実用的なGroovy: JavaアプリケーションにGroovyを混ぜ込む

単純で容易に書けるスクリプトを埋め込み、Groovyの単純さを生かす

Andrew Glover

2005年 5月 24日

皆さんは、単純で容易に書けるGroovyのスクリプトを、もっと複雑なJava™プログラムに埋め込むことを考えたことがあるでしょうか？ 今回の実用的なGroovyでは、JavaコードにGroovyを組み込むための様々な方法を、Andrew Gloverが紹介します。そして、それが、いつ、どういう場合に適切なのかを解説します。

このシリーズを継続して読んでいる人であれば、Groovyには非常に様々な面白い使い方があること、そしてGroovyの一番の利点が生産性の高さであることを学んでいると思います。Groovyのコードは、Javaコードよりも容易に、速く書くことができる場合が多いため、開発ツールに加えるだけの価値があります。その一方で、私がこのシリーズで繰り返し強調してきたように、GroovyはJava言語の置き換えではなく、置き換えを意図したものでもありません。では、Javaプログラミングの中にGroovyを組み込むことができるのでしょうか、そしてそれは適切なのでしょうか、また、どういった場合に適切なのでしょうか。

今回は、この質問に答えることにします。まず、おなじみの質問から始めます。Groovyスクリプトを、どのようにしてJava互換のクラス・ファイルへとコンパイルするのでしょう。次に、秘密のベールの下にもぐり込み、Groovyのコンパイル機構（groovyc）が、この魔術をどのように実現しているかを探ります。Groovyがベールの下で何をしているかを理解することが、Javaコードの中でGroovyを使うための第一歩です。

注意して欲しいのですが、今回の記事の例で示すプログラム手法の一部は、以前の記事で議論した、GroovletsフレームワークとGroovyのGroovyTestCaseの核心をなしているものです。

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。

天国での結婚？

以前このシリーズで、Groovyを使って、より高速にJavaコードをユニット・テストする方法を説明した際に、皆さんは面白いことに気が付いたかも知れません。つまり、私はこうしたGroovyスクリプトをコンパイルしたのです。実際私は、私のgroovyユニット・テストを通常のJava .classクラス・ファイルにコンパイルし、Mavenビルドの一部として実行したのです。

このタイプのコンパイルは、groovycコマンドを呼び出すことによって行われます。このコマンドは、Groovyスクリプトを、単純な昔ながらのJava互換 .classファイルにとコンパイルします。例えば、そのスクリプトが3つのクラスを宣言している場合には、groovycを呼ぶと、少なくとも3つの .classファイルができます。こうしたファイル自体は標準のJavaルールに従い、.classファイルの名前は、宣言されたクラス名と一致します。

一例として、リスト1を見てください。これは、幾つかのクラスを宣言する単純なスクリプトを作っています。これを見ると、groovycコマンドが実際にどんなものを生成するかが分かるでしょう。

リスト1. Groovyでのクラス宣言とコンパイル

```
package com.vanward.groovy

class Person {
  fname
  lname
  age
  address
  contactNumbers
  String toString(){

    numstr = new StringBuffer()
    if (contactNumbers != null){
     contactNumbers.each{
       numstr.append(it)
       numstr.append(" ")
     }
    }
    "first name: " + fname + " last name: " + lname +
    " age: " + age + " address: " + address +
    " contact numbers: " + numstr.toString()
  }
}
class Address {
  street1
  street2
  city
  state
  zip
  String toString(){
   "street1: " + street1 + " street2: " + street2 +
     " city: " + city + " state: " + state + " zip: " + zip
  }
}
class ContactNumber {
  type
  number
  String toString(){
   "Type: " + type + " number: " + number
  }
}
nums = [new ContactNumber(type:"cell", number:"555.555.9999"),
 new ContactNumber(type:"office", number:"555.555.5598")]
```

```
addr = new Address(street1:"89 Main St.", street2:"Apt #2",
 city:"Utopia", state:"VA", zip:"34254")
pers = new Person(fname:"Mollie", lname:"Smith", age:34,
 address:addr, contactNumbers:nums)
println pers.toString()
```

