Railsを使い倒そう

ネットワーク応用通信研究所 小田勝也

自己紹介

- Railsを使ったWeb開発は6年目です。
- 最近はscalaでAndroidアプリを書いています。

Railsの現状

- 成熟度
  - 最初のリリースからもう7年
  - 長期間に渡って人気があるフレームワーク
- 人気度
  - 依然として高い人気
  - stackoverflowでタグの数で2番目
  - GitHubでstarの数で1番目

RubyGemsの現状

- 豊富なライブラリ
  - Webアプリで必要なライブラリはほぼある
  - RubyToolBox
    - わかりやすいカテゴリづけと評価づけ
    - 各指標をもとに使用するライブラリを判断
- Javaのライブラリも使用可能
  - JRubyでJavaの資産も使える。

RubyでのWeb開発の現状

道具は全て揃っている。後はそれを組み合わせるだけの簡単なお仕事です。

だけど…

実際は作り込みが必要だよね。

今日のお話

実際の受託開発の現場でRailsの各機能を活用してどのように作り込みを行っているかを紹介します。

今日のお話2

1. 背景
2. 開発事例
   2.1. Rails::Generatorによるコード生成
   2.2. Rails::Engineを使ったアプリケーション開発
   2.3. Rackによるエラーメッセージのカスタマイズ
3. 質問

受託開発とは

顧客が利用・販売する製品の開発を請け負うこと。特に、顧客企業が業務の遂行に利用する情報システムや業務用ソフトウェアなどの開発を受注すること。

-- e-Words

受託開発でよくあること

お客様のワークフローに沿ったシステムを開発するため、もともとのRailsの枠に沿っていないこともしばしば。

事例

- 常に確認画面が欲しい。
  - お客様によって画面遷移が特殊な場合がある。
  - カスタムscaffoldを実装して対応。
- パッケージ毎にアプリの構成を変えたい。
  - DMZとセキュアゾーンで構成が変わる。
  - 各機能を別々のEngineに実装して対応

ここから本題

Rails::Generatorによるコード生成

大量の管理機能を短い開発期間で実装したい。

- 実装する管理機能は10数機能、画面遷移・デザインなどは共通。
- Rails::Generatorを使用して対応。

管理機能を大量に実装する方法

- メタプログラミング(動的)
  - RailsAdminがこのアプローチ。
  - 似て非なる管理機能があると大変。
  - 簡単な初期設定だけで動作する。
- Rails::Generators(静的)
  - 今回のアプローチ。
  - 似て非なる管理機能がある場合は、生成したコードを修正するだけ。
  - 生成した後に画面毎の作り込みが必要。

Rails::Generatorsとは

コードを生成するRailsの機能。コマンドラインから利用する。コントローラ、モデル、テストなどの雛形を作成するのに使用されている。

MyGeneratorの生成

```
~/rails_app> bundle exec rails g generator my
      create  lib/generators/my
      create  lib/generators/my/my_generator.rb
      create  lib/generators/my/USAGE
      create  lib/generators/my/templates
```

MyGeneratorクラスの定義

```
class MyGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)


  def create_spec_file
    template 'spec_template.rb', "spec/%name%.rb"
  end
end
```

ERBテンプレートの作成

```
require 'spec_helper'


describe <%= name.camelize %> do
  it "#sample_it" do
    # do something...
  end
end
```

MyGeneratorの実行

```
~/rails_app> bundle exec rails g my my_model
       exist  spec
      create  spec/my_model.rb
```

spec/my_model.rb

```
require 'spec_helper'


describe MyModel do
  it "#sample_it" do
    # do something...
  end
end
```

Rails::Generatorsとは

- Generator(ファイルを操作するクラス)
  - ファイルの雛形からファイルを作成し、どのパスに配置するかを決める。
  - 必要な場合はディレクトリなどを作成する。
- ERBテンプレート(ファイルの雛形)
  - ファイルの雛形はERBテンプレートに記述する。
  - ERBテンプレートには動的に取得する情報を埋め込むことができる。

