実用的なGroovy: Groovyテンプレートを使ったMVCプログラミング

Groovyのテンプレート・エンジン・フレームワークでレポート・ビューを単純化する

Andrew Glover

2005年 2月 15日

ビューはMVCプログラミングに統合された一部であり、エンタープライズ・アプリケーション開発では、どこにでもあるコンポーネントです。実用的なGroovyの今回は、Groovyのテンプレート・エンジン・フレームワークを使用してビュー・プログラミングを単純化し、長期間に渡ってコードを維持管理しやすくする方法について、Andrew Gloverが解説します。

私は実用的なGroovyシリーズの最近の記事の中で、レポート・アプリケーションを構築する上で、Groovyが素晴らしいツールであることを説明してきました。GroovyによるAntスクリプト では、チェックサムをレポートするアプリケーション例を使って、また、GroovyによるJDBCプログラミングでは、データベース統計レポートのアプリケーション例を使って、Groovyの利点を説明しました。これらのレポート例は、どちらもGroovyスクリプト内部で生成され、また、どちらも当然ながら、それぞれに付随して「ビュー」を持っています。

チェックサム・レポートの例では、ビュー・コードは少しばかり、やっかいでした。さらに悪いことに、維持管理も面倒なことが分かります。つまり、ある特定な面のビューを変更しようとすると、スクリプト内部のコードを変更しなくてはなりません。そこで今回は、レポート・ビューを単純化する方法を、Groovyのテンプレート・エンジンと、前回の記事で取り上げたチェックサム・レポートのアプリケーションを使って解説します。

テンプレート・エンジンはXSLTと似ており、テンプレートを定義する任意のフォーマット、つまりXMLやHTML、SQLそしてGroovyコードなど、を生成することができます。JSPと同様、テンプレート・エンジンを使うことによって、ビューを別の実体（例えばJSPやテンプレート・ファイル）に分離することが容易になります。例えば、あるレポートがXMLを出力すべき場合であれば、実行時に置換される値に対するプレースホルダーを含んだXMLテンプレートを作ることができるのです。そうするとテンプレート・エンジンは、そのテンプレートを読み取り、ランタイム値でプレースホルダーにマッピングすることによって、テンプレートを変換します。そのプロセスの結果、XMLフォーマットされた文書が出力されます。

このシリーズについて

どのようなツールであれ、開発作業の中に採り入れるためには、どういう場合に使うべきか、または使うべきではないかをよく知る必要があります。スクリプト言語は非常に強力なツールですが、その強力さは、適切なシナリオで適切な使い方をした場合にのみ発揮されます。実用的なGroovy シリーズはそうした点を念頭に置き、Groovyの実用的な使い方に焦点を絞って、どういう場合に、どのように使うのかを解説して行きます。

Groovyテンプレートに踏み込む前に、Groovyでの`String`型を復習したいと思います。スクリプト言語はどれも、ストリングの扱いを非常に単純にするように、色々な工夫をしていますが、Groovyも例外ではありません。まず、このページの最初または最後にあるCodeアイコンをクリックして（または、ダウンロード セクションを見てください）、この記事で使用するソースをダウンロードしてください。

Stringsは、もうたくさんだ

残念なことに、単純なJava™ コードでのストリングは、非常に限定されています（ただし、Java 5.0アプリケーションでは、なかなか素敵な機能が約束されています）。Groovyは、今日のJava`String` の持つ最悪の制約のうち2つを解決しており、複数行のストリングが書きやすく、またランタイム置換を行いやすくなっています。この記事で扱う幾つか単純な例を見れば、Groovyの`String` 型について、皆さんもすぐに追いつくことができるでしょう。

単純なJavaコードで複数行の`String` を書いたことのある人であれば、例の面倒な`+` を、大量に使う羽目に陥ったと思います。ところが、リスト1を見ればお分かりの通り、Groovyではこうした`+` を使う必要がなく、ずっとスッキリと単純なコードが書けるのです。

リスト1.Groovyでの複数行ストリング

```
String example1 = "This is a multiline
  string which is going to
  cover a few lines then
  end with a period."
```

Groovyはまた、リスト2に示すように、here-docの概念もサポートしています。here-docは、HTMLやXMLなど、フォーマットした`String`を作るために便利な仕組みです。Python風の三重引用符が必要なこと以外、here-doc構文は、通常の`String` 宣言とあまり変わらないことに注意してください。

リスト2.GroovyでのHere-doc

```
itext =
"""
This is another multiline String
that takes up a few lines. Doesn't
do anything different from the previous one.
"""
```

