前言

我相信你已經注意到了，儘管責罵這個語言的怪異之處或許是我們 JavaScript 程式設計師自我認同的一部分，不過本系列書名中的「JS」並不是某些用來咒罵 JavaScript 的字眼之縮寫。

從 Web 草創初期，JavaScript 就已經是為我們所閱讀的內容帶來互動體驗的一項基礎技術。雖然 JavaScript 最初所帶來的，可能是閃爍的滑鼠軌跡和惱人的彈出式提示，但在將近二十幾年之後，JavaScript 相關的技術與能力已經成長了好幾倍，很少人會懷疑它在 Web 這個世界上最廣泛被採用的軟體平台上的關鍵重要性。

然而，作為一個程式語言，它一直都是許多批評的對象，有部分是因為它得遵循的某些傳統機制，但更多的是出於它的設計哲學。即便是它的名稱，如同 Brendan Eich 有次所說的，也讓它變得像是它更成熟的大哥 Java 的「愚笨小弟」（dumb kid brother）。但這個名稱實際上僅是商業政策與行銷所造成的一個意外。這兩個程式語言在許多重要的方面都相當的不同。「JavaScript」與「Java」的關係，就如同「Carnival」（狂歡節）與「Car」（車子）的關係一樣。

因為 JavaScript 從數個程式語言借取了幾個概念與語法慣例，包括了驕傲的 C 程序式（procedural）根源，還有比較不那麼明顯的 Scheme/Lisp 函式型（functional）根源，這讓為數眾多的開發人員都可以非常輕易地接近它，即便對沒有什麼程式設計經驗的人來說，也是如此。JavaScript 的「Hello World」是如此的簡單，讓這個語言容易親近，只要接觸一小段時間，就能上手。

雖然 JavaScript 或許是最輕易能夠上手，並且馬上開始使用的程式語言，但它的一些古怪之處，使得它比許多其他的程式語言更難以確實地精通。對於 C 或 C++ 這類的程式語言來說，要寫出功能完備的程式，需要擁有該語言相當深入的知識才行。然而，要寫出能夠立即上線的 JavaScript 程式，有可能而且通常只需用到該語言相當表淺的知識，只發揮了該語言所能做到的一小部分能力而已。

深植在該語言中複雜精深的概念，常常會以**看似**簡單的方式出現，例如把函式當作 callbacks（回呼函式）來到處傳遞，這讓 JavaScript 開發人員只需要依循該語言的慣例來使用它，而不必太過擔心底層發生了什麼事。

它同時是個容易使用，能夠吸引許多人的簡單語言，也是集合了精密語言機制的複雜語言，如果沒有仔細研究，就連最資深的 JavaScript 開發人員也不容易**真正地理解**它。

這就是 JavaScript 的矛盾之處，是該語言最大的弱點，也是我們目前正在處理的挑戰。因為 JavaScript 是一個**可以**不用了解就開始使用它的程式語言，也很少有人真正地理解它。

任務

如果每次在 JavaScript 中遭遇到了意外或挫折時，你的反應都是將那些功能加入黑名單（就像某些人慣於做的那樣），你很快就會發現自己退居至捨棄了 JavaScript 豐富內涵的一個空殼中。

雖然這個功能子集曾被賦予「優良部分（The Good Parts）」這個著名的稱呼，不過親愛的讀者們，我懇求你們，應該將它們視為「簡單的部分（The Easy Parts）」、「安全的部分（The Safe Parts）」，或甚至是「不完整的部分（The Incomplete Parts）」。

《**你所不知道的** *JS*（*You Don't Know JS*）》系列處理的，則是完全相反的挑戰：學習並深入理解 JavaScript 的**所有**部分，甚至是而且特別是「困難的部分（The Tough Parts）」。

這裡，我們要解決的問題是 JS 開發人員「只想要學習足夠應付手上工作的部分就好，而從不強迫自己確實地去了解該語言如何以及為何展現其行為」的這種傾向。此外，我們不採納「如果路變得難走就撤退」這種常見的建議。