Rails::Generatorsとは2

コードを生成するだけでなく、コード生成に役立つ便利な機能を適用してくれる。

Rails::Generatorsとは3

- 上書き防止機能
  - 再生成したときに前回との差分を表示してくれる。
- 取り消し機能
  - 生成したコードを削除してくれる。
- ファイルの変更機能
  - ファイルの中身を変更する便利な機能がある。

上書き防止機能

```
~/rails_app> bundle exec rails g my my_model
    conflict  spec/my_model.rb
Overwrite rails_app/spec/my_model.rb? (enter
"h" for help) [Ynaqdh] d
```

```
---
/home/katsuya/rails_app/spec/my_model.rb 2013-12-15...
+++
/home/katsuya/rails_app/spec/my_model.rb20131215...
@@ -1,7 +1,7 @@
 require 'spec_helper'

 describe MyModel do
-  it "#do_something" do
+  it "#do_something2" do
     # do something...
   end
```

```
Overwrite
/home/katsuya/work/rails_seminar_2013_12_17/rails_app/spec
/my_model.rb? (enter "h" for help) [Ynaqdh] h
Y - yes, overwrite
n - no, do not overwrite
a - all, overwrite this and all others
q - quit, abort
d - diff, show the differences between the old and the new
h - help, show this help
```

上書き防止機能

前回生成したファイルに変更があったら、上書き防止機能が働く。例えば、ファイルの場合はファイルの内容が異なっていたら、リンクの場合はFile.identical?がfalseを返したら。

取り消し機能

```
~/rails_app> bundle exec rails d my my_model
      remove   spec
      remove   spec/my_model.rb
```

取り消し機能2

- 取り消し防止機能はない。
  - empty_directoryメソッドは問答無用にディレクトリを消すので注意。
- 取り消し機能に対応しているメソッド
  - create_file、empty_directory、create_link、directory、insert_into_fileの5つのメソッド。
  - 他のメソッドはだいたい中で上の5つのメソッドのいずれかを使用している。

ファイルの変更機能

```
class MyGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)


  def add_route
    insert_into_file 'config/routes.rb',
      "\n  get '/welcome' => 'welcome#index'\n\n",
      {after: /\.routes\.draw do(?:\s*\|map\|)?\s*$/}
  end
end
```

ファイルの変更機能2

```
katsuya@kyle ~/w/r/rails_app> bundle exec
rails g my my_model
      insert  config/routes.rb
```

ファイルの変更機能3

```
class MyGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)


  def add_route
    route "get '/welcome' => 'welcome#index'\n"
  end
end
```

ファイルの変更機能

insert_into_fileは上書き防止や取り消しに対応しているので意外と便利。

実例

- 一つの管理機能を実装し、その実装した管理機能をもとにカスタムscaffoldを作成している。
- 大量のテストを生成している。特に認証・認可に関連するテストが自動生成されて助かっている。
- routes.rb、i18n用のyamlファイル、configファイルに動的に設定を追加している。

実例

- 管理画面毎に異なる機能は、別ファイルに分ける。メソッドで分けてもいいよ。
  - RubyのオープンクラスやRailsの部分テンプレートといった機能を利用する。

知っておいた方がいいこと

- Generatorや雛形に不具合がある場合に、再生成しなければならない。生成したファイルに変更を加えている場合はそれなりに修正が大変。
- 再生成を怖がらない。git diffに任せる。
- Rails::GeneratorsはThorの上に構築されている。lib/thor/actions/**.rbをよく読むこと。

Thorについて

コマンドの作成を助けてくれるライブラリ。コマンド引数のparse、コマンドのグループ化、コマンドのドキュメント生成などを行ってくれる。

標準添付のOptionParserの置き換え。

コマンドについて

受託開発においてコマンドの作り込みは以外と重要。だけど、意外とメンテナンスは大変。

わかりやすいエラーメッセージ・統一されたオプションなどが必要。

コマンドの作成

- コマンドの配置先
  - Thorfileもしくはlib/tasks/*.thor。Rakeタスクと似ている。
- コマンド用のクラス
  - Thorを継承したクラスを作成。そのクラスに定義したpublicメソッド一つ一つがコマンドになる。

```ruby
class Ci < Thor
  desc "create JOB", "Create a job"
  method_option :branch,
    :aliases => "-b", :desc => "Specify branch name"
  method_option :ruby_version, :default => '1.9.3-p327',
    :aliases => "-r", :desc => "Specify ruby version"
  def create(job)
    options[:ruby_version]

    # do something ...
  end
end
```

