vol.4

Magazine

Japan Grails/Groovy User Group

Contents

祝 !! Grails 2.0.0 リリース

検証！Grails 2.0 の世界 Pt.2

玉田 幸治（株式会社ニューキャスト）
基本的にきゃっきゃ♪ゆうとるひげのおっさん。

Grails 2.0 - 1.4にしておくにはもったいない!

今から半年ほど前の2011年6月27日。Grails開発チームの発表にあったように、1.4系にしておくのはもったいないとのことで、Grails1.4系は2.0系となりました。詳細はJGGUGのG*ワークショップにて解説されたtyamaさんのスライドをご覧ください。

さてどの辺りが「もったいない！」程の内容なのかを探るシリーズ第二弾ですが、企画開始(前々号)から、今回分を(多忙な日々をかいくぐり)執筆してる間に、ついに2011年12月15日(JST)に、Grails-2.0が正式リリースされました！

このシリーズは、今後**「Grails 2.0の世界」**として継続していく"ハズ"です！

では、今回は、大きな変更点の一つ「Resourcesプラグイン」を通称"ヒゲのおっさん"がお送りします！

■Resourcesプラグイン

Grails-2.0には、今回の目玉商品である"Resourcesプラグイン"がデフォルトでインストールされています。Grails 2.0.0には、バージョン1.1.6のResourcesプラグインがインストールされます。このResoucesプラグインとは静的リソースを管理する為の"リソースフレームワークプラグイン"です。

通常のGrailsアプリケーションではリソース指定のタグを利用してリソースを管理(指定)していました。しかし、最近のWebアプリケーションはJavaScript & CSSのライブラリやフレームワークに依存して開発することが多くなってきているため、管理などが複雑で大変になってきます。

そこで、Resourcesプラグインでは、静的リソースの管理、リソース処理の為のマッパー、リソース提供用のサーブレットフィルタ等を、わかりやすいDSLで管理可能な"フレームワーク"として提供しています。

まずは、Grails2.0のBuildConfig.groovyを確認してみます。

```
plugins {
    compile ":hibernate:$grailsVersion"
    compile ":jquery:1.7.1"
    compile ":resources:1.1.5"

    build ":tomcat:$grailsVersion"
}
```

pluginsの中でしっかりとResourcesプラグインが定義されています。Resourcesプラグイン対応のjqueryプラグインもインストールされています。

使用方法

■準備編

grails-app/views/layouts/main.gspのheadタグ内とコンテンツの最後に<r:layoutResources/>タグを挿入します。

```
<!doctype html>
            ...
    <head>
        <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8">
        <meta http-equiv="X-UA-Compatible"
          content="IE=edge,chrome=1">
        <title><g:layoutTitle default="Grails"/></title>
            ...
        <r:layoutResources/>
        <g:layoutHead/>
    </head>
    <body>
            ...
        <r:layoutResources/>
        <g:javascript library="application"/>
    </body>
</html>
```

head内とコンテンツの最後の部分へタグを挿入します。これでResourcesプラグインを使用する準備は完了です。静的リソースはこの挿入したタグの部分に読み込まれる事になります。

■JavaScriptを挿入

準備編で挿入したlayoutResourcesタグ部分へjavascriptを挿入してみます。

grails-app/views/index.gspのbody開始タグ直後に以下のコードを書いてgrailsアプリケーションにブラウザからアクセスしてみましょう。

```
<r:script disposition="head">alert('コンテンツ前');</r:script>
<r:script disposition="defer">alert('コンテンツ後');</r:script>
```

コンテンツの読み込み前と読み込み後にalertが表示されます。htmlのソースコードを見てみましょう。

```
<!doctype html>
            ...
    <head>
            ...
        <script type="text/javascript">
                alert('コンテンツ前');
        </script>
            ...
    </head>
    <body>
            ...
        <div class="footer" role="contentinfo"></div>
        <div id="spinner" class="spinner"
         style="display:none;">
                Loading&hellip;
        </div>
        <script type="text/javascript">
                alert('コンテンツ後');
        </script>
    </body>
</html>
```

scriptタグの差し込み先がheadタグ内とコンテンツの最後にわかれて挿入されているのが確認できます。disposition="head"でhead内のlayoutResourcesへ挿入され、disposition="defer"でコンテンツの最後のlayoutResourcesへ挿入されます。

※headタグ属性で指定されたものは最初にみつかったlayoutResourcesタグ内に挿入されます。headタグ内にlayoutResourcesタグがなくてコンテンツの最後の箇所にしかlayoutResourcesタグがない場合はそちらに挿入されます。

cssはdisposition=headでヘッダー内のlayoutResoucesに読み込みを行い、コンテンツ読み込み後に実行させたいjavaScriptや、ページロードに影響を与えるjavaScriptはdisposition=deferでコンテンツの最後に読み込むべきです。

※<r:script>タグで定義されたjavaScriptはrequireで読み込まれたライブラリより後に配備されます。必要なライブラリが先に読み込まれている必要がある為にこのような動きになっています。

コンテンツ後は読み込まれた後に実行されていることがわかります。

■画像の挿入

web-app/images内にある懐かしい画像を挿入してみましょう。

```
<r:img dir="images" file="grails_logo.jpg" />
```

以下のように変換され懐かしい画像が表示されます。

```
<img src="/あなたのアプリ/static/images/grails_logo.jpg" />
```

設定ファイル

grails-app/conf内にリソース定義ファイル(SomeNameResources.groovyという形で'SomeName'に任意の名前を付けたgroovyファイルを配置)、もしくは
grails-app/conf/Config.groovyに以下のように記述します。

```
grails.resources.modules = {
    module名 {
        dependsOn 'jquery' //依存がある場合
        defaultBundle 'defaultBundle' //バンドル名
        resource url:リソースファイルのパス,
                disposition: 'head', or 'defer'
    }
}
```

リソース定義ファイルは複数のファイルを使用することが可能です。jqueryプラグインなどはプラグインの中に既にResourcesプラグイン用の設定ファイルが含まれています。プラグインの中にリソース定義ファイルを配置することでResourcesプラグイン対応のプラグインを作成することが可能です。

■モジュール

Resourcesプラグインではリソースのファイルをモジュールという単位で管理しています。複数のリソースファイルをモジュールという単位に統合し、gspでモジュールをrequireタグで指定することで利用が可能です。基本的なリソースをcommonモジュールとしてまとめて、管理画面等の別のリソースを必要とする場合、commonモジュールをdependsOnした管理画面用のadminモジュールを作るなどの手法が考えられます。

```
モジュール名称 {
    dependsOn 'jquery' // 依存関係の定義
                       設定してある場合は設定して
                       あるモジュールが先に読み込
                       まれます。
    defaultBundle 'monolith' // バンドル名称
    resource url: '/js/core.js', disposition: 'head'
                              // モジュールのリソース
}
```

■バンドル

バンドルというセットを利用してリソースの上書きが可能になります。リソースを上書きする場合、"overrides"を利用します。

```
overrides {
    jquery { defaultBundle '上書するためのバンドル' }
}
```

このようにdefaultBundleを別のバンドルでoverridesすることができます。

たとえば、jQueryと他のライブラリをまとめてバンドルして、上書きする場合は以下のようになります。

```
modules = {
    core {
        dependsOn 'jquery, utils'
        defaultBundle 'monolith'
        resource url: '/js/core.js', disposition: 'head'
    }
    utils {
        dependsOn 'jquery'
        defaultBundle 'monolith'
        resource url: '/js/utils.js'
    }
    forms {
        dependsOn 'core,utils'
        defaultBundle 'monolith'
        resource url: '/css/forms.css' resource url: '/js/forms.js'
    }
    overrides {
        jquery { defaultBundle 'monolith' }
    }
}
```

■マッパー定義

リソースフレームワークは"マッパー"を使用して、最終フォーマットへリソースを変化させてユーザへ配信します。時にはこのマッパーで処理させたくないリソースもあります。

特定のファイルを指定してマッパーの処理を行わないように定義することも可能です。例えば、"hashandcache"マッパーの処理をさせたくない場合は、

```
resource url: '/css/my.css', nohashandcache: true
```

上記のように、noマッパー名称:trueのように指定します。

また、ファイルタイプによって、マッパーの定義をする場合はgrails-app/conf/Config.groovy内に以下のように指定が可能です。

```
grails.resources.zip.excludes = ['**/*.zip', '**/*.exe']
```

※zipファイルとexeファイルをzipマッパーを通さない場合の設定(zipマッパーはzipped-resourcesで提供されています。)

```
grails.resources.bundle.excludes = ['**/*.less']
```

