vol.6

Magazine

Japan Grails/Groovy User Group

Contents

Groovy臨機応変（第一回）
～動中の静…Groovy 2.1.0の新機能 その1～

series 10

上原 潤二（うえはら じゅんじ）

NTTソフトウェア株式会社Grails推進室所属。JGGUG（日本Grails/Groovyユーザ会）運営委員。
「Grails徹底入門」（翔泳社）、「プログラミングGROOVY」（技術評論社）執筆メンバーの1人。
Groovy技術に関するブログ「Grな日々」を主宰している。

はじめに

こんにちは、JGGUGの上原（NTTソフトウェア株式会社所属）です。G*マガジンに記事を書くのは初めてとなります。このシリーズでは、Groovyの機能について紹介していきます。今回および次回では、この記事が掲載されているころには既にリリース済みの予定である、Groovy 2.1.0で導入された一連の新機能について紹介していく予定です。なお、今回の記事の内容は記事執筆時点で入手可能なGroovy 2.1.0 RC1バージョンに基づいており、最終リリース版では仕様や動作が変更されている可能性もあることをあらかじめご了承ください。

Groovy 2.1.0は、メジャーバージョンアップであった2.0に引き続くマイナーバージョンアップです。Groovy 2.0リリースはインフラレベルの大きな機能拡張をドカンと提供するものでしたが、それを踏まえての2.1.0リリースは、2.0で導入された機能をより使いやすく発展させるための機能が含まれており、実用性を高めるための周辺手段が整って来たような印象です。Groovy 2.0の新機能については参考[1]、[2]などを参考下さい。

今回は、Groovy 2.1.0の新機能の中で「静的Groovy」機能に関連のある以下について紹介します。

- @DelegatesTo
- 型チェックを拡張し、カスタマイズする
- コンパイラ設定スクリプト
- コンパイラ設定ビルダ

Groovy 2.0及び2.1の新機能については、参考[1]（プログラミングGROOVY別冊：第8章 Groovy 2.0の新機能）なども参考にしてください。

@DelegatesTo

@DelegatesToは静的型チェックを強化するために導入された新規のAST変換アノテーションであり、groovy.transform.*パッケージに所属します。

@DelegatesToアノテーションを解説する前に、delegateプロパティについて補足しておきます。クロージャにおけるdelegateプロパティとは、クロージャ中のコードにおいて、レシーバとなるオブジェクトを明示的に指定していないメソッド呼び出しやプロパティ参照における「暗黙のレシーバ」を示す動的に変更可能なプロパティです。インスタンスメソッド呼び出し「object.method()」において、「object」がレシーバです。インスタンスメソッド中のコードでレシーバを省略したときには通常メソッドが所属するクラスのインスタンスthisがレシーバとして扱われますが、それと同様にクロージャではdelegateプロパティが保持するオブジェクトを暗黙のレシーバとして呼び出そうとします。要はdelegateは、クロージャにおける「this」のようなものです。

delegateは動的に定まるプロパティであるが故に静的型チェックには本来参与できないのですが、@DelegatesToアノテーションでdelegateプロパティが保持するオブジェクトの型を明示的に指定してやることで、delegateプロパティが示すオブジェクトのメソッド呼び出しやプロパティ参照について静的型チェックができるようになります。

このような言葉での説明だけでは理解しにくいと思うので、以降では例を示しながら解説します。

■@DelegatesToの使用例

まず、Groovy標準のwith句と同じような効果を発揮する以下のようなメソッド「myWith」を定義することを考えてみます。本来のwith句は、Objectクラスに追加されたGDKメソッドですが、ここでは通常のメソッドとして定義します。

```
def myWith(Object self, Closure cl) {
  cl.delegate = self
  cl.call()
}
```

これは以下のようにwith句と同じように（少し違いますが）呼び出すことができ、"ABC"が表示されます。

```
myWith("abc") {
  println toUpperCase()
}
// ==> "ABC"が表示される
```

ここまでは、delegateプロパティの典型的な使用例の1つです。

さて、このコードでtoUpperCase()の綴り間違いなども検出できるようにするために、以下のように静的型チェックを適用したいとします。

```
import groovy.transform.*
@TypeChecked
def d() {
  myWith("abc") {
    println  toUpperCase()
  }
}
```

@TypeCheckedを適用するために、対象コードをメソッド定義で囲みます。しかし、上記は以下のようにエラーになってしまいます。

```
wyWithTest.groovy: 10: [Static type
checking] - Cannot find matching method
myWithCheck#toUpperCase(). Please check if the
declared type is right and if the method exists.
 @ line 10, column 14.
       println  toUpperCase()
                ^
1 error
```

静的型チェック配下のクロージャ中ではdelegateプロパティを参照できないし、もし参照できたとしてもdelegateプロパティにString型のインスタンスが保持されることがコンパイル時には決定できないので、toUpperCase()メソッド呼び出しを静的に解決することはできません。なので、delegateに対するメソッド呼び出しやプロパティ参照を含むクロージャは、通常は静的型チェック・静的コンパイルができません。仮に、

```
def myWith(String self, Closure cl) {
  cl.delegate = self
  cl.call()
}
```

のようにmyWithの引数selfをString型に限定しても同じです。myWithの処理内容「delegateプロパティにStringインスタンスを代入する」という処理内容を、myWithの呼び出し側は知らないからです。現在のGroovyが、メソッド呼び出しをまたがっての大域的な型推論を行わないから、とも言えます。

ちなみに、以下のように通常のwithを使う場合、

```
import groovy.transform.*
@TypeChecked
def d() {
  "abc".with {
    println toUpperCase()
  }
}
```

これはコンパイルも通り実行可能であり、toooUpperCase()のように綴り間違いをしていた時には以下のように「そのようなメソッドは無い」という静的型エラーになります。

```
org.codehaus.groovy.control.MultipleCompilationEr
rorsException: startup failed:
with.groovy: 6: [Static type checking] - Cannot
find matching method with#toooooUpperCase().
Please check if the declared type is right and if
the method exists.
 @ line 6, column 13.
       println toooooUpperCase()
               ^

1 error
```

これが可能なのは、Groovyの静的型チェッカがwithメソッドの呼び出しを特別扱いし、delegateプロパティの値がwithメソッドの対象メソッドの型であると型推論して処理しているからです。

同種のことを、ユーザ定義メソッドや、サードパーティ提供のライブラリやフレームワークなどのAPIでもできるようにすることが、@DelegatesToの存在意義の1つです。myWith定義を例にすると、以下のようにクロージャ引数clに@DelegatesToを指定します。

```
import groovy.transform.*

def myWith(Object self, @DelegatesTo(String)
Closure cl) {
  cl.delegate = self
  cl.call()
}

@TypeChecked
def b() {
  myWith("abc") {
      println  toUpperCase()
  }
}
```

このように「このメソッド呼び出しに**即値の引数として**与えられたクロージャのdelegateプロパティにはString型インスタンスが割り当てられる」ということをコンパイラに伝えることで、クロージャ中のdelegateに対するメソッド呼び出しの静的型チェックや静的コンパイルを可能とします。なお、クロージャが即値ではなく例えば変数などに代入されたクロージャを渡すようなときには、@DelegatesToは機能しません。このことは原理的に明らかでしょう。

■@DelegatesToの使用例2

delegateプロパティは、ExpandoMetaClass(EMC)を用いてクラスに動的にメソッドを追加する際にも良く使われます。この際にメソッドとして追加するクロージャを静的コンパイル・静的型

チェックするのにも@DelegatesToを使用することができます。以下は、Stringにhello()メソッドを動的に追加する例です。

```
import groovy.transform.*

def addMethodToString(name,
  @DelegatesTo(String) Closure cl) {
  // 引数クロージャclのdelegateプロパティには
  // String型が与えられると宣言
  String.metaClass."$name" = cl
}

@CompileStatic
def c() {
  addMethodToString("hello") {
    println toUpperCase()
    // toUpperCase()の暗黙のレシーバの型を解決でき、
    // 静的コンパイルも可能
  }
}
c()

"def".hello()
```

EMCを用いるためのmetaClassプロパティ自体は静的型チェック配下では使用できませんが、静的型チェックを適用しないメソッドを介してクロージャを渡すことで静的コンパイルしたクロージャを既存クラスにメソッドとして追加することができます。とはいえ、このように動的に追加したメソッドは、静的コードからは呼び出せませんので、上記ができることの有用性にはちょっとした疑問が残るかもしれません。

