Webエンジニア フルタイムコース
プロを目指すためのRuby入門 第8章

- モジュールの利用法について –
- 8.6 ~ 8.8 まで -

DIVE INTO CODE

WEF2105_太田　裕介

8.6 モジュールを利用した名前空間の作成

自分だけが使うプログラムだけならば、問題ないが大規模なプログラムやライブラリの作成では、クラス名の重複が問題になります。

Aさんのクラス

```
class Second
  def initialize(player,uniform_number)
    @player = player
    @uniform_number = uniform_number
  end
end
```

Bさんのクラス

```
class Second
  def initialize(digits)
    @digits = digits
  end
end
```

2つのクラスを同時に使う場合、どのように区別をすべきか？

```
#二塁手のAliceを作成する
#sBaseball::Second.new('Alice',13)
```

```
#時計の13秒を作成する
Clock::Second.new(13)
```

8.6 モジュールを利用した名前空間の作成

この場合、「名前空間(ネームスペース)」としてのモジュールです。
モジュール構文の中にクラス定義を書くと「そのモジュールに属するクラス」となり、同名のクラスがあっても外側のモジュール名が異なれば名前の衝突はなくなります。

```
module Baseball
  #これはBaseballモジュールに属するSecondクラス
  class Second
    def initialize(player,uniform_number)
      @player = player
      @uniform_number = uniform_number
    end
  end
end
```

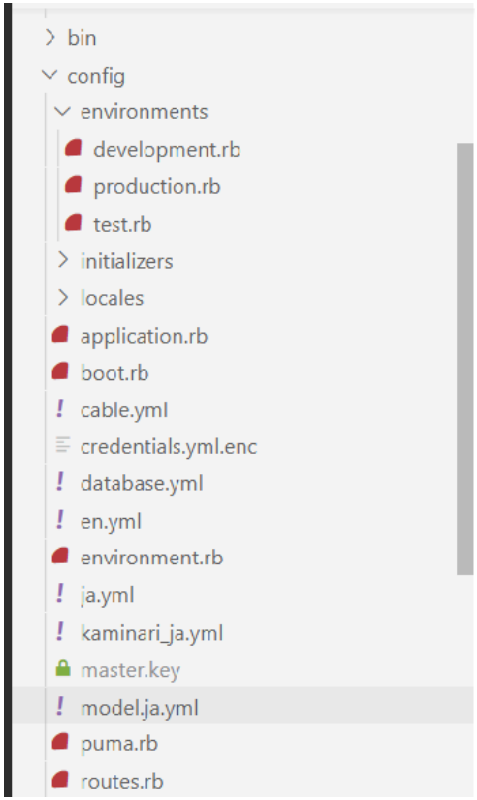
```
module Clock
  # これはClockモジュールに属するSecondクラス
  class Second
    def initialize(digits)
      @digits = digits
    end
  end
end
```

```
#二塁手のAliceを作成する
#Baseball::Second.new('Alice',13)
→　@player="Alice",
@uniform_number=13
```

```
#時計の13秒を作成する
Clock::Second.new(13)
→　@digits=13
```

8.6 モジュールを利用した名前空間の作成

名前空間は名前の衝突を防ぐためでなく、クラスのグループ分け/カテゴリ分けをする目的で使う場合もあります。
たとえば、RailsのGitHubのリポジトリです。多数のディレクトリに分かれています。



「〇〇フォルダの□□フォルダにある△△クラス」として名前空間が作られます。

```
Require
"active_support/core_ext/string/conversions"

Module ActiveRecord
  module Associations
   #Keeps track of tablealiases・・・・
```

8.6 モジュールを利用した名前空間の作成

名前空間として使うモジュールがどこかで定義されている場合は、
モジュール構文やクラス構文をネスト（入れ子）にしなくても
“モジュール名：：クラス名”の形でクラスを定義できます。

```
#すでにBaseballモジュールが定義されている。
module Baseball
End

 # モジュール名::クラス名の形でクラスを定義できる。
 class Baseball::Second
   def initialize(player,uniform_number)
     @player = player
     @uniform_number = uniform_number
   end
 end
```

8.6 モジュールを利用した名前空間の作成

コラム：トップレベルの同名クラスを参照する。（意地悪な例）
上側クラスが名前空間なしで定義され、下側がモジュールに属している。

```
#トップクラスのSecondクラス
class Second
  def initialize(player,uniform_number)
    @player = player
    @uniform_number = uniform_number
  end
end

module Clock
  # ClockモジュールのSecondクラス
  class Second
    def initialize(digits)
      @digits = digits
    end
  end
end
```

これをClockモジュールの中でSecondクラスでトップクラスのSecondクラスを参照する。

```
module Clock
  class Second
    def initialize(digits)
      @digits = digits
  # クラス名の前に :: を付けるとトップクラスのSecondを指定したことになる。
      @baseball_second = ::Second.new('Clock',10)
    end
  end
end
#iniitializeメソッドの中でトップレベルのSecondを参照
Clock::Second.new(13)
```