我不滿足，你也不應該滿足，不要停在「可以運作就好，但並非真正知道**為何**可行」的狀態。我溫和地向你發出挑戰，希望你踏向那條崎嶇的「少有人走的路（road less traveled）」，擁抱 JavaScript 的原貌，以及它能做到的事。只有擁有這種知識，就沒有你無法理解的技巧、框架，或是本週熱門流行詞的首字母縮寫。

本系列中的每一本書都針對該語言常被誤解或理解不足的核心部分，深入詳盡地解說它們。讀完之後，你會對你的理解深具信心，而且不只是理論上的，是理解實務上「你必須要知道」的那些細節。

你現在所擁有的 JavaScript 知識或許有部分源自於那些自己也理解不完整的人。**那部分**的 JavaScript 只不過是這個語言真正實體的一個影子。你**尚未**真正了解 JavaScript，但只要你深入研讀這個系列，你就會得到真正的理解。繼續往下讀吧，我的朋友，JavaScript 正等著你呢！

複習

JavaScript 很棒。要學會它的部分功能，是很容易的事情，但要完全掌握它（甚至只是掌握到**充分**的程度），就難上許多了。每當開發者遭遇到令人困惑之處，他們通常會責怪這個語言，而非自己缺乏理解。本系列的書籍就是要彌補這個遺憾，希望能夠鼓舞你好好去欣賞這個你能夠了解，也**應該**深入去了解的程式語言。

本書中的許多範例都假設所用的環境是現代（並緊緊跟隨未來發展）的 JavaScript 引擎，例如 ES6。如果在較舊的（ES6 以前的）引擎上執行，某些程式碼可能無法如所描述的那樣運作。

本書所用的慣例

本書使用下列格式體裁：

斜體字（*Italic*）

代表新出現的術語、URL、電子郵件地址、檔案名稱，以及延伸檔名。中文以楷體字表示。

定寬字（`Constant width`）

用於程式碼列表，或是在字裡行間參考到程式組成元素的地方，例如變數或函式名稱、資料庫、資料型別、環境變數、述句與關鍵字。

定寬粗體字（**`Constant width bold`**）

用於應該逐字由讀者鍵入的命令或文字。

序

有人曾經說過：「JavaScript 是唯一一個開發人員使用它之前不用先學習它的程式語言」。

我每次聽到這段話時，都會大笑，因為對我來說，這是真的，而我懷疑對其他許多開發人員來說，也是如此。JavaScript，或許還有 CSS 及 HTML，都不是 Internet 草創初期會在大學課程中教授的電腦科學核心語言，所以個人的學習發展過程，幾乎都得靠這些開發新手自行搜尋，並且運用「檢視原始碼（view source）」的功能來拼湊出這些基本 web 語言的樣貌。

我仍然記得我高中第一個網站專題。我們的任務是要建立出某種網路商店，而我因為是個龐德（James Bond）迷，我決定建立一個電影 007：黃金眼（Goldeneye）的商店。它應有盡有：背景播放的是黃金眼的 MIDI 主題曲，還有由 JavaScript 所控制的十字準星會跟著滑鼠游標在螢幕上跑來跑去，而且每次點擊都會播放槍聲。Q 應該會以這個網站傑作為榮。

我之所以會說這個故事，是因為我當時所做的事情，就跟現在許多開發人員做的一樣：我將一塊又一塊的 JavaScript 程式碼複製貼上到我的專案中，但完全不知道實際上到底發生了什麼事。像是 jQuery 這樣廣泛受到採用的 JavaScript 工具集，以它們自己的方式，延續了這種「不學習核心 JavaScript」模式的生命。

我並不是在詆毀這些 JavaScript 工具集，畢竟我自己也是 MooTools JavaScript 團隊的一員！但這些 JavaScript 工具集之所以會這麼強大，原因就是它們的開發人員了解基礎的知識，以及應該特別注意的地方，還有如何有效運用。儘管這些工具集的功能強大，但了解這個語言的基本知識，仍然是非常重要的，而且有了像是 Kyle Simpson 的**你所不知道的 JS**（*You Don't Know JS*）系列書籍的幫助，你沒有藉口不去學習它們。