■@DelegatesToの利用目的

@DelegatesToが使われるシチュエーション、すなわちdelegateに指定された固定クラスに対するメソッド呼び出しは、SpockやGradleなどでも使われているそうです（参考[3]）。なのでこれらのツールで将来的に@DelegatesToが使用されるようになることで、これらのツールの入力となるDSLについて静的型チェックがより有効に活用され、論理的なエラーがより早い段階で検出でき、またより的確になるのかもしれません（ただし、静的型チェックに相当することをAST変換で実装することは原理的に可能なので、同等のことをこれらのツールが既に行なっていないと仮定して、の話です）。また、@DelegatesToは以下を目的として使えるとのことです（参考[1]）。

- APIのドキュメントとして
- IDEでの補完など

IDEのDSLの補完のための情報は、従来はIDE独自の設定ファイル（EclipseのDSLディスクリプタやIDEA IntelliJのGDSL）として提供されてきましたが、これらを一部置きかえ得るものでもあるようです。

型チェックを拡張し、カスタマイズする

Groovy 2.1.0では、静的型チェックの振舞いをGroovy利用者がカスタマイズすることができるようになりました。具体的には、以下のように静的型チェックのための@TypeCheckedアノテーションに、型チェックを行うためのスクリプトを指定できるようになりました。

```
@TypeChecked(extensions='MyExtension.groovy')
  // 型チェックを行うスクリプト
  // 'MyExtension.groovy'を指定
void exec()
{
  "test".toUpperCase()
}
```

上記のMyExtension.groovyは、Type Checking DSLと呼ばれる特別な記法で記述されたGroovyコードであり、コンパイルしておく必要はありません。留意点としては、このType Cheking DSLスクリプトは@TypeCheckedを指定したクラスやスクリプトがあるディレクトリから相対的な位置に置き、相対指定のファイル名で指定する必要があるようです（試した限りでは）。

Type Checking DSLは以下のような記法です（参考[2]より）。

```
onMethodSelection { expr, method -> ... }
afterMethodCall { mc -> ... }
unresolvedVariable { var -> .. }
incompatibleAssignment { receiver, name, argList,
argTypes, call -> .. }
```

これらは、Groovy標準の静的型チェッカから呼び出されるイベントハンドラの定義であり、静的型チェックの各状況下で呼び出されます。これらのハンドラからの返り値を指定することで、デフォルトの静的型チェッカであれば報告するはずであった静的型エラーを握り潰す（＝静的型エラーを静的型エラー扱いしない）こともできます。以下は指定できる静的型チェックイベントハンドラの一覧です。

- onMethodSelection
- afterMethodCall
- beforeMethodCall
- unresolvedVariable
- unresolvedProperty
- unresolvedAttribute
- methodNotFound
- afterVisitMethod
- beforeVisitMethod
- afterVisitClass
- beforeVisitClass
- incompatibleAssignment
- setup
- finish

Type Checking DSLのイベントハンドラで書ける処理の詳細はここでは説明しませんが、AST情報を駆使して自由にロジックを組むことができます。テストコードを見る限り、型チェック拡張によって、例えば以下のような機能が実現できそうです。

- 存在しないメソッド呼び出しの静的型チェック時に@Grab指定を追加し、拡張モジュールを実行時に導入させる。そして拡張モジュールが追加したメソッドによって、実行は成功する。
- sprintfでのフォーマット指定と、実引数の型チェック
- メソッド呼び出しを大文字小文字を無視して比較するようにする
- @DelegatesTo相当の機能を実行する。つまりDelegatesToで型チェックさせる。

Type Checking DSLの詳細を調べたい場合は、今のところGroovyのソースコードに含まれるテストコードおよびorg.codehaus.groovy.transform.stc.GroovyTypeCheckingExtensionSupportクラスのソースコードを読むのが最善です。

なお、「型チェック拡張がコード生成に介在できるか」が興味深いところです。もしもそれが可能なら、例えば、動的メソッドの呼び出しやプロパティ参照の静的型エラー時に、その呼び出しコードをinvokeMethod()やgetProperty()の呼び出しに置き替えることにより、Groovy++のmixed modeコンパイルに似たことができるなど、応用範囲が広くなるからです。ということでGroovy Userのメーリングリストで聞いてみたところ、静的Groovy開発者のCédric Champeau氏から回答頂き、「型チェック拡張は「型チェック」なので本来はASTの書き換えをするものじゃないけども、技術的には可能」とのことでした。また、コーディング規約の統一のためのチェッカやCodeNarcのような静的解析など汎用的なチェックに使用することも狙っているとのことでした。

■型チェック拡張の意義

参考[1]では、

> *著者の私見では、「静的型チェック」だけではメリットが十分ではないので、次節で紹介する、性能向上をメリットとする「静的コンパイル」を通じて利用することが主となるでしょう。*

などと書いてしまいましたが、型チェック拡張によるカスタマイズができることを前提とすれば、「静的型チェックを行うDSLの容易な実現」という目的においては、@CompileStaticによる性能向上が無くとも、DSLに対してより的確なコンパイル時のエラーメッセージが出せたり、IDEサポートが得られるようになるなら、静的型チェックの有用性はとても高いと言えます。しかしその場合でも、Groovyの一般利用者から見ると、直接@TypeCheckedや型チェック拡張を使用するのではなく、Gradleなどのツールやフレームワークがそれを使用し、間接的に利点が享受される、という形になるのかもしれません。

コンパイラ設定スクリプト

次にコンパイラ設定スクリプトについて説明します。コンパイラ設定スクリプトは、groovycコマンドなどのコマンドラインオプションの一つとして「コンパイラ設定」を指定できるというものです。コンパイラ設定とは、具体的にはorg.codehaus.groovy.control.CompilerConfigurationクラスのことであり、従来から明示的にこのクラスを使うことで、任意のコンパイラ設定のもとでGroovyShellなどを呼び出したりすることはできました。Groovy 2.1.0以降では、そのようなプログラムを書かなくとも手軽にコマンドラインからコンパイラ設定を指定することができるようになります。

コンパイラ設定では、任意のクラスをimport/static importした扱いにできたりなど様々な設定ができますが、特に「コンパイル対象となるスクリプトやクラスに任意のAST変換を適用する」という設定を行うことができるので有用です。例えば、コンパイラ設定スクリプトで@CompileStaticや@TypeCheckedなどのAST変換を適用することで、コンパイル対象全部に対して静的コンパイルや静的チェックを行うことができます。あるいは、全クラスに@ToStringを適用し、デフォルトの動作としてtoString()メソッドを自動生成させる、といったこともできます。

コンパイラ設定は、コンパイラ設定スクリプト（以下ではcompConfig.groovy）と呼ばれるGroovyスクリプトによる設定ファイルとして準備し、以下のようにコマンドラインオプション-configscriptでそのファイル名を指定します。

```
groovyc -configscript compConfig.groovy Test.groovy
```

コンパイラ設定スクリプトの記述例は以下のとおりです。

```
import groovy.transform.CompileStatic
import org.codehaus.groovy.control.customizers
 .ASTTransformationCustomizer

configuration.addCompilationCustomizers(
  new ASTTransformationCustomizer(CompileStatic))
```

なお、Groovy 2.1.0-rc-1時点ではコンパイラ指定スクリプトで指定できるのはgroovycコマンドのみですが、2.1.0-finalまでにgroovyコマンドでも対応する予定とのことです（あくまで予定）。

コンパイラ設定ビルダ

さて最後に、コンパイラ設定ビルダについてですが、これは前節でのコンパイラ設定スクリプト（もしくは通常のプログラム中でのコンパイラ設定）を、より簡潔に記述するためのDSL記法です。Groovy 2.1.0では、設定ファイルはなんでもかんでもDSLにする方針のようですね。前項での「全ファイルに対して@CompileStaticを適用する」というコンパイラ設定スクリプトは、コンパイラ設定ビルダを用いて以下のように記述できます。

```
import groovy.transform.*

withConfig(configuration) {
   ast(CompileStatic)
}
```

さらに、コンパイラ設定ビルダでは、例えば拡張子が「.sgroovy」であるようなファイルのみを静的コンパイルする、といった指定も以下のように簡単にできます。

```
import groovy.transform.*

withConfig(configuration) {
   source(extension: 'sgroovy') {
      ast(CompileStatic)
   }
}
```

他にもファイル名のベース名や拡張子などで柔軟かつ一括の設定を行なうことができます。コンパイラ設定ビルダ記法の例については現時点で非常に情報が少ないのですが、org.codehaus.groovy.control.customizers.builder.SourceAwareCustomizerFactoryクラスの冒頭コメントやorg.codehaus.groovy.control.customizers.builder.CompilerCustomizationBuilderのソースコードがかろうじて参考になるでしょう。逆に言うとソースコード以外の情報は見付けられませんでした。

コンパイラ設定ビルダ及びコンパイラ設定スクリプトは、特に静的Groovyに限定された機能ではありませんが、上の@CompileStaticやもしくは@TypeCheckedを全体に適用することが有用であるため今回紹介しました。

