G Magazine

vol.7

Japan Grails/Groovy User Group

Contents

Grails 2.3.0 新時代への幕開け。

series 14

山本　剛（やまもと つよし／株式会社ニューキャスト）

出版・印刷関連のシステム設計開発等に従事するテクニカルDTPアーキテクト。
日本Grails/Groovyユーザーグループ名古屋支部長。2006年よりGrailsのドキュメント翻訳、
その後、Grails公式のAcegiプラグインを開発。書籍『Grails徹底入門』(翔泳社発行)の9、10、11章を執筆。

Grails 2.3.0 リリース。新時代への幕開け。

2013年9月10日、Grails 2.3.0がリリースされました。このバージョンでは次期バージョン3.0をターゲットに、大幅に改善・向上が行われています。例えば、以前から課題になっていたXSS問題を解決すべくGSP関連のエスケーピングの考え方、Grailsの振る舞いに適したリクエストバインディング、さらには今後Webアプリケーションには必須となるRESTの実装、開発時の動作を改善するための仕組み改良などです。

この記事では、そんな次世代へ向けたGrails 2.3.0の更新内容を、公式ドキュメントの内容を拝借しながら紹介したいと思います。

Grails 2.3.0 の更新内容

始めにGrails 2.3.0での更新概要を以下に示します。整頓すると、大まかに分けると五つの項目に分類できます。

コントローラ・ビュー

- コントローラにネームスペース定義が可能に。
- コントローラ例外ハンドリング。
- フレキシブルで保守性の高く簡単な新データバインダー。
- リクエストボディをコマンドオブジェクトにバインド。
- ドメインクラスをコマンドオブジェクトとして使用。
- エンコーディング/エスケーピングの向上 - クロスサイトスクリプティング（XSS）防御を実装。
- URLマッピングでリダイレクトが定義可能に。

REST

- サーバサイドRESTサポート大幅改良。

開発支援

- 依存管理の向上 - 依存管理エンジンをAetherに変更。
- フォーク実行 - 全てのメジャーなコマンドが別々のJVMへフォーク。
- テストランナーデーモン。
- Unitテストのデフォルトが Spockに。
- デフォルトでのgrailswラッパー生成。

コア機能

- 非同期サポート - リクエストの非同期処理とGORMとシームレスに統合した新たな非同期プログラミングAPI。

新規プラグイン

- Hibernate3と4をサポート。
- スカッフォルディングがプラグインへ移動。

その中でも、今回（個人的にですが）注目しておきたいポイントは、REST機能！と言いたいですが、やはり、いちばん実装に直結してくる、コントローラ・ビューとバインド廻りの実装だと思います。先ずはその辺りから追っていきましょう。

■コントローラ・ビュー実装が便利にそしてセキュアに

●コントローラネームスペース定義

Grailsは2.3以前では、例えばjp.grails.UserControllerとjp.grails.admin.UserControllerのようにパッケージ違い同名称のコントローラを作成するとができませんでした。（※元々できないと思ってるので同名称で作る事も無かったのですが）

Grails 2.3から「コントローラネームスペース定義」が実装されたことにより、別々のパッケージでの同じ名称のコントローラを使用できるようになりました。使い方は以下のようになります。

以下、異なる二つのAdminControllerがあるとします。それぞれに異なる名称のnamespaceプロパティを定義します。

```
package com.app.reporting // パッケージ違い
class AdminController {
  static namespace = 'reports' //namespaceプロパティ
  // …
}
```

```
package com.app.security // パッケージ違い
class AdminController {
  static namespace = 'users' //namespaceプロパティ
  // …
}
```

UrlMappings.groovyにてURL定義するさいに namespace を指定します。

```
class UrlMappings {
    static mappings = {
        '/userAdmin' {
            controller = 'admin'
            namespace = 'users'
        }

        '/reportAdmin' {
            controller = 'admin'
            namespace = 'reports'
        }

        "/$namespace/$controller/$action?"()
    }
}
```

リンクタグではnamespace属性に指定することができます。

```
<g:link controller="admin" namespace="reports">
    Click For Report Admin
</g:link>
<g:link controller="admin" namespace="users">
    Click For User Admin
</g:link>
```

●コントローラ例外ハンドリング

コントローラアクションが投げたExceptionをハンドリングするメソッドをコントローラに定義できるようになりました。例外処理をコントローラ内で共通化するのに便利です。

```
package demo
class DemoController {

    def someAction() {
        // アクション実装
    }

    def handleSQLException(SQLException e) {
        render 'SQLExceptionをハンドリング'
    }

    def handleBatchUpdateException(
        BatchUpdateException e) {
        redirect controller: 'logging',
        action: 'batchProblem'
    }

    def handleNumberFormatException(
        NumberFormatException nfe) {
        [problemDescription:
            'A Number Was Invalid']
    }
}
```

●コントローラはシングルトン

Grails 2.3から、新規アプリケーション作成で、Config.groovyでの grails.controllers.defaultScope の指定が'singleton'となっています。以前のバージョンまでは'prototype'でした。

```
grails.controllers.defaultScope = 'singleton'
```

デフォルトがシングルトンになるので、アクションの実装はメソッド推奨です。

一応ドキュメントの注意事項を、ここにも転記しておきます。

「コントローラのスコープは使用状況を考えてお使いください。例えば、シングルトンのコントローラは、_全ての_リクエストで共有されるためプロパティを持つべきではありません。デフォルトで定義されているスコープのprototype を変更することで、プラグインなどで提供されたコントローラがprotopypeを前提で実装されている場合があるので注意しましょう。」

●新データバインダ

Grails 2.3では、以前よりもフレキシブルで保守性の高くシンプルなデータバインダを提供しています。このデータバインダの実装は今までのSpringを使用したの実装とは違い、Grailsに必要なデータバインダとして新たに開発されています。もちろん下位互換として旧データバインダを使用する事も可能です。旧データバインダを使用するには、grails-app/conf/Config.groovy に次のように指定します。

```
grails.databinding.useSpringBinder=true
```

では、何が追加されたか見ていきましょう。

フィールドごとのカスタム日付フォーマット

特定のフィールドに対してBindingFormatアノテーションで日付フォーマットを指定できます

```
import org.grails.databinding.BindingFormat
class Person {
    @BindingFormat('MMddyyyy')
    Date birthDate
}
```

グローバル定義としてConfig.groovyに設定することも可能です。次のようにマッチさせたいフォーマットをリスト順に渡します。

```
grails.databinding.dateFormats = ['MMddyyyy',
    'yyyy-MM-dd HH:mm:ss.S',
    "yyyy-MM-dd'T'hh:mm:ss'Z'"]
```

フィールドにBindingFormatが指定されている場合は、BindingFormatの内容が優先となります。

他に、日付に関してはi18n/messages.propertiesを使用してローカライズ定義も可能です。例えば次の例では、i18nに定義

したdate.formats.birthdaysの値からローカライズ別に判定してバインディングを行います。

```
# grails-app/conf/i18n/messages.properties
date.formats.birthdays=MMddyyyy
# grails-app/conf/i18n/messages_ja.properties
date.formats.birthdays=yyyyMMdd
```

以下が使用方法。

```
class Person {
    @BindingFormat(code='date.formats.birthdays')
    Date birthDate
}
```

ユーザ定義可能なデータコンバータ

データコンバートを行うためには、内部的に自動で多くの自動型変換を行っていますが、アプリケーションの実装によっては独自の変換が必要となります。そのようなケースのためにデータコンバータを定義できるようになりました。

仕組みとして簡単です。ValueConverterを実装したクラスを作成して、resources.groovyでビーン定義して、Springコンテキストに登録するだけです。例として次のコードを見てみましょう。（※サンプルコードに賛否両論あるとは思いますが。。。）

例えば次の２つのドメインクラスがあるとします。

```
class Address {
    Zipcode zipcode
    String pref
    String city
}
class Zipcode {
    String zip3
    String zip4

    String toString(){
        return "${zip3}-${zip4}"
    }
}
```

Addressのデータバインドで、郵便番号zipcodeを"100-0001"のように文字列で受けた物を、Zipcode側のzip3（３桁）とzip4（４桁）それぞれのフィールドに入れるとします。

```
def address = new Address()
address.properties = [zipcode:'123-0001',
    pref:'東京都', city:'品川区']
```

この動作でそれぞれのフィールドにバインドできるようにValueConverterを用意します。コードの配置場所はsrc/groovy以下で問題無いです。

```
class ZipcodeValueConverter implements
ValueConverter {

    @Override
    boolean canConvert(Object o) {
        o instanceof String
    }

    @Override
    Object convert(Object o) {
        def zips = o.split('-')
        return new Zipcode(zip3:zips[0],
            zip4:zips[1])
    }

    @Override
    Class<?> getTargetType() {
        return Zipcode
    }
}
```

コンバータを作成したらビーンの登録定義をします。ビーン名はzipcodeConverterとしていますが何でも構いません。

```
beans = {
    zipcodeConverter ZipcodeValueConverter
}
```

これで先程のようなデータコンバートが可能になります。

```
def address = new Address()
address.properties = [zipcode:'123-0001',
    pref:'東京都', city:'品川区']

assert address.zipcode.zip3 == '123'
```

このようにユーザ定義データコンバータは、フォーム等からリクエストに投げられた内容を、特定したモデル型に対してコンバートするのには便利な仕組みだと思います。ただ何もかも当てはめると、これでも最低限で済んでいると思いますが、少々コードが増えますね。

ユーザ定義フォーマットデータコンバータ

BindingFormatの機能を拡張することで、簡単なデータフォーマッタをユーザ定義実装することができます。これはFormattedValueConverterを実装することで、データバインド時に文字列などを加工することができます。

公式ドキュメントの例では、以下のコードのような、大文字小文字コンバートを例としています。

FormattedValueConverterの実装クラスを作成します。Class

getTargetType() で返す型に対してのコンバータとして認識されます。このサンプルの場合はString型です。def convert（value, String format）メソッドの２番目引数でformatとあります。このformat取得できる内容で分岐を記述して、それぞれの処理を実装します。BindingFormatアノテーションの引数がformatで参照できる原理となっています。

```
import org.grails.databinding.converters.
FormattedValueConverter
class FormattedStringValueConverter implements
FormattedValueConverter {
    def convert(value, String format) {
        if('UPPERCASE' == format) {
            value = value.toUpperCase()
        } else if('LOWERCASE' == format) {
            value = value.toLowerCase()
        }
        value
    }

    Class getTargetType() {
        // specifies the type to which this
        // converter may be applied
        String
    }
}
```

こちらの実装コードも先程のValueConverterと同様に、resources.groovyに定義します。

```
beans = {
  formattedStringConverter FormattedStringValueConverter
}
```

使用方法は次のようにBindingFormatアノテーションに指定します。

```
class Person {
  @BindingFormat('UPPERCASE') // UPPERCASEでの実装を処理
  String someUpperCaseString

  @BindingFormat('LOWERCASE') //LOWERCASEでの実装を処理
  String someLowerCaseString

  String someOtherString
}
```

ここまでに説明してきた、これらのValueConverter系でのカスタムコンバートは、もちろんGrailsのバインディングのコアである部分とも同じ方式を使用しています。深い部分に興味がある場合はGrailsのソースを覗いてみるのも良いかもしれません。手始めにorg.grails.databinding.SimpleDataBinder、org.codehaus.groovy.grails.web.binding.GrailsWebDataBinder、org.grails.databinding.ClosureValueConverterなどをみてみるとよいのでは。

クラス・フィールドごとのカスタムバインディング

BindUsingアノテーションを使用して特定のクラスまたはフィールドに対して専用のバインディングを定義することができるようになりました。

特定のクラスに対しては、BindUsingアノテーションに、BindingHelperを実装したクラスを指定してクラスに定義します。

以下の例はGrailsソースコード用のテストコードorg.grails.databinding.BindUsingSpecから拝借しました。

実装の概要は、各フォールド名がマッチしたら、それぞれ特定の演算を行う内容となっています。ClassWithBindUsingクラスは、BindUsingアノテーションにMultiplyingBindingHelperを指定してあります。

```
@BindUsing(MultiplyingBindingHelper)
class ClassWithBindUsing {
    Integer leaveIt
    Integer doubleIt
    Integer tripleIt
}
```

MultiplyingBindingHelperは次のように、BindHelperの実装として作成します。

```
class MultiplyingBindingHelper implements
BindingHelper<Integer> {
  Integer getPropertyValue(Object obj,
  String propertyName,
  DataBindingSource source) {
    def value = source[propertyName]
    def convertedValue = value
    switch(propertyName) {
      case 'doubleIt':
        convertedValue = value * 2 //値を２倍する
        break
      case 'tripleIt':
        convertedValue = value * 3 //値を３倍する
        break
    }
    convertedValue
  }
}
```

このコードを実行するとBindUsingでの変換が行われて、結果は次のようになります。

```
def obj = new ClassWithBindUsing()
obj.properties = [doubleIt: 9, tripleIt: 20,
    leaveIt: 30]

assert obj.doubleIt == 18 // 与えた値9に対しての2倍
assert obj.tripleIt == 60 // 与えた値9に対しての3倍
assert obj.leaveIt == 30
```

このMultiplyingBindingHelperを使用するのはクラス全体への実装となりますが、特定のフィールドに対してBindUsingアノテーションを使用する場合は次のようになります。

BindUsingアノテーションの引数に、クロージャでコードを直接記述します。

```
class ClassWithBindUsingOnProperty {
    @BindUsing({
        obj,
        source -> source['name']?.toUpperCase()
    })
    String name
}
```

このコード例の結果は、toUpperCaseと有るようにnameに与えられた内容は全て大文字となります。

デフォルトでの空文字列とトリミング

デフォルトで全ての空文字列はデータバインディングでnullに変換されます。そして前後の空白をトリムします。これらの設定はConfig.groovy変更が可能です。デフォルトでは両方とオン（true）になっています。

```
// デフォルトはtrueです。
grails.databinding.trimStrings = false
// デフォルトtrueです。
grails.databinding.convertEmptyStringsToNull =
false
```

