前言

我開始使用 Laravel 的故事很簡單。在使用它之前，我已經寫了好幾年的 PHP，但也一直盼望 Rails 與其他現代 web 框架的強大功能。Rails 擁有活躍的社群、完美地結合主觀性強大的內定值與彈性，而且擁有強大 Ruby Gems，可善用預先包裝的常用程式碼。

但是有些事情讓我跳槽了，當我發現 Laravel 時，我覺得很開心。它具備 Rails 吸引我的所有元素，但它並非只是 Rails 的複製品，而是一個創新的框架，擁有了不起的文件、友善的社群，顯然也受了許多語言與框架的影響。

我從那天開始，就開始用部落格與 Podcast 分享學習 Laravel 的旅程，並且在各個會議上演說；我用 Laravel 寫了數十個工作及業餘 app，並且與上千位 Laravel 開發者在網路上交談或直接碰面。我的工具組裡面有許多工具，但坦白說，我最喜歡在命令列上輸入 `laravel new projectName`。

本書的內容

這不是第一本探討 Laravel 的書籍，也不會是最後一本。我不想要在本書中涵蓋每一行程式或每一個模式實作。我不希望這本書是那種在新版的 Laravel 發表時就過時的書籍，而是希望透過它來提供高階的概述與具體的範例，讓開發人員知道，當他們需要在任何一個 Laravel 基礎程式中工作時，需要用哪些 Laravel 功能與子系統來做哪些事情。我希望協助你瞭解 Laravel 背後的基本概念，而非只是根據文件照本宣科一番。

Laravel 是一種強大且彈性的 PHP 框架。它有蓬勃的社群與廣泛的工具生態系統，因此它的吸引力與影響力也愈來愈大。本書的對象是已經知道如何製作網站與應用程式，並且想要快速使用 Laravel 來製作它們的開發者。

Laravel 的文件既詳盡且優秀。如果你覺得我對任何主題所做的介紹不夠深入，建議你可以參考線上文件（*https://laravel.com/docs*），更深入地瞭解該主題。

你應該會覺得這本書在高階的介紹與具體的應用之間取得令人舒適的平衡點，讀完這本書之後，你可以輕鬆地從頭開始使用 Laravel 來編寫完整的應用程式。而且，如果我善盡職責，你會很興奮地進行嘗試。

本書對象

本書假設你擁有基本的物件導向程式實務知識、PHP（或至少知道 C 家族的一般語法），與 Model-View-Controller（MVC）模式與模板化的基本概念。從未做過網站的人可能會覺得力不從心，但是只要你有一些程式設計經驗，在閱讀這本書之前，就不需要知道任何關於 Laravel 的知識，我們會從最簡單的“Hello, world!”開始，介紹你需要知道的所有事情。

Laravel 可以在任何作業系統上執行，但本書的 bash（殼層）指令是最容易在 Linux/Mac OS 上執行的那些。Windows 用戶比較難以使用這些指令進行現代 PHP 開發，但如果你按照指示來啟動 Homestead（Linux 虛擬機器），就可以在那裡執行所有的指令。

本書的架構

這本書是按照我想像的時間順序來架構的，如果你想要使用 Laravel 來建構自己的第一個 web app，前面幾章有你必須先瞭解的基本元素，後面的章節則介紹較進階或較複雜的功能。

本書的每一個部分都可以視為獨立的單元來閱讀，但我嘗試為不熟悉這個框架的讀者排列這些章節，讓他們可以用合理的閱讀讀序，從頭讀到結束。

在適當的情況下，每一章的結尾都有兩個小節：“測試”與“TL;DR”。“TL;DR”的意思是“過長；未讀”。這兩個最終的小節將告訴你如何為各章介紹的功能編寫測試程式，並在高階的層面上概述各章的內容。

本書是為 Laravel 5.8 而寫的，但也涵蓋 Laravel 5.1 以上的功能與語法變動。

第二章

設置 Laravel 開發環境

PHP 之所以成功，部分的原因在於你很難找到一台無法提供 PHP 的 web 伺服器。但是，現代的 PHP 工具比過往的工具有更嚴苛的要求。開發 Laravel 時，最好讓程式在彼此一致的本地及遠端伺服器環境運行，幸運的是，Laravel 生態系統已經有一些工具可以滿足這個需求了。