まとめ

今回紹介した機能群によって、Groovy 2.0で登場した静的Groovy機能が、「Javaのように書ける」といったある意味後ろ向きな目的だけではなく、「より有用なDSLを書く」ということにも寄与するようになってきました。DSLや動的なGroovyコードにおいても静的型チェックが(実装が容易に)できることは有用でしょう。このような、「**動的言語・動的型言語をあくまでもベースにした上での静的型の有用性の回収**」はGroovyのユニークな特長であり、今後のGroovyの進化の方向性の一つでもあると思います。今後のGradleやGrailsなどのツールの対応に期待です。

参考

- [1] プログラミングGROOVY別冊：第８章 Groovy 2.0の新機能
  http://beta.mybetabook.com/b/ksky/%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9F%E3%83%B3%E3%82%B0GROOVY%E5%88%A5%E5%86%8A%EF%BC%9A%E7%AC%AC%EF%BC%98%E7%AB%A0+Groovy+2.0%E3%81-%AE%E6%96%B0%E6%A9%9F%E8%83%BD
- [2]Groovy 2.0 and Beyond
  http://www.slideshare.net/glaforge/groovy-2-and-beyond
- [3] Groovy devメーリングリストの「Detailed proposal for @DelegatesTo annotation」というスレッド
  http://groovy.329449.n5.nabble.com/Detailed-proposal-for-DelegatesTo-annotation-td5710777.html
- [4] Groovy 2.1 release notesGroovy 2.1 release notes
  http://groovy.codehaus.org/Groovy%202.1%20release%20notes

series 11

アプリケーションをGroovyでコントロールする

奥 清隆（おく きよたか）
仕事でもがっつりGroovyと戯れるプログラマ。
日本Grails/Groovyユーザーグループ関西支部長。
著書：『Seasar2によるWebアプリケーションスーパーサンプル』

Groovy Remote Control

今回は個人的にお気に入りなライブラリの1つである、Groovy Remote Controlをご紹介します。

http://groovy.codehaus.org/modules/remote/

Groovy Remote Controlはその名の通り、リモートのJVM上で起動しているプログラムをコントロールすることができるライブラリです。プログラミングGROOVYを読まれた方はお気づきだと思いますが、Groovyの使いどころ7つのパターンの1つである内視鏡手術を導入するのに適したライブラリです。

■内視鏡手術

内視鏡手術とは、アプリケーションを再起動することなく実行中にアプリケーション内部の状態を確認したり変更することを言います。例としては、JenkinsのCLIやスクリプトコンソールなどがあります。Jenkinsのスクリプトコンソールでは、Groovyのコードを実行中のJenkins内で実行することができ、Jenkinsの状態を確認したり設定を変更したりすることができます。

このような内視鏡手術は、groovy.lang.GroovyShellを使って簡単に実装できます。

```
def script = … // 外部から受け取ったGroovyスクリプト
def shell = new GroovyShell()
shell.evaluate(script)
```

クラスローダやバインディング変数、またImportCustomizerなどのコンパイラ設定などを設定することで、より特化した仕組みを提供することが出来ます。GroovyShellだけでも十分な内視鏡手術が可能になりますが、スクリプトは文字列でしか受け取れなかったり、HTTPで結果を返すときも文字列で返すことになります。Groovy Remote Controlを使うと、サーバ側にはGroovyのクロージャを渡すことができ、戻り値もオブジェクトのまま受け取ることができます。

■Hello World

それでは、Groovy Remote Controlを使ってHello Worldを書いてみましょう。コードはサーバ側とクライアント側の2つに分かれます。クライアント側ではHello Worldを表示するクロージャを送り、サーバ側では受け取ったクロージャを実行します。

●server.groovy

```
@Grab('org.codehaus.groovy.modules.remote:remote-
transport-http:0.3')
import com.sun.net.httpserver.HttpServer
import groovyx.remote.transport.http
  .RemoteControlHttpHandler
import groovyx.remote.server.Receiver

def receiver = new Receiver()
def handler = new RemoteControlHttpHandler(
  receiver)

def server = HttpServer.create(
  new InetSocketAddress(8080), 0)
server.createContext("/groovy-remote-control",
  handler)
server.start()
```

●client.groovy

```
@Grab('org.codehaus.groovy.modules.remote:remote-
transport-http:0.3')
import groovyx.remote.client.RemoteControl
import groovyx.remote.transport.http
  .HttpTransport

def transport = new HttpTransport(
  "http://localhost:8080/groovy-remote-control")

def remote = new RemoteControl(transport)
remote.exec {
    println "Hello World!"
}
```

サーバを起動した後で、クライアント側のコードを実行するとサーバ側で「Hello World!」と表示されたと思います。もしくは以下のようなエラーが出たかと思います。

```
Caught: java.lang.IllegalStateException: Could
not find class file for class class client$_run_
closure1
java.lang.IllegalStateException: Could not find
class file for class class client$_run_closure1
```

Groovy Remote Controlはクライアントからクロージャを送るときに、クロージャをシリアライズしてサーバ側に送ります。

サーバ側ではクロージャをデシリアライズするためにクロージャのクラスが必要になります。Groovy Remote Controlはクラスをサーバ側に送信するときにクロージャのクラスファイルを読み込んで、そのバイナリを一緒にサーバ側に送っています。このため、クライアント側のコードはコンパイルされてクラスファイルが生成されている必要があります。groovyコマンドやGroovyConsoleを使ってクライアント側のコードを実行してしまうと、クラスファイルは生成されないので上記のようなエラーが発生してしまいます。

コンパイルしてから実行するのは少し面倒なので、GroovyConsoleからでも実行できるように手を加えたGroovyConsoleを用意しました。groovyコマンドで次のようにして起動します。

```
groovy https://raw.github.com/kiy0taka/groovy-consoles/master/GroovyRemoteConsole.groovy
```

見た目は通常のGroovyConsoleですが、Groovy Remote Controlのクライアントとして使用できるように拡張しています。初回起動時はGroovy Remote Controlのライブラリを取得するので起動に時間がかかります。このGroovyConsoleにはgroovyx.remote.client.RemoteControlのインスタンスをバインディング変数に設定していますので、以下のコードだけで実行できるようになります。

```
remote.exec {
    println "Hello World!"
}
```

もう少し簡単に次のように書くことも出来ます。

```
remote {
    println "Hello World!"
}
```

戻り値については、クライアント/サーバの両方にあるクラスでシリアライズ可能なオブジェクトであれば返すことが出来ます。

```
assert remote.exec { 1 + 1 } == 2
```

クロージャの引数を渡すことは出来ませんが、戻り値と同じようにシリアライズ可能なオブジェクトであれば、リモートに渡すクロージャの中で使用することが出来ます。

```
def name = 'JGGUG'
remote.exec {
    def message = 'Hello ' + name + '!'
    println message
}
```

複数のクロージャをまとめて渡すことで、チェーンして実行することが出来ます。この場合、前のクロージャの結果が次のクロージャの引数となります。

```
assert 16 == remote({ 1 + 1 }, { it * 2 }, { it * it })
```

内視鏡手術として利用するためにはアプリケーションにサーバ側のコードを組み込む必要があります。GrailsやJenkinsにはそれぞれプラグインがあり、簡単にGroovy Remote Controlのサーバとしての機能を統合することが出来ます。

■Grailsで利用する

GrailsでGroovy Remote Controlと連携するためには、Remote Controlプラグイン（http://grails.org/plugin/remote-control）を利用します。Remote Controlプラグインの主な目的は、Grailsアプリケーションを立ち上げておき、そこに対してRemote Control経由でテストを実行するためのプラグインですが、普通にGroovy Remote Controlのサーバ側として利用することも可能です。

適当なGrailsアプリケーションを用意して、Remote Controlプラグインをインストールしましょう。記事執筆時点での最新バージョンは1.3です。BuildConfig.groovyに設定を追記しましょう。

```
compile ":remote-control:1.3"
```

または、コマンドからインストールします。

```
grails install-plugin remote-control
```

デフォルトではテスト環境でのみGroovy Remote Controlが有効になりますが、今回は普通にrun-appしたアプリケーションをコントロールするために、設定を変更します。Config.groovyに以下の設定を追記します。

```
remoteControl.enabled = true
```

Remote Controlプラグインを入れて上記の設定をした場合、本番環境も外部からコントロールできてしまうため危険です。必要に応じて環境ごとに設定をしましょう。

それではrun-appコマンドでGrailsアプリケーションを起動し、ブラウザで以下のURLにアクセスしてみましょう。

```
http://localhost:8080/アプリケーション名/grails-remote-control
```

