CHAPTER 一

Ch1 : 學習寫出好的 *Rails code*

序

我剛開始接觸 Rails 是在 2007 年。Rails 在世界剛竄紅。當時遑論說要一個 Developer 寫出容易維護的程式碼，基本上，在那個時代能夠使用 Rails 「兜」*(註）出一個堪用的網站，你都算一個高手了。

經過了四年，這個框架一路發展到現在。能夠自在的使用 Rails 的 Developer 在台灣卻也都還不算多。其中有很多原因，除了本地語言的文件不夠多之外，文件和 API 更迭過速導致開發環境破碎，也對後進者造成很大程度的學習障礙。

這一行可以說大家都是邊做專案，邊熬夜翻網路文件，跌跌撞撞把網站「拼」出來的。

<u>維護擴充性的重要性</u>

有些人，幸運的只是接案維生的 Developer，在練功「拼」完專案之後，就不需要再維護這些急就章的程式碼。

但在這個世界上，絕大多數的人，還是屬於服務於企業中，需要持久維護一個專案的 Developer。

這兩種人，我都當過。

因為浸淫在 Rails 的開發世界夠長，本身培育過 Developer，也開發過不少專案。在經歷維護專案，並持續升級 Rails 版本後，我漸漸能體悟到寫出好維護的程式碼是多麼重要的一件事。

如果程式碼亂七八糟：

* 你根本無法進行網站下一次的改版擴充動作，即便是功能上的小改版
* 你根本無法進行 Rails 版本升級的動作，因為你會被過去造成的錯誤（不管是自己或者是夥伴）不斷懲罰，導致寸步難行。

接著業務拓展和工作進度就會開始被這些程式碼開始扯到後腿...

寫程式變成了一件不快樂的事。

沒有書教你怎麼寫出好的（ Rails ）程式碼

但話說回來，寫出這些亂七八糟的程式碼，也不是開發者本身故意的。

畢竟 Rails 是一個進化很快的 Framework，要能跟上這個 Framework API 與架構的最新進度，已經很吃力了。很難奢求一般的 Developer 去追求所謂的 Rails Best Practices。

而大多數的狀況是，坊間的書籍與網站往往只教你：

* 如何快速把開發環境建置起來，開發一個示範站台，利用 plugins 建構出簡單的 feature。 --- 看完了這本書只學會 CRUD。

* 另外一個極端，將所有 API 全數羅列出來， --- 翻的時候只想睡，翻完之後不知如何應用在自己 project 之內

其他時候，只能全憑自己經驗或碰運氣看網路上找來的文件。功能可以勉強在時程內湊出來就萬幸，更別提寫出「好」的程式碼了。

This book is for you

網路上不乏許多關於 Rails 的 Best Practices，Antipattern 的演講。比如 ihower 的 [Rails Best Practices] 就是一篇對於寫出高品質 Rails code 很棒的指南。

但在實戰和平日社群交流中，我發覺大家需要的卻不是 Best Practices。Best Practices 離大家的生活太遙遠。大家所需要的是「Practices」：

* 如何寫出容易維護的程式碼
* 如何避免寫出的糟糕程式碼
* 如何遵循 Rails 架構，寫出標準好維護的程式碼
* 如何透過 Rails 內建機制整理，經過專案積累而日益肥大的 Code Base
* 如何搭配 RubyGems 讓自己的程式碼更乾淨、整潔，更易讀。

在開發功能，專案更迭中，自然持續的維持著專案的整潔。保持著可以繼續高速前進的心情與步調，愉快的工作。

這才是寫 Ruby / Rails 的初衷不是嗎？

本書架構

在接觸過無數的專案之後，我發現許多讓人無法繼續維護的 Rails code。除去開發者偷懶的因素，剩下的原因幾乎都是肇因對於 Ruby / Rails 基礎 API 以及架構不了解，所寫出的問題代碼。

