前言

本書涵蓋 Web 瀏覽器和 Node 所實作的 JavaScript 語言及 JavaScript API。我的目標讀者是之前有過一些程式設計經驗而且想要學習 JavaScript 的人，以及已經在使用 JavaScript 並且想要將他們的理解帶到下一個新階段、真正精通此語言的程式設計師。我撰寫本書的目的是詳盡且全面地記錄 JavaScript 這個語言，並為 JavaScript 程式能取用的最重要的客戶端和伺服端 API 提供深入的介紹。因此，這是一本篇幅很長且充滿細節的書籍。然而，我的希望是，你仔細研究而投入的心力，以及花在閱讀本書的時間，能以較高的程式設計生產力的形式，輕鬆得到回報。

本書的前面幾版包含了廣泛的參考章節。我不再認為有必要把那些材料以印刷形式呈現於此，因為在線上很快速且輕易就能找到最新的參考資訊。如果你有需要查找關於核心或客戶端 JavaScript 的任何資訊，我推薦你拜訪 MDN 網站（*https://developer.mozilla.org*）。至於伺服端的 Node API，我建議你直接前往源頭，查閱 Node.js 的參考說明文件（*https://nodejs.org/api*）。

本書編排慣例

本書使用下列的編排慣例：

斜體字（*Italic*）

用於強調或指出某個詞彙的第一次使用。**斜體**也用於電子郵件位址、URL 及檔案名稱。中文以楷體表示。

第一章

JavaScript 簡介

JavaScript 是 Web 的程式語言。絕大部分的網站都使用 JavaScript，而所有現代的 Web 瀏覽器，不管是桌上型電腦、平板電腦或手機所使用的，都包含 JavaScript 直譯器（interpreters），使得 JavaScript 成為歷史上最廣為部署的程式語言。在過去十年來，Node.js 讓 JavaScript 能在 Web 瀏覽器之外進行程式設計，而 Node 戲劇化的成功，正意味著 JavaScript 現在也是在軟體開發人員之間，最廣為使用的程式語言。無論你是從頭開始，或已經是專業的 JavaScript 使用者，這本書都將會協助你精通此語言。

如果你已經熟悉其他的程式語言，知道 JavaScript 是高階的、動態的、直譯式的程式語言，適用物件導性或函式型（functional）程式設計風格，可能會對你有所幫助。JavaScript 的變數是不具型（untyped）的。其語法大致以 Java 為基礎，但除此之外，這兩個語言就沒有相關之處。JavaScript 的一級函式（first-class functions）衍生自 Scheme，而它基於原型的繼承（prototype-based inheritance）則源自於較少人知道的語言 Self。但你並不需要知道任何的這些其他語言，或熟悉這些術語，就能運用本書並學習 JavaScript。

「JavaScript」這個名稱相當有誤導之虞。除了膚淺的語法相似度，JavaScript 是與 Java 完全不同的程式語言，而且 JavaScript 早就成長到超越其指令稿語言（scripting-language）根基，成為了穩健且有效率的通用語言，適用於嚴肅認真的軟體工程，以及具有大型源碼庫（codebases）的專案。

JavaScript：名稱、版本與模式（Modes）

JavaScript 是在 Web 的早期於 Netscape 被創造出來的，嚴格來說，「JavaScript」是 Sun Microsystems（現在的 Oracle）授權的商標，用以描述 Netscape（現在的 Mozilla）對於此語言的實作。Netscape 把這個語言提送到 ECMA（European Computer Manufacturer's Association）進行標準化，而由於商標的問題，此語言標準化的版本被困在了「ECMAScript」這個怪異名稱之中。實務上，每個人都單純稱呼這個語言為 JavaScript。本書使用「ECMAScript」這個名稱及其縮寫「ES」來指稱此語言的標準（standard）及該標準的各個版本（versions）。

2010 年代的大部分時間，所有的 Web 瀏覽器都有支援 ECMAScript 標準的第 5 版。本書將 ES5 視為相容性（compatibility）的基準線，並且不再討論這個語言更之前的版本。ES6 在 2015 年發行，新增了主要的幾個新功能，包括類別（class）和模組（module）語法，使得 JavaScript 從指令稿語言轉化為適用於大規模軟體工程的嚴肅且通用的語言。從 ES6 開始，ECMAScript 規格（specification）的發展步調就變為每年發行（yearly release），而此語言的版本，例如 ES2016、ES2017、ES2018、ES2019，現在也都是以發行年分來識別。