405エラーが表示されると成功です。Groovy Remote ControlはGETメソッドはサポートしません。

それでは先ほどのGroovyConsoleからアクセスしてみましょう。GroovyConsoleのメニューから「Remote Control」→「Configuration」を選択し、先ほどブラウザでアクセスしたURLを設定します。Remote Controlプラグインを使うとリモートで実行されるクロージャの中でSpringのコンテキストやGrailsApplicationのインスタンスを利用することが出来ます。

例えば、リモートからGrailsのサービスを利用して処理を実行したい場合は、変数ctxを使用して以下のように実行します。

```
remote {
    // Springコンテキストからサービスを取得
    def service = ctx.myService
    // サービスのメソッドを実行
    service.serviceMthod()
}
```

リモートからGrailsの設定を動的に変更したい場合は、変数appを使用して以下のように実行します。

```
remote {
    // Configを取得
    def config = app.config
    // 設定を変更
    config.myconfig = 'foo'
}
```

ドメインを使う場合、クライアント側でドメインクラスをクラスパスに通しておけばそのまま使うことができますが、少し手間なのでSpringコンテキストからドメインのインスタンスを取得して利用できます。例えばBookというドメインクラスがあり、新しいBookを登録するには次のように実行します。

```
remote {
    // 新しいBookインスタンスを取得
    def book = ctx.getBean('myapp.Book')
    book.name = 'G*Magazine Vol.6'
    // 保存する
    book.save()
}
```

このように、実行中のGrailsアプリケーションに対してGroovy Remote Controlを利用して、設定の変更やデータの登録をすることができます。

■Jenkinsで利用する

Groovy Remote ControlはJenkinsでも利用することができます。Jenkinsで利用する場合はJenkinsのGroovy Remote Controlプラグイン（https://wiki.jenkins-ci.org/display/JENKINS/Groovy+Remote+Control+Plugin）をインストールします。JenkinsのRemote Controlにつなぐ場合は、以下のURLにアクセスします。

```
http://JenkinsのURL/plugin/groovy-remote/
```

ローカルでデフォルトのままJenkinsを起動している場合は「http://localhost:8080/plugin/groovy-remote/」になります。

リモートクロージャ内ではjenkinsという変数でJenkinsインスタンスを利用できます。

```
remote {
    // Jenkinsのバージョンを表示
    def version = jenkins.version.value
    pritnln version
}
```

このようにRemote Controlプラグインを入れることでJenkinsをリモートから操作することができるようになります。CLIやスクリプトコンソールとの違いは、クライアント側で処理した結果をそのままJenkinsに渡してビルドをしたり、Jenkins側から返ってきた結果を使ってクライアント側で処理をすることがよりシームレスにできるようになることです。

Remote Controlプラグインには他にもJenkinsから他のアプリケーションをRemote Control経由で操作する機能もあります。これによって、アプリケーションからJenkinsのコントロールと、Jenkinsからアプリケーションをコントロールするという双方向のコントロールが可能になります。Jenkinsから外部アプリケーションをコントロールする方法についてはプラグインのWikiページを参照してください。

■Grails以外のアプリケーションで利用する

Grailsにはプラグインがあるため簡単に利用できますが、Grails以外のアプリケーションでもGroovy Remote Controlを利用するための仕組みが用意されています。最初のサーバ側のサンプルで利用したRemoteControlHttpHandlerはJDKにバンドルされている簡易HTTPサーバ用のハンドラの実装です。他にサーブレットとして機能する仕組みもあります。WebアプリケーションにRemote Controlを組み込むためにはgroovyx.remote.transport.http.RemoteControlServletクラスを使用します。Groovy Remote Controlのライブラリをwarに含めてweb.xmlにRemoteControlServletの設定を追記するだけで使えます。

GrailsやJenkinsのプラグインのように、独自のバインディング変数を設定する場合は、RemoteControlServletを継承しcreateReceiver()メソッドをオーバーライドし、コンテキストを設定したgroovyx.remote.server.Receiver型のインスタンスを返します。

```
protected Receiver createReceiver() {
    def ctx = [date:new Date(),
        props:System.properties]
    new Receiver(ctx)
}
```

Receiverクラスでは他にクラスローダを変更することも可能です。サーバ側の処理のほとんどはこのReceiverが行っているため、フレームワーク固有のコントローラと統合することも可能です。

■まとめ

内視鏡手術は使い方を間違えるととても危険ですが、その反面とても強力な仕組みとなります。Groovy Remote Controlを使うことでより強力な内視鏡を簡単にアプリケーションに組み込むことができます。ぜひ、Groovy Remote Controlを使って内視鏡手術を行ってみてください。そして、内視鏡以外の使い方があればブログなどで紹介してもらえればと思います。

オレオレ・プログラミング GROOVY

series 12

綿引 琢磨（わたびき　たくま）
Groovy/Grails/Griffon/Java 界隈のやさぐれエンジニア。
日本 Grails/Groovy ユーザーグループ運営委員。

G* な皆さん、こんにちは。Groovy を学ぶ際に最初に読むべき書籍といえば「プログラミングGROOVY」ですよね？しかし、書籍で紹介されているのは、Groovy の魅力のごく一部に過ぎません。そこで、このコラムでは「プログラミングGROOVY」を読んで、これから Groovy を活用していこうと考えている人向けに、即戦力となるであろう API を紹介したいと思います。

Groovy のテンプレートエンジン

Java では、Apache の Velocity Engine が代表的なテンプレートエンジンとして知られていますが、Groovy では言語本体※でテンプレートエンジンの API を提供しています。テンプレートエンジンとは、ひな形となる定型のテンプレートにデータをバインドして、文字列を出力したり、ファイルの生成を行うための機能です。身近な例を挙げると、Grails の generate 系コマンドで生成されるコントローラやビューなどは、Groovy のテンプレートエンジンを利用して生成しています。

Groovy のテンプレートエンジンは groovy.text パッケージ（groovy-templates-x.x.x.jar）に格納されています。僅か 5 ファイルだけの小さなパッケージですので、それぞれについて解説します。

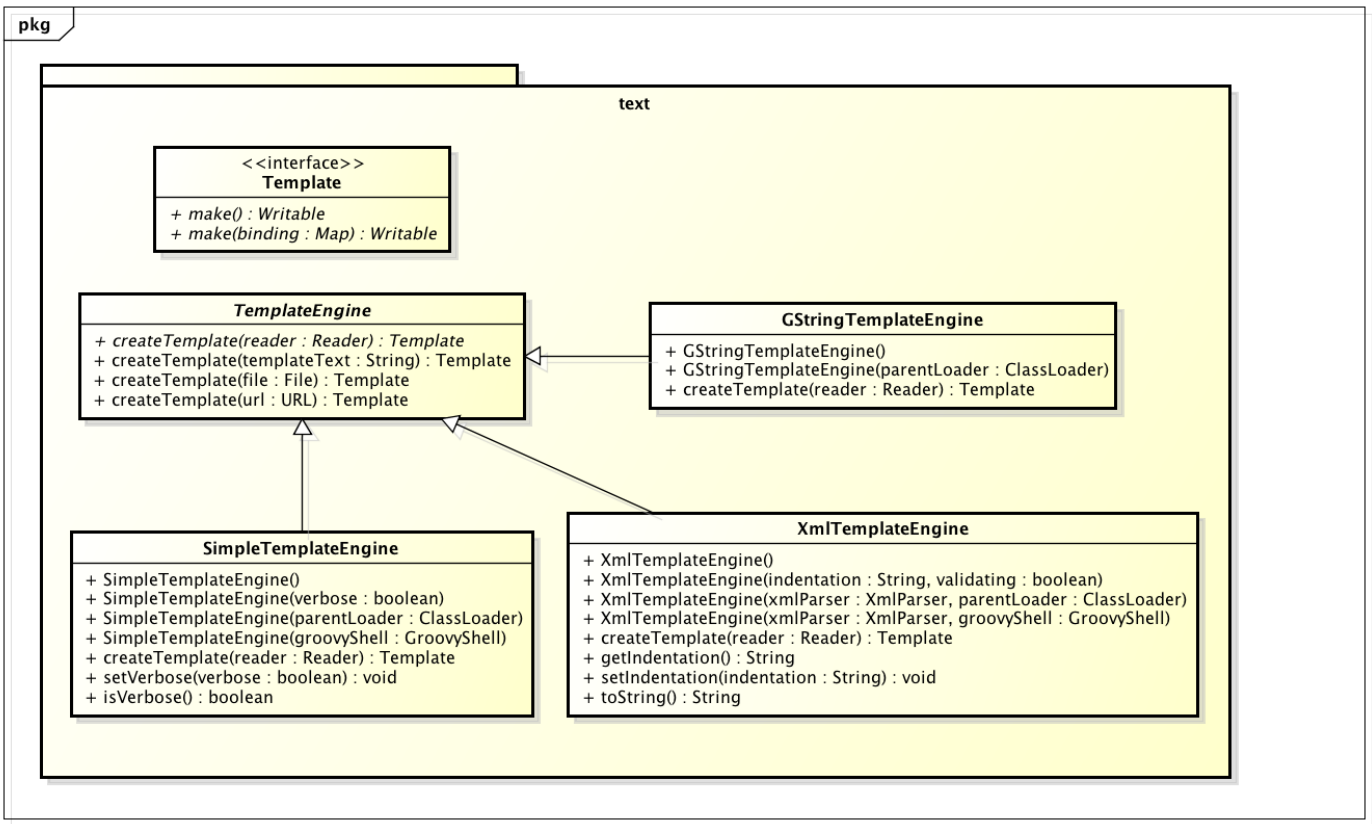
※Groovy 2.0 からのモジュール化により、物理的には別モジュールになっています。