系統需求

我們在這一章討論的程式都可以在 Windows 機器上執行，但你需要閱讀好幾十頁的自訂指令與注意事項。請使用 Windows 的人自行閱讀這些指令與注意事項，在此與本書其餘部分的範例，都是針對 Unix/Linux/Mac OS 開發者。

無論你選擇在本地機器上安裝 PHP 及其他工具來提供網站服務、透過 Vagrant 或 Docker 從虛擬機器提供開發環境服務，或使用 MAMP/WAMP/XAMPP 等工具，你的開發環境都需要依序安裝以下程式來提供 Laravel 網站服務：

- 使用 Laravel 5.6 至 5.8 版時，PHP >= 7.1.3；使用 5.5 版時，PHP >= 7.0.0；使用 5.4 版時，PHP >= 5.6.4；使用 5.3 版時，PHP 的版本為 5.6.4 及 7.1.*，使用 5.2 與 5.1 版時，PHP >= 5.5.9
- OpenSSL PHP 擴展
- PDO PHP 擴展
- Mbstring PHP 擴展
- Tokenizer PHP 擴展

- XML PHP 擴展（Laravel 5.3 以上）
- Ctype PHP 擴展（Laravel 5.6 以上）
- JSON PHP 擴展（Laravel 5.6 以上）
- BCMath PHP 擴展（Laravel 5.7 以上）

Composer

無論你用哪一種電腦進行開發，都要全域安裝 Composer（*https://getcomposer.org/*）。Composer 是多數的現代 PHP 開發過程中都用得到的基本工具。Composer 是 PHP 的依賴項目管理器，很像 Node 的 NPM，或 Ruby 的 RubyGems。但是與 NPM 一樣，Composer 也是許多測試程式、本地腳本載入、安裝腳本，以及許多其他要素的基礎。有 Composer 才能安裝 Laravel、更新 Laravel，及帶入外部依賴項目。

本地開發環境

對許多專案而言，使用簡單的工具組來承載開發環境就夠了。當你安裝 MAMP 或 WAMP 或 XAMPP 之後，你應該就可以執行 Laravel 了。如果在你的系統上的 PHP 是正確的版本，你也可以使用 PHP 內建的 web 伺服器來直接執行 Laravel。

為了開始使用，你真正要做的只有讓 PHP 可以執行。除此之外的任何事情都是由你自己決定。

但是，Laravel 提供兩種工具來供你進行本地開發：Valet 與 Homestead，我會簡單地介紹兩者。如果你不確定該使用哪一種，我建議使用 Valet，你只要大略看一下 Homestead 小節就好，不過，這兩種工具是有價值且值得瞭解的。

Laravel Valet

如果你想要使用 PHP 的內建 web 伺服器，最簡單的選項就是從 *localhost* URL 伺服每一個網站。當你在 Laravel 網站的根資料夾執行 `php -S localhost:8000 -t` 時，PHP 內建的 web 伺服器會在 *http://localhost:8000/* 伺服你的網站。你也可以在設定應用程式之後執行 `php artisan serve`，快速地啟動等效的伺服器。

但如果你想要使用特定的開發網域來嘗試各個網站，就要瞭解作業系統的承載檔案，並使用 dnsmasq（*http://bit.ly/2eNPJ5T*）這類的工具。我們先來試一下比較簡單的東西。

如果你使用 Mac（也有供 Windows 與 Linux 使用的非官方分支版本），使用 Laravel Vale 可免於將網域連接至應用程式資料夾。Valet 會安裝 dnsmasq 與一系列的 PHP 腳本，可讓你輸入 `laravel new myapp && open myapp.test`，並讓它**可以動作**。如果你使用 Homebrew 的話，就要安裝一些工具，文件會輔助你進行，但從最初的安裝到伺服 app 所需的步驟比較少，也比較簡單。

安裝 Valet（見最新的安裝說明文件（*http://bit.ly/2U7uy7b*），並將它指向你將放置網路的一或多個目錄。我在 *~/Sites* 目錄中執行 `valet park`，那是我放置正在開發的 app 的地方。你可以在目錄名稱的後面加上 *.test*，並用瀏覽器來造訪它。

Valet 可讓你輕鬆地使用 `valet park` 來伺服 *{foldername}.test* 內的所有資料夾、使用 `valet link` 來伺服單一資料夾、使用 `valet open` 來打開 Valet 用資料夾伺服的網域、使用 `valet secure` 來以 HTTPS 伺服 Valet 網站，以及使用 `valet share` 來開啟一個 ngrok 通道，與他人共享網站。

