Reading Rails

ActiveRecord::Base / ActiveSupport::Concern

8章

Railsツアーの準備

今日学ぶこと

- *ActiveRecord*の設計と*ActiveRecord::Base*の役割
- Rails独自のクラスとメソッドの依存関係の管理手法
- *ActiveSupport::Concerns*の役割と仕組み
- 実践でのメタプログラミングの使いどこ

Railsの重要なコンポーネント

- *Active Record:* アプリケーションのオブジェクトをデータベースのテーブルにマッピングする
- *Action Pack:* “Web“の部分全般を扱う
- *Active Support:* 時刻計算やロギングなどの汎用的な問題を扱うユーティリティ

ソースコードの取得

```
$ git clone https://github.com/rails/rails.git

                or

$ gem unpack activerecord -v=4.1.0
```

Railsでかい

- *.rb: 279929行 (2015/11/01)

```
$ wc -l `find ./* -name '*.rb'`

279929 total
```

- 約4,000名のcontributor (Rails Contributors - All time)

9章

ActiveRecordの設計

ActiveRecordでかい

```
$ wc -l `find ./activerecord/* -name '*.rb'`

 98991 total
```

- activerecord/*.rb: 98991行 (2015/11/01)
- 98991 / 279929 = 0.35

ActiveRecord 復習

```
class Duck < ActiveRecord::Base
  validate do
    errors.add(:base, “Illegal duck name, “) unless name[0] == ‘D’
  end
end
```

- *ActiveRecord::Base*がマッピングクラスのスーパークラスになる。
- *Duck*クラスを*ducks*テーブルにマッピングして、そのattributeにアクセスするゴーストメソッドを定義する。

ActiveRecordはどのようにまとめられているか

- *ActiveSupport*と*ActiveModel*の２つのライブラリに大きく依存している。*(後述)*

- *autoload*でモジュールを*require*する
    - e.g. 最初に*ActiveRecord::Base*を使う時に、そのクラスを定義しているファイル *active_record/base.rb*のファイルを*autoload*が自動的に*require*する

- *extend ActiveSupport::Autoload* してるので*autoload*は*ActiveRecord*モジュールのクラスメソッドになる。*(クラス拡張)*

ActiveRecord

```
# activerecord/lib/active_record.rb

require 'active_support'

require 'active_model'

# …


module ActiveRecord

  extend ActiveSupport::Autoload


  autoload :Base

  autoload :Callbacks

  # …
```

ActiveRecord::Base

- 大量のモジュールを *extend && include*
- *run_load_hooks* オートロードされたモジュールが設定用のコードを呼び出せるようにするもの。=> *Base*が*include*したモジュールがさらにモジュールを*include*している
- *autoload*のおかげで、ソースコードを*ActiveRecord::Base*で*require*してから*include*する必要がない。*(include*だけで簡潔に！*)*
- ここを起点にどのメソッドがどのファイルで定義されてるのか探せばいい感じ！*(多分)*
    - *save*などの永続化のメソッドは、*ActiveRecord::Persistence*にある

```
# activerecord/lib/base.rb
module ActiveRecord
  class Base
    extend ActiveModel::Naming

    extend ActiveSupport::Benchmarkable
    extend ActiveSupport::DescendantsTracker

    extend ConnectionHandling
    extend QueryCache::ClassMethods
    extend Querying
    extend Translation
    extend DynamicMatchers
  # …
```

```
    extend Enum
    extend Delegation::DelegateCache

    include Core
    include Persistence
    include ActiveModel::SecurePassword
    include AutosaveAssociation
    # …
    include AutosaveAssociation
  end

  ActiveSupport.run_load_hooks(:active_record,
                               Base)
end
```

validateはどこに？

- ActiveRecord::Validationsというモジュールをインクルードしてる、ここっぽい。

- valid?はある

```
# activerecord/lib/active_record/validations.rb

module ActiveRecord

  # …

  module Validations

    include ActiveModel::Validations

    # …

    def valid?(context = nil) # …
```

validateはどこに？

- ActiveRecord::ValidationsがActiveMode::Validationsとやらをinclude している
- rails/activemodel/lib/active_model/validations.rbのL150にあった

```
 # rails/activemodel/lib/active_model/validations.rb
module ActiveModel
  module Validations
  # …
  def validate(*args, &block)
  # …
```

validateはどこに？ - ActiveModelってなんぞ？

- ActiveRecordからは独立している => ActionPackが非ActiveRecordなモデルとやりとりするために使うHelperの役割も持ってる (Rails Guides)
- *ActiveSupport::Concern*を使ってるので、クラスがモジュールをincludeするとインスタンスメソッドと同時に*validate*みたいなクラスメソッドまで一緒に手に入る (10章で謎解き)