リスト1では、3つのクラス、つまりPersonとAddress、そしてContactNumberを宣言しています。それに続くコードでは、新たに定義されたタイプのオブジェクトを作り、toString()メソッドを呼んでいます。ここまではごく単純ですが、今度は、この結果としてgroovycが生成したものを見てください（リスト2）。

リスト2. groovycコマンドが生成したクラス

```
aglover@12d21 /cygdrive/c/dev/project/target/classes/com/vanward/groovy
$ ls -ls
total 15
 4 -rwxrwxrwx+ 1 aglover  user   3317 May  3 21:12 Address.class
 3 -rwxrwxrwx+ 1 aglover  user   3061 May  3 21:12 BusinessObjects.class
 3 -rwxrwxrwx+ 1 aglover  user   2815 May  3 21:12 ContactNumber.class
 1 -rwxrwxrwx+ 1 aglover  user   1003 May  3 21:12
   Person$_toString_closure1.class
 4 -rwxrwxrwx+ 1 aglover  user   4055 May  3 21:12 Person.class
```

なんと、5つの .classファイルができています。PersonファイルとAddressファイル、ContactNumberファイルは分かりますが、他の2つは一体何なのでしょう。

Person$_toString_closure1.classは、PersonクラスのtoString()メソッドの中にあるクロージャー（closure）の結果であることが分かります。これは、内部的にはPersonの内部クラスですが、では、BusinessObjects.classファイルというのは何でしょう。

リスト1をよく見ると、このスクリプトの中心部分に書かれているコードが、（こうした3つのクラスを宣言した後で）.classファイルになり、スクリプトの名前から名前を付けられているのです。この場合、スクリプトにはBusinessObjects.groovyという名前が付いています。ですから、クラス定義に含まれていないコードは、BusinessObjectsという名前の .classファイルへとコンパイルされるのです。

逆にコーディングする

こうしたクラスをデコンパイルしてみると、面白いことが分かります。Groovyによる、ベールの下にあるコードの性格から、コンパイルの結果できる .javaファイルは、かなり大きくなっています。しかしそれよりも皆さんの目を引くのは、Groovyスクリプトの中で宣言されたクラス（例えばPerson）と、クラスの外にあるコード（例えばBusinessObjects.classの中のコード）の差でしょう。Groovyファイルの中で定義されたクラスは最終的にGroovyObjectを実装し、クラスの外で定義されたコードは、Scriptを継承するクラスの中にバンドルされています。

例えば、コンパイルの結果できる、BusinessObjects.classという .Javaファイルを調べると、main()メソッドとrun()メソッドを定義していることが分かります。当然ながら、run()メソッドは、こうしたオブジェクトの新しいインスタンスを作るために私が書いたコードを含んでおり、main()メソッドは、run()メソッドを呼んでいます。

繰り返しますが、こうした詳細を説明する理由は、Groovyをよく理解しておいた方が、Javaプログラムの中にGroovyを取り込むことが容易になるためです。そうすると皆さんは、「なぜ私がそ

んなことを望むのか？」と尋ねるかも知れません。例えば、皆さんがGroovyで何か素晴らしいものを開発したとしましょう。それをJavaプログラムの中に組み込めた方が良いと思いませんか？

議論を進めるために、まず、Groovyで何か便利なものを作ってみようと思います。その後で、それを通常のJavaプログラムの中に埋め込むための方法を探ることにします。

Groovyで音楽を扱う

私は音楽が大好きです。実際のところ、私のCDコレクションは、コンピューター関係の本のコレクションと匹敵するほどです。これまで長年の間、私は音楽を様々なコンピューターに取り込んできましたが、その過程でMP3のコレクションがあふれかえり、音楽の豊かさを表すディレクトリーを数限りなく持つまでに至ってしまいました。

そこで私は最近、私の音楽コレクションを、再度きちんと整理するための第一歩を踏み出しました。ディレクトリーの中にあるMP3ファイルのコレクションに対して繰り返しを行い、各ファイルに対する詳細情報（演奏者やアルバム・タイトルなど）が得られるような、ちょっとしたGroovyスクリプトを書いたのです。このスクリプトをリスト3に示します。