コマンドの実行

```
~/rails_app> bundle exec thor ci
Commands:
  thor ci:create JOB       # Create job
  thor ci:help [COMMAND]   # Describe
available commands or one specific command
```

```
~/rails_app> bundle exec thor help ci:create
Usage:
  thor ci:create JOB

Options:
  -b, [--branch=BRANCH]                  # Specify branch name
  -r, [--ruby-version=RUBY_VERSION]      # Specify ruby
version
                                         # Default: 1.9.3-p327

Create job
```

OptionParserのいやなとこ

- 統一されたエラーメッセージを表示するのがそれなりに大変。
  - 結構なコード量になる。しかも、OptionParserを読み込まないとできない。読み込むのは大変。
- 細かいテストをするのが難しい。
  - thorは引数やオプション、一つ一つのテストを簡単に行える。
    - 影響が大きいコマンドだけをテストする等。

Thorのいいとこ

- 前のスライドで列挙したOptionParserのいやなところを解消してくれる。
- 意味ごとにコマンドをグループ化できる。
  - 大量のコマンドをメンテナンスしやすい。
- 定義したオプションの情報からドキュメントを出力してくれる。
  - OptionParserより高機能。

実例

- redmineを操作するコマンド群
  - CSVからチケットを作成
  - 複数のチケットの担当者を変えたりなど。
- Jenkinsを操作するコマンド群
  - XMLから特定のブランチ用のジョブの作成。
  - 複数のジョブをつなぎ合わせたりなど。
- リリースを行うコマンド群
  - リリース時に検証環境にリリースアプリを展開

次のお話

Rails::Engineを使ったアプリ開発

- 顧客毎にアプリケーションの構成を変えたい。
  - 顧客毎に機能を制限したい。
  - 特定の顧客には機能を拡張して導入したい。
  - Rails::Engineを使用して対応。

顧客毎にカスタマイズする方法

- バージョン管理で対応(静的)
  - 顧客毎にブランチを分ける。
  - アップグレードやバグのバックポートが大変。
- Rails::Engineで対応(動的)
  - 顧客毎のカスタマイズは各Engineに実装し、リリース時に構成を変更できるようにする。
  - アップグレードとバグのバックポートが不要なため管理がしやすい。

Rails::Engineとは

Railsアプリケーションそのもの。Railsアプリケーションも Rails::Engineからできている。

Rails::Engineを使用すると、Railsアプリケーションの中に、他のRailsアプリケーションに埋め込むことができる。

My::Engineの生成

```
~/rails_app> bundle exec rails plugin new vendor/my --
mountable
      create
      create  Rakefile
      create  my.gemspec
      create  Gemfile
      create  app
      create  config/routes.rb
      create  lib/my/engine.rb
```

lib/my/engine.rb

```
module My
  class Engine < ::Rails::Engine
    isolate_namespace My
  end
end
```

app/controlers/my/application_controoler.rb

```ruby
module My
  class ApplicationController < ActionController::Base
    def index
      render text: 'Hello Engine!'
    end
  end
end
```

config/routes.rb

```
My::Engine.routes.draw do

  get '/engine' => 'application#index'

end
```

```ruby
module My
  class Engine < ::Rails::Engine
    isolate_namespace My

    initializer 'my.update_app_routes',
      :after => :add_routing_paths do |app|
      app.routes.prepend do
        mount My::Engine => '/my', :as => :my
      end
    end
  end
end
```

Gemfile

```
source 'https://rubygems.org'


# Bundle edge Rails instead: gem 'rails', github:
'rails/rails'
gem 'rails', '4.0.0'


gem 'my', path: 'vendor/my'
```

Hello Engine!

