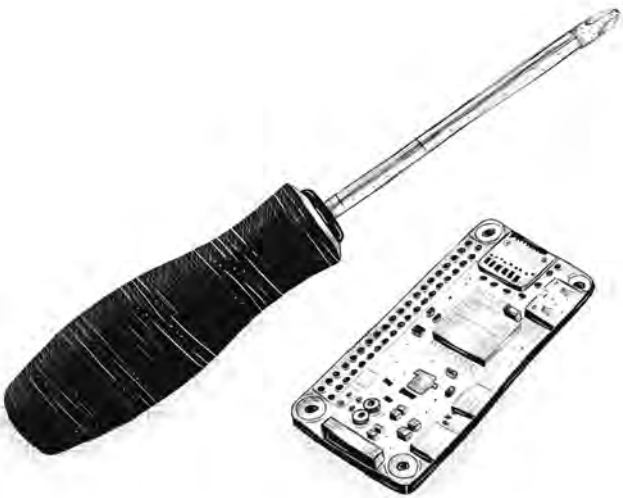
「當我們以為自己是依照自身規劃的目的進行系統開發，相信一切正照著我們的想像進行，但是……電腦真的跟我們想的不一樣。它只能投射出我們想法中非常微小的一部份：這部分主要是貢獻在邏輯、順序、規則和清晰度。」

— Ellen Ullman，
《*Close to the Machine: Technophilia and its Discontents*》作者



前言

本書的目的是教你如何對電腦下指令。在這個年代，電腦的存在就跟螺絲起子一樣普遍，但相較於螺絲起子，電腦當然是十分複雜的東西，要讓它們依照我們的意願行動，並非總是那麼容易。

如果你想要借助電腦完成的工作任務，是屬於常見而且易於理解的那一類，例如，讓電腦顯示電子郵件或充當計算機，目前電腦上都有適合的應用程式可以幫助你處理這類的工作，你只要開啟應用程式即可。然而，針對那些特定或是開放性的工作任務，或許就沒有現成的應用程式可以直接使用。

這就是程式設計出場的時刻。**程式設計**就是指建立**程式**的動作——利用一組精確的指令，告訴電腦該做什麼事，而且，因為電腦是一隻笨拙又頑冥不靈的野獸，所以程式設計的過程基本上十分乏味又令人沮喪。

幸好，如果你能克服並且接受現實情況就是如此，當你思考如何讓這些笨拙的機器幫你處理工作時，或許還能享受絞盡腦汁所帶來的樂趣。程式設計是一件十分有意義的事，原本手動處理要做到地老天荒的工作，有了程式的幫助，可以讓你在短短幾秒內快速完成。程式設計不僅能讓電腦這項工具完成以前做不到的事，還能就抽象思考方面，提供一種很棒的練習方式。

絕大部分的程式設計都是利用**程式語言**編寫而成，程式語言是一種人造語言，其創造目的是用來對電腦下指令。有趣的一點是，我們發現和電腦溝通時，最有效的做法是大量借用人類彼此之間溝通的方式。電腦語言和人類語言一樣，允許以新的方式將單字和片語組合在一起，從而創造出表達新概念的機會。

以文字語言作為人類與電腦互動的介面曾經蔚為風潮，例如，1980 年代的 BASIC 和 1990 年代的 DOS 命令提示字元。大部分的方法現在已經被視覺介面所取代，這種介面雖然易於學習，但提供的自由度較低。電腦語言依舊存在，只要你知道該去哪裡找它們。當今每一個網頁瀏覽器裡，都有內建像 JavaScript 這樣的程式語言，因此幾乎所有設備都能使用。

本書希望幫助你對這種程式語言擁有足夠的熟悉度，讓你能用程式語言創造出有用和有趣的事物。

程式設計的基本原則

除了說明 JavaScript 的相關知識，本書還會介紹程式設計的基本原則。說真的，程式設計很難。雖然基本規則很簡單也很清楚，但是建立在這些基本規則之上的程式，往往會因為導入程式自身的規則與複雜度，而變得更為複雜。某種程度上，寫程式這件事就像是在建立你自己的迷宮，而你可能會在建造過程裡迷失其中。

閱讀本書時，有時會陷入令人相當沮喪的情況。如果你是程式新手，會有很多新知識等你去消化。然後，大部分的知識都需要你結合其他方法來進行組合。

學習過程中是否需要投入額外的努力，取決於你自己。在你奮力掙扎著跟上本書的內容時，千萬不要先對自己的能力下任何結論。一切都會順利，你只需要堅持下去。遇到挫折時，請讓自己休息一下，然後再重新閱讀某些內容，確定自己看了也理解範例程式和練習題。學習是相當艱苦的工作，但你學到的一切都將成為你的知識，而且會讓後續的學習變得更加容易。