リスト3.　非常に便利なGroovyスクリプト

```
package com.vanward.groovy

import org.farng.mp3.MP3File
import groovy.util.AntBuilder

class Song {

  mp3file
  Song(String mp3name){
    mp3file = new MP3File(mp3name)
  }
  getTitle(){
    mp3file.getID3v1Tag().getTitle()
  }
  getAlbum(){
    mp3file.getID3v1Tag().getAlbum()
  }
  getArtist(){
    mp3file.getID3v1Tag().getArtist()
  }
  String toString(){
    "Artist: " + getArtist() + " Album: " +
      getAlbum() + " Song: " + getTitle()
  }
  static getSongsForDirectory(sdir){
    println "sdir is: " + sdir
    ant = new AntBuilder()
    scanner = ant.fileScanner {
       fileset(dir:sdir) {
         include(name:"**/*.mp3")
       }
    }
    songs = []
    for(f in scanner){
      songs << new Song(f.getAbsolutePath())
    }
    return songs
  }
}
songs = Song.getSongsForDirectory(args[0])
```

```
songs.each{
 println it
}
```

ご覧の通り、スクリプトはごく単純であり、特に私のようなタイプの人間には非常に便利にできています。特定なディレクトリー名を渡すだけで、そのディレクトリーの中にある全てのMP3ファイルについての情報（演奏者名、曲名、アルバム名）を返してくれるのです。

では、この便利なスクリプトを、通常のJavaプログラムの中に取り込むために何が必要かを見てみましょう（このJavaプログラムを、データベースを使って音楽を整理できるように、さらにはMP3の演奏まで行えるようにしようというわけです）。

クラス・ファイルはクラス・ファイルです

先に議論した通り、第1の選択肢は、groovycを使って単純にスクリプトをコンパイルすることです。この場合、groovycは、少なくとも2つの .classファイル（1つはSongクラスに対して、もう1つは、Songの宣言に続くスクリプト・コードに対して）を作るはずです。

groovycは、実際には5つの .classファイルを作ります。これは、Songs.groovyが3つのクロージャーを含む（getSongsForDirectory()メソッドの中に2つ、スクリプト本体の中に1つ）ことと関係しています。このクロージャーで、Songの集合に対して繰り返しを行い、printlnを呼んでいます。

.classファイルのうち3つは、実際にはSong.classとSongs.classの内部クラスなので、注目する必要があるのは、2つの .classファイルのみです。Song.classは、Groovyスクリプトの中のSong宣言に直接マップされ、GroovyObjectを実装します。一方Songs.classは、私がSongを定義した後のスクリプト・コードを表すので、Scriptを継承します。

この時点で、新しくコンパイルしたGroovyコードをJavaコードの中に取り込む方法として、2つの選択肢があります。Songs.classファイルはScriptを継承しているので、Songs.classファイルの中にあるmain()メソッドから実際にコードを実行するか、あるいはSong.classをクラスパスの中に含め、Javaコードの中で他のオブジェクトを使う場合と同じように使うのです。

気楽に考える

javaコマンドによってSongs.classを呼ぶのは呆れるほど簡単ですが、Groovyに関連付けられた依存関係や、Groovyスクリプトが要求するような依存関係を忘れないようにする必要があります。Groovyが要求するクラスを含めるために最も容易なのは、Groovyに埋め込み可能で一式揃ったjarファイルを、クラスパスに挿入する方法です。この場合では、そのファイルはgroovy-all-1.0-beta-10.jarです。Songs.classクラスを実行するためには、私が使ったMP3ライブラリー（jid3lib-0.5.jar>）も忘れずに含める必要があります。また、私はAntBuilderを使っているので、クラスパスにはAntも含める必要があります。リスト4は、これらをまとめたものです。

