前言

本書對象

本書的對象是想要使用 JavaScript、Node 與 Express 製作 web app 的程式員（傳統網站，或使用 React、Angular 或 Vue 製作單頁 app，或 REST API，或介於兩者之間的任何東西）。Node 開發有個令人興奮的層面：它吸引了一群全新的程式員。JavaScript 的親切性與靈活性吸引了世界各地自學的程式員。在電腦科學的歷史中，程式設計從來沒有如此便利，教導程式設計的（以及在你卡住時可以尋求協助的）線上資源的數量與品質不僅令人驚訝，也鼓舞人心。所以，如果你是新的（或許是自學的）程式員，歡迎加入我們。

當然，有些人跟我一樣，已經寫了一陣子程式了。如同我這個領域的許多程式員，我最初使用組合語言與 BASIC，後來用過 Pascal、C++、Perl、Java、PHP、Ruby、C、C# 與 JavaScript。我在大學接觸許多小眾語言，例如 ML、LISP 與 PROLOG。雖然我熟悉許多其中的語言，但是我認為這些語言都不如 JavaScript 那麼有前途。所以這本書也是為了我這種程式員寫的，他們有很多經驗，可能對於特定的技術有更深入的哲學觀點。

看這本書之前不需要任何 Node 經驗，但需要有 JavaScript 的經驗。如果你是程式新手，我推薦 Codecademy（*http://bit.ly/2KfDqkQ*）。如果你是中階或資深的程式員，容我推薦我自己的書，*Learning JavaScript, 3rd Edition*（O'Reilly）。本書的範例可以在任何可以運行 Node 的系統上執行（包含 Windows、macOS、Linux 及其他）。這些範例是為命令列（終端機）用戶設計的，所以你要稍微瞭解系統的終端機。

最重要的是，本書是寫給渴望學習的程式員看的。他們對將來的網際網路感到興奮，而且想要加入它。他們熱切希望學習新東西、新技術，以及 web 開發的新觀點。親愛的讀者，如果你還沒有興奮的感覺，希望你看完這本書時有這種感覺…

第二版說明

撰寫本書的第一版是一件開心的事情，直到今天，我仍然很高興當時能在書中提出實用的建議，以及收到讀者熱烈回應。第一版是在 Express 4.0 發表的時候出版的，雖然目前 Express 仍然是 4.x 版，但是圍繞著 Express 的中介函式（middleware）和工具已經有了**巨大的**變化。此外，JavaScript 本身也有所演變，甚至 web app 的設計方式也經歷了結構性的轉變（從純伺服器端算繪（rendering）變成單頁 app [SPA]）。雖然第一版介紹的許多原則仍然是實用且有效的，但具體的技術和工具已經幾乎完全不一樣了。那個版本已經過期了。由於 SPA 的優勢，第二版的重點變成「將 Express 當成 API 與靜態資產的伺服器」，並且加入一個 SPA 範例。

本書的結構

第 1 章與第 2 章介紹 Node 與 Express，以及後續內容即將使用的工具。第 3 章與第 4 章開始使用 Express 建構一個範例網站框架，它將成為本書其餘部分的主軸範例。

第 5 章討論測試與 QA，第 6 章介紹 Node 的重要結構，以及 Express 如何擴展和使用它們。第 7 章介紹製模（使用 Handlebars），建立基礎來使用 Express 建構實用的網站。第 8 章與第 9 章介紹 cookie、session 以及表單處理式（handler），以上就是建構基本的 Express 網站必須知道的事項。

第 10 章研究中介函式，它是 Express 的核心概念。第 11 章解釋如何使用中介函式從伺服器寄出 email，以及探討 email 固有的安全和版面配置問題。

第 12 章介紹生產問題。雖然在這個階段，你尚未完全掌握如何建構準生產網站，但是提前考慮生產環境可以免除後患。

第 13 章討論持久保存，把重點放在 MongoDB（文件資料庫龍頭之一）以及 PostgreSQL（流行的開放原始碼關聯資料庫管理系統）。

第 14 章詳述如何用 Express 來安排路由（如何將 URL 對映至內容），第 15 章則討論如何使用 Express 編寫 API。第 17 章探討提供靜態內容的細節，重點是將性能最大化。

第 18 章討論安全防護：如何在 app 中建立身分驗證與授權機制（重點是使用第三方身分驗證供應者），以及如何用 HTTPS 來運行網站。

第 19 章解釋如何整合第三方服務。我們將以 Twitter、Google Maps、US National Weather Service 為例。

第 16 章運用學過的 Express 知識來將主軸範例重構為 SPA，將 Express 當成後端伺服器，來提供第 15 章製作的 API。

第 20 章與第 21 章為你的大日子做好準備：推出你的網站。這兩章也會介紹除錯，讓你可以在推出之前根除任何缺陷，順利上線。第 22 章介紹下一個重要（也不受重視）的階段：維護。

本書以第 23 章結束，如果你想要更深入學習 Node 與 Express，這一章介紹額外的資源，以及哪裡可以提供幫助。

範例網站

從第 3 章開始，本書使用一個主軸範例：Meadowlark Travel 網站。本書的第一版是我到 Lisbon 旅遊之後撰寫的，那趟旅行讓我充滿回憶，所以我將家鄉 Oregon 州的虛構旅遊公司當成範例網站（Western Meadowlark 是 Oregon 的州鳥）。Meadowlark Travel 可讓旅客聯絡當地的「業餘導遊」，也和自行車及摩托車租賃服務公司合作，網站的服務重點是生態旅遊。