型別與文法（*Types & Grammar*）是這系列的第三本書，它以非常清晰的方式，帶你一觀那些複製貼上動作和 JavaScript 工具集沒辦法教你，也可能永遠不會教你的核心 JavaScript 基礎知識。強制轉型（coercion）及其陷阱、作為建構器（constructors）的 natives，以及全部的 JavaScript 基本知識都以專門的程式碼範例詳盡地解說。就像此系列的其他書籍一樣，Kyle 直指要點，不囉嗦不贅言，完全就是我愛的那種技術書籍。

好好享受**型別與文法**的閱讀過程吧，別把它放在離書桌太遠的地方！

—*David Walsh*（*http://davidwalsh.name*），
Mozilla 的資深 *Web* 開發人員

[4]

型別

大多數的開發人員都會說動態語言（dynamic language，像是 JS）沒有型別（*types*）。讓我們看看 ES5.1 規格（*http://www.ecma-international.org/ecma-262/5.1/*）對這點怎麼說：

> 本規格中的演算法所操作的每個值（values）都有一個關聯的型別。可能的值型別（value types）正是定義於此條款中的那些。型別會再進一步被分為 ECMAScript 語言型別（language types）和規格型別（specification types）。
>
> 一個 ECMAScript 語言型別對應到由 ECMAScript 程式設計師使用 ECMAScript 語言所直接操作的值。ECMAScript 語言型別有 Undefined、Null、Boolean、String、Number 與 Object。

聽到這裡，如果你是強型別（strongly typed，或稱「靜態型別」，statically typed）語言的擁護者，你可能會抗議這裡不該使用「型別（type）」這個詞。在那些語言中，「型別」一詞所代表的意義，比 JS 這裡所用的還要多很多。

某些人表示 JS 不應該宣稱擁有「型別」，認為它們應該被稱為「標記（tags）」或者是「子型別（subtypes）」才對。

呸！我們會在此使用這種粗略的定義（與規格的寫法大略相同）：一個**型別**（*type*）是一組固有的、內建的特徵，它們唯一地識別了特定值的行為，讓它與其他值有所分別，對於引擎（engine）及開發人員來說皆是如此。

換句話說，如果引擎及開發人員兩者都以不同的方式對待值 `42`（數字）與值 `"42"`（字串），那麼就說這兩個值有不同的**型別**，分別是 `number` 與 `string`。

你使用 42 的時候，你是**想要**做一些數值處理，像是算術運算。不過當你使用 "42" 的時候，你就是**想要**進行某些字串操作，例如輸出到頁面上等。這兩個值具有不同的型別。

這絕非一個完美的定義。但對於我們的討論而言，已經夠用了，而這也與 JS 描述自身的方式一致。

就算「型別」的名稱不叫做「型別」

除了學術定義上的爭議，JavaScript 到底有沒有**型別**（*types*）有何重要性可言呢？

要知道如何恰當且精確地將值轉換為不同的型別（參閱第 7 章），對於每個**型別**及其固有的行為有正確的理解是絕對不可或缺的。幾乎每個被寫出來的 JS 程式都會需要以某種方式去處理值的強制轉型（value coercion），所以能夠可靠且有自信地這樣做，是很重要的。

如果你有 42 這個 number 值，但你想要將它視為一個 string 來對待，例如取出在位置 1 上的 "2" 這個字元，你顯然就必須先將這個值從 number（強制）轉為 string。

這看起來似乎很簡單。

不過這種強制轉型可能發生的方式有許多種。有些是明確的，很容易推理，而且可靠。但如果你不小心留意，強制轉型就可能會以非常怪異且出乎意料的方式發生。

強制轉型所造成的混淆可能是 JavaScript 開發人員會遇到的最深刻的挫折之一。它時常被批評為過度**危險**，被視為是該語言設計上的一個缺陷，應該避之唯恐不及。

具備對 JavaScript 型別的完整理解，我們要說明為何強制轉型的**壞名聲**主要是出於過度炒作，而且有點冤枉，藉此逆轉你的觀點，讓你看得見強制轉型的強大之處及實用性。不過首先我們得對值和型別有更進一步的了解。