Groovyは、`GString`を使ってランタイム置換を実現します。`GString`とは何か、といぶかる人もいるかもしれませんが、これは皆さんも見たことがあり、実際に使ったこともあるはずです。単純に言えば、`GString`を使うと、bash風の`${}` 構文を使って値を置換できるのです。`GString`が素晴らしいのは、`GString` 型を使っていることさえ気にしなくてよい、という点です。皆さんが通常Javaコードで行うのと同じように、Groovyの中で`String`をコード化すればよいのです。

テンプレート・エンジンについて

テンプレート・エンジンは昔からあり、最近の言語では、どれにでも用意されています。通常のJava言語にはVelocityとFreeMarkerという2つがあり、PythonではCheetah、RubyではERB、Groovyにも独自のテンプレート・エンジンがあります。テンプレート・エンジンについて詳しくは、参考文献を見てください。

リスト3.GroovyでのGString

```
lang = "Groovy"
println "Uncle man, Uncle man, I dig ${lang}."
```

リスト3では、`lang` という変数を作り、その値を「Groovy」と設定しています。ここでは`GString`型の`String`を出力し、単語「dig」の後を、`${lang}`という値で置換するように要求しています。現実に実行すると、「Uncle man, Uncle man, I dig Groovy.」と出力されます。大したものだと思いませんか。

実はランタイム置換は、動的言語では一般的な機能です。そして、ここでもGroovyは一歩先を進んでいます。Groovyの`GString`を使うと、置換された値に対してメソッドを自動呼び出し（autocall）できるため、動的なテキストを構築する上で大きな可能性が生まれてくるのです。例えばリスト4では、`String`型のオブジェクトに対して、必要な値に対するメソッド（この場合では、メソッド`length()`）を呼ぶことができます。

リスト4.GStringによる自動呼び出し

```
lang = "Groovy"
println "I dig any language with ${lang.length()} characters in its name!"
```

リスト4のコードは、「I dig any language with 6 characters in its name!」を出力します。これから先のセクションでは、Groovyによる自動呼び出し機能を使って、テンプレートに高度な機能を持たせる方法を説明します。

Groovyのテンプレート

テンプレートを扱う作業は、本質的に2つに分割できます。第1が、テンプレートを作ることであり、第2が、マッピング・コードを提供することです。Groovyのテンプレート・フレームワークを使ってテンプレートを作るのは、JSPを作るのと非常によく似ています。（これは、JSPにある構文を再利用できるためです。）こうしたテンプレートを作る上での鍵は、ランタイムで置換される変数を定義することです。例えばリスト5では、`GroovyTestCase`を作るためのテンプレートを定義しています。

リスト5.GroovyTestCaseを作るためのテンプレート

```
import groovy.util.GroovyTestCase
class <%=test_suite %> extends GroovyTestCase {
  <% for(tc in test_cases) {
     println "\tvoid ${tc}() { } "
  }%>
}
```

リスト5のテンプレートは、`<%` 構文と`<%=` 構文を使っているため、JSPファイルと非常に似ています。しかし、Groovyの柔軟性を真に生かすとすれば、JSP構文に限定する必要はありません。リスト6に示すように、Groovyの素晴らしい`GString`も、自由に使えるのです。

リスト6.実際のGString

```
<person>
  <name first="${p.fname}" last="${p.lname}"/>
</person>
```

リスト6では、person要素のコレクションを定義するXML文書を表現した、単純なテンプレートを作りました。このテンプレートが想定しているのは、pという名前を持ち、fname とlnameというプロパティーを持つオブジェクトであることが分かるでしょう。

Groovyではテンプレートが簡単に定義できますが、正にそうあるべきなのです。Groovyのテンプレートはロケット工学のように複雑なものではなく、モデルからビューを分離しやすくするための、一つの手段に過ぎないのです。次のステップでは、ランタイム・マッピング・コードを書きます。

ランタイム・マッピング

テンプレートは定義できたので、定義した定数をランタイム値にマッピングするために、このテンプレートを使ってみましょう。Groovyでは毎度のことですが、複雑そうに見えることも、実は非常に簡単なのです。必要なものは、map だけです（mapのキーはテンプレートでの変数名であり、mapのキー値はランタイムでの値となるべきものです）。

例えば、ある単純なテンプレートがfavlangという変数を持っている場合であれば、favlangというキー値を持つmap を定義する必要があります。キー値は、任意のスクリプト言語（この場合ではもちろん、Groovy）です。