如同任何教學範例，Meadowlark Travel 網站有點刻意打造，但是這個例子涵蓋了真實世界的許多挑戰：第三方元件整合、地理定位、電子商務、性能與安全防護。

由於本書的重點是後端基礎設施，所以範例網站並不完整，它只是個真實網站的虛構案例，目的是讓許多小範例更有深度，為它們提供背景。如果你正在開發自己的網站，你可以把 Meadowlark Travel 範例當成模板。

第一章

Express 簡介

JavaScript 革命

在介紹本書的主題之前，我一定要介紹一些歷史背景，談談 JavaScript 與 Node。JavaScript 的時代已然來臨。它最初只是卑微的用戶端腳本語言，現在不僅遍布世界各地的用戶端，拜 Node 之賜，它也異軍突起，開始成為伺服器端語言。

我們已經可以清楚地看到「完整的 JavaScript 技術堆疊」這個願景，再也不需要切換不同的背景了！你再也不需要將心智齒輪從 JavaScript 換檔到 PHP、C#、Ruby 或 Python（或任何其他伺服器端語言）了。它也促使前端工程師跳到伺服器端編寫程式。我的意思不是伺服器端程式設計只和語言有關，除了語言之外還有很多東西要學，但是透過 JavaScript，語言至少不是一種障礙。

本書是為看到 JavaScript 技術堆疊的願景的人而寫的。或許你是前端工程師，希望將經驗延伸到後端開發工作。或許你跟我一樣是資深的後端開發者，希望 JavaScript 可以取代根深蒂固的伺服器端語言。

如果你和我一樣是軟體工程師，你一定看過許多語言、框架和 API 的興起。其中有些已經展翅高飛，有些成為昨日黃花。或許你認為自己可以快速學習新語言、新系統，每當你遇到一種新語言，你都可以體驗這種感受：你可以在這裡發現大學學過的東西，在那裡發現幾年前在職場學過的東西，雖然這種洞察力確實令人自豪，但也會令人感到厭倦，因為有時你只**想要完成工作**，不想要學習新技術，或重拾好幾年或好幾個月沒有用過的技能。

很多人原本覺得 JavaScript 不太可能成為王者，相信我，我也有這種感受。如果你在 2007 年告訴我：「你不但會把 JavaScript 當成首選語言，也會幫它寫一本書」，我會說你瘋了。我對 JavaScript 也抱持常見的偏見：我認為它是一種「玩具」語言，一種讓業餘或半吊子的人隨便玩玩的東西。平心而論，JavaScript 確實降低業餘玩家的門檻，外界也有很多問題重重的 JavaScript 程式，這些現象確實損害這種語言的形象。我們應該把一句流行語倒過來講「怨恨參與者，而不是怨恨體系」。

不幸的是，這種對於 JavaScript 的偏見使人無法看到它的強大、靈活和優雅。儘管我們知道 JavaScript 早在 1996 年就出現了（雖然很多迷人的功能是在 2005 年才加入的），但很多人直到現在才認真地看待 JavaScript。

看了這本書之後，你應該就不會有這種偏見了，或許是你已經和我一樣擺脫偏見，或許你從一開始就沒有偏見。無論如何，你都很幸運，我準備介紹的 Express 是用這種令人愉快且驚奇的語言實現的技術。

在 2009 年，當大家開始發現 JavaScript 這種腳本語言的強大與表達能力之後，Ryan Dahl 看到把 JavaScript 當成伺服器端語言的潛力，促使 Node.js 的誕生。這是網際網路技術的黃金階段。Ruby（Ruby on Rails）從計算機科學的學術界汲取一些偉大的思想，結合他自己的一些新點子，告訴全世界更快速地建構網站與 web app 的方式。Microsoft 在網際網路時代做了勇敢的嘗試，用 .NET 做了很多了不起的事情，他們不僅借鑑 Ruby 與 JavaScript，也從 Java 的錯誤中學習，同時大量引用學術界的觀點。

如今，web 開發人員可以自由地使用最新的 JavaScript 功能，不必擔心用戶使用的是舊的瀏覽器，這要歸功於 Babel 之類的轉譯技術。負責在 web app 中管理依賴項目以及確保性能的 Webpack 已經變成一種很普遍的解決方案了，而 React、Angular 與 Vue 等框架正改變大家開發 web 方式，讓宣告式文件物件模型（DOM）處理程式庫（jQuery）逐漸邁入歷史。

現在是參與網際網路技術的時刻，到處都有驚奇的新點子（或是被重新點燃的驚奇舊點子）。現在創新的浪潮比以往任何時刻都要強烈。

Express 簡介

Express 網站說 Express 是一種「極簡且靈活的 Node.js web app 框架，為 web 和行動 app 提供穩健的功能組。」但是這句話到底是什麼意思？我們將它拆開討論：

極簡

這是 Express 最吸引人的層面之一。框架開發者經常忘記通常「少即是多」。Express 的理念是在你的大腦與伺服器之間提供**最少的**階層。這不代表它不夠強健或沒有足夠的功能，而是意味著它既不會阻礙你充分表達想法，也能提供一些有用的功能。Express 提供極簡框架，你可以視需求加入各種 Express 零件，將不需要的東西換掉。這是一種清新的感覺。很多框架都給你**所有東西**，在你編寫任何程式之前，就塞給你一個臃腫、神秘、複雜的專案。你的第一個動作通常是花時間砍掉不需要的功能，或換掉無法滿足需求的功能。Express 採取另一種做法，讓你在必要時才加入需要的東西。

