はデフォルトで、`it` という変数名を付けます。コマンド・プロンプトで `groovy listTest` と入力すると listTest.groovy が実行されます。

リスト 5 の 2 種類のコードは、リスト 4 のコードを短く、そしてさらに短くしたものです。

リスト 5. Groovy の `it` 変数を使った繰り返し処理

```
// shorter, using the default it variable
def list = ["Java", "Groovy", "JavaScript"]
list.each{ println it }

// shorter still, using an anonymous list
["Java", "Groovy", "JavaScript"].each{ println it }
```

Groovy では配列と `List` のどちらにも `each()` メソッドを使うことができます。`ArrayList` を `String` 配列に変更するためには、その行の最後に `as String[]` を追加します (リスト 6)。

リスト 6. Groovy での配列に対する繰り返し処理

```
def list = ["Java", "Groovy", "JavaScript"] as String[]
list.each{println it}
```

Groovy では何に対しても `each()` メソッドを使うことができ、またショートカットのゲッター構文 (`getClass()` と `class` は同じ呼び出しです) を使えることから、簡潔であると同時に表現力に富んだコードを作成することができます。例えば、あるクラスの public メソッドをすべて、リフレクションを使って表示したいとします。リスト 7 はその一例です。

リスト 7. Groovy のリフレクション

```
def s = "Hello World"
println s
println s.class
s.class.methods.each{println it}

//output:
$ groovy reflectionTest.groovy
Hello World
class java.lang.String
public int java.lang.String.hashCode()
public volatile int java.lang.String.compareTo(java.lang.Object)
public int java.lang.String.compareTo(java.lang.String)
public boolean java.lang.String.equals(java.lang.Object)
...
```

このスクリプトの最後の行では `getClass()` メソッドを呼び出しています。`java.lang.Class` には配列を返す `getMethods()` メソッドが用意されています。これらをすべて連結し、作成された `Method` の配列に対して `each()` を呼び出すと、1 行のコードで非常に多くのことを実現することができます。

`each()` メソッドは多用途であり、Java の for-each 構造とは違って使い道が `List` や配列に限定されているわけではありません。Java 言語の場合、ここまでの説明で話は終わりですが、Groovy の場合は単なるウォーミングアップにすぎません。

Map に対する繰り返し処理

先ほど説明したように、Java 言語では `Map` に対して直接繰り返し処理をすることはできません。しかし Groovy では問題なく行うことができます (リスト 8)。

リスト 8. map に対する Groovy での繰り返し処理

```
def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]
map.each{ println it }
```

名前と値のペアを取得するためには、暗黙的な `getKey()` メソッドと `getValue()` メソッドを使用するか、あるいはクロージャーの先頭で変数に明示的に名前を付けます (リスト 9)。

リスト 9. map からキーと値を取得する

```
def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]
map.each{
  println it.key
  println it.value
}

map.each{k,v->
  println k
  println v
}
```

これを見るとわかるように、`Map` に対する繰り返し処理は、他の任意のコレクションに対する繰り返し処理の場合と同じようにごく自然です。

次の繰り返し処理の例に移る前に、もう 1 つ、Groovy での `Map` に関する構文糖について知っておく必要があります。Java 言語の場合のように `map.get("Java")` を呼び出すのではなく、呼び出しを短縮して `map.Java` にすることができるのです (リスト 10)。

リスト 10. map の値を取得する

```
def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]

//identical results
println map.get("Java")
println map.Java
```

`Map` に対する Groovy のショートカット構文がスマートであることは否定できませんが、この構文は `Map` に対してリフレクションを使用する場合に早合点の元にもなりがちなのです。`list.class` を呼び出すと `java.util.ArrayList` が返されますが、`map.class` を呼び出すと `null` が返されます。これは、map 要素を取得するためのショートカットによって通常のゲッターの呼び出しがオーバーライドされるためです。`Map` の中のどの要素も `class` のキーを持っていません。そのため、この呼び出しによって `null` が返されるのは適切なのです (リスト 11)。

