Practical Meta Programming on Rails Application

Railsアプリでの

実用的メタプログラミング

@2013-12-17 Rubyの技術を語る1日 in 品川

MOROHASHI Kyosuke


諸橋恭介 (@moro)
Kyosuke MOROHASHI

はじめる！ Cucumber

諸橋恭介

日本語で書かれた受け入れテストをRubyで実行できるようにする。


ESM
Eiwa System Management

Agenda

- ✓ メタプログラミングとはなにか
- ✓ リフレクションtoolbox
- ✓ 実用メタプログラミング
- ✓ 共通機能の抽出
- ✓ ライブラリの作り方

今日のゴール

- Rubyではメタプログラミングは"ふつう"だと思えるようになる
- メタプログラミング用のRubyの機能について、概略をつかむ
- 実践的にメタプログラミングするときの注意点・考え方を理解する

メタプログラミング

Ruby

Paolo Perrotta 著
角征典 訳

メタプログラミングは

- Rubyではメタプログラミングは"ふつう"だと思えるようになる
- メタプログラミング用のRubyの機能について、概略をつかむ
- 実践的にメタプログラミングするときの注意点・考え方を理解する

メタプログラミングとはなにか

“

メタプログラミングとは、言語要素を実行時に操作するコードを記述すること

「メタプログラミングRuby」

Rubyの言語要素

- ✓ クラス
- ✓ モジュール
- ✓ オブジェクト(とその状態:インスタンス変数)
- ✓ メソッド
- ✓ 手続き

```
class Book
  def title
    @title
  end

  def title=(title)
    @title = title
  end
end
```

```
class Book
  def title
    @title
  end

  def title=(title)
    @title = title
  end
end
```

言語要素を操作して
何度も出ている
'title'からいろ
いろできないか？

```
class Book
  attr_accessor :title
end
```

```
def attr_accessor(name)
  define_method(name) do
    instance_variable_get("@#{name}")
  end
  define_method("#{name}=") do |val|
    instance_variable_set("@#{name}", val)
  end
end
```

```
define_method(:title) do
  instance_variable_get("@title")
end

define_method("title=") do |val|
  instance_variable_set("@title", val)
end
```

```ruby
define_method(:title) do
  instance_variable_get("@title")
end

define_method("title=") do |val|
  instance_variable_set("@title", val)
end
```

メソッドを動的に追加

```ruby
define_method(:title) do
  instance_variable_get("@title")
end

define_method("title=") do |val|
  instance_variable_set("@title", val)
end
```

インスタンス変数を

変数名を使って操作

Rubyでは、メタプログラミングとそう

でないものとに

明確な区別はない

まとめ: メタプログラミングとは

- ✓ 言語要素を操作する
- ✓ 実は特別なものでない
- ✓ Rubyでは日常的にやっている

リフレクション

toolbox

“

リフレクション (reflection) とは、プログラムの実行過程でプログラム自身の構造を読み取ったり書き換えたりする技術のことである。

http://ja.wikipedia.org/wiki/リフレクション_(情報工学)

メタプログラミング

Ruby

Paolo Perrotta 著
角征典 訳

メタプログラミングは

```
Class.is_a?(Object) #=> true
```

- Rubyではクラスそれ自身もオブジェクトである
  - Moduleクラスを継承した、Classクラスのオブジェクト
- 変数に入れたり、引数にしたりできる

```
klasses = {
  'array klass' => Array,
  'hash klass'  => Hash,
}

def class_name(klass)
  klass.name
end

klasses['array klass'].new #=> []

class_name(klasses['hash klass']) #=> "Hash"
```

```
Class.is_a?(Object) #=> true
```

- ✓ クラス定義を変更するメソッドがある
  - ✓ includeやextendやdefine_method もModuleクラスのメソッド

ブロック (Proc)

- 処理そのものをオブジェクトとして扱える
- Rubyの大きな魅力の一つ

ブロック (Proc)

```
# イテレータや処理の差し替え
multiplier = ->(i) { i * 2 }
square     = ->(i) { i ** 2 }

[1, 2, 3].map(&multiplier)
[1, 2, 3].map(&square)

# リソースの開閉
File.open(path, 'w') {|f| f.puts('content') }

# 遅延評価
scope :fresh,
      -> { where('created_at > ?', 3.hour.ago) }
```

2010年12月4日(土) 札幌Ruby会議03

Rubyの教えてくれたこと

— You must unlearn what you have learned.

