推薦序

西元 1995 年，當我在 Netscape 建立 JavaScript 時，我完全沒想到它會變成在網際網路上被廣泛運用的一種程式語言。我知道我有一點點時間可將它變成「最低限度可運作的版本」狀態，因此我讓它具備可擴充性，並可自全域物件向下調整，甚至到基礎階層的資料描述物件協議（例如：`toString` 和 `valueOf`，以 Java 相同名稱的方法命名）。

儘管它持續演進並且廣受歡迎，JavaScript 總是受惠於一種漸進式的、謹慎的指導原則，讓最重要的事情優先處理。我認為這個觀念一定是來自於匆忙的設計和有意提供的擴充性。我還多加了兩個核心元件，函式和物件，這樣程式開發人員可以在各方面運用它們，功能上作為每種特殊工具的通用替代方案。這意味著，學生必須學習哪些工具在什麼工作下是最適合使用，以及如何正確地使用它們。

Netscape 對我來說像是一陣旋風，或許對自西元 1995 年初的每一個人都是。它在首次公開募股時，透過蹩腳的「Netscape ＋ Java 打倒 Windows」慣用語，誓言要與 Microsoft 競爭；在該年的 IPO 巡迴演說中，Marc Andreesen 不斷重述這句話。Java 在當時的程式語言領域是居於老大哥，類似於「蝙蝠俠」的地位，而 JavaScript 只是扮演「羅賓」角色的一個「直譯式語言」。

但是，在我撰寫初版時（代號為「Mocha」），JavaScript 被深植於 Netscape 瀏覽器，而非 Java；且在同個時間點，我建立了文件物件模型（Document Object Model）。在當時已無人可以超越 Netscape/Sun 所建立的障礙，或瀏覽器 /JVM 的程式基礎架構，將 Java 或其他技術嵌入成為一種外掛套件。

所以我的確有一種隱約的感覺，JavaScript 若不是會隨著時間成功並佔有一席之地，就是很快地消失無影蹤。我記得曾經告訴我的朋友 Jeff Weinstein，當他問我，在接下來的二十年之間會做哪些事情；我回答，「JavaScript 或破產」。儘管如此，由於我在「極短的時程」中所選擇的設計理念，和管理階層指定「讓它看起來像 Java」的方針，我感覺對 JavaScript 的使用者有一種強烈的使命感。

模組化的 JavaScript 系列書籍實現了我逐步加強和直覺操作的指導理念；從簡單可應用的程式範例，逐漸擴展至模組化的應用程式。這一系列的書籍出色地含括了最佳的測試案例，和部署 JavaScript 應用程式的決戰技巧。在 O'Reilly 的 JavaScript 領域暢銷叢書中，像是一顆閃耀的珍珠。

我非常欣喜能夠在此支持 Nicolás 的努力，因為他的書籍看起來的確就是新進入 JavaScript 領域所需要的。我初次遇見 Nicolás 是在巴黎的一頓晚宴上，並在那時對他有一些認識，後來在網路上也保持聯繫。他本身的實用主義理念，結合著為新進使用者的思考觀點和一些風趣幽默感，說服了我為他審核本書的草稿；而完成後的作品，是有趣且容易閱讀的。在此我鼓勵你深入地研究、發掘、擁抱 JavaScript 技術，並為更好的網際網路作出屬於你的貢獻，嘉惠世人。

—Brendan Eich
JavaScript 創始人
Brave Software 創立者與 CEO

前言

早在西元 1998 年前，當我還在學校使用 FrontPage 這個軟體設計所有網頁時，如果有人說我將靠這個生活，我可能會暗自竊笑。JavaScript 一路跟著我們成長，真的很難想像如果沒有 JavaScript 技術，網路將如何興盛。這本書將一步步帶給你整體的現代 JavaScript 觀念。

誰該讀這本書

這本書主要是寫給網頁開發人員、技術狂熱者和具備 JavaScript 知識的專業人士閱讀。這些開發人員和其他想要精進 JavaScript 知識的人，都可以從閱讀《現代 *JavaScript* 實務應用》這本書得到收穫。

為什麼是現代的 JavaScript

這本書的宗旨，為提供一個簡易的方式來學習 JavaScript 的最新發展功能：ES6 和後續更新。ES6 版本對這個語言來說，是一個巨大的變動。大約在此時，我在部落格上撰寫了一些文章，是有關於 ES6 所推出的各項不同特徵功能。在市面上只有少數的書探討 ES6，且與我想要在書中討論的重點有些不同。本書會嘗試說明詳細的特徵功能細節，但不鑽研規格文件、實作的細節、或一些極端案例；極端案例若真的發生，實際上也必須上網討論鑽研才行。

除了這些極端的案例之外，本書會將大部分的焦點關注於學習過程，將內容素材以漸進的方式安排呈現；再搭配上一些實務案例，《*現代 JavaScript 實務應用*》一書內容不僅有關於 ES6 的功能，還包括自西元 2015 年 6 月以來的變更—當 ES6 技術規格最終確立—包含非同步函式、物件解構賦值、動態匯入、`Promise#finally`、和非同步產生器。