> 如果行動沒有收穫，那就搜集情報；
> 如果搜集情報也沒有收穫，那就乾脆睡大覺。
>
> — Ursula K. Le Guin，《黑暗的左手（三版）》作者

程式有許多面向。你可以說程式是程式設計師輸入的一段文字，也可以說程式是一種引導電腦完成工作的力量，它是電腦記憶體裡的資料，卻能控制同一塊記憶體上所要執行的動作。如果要試著以我們熟悉的事物來比喻程式，往往不足，很難找到適合的比喻。表面上勉強適合的比喻是，程式就像是一台機器，通常會包含大量的個別零件，而且會運作整體事物；我們必須思考出某些方法，讓這些零件互相連結，而且對整體運作做出貢獻。

電腦這個實體機器掌管了許多無形的機器，其本身只會傻傻地做簡單的事。電腦之所以如此有用的原因在於，它們能以令人無法置信的速度，快速完成許多工作。程式的作用是巧妙地將大量簡單的動作結合在一起，以完成非常複雜的事情。

程式的工作是建立想法，不需要建構成本而且沒有重量，只要手指在鍵盤上敲敲打打就能輕易發展出程式。

但是，程式如果缺乏照顧，任其發展，程式大小和複雜度會失去控制，甚至會讓程式開發者感到困惑。因此，程式設計最主要的問題是，在我們能掌控的範圍內發展程式。程式運作時很美，程式設計的藝術可說是一門控制複雜度的技術。最棒的程式是受到控制的程式，是能簡化複雜度的程式。

某些程式設計師相信，要管理程式複雜度最好的方法是，只在程式裡使用一小部分自己完全理解的技術。他們擬了嚴格的規則（「最佳實務做法」），規定程式應該具備的形式，並且小心地待在自己的小小安全區內。

這樣的做法不僅無趣，而且不具效益。新問題通常需要新的解決方案，程式設計領域相當年輕，還在快速發展中，變化很大，有足夠的空間容納廣大不同的方法。程式設計的過程中會發生許許多多可怕的錯誤，但你應該繼續犯錯，才能理解這些錯誤。從開發實務中感受何謂出色的程式，而非從一大串規則中去學習。

程式語言的重要性

電腦誕生之初，那時還沒有程式語言，程式看起來就像下面這個樣子：

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

上面這個程式的內容是將數字 1 到 10 相加，然後印出計算結果：`1 + 2 + ... + 10 = 55`，只要簡單的機器就能執行。早期的電腦要寫程式，必須將大量的開關設置在正確的位置，或是在打孔卡上打洞，再讓電腦讀取打孔卡。你或許能想像得到，這是多麼繁瑣而且容易出錯的程序。即使是寫一個簡單的程式也需要很多聰明和紀律，其複雜的程度幾乎令人無法想像。

當然啦，手動輸入這些晦澀難懂的神秘位元（1 和 0），確實讓當時的程式設計師強烈感受到自己是一名偉大的巫師。某種程度來說，工作滿意度上值得這麼做。

上述所列程式中的每一行都包含一項指令，改以文字說明如下：

1. 將數字 0 儲存在記憶體位置 0。
2. 將數字 1 儲存在記憶體位置 1。
3. 將記憶體位置 1 的值儲存在記憶體位置 2。
4. 將記憶體位置 2 的值減掉 11。
5. 如果記憶體位置 2 的值為數字 0，則繼續執行指令 9。
6. 將記憶體位置 0 加上記憶體位置 1 的值。
7. 將記憶體位置 1 的值加上數字 1。
8. 繼續執行指令 3。
9. 輸出記憶體位置 0 的值。

改以文字說明程式內容，雖然已經比大量位元更具有可讀性，但仍然相當艱深難懂。因此，我們改以名稱取代指令中的數字和記憶體位置，幫助我們閱讀程式。

```
 Set "total" to 0.
 Set "count" to 1.
[loop]
 Set "compare" to "count".
 Subtract 11 from "compare".
 If "compare" is zero, continue at [end].
 Add "count" to "total".
 Add 1 to "count".
 Continue at [loop].
[end]
 Output "total".
```

現在你看出來這個程式是怎麼運作的嗎？最前面兩行是為兩個記憶體位置設定初始值：`total` 值用於累計計算的結果，`count` 值則用於追蹤我們目前的數字。使用 `compare` 的那幾行可能是最怪異的。這個程式會檢查 `count` 值是否等於 11，藉此決定是否要停止執行程式。由於我們假設的機器相當原始，所以只能測試數字是否為零，再根據這個結果來做決定。利用標記為 `compare` 的記憶

體位置來計算 count - 11 的值，再根據這個值做決定。只要程式判斷 count 值還不是 11，下兩行就會將 count 值加到累計結果裡，並且將 count 值增加 1。

以下程式和前面做的事一樣，只是改以 JavaScript 撰寫：

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// → 55
```

在這個版本裡，我們做了更多改進。最重要的改變是，我們不需要再指定方式讓程式來回跳來跳去。while 結構幫我們解決了這個問題，只要指定的條件成立，就會繼續執行下方的程式區塊（用大括號括起來的部分）。這裡指定的條件是 count <= 10，意思是「*count* 值小於或等於 10」。此外，我們也不需要再創一個臨時值並且將其與零進行比較，這部分充其量只是一個無趣的細節，程式語言的力量之一就是為我們處理這類枯燥乏味的細節。

執行完 while 結構之後，程式最後會利用 console.log 這項操作寫出結果。

如果恰巧有方便使用的運算函式 range 和 sum，就能分別建立某個範圍內的數字集合，以及計算這個數字集合的總和，最後的程式看起來會像以下這樣：