※ファイルタイプlessをバンドルしないようにする。

コラム

jqueryプラグインの中の設定ファイルはどうなっているのか

jqueryプラグインの中のresourcesプラグイン用のリソース設定ファイルJQueryPluginResources.groovyを見てみます。

```
// Resource declarations for Resources plugin
def jqver = org.codehaus.groovy.grails.plugins.
jquery.JQueryConfig.SHIPPED_VERSION
modules = {
    'jquery' {
        resource id:'js', url:[plugin: 'jquery',
        dir:'js/jquery', file:"jquery-${jqver}.min.js"],
        disposition:'head', nominify: true
    }

    'jquery-dev' {
        resource id:'js', url:[plugin: 'jquery',
        dir:'js/jquery', file:"jquery-${jqver}.js"],
        disposition:'head'
    }
}
```

<r:require modules="jquery"/>を利用することにより圧縮されたjquery-minを読み込むことができ、<r:require modules="jquery-dev"/>で、圧縮前のjqueryライブラリを読み込むことが可能になっているのがわかります。このように開発環境とプロダクション環境をモジュールによって分けられるのもresourcesプラグインの利点といえます。

開発中時のデバッグ指定

Resourcesプラグインにはデバッグの際に使用する機能が幾つか提供されています。開発環境ではX-Grails-Resources-Original-Srcヘッダーが追加されます。このヘッダーにはリソースを構成した元のリソースが示されています。

また、URLに_debugResources=yのクエリパラメータを付加することによりResourcesプラグインは元のリソースファイルを読み込むことが可能となります。また、ブラウザによるキャッシュを防ぐため、ユニークなタイムスタンプが付加されます。毎回クエリに_debugResources=yを付けなくてもいいように、

```
grails.resources.debug=true
```

とgrails-app/conf/Config.groovyに記述することができます。これにより、_debugResources=yのクエリパラメータを省いての開発が可能になります。開発中はdevelopment環境内に追記しておくことをおすすめします。

関連プラグイン

■zipped-resources プラグイン

ページロードの時間と帯域を短縮するためのプラグインです。gif, jpeg, pngの既に圧縮されているファイルを除いたコンテンツを自動的に圧縮するマッパーが用意されています。

```
grails install-plugin zipped-resources
```

このように、zipped-resourcesプラグインはインストールするだけで使用できます。

■cached-resources プラグイン

クライアントのブラウザにリソースをキャッシュさせる仕組みのプラグインです。リソースのファイル名をそれぞれのハッシュ値に変換し、キャッシングヘッダーを全てのリソースのレスポンスにセットすることによりブラウザにキャッシュさせ、ページロード時間の短縮を行います。

```
grails install-plugin cached-resources
```

同じくcached-resourcesプラグインはインストールするだけで使用できます。

まとめ

Grails 2.0の世界として、駆け足でResourcesプラグインの紹介をしてきましたがこのプラグインを使用することで静的リソースの管理が楽になることがおわかりいただけたでしょうか？まだまだ紹介しきれていない部分もありますので公式のドキュメントを一度みていただくことをおすすめします。それではまたお会いできる時まで! 基本的にきゃっきゃうふとるヒゲのおっさんがお送りしました。

リンク

- tyamaさんのjggug発表スライド http://slidesha.re/n43zfq
- Resouces プラグインドキュメント http://grails-plugins.github.com/grails-resources/

series 07

JGGUG合宿2011企画アプリ
GDK48 投票アプリの作り方（前編）

奥 清隆（おく きよたか）
仕事でもときどき 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』

GDK48投票アプリの作り方(前編)

今回は「Griffon不定期便」をお休みして、JGGUG合宿2011で行われたGDK48という企画で使われたアプリケーションついてお話ししたいと思います。

GDK48とは、みんなでG*なコードを書いて、それをみんなで投票して1位を決めようという企画でした。企画の詳細な内容については合宿運営レンジャーレッドの資料を御覧ください。

http://www.slideshare.net/kazuchika/gdk48

GDK48と言っておきながら、残念なことに48本までは集まりませんでしたが、大変役に立つコードが集まりました。合宿で使用したアプリケーションは下記のURLで現在も見ることができます。

http://gdk48.kiy0taka.org/gdk48/

今回はこのGDK48総選挙システムの作り方についてお話ししていきます。GDK48総選挙システムのソースコードはGitHub上で公開していますので、合わせて御覧ください。

http://git.io/gdk48

■GDK48総選挙システム概要

GDK48総選挙システムはとてもシンプルなものです。ストーリーとしては次の2つになります。

- Gistに投稿したURLをツイートしてGDK48にエントリーする
- Twitterアカウントで認証済みのユーザがGDK48に投票する

今回はこの「Gistに投稿したURLをツイートしてGDK48にエントリーする」の実装についてお話ししていきます。

■アプリケーションの作成

GDK48総選挙システムはシンプルなアプリケーションなのでGaelykで作るという選択肢もありましたが、Twitter連携などの必要な機能がプラグインで提供されているためGrailsを使って実装しました。Grailsのバージョンは1.3.7を使いました。Grails2.0が出たところですが、この記事でも1.3.7で作成していきます。Grails2.0については山本さんの記事を熟読してください。それではgdk48という名前でアプリケーションを作成しましょう。

```
grails create-app gdk48
```

Gistに投稿したURLをツイートしてGDK48にエントリーする

GDK48にエントリーする流れは次のような感じです。

1. GDK48エントリー用のコードを書く
2. 書いたコードをGist(http://gist.github.com)に貼り付ける
3. 貼りつけたGistのURLとハッシュタグ「#gdk48」を付けてツイートする
   例）「GroovyでGitコマンド。
   https://gist.github.com/1341311 #gdk48」

ハッシュタグとGistのURLがあればGDK48総選挙システムが勝手に拾ってくれてエントリーされます。

GDK48エントリ用ドメインクラスの作成

それではエントリ用のドメインクラスを作成しましょう。エントリ用ドメインクラスと言っても、総選挙システムで管理するのはGistのIDだけにします。GistのIDとは、Gistにコードを投稿したときに割り当てられるURLの後ろにつく数字です。例えば、「https://gist.github.com/1341311」というURLなら、GistのIDは「1341311」になります。コードの内容などはエントリー後に修正されることもあるのでシステムではコード内容を保持せず、Gistへのリンクが生成できれば良しとします。クラス名はGistとしてドメインクラスを作成します。

```
cd gdk48
grails create-domain-class Gist
```

grails-app/domian/gdk48/Gist.groovyが生成されました。GistクラスにはGistのIDだけあればいいので、次のように編集します。

```
package gdk48

class Gist {

    Long gistNo

    static constraints = {
        gistNo nullable:false, unique:true
    }
}
```

プロパティgistNoがGistのIDとなります。gistNoはnull値を許可せず、ユニークな値になるように制約を定義しておきます。

Gistコントローラの作成

ドメインクラスが出来たのでコントローラを作成しましょう。

```
grails create-controller Gist
```

grails-app/controllers/gdk48/GistController.groovyが生成されました。GDK48総選挙システムでは、CRUDの画面は必要ないのですが、とりあえず動かしてみたい人はスカッフォルドで動きを確認してみましょう。GistController.groovyを次のように編集します。

```
package gdk48

class GistController {

    static scaffold = true
}
```

この状態でアプリケーションを起動すれば、スカッフォルドされたCRUDの画面を動かせます。「grails run-app」コマンドを実行してからブラウザで「http://localhost:8080/gdk48/gist」にアクセスしてみましょう。最終的にこのCRUDの画面は必要ではないので詳細は省きますが、スカッフォルドでアプリケーションを動かすことで、システムのイメージが掴めることもあるかと思います。コントローラに少し工夫をするだけでCRUDの画面が確認できるスカッフォルド機能はとても便利ですね。

それでは、実際に必要な処理を実装しましょう。GistControllerに必要なアクションはエントリされたGistの一覧を表示するためのアクションです。一覧表示用にlistという名前のクロージャを定義します。

```
package gdk48

class GistController {

    def list = {
        def gists = Gist.list(max:5,
            offset:(params.offset?:'0').toLong(),
            sort:'gistNo',
            order:'desc')
        [gists:gists]
    }
}
```

listクロージャではエントリされたGistの最大5件まで取得します。リクエストパラメータoffsetで開始件数を指定すればページングするようにします。検索結果はGistの新しい順（gistNoの昇順）でソートします。リクエストパラメータの入力チェックなど細かいことは気にしないことにします。