Rails::Engineとは

- Railsアプリケーションの中にRailsアプリケーションを埋め込むことができる。
- 特定の顧客用の機能を分離して実装できる。
- 利用しやすく外しやすい、使いまわしができる。

Rails::Engineとは2

Railsを拡張するための機能。昔のプラグインと異なり、仕様が明確化されているためとても便利。

routes、generator、Rackアプリ、Rakeタスクの追加等を行える。

Rackアプリの追加

- Railsアプリケーションのスタックに Rackアプリを追加することができる。
- Engineのスタックに Rackアプリも追加することができる。

lib/my/middleware.rb

```
module My
  class Middleware
    def initialize(app)
      @app = app
    end

    def call(env)
      return @app.call(env)
    end
  end
```

lib/my/engine.rb

```
require 'my/middleware'
module My
  class Engine < ::Rails::Engine
    isolate_namespace My

    initializer 'my.middleware' do |app|
      app.middleware.use My::Middleware
    end
  end
end
```

Rackアプリの追加4

```
~/rails_app> bundle exec rake middleware
use Rack::ETag
use My::Middleware
run RailsApp::Application.routes
```

lib/my/engine.rb

```
require 'my/middleware'


module My
  class Engine < ::Rails::Engine
    isolate_namespace My


    middleware.use My::Middleware
  end
end
```

Rakeタスクの追加

- RailsアプリケーションにRakeタスクを追加することができる。
- もともとのRakeタスクを拡張することができる。

lib/tasks/my_task.rake

```
namespace :my do
  desc "Explaining what the task does"
  task :task do
    # Task goes here
  end
end


Rake::Task['db:seed'].enhance do
  My::Engine.load_seed
end
```

Rakeタスクの追加3

```
~/rails_app> bundle exec rake -T
rake db:create                # Create the database from
DATABASE_URL or conf...
rake middleware               # Prints out your Rack
middleware stack
rake my:task                  # Explaining what the task
does
```

実例1

複数のEngineを跨いで使えるconfig機能の作成

- 各Engineの初期化時もconfigを使いたい。
- 開発しやすいように、requestを受け取る毎にreloadしたい。

```ruby
module My
  class Engine < ::Rails::Engine
    config.before_initialize do
      My.load_configuration(
        My::Engine.config.root + 'config/my.rb')
    end

    initializer 'demeter_mp.load_admin_config' do |app|
      if My.config.reload_config_per_request
        config.to_prepare do
          My.reload_configuration!
        end
```

実例2

既存機能のデザインを特定の顧客だけ変えたい。
コントローラのview_pathsを変更することでカスタマイズ可能。

コントローラはビューのテンプレートを探す際に、view_pathsに設定されているディレクトリを頭から舐めて探していく。

app/controlers/my/application_contr ooler.rb

```ruby
module My
  class ApplicationController < ActionController::Base
    def index
      render :index
    end
  end
end
```

templates/my/application/index.html.erb

```
<html>
  <body>
    <p><%= 'Hello Custom Template!' %></p>
  </body>
</html>
```

```ruby
module My
  class Engine < ::Rails::Engine
    initializer 'my.add_view_path' do
      ActiveSupport.on_load(:my) do
        My::ApplicationController.prepend_view_path \
          Engine.config.root + 'templates'
      end
    end
  end
  ActiveSupport.run_load_hooks(:my, Engine)
end
```

さらにカスタマイズしたい

Rails::Engineが提供する機能で満足できない場合は、メタプログラミングで解決する。

既存のコントローラやモデルをカスタマイズしたい場合は、Rubyのオープンクラスの性質、Module#prepend、Module#includeなどを使う。

知っておいた方がいいこと

- Rails::Engine、Rails::ApplicationはRails::Railtieの上に構築されている。
- ActiveRecord、ActionController、ActionViewもRails::Railtieの上に構築されている。
- lib/rails/application.rbの「Bootint Process」をよく読む。