所以這本書的順序將會是這樣的：

1. **介紹** Ruby **通用的的基本** Coding Style，**包含程式碼的寫作慣例**
2. **介紹** Rails **容易被誤用，但在開發中相當重要的基本工具。**
3. Application **的** AntiPattern，**什麼是絕對錯誤的架構設計**
4. **利用** Rails **原生架構實現的程式碼整理術**
5. **利用** 3rd Party Gem **實現的程式整理術**

CHAPTER 二

Ch2：基本的 Ruby Style

縮排慣例

雖然 Ruby **是一門能夠讓開發者發揮十足創意的魔幻語言，你可以用很多種方式完成同一件事。但這個語言的圈子內的開發者，在** Coding Style **上其實還是有一定的默契存在。**

Ruby **的縮排是** 2 **個空格。不是** 3**，也不是** 4**。**

```
# good
def hello
  "Hello World"
end

# bad
def hello
    "Hello World"
end
```

編輯器的自動縮排功能

最多開發者使用的兩套文字編輯器是Vim **與** TextMate**。**

* TextMate

Texmate **內建有** Ruby **與** Ruby on Rails **兩套** Bundle**。開發者在撰寫的時候就會自動進行縮排。而如果在開發時遇到其他人不小心將程式碼排的歪七扭八。也可以使用** Code Beautifier **這套** Bundle **自動整理程式碼。**

* Vim

可以使用 vim-ruby **和** vim-rails **這兩套** plugin**。**

命名慣例

變數或者是 method 名稱，採用 snake_case

```
def credit_card_discount
  original_price * 0.9
end
```

Class 和 Module 名稱，採用 CamelCase

```
class UserProfile
  def initialize(name)
    @name = name
  end
end
```

CONSTANT 使用 SCREAMING_SNAKE_CASE

```
class Invoice
  CREDIT_CARD_TYPE = ["VISA","MASTER"]
end
```

迴圈慣例

單行的迴圈使用 { } ，多行的迴圈使用 do end

```ruby
10.times {|n| puts "This is #{n}"}

10.times.do |n|
  puts "This is #{n}"
  puts "That is #{n*2}"
end
```

大師 Jim Weirich 在 Braces VS DO/END 這篇提出他的觀點 這篇提出他的觀點：

* Use { } for blocks that return values
* Use do / end for blocks that are executed for side effects

```ruby
# block used only for side effect
list.each do |item| puts item end

# Block used to return test value
list.find { |item| item > 10 }

# Block value used to build new value
list.collect { |item| "-r" + item  }
```

使用 each 而非 for

Ruby 中跑迴圈的方式可以有很多種，最常使用的是 for 和 each

使用無窮迴圈測試四種常見迴圈寫法。

```ruby
Benchmark.bm do |x|

  x.report('while1') do
    n = 0
    while 1 do
      break if n >= 1000000
      n += 1
    end
  end

  x.report('loop') do
    n = 0
```

```
  loop do
    break if n >= 1000000
    n += 1
  end
end

x.report('Infinite.each') do
  n = 0
  (0..(1/0.0)).each do
    break if n >= 1000000
    n += 1
  end
end

x.report('for Infinite') do
  n = 0
  for i in (0..(1/0.0)) do
    break if n >= 1000000
    n += 1
  end
end

end
```

跑出來的數值如下

```
                user     system      total        real
while1      31.960000   0.150000  32.110000 ( 36.749597)
loop        42.240000   0.190000  42.430000 ( 45.533708)
Infinite.each  108.400000   0.970000 109.370000 (117.421059)
for Infinite   112.070000   1.010000 113.080000 (126.712828)
```

for **明顯比** each **慢上許多。**

資料來源：http://www.ruby-forum.com/topic/179264

括號慣例

定義 method 要加括號。除非沒有參數。

```
# 有括號
def find_book(author,title, options={})
  # ....
end

# 無括號
def word_count
  # ....
end
```

