#001

Rails 101

Learning Rails in 7 days

by xdite

http://leanpub.com/rails-101

Rails 101

xdite

This book is for sale at http://leanpub.com/rails-101

This version was published on 2013-12-30

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By xdite

Rspec 101

Maintainable Rails View

Lean SaaS

中文世界唯一一本 *Rails 4.0.0 + Ruby 2.0.0* 的自學書籍

Contents

CHANGELOG

- 2013/12
  - Fix 第四章錯誤
  - Upgrade Bootstrappers
  - Fix Bootstrappers 錯誤
  - 把內建 mysql 改為 SQLite 避免更多錯誤
  - Fix production hack
  - 把 TODO 解說補完
  - 新增 RESTFul 兩篇文章
  - 加上 Recap
  - 加上 Facebook 社團 https://www.facebook.com/groups/rails101/
- 2013/11
  - 修復 Bootstrapper, 更新專案至 Rails 4.0.0 書中錯誤, 加上程式碼的 Repo
  - 加上 Helper 章節
  - 加上 Partial 章節
  - 加上 Scope 章節
  - 加上 Capistrano 章節
  - 加上 Asset Pipeline 章節
  - 加上 Rails 4.0.0 production hack
- 2013/06 更新至 Ch6

作者介紹

我是 xdite。以 Ruby on Rails 撰寫網站已經累積接近 6 年的時間 (Since 2007)。

我有一個以 Web 開發經驗為主的 blog Blog.XDite.net，不定期會發表各式各類以 Rails 開發為主軸相關的文章。

我曾經受邀至 Ruby Taiwan Conf、Ruby China Conf、Reddot Conf (Singapore Ruby Conf) 發表 Rails 開發相關的演說。

我曾經以 Ruby on Rails 作為開發技術，奪得 Facebook World Hack 2013 Global Grand Prize。

我現在在 Rocodev 工作，這是我持有的軟體公司。

Logdown

Logdown 是 Rocodev 開發的一個 Blog 系統，非常適合拿來寫技術筆記。

購買本書的讀者可以使用 rails101 這個 coupon 獲得 USD $10 的 Logdown 折扣。

致謝

感謝讀者 sdlong 幫忙此書的校對與範例圖片繪製

如何排除障礙

1. 加入 https://www.facebook.com/groups/rails101/ FB 社團。在社團上問問題
2. 在 Google 或 Stack Overflow 上用錯誤訊息關鍵字找答案。
3. 你是不是忘了加 @ 呢?
4. 你是不是忘了跑 migration 呢?
5. 你是不是忘記加欄位呢?
6. 你是不是忘記 touch tmp/restart.txt 呢?
7. 在 http://guides.rubyonrails.org 找線索
8. 檢查 development.log 或者是 Chrome 的 DevTools Console

本書使用方法

我自 2009 年以來，就開始訓練 Rails Developer。訓練方式是使用一系列題目，培養開發者對於 Rails 相關工具的熟悉度。這一本書是該系列題目的答案本。

這本書的前身，其實是一套訓練新進 Developer 的基礎教材。目的是希望能夠讓一個剛接觸 Rails 的開發者，快速熟悉 Rails 生態圈裡面的基本工具，以及練熟 / 背熟日常所需要用的知識。

我不建議各位讀者以「讀」的方式去使用這本書，相反地，我希望你動手實作。

解完這本書裡面所有的題目才是重點，你會在解題的過程裡面學到從無到有 build 起一個 Rails 網站，所需要的所有基本常識。

如果你真的解不開這裡面的題目，我才希望你看解答。

如果你偏好當面問也住在台北，我們 host 了一個每週二舉辦的 Rails Meetup http://www.meetup.com/taipei-rails-meetup/ ，歡迎帶著你手邊的問題來這裡問。這邊的同好對於新手都很友善，會十分樂於回答你的問題而已。

如果你不住在台北，我開了一個 Facebook 社團 https://www.facebook.com/groups/rails101/，這裡可以詢問本書相關的問題。

發問請貼程式碼，可將程式碼貼在 Gist 上，再轉貼到論壇上發問。

本書程式碼

本書程式碼放在：https://github.com/xdite/groupme

開發環境

我力求提供讀者一個能夠上手的 Rails 開發環境。但是要能夠建築出一個開發 Rails 的環境，變因實在太大。從

- MacOS 的版本 10.7, 10.8, 10.9 的 command line 工具安裝方式與位置都不同
- Debian 5,6 / Ubunbtu 10,11,12 的 apt-get 安裝 lib 位置與相依都不同
- Ruby 從 1.8.7 -> 1.9.3 -> 2.0.0 的語言變更導致 library 不相容
- Rails 從 3.0 -> 3.1 -> 3.2 到 4.0.0 rc1, beta1, official 的官方配置與 deploy 環境都不同

如果本書提供的環境安裝方式無效，請儘量試試在 Stackoverflow 上找找，也許很快就能解決你的問題。

學習 Rails 前置所需要的技能

經過這些年的 Rails Developer 培訓之後，我强烈建議開始學習 Rails 之前，請先學會以下相關技能:

- Git
- 熟習 Vim 以及 SublimeText2 其中一種開發工具
- 熟悉 Linux Command Line 的操作

Git 學習資源

Codeshool 出了三套 Git 課程。請先練習過後，再開始學習 Rails 開發，我相當不建議讀者跳過 Git 技巧的練習。

- TryGit http://www.codeschool.com/courses/try-git
- GitReal http://www.codeschool.com/courses/git-real
- GitReal2 http://www.codeschool.com/courses/git-real-2 （進階技巧，可以之後再學）

練習作業

1. 上 Github 註冊一個帳號
2. 練習以下 Git 指令

- git commit
- git push
- git pull
- git branch
- git checkout
- git merge

編輯器學習資源

Vim

- c9s 的 Vim Hacks 是相當好的 vim 學習資源

Sublimtext 2

- Sunlimtext 2 下載網址: http://www.sublimetext.com/2
- Sublime Text 台灣 社群最近也撰寫了一本相當不錯的SublimeText 學習手冊

Linux Command Line 學習資源

- PeepCode 的 Meet the Command Line
- PeepCode 的 Advanced Command Line

練習作業

- 練習 Meet the Command Line
- 練習 Advanced Command Line

Ruby on Rails 安裝最佳實踐

打造 Bug Free 的 Rails 開發環境

許多新手初入門 Rails，除了對 Rails 版本號更迭過快感到十分抓狂，開發環境的建置也常令人一個頭兩個大。不是卡在 lib 編不過，就是 gem 抓不到。

OSX 10.8、MySQL、ImageMagick、readline ....

project 還沒開始寫半行，先被困在莫名其妙的套件相依性上。有沒有比較簡單且不容易踩中蟲的安裝步驟呢？有。

我們公司 Rocodev 寫了一份 Ruby on Rails 安裝最佳實踐，這份教學幾乎是 Bug Free。

最新版本

最新版本在此：https://github.com/rocodev/guides/wiki/setup-mac-development 如果書中的內容過期，請到此頁面找尋最新版解法。

Mac 是最好的 Ruby on Rails 開發環境，馬上買一台！

看到標題，也許你心裡浮出了問號？我只是想試看看 Rails，有必要這樣大手筆的購買設備嗎？

有！如果你立志相成為一個 Rails Developer 的話。

世界上絶大多數的 Rails Developer 開發都是使用 MacBook +。這不僅僅只是 Best Practices / 神兵利器（brew、Livereload ...）只存在 Mac 的關係。更重要的是每當 OS 更新、Gem 版本更新、Rails 地雷 ...

Linux 使用者都會因為 package system maintainer 不是 Rails Developer 而不熟生態圈的關係，被雷炸的像次等公民。

開發者的時間就是金錢，想想好的設備會為你的生產力帶來多大的改善？好的 Framework 會多節省你的開發時間？

你都已經打算開始學習 Rails 改善你悲慘的開發人生了？為什麼不買一台 Mac 讓自己更節省力氣呢？

安裝步驟

強烈警告：請絕對不要跳著裝！如果疏漏步驟有可能導致無法復原需要重灌。

系統套件

1. 進行 Mac 系統更新
2. 從 Apple Store 上取得 Xcode 4.4 安裝
3. `Xcode -> Preferences -> Downloads tab then install the` “`Command Line Tools.`” 裝完請重開機（保險起見）。

安裝 Homebrew

```
$ ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
$ brew install git
$ brew update

$ brew tap homebrew/dupes
$ brew install apple-gcc42
```

安裝 XQuartz

安裝 ImageMagick 需先有 X11 的 support，OSX 10.8 拿掉了...

http://xquartz.macosforge.org/landing

ImageMagick / MySQL

安裝 Imagemagick

```
$ brew install imagemagick
```

安裝 MySQL

```
$ brew install mysql
$ unset TMPDIR
$ mysql_install_db --verbose --user=`whoami` --basedir="$(brew --prefix mysql)" -\
-datadir=/usr/local/var/mysql --   tmpdir=/tmp
$ mysql.server start
$ mysqladmin -u root password '123456'
$ mkdir -p ~/Library/LaunchAgents
$ find /usr/local/Cellar/mysql/ -name "homebrew.mxcl.mysql.plist" -exec cp {} ~/L\
ibrary/LaunchAgents/ \;
$ launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.mysql.plist
```

若無法順利進行安裝，可換下載 MySQL 官網 上的 Mac OS X ver. 10.6 (x86, 64-bit), DMG Archive 來安裝 MySQL。

安裝 RVM 與 Ruby 2.0

在建制 Rails 環境的時候，我們可能會有跑不同版本的 Ruby 或者不同的 getsemt 的需求。Ruby Version Manager 是一個能夠讓我們用很優雅的方式切換 Ruby 版本的工具。同時使用系統 Ruby，其實很容易弄髒環境和產生一些靈異現象的 bug。於是我們在建制環境時，通常第一時間就會裝起 RVM。

安裝 RVM

```
$ bash -s stable < <(curl -s https://raw.github.com/wayneeseguin/rvm/master/binsc\
ripts/rvm-installer)
$ . ~/.profile
$ source ~/.profile
```

安裝 Ruby 2.0

```
$ brew install libyaml
$ rvm pkg install openssl
$ rvm install 2.0.0 \
    --with-openssl-dir=$HOME/.rvm/usr \
    --verify-downloads 1
$ rvm use 2.0.0
```

注意事項

使用 RVM 安裝 gem 和 passenger-install-apache2-module 不需要加上 sudo , 因為使用 sudo 會使用非 RVM 的 ruby 環境, 安裝目錄也不一樣.)

安裝必要 Ruby gems

```
$ gem install rails --version 4.0.0
$ gem install mysql2
$ gem install capistrano
$ gem install capistrano-ext
```

設定 HTTP Server (使用 Pow)

使用 Pow 作為 HTTP Server

Pow 是 37 Signals open-source 出來的一套 Rack Server。其標榜的就是 Zero Config。

Pow 的原理原理是攔截 routing，導到 Pow 上。所以新增 project 不需要更改 /etc/hosts 就會生效。也因為 Pow 是 rack-based，支援 rack 的 framework 掛了就能跑。

相較起來，以往的 Passenger 搭配 Mac 本機端的 apache 的 solution 就顯得太笨重了。

Installation

Pow 的安裝相當簡單。

```
1 $ curl get.pow.cx | sh
```

即完成安裝。

Setting

Pow 預設的目錄是在 ~/.pow 下。

因此若要讓 project 跑在 Pow 之下。以我的 wiki 為例:

```
1 $ cd ~/.pow/
2 $ ln -s ~/projects/wiki
```

打開瀏覽器，輸入 http://wiki.dev 就完成了。以往的 http://localhost:3000/ 實在太噁心了，別再用它了！

小技巧

或者是直接在 project/wiki 下打 `powder link` 也可以。

使用 Powder 管理 Pow

Powder 是後來衍生出來的一套管理工具。

因為 Pow 的管理有點不易，所有有人寫了這個工具把一些常用的功能包裝起來。

安裝方法:

```
$ gem install powder
```

通常我只拿來做 `powder restart` 和 `powder log` 而已。

注意事項

若電腦預設非使用系統 Ruby 的開發者需注意此點。Pow 很可能會抓到系統 Ruby 及其 gemset 而無法啓動。我個人的解法是安裝 RVM 管控 Ruby，再在欲使用 Pow 之 project 目錄放置.rvmrc 即可。

.rvmrc 內容如下：

```
rvm 2.0.0
```

練習作業 0 - Hello World

作業目標

建立一個 Rails 專案，邁開第一步。

吸收觀念

建立一個 Rails 專案，實作第一個頁面 Hello World。

- Bundler
- Pow
- Rails 目錄結構

作業解答

安裝 Rails 4.0.0

```
gem install rails --version 4.0.0
```

打開 Terminal，在 ~/projects 下輸入指令，建立一個叫做 groupmy 的 Rails 專案

```
rails new groupmy
```

進入 groupmy 目錄

```
$ cd groupmy
```

此時，`rails new groupmy` 已經幫我們建立了一系列會用到的目錄檔案。

Bundler

在這裡我們要先岔開話題，先介紹幾個工具，第一個是 Bundler。

Bundler 是一套可以解決外部工具及其相依關係的好用工具，現在基本上所有以 Ruby 開發的工具，基本上都是使用這套工具管理專案上的套件相依關係。

接下來，我們要先使用 Bundler 安裝這個專案會用到的套件。

Bundler 依據 `Gemfile` 這個檔案安裝以及判斷套件相依性。你的 Rails 中的 Gemfile 內會是長這樣

```
1  source 'https://rubygems.org'
2
3  # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
4  gem 'rails', '4.0.0'
5
6  # Use sqlite3 as the database for Active Record
7  gem 'sqlite3'
8
9  # Use SCSS for stylesheets
10 gem 'sass-rails', '~> 4.0.0'
11
12 # Use Uglifier as compressor for JavaScript assets
13 gem 'uglifier', '>= 1.3.0'
14
15 # Use CoffeeScript for .js.coffee assets and views
16 gem 'coffee-rails', '~> 4.0.0'
17
18 # See https://github.com/sstephenson/execjs#readme for more supported runtimes
19 # gem 'therubyracer', platforms: :ruby
20
21 # Use jquery as the JavaScript library
22 gem 'jquery-rails'
23
24 # Turbolinks makes following links in your web application faster. Read more: htt\
25 ps://github.com/rails/turbolinks
26 gem 'turbolinks'
27
28 # Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
29 gem 'jbuilder', '~> 1.2'
30
31 group :doc do
32   # bundle exec rake doc:rails generates the API under doc/api.
```

```
33    gem 'sdoc', require: false
34  end
35
36  # Use ActiveModel has_secure_password
37  # gem 'bcrypt-ruby', '~> 3.0.0'
38
39  # Use unicorn as the app server
40  # gem 'unicorn'
41
42  # Use Capistrano for deployment
43  # gem 'capistrano', group: :development
44
45  # Use debugger
46  # gem 'debugger', group: [:development, :test]
```

常用 Bundler 指令：

bundle check : 檢查此專案的套件是否漏失

bundle install : 安裝此專案所需要的套件

我們先會使用 `bundle install` 這個指令確定這個專案裡面，套件都被安裝了。(套件必須都被安裝，才能啓動專案。)

Pow

下一個步驟是啓動專案，Rails 內建一個 Webserver，只要使用 `rails s` 就可以跑起來，這個專案會跑在 port 3000，打開 http://localhost:3000 就可以訪問。

但是在這本書裡面，我們推薦另外一種方式掛起我們的開發版本網站：Pow。

我們在上一章介紹過 Pow 這個軟體，使用 Pow 最大的好處是我們可以使用 http://groupmy.dev 這種網址掛起網站，而非使用 `rails s` 跑在 port 3000。