●コマンドオブジェクト

リクエストボディをコマンドオブジェクトにバインド

コントローラアクションがコマンドオブジェクトを引数として定義されている状態で、リクエストがボディを含んでいる場合、ボディの内容はパースされコマンドオブジェクトにデータバインドされます。この機能を使用すると、コマンドオブジェクトに対応したJSONまたXMLをリクエストボディとしての送信を受け取る等の実装が非常に楽になります。例として公式ドキュメントより。

次のように、コントローラとアクションの引数にコマンドオブジェクトを指定します。

```
class DemoController {
  def createWidget(Widget w) {
    render "Name: ${w?.name}, Size: ${w?.size}"
  }
}
class Widget {
  String name
  Integer size
}
```

リクエストがボディを含んでいるリクエストをcurlなどで実行。

```
$ curl -H "Content-Type: application/json"
-d '{"name":"Some Widget","size":"42"}'
localhost:8080/myapp/demo/createWidget
```

結果はコントローラのrenderで返したようになります。

```
Name: Some Widget, Size: 42
```

ドメインクラスをコマンドオブジェクトとして使用

同じく、でも若干異なりますが、アクションのコマンドオブジェクトとしてドメインクラスを引数として定義されている状態で、リクエストパラメータにidが存在した場合は、自動的にidを使用してデータベースから、対象のドメインクラスインスタンスを取り出します。

次のようなPersonドメインクラスがあったとします。

```
class Person {
    String name
}
```

以下のようにPersonをアクションの引数に与えます。

```
class PersonController {
    def updatePerson(Person person) {
        println person
       //リクエストパラメータに id が有れば
       //データベースから自動取得。
    }
}
```

つまり、このドメインクラスを指定した場合の挙動では、def updatePerson(Person person)での、personはドメインクラスPersonのインスタンスが必ず生成されます。

●エンコーディング/エスケーピングの向上

Grails 2.3 では専用の クロスサイトスクリプティング（XSS）防御を実装しています。大まかには以下の内容となります。

- デフォルトでGSPエクスプレッションとスクリプトレットをHTMLエスケーピング
- タグ固有のエンコーディング定義
- 多重エンコーディング防止
- GSPページ内の全てのデータに対して安全と判断しない場合の自動エンコーディング

デフォルトがHTMLエスケープされます

GSPでの全ての ${} はデフォルトでHTMLエスケープされるようになりました。

信頼できる値をエスケープしないようにするには、raw() メソッドを使用します。

```
${raw(page.content)}
```

エンコーディング/エスケーピング設定

Grails 2.3ではConfig.groovyに以下の内容が追加されています。

```
// GSP settings
grails {
    views {
        gsp {
            encoding = 'UTF-8'
            htmlcodec = 'xml'
            codecs {
                expression = 'html'
                scriptlet = 'html'
                taglib = 'none'
                staticparts = 'none'
            }
        }
        filteringCodecForContentType {
            //'text/html' = 'html'
        }
    }
}
```

- expression - ${..} エクスプレッションを対象にしたコーデックの設定。
- scriptlet - スクリプトレット（<% %>, <%= %> ブロック）を対象にしたコーデックの設定。
- taglib - タグライブラリを対象にしたコーデックの設定。タグを作成するさいに定義ができるのと下位互換のためデフォルトはnone。
- staticparts - GSP内の静的マークアップを対象にしたコーデックの設定。
- filteringCodecForContentType - もっとセキュリティを高めるために、全てのレスポンス出力に対しての指定もできます。デフォルトではオフになっています。この設定を有効にした場合は、staticpartsの指定を'raw'にする必要があります。

特定のプラグインに対して設定えお行う場合は、次のように例えばFooプラグインを対象の場合は、最初に foo. を付加します。

```
foo.grails.views.gsp.codecs.expression = "none"
```

ページ固有の定義

各GSPページに固有の定義も可能です。

```
<%@page expressionCodec="none" %>
```

タグライブラリでのエンコーディング定義

各タグライブラリでは"defaultEncodeAs"プロパティを指定することで個別のコーデックを指定できます。

タグライブラリクラス全体に定義する場合は、defaultEncodeAsで指定します。

```
static defaultEncodeAs = 'html'
```

各タグ（メソッド）に定義する場合は、encodeAsForTagsでそれぞれ定義することが可能です。

```
static encodeAsForTags = [tagName: 'raw']
```

また、タグを使用する際に指定することもできます。

```
<g:message code="foo.bar" encodeAs="JavaScript" />
```

コード内で異なるエンコーディングに切り替える事も可能です。次の例はHTMLタグはhtmlコーデックのまま、Javascriptのコードに対してJavascriptコーデックを指定する例です。

```
out.println '<script type="text/javascript">'
withCodec("JavaScript") {
    out << body()
}
out.println()
out.println '</script>'/>
```

このように、細かな振る舞いができるようになり、より安全なWebアプリケーションが構築可能です。まだデフォルトの設定では完全とは言えませんが、下位互換を考えての事だと思います。ただXSSに関して言うと今後のWebの進化と共にまだまだ問題点も出てきそうですが、このGrails 2.3からの実装では一安心といったとこでしょうか。

●URLマッピングでリダイレクトが定義可能に

URLマッピングでリダイレクトが指定できるようになりました。以下の例のように redirect: で指定します。

```
class UrlMappings {
  static mappings = {
    "/viewBooks"(redirect: '/books/list')
    "/viewAuthors"(redirect:
      [controller: 'author', action: 'list'])
    "/viewPublishers"(redirect:
      [controller: 'publisher', action: 'list',
      permanent: true])
  }
}
```

URLマッピングは今回のアップデートで他にも多くの追加実装があります。これからのWebアプリケーションに欠かせないREST関連の実装です。REST関連の実装に関しては、記事の続きのREST関連部分で紹介します。

■サーバサイドRESTサポート大幅改良

GrailsのREST関連機能は大幅に改良され、さらにRESTが身近で開発が容易になるような新機能が多数追加されています。

- ドメインクラスでのRESTリソース定義。
- URLマッピングでのRESTリソースマッピング用機能追加。
  - シングルリソース、ネストリソース、バージョニング。
- RestfulControllerスーパークラス。
- 拡張可能なレスポンスレンダリングとバインディングAPI。
- RESTコントローラのスカッフォルディング。
- HAL, Atom, Hypermedia（HATEAOS）対応。

RESTサポートの実装内容はかなり広範囲になるので、深い部分は今後機会があれば別で記事にするとします。今回はRESTサポート内容の一部概要を解説します。詳しくは公式ドキュメントを参考にしてください。有志による日本語ドキュメントプロジェクトで日本語版があります！ http://bit.ly/1bsfjJv

では、概要的に一部だけ紹介します。

●ドメインクラスでのRESTリソース定義

ドメインクラスにResourceアノテーションを指定するだけで、RESTFul APIをもっと簡単に作成する方法が新機能として追加されています。次の例のように grails.rest.Resource と引数に対象の uri 指定を追加します。

```
import grails.rest.Resource

@Resource(uri='/tasks')
class Task {
    String title
    Date period
    Boolean done
    static constraints = {
        period nullable:true
        done nullable:true
    }
}
```

この指定をするだけで、RESTリソースマッピングの設定とドメインと同じ名称のコントローラ（例の場合はTaskController）を、自動的に動的（※実ファイルでは無く）作成します。

例のように、@Resource(uri='/tasks') のみ指定するとJSONとXML両方に対応したRESTful APIを生成します。そしてデフォルトではXMLを返すようになります。JSONを取得したい場合はURLにConfig.groovyの grails.mime.types に設定されているJSON識別用拡張子追加します。例えば http://localhost:8080/example/tasks の場合は、http://localhost:8080/example/tasks.json となります。この指定は formats の指定をすることで順序を変更することが可能です。次のようにjsonを先に指定すると拡張子なしでjsonを取得できます。

```
@Resource(uri='/tasks', formats=['json', 'xml'])
// jsonを先にする
class Task {
....
```

Grailsのコンテントネゴシエーションの機能も連動しているため、コンテントタイプの指定は拡張子のみでは無くACCEPTヘッダで指定することもできます。

```
curl -i -H "Accept: application/json" http://
localhost:8080/example/tasks
```

ACCEPTヘッダで指定しても拡張子を指定した場合は拡張子が優先となります。

簡単なGETはもちろん他にPOST（作成）,PUT（更新）,DELETE（削除）などメソッドを変えることで対象ドメインクラスをRESTfulに操作出来ます。

```
curl -i -X POST -H "Content-Type: application/
json" -d '{"title":"色鉛筆を買って帰る"}'
localhost:8080/example/tasks
curl -i -X PUT -H "Content-Type: application/
json" -d '{"title":"赤鉛筆を買って帰る"}'
localhost:8080/example/tasks/1
curl -i -X DELETE localhost:8080/example/tasks/1
```

そして、単純にドメイン参照用だけのAPIにしたい場合はreadOnlyを指定することで読み取り専用にできます。

```
@Resource(uri='/tasks', readOnly=true) //
readOnlyをtrueにする
class Task {
....
```

ドメインクラスに1行指定する程度でできるとか簡単で便利ですね。ただ、簡単に終わらせられる物だけでは無いので、いろいろな実装は必要と思いますが、それにしても必要最低限の機能は揃っている思います。ここから先はカスタマイズなどを行うときに使用できる機能の紹介です。

●URLマッピングでのRESTリソースマッピング

先ほどのは機能は、ドメインクラスに @Resource(uri='/tasks') のみ指定でURLマッピングを自動で設定してくれるという便利な物でしたが、直接URLマッピングを定義することができます。

次のようにUrlMappings.groovyに1行追加します。

```
"/tasks"(resources:"task")
```

この1行のみでいちばん単純なRESTリソースマッピングを設定することができます。

どのようなURLが実際に設定されたのかを確認してみます。確認には、URLマッピングレポート用コマンド url-mappings-report を実行します。シェルコンソールの設定よってはカラフルに、次のような内容が表示されます。（※このコマンドでは、ドメインクラスにResourceアノテーションを付加した内容はリストされないようです。）

```
| URL Mappings Configured for Application
| ---------------------------------------

Dynamic Mappings
  |    *     | /${controller}/${action}?/${id}?(.${format})?              | Action: (default action) |
  |    *     | /                                                          | View:   /index           |
  |    *     | ERROR: 500                                                 | View:   /error           |

Controller: dbdoc
  |    *     | /dbdoc/${section}?/${filename}?/${table}?/${column}?       | Action: (default action) |

Controller: task
  |   GET    | /tasks                                                     | Action: index            |
  |   GET    | /tasks/create                                              | Action: create           |
  |   POST   | /tasks                                                     | Action: save             |
  |   GET    | /tasks/${id}                                               | Action: show             |
  |   GET    | /tasks/${id}/edit                                          | Action: edit             |
  |   PUT    | /tasks/${id}                                               | Action: update           |
  |  DELETE  | /tasks/${id}                                               | Action: delete           |
```

内容を説明すると表になっている最初のカラムがメソッド、その次が ${id} などの変数を間に含んだURL、そして最後が対象コントローラでのアクション名になっています。開発の順番はそれぞれですが、先に UrlMappings.groovy へ必要なRESTリソースマッピングを設定しておいて、レポートの内容を見ながら実装していく何てことも想像できます。

RESTリソースマッピングでは includes また excludes を指定することで、指定した対象のみ、あるいは除外することが可能です。

アクションindexとshowのみ。

```
"/tasks"(resources:"task", includes:\['index', 'show'])
```

更新削除系のアクションdeleteとupdateを除外。

```
"/tasks"(resources:"task", excludes:\['delete', 'update'])
```

この簡単な resources: の指定だけでなく、他にもRESTリソースマッピングの設定機能があります。

シングルリソース

先ほどの resources: 指定に似ていますが、"s"が無くなっただけの resource: で指定すると意味が変わりシングルリソースの設定になります。

```
"/info"(resource:'personalInfo')
```

次のようになります。

```
Controller: personalInfo
 |   GET    | /info/create                                          | Action: create          |
 |   POST   | /info                                                 | Action: save            |
 |   GET    | /info                                                 | Action: show            |
 |   GET    | /info/edit                                            | Action: edit            |
 |   PUT    | /info                                                 | Action: update          |
 |  DELETE  | /info                                                 | Action: delete          |
```

最初の resources: との違いは ${id} の変数指定が含まれない状態になります。例えば単一な情報API用などに使用します。

ネストリソース

子リソースを持つ場合に使用します。

```
"/books"(resources:'book') {
  "/authors"(resources:"author")
}
```

この指定をすると、次のようなURLマッピングになります。

```
Controller: author
 |   GET    | /books/${bookId}/authors                              | Action: index           |
 |   GET    | /books/${bookId}/authors/create                       | Action: create          |
 |   POST   | /books/${bookId}/authors                              | Action: save            |
 |   GET    | /books/${bookId}/authors/${id}                        | Action: show            |
 |   GET    | /books/${bookId}/authors/${id}/edit                   | Action: edit            |
 |   PUT    | /books/${bookId}/authors/${id}                        | Action: update          |
 |  DELETE  | /books/${bookId}/authors/${id}                        | Action: delete          |

Controller: book
 |   GET    | /books                                                | Action: index           |
 |   GET    | /books/create                                         | Action: create          |
 |   POST   | /books                                                | Action: save            |
 |   GET    | /books/${id}                                          | Action: show            |
 |   GET    | /books/${id}/edit                                     | Action: edit            |
 |   PUT    | /books/${id}                                          | Action: update          |
 |  DELETE  | /books/${id}                                          | Action: delete          |
```

通常のURLマッピングも指定できます。

```
"/books"(resources: "book") {
    "/publisher"(controller:"publisher")
}
```