リスト 11. Groovy での map と `null`

```
def list = ["Java", "Groovy", "JavaScript"]
println list.class
// java.util.ArrayList

def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]
println map.class
// null

map.class = "I am a map element"
println map.class
// I am a map element

println map.getClass()
// class java.util.LinkedHashMap
```

これは「驚き最小の原則」に Groovy が違反してしまう稀な例ですが、map から要素を取得する場合の方がリフレクションを使用する場合よりもはるかに一般的なため、この例外は我慢できる範囲のうちと言えます。

ストリングに対する繰り返し処理

each() メソッドに慣れてくると、ありとあらゆる興味深い場所に `each()` メソッドを活用することができます。例えば、ある `String` に対して 1 文字ずつ繰り返しの処理を行いたいとします。この場合にも `each()` メソッドを使用することができます (リスト 12)。

リスト 12. `#####`に対する繰り返し処理

```
def name = "Jane Smith"
name.each{letter->
  println letter
}
```

この手法によって、あらゆる可能性が生まれます。例えばすべての空白をアンダーバーで置き換えることができます (リスト 13)。

リスト 13. 空白をアンダーバーで置き換える

```
def name = "Jane Smith"
println "replace spaces"
name.each{
  if(it == " "){
    print "_"
  }else{
    print it
  }
}

// output
Jane_Smith
```

もちろん、1 文字のみを置き換える場合には、Groovy にはもっと簡潔な replace メソッドが用意されています。リスト 13 のすべてのコードを 1 行に統合し、`"Jane Smith".replace(" ", "_")` とすることもできます。しかし複雑な `String` 操作の場合には、`each()` メソッドが完璧なソリューションです。

Range に対する繰り返し処理

Groovy には、実質的に繰り返しの処理に使用できるネイティブの `Range` 型が用意されています。連続する 2 つのドットで区切られたもの (例えば `1..10` など) はすべて `Range` です。リスト 14 はその一例です。

リスト 14. Range に対する繰り返し処理

```
def range = 5..10
range.each{
  println it
}

//output:
5
6
7
8
9
10
```

`Range` は単純な `Integer` にしか使えないわけではありません。リスト 15 のコードを考えてみてください。このコードは、ある範囲 (`Range`) の日付 (`Date`) に対して繰り返し処理を行います。

リスト 15. `Date` に対する繰り返し処理

```
def today = new Date()
def nextWeek = today + 7
(today..nextWeek).each{
  println it
}

//output:
Thu Mar 12 04:49:35 MDT 2009
Fri Mar 13 04:49:35 MDT 2009
Sat Mar 14 04:49:35 MDT 2009
Sun Mar 15 04:49:35 MDT 2009
Mon Mar 16 04:49:35 MDT 2009
Tue Mar 17 04:49:35 MDT 2009
Wed Mar 18 04:49:35 MDT 2009
Thu Mar 19 04:49:35 MDT 2009
```

こうしたことからわかるように、まさに `each()` メソッドを使いたい場所で `each()` メソッドが登場します。Java 言語にはネイティブの `Range` 型がありませんが、同様の概念は `enum` の形で用意されています。そしてもちろん、その場合にも `each()` メソッドを使用することができます。

列挙に対する繰り返し処理

Java の `enum` は特定の順序で保存される一連の値です。リスト 16 は、(`Range` 演算子の場合とまったく同じように) いかに自然な形で `each()` メソッドを `enum` に使用できるかを示しています。

リスト 16. `enum` に対する繰り返し処理

```
enum DAY{
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}

DAY.each{
  println it
}

(DAY.MONDAY..DAY.FRIDAY).each{
  println it
}
```

Groovy では、`each()` というメソッドの名前は、その役割を表現するには十分でない場合があります。以下のいくつかの例では、使用されるコンテキストに特有の方法で `each()` メソッドが修飾されています。その好例が Groovy の `eachRow()` メソッドです。