一覧画面の作成

Gistの一覧を表示する画面を作りましょう。grails-app/views/gist/list.gspファイルを次のように作成します。

```
<%@ page import="gdk48.*" %>
<html>
<head>
  <meta name="layout" content="main" />
  <title>GDK48総選挙</title>
</head>
<body>
<div class="container">
  <h1>GDK48 総選挙</h1>
  <div class="paginateButtons">
    <g:paginate controller="gist" action="list"
        total="${Gist.count()}" max="5" />
  </div>
  <div>
    <g:each var="gist" in="${gists}">
    <div class="gist">
      <script
        src="https://gist.github.com/${gist.gistNo}.js">
      </script>
    </div>
    </g:each>
  </div>
  <div class="paginateButtons">
    <g:paginate controller="gist" action="list"
        total="${Gist.count()}" max="5" />
  </div>
</div>
</body>
</html>
```

ページング用のタグと、scriptタグを使ってGistのコードを表示しています。Gistでは「https://gist.github.com/1341311.js」のようなURLでGistコードを表示するためのJavaScriptコードを取得することができます。

■テストデータの登録

この段階で一覧画面を確認しても、Gistを登録する部分を実装していないのでデータは何も表示されません。しばらくはBootStrapを使って適当なGistを登録しておきましょう。grails-app/conf/BootStrap.groovyを次のように編集します。

```
import grails.util.Environment
import gdk48.Gist

class BootStrap {

    def init = { servletContext ->
        if (Environment.current == Environment.DEVELOPMENT) {
            [1261979, 1129161, 1059105, 1031115,
             1020384, 1020286, 1015558, 1012403].each {
              new Gist(gistNo:it).save(failOnError:true,
                                       flush:true)
             }
        }
    }
    def destroy = {
    }
}
```

アプリケーション初期化のタイミングで、適当なGistを何件か登録する処理を追加しました。それでは「grails run-app」コマンドでアプリケーションを起動して「http://localhost:8080/gdk48/gist/list」にアクセスしてみましょう。

次のような画面が表示されれば成功です。

■UIを改善する

Gistを一覧することが出来ましたが、見た目がすこし残念な感じなので、少しUIを改善しましょう。Blueprint CSS(http://blueprintcss.org/)というCSSフレームワークを使ってみます。GrailsにはBlueprint CSS用のプラグインがありますのでインストールします。

```
grails install-plugin blueprint
```

それではBlueprint CSSを使ってlist.gspを改善しましょう。

```
<%@ page import="gdk48.*" %>
<html>
<head>
  <meta name="layout" content="main" />
  <title>GDK48総選挙</title>
  <blueprint:resources/>
  <style>
  body {
    background-color: #FAEBD7;
  }
  .gist {
    margin-bottom: 30px;
  }
  </style>
</head>
<body>
<div class="container">
  <h1>GDK48 総選挙</h1>
  <div class="paginateButtons">
    <g:paginate controller="gist" action="list"
        total="${Gist.count()}" max="5" />
  </div>
  <div class="span-26 last"
        style="background-color: #FFB6C1; padding: 30px;">
    <g:each var="gist" in="${gists}">
    <div class="gist">
      <script src="https://gist.github.com/${gist.gistNo}.js">
      </script>
    </div>
    </g:each>
  </div>
  <div class="paginateButtons">
    <g:paginate controller="gist" action="list"
        total="${Gist.count()}" max="5" />
  </div>
</div>
</body>
</html>
```

これで次のような画面になりました。

まだ残念な感じがするかもしれませんが、これが私の限界です。Grailsには他にもUI系のプラグインがたくさんあるので、好きなものを使ってみるといいかと思います。

GDK48エントリTweetを取得する

それでは、GDK48エントリTweetを取得する処理を実装しましょう。処理の内容は次のとおりです。

1. 定期的にハッシュタグ「#gdk48」が含まれるツイートをTwitterから検索
2. 検索結果のツイートにGistのURLが含まれていればデータを登録

■Jobの作成

定期的に処理を実行するためにQuartzプラグインを使用します。Quartzプラグインは、スケジューリングされた処理を実行するためのJavaライブラリであるQuartz(http://quartz-scheduler.org/)のGrailsプラグインです。

まず、Quartzプラグインをインストールしましょう。

```
grails install-plugin quartz
```

Quartzプラグインをインストールすると「create-job」コマンドを使って定期的に行う処理を実装するJobクラスを作成できます。それではGDK48エントリツイート取得用Jobを作成しましょう。

```
grails create-job TweetCrawler
```

これで、grails-app/jobs/gdk48/TweetCrawlerJob.groovyというファイルが生成されました。TweetCrawlerJobは少しコードが長いので、紙面の都合上ポイントだけ解説しておきます。完全なソースはサンプルコード(http://git.io/gdk48job)を御覧ください。

■トリガーの定義

今回は1分おきにこのJobを実行するようトリガーを定義します。Quartzではクーロン式を使ったトリガーなどいくつかの形式でトリガーを定義できるようになっています。ここでは「simple」タイプのトリガーを定義します。

```
static triggers = {
    simple name: 'TweetCrawlerTrigger',
    startDelay: 10000, repeatInterval: 60000
}
```

name属性は任意のトリガー名、startDelayはアプリケーションが起動してから初めてトリガーが起動されれるまでの時間をミリ秒で指定、repeatIntervalはトリガーを起動する間隔をミリ秒で指定します。

この設定では、「TweetCrawlerTrigger」という名前でアプリケーションが起動してから10秒後に最初のJobが実行され、その後1分間隔でJobが実行されます。

■Jobの並列実行

Jobは1分おきに実行されますが、1分以上処理に時間がかかった場合、2つのJobが平行して実行されます。TweetCrawlerJobは並列に実行されても問題はありませんが、意味が無いので並列に実行されないようにしておきます。TweetCrawlerJobにプロパティconcurrentを定義しておけば、並列実行を制御することができます。

```
def concurrent = false
```

プロパティconcurrentを定義しなかったり、値をtrueにした場合は並列に実行されることを許可します。

■テスト実行時のトリガーの制御

今回はテストコードについては紹介しませんが、サンプルコードにはいくつかテストコードを書いています。デフォルトの設定ではテスト実行時にもトリガーが起動されてしまうので、予期しないタイミングで実行されテストが失敗するケースも出てきてしまいます。テスト時にトリガーを起動しないように次のようなファイルをgrails-app/conf/QuartzConcifg.groovyとして作成しておきます。

```
quartz {
    autoStartup = true
    jdbcStore = false
    waitForJobsToCompleteOnShutdown = true
}

environments {
    test {
        quartz {
            autoStartup = false
        }
    }
}
```

■Twitter4Jのライブラリを追加

Twitterからハッシュタグ「#gdk48」で検索するためにTwitter4Jを使用します。grails-app/config/BuildConfig.groovyにTiwtter4Jの依存関係を追記しておきましょう。

```
repositories {
    grailsPlugins()
    grailsHome()
    grailsCentral()
    mavenLocal()
    mavenCentral()
}
dependencies {
    compile 'org.twitter4j:twitter4j-core:jar:2.2.5'
}
```

■処理の実装

TweetCrawlerJobのメソッドexecute()を実装します。

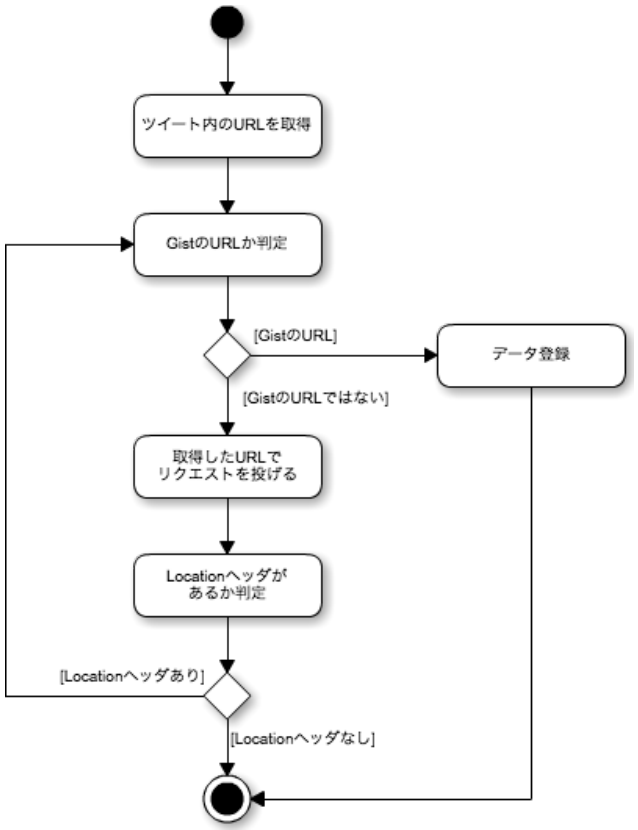
```
def execute() {
    def twitter = new TwitterFactory().instance
    twitter.search(new Query('#gdk48'))
                        .tweets.each { parse it }
}
```