■Template

テンプレートのインタフェースで、オーバーロードによる 2 つの make() メソッドが定義されています。このインタフェースの実装クラスの中でバインドしたものを出力するための Writable なクラスを生成します。

■TemplateEngine

テンプレートエンジンの抽象クラスで、オーバーロードによる 4 つの createTemplate() メソッドが定義されています。最終的には抽象メソッドとなっている Reader を引数とする createTemplate() が呼ばれ、具象クラスのテンプレートエンジンが生成されます。



groovy.text パッケージ概略図

■SimpleTemplateEngine

３つのテンプレートエンジンの中で最も汎用的に使えるテンプレートエンジンです。JSP形式（<% %>, <%= %>）とGString形式（${}）のプレースホルダを記述できます。例えば、テンプレートファイルを読み込んでクラスファイルを生成するコードは以下のようになります。

●テンプレート

```
package $packageName

class $className {
<% fields.each { type, fieldName -> %>
    private $type $fieldName<% } %>
}
```

● サンプルコード

```
import groovy.text.SimpleTemplateEngine

def path
  = '/Users/bikisuke/sandbox/templateEngine'
def className = 'SampleBean'
def binding = [
        packageName:'org.jggug.bean',
        className:className,
        fields:['String':'name', 'int':'value']
    ]
def file = new File("${path}/simple.template")
def engine = new SimpleTemplateEngine()
def template = engine.createTemplate(file)
  .make(binding)
new File("${path}/${className}.groovy")
  .write(template.toString())
```

●実行結果

```
package org.jggug.bean

class SampleBean {

    private String name
    private int value
}
```

SimpleTemplateEngine では、デバッグ確認用の verbose フラグがあり、コンストラクタまたは setVerbose() で値を true にすることで、実行時にテンプレートを解析した結果を標準出力することができます。前述のテンプレートの場合は以下のような出力になります。

```
-- script source --
out.print("""package $packageName

class $className {
"""); fields.each { type, fieldName -> ;
out.print("""
    private $type $fieldName"""); } ;
out.print("""
}
""");

/* Generated by SimpleTemplateEngine */
-- script end --
```

■GStringTemplateEngine

GStringTemplateEngine は SimpleTemplateEngine と同等の機能を持ちながら、クロージャを用いることで、SimpleTemplateEngine よりも拡張性のあるテンプレートエンジンとなっています。クロージャ内では出力を out という変数に委譲するため、スクリプトレットで out 変数を使いながらロジックを記述することができます。一つのスクリプトレット内で値の判定をさせて out に出力すると、SimpleTemplateEngine よりも柔軟で読みやすいテンプレートを作成することができます。

●テンプレート

```
$familyName $firstName 様、おめでとうございます。

厳正な抽選の結果、<%= familyName %>様に
<% out << (hasLoan? ' 7億' : ' 3億') %>円プレゼントが当選しました。

ご連絡いただければ、すぐにお振込いたしますので、
下記 URL へのアクセスお願いいたします。
http://akutoku.com
```

●サンプルコード

```
import groovy.text.GStringTemplateEngine

def path
  = '/Users/bikisuke/sandbox/templateEngine'
def binding = [
        familyName:'御金',
        firstName:'内造',
        hasLoan:true
    ]
def file = new File("${path}/gstring.template")
def engine = new GStringTemplateEngine()
def template = engine.createTemplate(file)
  .make(binding)
println template.toString()
```

● 実行結果

```
御金 内造 様、おめでとうございます。

厳正な抽選の結果、御金様に
7億円プレゼントが当選しました。

ご連絡いただければ、すぐにお振込いたしますので、
下記 URL へのアクセスお願いいたします。
http://akutoku.com
```

■XmlTemplateEngine

3つ目は名前の通り XML を出力するためのテンプレートエンジンです。テンプレートの解析に XmlParser を使用し、テンプレート自体も XML 書式で記述します。このテンプレートでは <gsp:scriptlet> と <gsp:expression> の2つのタグを使ってデータとのバインドを行います。

●テンプレート

```
<?xml version="1.0" encoding="UTF-8"?>
<WizardRings xmlns:gsp='http://groovy.codehaus.
org/2005/gsp'>
  <gsp:scriptlet>rings.each {</gsp:scriptlet>
  <TransformationRing style="${it.style}">
    <stone element="${it.stone.element}" >
      <gsp:expression>it.stone.base</
gsp:expression>
    </stone>
    <chants><gsp:expression>it.chants</
gsp:expression></chants>
  </TransformationRing>
  <gsp:scriptlet>}</gsp:scriptlet>
</WizardRings>
```

●サンプルコード

```
import groovy.text.XmlTemplateEngine

def path
  = '/Users/bikisuke/sandbox/templateEngine'
def rings = []
rings << [style:'Flame', stone:[element:'Fire',
base:'Ruby'], chants:'Hi']
rings << [style:'Water', stone:[element:'Water',
base:'Sapphire'], chants:'Sui']
rings << [style:'Hurricane',
stone:[element:'Wind', base:'Emerald'],
chants:'Fu']
rings << [style:'Land', stone:[element:'Earth',
base:'Yellow-Topaz'], chants:'Do and Don']

def file = new File("${path}/xml.template")
def engine = new XmlTemplateEngine()
def template = engine.createTemplate(file)
  .make([rings:rings])
println template.toString()
```

● 実行結果

```
<WizardRings>
  <TransformationRing style='Flame'>
    <stone element='Fire'>
      Ruby
    </stone>
    <chants>
      Hi
    </chants>
  </TransformationRing>
  <TransformationRing style='Water'>
    <stone element='Water'>
      Sapphire
    </stone>
    <chants>
      Sui
    </chants>
  </TransformationRing>
  <TransformationRing style='Hurricane'>
    <stone element='Wind'>
      Emerald
    </stone>
    <chants>
      Fu
    </chants>
  </TransformationRing>
  <TransformationRing style='Land'>
    <stone element='Earth'>
      Yellow-Topaz
    </stone>
    <chants>
      Do and Don
    </chants>
  </TransformationRing>
</WizardRings>
```

テンプレートエンジンの使いどころ

SimpleTemplateEngine は、オールラウンドで使用することができるので、ソースコード生成のような自動生成ツールを作成する際には非常に役に立つクラスです。筆者はこれまで生成ツールをいくつか作成してきましたが、使い勝手の良さからSimpleTemplateEngine を使うケースが多いです。

GStringTemplateEngine は、データの値に応じて出力時の内容を変更させたいような場合に有効です。ただし、テンプレートでのロジックが実装しやすい反面、肥大したテンプレートになってしまう可能性もあるため、簡単なデータ変換や出力元の切替など効果的に利用すると良いと思います。

XmlTemplateEngine は使いどころは限られますが、XML に特化したテンプレートエンジンですので、XML を出力する際には検討する価値はあると思います。

まとめ

簡単ではありますが、Groovy のテンプレートエンジンをおさらいしてみました。2.x 系になって、魅力的な新機能が増えていますが、まだまだ Groovy にはたくさんの実用的な機能があります。皆さんも API を眺めてみて、オレオレGROOVY を探求してはいかがでしょうか。

『Yokohama.groovy』について
～活動報告など～

しんや（@shinyaa31）

はじめまして。しんや（@shinyaa31）と申します。横浜在住、都内勤務のエンジニアです。

2年程前から社外の勉強会に参加するようになりました。そして今年（2012年）の夏に『**Yokohama.groovy**』というコミュニティを作り読書会・写経会という形で勉強会を実施、継続しています。

今回はその『Yokohama.groovy』について、発足の経緯や実施内容等についてご紹介してみたいと思います。

Yokohama.groovy発足のきっかけ

これまでに、私自身 #jggug が主催するイベント（主にG*ワークショップですが）には数回足を運んでいました。そこでの内容は毎回とても参考になる事が多く得るものも多かったのですが、基本的には『聴講』スタイル。Groovy界隈に限った事では無かったのですが、手を動かして何かしらのアウトプットを出し、より理解を深める機会が欲しいなと思うようになりました。