SQL に対する繰り返し処理

リレーショナル・データベースのテーブルを扱う場合、「テーブルの中の各行に対して、その行に特有の何かを行う必要がある」と言うことがよくあります。これを、これまでの例と比べてみてください。皆さんはおそらく「リストの中にある各言語に対して、その言語に特有の何かをする必要がある」と言いたくなるのではないでしょうか。こうした場合のために、`groovy.sql.Sql` オブジェクトには `eachRow()` メソッドが用意されています (リスト 17)。

リスト 17. `ResultSet` に対する繰り返し処理

```
import groovy.sql.*

def sql = Sql.newInstance(
   "jdbc:derby://localhost:1527/MyDbTest;create=true",
   "username",
   "password",
   "org.apache.derby.jdbc.ClientDriver")

println("grab a specific field")
sql.eachRow("select name from languages"){ row ->
    println row.name
}

println("grab all fields")
sql.eachRow("select * from languages"){ row ->
    println("Name: ${row.name}")
    println("Version: ${row.version}")
    println("URL: ${row.url}\n")
}
```

このスクリプトの最初の行では、JDBC 接続ストリング、ユーザー名、パスワード、JDBC ドライバー・クラスを設定し、新しい `Sql` オブジェクトをインスタンス化しています。このインスタンスからは `eachRow()` メソッドを呼び出すことができ、メソッドの引数として SQL の `select` 文を渡すことができます。クロージャーの中では、あたかも実際に `getName()` メソッド、`getVersion()` メソッド、`getUrl()` メソッドがあるかのように、列の名前 (`name`、`version`、`url`) を参照することができます。

これは Java で同じことを行う場合よりも、はるかにすっきりとしています。Java の場合には `DriverManager`、`Connection`、`Statement`、`JDBCResultSet` を別々に用意し、それらをすべて、ネストされた無数の `try`/`catch`/`finally` ブロックの中でクリーンアップしなければなりません。

`Sql` オブジェクトの場合には、`each()` でも `eachRow()` でもメソッド名としては適切かもしれません。しかし以下に示すいくつかの例では、`each()` という名前ではその役割を十分に表現できないことに皆さんも同意するはずです。

ファイルに対する繰り返し処理

私は `java.io.File` に対して Java コードで直接 1 行ずつ繰り返し処理を行うことを考えたくはありません。ネストされたすべての `BufferedReader` と `FileReader` に対する処理を終え、さらにはプロセスの最後で例外処理をすべて終える頃になると、そもそも何の作業を行っていたのかを忘れてしまうかもしれません。

リスト 18 は Java 言語でのプロセスの全体を示しています。

リスト 18. ファイルに対する繰り返し処理を Java で行う場合

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class WalkFile {
   public static void main(String[] args) {
      BufferedReader br = null;
      try {
         br = new BufferedReader(new FileReader("languages.txt"));
         String line = null;
         while((line = br.readLine()) != null) {
            System.out.println("I know " + line);
         }
      }
      catch(FileNotFoundException e) {
         e.printStackTrace();
      }
      catch(IOException e) {
         e.printStackTrace();
      }
      finally {
         if(br != null) {
            try {
               br.close();
            }
            catch(IOException e) {
               e.printStackTrace();
            }
         }
      }
```

```
    }
}
```

リスト 19 は同様のプロセスを Groovy で行う場合を示しています。

リスト 19. ファイルに対する繰り返し処理を Groovy で行う場合

```
def f = new File("languages.txt")
f.eachLine{language->
  println "I know ${language}"
}
```

こうした場合に、Groovy による簡潔さが非常に際立ちます。私がなぜ Groovy を「Java プログラマーのための DSL としての Groovy」と呼ぶのか、この例を見れば理解していただけると思います。

ここで、同じ `java.io.File` クラスを Groovy と Java 言語の両方で処理していることに注意してください。ファイルが存在しない場合には、Groovy コードは Java コードと同じように `FileNotFoundException` をスローします。Java との違いは、Groovy にはチェック例外がない点です。`eachLine()` 構造が `try/catch/finally` ブロックの中にラップされていますが、これは私がそうしているだけで、言語による要件ではありません。単純なコマンドライン・スクリプトの場合、私はリスト 19 の簡潔なコードを使用しています。しかし、アプリケーション・サーバーの中で実行する際にこの繰り返し処理を行うのであれば、とても例外をキャッチせずに放っておけるほどにはなれません。その場合には Java のバージョンと同じように `try/catch` ブロックの中に `eachLine()` ブロックをラップすることになるでしょう。

`File` クラスには `each()` メソッドのバリエーションがいくつか用意されています。その 1 つが `splitEachLine(String separator, Closure closure)` です。これは、ファイルに対して 1 行ずつ繰り返し処理を行えるだけではなく、ファイルをトークンに分割することも同時にできるということです。リスト 20 はその一例です。

リスト 20. ファイルの各行を分割する