Twitter4Jの検索APIを使ってハッシュタグで検索し、解析してデータを登録するためのメソッドparse()に検索結果を渡します。このまま実行してもおそらく「#gdk48」では検索結果が返ってこないので、適当に変更して実行してみてください。

メソッドparse()については詳細な解説を省きますが、ツイートの本文にGistのURLが含まれるかどうかを判定してデータを登録するように実装します。この処理のポイントとしては次の2点が考えられます。

- URLは短縮URL(http://t.co/BOFnltlQ のような形式など)の場合がある
- 短縮URLは他の短縮URLサービスで短縮されたURLをさらに短縮している場合がある

この2点を考慮して、ツイート内容からGistのURLを取得するために、ツイート内のURLでリクエストを試みてから、短縮URLの場合に返されるLocationヘッダを取得し判定するようにします。処理の流れは次の図のようになります。



Locationヘッダの取得にはjava.net.URLクラスを使って次のようにヘッダだけ取得しています。

```
def conn = new URL(url).openConnection()
conn.followRedirects = false
def location = conn.getHeaderField('Location')
```

まとめ

今回は、GDK48にエントリされたGistを登録し、アプリケーションで一覧するところまでを紹介しました。本当は1回ですべてを紹介する予定でしたが、文章にすると予想以上にボリュームがあったので、続きは次回に回したいと思います。次回はTwitterを使った認証と投票の仕組みについてです。それではまた次回お会いしましょう！

series 08

GroovyFX で遊ぼう！（導入編）

関谷 和愛（せきや かずちか）
Groovy、Java、Apple 製品を愛するおっさんエンジニア。日本 Grails/Groovy ユーザーグループ運営委員長。
共著『プログラミング GROOVY』、共訳『Groovy イン・アクション』

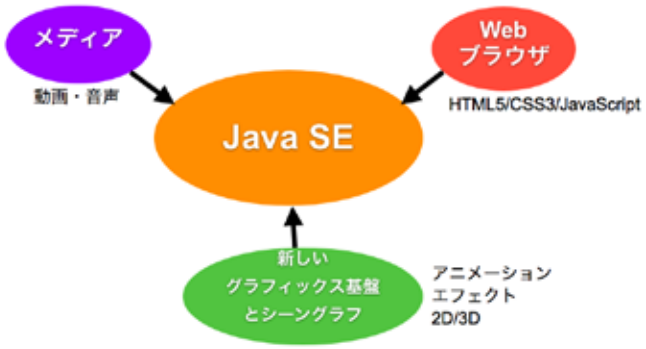
今年10月に開催されたJavaOne 2011で、Javaの次世代クライアントソリューションであるJavaFX 2.0がついに正式リリースされました。そしてG*エコシステムにも、このJavaFXをGroovyに使いこなすためのGroovyFXという新しい仲間が増えました。

このシリーズでは、このGroovyFXを取り上げ、その使い方や魅力を紹介していきたいと思います。

JavaFXって何だっけ？

JavaFXは、Java SEプラットフォームに、従来Javaがあまり得意でなかった下記のような機能を追加する拡張ライブラリです：

- アニメーションやエフェクト、3Dといった高度なグラフィックス
- メディア（動画や音声）の再生と制御
- webブラウザとの統合・連携



Java SEを拡張するJavaFX

JavaFX 2.0は、今日のGPUの性能をフルに引き出せるよう設計された新しい描画エンジンを備え、シーングラフという形で表現されたグラフィックス/UI要素を高速に描画します。シーングラフの各ノードには多彩なエフェクトやトランジションを適用することができます。さらに、WebKitベースのブラウザ実装や、GStreamerベースのメディアプレーヤを内蔵し、これらをシーングラフに埋め込んだり、Javaから自在に制御することもできます。

JavaFXの開発が始まったのはOracleによるSun買収の数年前までさかのぼり、途中何度かの開発方針の見直しなどを経て、現在の形に落ち着きました。現時点では、Windows用の正式版と、Mac OS X用の開発者プレビュー版がリリースされており、2012年にはLinux版も予定されています。

今年のJavaOneでは、オープンソース化とJCP(Java Community Process)での標準化の計画も発表され、さらに、今後のJDK7リリースにバンドルされるようになること、そして次期メジャーアップデートであるJDK8に標準APIとして統合されることなども明らかにされました。

これにより、JavaFXはもはや単なる拡張ライブラリの一つではなく、AWTやSwingの後継として、今後のJavaのクライアントUI開発の主役の座につくことが確実になったのです。

GroovyFX登場！

GroovyFXは、Jim Clarke (Oracle)とDean Iverson (Virginia Tech)によって開発された、GroovyによるJavaFXのラッパーです。Groovy活用法7パターンでいうところの、"Lipstick"パターンに該当します。(「Javaライブラリをラップして使いやすく」)

JavaFX 2.0のAPIはすべてJavaベースなので、Groovyから普通のJavaライブラリのように利用することも可能ですが、GroovyFXでは、さらに次のような機能を提供することによって、JavaFXをよりGroovyらしく、簡単・便利に利用できるようにしています：

- Groovyのビルダーを活用したDSL
  - SceneGraphBuilder: シーングラフの構築
  - TimelineBuilder: タイムラインの構築
- Groovyの言語機能を活用した便利な記法
  - Durationリテラル: 100.ms, 5.s, 3.hなど
  - 疑似定数: red, vertical, ease_outなど
  - 属性値の自動型変換: 文字列→Font、リスト→Point3Dなど

JavaFX 1.xまではJavaFX Scriptという独自の言語が使われており、この言語によって、JavaFXのシーングラフ構築などを簡潔かつ見通しよく記述することができました。しかし、JavaFX 2.0ではこの言語が廃止され、すべてがJavaベースになったため、記述は煩雑になってしまいました。JavaFX Scriptの後継としてGroovyFXを使えば、Groovyが持つ豊かな言語機能と、GroovyFXが提供するJavaFXに特化したサポートの組み合わせが、JavaFX Scriptの抜けた穴を補って余りあるでしょう。

■入手とビルド

残念ながらGroovyFXはまだアルファリリースの段階です。zipやjarなどの便利なバイナリディストリビューションもありませ

ん。利用するためにはまずソースを入手し、自分でビルドする必要があります。

とは言ってもビルドには難しいところはありませんので、下記の手順にそってぜひやってみてください：

1.必要なソフトウェアの用意

- JavaFX SDK (最新バージョン: 2.0.2)
  http://www.oracle.com/technetwork/java/javafx/downloads/index.html
  JavaFXのランタイム。Java SE 7 Update 2からは、Java SEにもバンドルされています。インストール後、環境変数JAVAFX_HOMEにインストール先を設定しておいてください。

- Groovy (最新バージョン: 1.8.4)
  http://groovy.codehaus.org/Download
  Groovyのランタイム。(G* Magazineの読者はインストール済みだと思いますが！)

- Gradle (最新バージョン: 1.0-milestone-6)
  http://www.gradle.org/downloads
  Groovyで作られた汎用ビルドツール。

2.ソースを入手

リポジトリからソースをチェックアウトします。

```
% svn co http://svn.codehaus.org/gmod/groovyfx/trunk/
```

執筆時点のリビジョンは1072でした。

3.ビルド実行

build.gradleのあるディレクトリでgradleコマンドを実行します。

```
% cd groovyfx/trunk/groovyfx/
% gradle build
```

4.クラスパスの設定

build/libsの下にできたgroovyfx-0.1-SNAPSHOT.jarにクラスパスを設定します。もしまだなら、$JAVAFX_HOME/rt/lib/jfxrt.jarもクラスパスに入れておきます。

以上で、GroovyFXの利用準備が整いました。src/demo/groovyの下にたくさんのデモスクリプトがあるので、実行してみたりソースを眺めてみるとよいでしょう。