2012年に入り、『アジャイルサムライ読書会　横浜道場』（以下、“横浜道場”）等で横浜界隈での勉強会参加者と交流する機会も多くなり、そこでのやり取りを経て、『横浜をベースに何か勉強（会）をやろう！』という流れに。イベントの名前はシンプルに『Yokohama.groovy』。対象となったメンバー間での興味関心がGroovyに多かった事で比較的すんなり決まりました。

会の概要等

会の名前が示す通り、開催場所は横浜近辺としています。これは構成が横浜道場メンバーである事が多分に影響しています。

開催ペースは現在のところ隔週開催。今後については不明ですが、メンバー間で都合の良い曜日を決め、調整が付けやすいように曜日を固定して続けています。幸いここまで、日程微調整が少しあったものの、割と安定したペースで続けられています。

開催時間帯については20:00～22:00という遅目の時間帯にしました。これは前述の横浜道場にも言える事なのですが、横浜・神奈川在住者が都内に勤務する場合どうしても横浜方面に帰宅するのは若干遅目の時間帯になってしまう事から開始時間を遅らせて（その分終了・懇親会の時間もシフトし）より参加し易くしています。どの勉強会に於いても19:00、19:30開催等が多く、定時との兼ね合いだと『行きたいけど時間の都合で…』というケースも割と多く聞きますので“時間帯を遅らせて開催する”というのは横浜・神奈川に限らず、都内開催に於いても参加者増に効果があるのではないでしょうか。

開催場所は現在まで、ほぼ大半を『**タネマキ**』で執り行っています。

タネマキ【コワーキング & シェアオフィススペース】
http://tane-maki.net/

本棚にはプログラマの必読書？「ジョジョ」全巻が！

2011年末に横浜駅西口方面にオープンしたコワーキングスペースで、寛ぎやすい空間の中、横浜道場譲り（？）のゆる～い雰囲気で会を進行しております。

会の進行に関しては、冒頭にも書いたように『読むだけではなく手や頭を動かして理解を深める』事を主目的としています。なので読書会＆写経会、というスタイルが近いでしょうか。当回に読む範囲を定めておき、当日は気になる所をかいつまみながら時には深掘り、時には横道に逸れつつ…と言った感じで“持続可能なペースで”続けてきました。

対象図書第1弾は『プログラミングGroovy』

対象図書については、G*ワークショップ等でも常々テーマや話題にも挙がり、教材とするには絶好の書籍だろう、という訳でこちらも迷いなく対象書籍を『プログラミングGroovy』とする事に。第3章～第5章の深掘り可能なテーマがメインとなりました。

2012年度中のテーマは『プログラミングGroovy』

これまでの活動記録

第1回からのイベント参加記録を以下に並べてみます。どのような内容で開催されて来たか、雰囲気だけでも掴んで頂ければ幸いです（※個人ブログで申し訳ありません…）。

第1回 2012/08/07
http://d.hatena.ne.jp/absj31/20120807/1344484292
第2回 2012/08/20
http://d.hatena.ne.jp/absj31/20120820/1345539843
第3回 2012/09/04
http://d.hatena.ne.jp/absj31/20120904/1346856008
第4回 2012/09/18
http://d.hatena.ne.jp/absj31/20120918/1348888991
第5回 2012/10/02
http://d.hatena.ne.jp/absj31/20121002/1349193477
第6回 2012/10/16
http://d.hatena.ne.jp/absj31/20121016/1350402234
第7回 2012/10/30
http://d.hatena.ne.jp/absj31/20121030/1351611802
第8回 2012/11/13
http://d.hatena.ne.jp/absj31/20121113/1352871838
第9回 2012/11/27
http://d.hatena.ne.jp/y_sumida/20121127/1354423092
第10回 2012/12/18
http://d.hatena.ne.jp/y_sumida/20121218/1356238928

開催のハードルは極力低く

昨今本当に数が多くなった勉強会の開催数ですが、勉強会を開催する上でも幾つかのハードルがあると思います。開催地の問題、時間の問題、長期化・シリーズ化するような規模の場合は如何に勉強会自体を継続させて行くかの問題等など…。

Yokohama.groovyでは参加メンバーが皆勉強会参加の“常連”でもあったため、その辺の事情を把握していたのでこの点については『なるべく開催のコストを掛けない方向で』と一致していました。また人数についても小回りの効く規模に留めています（3〜6名）。これは主に利用している会場（横浜タネマキ）の都合もありますが、いざという時の予定調整も行いやすいです。場所についても、コワーキングスペースを利用する事で開催場所決定の手間を削減しています。現在の人数規模感であれば、いざとなったら喫茶店等でも開催可能ですし。書籍に関する予習等は行うものの、基本それ以外の手間等は極力掛けないような体制で行こうという部分を崩さないで居るのも、ここまで定期的に続けられている要因なのかなと思います。

もっとG*関連イベントを！

私の観測範囲（ここ1〜2年、主に関東）内だと、Groovy, G*関連の実践系勉強会と言えば以下のようなイメージでした（※この他にも開催してるぞ！と思われた方居ましたら申し訳ありません…m(_ _)m）。

- G*ワークショップ
- TDDBC等でのテスト系イベントの対象言語の1つとして
- JGGUG主催の合宿系イベント
- @kyon_mm氏によるGroovy関連イベント
- その他、単発でちらほらと…

そんな感じでしたので、最近になって『Groovy/G*関連の実践系勉強会少ないな〜、もう少しあっても良いのに/あると嬉しいのに…』という思いは確かにありました。

今回この原稿を書くにあたり綿引さん（@bikisuke）にその辺りのお話を伺ったところ、『以前はG*ワークショップや関連イベントでも、実践系のものをやってたりしていた』そうです（存じ上げておりませんでした…）。

書籍『プログラミングGroovy』で紹介されている『Groovyエコシステム』の諸技術、またはそれ以外にもGroovy/G*界隈には魅力的且つ実用的な技術が数多く存在していると思います。状況や各種課題等もある事かと思いますが、“文字通り”ワークショップやハンズオンの機会が増え、Groovy/G*の技術がより多くの人に使われるようになるとまた面白い展開も出来てくるのではないでしょうか。

Yokohama.groovyの今後について

2012年12月一杯で『プログラミングGroovy』の読書・写経会についてはひとまず終了。

2013年からは活動曜日を平日から土日に移し、午後イチ〜夕方の長めの時間を取って、Groovy及びG*プロダクトに関する『もくもく作業』メインの中身となりました。時間も内容もより余裕を持たせる事によって、より充実した方向へとシフトして行こうと思っています。

Twitterハッシュタグ：#yokohamagroovy で適宜情報は発信していく予定ですので、興味のある方はチェックしてみてください。

また、実施しているイベントについて参加してみたいという方がおりましたら、告知サイト等の告知があればそちらを是非ご参照ください。またそうでない場合も参加メンバーに気軽に声を掛けてみてください。

活動拠点『タネマキ』の手書き看板。

Grails Plugin 探訪

第 7 回

~ Remote Methods プラグイン ~

URL: http://grails.org/plugin/grails-remote-methods
プラグインのバージョン: 0.2
対応する Grails のバージョン: 2.0 以上

杉浦孝博

最近は Grails を使用したシステムの保守をしている自称プログラマ。
日本 Grails/Groovy ユーザーグループ事務局長。
共著『Grails 徹底入門』、共訳『Groovy イン・アクション』

はじめに

今回ご紹介する Grails プラグインは、Remote Methods プラグインです。

本記事は、次の環境で動作確認をしております。

- OS : Mac OS X 10.8.2
- Java : 1.7.0_10
- Grails : 2.2.0

なお、コマンドの実行結果については、紙面の都合上、出力結果を省略しており、実際の出力と異なる場合があります。ご了承願います。

Remote Methods プラグインとは

Remote Methods プラグインは、コードを書かずに JavaScript からコントローラのメソッドを呼び出すコードを生成するためのプラグインです。

プラグインのインストール

Remote Methods プラグインのインストールは、grails の install-plugin コマンドでインストールするか、あるいは BuildConfig.groovy に記述することで行います。

install-plugin コマンドでインストールする場合、次のように行います。

```
$ grails create-app appgp07
$ cd appgp07
$ grails install-plugin grails-remote-methods
```

BuildConfig.groovy に記述する場合、次のように記述します。

```
    plugins {
        ....
        compile ":grails-remote-methods:0.2"
    }
```

サンプルのドメインクラスとコントローラ

今回のサンプルとして、本をあらわす Book ドメインクラスと、そのコントローラである BookController を使用します。

```
package appgp07

class Book {
    String title
    int price
    String isbn13
    static constraints = {
        title blank: false
        price min: 0
        isbn13 blank: false, maxSize: 13
    }
}
```

```
package appgp07

class BookController {

// (省略)

    def getTitle() {
        // (省略)
    }

    def getPrice() {
        // (省略)
    }

    def getIsbn13() {
        // (省略)
    }
}
```