使用 `rails s` 的壞處在於，如果你更改了什麼設定需要改變的話，你必須要要 `Ctrl-C` 終止這個指令，然後再重新啓動一次，這個過程十分緩慢。

而使用 Pow，我們只要 `touch tmp/restart.txt` 就可以重啓 webserver，重新訪問被更新的網站了。

什麼時候需要重啓網站？

大多時候更改網站內容，不需要重開 Server。但是若更改 `config/routes.rb`、`config/enviroments/*.rb` 、`config/database.yml` 等等的檔案，都需要重開 Server（終止 Ctrl-C 再 `rails s`）。

Pow 與.rvmrc 與.powrc

我們推薦使用 `powder` 這個 gem 管理 Pow。安裝 powder 的方式是 `gem install powder`。我們可以用 `powder install` 這個指令把 Pow 裝起來。

再使用 `powder link`，把這個專案掛起來。比如你的專案目錄夾是 `groupmy/`，在這個目錄打 `powder link`，則建立的連結關係就會是 http://groupmy.dev/ 連結到 groupmy/ 這個專案。

我們還要再作幾件事讓 http://groupmy.dev/ 可以跑起來。

第一件事是開一個空的新檔案.rvmrc。在裡面置入：

```
rvm 2.0.0
```

第二件事是再開一個空的新檔案.powrc。在裡面置入：

```
1 if [ -f "$rvm_path/scripts/rvm" ] && [ -f ".rvmrc" ]; then
2   source "$rvm_path/scripts/rvm"
3   source ".rvmrc"
4 fi
```

這時候在終端機打 `powder open`，基本上應該就可以把專案開起來了。

網站開不起來怎麼辦？

試試這幾個指令:

rvm reload

bundle install

powder restart

touch tmp/restart.txt

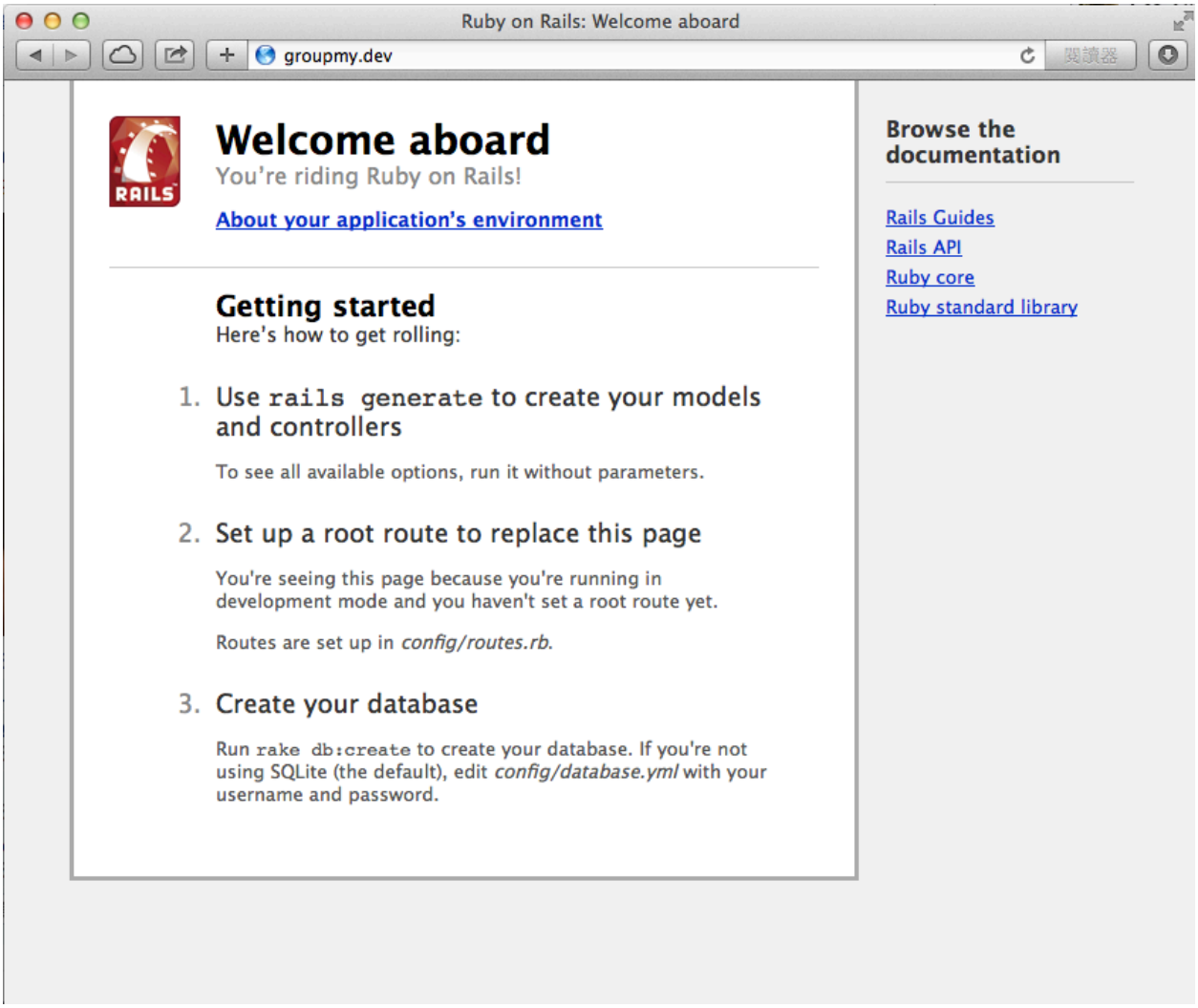
監視 development.log

`rails s` 的壞處是，只要設定檔案變更，就要結束掉重起。但它也有好處，就是可以直接看 Live Log，方便 Debug。那麼使用 Pow 之後，要怎麼樣繼續看 Log 去 debug 呢?

答案是：使用 `tail -f log/development.log` 這個技巧，持續追蹤 Log。

建立 Hello World 頁面

當首次打開 http://groupmy.dev/ ，映入眼簾的會是這樣一個畫面。



這是 Rails 預設的最初歡迎頁面。接下來我們要將根目錄顯示的預設畫面，換成自訂的頁面，顯示 “Hello World”。

而 Hello World 將會放在 pages 這個 controller 下的 wecleom 這個 action。

步驟 1: 產生 pages controller

```
[~/projects/groupmy] (master) $ rails g controller pages
  create  app/controllers/pages_controller.rb
  invoke  erb
  create    app/views/pages
  invoke  test_unit
  create    test/functional/pages_controller_test.rb
  invoke  helper
  create    app/helpers/pages_helper.rb
  invoke    test_unit
  create      test/unit/helpers/pages_helper_test.rb
```

步驟 2: 建立 welcome action

打開 app/controllers/pages_controller.rb，填入

```
class PagesController < ApplicationController
   def welcome
   end
end
```

步驟 3: 建立 welcome 的 HTML view

新增 app/views/pages/welcome.html.erb，填入

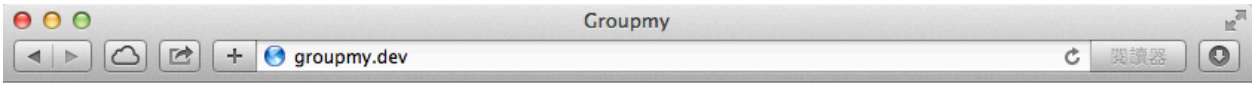
```
    Hello World!
```

步驟 4 :

設定 config/routes.rb 將 root 設到 pages#welcome 上。

```
    root :to => "pages#welcome"
```

現在你可以成功的見到了 Hello World!



Hello World!

Hello world

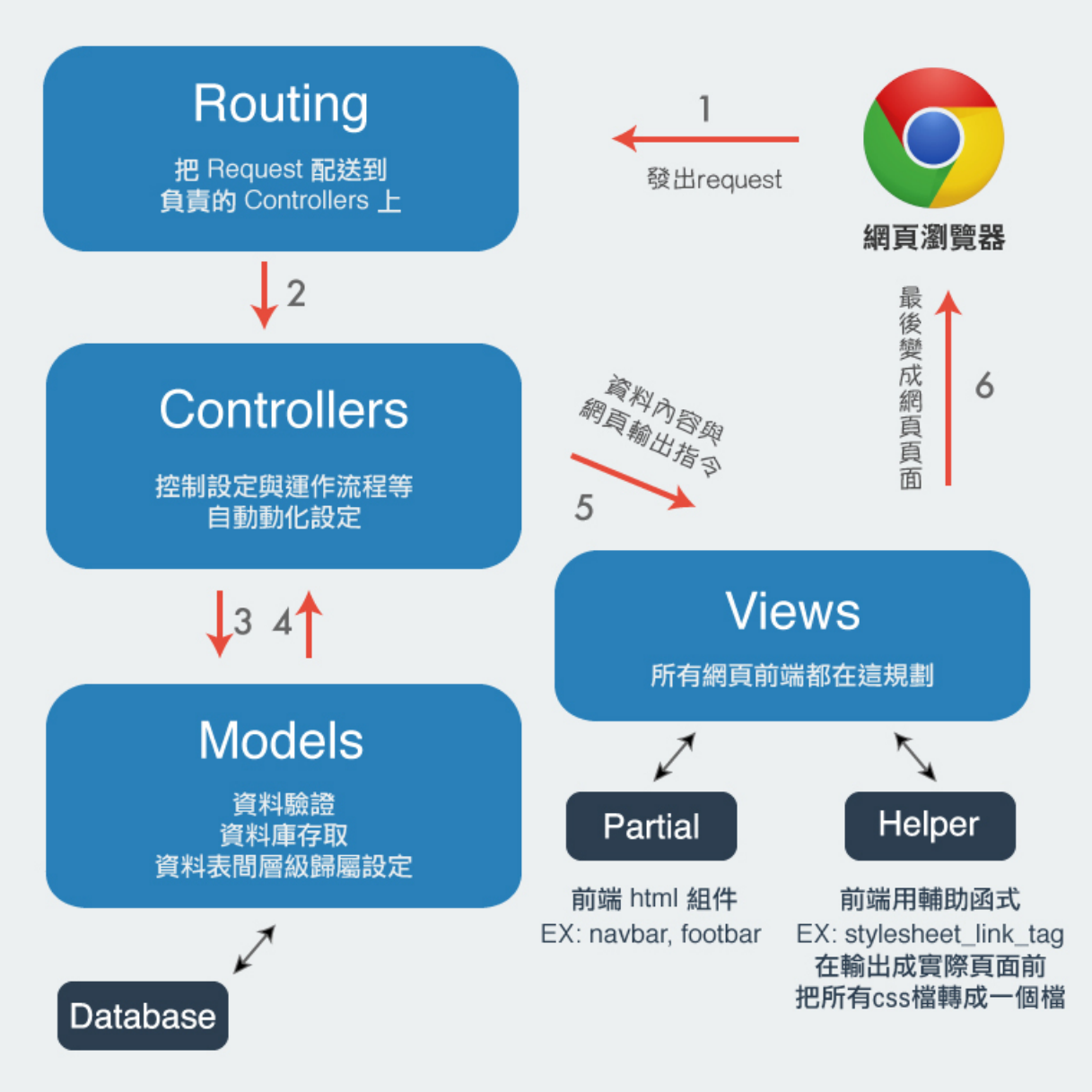
Rails 的 Routing

Rails 的路徑，是透過 `config/routes.rb` 下的設定進行 mapping。

常見出現的 routing 寫法，有以下幾種：

```
get "subscriptions/new"

resources :posts do
  resources :comments
end

namespace :admin do
  resources :posts
end

match '/search' => "search#index", :as => "search"
root :to => "pages#welcome"
```

關於它們的作用，我們會在後面的章節進行解釋。



Ruby on Rails 框架運作原理

練習作業 1 - 建立 Group - CRUD 與 RESTful

作業目標

本書的練習專案會是一個以 Group 為主的討論區

- 使用者可以建立、管理 Group。
- 使用者可以加入、退出社團。
- 使用者加入此社團後可以發表文章

開發一個簡易社群討論系統。系統要有 Group 與 Post 兩個 model，寫出 CRUD 介面，並且文章網址是使用 http://groupme.dev/group/1/post/2 這種表示。

本章練習主題

- 學會寫出 CRUD 七個 action 的 controller 與 view
- 學會利用 migration 新增資料庫欄位
- 學會撰寫「表單」
- 學會設定 Route
- 學會 resources 的設定（單層 resources)
- 對 Rails RESTful 有初步的理解
- 知道 before_action 使用的場景，並如何應用

Ch 1.0 CRUD

CRUD 指的是 Create(新增)、Read(讀取)、Update(更新)、Destroy(刪除) 四種操作資料的基本方式，這也是開發網站時幾乎大家最常寫到的四種功能。

Rails 在開發上極具優勢的其中一個原因，就是使用了 RESTful 的機制以及內建的 Convention，在實作 CRUD-like 功能時，隔外顯現開發速度上的優勢。

實作課題

本章將會實作以下課題

- 產生 Group 與 Post 這兩個 Model
- Group 需要有名稱 title, description
- Post 需要有文章標題文章內容 content
- 寫出 Group 的 CRUD controller 與 View
- 寫出 Posts 的 CRUD controller 與 View
- 在 routing 中對 Group 和 Posts 分別宣告它們都是 resources

Ch 1.0（補充）CRUD 懶人大法 Scaffold

Rails 裡面預設了相當 powerful 的一個指令：`scaffold`，字面上的意思是「鷹架」。

懶得寫 Rails 程式碼嗎？使用 rails g scaffold [MODEL]，可以快速的產出 model 及其 controller 內的 CRUD action 與 view，快速的寫出網站程式。

以下只是示範，請停住你的鍵盤不要跟著按。

scaffold 產生 Groups 與 Posts

以 1.1 內的作業題目為例，其實你可以這樣進行快速解

- Group 需要有名稱 name 與 description

自動產生 group scaffold

```
$ rails g scaffold group title:string description:text
```

- Post 需要有文章內容 content。

自動產生 post scaffold

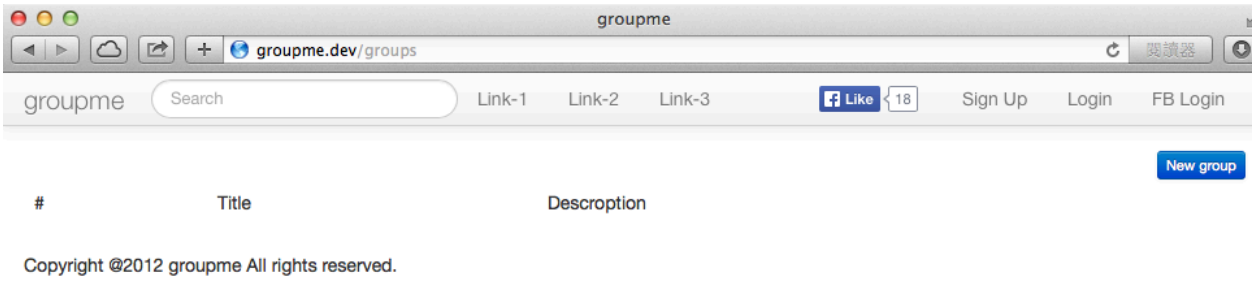
```
$ rails g scaffold post content:text
```

- 執行 db:migrate

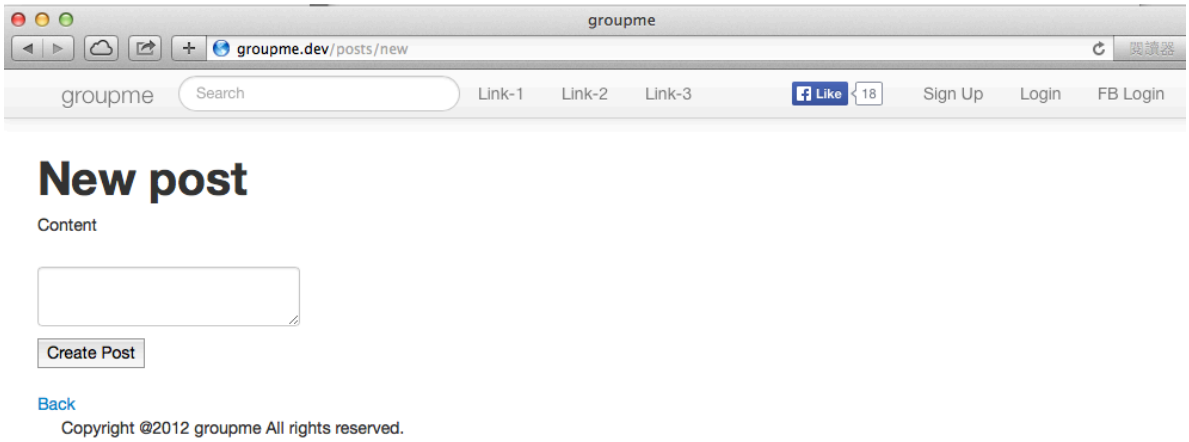
```
$ rake db:migrate
```

```
[~/projects/groupmy] (master) $ rake db:migrate
(in /Users/xdite/projects/groupmy)

==  CreateGroups: migrating ====================================================
-- create_table(:groups)
   -> 0.0013s
==  CreateGroups: migrated (0.0014s) ===========================================

==  CreatePosts: migrating ====================================================
-- create_table(:posts)
   -> 0.0012s
==  CreatePosts: migrated (0.0013s) ============================================
```

打開 http://groupme.dev/groups



打開 http://groupme.dev/posts/new



Scaffold 作了哪些事？

- 自動產生 groups 與 post 的 model 及其 migration
- 自動產生 groups 與 posts 的 CRUD 之 controller / action / view
- 自動在 config/routes 裡加上 resources :groups 與 resources :posts

What is「db migration」?