次のようなURLマッピングになります。

```
Controller: publisher
 |    *     | /books/${bookId}/publisher                            | Action: (default action) |

Controller: book
 |   GET    | /books                                                | Action: index           |
 |   GET    | /books/create                                         | Action: create          |
 |   POST   | /books                                                | Action: save            |
 |   GET    | /books/${id}                                          | Action: show            |
 |   GET    | /books/${id}/edit                                     | Action: edit            |
 |   PUT    | /books/${id}                                          | Action: update          |
 |  DELETE  | /books/${id}                                          | Action: delete          |
```

RESTリソースのバージョニング

REST APIの実装バージョンを増やすなどの実装もURLマッピングで指定できます。一般的な例としては、次のように、namespace: 指定をする方法があります。

```
"/books/v1"(resources:"book", namespace:'v1')
"/books/v2"(resources:"book", namespace:'v2')
```

コントローラ側にも namespace の指定をすることで実装できます。

```
package myapp.v1
class BookController {
    static namespace = 'v1'
}

package myapp.v2

class BookController {
    static namespace = 'v2'
}
```

Accept-Versionヘッダを使ったバージョンニングもサポートしています。この場合は version: を指定します。

```
"/books"(version:'1.0', resources:"book", namespace:'v1')
"/books"(version:'2.0', resources:"book", namespace:'v2')
```

次のようにURLではなく Accept-Version: 1.0 ヘッダでバージョンを指定してアクセスします。

```
curl -i -H "Accept-Version: 1.0" -X GET http://
localhost:8080/myapp/books
```

バージョンの指定方法としては、ハイパーメディア/MIMEタイプを使ったバージョニングに対応させることもできます。この辺りに関しては公式ドキュメントを参考にしてください。

●RestfulController

RESTリソースマッピングの方法がわかった所で、ここからはRESTfulコントローラ実装機能の話です。

RESTfulコントローラを実装するために便利なのが、Grails 2.3で追加されたスーパークラス RestfulController です。

RESTリソースに対してコントローラを実装するには、次のように RestfulController の拡張を実装します。

```
class BookController extends RestfulController {
    static responseFormats = ['json', 'xml']
    BookController() {
        super(Book)
    }
}
```

基本的には、この例のように記述すれば、必要な機能は動作します。いわゆる scaffold と同じような感覚です。実装をカスタマイズするには、カスタマイズ対象のアクションメソッドをオーバーライドします。アクションの対応表をのせておきます。

GET	/books	index
GET	/books/create	create
POST	/books	save
GET	/books/$\{id}	show
GET	/books/$\{id}/edit	edit
PUT	/books/$\{id}	update
DELETE	/books/$\{id}	delete

RestfulControllerについて、詳しくは公式ドキュメントも参考にしてください。（http://bit.ly/1gDjyjc）

もちろん RestfulController を使用しなくてもRESTfulなコントローラは開発可能です。今まで通りコントローラアクションを実装すれば良いわけですが、公式ドキュメントが非常にわかりやすいので、是非そちらを参考にしてください。（http://bit.ly/18cl6yz）

●レスポンスレンダリング

respondメソッド

公式ドキュメントの「RESTコントローラの実装ステップバイステップ（http://bit.ly/18cl6yz）」の一部にも記載されている重要な部分でGrails 2.3の新機能でもある「レスポンスレンダリング」。この機能がアクションからのレスポンスを返す重要な機能となっています。

アクションで使用する respond メソッドはリクエストのコンテントタイプを判別して、JSON、XML、HTMLなどの適切なレスポンスを行います。この機能は今までも実装されていたコンテントネゴシエーション機能の withFormat と似ていますが、よりRESTfulな実装をスッキリと記述できるようになっています。

```
// この1行でコンテントタイプを判別してレスポンスします。
respond Book.get(1)
// formatsの配列を指定するとその中から判別して
// レスポンスします。
respond Book.get(1), [formats:['xml', 'json']]
```

使用方法は、先ず1つ目のパラメータは、レンダリング対象のオブジェクトで、2つ目にその他の引数をMapで渡します。その他の引数には、formats: , model: , view: などを設定できます。model: , view: に関してはHTMLでレンダリングする場合に使用します。

RESTレンダリングカスタマイズ

XML・JSONのレンダラーは grails.rest.render.xml , grails.rest.render.json のパッケージ内にあり、これらは grails.converters.XML と grails.converters.JSON を使用してレスポンスをレンダリングします。このレンダラーを使用してレスポンスのカスタマイズを行います。

```
import grails.rest.render.xml.*
beans = {
    bookRenderer(XmlRenderer, Book) {
        excludes = ['isbn']
    }
}
```

このようにビーンを追加してカスタマイズ実装します。他にも以前からXML・JSONコンバータのカスタマイズ方法が幾つか存在します。マーシャラレジスタ・ObjectMarshaller・コンバータAPI・AbstractRenderer・ContainerRenderer・GSP使用など全部説明をすると少し話が長くなるので、詳しくは公式ドキュメントを参照してください。

●RESTコントローラのスカッフォルド

とりあえずGrails 2.3のRESTを試したい場合は、もちろんRESTのスカッフォルド生成も準備されています。

```
grails generate-async-controller example.Book
```

コードを書き出して参考にしてみると良いと思います。

●HAL, Atom, Hypermedia（HATEAOS）対応

SpringでのAPIなどのラッパーとしているのか、自分も勉強不足であまり聞き慣れていない、HAL・HATEAOSなどに、Grails 2.3から対応しています。

HATEOASとHALに関しては以下を参照してください。

- HATEOAS - http://en.wikipedia.org/wiki/HATEOAS
- HAL - http://stateless.co/hal_specification.html
- Grails公式ドキュメントから「アプリケーション状態エンジンとしてのハイパーメディア」- http://bit.ly/15FDmkH

■非同期プログラミングAPI

Grails 2.3では、Promiseの概念と統一されたイベントモデルを取り入れることで、フレームワーク内での並行プログラミングを簡単にする非同期プログラミングAPIを提供しています。

（※公式ドキュメントより抜粋して紹介します。詳しくは有志が翻訳をした日本語版ドキュメントを参照しましょう！http://bit.ly/158V6kH）

今回追加された内容は、Grails 2.3ドキュメントでの分類からみると以下の項目に分けられます。

- Promises
- 非同期GORM
- 非同期リクエストハンドリング

「RESTコントローラのスカッフォルド」でも解説しましたが、スカッフォルド生成ができますので、実際にコードを生成して参考にすると、実装例としてわかりやすいと思います。

```
grails generate-async-controller example.Book
```

ドキュメントに多く記載されていますので、一部を見出しにそってダイジェストで紹介します。 公式ドキュメントはコチラ http://bit.ly/158V6kH

●Promise API

Grails 2.3では、例外ハンドリングモデルや、非同期処理の連鎖、リスナーの追加などの便利な機能を持つ、Promiseの概念と統一されたイベントモデルを取り入れ並行プログラミングを簡単にする非同期機能を追加しました。その実装として

grails.async.Promisesクラスを提供しています。

```
import static grails.async.Promises.*
```

promiseを作成するには、grails.async.Promiseインタフェースのインタンスを返すtaskメソッドを使用します。

現在のスレッドをブロックした状態で処理を待つ場合はwaitAllメソッドを使用します。

```
def p1 = task { 2 * 2 }
def p2 = task { 4 * 4 }
def p3 = task { 8 * 8 }
assert [4,16,64] == waitAll(p1, p2, p3)
```

スレッドをブロックしたくない場合はonCompleteメソッドを使用します。

```
def p1 = task { 2 * 2 }
def p2 = task { 4 * 4 }
def p3 = task { 8 * 8 }

onComplete([p1,p2,p3]) { List results ->
   assert [4,16,64] == results
}

def promiseList = tasks { 2 * 2 }, { 4 * 4}, { 8 * 8 }
assert [4,16,64] == promiseList.get()
```

Promiseのチェーン

thenメソッドを使用してpromiseをチェーンする事が可能です。途中のチェーンで例外が発生した場合に次のチェーンが呼ばれることはありません。

```
final polish = { … }
final transform = { … }
final save = { … }
final notify = { … }
Promise promise = task {
    // 長い実行タスク
}
promise.then polish then transform then save then {
    // 最終結果の呼出
}
```

Promiseのリストとマップ

リストとマップの概念を、ぞれぞれ grails.async.PromiseList 、grails.async.PromiseMap のクラスとして実装されています。

tasksメソッドを使用してpromiseのリストやマップを作成できます。

```
def promiseList = tasks { 2 * 2 }, { 4 * 4}, { 8 * 8 }
def promiseMap = tasks one:{ 2 * 2 },
                       two:{ 4 * 4},
                       three:{ 8 * 8 }
```

Promiseファクトリ

Promiseインスタンスを生成するためのファクトリを、Promises.promiseFactoryの値を変更することで差し替える事が可能です。

```
import org.grails.async.factory.*
import grails.async.*
Promises.promiseFactory = new SynchronousPromiseFactory()
```

DelegateAsync変換

同じAPIにおいて、同期API/非同期APIの両方が必要となる場合において、DelegateAsync変換を使用して同期APIを非同期APIに変換することが可能です。

```
//サービス
class BookService {
    List<Book> findBooks(String title) {
    }
}
// 非同期API - AsyncBookServiceとして@DelegateAsync追加
import grails.async.*
class AsyncBookService {
   @DelegateAsync BookService bookService
}
```

●非同期GORM

Grails 2.3から、GORMでサポートされているすべてのデータストアに対して、横断的に動作する非同期プログラミングモデルを提供しています。

Asyncネームスペース

非同期GROM APIは、すべてのドメインクラスに対しasyncネームスペースを通じて提供されます。

```
import static grails.async.Promises.*
def p1 = Person.async.get(1L)
def p2 = Person.async.get(2L)
def p3 = Person.async.get(3L)
def results = waitAll(p1, p2, p3)
```

他にも非同期関連には多くの仕組みや注意点なども存在します。必ずドキュメントを一度読んでみましょう。

■開発支援・その他

●依存管理の向上 - 依存管理エンジンを Aether に変更

Grails でデフォルトで使用されている依存管理エンジンが、Maven で使用されているエンジンの Aether に変更されました。この変更は BuildConfig の設定を変更することで以前の依存管理エンジン ivy に戻すことも可能です。

```
grails.project.dependency.resolver = "maven" // or ivy
```

依存管理に Aether を使用する事で Grails では Maven ビルドツールと同じ振る舞いをします。このことで以前と比べてスナップショットハンドリングの向上やカスタムパッケージング等の機能向上がはかられています。

注意点として Aether では初期オフラインモードがありません。Aether はフラットファイルから依存解決しないので、Maven セントラルからライブラリを取得します。GRAILS_HOME/lib で配布されたライブラリは最初の依存解決では使用されません。そのため一度 Maven セントラルリポジトリから取得するまでオフラインは使用できません。

そしてさらに、dependency-report コマンドも改良され、依存問題の解決を手助けするために、依存関係のグラフがコンソールに表示されるようになりました。

```
runtime - Dependencies needed at runtime but not
for compilation (total: 107)
+--- org.codehaus.groovy:groovy-all:2.1.6
+--- org.grails:grails-plugin-rest:2.3.0
|    \--- org.slf4j:slf4j-api:1.7.5
|    \--- org.grails:grails-plugin-
datasource:2.3.0
|         \--- org.apache.tomcat.embed:tomcat-
embed-logging-log4j:7.0.42
|         \--- org.apache.tomcat:tomcat-
jdbc:7.0.42
|         \--- org.springframework:spring-
jdbc:3.2.4.RELEASE
|    \--- org.grails:grails-web:2.3.0
|         \--- commons-collections:commons-
collections:3.2.1
|         \--- taglibs:standard:1.1.2
|         \--- commons-fileupload:commons-
fileupload:1.2.2
|         \--- xpp3:xpp3_min:1.1.4c
|         \--- org.grails:grails-
databinding:2.3.0
|         \--- commons-el:commons-el:1.0
|         \--- javax.servlet:jstl:1.1.2
|         \--- org.springframework:spring-
webmvc:3.2.4.RELEASE
|         \--- opensymphony:sitemesh:2.4
|         \--- commons-beanutils:commons-
beanutils:1.8.3
|         \--- org.objenesis:objenesis:1.3
|         \--- org.grails:grails-spring:2.3.0
|    \--- com.google.code.gson:gson:2.2.4
```

●フォーク実行

全てのメジャーなコマンドが別々の JVM へフォークされるようになり、ビルドパスをランタイム/テストから分離します。フォーク実行のパラメータは BuildConfig に定義できます。

```
grails.project.fork = [
   test: [maxMemory: 768, minMemory: 64,
        debug: false, maxPerm: 256, daemon:true],
       // test-app用JVMへの設定
   run: [maxMemory: 768, minMemory: 64,
       debug: false, maxPerm: 256],
       // run-app用JVMへの設定
   war: [maxMemory: 768, minMemory: 64,
       debug: false, maxPerm: 256],
       // run-war用JVMへの設定
   console: [maxMemory: 768, minMemory: 64,
       debug: false, maxPerm: 256]
       // console用JVMへの設定
]
```

●テストランナーデーモン

インタラクティブモードでのテストを実行時にバックグラウンドでデーモンが起動してテストがフォーク実行されテスト効率がアップします。デーモンはインタラクティブモードで restart-daemon コマンド実行すると再起動できます。

```
$ grails> restart-daemon
```

テストランナーデーモンの制御は、BuildConfig の grails.project.fork で行います。daemon: を false にすることでデーモンでのテスト実行を行わないように設定できます。

```
grails.project.fork = [
    test: [maxMemory: 768, minMemory: 64,
        debug: false, maxPerm: 256, daemon:true],
    ..
```

●その他の向上など

Unit テストのデフォルトが Spock に

Unit テストのデフォルトが Spock になりました。したがって spock が組み込まれているため、今後 spock プラグインは不要になります。Grails コマンドで生成されるテストコードのテンプレートも Spock 仕様になっています。

デフォルトでの grailsw ラッパー生成

create-app コマンドがデフォルトで grailsw ラッパーを生成するようになりました。ラッパーを生成しない場合は --skip-wrapper オプションを指定してください。

```
grails create-app appname --skip-wrapper
```

Hibernate3と4をサポート

Grails 2.3からプラグインを切り替えることによってHibernate3と4の両方をサポートしています。

スカッフォルディング2.0プラグイン

これまで長いGrailsの歴史の中、重要な開発用ツールとして組み込まれていたスカッフォルディング機能が、Grails 2.3からプラグインでの提供となりました。これは別に後ろ向きな事ではなく、プラグインとして提供することでさらなる多様性が考えられると言うことです。このスカッフォルディングプラグイン2.0では、RESTコントローラ、Asyncコントローラと Spockユニットテストをサポートしています。

■おまけ

●upgradeコマンド!?