```
console.log(sum(range(1, 10)));
// → 55
```

這個故事本身蘊含的意義是，同一個程式的表達方式可長、可短，也有好懂、難懂的做法。好比這個程式的第一個版本十分艱澀難懂，最後一個版本則幾乎是以英文寫成：記錄（log）範圍（range）1 到 10 的數字總和（sum），後續章節會介紹如何定義 sum 和 range 這類的運算函式。

優秀的程式語言會幫助程式設計師思考，討論電腦在更高階層必須執行的動作，有助於省略細節，提供方便的程式建構區塊（例如，while 和 console.log），讓程式設計師自己也能定義建構區塊（例如，sum 和 range），並且方便組合這些建構區塊。

JavaScript 的歷史與發展

JavaScript 於 1995 年導入 Netscape Navigator 這個瀏覽器裡，作為在網頁裡加入程式的一種方式，此後，逐漸被其他主流圖形介面網頁瀏覽器採用。JavaScript 是實現現代網頁應用程式的契機，現在你跟網頁應用程式進行互動時，不會每進行一個動作就重新載入網頁一次。在一些比較傳統的網站上，JavaScript 仍舊以各種形式繼續為使用者提供互動性和靈巧性。

有一個重點要請讀者注意，JavaScript 和 Java 這兩個程式語言之間幾乎沒有任何關係。JavaScript 當初會取如此相似的名稱，是出於行銷考量，而非讓大家好判斷。因為 JavaScript 導入之初，Java 已經是非常著名的程式語言，而且十分普遍，於是，有人就想尾隨 Java 的成功，認為趁勢而上是個不錯的主意，因而造成了大家現在對名稱的困擾。

Netscape 瀏覽器採用之後，出現了一份標準文件，說明 JavaScript 語言應該如何運作，隨後有各種領域的軟體宣布支持 JavaScript，實際以相同的程式語言進行討論，制定這項標準的 ECMA 國際組織（Ecma International organization）稱其為 ECMAScript 標準。實務開發上，ECMAScript 和 JavaScript 這兩個標準可以互換，請將它們當作是同一個語言的兩個名稱。

有些人會說 JavaScript 是很糟糕的程式語言，當中有許多說法確實是真的。當年我第一次需要用 JavaScript 撰寫某些程式時，心裡也是很快就看不起它，因為我輸入的任何內容，JavaScript 幾乎都照單全收，只是它闡述的方式跟我想的完全不同。當然，有很大的原因是因為我當時真的不知道自己在做什麼，可是，真正的問題在於：JavaScript 的接受尺度，大到不像話。這種設計背後的想法是，希望初學者在 JavaScript 的環境下能更順利寫出程式，但事實上，因為系統不會幫你指出程式中的問題，所以在多數情況下，要自己找出問題會比較困難。

然而，JavaScript 這種彈性也具有優勢，預留了很多技術空間，在其他更為嚴謹的程式語言中是不可能出現這種情況，後續在第 10 章的範例中，你會看到 JavaScript 如何使用這種特性來克服語言本身的某些缺點。我自己則是在以正確的方式學習 JavaScript 並且使用了一段時間之後，才真正地喜歡上這套程式語言。

JavaScript 從以前發展到現在有好幾種版本。約在 2000 年至 2010 年期間，ECMAScript 3 是大家普遍支持的版本，這段期間 JavaScript 不僅快速佔據主導地位，同時還野心勃勃地發展 ECMAScript 4，打算對這個語言進行徹底改造

和擴展。然而，要對一個大家廣泛使用的活躍語言進行徹底改造，以官方的立場來說還是十分困難，所以 ECMAScript 4 在 2008 年放棄開發，轉而發展要求沒那麼高的 ECMAScript 5，結果最後只做了一些沒有爭議性的修改，並且在 2009 年釋出版本。直到 2015 年發布了 ECMAScript 6 才出現重大更新，其中也包含一些原本打算要在 ECMAScript 4 版本裡推出的想法。此後，JavaScript 每年都會有一個小型更新。

程式語言不斷發展的現實面就是，瀏覽器也必須持續跟上腳步。如果你還在使用舊版的瀏覽器，可能無法支援所有功能。因此，程式設計師在進行任何修改時，要小心不要破壞現有的程式，原則上，新版瀏覽器還是可以執行舊版的程式。本書中的程式是使用 2017 年版的 JavaScript。