最後，本書的目標是為閱讀其他的模組化 JavaScript 系列叢書建立基礎知識。在這本書學習完最新的語言特徵後，我們就可以準備好進一步討論模組化設計、測試、和部署，而不需要在範例程式中所使用的新特徵功能上分神討論。這樣漸進式和模組化的學習方法，會普遍運用於整個系列叢書的每一本、每一章、和每一小節中。

本書架構

第 1 章，我們從 JavaScript 的簡介開始，以及它的標準發展流程，和這些年來的進化發展；它現在所處的階段，和未來的走向。你也會初步認識 Babel 和 ESLint，這些現代化的工具可以幫助我們認識現代的 JavaScript。

第 2 章內容包含了 ES6 所有重要的變更，包含箭頭函式、解構賦值、`let` 和 `const`、字串樣板和一些語法上的潤飾。

第 3 章我們會討論一個新增 `class` 類別的語法，來宣告物件原型；還有新的基礎型別稱為 `Symbol` 符號，以及一些新的 `Object` 方法。

第 4 章則會探討 ES6 中所有新的流程控制方法。我們會詳細地討論 Promise、迭代器、產生器和非同步函式；並配合大量的範例協助解說，發掘所有不同技巧間可協同合作的特質。你將不只會學習到流程控制的技巧，還能夠推論背後的邏輯意義，這樣才能夠運用它們來簡化你的程式碼。

第 5 章描述 ES6 中新的內建集合，我們可以使用它來建立物件映射和資料項唯一的集合。這些集合類型均會有使用範例提供參考。

第 6 章涵蓋了新的 `Proxy` 代理器和 `Reflect` 反射的內建功能。我們將學習有關代理器操作的方法，以及需注意的事項。

第 7 章則討論 ES6 中其他尚未介紹到的內建功能，特別是 `Array`、`Math`、數值、字串、萬國碼、及正規表示式。

第 8 章專注於原生的 JavaScript 模組，並簡述它的歷史。接著充分地討論它們的語法意涵。

第 9 章，也就是最終章—稱為「實務操作的考量」—這部分在程式語言的書籍中較少被提及。除了在書中的內容均會傳遞個人的觀點看法之外，在最後一章的內容中我也將它們彙整在一起。在此章節中，可以找到各種選擇其背後的支持論點，如何（How）決定何時（When）應該使用何種（What）變數宣告、或字串實字的標示引號，對非同步程式碼流程的處理建議，以及採用類別或代理器是否合適，和一些分析見解。

對已經有點熟悉 ES6 的人，可以在本書中選擇適合的章節開始閱讀；我建議第 4 章可以仔細的閱讀理解，因為你會深刻理解流程控制的價值。第 7 章和第 8 章也是必讀，它們可以提供你 ES6 領域的細節知識，且這些細節很少被公開討論。最後一章將激發你深度地思考—無論你是否同意本章內容所表述的觀點—在現代化 JavaScript 新的秩序規則下，JavaScript 應用程式中可行和不可行的部分。

本書編排慣例

本書以下列各種字體來達到強調或區別的效果：

斜體字（*Italic*）

代表新名詞、網址 URL、電子郵件、檔案名稱、以及檔案屬性。中文以楷體表示。

定寬字（`Constant width`）

用於標示程式碼，或是在本文段落中標註程式片段，如變數或函式名稱、資料庫、資料型別、環境變數、陳述式、關鍵字等等。

這個圖示代表提示或建議。

這個圖示代表警告或需要特別注意的地方。

第一章

ECMAScript 和 JavaScript 的未來

JavaScript 從西元 1995 年開始，還只是一個為了取得戰略優勢的行銷策略；直到西元 2017 年才在應用程式執行端轉變為全世界廣泛運用的核心程式體驗。這個語言不僅僅在瀏覽器裡運行，同時也能夠在桌面、行動裝置、硬體設備、甚至 NASA 的太空衣設計中存在。

JavaScript 是如何辦到的，而它的下一步在哪？

1.1 JavaScript 標準的發展概述

回溯至西元 1995 年，Netscape 擘畫出一個超越 HTML 語法的網頁動態網頁技術。Brendan Eich 進入 Netscape，就是為了研發一種類似於 Scheme 的程式語言，但是能夠在瀏覽器上運作。當他加入之後，便瞭解到高層希望此程式語言類似 Java，而這想法也早已在進行中。

Brendan 在十天之內，以 Scheme 的第一階層功能和 Self 的原型為主要元素，建立出第一個 JavaScript 原型。最原始的 JavaScript 版本被稱為 Mocha，它並沒有陣列或是物件實字，且每個錯誤都以警告方式（alert）顯示。因為缺乏處理例外事件的能力，導致現在許多的運作會產生 `NaN` 或是 `undefined` 的結果。Brendan 致力於 DOM 階層 0 和初版的 JavaScript 的發展，並在此階段建立了基礎標準的階段。

西元 1995 年 9 月，修正版本的 JavaScript 以 LiveScript 之名開始推廣，並由 Netscape Navigator 2.0 的 beta 版支援。當 Navigator 2.0 的 beta 3 版本於西元 1995 年 12 月發行之後，它就被重新命名為 JavaScript（原為 Sun 的註冊商標，目前屬於 Oracle）。在這個版本發行不久後，為了在 Netscape Enterprise Server 中也能夠撰寫 JavaScript，Netscape 發表了一個伺服器端的 JavaScript 實作方法，並命名為 LiveWire[1]。JScript，是 Microsoft 的 JavaScript 反向工程的實作，在西元 1996 年由 IE3 支援。JScript 在 IIS 伺服器端的也是可以使用的。