在傳統開發 web 應用程式的流程中，開發者多半是使用 phpmyadmin 開設 db 欄位。

因此在學習 Rails 之初，剛入門的開發者往往會提出一個質疑：為何必須撰寫 db migration 檔去生欄位，而不是使用傳的 phpmyadmin?

理由是 db migration 是一個經過實證的最佳解。當多人同時在一個 project 內工作時，縱然程式碼可以受到版本控制，但是 db schema 卻可能是無法透過版本控制的一個隱憂。

當多人都可以自由的刪改 db 欄位不受監督時，程式碼就很難與 db 欄位有著一致性，將會有出錯的可能性。為了 automation 以及也能對 db schema 做版本控制，所以才有了 db migration file 的設計。

小結

至此，我們擁有了 group 與 post 兩份的 CRUD。

警告

在新手的學習過程中，我並不推薦使用 `scaffold` 這個指令，因為 CRUD 及其背後 convention 的原理是基本中的基本，「應」熟練再熟練。寫個十幾遍都不為過，練到閉著眼睛都會寫才行。

然而新手在練習作業中，很容易不小心就忘記 CRUD 的寫法。故此章先行提示 Rails 內建自動 CRUD 的指令 `scaffold`，但我建議日常開發中不要使用這個指令。

Ch 1.0（補充）建站懶人包 Bootstrappers

Bootstrappers 是我在 2012 年開發的一個懶人架站機。

它的特點是內建 Boostrap 這個 Theme、一些常用熱門 Gem，還有一些 Rails Best Practice。可以讓你在寫網站的一剛開始就加速好幾倍...

改用 `bootstrappers` 生專案。

安裝 bootstrappers

```
1  gem install bootstrappers -v=4.2.1
```

用 bootstrappers 建立新專案

```
1  bootstrappers groupme
```

在這裡我建議你用 Bootstrappers 重新建立一個專案，再循 Chapter 1.1 的方式，把 `groupme.dev` 用 Pow 掛起來，因為本書接下來都會以 Bootstrappers 生成的專案作為示範。

Ch 1.1 建立 Group

建立 Group 這個 model

執行

```
rails g model group title:string description:text
```

```
      invoke  active_record
      create    db/migrate/20130529180541_create_groups.rb
      create    app/models/group.rb
      invoke    test_unit
      create      test/models/group_test.rb
      create      test/fixtures/groups.yml
```

生成 Group 這個 model。

然後跑 `rake db:migrate`。

```
==  CreateGroups: migrating ====================================================
-- create_table(:groups)
   -> 0.1951s
==  CreateGroups: migrated (0.1952s) ===========================================
```

建立 groups 這個 controller

執行 `rails g controller groups`

```
      create  app/controllers/groups_controller.rb
      invoke  erb
      create    app/views/groups
      invoke  test_unit
      create    test/controllers/groups_controller_test.rb
      invoke  helper
      create    app/helpers/groups_helper.rb
      invoke    test_unit
      create      test/helpers/groups_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/groups.js.coffee
      invoke    scss
      create      app/assets/stylesheets/groups.css.scss
```

把 groups 加入 routing

到 `config/routes.rb` 加入

```
resources :groups
```

Ch 1.1 (補充) RESTful

Rails 在 1.2 版本的時候引進了 RESTful 這一套設計風格。而在 2.0 版强迫正式成為開發預設值。

說到 RESTful，幾乎每個剛踏進 Rails 框架中的開發者都皺起眉頭。幾乎沒有人能夠在初學階段搞的懂這是什麼玩意。

偏偏初學者往往有一個「死扣細節」的習慣：沒「弄懂」就「不敢用」，結果在第一個關卡挫折感就堆的如山高，把玩 RESTful 後打退堂鼓，從此放棄 Rails。

想學 Rails 就先把 RESTful 背起來

RESTful 是 Rails 裡面使用的最基礎最頻繁的一個設計手法，偏偏它也是最難一次講解清楚讓剛入門者理解的一個主題。因為在傳統 web 應用程式開發流程中，根本沒有這樣的概念。更別提在市面上這麼多網頁框架，只有 Rails 將之視為預設值。

別說是初學者，就連筆者和一些 Rails 界的前輩，在 Rails 剛納入 RESTful 為預設風格時，也沒能通透其中原理。

但是不熟練 RESTful 的開發架構，在開發 Rails 時幾乎會寸步難行。在訓練新 Developer 時如何讓他短時間就熟練便上手呢?

方法很簡單：强迫他背起來，然後重複寫十遍。

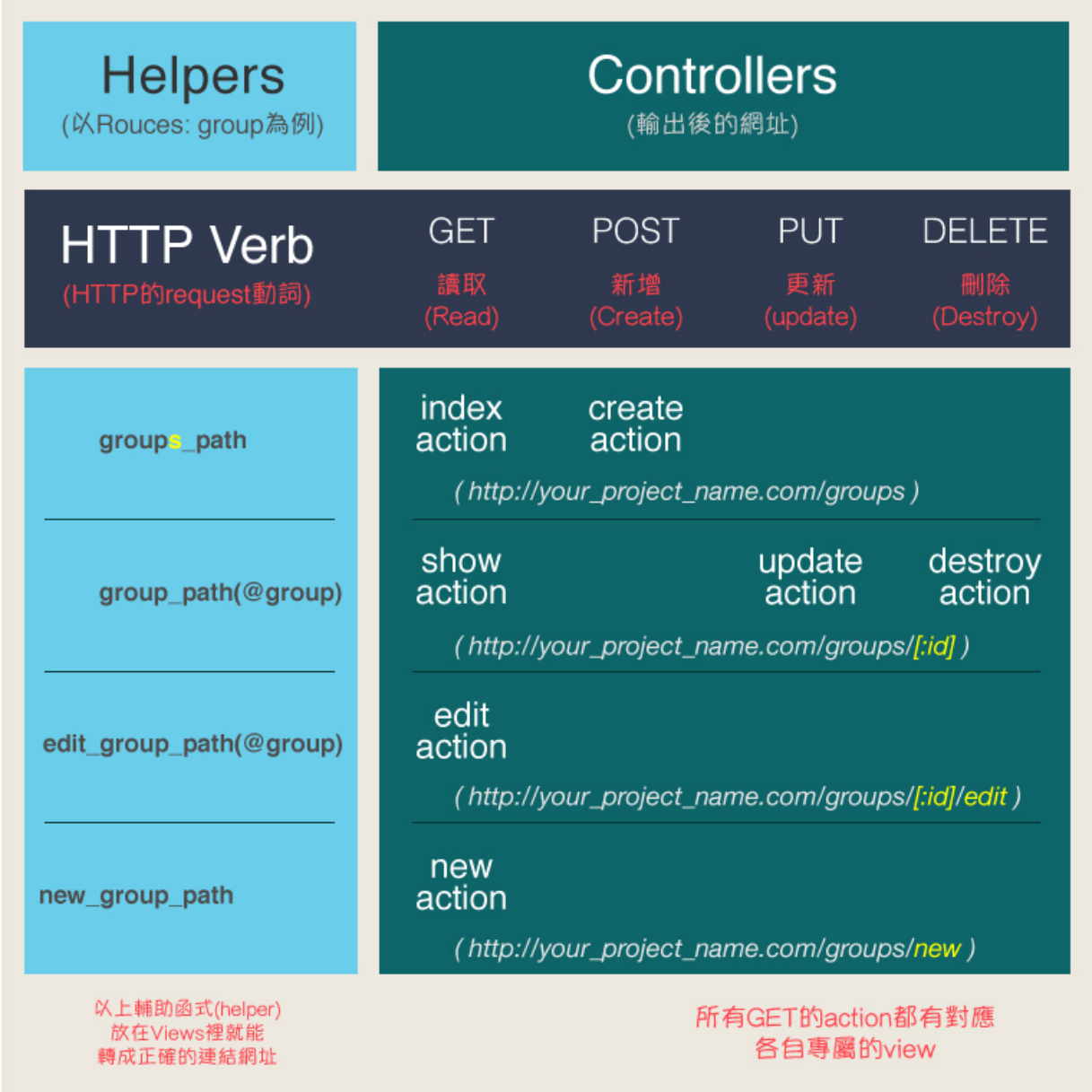
聽起來很唬爛，但其實真的紮紮實實寫過十遍以後。這時候再重新講解一次 RESTful 的概念，原本模糊的概念一下就變得豁然開朗了。

初步熟悉使用 Rails RESTful

在 config/routes.rb 內加入 resources

在 config/routes.rb 加入 `resources :groups`，就是具體在 rails 對一個 controller 實作 RESTful 的方式（對一個 controller 宣告成為 resources)。

```
1 Groupme::Application.routes.draw do
2   resources :groups
3 end
```



RESTful 路徑對照表

Ch 1.1.1 建立 Groups Controller 裡的 index

到 app/controllers/groups_controller.rb 加入

```
def index
  @groups = Group.all
end
```

補上 **view**：

```
touch app/views/groups/index.html.erb
```

```
<div class="span12">
  <div class="group">
    <%= link_to("New group", new_group_path , :class => "btn btn-mini btn-primary\
 pull-right" ) %>
  </div>
  <table class="table">
    <thead>
      <tr>
        <td> # </td>
        <td> Title </td>
        <td> Description </td>
      </tr>
    </thead>

    <tbody>
      <% @groups.each do |group| %>
      <tr>
        <td> # </td>
        <td> <%= link_to(group.title, group_path(group)) %>  </td>
        <td> <%= group.description %>  </td>
        <td> <%= link_to("Edit", edit_group_path(group), :class => "btn btn-mini"\
 ) %>
         <%= link_to("Delete", group_path(group), :class => "btn btn-mini", :meth\
od => :delete, :confirm => "Are you sure?" ) %>
       </td>
     </tr>
     <% end %>
   </tbody>
 </table>

</div>
```

解說

通常會放置列表。因此 Group.all 是拉出 group 這個 model 所有的資料。

```
1   def index
2     @groups = Group.all
3   end
```

Ch 1.1.2 建立 Groups Controller 裡的 show

在 app/controllers/groups_controller.rb 加入 show 這個 action

```
  def show
    @group = Group.find(params[:id])
  end
```

補上 view

touch app/views/groups/show.html.erb

加入

```
<div class="span12">

  <div class="group">
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-mini btn-pri\
mary pull-right" ) %>
  </div>

  <h2> <%= @group.title %> </h2>

  <p> <%= @group.description %> </p>

</div>
```

解說

秀出單筆資料。Group.find(123) 是指找 Post model 裡 id 為 123 的資料。http://groupme.dev/groups/123 的 groups 是 controller、show 是 action；如果在 route 裡面沒有特別指定，則 123 通常就是 params[:id] 。

```
1  def show
2    @group = Group.find(params[:id])
3  end
```

Ch 1.1.3 建立 Groups Controller 裡的 new

在 app/controllers/groups_controller.rb 加入 new 這個 action

```
  def new
    @group = Group.new
  end
```

補上 view

touch app/views/groups/new.html.erb

加入

```
<div class="span12">

 <%= simple_form_for @group  do |f| %>
    <%= f.input :title, :input_html => { :class => "input-xxlarge"} %>
    <%= f.input :description, :input_html => { :class => "input-xxlarge"} %>

    <div class="form-actions">
       <%= f.submit "Submit", :disable_with => 'Submiting...', :class => "btn btn\
-primary" %>
    </div>
  <% end %>

</div>
```

解說

initial 一個新的 Post object。

```
1   def new
2     @group = Group.new
3   end
```

Ch 1.1.4 建立 Groups Controller 裡的 create

在 app/controllers/groups_controller.rb 加入 create 這個 action

```
  def create
    @group = Group.new(params[:group])

    if @group.save
      redirect_to groups_path
    else
      render :new
    end
  end
```

不過此時，我們發現好像少了一種實際狀況？如果一個 Group 沒有 title，應該算是不合法的 group 吧？

我們應該要限制沒有輸入 title 的 group，必須要退回到 new 這個表單。

修改 app/models/group.rb，限制 group 一定要有標題

```
class Group < ActiveRecord::Base

  validates :title, :presence => true

end
```

同時修改 app/controllers/groups_controller.rb 中 create 這個 action

改成以下內容

```
  def create
    @group = Group.new(group_params)
    @group.save

    redirect_to groups_path
  end

  private


  def group_params
    params.require(:group).permit(:title, :description)
  end
```

解說

這邊要搭配 app/views/groups/new.html.erb 這個 view 並列一起看。

```
<div class="span12">

 <%= simple_form_for @group  do |f| %>
    <%= f.input :title, :input_html => { :class => "input-xxlarge"} %>
    <%= f.input :description, :input_html => { :class => "input-xxlarge"} %>

    <div class="form-actions">
       <%= f.submit "Submit", :disable_with => 'Submiting...', :class => "btn btn\
-primary" %>
    </div>
  <% end %>

</div>
```

在 RESTful Rails 的寫法中，對 groups_path 丟 POST 就是對應到 create 的動作。而 Rails 在設計上，form 是綁 model 的，因此整個 form 的內容會被包成一個 hash，在這裡就是 params[:group]。create action 初始一個 object，並把 params[:group] 整包塞進這個 object 裡。如果 @group 能夠成功的儲存，就「重導」到 index action，失敗則「退回」到 new action。

Ch 1.1.4 (補充) Strong Parameters

還記得 2012 年初開發圈曾經騰動一時的：Github 被入侵事件 嗎?