株式会社 えにしテック

島田 浩二

koji.shimada@enishi-tech.com

http://www.slideshare.net/snoozer05/20101204-youmustunlearnwhatyouhavelearned

Object#send

- 引数で指定したメソッドを呼び出す
  - つまり、呼び出すメソッドをプログラム的に変更できる

Object#send

```
def up_or_down(condition)
  up_or_down = condition ? :upcase ? :downcase
  'Ruby'.send(up_or_down)
end

up_or_down(true)  # => 'RUBY'
up_or_down(false) # => 'ruby'
```

define_method

- モジュールやクラスに、インスタンスメソッドを追加するメソッド
  - defのリフレクション版
- メソッドの中身をブロックで書く
  - そのためブロック外の変数が見える

define_method

```
class Foo
  foo = 'FOO'
  define_method(:foo1) do
    puts foo + '1'
  end

  def foo2
    puts foo + '2'
  end
end

obj = Foo.new
obj.foo1 #=> 'FOO1'
obj.foo2 #=> NameError
```

class_eval/ instance_eval

- 文字列で書かれたRuby式やブロックを、クラスやインスタンスのコンテキストで評価する

class_eval/ instance_eval

- class_eval (module_eval)を使うシーン
  - Classクラスのprivateメソッドを呼んだり、
  - Classにあとからメソッドを追加したりする
- instance_eval を使うシーン
  - インスタンスのインスタンス変数を参照する
  - privateメソッドを呼ぶ

```ruby
class Foo
  def initialize(message)
    @message = message
  end
end

Foo.class_eval %q[
  def greet
    "#{@message} you!"
  end
]

foo = Foo.new('hi')

p foo.greet                              #=> 'hi you!'
p foo.instance_eval { @message } #=> 'hi'
```

instance_variable_set/ instance_variable_get

- インスタンス変数を名前ベースで取得したり設定したりできる
- おなじようにclass_varaible_set/getもある

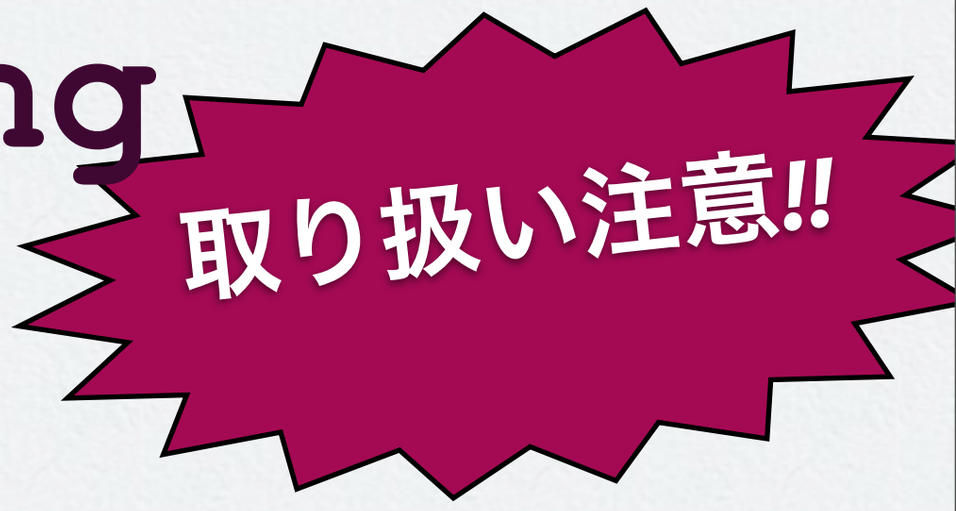
```ruby
class Foo
  def initialize(message)
    @message = message
  end
end

foo = Foo.new('hi')
foo.instance_variable_get('@message') #=> 'hi'
```

method_missing

取り扱い注意!!