這個語言在西元 1996 年以 ECMAScript（ES）之名開始進行標準化，並定義於 ECMA-2625 規格中，由 ECMA 的技術委員會（稱為 TC39）審核管理。Sun 並沒有將 JavaScript 註冊商標的所有權轉讓給 ECMA；而同時間 Microsoft 也提供 JScript 使用，其他會員公司並不想使用此名稱，所以 ECMAScript 的發展就暫時停滯。

由於在實作方式上的競爭爭議，Netscape 的 JavaScript 和 Microsoft 的 JScript 主導了大部分 TC39 標準的委員會議。即便如此，委員會也已擬出準則規範：需建立向下相容性，並將之視為至高優先法則，例如嚴格等於運算子（`===` 與 `!==`）的定義，並不會破壞既有使用寬鬆等於比較的程式。

ECMA-262 的初版於西元 1997 年 6 月發行。一年之後，西元 1998 年 6 月，這個規格依據 ISO/IEC 16262 的國際標準進行修正，歷經國家 ISO 機構審核之後，正式成為第二版。

西元 1999 年 12 月發表了第三版，將正規表示式標準化、`switch` 敘述、`do/While`、`try/catch`、`Object#hasOwnProperty`，以及一些其他改變。大部分的這些特徵功能都可運用在 Netscape 的 JavaScript 執行期間和 SpiderMonkey。

1 西元 1998 年的一本書籍（*https://mjavascript.com/out/livewire*）利用 LiveWire 解釋伺服器端使用 JavaScript 的複雜性。

在不久之後，TC39 很快就公佈了 ES4 的規格草稿。早期的 ES4 也影響了在西元 2000 年中期所推出的 JScript.NET[2]，至西元 2006 年的 Flash 所支援的 ActionScript[3]。

針對 JavaScript 如何發展的意見衝突，讓規格制定的發展工作停頓了下來。對於網頁標準制定來說，是個黑暗時期：Microsoft 獨佔了所有的網路資源，但對於標準的發展則興趣缺缺。

當 AOL 在西元 2003 年解雇了 50 位 Netscape 的員工[4]，此時 Mozilla 成立了；然而，在 Microsoft 掌握超過 95% 的網頁瀏覽市占率時，TC39 卻已解散。

Mozilla 的 Brendan 花了兩年的時間，讓 ECMA 在 TC39 上復活，透過 Firefox 持續成長的市占率迫使 Microsoft 改變心意。西元 2005 年中，TC39 開始再次重新舉行定期會議。討論 ES4 技術規格時，也有計畫企圖導入模型系統、分類、迭代器、產生器、解構賦值、形式註解、以及其他特徵組合的計畫。但因為這項計畫野心勃勃，也使得 ES4 不斷地延宕。

直到西元 2007 年，這個技術委員會一分為二：ES3.1 相較於 ES3 有更多持續性的改善；而 ES4 則是過度設計且定義不清。到了西元 2008 年 8 月[5]，當 ES3.1 已經確認為是標準發展的正確途徑，但後來被改名為 ES5。而雖然 ES4 被放棄了，但它的許多特徵最終也形成了 ES6（在當時被稱為 Harmony），還有一些仍在考慮中，而有些則被放棄、回絕或是被撤銷。ES3.1 的更新可以視為是 ES4 技術規格發展的基石。

在西元 2009 年 12 月，ES3 發行十週年的當時，ECMAScript 第五版也發行了。這個版本將實際上的延伸操作編寫入語言技術規格中，在瀏覽器實作普遍運用；它新增了 `get` 和 `set` 存取器、`Array` 原型的功能改善、反射等，同時也以原生方式支援 JSON 文件的解析以及嚴格模式。

---

2 你可以在 Microsoft 網站上看到當時的原始聲明（*https://mjavascript.com/out/jscript-net*）（西元 2000 年 7 月）。

3 Brendan Eich 在 JavaScript Jabber 播客節目（podcast）中，談到 JavaScript 的起源（*https://mjavascript.com/out/brendan-devchat*）。

4 你可以在西元 2003 年 7 月的 *The Mac Observer* 中閱讀到這篇新聞報導（*https://mjavascript.com/out/aolnetscape*）。

5 Brendan Eich 於西元 2008 年發送一封電子郵件，內容概述了當時的狀況（*https://mjavascript.com/out/harmony*），時間大約在 ES3 發行之後的 10 年。

幾年之後，於西元 2011 年 6 月，技術規格再度進行檢視，並編輯成國際標準 ISO/IEC 16262:2011 的第三版，並正規化於 ECMAScript 5.1 下。

TC39 又花了四年的時間，在西元 2015 年 6 月將 ECMAScript 6 完成正規化。第六版對此語言來說，是一次最大的更新，並進行規格出版；同時也實作了許多 ES4 的提案，這些提案部分是 Harmony 專案中被延緩的提案。透過本書的內容，我們將會深入地探討 ES6。