內建型別

JavaScript 定義了七個內建型別（built-in types）：

- null
- undefined
- boolean
- number
- string
- object
- symbol（ES6 中所新增的！）

除了 object 之外，所有的這些型別都叫做「primitives」（基本型別值，或簡稱「基型值」）。

typeof 運算子會檢視給定值之型別，並以字串形式回傳七個值之一，令人意外的是，這與我們剛列出的七個內建型別並不會一對一地完全吻合：

```
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";    // true
typeof { life: 42 }  === "object";    // true

// ES6 中所新增的！
typeof Symbol()      === "symbol";    // true
```

這裡列出的六個型別都有對應型別的值，並會回傳相同名稱的一個字串值，如上所示。Symbol 是 ES6 的一個新的資料型別（data type），我們將會在第 3 章中涵蓋它。

你可能已經注意到，上面列表中我排除了 null。它是**特別的**，特別之處在於，它與 typeof 運算子搭配使用時，很容易產生臭蟲：

```
typeof null === "object"; // true
```

如果它所回傳的是 "null" 就好了（那才是正確的）！不過這個 JS 中原生的臭蟲已經存在將近二十年了，而且很有可能永遠都不會被修補，因為有非常多的 web 內容仰賴它這種錯誤的行為，因此「修補」這個臭蟲只會產生更多的「臭蟲」，使得很多的 web 軟體無法運行。

如果你想要藉由其型別來測試一個 null 值，你會需要一個複合條件：

```
var a = null;

(!a && typeof a === "object"); // true
```

null 是唯一「falsy」（即假值的，參閱第 7 章）的基型（primitive）值，而 typeof 檢查對它則會回傳 "object"。

那麼 typeof 會回傳的第七個字串值是什麼呢？

```
typeof function a(){ /* .. */ } === "function"; // true
```

你很容易會認為這個 function 是 JS 中頂層的內建型別，尤其是在看到了 typeof 運算子的這個行為之後。然而，如果你閱讀規格，你會發現它實際上是 object 的一個「子型別（subtype）」。更確切的說，一個函式（function）被稱作是一個「可呼叫的物件（callable object）」，一個擁有 [[Call]] 內部特性，讓它能夠被調用（invoke）的物件。

「函式實際上是物件」的這個事實相當有用。更重要的是，它們能夠擁有特性（properties）。例如：

```
function a(b,c) {
    /* .. */
}
```

函式物件具有一個 length 特性，它被設為它所宣告的形式參數（formal parameters）之數目：

```
a.length; // 2
```

既然你以兩個具名的形式參數（b 與 c）宣告了這個函式，「該函式的 length（長度）」就會是 2。

那陣列（arrays）呢？它們是 JS 原生（native）的東西，所以它們是一種特殊的型別嗎？

```
typeof [1,2,3] === "object"; // true
```

不，它們只是物件。最適當的方式是也把它們想成是 object（參閱第 6 章）的一個「子型別」，但是具有「能以數值化的方式來索引」（相對於普通物件只能以字串作為鍵值）及「維護了一個會自動更新的 .length 特性」這些額外的特徵。

作為型別的值

在 JavaScript 中，變數（variables）沒有型別，**值**（*values*）**才有型別**。變數在不同時間點上可以持有任何的值。

思考 JS 型別的另一個方式是：JS 沒有「強制施加的型別（type enforcement）」，JS 引擎不會堅持一個**變數**必須永遠持有與其**初始型別相同**的值。一個變數可能會在某個指定述句（assignment statement）後持有一個 string，而在下一個述句後持有一個 number，依此類推。

42 這個**值**（*value*），具有一個固有的型別 number，而其**型別**（*type*）無法改變。我們可以透過一個叫做**強制轉型**（*coercion*，參見第 7 章）的過程，從 number 值 42 建立出型別為 string 的另一個值 "42"。

當你看到 typeof 被用在一個變數上，它不是在問說「該變數的型別為何？」，雖然表面上看起來像那樣，但 JS 的變數是沒有型別的。它所問的問題實際上是「在該變數**中**的那個值是什麼型別？」

```
var a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"
```

typeof 運算子回傳的一定是個字串。所以：

```
typeof typeof 42; // "string"
```

第一個 typeof 42 回傳 "number"，而 typeof "number" 的結果則是 "string"。

未定義（undefined）vs.「未宣告（undeclared）」

目前沒有值的變數實際上擁有 undefined 這個值。對這種變數呼叫 typeof 會回傳 "undefined"：

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// 之後
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