メソッドの定義とGSPの記述

メソッドの定義と、それを呼び出すタグをGSPに記述します。

タグの記述方法には大きく二通りの記述方法があります。呼び出すメソッドを明示する方法と、呼び出すメソッドを明示しない方法です。

■呼び出すメソッドを明示する

メソッドのリモート呼び出しのタグ名は「rm:defineRemote」となります。呼び出すコントローラはcontroller属性で、メソッド名はmethods属性で指定します。methods属性はリストで、複数指定することができます。

```
<rm:defineRemote controller="book"
methods="['getTitle', 'getPrice', 'getIsbn13']"/>
```

■呼び出すメソッドを明示しない

呼び出すコントローラはcontroller属性で指定します。

```
<rm:defineRemote controller="book"/>
```

メソッド呼び出しの対象となるメソッドを、コントローラのremoteMethodsプロパティで指定します。

```
class BookController {
    static remoteMethods
        = ['getTitle', 'getPrice', 'getIsbn13']
```

なお、GSPでmethods属性を指定し、かつremoteMethodsプロパティも指定した場合は、methods属性の指定内容が優先されます。

JavaScriptコードの生成

rm:defineRemoteタグによって、以下のコードが生成されます。

例えば、次のGSPの記述から

```
<rm:defineRemote controller="book"
    methods="['getTitle']"/>
```

実行時に次のJavaScriptコードが生成されます（紙面の都合上、コードは整形しています）。

```
<script src="/appgp07/static/js/application.js"
  type="text/javascript" ></script>
<script type="text/javascript">
var book = {
  getTitle: function (params, success, error) {
    jQuery.ajax({
      type: 'POST',
      data: params,
      url: '/appgp07/book/getTitle',
      success: function(data, textStatus) {
        success(data, textStatus);
      },
      error: function(XMLHttpRequest, textStatus,
        errorThrown) {
          if (error) {
            error(XMLHttpRequest,
                    textStatus, errorThrown)
          } else {
            window.errorHandler(XMLHttpRequest,
                          textStatus, errorThrown)
          };
        }
    });
  },
  _: 0
}
</script>
```

生成されるコードは、jQueryを使用したAjaxのコードとなります。

呼び出すコードの作成

最後に、生成されたJavaScriptコードを呼び出すコードをGSPに記述します。

```
<script type="text/javascript">
// 省略

function getTitle(params) {
    book.getTitle(params, function() {},
        function() {})
}

// 省略
</script>
```

コードリーディング

プラグインの機能の説明は以上となります。プラグインがしていることは、簡単に言ってしまえば、タグリブ(のクラス)を提供することになります。

タグリブのコード自体短いものなので、次に前ソースを載せます。ここからは、簡単ではありますが、タグリブの仕組みを説明したいと思います。

```
package remote.method

class UtilsTagLib {    // [1]

    def static namespace = 'rm'    // [2]

    def grailsApplication

    def defineRemote = { attrs ->    // [3]
        javascript(null, {    // [6]
            def controller = getController(attrs.controller)
            def methods = getMethods(controller, attrs.methods)

            out << "var ${controller.controllerName} = {"    // [4]
            methods.each {
                out << "${it}:function (params, success, error) {"
                out << remoteFunction(    // [6]
                        controller: controller.controllerName,
                        action: it,
                        params: "params",
                        onSuccess: "success(data, textStatus)",
                        onFailure: """
                    if (error) {
                        error(XMLHttpRequest,textStatus,errorThrown)
                    } else {
                        window.errorHandler(XMLHttpRequest,textStatus,errorThrown)
                    }
                    """
                )
                out << "},"
            }
            out << "_:0}"

            null
        })
    }

    private getController(name) {    // [5]
        return grailsApplication.mainContext.getBean(
            grailsApplication
                .getArtefactByLogicalPropertyName
                    ("Controller", name ?: controllerName).clazz.name
        )
    }

    private getMethods(controller, methods) {    // [5]
        if (controller.hasProperty('remoteMethods') == null &&
            methods == null) {
            throw new IllegalArgumentException(
                "In tag 'defineRemote' you should specify methods attribute " +
                "in tag or define list methods in controller")
        }

        methods ?: controller.remoteMethods
    }
}
```

■ファイルの置く場所とクラス名

タグリブのファイルを置く場所は、grails-app/taglibディレクトリ配下になります。

タグリブのクラス名は「～TagLib」で終わるように、ファイル名は「～TagLib.groovy」で終わるように命名します(コード［1］)。

■タグの名前空間とタグの定義

タグの名前空間の接頭辞は、namespaceプロパティで指定します(コード［2］)。

タグの定義は、タグリブのクラスにクロージャで定義します(コード［3］)。クロージャとバインドされた変数名が、タグの名前となります。

タグ定義のクロージャはパラメータを1つ受け取り、タグに記述された属性のマップとなります(コード［3］)。

■out変数

タグ定義のクロージャで処理した内容は、out変数に出力することで、GSPの処理結果の一部としてクライアントに返すことができます(コード［4］)。

■メソッドの定義

タグリブとはいえ、通常のクラスと変わりありませんので、メソッドの定義をすることができます(コード［5］)。

■他のタグリブの呼び出し

javascriptやremoteFunctionといったタグリブを、通常のメソッド呼び出しと同じ使い方で利用することができます(コード［6］)。この場合、呼び先のタグリブでout変数に出力をしていれば、それらの内容もout変数に出力順に出力されます。

終わりに

今回はタグの記述のみで、Ajaxでの単純なメソッド呼び出しができるようになるプラグインをご紹介しました。凝った呼び出し方をする場合には物足りないかもしれませんが、お手軽にメソッド呼び出しの記述ができるのは魅力的ではないかと思います。

あと、今回は機能や実装がシンプルだったこともありましたので、コードリーディングをしてみましたがいかがだったでしょうか。今後も、機会があればやってみたいと思います。

▼ぐるーびーたん　第5話までのあらすじ

プログラマを目指して熊本から上京したぐるーびーたん。今度はJGGUG温泉合宿に参加!?

※この作品は、たいがいフィクションです。実在の人物、団体とは関係ありません。

リリース情報 2013.02.19

Grails

Grailsは、GroovyやHibernateなどをベースとしたフルスタックのWebアプリケーションフレームワークです。
URL: http://grails.org/
バージョン: 1.3.9, 2.1.3, 2.2.0

■更新情報

- 2.1.3では、grailsラッパーが使えない問題やfindAllクエリが失敗する問題などのバグ対応が行われています。
- 2.1.3リリースノート: http://grails.org/2.1.3+Release+Notes
- 山本さんのブログ: http://d.hatena.ne.jp/mottsnite/20121219/1355931008
- 2.2.0では、Groovy 2.0対応や、ネームスペース対応、フォークモードの他、各種バグ対応が行われています。
- 2.2.0リリースノート: http://grails.org/2.2.0+Release+Notes
- 山本さんのブログ: http://d.hatena.ne.jp/mottsnite/20121220/1356016199

Groovy

Groovyは、JavaVM上で動作する動的言語です。
URL: http://groovy.codehaus.org/
バージョン: 1.8.9, 2.0.7, 2.1.1

■更新情報

- 1.8.9では、GStringにgetBytes()メソッドが追加されたり、@ImmutableがCloneableフィールドと一緒に使えるようになったり、多くのバグ対応が行われています。
- 1.8.9リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=18778
- 2.0.7では、Fileクラスに createTempDir()メソッドが追加されたり、ソースファイルのクラスパスを検索するかどうかを制御するためにgroovyc antタスク用フラグが提供されたり、多くのバグ対応が行われています。
- 2.0.7リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=19028
- 2.1.1では、@Delegateを@DelegatesToと一緒に動作できるようになったり、バイトコード周りが幾つか改良されたり、多くのバグ対応が行われています。
- 2.1.1リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=18990

Griffon

Griffonは、デスクトップアプリケーションを開発するためのアプリケーションフレームワークです。
URL: http://griffon.codehaus.org/
バージョン: 1.2.0

■更新情報

- 1.2.0では、リリースパッケージをアップロードするコマンドが追加されたり、カスタマイズしたログ用のファクトリサービスが追加されたり、いくつかのバグ対応が行われています。
- 1.2.0リリースノート: http://docs.codehaus.org/display/GRIFFON/Griffon+1.2.0#Griffon120-120

Gant

Gantは、XMLの代わりにGroovyでAntタスクを記述し実行するビルド管理ツールです。
URL: http://gant.codehaus.org/
バージョン: 1.9.9