ES6 努力發展的同時，在西元 2012 年 *WHATWG*（一項促進網頁發展的標準）也開始從相容性（compatibility）和互用性（interoperability）的觀點，來記錄 ES5.1 和瀏覽器實作之間的差異。這項工作強制將 `String#substr` 標準化，這個方法在之前並未被具體說明；用統一的方法將字串包裹於 HTML 標籤之中，這在各瀏覽器間的方法並不一致；並記錄 `Object.prototype` 特性，例如 `__proto__` 和 `__defineGetter__`，以及其他改善 [6]。這些努力成果均聚集於一個獨立出來的 Web ECMAScript 技術規格中，最終在西元 2015 年完成。其中附錄 B 的內容是一些關於核心 ECMAScript 技術規格的有用資訊，指出在技術實作上則不需要完全依照它的建議。綜合這些更新，附錄 B 也成為了規範，並提供網頁瀏覽器遵循。

第六版在 JavaScript 的歷史上是一個重要的里程碑。除了大量的新特徵功能外，ES6 帶來了一個關鍵性的影響，它讓 ECMAScript 成為一個持續更新的標準。

1.2 ECMAScript 是一個持續更新的標準

在 ES3 之後，有超過 10 年的時間在語言規格方面沒有重大變化；而 ES6 則花了約 4 年時間才具體化。明顯的是，TC39 的流程必須要再改善進化。這個版本修改流程通常要以截止期限來驅動進行。在達成共識前的任何延遲，都會造成兩個修訂版本中間漫長的等待，這會導致特徵蔓延，並產生更多的延遲。小部分的修改會受到大量加入的技術規格而延遲，而大量加入的規格需面臨最後定版的壓力，以避免爾後延遲。

6 欲瞭解完整的變更，可參考 WHATWG 部落格（*https://mjavascript.com/out/javascript-standard*）。

當 ES6 上市，TC39 簡化[7]它的提案修訂流程，同時調整流程以符合現代的期望：必須更頻繁和持續的進行，且讓規格發展更為民主化。此時，TC39 從古老的以 Word 文件為基礎的流程，轉變為使用 Ecmarkup 標示語言（一種特殊的 HTML 用來格式化 ECMAScript 文件規格）；以及運用 GitHub 拉下需求，大幅增加提案[8]建立的數量，以及擴大非會員自外部參與修訂。新的流程具連續性且更加透明：以往你必須要先從網頁下載一個 Word 或 PDF 檔案，現在最新規格的草稿均可以自線上參考（*https://mjavascript.com/out/spec-draft*）。

Firefox、Chrome、Edge、Safari、以及 Node.js 都對 ES6 技術規格有超過 95% 的相容[9]，但是我們已經能夠在這每一個瀏覽器中使用 ES6 所推出的特徵功能，而不需要等到所有功能均 100% 完成後才轉換過去。

新的流程依據成熟度共包含了四個階段[10]。提案內容越發成熟可行，越有可能進入最終技術規格。

任何對於功能變更或是新增的討論、想法、或是提案，只要尚未提交為正式提案之前，都被定義為「稻草人（strawman）」（第零階段），但只有 TC39 的會員可以提出「稻草人」提案。截至撰稿為止，已經有超過 12 個討論中的稻草人提案[11]。

在第一階段時，會將提案正式化並希望著重於跨功能的要點、與其他提案的互動性、以及實作上的考量。在此階段的提案必須將問題定義釐清，並針對問題提出具體的解決方法。在第一階段的提案通常包含一個高階的 API 描述、舉例性的使用範例、和功能內部語法和演算法的討論。第一階段提案在經過審核流程後，是有可能會進行重大的調整變更。

7 查看「後 ES6 技術規格制定流程」的簡報（*https://mjavascript.com/out/tc39-improvement*），自西元 2013 年 9 月導入簡化提案修訂流程。

8 查看所有 TC39 考量的提案（*https://mjavascript.com/out/tc39-proposals*）。

9 查看此表格，詳細記錄 ES6 在各瀏覽器之中的相容性（*https://mjavascript.com/out/es6-compat*）。

10 查看 TC39 提案流程文件（*https://mjavascript.com/out/tc39-process*）。

11 你可以追蹤稻草人提案（*https://mjavascript.com/out/tc39-stage0*）。

在第二階段時，提案會有一個初步的規格草稿。在此同時開始進行實驗性質的實作也是可行的。實作可以向下相容程式的方式發展，並遵循提案內容；在引擎上的實作，需以原生的方式對提案內容提供支援；或使用建置期間的工具來轉換原始程式碼，編譯為某些既定項目，讓引擎可以執行。

在第三階段時，提案已成為候選推薦案。技術規格編輯與指定的技術審核人，必須對進入此階段的每個提案表示意見，同意後才會進入最終的技術規格；實作人員不需要表示對提案的興趣偏好。在實務上，進入這個階段的提案都至少已在一個瀏覽器平台上完成實作，產出一個高度向下相容的程式，或由建置期間的編譯器支援，如：Babel。第三階段的提案已不太會變更，除了修正一些極端的案例。

兩個獨立的實作都必須通過驗收測試（acceptance tests），以取得該提案在第四階段的狀態。進入第四階段的提案將會被納入下一版本的 ECMAScript 中。