靈活

Express 做的事情其實很簡單：它從用戶端（可能是瀏覽器、行動裝置、另一個伺服器、桌上型 app…任何一種說 HTTP 的東西）接收 HTTP 請求，再回傳一個 HTTP 回應。這個基本模式幾乎可以涵蓋所有連接網際網路的東西，所以 Express 的用法非常靈活。

web app 框架

比較精準的說法應該是「web app 框架的伺服器端」。現在當你聽到「web app 框架」時，腦海中浮現的通常是單頁 app 框架，例如 React、Angular 或 Vue。但是除了少數獨立的 app 之外，大部分的 web app 都必須和其他服務共享資料並且與之整合。它們通常透過 web API 來做這件事，web API 可視為 web app 框架的伺服器端元件。注意，有時你也會（有時最好這樣做）單純用伺服器端算繪來建立整個 app，此時 Express 也可以妥善地構成整個 web app 框架。

除了 Express 在它自己的簡介中提到的特性之外，我也想要加入兩個我的看法：

快速

隨著 Express 成為 Node.js 開發的首選 web 框架，許多運行高性能、高流量網站的大公司也開始關注它，這給 Express 團隊帶來壓力，讓他們開始專注於性能的提升，現在 Express 已經可以為高流量的網站提供領先群雄的性能了。

非強制性的

JavaScript 生態系統有一個特點在於它的規模與多樣化。雖然 Express 通常是 Node.js web 開發的核心工具，但 Express app 有上百種（甚至上千種）社群程式包可用。Express 團隊理解這種生態系統多樣性，所以提供了相當靈活的中介函式系統，讓你輕鬆地使用你喜歡的元件來建立 app。在 Express 開發過程中，你可以看到它捨棄「內建」的元件，轉而選擇可設置的中介函式。

我說過 Express 是 web app 框架的「伺服器端部分」…或許我們也要考慮伺服器端與用戶端 app 之間的關係。

伺服器端與用戶端 app

伺服器端 ***app*** 就是 app 裡面的網頁都會先在伺服器算繪（變成 HTML、CSS、圖像與其他多媒體資產、JavaScript）再送給用戶端的 app。另一方面，**用戶端** ***app*** 只收到一次最初的 app 包裝，並且用裡面的東西算繪大多數的用戶介面。也就是說，一旦瀏覽器收到初始的（通常是很精簡的）HTML 之後，它就會使用 JavaScript 來動態修改 DOM，不需要依靠伺服器顯示新網頁（不過原始資料通常仍然來自伺服器）。

在 1999 年之前，伺服器端 app 是標準做法，事實上，*web app* 這個名詞是那一年正式提出的。我認為大約在 1999 年至 2012 年之間是 Web 2.0 時代，許多後來的用戶端 app 的製作技術都是在那段期間開發出來的。在 2012 年，隨著智慧手機的普及，大家都盡量避免用網路傳輸的資訊，這種做法有利於用戶端 app 的出現。

伺服器端 app 通常稱為**伺服器端算繪**（SSR），用戶端 app 通常稱為**單頁** ***app***（SPA）。用戶端 app 完全可以用 React、Angular 與 Vue 等框架來實現。我一直覺得「單頁」有點用詞不當，因為從用戶的角度來看，它其實不只一頁，兩者唯一的區別在於網頁究竟是從伺服器送來的，還是在用戶端動態算繪的。

事實上，在伺服器端 app 與用戶端 app 之間有許多模糊的界線。許多用戶端 app 都有兩到三個可以送給用戶端的 HTML 包裹（例如公用介面與登入後的介面，或常規的介面與管理介面）。此外，SPA 經常與 SSR 結合，來提升第一頁的載入性能，和協助進行搜尋引擎優化（SEO）。

一般來說，如果伺服器只傳送少量的 HTML 檔案（通常一到三個），而且用戶可以看到動態 DOM 產生的豐富多畫面，它就可以視為用戶端算繪。各個畫面的資料（通常是 JSON 形式）與多媒體資產通常同樣來自網路。

當然，Express 不太關心你製作的究竟是伺服器端還是用戶端 app，它樂於扮演任何一種角色。你究竟只提供一個 HTML 包裹，還是提供一百個包裹，對 Express 而言都沒差。

雖然 SPA 無疑「贏得」web app 架構龍頭寶座，但本書的前幾個範例都是伺服器端 app，它們依然有其重要性，而且提供一個或是多個 HTML 包裹在概念上沒有太大的差異。第 16 章有一個 SPA 範例。

Express 歷史簡介

Express 的創作者 TJ Holowaychuk 說 Express 這種 web 框架的靈感來自 Sinatra。Sinatra 是一種建構在 Ruby 之上的 web 框架，所以 Express 借鑑以 Ruby 為基礎的框架是很自然的事情：Ruby 催生了大量優秀的 web 開發方法，它的目的是讓 web 開發工作更快速、更高效，且更易於維護。

雖然 Express 的靈感來自 Sinatra，它也和 Node 的「外掛」程式庫 Connect 有很密切的關係。Connect 以**中介函式**（*middleware*）這個名稱來代表可插的（pluggable）Node 模組，它們可以在不同程度上處理 web 請求。雖然 Express 在 2014 年的 4.0 版移除了它與 Connect 的關係，但是它的中介函式概念依然要歸功於 Connect。

Express 在 2.x 與 3.0 之間經歷了重大的改寫，在 3.x 與 4.0 之間也有一次。本書的重點是 4.0 版。

Node：一種新的 web 伺服器

在某種程度上，Node 與其他流行的 web 伺服器有很多共同點，包括 Microsoft 的 Internet Information Services（IIS）和 Apache。但是我們感興趣的是它究竟哪裡不同，見以下內容。