明示的にトップレベルのモジュールやクラスを指定するには：：にする。
（私の環境では参照できませんでした…。）

8.7 関数や定数を提供するモジュールの作成

モジュールのメソッドにクラスなどを入れることを説明しましたが、場合によってはモジュール単体でそのメソッドを呼び出すケースがあります。モジュール自身に特異メソッドを定義すれば、直接“モジュール.メソッド名”で呼び出すことができます。

```
Loggable.log('Hello.') #=>[LOG] Hello.
```

```
module Loggable
  #  特異メソッドとしてメソッドを定義する。
  def self.log(text)
    puts "[LOG] #{text}"
  end
end
```

```
module Loggable
  class << self
    def log(text)
      puts "[LOG] #{text}"
    end

  # 以下、ほかの特異メソッドを定義する。
  end
end
```

上記だと、特異メソッド(クラスメソッド)を定義した場合とほとんど同じです。単なるメソッドの集まりになります。
モジュールでもクラス同様に class << self を使って他の特異メソッドを定義することができます。メソッド名からself.を減らすことが出来ます。

※class << self
self.を作成しない手法。

8.7 関数や定数を提供するモジュールの作成

```
module Loggable
    #   特異メソッドとしてメソッドを定義する。
    def log(text)
      puts "[LOG] #{text}"
    end
  #モジュールの特異メソッドとしても使える。
  #対象メソッド定義よりも下で呼び出すこと
   module_function :log
end

# 特異メソッドとしてlogメソッドを呼び出す
Loggable.log ('  Hello.')  #=>[LOG]Hello.

#Loggableモジュールをincludeしたクラス
class Product
  include Loggable

  def title
    log 'title is called'
  end
end
```

モジュールではmixin(**ミックスイン**とは、クラスにモジュールで定義したメソッドを取り込み拡張すること。)として使えて、さらにほかの特異メソッドとしても使える,Module_functionメソッドがあります。この様に使えるメソッドをモジュール関数と呼びます。

```
#ミックスインとしてlogメソッドを呼び出す。
Product = Product.new
Procuct.title
#=>[LOG] title is called
```

8.7 状態を保持するモジュールの作成

クラスに定数を定義できたように、
モジュールにも定数を定義が出来ます。

```
module Loggable
  #定数を定義する。
  PREFIX = '[LOG]'.freeze

  def log(text)
    puts {PREFIX}#{text}"
  end
End

 #定数を参照する。
 Loggable:: PREFIX #=> "[LOG]"
```

※freeze　モジュールを凍結（内容の変更を禁止）します。

モジュールにも関数や定数を持つライブラリがありMathモジュールがあります。これは数学で使う関数が数多く定義されています。

```
#特異メソッドとしてsqrt（平方根）を利用
Math.sqrt(2) #=> 1.41421356・・・

Class Calculator
  include Math
  def calc_sqrt(n)
  #ミックスインとしてMathモジュールのsqrtを使う
    sqrt(n)
  end
End

calculator = Calculator.new
calculator.calc_sqrt(2) #=> 1.41421356・・・
```

他にはKernelモジュールがあります。
put p print require loop　…　他　多数

8.8 関数や定数を提供するモジュールの作成

クラスインスタンス変数を使って、クラス自身にデータを保持する方法がありますが、この方法はモジュールでも使うことが出来ます。

```
module AwesomeApi
# 設定時を保持するクラスインスタンス変数を用意する。
  @base_url =''
#クラスインスタンス変数を読み書きの特異メソッドを定義する。
   class << self
      def base_url = (value)
        @base_url = value
      end
      def base_url
       @base_url
      end

   end
end

#設定を保持・参照する
AwesomeApi.base_url = 'http://example.com'
AwesomeApi.base_url #=> "http://example.com"
```

モジュールの利点はインスタンス化できない点がクラスと異なります。オブジェクトの中で値を保存しておくために利用されるのがインスタンス変数ですが、何も操作しない場合はモジュールにしておけば、心配がありません。

左の例文について
※attr_accessor を使えば一行ですみます。読み取りメソッドと書き込みメソッドの両方を定義します。
attr_accessor　：base_url

8.8 関数や定数を提供するモジュールの作成

コラム （コード例は割愛。）

Singleton モジュール

モジュールではなく、クラスを使ってデザインパターン本来の「シングルトンパターン」を実現する場合は、Singletonモジュールを使うと便利です。

利点

・newメソッドがprivateメソッドになり、外部から呼び出せなくなる。
・クラスの特異メソッドとしてinstanceメソッドが定義され、ここから「唯一、１つだけ」のインスタンスを取得する。

まとめ

この章ではモジュールの利用として以下の4つを紹介しました。

・モジュールのミックスイン(includeとextend)
・モジュールを利用した名前空間の作成
・関数や定数を提供するモジュールの作成
・状態を保持するモジュールの作成

おわり