前言

Flask 在眾多框架中鶴立雞群，因為它可讓開發者掌握方向盤，完全控制 app 的創新。你或許聽過“與框架拉扯”這句話。當你想要用非官方手段來解決問題時，多數的框架都會讓你有這種感覺，原因可能出在你想要用不同的資料庫引擎，或用不同的方法來驗證使用者，每當你想要採取有異於框架開發者的做法時都會頭痛不已。

但是 Flask 並非如此。你喜歡關聯式資料庫嗎？太好了！ Flask 支援所有的關聯式資料庫！你比較喜歡 NoSQL 資料庫？絕對沒問題，Flask 也可以使用它們。想要用你自製的資料庫引擎？完全不想使用資料庫？全部沒問題。使用 Flask 時，你甚至可以編寫自己的 app 零組件。一切毫無問題！

這麼自由的關鍵在於 Flask 最初的設計就是要讓人擴充的。它具備一個穩健的核心，這個核心包含所有 web app 都需要的基本功能，並且可讓其他的功能採用第三方提供的擴充功能，當然，第三方包括你！

你可以在本書看到我用 Flask 來開發 web app 的流程，這個流程當然不是用這個框架來建構 app 的唯一方式。你應該將我的做法當成建議，而不是教條。

多數的軟體開發書籍都提供小型且專注於某個主題的範例程式來獨立展示該技術的某種功能，使得讀者必須自行摸索將這些功能變成完整、可運作的 app 所需的“膠水”程式。我採取不同的方法，本書的範例都是同一個 app 的某個部分，最初這個 app 很簡單，但是它會隨著章節的進行而逐漸擴充。這個 app 一開始只有幾行程式，最後會成為一個功能完整的部落格與社群網路 app。

本書對象

你需要有一些 Python 程式寫作經驗才能閱讀本書的多數內容。雖然本書預設讀者可以不具備任何 Flask 知識，但你應該充分瞭解一些 Python 概念，例如套件、模組、函式、裝飾器（decorator），以及物件導向程式設計。瞭解異常（exception）以及如何用堆疊追蹤（stack trace）來尋找問題對你也有很大的幫助。

當你研究本書的範例時，會花很多時間在命令列上。你必須熟悉作業系統的命令列。

現代的 web app 難免會用到 HTML、CSS 與 JavaScript。本書開發的 app 當然也會使用它們，但本書不會詳細說明這些技術及其用法。如果你想要自行開發完整的 app 且不想要麻煩瞭解用戶端技術的開發者，建議你熟悉這些語言。

我將本書的 app 原始碼公開放在 GitHub 上。雖然你可以在 GitHub 以 ZIP 或 TAR 檔案的格式下載 app，但強烈建議你安裝 Git 用戶端，並熟悉原始檔版本控制系統（至少瞭解從存放區直接 clone 和 check out 各種版本的基本命令）。第 xv 頁的“如何使用範例程式”有你需要的命令清單。既然你以後一定會在自己的專案中使用版本控制系統，那就將閱讀本書當成學習 Git 的藉口吧！

最後，本書不是完整且詳盡的 Flask 框架參考書，雖然本書會討論多數的功能，但你應該在閱讀的過程配合官方的 Flask 文件（*http://flask.pocoo.org*）。

本書的架構

本書有三個部分。

第一部分**詳細介紹** *Flask* 會用 Flask 框架與它的一些擴充功能來開發 web app，讓你從中學習基本知識：

- 第 1 章會說明如何安裝與設定 Flask 框架。
- 第 2 章會用基本的 app 來深入說明 Flask。
- 第 3 章介紹如何在 Flask app 中使用模板。
- 第 4 章介紹 web 表單。
- 第 5 章介紹資料庫。
- 第 6 章介紹 email 支援。