Node 和 Express 很像，也是以極簡的方式製作 web 伺服器。與需要好幾年才能精通的 IIS 或 Apache 不同的是，Node 很容易設定與配置。我不是說你很容易就可以在生產環境中微調 Node 伺服器來取得最大的性能，而是它的組態設置選項更簡單而且更直觀。

Node 與比較傳統的 web 伺服器的另一項主要差異是 Node 是單執行緒的。乍看之下，這是一種退步，事實上，這是神來之筆。單執行緒可以大幅簡化 web app 的編寫工作，如果你需要多執行緒 app 的性能，你只要啟動更多 Node 實例就可以得到多執行緒的性能優勢。敏銳的讀者可能認為這聽起來很像虛幻的把戲，藉著平行執行伺服器（而不是平行執行 app）來製造多執行緒的效果，難道不是只是把複雜的東西換個位置，而不是消除它？或許是吧，但是根據我的經驗，它只是將複雜的東西搬到它本來的地方。此外，隨著雲端運算日益流行，以及伺服器逐漸成為通用商品，這種做法比較合理。IIS 與 Apache 確實很強大，它們設計上就是為了榨乾當今強大硬體的所有性能，不過這是需要代價的，要達成這種性能需要具備可觀的設定和微調專業知識。

就編寫 app 的方式而言，Node app 和 PHP 或 Ruby app 相同的地方比 Node app 和 .NET 或 Java app 更多。雖然 Node 使用的 JavaScript 引擎（Google 的 V8）會將 JavaScript 編譯成原生機器碼（很像 C 或 C++），但它會透明地做這件事[1]，所以從用戶的角度來看，它的行為很像一種純直譯語言。免除單獨編譯步驟可以減少維護與部署方面的麻煩，因為你只要更新一個 JavaScript 檔案就可以讓變動自動生效了。

Node app 另一個引人注目的好處是 Node 不需要依賴特定的平台。雖然它不是第一種或唯一不需要依賴特定平台的伺服器技術，但技術與平台的關係比較類似包含各種程度的頻譜，而不是非有即無的關係。例如，雖然拜 Mono 之賜，你可以在 Linux 伺服器執行 .NET app，但是因為文件的欠缺以及系統的不相容，這種做法令人非常痛苦，同樣的，雖然你可以在 Windows 伺服器上執行 PHP app，但通常很難像在 Linux 電腦上那樣設定它。另一方面，Node 可以在所有主流作業系統（Windows、macOS 與 Linux）上輕鬆地設定，而且可以輕鬆地進行協作。網站設計團隊通常混合使用 PC 與 Mac。有些平台（例如 .NET）會給前端開發者與設計者帶來挑戰，因為他們通常使用 Mac，對協作與效率造成很大的影響。能夠在任何作業系統，在幾分鐘（甚至幾秒鐘！）之內啟動伺服器簡直是美夢成真。

1 通常稱為 *just in time*（JIT）編譯。

Node 生態系統

Node 當然是技術堆疊的核心。這種軟體可以讓 JavaScript 在伺服器上運行，切斷與瀏覽器的關係，讓你可以使用以 JavaScript 寫成的框架（例如 Express）。資料庫是另一項重要的元件，第 13 章會探討它。除非 web app 非常簡單，否則 web app 都需要資料庫，在 Node 生態系統中，有些資料庫比其他的更合用。

不意外的，所有主要的關聯資料庫（MySQL、MariaDB、PostgreSQL、Oracle、SQL Server）都有資料庫介面可用，忽略這些著名的巨頭並不聰明。但是，Node 開發的出現導致一種新的資料庫儲存法的興起：所謂的 NoSQL 資料庫。用**否定**的名稱定義一項東西不見得是好事，所以我們認為將 NoSQL 改稱為「文件資料庫」或「鍵 / 值資料庫」比較適當。它們提供一種概念上比較簡單的資料儲存方法。這種資料庫有很多種，但 MongoDB 是領導者之一，它也是本書將採用的 NoSQL 資料庫。

因為建立一個可以運作的網站需要使用很多種技術，所以我們用縮寫字來說明建構網站的「堆疊」。例如，Linux、Apache、MySQL 與 PHP 稱為 *LAMP* 堆疊。MongoDB 的工程師 Valeri Karpov 創造了 *MEAN* 縮寫：Mongo、Express、Angular 與 Node。雖然縮寫很吸睛，但它也有局限性：在生態系統中，很多資料庫與 app 框架都是「MEAN」這個縮寫無法涵蓋的（它也忽略我認為很重要的元件：算繪引擎）。

創造包含萬物的縮寫字是一種有趣的練習。Node 當然是我們不可或缺的元件，雖然坊間還有其他伺服器端 JavaScript 容器，但 Node 已經成為主流了。Express 也不是唯一的 web app 框架，儘管它的主導地位已經很接近 Node 了。在開發 web app 時，另外兩種通常不可或缺的元件是資料庫伺服器與算繪引擎（無論是 Handlebars 之類的製模（templating）引擎，還是 React 之類的 SPA 框架）。這兩種元件沒有明顯的領先者，這就是我認為侷限技術堆疊沒有幫助的原因。

因為 JavaScript 是整合這些技術的東西，所以我將它們稱為 *JavaScript* **堆疊**來涵蓋所有工具，在這本書，這個堆疊包含 Node、Express 以及 MongoDB（第 13 章也有一個關聯資料庫範例）。

授權

當你開發 Node app 時，可能會發現必須比以前更注意授權（我當然就是如此）。Node 生態系統的美妙之處在於你可以使用大量的程式包，但是每一個程式包都有它自己的授權規則，更糟糕的是，有的程式包需要使用其他的程式包，這意味著你可能難以瞭解 app 的各個部分的授權規定。