隨著 JavaScript 的演進，語言設計師們試著更正早期（ES5 之前）版本的缺陷。為了維持回溯相容性（backward compatibility），我們不可能把傳統的功能移除，不論它們的缺點有多大。但在 ES5 與之後的版本中，程式可以選擇使用 JavaScript 的 *strict mode*（嚴格模式），其中有幾個早期的語言錯誤都被更正了。選用的機制是會在 §5.6.3 中描述的「use strict」指引（directive）。那節也總結了傳統 JavaScript 和嚴格 JavaScript 之間的差異。在 ES6 和後續版本中，對於新語言功能的使用通常隱含著嚴格模式的調用。例如，假設你使用 ES6 的 `class` 關鍵字，或建立了一個 ES6 的模組，那麼在該類別或模組中的所有程式碼，都會自動變成嚴格的，而舊的、有缺陷的功能在那些情境下將無法取用。本書會涵蓋 JavaScript 的傳統功能，但會小心指出它們在嚴格模式中無法使用。

為了發揮用處，每個語言都必須有一個平台（platform）或標準程式庫（standard library），以進行像是基本輸入輸出那類的事情。核心的 JavaScript 語言定義了一個最小的 API 用以處理數字、文字、陣列、集合、映射等等的東西，但並不包含任何輸入

或輸出的功能性。輸入與輸出（以及更為精密的功能，例如網路、儲存空間和繪圖）是 JavaScript 內嵌的「host environment（宿主環境）」所負責的。

JavaScript 原本的 host environment 是 Web 瀏覽器，而這仍是 JavaScript 程式碼最常見的執行環境。Web 瀏覽器環境能讓 JavaScript 程式碼獲取來自使用者滑鼠和鍵盤或發出 HTTP 請求（requests）所取得的輸入。而它允許 JavaScript 程式碼以 HTML 和 CSS 顯示輸出給使用者看。

從 2010 年開始，JavaScript 程式碼就有另一個 host environment 可用。不限制 JavaScript 非得使用 Web 瀏覽器所提供的 API，Node 賦予 JavaScript 存取整個作業系統（operating system）的能力，讓 JavaScript 程式能夠讀寫檔案、透過網路發送與接收資料，以及發出或回應 HTTP 請求。Node 是實作 Web 伺服器相當受歡迎的一種選擇，也是撰寫簡單工具指令稿（utility scripts）來取代 shell scripts 的一種便利工具。

本書主要專注於 JavaScript 語言本身。第 11 章含有 JavaScript 標準程式庫的說明文件、第 15 章介紹 Web 瀏覽器的 host environment，而第 6 章介紹 Node 的 host environment。

本身會先涵蓋底層的基本知識，然後以它們為基礎來說明更複雜且高階的抽象層（abstractions）。這些章節的設計或多或少是要讓讀者循序閱讀的。不過學習一個新程式語言的過程從來都不是線性（linear）的，而對於一個語言的描述也同樣不是線性的：每個語言功能都與其他功能有關，因此本書滿是對相關材料的交互參考，有時往後，有時往前。這個簡介章節讓我們快速地看過這個語言一遍，介紹會讓後續章節的深入討論更容易理解的關鍵功能。如果你已經是在實務上使用 JavaScript 的程式設計師，你大概可以跳過本章（不過繼續移動之前，你可能會覺得本章結尾的範例 1-1 讀起來很有趣）。

1.1 探索 JavaScript

學習一個新的程式語言時，很重要的是要去嘗試書中的範例，然後修改它們，並再試一次，以測試你對該語言的理解。為了那麼做，你會需要一個 JavaScript 直譯器（interpreter）。

要試試數行的 JavaScript 程式碼，最簡單的方式就是開啟你 Web 瀏覽器的 Web 開發人員工具（developer tools，使用 F12、Ctrl-Shift-I 或 Command-Option-I），然後選取 Console（主控台）分頁。然後你就能在命令提示列（prompt）輸入程式碼，並在輸入過程中觀察結果。瀏覽器的開發人員工具經常會以分隔窗格的形式出現在瀏覽器視窗的底部或右邊，但你通常也能將它們拆離，作為分別的視窗（如圖 1-1 所示），那樣會很方便。