リスト4. JavaコマンドラインからのGroovy

```
c:\dev\projects>java -cp  ./target/classes/;c:/dev/tools/groovy/
  groovy-all-1.0-beta-10.jar;C:/dev/tools/groovy/ant-1.6.2.jar;
  C:/dev/projects-2.0/jid3lib-0.5.jar
  com.vanward.groovy.Songs c:\dev09\music\mp3s
Artist: U2 Album: Zooropa Song: Babyface
Artist: James Taylor Album: Greatest Hits Song: Carolina in My Mind
Artist: James Taylor Album: Greatest Hits Song: Fire and Rain
Artist: U2 Album: Zooropa Song: Lemon
Artist: James Taylor Album: Greatest Hits Song: Country Road
Artist: James Taylor Album: Greatest Hits Song: Don't Let Me
  Be Lonely Tonight
Artist: U2 Album: Zooropa Song: Some Days Are Better Than Others
Artist: Paul Simon Album: Graceland Song: Under African Skies
Artist: Paul Simon Album: Graceland Song: Homeless
Artist: U2 Album: Zooropa Song: Dirty Day
Artist: Paul Simon Album: Graceland Song: That Was Your Mother
```

JavaコードにGroovyを埋め込む

コマンドラインによる方法は容易で楽しいものですが、この方法だけが全てではありません。より高度な方法に興味がある方であれば、MP3の音楽のユーティリティーを、Javaプログラムの中に直接インポートすることもできます。この場合には、Song.classをインポートし、それを、Java言語で他のクラスを使う場合と同じように使います。クラスパスの問題は、上記と同様です。Groovyを超えるjarファイルであるAntと、jid3lib-0.5.jarファイルは必ず含める必要があります。リスト5を見ると、GroovyのMP3ユーティリティーを単純なJavaクラスにインポートする方法が分かると思います。

リスト5. 埋め込まれたGroovyコード

```
package com.vanward.gembed;

import com.vanward.groovy.Song;
import java.util.Collection;
import java.util.Iterator;

public class SongEmbedGroovy{

 public static void main(String args[]) {
  Collection coll = (Collection)Song.getSongsForDirectory
    ("C:\\music\\temp\\mp3s");
  for(Iterator it = coll.iterator(); it.hasNext();){
    System.out.println(it.next());
  }
 }
}
```

Groovyクラスローダー

これで全てを学んだと思うかも知れませんが、他にもまだ、Java言語でGroovyを使って遊ぶ方法があります。直接コンパイルによってGroovyスクリプトをJavaプログラムに取り込む方法の他に、スクリプト自体を埋め込む方法に関しても、幾つかの選択肢があるのです。

例えば、GroovyのGroovyClassLoaderを使ってGroovyスクリプトを動的にロードし、その振る舞いを実行することができます。これをリスト6に示します。

リスト6. GroovyClassLoaderがGroovyスクリプトを動的にロードし、実行する

```
package com.vanward.gembed;

import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;
import groovy.lang.MetaMethod;
import java.io.File;

public class CLEmbedGroovy{

 public static void main(String args[]) throws Throwable{

  ClassLoader parent = CLEmbedGroovy.class.getClassLoader();
  GroovyClassLoader loader = new GroovyClassLoader(parent);

  Class groovyClass = loader.parseClass(
    new File("C:\\dev\\groovy-embed\\src\\groovy\\
      com\\vanward\\groovy\\Songs.groovy"));

  GroovyObject groovyObject = (GroovyObject)
    groovyClass.newInstance();

  Object[] path = {"C:\\music\\temp\\mp3s"};
  groovyObject.setProperty("args", path);
  Object[] argz = {};

  groovyObject.invokeMethod("run", argz);

 }
}
```

メタ化してください

リフレクションや、リフレクションを使ってできることが大好きな変人であれば、GroovyのMetaクラスに狂喜することでしょう。リフレクションの場合と同じように、Metaクラスを使うと、GroovyObjectに関するアスペクトを（GroovyObjectのメソッドと同じように）発見することができ、実際に新しい振る舞いを作成し、実行することができるのです。実は、これがGroovyの心臓部分なのです。スクリプトを実行している時に、これがどのように動作しているか、ちょっと想像してみてください

クラスローダーは、デフォルトとして、スクリプト名に対応するクラス（この場合にはSong.class>ではなく、Songs.classです）をロードすることに注意してください。Songs.classはGroovyのScriptクラスを継承していることが分かっているので、次にすべきことはrun()メソッドの実行、ということは明らかでしょう。