但是我也要告訴你一些好消息，Node 程式包最流行的授權是 MIT 授權，它是毫無限制的，**幾乎**可讓你做你想做的任何事情，包括在非開放原始碼的軟體中使用程式包。然而，你不能假設你使用的每一個程式包都採用 MIT 授權。

npm 有一些程式包會試著找出專案中的各個依賴項目的授權，你可以在 npm 搜尋 `nlf` 與 `license-report`。

雖然 MIT 是最常見的授權，但你可能也會看到這些授權條款：

GNU 通用公眾授權條款（GPL）

GPL 是一種流行的開放原始碼授權，它的設計巧妙地維持軟體的免費，也就是說，當你在專案中使用 GPL 授權的程式碼時，你的專案**也**必須是 GPL 授權的，當然，這意味著你的專案必須開放原始碼。

Apache 2.0

這個授權條款很像 MIT，可讓專案使用不同的授權，包括非開放原始碼授權。但是你必須說明哪些元件使用 Apache 2.0 授權。

Berkeley Software Distribution（BSD）

類似 Apache，這種授權可讓你在專案中使用任何你想用的授權，只要你說明哪些元件使用 BSD 授權即可。

有的軟體使用**雙授權**（遵守兩個不同的授權條款），通常是為了讓軟體可以在 GPL 專案中，以及在授權條款比較寬鬆的專案中使用（如果你要在 GPL 軟體中使用某個元件，那個元件必須是 GPL 授權的）。GPL 與 MIT 雙授權是我經常在個人的專案中採取的方案。

最後，如果你正在編寫自己的程式包，你應該當一位好公民，為你的程式包選擇一種授權條款，並且正確地撰寫它的文件。對開發人員來說，最痛苦的事情莫過於在使用別人的程式包時，被迫在原始碼裡面四處尋找授權條款，更糟的是根本找不到授權。

總結

希望這一章可以讓你瞭解什麼是 Express，以及它在更大型的 Node 和 JavaScript 生態系統中扮演什麼角色，並且幫助你釐清伺服器端與用戶端 web app 之間的關係。

如果你仍然不清楚 Express 究竟是什麼，別擔心：有時直接使用一樣東西比較容易瞭解它，這本書將幫助你使用 Express 來建構 web app。但是在使用 Express 之前，下一章要介紹 Node，它是瞭解 Express 如何工作的重要背景資訊。

第二章

Node 入門

如果你沒有任何 Node 經驗，本章是為你而寫的，你必須初步瞭解 Node 才能瞭解 Express 與它的實用性。如果你曾經使用 Node 建構 web app，你可以放心地跳過這一章。在這一章，我們將使用 Node 建構非常簡單的 web 伺服器，在下一章，我們將瞭解如何用 Express 來完成同一件事。

取得 Node

在系統中安裝 Node 再簡單不過了，Node 團隊竭盡所能地確保安裝程序在所有主要平台上都同樣簡單明瞭。

請到 Node 首頁（*http://nodejs.org*）。按下有個版本號碼、後面加上「LTS」字樣的綠色按鈕（推薦多數用戶使用）。LTS 代表 *Long-Term Support*（**長期支援**），它比 Current 版本更穩定（後者包含最新功能以及一些性能改善）。

按下按鈕之後，Windows 與 macOS 用戶會下載一個安裝程式，它可以帶領你完成整個程序。對使用 Linux 的讀者而言，使用程式包管理器（*http://bit.ly/36UYMxI*）或許可以更快地啟動與運行。

如果你使用 Linux，而且想要使用程式包管理器，你一定要按照上述網頁的指示執行。如果你沒有加入適當的程式包庫（package repository），許多 Linux 版本都會安裝很早之前的 Node 版本。

你也可以下載獨立的安裝程式（*https://nodejs.org/en/download*），當你想要將 Node 傳給你的機構時，可以使用這種方便的做法。

使用終端機

我一直深信終端機（也稱為**主控台**（*console*）或**命令提示**（*command prompt*））帶來的威力與生產力。本書的所有範例都假設你使用終端機。如果你不熟悉終端機，強烈建議你花一些時間熟悉你的終端機。本書的許多公用程式都有對應的 GUI 介面，所以如果你堅決不想使用終端機，你也有其他的選擇，但是你必須自行尋找做法。

如果你使用 macOS 或 Linux，你有大量德高望重的 shell（終端命令解譯器）可選擇。bash 是截至目前為止最流行的 shell，但 zsh 也有其擁護者。我比較喜歡 bash 的主因（除了一直都很熟悉它之外）是它的普遍性，當你坐在任何 Unix 類的電腦前面時，在 99% 的情況下，它的預設 shell 都是 bash。

如果你是 Windows 用戶，事情就沒那麼輕鬆了。Microsoft 對於提供愉快的終端機體驗沒有太大興趣，所以你必須多做一些事情。Git 很方便地加入一個「Git bash」shell，它提供了類 Unix 的終端機體驗（雖然只有一小組常見的 Unix 命令列公用程式，但它們很實用）。雖然 Git bash 提供一種精簡的 bash shell，但它也使用內建的 Windows 主控台應用程式，產生令人氣餒的體驗（就連最簡單的功能，例如調整主控台視窗的大小、選擇文字、剪下、貼上都很不直觀且彆扭）。所以建議你安裝比較精密的終端機，例如 ConsoleZ（*https://github.com/cbucher/console*） 或 ConEmu（*https://conemu.github.io*）。Windows 超級用戶（尤其是 .NET 開發者，或 Windows 系統或網路的核心管理員）有另一個選項：Microsoft 自己的 PowerShell。PowerShell 名符其實：很多人用它做了了不起的事情，熟練的 PowerShell 用戶和 Unix 命令列大師不分軒輊。但是如果你經常游走 macOS/Linux 與 Windows 之間，我建議你維持使用 Git bash 來利用它提供的一致性。