- 存在しないメソッドを呼ばれた時に呼ばれるメソッド
- ActiveRecordの属性取得メソッドなどに使われている

method_missing




```
class Foo
  def method_missing(method, *args, &block)
    if method = :hi
      'hello'
    else
      super
    end
  end
end

Foo.new.hi #=> 'hello'
```

include,extend

- mixinもクラスやモジュールの言語要素を操作している
- includeはそのクラスのインスタンスメソッドを追加する
- extendはそのオブジェクトに(特異)メソッドを追加する

included, extended

- ✓ モジュールがincludeされたりextendされたりすると実行されるフック
- ✓ 継承のタイミングで実行されるinherited メソッドもある

included, extended

```
module MyModule
  def self.included(base)
    "included by #{base}"
  end

  def self.extended(obj)
    "extended by #{obj}"
  end
end

class MyClass
  include MyModule              # => "included by MyClass"
end

MyClass.new.extend(MyModule) # => "extended by #<MyClass:0x007fc
```

ActiveSupport::Concern

- Railsによるincludeの拡張
- そのモジュールに ClassMethods というモジュールがあれば、それをextendする
- included {} という、includedフック相当のクラスマクロがある

ActiveSupport::Concern

```
module MyModule
  included(base)
    "included by #{base}"
  end
  module ClassMethods
    def hi; 'hello' ; end
  end
end

class MyClass
  include MyModule  # => "included by MyClass"
end

MyClass.hi          # => "hello"
```

まとめ: リフレクション toolbox

- ✓ Rubyのクラスは「やわらかい」
- ✓ 操作するためのメソッドがたくさんある
- ✓ 「メタプログラミングRuby」おすすめ

実用メタプログラミング

メタプログラミングの落とし穴

```
Book = Class.new do
  define_method(:initialize) do |attrs|
    attrs.each do |key, value|
      if respond_to?("#{key}=")
        send("#{key}=", value)
      else
        instance_variable_set("@#{key}", value)
      end
    end
  end

  define_method(:price=) do |price_str|
    instance_variable_set(
      '@price',
       price_str.delete(',').to_i
    )
  end
end

p Book.new(author: 'moro', price: '1,980')
```

```ruby
class Book
  def initialize(attrs)
    @author    = attrs[:author]
    self.price = attrs[:price]
  end

  def price=(price_str)
    @price = Integer(price_str.delete(','))
  end
end

Book.new(author: 'moro', price: '1,980')
#=> #<Book:0x007faddb148b48 @author="moro",
@price=1980>
```

- メタプログラミングの
  コードは"難しい"

- とりわけリフレクション

- send期, eval期

メタに考えて

ベタに作る

`ActiveRecord::Base.has_many` *`assoc_name`*

`assoc_name` で指定された、複数の
子レコードへの関連を定義する。

```ruby
class Books < ActiveRecord::Base
  has_many :reviews
end

Book#reviews
    #reviews=
    #review_ids
    #review_ids=
    ...

Book#reviews.build
     reviews.create
     reviews.each {|review| ... }
     reviews.where(...)
```

たくさんメソッドができる

```
def has_many(name, opts = {})
  class_eval <<-RUBY
    def #{name}
      klass = #{name.to_s.classify}
      klass.where(#{fk}: id)
    end

    def #{name}=(value)
      values.each do |value|
        #{name.to_s.classify}.create!(...)
      end
    end
  RUBY
end
```

難しそうな実装イメージ

ActiveRecord::Base.has_many *assoc_name*

assoc_name で指定された、複数の
子レコードへの**関連を扱うオブジェクト**のための
ラッパーメソッドを定義する。

Book#reviews

- 紐付いているレビューを返す、
  のではない
- 親に紐づくレビューがあるという
  関連を表すオブジェクトを返す

```
def define_readers
  mixin.class_eval <<-CODE, __FILE__, __LINE__ +
    def #{name}(*args)
      association(:#{name}).reader(*args)
    end
  CODE
end
```