私のGroovyスクリプトが、ランタイム引数にも依存していることを思い出してください。この場合には、ディレクトリー名に対する最初の要素を設定することになるので、args変数を適切に設定する必要があります。

より動的な選択肢

コンパイルしたクラスを使ったり、クラスローダーでGroovyObjectを動的にロードしたりする方法の他に、Groovyの賢いGroovyScriptEngineやGroovyShellを使って、動的にGroovyスクリプトを実行することもできます。

通常のJavaクラスの中にGroovyShellオブジェクトを埋め込むことによって、ちょうどクラスローダーが行うように、動的にGroovyスクリプトを実行することができます。しかも、スクリプトを

どのように実行するかに関して、幾つかのオプションが選べるのです。リスト7を見ると、通常のJavaクラスの中に、どのようにGroovyShellが埋め込まれているかが分かると思います。

リスト7. GroovyShellを埋め込む

```
package com.vanward.gembed;

import java.io.File;
import groovy.lang.GroovyShell;

public class ShellRunEmbedGroovy{

 public static void main(String args[]) throws Throwable{

  String[] path = {"C:\\music\\temp\\mp3s"};
  GroovyShell shell = new GroovyShell();
  shell.run(new File("C:\\dev\\groovy-embed\\src\\groovy\\
    com\\vanward\\groovy\\Songs.groovy"),
    path);
 }
}
```

ご覧の通り、Groovyスクリプトの実行は極めて容易です。単純にGroovyShellのインスタンスを作ってスクリプト名を渡し、run()メソッドを呼ぶだけです。

これだけではありません。ご希望とあれば、GroovyShellインスタンスに、スクリプトのScriptタイプを尋ねることもできるのです。Scriptタイプが分かると、任意のランタイム値を含むBindingオブジェクトを渡してから、run()メソッドを呼ぶように進むことができるのです。これをリスト8に示します。

リスト8. GroovyShellで遊ぶ

```
package com.vanward.gembed;

import java.io.File;
import groovy.lang.Binding;
import groovy.lang.GroovyShell;
import groovy.lang.Script;

public class ShellParseEmbedGroovy{

 public static void main(String args[]) throws Throwable{
  GroovyShell shell = new GroovyShell();
  Script scrpt = shell.parse(
    new File("C:\\dev\\groovy-embed\\src\\groovy\\
      com\\vanward\\groovy\\Songs.groovy"));

  Binding binding = new Binding();
  Object[] path = {"C:\\music\\temp\\mp3s"};
  binding.setVariable("args",path);
  scrpt.setBinding(binding);

  scrpt.run();
 }
}
```

Groovyのスクリプト・エンジン

GroovyScriptEngineオブジェクトは、動的にスクリプトを実行するという点で、GroovyShellとほとんど同じように動作します。異なる点は、GroovyScriptEngineをインストールすると、その時点で

一連のディレクトリーがGroovyScriptEngineに与えられるのです。そして必要に応じて、それらに複数のスクリプトを評価させることができるのです。これをリスト9に示します。

リスト9. GroovyScriptEngineの実際

```
package com.vanward.gembed;

import java.io.File;
import groovy.lang.Binding;
import groovy.util.GroovyScriptEngine;

public class ScriptEngineEmbedGroovy{

 public static void main(String args[]) throws Throwable{

  String[] paths = {"C:\\dev\\groovy-embed\\src\\groovy\\
    com\\vanward\\groovy"};
  GroovyScriptEngine gse = new GroovyScriptEngine(paths);
  Binding binding = new Binding();
  Object[] path = {"C:\\music\\temp\\mp3s"};
  binding.setVariable("args",path);

  gse.run("Songs.groovy", binding);
  gse.run("BusinessObjects.groovy", binding);
 }
}
```

リスト9では、私が必要とするパスを含んだ配列を、インスタンス化したGroovyScriptEngineに渡しています。そして、昔からおなじみのBindingオブジェクトを作ってから、これもおなじみのSongs.groovスクリプトを実行しています。また遊びとして、BusinessObjects.groovyスクリプト（最初の方に出てきました）も実行しています。