網頁瀏覽器不是唯一使用 JavaScript 的平台，某些資料庫也有使用 JavaScript 作為腳本和查詢語言，例如，MongoDB 和 CouchDB。用於桌面應用和伺服器程式設計的幾個平台裡，最著名的是 Node.js 專案（後續第 20 章會介紹的主題），其功能在於提供外部的程式設計環境，讓程式設計師可以在瀏覽器之外的環境裡撰寫 JavaScript 的程式。

程式碼的學習與使用方法

組成程式的文字就是**程式碼**（code），本書裡絕大部分的章節都納入了大量的程式碼。在學習程式設計的過程中，我相信閱讀和撰寫程式碼絕對是不可或缺的一部份。因此，不要只是用眼睛瀏覽本書中的程式範例，請你試著用心閱讀並且加以理解這些程式碼。剛開始進行的速度可能會很慢，而且會有很多看不懂的地方，但我保證，你很快就會抓到訣竅。練習題的部分也是一樣，在你確實能寫出程式，作為可行的解決方案之前，請不要以為自己真的懂了。

我建議各位讀者在 JavaScript 的直譯器中進行實作練習，試著自己找出解決方案。透過這樣的方式，你能立即獲得回饋，馬上就知道自己的做法是否可行。除了本書的練習題，你可以多方嘗試解決不同的問題。

如果你想拿書中的範例程式碼做一些實驗，最簡單的方法是在本書的線上版（*https://eloquentjavascript.net*）尋找你想執行的程式碼。所有程式碼都可以點擊，點擊之後就能對程式碼進行編輯和執行，以及看到程式輸出的結果。如果要寫練習題的程式，請前往本書建置的測試環境（*https://eloquentjavascript.net/code*），每一題都有提供第一行程式碼，還可以在這裡找到所有練習題的解決方案。

如果你想在其他環境（非本書提供的網站環境）執行書中說明的程式，則需要特別謹慎。雖然多數的範例程式都可以在任何 JavaScript 環境中執行，不會受到影響，可是，後面幾章的程式碼通常是針對特定環境（例如，特定瀏覽器或 Node.js）撰寫，所以只能在這個環境下執行。此外，有許多章節裡說明的程式較大，這些程式裡的程式碼片段會相互依賴或者是需要使用外部檔案，因此，在本書網站建置的封閉測試環境中會提供數個 Zip 檔案連結，包含所有需要用到的腳本和資料檔案，作為執行某幾個章節的程式碼之用。

本書內容簡介

本書分為三大部分。前面 12 章的內容是討論 JavaScript 語言，接下來 7 章是介紹網頁瀏覽器以及如何利用 JavaScript 設計出瀏覽器程式，最後兩章是專門介紹另一種 JavaScript 程式的開發環境：Node.js。

書中共提出五個**章節的實作專案**，說明比較大型的程式，讓各位體驗一下真正的程式設計，依序帶領讀者開發宅配機器人、小型的程式語言、2D 平面遊戲、小畫家程式和動態網站。

第一部分的內容是程式語言，會先利用前四個章節的篇幅介紹 JavaScript 語言的基本結構，包含控制結構（例如，你在前言裡看到的關鍵字 `while`）、函式（自己開發與撰寫的程式區塊）和資料結構。學會這些之後，你就能撰寫基本的程式。接下來的第 5、第 6 章則會介紹函式和物件的使用技巧，讓你寫出更多**抽象**程式碼和控制程式的複雜度。

看完第一個實作專案後，程式語言的部分會繼續帶讀者了解錯誤處理、臭蟲修復、規則運算式（處理文字的重要工具）、模組化（另一種防範程式複雜度升高的方法）和非同步程式設計（處理耗時的事件），再以第二個實作專案總結第一部分的內容。

第 13 章到 19 章是第二部分的內容，介紹 Javascript 在瀏覽器上可以使用的工具。你會學到如何將內容顯示在螢幕上（第 14 章和第 17 章）、回應使用者輸入的內容（第 15 章）以及網路通訊（第 18 章）。這個部分會再提供兩個章節的實作專案。

看完前面兩大部分之後，第 20 章會介紹另外一個開發環境——Node.js，第 21 章則是使用這項工具建置一個小型的網站。

最後在第 22 章裡，我們會討論提升 JavaScript 程式的執行速度，也就是 JavaScript 程式最佳化時，應該提出哪些考量。

本書編排慣例

本書內文中的定寬字，代表它是程式裡的某些元素；有些文字片段本身就具有意義，有些則只是引用前後內容裡的部分程式。在前面的內容裡，我們已經看過幾個程式，其編寫方式如下：

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}
```