```
// languages.txt
// notice the space between the language and the version
Java 1.5
Groovy 1.6
JavaScript 1.x

// splitTest.groovy
def f = new File("languages.txt")
f.splitEachLine(" "){words->
  words.each{ println it }
}

// output
Java
1.5
Groovy
1.6
JavaScript
1.x
```

バイナリー・ファイルを扱う場合には、Groovy には `eachByte()` メソッドも用意されています。

もちろん、Java 言語での `File` は必ずしもファイルとは限らず、ディレクトリーの場合もあります。Groovy にはサブディレクトリーを扱うための `each()` のバリエーションもいくつか用意されています。

ディレクトリーに対する繰り返し処理

Groovy を使用するとファイルシステムへのアクセスや操作を容易に行えるため、Groovy を簡単にシェル・スクリプト (つまりバッチ・スクリプト) の置き換えとして使うことができます。カレント・ディレクトリーのディレクトリー・リストを取得するためにはリスト 21 のようにします。

リスト 21. ディレクトリーに対する繰り返し処理

```
def dir = new File(".")
dir.eachFile{file->
  println file
}
```

`eachFile()` メソッドによってファイルとサブディレクトリーの両方が返されます。Java 言語の `isFile()` メソッドと `isDirectory()` メソッドを使用すると、もっと高度なことを行うことができます。リスト 22 はその一例です。

リスト 22. ファイルとディレクトリーとを分離する

```
def dir = new File(".")
dir.eachFile{file->
  if(file.isFile()){
    println "FILE: ${file}"
  }else if(file.isDirectory()){
    println "DIR:  ${file}"
  }else{
    println "Uh, I'm not sure what it is..."
  }
}
```

どちらの Java メソッドも単純な `boolean` 値を返すため、Java の三項演算子を使ってこのコードを簡潔にすることができます (リスト 23)。

リスト 23. 三項演算子

```
def dir = new File(".")
dir.eachFile{file->
  println file.isDirectory() ? "DIR:  ${file}" : "FILE: ${file}"
}
```

ディレクトリーのみに関心がある場合には、`eachFile()` の代わりに `eachDir()` を使います。また `eachDirMatch()` メソッドと `eachDirRecurse()` メソッドもあります。

これを見るとわかるように、`File` を扱う場合には単純な `each()` メソッドでは、その役割を表現するには十分ではありません。`File` の場合にも典型的な `each()` メソッドのセマンティクスは保たれますが、メソッド名は高度な機能を持つことを追加情報として示すために、より説明的な名前にします。

URL に対する繰り返し処理

`File` に対する繰り返し処理の方法を理解できると、同じ原理を使って HTTP リクエストへのレスポンスに対して繰り返し処理を行うことができます。Groovy には、`java.net.URL` に対する便利な (そしておなじみの) `eachLine()` メソッドが用意されています。

例えば、リスト 24 は ibm.com のホームページの HTML を 1 行ずつウォークスルーします。

リスト 24. URL に対する繰り返し処理