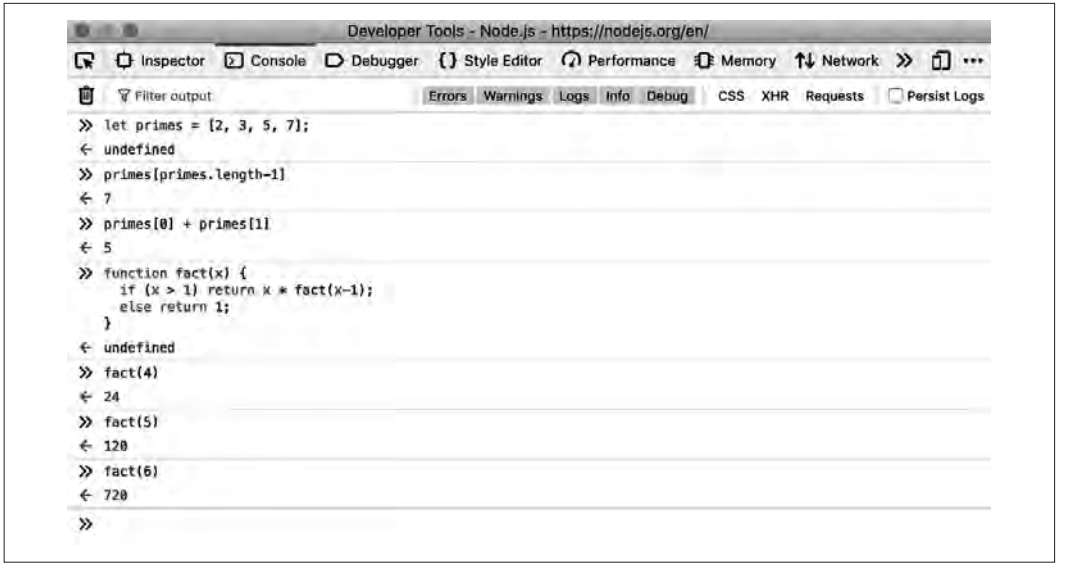

圖 1-1　Firefox 開發人員工具中的 JavaScript 主控台

嘗試 JavaScript 程式碼的另一種方式是從 *https://nodejs.org* 下載並安裝 Node。一旦在你的系統上安裝好 Node，你只要打開一個 Terminal（終端機）視窗，然後輸入 **node**，就能開啟一個互動式的 JavaScript 工作階段（session），像這一個：

```
$ node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press ^C to abort current expression, ^D to exit the repl
> let x = 2, y = 3;
undefined
> x + y
5
```

```
> (x === 2) && (y === 3)
true
> (x > 3) || (y < 3)
false
```

1.2 Hello World

當你準備好開始以較長的程式碼片段進行實驗，那些逐行輸入的互動式環境可能不再合適，而你大概會偏好在文字編輯器（text editor）裡面撰寫你的程式碼。在那裡，你可以把程式碼複製貼上到 JavaScript 主控台或 Node 的工作階段。又或者你可以把程式碼儲存到一個檔案（JavaScript 程式碼傳統的延伸檔名是 *.js*），然後以 Node 執行那個 JavaScript 程式碼檔案：

```
$ node snippet.js
```

如果你以這種非互動式的方法使用 Node，它不會自動印出你所執行的所有程式碼的值，所以你得自己那麼做。你能使用函式 `console.log()` 在你的終端機視窗或瀏覽器的開發人員工具主控台中顯示文字和其他的 JavaScript 值。所以，舉例來說，如果你建立了一個 *hello.js* 檔案，其中含有這行程式碼：

```
console.log("Hello World!");
```

並以 `node hello.js` 執行該檔案，你就會看到訊息「Hello World!」被印出來。

如果你希望相同的那個訊息被印出到 Web 瀏覽器的 JavaScript 主控台中，就創建名為 *hello.html* 的一個新檔案，然後把這段文字放進去：

```
<script src="hello.js"></script>
```

然後使用像下面這樣的一個 `file://`URL 把 *hello.html* 載入到你的 Web 瀏覽器中：

```
file:///Users/username/javascript/hello.html
```