- 第 7 章展示適合中大型 app 的架構。

在第二部分**範例：社群部落格 *app*** 中，我會用 Flasky 來建構專為本書開發的開放原始碼部落格社群網路 app：

- 第 8 章製作使用者身分驗證系統。
- 第 9 章製作使用者角色與使用權限。
- 第 10 章製作使用者個人資訊網頁。
- 第 11 章建立部落格介面。
- 第 12 章製作追隨者。
- 第 13 章製作部落格文章的使用者評論。
- 第 14 章製作應用程式開發介面（API）。

第三部分，**最後一里路**會說明一些在公開 app 之前應該執行的重要工作，它們與編寫 app 程式沒有直接關係：

- 第 15 詳細說明各種單元測試策略。
- 第 16 章概述效能分析技術。
- 第 17 章說明 Flask app 的部署選項，包括傳統、雲端與容器解決方案。
- 第 18 章列舉其他的資源。

如何使用範例程式

本書的範例程式可從 *https://github.com/miguelgrinberg/flasky* 下載。

我很細心地建構這個存放區的提交紀錄（commit history），讓它符合本書的順序。建議你在使用這些程式時，先從最舊的提交版本開始 check out，再按照提交清單，隨著閱讀的進度依續 check out。或者，GitHub 也可以讓你用 ZIP 或 TAR 檔來下載各個提交版本。

如果你要用 Git 來處理原始程式，就要先安裝 Git 用戶端。你可以在 *http://git-scm.com* 下載它，再用下面的 Git 命令下載範例程式：

```
$ git clone https://github.com/miguelgrinberg/flasky.git
```

`git clone` 命令會將 GitHub 的原始程式安裝至你在目前的目錄裡面建立的 *flasky2* 資料夾。這個目錄除了有原始程式碼之外，也有 Git 存放區的複本，包括我在這個 app 做過的所有修改紀錄。

第 1 章會帶領你 *check out* 這個 app 的最初版本，之後在適當的時機也會指示你 check out 後續的版本。check out 後續版本的 Git 命令是 `git checkout`。例如：

```
$ git checkout 1a
```

命令中的 `1a` 是**標籤**，它是專案提交紀錄的某一個時間點的名稱。這個存放區是根據書中的章節來標記的，所以本例的 `1a` 標籤代表這個 app 檔案是第 1 章使用的初始版本。多數的章節都有多個標籤，例如 `5a`、`5b` 等標籤是第 5 章的後續版本。

當你執行剛才的 `git checkout` 命令時，Git 會顯示一個警告訊息，說你正處於“detached HEAD”狀態，代表你目前查看的版本不是可以送出新版本的分支，而是位於專案更改歷程中間的某個版本。當你看到這個訊息時不用驚慌，但請記得，你無法在這個狀態之下修改任何檔案並發出另一個 `git checkout`，因為 Git 不知道如何處理你做的修改。所以，為了繼續使用這個專案，你必須將修改過的檔案還原成它們的原始狀態，最簡單的做法是使用 `git reset` 命令：

```
$ git reset --hard
```

這個命令會銷毀你在本地端做過的任何修改，所以在使用這個命令之前，你要儲存想要保留的東西。

除了 check out 某個版本的原始檔案外，有時你會執行其他的設定，例如安裝新的 Python 套件或更新資料庫，我們會在適當的時機告訴你做法。

有時你會將本地端的存放區更新為 GitHub 存放區的內容，以便套用 bug 的修復與程式的改善，此時可執行這個命令：

```
$ git fetch --all
$ git fetch --tags
$ git reset --hard origin/master
```

`git fetch` 命令的用途是將本地存放區的提交紀錄與標籤更新成 GitHub 上面的，不過它們不會影響實際的原始檔案，原始檔案要用它下面的 `git reset` 命令來更新。再次提醒你，每當你使用 `git reset` 時，就會失去你在本地端做過的任何更改。

另外還有一種好用的功能可讓你查看兩個版本之間的差異，來幫助你瞭解變動的細節：在命令列輸入 `git diff` 命令。例如，要查看 `2a` 與 `2b` 版本之間的差異，可使用：

```
$ git diff 2a 2b
```