使用 method 要加括號。除非是 statement 或 command

```
# 有括號

link_to("Back", post_path(post))

# 無括號

redirect_to post_path(post)
```

statement 多數時候要加括號，但如果狀況單純可不加括號

```
# 無括號

if post.content_size > 100
  # ....
end

# 有括號

if (post.content.size > 100) && post.is_promotion?
  # ....
end
```

「使用括號」是一個相對比較好的習慣

看到這裡都開始讓人搞迷糊了。到底是什麼樣的情況要加括號？什麼樣的情況不加括號？筆者的建議是，一旦沾到相當較複雜的陳述句時，儘量加上括號。Ruby 雖然是一個相當自由的語言，可以用空格(space)或是括號(parentheses)傳

入參數。但濫用空格可能造成 Ruby 在 parsing 陳述句的時候解讀異常，產出非正確的結果，這就是大家所不願意樂見的狀況了。

括號的唯一例外：super 與 super()

在 Ruby 中，唯一加括號和不加括號會有別的是 super 與 super()。它們代表了不同的意思：

* super 不加括號表示呼叫父類別的同名函式，並且將本函式的所有參數傳入父類別的同名函式。

```
class Foo
  def initialize(*args)
    args.each {|arg| puts arg}
  end
end

class Bar < Foo
  def initialize( a, b, c)
    super
  end
end
```

如果執行

```
> a, b, c = *%W[a b c]
> Bar.new a, b, c
```

將會印出 a b c

* super() 帶括號則表示呼叫父類別的同名函式，但是不傳入任何參數。

但如果是

```
class Foo
  def initialize(*args)
    args.each {|arg| puts arg}
  end
end

class Bar < Foo
  def initialize( a, b, c)
    super()
  end
```

```
end
```

執行

```
> a, b, c = *%W[a b c]
> Bar.new a, b, c
```

則不會印出任何東西。

布林慣例

* 布林邏輯使用 && 與 || ，而非 and 和 or

不少人在寫邏輯判段式時，會誤以為 && 和 || 與 and / or 是等價的，因此產出不少 bug。其實 and 和 or 跟 if / else 是等價的。所以請千萬不要在 boolean 邏輯內使用 and / or。

* 被判斷為 foo = (42 && foo) / 2

```
foo = 42 && foo / 2
=> NoMethodError: undefined method '/' for nil:NilClass
```

* 使用 and

```
food = 42 and foo / 2
=> 21
```

and 與 or 真正的用法

and 的用法

```
next if widget = widgets.pop
```

相當於

```
widget = widgets.pop and next
```

or 的用法

```
raise "Not ready!" unless ready_to_rock?
```

相當於

```
ready_to_rock? or raise "Not ready!"
```

參考資料

* Avdi 的 Using “and” and “or” in Ruby

邏輯慣例

if / unless 的寫作觀念

Ruby 在邏輯控制的部份，除了提供 if 還提供了 unless。

unless 等於 「if not something 」等於 「if !something」。

不過雖說 Ruby 提供了 unless 這個用法，但在實務上來說，一般還是不太推薦使用 unless。除了以下幾種狀況：

當語意較適合時，使用 unless

```
unless content.blank?
  # ....
end
```

沒有 else 的時候，使用 unless

當沒有 else 的時候，看起來還算 OK

```
unless foo?
  # ....
end
```

但加上一個 else，看起來就不是那麼直觀了

```
unless foo?
  # ....
else
  # ....
end
```

如果專案當中有這樣的 code，相信我，換成 if 的陳述會直觀許多。

```
if !foo?
  # ....
else
  # ....
end
```

當只有一個條件時，使用 unless 很適合。但多個條件時，使用 unless 很糟糕。

```
unless foo? && baz?
  # ....
end
```

相同的，改成 if **也會直觀許多**

```
if !foo? || !baz?
  # ...
end
```

if / else **牽扯到的** Fail Close **安全觀念**

在撰寫安全性程式碼時，使用 if **和** unless**，也會產生截然不同的差別。在** Security on Rails **中有介紹** Fail open **與** Fail close **兩種觀念：**