期望能夠從現在開始，都能夠每年釋出新的技術規格。為了達成每年更新的時程，每個釋出的版本會用它的出版年份作為參考。因此，ES6 就變成了 ES2015，後續釋出的 ES7 為 ES2017，諸如此類。但實際上，ES2015 的名稱並未被使用，因此仍然以 ES6 表示；而 ES2016 釋出時也還未有這樣的命名參考，因此有時也還是稱為 ES7。當在社群中已經普遍使用 ES6 名稱來稱呼此版本時，我們可以總結出較常使用的名稱應為：ES6、ES2016、ES2017、ES2018，後續依此類推。

提案流程會結合每年度的甄選，以將甄選出來的提案轉換為可行的標準，並轉換為一致的出版流程，這也意味著文件版本號碼變得不是那麼重要，關注的重點就會在於提案階段；而我們可以預期 ECMAScript 標準的版本號碼會變得更不常使用。

1.3 瀏覽器支援度和輔助工具

在第三階段的推薦候選提案，非常有可能被甄選出來作為定版的技術規格，最後在兩個 JavaScript 引擎獨立的實作領域中提供。第三階段提案會被視為是可於真實世界的應用程式中安全使用，透過實驗性引擎的實作、向下相容程式、或編譯器的使用測試。第二階段和更早的提案則用

於 JavaScript 開發人員，透過實作人員和使用者的操作回饋，持續進行調整。

Babel 和類似功能的編譯器能夠輸入一段程式碼，產生網頁平台（HTML、CSS、或 JavaScript）可理解的輸出結果；通常稱這類的工具為**轉譯器**（*transpiler*），它可歸類於編譯器的子類別中。當我們想要在程式碼中使用一項提案，但它尚未廣泛地於 JavaScript 引擎中實作時，Babel 和類似功能的編譯器可以將這段程式碼以新提案的方式，轉換為目前 JavaScript 引擎可以支援的方式執行。

這樣的轉換可以在建置期間（build time）進行，如此使用者所取得的程式碼，就可以在 JavaScript 執行期間被完整支援。這個機制可以改善執行期間（runtime）的支援度，讓 JavaScript 開發人員能夠很快採用新的語言特徵和語法。對技術文件的撰寫者和實作人員也受益良多，因為他們就可以蒐集來自於使用者的回饋，有關於功能的可用性、期待性，甚至臭蟲或特殊案例的意見。

轉譯器能夠輸入我們所撰寫的 ES6 原始碼，並輸出 ES5 的程式碼，使得瀏覽器可以有一致性的理解。這是在目前階段運作 ES6 程式碼最可靠的方法：於建置期間產出 ES5 程式碼，可被舊版和新版瀏覽器運作執行。

相同機制也可以套用至 ES7 和後續版本。在每年新語言版本的釋出，我們也可以期待編譯器支援新的 ES2017 輸入、ES2018 輸入等等。相同地，當瀏覽器的支援度更好時，我們可以期待編譯器降低 ES6 輸出的複雜度，接著是 ES7 輸出的複雜度，依此類推。按照這樣的機制，我們可以將 JavaScript 對 JavaScript 的轉譯器視為是一個移動的視窗，它可以將最新的語言語法的程式碼輸入，並產出目前瀏覽器可支援的程式碼。

接下來讓我們來談談，如何將 Babel 運用於你的工作流程中。

1.3.1 Babel 轉譯器簡介

Babel 可以將運用 ES6 特徵功能的 JavaScript 程式碼轉譯為 ES5 版本的程式碼。它會產生人類可閱讀的程式碼，當我們無法對所使用的新特徵有穩固的操控能力時，它特別的有效且受到歡迎。

線上的 Babel REPL（讀取、求值、輸出迴圈，Read-Evaluate-Print Loop）（*https://mjavascript.com/out/babel-repl*）是一個學習 ES6 的絕佳地點，可省去安裝 Node.js 和 `babel` 工具的麻煩。

REPL 提供我們一個程式碼輸入區，它可以即時地自動將程式碼編譯；我們可以在原始碼右方的區域看到編譯後的程式碼。

那麼我們就來在 REPL 撰寫一些程式，你可以先用以下的程式碼來開始試試：

```
var double = value => value * 2
console.log(double(3))
// <- 6
```

在我們所輸入的程式碼右方，可以看到轉譯後 ES5 的相同功能程式碼，如圖 1-1 所示。當更新原始碼時，轉譯器的結果也會即時更新。

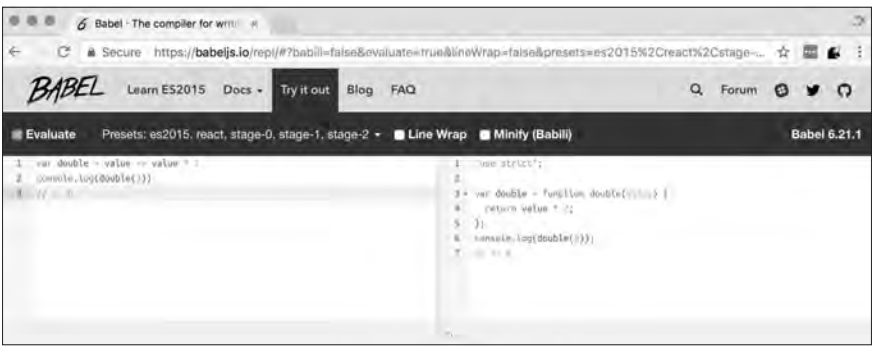

圖 1-1　線上的 Babel REPL 的實際畫面－非常棒的互動方式學習 ES6 特徵功能

Babel REPL 是一個有效的工具，可以嘗試本書所介紹的特徵功能。然而需注意的是，Babel 並不會轉譯新的內建功能，例如：`Symbol`、`Proxy` 和 `WeakMap`。對這些內建功能則不會進行轉譯，而是依據執行期間執行 Babel 的輸出，來提供這些內建功能。如果想要在執行期間支援尚未被實作的內建功能，我們可以在程式碼中匯入 `babel-polyfill` 套件。