既存アプリケーションをアップグレードする時に、コマンド1発とか、実装を重ねたアプリケーションでは無理でしょう。とは思いますが。慎重に扱えば使っておくのも良いのでは？

一応一般的な手順です。

1. gitなどSCMでしっかり最新をコミットして起きましょう。
2. gvm u grails 2.3.0 を実行して手元のGrailsのバージョンを変更。（gvm使えない人は手動でGrailsバージョンを変更）
3. grails upgrade コマンドを決め込む！
4. 差分をみて問題無ければベースはクリア。

まだ続きます。

コマンドを実行すると、もろもろ更新されますが、tomcatとhibernateのプラグインバージョンは自動更新されません。両プラグインのバージョンを調整します。

```
plugins {
    //....
    build ":tomcat:7.0.42"
    runtime ":hibernate:3.6.10.1"
        // or ":hibernate4:4.1.11.1"
    //....
}
```

※ドキュメントによるとupgradeコマンドは無くなり、今後はuse-current-grails-version（長い!）を使用する事になるとかいてありますが、そんな形跡はドキュメント以外に何処にも無く結局upgradeコマンドを使用することになります。

まとめ

Grails 2.3がでて（執筆時2013/9）すぐですが、やはり最初のバージョンは、いつも難ありな感覚だと思います。記事がリリースされてる頃には、バグフィックス版の2.3.1が出ていることかと思います。そして、2014年の前半には、もう Grails 2.4がでて、後半にはGrails 3.0へと向かっていきます。かなりの速度で進化して行きますがGrailsの根本は変わってません。新機能が多いと少し安定度に不安がありますが、出たら少しずつ検証しながら、実際のプロジェクトで使っていくと良いでしょう。今後も最新情報を紹介できたらと思います。ではまた次回のバージョンアップで！

series 10

Groovy臨機応変（第二回）
～indyで遊ぼう！（但しGroovy流に）～

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。Groovy臨機応変の第二回目、今回のテーマは、「indyで遊ぼう!（但しGroovy流に）」です。

本記事のコードはGroovy 2.1.4で動作確認をしています。JDKは1.7以降が必要です。

indyって何？

Java 7では、Java VM上での動的言語の実行を効率化することを目的とした一連の機能拡張（JSR 292: Supporting Dynamically Typed Languages on the Java Platform）が導入されました。この機能拡張で追加された新規バイトコード命令「invokedynamic」にちなみ、この機能を愛称としてindyと呼びます。

indyについてご存知の方は多いと思いますが、Javaプログラマが意識直接する機会はあまり多くは無いでしょう。なぜならindyの導入目的は、動的言語の処理系実装の効率化なので、どんなJavaコードを書こうともindy命令が生成されることはないからです（しかしこの事情はJava 8ではちょっと様相が違ってきて、lambda式を使用するとindy命令が使われるのですが、その件についてはブログ記事[1]を書いてみました。宮川さんの素晴しい資料[2]もご参照ください）。

いずれにせよindyは、言語処理系のための機構であり、Javaプログラミングで意識する必要はほぼ無く、仮に遊びたいと思ってもなかなか敷居が高いのです。例えば、バイトコード生成や解析のためのライブラリであるASMライブラリ[3]を使用することでindy命令を含むクラスファイルを生成できますが、APIの呼び出しとしてバイトコード命令列を構築する必要があるのでちょっと手を出しずらい感があります。バイトコードのアセンブラとしてJasmineXTというのもありますが、invokedynamicには実際には対応していないようです。

あるいはそんなことをせずとも、indy対応の動的言語（GroovyやJRuby）を使用すれば、内部的にindy命令を含むバイトコードが生成されるのですが、残念ながら「遊んでいる感じ」が得られません。

Bytecode DSL

ということで、indyを扱うための手軽な方法として、Bytecode DSL[4]を紹介します。これはCedric Champeau氏が2011年に公開を開始した「Groovyの内部DSLで実現されたJavaバイトコードのアセンブラ」であり、実装としてはASMライブラリのラッパーです。以下は、Bytecode DSLのサイトにあるサンプルを元にした例です。

```
@GrabResolver(name="maven-repox", root="https://
raw.github.com/uehaj/maven-repo/gh-pages/
snapshot")
@Grab('groovyx.ast.bytecode:groovy-bytecode-
ast:0.2.0-separate-asm')
import groovyx.ast.bytecode.Bytecode

@groovyx.ast.bytecode.Bytecode
int fib(int i) {
    iload 1
    iconst_2
    if_icmpge l1
    iload 1
    _goto l2
   l1
    aload 0
    iload 1
    iconst_2
    isub
    invokevirtual '.fib', '(I)I'
    aload 0
    iload 1
    iconst_1
    isub
    invokevirtual '.fib' ,'(I)I'
    iadd
   l2
    ireturn
}

println fib(40)
```

驚くべきことに、これは正しいGroovyコードであり以下のよ

うに実行できます。

```
$ groovy fib.groovy
102334155
```

上記コード中の「@Bytecode」は、AST変換（groovyx.ast.bytecode.Bytecode）の指定です。AST変換はGroovyのコンパイル処理に介在して機能追加したりできるのものですが、ここではコード生成フェイズを置換しているのです[5]。上記のコードは、前述のようにgroovyコマンドで実行したり、groovycでコンパイルして.classファイルを生成することもできます。

Bytecode DSLは本記事執筆時点でMaven Centralリポジトリなどにモジュールが登録されていないようですので、Grabで取得して試せるように筆者が少しだけ修正[6]してコンパイルしたものをgithub上にアップロードしておきました。上記のコード冒頭の@GrabResolver指定はそのための指定です。

ちなみに前述のコードは、フィボナッチ数を計算する

```
public int fib(int i) {
   return i<2?i:fib(i-2)+fib(i-1);
}
```

のようなJavaコードに対応するバイトコード列です。本稿ではJavaバイトコードについては説明しませんが詳細はJava仮想機械仕様[7]などを参照ください。

Bytecode DSLはラベルやgoto,tablejumpなどの命令にも対応しており、さらに重要なことに、invokedynamic命令にも対応しているのです。これを使って遊んでみましょうというのが今回の記事の趣旨です。

indyの基本

ここでindy/invokedynamicの動作について簡単に説明しようと思いましたが、簡単に説明するのが難しいので結構長いです。詳細な説明は、Oracleのサイトにあるinvdy関連のAPIリファレンス[8]を見てください。

■従来のinvoke系命令によるメソッド呼び出し

まず、invokedynamic以前のメソッド呼び出し命令にはinvokevirtual, invokestatic, invokeinterfaceなどがありますが、これらの命令を使ったメソッド呼び出しの様子を（図1）に示します。クラス名やメソッド名、引数のシグネチャなどによって指定された呼び出し先メソッドの参照が、ロード時もしくは呼び出し時にJavaのルールと機構を使ってJava VMによって解決されて呼び出されます（（図1）の（1））。

（図1）

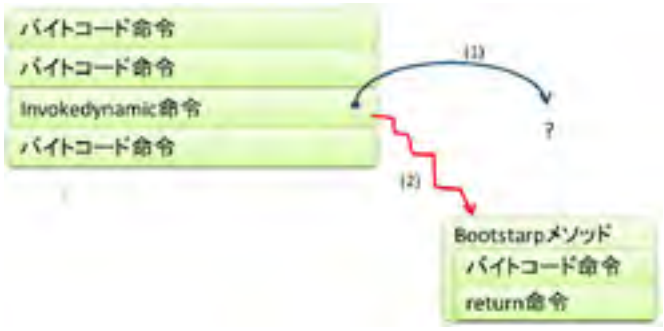
■invokedynamic命令によるメソッド呼び出し

これに対し、invokedynamicでは呼び出し先メソッドは、言語処理系が提供するランタイムによって、その言語処理系のルールに基づいて解決されます（（図2）の（1））。

メソッド参照を解決する方法をJava VMは知りません。

（図2）

その代りに、すべてのinvokedynamic命令には、「Bootstrapメソッド」と呼ばれるメソッドが結び付けられています（（図3）の（2））。Bootstrapメソッドは言語処理系のランタイムライブラリとして提供され、その言語処理系特有のメソッド選択のルールと機構を実装します。



(図3)

■Bootstrapメソッドによるメソッド解決

Bootstrapメソッドによるメソッド解決の様子を（図4）に示します。まず、invokedynamicは「java.lang.invoke.MethodHandle（メソッドハンドル、以降MHと略する）」を指すコンスタントプール上のオブジェクトを介してBootstrapメソッドと結び付いています（（図4）の（1））。MHはinvokedynamicで実現されるメソッド呼び出し処理そのものにも用いられており、むしろその用途がメインです。MHはメソッド呼び出しやフィールド参照などの操作を抽象化したもので、メソッド呼び出しについて言えば、リフレクションAPIのjava.lang.Methodのようなものです。

(図4)

（図4）における、メソッド解決までの処理の流れを以下に示します。

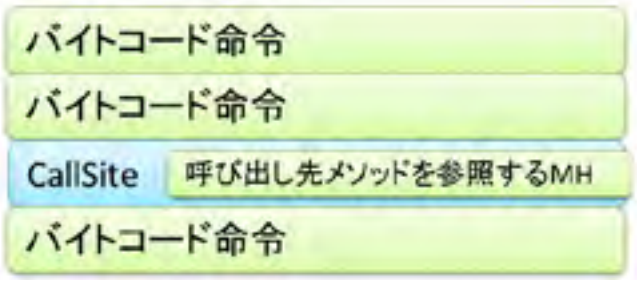
（1）Invokedynamic命令の **初回実行時** に、Bootstrapメソッドの呼び出しが **1回だけ** 実行されます。
（2）言語処理系が提供するBootstrapメソッドとしての「呼び出し先決定メソッド」が、型情報とメソッド名などを元に、適切な呼び出し先を決定します。
（3）Bootstrapメソッドは、決定された呼び出し先メソッドを参照するMHを含むCallSiteオブジェクトを返します。

■CallSiteの返却とインストール

Bootstrapが返したCallSiteオブジェクトが、invokedynamic命令に結び付けられます（（図5）の（1））。

(図5)

この結び付きは、1つのinvokedynamic命令について1回しか生じず、不可逆なので、以下のようにinvokedynamicがCallSiteに置き変わると考えても良いかも知れません\[9]\。ただし、CallSiteオブジェクトは複数のinvokedynamic命令から共有され得ることに注意してください。Bootstrapメソッドから返されるCallSiteオブジェクトが、シングルトン的に同じCallSiteのインスタンスである、という実装方針も許容されるためです。



(図6)

■メソッドの呼び出し

メソッド解決後の実際のメソッド呼び出しは以下のようになり、CallSite/MHを経由することを除けば、invokevirtualやinvokestaticと同様の状態です。CallSiteおよびMHを通じた呼び出しはJava VMが完全把握しており、処理系ランタイムのロジックは一切介在しないので、JITコンパイルの最適化がフルに効くことが期待されます。また、2回目以降の呼び出しは以下の状態から始まります。

(図7)

■CallSiteの変更

最後に重要なこととして、CallSite#setTarget()のメソッド呼び出しにより、MHを指し変えることができます。但し抽象クラスであるCallSiteを継承したクラスであるMutableCallSite,VolatileCallSiteクラスのインスタンスの場合のみです（（図8）の（1））。

（図8）

Groovyではmetaclassの変更で発生するメソッドの差し替えは、CallSiteのターゲットの置換によって実現することができます。

■理解のために、そして最適化の期待効果

invokedynaic命令は、実質的には「Bootstrapメソッドの呼び出し命令」と考えることができます。そして「Bootstrapメソッドの呼び出し命令」の呼び出しの結果得られるCallSiteオブジェクトでinvokedynamic命令が上書き（自己書き換え）されます。なおこの説明は、Java VM上の作りとしてそうなっているということではなく、概念の理解のため例えばそう理解することもできるでしょう、という例え話です。

invokedynamic命令がCallSiteで上書きされた後は、その命令のあった場所は「MHの参照先メソッドの呼び出し命令」としてJava VMに扱われます。Bootstrapメソッド呼び出しのオーバーヘッドは初回実行時のみにかかり、実際のターゲットメソッドの呼び出しについてはConstantCallSiteならオーバーヘッドは皆無となるでしょう。MutableCallSiteやVolatileCallSiteではMHの間接参照のオーバーヘッドが残ります。しかしながら、Hotspot JVMにおいては、invokevirtualの仮想メソッドテーブルの参照（間接参照）経由のメソッド呼び出しがインライン展開され得るのと同じように、MHの間接参照経由のメソッド呼び出しがインライン展開されることを期待しても良いでしょう。Hotspot JVMで行なわれる、外部クラスローディング時の最適化戻し（デオプティマイゼーション）と同等の処理が、CallSiteの呼び出しターゲットの変更によって行なわれると期待することにも無理はないでしょう。