次のお話

Rackによるエラーのカスタマイズ

エラー画面をカスタマイズしたい。

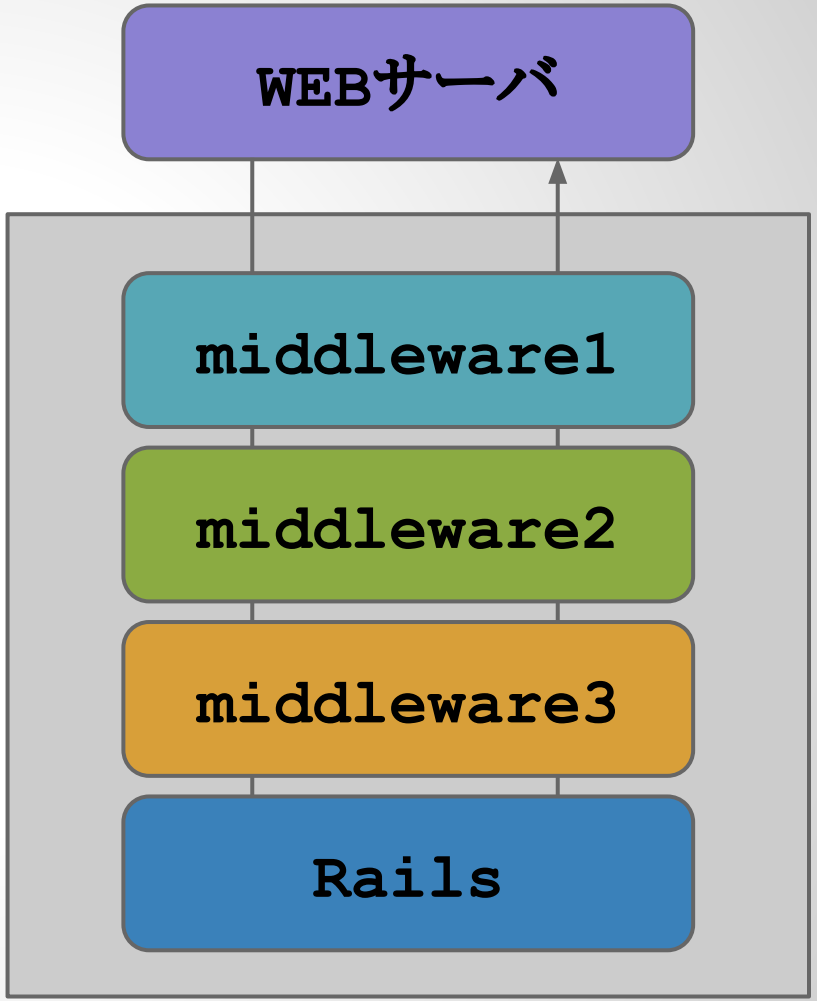
- データベースエラーやルーティングエラーをユーザに見せたくない。
- エラーの種類毎にエラー画面を変えたい。
- Rackアプリをカスタマイズすることで対応

Railsにおけるエラー

- エラーの種類を問わず、Rubyにおけるエラーはすべて例外という形で表現される。
- 起こりうるエラーの種類とエラーの発生場所を把握する必要がある。
- Railsアプリの構成の理解が必須。

Railsアプリの構成

- 複数のRackアプリ(middleware)が連なって構成されている。
- Railsアプリケーションも Rackアプリの一つ
- 全体もRackアプリ



Rackアプリとは

リクエストを受け取って、レスポンスを返すRubyのオブジェクト。

リクエストの受け取り方と、レスポンスの返し方が仕様化されている。

```ruby
class Middleware1
  def initialize(app)
    @app = app # middleware2の設定
  end

  def call(env)
    # do something...
    status, headers, body = @app.call(env)
    # do something...
    return [status, headers, body]
  end
end
```

Rackアプリとは

機能を分離し、複数のWebサーバ、複数のフレームワークで動作させることができる。

- 機能毎に利用しやすくはずしやすい、使いまわしができる。
- 機能毎に切り離してテストがしやすい。
- 複数のWebサーバで利用できる。unicornでもPassengerでも。
- 複数のフレームワークで利用できる。RailsでもSinatraでも。