事件的起因是因為 Rails 內建的 mass assignment，無法容易保護資料的安全性。而原先內建的 attr_accessible / attr_protected 的設計並不足夠實務使用（無法强制 Developer 使用）。

在該事件發生後，Rails 核心團隊在 3.2.3 之後的版本，都開啓了 config.whitelist_attributes = true 的選項作為預設。也就是專案自動會對所有的 model 都自動開啓白名單模式，你必須手動對每一個 model 都加上 attr_accessible。這樣表單送值才會有辦法運作。

此舉好處是:「夠安全」，能强迫開發者在設計表單時記得審核 model 該欄位是否適用於 mass-assign。但這樣的機制也引發開發者「不實用」「找麻煩」的議論。

臨時 Hack 找麻煩

首先會遇到的第一個問題是:「新手容易踩中地雷」

首先最麻煩的當然是，新手會被這一行設定整到。新手不知道此機制為何而來，出了問題也不知道如何關掉這個設定。更麻煩的是撰寫新手教學的人，必須又花上一大篇幅解釋 mass-assignment 的設計機制，為何重要，為何新手需要重視⋯etc.

第二個問題:「不是很實用」

手動一個一個加上 attr_accessible 真的很煩人，因為這也表示，若新增一個欄位，開發者也要手動去加上 attr_accessible，否則很可能在某些表單直接出現異常現象。

而最麻煩的還是，其實 attr_accessible 不敷使用，因為一個系統通常存在不只一種角色，普通使用與 Admin 需要的 mass-assignment 範圍絕對不盡相同。

雖然 Rails 在 3.1 加入了 scoped mass assignment。但這也只能算是 model 方面的解決手法。一旦系統內有更多其他流程需求，scoped mass assignment 的設計頓時就不夠解決問題了⋯

癥結點：欄位核准與否應該由 controller 管理，而非 model

大家戰了一陣子，終於收斂出一個結論。一切的癥結點在於之前的設計想法都走錯了方向，欄位核准與否應該由 controller 決定。因為「流程需求」本來就應該作在 controller 裡面。新的 solution strong_parameters 就是這個解法。

在當時：Rails 之父 DHH 提供了他的最佳實務：

```
class PostsController < ActionController::Base
  def create
    Post.create(post_params)
  end

  def update
    Post.find(params[:id]).update_attributes!(post_params)
  end

  private
    def post_params
      params[:post].slice(:title, :content)
    end
end
```

使用 slice 去把真正需要的部分切出來，所以就算 hacker 打算送其他 parameter 也會被過濾掉 (不會有 exception)。

Strong Parameters 的作法

而 strong_parameters 的作法是必須過一段 permit，允許欄位。如果送不允許的欄位進來，會 throw exception。

```
class PeopleController < ActionController::Base
  def update
    person.update_attributes!(person_params)
    redirect_to :back
  end

  private
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

進階 Strong Parameters

當然，每一段 controller 都要來上這麼一段，有時候也挺煩人的。Railscast 也整理了一些進階招數:

- Nested Attributes
- Orgngized to Class

Ch 1.1.5 建立 Groups Controller 裡的 edit

在 app/controllers/groups_controller.rb 加入 edit 這個 action

```
  def edit
    @group = Group.find(params[:id])
  end
```

補上 view

touch app/views/groups/edit.html.erb

加入

```
<div class="span12">

 <%= simple_form_for @group  do |f| %>
    <%= f.input :title, :input_html => { :class => "input-xxlarge"} %>
    <%= f.input :description, :input_html => { :class => "input-xxlarge"} %>

    <div class="form-actions">
       <%= f.submit "Submit", :disable_with => 'Submiting...', :class => "btn btn\
-primary" %>
    </div>
  <% end %>

</div>
```

解說

query 出指定的 Group model object，然後進行編輯。

```
1   def edit
2     @group = Group.find(params[:id])
3   end
```

Ch 1.1.6 建立 Groups Controller 裡的 update

在 app/controllers/groups_controller.rb 加入 update 這個 action

```
  def update
    @group = Group.find(params[:id])

    if @group.update(group_params)
      redirect_to group_path(@group)
    else
      render :edit
    end
  end
```

解說

這邊也要翻回 app/views/posts/edit.html.erb 這個 view 一起來看。

```
<div class="span12">

 <%= simple_form_for @group  do |f| %>
    <%= f.input :title, :input_html => { :class => "input-xxlarge"} %>
    <%= f.input :description, :input_html => { :class => "input-xxlarge"} %>

    <div class="form-actions">
       <%= f.submit "Submit", :disable_with => 'Submiting...', :class => "btn btn\
-primary" %>
    </div>
  <% end %>

</div>
```

在 RESTful Rails 的寫法中，對 group_path 丟 PUT 就是對應到 update 的動作。form 會背包成一個 hash，如果 @group 能夠吃進 params[:group] 進行更新且成功儲存，就會「重導」到 show action，失敗則「退回」到 edit action。

Ch 1.1.7 建立 Groups Controller 裡的 destroy

在 app/controllers/groups_controller.rb 加入 destroy 這個 action

```
1   def destroy
2     @group = Group.find(params[:id])
3
4     @group.destroy
5
6     redirect_to groups_path
7   end
```

至此，我們完成了 Group 的 CRUD。

把首頁改成 Group 的 Index 頁

最後修改 config/routes.rb，把 root 指向 group 的 index 頁

```
root :to => "groups#index"
```

解說

這邊要翻回 app/views/posts/index.html.erb 這個 view 一起來看。

```
      <tr>
        <td> # </td>
        <td> <%= link_to(group.title, group_path(group)) %>  </td>
        <td> <%= group.description %>  </td>
        <td> <%= link_to("Edit", edit_group_path(group), :class => "btn btn-mini"\
 ) %>
         <%= link_to("Delete", group_path(group), :class => "btn btn-mini", :meth\
od => :delete, :confirm => "Are you sure?" ) %>
       </td>
     </tr>
```

找到該筆資料並刪除之。在 RESTful Rails 的寫法中，對 group_path 丟一個 DELETE，就會對應到 destroy 的動作。可看一下 link_to "Destroy" 那一行。

補充章節：RESTful on Rails

Representational State Transfer，簡稱 REST。是 Roy Fielding 博士在 2000 年他的博士論文中提出來的一種軟體架構風格。目前是一種相當風行的 Web Services 實現手法，因為 REST 風格的 Web Services 遠比傳統的 SOAP 與 XML-RPC 來的簡潔。在近年，幾乎所有各大主流網站的 API 都已採用此種風格進行設計。

What is REST?

REST 提出了一些設計概念和準則:

1. 網路上的所有事物都被將被抽象成資源 (resource)
2. 每個資源對應一個唯一的 resource identifier
3. 通過通用的介面 (generic connector interface) 對資源進行操作
4. 對資源的各種操作不會改變 resource idetifier
5. 所有的操作都是無態 (stateless) 的

對照到 web services 上來說:

- resource indentifier 是 URI
- generic connector interface 是 HTTP

RESTful Web Services

RESTful Web Services 是使用 HTTP 並遵循 REST 設計原則的 Web Services。它從以下三個方面資源進行定義:

- URI，比如：http://example.com/resources/。
- Web Services accept 與 return 的 media type，如：JSON，XML ， YAML 等。
- Web Services 在該 resources 所支援的一系列 request methid : 如：POST，GET，PUT 或 DELETE。

以下列出在實現 RESTful Web 服務時 HTTP 請求方法的典型用途。

GET

資源	GET
一組資源的 URI，比如 http://example.com/resources/	使用給定的一組資源替換當前整組資源。
單個資源的 URI，比如 http://example.com/resources/142	獲取指定的資源的詳細信息，格式可以自選一個合適的 media type（如：XML、JSON 等）

PUT

資源	PUT
一組資源的 URI，比如 http://example.com/resources/	列出 URI，以及該資源組中每個資源的詳細信息（後者可選）。
單個資源的 URI，比如 http://example.com/resources/142	替換/創建指定的資源。並將其追加到相應的資源組中。

POST

資源	POST
一組資源的 URI，比如 http://example.com/resources/	在本組資源中創建/追加一個新的資源。該操作往往返回新資源的 URL。
單個資源的 URI，比如 http://example.com/resources/142	把指定的資源當做一個資源組，並在其下創建/追加一個新的元素，使其隸屬於當前資源。

DELETE

資源	DELETE
一組資源的 URI，比如 http://example.com/resources/	刪除整組資源。
單個資源的 URI，比如 http://example.com/resources/142	刪除指定的元素。

制約即解放

DHH 在 Discovering a world of Resources on Rails 提到一個核心概念：Constraints are liberating 。

很多剛踏入 Rails 這個生態圈的開發者，對於 REST 總有股强烈的反抗心態，認為 REST 是一個討厭的限制。但事實上，DHH 卻認為引入 REST 的制約卻反為 Rais 開發帶來了更大的解放，而這也是他引入這個設計的初衷。

維護性：解決了程式碼上的風格不一

在傳統的開發方式中，對於有效組織網頁應用中的程式碼，大家並沒有什麼共識。所以很可能在專案中會出現這種風格不一致的程式碼:

```
def add_friend
end

def remove_friend
end

def create_post
end

def delete_post
end
```

透過 REST 的包裝（對於 resource 的操作），可以變得更簡潔直觀，且具有統一的寫法。

```
class FriendShipController
   def create
   end

   def destroy
   end
end

class PostController
  def create
  end

  def destroy
  end
end
```

介面統一：Action as Resource

大眾最常對 REST 產生的一個誤解，就是以為 resource 只有指的是 data。其實 resource 指的是：data + representation (表現形式，如 html, xml, json 等)。

在網頁開發中，網站所需要的表現格式，不只有 HTML 而已，有時候也需要提供 json 或者 xml。Rails 透過 responder 實現了

在 Resource Oriented Architecture (ROA，一組 REST 架構實現 guideline) 中有一條：`A resource can use file extension in the URI, instead of Content-Type negotiation` 。

Rails 透過 responder 的設計，讓一個 action 可以以 file extension (.json, .xml, .csv ...etc.) 的形式，提供不同類型的 representation。

```
1  class PostsController < ApplicationController
2    # GET /posts
3    # GET /posts.xml
4    def index
5      @posts = Post.all
6
7      respond_to do |format|
8        format.html # index.html.erb
9        format.xml  { render :xml => @posts }
10     end
11   end
```

開發速度：透過 CRUD 七個 action + HTTP 四個 verb + 表單 Helper 與 URL Helper 達到高速開發

REST 之所以能簡化開發，是因為其所引入的架構約束。Rails 中的 REST implementation 將 controller 的 method 限制在七個：

- index
- show
- new
- edit
- create
- update
- destroy

實際上就是整個 CRUD。而在實務上，web services 的需要的操作行為，其實也不脫這七種。Rails 透過 HTTP 作為 generic connector interface，使用 HTTP 的四種 verb：GET、POST、PUT、DELETE 對資源進行操作。

GET

```
<%= link_to("List", posts_path) %>
<%= link_to("Show", post_path(post)) %>
<%= link_to("New", new_post_path) %>
<%= link_to("Edit", edit_post_path(post)) %>
```

POST

```
  <%= form_for @post , :url => posts_path , :html => {:method => :post} do |f| %>
```

PUT

```
  <%= form_for @post , :url => post_path(@post) , :html => {:method => :put} do |\
f| %>
```

Destroy

```
  <%= link_to("Destroy", post_path(@post), :method => :delete )
```

Form 綁定 Model Attribute 的設計

Rails 的表單欄位，是對應 Model Attribute 的：

```
<h1>New post</h1>

<%= form_for @post , :url => posts_path do |f| %>
    <%= f.error_messages %>
    <div><label>subject</label><%= f.text_field :subject %></div>
    <div><label>content</label><%= f.text_area :content %> </div>
    <%= f.submit "Submit", :disable_with => 'Submiting...' %>
<% end -%>


<%= link_to 'Back', posts_path %>
```

透過 form_for 傳送出來的表單，會被壓縮包裝成一個 parameter: params[:post]，

```
def create
  @post = Post.new(params[:post])
  if @post.save
    flash[:notice] = 'Post was successfully created.'
    redirect_to post_path(@post)
  else
    render :action => "new"
  end
end
```

如此一來，原本創造新資源的一連串繁複動作就可以大幅的被簡化，開發速度達到令人驚艷的地步。

參考資料：

- 百度百科：REST
- Wikipedia: REST
- Wikipedia: ROA
- DHH: Discovering a world of Resource on Rails
- ihower: Rails RESTful 制約即解放
- ihower: 什麼是 REST 與 RESTful

補充章節：More about RESTful on Rails

有時候開發中也會出現超過這七種 action 之外的動作。那要如何整合進 RESTful 的 route 中呢?

Collection & Member

Member

宣告這個動作是屬於單個資源的 URI。可以透過 `/photos/1/preview` 進行 GET。有 `preview_photo_path(photo)` 或 `preview_photo_url(photo)` 這樣的 Url Helper 可以用。

```
resources :photos do
  member do
    get 'preview'
  end
end
```

也可以使用這種寫法:

```
resources :photos do
  get 'preview', :on => :member
end
```

Collection

宣告這個動作是屬於一組資源的 URI。可以透過 `/photos/search` 進行 GET。有 `search_photos_path` 或 `search_photos_url` 這樣的 Url Helper 可以用。

```
resources :photos do
  collection do
    get 'search'
  end
end
```

也可以使用這種寫法:

```
resources :photos do
  get 'search', :on => :collection
end
```

Nested Resources

有時候我們會需要使用雙重 resources 來表示一組資源。比如說 Post 與 Comment :

```
resources :posts do
  resources :comments
end
```

可以透過 `/posts/123/comments` 進行 GET。有 `post_comments_path(post)` 或 `post_comments_url(post)` 這樣的 Url Helper 可以用。

而這樣的 URL 也很清楚的表明，這組資源就是用來存取編號 123 的 post 下所有的 comments。

而要拿取 Post 的 id 可以透過這樣的方式取得:

```
class CommentsController < ApplicationController

  before_filter :find_post

  def index
    @comments = @post.comments
  end

  protected

  def find_post
    @post = Post.find(params[:post_id])
  end
end
```

Namespace Resources

但是像 /admin/posts/1/edit 這種 URL，用 Nested Resources 應該造不出來吧？沒錯，這樣的 URL 應該要使用 Namespace 去建構：

```
1       namespace :admin do
2         # Directs /admin/products/* to Admin::PostsController
3         # (app/controllers/admin/posts_controller.rb)
4         resources :posts
5       end
```

可以透過 `/admin/posts/123/edit` 進行 GET。有 `edit_admin_post_path(post)` 或 `edit_admin_post_url(post)` 這樣的 Url Helper 可以用。

延伸閱讀

- Rails Guide: Rails Routing from the Outside In

練習作業 2 - 在 Group 裡面發表文章 - 雙層 RESTFul

上一章我們完成了 Group 的 CRUD，這一章的練習目標是在 Group 裡面發表文章。並且文章網址是使用 http://groupme.dev/group/1/post/2 這種網址表示。

本章練習主題

- 學會 has_many, belongs_to
- 學會 resources 的設定（雙層 resources)
- 使用 before_action 整理程式碼

Ch 2.1 建立 Post

建立 Post 這個 model

執行

```
rails g model post content:text group_id:integer
```

```
      invoke  active_record
      create    db/migrate/20130529200707_create_posts.rb
      create    app/models/post.rb
      invoke    test_unit
      create      test/models/post_test.rb
      create      test/fixtures/posts.yml
```

生成 Post 這個 model。

然後跑 `rake db:migrate`。

```
==  CreatePosts: migrating ====================================================
-- create_table(:posts)
   -> 0.1297s
==  CreatePosts: migrated (0.1298s) ===========================================
```

建立 posts 這個 controller

執行 `rails g controller posts`