它們的差異會用 *patch* 來展示，如果你不習慣使用 patch 檔，或許無法直接用它來瞭解差異，此時或許使用 GitHub 顯示的比較圖形比較容易瞭解。例如，你可以在 GitHub 的 *https://github.com/miguelgrinberg/flasky/compare/2a...2b* 查看 `2a` 與 `2b` 版本之間的差異。

使用範例程式

本書的目的是協助你完成工作。一般來說，你可以在自己的程式或文件中使用本書的程式碼而不需要聯繫出版社取得許可，除非你更動了程式的重要部分。舉例來說，為了撰寫程式而使用本書的幾段程式碼時，不需要取得授權，但是將 O'Reilly 書籍的範例製成光碟來銷售或散佈，就絕對需要我們的授權。引用這本書的內容與範例程式碼來回答問題不需要取得許可。在你的產品文件中加入本書大量的程式碼需要取得許可。

如果你在引用它們時能標明出處，我們會非常感激（但不強制要求）。在標明出處時，內容通常包括標題、作者、出版社與國際標準書號。例如：“*Flask Web 開發*，第二版，Miguel Grinberg 著（O'Reilly）。版權所有人 2018 Miguel Grinberg，978-1-491-99173-2”。

如果你覺得自己使用範例程式的程度超出上述的允許範圍，歡迎隨時與我們聯繫：*permissions@oreilly.com*。

本書編排方式

本書使用以下的編排規則：

斜體字（*Italic*）

代表新的術語、URL、電子郵件地址、檔案名稱及副檔名。中文以楷體表示。

定寬字（`Constant width`）

代表命令列輸出與程式，在文章中代表命令與程式元素，例如變數或函式名稱、資料庫、資料型態、環境變數、陳述式與關鍵字。

第二章

基本 app 結構

本章會讓你學到 Flask app 的各個零組件，並教你編寫和執行你的第一個 Flask web app。

初始化

所有的 Flask app 都必須建立一個 *app 實例*。web 伺服器會使用一種稱為 Web Server Gateway Interface（WSGI，讀成“wiz-ghee”）的協定來將用戶端傳來的所有請求送給這個物件來處理。app 實例是 `Flask` 類別的物件，建立它的方式通常是：

```
from flask import Flask
app = Flask(__name__)
```

`Flask` 類別建構式唯一需要的引數是 app 的主模組或套件的名稱。對多數 app 而言，這個引數的值是 Python 的 `__name__` 變數。

Flask 新手經常無法理解傳給 Flask app 建構式的 `__name__` 引數。Flask 會用這個引數來確定 app 的位置，以便找到構成 app 的其他檔案，例如圖像與模板。

稍後會說明比較複雜的 app 初始化方式，但是就簡單的 app 而言，上面的做法就夠了。

路由與 view 函式

網頁瀏覽器這類的用戶端會將 *request*（**請求**，譯注：本書用原文來表示名詞的 request，以幫助讀者理解）送給網頁伺服器，接下來網頁伺服器將它們送給 Flask app 實例。Flask app 實例必須知道應該用哪些程式來處理用戶端請求的每一個 URL，所以它有一個 URL 到 Python 函式的對應（mapping）。URL 與處理它的函式之間的關係稱為**路由**（*route*）。

在 Flask app 裡面定義路由最方便的方式是使用 app 實例公開的 `app.route` 裝飾器。下面的範例說明如何使用這個裝飾器來宣告路由：

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

裝飾器是 Python 語言的標準功能。裝飾器經常用來將函式註冊成在某些事件發生時呼叫的處理函式。

上面的範例會將函式 `index()` 註冊成 app 的根 URL 的處理函式。雖然 `app.route` 裝飾器是註冊 view 函式最好的方法，但是 Flask 也提供一種比較傳統的方式，用 `app.add_url_rule()` 方法來設定 app 路由，它最基本的形式接收三個引數：URL、端點名稱以及 view 函式。下面的範例使用 `app.add_url_rule()` 來註冊 `index()` 函式，其效果與上一個範例一樣：

```
def index():
    return '<h1>Hello World!</h1>'

app.add_url_rule('/', 'index', index)
```