使ってみよう indy&Bytecode DSL

では早速、Bytecode DSLでIndyを使ってみます。

■indy呼び出し（例1）と実行方法

まずは「Hello Indy」という文字列を表示するだけのコード例です。

（例1）

```
1  @GrabResolver(name="maven-repo", root="https://raw.github.com/uehaj/maven-repo/gh-pages/snapshot")
2  @Grab("groovyx.ast.bytecode:groovy-bytecode-ast:0.2.0-separate-asm")
3  import groovyx.ast.bytecode.Bytecode
4  import java.lang.invoke.CallSite
5  import java.lang.invoke.ConstantCallSite
6  import java.lang.invoke.MethodType
7  import java.lang.invoke.MethodHandle
8  import java.lang.invoke.MethodHandles.Lookup
9  import static org.objectweb.asm.Opcodes.H_INVOKESTATIC
10
11 class HelloIndy {
12
13     public static CallSite bootstrap( Lookup lookup, String methodName,
       MethodType type) {
14         assert methodName == 'xx'
15         MethodHandle mh = lookup.findVirtual(
           java.io.PrintStream, "println",
           MethodType.methodType(void, [String]))
16         return new ConstantCallSite(mh)
17     }
18
19     @Bytecode
20     static main(args) {
21         getstatic 'java/lang/System.out', 'Ljava/io/PrintStream;'
22         ldc 'Hello Indy'
23         invokedynamic 'xx', '(Ljava/io/PrintStream;Ljava/lang/String;)V',
               [H_INVOKESTATIC, 'HelloIndy', 'bootstrap', [CallSite, Lookup, String, MethodType]]
24         vreturn
25     }
26 }
```

これを単にgroovyコマンドで実行しようとすると以下のエラーが発生するかもしれません。

```
$ groovy HelloIndy.groovy
Caught: java.lang.ClassFormatError: Class file
version does not support constant tag 15 in class
file HelloIndy
java.lang.ClassFormatError: Class file version
does not support constant tag 15 in class file
HelloIndy
```

これは生成コードがindy命令を含むため、ターゲットとするクラスファイルのバージョンをJava7に設定しなければならないためであり、回避するには以下のようにします[10]。

```
$ groovy -Dgroovy.target.bytecode=1.7 HelloIndy.
groovy
Hello Indy
```

■indy呼び出し（例1）の説明

以降（例1）のコードを説明していきます。まず、

```
13  public static CallSite bootstrap(Lookup
    lookup, String methodName, MethodType type) {
```

ここはBootstrapメソッドの定義です。Bootstrapメソッド引数は以下のうち最初の1～3が必須であり、さらに追加的な引数を取ることもできます。

1. **Lookup lookup**: MHを検索・取得するためのルックアップオブジェクト「java.lang.invoke.MethodHandles.Lookup」が与えられる。
2. **String methodName**: invokedynamic命令で「メソッド名」として指定された文字列。これをメソッド解決にどのように使用するかは処理系に任されている。従って実体としてその「メソッド名」を持つJavaレベルのメソッドが存在する必要は全くない。例えばGrailsのdynamic finderのようにこの引数をもとに処理内容を変化させも良いわけである。
3. **MethodType type**: 呼び出そうとするメソッド名の返り値と引数の型。
4. （オプション）追加的引数として、プリミティブ型の値（及びコンスタントプールの特定の種別のエントリ）を受けとることができる。

```
15      MethodHandle mh = lookup.findVirtual(
        java.io.PrintStream, "println",
        MethodType.methodType(void, [String]))
16      return new ConstantCallSite(mh)
```

15行目では、ルックアップオブジェクトを使ってPrintStream#println()を参照するMHを取得しています。

16行目では、そのMHをターゲットとする、ターゲット先が変更できないCallSiteであるConstantCallSiteを作成し、返却しています。

このCallSiteがインストールされることで、このBootstrapメソッドに紐付いたinvokedynamic命令（23行目）は「println を呼び出す」処理に置き換えられることになります。

```
19    @Bytecode
20    static main(args) {
21      getstatic 'java/lang/System.out',
    'Ljava/io/PrintStream;'
22      ldc 'Hello Indy'
23      invokedynamic 'xx', '(Ljava/io/
    PrintStream;Ljava/lang/String;)V',
    [H_INVOKESTATIC, 'HelloIndy',
        'bootstrap', [CallSite, Lookup, String,
    MethodType]]
24      vreturn
25    }
26  }
```

19行目がBytecodeアノテーションの指定です。ここではmainメソッドをbytecodeで記述しています。

21、22行目は引数の準備です。invokevirtualと同型なのですが、21行目ではthisにあたるオブジェクト（System.out）をスタックにプッシュ、22行目では引数として文字列定数"Hello Indy"をスタックに積んでいます。

23行目でいよいよinvokedynamicの登場です。基本的にやってることはBootstrapメソッドの指定とその呼び出しについての情報の指定であり、一連の引数の意味は以下のとおりです。

- メソッド名（ここでは'xx'）。Bootstrapメソッドに「メソッド名」として渡る文字列。実際にそのメソッドが存在する必要はない。
- invokedynamicで呼び出そうとするメソッドの引数と返り値の型（ここでは'（Ljava/io/PrintStream;Ljava/lang/String;）V'）。こちらは重要。
- Bootstrapメソッドを特定する情報。JVM仕様の4.7.21. The BootstrapMethods attributeのbootstrap_methods[]で使用される、4.4.8. The CONSTANT_MethodHandle_info StructureのCONSTANT_MethodHandle_infoの内容に相当する情報。Bytecode DSLでは以下のリストとして表現される。
  - 第1要素（ここではH_INVOKESTATIC）：Bootstrapメソッドを参照するMHの種別だが、H_INVOKESTATICがほぼ一択。
  - 第2要素（ここでは'HelloIndy'）：Bootstrapメソッドが所属するクラスのクラス名
  - 第3要素（ここでは'bootstrap'）：Bootstrapメソッド名
  - 第4要素（ここでは[CallSite, Lookup, String, MethodType]）：Bootstrapメソッドの引数の型を表すリスト。追加的な引数があればその型を必要な個数分を追加指定する。
- （オプション）Bootstrapメソッドに渡す追加的な引数。

■indy呼び出し（例2）

（例1）ではMHを1つだけ参照するConstantCallSiteを返すBootstrapメソッドを使用しており、indyを使う意味が皆無に等しいものでした。次はMutableCallSiteを使ってみた例です。文字列を出力するメソッドをinvokedynamicを使って呼び出すのですが、MutableCallSiteを使って、「標準出力に表示するメソッド」と「ファイルに出力するメソッド」を切り替えています。

（例2）

```
12 class HelloIndy2 {
13
14     static void logToSysOut(String msg) {
15         println(msg)
16     }
17
18     static void logToFile(String msg) {
19         new File("output.log") << msg << "\n"
20     }
21
22     static CallSite cs
23
24     static MethodHandle methodSelect(Lookup lookup, String name) {
25         return lookup.findStatic(HelloIndy2, name, MethodType.methodType(void, [String]))
26     }
27
28     public static CallSite bootstrap(Lookup lookup, String methodName, MethodType type) {
29         MethodHandle mh = methodSelect(lookup, methodName)
30         cs = new MutableCallSite(mh)
31         return cs
32     }
33
34     @Bytecode
35     static void test(String s) {
36         aload 0
37         invokedynamic 'logToSysOut', '(Ljava/lang/String;)V',
               [H_INVOKESTATIC, 'HelloIndy2', 'bootstrap', [CallSite, Lookup, String, MethodType]]
38         vreturn
39     }
40
41     static main(args) {
42         test("output to sysout") // 標準出力に出力される
43         MethodHandle mh = methodSelect(MethodHandles.lookup(), "logToFile")
44         cs.setTarget(mh) // CallSiteのターゲットをファイルへの出力を行うメソッドに置き換える
45         test("output to file")  // ファイルに出力される
46     }
47 }
```

以降、解説します。まず、メソッド選択を行うメソッドを以下のように準備します。ここでは指定したメソッド名をもつHelloIndy2クラスの静的メソッドから選択します。言語処理系を実装する際には、ターゲット言語のランタイムライブラリから、メソッド呼び出しやフィールド値の参照をするMHを作ったり探してくる処理になるはずです。

```
24    static MethodHandle methodSelect(Lookup
    lookup, String name) {
25      return lookup.findStatic(HelloIndy2, name,
    MethodType.methodType(void, [String]))
26    }
```

さらに、例1では、ConstantCallSiteを返していたところで

```
30      cs = new MutableCallSite(mh)
```

のようにMutableCallSite()を作成して変数csに保存するようにします。このcsを使って

```
43      MethodHandle mh =
    methodSelect(MethodHandles.lookup(),
    "logToFile")
44      cs.setTarget(mh)
```

で別のメソッドハンドルでターゲットメソッドの差し替えを行ないます。

■メソッドハンドルを加工する

indyの真骨頂は、私見ですが、メソッドハンドルの操作API群にあります。indyの用途、すなわち言語処理系の実装の文脈では、invokedynaic命令は「他言語」のメソッドの呼び出しであり、異なる言語セマンティクスの橋渡し処理をしなければならない場合があります。たとえば、

- 言語Xでのリストデータ構造は、Javaのjava.util.Listとは異なるため、メソッド呼び出し時にリストの引数や返り値を相互変換する
- 言語Xのデータは、Javaから見るとラッパーオブジェクトを介して扱う必要があり、メソッド呼び出し時に引数・返り値をラッピング・アンラッピング変換する
- 言語Xの実行コンテキスト（環境）を持ち回る必要があり、そのコンテキストを常にメソッドの第一引数に渡す必要がある

などのような場合です。indyにおいては、MHに対する高階操作、すなわち既存のMHを加工し「上記のような処理を追加的に実行するようなMH」を作り出すようなAPIが多数とりそろえられています[11]。具体的には、以下のMethodHandleおよびMethodHandlesの静的メソッド群です（主なもののみ抜粋）。

クラス	メソッド	説明
java.lang.invoke.MethodHandle	asCollector	複数引数を配列で指定するようなMHを得る
java.lang.invoke.MethodHandle	asSpreader	配列引数を複数引数で指定するようなMHを得る
java.lang.invoke.MethodHandle	bindTo	最初の引数を固定したMHを得る
java.lang.invoke.MethodHandle	asType	結果や引数をキャストや格上げなどをして適合させて呼び出すようなMHを得る
java.lang.invoke.MethodHandles	dropArguments	いくつかの引数を捨てるMHを返す
java.lang.invoke.MethodHandles	filterArguments	引数それぞれに指定したフィルターMHを適用した結果を引数としてtargetとなるMHを呼びだすようなMHを返す
java.lang.invoke.MethodHandles	guardWithTest	テスト用MHと、テスト用MHを実行した結果が真の場合に実行するMHと、偽の場合に実行するMHを渡し、その条件判断を実行するMHを返す
java.lang.invoke.MethodHandles	permuteArgument	引数の順序を任意に変更したMHを返す

これらの機能でMHを複合的に組み合わせたものがinvokevirtualにCallSiteとしてインストールされたときでも、単純な単一のMHの場合と同様に全体が依然としてJava VMの完全な制御下にあり、高度な最適化（主にインライン展開）がなされることを期待できるのです（たぶん）。従来のJava VM上の動的言語でも、CallSite最適化は良くなされてきたわけですが、それらでは達成できなかったような性能向上をindyが生じさせる可能性はたぶんここにあります。言語セマンティクスの吸収層を、Java VMが良く把握可能な形で実装するということです。だとすると、Javaとの言語間のセマンティクスの差が少ない言語（Groovyとか）では性能向上の余地が少ない可能性もありますが、これは私の憶測です。

■メソッドハンドルの加工の例（bindTo）

（例1）を少しだけ変更して、PrintStream#printlnの第一引数（printlnはインスタンスメソッドなのでthisオブジェクトに対応する）に「System.out」を固定したMHをターゲットとするCallSiteをBootstrapメソッドから返すようにしてやります。

```
1   class HelloIndy3 {
2
3       public static CallSite bootstrap(Lookup lookup, String methodName, MethodType type) {
4           MethodHandle mh = lookup.findVirtual(java.io.PrintStream, "println",
                MethodType.methodType(void, [String])).bindTo(System.out)
5           return new ConstantCallSite(mh)
6       }
7
8       @Bytecode
9       static main(args) {
10          ldc           'Hello Indy'
11          invokedynamic 'xx', '(Ljava/lang/String;)V', [H_INVOKESTATIC, 'HelloIndy3',
                'bootstrap', [CallSite, Lookup, String, MethodType]]
12          vreturn
13      }
14  }
```

4行目でSystem.outをbindToしてやることで、10～11行目における呼び出しでSystem.outを引数として渡す必要がなくなりました。

■メソッドハンドルの加工の例（filterReturnValue）

2つのMHを連続して実行するようなMHを作成してみます（これが必要になる例を後で示します）。

MethodHandles#filterReturnValueは引数に与えた二つのMH（target, filter）を使い、targetを呼び出した上でその結果の値をfilterの引数に渡して呼び出し、その結果を返すようなMHを返すメソッドです。その特別な場合として、targetの戻り値の型がvoid、filterの引数が無しであれば、単にtarget, filterの順にMHを呼び出してくれます。これを利用すると2つのMHを順に呼びだすMHが作れます。

```
class HelloIndy4 {

    public static CallSite bootstrap(Lookup lookup, String methodName, MethodType type) {
        MethodHandle mh1 = lookup.findVirtual(java.io.PrintStream, "println",
        MethodType.methodType(void, [String])).bindTo(System.out).bindTo("Hello")
        MethodHandle mh2 = lookup.findVirtual(java.io.PrintStream, "println",
        MethodType.methodType(void, [String])).bindTo(System.out).bindTo("Indy")
        MethodHandle mh3 = MethodHandles.filterReturnValue(mh1, mh2)
        return new ConstantCallSite(mh3)
    }

    @Bytecode
    static main(args) {
        invokedynamic 'xx', '()V', [H_INVOKESTATIC, 'HelloIndy4', 'bootstrap',
            [CallSite, Lookup, String,MethodType]]
        vreturn
    }
}
```