```
      create  app/controllers/posts_controller.rb
      invoke  erb
      create    app/views/posts
      invoke  test_unit
      create    test/controllers/posts_controller_test.rb
      invoke  helper
      create    app/helpers/posts_helper.rb
      invoke    test_unit
      create      test/helpers/posts_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/posts.js.coffee
      invoke    scss
      create      app/assets/stylesheets/posts.css.scss
```

把 posts 加入 routing

到 `config/routes.rb` 修改原有的

```
  resources :groups
```

變成

```
  resources :groups do
    resources :posts
  end
```

加入 Group 與 Post 的關聯

修改 app/models/group.rb 加入 has_many :posts

```
class Group < ActiveRecord::Base

  has_many :posts
  validates :title, :presence => true

end
```

修改 app/models/post.rb 加入 belongs_to :group

```
class Post < ActiveRecord::Base
  belongs_to :group
end
```

Ch 2.1.1 在 Groups controller 的 show 裡面撈出相關的 Post

到 app/controllers/groups_controller.rb，將 show 修改成以下內容

```
def show
  @group = Group.params[:id]
  @posts = @group.posts
end
```

補上 view：

修改 app/views/groups/show.html.erb

```
<div class="span12">
  <div class="group pull-right">
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-mini ")%>

    <%= link_to("New Post", new_group_post_path(@group) , :class => "btn btn-mini\
 btn-primary")%>
         
  </div>
  <h2> <%= @group.title %> </h2>
  <p> <%= @group.description %> </p>


  <table class="table">

  <tbody>
    <% @posts.each do |post| %>
    <tr>
      <td> <%= post.content %> </td>
      <td> <%= link_to("Edit", edit_group_post_path(post.group, post), :class => \
"btn btn-mini")%>
        <%= link_to("Delete", group_post_path(post.group, post), :class => "btn b\
tn-mini", :method => :delete, :confirm => "Are you sure?" ) %> </td>
      </tr>
      <% end %>
    </tbody>
  </table>
</div>
```

Ch 2.1.2 建立 Posts Controller 裡的 new

在 app/controllers/posts_controller.rb 加入 new 這個 action

```
  def new
    @group = Group.find(params[:group_id])
    @post = @group.posts.build
  end
```

補上 view

touch app/views/posts/new.html.erb

加入

```
<div class="span12">

 <%= simple_form_for [@group,@post]  do |f| %>
    <%= f.input :content, :input_html => { :class => "input-xxlarge"} %>

    <div class="form-actions">
       <%= f.submit "Submit", :disable_with => 'Submiting...', :class => "btn btn\
-primary" %>
    </div>
  <% end %>

</div>
```

Ch 2.1.3 建立 Posts Controller 裡的 create

在 app/controllers/posts_controller.rb 加入 create 這個 action

```
  def create
    @group = Group.find(params[:group_id])
    @post = @group.posts.new(post_params)

    if @post.save
      redirect_to group_path(@group)
    else
      render :new
    end
  end

  private


  def post_params
    params.require(:post).permit(:content)
  end
```

不過此時，我們發現 Post 其實也需要被驗證。如果一個 Post 沒有 content，應該也算是不合法的 post?

我們應該要限制沒有輸入 content 的 post，必須要退回到 new 這個表單。

修改 app/models/post.rb，限制 post 一定要有內容

```
class Post < ActiveRecord::Base

  belongs_to :group
  validates :content, :presence => true

end
```

解說

Rails 的 ORM 內建一系列 Validation 的 API，可以幫助 Developer 快速驗證資料的類型與格式。

http://edgeguides.rubyonrails.org/active_record_validations.html

Ch 2.1.4 建立 Posts Controller 裡的 edit

在 app/controllers/groups_controller.rb 加入 edit 這個 action

```
  def edit
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])
  end
```

補上 view

touch app/views/posts/edit.html.erb

```
<div class="span12">

 <%= simple_form_for [@group,@post]  do |f| %>
    <%= f.input :content, :input_html => { :class => "input-xxlarge"} %>

    <div class="form-actions">
       <%= f.submit "Submit", :disable_with => 'Submiting...', :class => "btn btn\
-primary" %>
    </div>
  <% end %>

</div>
```

Ch 2.1.5 建立 Posts Controller 裡的 update

在 app/controllers/postss_controller.rb 加入 update 這個 action

```
1   def update
2     @group = Group.find(params[:group_id])
3     @post = @group.posts.find(params[:id])
4
5     if @post.update(post_params)
6       redirect_to group_path(@group)
7     else
8       render :edit
9     end
10  end
```

Ch 2.1.6 建立 Posts Controller 裡的 destroy

在 app/controllers/groups_controller.rb 加入 destroy 這個 action

```
1   def destroy
2     @group = Group.find(params[:group_id])
3     @post = @group.posts.find(params[:id])
4
5     @post.destroy
6
7     redirect_to group_path(@group)
8   end
```

至此，我們完成了 Post 的 CRUD。

Ch 2.1.7 以 before_action 整理重複的程式碼

有沒有覺得 Posts 裡的每個 action 裡面都有一行

```
1     @group = Group.find(params[:group_id])
```

很冗餘呢?
我們可以利用 before_action 這個技巧，把重複的程式碼去掉:
首先在 private 底下，新增一個 find_group method

```
1   def find_group
2     @group = Group.find(params[:group_id])
3   end
```

然後在 class PostsController < ApplicationController 下，加一行

```
1   before_action :find_group
```

再把每個 action 的這一行砍掉

```
1     @group = Group.find(params[:group_id])
```

最後的成品會長這樣

```
1  class PostsController < ApplicationController
2
3    before_action :find_group
4
5    def new
6      @post = @group.posts.build
7    end
8
9    def create
10
11     @post = @group.posts.new(post_params)
12
13     if @post.save
14       redirect_to group_path(@group)
15     else
```

```
16        render :new
17      end
18    end
19
20    def edit
21      @post = @group.posts.find(params[:id])
22    end
23
24    def update
25
26      @post = @group.posts.find(params[:id])
27
28      if @post.update(post_params)
29        redirect_to group_path(@group)
30      else
31        render :edit
32      end
33    end
34
35
36    def destroy
37
38      @post = @group.posts.find(params[:id])
39
40      @post.destroy
41
42      redirect_to group_path(@group)
43    end
44
45
46    private
47
48
49    def find_group
50      @group = Group.find(params[:group_id])
51    end
52
53    def post_params
54      params.require(:post).permit(:content)
55    end
56  end
```

解說

before_action 是一個常見的 controller 技巧，用來收納重複的程式碼。
before_action 可以用 only，指定某些 action 執行：

```
1 before_action :find_group, :only => [:edit, :update]
```

或者使用 except，排除某些 action 不執行：

```
1 before_action :find_group, :except => [:show, :index]
```

練習作業 3 - 為 Group 與 Post 加入使用者機制

在上一章我們完成了在 Group 裡面發表文章的功能。但是通常一個討論區的機制，必須是先加入會員才能進行相關動作。所以我們必須為討論區加入使用者機制:

使用者必須能夠註冊 / 登入，登入後才可以開設 Group，與發表 Post，不然只能瀏覽。只有自己的 Group，Post 才能進行修改與刪除。

本章練習主題

- 安裝 gem
- 設定 devise
- 撰寫全域的 method `login_required`
- 利用 before_action 結合 login_required 加入登入判斷
- session 的使用：current_user

本章參考資料

- devise

Ch 3.0 devise 與 Rails 4

安裝 gem

Rails 內，安裝 gem 的方式是透過 Bundler 這個工具。具體方式是修改 `Gemfile` 這個檔案，加入所需要安裝的 gem，再執行 `bundle install`，即能安裝完畢。

安裝 devise

devise 是目前 Rails 界最被廣為使用的認證系統。其彈性的設計支援很多實務上的需求，如：鎖定賬號（Lockable）、需要認證（Confirmable）、取回帳號（Recoverable）、與第三方認證如 Facebook 整合（OmniAuthable）。

在本書裡面我們使用的 Bootstrappers 內建即幫我們安裝好 `devise` 這個 gem。

不過你還是可以開一個空專案，實際裝一下練習看看。

修改 `Gemfile`

```
1  gem 'devise'
```

使用 bundle 安裝

```
bundle install
```

產生必要檔案

```
rails g devise:install
```

產生 user model 檔案

```
rails generate devise user
```

devise 相關連結

- 註冊 `<%= link_to( "Sign Up" ,new_user_registration_path) %>`
- 登入 `<%= link_to( "Login", new_user_session_path ) %>`
- 登出 `<%= link_to("Logout",destroy_user_session_path, :method => :delete ) %>`

devise 相關 method

1. 判斷現在使用者是否登入了，可以使用 current_user.blank?。
2. 要取現在這個登入的使用者資料，可以使用 current_user

login_required

Bootstrappers 在 app/controller/application_controller.rb 也先預幫開發者準備好一個 method：login_required。

```
  def login_required
    if current_user.blank?
      respond_to do |format|
        format.html  {
          authenticate_user!
        }
        format.js{
          render :partial => "common/not_logined"
        }
        format.all {
          head(:unauthorized)
        }
      end
    end

  end
```

如果開發者只是單純想限制哪一個 action 需要登入才能使用，只要掛上 before_action，再指定即可。

客製化 devise

原始的 devise 設計，只有 email 與 password 的設計，並沒有 name 的欄位。雖然 Bootstrappers 幫忙也生了 name 的欄位，但是我們還是得在 devise 的註冊表單加進去 name 才行，否則註冊進去的 user 都會沒有 name。

devise 的 view 預設是隱藏起來，直接使用 gem 內的 view。要客製化必須要先使用 generator 生出來，再修改複寫。

具體的步驟是執行:

```
rails g devise:views
```

會生成一堆檔案

```
      invoke  Devise::Generators::SharedViewsGenerator
      create    app/views/devise/shared
      create    app/views/devise/shared/_links.erb
      invoke  simple_form_for
      create    app/views/devise/confirmations
      create    app/views/devise/confirmations/new.html.erb
      create    app/views/devise/passwords
      create    app/views/devise/passwords/edit.html.erb
      create    app/views/devise/passwords/new.html.erb
      create    app/views/devise/registrations
      create    app/views/devise/registrations/edit.html.erb
      create    app/views/devise/registrations/new.html.erb
      create    app/views/devise/sessions
      create    app/views/devise/sessions/new.html.erb
      create    app/views/devise/unlocks
      create    app/views/devise/unlocks/new.html.erb
      invoke  erb
      create    app/views/devise/mailer
      create    app/views/devise/mailer/confirmation_instructions.html.erb
      create    app/views/devise/mailer/reset_password_instructions.html.erb
      create    app/views/devise/mailer/unlock_instructions.html.erb
```

修改註冊表單

註冊表單的檔案是 app/views/devise/registrations/new.html.erb，加入一行 name 使之成為

```
<h2>Sign up</h2>

<%= simple_form_for(resource, :as => resource_name, :url => registration_path(res\
ource_name)) do |f| %>
  <%= f.error_notification %>

  <div class="form-inputs">
    <%= f.input :email, :required => true, :autofocus => true %>
    <%= f.input :name, :required => true, :autofocus => true %>
    <%= f.input :password, :required => true %>
    <%= f.input :password_confirmation, :required => true %>
  </div>

  <div class="form-actions">
    <%= f.button :submit, "Sign up" %>
  </div>
<% end %>

<%= render "users/shared/links" %>
```

加入 strong_parameters 與 devise 整合的 hack

在 app/controller/application_controller 加上這些設定:

```
  before_filter :configure_permitted_parameters, if: :devise_controller?


  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.for(:sign_up) { |u| u.permit(:name, :email, :passw\
ord, :password_confirmation) }
  end
```

https://github.com/plataformatec/devise/tree/rails4#strong-parameters

Ch 3.1 對需要登入才能使用的 Action 加入限制

根據需求：使用者必須能夠註冊 / 登入，登入後才可以開設 Group，與發表 Post。
所以我們要在 Groups controller 加入：

```
1  class GroupsController < ApplicationController
2
3    before_action :login_required, :only => [:new, :create, :edit,:update,:destroy]
```

在 Posts controller 加入：

```
1  class PostsController < ApplicationController
2
3    before_action :login_required, :only => [:new, :create, :edit,:update,:destroy]
```

Ch 3.2 讓 Group 與 User 產生關聯:

新增一條 migration: `rails g migration add_user_id_to_group`

```
      invoke  active_record
      create    db/migrate/20130531141923_add_user_id_to_group.rb
```

填入以下內容

```
class AddUserIdToGroup < ActiveRecord::Migration
  def change
    add_column :groups, :user_id, :integer
  end
end
```

執行 `rake db:migrate`

```
==  AddUserIdToGroup: migrating ===============================================
-- add_column(:groups, :user_id, :integer)
   -> 0.0213s
==  AddUserIdToGroup: migrated (0.0214s) ======================================
```

修改 `app/models/user.rb` 加入 `has_many :groups`
內容如下

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable

  has_many :groups

  extend OmniauthCallbacks

  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable, :omniauthable


end
```

修改 app/models/group.rb 加入

```
  belongs_to :owner, :class_name => "User", :foreign_key => :user_id

  def editable_by?(user)
    user && user == owner
  end
```

內容如下：

```
class Group < ActiveRecord::Base

  belongs_to :owner, :class_name => "User", :foreign_key => :user_id
  has_many :posts

  validates :title, :presence => true


  def editable_by?(user)
    user && user == owner
  end
end
```

接著我們要把 Groups 的幾個 action 內容替換掉:

create

```
1  def create
2    @group = current_user.groups.build(group_params)
3    if @group.save
4      redirect_to groups_path
5    else
6      render :new
7    end
8  end
```

edit

```
1  def edit
2    @group = current_user.groups.find(params[:id])
3  end
```

update

```
1  def update
2    @group = current_user.groups.find(params[:id])
3
4    if @group.update(group_params)
5      redirect_to group_path(@group)
6    else
7      render :edit
8    end
9  end
```

destroy

```
1  def destroy
2    @group = current_user.groups.find(params[:id])
3
4    @group.destroy
5
6    redirect_to groups_path
7  end
```

把 app/views/groups/index.html.erb 的內容換掉

```
<div class="span12">
  <div class="group">
    <%= link_to("New group", new_group_path , :class => "btn btn-mini btn-primary\
 pull-right")%>
  </div>
  <table class="table">
    <thead> <tr>
      <td> # </td>
      <td> Title </td>
      <td> Description </td>
      <td> Owner </td>
    </tr>
  </thead>
  <tbody>
    <% @groups.each do |group| %>
    <tr>
      <td> # </td>
      <td> <%= link_to(group.title, group_path(group)) %> </td>
      <td> <%= group.description %> </td>
      <td> <%= group.owner.name %> </td>
      <td>
        <% if current_user && group.editable_by?(current_user) %>
        <%= link_to("Edit", edit_group_path(group), :class => "btn btn-mini")%>
        <%= link_to("Delete", group_path(group), :class => "btn btn-mini", :metho\
d => :delete, :confirm => "Are you sure?" ) %>
        <% end %>
      </td>
    </tr>
    <% end %>
  </tbody>
</table>
</div>
```

解說

在前面一章的例子裡面。我們都是使用 ‘ @group = Group.find(params[:id])‘。

但是這樣無法確保這個 action 的安全性。其實這幾個 action 必須是要「限定」本人才能修改的。但是初學者可能會寫出這樣的 code。

```
1  def edit
2    @group = Group.find(params[:id])
3
4    if @group.user != current_user
5      flash[:warning] = "No Permission"
6      redirec_to root_path
7    end
8
9  end
```

事實上，我們可以只用 current_user.groups 這樣的天然限制。就足以阻擋掉這樣的非法存取了。

```
1  def edit
2    @group = current_user.groups.find(params[:id])
3  end
```