Laravel Homestead

Homestead 是另一種設定本地開發環境的工具。它是以 Vagrant（管理虛擬機器的工具）為基礎的組態工具，提供預先設置好的虛擬機器映像，這個映像是完全為了 Laravel 開發而設定的，**並且**對映最常見的生產環境（許多 Laravel 網站都是在這種環境上運行的）。對使用 Windows 電腦的開發者而言，Homestead 應該是最好的本地開發環境。

Homestead 的文件（*http://bit.ly/2FwQ7EZ*）很詳盡，而且經常更新內容，所以如果你想要學習它的工作方式以及如何設定它，建議你參考那些文件。

Vessel

雖然不是 Laravel 官方專案，但 Servers for Hackers（*https://serversforhackers.com/*）及 Shipping Docker（*https://shippingdocker.com/*）的 Chris Fidao 創造了簡單的工具 Vessel（*https://vessel.shippingdocker.com/*），可用來建立 Docker 環境以便進行 Laravel 開發。欲知詳情，請參考 Vessel 文件。

建立 Laravel 新專案

建立 Laravel 新專案的方式有兩種，它們都是在命令列上執行的。第一種是全域安裝 Laravel 安裝器工具（使用 Composer），第二種是使用 Composer 的 `create-project` 功能。

你可以到 Installation 文件網頁瞭解這兩種做法的細節（*http://laravel.com/docs/installation*），不過我比較建議使用 Laravel 安裝工具。

使用 Laravel 安裝工具來安裝 Laravel

如果你已經全域安裝 Composer 了，執行以下的命令即可安裝 Laravel 安裝工具：

```
composer global require "laravel/installer"
```

當你安裝 Laravel 安裝工具之後，啟動新的 Laravel 專案就很簡單了。你只要在命令列執行這個命令：

```
laravel new projectName
```

就會在目前的目錄內建立一個新的子目錄，稱為 *{projectName}*，並在裡面安裝一個 Laravel 專案。

使用 Composer 的 create-project 功能來安裝 Laravel

Composer 也提供一種稱為 `create-project` 的功能，讓你可用特定的骨架來建立新專案。執行以下的命令即可使用這個工具來建立新的 Laravel 專案：

```
composer create-project laravel/laravel projectName
```

如同安裝器工具，它會在你目前的目錄內建立一個子目錄，稱為 *{projectName}*，裡面裝了一個 Laravel 骨架，可讓你用來開發。

Lambo：超強的“Laravel New”

因為我經常在建立新的 Laravel 專案之後執行一系列相同的步驟，所以我做了一個簡單的腳本，稱為 Lambo（*http://bit.ly/2TCcQo8*），以便在每次建立新專案時自動執行這些步驟。

Lambo 會執行 `laravel new`，接著將你的程式碼提交至 Git，用合理的預設值設定 *.env* 憑證，在瀏覽器裡面打開專案，並且（可選）在你的編譯器裡面打開它，以及執行一些其他實用的組建步驟。

你可以用 Composer 的 `global require` 來安裝 Lambo：

```
composer global require tightenco/lambo
```

也可以像使用 `laravel new` 那樣使用它：

```
cd Sites
lambo my-new-project
```

Laravel 的目錄結構

當你打開含有 Laravel 骨架的目錄之後，會看到以下的檔案與目錄：

```
app/
bootstrap/
config/
public/
resources/
routes/
storage/
tests/
vendor/
.editorconfig
.env
.env.example
.gitattributes
.gitignore
artisan
composer.json
composer.lock
package.json
phpunit.xml
readme.md
server.php
webpack.mix.js
```

Laravel 5.4 之前的其他組建工具

在使用 Laravel 5.4 之前的版本建立的專案裡面，你可能會看到 *gulpfile.js*，而非 *webpack.mix.js*；這代表專案正在運行 Laravel Elixir（*http://bit.ly/2JCToYp*），而非 Laravel Mix（*http://bit.ly/2U4X09P*）。