ここではbindTo()を二回呼び出して、printlnで出力する文字列も固定して引数なしのMHを作り出しています。

以下は実行例です。

```
% groovy  -Dgroovy.target.bytecode=1.7
HelloIndy4.groovy
Hello
Indy
```

indy で Brainfuck を実装してみよう

最後に indy を使用する言語処理系を実装してみます。実装するのは Brainfuck という言語[12]です。Brainfuck はいわゆる一つの奇妙なプログラミング言語ですが、処理系の実装が容易であることを重視して設計されたチューリング完全な言語であり、試しに実装してみるのには良いでしょう。

とはいえ、残念ながら Brainfuck は動的言語ではないので、あんまり indy が役にたちません。

そこで、一捻りとして、"++--" といった命令列を文字列として保持し、Bootstrap メソッドの実行時に「メソッド呼び出しの列を表現する MH」として生成してみます。"+" が p()、"-" が m() というメソッド呼び出しに対応するとすると、"++--" というオプション引数を受けとったならば、「p(), p(), m(), m(),」という一連の呼び出しに対応する MH を生成し、CallSite にして返すような Bootstrap メソッドを実装します。これによってクラスファイルのサイズが減少することを期待します。

以下が Brainfuck コンパイラのコード（compile.groovy）です。

```
// ネストしたループにおけるラベル名を管理するためのデータとコード。
def list = [].withDefault{0}
def nestLevel = 0
def level = { it << list[0..nestLevel-1].join('_') }.asWritable()
def increaseLevel = { list[nestLevel++]++; level(it) }.asWritable()
def decreaseLevel = { level(it); nestLevel-- }.asWritable()

// Brainfuck命令をJVMバイトコードにコンバートする
def genCode = {
    new File(args[0]).text.replaceAll(/[\^\\\\+\\\\-\\\\<\\\\>\\\\.\\\\,\\\\[\\\\]]/, '').eachMatch(/
    [\^\\\\[\\\\]]+|[\\\\[\\\\]]/) { g0 ->
      if (g0 == '[') {
         it << """        _GOTO          tmp$increaseLevel
    lab$level:
"""
      }
      else if (g0 == ']') {
        it << """    tmp$level:
        getstatic     '.data','[B'
        getstatic     '.dp','I'
        baload
        ifne          lab$decreaseLevel
"""
      }
      else {
         it << """        invokedynamic 'dummy', '()V', [H_INVOKESTATIC, 'Brainfuck', 'bootstrap',
    [CallSite, Lookup, String, MethodType, String]], '$g0'
"""
      }
    }
}.asWritable()

println """\
@GrabResolver(name="maven-repo", root="https://raw.github.com/uehaj/maven-repo/gh-pages/snapshot")
@Grab("groovyx.ast.bytecode:groovy-bytecode-ast:0.2.0-separate-asm")
import groovyx.ast.bytecode.Bytecode
import java.lang.invoke.*
import java.lang.invoke.MethodHandles.Lookup
import static groovyjarjarasm.asm.Opcodes.H_INVOKESTATIC

class Brainfuck {

    // Bootstrapメソッド
    public static CallSite bootstrap(Lookup lookup, String methodName, MethodType type, String
    instructions) {
        MethodHandle result = null
        instructions.tr('<>.','lgo').each { insn ->
            MethodHandle mh = lookup.findStatic(Brainfuck, insn, MethodType.methodType(void.class))
            result = (result == null) ? mh : MethodHandles.filterReturnValue(result, mh)
        }
```

```
        new ConstantCallSite(result)
    }

    // Brainfuck実行のためのデータ構造
    static int dp // データポインタ
    static byte[] data = new byte[30000] // メモリ

    // Brainfuck命令群
    static void '+'(){data[dp]++}
    static void '-'(){data[dp]--}
    static void l(){dp--} // メソッド名が'<'だとクラスファイルとして違法なメソッド名になるので妥協
    static void g(){dp++} // '>'
    static void o(){print((char)data[dp])} // '.'
    static void ','(){System.in.read(data, dp, 1)}

    @Bytecode
    static void main(String[] args) throws Exception {
        // Brainfuckからコンバートされたコード
$genCode
        return
    }
}"""
```

このコードは、Brainfuck言語のソースファイルを引数として起動すると、Bytecode DSLを用いたアセンブラコードを含むGroovyコードを標準出力に出力します。

以下は実行例です。

```
% cat hello.bf
>+++++++++[<++++++++>-]<.>+++++++[<++++>-]<+.+++++++..+++.[-]>++++++++[<++++>-]<.>++++++++++[<+++++>-
    ]<.>++++++++[<+++>-]<.+++.------.--------.[-]>++++++++[<++++>-]<+.[-]++++++++++.

% groovy compile.groovy hello.bf > a.groovy
% groovy -Dgroovy.target.bytecode=1.7 a.groovy
Hello World!
```

上でa.groovyにはこんなコードが生成されています（抜粋）。

```
@Bytecode
static void main(String[] args) throws Exception {
    // Brainfuckからコンバートされたコード
    invokedynamic 'dummy', '()V', [H_INVOKESTATIC, 'Brainfuck', 'bootstrap',
        [CallSite, Lookup, String, MethodType, String]], '>++++++++++'
    _GOTO          tmp1
lab1:
    invokedynamic 'dummy', '()V', [H_INVOKESTATIC, 'Brainfuck', 'bootstrap',
        [CallSite, Lookup, String, MethodType, String]], '<+++++++++>-'
tmp1:
    getstatic     '.data','[B'
    getstatic     '.dp','I'
    baload
    ifne          lab1
    invokedynamic 'dummy', '()V', [H_INVOKESTATIC, 'Brainfuck', 'bootstrap',
        [CallSite, Lookup, String, MethodType, String]], '<.>++++++++'
    _GOTO          tmp2
lab2:
    invokedynamic 'dummy', '()V', [H_INVOKESTATIC, 'Brainfuck', 'bootstrap',
        [CallSite, Lookup, String, MethodType, String]], '<++++>-'
:
```

まとめ

以上、かけ足でしたがいかがでしょうか。振り返るに、indyはJava VMのJITコンパイラのオープン化の一種です。JITコンパイラの一部であった、メソッド選択機構の内部データ構造をSPI化し、フック等を通じて言語処理系から利用可能になるように公開したものです。

Java VM上で言語処理系を実装するとき以外にはあまり役に立たないかもしれません。それ以外には、今いち用途が思いつきませんね…。

この記事が誰かの楽しみになりますことを希望しています。感想をお待ちしております（Twitter @uehaj）。

脚注・参考リンク

[1] http://uehaj.hatenablog.com/entry/indyandlambda

[2] http://www.slideshare.net/miyakawataku/lambda-meets-invokedynamic

[3] http://asm.ow2.org/

[4] https://github.com/melix/groovy-bytecode-ast/wiki
http://www.jroller.com/melix/entry/yes_fibonacci_in_groovy_can
http://www.jroller.com/melix/entry/groovy_bytecode_ast_transformation_released

[5] Groovy++でも、同じようにコード生成を置き変えることによって、静的Groovyを実現していました。

[6] https://github.com/uehaj/groovy-bytecode-ast

[7] http://docs.oracle.com/javase/specs/jvms/se7/html/

[8] http://docs.oracle.com/javase/jp/7/api/java/lang/invoke_package-summary.html

[9] Java VM仕様上の用語としては、バイトコード列中のinvokedynamic命令の出現のことをdynamic call siteと呼びます。クラスとしてのCallSiteオブジェクトは、その意味でのcall siteとはちょっと違う気がします。

[10] groovycではこのプロパティの設定ではターゲットクラスファイルバージョンが変更できないため「configuration.setTargetBytecode("1.7")」という内容のファイルを作成し、--configscriptオプションで指定する。

[11] このAPI群を見ると、MHこそがLambdaだ、と思うのは私だけでしょうか？ MHだけでプログラミングができないか？とも。でも「条件MHが成立する間、ループ本体MHを実行するようなMHを返す」などの重要なAPIがありません。（今のところは）そういうものじゃないのです。残念。

[12] http://ja.wikipedia.org/wiki/Brainfuck

組織内ルールや共通設定が自動適用される独自Gradleを作ろう

series 15

林 政利（はやし まさとし）
フリーランスのエンジニアとして日々Web系システムを開発中。
G*系の活動ではGradleのドキュメントを翻訳したりGrailsのプラグインを公開したりしています。

こんにちは、LITERAL-ICEの林です。

G*の誇るビルドツールGradleも最近は(Groovyというか主にAndroid方面から)よく名前を聞くようになってきており、興味を持っておられる方も増えているのではと感じています。

さて、今回は「組織で使うGradle」という観点から、既にGradleを触ったことがある方向けに、社内など組織内のビルドルールをある程度自動で共有する方法を紹介したいと思います。

プロキシの設定はgradle.propertiesに書いてどうのこうの、社内リポジトリのURLはここに書いてるからbuild.gradleのrepositoriesセクションに追記して、あそうだライブラリのバージョン1.8はライセンスに問題があるから使わないでねなどと何処かに書く前に、その辺を自動化できないか、そういうお話しですね。

具体的には、本記事では以下の各設定を組織内の全ビルドスクリプトで共有するようにしてみます。

- ビルドスクリプトの作成者も使用者も意識することなく社内リポジトリを設定する。
- 同じように規定のプロキシ設定も自動適用する。
- あるライブラリやあるバージョンの使用を禁止する。
- あるライブラリを使用するときは、組織で決めた特定のバージョンを使用する。

このような各設定やルールの共有は、それらを入れ込んだ専用のGradleを作り、それを使ってビルドさせることで簡単に実現できます。

組織用の独自Gradleを作る

ここでキーとなるのは「Gradleラッパー」と「初期化スクリプト」というGradleの機能です。

これらについては後述しますが、まずは百聞は一見に如かずということで、リポジトリ設定済みの独自Gradleを試してみましょう。

■社内リポジトリ設定済みのGradleを使用する

以下にサンプルのGradleプロジェクトを用意しました。

- https://github.com/literalice/gradle-sample-preconfigure-repository

このビルドスクリプトは以下のようになっています。

```
// ./build.gradle

configurations {
    runtime
}

dependencies {
    runtime name: "hoge-hoge", ext: "txt"
}

task showDependencies << {
    configurations.runtime.each { File file ->
println file.absolutePath }
}
```

このスクリプトでは「"hoge-hoge.txt"」という依存関係を使用していますが、この依存関係の取得先であるリポジトリが設定されていません(repositoriesセクションがない)。

なので、公式のGradleを使って(システムのgradleコマンドなどで)ビルドするとビルドに失敗します。

しかし、同梱のラッパー「gradlew」を使ってshowDependenciesタスクを実行すると以下のように問題なく結果が表示されるはずです。

```
> ./gradlew showDependencies

:showDependencies
xxxx\preconfigure-repository\lib\hoge-hoge.txt

BUILD SUCCESSFUL
```

つまり、ビルドスクリプトに何も書いてなくても、既にリポジトリが設定済みになっているわけです。

(ここでは、プロジェクトディレクトリの「lib」ディレクトリをリポジトリに設定しています)

これを実現しているのが、Gradleの「Gradleラッパー」および「初期化スクリプト」という機能です。

■初期化スクリプト

初期化スクリプトは、ユーザーがビルドを開始した際、その実行前にあらかじめ実行されるスクリプトです。

初期化スクリプトは以下の方法でビルドに渡すことができ、ビルド実行前に呼び出されて実行されます。

- gradleコマンド実行時に、オプションの-Iか--init-scriptオプションに初期化スクリプトのパスを渡す。
- init.gradleという名前のファイルをホームディレクトリの.gradleディレクトリに置く。
- .gradleという拡張子のファイルをホームディレクトリの.gradle/init.dディレクトリに置く。
- .gradleという拡張子のファイルをGradle本体のinit.dディレクトリに置く。

上から順番に実行されます。渡し方はたくさんありますが、ここで注目するのは一番下(=もっとも優先されるスクリプト)、つまりGradle本体の「init.d」ディレクトリに初期化スクリプトを入れ込めるという点です。

■Gradleラッパー

Gradleラッパーは Gradleの中でもかなり有名な機能だと思います。この機能を使うと、Gradleをインストールしていないマシンでも Gradleビルドを実行することができます。

これは、プロジェクトに同梱されたラッパースクリプト(gradlew.batまたはシェルスクリプト gradlew)が Gradle本体をダウンロードし、そのGradleを使ってビルドを実行してくれるからです。

Gradleラッパーは Gradleが生成したものをプロジェクトに同梱するだけで非常に便利に機能しますが、今回ポイントとなるのは、「ラッパースクリプトがダウンロードするGradleのダウンロード先URLは設定で変更することができる」という点です。

先ほど実行したGradleビルドで確認してみます。

実行したファイル、「gradlew(.bat)」がラッパースクリプトで、ラッパーの設定ファイルはプロジェクトディレクトリの「gradle/wrapper/gradle-wrapper.properties」ファイルです。

```
gradleVersion=1.7-preconfigure-repository
archiveBase=PROJECT
archivePath=wrapper/dists
distributionBase=PROJECT
distributionPath=wrapper/dists
zipStoreBase=PROJECT
zipStorePath=wrapper/dists
distributionUrl=https://dl.dropboxusercontent.
com/u/11287929/material/g-aster-7/gradle-1.7-
preconfigure-repository-bin.zip
```

一番下の設定値「distributionUrl」を見てください。

ここで、このビルドを実行するときに使用されるGradleのダウンロード先URLを指定しています(その他の設定はダウンロードしたGradleの保存先や解凍先です)。

設定値を見ると、「gradle-1.7-preconfigure-repository-bin.zip」というGradleが指定されていることが分かると思います。