為了顯示某個程式的輸出結果，有些程式的後方會加上兩個斜線和一個箭頭，再將預期的輸出結果寫在箭頭後面。

```
console.log(factorial(8));
// → 40320
```

祝各位學習順利！

1

資料值、資料型態與運算子

在電腦的世界裡，所有一切都是資料。你可以讀取資料、修改資料、創建新資料，唯有資料才能在這個世界佔有一席之地。所有資料都是以位元的型態存在電腦裡，儲存為一長串的序列，因此，資料基本上都長得很像。

任何具有兩個值的東西都可以用來表示**位元**（*bit*），通常是使用 0 和 1。在電腦裡，位元以各種形式存在，例如，電荷的高低、訊號的強弱或是光碟片表面的亮暗點。所有分散的資訊都能簡化為 0 和 1 的序列，以位元的方式呈現。

舉個例子，假設我們要以位元來表示 13 這個數字。雖然位元的運算方法和十進位數字一樣，可是不同於十進位有十個數字，位元只有**兩個數字**。從最右邊的位元開始依序往左邊的位元移動，每次增加的權重是 2 的倍數。以下這些位元會組成數字 13，每個位元底下顯示的數字是權重：

```
  0   0   0   0   1   1   0   1
128  64  32  16   8   4   2   1
```

所以上述的二進位數字 00001101 裡，數字 1（非 0 的數字）底下的數字有 8、4 和 1，加起來就是 13。

資料值

請想像一下，現在你眼前有一片滿是位元的海洋。當前在一般電腦裡，揮發性資料儲存設備（工作記憶體）的容量就超過 300 億位元；非揮發性儲存設備（硬碟或具有同樣效用的設備）的容量，通常還會比揮發性設備高出數十倍。

為了處理這麼大量的位元還不能流失資料，我們必須將這些位元分成不同的區塊，讓每個區塊表示一段資訊。在 JavaScript 的環境中，這些區塊就稱為**資料值**（value）。雖然所有資料值都是由位元組成，但它們各自扮演不同的角色，這會取決於每個資料值本身的資料型態。有些資料值的型態是數字，有些資料值是一段文字，還有些資料值是函式等等。

在 JavaScript 的世界裡，要創造一個資料值很簡單，你只要呼叫這個值的名字，它就會**咻地一聲出現**，讓你使用，而且還不需要為它收集建造材料或者是為此付出使用費。當然，這些資料值不是真的憑空創造出來的，每個值都必須儲存在記憶體裡的某個位置，而且，萬一你想同時取用巨量的資料值，還可能會一口氣用光電腦所有的記憶體。幸好，只有當你同時需要用到所有資料值，才會出現這個問題。再說，一旦你不再使用某個資料值，它就會消失，然後被回收作為產生下一代資料值的建造材料。

本章將介紹簡單的資料型態和處理資料值的運算子，這些都是組成 JavaScript 程式的原子元素。

數字

毫無懸念，數字型態的資料值就是以數字表示的值。在 JavaScript 的程式裡，寫法如下：

```
13
```

在程式中使用上述的寫法，會將數字 13 轉換成位元型態，然後儲存在電腦的記憶體裡。

JavaScript 用來儲存單一數字資料值的位元數是固定為 64 位元，但 64 位元可以表示的型態其實沒有表面上那麼多，也就是說你能表示不同數字的數量有限。當你使用 N 個十進位數字，你可以表示出 10^N 個數字。同樣地，給定 64 個二進位數字，你可以表示出 2^{64} 個不同的數字，大約是 18 百京（1800 千兆，也就是數字 18 後面再加上 18 個零），這個數字真的很大。

過去的電腦記憶體容量比現在小很多，所以人們常常會使用好幾組 8 位元或 16 位元來表示他們需要的數字。只用這麼少的位元數當然很容易發生**溢位**（overflow）的意外情況，就是程式最後得到的數字無法塞進指定的位元數裡。不過，今日即使是能放進口袋大小的電腦也有大量的記憶體，所以你可以自由地使用 64 位元區塊，只有在你真的需要處理天文數字時才要擔心溢位的問題。

可是，也不是所有小於 18 百京的數字都能塞進 JavaScript 的數字型態裡。64 位元還要用來儲存負數，所以會有一個位元是用來表示數字的正負號，另一個更大的問題是，我們還一定要表示出非整數部分的數字，所以會有某些位元被用於儲存小數點的位置。即使如此，64 位元實際上可以儲存的最大整數範圍還是有 9 百兆（15 個零）之多，這仍舊是一個大家可以開心接受的可觀數字。

小數類型的數字寫法會加上一個小數點。

```
9.81
```

要表示非常大或非常小的數字時，你還可以使用科學記號法，加上 *e* 代表指數，後面跟著表示指數次方的數字。

```
2.998e8
```

上面表示的數字就是 $2.998 \times 10^8 = 299{,}800{,}000$。

只要你計算出來的全部數字（通常也稱為**整數**）是小於前面提到的 9 百兆，就一定能保證計算結果的精確度，但不幸的是，小數計算通常就不是這麼回事。就像我們無法以十進位的有限實數精確表示出 π 的值一樣，當你只有 64 位元可以儲存，一樣會因為捨棄許多數字而失去精確度。雖然有些遺憾，但也只有在特定情況下才會引發真正的問題。重點是要知道有這個情況，然後把小數部份的數字視為近似值，而非精確值。

算術

和數字最息息相關的部分就是算術，例如，加法或乘法這類的算術運算是取兩個數字型態的資料值，加以運算之後產生一個新的數字。JavaScript 裡的算術運算看起來就會像以下這種寫法：

```
100 + 4 * 11
```

在上面的寫法裡，『+』和『*』這兩個符號稱為**運算子**（operator）；前者表示加號，後者表示乘號。將某一個運算子放在兩個數值之間，就能用來運算這些數值，然後產生一個新的值。