我們來逐一瞭解它們。

資料夾

預設情況下，根目錄有以下的資料夾：

app

保存大部分的實際應用程式的地方。model、controller、路由定義、命令及你的 PHP 網域碼都在這裡。

bootstrap

存有 Laravel 框架在每次執行時，用來啟動的檔案。

config

放置所有組態檔的地方。

database

放置資料庫 migration、種子與工廠。

public

當伺服器伺服網站時指向的目錄。它裡面有 *index.php*，這是前端 controller，會開始啟動程序，並妥善地路由所有的請求。它也是保存所有可被公眾看見的檔案的地方，例如圖像、樣式表、腳本與下載檔案。

resources

保存其他腳本需要的檔案的地方。畫面、語言檔案與（可選）Sass/LESS/CSS 原始檔，及 JavaScript 原始檔案都在這裡面。

routes

存有所有的路由定義，包括 HTTP 路由與“主控台路由”或 Artisan 命令。

storage

保存快取、log 與編譯過的系統檔案。

tests

保存單元與整合測試。

vendor

Composer 安裝它的依賴項目的地方。Git 會忽略它（標記為被版本管理系統排除），因為 Composer 是在任何遠端伺服器上作為部署程序的一部分來執行的。

零星的檔案

根目錄也有以下的檔案：

.editorconfig

提供 Laravel 的編寫標準（例如縮排大小、字元集、是否移除尾部空格）給你的 IDE / 文字編輯器。你會在運行 5.5 以上版本的所有 Laravel app 中看到它。

.env 與 *.env.example*

主宰環境變數的檔案（在各個環境之下應該會不一樣的變數，因此不會被提交到版本管理系統）。*.env.example* 是個模板，每一個環境都應該複製它，來建立它們自己的 *.env* 檔，Git 會忽略它。

.gitignore 與 *.gitattributes*

Git 組態檔。

artisan

可讓你在命令列上執行 Artisan 命令的檔案（見第 8 章）。

composer.json 與 *composer.lock*

Composer 的組態檔；*composer.json* 是可讓用戶編輯的，*composer.lock* 不是。這些檔案共享同一些關於專案的基本資訊，也會定義它的 PHP 依賴關係。

package.json

很像 *composer.json*，但用於前端資產與組建系統的依賴項目；負責告訴 NPM 拉入哪個以 JavaScript 寫成的依賴項目。

phpunit.xml

PHPUnit 的組態檔，PHPUnit 是 Laravel 內建的測試工具。

readme.md

提供 Laravel 的簡介的 Markdown 檔。當你使用 Laravel 安裝器時不會看到這個檔案。

server.php

後備伺服器，它會試著讓能力較差的伺服器仍然可以預覽 Laravel 應用程式。

webpack.mix.js

（選用）Mix 組態檔。如果你使用 Elixir 的話，會變成看到 *gulpfile.js*。這些檔案負責告訴你的組建系統如何編譯與處理前端資產。

設置

Laravel app 的核心設定（包括資料庫連結設定、佇列與郵件設定等等）都位於 *config* 資料夾的檔案裡面。這些檔案都會回傳一個陣列，陣列的每一個值都可以用一個組態鍵來存取，這個組態鍵是由檔名與所有後繼鍵組成的，之間以句點分隔（`.`）。

所以，如果你在 *config/services.php* 裡面建立一個這樣的檔案：

```
// config/services.php
<?php
return [
    'sparkpost' => [
        'secret' => 'abcdefg',
    ],
];
```

就可以使用 `config('services.sparkpost.secret')` 來取得那一個組態變數。

所有因環境而異的組態變數（因此不會被提交到版本管理系統）都在 *.env* 檔案裡面。假設你想要在各個環境中使用不同的 Bugsnag API 鍵。你要設定組態檔，從 *.env* 將它拉入：

```
// config/services.php
<?php
return [
    'bugsnag' => [
        'api_key' => env('BUGSNAG_API_KEY'),
    ],
];
```

這個 `env()` 輔助函式會從你的 *.env* 檔案拉出那個鍵的值。現在將那個鍵加入你的 *.env*（設定這個環境）與 *.env.example*（所有環境的模板）檔案：

```
# 在 .env
BUGSNAG_API_KEY=oinfp9813410942

# 在 .env.example
BUGSNAG_API_KEY=
```

現在 *.env* 檔案裡面有一些框架需要的環境專屬變數了，例如你即將使用的郵件驅動程式，以及基本資料庫設定。

在組態檔外面使用 env()

當你在組態檔之外的任何地方使用 env() 呼叫式時，有些 Laravel 的功能，包括一些快取與最佳化功能，將無法使用。

拉入環境變數的最佳方式，就是幫環境專用的東西設定組態項目。讓這些組態項目讀取環境變數，接著在 app 內的任何地方引用組態變數：

```
// config/services.php
return [
    'bugsnag' => [
        'key' => env('BUGSNAG_API_KEY'),
    ],
];

// 在 controller 或其他地方
$bugsnag = new Bugsnag(config('services.bugsnag.key'));
```