これが、リポジトリ設定を同梱した独自Gradleです。

独自Gradleとは言っても、Gradle1.7の「init.d」ディレクトリに「init.gradle」という初期化スクリプトを配置してアーカイブしただけのものです。

「wrapper/dists」にビルドに使用されたこのGradleが保存されているので、確認してみてください。

```
// init.d/init.gradle

allprojects {
    repositories {
        flatDir {
            dirs "lib"
        }
    }
}
```

スクリプトの中で、リポジトリが一つ定義されています。

このスクリプトが、ビルド実行前に実行されたために、必要なリポジトリが設定されていたのです。

上記のスクリプトではフラットファイルリポジトリ設定でプロジェクトディレクトリの「lib」ディレクトリをリポジトリとして設定していますが、mavenCentral()や社内のリポジトリをここで設定することでその他のリポジトリを設定することもできます。

プロキシ設定済みの状態でGradleを使用させる

ここからは、初期化スクリプトとGradleラッパーの仕組みを使用してできることをいくつか紹介していきます。

まず、社内プロキシなど、プロキシサーバーの設定です。ライブラリをMavenリポジトリからダウンロードする際など、インターネット接続にプロキシを通す必要がある環境は多いと思います。

本来、gradleが使用するプロキシ設定は、プロジェクトディレクトリに「gradle.properties」を配置して、以下のように設定値を書き込めば適用されます。

```
systemProp.http.proxyHost=localhost
systemProp.http.proxyPort=8080
systemProp.http.proxyUser=userid
systemProp.http.proxyPassword=password
systemProp.http.nonProxyHosts=127.0.0.1|localhost
```

一方、いちいち組織内の全てのgradleプロジェクトに設定するのが面倒くさい、または設定が変更されたときに簡単に追随したいという場合は、以下のように独自Gradleの初期化スクリプトで設定しておくことができます。

```
// init.d/init.gradle

System.setProperty("http.proxyHost", "localhost")
System.setProperty("http.proxyPort", "8080")
System.setProperty("http.nonProxyHosts",
"127.0.0.1|localhost")

allprojects {
    repositories {
        mavenCentral()
    }
}
```

上記のスクリプトは、拡張子を.gradleにして独自Gradleの「init.d」ディレクトリに同梱しておきます。

以下にサンプルプロジェクトを上げていますので、ローカルプロキシなどを立てて試してみてください。

- https://github.com/literalice/gradle-sample-preconfigure-proxy

サンプルプロジェクトで使用された独自Gradleは、リポジトリ設定の例と同様、プロジェクトディレクトリの「wrapper/dists」に格納されています。中の初期化スクリプトを確認してみてください。

ちなみに、プロキシサーバーがない場合以下のようなエラーになるはずです。

```
> gradlew show --refresh-dependencies

:showDependencies FAILED

FAILURE: Build failed with an exception.

* Where:
Build file '~preconfigure-proxybuild.gradle' line:
10

* What went wrong:
Execution failed for task ':showDependencies'.
> Could not resolve all dependencies for
configuration ':runtime'.
   > Could not resolve org.apache.
commons:commons-lang3:3.1.
     Required by:
         :preconfigure-proxy:unspecified
      > Could not HEAD 'http://repo1.maven.org/
maven2/org/apache/commons/commons-lang3/3.1/
commons-lang3-3.1.pom'.
         > Connection to http://localhost:12000
refused

* Try:
Run with --stacktrace option to get the stack
trace. Run with --info or --debug option to get
more log output.
```

あるライブラリやあるバージョンの使用を禁止する

例えば、社内で開発しているライブラリが社内リポジトリに上がっているけれども、そのうちいくつかのバージョンについては使用させたくないとか、オープンソースライブラリのうちいくつかのバージョンからライセンスが変わったために社内規定で使用が禁止されたなどという場合に対応するための設定を紹介します。

ここでは、比較的身近な例と思われるiTextライブラリを例にします。

「AGPLに変更されたバージョン5系のiTextをライセンス上の理由により使用させたくない」というパターンを想定してください。

ビルドスクリプトでこのライブラリが使用されている場合、例外を投げてビルドを失敗させます。

サンプルプロジェクトを以下にアップロードしています。

- https://github.com/literalice/gradle-sample-forbidden-versions

このプロジェクトでは、以下のように、iTextのバージョン5を使用しています。

```
// ./build.gradle

configurations {
    runtime
}

dependencies {
    // runtime "com.lowagie:itext:2.1.7"
    runtime "com.itextpdf:itextpdf:5.4.4"
}

task showDependencies << {
    configurations.runtime.each { File file ->
        println file.absolutePath }
}
```

クローンして実行すると、以下のようにエラーが出力されるはずです。

```
> gradlew show

Execution failed for task ':showDependencies'.
> Could not resolve all dependencies for
configuration ':runtime'.
   > Could not resolve com.
itextpdf:itextpdf:5.4.4.
     Required by:
         :forbidden-versions:unspecified
      > Only version 2.+ or 4.+ are allowed for
iText.
```

一方、依存関係のバージョンを以下のように2系に変更すると、

エラーが発生することなくビルドを完了できます。

```
// ./build.gradle

dependencies {
    runtime "com.lowagie:itext:2.1.7"
    // runtime "com.itextpdf:itextpdf:5.4.4"
}
```

これらのルールも、独自Gradleの初期化スクリプトで設定されています。今までの例と同じように、プロジェクトディレクトリの「wrapper/dists」にビルドで使用しているディストリビューションが格納されているので、中の初期化スクリプトを確認してみてください。

```
// init.d/init.gradle

allprojects {
  repositories {
    mavenCentral()
  }
  configurations.all {
    // 解決した全ての依存関係を走査する
    resolutionStrategy.eachDependency {
      DependencyResolveDetails details ->
        final requested = details.requested

      logger.lifecycle "library: {}", requested

      if (requested.name == "itext" ||
      requested.name == "itextpdf") {
        if(!(requested.version==~/^[2|4]\..+$/)){
          throw new IllegalDependencyNotation(
"Only version 2.+ or 4.+ are allowed for iText.")
        }
      }
    }
  }
}
```

ここでは、「依存関係解決ルール」というGradleの機能を使って全ての依存関係を走査し、iTextの2系と4系以外が使用されている場合に例外を投げています。

ルールは推移的な依存関係にも適用されるので、この独自Gradleを普通に使う限りiTextのバージョン5が紛れ込むことはないでしょう。

ユーザーガイドには、この他にも依存関係解決ルールを使って実現できるパターンがいくつか紹介されています。

http://gradle.monochromeroad.com/docs/userguide/dependency_management.html#sec:dependency_metadata_manipulation

- 禁止バージョンを検知したとき、上記のように失敗させるのではなく、別のバージョンに置換する
- バージョン番号をプレースホルダで指定させ("com.lowagie:itext:*org-default*"など)、ルール内で社内データベースを参照して推奨バージョンに置換する

なお、依存関係解決ルールはGradle1.9現在「試験的(Incubating)」としてマークされており、将来のバージョンでAPI等が変更される可能性があります。

変更される場合はリリースノートで通知されることになっていますので、もし現在使用している場合はリリースノートをよく確認されることをお勧めします。

まとめ

本記事では、Gradleラッパーを使って、単にGradleをインストールしていないマシンでビルドを実行させるだけでなく、初期化スクリプトや共通ルールを同梱した独自Gradleをダウンロードさせることで様々な処理を共有できることを紹介しました。

さらに加えて依存関係解決ルールなどを使いこなせば、組織内で暗黙のうちに了解されていたりWikiなどに記載していたルールを上手くビルドスクリプトに反映させることができそうですね。

Gradleは開発、リリース速度が速く、先日も1.9がリリースされたばかりですが、一方で常に「エンタープライズ向け」という視点を保ちながら進化しています。Gradleの機能ライフサイクル規定などからは、速度だけでなく、安定性にも気を遣って開発を進めたいという姿勢が感じられるかと思います。

もちろん、新しい機能を使っていこうとしたときは不安定なバグに遭遇することもありますが、組織内で検証など進めつつ、自動化できるところはどんどんGradleで自動化して、イージーなライフをお楽しみいただければと思います。それでは。

Grails Plugin 探訪

第 8 回

~ Grails Markdown プラグイン ~

URL: http://grails.org/plugin/markdown
プラグインのバージョン: 1.1.1
対応する Grails のバージョン: 1.3.5 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、Grails Markdown 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 10.8.5
- Java : 1.7.0_45
- Grails : 2.3.2

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。

Grails Markdown プラグインとは

Grails Markdown プラグインは、Markdown 用のタグライブラリやサービスクラスを提供するためのプラグインです。タグライブラリ、サービスクラスとも、Markdown を HTML に変換できます。

また、String クラスにメソッドを追加し、Markdown の文字列と HTML の文字列を相互に変換します。

なお、Markdown 処理のためのライブラリとして、Pegdown と Remark というライブラリを使用しています。

Markdown とは

Markdown とは、文書を記述するためのマークアップ言語の一つです。詳しくはウィキペディアを参照してください。

また、Markdown の文法については、このページを参照してください。

プラグインのインストール

Grails Markdown プラグインのインストールは、BuildConfig.groovy に次のように記述することで行います。

```
    plugins {
        ....
        compile ":markdown:1.1.1"
    }
```

タグライブラリ

タグライブラリを使うと、オン・ザ・フライで Markdown を HTML に変換します。

タグは、<markdown:renderHtml> です。

■タグのボディに Markdown を指定

<markdown:renderHtml> タグのボディに Markdown 形式のテキストを指定すると、<markdown:renderHtml> と </markdown:renderHtml> で囲まれた部分に記述された内容が HTML に変換されます。

```
<markdown:renderHtml>これは Markdown の *テスト* です。</markdown:renderHtml>
```

上記コードは次の HTML コードに変換されます。

```
<p>これは Markdown の <em>テスト</em> です。</p>
```

■タグの属性にMarkdownを指定

<markdown:renderHtml>タグのtext属性にMarkdown形式のテキストを指定すると、text属性の値がHTMLに変換されます。

```
<markdown:renderHtml text="**別の** Markdownのテストです。"/>
```

上記コードは次のHTMLコードに変換されます。

```
<p><strong>別の</strong> Markdownのテストです。</p>
```

■タグの属性にMarkdownを記述したテンプレートファイルを指定

<markdown:renderHtml>タグには、text属性の他にtemplate属性を指定することができ、Markdown形式のテキストを記述したテンプレートファイルを指定することができ、HTMLに変換され取り込まれます。

テンプレートファイル_readme.gspに、次のように<markdown:renderHtml>タグを含む内容を記述し、

```
<p>
<markdown:renderHtml>これはMarkdownの*テスト*です。</markdown:renderHtml>
</p>
<p>
<markdown:renderHtml text="**別の** Markdownのテストです。"/>
</p>
```

別のGSPファイルからtemplate属性で上記テンプレートファイルreadme.gspを指定します。

```
<markdown:renderHtml template="readme" />
```

実行結果は次のHTMLコードに変換されます。

```
<p>
<p>
<p>これはMarkdownの<em>テスト</em>です。</p>
</p>
<p>
<p><strong>別の</strong> Markdownのテストです。</p>
</p>
</p>
```

なお、template属性とtext属性が両方とも指定された場合、template属性が優先されます。

Stringクラスの拡張

Grails Markdownプラグインは、Stringクラスを拡張し、次のメソッドを追加します。

- String markdownToHtml()
- String htmlToMarkdown()

■markdownToHtmlメソッド

Markdown形式の文字列をHTML形式の文字列に変換します。

```
println "これはMarkdownの*テスト*です。"
    .markdownToHtml()
```

上記コードの実行結果は次のとおりです。

```
<p>これはMarkdownの<em>テスト</em>です。</p>
```

■htmlToMarkdownメソッド

HTML形式の文字列をMarkdown形式の文字列に変換します。

```
println "これはMarkdownの<em>テスト</em>です。"
    .htmlToMarkdown()
```

上記コードの実行結果は次のとおりです。

```
これはMarkdownの*テスト*です。
```

サービスクラス

Grails Markdownプラグインは、Markdown処理用にMarkdownSeriviceクラスを提供しており、markdownServiceという名前でコントロールクラスやサービスクラスにインジェクトできます。

```
class BookController {
    ....
    def markdownService
    ....
    String convertToHTML(String text) {
        ....
        markdownService.markdown(text)
        ....
}
```

また、markdownServiceはprocessorとremarkというプロパティを提供しており、それぞれ、org.pegdown.PegDownProcessorクラスのインスタンス、remarkはcom.overzealous.remark.Remarkクラスのインスタンスとなります。

MarkdownServiceクラスは次のメソッドが定義されています。

- String markdown(def text, def conf = null)
- String htmlToMarkdown(def text, def customBaseUri = "", def conf = null)
- String sanitize(def text, def conf = null)

■markdownメソッド

markdownメソッドは、Markdown形式の文字列をHTML形式の文字列に変換します。オプションで設定(後述)を指定できます。

```
println markdownService.markdown(
    "これはMarkdownの*テスト*です。")
```

の実行結果は次のとおりです。

```
<p>これはMarkdownの<em>テスト</em>です。</p>
```

■htmlToMarkdownメソッド

htmlToMarkdownメソッドは、HTML形式の文字列をMarkdown形式の文字列に変換します。

```
println markdownService.htmlToMarkdown(
    "これはMarkdownの<em>テスト</em>です。")
```

の実行結果は次のとおりです。

```
これはMarkdownの*テスト*です。
```

■sanitizeメソッド

sanitizeメソッドは、Markdown形式の文字列をサニタイジングします。

```
println markdownService.sanitize(
    "これはMarkdownの<em>*テスト*</em>です。")
```

の実行結果は次のとおりです。

```
これはMarkdownの*テスト*です。
```

また、

```
println markdownService.sanitize(
    "これはMarkdownの<p>*テスト*</p>です。")
```