在舊版的 JavaScript，這些特徵功能的語義更正在實作上是很難達成的，或是說幾乎不可能。向下相容程式可能減輕這個問題，但是它們通常無法涵蓋所有的案例，且在操作上必須做一些妥協。因此我們必須謹慎使用，並在釋出有使用內建功能或向下相容的轉譯程式碼之前，完整測試。

若想要使用內建功能，最好等到瀏覽器可以完整支援新的內建功能，再開始使用。通常建議你可以考慮內建功能的替代方案。同時，最重要是學習這些特徵功能，而不要讓我們的 JavaScript 語言知識落後。

現代化的瀏覽器如 Chrome、Firefox 和 Edge，現在都可支援大部分的 ES2015 以及更新的功能；當我們想要嘗試一個指定特徵功能的語義時，使用它們的開發工具更為有效。當談到了需仰賴現代 JavaScript 特徵功能的線上服務等級的應用程式，則建議需要有一個轉譯建置的步驟，這樣你的應用程式才能夠於執行期間支援較多的 JavaScript 特徵功能。

除了 REPL 之外，Babel 提供了一個以 Node.js 套件撰寫的命令列工具。你可以透過 `npm` 這個 Node 套件管理員程式來安裝它。

下載 Node.js（*https://mjavascript.com/out/node*）。在安裝了 `node` 之後，將能夠在你的終端機上使用 `npm` 命令列工具。

在開始之前，我們會先建立一個專案目錄，以及一個 *package.json* 檔案；此檔案是一個用來描述 Node.js 應用程式的資源配置文件。我們可以透過 `npm` 命令列工具建立 *package.json* 檔案：

```
mkdir babel-setup
cd babel-setup
npm init --yes
```

將 `--yes` 參數傳遞至 `init` 命令，可設定 *package.json* 使用 `npm` 所提供的預設值，而不需要再詢問我們任何問題。

讓我們來建立一個名稱為 *example.js* 的檔案，包含以下的 ES6 程式碼。將它儲存至方才所建立的 *babel-setup* 目錄，位於 *src* 子目錄之下。

```
var double = value => value * 2
console.log(double(3))
// <- 6
```

欲進行 Babel 安裝，請將下列指令輸入至你的控制終端中：

```
npm install babel-cli@6 --save-dev
npm install babel-preset-env@6 --save-dev
```

透過 npm 安裝的套件會放置於專案目錄下的 *node_modules* 目錄。我們接著就可以建立 npm 直譯程式，或在應用程式中使用 `require` 敘述來存取這些套件。

使用 `--save-dev` 參數，可以將這些套件加入至 *package.json* 資源配置文件中，作為程式發展的相依套件；如此當複製我們的專案至新環境時，就可以藉由執行 `npm install` 來重新安裝所有的相依套件。

`@` 符號表示我們想要安裝一個特定版本的套件。因此，`@6` 表示我們告訴 npm 需安裝 `6.x` 範圍中最新版本的 `babel-cli`。這樣的偏好設定可以方便地讓應用程式不受到未來新功能的影響，因為它將不會安裝 `7.0.0` 或更新的版本，這些新版本可能會包含一些截至撰稿為止所無法預期的變更。

在接下來的步驟，我們將 *package.json* 檔案中 `scripts` 特性的值，以如下的方式取代。`babel` 命令列工具，由 `babel-cli` 所提供，能夠取得 *src* 目錄下的所有內容，並將它們編譯為所需的輸出格式，儲存結果至 *dist* 目錄中，同時將原始目錄結構保存於不同的根目錄下。

```
{
  "scripts": {
    "build": "babel src --out-dir dist"
  }
}
```

與上一個步驟我們所安裝的套件共同運作的 *package.json* 檔案，它最少需要以下的描述內容：

```
{
  "scripts": {
    "build": "babel src --out-dir dist"
  },
  "devDependencies": {
    "babel-cli": "^6.24.0",
    "babel-preset-env": "^1.2.1"
  }
}
```

在 scripts 物件中可被列舉出的所有指令，均可透過 npm run <name> 的方式執行；它會暫時地變更 $PATH 環境變數，如此我們就可以不需要安裝 babel-cli 在系統中，亦可執行 babel-cli 中的命令列工具。

如果在控制終端中執行 npm run build，你會注意到 *dist/example.js* 檔案會被建立出來。輸出的檔案會與我們原始的檔案相同，因為 Babel 不進行猜測，我們必須先對它進行設定。於 *package.json* 旁邊建立一個 *.babelrc* 檔案，並於檔案中撰寫以下的 JSON 內容：

```
{
  "presets": ["env"]
}
```