```
def url = new URL("http://www.ibm.com")
url.eachLine{line->
  println line
}
```

もちろん、これのみを行いたい場合のために、Groovy には 1 行のソリューションが用意されており、すべての `String` に `toURL()` メソッドを追加することができます (`"http://www.ibm.com".toURL().eachLine{ println it }`)。

しかし、HTTP レスポンスを使って何かもっと有用なことをしたい場合にはどうするのがよいのでしょう。特に、RESTful Web サービスに対してリクエストを送信する場合、レスポンスに XML が含まれており、この XML を構文解析したい場合にはどうするのがよいのでしょう。この場合にも `each()` メソッドが役に立ちます。

XML に対する繰り返し処理

先ほど、ファイルと URL に対して `eachLine()` メソッドを使う方法を説明しました。しかし XML の場合は少し問題が異なります。おそらく皆さんは、XML 文書を 1 行ずつウォークスルーすることにはあまり関心がなく、要素ごとにウォークスルーすることに関心があるはずです。

例えば言語の一覧が languages.xml という名前のファイルに保存されているとします (リスト 25)。

リスト 25. languages.xml ファイル

```
<langs>
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
```

Groovy には `each()` メソッドが用意されており、このメソッドを使うことができます。しかしそのためには少し工夫が必要です。Groovy のネイティブ・クラスである `XmlSlurper` を使って XML を構文解析する場合、`each()` を使って要素に対して繰り返し処理を行うことができます。リスト 26 はその一例です。

リスト 26. XML に対する繰り返し処理

```
def langs = new XmlSlurper().parse("languages.xml")
langs.language.each{
  println it
}

//output
Java
Groovy
JavaScript
```

`langs.language.each` ステートメントによって、`<langs>` の下にある `<language>` という名前の要素をすべて取得することができます。`<langs>` の下に `<format>` 要素や `<server>` 要素がある場合にも、それらの要素が `each()` メソッドの出力の中に現れることはありません。

これでも目を引かれないという人のために、この XML がファイルシステムの中のファイルではなく、RESTful Web サービスから得られるものとしてみましょう。その場合にも、ファイルへのパスを URL で置き換えるだけでよく、コードの他の部分はそのままです (リスト 27)。

リスト 27. Web サービスへの呼び出しによって得られる XML に対して繰り返し処理を行う

```
def langs = new XmlSlurper().parse("http://somewhere.com/languages")
langs.language.each{
  println it
}
```

これは `each()` メソッドを非常に簡潔に、うまく使っていると言えないでしょうか。

まとめ

今回は `each()` の使い方を説明しましたが、`each()` の最も良いところは、Groovy 全体にわたって非常にさまざまな使い方がとても簡単にできるところです。`each()` メソッドを理解できてしまうと、Groovy での繰り返し処理には新しいものがほとんどありません。そして Raymond が言うように、まさにそこがポイントなのです。`List` に対する繰り返し処理の方法を理解できると、配列、`Map`、`String`、`Range`、`enum`、SQL の `ResultSet`、`File`、ディレクトリー、`URL`、さらには XML 文書の要素に至るまで、どの場合の繰り返し処理の方法も瞬時に理解することができます。

この記事の最後の例では `XmlSlurper` を使って XML を構文解析する方法について簡単に触れました。次回の記事ではこの点に戻り、Groovy では XML の構文解析がいかに容易かを説明します。そこでは `XmlParser` と `XmlSlurper` の両方の実際を見ながら、そもそもなぜ Groovy には似て非なる 2 つのクラスが XML の構文解析用に用意されているのかを学びます。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 this article	j-pg04149.zip	17KB

関連トピック

- 『The Art of Unix Programming』(Eric Raymond 著、2003年 Addison-Wesley 刊) の「Applying the Rule of Least Surprise」の章では、最も驚きの少ないことを行うための道を解説しています。
- 「for/inでJava 5.0 のループを拡張」(Brett McLaughlin 著、developerWorks、2004年11月) を読んでください。Java 5.0 で便利な機能として導入された for/in (for-each としても知られています) を解説しています。
- Scott Davis による関連のシリーズとして、Web 開発のための Groovy ベースのプラットフォームに焦点を絞った「Grails をマスターする」を読んでください。
- Scott Davis による最新の本、『Groovy Recipes』(Pragmatic Programmers、2008年刊) を読み、Groovy と Grails について学んでください。
- Groovy のメーリング・リストをブラウズ、検索、または購読してください。
- 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。

© Copyright IBM Corporation 2009
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)