Railsのmiddlewareのスタック

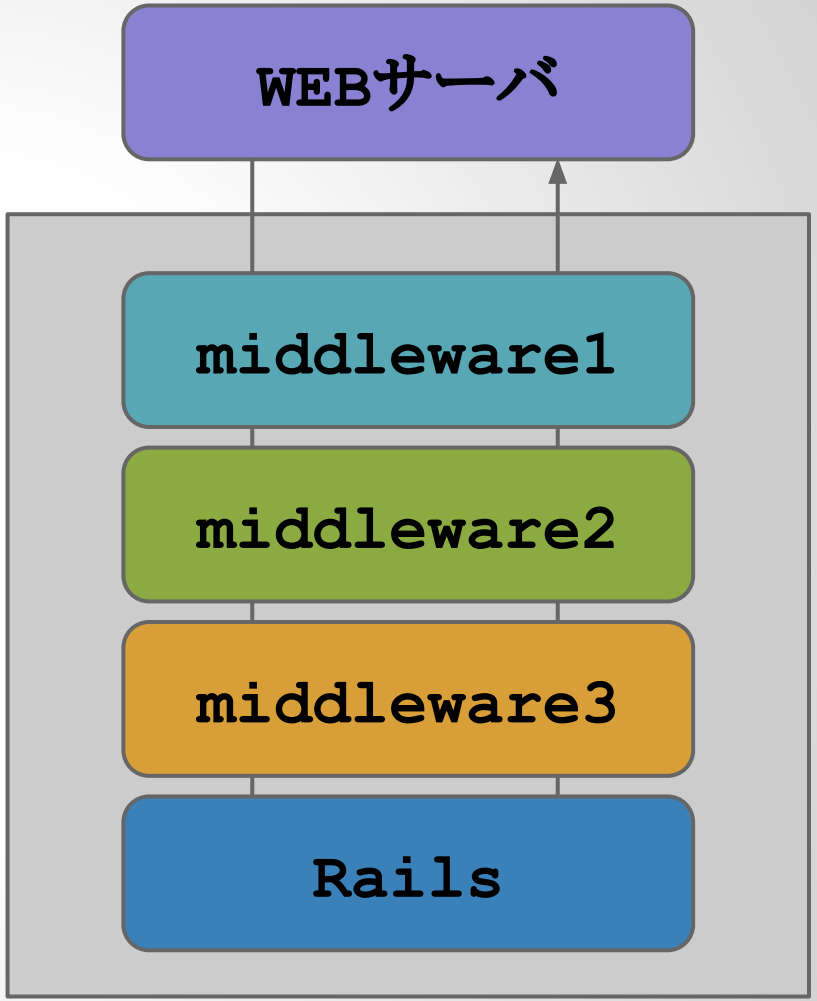
```
~/rails_app> bundle exec rake middleware
use ActionDispatch::Static
use ActionDispatch::ShowExceptions
use ActiveRecord::QueryCache
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use ActionDispatch::ParamsParser
run RailsApp::Application.routes
```

よくエラーが発生するRackアプリ

- データベースと接続できない場合
  - ActiveRecord::QueryCacheで例外
- invalidなjsonやxmlが送信された場合
  - ActionDispatch::ParamsParserで例外

エラーを掴むには

- スタックの最初の方に詰まれているmiddlwareで対応する。
- RailsはデフォルトでActionDispatch::ShowExceptionsでエラーを処理している。



Rails標準のエラー対応機能

- middleware(ActionDispatch::ShowExceptions)として実装されている。
- エラー情報を設定して、指定したmiddleware(デフォルトはPublicExceptions)を呼び出す。指定したmiddlewareで例外が発生した場合は決め打ちのエラーを上位のmiddlewareに返す。

Rails標準のエラー対応機能2

- middleware(ActionDispatch::PublicExceptions)として実装されている。
- public以下に置かれている500.htmlをブラウザに返す。

エラー対応機能の置き換え

```
module Foo
  class Application < Rails::Application
    require 'my/public_exceptions'
    config.exceptions_app =
      My::PublicExceptions.new(Rails.public_path)
  end
end
```

```ruby
class My::PublicExceptions < ActionDispatch::PublicExce...
  def call(env)
    status = env["PATH_INFO"][1..-1]
    path = "#{public_path}/#{status}.html.erb"
    if File.exist?(path)
      eruby = Erubis::Eruby.new(File.read(path))
      render(status, eruby.result(binding()))
    else
      [404, { "X-Cascade" => "pass" }, []]
    end
  end
end
```

事例

- 認証機能をRackアプリで実装。複数のEngine毎に認証機能を切り替えている。
- OAuth2のAuthorization Serverを実装する際に利用。gemライブラリで実装されているRackアプリをaction内で呼ぶだけで動作する。

質問