validateはどこに？ - ActiveModelはなぜ必要か

- *validate*はもともと*ActiveRecord::Validations*で定義されていた
- 「DBの操作」と「オブジェクトモデルの操作」は別々に分離する方がいいんじゃないか (責務の単一化)
    - DBの操作...保存、読み込み => ActiveRecord
    - オブジェクトモデルの操作...属性を保持、属性が妥当かを追跡する => ActiveModel
- *valid?* : オブジェクトがデータベースに保存されたかどうかを確認する必要があるのでActiveRecord
- *validate* : オブジェクトの属性の妥当性をチェックするだけなのでActiveModel

validate?の旅

- 実際は*ActiveRecord::Validations*だけじゃなくて*ActiveModel::Validations*にも定義してあった
- *ActiveRecord::Validations#validate?*で*super*を呼び出してるし、ラップして機能追加してる

ActiveRecordの設計まとめ

- *ActiveRecord::Base*は、モジュールの集まり
- それぞれのモジュールが、*ActiveRecord::Base*にインスタンスメソッドとクラスメソッドを追加する
- *Base*クラスには、インスタンスメソッドが300個以上、クラスメソッドが500個以上(!!)
- これがRubyの設計技法 (設計技法は絶対ではなく、言語によって違ってくる)
- 疎結合性、テスト容易性、モジュールの再利用性
  - *ActiveRecord::Base*を無視して*ActiveModel::Validations*クラスだけ*include*して*validate*メソッドを利用することもできる

10章

Concernの設計

Concern以前のRails - includeとextendのトリック

```
module ActiveRecord
  module Validations
    def self.included(base)
      base.extend ClassMethods
      # …
    end

    module ClassMethods
      def validates_length_of(*attrs)
        # …
      end
    end

    def valid?
      # …
    end
  end
end
```

1. *ActiveRecord::Base*が*Validations*を*include*
2. *ActiveRecord::Base*を引数(*base*)にして*Validations*の*included*メソッドが呼ばれる
3. フックメソッドの中で、*Base*が*ClassMethods*モジュールを*extend*する
4. *ClassMethods*のインスタンスメソッド達が、*Base*のクラスメソッドとして追加される！

Concern以前のRails - includeとextendのトリック

- モジュールをincludeする一行だけでインスタンスメソッドとクラスメソッドの両方を取り込める！
- でもクラスメソッドを使うあらゆるクラスでincludedメソッドを実装しなきゃいけない
- includerに一行足す vs 取り込まれるモジュールでincluded全部定義する

```
class Base
  include Validations
  extend Validations::ClassMethods
  # …
end
```

```
module Validations
  def self.included(base)
    base.extend ClassMethods
    # …
  end
  # …
end
```

includeとextendのトリックの問題点

```
module SecondLevelModule
  def self.included(base)
    base.extend ClassMethods
  end

  def second_level_instance_method; ‘ok’; end

  module ClassMethods
    def second_level_class_method; ‘ok’; end
  end
end
```

```
module FirstLevelModule
  def self.included(base)
    base.extend ClassMethods
  end

  def first_level_instance_method; ‘ok’; end
  module ClassMethods
    def first_level_class_method; ‘ok’; end
  end

  include SecondLevelModule
end

class BaseClass
  include FirstLevelModule
end
```

includeとextendのトリックの問題点

```
$ curl -O https://raw.githubusercontent.com/
totzyuta/magick-of-ruby/master/workshop/chapter10/
chained_inclusions_broken.rb

$ pry -r ‘./chained_inclusions_broken.rb’


pry>BaseClass.new.first_level_instance_method
pry>BaseClass.new.second_level_instance_method
pry>BaseClass.first_level_class_method
pry>BaseClass.second_level_class_method
```

includeとextendのトリックの問題点

```
$ pry -r ‘./chained_inclusions_broken.rb’


pry>BaseClass.second_level_class_method”
=> NoMethodError
```

- *SecondLevelModule*を*include*するときの*Base*は、*FirstLevelModule*になってる！

Rails2での改修工事

```
module FirstLevelModule
  def self.included(base)
    base.extend ClassMethods
    base.send :include, SecondLevelModule
end
```

includeが以前はprivateだった。

答え

http://stackoverflow.com/questions/4213837/on-ruby-why-include-is-private-and-extend-is-public