像 `index()` 這種處理 app URL 的函式稱為 *view* **函式**。如果 app 是用 *www.example.com* 網域名稱在伺服器上部署的，用瀏覽器前往 *http://www.example.com/* 就會在伺服器觸發 `index()` 的執行。這個 view 函式的回傳值就是用戶端收到的**回應**（*response*）。如果用戶端是網頁伺服器，這個回應就是在瀏覽器視窗顯示給使用者看的文件。view 函式的回應可能是含有 HTML 內容的簡單字串，也可能是比較複雜的表單，稍後會展示較複雜的案例。

在 Python 原始碼檔案裡面放入以 HTML 碼寫成的回應字串會讓程式難以維護。本章的範例採取這種做法只是為了介紹回應的概念。第 3 章會介紹比較好的 HTML 回應產生方式。

如果你仔細觀察在日常生活中使用的服務的 URL，可以發現很多 URL 都有可變的部分。例如，Facebook 個人資訊網頁的 URL 格式是 *https://www.facebook.com/<your-name>*，裡面有你的使用者名稱，所以每位使用者的這個 URL 都不同。Flask 在 `app.route` 裝飾器裡面使用一種特殊的語法來支援這種 URL。下面的範例定義了一個有動態成分的路由：

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

角括號裡面的路由 URL 是動態的部分。所有符合固定部分的 URL 都會被對應到這個路由，當 view 函式被呼叫時，它就會從引數收到動態的部分。上面的範例用 `name` 引數來產生一個含有個人化歡迎訊息的回應。

路由的動態成分預設使用字串，但你也可以使用不同的型態。例如，路由 `/user/<int:id>` 只能匹配 `id` 動態部分是一個整數的 URL，例如 */user/123*。Flask 支援的路由型態有 `string`、`int`、`float` 與 `path`。`path` 是一種特殊的字串型態，與 `string` 不同的地方在於它可以使用正斜線。

完整的 app

上一節說明了 Flask web app 的各個部分，接著你要編寫自己的第一個 app 了。範例 2-1 的 *hello.py* app 腳本定義了一個 app 實例、一個路由以及 view 函式。

範例 2-1　hello.py：完整的 Flask app

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

如果你有複製 GitHub 上的 app Git 存放區，可以執行 `git checkout 2a` 來 check out 這個版本。

開發 web 伺服器

Flask app 有個開發 web 伺服器，你可以用 `flask run` 命令啟動它。這個命令會尋找含有 `FLASK_APP` 環境變數指定的 app 實例的 Python 腳本名稱。

若要啟動上一節的 *hello.py* app，你要先啟動已經建立好的虛擬環境，並且在裡面安裝了 Flask。對 Linux 與 macOS 使用者而言，啟動 web 伺服器的方法是：

```
(venv) $ export FLASK_APP=hello.py
(venv) $ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

對 Microsoft Windows 使用者而言，唯一的差異是設定 `FLASK_APP` 環境變數的方式：

```
(venv) $ set FLASK_APP=hello.py
(venv) $ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