但是，上面舉的這個例子是「將 4 和 100 相加之後的結果乘以 11」？還是要在相加之前先做乘法運算？我想你可能已經猜到了，乘法運算會先發生。不過，就跟一般的數學一樣，你可以把加法部分的運算加上括號，就能改變這個情況。

```
(100 + 4) * 11
```

如果要進行減法運算，請使用『-』運算子，除法則是用『/』運算子。

在數個運算子一起出現又沒有括號的情況下，計算的順序會由運算子的**優先序**（precedence）決定。從前面的例子裡可以看到，乘法的優先序高於加法。『/』運算子和『*』運算子的優先序相同；同樣地，『+』和『-』這兩個運算子的優先序也是一樣。當多個優先序相同的運算子一起出現時，例如，`1 - 2 + 1`，運算的順序則是從左到右：`(1 - 2) + 1`。

你不必擔心這些優先序的規則，如果真有疑慮，只要加上括號因應即可解決。

此外，還有一個你不會立刻認出來的符號，其實也是算術運算子，就是用來計算**餘數**的『%』。`X % Y` 的運算結果是 `X` 除以 `Y` 的餘數，例如，`314 % 100` 產生的結果是 `14`，`144 % 12` 則會是 `0`。餘數運算子的優先序和乘法、除法相同，它還有另外一個常見的名稱是 *Mod* **運算子**（modulo）。

特殊數字

JavaScript 把三個特殊值視為數字，不過它們的行為表現和一般數字不同。

前兩個特殊數字是 `Infinity` 和 `-Infinity`，分別代表正無限大和負無限大，而且，`Infinity - 1` 還是 `Infinity`，依此類推。請不要過於信任以 Infinity 為主的計算，從數學上來看，這不是很可靠的計算，所以讓我們立刻來看下一個和這有關的特殊數字：`NaN`。

`NaN`（not a number）的意思是指一個資料值「不為數字」，就算這個值本身是數字型態也一樣不會被視為數字。例如，想要計算 `0 / 0`（零除以零）、`Infinity-Infinity` 或是任何其他無法產生有意義結果的數字運算時，就會得到這樣的結果。

字串

下一個要介紹的基本資料型態是字串（string）。字串是用來表示文字，其寫法是將字串的內容用引號括起來。

```
`Down on the sea`
"Lie on the ocean"
'Float on the ocean'
```

標記字串時，你要使用單引號、雙引號或反引號都可以，只要括住字串開頭和結尾的引號是成對的即可。

幾乎所有放在兩個引號之間的資料值，JavaScript 都會視為字串值，但是，有少數字元處理起來會比較麻煩，請想像一下，要在兩個引號之間放入多個引號會是多麼困難的事。如果不用跳脫字元（escaping），可以利用**換行字元**（按下 ENTER 鍵時就可以獲得）來處理這個情況，但只有在使用反引號（`）時才有效。

因此，為了在字串裡加入這類的字元，就需要使用以下的表示方法：引號裡面的字串中只要有出現反斜線字元（\），就表示後面的字元具有特殊意義，這樣的做法稱為**跳脫**字元。當引號前有反斜線字元，引號就不會被當成字串結尾，而是視為字串的一部分。例如，反斜線後出現字元 n，程式會闡述為換行；同樣地，反斜線後出現字元 t，就表示 tab 字元。請看以下的字串範例：

```
"This is the first line\nAnd this is the second"
```

包含在引號裡的文字，實際顯示如下：

```
This is the first line
And this is the second
```

當然，也會出現這種情況，你希望字串裡的反斜線不是特殊字元，而是單純的反斜線符號時，你只要將兩個反斜線疊在一起，也就是一個反斜線後跟著另一個反斜線，在這個情況下，最後顯示的字串值中就只會剩下一個反斜線。例如，字串「`A newline character is written like "\n".`」在程式裡的表示方法如下：

```
"A newline character is written like \"\\n\"."
```

字串也必須模擬成一連串的字元，才能儲存在電腦裡，JavaScript 的做法是以 *Unicode* 標準（標準萬國碼）為主。這項標準會指定一個數字給每個字母，你會用到的字母幾乎都有納入這項標準，包括希臘語、阿拉伯語、日語、亞美尼亞語等語言的字母。只要每一個字母都有一個代表它們的數字，我們就可以用一串數字來表示字串。

這就是 JavaScript 的做法，但其實很複雜：JavaScript 是以 16 位元來表示每一個字串元素，所以最多可以有 2^{16} 個不同的字元。然而，Unicode 定義的字元超過這個數量，目前大約有兩倍之多，所以 JavaScript 字串裡的某些字元（例如，許多表情符號）就會佔掉兩個「字元位置」。後續在第 5 章的「字串與字元編碼」那一節裡，我們會再回過頭來討論這一點。

除法、乘法或減法不能用在字串上，只有『`+`』運算子**可以**，但不是真的加法運算，而是**連接**的概念——把兩個字串黏在一起，例如，下面這一行寫法會產生字串「`concatenate`」：

```
"con" + "cat" + "e" + "nate"
```