2.0.0 だとNoMethodErrorになった

- *included*と*extend*のトリックは*FirstLevelModule*だけで使うことに

- *FirstLevelModule#included*の中で、インクルーダーに*SecondLevelModule*を*include*させる

- => *include*される*module*は、自分がどのレベルで*include*されるのか気にしておく必要ある！（つらそう）

ActiveSupport::Concern

ActiveSupport::Concern

- *ActiveSupport::Concern*を*extend*したモジュールを、*concern*と呼ぶ

- *concern*を*include*したクラスは、クラスメソッドとインスタンスメソッドの両方が手に入る

- 「クラスメソッドをインクルーダーに追加する」機能をうまいことカプセル化してる

- 以前は*ActiveSupport::Concern*はなかった！

ActiveSupport::Concern 使い方

```
require "active_support"

module IncludedClass
  extend ActiveSupport::Concern

  def an_instance_method; "instance method"; end

  module ClassMethods
    def a_class_method; "class method"; end
  end
end

class BaseClass
  include IncludedClass
end
```

```
pry>BaseClass.new.an_instance_m
ethod

=> ???

pry>BaseClass.a_class_method

=> ???
```

ActiveSupport::Concern仕組み

- モジュールが*Concern*を*extend*すると、*Ruby*がフックメソッドである*extended*を呼び出す => *@_dependencies*という依存関係を管理する**空の配列**をセットする

- *Concern#append_features*

Module#append_features (寄り道)

- Rubyの標準ライブラリ
- *Module#included*と似てる。けど違う。
- includeされたモジュールが、インクルーダーの継承チェーンに含まれているかどうかを確認してから追加する
- *append_features*をオーバーライドすると、モジュールがincludeされなくなってしまう (!!) => 普通は、もともとは空なフックメソッドであるincludedとかをオーバーライドしよう

Concern#append_features

- 基本コンセプト：concernのなかで、別のconcernをincludeしない
- concernではないモジュールにincludeされたら、全ての依存関係をインクルーダーに一気に組み込む

Concern#append_features

```
module ActiveSupport
  module Concern
    def append_features(base)
      if base.instance_variable_defined?(:@_dependencies)
        base.instance_variable_get(:@_dependencies) << self
        return false
      else
        return false if base < self
        @_dependencies.each { |dep| base.include(dep) }
        super
        base.extend const_get(:ClassMethods) if const_defined?(:ClassMethods)
        base.class_eval(&@_included_block) if instance_variable_defined?(:@_included_block)
      end
    end
  end
end
```

Concern#append_features 詳細①

```
def append_features(base)
  if base.instance_variable_defined?(:@_dependencies)
    # concernのとき
    base.instance_variable_get(:@_dependencies) << self
    return false
  else
    # concernじゃないとき
    # …
  end
end
```

インクルーダーがconcernのとき

- *@_dependencies*に*self*を入れる
- => 依存関係情報に自分自身(concernクラス)を追加する
- includeしてないことを示すために*false*を返す

Concern#append_features 詳細②

const_getを使わない場合、
ActiveSupport::Concern::ClassMethods
を参照してしまう。

```
# concernじゃないとき

return false if base < self

@_dependencies.each { |dep| base.include(dep) }

super

base.extend const_get(:ClassMethods) if const_defined?(:ClassMethods)

base.class_eval(&@_included_block) if instance_variable_defined?(:@_included_block)
```

継承関係を判定してbooleanを返す

Concernモジュールではなく、self(concern)のスコープで定数を読み込む

インクルーダーがconcernでないとき

- **継承チェーンに自分がいるときは、includeせずに*false*を返す**
- **インクルーダー(こいつが例えば*Base*とかになってる！)に今まで依存関係配列にストックしてきたクラスをどかどかincludeしていく**
- ***Module.append_features*で自分自身を継承チェーンに追加する**
- ***ClassMethods*を*Base*でextendしてクラスメソッドを追加する**

ActiveModel::Validationsでの使われ方

```
module ActiveModel
  module Validations
    extend ActiveSupport::Concern
    # …

    module ClassMethods
      def validate(*args, &block)
        # …
```

ActiveSupport::Concernまとめ

- 重複だらけの*include*と*extend*のトリックを外部ツールとしてカプセル化したもの
- 「concernのなかで、別のconcernをincludeせずに後でまとめてincludeする」ことで実現
- 対立する意見
  - *Concerns*は*include*の裏側で大量の魔法を使ってるので、コストになるのでは？
  - *Concern*のおかげでRailsのモジュールがスリムでシンプルになった

まとめ

まとめ

- Railsは、苦しみながら独自の依存管理システムを少しずつ成長させてきた
- シンプルな設計を目指すこと。
    - 「コードをシンプルに保ち、その仕事を成し遂げるための最も明白な技術」を使う
- メタプログラミングは、ある時点のコードが複雑になる、排除しずらい重複が生まれるときにだけここぞと使う、切れ味の鋭いツール

“メタプログラミングは賢くなるためのもので
はない。柔軟になるためのものである。”

– Paolo Perrota