開啟開發人員工具視窗在主控台中查看這個招呼訊息。

1.3 JavaScript 導覽

本節透過程式碼範例提供了快速的 JavaScript 語言簡介。在這個介紹章節之後，我們會深入 JavaScript 的最底層：第 2 章說明像是註解、分號和 Unicode 字元集之類的東西。第 3 章就開始變得更加有趣了：它解說 JavaScript 的變數，以及你能夠指定給那些變數的值。

這裡有些範例程式碼用於演示那兩章的重點：

```
// 跟在雙斜線（double slashes）之後的任何東西都是註解（comment）。
// 仔細閱讀這些註解：它們解說 JavaScript 程式碼。
// 一個變數（variable）是一個值（value）的符號名稱。
// 變數是以 let 關鍵字來宣告的：
let x;                    // 宣告一個名為 x 的變數。

// 值能以一個 = 符號被指定給變數
x = 0;                    // 現在變數 x 有 0 這個值
x                         // => 0：一個變數估算（evaluates）為它的值。

// JavaScript 支援數種型別的值
x = 1;                    // 數字。
x = 0.01;                 // 數字可以是整數或實數。
x = "hello world";        // 在引號中的文字字串。
x = 'JavaScript';         // 單引號也能用來界定字串。
x = true;                 // 一個 Boolean 值。
x = false;                // 另一個 Boolean 值。
x = null;                 // Null 是一個代表「沒有值（no value）」的特殊值。
x = undefined;            // Undefined 是像 null 的另一個特殊值。
```

JavaScript 程式可以操作的另外兩個重要**型別**（*types*）是物件（objects）和陣列（arrays），它們是第 6 章和第 7 章的主題，但它們重要到你在抵達那些章節前，就會碰到它們很多次：

```
// JavaScript 最重要的資料型別是物件。
// 一個物件是名稱與值對組（name/value pairs）的一個群集，或字串對值的一種映射（map）。
let book = {              // 物件是以大括號（curly braces）圍起。
    topic: "JavaScript",  // 特性「topic」有「JavaScript」這個值。
    edition: 7            // 特性「edition」有值 7
};                        // 這個大括號標示一個物件的結尾。

// 使用 . 或 [] 來存取一個物件的特性：
book.topic                // =>「JavaScript」
book["edition"]           // => 7：存取特性值的另一種方式。
book.author = "Flanagan"; // 藉由指定創建新的特性。
```

```
book.contents = {};        // {} 是沒有特性的一個空物件。

// 以 ?. 條件式存取特性（ES2020）：
book.contents?.ch01?.sect1 // => undefined：book.contents 沒有 ch01 特性。

// JavaScript 也支援值的陣列（數值索引的串列）：
let primes = [2, 3, 5, 7]; // 有四個值的一個陣列，以 [ 和 ] 界定。
primes[0]                  // => 2：陣列的第一個元素（索引 0）。
primes.length              // => 4：陣列中有多少個元素。
primes[primes.length-1]    // => 7：陣列的最後一個元素。
primes[4] = 9;             // 藉由指定新增一個新的元素。
primes[4] = 11;            // 或透過指定更動一個現有的元素。
let empty = [];            // [] 是沒有元素的一個空陣列。
empty.length               // => 0

// 陣列和物件可以存放其他的陣列和物件：
let points = [             // 有兩個元素的一個陣列。
    {x: 0, y: 0},          // 每個元素都是一個物件。
    {x: 1, y: 1}
];
let data = {                 // 有兩個特性的一個物件
    trial1: [[1,2], [3,4]],  // 每個特性的值都是一個陣列。
    trial2: [[2,3], [4,5]]   // 這個陣列的元素也是陣列。
};
```

程式碼範例中的註解語法

你可能已經注意到，在前面的程式碼中，某些註解（comments）是以一個箭號（=>）開頭的，這顯示出註解前的程式碼所產生的值，我以這種方式試著在印刷書籍中模擬互動式的 JavaScript 環境（像是 Web 瀏覽器主控台）。