リスト7では、この単純なテンプレートを定義しています。またリスト8では、対応するマッピング・コードを示しています。

リスト7.マッピングを示すための単純なテンプレート

```
My favorite dynamic language is ${favlang}
```

リスト8は、5つのことをする単純なクラスを示していますが、そのうちの2つが重要なものです。どれがその2つか、皆さんには分かるでしょうか。

リスト8.単純なテンプレートに対して値をマップする

```
package com.vanward.groovy.tmpl
import groovy.text.Template
import groovy.text.SimpleTemplateEngine
import java.io.File
class SimpleTemplate{
  static void main(args) {
    fle = new File("simple-txt.tmpl")
    binding = ["favlang": "Groovy"]
    engine = new SimpleTemplateEngine()
    template = engine.createTemplate(fle).make(binding)
    println template.toString()
  }
}
```

リスト8に示す単純なテンプレートに対して値をマップするのは、驚くほど容易です。

まず、テンプレート`simple-txt.tmpl`を指す、`File`インスタンスを作ります。

次に、`binding`オブジェクトを作ります。これがつまり、`map`です。テンプレート`favlang`にある値を、`String`Groovyにマップします。これはGroovyで、（あるいはテンプレート・エンジンを持つどんな言語でも）テンプレートを使う上で、最初の重要ステップです。

次に、`SimpleTemplateEngine`のインスタンスを作ります。これは、Groovyでのテンプレート・エンジン・フレームワークを実装した具体例です。次にこのエンジン・インスタンスに対して、テンプレート（`simple-txt.tmpl`)と`binding`オブジェクトを渡します。テンプレートと、その`binding`オブジェクトを一緒にまとめることが、リスト8で2番目に重要なステップであり、テンプレート・エンジンを扱う場合のハイライトに当たる部分です。内部的には、このフレームワークが、`binding`オブジェクトにある値を、対応するテンプレートにある名前と共にマップします。

リスト8での最後のステップは、このプロセスの結果を出力することです。ご覧の通り、`binding`オブジェクトを作り、適切なマッピングを提供することは、（少なくとも、この例では）ごく単純です。次のセクションでは、Groovyのテンプレート・エンジンを、もっと複雑な例で試してみます。

より複雑なテンプレート

リスト9では、リスト6で定義される`person`要素を表す、`Person`クラスを作っています。

リスト9.GroovyでのPersonクラス

```
class Person{
 age
 fname
 lname
 String toString(){
  return "Age: " + age + " First Name: " + fname + " Last Name: " + lname
 }
}
```

リスト10は、上記で定義した`Person`クラスのインスタンスをマップするマッピング・コードです。

リスト10.テンプレートでPersonクラスをマップする

```
import java.io.File
import groovy.text.Template
import groovy.text.SimpleTemplateEngine
class TemplatePerson{
  static void main(args) {
    pers1 = new Person(age:12, fname:"Sam", lname:"Covery")
    fle = new File("person_report.tmpl")
    binding = ["p":pers1]
    engine = new SimpleTemplateEngine()
    template = engine.createTemplate(fle).make(binding)
    println template.toString()
  }
}
```

上記のコードは、どこかで見たように思えませんか？ 実際これは、`pers1`インスタンスを作る一行が追加されている以外、リスト8とほとんど同じなのです。では、リスト6のテンプレート

を、再度ちょっと見てください。テンプレートが、どのように`fname`プロパティーと`lname`プロパティーを参照しているか、分かるでしょうか。ここでは、プロパティー`fname`が「Sam」に設定され、プロパティー`lname`が「Covery」に設定された`Person`のインスタンスを作っています。

リスト10のコードが実行されると、その出力は`person`要素を定義するXMLになります。これをリスト11に示します。

リスト11.Personテンプレートの出力

```
<person>
  <name first="Sam" last="Covery"/>
</person>
```

リストをマップする

リスト5では、`GroovyTestCase`に対するテンプレートを定義しました。このテンプレートを再度見てもらえば、この定義が、コレクションに対して繰り返すための論理を持っていることに気がつくでしょう。リスト12にも似たようなコードがありますが、このコードは、テスト・ケースの`list`をマップするための論理を含んでいます。

リスト12.テスト・ケースのリストをマップする

```
fle = new File("unit_test.tmpl")
coll = ["testBinding", "testToString", "testAdd"]
binding = ["test_suite":"TemplateTest", "test_cases":coll]
engine = new SimpleTemplateEngine()
template = engine.createTemplate(fle).make(binding)
println template.toString()
```

リスト5を見ると、このテンプレートは、「test_cases」という名前の`list`を想定していることが分かります。「test_cases」は、リスト12で、3つの要素を含む`coll`として定義したものです。ここでは単純に、`coll`をバインディング・オブジェクトの「test_cases」キーに設定しており、これでコードは完成です。