■更新情報

- 1.9.9では、-Dオプションに=が含まれる場合にパースが正しくない問題が対応されています。
- 1.9.9リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=11660&version=19046

Gradle

Gradleは、Groovyでビルドスクリプトを記述し実行するビルド管理ツールです。
URL: http://www.gradle.org/
バージョン: 1.4

■更新情報

- 1.4では、依存解決が改良されたり、GroovyCompile/Groovydocタスクを使ってGroovyの依存性を自動構成したり、いくつかバグ対応が行われています。
- 1.4リリースノート: http://www.gradle.org/docs/1.4/release-notes

Gaelyk

Gaelykは、Groovyで記述するGoogle App Engine for Java用のライトウェイトなフレームワークです。
URL: http://gaelyk.appspot.com/
バージョン: 1.2

Google App Engine SDK for Java

Google App Engine SDK for Javaは、JavaでGoogle App Engine用のWebアプリケーションを開発するためのSDKです。
URL: http://code.google.com/intl/ja/appengine/
バージョン: 1.7.5

■更新情報

- 1.7.5では、F4_1GとB4_1Gという新しいインスタンスが利用可能になったり、DataNucleusプラグインが2.1.2にバージョンアップされたり、いくつかバグ対応が行われています。
- 1.7.5リリースノート: http://code.google.com/p/googleappengine/wiki/SdkForJavaReleaseNotes

GPars

GParsは、Groovyに直感的で安全な並行処理を提供するシステムです。
URL: http://gpars.codehaus.org/
バージョン: 1.0.0GA

■更新情報

- 1.0.0GAでは、dataflow用に同期チャネルが追加されたり、静的にディスパッチするアクターが追加されたり、いくつかバグ対応が行われています。
- 1.0.0GAリリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=12030&version=17007

Groovy++

Groovy++は、Groovy言語に対して静的な機能を拡張します。
URL: http://code.google.com/p/groovypptest/
バージョン: 0.9.0

Spock

Spockは、JavaやGroovy用のテストと仕様のためのフレームワークです。
URL: http://code.google.com/p/spock/
バージョン: 0.7

■更新情報

- 0.7では、リファレンスドキュメントが新しくなったり、スタブやモックがいろいろと改良されたり、Groovy 2.x対応されたりしています。
- 0.7リリースメール: http://docs.spockframework.org/en/spock-0.7-groovy-2.0/new_and_noteworthy.html#id1

GroovyServ

GroovyServは、Groovy処理系をサーバとして動作させることでgroovyコマンドの起動を見た目上高速化するものです。

URL: http://kobo.github.com/groovyserv/
バージョン: 0.11

■更新情報

- 0.11 では、リモートアクセスがサポートされたり、“cookie”が“authtoken”にリネームされたり、いくつかバグ対応が行われています。
- 0.11 リリースノート: http://kobo.github.com/groovyserv/changelog.html#ref-changelog

Geb

Gebは、Groovyを使用したWebブラウザを自動化する仕組みです。
URL: http://www.gebish.org/
バージョン: 0.7.2

■更新情報

- 0.7.2 では、Java 7 で実行した場合にGebAssertionErrorが発生する問題の対応が行われています。
- 0.7.2 リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=12101&version=18756

Easyb

Easybは、ビヘイビア駆動開発(Behavior Driven Development:BDD)用のフレームワークです。
URL: http://www.easyb.org/
バージョン: 0.9.8

Gmock

Gmockは、Groovy用のモック・フレームワークです。
URL: http://code.google.com/p/gmock/
バージョン: 0.8.2

HTTPBuilder

HTTPBuilderは、HTTPベースのリソースに簡単にアクセスするための方法です。
URL: http://groovy.codehaus.org/modules/http-builder/
バージョン: 0.6.0

■更新情報

- 0.6.0 では、HttpClientが4.2.1に、Nekohtmlが1.9.16に対応し、JSONの扱いがjsonlibからJsonSlurperに変更となりました。
- 0.6.0 チェンジログ: http://groovy.codehaus.org/modules/http-builder/changes.html

CodeNarc

CodeNarcは、Groovy向けの静的コード解析ツールです。
URL: http://codenarc.sourceforge.net/
バージョン: 0.18.1

■更新情報

- 0.18.1 では、SpaceAroundOperatorRuleやBracesForClassRuleなど、いくつかバグ対応が行われています。
- 0.18.1 リリースノート: http://sourceforge.net/projects/codenarc/files/codenarc/CodeNarc%20Version%200.18.1/

GMetrics

GMetricsは、Groovyソースコードのサイズや複雑さを計算したり報告するためのツールです。
URL: http://gmetrics.sourceforge.net/
バージョン: 0.6

■更新情報

- 0.6では、メトリクスやフューチャーが追加されたり、メトリクスのインタフェースが変更されたり、いくつかバグ対応が行われています。
- 0.6 リリースノート: http://sourceforge.net/projects/gmetrics/files/gmetrics-0.6/

GContracts

GContractsは、Groovyで契約プログラミングを行うためのフレームワークです。
URL: https://github.com/andresteingress/gcontracts
バージョン: 1.2.10

■更新情報

- 1.2.10では、Groovy 1.8.8やJava 7で使った時のバグ対応など、いくつかバグ対応が行われています。
- 1.2.10 リリースメール: http://blog.andresteingress.com/2012/10/17/gcontracts-1-2-10-released/

GroovyFX

GroovyFXは、JavaFXをGroovyで書きやすくするためのフレームワークです。
URL: http://groovyfx.org/
バージョン: 0.3.1

■更新情報

- 0.3.1 での変更箇所は不明です。

GBench

GBenchは、Groovyのためのベンチマーク・フレームワークです。
URL: http://code.google.com/p/gbench/
バージョン: 0.4.0

■更新情報

- 0.4.0 では、パッケージ名が "groovyx.gbench" となったり、BenchmarkBuilderを実行する拡張モジュールが追加されたり、@Benchmarkアノテーションと BenchmarkBuilderの両方のパフォーマンスが改良されたりしています。
- 0.4.0 リリースノート: http://code.google.com/p/gbench/wiki/ReleaseNotes040

Betamax

Betamaxは、HTTP通信の内容を保存し再生するテストツールです。
URL: http://freeside.co/betamax/
バージョン: 1.1.2

■更新情報

- 1.1.2 での変更箇所は不明です。

Caelyf

Caelyfは、Groovyで記述するCloud Foundry用のライトウェイトなツールキットです。
URL: http://caelyf.cloudfoundry.com/
バージョン: 0.1

Vert.x

Vert.xは、非同期アプリケーション開発のためのフレームワークです。
URL: http://vertx.io/
バージョン: 1.3.1

■更新情報

- 1.3.1 では、coffee-script.jsを事前にコンパイルしたり、sockjsから切断する前にunregisterフックを呼び出すようにしたり、各種バグ対応が行われています。
- 1.3.1 リリースissue: https://github.com/vert-x/vert.x/issues?milestone=25&state=closed

GVM (Groovy enVironment Manager)

様々なGroovy関連のツールをインストールし、複数のバージョンを切り替えて使用するためのツールです。
URL: http://gvmtool.net/
バージョン: -

■企業スポンサー

NTTソフトウェア

Delight Technologies

NEWCAST

metabolics

■個人サポーター

須江 信洋 様

関谷 和愛 様

田中　明 様

個人サポータ制度のお知らせ

JGGUGでは,昨年に引き続き2013年度も個人サポータを募集いたします。.

個人サポータとなっていただいた方には、一年間にわたってJGGUGが発行するG* Magazine（年数回刊。基本的に電子版として配布予定）に個人サポータとしてお名前を掲載します（掲載を希望しない旨お申し出いただけば掲載しません）。

個人サポータとなるには、まずsupporters@jggug.orgにメイルで

- お名前
- 予定金額

G* Magazineへのご芳名掲載の可否

をお知らせください。追って、運営委員より振込先の情報などを返信します。

皆様のサポートをお待ちしております。

日本 Grails/Groovy ユーザーグループ
代表　山田 正樹

G* Magazine vol.6 2013.02
http://www.jggug.org

発行人：日本 Grails/Groovy ユーザーグループ
編集長：川原正隆
編集：G* Magazine 編集委員（杉浦孝博、奥清隆）
デザイン：㈱ニューキャスト
表紙：川原正隆
編集協力：JGGUG 運営委員会
Mail：info@jggug.org

Publisher：Japan Grails/Groovy User Group
Editor in Chief：Masataka Kawahara
Editors：G* Magazine Editors Team
(Takahiro Sugiura, Kiyotaka Oku)
Design：NEWCAST inc.
CoverDesign：Masataka Kawahara
Cooperation：JGGUG Steering Committee
Mail：info@jggug.org