如果你使用 Windows 10 以上，現在你可以在 Windows 直接安裝 Ubuntu Linux 了！它不是雙開機（dual-boot）或虛擬系統，而是 Microsoft 的開放原始碼團隊將 Linux 體驗帶到 Windows 的創舉。你可以用 Microsoft App Store（*http://bit.ly/2KcSfEI*）在 Windows 安裝 Ubuntu。

Windows 用戶的最後一個選項是虛擬化。由於現代電腦有強大的能力與結構，虛擬機器（VM）的性能與實際的電腦幾乎沒有區別。我很幸運可以使用 Oracle 免費的 VirtualBox（*https://www.virtualbox.org/*）。

最後，無論你使用哪一種系統，你都可以使用優秀的雲端開發環境，例如 Cloud9（*https://aws.amazon.com/cloud9/*）（現在是 AWS 產品）。Cloud9 會啟動一個新的 Node 開發環境，可讓你輕鬆快速地使用 Node。

選擇喜歡的 shell 之後，建議你花一些時間瞭解它的基本知識。網路上有許多很棒的課程（The Bash Guide（*https://guide.bash.academy*）是很棒的入門網站），稍微學習它們可以為你省下日後的許多痛苦。你至少要知道如何瀏覽目錄、複製、移動、刪除檔案，以及跳出命令列程式（通常使用 Ctrl-C）。如果你想要成為終端機大師，建議你學習如何搜尋檔案內的文字、搜尋檔案與目錄、將命令串接起來（古老的「Unix 哲學」），以及將輸出轉至別處。

在許多類 Unix 系統上，Ctrl-S 的功能比較特殊：它會「凍結」終端機（以前是用來暫停快速捲動的輸出畫面），因為 Ctrl-S 通常是「儲存」的快速鍵，很多人會下意識地按下它，造成令人困惑的情況（很慚愧的是，我也經常如此）。要解開終端機凍結狀態，你只要按下 Ctrl-Q 即可。所以如果你不知道為什麼終端機看起來凍結了，試著按下 Ctrl-Q，看看能不能將它解開。

編輯器

程式員經常爭論究竟要使用哪一種編輯器，原因很簡單：編輯器是主要工具。我選擇的編輯器是 vi（或有 vi 模式的編輯器）[1]。雖然 vi 不符合大眾的口味（當我向同事說 vi 可以輕鬆地完成他們的工作時，他們總會翻我白眼），但找到強大的編輯器並且學習它們可以明顯提升生產力，我保證也可以提升工作樂趣。我特別喜歡 vi 的原因之一在於它和 bash 一樣無處不在（但這不是最重要的原因）。一旦你進入 Unix 系統，你就可以使用 vi。多數流行的編輯器都有「vi 模式」，可讓你使用 vi 鍵盤命令。習慣它之後，你就很難使用其他東西，vi 不好上手，但可帶來可觀的回報。

如果你跟我一樣，認同隨處可用的編輯器的價值，你的另一個選擇是 Emacs。Emacs 跟我沒緣分（通常你只會在 Emacs 和 vi 裡面選擇一個），但我百分之百欣賞 Emacs 提供的功能和彈性。如果 vi 的 modal 編輯法不適合你，建議你可以瞭解一下 Emacs。

1 目前 vi 基本上等於 vim（vi improved）。大多數的系統都將 vi 改稱為 vim，但我通常會輸入 `vim` 來確保我使用的是 vim。

雖然主控台編輯器（例如 vi 或 Emacs）非常方便，但有時你需要比較現代化的編輯器。Visual Studio Code 是一種流行的選項（*https://code.visualstudio.com/*）（不要把它看成名字沒有「Code」的 Visual Studio）。我很欣賞 Visual Studio Code，它有很好的設計、快速、高效，很適合用來進行 Node 與 JavaScript 開發。另一種流行的選項是 Atom（*https://atom.io*），它在 JavaScript 社群裡面也很流行，這兩種編輯器都可以在 Windows、macOS 與 Linux 上免費使用（而且它們都有 vi 模式！）。

有了優秀的程式編輯工具之後，我們將焦點轉向 npm，它可以協助我們取得別人寫的程式包，讓我們活用龐大且活躍的 JavaScript 社群。

npm

npm 是普及的 Node 程式包管理器（也是我們取得和安裝 Express 的手段）。在 PHP、GNU、WINE 和其他工具的傳統玩笑中，*npm* 不是首字母縮寫（所以不是大寫），而是「npm is not an acronym」的遞迴式縮寫。

一般來說，「安裝程式包」與「管理依賴項目」是程式包管理器的兩大功能。npm 是一種快速、能幹的、無痛的程式包管理器，我認為它對 Node 生態系統的快速成長和多樣性起了很大的作用。

它有一種流行的程式包管理器對手，稱為 Yarn，Yarn 使用與 npm 一樣的程式包資料庫，我們會在第 16 章使用 Yarn。

npm 會在你安裝 Node 的時候安裝，所以如果你按照上述的步驟操作，現在你就可以使用它了。讓我們開始工作吧！

當我們使用 npm 時，最主要的命令就是 `install`（一點都不奇怪），例如，要安裝 nodemon（一種流行的工具程式，可以在你修改原始碼時自動重啟 Node 程式），你要執行下面的命令（在主控台上）：

```
npm install -g nodemon
```