此處 env 的預先設定，是我們稍早透過 npm 所安裝，安裝時加入了一系列的外掛程式至 Babel 以將 ES6 版本的程式碼轉換為 ES5 版本。這個預先設定可以將如 *example.js* 內容中的箭頭函式，轉換為 ES5 版本的程式碼。env 設定可以啟用 Babel 程式碼轉換的外掛程式，依據最新版的瀏覽器所支援的特徵功能。這個設定是可以調整的，意味著我們可以決定需要支援到多久以前的瀏覽器；越多瀏覽器需要支援，我們轉譯出來的程式碼就越大；支援越少瀏覽器，則可滿足的使用者就越少。如往常一般，需要深入研究才能夠正確調整 Babel 的 env 設定。在預設的狀況下，每一種轉換都是啟用的，以在執行期間提供廣泛的支援度。

一旦我們再次執行所建置的程式碼，可以觀察到現在的輸出結果是有效的 ES5 程式碼：

```
» npm run build
» cat dist/example.js
"use strict"

var double = function double(value) {
  return value * 2
}
console.log(double(3))
// <- 6
```

接下來我們再看看另一個類型的工具，稱為 eslint 程式碼靜態分析工具，它可以為我們的應用程式建立一個程式碼品質基準（quality baseline）。

1.3.2 程式碼品質和一致性與 ESLint 工具

當發展一個程式碼時，我們會分析程式碼中是否有重複的部分，或有些已不再使用；並撰寫一段新的程式碼，刪除這些不需要的部分，調整程式碼以符合新的架構。當程式碼逐漸成長時，它的開發團隊也會產生變化：一開始團隊可能有幾個成員，或甚至只有一個成員；但是當專案功能逐漸成長，團隊成員數量可能也會跟著增加。

Lint 靜態語義分析工具（稱為 linter）可用於判斷語法錯誤。現代的 linter 通常可以進行自訂，協助建立程式風格規範，讓團隊中的每個人可以使用。透過遵循一致的程式撰寫規則和一個品質基準，我們可以從程式撰寫風格層面，讓團隊成員更方便協同合作。每位團隊成員對程式撰寫風格可能都有不同的看法意見；但是當我們使用 linter 並同意它的設定後，這些意見可以被濃縮為風格規則。

除了確認程式可以被解析之外，在實務上我們可能希望避免 `throw` 敘述拋出字串實字的例外錯誤；或是不允許在上線版本的程式碼中使用 `console.log` 和 `debugger` 敘述。然而，若是定義一個規則要求每個函式呼叫只允許傳遞一個引數，這樣的規則可能太過於嚴格。

當 linter 在定義和執行一個程式風格規範是有效時，在設計這些規則時我們必須特別的謹慎小心。如果 lint 的規則過於嚴格，開發人員會感覺到挫折並影響工作產出；如果 lint 的規則過於寬鬆，則無法產生具一致性的程式碼風格。

為了達到平衡，我們應該要定義可以改善應用程式中大多數使用狀況的規則。當考量一項新的規則時，我們應該問問自己這項規則是否可以顯著地改善目前的程式碼，讓新的程式碼可以持續順暢發展。

ESLint 是一個現代的 linter，它包含了數個外掛程式，並支援各類不同的規則，可以讓我們挑選使用。我們可以決定，當程式碼未遵循定義的規則時，應該將警告訊息作為輸出訊息的一部分；或是直接中止執行並拋出錯誤。我們將使用 `npm` 安裝 `eslint`，就如上一節安裝 `babel` 一樣：

```
npm install eslint@3 --save-dev
```

接著，我們需要設定 ESLint。當安裝完 `eslint` 後，我們會在 *node_modules/.bin* 中找到它的命令列工具。執行以下指令將會進入 ESLint 設定步驟，對我們的專案進行初次設定。在開始之前，訊息會詢問你需使用何種程式風格規範，請選擇「Standard」[12]，接著為設定檔格式選擇「JSON」格式：

```
./node_modules/.bin/eslint --init
? How would you like to configure ESLint?
  Use a popular style guide
? Which style guide do you want to follow? Standard
? What format do you want your config file to be in? JSON
```

除了每個規則之外，`eslint` 允許我們繼續延伸既定的規則集合，Node.js 就是一種延伸出來的模組。當需要將設定檔於多個專案或甚至社群之間分享時，這樣的功能就非常有用。在選擇「Standard」規則集合之後，我們將會看到 ESLint 於 *package.json* 中加入了一些相依設定，就是此套件選擇的是預先定義的「Standard」規則集合，接著會建立一個設定檔。名稱為 *.eslintrc.json* 並包含以下內容：

```
{
  "extends": "standard",
  "plugins": [
    "standard",
    "promise"
  ]
}
```

查看 *node_modules/.bin* 目錄是 npm 運作的實作細節，但在實作上距離理想還有一段差距。當操作它進行初始化 ESLint 的設定時，我們無法保留這個設定檔，也無法在分析程式碼時將它輸出。為了解決這個問題，可以加入一段如下方內容的 `lint` 程式碼至 *package.json*：

```
{
  "scripts": {
    "lint": "eslint ."
  }
}
```

12 請注意，「Standard」只是一個自我識別的名稱，不是實際上任何正式官方的定義。它並不是一種特別的程式風格規則，只是要求程式撰寫能夠持續一致。程式具備一致性可以協助閱讀專案程式的效率。Airbnb 的程式風格規範是目前熱門的風格，它在預設狀況下不能夠省略分號，與「Standard」不同。