在這個情況下，Rails 在 Production 環境上面會直接跳 404 找不到頁面。直接確保該頁面的安全性。

以下的 create 這個例子也是類似的狀況。原先開發者可能會寫出

```
1  def create
2    @group = Group.new(group_params)
3    @group.user = current_user
4    if @group.save
5      redirect_to groups_path
6    else
7      render :new
8    end
9  end
```

利用內建的 association 就可以改寫成以下範例：

```
1   def create
2     @group = current_user.groups.build(group_params)
3     if @group.save
4       redirect_to groups_path
5     else
6       render :new
7     end
8   end
```

效果是一樣的。

Ch 3.3 讓 Post 與 User 產生關聯:

新增一條 migration: `rails g migration add_user_id_to_post`

```
      invoke  active_record
      create    db/migrate/20130531153435_add_user_id_to_post.rb
```

填入以下內容

```
class AddUserIdToPost < ActiveRecord::Migration
  def change
    add_column :posts, :user_id, :integer
  end
end
```

執行 `rake db:migrate`

```
==  AddUserIdToPost: migrating ================================================
-- add_column(:posts, :user_id, :integer)
   -> 0.0176s
==  AddUserIdToPost: migrated (0.0177s) =======================================
```

修改 `app/models/user.rb` 加入 `has_many :posts`
內容如下

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable

  has_many :groups
  has_many :posts

  extend OmniauthCallbacks

  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable, :omniauthable


end
```

修改 `app/models/post.rb` 加入

```
  belongs_to :author, :class_name => "User", :foreign_key => :user_id

  def editable_by?(user)
    user && user == author
  end
```

內容如下：

```
class Post < ActiveRecord::Base

  belongs_to :group
  validates :content, :presence => true

  belongs_to :author, :class_name => "User", :foreign_key => :user_id

  def editable_by?(user)
    user && user == author
  end

end
```

接著我們要把 Posts 的幾個 action 內容替換掉:

create

```
  def create

    @post = @group.posts.new(post_params)
    @post.author = current_user

    if @post.save
      redirect_to group_path(@group)
    else
      render :new
    end
  end
```

edit

```
  def edit
    @post = current_user.posts.find(params[:id])
  end
```

update

```
  def update

     @post = current_user.posts.find(params[:id])

    if @post.update(post_params)
      redirect_to group_path(@group)
    else
      render :edit
    end
  end
```

destroy

```
  def destroy

    @post = current_user.posts.find(params[:id])

    @post.destroy

    redirect_to group_path(@group)
  end
```

把 app/views/groups/show.html.erb 的內容換掉

```
<div class="span12">


 <div class="group pull-right">
 <% if current_user && @group.editable_by?(current_user) %>
  <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-mini ")%>
  <% end %>

  <%= link_to("New Post", new_group_post_path(@group) , :class => "btn btn-mini b\
tn-primary" if current_user )%>
   
  </div>

<h2> <%= @group.title %> </h2>

<p> <%= @group.description %> </p>


<table class="table">

  <tbody>
    <% @posts.each do |post| %>
    <tr>
      <td>

        <span clas="author"> <strong> Author : <%= post.author.name %> </strong> \
</span>
        <p>
        <%= post.content %>
```

```
30        </p>
31        </td>
32
33         <% if current_user && post.editable_by?(current_user) %>
34        <td> <%= link_to("Edit", edit_group_post_path(post.group, post), :class => \
35  "btn btn-mini")%>
36          <%= link_to("Delete", group_post_path(post.group, post), :class => "btn b\
37  tn-mini", :method => :delete, :confirm => "Are you sure?" ) %> </td>
38        </tr>
39        <% end %>
40        <% end %>
41      </tbody>
42    </table>
43  </div>
```

練習作業 4 - User 可以加入、退出社團

作業目標

- user 可以在 group 頁面加入社團 / 退出社團
- user 必須要是這個社團的成員才能發表文章

本章練習主題

- has_many_belongs_to
- custom routes

Ch 4.1 使用者必須要是這個社團的成員才能發表文章

在第 2 章、第 3 章我們完成了 Group 與 Post 的新增與管理。在這一章我們要挑戰一點進階的課題。

4.1.1 has_many :through

新增 group_user 這個 model: `rails g model group_user group_id:integer user_id:integer`

執行 migration: `rake db:migrate`

```
==  CreateGroupUsers: migrating ===============================================
-- create_table(:group_users)
   -> 0.0238s
==  CreateGroupUsers: migrated (0.0239s) ======================================
```

在 User model 加入

```
  has_many :group_users
  has_many :participated_groups, :through => :group_users, :source => :group
```

內容變成以下

```
class User < ActiveRecord::Base

  has_many :groups
  has_many :posts

  has_many :group_users
  has_many :participated_groups, :through => :group_users, :source => :group

  extend OmniauthCallbacks

  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable, :omniauthable


  def join!(group)
    participated_groups << group
  end

```

```
19    def quit!(group)
20      participated_groups.delete(group)
21    end
22
23    def is_member_of?(group)
24      participated_groups.include?(group)
25    end
26
27  end
```

在 Group model 加入

```
  has_many :group_users
  has_many :members, :through => :group_users, :source => :group
```

內容變成以下

```
class Group < ActiveRecord::Base

  belongs_to :owner, :class_name => "User", :foreign_key => :user_id
  has_many :posts
  has_many :group_users
  has_many :members, :through => :group_users, :source => :group

  validates :title, :presence => true


  def editable_by?(user)
    user && user == owner
  end
end
```

修改 GroupUser model 成

```
class GroupUser < ActiveRecord::Base
  belongs_to :group
  belongs_to :user
end
```

解說

Many to Many 關係

```
  has_many :group_users
  has_many :members, :through => :group_users, :source => :user
```

Rails 常見的關係有 has_one,belongs_to 與 has_many。has_many :through 是最後一種。多數用在多對多（表間列表）的關係上。

Ch 4.1.2 model method 與 after create

實作以下 method 在 User model 內

```
  def join!(group)
    participated_groups << group
  end

  def quit!(group)
    participated_groups.delete(group)
  end

  def is_member_of?(group)
    participated_groups.include?(group)
  end
```

這樣我們之後，就可以在 controller，使用 current_user.join!(group) 這樣的語法來加入 group。

產生 Group 後自動加入 group 作為 group 的一員

在一般的認知中，Group 的開創者應該就要是 group 的一員。這有兩種作法，一種是到 `app/controller/groups_controller.rb` 裡的 `create` action 裡面加入 `current_user.join!(group)`

```
  def create
    @group = current_user.groups.build(group_params)
    if @group.save
      current_user.join!(group)
      redirect_to groups_path
    else
      render :new
    end
  end
```

或者是你也可以使用 ActiveRecord 的 after_create，更漂亮的實作：

```ruby
class Group < ActiveRecord::Base

  belongs_to :owner, :class_name => "User", :foreign_key => :user_id
  has_many :posts

  has_many :group_users
  has_many :members, :through => :group_users, :source => :user


  validates :title, :presence => true

  after_create :join_owner_to_group

  def editable_by?(user)
    user && user == owner
  end

  def join_owner_to_group
    members << owner
  end

end
```

解說

after_create

`after_create` 是 ActiveRecord 提供的 callbacks，意旨在簡化程式碼。類似的 callbacks 還有

- before_create
- before_update
- after_create
- after_update

等等...

Ch 4.1.3 join 與 quit action

在 Groups Controller 加入以下這兩個 action

```
  def join
    @group = Group.find(params[:id])

    if !current_user.is_member_of?(@group)
      current_user.join!(@group)
    else
      flash[:warning] = "You already joined this group."
    end
    redirect_to group_path(@group)
  end

  def quit
    @group = Group.find(params[:id])

    if current_user.is_member_of?(@group)
      current_user.quit!(@group)
    else
      flash[:warning] = "You are not member of this group."
    end

    redirect_to group_path(@group)

  end
```

然後在 config/routes.rb 修改 `resources :groups` 這一段，變成:

```
  resources :groups do
    member do
      post :join
      post :quit
    end
    resources :posts
  end
```

再修改 `app/views/groups/show.html.erb` 加入這一段:

```
  <% if current_user %>
    <div class="group pull-right">
    <% if current_user.is_member_of?(@group) %>
      <%= link_to("Quit Group", quit_group_path(@group), :method => :post, :class\
 => "btn btn-mini") %>
    <% else %>
      <%= link_to("Join Group", join_group_path(@group), :method => :post, :class\
 => "btn btn-mini") %>
    <% end %>
    </div>
  <% end %>
```

修改 app/views/groups/show.html.erb 原先的

```
  <%= link_to("New Post", new_group_post_path(@group) , :class => "btn btn-mini b\
tn-primary") if current_user %>
```

變成

```
  <%= link_to("New Post", new_group_post_path(@group) , :class => "btn btn-mini b\
tn-primary") if current_user.is_member_of?(@group) %>
```

在 controller 裡加入 member_required

在 Post 這個 controller 裡面加入另一個 before_action

```
  before_action :member_required, :only => [:new, :create ]
```

在 private 下新增 member_required 這個 method

```
  def member_required
    if !current_user.is_member_of?(@group)
      flash[:warning] = " You are not member of this group!"
      redirect_to group_path(@group)
    end
  end
```

練習作業 5 - 實作簡單的 Account 後台機制

作業目標

- user 可以在 account 頁面找到自己曾經加入的社團
- user 可以至 account 找到自己曾經發表過的文章
- 每個 Group 要秀出現在有多少 post 數量
- 文章列表的排序要以發表時間以 DESC 排序。

練習主題

- 使用 counter_cache 取代直接對資料庫的 count
- 了解 ORM 的基本用法

Ch 5.1 User 必須要在使用者後台可以看到自己參加的 Group

新增 account/groups controller: `rails g controller account/groups`

```
   identical  app/controllers/account/groups_controller.rb
      invoke  erb
       exist    app/views/account/groups
      invoke  test_unit
   identical    test/controllers/account/groups_controller_test.rb
      invoke  helper
   identical    app/helpers/account/groups_helper.rb
      invoke    test_unit
   identical      test/helpers/account/groups_helper_test.rb
      invoke  assets
      invoke    coffee
   identical      app/assets/javascripts/account/groups.js.coffee
      invoke    scss
   identical      app/assets/stylesheets/account/groups.css.scss
```

新增 app/controllers/account/groups_controller.rb 裡的內容

```ruby
class Account::GroupsController < ApplicationController
  before_action :login_required

  def index
    @groups = current_user.participated_groups
  end
end
```

touch app/views/account/groups/index.html.erb 新增裡面的內容

```
<div class="span12">

  <h2> My Groups </h2>

  <table class="table">
    <thead> <tr>
      <td> # </td>
      <td> Title </td>
      <td> Descroption </td>
      <td> Post Count </td>
```

```
      <td> Last Update </td>
    </tr>
  </thead>
  <tbody>
    <% @groups.each do |group| %>
    <tr>
      <td> # </td>
      <td> <%= link_to(group.title, group_path(group)) %> </td>
      <td> <%= group.description %> </td>
      <td> <%= group.posts.count %> </td>
      <td> <%= group.updated_at %> </td>
    </tr>
    <% end %>
  </tbody>
</table>
</div>
```

修改 config/routes.rb 加入

```
  namespace :account do
    resources :groups
  end
```

修改 app/common/_user_nav.html.erb 裡的

```
      <%= render_list :class => "dropdown-menu" do |li|
          li << link_to("Logout",destroy_user_session_path, :method => :delete )
      end %>
```

變成

```
      <%= render_list :class => "dropdown-menu" do |li|
          li << link_to("My Group", account_groups_path)
          li << link_to("Logout",destroy_user_session_path, :method => :delete )
      end %>
```

Ch 5.2 User 必須要在使用者後台可以看到自己發表的文章

新增 account/posts controller: `rails g controller account/posts`

```
      create  app/controllers/account/posts_controller.rb
      invoke  erb
      create    app/views/account/posts
      invoke  test_unit
      create    test/controllers/account/posts_controller_test.rb
      invoke  helper
      create    app/helpers/account/posts_helper.rb
      invoke    test_unit
      create      test/helpers/account/posts_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/account/posts.js.coffee
      invoke    scss
      create      app/assets/stylesheets/account/posts.css.scss
```

新增 `app/controllers/account/groups_controller.rb` 裡的內容

```ruby
class Account::PostsController < ApplicationController

  before_action :login_required

  def index
    @posts = current_user.posts
  end

end
```

touch app/views/account/posts/index.html.erb 新增裡面的內容

```
<div class="span12">

  <table class="table">

    <thead>
     <tr>

      <td> Content </td>
      <td> Group name </td>
      <td> Last Update </td>
    </tr>
  </thead>

  <tbody>
    <% @posts.each do |post| %>
    <tr>
      <td>        <p>       <%= post.content %>     </p>     </td>

      <td>      <%= post.group.title %>    </td>

      <td> <%= post.updated_at %> </td>


      <td> <%= link_to("Edit", edit_group_post_path(post.group, post), :class => \
"btn btn-mini")%>
        <%= link_to("Delete", group_post_path(post.group, post), :class => "btn b\
tn-mini", :method => :delete, :confirm => "Are you sure?" ) %> </td>
      </tr>
      <% end %>
    </tbody>
  </table>
</div>
```

修改 config/routes.rb 加入 resources :posts

```
  namespace :account do
    resources :groups
    resources :posts
  end
```

修改 app/common/_user_nav.html.erb 裡的

```
1       <%= render_list :class => "dropdown-menu" do |li|
2           li << link_to("My Group", account_groups_path)
3           li << link_to("Logout",destroy_user_session_path, :method => :delete )
4       end %>
```

變成

```
1       <%= render_list :class => "dropdown-menu" do |li|
2           li << link_to("My Group", account_groups_path)
3           li << link_to("My Post", account_posts_path)
4           li << link_to("Logout",destroy_user_session_path, :method => :delete )
5       end %>
```

Ch 5.3 文章列表的排序要以發表時間 DESC 排序

你應該有注意到，在 My Post 裡的每篇文章是以文章先後發表順序排列的，也就是越晚發表的文章排越下面。但是這樣預設的排列方式違反一般開發者的使用習慣。

比較適當的排列方式應該是以文章的「修改時間」「倒序排列」。也就是最近修改的文章要排在越上面才行。

修改 app/controllers/account/groups_controller.rb 裡的內容

```
1  class Account::PostsController < ApplicationController
2
3    before_action :login_required
4
5    def index
6      @posts = current_user.posts.order("updated_at DESC")
7    end
8
9  end
```

這樣就能達到想要的效果了。

Ch 5.4 Group 的排序要以文章數量的熱門度 ASC 排序

在 app/views/account/groups/index.html.erb 裡面你應該有發現這一段 <%= group.posts.count %> code。

```
<div class="span12">

  <tbody>
    <% @groups.each do |group| %>
    <tr>
      <td> # </td>
      <td> <%= link_to(group.title, group_path(group)) %> </td>
      <td> <%= group.description %> </td>
      <td> <%= group.posts.count %> </td>
      <td> <%= group.updated_at %> </td>
    </tr>
    <% end %>
  </tbody>
</table>
</div>
```

group.posts.count 這一段 code 是不太健康的，因為它產生了這樣的 SQL query。

```
(0.2ms)  SELECT COUNT(*) FROM `posts` WHERE `posts`.`group_id` = 1
```

如果你有開發過網頁程式 application 的經驗，就知道在迴圈裡面跑 count 對效能是相當傷的。實務上我們相當不建議這麼做。

那麼這一段程式碼要怎麼改善呢？比較直觀的想法，就是在 groups 這個 table 再開一欄叫作 posts_count 的欄位。然後在 create action 裡面對 posts_count +1