伺服器會在啟動後執行迴圈來接收 request 並且服務它們。這個迴圈會持續執行，直到你按下 Ctrl+C 來停止 app 為止。

請在伺服器開始運行之後，打開網頁瀏覽器並且在網址列輸入 **`http://localhost:5000/`**。圖 2-1 是連接 app 之後的畫面。

圖 2-1　hello.py Flask app

如果你在基礎 URL 後面輸入任何其他文字，app 將無法知道如何處理它，並回傳錯誤碼 404 給瀏覽器—當我們瀏覽不存在的網頁時經常看到這種錯誤。

Flask 提供的 web 伺服器僅供開發與測試使用，第 17 章會介紹產品 web 伺服器。

你也可以用程式呼叫 `app.run()` 方法來啟動 Flask 開發 web 伺服器。當你使用沒有 `flask` 命令的舊版 Flask 時，必須執行 app 的主腳本才能啟動伺服器，且該主腳本的結尾必須有這段程式：

```
if __name__ == '__main__':
    app.run()
```

雖然當我們可以執行 `flask run` 命令時不需要採取這種做法，但 `app.run()` 方法在某些情況下依然可派上用場，例如在單元測試時，你會在第 15 章看到這種情況。

動態路由

範例 2-2 是 app 的第二個版本，它加入第二個路由，這個路由是動態的。當你在瀏覽器前往動態 URL 時，會看到一個個人化的歡迎訊息，裡面有以 URL 提供的名字。

範例 *2-2　hello.py*：使用動態路由的 *Flask app*

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

如果你有複製 GitHub 上的 app Git 存放區，可以執行 `git checkout 2b` 來 check out 這一版的 app。

在測試動態路由時，請確定伺服器正在運行，接著前往 *http://localhost:5000/user/Dave*。app 會用 `name` 動態引數來回應個人化的歡迎訊息。試著在 URL 使用不同的名字，看看 view 函式是否一定會用指定的名字來回應。圖 2-2 是其中一個案例。

圖 2-2　動態路由

除錯模式

你可以選擇用**除錯模式**來執行 Flask app。這個模式預設啟用兩種非常方便的開發伺服器模組，稱為 *reloader* 與 *debugger*。

啟用 reloader 時，Flask 會查看專案的所有原始碼檔案，並且在任何檔案被修改時自動重新啟動伺服器。在開發的過程中啟用 reloader 來運行伺服器很有幫助，因為每當你修改與儲存原始檔案時，伺服器就會自動重新啟動，並且採用你做的修改。

debugger 是一種 web 工具，當你的 app 出現未處理的異常狀況時，debugger 會在瀏覽器上出現，此時網頁瀏覽器視窗會變成互動式堆疊追蹤（stack trace），讓你可以檢視原始碼，並且在堆疊追蹤的任何地方執行運算式。圖 2-3 是 debugger 的外觀。