の実行結果は次のとおりです。

```
これはMarkdownの
*テスト*
です。
```

設定

Grails Markdownプラグインは、デフォルトの設定以外に、Config.groovyに記述することで、またはメソッドの呼び出しの引数にマップの形式で値を指定することができます。

■テキストの折り返し

```
markdown.hardwraps = true          // Config.groovy
[hardwraps: true]                  // メソッドの引数
```

Markdownでテキストの折り返しをする場合、行の末尾にスペースを2つ置く必要があります。

hardwrapsにtrueを設定した場合、行の末尾にスペースを2つ置くことなく、改行位置でテキストを折り返します。

```
<markdown:renderHtml>これは
Markdownの*テスト*です。</markdown:renderHtml>
```

というテキストに対し、hardwrapをtrueに設定した場合

```
<p>これは<br/>Markdownの<em>テスト</em>です。</p>
```

と出力され、hardwrapをfalseに設定した場合

```
<p>これは Markdownの<em>テスト</em>です。</p>
```

と出力されます。

■自動リンク

```
markdown.autoLinks = true          // Config.groovy
[autoLinks: true]                  // メソッドの引数
```

Markdown形式のテキスト中に、http:やhttps:で始まるURLをリンクに変換します。

Markdown形式の場合、リンクを出力するには次のように鍵括弧と丸括弧を使用します。

```
<markdown:renderHtml>[Google](http://www.google.
co.jp/)</markdown:renderHtml>
```

次のように単純に記述されたURLに対し、

```
<markdown:renderHtml>http://www.google.co.jp/</
markdown:renderHtml>
```

autoLinksをtrueに設定した場合

```
<p><a href="http://www.google.co.jp/">http://www.
google.co.jp/</a></p>
```

と出力され、autoLinksをfalseに設定した場合

```
<p>http://www.google.co.jp/</p>
```

と出力されます。

■略語

```
markdown.abbreviations = true    // Config.groovy
[abbreviations: true]            // メソッドの引数
```

Markdown形式で略語の指定ができます。略語は<abbr>タグで囲まれます。

次のようにMarkdown形式で記述した場合、

```
<markdown:renderHtml>
This is GSP.

*[GSP]: Groovy Server Pages
</markdown:renderHtml>
```

abbreviationsをtrueに設定した場合

```
<p>This is <abbr title="Groovy Server
Pages">GSP</abbr>.</p>
```

と出力され、abbreviationsをfalseに設定した場合

```
<p>This is GSP.</p><p>*[GSP]: Groovy Server
Pages</p>
```

と出力されます。

■定義リスト

```
markdown.definitionLists = true    // Config.groovy
[definitionLists: true]            // メソッドの引数
```

Markdown形式で<dl>タグのリストを可能にします。
次のようにMarkdown形式で記述した場合、

```
<markdown:renderHtml>
Grails
:   A rapid web-application development platform
for the JVM.
</markdown:renderHtml>
```

definitionListsをtrueに設定した場合

```
<dl><dt>Grails</dt><dd>A rapid web-application
development platform for the JVM.</dd>
</dl>
```

と出力され、definitionListsをfalseに設定した場合

```
<p>Grails : A rapid web-application development
platform for the JVM.</p>
```

と出力されます。

■スマートな引用符、句読点

```
markdown.smartQuotes = true        // Config.groovy
[smartQuotes: true]                //メソッドの引数
markdown.smartPunctuation = true   // Config.groovy
[smartPunctuation: true]           //メソッドの引数
※両方ともtrueの場合の別の記法
markdown.smart = true              // Config.groovy
[smart: true]                      //メソッドの引数
```

引用符やハイフンをHTMLエンティティに変換します。
次のようにMarkdown形式で記述した場合、

```
<markdown:renderHtml>
"Grails" --- 'Groovy'
</markdown:renderHtml>
```

smartをtrueに設定した場合

```
<p>&ldquo;Grails&rdquo; &mdash;
&lsquo;Groovy&rsquo;</p>
```

と出力され、smartをfalseに設定した場合

```
<p>"Grails" --- 'Groovy'</p>
```

と出力されます。

■コードブロック

```
markdown.fencedCodeBlocks = true   // Config.groovy
[fencedCodeBlocks: true]           //メソッドの引数
```

Markdown形式でそのままコードなどを表示する際、スペース4つのインデントを付ける必要がありますが、チルダを3つ、あるいはバッククォートを3つで囲むことで、同様のことができます。
次のようにMarkdown形式で記述した場合、

```
<markdown:renderHtml>
~~~
Map<String, Object> map = new HashMap<String,
Object>();
map.put("foo", 1);
map.put("bar", 2);
~~~
</markdown:renderHtml>
```

fencedCodeBlocksをtrueに設定した場合

```
<pre><code>Map&lt;String, Object&gt; map = new
HashMap&lt;String, Object&gt;();
map.put(&quot;foo&quot;, 1);
map.put(&quot;bar&quot;, 2);
</code></pre>
```

と出力され、fencedCodeBlocksをfalseに設定した場合

```
<p>~~~ Map&lt;String, Object&gt; map = new
HashMap&lt;String, Object&gt;(); map.put("foo",
1); map.put("bar", 2); ~~~</p>
```

と出力されます。

■テーブル

```
markdown.tables = true          // Config.groovy
[tables: true]                  // メソッドの引数
```

Markdown ExtraまたはMultimarkdown形式でHTMLテーブルを出力することができます。

次のようにMarkdown形式で記述した場合、

```
<markdown:renderHtml>
|              |          Grouping           ||
| First Header | Second Header | Third Header |
|:------------ |:-------------:| ------------:|
| Content      |         *Long Cell*         ||
| Content      |   **Cell**    |         Cell |
| New Section  |     More      |         Data |
| And more     |          And more           ||
</markdown:renderHtml>
```

tablesをtrueに設定した場合

```
<table>
  <thead>
    <tr>
      <th align="left"> </th>
      <th align="center" colspan="2">Grouping </th>
    </tr>
    <tr>
      <th align="left">First Header </th>
      <th align="center">Second Header </th>
      <th align="right">Third Header </th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td align="left">Content </td>
      <td align="center" colspan="2">
        <em>Long Cell</em>
      </td>
    </tr>
    <tr>
      <td align="left">Content </td>
      <td align="center"><strong>Cell</strong> </td>
      <td align="right">Cell </td>
    </tr>
    <tr>
      <td align="left">New Section </td>
      <td align="center">More </td>
      <td align="right">Data </td>
    </tr>
    <tr>
      <td align="left">And more </td>
      <td align="center" colspan="2">And more </td>
    </tr>
  </tbody>
</table>
```

と出力され、tablesをfalseに設定した場合

```
<p>| | Grouping || | First Header | Second Header
| Third Header | |:------------ |:-------------
:| ------------:| | Content | <em>Long Cell</em>
|| | Content | <strong>Cell</strong> | Cell | |
New Section | More | Data | | And more | And more
||</p>
```

と出力されます。

■全部入り

```
markdown.all = true             // Config.groovy
[all: true]                     // メソッドの引数
```

次に挙げる設定を一括でtrueまたはfalseにします。

- Hardwraps
- Auto Links
- Abbreviations
- Definition Lists
- Smart Quotes
- Smart Punctuation
- Fenced Code Blocks
- Tables

■HTMLのタグの削除

```
markdown.removeHtml = true      // Config.groovy
[removeHtml: true]              // メソッドの引数
```

Markdown形式の文字列をHTML形式の文字列に変換する際、HTMLのタグを削除するかどうかを指定します。

次のようなコードがある場合、

```
println "これはMarkdownの<html><body><p>*テスト
*</p></body></html>です。".markdownToHtml()
```

removeHtmlをtrueに設定した場合

```
<p>これはMarkdownの<em>テスト</em>です。</p>
```

と出力され、removeHtmlをfalseに設定した場合

```
<p>これはMarkdownの<html><body><p><em>テスト</
em></p></body></html>です。</p>
```

と出力されます。

■テーブルの削除

```
markdown.removeTables = true     // Config.groovy
[removeTables: true]             // メソッドの引数
```

HTML形式の文字列をMarkdown形式の文字列に変換する際、テーブルのタグを削除するかどうかを指定します。

次のようなコードがある場合、

```
println "これはMarkdownの<em>テスト</em>です。これは<table><tr><td>テーブル</td></tr><table>です。".htmlToMarkdown()
```

removeTablesをtrueに設定した場合

```
これはMarkdownの*テスト*です。これは
```

と出力され、removeTablesをfalseに設定した場合

```
これはMarkdownの*テスト*です。これは
<table>
 <tbody>
  <tr>
   <td>テーブル</td>
  </tr>
 </tbody>
</table>
<table>
 です。
</table>
```

と出力されます。

終わりに

Markdownは、GitHubのREADMEやTumblrなどでも採用され、さらに電子書書籍の入稿フォーマットとしても使っているところもあると聞きます。HTMLが良い場合もあると思いますが、Markdownが適切なところでは、Grails Markdownとともに使用を検討してみてはいかがでしょうか。

リリース情報 2013.11.17

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, 2.1.5, 2.2.4, 2.3.2

■更新情報

- 2.1.5では、ユニットテストの改良やGORMメソッドの実行結果の改善、いくつかバグ対応が行われています。
- 2.1.5リリースノート: http://grails.org/2.1.5+Release+Notes
- 山本さんのブログ: http://d.hatena.ne.jp/mottsnite/20130429/1367230722
- 2.2.4では、いくつかバグ対応が行われています。
- 2.2.4リリースノート: http://grails.org/2.2.4+Release+Notes
- 山本さんのブログ: http://d.hatena.ne.jp/mottsnite/20130730/1375191304
- 2.3.2では、開発モードでドメインクラスの変更後セッションファクトリの最初期化を不可にできるようにしたり、生成されるテストコードが修正されたり、いくつかバグ対応が行われています。
- 2.3.2リリースノート: http://grails.org/2.3.2+Release+Notes
- 山本さんのブログ: http://d.hatena.ne.jp/mottsnite/20131108/1383858545

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, 2.0.8, 2.1.9, 2.2.0-rc-3

■更新情報

- 2.0.8では、いくつかバグ対応が行われています。
- 2.0.8リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=19099
- 2.1.9では、@CompileStaticを使用した際のバグを含め、いくつかバグ対応が行われています。
- 2.1.9リリースノート: https://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=19607
- 2.2.0-rc-3では、Groovyshで使用するクラスが移動され、いくつかバグ対応が行われています。
- 2.2.0-rc-3リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=19632

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: 1.4.0

■更新情報

- 1.4.0では、ライセンスの自動パッケージ機能が追加されたり、InjectedResourceアノテーションによるプロパティインジェクションが改良されたり、いくつかバグ対応が行われています。
- 1.4.0リリースノート: http://docs.codehaus.org/display/GRIFFON/Griffon+1.4.0

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

GMaven

GMavenは、Maven用のGroovyプラグインです。
URL: http://gmaven.codehaus.org/
バージョン: 1.4

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: 1.8

■更新情報

- 1.8では、C/C++/アセンブリに対応したり、実行速度や使用メモリが改善されたり、いくつかバグ対応が行われています。
- 1.8リリースノート: http://www.gradle.org/docs/1.8/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: 2.0

■更新情報

- 2.0では、GAE SDK 1.8.0とGroovy 2.1.3に対応し、新しい検索用DSLが追加されたり、いくつかバグ対応が行われています。
- 2.0リリースノート: http://gaelyk.appspot.com/download

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: 1.8.7

■更新情報

- 1.8.7では、クラウド・エンドポイントがGA(General Availability)となり、max_concurrent_requests設定がバージョン/モジュールごとに設定できるようになったり、いくつかバグ対応が行われています。
- 1.8.7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: 1.1.0GA

■更新情報

- 1.1.0GAでは、LazyDataflowVariableやPromiseベースのAPIが追加されたり、いくつかバグ対応が行われています。
- 1.1.0GAリリースノート: http://docs.codehaus.org/display/GPARS/2013/07/25/Here+comes+GPars+1.1

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

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: 0.13

■更新情報

- 0.13では、groovyclientでサーバの操作ができるオプションが追加されたり、GROOVYSERVER_HOST環境変数がサポートされたり、いくつかバグ対応が行われています。
- 0.13リリースノート: http://kobo.github.io/groovyserv/changelog.html

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: 0.9.2

■更新情報

- 0.9.2では、NavigatorにisEnabled()とisEditable()が追加されたり、Grails 2.3互換となったり、いくつかバグ対応が行われています。
- 0.9.2リリースノート: http://www.gebish.org/manual/0.9.2/project.html#092

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: 1.5

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: 0.8.3

■更新情報

- 0.8.3では、Groovy 1.8.4にアップグレードし、Grails 2.0と互換がとられ、いくつかバグ対応が行われています。
- 0.8.3リリースノート: http://grails.1312388.n4.nabble.com/Release-of-gmock-0-8-3-td4646065.html

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

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: 0.19

■更新情報

- 0.19では、新しく13ルールが追加され、いくつかバグ対応が行われています。
- 0.19リリースノート: http://groovy.329449.n5.nabble.com/ANN-Announcing-CodeNarc-0-19-td5716250.html

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: http://gcontracts.org/
バージョン: 1.2.12

■更新情報

- 1.2.12のリリース内容は不明です。

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

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: 0.4.2

■更新情報

- 0.4.2では、新しいシステムプロパティをサポートしたり、いくつかバグ対応が行われています。
- 0.4.2リリースノート: https://code.google.com/p/gbench/wiki/ReleaseNotes042

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

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.cfapps.io/
バージョン: 1.1

■更新情報

- 1.1では、Groovy 2.1.xに対応しました。
- 1.1リリースissue: http://caelyf.cfapps.io/download

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: 2.0.2, 2.1M1

■更新情報

- 2.0.2、2.1M1とも更新情報は不明です。

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.7 2013.12
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