"Fail open" way, it's bad

```
def show
  @invoice = Invoice.find(params[:id])
  unless @user.validate_code( @invoice.code )
    redirect_to :action => 'not_authorized'
  end
end
```

"Fail close" way

```
def show
  @invoice = Invoice.find(params[:id])
  if @user.validate_code( @invoice.code )
    redirect_to :action => 'authorized
  else
    redirect_to :action => 'not_authorized'
  end
end
```

使用 Fail open**：「條件不成功，才不允許，不然就允許。」出包的機率遠比** Fail close**：「如果條件成功，才允許進行，不然就不允許。」高的許多，這也是值得注意的地方。**

使用三重運算子簡化 if / else

比較簡單的 if / else statement

如:

```
if some_condition
  something
else
  something_else
end
```

其實可以簡化成 :

```
some_condition ? something : something_else
```

不要濫用三重運算子

雖說 Ruby 提供這樣的簡化方式，但儘量還是不要濫用。如果超過兩層還是要將之拆開

```
# bad
some_condition ? (nested_condition ? nested_something :
nested_something_else) : something_else

# good
if some_condition
  nested_condition ? nested_something : nested_something_else
else
  something_else
end
```

其他慣例

字串慣例

使用 "#{}" 而非 + 串接字串。

```
# bad
full_name = first_name + ' <' + last_name + '>'

# good
full_name = "#{first_name} <#{last_name}>"
```

原因：使用 string interpolation **可以讓程式顯得更直觀。而且** + **會產生一堆不必要的** new object。

使用 << 而非 + 串接字串

```
html << "<h2>Post Title </h2>"
```

原因：String#<< **比** String#+ **速度快的多。而且** + **會產生一堆不必要的** new object。

偏好使用 " " (double quote) 而非 ' '(single quote) 包覆字串

```
name = "foobar"
string = "#{name}"

# => foobar

string = '#{name}'

# => #{name}
```

原因： 雖然 ' ' **和** " " **都可以用來宣告字串。但** " " **才有** string interpolation (double quote)**效果。**

使用 %() 處理需要 string interpolation 但同時也需要 " " (double quote) 的狀況

有時候我們不避免的需要寫出這樣的 code

```
<%= "<div class=\"name\"> #{name} </div>" %>
```

雖然可以透過將 " " **換成** ' ' **讓** code **不那麼粘膩。**

```
<%= "<div class='name'> #{name} </div>" %>
```

但其實還有這一招使用 `%()`，可以確保事情不會變得更加複雜。

```
<%= %(<div class="name">#{name}</div>) %>
```

Array 慣例

當要宣告一個擁有多字串的 Array 陣列時，偏好使用 %w

```
array = ["A", "B", "C", "D"]

# better
array = %w(A B C D)
```

Hash 慣例

使用 :symbol 而非 "string" 作為 Hash 的 key

```
# bad
{ "foo" => "bar" }

# good
{ :foo => "bar" }
```

原因：若使用 "string"，Ruby 每次都會為 key 在不同的記憶體位置再產生一個新 string object 。

使用 "string" :

```
string1 = { "foo" => "bar" }
string2 = { "foo" => "rab" }
string1.each_key {|key| puts key.object_id.to_s}
# => 2191071500
string2.each_key {|key| puts key.object_id.to_s}
# => 2191041700
```

改採用 :symbol :

```
string1 = { :foo => "bar" }
string2 = { :foo => "rab" }
string1.each_key {|key| puts key.object_id.to_s}
```

```
# => 304828
string2.each_key {|key| puts key.object_id.to_s}
# => 304828
```

這就是鼓勵使用 [:symbol] **而非** "string" **作為** Hash **中的**key**的原因，在很多記憶體使用量的膨脹和浪費的** case **中，多數都是因為產生了過多不必要的** object **所造成的問題。**