```
  def create

    @post = @group.posts.new(post_params)
    @post.author = current_user

    if @post.save
      Group.increment_counter(:posts_count, @group.id)
      redirect_to group_path(@group)
    else
      render :new
    end
  end
```

在 destroy action 裡面對 posts_count - 1

```
  def destroy

    @post = current_user.posts.find(params[:id])
    @post.destroy

    Group.decrement_counter(:posts_count, @group.id)
    redirect_to group_path(@group)
  end
```

不過，其實倒也不用這麼麻煩。Rails 內建一個叫作 counter_cache 的機制，只要內建子關係的 count 欄位，如 posts_count。

```
rails g migration add_posts_count_to_group
```

```
      invoke  active_record
      create    db/migrate/20130531183331_add_posts_count_to_group.rb
```

填入

```
class AddPostsCountToGroup < ActiveRecord::Migration
  def change
    add_column :groups, :posts_count, :integer, :default => 0
  end
end
```

執行 rake db:migrate

```
==  AddPostsCountToGroup: migrating ============================================
-- add_column(:groups, :posts_count, :integer, {:default=>0})
   -> 0.0232s
==  AddPostsCountToGroup: migrated (0.0232s) ===================================
```

然後在子關係的欄位裡，這樣設定:

```
  belongs_to :group, :counter_cache => true
```

以後只要 post 遇到 create 或者是 destroy，就會自動對這個欄位 +1 / -1 。不需要在 controller 裡面另外動作。

而上了 counter_cache 之後，<%= group.posts.count %> 以後執行時，也會優先去找 posts_count 裡的值，而不是真的下 MySQL 的 COUNT 去實際算值。

而加了這個 posts_count 欄位之後，我們就可以修改 index 裡的排序規則變成以文章數量的熱門度 ASC 排序

```
1  def index
2    @groups = current_user.participated_groups.order("posts_count ASC")
3  end
```

練習作業 6 - Refactor code

作業目標

- 使用系統 helper / 自己撰寫的 helper 包裝 html
- 使用 partial 整理 html
- 使用 scope 整理 query

練習主題

- 練習使用內建 helper
- 練習拆 partial
- 練習使用 scope 機制包裝不同 condition 的 SQL query

Ch 6.1 使用系統 helper 整理 code

我們已經初步完成了這個系統。但有一些設計看起來還是令人不太滿意的。比如說 `post.updated_at` 與 `group.updated_at`。

顯示出來的會是 `2013-05-31 14:47:31 UTC` 這種格式。

date.to_s

但其實我們想要顯示的好看一點，可以改成這樣的寫法:

- `post.updated_at.to_s(:long)` ⇒ May 31, 2013 18:04
- `group.updated_at.to_s(:short)` ⇒ 31 May 18:04

simple_format

當我們在顯示 `post.content` 時，有個困擾。我們在輸入框輸入

```
a
b
c
```

但在系統輸出的時候，HTML 並不會自動斷行，會顯示成

```
a b c
```

用 Enter 斷行的 `\r` 或 `\n` 在 HTML 裡面並不起作用。若要在 HTML 裡面正確斷行必須要使用 `<br>`。

若在以往，要漂亮顯示文本，開發者必須自行寫一段 nl2br 的轉換 helper。不過 Rails 裡面內建 `simple_format` 這個 helper，可以自動幫我們處理這種雜事。

```
<%= simple_format(post.content) %>
```

truncate

在顯示 group.title 時，有時候我們也會遇到標題太長的問題，導致破版。所以要砍字，再加上 “...”

Rails 內建了一個 `truncate` helper 可以快速辦到這個效果

```
<%= truncate(@group.title, :length => 17 ) %>
```

Ch 6.2 自己撰寫的 helper 包裝 html

Helper 是一些使用在 Rails 的 View 當中，用 Ruby 產生/整理 HTML code 的一些小方法。通常被放在 `app/helpers` 下。預設的 Helper 名字是對應 Controller 的，產生一個 Controller 時，通常會產生一個同名的 Helper。如 `PostsController` 與 `PostsHelper`。

使用情境

使用 Helper 的情境多半是:

- 產生的 HTML code 需要與原始程式碼進行一些邏輯混合，但不希望 View 裡面搞得太髒。
- 需要與預設的 Rails 內建的一些方便 Helper 交叉使用。

使用 Helper 封裝程式碼可以帶給專案以下一些優點:

- Don't repeat yourself（DRY）程式碼不重複
- Good Encapsulation 好的封裝性
- 提供 view 模板良好的組織
- 易於修改程式碼

範例

在剛剛的專案當中，顯示 Post 的程式碼如下:

```
1   <%= @post.content %> %>
```

隨著專案變遷，這樣的程式碼，可能會依需求改成:（需要內容斷行）

```
1   <%= simple_format(@post.content) %> %>
```

之後又改成（只顯示頭一百字）

```
1   <%= truncate(simple_format(@post.content), :lenth => 100) %>
```

最後又改成（內容若有網址需要自動超連結）

```
1     <%= auto_link(truncate(simple_format(@post.content), :lenth => 100)) %>
```

而麻煩的是，這樣類似的內容，常常在專案出現。每當需求變更，開發者就需要去找出來，有十個地方，就需要改十遍，很是麻煩。

Helper 就是用在這樣的地方。與其一開始寫下

```
1   <%= @post.content %> %>
```

不如，一開始就設計一個 Helper ‘ <%= render_post_content(@post) %>‘

```
1   def render_post_content(post)
2     auto_link(truncate(simple_format(@post.content), :lenth => 100))
3   end
```

以後變更需求就只要修改一個地方即可。

更多的 Partial 用法

http://guides.rubyonrails.org/layouts_and_rendering.html#using-partials

Ch 6.3 使用 partial 整理 html

Partial 也是 Rails 提供用來整理 View 的一種方式，開發者可以使用 Partial 技巧將太長的 View 整理切該成好維護的小片程式碼。

Helper 是一些使用在 Rails 的 View 當中，用 Ruby 產生/整理 HTML code 的一些小方法。通常被放在 `app/helpers` 下。預設的 Helper 名字是對應 Controller 的，產生一個 Controller 時，通常會產生一個同名的 Helper。如 `PostsController` 與 `PostsHelper`。

使用情境

什麼時候應該將把程式碼搬到 Partial 呢?

* long template | 如果當檔 HTML 超過兩頁 * highly duplicated | HTML 內容高度重複 * independent blocks | 可獨立作為功能區塊

常見範例

- nav/user_info
- nav/admin_menu
- vendor_js/google_analytics
- vendor_js/disqus_js
- global/footer

本書範例 project 裡面的 layout/application.html.erb 就是很好的範例。

```
<%= render :partial => "common/menu" %>

  <div class="container">

      <%= notice_message %>

      <div class="content">
        <div class="row">
          <%= yield %>
          <%= yield (:sidebar) %>
        </div>

      </div>

      <%= render :partial => "common/footer" %>
  </div>



  <%= render :partial => "common/bootstrap_modal" %>
  <%= render :partial => "common/facebook_js" %>
  <%= render :partial => "common/google_analytics" %>

  <%= javascript_include_tag "application" %>

  <%= yield :javascripts %>
```

更多的 Partial 用法

http://guides.rubyonrails.org/layouts_and_rendering.html#using-partials

Ch 6.4 使用 scope 整理 query

在 Ch 5.3 裡面，我們在 index controller 裡面設計了一行程式碼，讓文章永遠按照最新的時間降序排列。

```
class Account::PostsController < ApplicationController

  def index
    @posts = current_user.posts.order("updated_at DESC")
  end

end
```

`posts.order("updated_at DESC")` 其實是一段可能會很常用到的程式碼。如果 View 需要 Helper 包裝整理，其實我們也需要工具來整理 Query。

這段程式碼比較理想的方式其實是：

```
class Account::PostsController < ApplicationController

  def index
    @posts = current_user.posts.recent
  end

end
```

讓維護者可以從程式碼裡面知道這一個 controller 提供的是「最近的文章」。Rails 在 ORM 中提供了 Scope，可以讓開發者包裝 query：

```
class Post < ActiveRecord::Base
  scope :recent, -> { order("updated_at DESC") }
end
```

Scope 串接

甚至也提供串接的功能:

```
class Post < ActiveRecord::Base
  scope :recent, -> { order("updated_at DESC") }
  scope :published, -> { where(:published => true) }
end
```

你可以用不同的條件用「語意」串接組起想要撈出的資料。

```
class Account::PostsController < ApplicationController

  before_action :login_required

  def index
    @posts = current_user.posts.published.hot
  end

end
```

更多的 scope 用法

http://edgeguides.rubyonrails.org/active_record_querying.html#scopes

練習作業 7 - 撰寫自動化 Rake 以及 db:seed

作業目標

用 Rake 撰寫自動化步驟，生假資料。

寫一個 rake 可以達成以下步驟:「砍 db ⇒ 建 db ⇒ 跑 migration ⇒ 生種子資料」，另一個 rake 是生假 Group 與假文章。

練習主題

- 操作 rake -T
- 撰寫一個 task 可以自動連續執行 rake db:drop ; rake db:create ; rake db:migrate ; rake db:seed
- 撰寫一個 task 可以執行 rake dev:fake 生假資料 (自己寫 namespace : dev, 裡面放一個 task 叫做 fake，fake 資料用 Populator 生）# 請自行練習

參考資料

- Ruby on Rails Rake Tutorial
- What’s New in Edge Rails: Database Seeding

Ch 7.1 Rake

Rake 的意思就是字面上直觀的 Ruby Make，Ruby 版 Make：用 Ruby 開發的 Build Tool。

你也許會問，Ruby 是直譯語言，不需要 compile，為何還需要 Build Tool。

與其說 Rake 是一套 Build Tool，還不如說是一套 task 管理工具。我們通常會使用 Ruby based 的 Rake 在 Rails 專案裡編寫 task。

Rails 內的 rake tasks

`rake -T` 會秀出一大串這個 Rails project 可用的 rake tasks (包含 plugin 內建的 task 也會秀出來)。

```
rake about              # List versions of all Rails frameworks and the environme\
nt
rake assets:clean       # Remove compiled assets
rake assets:precompile  # Compile all the assets named in config.assets.precompile
rake db:create          # Create the database from config/database.yml for the cu\
rrent Rails.env (use db:create:all to create all dbs in the config)
rake db:drop            # Drops the database for the current Rails.env (use db:dr\
op:all to drop all databases)
 .....
```

清空系統重跑 migration 與種子資料的 rake dev:build

在還未上線的開發階段，我們有時會不是那麼喜歡被 db migration 拘束，有時候下錯了 migration，或想直接清空系統重來，實在是件麻煩事。因此寫支 rake 檔自動化是個良招。

新增 `lib/tasks/dev.rake`

```
namespace :dev do
  desc "Rebuild system"
  task :build => ["tmp:clear", "log:clear", "db:drop", "db:create", "db:migrate"]\
 task :rebuild => [ "dev:build", "db:seed" ]
end
```

執行 rake dev:build 看看會發生什麼事吧：

```
    $ rake dev:build
```

`rake dev:build` 是個無敵大絕：清空系統重來！

不過系統清空重來還是很麻煩，我們還是需要一些種子資料，如種子看板，種子文章，管理者帳號，測試用一般帳號。

rake db:seed

db/seed.rb 就是讓你撰寫這些種子資料，可以在專案開始時自動產生一些種子資料。

注意

請注意！seed.rb 裡放的是種子資料！不是假資料！假資料應該要放在 rake 檔裡，而不是放在種子檔裡！

```
admin = User.new(:email => "admin@example.org", :password => "123456",
  :password_confirmation => "123456")
admin.is_admin = true
admin.save!

normal_user = User.new(:email => "user1@example.org", :password => "123456",
  :password_confirmation => "123456")
normal_user.save!

board = Board.create!(:name => "System Announcement")

post = board.posts.build(:title => "First Post", :content => "This is a demo post\
")
post.user_id = admin.id
post.save!
```

練習作業

- 撰寫一個 task 可以自動連續執行 rake db:drop ; rake db:create ; rake db:migrate ; rake db:seed
- 撰寫一個 task 可以執行 rake dev:fake 生假資料 (自己寫 namespace : dev, 裡面放一個 task 叫做 fake，fake 資料用 Populator 生）# 請自行練習

練習作業 8 - 將專案 deploy 到租來的 VPS

作業目標

在租來的 VPS 上面建置 Ruby on Rails production 環境，使用 Ruby Enterprise Edition 與 mod_rails。使用 Capistrano 佈署 application。

練習主題

- 學會如何自動部署專案
- 使用 capistrano 自動部署專案
- 操作 cap deploy:setup
- 操作 cap deploy
- 操作 cap deploy:rollback
- 操作 cap deploy:restart

參考資料

- rails-nginx-passenger-ubuntu
- AWDR4 deploy 章節

Ch 8.1 佈署 Rails Production 所需要的環境

佈署 Rails Application 是件說簡單很簡單，說複雜也很複雜的事。佈署的手法有很多種搭配，筆者最推薦的其實是在 Ubuntu / Debian 安裝 Ruby Enterprise Edition，web sever 使用 nginx + mod_rails 的組合。之後再撰寫 Capistrano 的 recipe 來 deploy。

為什麼要用獨立的 Ruby 版本，而不使用系統 Ruby？

因為系統 Ruby 通常綑綁了背後的套件系統（RubyGem），Rails 是個腳步前進很快的生態圈。而各樣相依套件有時候也會限定 RubyGem 的版本。很多時候，在開發或佈署上就會踩到大地雷。

所以會建議在系統上跑獨立的 Ruby。

為什麼要用 Ubuntu / Debian，而不是使用 CentOS？

還是跟 Rails 是個腳步前進很快的生態圈有相當大的關係。Gem 的前進腳步很快，很多時候只 compatible 新的 library，而 CentOS 上很多 package 都已經 outdate 了。實際佈署上會踩到很多雷。而 Ubuntu / Debian 上的 package 更新速度非常快。所以也是首選。

有沒有 Best Practice 懶人包？

有。其實佈署真的不算是件易事，在佈署中最容易踩到的雷當數 ImageMagick (rmagick gem) 與 MySQL (mysql2 gem)。偏偏這幾乎是每個網站最常會用到的兩個 gem。而且裝爛了很難重裝。

這是我們公司標準用來裝機的 Step by Step guide：https://github.com/rocodev/guides/wiki/setup-production-development 基本上已經排除不少裝機時可能會遇到的狀況。

哪裡買域名和租 VPS？

我是在 enom.com 買域名，管理介面還算蠻好用的。

至於 VPS 是去 Linode 租的。算便宜大碗，速度上也能接受。若純練習也可到 AWS EC2 租東京的 micro instance。

Ch 8.2 Capistrano

Capistrano 是 37 signals 開發的一套 automate deploy tool。也是許多 Rails Developer 推薦的一套佈署工具。

也許你要問，為什麼要用工具 deploy 專案呢? 那是因為佈署 application 是由數道繁瑣的手續構成。首先，Rails 佈署程式並不像 php 那樣簡單，上傳檔案成功就完事了。要佈署一個 Rails Application，你必須連到遠端 server，然後 checkout 程式碼，跑一些 migration，重開 server 生效，重開 memcached，重開 search daemon ....

這些連續動作有時候往往一閃神即是災難。

而手動 deploy 在只有一個人一台機器時還勉强説得過去。當開發人員一多或機器一多，馬上就會要了大家的命。

Capistrano 固然强大，但官方網站的文件卻讓人不易讀懂。Bootstrappets 內建了一個已經寫好的 recipe，可以直接使用。

（放在 config/deploy.rb ）