簡単なプログラムの例

では実際にGroovyFXプログラムの例を見てみましょう。

```
import groovyx.javafx.*

key = '※Google+のAPIキー'
uid = 110611905999186598367  // user ID
url = "https://www.googleapis.com/plus/v1/
people/$uid?key=$key".toURL()
json = new groovy.json.JsonSlurper().
parseText(url.text)

GroovyFX.start {
  def sg = new SceneGraphBuilder()

  stage = sg.stage(title: "Profile", width: 640, height:380) {
    scene(fill: black) {
      imageView(x: 20, y: 40, rotationAxis: [0, 1.0, 0]) {
        image(json.image.url.replaceAll(/sz=50/, 'sz=200'))
        effect reflection(fraction: 0.25)
        transition =
          rotateTransition(1.s, from:0, to:360, tween: ease_out)
        onMouseClicked { transition.play() }
      }
      text(x: 240, y: 60, text: json.displayName,
        fill: white, font: "32pt", textOrigin: "top") {
        effect bloom()
      }
      text(x: 240, y: 120, text: json.tagline,
        fill: white, font: "16pt", textOrigin: "top")
    }
  }
  stage.show()
  transition.play()
}
```

※下記からGoogle+のAPIキーを入手し、key変数に設定してください。
https://code.google.com/apis/console/

このプログラムは、Google+のAPIを使って特定のユーザのプロフィール情報を取得し、次のような画面を出力します：

実行結果

簡単なアニメーション要素として、起動時に一度、そしてクリックするたびにプロフィールイメージが回転するようにしてあります。

■コードの概要説明

最初の数行では、Google+のAPIにアクセスして、JSONのレスポンスを受け取っています。JavaFXにはJSONパーサが盛り込まれる予定もあったのですが、最終的には入りませんでした。Groovyには標準でJSONパーサが装備されており、このようなweb APIの利用では大いに役立ちます。

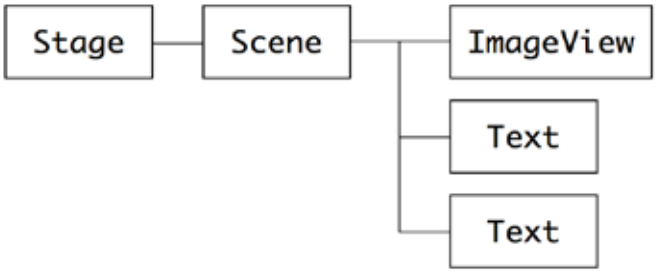
```
GroovyFX.start {
  def sg = new SceneGraphBuilder()
```

この部分はGroovyFXアプリの決まり文句です。

GroovyFXクラスはJavaFXのApplicationクラスのサブクラスで、startメソッドもApplicationクラスのものと同様の意味を持ちます。ただし、GroovyFXではクロージャを渡せるようになっており、このクロージャがJavaFXのアプリケーションスレッドで実行されます。シーングラフの構築などはこのクロージャの中で行われる必要があります。

次の行ではSceneGraphBuilderのインスタンスを生成しています。以下、このビルダーを使ってアプリケーションのシーングラフを構築しています。

このサンプルのシーングラフは下図のようになっています：



サンプルのシーングラフ構造

トップレベルのウィンドウに相当するStageの下に、各ノードを収容するコンテナであるScene、さらにその下に1つのImageViewと2つのTextノードがあります。各コンテナやノードの属性は、Groovyの他のビルダーと同様、マップのリテラル形式（key:value）で記述され、エフェクトやトランジション、イベントハンドラなども各ノードの子要素として記述されています。

上記の構造図と比較しながらコードを眺めると、シーングラフの構造と各ノードの属性や振る舞いが、簡潔かつ明快に表現されていることがよくわかると思います。

もう少し細かいところも見てみましょう。

```
rotateTransition(1.s, from:0, to:360, tween: ease_out)
```

この行には1.sという表現がありますが、ここは本来JavaFXではDurationを指定するところです。GroovyFXでは、Groovyのメタプログラミング機能を使ってNumberクラスにgetS、getMsなどのメソッドを追加して、NumberからDurationへの変換を実現しています。これにより、5.s、100.ms、3.hなどの表現が可能になっています。

また、上のコードでのease_outや、色を指定しているblack、whiteなどはいったい何者でしょうか？これらは一見定数のように見えますが、実はそうではありません。SceneGraphBuilderクラスの中に定義したpropertyMissingメソッドで未知のプロパティへのアクセスをフックし、対応する値に変換するというからくりで、簡潔な表現の疑似定数として実現されています。

さらに、JavaFXならFontやPoint3D、VPosといった型で指定すべき属性に対して、リストや文字列などが使われていることにも気づかれたかもしれません。これらはFXHelperというクラスで、渡された型に応じて、しかるべきクラスに自動変換する仕組みが実装されています。

以上のように、GroovyFXでは、ビルダーによるシーングラフの宣言的な構造記述に加え、Groovyの機能を駆使して記述を簡潔にするさまざまな仕組みによって、JavaFXの快適な利用を実現しているのです。

まとめ

今回はGroovyFXの概要とインストール、そしてごく簡単なコードを例にその特徴を見てきました。

GroovyFX(とJavaFX)には、変数のバインド機構やFXML、SwingやGriffonとの統合、HTML5コンテンツとの連携など、面白い話題がまだまだたくさんあります。次回以降はそういった話題を取り上げ、サンプルコードを使ってみなさんと一緒に「遊んで」いきたいと思います。どうぞお楽しみに！

■リンク

- JavaFX http://javafx.com/
- GroovyFX http://groovy.codehaus.org/GroovyFX
- 使用したサンプルコード https://gist.github.com/1341142

series 09

Betamax ことはじめ

須江 信洋（日本アイ・ビー・エム株式会社）
エンタープライズ Java を中心にミドルウェア製品のプリセールスを担当。日本 Grails/Groovy ユーザーグループ サポートスタッフ。『Groovy イン・アクション』（毎日コミュニケーションズ）翻訳チーム、および『プログラミング GROOVY』（技術評論社）執筆チームの一員。本稿は著者個人の考えおよび経験に基づいて記述したものであり、所属する会社や組織の意見を表すものではありません。

JGGUGでサポートスタッフを担当させていただいている@nobusueと申します。

今回はGroovyベースのテスト支援ツールであるBetamaxの紹介記事を書かせていただくことになりました。G*Magazineには初登場となりますが、よろしくお願いいたします。

Betamaxとは

BetamaxはWebへのアクセスを記録して再生することのできるrecord/playback proxyです。

最近のアプリケーションはTwitterやFacebookなど外部のWebAPIと連携するものが多くなってきていますが、Betamaxを利用すると実際にWebAPIやWebサイトへのアクセスを行わずにアプリケーションのテストを行うことができます。

BetamaxはHTTPリクエスト・レスポンスのペアを、HTTPリクエストの内容をキーとして「tape」というテキストファイル(YAML)に記録します。テストケースに対してtapeを指定してやることで、既に記録済みのHTTPリクエストに対してはtapeの内容が再生されます。未記録のHTTPリクエストであれば、実際に得られたレスポンスをキャプチャしてtapeに記録します。

Betamaxを使うことで、例えば以下のような効果が期待できます。

- WebAPIを利用するアプリケーションのテストをオフラインで行うことができる。
  新幹線で移動中でもOK。
- いちいちWebAPIにアクセスする必要がなくなるため、スローテスト問題を解決できる。
- WebAPI側の障害や仕様変更などからテストを分離できるため、テストを安定化することができる。
- 再現が難しいレアケースなどに対するテストを容易にできる。
  例えばTwitter APIの呼び出し回数制限に引っかかった場合など。
- tapeを資産として再利用したり、加工したりできる。
  Betamaxの発想の元となったvcr(Rubyベース)というプロダクトでは、既に多くの資産が蓄積されています。

Betamaxの開発者はRob Fletcher氏です。RobはGebのコミッタでもあり、BetamaxにはGeb/Spock/GrailsといったG*エコシステムとの連携があらかじめ織り込まれています。

Hello, Betamax

何はともあれ、まず実際にBetamaxを動かしてみましょう。

http://grails.org/にアクセスして、HTTPレスポンスのステータスコードが200であることを確認するテストケースをSpockで記述してみます。テストケースは以下のようになります。