//=> 也作**斷言**（*assertion*）之用，而我也撰寫了一個工具來測試程式碼，驗證產生的正是註解中所指定的值。我希望這對減少本書中的錯誤有所幫助。

關於這些註解與斷言，有兩種相關的風格存在。如果你看到像 `//a == 42` 這種形式的註解，它意味著註解前的程式碼執行後，該變數會有值 42。如果你看到 `//!` 這種形式的註解，它代表該行中註解前的程式碼會擲出一個例外（而驚嘆號之後其餘的註解通常會說明所擲出的是何種例外）。

你在本書各處都會看到以這種方式使用的註解。

在此所展示的，在中括號（square braces，或稱「方括號」）中列出陣列元素，或在大括號（curly braces，或稱「曲括號」）內映射物件特性名稱至特性值的語法，被稱為**初始器運算式**（*initializer expression*），它僅是第 4 章的主題之一。一個**運算式**（*expression*）是 JavaScript 的一種片語（phrase），可被**估算**（*evaluated*）而產生一個值。舉例來說，使用 . 和 [] 來參考一個物件特性或陣列元素的值，就是一種運算式。

在 JavaScript 中構成運算式最常見的方式之一就是使用**運算子**（*operators*）：

```
// 運算子作用於值（運算元）來產生一個新的值。
// 算術運算子是最簡單的一種：
3 + 2                    // => 5：加
3 - 2                    // => 1：減
3 * 2                    // => 6：乘
3 / 2                    // => 1.5：除
points[1].x - points[0].x  // => 1：更複雜的運算元也行得通
"3" + "2"                // => "32"：+ 相加數字、串接字串

// JavaScript 定義了一些簡寫的算術運算子
let count = 0;           // 定義一個變數
count++;                 // 遞增該變數
count--;                 // 遞減該變數
count += 2;              // 加 2：等同於 count = count + 2;
count *= 3;              // 乘 3：等同於 count = count * 3;
count                    // => 6：變數名稱也是運算式。

// 相等性和關係運算子測試兩個值是否相等、
// 不等、小於、大於等等。它們會估算為 true 或 false。
let x = 2, y = 3;        // 這些 = 符號是指定而非相等性測試
x === y                  // => false：相等性（equality）
x !== y                  // => true：不等性（inequality）
x < y                    // => true：小於（less-than）
x <= y                   // => true：小於或等於（less-than or equal）
x > y                    // => false：大於（greater-than）
x >= y                   // => false：大於或等於（greater-than or equal）
"two" === "three"        // => false：這兩個字串不同
"two" > "three"          // => true：「tw」在字母順序（alphabetically）大於「th」
false === (x > y)        // => true：false 等於 false

// 邏輯運算子結合或反轉 boolean 值
(x === 2) && (y === 3)   // => true：兩個比較都是 true。&& 是 AND
(x > 3) || (y < 3)       // => false：兩個比較都不是 true。|| 是 OR
!(x === y)               // => true：! 反轉一個 boolean 值
```

如果 JavaScript 運算式像是片語，那麼 JavaScript **述句**（*statements*）就像是完整的句子（sentences）。述句是第 5 章的主題。粗略來說，一個運算式是會計算出一個值，但不會做任何事情的東西：它不會以任何方式更動程式的狀態（program state）。另一方面，述句則沒有值，但它們會更動狀態。你已經在上面見過變數的宣告（declarations）和指定（assignment）述句。另一大類述句是**控制結構**（*control structures*），例如條件式（conditionals）或迴圈（loops）。你會在下面看到例子，在我們涵蓋函式之後。

一個**函式**（*function*）是具名（named）且參數化（parameterized）的一個 JavaScript 程式碼區塊，只要定義一次，就能不斷調用（invoke）。函式的正式介紹要等到第 8 章，但就跟物件和陣列一樣，到達那章之前你就會多次看到它們。這裡是一些簡單的例子：

```
// 函式是我們能夠調用的參數化 JavaScript 程式碼區塊。
function plus1(x) {         // 定義名為「plus1」並具有參數「x」的一個函式
    return x + 1;           // 回傳比傳入之值大 1 的一個值。
}                           // 函式包在大括號中

plus1(y)                    // => 4：y 是 3，所以此次調用回傳 3+1

let square = function(x) { // 函式是能指定給變數的值
    return x * x;           // 計算函式的值
};                          // 分號（semicolon）標示這個指定的結尾。

square(plus1(y))            // => 16：在一個運算式中調用兩個函式
```