.env 檔

接著簡單看一下 *.env* 檔的預設內容。確切的鍵將會根據你所使用的 Laravel 版本而改變，範例 2-1 是它們在 5.8 版的樣子。

範例 2-1　Laravel 5.8 的預設環境變數

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
```

```
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

BROADCAST_DRIVER=log
CACHE_DRIVER=file
QUEUE_CONNECTION=sync
SESSION_DRIVER=file
SESSION_LIFETIME=120

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
PUSHER_APP_CLUSTER=mt1

MIX_PUSHER_APP_KEY="${PUSHER_APP_KEY}"
MIX_PUSHER_APP_CLUSTER="${PUSHER_APP_CLUSTER}"
```

我不一一介紹它們，因為有些只是各種服務（Pusher、Redis、DB、Mail）的身分驗證資訊群組。不過你必須認識這兩個重要的環境變數：

`APP_KEY`

用來加密資料的隨機生成字串。如果它是空的，你可能會看到錯誤訊息“No application encryption key has been specified”。此時，你只要執行 `artisan key:generate`，Laravel 就會幫你製作一個。

`APP_DEBUG`

決定這個 app 實例的用戶是否應該看到 debug 錯誤——很適合本地及預備環境，絕對不適合生產環境。

其餘的非身分驗證設定（BROADCAST_DRIVER、QUEUE_CONNECTION 等）都被設為盡量不依靠外部服務的內定值，這對新手而言是最適合的設定。

當你啟動第一個 Laravel app 時，對大部分的專案而言，你唯一需要改變的地方可能是設定資料庫組態。我使用 Laravel Valet，所以將 DB_DATABASE 改成我的專案的名稱，將 DB_USERNAME 改成 root，將 DB_PASSWORD 改成空字串：

```
DB_DATABASE=myProject
DB_USERNAME=root
DB_PASSWORD=
```

我在我最喜歡的 MySQL 用戶端裡面用我的專案名稱建立一個資料庫，這樣就一切就緒了。

啟動並執行

現在要來啟動並執行基本的 Laravel 安裝版。執行 git init，使用 git add . 與 git commit 來提交基本檔案，你就可以開始寫程式了。就是這樣！如果你使用 Valet，可以執行以下的命令，立刻在瀏覽器中看到網站的運行：

```
laravel new myProject && cd myProject && valet open
```

每次我開始一個新專案時，都會執行以下的步驟：