▼src/test/groovy/MySpec.groovy

```
import betamax.Betamax
import betamax.Recorder
import spock.lang.*
import org.junit.*
import groovyx.net.http.RESTClient
import org.apache.http.impl.conn.
ProxySelectorRoutePlanner

class MySpec extends Specification {

  @Rule Recorder recorder = new Recorder()
                              // (1)Recorderの設定

  @Shared RESTClient http = new RESTClient()

  def setupSpec() {
    http.client.routePlanner =
        new ProxySelectorRoutePlanner(
                                  // (2)Proxyの設定
          http.client.connectionManager.schemeRegistry,
          ProxySelector.default)
  }

  @Betamax(tape = "my tape")   // (3)Tapeの設定
  def "simple http get response data"(){
    when:
      def response = http.get(uri:"http://grails.org/")
    then:
      response.status == 200
  }
}
```

(1)では、JUnitの@Ruleアノテーションを利用して、BetamaxのRecorderをMethodRuleとして定義しています。この設定により、テストケースでBetamaxの機能が利用できるようになります。

(2)では、RESTClientのproxy設定としてJVMのproxy設定が反映されるようにしています。（BetamaxはJVMのproxy設定を利用してリクエスト・レスポンスのキャプチャーを行います。）

(3)では、テストケースを実行する際にリクエスト・レスポ

ンスを記録するtapeを指定しています。ここで指定した名称がtapeのファイル名として利用されます。

このテストケースをビルドして実行するためのGradleビルドファイルは次のようになります。

▼build.gradle

```
apply plugin:"groovy"

repositories {
  mavenCentral()
}

dependencies {
  groovy "org.codehaus.groovy:groovy-all:1.8.4"
  testCompile "com.github.robfletcher:betamax:1.0"
  testCompile("org.codehaus.groovy.modules.
              http-builder:http-builder:0.5.1") {
    exclude module:"groovy"
    exclude module:"httpclient"
  }
  testCompile("org.spockframework:spock-core:0.5-groovy-1.8") {
    exclude mogule:"groovy-all"
  }
}
```

（これらのコードは https://github.com/nobusue/betamax-examples/tree/master/HelloBetamax から入手できます。こちらはGradleを導入していない環境でも実行できるようにGradle Wrapperを仕込んであります。）

それではテストケースを実行してみましょう。build.gradleの配置されているディレクトリから以下を実行します。

```
> gradle test
(Gradle Wrapperを使う場合) > gradlew test
```

「BUILD SUCCESSFUL」と表示されれば、テストは成功です。

このとき、src/test/resources/betamax/tapes を確認すると、my_tape.yaml というファイルが生成されているはずです。これがBetamaxが記録したHTTPリクエスト・レスポンスです。

▼my_tape.yaml(抜粋)

```
!tape
name: my tape
interactions:
- recorded: 2011-12-17T10:07:58.216Z
  request:
    method: GET
    uri: http://grails.org/
    headers:
      Accept: '*/*'
      Accept-Encoding: gzip,deflate
      Host: grails.org
      Proxy-Connection: Keep-Alive
  response:
    status: 200
    headers:
      Content-Encoding: gzip
      Content-Language: en-US
      Content-Type: text/html;charset=UTF-8
      Date: Sat, 17 Dec 2011 10:07:50 GMT
      Server: Apache/2.2.3 (Red Hat)
      Vary: Accept-Encoding
    body: "<!DOCTYPE html>\n<!--[if lt IE 7 ]>
                <html class=\"ie6\"> <![endif]-->
                \n<!--[if IE 7 ]>
. . .
```

HTTPリクエストの記録日時、URIとヘッダ、およびレスポンスのヘッダとボディが記録されていることがわかります。

それでは、tapeが記録済みの状態で再度テストを実行してみましょう。（初回実行時はGradleが依存ライブラリを取得するのに時間がかかるため、何回かテストを実行してみるとよいです。）記録済みのURIに対するリクエストをBetamaxが横取りしてtapeを再生するため、tapeなしでテストを実行した場合よりも早くテストが完了するはずです。私の環境では、tapeなしの場合はビルド完了まで10秒程度かかっていましたが、tapeありの場合は6秒程度まで短縮できました。

また、my_tape.yamlをリネームして保存しておき、再度tapeを記録して比較してみると、レスポンスの差分を確認することができます。grails.orgのコンテンツが変更されていなければ、recordedとDateヘッダの値のみが変わっていることが確認できるはずです。

Grails + Geb + Betamax サンプルコードの実行

もう少し実用的な使い方として、Grailsのテストケースで Betamaxを使う場合にどうすればよいかを見てみましょう。

Betamaxのソースを見てみると、https://github.com/robfletcher/betamax/tree/master/examples/grails-betamax 以下

にサンプルのGrailsアプリケーションがあります。これはTwitter API経由で「betamax」というキーワードを含むtweetを表示するアプリケーションですが、テストケースの中でBetamaxを利用して実際のAPI呼び出しをエミュレートするようになっています。

ただし、2012/12/18時点では、Twitter Webサイトの仕様変更によりオリジナルのBetamaxのサンプルではテストケースが通らなくなってしまっています。Twitterの仕様変更に対応した修正を施したバージョンを https://github.com/nobusue/betamax/tree/fix-grails-sample で公開していますので、「テストがfailするのは気持ち悪い」という方はこちらをご利用ください。参考までに、オリジナルからの修正点は以下のとおりです。

```
diff --git a/examples/grails-betamax/grails-app/views/twitter/tweets.gsp
 b/examples/grails-betamax/grails-app/views/twitter/tweets.gsp
index 7f84680..187706d 100644
--- a/examples/grails-betamax/grails-app/views/twitter/tweets.gsp
+++ b/examples/grails-betamax/grails-app/views/twitter/tweets.gsp
@@ -4,7 +4,7 @@
                <li>
                        <blockquote>
                                <p>${it.text}</p>
-                               <small><a href="http://twitter.com/${it.user}" rel="external">@${it.user}</a></small>
+                               <small><a href="http://m.twitter.com/${it.user}" rel="external">@${it.user}</a></small>
                        </blockquote>
                </li>
        </g:each>

diff --git a/examples/grails-betamax/test/functional/betamax/examples/TwitterPageSpec.groovy
 b/examples/grails-betamax/test/functional/betamax/examples/TwitterPageSpec.groovy
index 53e03d7..340a499 100644
--- a/examples/grails-betamax/test/functional/betamax/examples/TwitterPageSpec.groovy
+++ b/examples/grails-betamax/test/functional/betamax/examples/TwitterPageSpec.groovy
@@ -50,7 +51,7 @@ class TwitterPageSpec extends GebSpec {
                $('#tweets li').eq(0).find('small a').click()

                then:
-               title == "Christine Romero (@la_dyosa) on Twitter"
+               $('div.user-screen-name').text() == "la_dyosa (Christine Romero)"
        }

 }
```

サンプルアプリケーションの実行にはGrailsが必要ですので、別途導入しておいてください。(私の環境ではGrails-1.3.7で動作確認を行いました。)

実行方法は簡単で、ソースをcloneしてexamples/grails-betamax/に移動し、以下を実行するだけです。

```
> grails run-app
```

「Server running. Browse to http://localhost:8080/grails-betamax」が出力されたら、ブラウザでアクセスすると以下の画面が表示されます。

「twitter」リンクをクリックすると、Twitterから「betamax」を含むtweetを10件と、Twitterクライアントの分布を表示する画面に遷移します。

このアプリケーションは画面表示毎にTwitter APIを実行しているので、ためしにTwitterで「betamax最高！」などとつぶやいてから、画面を再表示してみてください。あなたのtweetが一覧に表示されるはずです。

アプリケーションの動作を確認したら、サーバーを停止しておいてください。

次に、テストを実行してみます。このサンプルではユニットテスト(test/unit/betamax/examples/)と機能テスト(test/functional/betamax/example)が提供されています。さきほどと同様に以下を実行してみてください。

```
> grails test-app
```

テストコードがビルドされ、ユニットテストおよび機能テストが実行されます。機能テスト実行時は自動的にアプリケーションが起動されるため、テスト完了まで少し時間がかかります。

```
Tests PASSED - view reports in target\test-reports
Application context shutting down...
Application context shutdown.
```