在 ES6 與後續版本中，函式的定義有一種簡寫語法可用。這種簡潔的語法使用 => 來分隔引數列（argument list）和函式主體（function body），因此以這種方式定義的函式被稱為**箭號函式**（*arrow functions*）。箭號函式最常用在你想要傳入一個不具名的函式作為引數給另一個函式之時。上面的程式碼以箭號函式改寫之後，會像這樣：

```
const plus1 = x => x + 1;   // 輸入 x 映射至輸出 x + 1
const square = x => x * x;  // 輸入 x 映射至輸出 x * x
plus1(y)                    // => 4：函式的調用也相同
square(plus1(y))            // => 16
```

以物件使用函式時，我們就得到**方法**（*methods*）：

```
// 當函式被指定給一個物件的特性，我們就稱呼
// 它們為「方法」。所有的 JavaScript 物件（包括陣列）都有方法：
let a = [];                 // 創建一個空的陣列
a.push(1,2,3);              // push() 方法新增元素到一個陣列
a.reverse();                // 另一個方法：反轉元素的順序

// 我們也能定義自己的方法。「this」關鍵字指涉方法
// 在其上被定義的那個物件：在此，就是前面的 points 陣列。
```

```
points.dist = function() { // 定義一個方法來計算點之間的距離
    let p1 = this[0];      // 我們在其上被調用的陣列之第一個元素
    let p2 = this[1];      // 「this」(這個)物件的第二個元素
    let a = p2.x-p1.x;     // x 坐標的差
    let b = p2.y-p1.y;     // y 坐標的差
    return Math.sqrt(a*a + // 畢氏定理(Pythagorean theorem)
                     b*b); // Math.sqrt() 計算平方根(square root)
};
points.dist()              // /=> Math.sqrt(2):我們的兩個點之間的距離
```

現在，如承諾過的，這裡有些函式在其主體中展示了常見的 JavaScript 控制結構述句：

```
// JavaScript 述句包括條件式和迴圈，使用
// C、C++、Java 與其他語言的語法。
function abs(x) {          // 計算絕對值(absolute value)的一個函式
    if (x >= 0) {          // if 述句 ...
        return x;          // 如果比較為 true 就執行這段程式碼。
    }                      // 這是 if 子句的結尾。
    else {                 // 選擇性的 else 子句，會在
        return -x;         // 比較為 false 時執行。
    }                      // 每個子句只有 1 個述句時，大括號是選擇性的。
}                          // 注意到回傳(return)述句在 if/else 中內嵌為巢狀。
abs(-10) === abs(10)       // => true

function sum(array) {      // 計算一個陣列之元素的總和(sum)
    let sum = 0;           // 從初始總和值 0 開始。
    for(let x of array) {  // 迴圈處理陣列，將每個元素指定給 x。
        sum += x;          // 把元素值加到總和。
    }                      // 這是迴圈的結尾。
    return sum;            // 回傳總和。
}
sum(primes)                // => 28:前 5 個質數的總和 2+3+5+7+11

function factorial(n) {    // 計算階乘(factorials)的一個函式
    let product = 1;       // 從乘積 1 開始
    while(n > 1) {         // 只要 () 中的運算式為 true 就重複 {} 中的述句
        product *= n;      // product = product * n; 的簡寫
        n--;               // n = n - 1 的簡寫
    }                      // 迴圈結尾
    return product;        // 回傳乘積(product)
}
factorial(4)               // => 24:1*4*3*2

function factorial2(n) {   // 使用不同迴圈的另一個版本
    let i, product = 1;    // 從 1 開始
    for(i=2; i <= n; i++)  // 自動遞增 i，從 2 到 n
        product *= i;      // 每次都這樣做。1 行的迴圈不需要 {}
```

```
    return product;         // 回傳階乘值
}
factorial2(5)               // => 120: 1*2*3*4*5
```