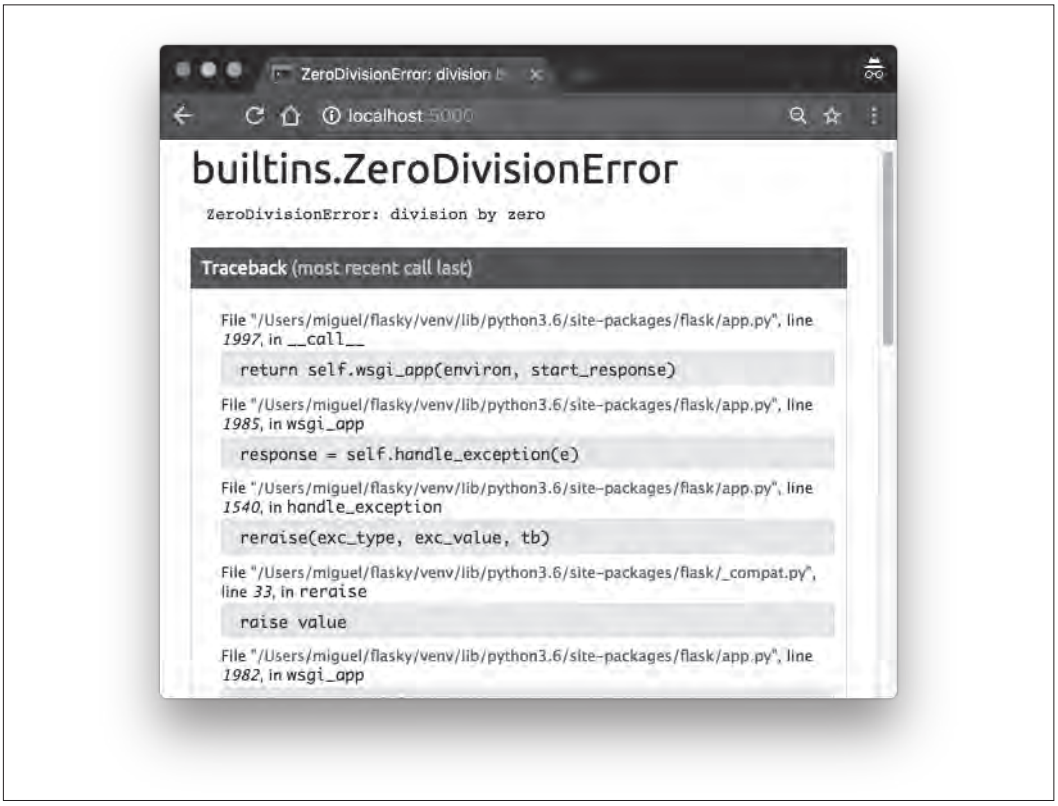

圖 2-3　Flask debugger

debug 模式在預設情況下是停用的。若要啟用它，你要先設定 FLASK_DEBUG=1 環境變數再呼叫 flask run：

```
(venv) $ export FLASK_APP=hello.py
(venv) $ export FLASK_DEBUG=1
(venv) $ flask run
 * Serving Flask app "hello"
 * Forcing debug mode on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN:273-181-528
```

當你使用 Microsoft Windows 時，要將 export 換成 set 來設定環境變數。

如果你用 app.run() 方法來啟動伺服器的話，就不需要使用 FLASK_APP 與 FLASK_DEBUG 環境變數了。若要以程式啟用 debug 模式，可使用 app.run(debug=True)。

絕對不要在產品伺服器上啟用 debug 模式。用戶端可藉由 debugger 請求執行遠端程式碼，讓產品伺服器易受攻擊。比較簡單的保護方法是用 PIN 來啟動 debugger，你可以用 flask run 命令在主控台的輸出畫面中看到 PIN。

命令列選項

flask 命令有一些選項可用。要查看有哪些選項，你可以執行 flask --help 或是單純執行 flask，不使用任何引數：

```
(venv) $ flask --help
Usage: flask [OPTIONS] COMMAND [ARGS]...

  This shell command acts as general utility script for Flask applications.

  It loads the application configured (through the FLASK_APP environment
  variable) and then provides commands either provided by the application or
  Flask itself.

  The most useful commands are the "run" and "shell" command.
```

```
  Example usage:

    $ export FLASK_APP=hello.py
    $ export FLASK_DEBUG=1
    $ flask run

Options:
  --version  Show the flask version
  --help     Show this message and exit.

Commands:
  run     Runs a development server.
  shell   Runs a shell in the app context.
```

`flask shell` 命令的用途是在 app 的環境（context）中啟動 Python 殼層 session。你可以使用這個 session 來執行維護工作、進行測試或除錯。稍後幾章會介紹這個命令實際的使用範例。

你已經瞭解 `flask run` 命令了，顧名思義，它會用開發 web 伺服器來執行 app。這個命令有許多選項：

```
(venv) $ flask run --help
Usage: flask run [OPTIONS]

  Runs a local development server for the Flask application.

  This local server is recommended for development purposes only but it can
  also be used for simple intranet deployments.  By default it will not
  support any sort of concurrency at all to simplify debugging.  This can be
  changed with the --with-threads option which will enable basic
  multithreading.

  The reloader and debugger are by default enabled if the debug flag of
  Flask is enabled and disabled otherwise.

Options:
  -h, --host TEXT                 The interface to bind to.
  -p, --port INTEGER              The port to bind to.
  --reload / --no-reload          Enable or disable the reloader.  By default
                                  the reloader is active if debug is enabled.
  --debugger / --no-debugger      Enable or disable the debugger.  By default
                                  the debugger is active if debug is enabled.
  --eager-loading / --lazy-loader
                                  Enable or disable eager loading.  By default
                                  eager loading is enabled if the reloader is
```

```
                                disabled.
  --with-threads / --without-threads
                                Enable or disable multithreading.
  --help                        Show this message and exit.
```