此時你可能會回憶起 Babel 的範例；當執行程式時，`npm run` 可以將 *node_modules* 加入至 `PATH` 中。要用 lint 分析程式碼時，我們可以執行 `npm run lint`，而 npm 會找到 ESLint 位於 *node_modules* 目錄中的命令列介面工具。

來看看以下的 *example.js* 檔案，它的內容故意排列混亂而沒有風格規範，這樣便可以試試 ESLint 可以為我們做到什麼事情：

```
var goodbye='Goodbye!'

function hello(){
  return goodbye}

if(false){}
```

當我們執行 `lint` 程式時，ESLint 會描述出檔案內容的問題，如下圖 1-2 所示。

圖 1-2　ESLint 是一個協助產生零語法錯誤程式碼及一致程式風格的好工具

ESLint 能夠自動地修正大多數的程式風格問題，如果我們有傳入 `--fix` 參數。可以將下面的內容加入至你的 *package.json* 中：

```
{
  "scripts": {
    "lint-fix": "eslint . --fix"
  }
}
```

這樣當執行 `lint-fix` 後，我們就只會得到兩個錯誤訊息：`hello` 未被使用，以及 `false` 是一個不變的常數條件。其他的錯誤都已經被修正了，最後產生如下的程式碼。剩下的錯誤無法被自動修正，是因為 ESLint 無法推論我們的程式碼的意義，所以選擇不針對語義進行變更。藉由這樣的方式，`--fix` 就成為了一個好用的工具，可以解決程式撰寫風格的問題，也不會有程式邏輯被變更的風險。

```
var goodbye = 'Goodbye!'

function hello() {
  return goodbye
}

if (false) {}
```

有一個類似的工具可於 `prettier` 中找到（*https://mjavascript.com/out/prettier*），它能夠自動地將你的程式碼排列格式。Prettier 可以設定為自動覆寫我們的程式碼，以確保程式碼依循我們喜好的格式，例如：一定數量的縮排、單引號或雙引號、尾端逗點（trailing commas）或單行最大長度。

既然你已知道如何將 JavaScript 編譯為瀏覽器可理解，以及如何使用 lint 來標準化你的程式碼，接下來就進行 ES6 特徵主題和 JavaScript 未來的討論。

1.4 ES6 中的特徵功能主題

ES6 是很巨大的：它是語言的技術規格，自 ES5.1 開始有 258 頁的文件，至 ES6 已經增加了一倍，達到共 566 頁。每次規格的變更項目都不出以下的類別：

- 語法糖（syntactic sugar）
- 新的機制
- 更好的語義
- 更多的內建功能和方法
- 對既有的限制提出相容的解決方案

語法糖是 ES6 最重要的革新動能之一。新版本中使用新的類別語法，以簡化的方式描述物件的繼承；函式，可使用簡化語法，通常稱為箭頭函式；以及特性，使用特性值指定簡化語法。我們還會看到許多的特徵功能，例如：解構賦值、其餘、和展開，也提供了撰寫程式的新方法。第 2 章和第 3 章會介紹 ES6 的這些特徵功能。

在 ES6 中我們也可以使用多種新的機制，來描述非同步程式流程：**承諾**（*promises*），它代表著一項操作的最終結果；**迭代器**（*iterators*），它代表著一系列的值；**產生器**（*generators*），它是一個特別的迭代器，可以產生一系列的值。在 ES2017，`async/await` 則建立在於這些新的觀念和建構元件之上。我們會在第 4 章說明這些迭代和流程控制機制。

在 JavaScript 中有一些常見的案例，就是開發人員會使用單純物件（plain objects）加上隨機的字串鍵，來建立出雜湊映射。如果我們沒有謹慎使用，且讓使用者輸入了既有定義的鍵，則會容易導致錯誤。ES6 推出了一些新的內建功能來管理集合和映射，使得字串鍵的使用不再受到限制。這些集合會在第 9 章進行探討。

代理器物件重新定義了 JavaScript 反射可達到的作用。代理器物件與其他領域的代理器相似，例如：網路封包的傳遞。它們可以攔截任何與 JavaScript 物件的互動，例如：定義、刪除、或存取一個特性。若瞭解代理器的運作機制，就會知道它們在功能上是無法被完整地進行向下相容：向下相容程式（polyfill）是存在，但是會有功能上的限制，使得在某些案例上是無法相容於規格定義。在第 6 章我們將會深入瞭解解代理器能為我們完成的工作為何。

除了新的內建功能之外，ES6 還針對 `Number`、`Math`、`Array`、和字串帶來多項更新。在第 7 章，我們會來看看這些內建功能的新的實例和靜態方法。

我們也可以操作 JavaScript 原生的新模組系統。在瞭解運用於 Node.js 的 CommonJS 格式之後，第 8 章將說明所期待的原生 JavaScript 模組使用語法。

因為 ES6 推出的大量改善與更新，要將新的特徵功能與我們既有的 JavaScript 知識整合，仍會感到困難。我們用整個第 9 章的內容，來分析 ES6 中每個特徵功能的優點和重要性，這樣你就會有一些實務上的概念，可以開始使用 ES6 所提供的強大功能。