ここまでで、Groovyのテンプレートが使いやすいことが、はっきりと分かったと思います。またGroovyのテンプレートによって、MVCパターンがどこでも自在に使えるようになりますが、もっと重要なこととして、ビューを具体化することによって、MVCコードへとリファクターできるようになるのです。次のセクションでは、ここで学んだことを基に、前回の記事の例をリファクターする方法を説明しましょう。

テンプレートでリファクターする

私はGroovyによるAntスクリプトに関するコラムの中で、クラス・ファイルのチェックサム・レポートを生成する単純なユーティリティーを書きました。皆さんも覚えているかもしれませんが、そこでは`println`を使って、不器用にXMLをコード化しました。このコードはリスト13で示すように、私でも認めざるを得ないほど見苦しいものです。

リスト13.うさん臭いコード

```
nfile.withPrintWriter{ pwriter |
  pwriter.println("<md5report>")
    for(f in scanner){
       f.eachLine{ line |
         pwriter.println("<md5 class='" + f.path + "' value='" + line + "'/>")
       }
    }
  pwriter.println("</md5report>")
}
```

記憶を呼び起こしてもらうために言うと、リスト13のコードは`PrintWriter`を使って、あるデータをファイル（`nfile`インスタンス）へと書き出します。レポート（XML）のビュー・コンポーネントを、どのように`println`内部にハードコード化したかに注意してください。この手法の問題点は、柔軟性に欠けることです。例えば、しばらく後に変更を加える必要が出てきた場合には、Groovyスクリプトの論理の中に入り込んで変更を加えなければなりません。もっと悪い場合として、プログラマー以外の人が変更を加えようとする場合を考えてみてください。このGroovyコードでは面倒すぎるのです。

関係者の誰にとっても、テンプレートを変更する方がずっと自然です。ですからスクリプトのビュー部分をテンプレートに移動すれば、維持管理が容易になります。そこで、テンプレートを変更することにしましょう。

テンプレートを定義する

まず、テンプレートを定義することから始めます。このテンプレートは、クラスのコレクションに対して繰り返しを行う論理を持ち、出力として意図されているものと、よく似たものになります。

リスト14.以前のコードをテンプレートへとリファクターする

```
<md5report><% for(clzz in clazzes) {println "<md5 class=\"${clzz.name}\" value=\"${clzz.value}\"/>"}%></
md5report>
```

リスト14が定義するテンプレートは、コレクションに対して繰り返しを行う論理を含むという点で、`GroovyTestCase`に対するテンプレートと似ています。JSP構文と`GString`が混在している点にも注意してください。

マッピング・コードを書く

テンプレートが定義できたので、次のステップは、ランタイムのマッピング・コードを書くことです。ファイル書き込みを行う古い論理を、`ChecksumClass`オブジェクトのコレクションを構築するコードで置き換え、こうしたオブジェクトを`binding`オブジェクトの中に置かなければなりません。

リスト15.Groovyで定義される`ChecksumClass`

Listing 15. CheckSumClass defined in Groovy

```
class CheckSumClass{
  name
  value
  String toString(){
   return "name " + name + " value " + value
  }
}
```

Groovyでは、クラス定義が非常に容易なことが分かると思います。

コレクションを作る

次に、以前はファイルへの書き出しを行っていたコード部分を、新しい`ChecksumClass`でリストに中身を入れる論理を使って、リファクターする必要があります。これをリスト16に示します。

リスト16.ChecksumClassesのコレクションを作る、リファクターされたコード

```
clssez = []
for(f in scanner){
  f.eachLine{ line |
   iname = formatClassName(bsedir, f.path)
   clssez << new CheckSumClass(name:iname, value:line)
  }
}
```

リスト16を見ると、Ruby風の構文を使って`list`にオブジェクトを追加することがいかに簡単か、分かるでしょう。実際、非常にGroovy（カッコ良い）なのです。まず、`[]`構文で`list`を作ります。次に簡略の`for`ループ概念と、続いてクロージャー（closure）を持つイテレーターを使っています。クロージャーは各`line`（この場合では、チェックサム値）から、新たに定義した`CheckSumClass`のインスタンスを（Groovyの自動生成コンストラクターを使って）作り、これらをコレクションに追加します。悪くありません。それに、書くのが楽しくなります。

テンプレート・マッピングを追加する

最後に必要なのは、テンプレート・エンジン特有のコードを追加することです。このコードはランタイム・マッピングを実行し、オリジナル・ファイルに対応してフォーマットされた、テンプレートを書きます。これをリスト17に示します。