Bean Scripting Framework

最後に忘れてはならないのが、長老格たる、JakartaのBSF（Bean Scripting Framework）です。BSFは、通常のJavaアプリケーション（Groovyも含みます）に、任意のスクリプト言語を埋め込むための共通APIを提供しようとしています。この標準は最小公倍数的な手法ですが、これを使うことによって、Groovyスクリプトを難なく埋め込むことができます。

先ほどのBusinessObjectsスクリプトを思い出してください。リスト10を見ると、BSFによって、このスクリプトを通常のJavaプログラムに容易にプラグインできることが分かるでしょう。

リスト10. BSFを使う

```
package com.vanward.gembed;

import org.apache.bsf.BSFManager;
import org.codehaus.groovy.runtime.DefaultGroovyMethods;
import java.io.File;
import groovy.lang.Binding;

public class BSFEmbedGroovy{

 public static void main(String args[]) throws Exception {
  String fileName = "C:\\dev\\project\\src\\groovy\\
    com\\vanward\\groovy\\BusinessObjects.groovy";
  //this is required for bsf-2.3.0
```

```
  //the "groovy" and "gy" are extensions
  BSFManager.registerScriptingEngine("groovy",
    "org.codehaus.groovy.bsf.GroovyEngine", new
      String[] { "groovy" });
  BSFManager manager = new BSFManager();
  //DefaultGroovyMethods.getText just returns a
  //string representation of the contents of the file
  manager.exec("groovy", fileName, 0, 0,
    DefaultGroovyMethods.getText(new File(fileName)));
  }
}
```

まとめ

この記事から1つ明確に分かることは、GroovyをJavaコードの中で再利用するための選択肢は無数にある、ということです。Groovyスクリプトを単純なJava .classクラスにとコンパイルする方法から、スクリプトを動的にロード、実行する方法に至るまで様々な方法がありますが、考慮すべき要点は、柔軟性と結合性(coupling)です。Groovyを通常の .classファイルの中にコンパイルする方法は、埋め込もうとしている機能を使用する上では最も単純な選択肢ですが、スクリプトを動的にロードする方法を使うと、コンパイルで時間を犠牲にすることなく、振る舞いの追加や修正が容易に行えます（当然ですが、この選択肢が使えるのはインターフェースが変化しない場合のみです）。

スクリプト言語を通常のJavaに埋め込むことは、常に必要なことではありませんが、時々、必要な場合があります。ここで取り上げた例では、単純なディレクトリー検索ユーティリティーをJavaベースのアプリケーションの中に埋め込みました。これによって、容易にMP3プレーヤー・アプリケーションや別のMP3ユーティリティーができます。もちろん、MP3ファイルの検索ユーティリティーをJavaコードで書き直すこともできますが、そうする必要はないのです。Groovyは完璧にJava言語と互換性があります。その上、色々な選択肢をいじり回す楽しみまであるのです！

関連トピック

- 実用的なGroovyシリーズの他の記事もぜひ読んでください。シリーズの進行と共に、それぞれの記事がお互いに関係してきます。今回の記事に特に関係するものとして、下記の記事があります。
  - 「Groovyを使って、より高速にJavaコードをユニット・テストする」（developerWorks, 2004年11月）は、通常のJavaプログラムを、GroovyとGroovyTestCaseを使ってユニット・テストする方法を紹介しています。
  - 「Groovyでサーバー側に対応する」（developerWorks, 2005年3月）はGroovletsを紹介しています。
- Teodor Zlatanovが「MP3とPerlで遊ぶ」（developerWorks, 2003年12月）の中で、MP3ファイルをスクリプト言語で扱う利点について解説しています。
- Bean Scripting FrameworkやStruts、Jythonなどを実際に体験するために、「Customizing WebSphere Studio to use the Struts Scripting tool with ActionClasses in Jython」（developerWorks, 2005年3月）を読んでください。
- オープンソースのJava ID3 Tag Libraryは、MP3ファイルから曲のタイトルや演奏者、アルバム名などの情報を読むことのできる楽しいライブラリーです。
- Bean Scripting Frameworkは、IBM alphaWorksのプロジェクトとして誕生しましたが、その後、ApacheのJakartaプロジェクトに寄贈されました。
- 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、developerWorksのJava technologyゾーンのtutorials pageには、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ありますので、ご覧ください。