還有很多相關函式（**方法**）可以用來對字串值執行其他操作，後續在本書第 4 章的「方法」一節裡會有更多的介紹。

此外，不論是以單引號或雙引號表示字串，兩者看起來幾乎一樣，唯一的差異在於跳脫字元需要使用其中哪一種類型的引號。如果字串是用反引號括起來，可以玩的花招更多，這種字串通常也稱為**字串樣板**（template literal）。除了能橫跨多行，還可以嵌入其他資料值。

```
`half of 100 is ${100 / 2}`
```

當你在字串樣板 `${}` 中寫入某些內容，程式會將這些內容的計算結果轉換成字串，並且放在字串樣板的這個位置上。以上述寫法為例，顯示的結果為「`half of 100 is 50`」。

一元運算子

然而，並非所有運算子都是符號，也有一些是以文字表示，例如，『`typeof`』運算子。這個運算子的運算結果會以字串值表示給定資料值的型態名稱。

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

在上述範例中，我們使用了 `console.log` 函式，目的是將某些我們想看到的程式評估結果，以文字方式顯示，下一章會進一步介紹這個函式。

之前顯示的其他運算子可以用於運算兩個值，但是 `typeof` 運算時只會取一個值；前者稱為**二元**運算子（binary operator），後者則稱為**一元**運算子（unary operator）。『`-`』（減號）運算子可以同時作為二元和一元運算子使用。

```
console.log(- (10 - 2))
// → -8
```

布林值

在只有兩種可能性的情況下（例如，「是」和「否」或「開」和「關」），如果能用一個值就可以區分，通常會很有幫助。JavaScript 針對這個目的設計了**布林型態**（Boolean），這個型態只有兩個值——就是寫成「true」和「false」。

比較大小

以下是產生布林值的方法之一：

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

『`>`』和『`<`』是傳統使用的符號，分別表示「大於」和「小於」，兩者都是二元運算子。套用在上面的範例中，回傳的布林值會顯示比較結果是否為真（true）。

這個方法同樣可以用於比較字串大小。

```
console.log("Aardvark" < "Zoroaster")
// → true
```

字串的排列大致上會依照字母順序，但不是你以為實際在字典裡看到的樣子：大寫字母一定會比小寫字母「小」，所以「Z」<「a」；此外，非字母的字元，例如，『!』、『-』等等，也包含在這個排序規則裡。JavaScript 比較字串時，會從左邊的字元開始依序往右邊檢查，一個接著一個比較每個字元的 Unicode 碼。

其他類似的運算子還有『>=』（大於或等於）、『<=』（小於或等於）、『==』（等於）和『!=』（不等於）。

```
console.log("Itchy" != "Scratchy")
// → true
console.log("Apple" == "Orange")
// → false
```

JavaScript 中只有一個值不會和自身相等，就是 NaN（not a number，「不為數字」）。

```
console.log(NaN == NaN)
// → false
```

NaN 表示無意義的計算結果，既然如此，就不會等於任何其他無意義的計算結果。

邏輯運算子

還有某些運算本身可以應用布林值。JavaScript 支援三種邏輯運算子：『和』、『或』以及『非』，都能用於「推論」布林值。

『&&』運算子表示邏輯『和』，屬於二元運算子，只有當兩個給定值均為真（true），其運算結果才會為真（true）。

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

『||』運算子表示邏輯『或』，只要給定值中的其中一個為真（true），其運算結果即為真（true）。

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

邏輯『非』的寫法是驚嘆號『!』，屬於一元運算子，作用是將給定值反轉——!true 會回傳 false，!false 則會回傳 true。

布林運算子、算數運算子和其他運算子混在一起使用時，需要使用括號的時機不見得很明顯。從我們目前看到的運算子中通常可以了解到，實務上優先序最低的運算子是『||』，再來是『&&』，然後是比較運算子（>、== 等等），最後是其他剩餘的運算子。以下的範例就是一個典型的表達式，會選擇這樣的順序是因為需要盡可能減少括號的數量：

```
1 + 1 == 2 && 10 * 10 > 50
```

最後一個要討論的運算子，不是一元，也不是二元，而是**三元**（ternary），可以運算三個值，其寫法是由一個問號和一個冒號組成，如以下範例所示：

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

這種運算子稱為**條件**運算子，或有時只稱為**三元**運算子，因為在 JavaScript 這個語言裡只有它是屬於這類的運算子。問號左邊的值會「挑選」出其他兩個值之中要出現哪一個。當左邊的值為 true，會選擇中間的值；為 false 時，則會選擇右邊的值。

空值

還有兩個特殊值，分別寫成 null（空值）和 undefined（未定義），用於表示**無意義**的值。這兩個雖然本身是值，卻不帶有任何資訊。

程式語言中有許多運算無法產生有意義的值（後續會看到一些例子），但還是必須產生某個值，所以在這種情況下，就會簡單地產生 undefined。

null 和 undefined 兩者之間在意義上的差異，其實是 JavaScript 設計上的意外，這一點在多數情況下並不重要。如果你真的很糾結這一點，我會建議你乾脆將這兩者視為可以互換的值。