--host 引數特別好用，因為它可讓 web 伺服器知道要監聽哪個網路介面來與用戶端連結。在預設情況下，Flask 開發 web 伺服器會在 *localhost* 監聽連結，所以只接受來自伺服器所在電腦的連結。下面的命令可讓 web 伺服器監聽公共網路介面的連結，讓同一個網路的其他電腦也可以與之連結：

```
(venv) $ flask run --host 0.0.0.0
 * Serving Flask app "hello"
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

現在，在 *http://a.b.c.d:5000* 網路的其他電腦也可以連接 web 伺服器了，*a.b.c.d* 是在網路上的伺服器所屬的電腦的 IP 位址。

--reload、--no-reload、--debugger 與 --no-debugger 選項可在 debug 模式提供更大程度的控制。例如，啟用 debug 模式時，你可以用 --no-debugger 來關閉 debugger，同時持續啟用 debug 模式與 reloader。

請求回應循環

寫好基本的 Flask app 之後，你可能想要更深入瞭解 Flask 如何施展魔法。接下來的各節要說明這個框架的設計。

application 與 request context

當 Flask 從用戶端收到 request 時，必須讓處理它的 view 函式能夠使用一些物件。其中一個很好的例子就是 *request 物件*，它封裝了用戶端送來的 HTTP request。

為了讓 view 函式讀取 request 物件，Flask 最直接的做法是用引數傳入 request 物件，但是這種做法會讓 app 的每一個 view 函式多出一個引數。如果 view 函式除了 request 物件之外還要處理其他物件才能滿足要求的話，情況就更複雜了。

為了避免 view 函式有許多不一定用得到的引數，Flask 使用 *context* 來讓某些物件暫時可被全域操作。拜 context 之賜，你可以這樣編寫 view 函式：

```
from flask import request

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is {}</p>'.format(user_agent)
```

注意，在這個 view 函式中，request 被當成全域變數來使用。事實上 request 不是全域變數，在多執行緒伺服器中，各個執行緒可以同時處理來自不同用戶端的不同 request，所以各個執行緒需要在 request 內看到不同的物件。Flask 可藉由 context 讓某一個執行緒可全域存取某些變數，且不會干擾其他的執行緒。

執行緒是可被獨立管理的最小指令序列。一個程序通常有多個活動的執行緒，它們有時會共用一些資源，例如記憶體或檔案控制代碼（file handle）。多執行緒 web 伺服器會啟動一個執行緒池，並且從池中選出一個執行緒來處理收到的各個 request。

Flask 有兩種 context：*application context* 與 *request context*。表 2-1 是各種 context 公開的變數。

表 2-1　Flask context 全域變數

變數名稱	Context	說明
current_app	application context	運行中的 app 的實例。
g	application context	當 app 處理 request 時，可將這個物件當成暫時存放區。這個變數會在各個 request 中重設。
request	request context	request 物件，封裝了用戶端送來的 HTTP request 內容。
session	request context	使用者 session，它是個字典，app 可用它來儲存需要在各個 request 之間“記住”的值。

Flask 會在發送 request 給 app 之前啟動（或**推動**（*push*））application 與 request context，並且在處理完 request 後移除它們。當 application context 啟動時，執行緒可使用 current_app 與 g 變數。類似的情況，當 request context 啟動時，執行緒可使用 request 與 session。如果你在 application 或 request context 沒有啟動的情況下存取這些變數，就會產生錯誤。本章與之後的章節會更深入討論這四種 context 變數，先不用擔心你還不瞭解它們的用法。

下面的 Python 殼層 session 展示 application context 的運作方式：