```
# -*- encoding : utf-8 -*-

raw_config = File.read("config/config.yml")
APP_CONFIG = YAML.load(raw_config)

require "./config/boot"
require "bundler/capistrano"
require "rvm-capistrano"

default_environment["PATH"] = "/opt/ruby/bin:/usr/local/bin:/usr/bin:/bin"

set :application, "groupme"
set :repository,  "git@github.com:example/#{application}.git"
set :deploy_to, "/home/apps/#{application}"

set :branch, "master"
set :scm, :git

set :user, "apps"
set :group, "apps"

set :deploy_to, "/home/apps/#{application}"
set :runner, "apps"
set :deploy_via, :remote_cache
set :git_shallow_clone, 1
set :use_sudo, false
set :rvm_ruby_string, '1.9.3'

set :hipchat_token, APP_CONFIG["production"]["hipchat_token"]
set :hipchat_room_name, APP_CONFIG["production"]["hipchat_room_name"]
set :hipchat_announce, false # notify users?

role :web, "groupme.com"                          # Your HTTP server, Apache/etc
role :app, "groupme.com"                          # This may be the same as your `\
Web` server
role :db,  "groupme.com"    , :primary => true # This is where Rails migrations wi\
ll run

set :deploy_env, "production"
set :rails_env, "production"
set :scm_verbose, true
set :use_sudo, false
```

```
43
44
45  namespace :deploy do
46
47    desc "Restart passenger process"
48    task :restart, :roles => [:web], :except => { :no_release => true } do
49      run "touch #{current_path}/tmp/restart.txt"
50    end
51  end
52
53
54  namespace :my_tasks do
55    task :symlink, :roles => [:web] do
56      run "mkdir -p #{deploy_to}/shared/log"
57      run "mkdir -p #{deploy_to}/shared/pids"
58
59      symlink_hash = {
60        "#{shared_path}/config/database.yml"   => "#{release_path}/config/database.\
61  yml",
62        "#{shared_path}/config/s3.yml"   => "#{release_path}/config/s3.yml",
63        "#{shared_path}/uploads"              => "#{release_path}/public/uploads",
64      }
65
66      symlink_hash.each do |source, target|
67        run "ln -sf #{source} #{target}"
68      end
69    end
70
71  end
72
73  namespace :remote_rake do
74    desc "Run a task on remote servers, ex: cap staging rake:invoke task=cache:clea\
75  r"
76    task :invoke do
77      run "cd #{deploy_to}/current; RAILS_ENV=#{rails_env} bundle exec rake #{ENV['\
78  task']}"
79    end
80  end
81
82  after "deploy:finalize_update", "my_tasks:symlink"
```

Ch 8.3 Capistrano 常用指令

cap deploy:setup

第一次使用，運行此行指令，Capistrano 就會遠端到機器上幫你把 Capistrano 所需的一些目錄和檔案先預備好

cap deploy

Deploy 專案到遠端

cap deploy:migrate

遠端執行 migration

cap deploy:rollback

deploy 的這一版本爛了，想回到沒爛的上一版本。

cap deploy:restart

純粹重開 application

Ch 8.4 Deploy with Rails 4

Rails 4 在部署上 Server 時有時候會遇到 Asset Compile 不過的問題。

Rails 3

config/production.rb

```
1   config.serve_static_assets = false
2   config.assets.compile = false
```

Rails 4

在 Rails 4 上要修改成

```
1   config.serve_static_assets = true
2   config.assets.compile = true
3   config.assets.compress = true
4   config.assets.configure do |env|
5    env.logger = Rails.logger
6   end
```

補充章節：Asset Pipeline

Rails 在 3.1 之後，提供一套 Asset 開發流程，稱之 Asset Pipeline 。是一套讓開發者很方便能夠削減 (minify) 以及壓縮 (compress) Javascript / CSS 檔案的框架。它同時也提供你直接利用其他語言如 CoffeeScript / SASS / ERB ， 直接撰寫 assets (指的是 stylesheets / javascripts / images 這些靜態檔案) 的可能性。

SCSS

SCSS 是一套新型的 Asset 語言，可以用「更合理的方式」撰寫 CSS。舉例來說：

```
.content{ margin: 2em 0;}
.content h1{ font-size: 2em;}
```

在 SCSS 裡可以寫成巢狀的

```
.content{
margin: 2em 0;
  h1{font-size: 2em;}
}
```

編譯器會自動幫你編譯成 CSS。如此一來好維護多了。

此外，SCSS 還支援一些強大的功能：如變數、函數、數學、繼承、mixin …等等。

這些內建功能，有多方便呢？就拿變色來說吧。在進行網頁 protyping 時，更改全站配色或者是直接提供兩個以上的設計，對設計師來說是家常便飯的事。

但更改全站配色卻是相當麻煩的一件事，因為「尋找 + 全數取代」，並不能保證最後會有正確的結果。很有可能：你更改了所有 CSS 中涉及連結的顏色，卻發現在全數取代的過程中，不小心也改到邊框的顏色。

但是 SCSS 可以讓我們使用變數去指定特定 style 的顏色。相當厲害。

```
$border-color: #3bbfce;
$link-color: #3bbfcf'
.content{
  border-color: $border-color;
  a{ color: $link-color; }
}
```

會產生出

```
.content{ border-color: #3bbfce; }
.content a{color: #3bbfcf; }
```

Compass

而 SCSS 上的 Compass 這套 Framework 更是厲害，它提供了不少 mixin 可以讓 Developer 更加省事。

就如我們常常使用的圓角框技巧來說好了，以往我們要設計一個圓角框，必須囉哩八嗦的這樣寫：

```
#border-radius {
  -moz-border-radius: 25px;
  -webkit-border-radius: 25px;
  -o-border-radius: 25px;
  -ms-border-radius: 25px;
  -khtml-border-radius: 25px;
  border-radius: 25px;
}
```

而使用 Compass 我們只要這樣寫就可以了：

```
1  #border-radius {  @include border-radius(25px); }
```

CoffeeScript

Asset Pipeline 也支援了 CoffeeScript。CoffeeScript 本身也是一種程式語言，開發者可以透過撰寫 CoffeeScript，編譯產生 JavaScript。它的語法有點像是 Ruby 與 Python 的混合體。

Plurk 前創辦人 amix 曾寫過一篇這樣的 post：CoffeeScript: The beautiful way to write Javascript 來這樣形容 CoffeeScript：「以更漂亮的方式撰寫 JavaScript」。

他認為目前 JavaScript 存在幾種問題：

- JavaScript 是 functional language
- 雖然是 OOP，但卻是 prototype-based 的 JavaScript 是 dynamic language，更像 Lisp 而不是 C/Java，但卻用了 C/Java 的語法。
- 名字裡面有 Java，但卻和 Java 沒什麼關係。
- 明明是 functional & dynamic laungaue，更偏向 Ruby / Python，卻使用了 C / Java 的 syntax，原本可以是一門很美的語言，卻活生生的變成了悲劇。

而 CoffeeScript 的誕生，原因就是就是為了扭正這樣的局面，重新讓寫 JavaScript 這件事也可以變得「很美」。

CoffeeScript : The Good Part

CoffeeScript 留下了 JavaScript 的 Good Parts，而在設計上極力消除 JavaScript 原生特性會產生的缺點，例如：

消除到處污染的全域變數

開發者在寫 JavaScript 時，常不自覺的使用全域變數，導致很多污染問題。而透過 CoffeeScript 生出來的 JavaScript，變數一律為區域變數（以 var 開頭）

Protected code

使用 CoffeeScript 撰寫 function，產生出來的 JavaScript 必以一個 anonymous function: function(){}(); 自我包裹，獨立運作不干擾到其他 function。

使用 -> 和 indent(縮排) 讓撰寫 function 更不容易出錯

在撰寫 JavaScript 時，最令人不爽的莫過於 `function(){}();`，這些複雜的括號和分號稍微一漏，程式就不知道死在哪裡了….

CoffeeScript 自動產生出來的 JavaScript 能夠確保括號們絕對不會被漏掉。

更容易偵測 syntax error 並攔阻

JavaScript 在 syntax error 時，非常難以偵測錯誤，幾乎是每個程式設計師的夢魘。而 CoffeeScript 是一門需要 compile 的語言，可以藉由這樣的特性擋掉 syntax error 的機會。

實作物件導向更簡單

Javascript 是一種物件導向語言，裡面所有東西幾乎都是物件。但 JavaScript 又不是一種真正的物件導向語言，因為它的語法裡面沒有 class（類別）。

在 JavaScript 中，我們要實作 OOP 有很多種方式，你可以使用 prototype、function 或者是 Object。但無論是哪一種途徑，其實都「不簡單」。

但 CoffeeScript 讓這件事變簡單了，比如以官網的這個例子：

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved #{meters}m."

class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5

class Horse extends Animal
  move: ->
    alert "Galloping..."
    super 45

sam = new Snake "Sammy the Python"
tom = new Horse "Tommy the Palomino"

sam.move()
tom.move()
```

在往常不容易寫的漂亮的 JavaScript OO，可以被包裝得相當乾淨好維護。

Asset Pipeline 的架構

Asset Pipeline 對於 assets 位置的定義。根據預設，你可以把 assets 放在以下三個資料夾內：

- app/assets
- lib/assets
- vendor/assets

理論上，你把 assets 丟在這三個資料夾內，在 application.css / application.js 內 require 都可以動。

掛上其他 library

要引用 3rd party vendor assets，只要在 application.css 或者 application.js 進行 require 就可以使用了。

```
1 //= require jquery
2 //= require bootstrap
```

目錄夾的分類

app/assets

在 Rails 3.1.x 之後的版本， rails g controler posts， 會自動在 assets/styelsheets/ 和 assets/-javascripts/ 中產生對應的 scss 與 coffeescript 檔案。所以 app/assets 是讓開發者放「自己為專案手寫的 assets」的地方。

lib/assets

lib 是 library 的簡寫，這裡是放 LIBRARY 的地方。所以如果你為專案手寫的 assets 漸漸形成了 library 規模，比如說 mixin 或者是自己為專案整理了簡單的 bootstrap，應該放在 lib/ 下。

vendor/assets

verdor 是「供應商」的意思，也就是「別人寫的」assets 都應該放在這裡。比如說：

- jquery.*.js
- fanfanfan icons
- tinymce / ckeditor

等等…

Rails Asset Gem

透過 Asset Pipleline 的架構，開發者可以很容易透過 Gem 的機制，將 3rd party 的 CSS Framework 打包封裝，直接掛在 Rails 專案裡面使用。

比如說知名的 Fontawesome 這個 icon 專案，也有自己的 Rails gem https://github.com/bokmann/font-awesome-rails

在 application.css 裡面掛上

```
1 //= require font-awesome
```

這樣就可以直接使用了。不需要像以前開發網站一樣，把整個 framwork COPY 到專案裡面，既髒又難維護。

Rails 4 with Asset Pipeline

在 Rails 4 上掛上 asset pipeline 跟 Rails 3 的方式有些許的不同。

Rails 3

在 Rails 3，asset gem 基本上是被放在 asset 這個 group 的。

```
group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.1'
  gem 'uglifier', '>= 1.0.3'
  gem "compass-rails"
end
```

Rails 4

但在 Rails 4 上，這些 gem 卻要移出來不放在 asset 這個 group 裡。另外若要使用 compass 的話，因為一些技術問題，也需要安裝 1.1.2 以上的 gem，才能正常引用 compass

```
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.0.0'
gem 'sass-rails', '~> 4.0.0'
gem "compass-rails", "~> 1.1.2"
```

總複習

到目前為止。這本書提到的主題有

- rvm
- pow
- CRUD
- RESTFUL
- 雙層 RESTFUL
- 登入（Devise)
- Scope
- Helper
- Partial
- Capistrano
- Asset Pipeline

這本書原始的想法，其實是把一個初學者需要自學「一年」（我沒有開玩笑）的主題，濃縮到幾個禮拜。所以基本上這本書不能用「看」的，必須要動手實作。並且重複練習 3 次以上。

所以看到這裡，還請你重新再來一次。（認真）

不過這次你可以嘗試不一樣的作法。

第二遍

每個 chapter 都開一個 git branch 實作。並且推到 Github 上。

第三遍

幫 Groupme 加上 admin 後台，以及其他功能。並且推到 Github 上。

相信做到第三遍之後，你應該可以自然而然看懂原本像天書一般的 Rails API。

推薦書單

基礎

如果你不懂任何 Ruby 或 Rails，你應該從這些書 / 教材開始練習：

- Code School Try Ruby
- Code School Try Git
- Code School Git Real
- Peepcode Meet Command Line
- Peepcode Advanced Command Line
- Zed Shaw Learn Ruby The Hard Way

Learning Rails

使用這些教材，寫出一個簡單 Application，例如一個小論壇...

- Code School Rails for Zombies Redux
- Code School Rails for Zombies 2

初階基礎網頁設計

- CodeSchool Try jQuery
- CodeSchool JavaScript Road Trip Part 1
- CodeSchool JavaScript Road Trip Part 1
- CodeSchool CSS Cross-Country
- Codecademy Javascripts

初階 Ruby on Rails

如果你不知道怎麼開始使用 TDD 寫程式，可以從這幾本書開始...

- Michael Hartl Rails Turtorial
- Ryan Bigg Rails in Action 4

UT on Rails is also a excellent learning material

- Schneems UT on Rails

測試 Testing

- Code School Rails testing for zombies
- Code School Testing with Rspec
- Noel Rappin Rails Test Prescriptions: Keeping Your Application Healthy
- Thougutbot Learn Test-Driven Development using RSpec and Capybara.

進階基礎網頁設計

- Code School Jounry into Mobile
- Code School The Anatomy of Backbone
- Code School CoffeeScript
- Code School Assembling Sass
- Code School Assembling Sass Part2

重構 Ruby / Rails code

- Codschool Rails Best Practices
- Chad Pytel / Tammer Saleh : Rails Antipattern
- John Athayde / Bruce Williamsp The Rails View: Create a Beautiful and Maintainable User Experience
- Eric Davis Refacotoring Redmine
- Code Climate 7 Patterns to Refactor Fat ActiveRecord Models

寫出更漂亮的 Ruby code

- Code School Code Ruby Bits
- Code School Code Ruby Bits Part 2
- David A. Black The Well-Grounded Rubyist
- Russ Olsen Eloquent Ruby
- Avdi Grimm Confident Ruby
- Avdi Grimm Exceptional Ruby
- Stefan Kaes Writing Efficient Ruby Code (Digital Short Cut)

Podcast / Journal of writing better Ruby/Rails code

- Ruby Tapas
- Destroy All Software
- Practicing Ruby

Object-oriend Design in Ruby on Rails

- thoughtbot Ruby Science
- Avdi Grimm Object on Rails
- Russ Olsen Design Patterns in Ruby
- Jay fields Refacoting : Ruby Edition
- Sandi Metz Practical Object-Oriented Design in Ruby: An Agile Primer

如何更瞭解 Rails 底層

- José Valim Crafting Rails Applications: Expert Practices for Everyday Rails Development
- Marc-André Cournoyer Owning Rails: The Rails Online Master Class
- Railscast Rails Initialization Walkthrough
- Railscast Rails Middleware Walkthrough
- Railscast Rack App from Scratch
- Railscast Rails Modularity
- Railscast Hacking with Arel
- Railscast Authorization from Scratch Part 1
- Railscast Authorization from Scratch Part 2
- Railscast Action Controller Walkthrough
- Railscast Action View Walkthrough

附錄

Resources of latest Ruby

- Ruby5
- Ruby Weekly
- Ruby Inside
- RubyFlow
- RubyRogues
- Thoughtbot Podcast
- Railscast
- Confreaks