関連を表す

オブジェクトを返す

Associationの抽出

- ✓ owner と target がいる
- ✓ owner の id とtarget の fk で検索する scope を作る
- ✓ target をロードし、適切にメモ化
- ✓ 追加削除のコールバックを管理する

```
= Active Record Associations

This is the root class of all associations
('+ Foo' signifies an included module Foo):

  Association
    SingularAssociation
      HasOneAssociation
        HasOneThroughAssociation
        + ThroughAssociation
      BelongsToAssociation
        BelongsToPolymorphicAssociation
  CollectionAssociation
    HasAndBelongsToManyAssociation
    HasManyAssociation
      HasManyThroughAssociation
      + ThroughAssociation
```

- ~~親レコードから子レコードを引くメソッドを作る~~

- 親レコードと子レコードとの関連を表すクラスを作る

```
def define_readers
  mixin.class_eval <<-CODE, __FILE__, __LINE__ +
    def #{name}(*args)
      association(:#{name}).reader(*args)
    end
  CODE
end
```

リフレクションで薄いラッパーを作る

まとめ:
実用メタプログラミング

- ✓ メタに考えてベタに作る
- ✓ リフレクションでつなぐ

共通機能の抽出

投稿チェック

機能の例

- 記事(Post)とコメント(Comment)と画像(Photo)の投稿内容をチェック
- それぞれごとの内容で外部サービスに投稿したい
  - Postはタイトルと本文、コメントは本文、画像は表示URL

```
class Post < AR::Base
  after_save :submit_content_monitoring

  private

  def submit_content_monitoring
    content = [title, body].join("\n\n")

    url = Rails.config.censoring_endpoint
    req = Net::HTTP::Post.new(url.path)
    req.set_form_data('content' => content)
    Net::HTTP.start(url.hostname, url.port) do |ht
      http.request(req)
    end
  end
```

```ruby
class Comment < AR::Base
  ...
  def submit_content_monitoring
     content = body
     ...
  end
end

class Photo < AR::Base
  ...
  def submit_content_monitoring
    content =
      "<img src='http://img.example.com/#{id}' />"
    ...
  end
end
```

共通部分を

探す

```ruby
class Post < AR::Base
  after_save :submit_content_monitoring

  private

  def submit_content_monitoring
    content = [title, body].join("\n\n")

    url = Rails.config.censoring_endpoint
    req = Net::HTTP::Post.new(url.path)
    req.set_form_data('content' => content)
    Net::HTTP.start(url.hostname, url.port) do |ht
      http.request(req)
    end
  end
```

```ruby
class Post < AR::Base
  after_save :submit_content_monitoring

  private

  def submit_content_monitoring
    Censoring.new(
      [title, body].join("\n\n"),
      Rails.config.censoring_endpoint
    ).submit
  end
end
```

```
class Censoring
  def initialize(content, url)
     @content = content
     @url     = url
  end

  def submit
     req = Net::HTTP::Post.new(@url.path)
     req.set_form_data('content' => @content)
     Net::HTTP.start(@url.hostname, @url.port) do |
       http.request(req)
     end
  end
end
```

```ruby
class Post < AR::Base
  after_save :submit_content_monitoring

  private

  def submit_content_monitoring
    Censoring.new(
      [title, body].join("\n\n"),
      Rails.config.censoring_endpoint
    ).submit
  end
end
```