リスト17.テンプレート・マッピングでリファクターする

```
fle = new File("report.tmpl")
binding = ["clazzes": clzzez]
engine = new SimpleTemplateEngine()
template = engine.createTemplate(fle).make(binding)
nfile.withPrintWriter{ pwriter |
  pwriter.println template.toString()
}
```

ここまで来れば、リスト17も皆さんに全く違和感のないものとなっているはずです。リスト16からの`list`を、`binding`オブジェクトの中に置きます。次に`nfile`オブジェクトをとりあげ、それに対応した出力を（リスト14にある、マップされたテンプレートから）書くのです。

これらの全てをリスト18としてまとめる前に、リスト13に示した、最初のうさん臭いコードをもう一度見てください。それから、下記に示した、新たにリファクターされたコードと比べてみてください。

リスト18.うさん臭さがずっと少ないコードが完成です！

```
/**
 *
 */
buildReport(bsedir){
 ant = new AntBuilder()
 scanner = ant.fileScanner {
   fileset(dir:bsedir) {
     include(name:"**/*class.md5.txt")
   }
 }
 rdir = bsedir + File.separator + "xml" + File.separator
 file = new File(rdir)
 if(!file.exists()){
   ant.mkdir(dir:rdir)
 }
 nfile = new File(rdir + File.separator + "checksum.xml")
 clssez = []
 for(f in scanner){
   f.eachLine{ line |
    iname = formatClassName(bsedir, f.path)
    clssez << new CheckSumClass(name:iname, value:line)
   }
 }
 fle = new File("report.tmpl")
 binding = ["clazzes": clzzez]
 engine = new SimpleTemplateEngine()
 template = engine.createTemplate(fle).make(binding)
 nfile.withPrintWriter{ pwriter |
   pwriter.println template.toString()
 }
}
```

私は美しいコードを書くと宣言したわけではありませんが、このコードは確かに、前回のコードほど見苦しいものではありません。よく考えてみれば分かると思いますが、私がしたことというのは、うさん臭い`println`を、Groovyを使った、ずっと素敵なテンプレート・コードで置き換えたことだけです。（熟達したリファクター派の人達であれば、抽出メソッド（Extract Method）を適用して、もっとコードを洗練できると言うことでしょう。）

まとめ

今回のレッスンでは、皆さんがGroovyに対して抱く「ビュー（view）」を拡張できたのではないかと思います。冗談は別として、Groovyのテンプレート・フレームワークは、ビュー側面を要求する単純なアプリケーションを素早く処理する必要のある場合には、単純なJane Javaコーディングに代わる、巧妙な手法です。テンプレートは美しい抽象化を実現するものであり、適切に使用することによって、アプリケーションを長期的に維持管理する上で、非常に役立つものです。

次回は、Groovyを使って、Groovletを持つWebアプリケーションを作る方法を解説します。それまでの間、Groovyでのテンプレート開発を楽しんでください。

ダウンロード可能なリソース

内容	ファイル名	サイズ
j-pg02155-source.zip	j-pg02155-source.zip	2.18KB

関連トピック

- このページの最初または最後にあるCodeアイコンをクリックして（または、ダウンロードセクションを見てください）、この記事で使用したコード例をダウンロードしてください。
- 実用的なGroovy シリーズの他の記事もお見逃しなく。GroovyによるJDBCプログラミング（developerWorks, 2005年1月）では、チェックサムをレポートするアプリケーション例を解説し、またGroovyによるAntスクリプト（developerWorks, 2004年12月）では、Groovyに組み込みのレポート・ツールを使って、より表現力豊かなAntビルドを実現する方法について解説しています。
- Malcolm Davisが、Struts、オープン・ソースMVC実装（developerWorks, 2001年2月）の中で、MVC設計パターンの概要を的確に説明しています。
- VelocityはJavaテンプレート・エンジンとして広く使われており、また非常に強力なものです。Sing LiがClient and server-side templating with Velocity（developerWorks, 2004年2月）の中でVelocityを説明しています。
- もう一つ、なかなか素晴らしいJavaテンプレート・エンジンとして、FreeMarkerがあります。
- 次回Pythonで遊ぶ時には、Pythonの力を利用した、非常に効果的なテンプレート・エンジンであるCheetahも少し見てください。
- Cheetahについてさらに学ぶには、Python-Powered Templates with Cheetah（2005年1月 OnLamp.com）を見てください。
- Rubyが好きな方は、Ruby用の美しいテンプレート・エンジンである、ERBについて調べてみてください。
- developerWorksのJava technologyゾーンには、Javaプログラミングのあらゆる面に関する記事が豊富に用意されています。
- また、Java technology zone tutorials pageには、developerWorksが提供する、Javaに焦点を当てた無料チュートリアルの完全なリストが用意されています。