```
laravel new myProject
cd myProject
git init
git add .
git commit -m "Initial commit"
```

我將所有網站都放在 *~/Sites* 資料夾內，我已經將它設為主要的 Valet 目錄了，所以在這個例子中，我可以立刻在瀏覽器造訪 *myProject.test*，不需要額外的工作。我可以編輯 *.env* 並將它指向一個特定的資料庫，在我的 MySQL app 中加入那個資料庫，就可以開始寫程式了。記得，如果你使用 Lambo，它已經為你完成以上的步驟了。

測試

之後每一章結尾的“測試”小節都會告訴你如何編寫程式來測試之前討論的功能。因為本章未討論可測試的功能，所以我們快速地說明一下測試。（要進一步瞭解如何用 Laravel 來編寫與執行測試，可前往第 12 章。）

Laravel 原本就有 PHPUnit 的依賴項目，並且已經將它設定為針對 *tests* 目錄內，名稱結尾為 *Test.php* 的所有檔案進行測試（例如 *tests/UserTest.php*）。

所以，編寫測試最快的方式，就是在 *tests* 目錄中使用結尾為 *Test.php* 的檔名來建立一個檔案。執行它們最簡單的做法是在命令列執行 `./vendor/bin/phpunit`（在專案根目錄）。

如果有任何測試需要存取資料庫，務必在承載資料庫的機器上執行測試，所以，如果你在 Vagrant 承載資料庫，請 ssh 到你的 Vagrant 箱子，並在那裡執行測試。同樣地，你可以在第 12 章進一步瞭解這個部分。

此外，如果你初次閱讀本書，有些測試小節會使用你還不熟悉的測試語法及功能，如果你看不懂測試小節的程式，可先跳過它，等你看完測試章節之後，再回來閱讀。

TL;DR

Laravel 是一種 PHP 框架，所以在本地伺服它很簡單。Laravel 提供兩種工具來管理本地開發：較簡單的工具稱為 Valet，它使用本地機器來提供依賴項目，另一種是 Homestead，它是預先設置的 Vagrant。Laravel 依賴 Composer，而且可以使用 Composer 來安裝，它內建一系列的資料夾與檔案，以反映其慣例以及和其他開放原始碼工具之間的關係。

第三章

路由與 controller

任何一種 web 應用程式框架的基本功能都是從用戶接收請求，並且提供回應，通常是透過 HTTP(S)。這意味著，當你學習 web 框架時，定義應用程式的路由是最重要，而且需要優先處理的事情。如果沒有路由，你就幾乎無法與用戶互動。

在這一章，我們要來看一下 Laravel 的路由，並展示如何定義它們、如何將它們指向它們應該執行的程式碼，及如何使用 Laravel 的路由工具來處理路由需要的各種陣列。

簡介 MVC、HTTP 動詞與 REST

本章大多數的內容都參考 Model–View–Controller（MVC）app 的建構方式，而且有許多範例都使用 REST 的路由名稱與動詞，所以我們來簡單介紹兩者。

什麼是 MVC ？

MVC 有三個主要概念：

model

代表單獨的資料庫資料表（或資料庫的一筆紀錄），你可以將它想成“Company”或“Dog”。

view

代表將你的資料輸出給最終用戶的模板，你可以將它想成“用 HTML、CSS 與 JavaScript 組成的登入網頁模板”。

controller

就像交通警察，從瀏覽器接收 HTTP 請求，從資料庫及其他儲存機制取出正確的資料，驗證用戶輸入，最後將回應送給用戶。

在圖 3-1 中，你可以看到最終用戶會先用他們的瀏覽器傳送 HTTP 請求，來跟 controller 互動。為了回應那個請求，controller 可能會將資料寫入 model（資料庫）或從那裡讀出資料。接下來 controller 可能會傳送資料給 view，接著 view 會被傳給最終用戶，在他們的瀏覽器上顯示出來。

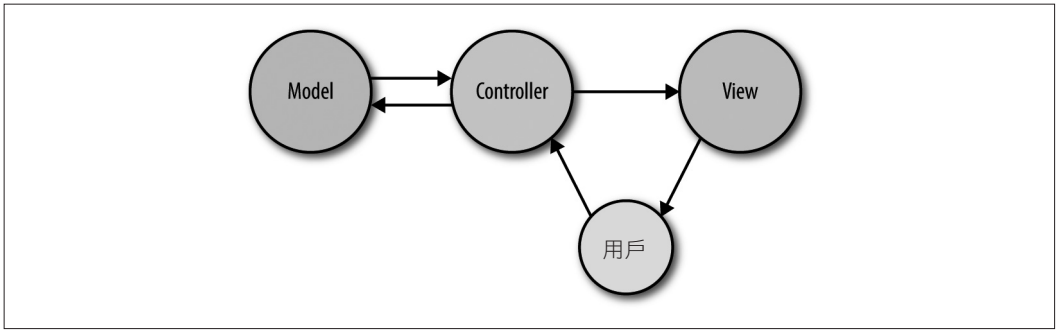

圖 3-1　MVC 的基本說明

稍後介紹的一些 Laravel 使用案例不適合這種相對簡單的 app 架構，所以無須太關注 MVC，但至少它可以為你做好準備，在本章其他地方討論 view 與 controller 時瞭解它們。

HTTP 動詞

最常見的 HTTP 動詞是 `GET` 與 `POST`，接下來是 `PUT` 與 `DELETE`。此外還有 `HEAD`、`OPTIONS` 與 `PATCH`，以及幾乎不會在一般的 web 開發中使用的 `TRACE` 及 `CONNECT`。

簡單地介紹如下：

`GET`

請求一項資源（或一系列資源）。

`HEAD`

請求只有標頭的 `GET` 回應。

POST

建立資源。

PUT

覆寫資源。

PATCH

修改資源。

DELETE

刪除資源。

OPTIONS

詢問伺服器這個 URL 可用的動詞有哪些。

表 3-1 是資源控制器提供的動作（第 48 頁的“資源控制器”會深入介紹）。每一項動作都期望你用特定的動詞來呼叫特定的 URL 格式，因此你可以大致瞭解各個動詞的用途為何。

表 3-1　Laravel 資源控制器的方法

動詞	URL	controller 方法	名稱	說明
GET	tasks	index()	tasks.index	顯示所有工作
GET	tasks/create	create()	tasks.create	顯示建立工作表單
POST	tasks	store()	tasks.store	從建立工作表單接收表單提交
GET	tasks/{task}	show()	tasks.show	顯示一項工作
GET	tasks/{task}/edit	edit()	tasks.edit	編輯一項工作
PUT/PATCH	tasks/{task}	update()	tasks.update	從編輯工作表單接收表單提交
DELETE	tasks/{task}	destroy()	tasks.destroy	刪除一項工作

什麼是 REST ？

我們會在第 349 頁的“類 REST JSON API 基礎”詳細說明 REST，簡單來說，它是一種建構 API 的架構風格。本書談到的 REST 主要有以下的特徵：

- 一次圍繞著一項主要資源來架構（例如 tasks）
- 由可預測的 URL 結構使用 HTTP 動詞（表 3-1）來進行的互動組成
- 回傳 JSON，並且通常是用 JSON 來請求的

此外還有一些特徵，但通常是“RESTful”，在本書，它的意思是“使用這種 URL 結構格式，因此我們可以發出可預測的呼叫，例如編輯網頁的 `GET /tasks/14/edit`”。這很重要（即使不是在建構 API 時），因為 Laravel 的路由結構是以類 REST 結構為基礎，如表 3-1 所示。

以 REST 為基礎的 API 主要遵循同樣的結構，不過它們沒有**建立**（*create*）路由或**編輯**（*edit*）路由，因為 API 只代表動作，而不是為動作而準備的網頁。

路由定義

在 Laravel 應用程式中，你會在 *routes/web.php* 裡面定義 web 路由，在 *routes/api.php* 中定義 API 路由。web 路由是讓最終用戶造訪的，API 路由是讓你的 API 使用的。目前我們的重點是 *routes/web.php* 內的路由。

Laravel 5.3 之前的路由檔案位置

使用 Laravel 5.3 之前的版本的專案只有一個路由檔案，位於 *app/Http/routes.php*。

定義路由最簡單的方式是使用 closure 來匹配路徑（例如 `/`），如範例 3-1 所示。

範例 3-1　基本的路由定義