`-g` 旗標要求 npm **全域性地**安裝程式包，也就是讓整個系統都可以使用它。當我們討論 *package.json* 檔案時，你會更清楚地看到這個區別。就目前而言，經驗上來說，JavaScript 公用程式（例如 nodemon）通常需要全域安裝，而你的 web app 或專案專用的程式包則不需要如此。

與 Python 之類的語言不同的是，Node 平台還很新，所以你應該會使用最新版的 Node（Python 經歷了從 2.0 到 3.0 的重大變化，所以需要設法切換不同的環境）。但是如果你需要支援多種 Node 版本，你可以參考 nvm（*https://github.com/creationix/nvm*）或 n（*https://github.com/tj/n*），它們可讓你在不同的環境之間切換。你可以輸入 `node --version` 來檢查電腦安裝了哪個 Node 版本。

用 Node 建構簡單的 web 伺服器

如果你做過靜態的 HTML 網站，或是具備 PHP 或 ASP 背景，你應該知道 web 伺服器（例如 Apache 或 IIS）可以提供靜態檔案，讓瀏覽器透過網路顯示它們。例如，如果你建立了 *about.html* 檔案，並將它放到正確的目錄裡面，你就可以前往 *http://localhost/about.html*，你甚至可以省略 *.html*，取決於 web 伺服器的配置，但 URL 與檔名之間的關係很明顯：web 伺服器只需要知道檔案在電腦的哪裡，並且將它提供給瀏覽器。

localhost，顧名思義，代表你現在的電腦。*localhost* 經常用來代表 IPv4 回送位址 127.0.0.1 或 IPv6 回送位址 ::1。雖然很多人使用 127.0.0.1，但本書將使用 *localhost*。如果你使用遠端電腦（例如使用 SSH），切記，瀏覽 *localhost* 不會連接那台電腦。

Node 提供的模式與傳統的 web 伺服器不同：你寫的 app **就是** web 伺服器，Node 只提供建構 web 伺服器所需的框架。

你可能會說「但是我不想要編寫 web 伺服器！」這是自然的反應：你想要寫的是 app，不是 web 伺服器。但是 Node 可以讓你輕鬆地編寫 web 伺服器（甚至只要用幾行程式），而且你可以得到有價值的回報：讓你更能夠掌控你的 app。

安裝 Node 並且和終端機成為好朋友之後，我們上工吧！

Hello World

我一直覺得藉由顯示「Hello world」這個無趣的訊息來介紹一項工具是很爛的做法，但是在這個節骨眼公然挑戰這種無趣的傳統幾乎是一種冒犯的行為，所以我們還是乖乖地先採取這種做法，再做一些比較有趣的事情吧！

在你最喜歡的編輯器裡面，建立一個稱為 *helloworld.js* 的檔案（本書程式存放區的 *ch02/00-helloworld.js*）：

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('Hello world!')
})

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate....'))
```

你可能會覺得這個範例沒有使用分號怪怪的，取決於你何時或是在何處學習 JavaScript。我曾經是死忠的分號推廣者，但是我在進行許多 React 開發（傳統的做法是省略它們）時，很不情願地捨棄它們。過了一陣子，當矇蔽雙眼的迷霧消失之後，我開始覺得以前愛用分號很奇怪！現在我堅定地支持「去掉分號」，本書的範例將反映這一點。這是個人的選擇，如果你喜歡，你也可以使用分號。

在 *helloworld.js* 的目錄裡面，輸入 `node hello world.js`，接著打開瀏覽器，前往 *http://localhost:3000*，**出現了！**你的第一個 web 伺服器。這個伺服器不提供 HTML，它只在瀏覽器顯示「Hello world!」純文字訊息。喜歡的話，你也可以傳送 HTML，你只要將 `text/plain` 改成 `text/html`，並將 `'Hello world!'` 改成一個包含有效 HTML 的字串即可。在此不展示這種做法，因為我不想在 JavaScript 裡面編寫 HTML，第 7 章會詳述原因。

事件驅動設計

Node 背後的核心哲學就是**事件驅動設計**（*event-driven programming*）。對身為程式員的你而言，它代表你必須瞭解你可以使用的事件有哪些，以及如何回應它們。很多人都用「用戶介面」來說明事件驅動設計：當用戶按下某個東西時，你就要處理**按鍵事件**。這是很好的例子，因為程式員無法掌握用戶何時按下哪個東西，以及是否按下它，所以事件驅動設計非常直觀。或許你很難將這個比喻轉換成在伺服器上回應事件，但它們的原理是相同的。

在上面的範例程式裡面的事件是隱性的，它處理的事件是 HTTP 請求。`http.createServer` 方法用引數接收一個函式，每當有人發出 HTTP 請求時，那個函式就會被呼叫。我們的程式做的事情只是將內容類型設為純文字，並傳送字串「Hello world!」。

當你開始以事件驅動設計的方式來思考時，你就可以看到四處都有事件。其中一個事件就是用戶從一個網頁或 app 的某個區域前往另一個地方。你的 app 回應那個導覽的動作稱為**路由**（*routing*）。

路由

路由就是將用戶端要求的內容傳給它們的機制。對 web 用戶端 / 伺服器 app 而言，用戶端要在 URL 裡面指定想要的內容，具體來說，URL 包含路徑與查詢字串（querystring）（第 6 章會更詳細討論 URL 的各個部分）。

伺服器路由通常會使用路徑與查詢字串，但有時也會使用其他資訊，例如標頭、網域、IP 位址等，用它們來掌握（舉例而言）用戶大概在哪個地方，以及他的首選語言。

我們來擴展「Hello world!」範例，做一些比較有趣的事情。我們要提供一個精簡的網站，它有一個首頁，一個 About 網頁，以及一個 Not Found 網頁。我們先沿用之前的範例，只提供純文字，不提供 HTML（本書程式存放區的 *ch02/01-helloworld.js*）：

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req,res) => {
  // 將 url 一般化，移除它的查詢字串、
  // 非必要的結尾斜線，並且將它改成小寫
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('Homepage')
      break
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('About')
      break
```

```
    default:
      res.writeHead(404, { 'Content-Type': 'text/plain' })
      res.end('Not Found')
      break
  } })

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate....'))
```