という表示が出たらテストは完了です。taget/test-reports/html/index.html を開いてテスト結果を確認してみてください。

テストが完了したところで、具体的にテストケースで何をしているのか確認してみましょう。

▼test/unit/betamax/examples/TwitterServiceSpec.groovy

```
class TwitterServiceSpec extends UnitSpec {

  File baseDir = BuildSettingsHolder.settings?.baseDir ?: new File("examples/grails-betamax")
  @Rule Recorder recorder = new Recorder(tapeRoot: new File(baseDir, "test/resources/tapes"))

  TwitterService service = new TwitterService()

  def setupSpec() {
    def log = Logger.getLogger("betamax")
      log.addHandler(new ConsoleHandler())
    }

  def setup() {
    def restClient = new RESTClient()
    restClient.client.routePlanner = new ProxySelectorRoutePlanner(
      restClient.client.connectionManager.schemeRegistry, ProxySelector.default)
    service.restClient = restClient
  }

  @Betamax(tape = "twitter success")
  def "returns aggregated twitter client stats when a successful response is received"() {
    when:
      def clients = service.tweetsByClient("betamax")

    then:   // (1)Twitterクライアントの分布のテスト
      clients.size() == 6
      clients["\u00DCberSocial for BlackBerry"] == 4
      clients["TweetDeck"] == 2
      clients["Echofon"] == 1
      clients["Mobile Web"] == 1
      clients["Snaptu"] == 1
      clients["Twitter for BlackBerry\u00AE"] == 1
  }

  @Betamax(tape = "twitter success")
  def "only retrieves tweets containing the search term"() {
    when:
      def tweets = service.tweets("betamax")

    then:   // (2)betamaxを含むtweetの検索結果件数と内容のテスト
      tweets.size() == 10
      tweets.every { it.text =~ /(?i)betamax/ }
  }

  @Betamax(tape = "twitter rate limit")
  def "sets an error status when twitter rate limit is exceeded"() {
    when:
      service.tweetsByClient("betamax")

    then:   // (3)異常系(例外発生)のテスト
      thrown TwitterException
  }
}
```

(1)と(2)はいずれもTwitter APIの実行結果に依存しているので、本来であればテスト用にモックやスタブを用意しなければなりません。しかし、ここではBetamaxのtapeを利用することで、Twitter APIの呼び出しをBetamaxのproxyで代替しています。

(3)についても同様で、Twitter APIの呼び出し回数制限に引っかかったケースをtapeとして記録しておき、それを再生することでテストを実行しています。このようなテストはWebAPIに負荷をかけるなどの問題があるため通常は再現が難しく、モックやスタブで置き換えるのも一筋縄ではいきません。Betamaxが効果を発揮する典型的なケースであるといえます。

Betamaxをソースからビルドする場合の注意点

■SonaTypeのアカウント設定

Betamaxのソースコードをcloneしてビルドしようとすると、「No such property: sonatypeUsername」という例外が発生してビルドができません。これは「sonatypeUsername」および「sonatypePassword」が設定されていないことが原因です。

これらの設定はSonaTypeに成果物(アーティファクト)をアップロードする際に必要となるもので、単にビルドして利用するだけなら利用されることはありません。回避策として、build.gradleと同じディレクトリに以下のファイルを作成してください。

▼gradle.properties

```
sonatypeUsername=
sonatypePassword=
```

■一部のテストが通らない

Betamaxの実装はエンコーディングやタイムゾーンの考慮が不十分な箇所があり、日本語環境で実行すると通らないテストがあります。単に使うだけであれば特に実害はないので無視していただいてかまいません。

我こそはと思われる方は、ぜひバグ取りにご協力ください。

まとめ

今回は、Groovyで実装されたrecord/playback proxyであるBetamaxについての簡単な紹介と、Grailsのテストケースでの活用について説明しました。次回はBetamaxのコンフィグレーションや、内部動作についてご紹介したいと思います。

Betamaxの最新安定バージョンは1.0ですが、まだまだ発展途上のプロダクトであり2012年12月現在も活発な開発が続いています。Betamaxについて興味をもたれた方は、ぜひ以下を参照してみてください。バグフィックスや機能拡張への貢献も歓迎です。

- Betamax Webサイト
  http://robfletcher.github.com/betamax/
- Betamax ソースコードリポジトリ
  https://github.com/robfletcher/betamax

Grails Plugin 探訪

第 5 回

~ RabbitMQ プラグイン ~

URL: http://www.grails.org/plugin/rabbitmq
プラグインのバージョン: 0.3.2
対応するGrailsのバージョン: 1.2 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、RabbitMQ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 10.6.8
- Java : 1.6.0_29
- Grails : 1.3.7

RabbitMQとは

RabbitMQとは、Advanced Message Queuing Protocol(AMQP)を使用したオープンソースのメッセージングシステムです。2010年にSpring Source社が買収し、現在は同社がRabbitMQの開発・サポートを行っています。

詳しくは、RabbitMQを参照してください。

RabbitMQプラグインとは

RabbitMQプラグインとは、RabbitMQとの統合・連携をするためのプラグインです。RabbitMQプラグインは、メッセージ送受信を高レベルに抽象化するため、Spring AMQPを使用しています。

RabbitMQプラグインのインストール

アプリケーションを作成し、RabbitMQプラグインをインストールします。

```
$ grails create-app rabbit
$ cd rabbit
$ grails install-plugin rabbitmq
```

RabbitMQサーバの設定

RabbitMQを利用するための設定は、grails-app/conf/Config.groovyに記述します。

```
rabbitmq {
    connectionfactory {
        username = 'guest'
        password = 'guest'
        hostname = 'localhost'
    }
}
```

設定できるプロパティは、次のとおりです。

プロパティ名	説明	デフォルト値
rabbitmq.connectionfactory.username	RabbitMQサーバに接続するためのユーザ名	なし
rabbitmq.connectionfactory.password	RabbitMQサーバに接続するためのアカウント用のパスワード	なし
rabbitmq.connectionfactory.hostname	RabbitMQサーバのホスト名	なし
rabbitmq.connectionfactory.virtualHost	RabbitMQサーバに接続するための仮想ホスト名	'/'
rabbitmq.connectionfactory.channelCacheSize	接続チャネルのキャッシュサイズ	10
rabbitmq.concurrentConsumers	メッセージハンドラごとに作成する同時コンシューマの数	1

キューの設定

キューの設定は、次の2つの方法があります。

- Spring Beanで定義する
- DSLで定義する

■Spring Beanで定義する

org.springframework.amqp.core.Queue型のSpring Beanの定義を、grails-app/conf/spring/resources.groovyにします。

例えば、次のようになります。

```
beans = {
    myQueue(org.springframework.amqp.core.Queue, 'myQueueName')
    myOtherQueue(org.springframework.amqp.core.Queue,
                        'myOtherQueueName') {
        autoDelete = false
        durable = true
        exclusive = false
        arguments = [arg1: 'val1', arg2: 'val2']
    }
}
```

■DSLで定義する

先程のSpring Beanの定義を、DSLでgrails-app/conf/Config.groovyに記述することができます。

```
rabbitmq {
    connectionfactory {
        username = 'guest'
        password = 'guest'
        hostname = 'localhost'
    }
    queues = {
        myQueueName()
        myOtherQueueName autoDelete: false,
        durable: true, exclusive: false,
        arguments: [arg1: 'val1', arg2: 'val2']
    }
}
```

■設定できる項目

両方の定義方法で設定できる項目は、次のとおりです。

項目	説明	デフォルト値
autoDelete	クライアントの接続がなくなった場合にブローカーからキューを削除するかどうか	false
durable	ブローカの再起動後もキューが存在するかどうか	false
exclusive	キューを作成したクライアントのみがキューに接続できるようにするかどうか	false
arguments	キュー作成時の引数	null

エクスチェンジの設定

エクスチェンジは、メッセージを受け取る役割を担います。エクスチェンジが受け取ったメッセージはバインドに従って、適切なキューに送られます。

エクスチェンジの設定は、grails-app/conf/Config.groovyで行います。

```
rabbitmq {
    connectionFactory {
        ...
    }
    queues = {
        exchange name: 'my.topic', type: topic
    }
}
```

exchangeに設定できる項目は、次のとおりです。

項目	説明	デフォルト値
name	エクスチェンジの名前	なし
type	エクスチェンジの種類。direct fanout, topic, headersから指定	なし
autoDelete	クライアントの接続がなくなった場合にブローカーからエクスチェンジを削除するかどうか	false
durable	ブローカの再起動後もエクスチェンジが存在するかどうか	false

キューとエクスチェンジのバインド

エクスチェンジとキューのバインドも、設定で指定できます。

バインドの設定は、grails-app/conf/Config.groovyで行います。

```
rabbitmq {
    connectionFactory {
        ...
    }
    queues = {
        exchange name: 'my.topic', type: topic,
        durable: false, {
                foo durable: true,
                binding: 'shares.#'
                bar durable: false,
                autoDelete: true,
                binding: 'shares.nyse.*'
        }
    }
}
```