```
// routes/web.php
Route::get('/', function () {
    return 'Hello, World!';
});
```

什麼是 closure ？

closure 是 PHP 版的匿名函式。你可以將這種函式當成物件四處傳遞、將它指派給變數、將它當成參數傳給其他函式或方法，甚至將它序列化。

我們剛才定義：每當有人造訪 `/`（網域的根目錄）時，Laravel 的路由器就會執行在此定義的 closure，並回傳結果。注意，我們是**回傳**內容，而不是 `echo` 或 `print` 它。

中間軟體簡介

你可能會覺得奇怪“為什麼要回傳‘Hello, World!’，而不是 echo 它？”

答案不只一個，最簡單的一種在是：Laravel 的請求與回應週期被很多東西包著，包括一些稱為**中間軟體**（*middleware*）的東西。當路由 closure 或 controller 方法完成工作時，還不會將輸出傳送給瀏覽器，而是回傳內容，讓它繼續流經回應堆疊及 middleware，再回傳給用戶。

許多簡單的網站都可以全部在 web 路由檔案內定義。如範例 3-2 所示，只要使用一些簡單的 GET 結合一些模板，你就可以輕鬆地伺服一個典型的網站了。

範例 *3-2*　網站範例

```
Route::get('/', function () {
    return view('welcome');
});

Route::get('about', function () {
    return view('about');
});

Route::get('products', function () {
    return view('products');
});

Route::get('services', function () {
    return view('services');
});
```

靜態呼叫

如果你有豐富的 PHP 開發經驗，當你在 Route 類別內看到靜態呼叫時或許會很驚訝。它本身其實不是個靜態方法，而是使用 Laravel 靜態介面（facade）的服務位置（service location），第 11 章將會介紹它。

如果你不想用靜態介面，可以用這種方式來做出相同的定義：

```
$router->get('/', function () {
    return 'Hello, World!';
});
```

路由動詞

你可能已經發現，我們在路由定義中使用 Route::get。它的意思是，我們要求 Laravel 只在 HTTP 請求使用 GET 動作時，才匹配這些路由。但如果它是個表單 POST，或是一些 JavaScript 傳送的 PUT 或 DELETE 請求時，又該如何？你也可以對著路由定義呼叫一些其他的方法，如範例 3-3 所示。

範例 3-3　路由動詞