JavaScript 支援物件導向（object-oriented）的程式設計風格，但它與「典型」的物件導向程式語言有很大的不同。第 9 章會詳細涵蓋 JavaScript 的物件導向程式設計，並以大量範例進行說明。這裡有一個非常簡單的例子，示範如何定義一個 JavaScript 類別（class）來表示 2D 的幾何點（geometric points）。是這個類別之實體（instances）的物件會有單一個方法，名為 distance()，計算該點與原點（origin）之間的距離：

```
class Point {               // 依照慣例，類別名稱會首字母大寫（capitalized）。
    constructor(x, y) {     // 初始化新實體用的建構器（constructor）函式。
        this.x = x;         // this 關鍵字是被初始化的那個新物件。
        this.y = y;         // 將函式引數儲存為物件的特性。
    }                       // 建構器函式中不需要回傳。

    distance() {            // 計算原點至該點間距離的方法。
        return Math.sqrt(   // 回傳 x² + y² 的平方根。
            this.x * this.x +  // this 指涉 distance 方法
            this.y * this.y    // 在其上被調用的那個 Point 物件。
        );
    }
}

// 以「new」使用 Point() 建構器函式來創建 Point 物件
let p = new Point(1, 1);    // 幾何點 (1,1)。

// 現在使用 Point 物件 p 的一個方法
p.distance()                // => Math.SQRT2
```

這個 JavaScript 基本語法和特色的簡介之旅就在此結束，但接下來本書會有自成一體的各個章節涵蓋此語言額外的功能：

第 *10* 章，模組

展示一個檔案或指令稿中的 JavaScript 程式碼如何使用定義在其他檔案或指令稿中的 JavaScript 函式和類別。

第 *11* 章，*JavaScript* 標準程式庫

涵蓋所有 JavaScript 程式都能取用的內建（built-in）函式和類別。這包括重要的資料結構，像是映射（maps）、集合、用於文字模式比對的一個正規表達式（regular expression）類別、序列化 JavaScript 資料結構的函式，等等更多功能。

第 *12* 章，迭代器與產生器

解釋 `for/of` 迴圈如何運作，以及你如何讓自己的類別變成是 `for/of` 可迭代（iterable）的。它也涵蓋產生器函式和 `yield` 述句。

第 *13* 章，非同步 *JavaScript*

本章深入探討了 JavaScript 的非同步程式設計（asynchronous programming），涵蓋回呼（callbacks）與事件（events）、基於 Promise 的 API，以及 `async` 和 `await` 關鍵字。雖然核心的 JavaScript 語言不是非同步的，在 Web 瀏覽器和 Node 中，非同步 API 都是預設值，而本章介紹使用這些 API 的技巧。

第 *14* 章，*Metaprogramming*

介紹 JavaScript 的一些進階功能，撰寫程式庫以供其他開發人員使用的程式設計師可能會對這些功能感興趣。

第 *15* 章，*Web* 瀏覽器中的 *JavaScript*

介紹 Web 瀏覽器的宿主環境（host environment），說明 Web 瀏覽器如何執行 JavaScript 程式碼，並涵蓋 Web 瀏覽器所定義的許多最為重要的 API。

第 *16* 章，使用 *Node* 的伺服端 *JavaScript*

介紹 Node 的宿主環境，涵蓋基本的程式設計模型，以及要了解的最重要的資料結構和 API。

第 *17* 章，*JavaScript* 工具和擴充功能

涵蓋值得知道的工具和語言擴充功能，因為它們被廣為使用，並且可能讓你成為更有生產力的程式設計師。

1.4 範例：字元次數的直方圖

本章以一個簡短但有其重要性的 JavaScript 程式作為總結。範例 1-1 是一個 Node 程式，它會從標準輸入（standard input）讀取文字，從那些文字計算出字元次數的直方圖（character frequency histogram），然後印出那個直方圖。你可以像這樣調用該程式來分析它自己原始碼的字元次數：

```
$ node charfreq.js < charfreq.js
T: ########## 11.22%
E: ########## 10.15%
R: ####### 6.68%
S: ###### 6.44%
A: ###### 6.16%
N: ###### 5.81%
O: ##### 5.45%
I: ##### 4.54%
H: #### 4.07%
C: ### 3.36%
L: ### 3.20%
U: ### 3.08%
/: ### 2.88%
```

此範例用到了幾個進階的 JavaScript 功能，主要是要展示真實世界中的 JavaScript 程式看起來可能是怎樣。你不應該預期目前能夠完全理解這段程式碼，但請放心，在接下來的章節中，這裡的各個部分都會加以說明。