執行它之後，你可以瀏覽首頁（*http://localhost:3000*）與 About 頁（*http://localhost:3000/about*），它會忽略任何查詢字串（所以使用 *http://localhost:3000/?foo=bar* 時會看到首頁），輸入任何其他的 URL（*http://localhost:3000/foo*）都會出現 Not Found 網頁。

提供靜態資源

我們已經完成一些簡單的路由了，接著要提供真正的 HTML 與 logo 圖像。它們之所以稱為**靜態**資源是因為它們通常不會變動（例如，相較於股票報價網頁，這種網頁的股價在每次重新載入時都有可能改變）。

當你進行開發以及執行小型專案時很適合使用 Node 來提供靜態資源，但是在進行大型專案時，你可能要使用 NGINX 或 CDN 之類的代理伺服器來提供靜態資源。詳情見第 17 章。

如果你曾經使用 Apache 或 IIS，你可能習慣建立一個 HTML 檔案，導覽至它那裡，將它自動傳給瀏覽器。Node 不是這樣運作的，你必須打開檔案，讀取它，再將它的內容傳給瀏覽器。所以我們在專案中建立一個稱為 *public* 的目錄（下一章會告訴你為何不稱它為 *static*）。在那個目錄裡面，我們建立 *home.html*、*about.html*、*404.html*，一個稱為 *img* 的子目錄，以及一張稱為 *img/logo.png* 的圖像。既然你已經在翻這本書了，你應該知道如何撰寫 HTML 檔以及找到一張圖像，所以我讓你自行完成這些工作。在 HTML 檔裡面這樣引用 logo：`<img src="/img/logo.png" alt="logo">`。

接著修改 *helloworld.js*（本書程式存放區的 *ch02/02-helloworld.js*）：

```
const http = require('http')
const fs = require('fs')
const port = process.env.PORT || 3000

function serveStaticFile(res, path, contentType, responseCode = 200) {
```

```
  fs.readFile(__dirname + path, (err, data) => {
    if(err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' })
      return res.end('500 - Internal Error')
    }
    res.writeHead(responseCode, { 'Content-Type': contentType })
    res.end(data)
  })
}

const server = http.createServer((req,res) => {
  // 移除 url 的查詢字串、非必要的結尾斜線，
  // 並且把它改成小寫來將它一般化
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      serveStaticFile(res, '/public/home.html', 'text/html')
      break
    case '/about':
      serveStaticFile(res, '/public/about.html', 'text/html')
      break
    case '/img/logo.png':
      serveStaticFile(res, '/public/img/logo.png', 'image/png')
      break
    default:
      serveStaticFile(res, '/public/404.html', 'text/html', 404)
      break
  }
})

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate....'))
```

當你前往 *http://localhost:3000/about* 時，你會收到 *public/about.html*。我們在製作這個範例的路由時沒有發揮太多想像力，你可以將路由改成任何你喜歡的東西，也可以將檔案改成任何你喜歡的東西。例如，要讓一週七天使用不同的 About 網頁，你可以使用檔案 *public/about_mon.html*、*public/about_tue.html* 等，並且在路由中加入邏輯，在用戶前往 *http://localhost:3000/about* 時提供適當的網頁。

注意，我們建立了一個輔助函式 `serveStaticFile` 來做這些工作。`fs.readFile` 是非同步的檔案讀取方法，這個函式有個同步版本，`fs.readFileSync`，但越早開始以非同步的方式思考越好。`fs.readFile` 函式使用一種稱為**回呼**（*callback*）的模式，你要提供一種稱為**回呼函式**的函式，當工作完成時，那個回呼函式會被呼叫（「被叫回（called back）」，所以它用這個名稱）。在這個例子中，`fs.readFile` 會讀取指定檔案的內容，並且在讀取檔案之後執行回呼函式；如果那個檔案不存在，或是有讀檔權限問題，`err` 變數就會被設定，函式會回傳 HTTP 狀態碼 500，代表伺服器錯誤。如果讀檔成功，檔案會被送給用戶端，連同指定的回應碼與內容類型。第 6 章會詳細說明回應碼。

`__dirname` 會被解析成正在執行的腳本所在的目錄，所以如果腳本位於 */home/sites/app.js*，`__dirname` 會解析為 */home/sites*。請盡可能地使用這個方便的全域變數，若非如此，當你在不同的目錄執行 app 時，可能會出現難以找出根源的錯誤。

前往 Express

Node 到目前為止應該還沒有讓你留下深刻的印象，我們基本上只是重複做了 Apache 或 IIS 自動為你做的事情，但是現在你已經瞭解 Node 如何做事，以及你擁有多少控制權了。雖然我們還沒有做什麼特別的事情，但是你可以看到我們如何將它當成起點，準備做更複雜的事情。當我們沿著這條路繼續編寫越來越複雜的 Node app 時，最終也會產生某種類似 Express 的東西。

幸運的是，我們不需要這樣做：Express 已經出現了，它可以幫你省下製作基礎架構的時間。具備一些 Node 經驗之後，接下來我們要開始學習 Express 了。