```
>>> from hello import app
>>> from flask import current_app
>>> current_app.name
Traceback (most recent call last):
...
RuntimeError: working outside of application context
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'hello'
>>> app_ctx.pop()
```

在這個範例中，current_app.name 會在 application context 未啟動的情況下失敗，但是會在 app 的 application context 啟動時正常運作。特別注意我們是如何對 app 實例呼叫 app.app_context() 來取得 application context 的。

指派 request

當 app 收到來自用戶端的 request 時，必須知道應該呼叫哪個 view 函式來服務它。所以，Flask 會在 app 的 *URL map* 中尋找 request 裡面的 URL，這個 map 存有 URL 與負責處理它的 view 函式之間的對應關係。Flask 會用以 app.route 裝飾器或無裝飾器的版本 app.add_url_rule() 提供的資料來建立這個 map。

你可以在 Python shell 查看 *hello.py* 的 map 來瞭解 Flask app 的 URL map 長怎樣。在做這件事之前，你要先啟動虛擬環境：

```
(venv) $ python
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
 <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

在 app 中，/ 與 /user/<*name*> 路由是用 app.route 裝飾器定義的。/static/<*filename*> 路由是 Flask 加入的特殊路由，用途是存取靜態檔案。第 3 章會詳細介紹靜態檔案。

URL map 裡面的 (HEAD, OPTIONS, GET) 元素是路由要處理的 *request 方法*。根據 HTTP 規格的定義，所有的 request 都是用方法來發出的，它通常會指明用戶端要求伺服器執行的動作。Flask 會將方法指派給各個路由，所以可以用不同的 view 函式來處理送到同一個 URL 的不同 request 方法。Flask 會自動管理 HEAD 與 OPTIONS 方法，所以實際上，

這個 app 的 URL map 裡面的三個路由都被指派給 GET 方法，當用戶端想要請求網頁這類的資訊時就會使用它。第 4 章會教你如何為其他 request 方法建立路由。

request 物件

之前提過，Flask 會用一個名為 request 的 context 變數來公開 request 物件，它是相當實用的物件，裡面有用戶端放在 HTTP request 裡面的所有資訊。表 2-2 是 Flask request 物件最常用的屬性與方法。

表 2-2　Flask request 物件

屬性或方法	說明
form	字典，裡面有以 request 送出的所有表單欄位。
args	字典，裡面有傳入“URL 的查詢字串”的所有引數。
values	字典，結合 form 與 args 的值。
cookies	字典，存有 request 之中的所有 cookie。
headers	字典，裡面有 request 內的所有 HTTP 標頭。
files	字典，裡面有放在 request 裡面的所有上傳檔案。
get_data()	回傳 request 內文的緩存資料。
get_json()	回傳一個 Python 字典，裡面有 request 內文中已解析的 JSON。
blueprint	處理 request 的 Flask 藍圖（blueprint）名稱。第 7 章會介紹藍圖。
endpoint	處理 request 的 Flask 端點名稱。Flask 會將 view 函式的名稱當成路由的端點名稱。
method	HTTP request 方法，例如 GET 或 POST。
scheme	URL 協定（http 或 https）
is_secure()	如果 request 是用安全連結（HTTPS）傳來的，則回傳 True。
host	在 request 內定義的主機，如果它是用戶端提供的話，也包含連接埠號碼。
path	URL 的路徑部分。
query_string	URL 的查詢字串部分，用原始二進位值表示。
full_path	URL 的路徑與查詢字串部分。
url	用戶端請求的完整 URL。
base_url	與 url 一樣，但沒有查詢字串部分。
remote_addr	用戶端的 IP 位址。
environ	request 的原始 WSGI 環境字典。