自動轉換型態

我在前言裡提過，JavaScript 是一個不嫌麻煩的程式語言，不管你給什麼程式，JavaScript 幾乎都可以接受，即使是要它做奇怪的事，一樣照單全收。以下這些表達式就是非常好的範例：

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

當運算子在運算過程中發現資料型態「錯誤」，JavaScript 會使用一組規則，悄悄地將資料值轉換為它需要的型態；而且，這些通常不是你想要的或是預期的規則，因此稱為**強制型態轉換**（type coercion）。在上述範例中，第一個表達式的 null 變成 0，第二個表達式將字串 "5" 轉換成數字 5；然而，在第三個表達式裡，在將數字相加之前，『+』會先嘗試連接字串，所以數字 1 會被轉換成字串 "1"。

當某個型態的值無法直接對應一個數字時（例如，"five" 或是 undefined），就會強制將其轉換為一個特殊數字值 NaN，即使繼續對 NaN 進行算數運算，產生的結果還是 NaN。所以，如果你意外發生上述範例中的某一種情況，請檢查看看是否意外遇到被強制轉換型態的情況。

使用『==』運算子比較相同型態的值，很容易預測比較結果：當兩個值相同時，除了 NaN 的情況，結果應該是 true；可是，當兩者的型態不同時，JavaScript 就會使用一套複雜而且令人困惑的規則來決定處理方式，在多數情況下，JavaScript 會先試著將其中一個值轉換為另一個值的型態。然而，當運算子兩邊的其中一側出現 null 或 undefined 時，只有兩邊均為 null 或 undefined，結果才會是 true。

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

當你要測試某個值是否為真正的值，而非 null 或 undefined 時，這種行為通常很有用，你可以利用『==』或『!=』運算子，把某個值和 null 進行比較。

但是，如果你想測試某個值是否確實指向 false，該怎麼做呢？例如，表達式 0 == false 和 "" == false 均為 true。當你不希望發生任何自動轉換型態的情況時，還有兩個運算子可以用：=== 和 !==。前者是測試某個值是否確實和另一個值相等，後者是測試某個值是否確實和另一個值不相等。因此，如我們所預期，"" === false 的結果為 false。

保險起見，我建議最好使用有三個字元的比較運算子，避免發生型態被強制轉換的意外情況。不過，要是你確定比較運算子兩側的型態均相同，使用字元較少的運算子也不會發生問題。

短路邏輯運算子

邏輯運算子『&&』和『||』是以比較奇特的方式來處理不同型態的值。這兩個運算子為了決定處理方法，會將左側的值轉成布林值型態，至於要回傳原本左手邊的值還是右手邊的值，則取決於運算子和轉換的結果。

以『||』運算子為例，當左側的值轉成 true，就回傳左側的值，否則就回傳右側的值。當值為布林值而且對其他型態的值進行類似操作，就具有預期效果。

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

我們可以利用這個功能性，將預設值作為退路。萬一有某個值可能變成空值，你可以將它的替代值放在『||』後面。所以，如果左邊的初始值變成 false，就會回傳右邊的替代值。將字串和數字轉換成布林值時，其規則是 0、NaN 和空字串（""）均為 false，其他剩下的值均為 true。所以，0 || -1 產生的結果是 -1、"" || "!?" 會產生 "!?"。

『&&』運算子的做法一樣，只是反過來。當運算子左側的值轉成 false，就回傳左側的值，否則就回傳右側的值。

這兩個運算子還有另一個重要的屬性就是，只有在必要時才評估其右側的值。在 `true || X` 的情況裡，由於結果為 true，所以不論 `X` 為何（即使 `X` 是一段糟糕的程式），永遠都不會評估 `X` 的值。同樣地，在 `false && X` 的情況裡，由於結果是 false，所以會忽略 `X` 的值。這種特性就稱為**短路評估**（short-circuit evaluation）。

條件運算子的作用也是一樣，只有在第一個值被選取的情況下，才會評估第二和第三個值。

本章重點回顧

本章檢視了 JavaScript 中四種型態的資料值：數字、字串、布林值和 undefined（未定義）。

輸入名稱（`true`、`null`）或值（`13`、`"abc"`），就可以建立這些型態的值，還可以利用運算子將資料值結合在一起以及轉換資料值。我們也看了二元運算子，用於算術（`+`、`-`、`*`、`/`、`%`）、字串連接（`+`）、比較（`==`、`!=`、`===`、`!==`、`<`、`>`、`<=`、`>=`）和邏輯（`&&`、`||`）；幾個一元運算子（負號 `-`、否定邏輯 `!`、用於找出數值型態的 `typeof`）和一個三元運算子（`?:`）（給定三個值作為運算基礎，從兩個值中挑選出一個結果）。

本章提供的資訊還不夠多，只夠你將 JavaScript 作為口袋裡的計算機使用。在下一章的內容裡，我們會開始將這些表達式放在一起，組出一個基本程式。