上記の例では、'my.topic'という名前のエクスチェンジを1つを定義し、'foo', 'bar'という名前のキューにバインドします。

どのメッセージをキューにバインドするかは、bindingで指定します。

ルーティングキーが'shares.'で始まるメッセージは、fooキューにバインドされます。

ルーティングキーが'shares.nyse.'で始まり1語続くメッセージは、barキューにバインドされます。

メッセージの送信

メッセージの送信は、rabbitSendメソッドで行います。rabbitSendメソッドは、コントローラ、サービス、タグリブといったGrailsアーティファクトにRabbitMQプラグインによって追加されます。

```
class MessageController {

    def sendMessage = {
        rabbitSend 'someQueueName', 'someMessage'
        ...
    }
}
```

rabbitSendメソッドは、引数を2つとります。1つ目はキューの名前で、2つ目は送信するメッセージそのものです。

メッセージの送信は、org.springframework.amqp.rabbit.core.RabbitTemplateインスタンスを使う方法もあります。この方法については、後ほど説明します。

メッセージの受信

メッセージの受信は、次の2通りの方法があります：

- エクスチェンジの購読による受信(発行/購読モデル)
- キューからの受信

エクスチェンジの購読による受信

エクスチェンジの購読による受信では、まずエクスチェンジの定義を行います。

Grailsアプリケーションが発行者となる場合、grails-app/conf/Config.groovyでエクスチェンジの定義をします。

次に、メッセージを購読する処理のため、rabbitSubscribeという名前の静的プロパティと、handleMessage()という名前のメッセージを処理するメソッドを持つサービスを作成します。

```
package rabbit

class SharesService {
    static rabbitSubscribe = 'shares'

    void handleMessage(message) {
        // handle message
        ...
    }
}
```

rabbitSubscribeプロパティに指定するのはエクスチェンジの名前で、指定されたエクスチェンジ宛てに送られたメッセージを購読することになります。

エクスチェンジが受け取ったメッセージは、handlerMessage()メソッドの引数として指定され、購読対象のサービスのhandlerMessage()メソッドが呼び出されます。

topicタイプのエクスチェンジの場合、ルーティングキーを指定することで、受信するメッセージをコントロールできます。

rabbitSubscribeプロパティをハッシュ形式で指定し、nameでエクスチェンジの名前、routingKeyでルーティングキー(のパターン)を指定します。

```
package rabbit

class SharesService {
    static rabbitSubscribe = [ name: 'shares',
                               routingKey: 'NYSE.GE' ]
    ...
}
```

キューからの受信

キューからの受信では、まずキューの定義を行います。

次に、メッセージを受信する処理のため、rabbitQueueという名前の静的プロパティと、handleMessage()という名前のメッセージを処理するメソッドを持つサービスを作成します。

```
package rabbit

class DemoService {
    static rabbitQueue = 'someQueueName'

    void handleMessage(message) {
        // handle message
        ...
    }
}
```

rabbitQueueプロパティに指定するのはキューの名前です。

メッセージはhandlerMessage()メソッドの引数として指定され、受信対象のサービスのhandlerMessage()メソッドが呼び出されます。

メッセージ

RabbitMQプラグインでは、メッセージとして、次の型のオブジェクトを使用することができます：

* 文字列
* バイト配列
* マップ
* シリアライザブル

handleMessage()メソッドをオーバーロードすることで、メッセージの型ごとに呼び出すメソッドを変えることができます：

```
package rabbit

class DemoService {
    static rabbitQueue = 'someQueueName'

    void handleMessage(String textMessage) {
        // handle String message
        ...
    }

    void handleMessage(Map mapMessage) {
        // handle Map message
        ...
    }

    void handleMessage(byte[] byteMessage) {
        // handle byte array message
        ...
    }
}
```

RabbitTemplateクラスを使用する。

自分で細かい制御をしたい場合、org.springframework.amqp.rabbit.core.RabbitTemplateクラスのインスタンスをBeanとして、コントローラやサービスにインジェクションし使用することができます。

Beanの名前は、rabbitTemplateです。

```
class MessageController {

    def rabbitTemplate

    def sendMessage = {
        rabbitTemplate.convertAndSend('someQueueName',
                                      'someMessage')
        ...
    }
}
```

RabbitTemplateクラスの詳細は、http://static.springsource.org/spring-amqp/docs/latest-ga/を参照してください。

終わりに

アプリケーションで非同期処理を行う際、メッセージシステムを使うことを、そしてRabbitMQを使うことを検討してみてはいかがでしょうか。

▼ぐるーびーたん　第3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。いろいろ覚えて楽しそうなんだけど。。。君の名は確か!?

※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 2011.12.15

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: 1.2.5, 1.3.7, 2.0.0

■更新情報

- 2.0.0では、ページスコープで<g:include>が<g:set>を壊してしまうバグや、"formula"派生プロパティが不正になるバグなど、いくつかバグの改修や、機能の改善が行われています。
- 2.0.0リリースノート: http://grails.org/2.0.0+Release+Notes
- 山本さんのブログ: http://d.hatena.ne.jp/mottsnite/20111215/1323965990

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: 1.7.10, 1.8.4, 2.0.0-beta-1

■更新情報

- 1.8.4では、匿名の内部クラスでプロパティにアクセスした際にNullPointerExceptionが発生するバグや、DefaultTypeTransformation.booleanUnboxのパフォーマンスの改善など、いくつかバグの改修や、機能の改善が行われています。
- 1.8.4リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=17852
- 2.0.0-beta-1では、Grabの簡略表記が正しく動作しないバグや、@Immutableがjava.util.UUIDをサポートする改良など、いくつかバグの改修や、機能の改善が行われています。
- 2.0.0-beta-1リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=17925

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: 0.9.4

■更新情報

- 0.9.4では、ビルド時にソースのエンコーディングを指定できるようになったり、griffon.core.ServiceManager型のインスタンスを介して全てのサービスのインスタンスにアクセスできるようになるなど、いくつかバグの改修や、機能の改善が行われています。
- 0.9.4リリースノート: http://griffon.codehaus.org/Griffon+0.9.4

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: 1.9.7

■更新情報

- 1.9.7では、ANT_HOMEに空白文字が含まれている場合gant.batが失敗するバグに対応しています。
- 1.9.7リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=11660&version=17496

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: 0.9.2, 1.0-milestone-6

■更新情報

- 1.0-milestone-6では、依存解決が以前のリリースより速くなったり、デーモンが改良されたりするなど、いくつかバグの改修や、機能の改善が行われています。
- 1.0-milestone-6リリースノート: http://wiki.gradle.org/display/GRADLE/Gradle+1.0-milestone-6+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: 1.1

■更新情報

- 1.1では、Groovyが1.8.4、App Engine SDKが1.6.0にアップデートされたり、データストアサービスにget()メソッドが追加されるなど、いくつかバグの改修や、機能の改善が行われています。
- 1.1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: 1.6.1

■更新情報

- 1.6.1では、プログラムでアプリケーションのログを読み取ることができるようになったり、ハイレプリケーションデータストアの移行ユーティリティが一般利用可能な機能となるなど、いくつかバグの改修や、機能の改善が行われています。
- 1.6.1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: 0.12GA

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

■更新情報

- 0.9.0では、Groovy 1.8.2対応されています。
- 0.9.0リリースメール: http://groups.google.com/group/groovyplusplus/browse_thread/thread/237434c427c57f11?pli=1

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: 0.5

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: 0.9

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: 0.6.0

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: 0.8.1

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: 0.5.1

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: 0.16.1

■更新情報

- 0.16.1では、Groovy 1.8でCodeNarcを実行した際にStackOverflowErrorが発生するバグに対応しています。
- 0.16.1リリースノート: http://sourceforge.net/projects/codenarc/files/codenarc/CodeNarc%20Version%200.16.1/

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: 0.4

■更新情報

- 0.4では、Groovyが1.7にアップグレードされたり、Groovy 1.7.7以上でGMetricsができないバグなど、いくつかバグの改修や、機能の改善が行われています。
- 0.4リリースノート: http://sourceforge.net/projects/gmetrics/files/gmetrics-0.4/

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: 1.2.4

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://groovy.codehaus.org/GroovyFX
バージョン: 1.0 Alpha

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: 0.2.2

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: https://github.com/robfletcher/betamax
バージョン: 1.0

■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，2011年度より個人サポータ制度を始めました．

今までJGGUGの経済的な基盤はすべて法人会員の年会費に依存していました。 しかし、個人会員も金銭面からサポートしたいという声があがり、JGGUG運営基盤の裾野を拡げるためにも、個人サポータという形で個人会員からの寄付を受け入れることにした次第です。

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.4 2011.12
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