メタに考えてベタに作る

"対象クラスごとに、投稿チェック内容を組み立て送信する"クラスがほしい

```
class CensorAdapter
  def initialize(endpoint, &block)
    @endpoint        = endpoint
    @content_builder = block
  end

  def submit(record)
    content = @content_builder.call(record)
    Censoring.new(content, @endpoint).submit
  end
end

# ----------------
post_adapter = CensorAdapter.new(endpoint) do |post|
  [post.title, post.body].join("\n\n")
end

post_adapter.submit(post)
```

```
class Post < AR::Base
  @@censor_adapter =
    CensorAdapter.new(config.endpoint) do |post|
      [post.title, post.body].join("\n\n")
    end

  after_save :submit_content_monitoring

  private

  def submit_content_monitoring
    @@censor_adapter.submit(self)
  end
end
```

```
class Comment < AR::Base
  @@censor_adapter =
    CensorAdapter.new(config.endpoint) do |comment|
      comment.body
    end

  after_save :submit_content_monitoring

  private

  def submit_content_monitoring
    @@censor_adapter.submit(self)
  end
end
```

```
class Photo < AR::Base
  @@censor_adapter =
    CensorAdapter.new(config.endpoint) do |photo|
      "<img src='http://img.example.com/#{photo.id}' />"
    end

  after_save :submit_content_monitoring

  private

  def submit_content_monitoring
    @@censor_adapter.submit(self)
  end
end
```

リフレクションでつなげる

投稿するためのメソッドを追加してまわるのを、なんとかできないか

```
module CeonsoringDsl
  def censor_content(url, &block)
     adapter = CensorAdapter.new(url, block)
     after_save {|record| adapter.submit(record) }
  end
end
```

```
class Post < AR::Base
  extend CensoringDsl

  censor_content(config.endpoint) do |post|
    [post.title, post.body].join("\n\n")
  end
end
```

protip:

作ったいろんなクラスを Concernにまとめる

```ruby
module Censorable
  extend ActiveSupport::Concern

  module ClassMethods
    def censor_content(url, &block)
       adapter = Censorable::Adapter.new(url, bloc
       after_save {|record| adapter.submit(record)
    end
  end

  class Adapter
    ...

  class Request
    ...
end
```

```
class Post < ActiveRecord::Base
  include Censorable

  censor_content(config.censoring_endpoint) do |po
    [post.title, post.body].join("\n\n")
  end
end
```

小さく分けてテストする

ベタに作られた共通部分はテストしやすい

- ✓ Censorable::Request
- ✓ Censorable::Adapter
- ✓ 拡張されるAR(AMo)モデル

Censorable::Request

`#initialize(content, url)`
　送信内容とエンドポイントを受け取り

`#submit`
　そこにPOSTする

`#submit` の通信部分をモックすると、

テストしやすい(FakeWeb, WebMockなど)

Censorable::Adapter

#initialize(url, &block)
　エンドポイントとデータ変換方法(Proc)を受け取り

#submit(record)
　初期化時のurlと、recordから作った送信内容で
　Requestを作り、submit()

&blockとrecordを変えながら試すことで

変換エラーなどのバリエーションテストOK

AR(AMo)モデル

#create!
　保存されるデータから適切に作られた内容が
　監視サービスに送信される

拡張されたモデルの動きは「外側から」の振る舞いをテストする。

(ここでも、必要に応じて通信部分をモック)

まとめ:
共通機能の抽出

- ✓ メタに考えてベタに作る
- ✓ リフレクションでつなぐ
- ✓ 小さく分けてテストする

まとめ

今日のゴール

- Rubyではメタプログラミングは"ふつう"だと思えるようになる
- メタプログラミング用のRubyの機能について、概略をつかむ
- 実践的にメタプログラミングするときの注意点・考え方を理解する

RubyやRailsの開発では、メタプログラミングと、そうでないものの区別は曖昧。

いつも使っている機能が、実は動的メソッド定義だったりするし、それを自分で作ることも簡単

そこでやり過ぎないために、「メタに考えベタに書く」こ

とを心がける

(いっぽうでリフレクションを使いまくる

"練習"もしてみるとたのしいです)

包括的な完成品を目指すのではなく、実アプリの機能を、少しずつ注意深く"メタ"にしていくのが大事

それができる柔軟性が

Rubyの良さ

Matz says:

“

Rubyは君を信頼する。Rubyは君を分別のあるプログラマとして扱う。Rubyはメタプログラミングのような強力な力を与える。ただし、大いなる力には、大いなる責任が伴うことを忘れてはいけない。それでは、Rubyでたのしいプログラミングを。

「メタプログラミングRuby」 序文より