範例 *1-1*　以 *JavaScript* 計算字元次數的直方圖

```
/**
 * 這個 Node 程式會從標準輸入讀取文字，計算那段文字中
 * 每個字母的出現次數，並為最常使用的字元顯示
 * 一個直方圖。這需要 Node 12 或更高版本才能執行。
 *
 * 在 Unix 類型的環境中，你可以像這樣調用此程式：
 *    node charfreq.js < corpus.txt
 */

// 這個類別擴充 Map 以讓 get() 方法在鍵值（key）
// 不在映射（map）中時，回傳指定的值而非 null
class DefaultMap extends Map {
    constructor(defaultValue) {
        super();                           // 調用超類別（superclass）的建構器
        this.defaultValue = defaultValue; // 記住預設值
    }

    get(key) {
        if (this.has(key)) {               // 如果該鍵值已經在映射中
            return super.get(key);         // 從超類別回傳其值。
        }
        else {
            return this.defaultValue;      // 否則回傳預設值
        }
```

```
    }
}

// 此類別計算並顯示字母次數直方圖
class Histogram {
    constructor() {
        this.letterCounts = new DefaultMap(0);  // 從字母到次數的映射
        this.totalLetters = 0;                  // 總共有多少字母
    }

    // 這個函式以文字的字母更新直方圖。
    add(text) {
        // 從文字移除空白，並將之轉為大寫（upper case）
        text = text.replace(/\s/g, "").toUpperCase();

        // 現在以迴圈處理過文字的字元
        for(let character of text) {
            let count = this.letterCounts.get(character); // 取得舊的次數
            this.letterCounts.set(character, count+1);    // 遞增它
            this.totalLetters++;
        }
    }

    // 把這個直方圖轉換為一個字串，顯示一個 ASCII 圖形
    toString() {
        // 把這個 Map 轉換為由 [key,value] 陣列所構成的一個陣列
        let entries = [...this.letterCounts];

        // 以次數（count）排序此陣列，然後再以字母順序排列
        entries.sort((a,b) => {             // 定義排序順序的一個函式。
            if (a[1] === b[1]) {            // 如果次數相同
                return a[0] < b[0] ? -1 : 1; // 依照字母順序排列。
            } else {                        // 如果次數不同
                return b[1] - a[1];         // 以最大的次數排序。
            }
        });

        // 將次數轉換為百分比（percentages）
        for(let entry of entries) {
            entry[1] = entry[1] / this.totalLetters*100;
        }

        // 捨棄小於 1% 的任何項目
        entries = entries.filter(entry => entry[1] >= 1);

        // 現在把每個項目轉換為一行文字
```

```
        let lines = entries.map(
            ([l,n]) => `${l}: ${"#".repeat(Math.round(n))} ${n.toFixed(2)}%`
        );

        // 並回傳串接起來的文字行，以 newline（換行）字元分隔。
        return lines.join("\n");
    }
}

// 這個 async（回傳 Promise 的）函式會創建一個 Histogram 物件，
// 非同步地從標準輸入讀取成塊的文字，並把那些文字塊加到
// 直方圖。抵達串流結尾時，它會回傳這個直方圖
async function histogramFromStdin() {
    process.stdin.setEncoding("utf-8"); // 讀取 Unicode 字串，而非位元組（bytes）
    let histogram = new Histogram();
    for await (let chunk of process.stdin) {
        histogram.add(chunk);
    }
    return histogram;
}

// 最後的這行程式碼是程式的主要部分。
// 它從標準輸入製作出一個 Histogram 物件，然後印出那個直方圖。
histogramFromStdin().then(histogram => { console.log(histogram.toString()); });
```

1.5 總結

本書以從下到上的方式解說 JavaScript。這意味著我們會從底層的細節開始，例如註解、識別字（identifiers）、變數和型別，接著再往上搭建運算式、述句、物件，以及函式，然後涵蓋高階的語言抽象層，像是類別和模組。我很認真看待本書書名中的**大全**（*definitive*）這個詞，而接下來的章節會以乍看之下可能令人抗拒的詳細程度解說這個語言。然而，真正精通 JavaScript 需要對細節的理解，而我希望你會撥出時間從頭到尾閱讀本書。但請不用認為你初次閱讀就得那樣做。如果你發現自己被困在某個章節中，請跳到下一節沒關係。一旦你對此語言有了整體的認知，並能運用它時，你可以再次回頭去掌握那些細節。