```
Route::get('/', function () {
    return 'Hello, World!';
});

Route::post('/', function () {
    // 處理有人傳送一個 POST 請求到這個路由的情況
});

Route::put('/', function () {
    // 處理有人傳送一個 PUT 請求到這個路由的情況
});

Route::delete('/', function () {
    // 處理有人傳送一個 DELETE 請求到這個路由的情況
});

Route::any('/', function () {
    // 處理被傳到這個路由的任何動詞請求
});

Route::match(['get', 'post'], '/', function () {
    // 處理被傳到這個路由的 GET 或 POST 請求
});
```

路由處理

你可能已經猜到了，除了傳遞 closure 給路由定義式之外，你也可以用其他方式教它如何解析路由。closure 既快速且簡單，但隨著應用程式越來越大，將所有的路由邏輯都放入一個檔案會讓它變得很臃腫。此外，使用路由 closure 的應用程式無法利用 Laravel 的路由快取（稍後會進一步說明）的好處，快取可讓每一個請求減少上百毫秒的時間。

另一種常見的做法，是改用字串來傳遞 controller 的名稱與方法，取代 closure，如範例 3-4 所示。

範例 3-4　路由呼叫 controller 方法

```
Route::get('/', 'WelcomeController@index');
```

這會要求 Laravel 將送到那個路徑的請求傳給 `App\Http\Controllers\WelcomeController` controller 的 `index()` 方法。這個方法收到的參數以及被處理的方式將會與你在這個位置使用 closure 時相同。

Laravel 的 controller / 方法參考語法

Laravel 規範了如何引用特定 controller 的特定方法：*ControllerName@methodName*。有人將它當成臨時性的通訊規範，但也有人將它用在實際的綁定，如範例 3-4 所示。Laravel 會解析 `@` 之前與之後的文字，並使用它們來辨識 controller 與方法。Laravel 5.7 也加入“tuple”語法（`Route::get('/', [WelcomeControl ler::class, 'index'])`），但比較常見的做法，仍然是使用 *ControllerName@methodName* 來描述方法。

路由參數

如果你定義的路由有參數（在 URL 結構之中有變數區段），則很容易在路由裡面定義它們並將它們傳入 closure（見範例 3-5）。

範例 3-5　路由參數

```
Route::get('users/{id}/friends', function ($id) {
    //
});
```

你也可以在參數名稱後面加上一個問號（`?`）來讓它變成選用的，如範例 3-6 所示。在這種情況下，你也要提供預設值給路由的對應變數。

範例 3-6　選用的路由參數

```
Route::get('users/{id?}', function ($id = 'fallbackId') {
    //
});
```

你也可以使用正規表達式（regexes）來定義唯有在某個參數符合特定情況下才匹配路由，如範例 3-7 所示。

範例 *3-7* 使用正規表達式來限制路由

```
Route::get('users/{id}', function ($id) {
    //
})->where('id', '[0-9]+');

Route::get('users/{username}', function ($username) {
    //
})->where('username', '[A-Za-z]+');

Route::get('posts/{id}/{slug}', function ($id, $slug) {
    //
})->where(['id' => '[0-9]+', 'slug' => '[A-Za-z]+']);
```

你可以猜到，當你造訪的路徑匹配路由字串，但 regex 不匹配參數時，該路徑就不匹配。因為路線的匹配方式是由上到下，users/abc 會跳過範例 3-7 的第一個 closure，但它會被第二個 closure 匹配，所以它會走那一個路線。另一方面，posts/abc/123 不匹配任何 closure，所以它會回傳 404 Not Found 錯誤。

路由參數與 Closure / Controller 方法的參數名稱之間的關係

從範例 3-5 可以看到，大部分的人都讓路由參數（{id}）與注入路由定義的方法參數（function ($id)）使用相同的名稱。這是必要的嗎？

除非你使用本章稍後介紹的路由 model 綁定，否則不一定需要如此。定義哪個路由參數對映哪個方法參數的唯一規則就是它們的順序（從左到右），例如：

```
Route::get('users/{userId}/comments/{commentId}', function (
    $thisIsActuallyTheUserId,
    $thisIsReallyTheCommentId
    ) {
    //
});
```

不過，可以使用不同的名稱不代表你**應該**這樣做。為了替將來的開發人員著想，建議你讓它們使用相同的名稱，因為開發人員可能會為了不一致的名稱而頭痛不已。