多數開發人員可能會傾向於把「undefined（未定義）」這個詞想成是「undeclared（未宣告）」的同義詞。然而在 JS 中，這兩個概念相當不同。

一個「undefined」變數，是已經在可取用的範疇（accessible scope）中被宣告了的變數，只不過在當下其中沒有其他的值。相較之下，一個「undeclared」變數，則是尚未在可取用的範疇中正式被宣告的變數。

請考慮：

```
var a;

a; // undefined
b; // ReferenceError: b is not defined
```

一個惱人的困惑之處是瀏覽器指定給這種情況的錯誤訊息。如你所見，這個訊息是「b is not defined」，這當然非常容易跟「b is undefined」搞混，這種混淆並非沒道理。但是我們要再次提醒，「undefined」與「is not defined」是非常不同的兩回事。如果瀏覽器能夠使用像「b is not found（找不到 b）」或「b is not declared（b 沒有宣告）」這樣的說法，那就好了，可以減少許多混淆！

typeof 與未宣告的變數（undeclared variables）搭配使用時，還會有另一個特別的行為，更進一步加深了這個混淆。

請考慮：

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

縱然是對「undeclared」（或「not defined」）的變數，typeof 運算子也會回傳 "undefined"。注意到執行 typeof b 時，並沒有錯誤被擲出，即便這裡的 b 是一個未宣告的變數。這是 typeof 行為中的一種特殊的安全防護。

類似於前面所述，如果與未宣告的變數（undeclared variable）搭配使用的 typeof 能夠回傳「undeclared」，而非把結果值與其他不同的「undefined」情況混為一談，那就太好了。

typeof Undeclared

儘管如此，在瀏覽器中處理 JavaScript 時，這個安全防護是一項實用的特色，因為在那種環境下，可能會有多個指令稿（script）檔案載入變數到共用的全域命名空間中（shared global namespace）。

許多開發人員相信，這個全域命名空間中永遠都不應該有任何的變數，而所有的東西都應該被包在模組或私有 / 個別的命名空間中。這在理論上當然很好，但實務上幾乎是不可能的，雖然它仍然是我們應致力達成一個好目標！幸好，ES6 為模組（modules）新增了一級的（first-class）支援，這最終會讓此目標變得更為實際。

作為一個簡單的例子，想像一下你的程式中有一種「除錯模式（debug mode）」，它是由一個叫做 DEBUG 的全域變數（旗標）所控制的。進行像是將訊息記錄到主控台這種除錯工作前，你會想檢查看看該變數是否已宣告。一個頂層的 var DEBUG = true 全域宣告只會被包含在一個「debug.js」檔案中，而你只會在開發與測試的過程中，將它載入到瀏覽器中，實際上線時不會載入。

然而，你必須在你其餘的程式碼中，特別留心你是如何檢查這個 DEBUG 全域變數的，如此才不會有 ReferenceError 被擲出。typeof 的這層安全防護，在這種情況中，能助我們一臂之力：

```
// 糟糕，這會擲出一個錯誤！
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// 這是存在與否的一個安全檢查
if (typeof DEBUG !== "undefined") {
```

```
    console.log( "Debugging is starting" );
}
```

即使你不是在處理使用者定義的變數（像是 DEBUG），這種檢查也有所用處。如果你是在檢查內建 API 的某個功能是否存在，你可能也會發現這種檢查有所幫助，防止擲出一個錯誤：

```
if (typeof atob === "undefined") {
    atob = function() { /*..*/ };
}
```

如果你是在定義某個功能不存在時，要代替它的一個「polyfill」，你大概會想要避免使用 var 來進行 atob 的宣告。如果你在 if 述句內宣告 var atob，這個宣告會被拉升（hoisted，參閱本系列的*範疇與 Closures* 一書）到該範疇的頂端，即使那個 if 條件沒有通過（因為全域的 atob 已經存在！）也是一樣。在某些瀏覽器中，或是對某些型別的全域內建變數（通常叫做「host 物件」）而言，這種重複的宣告可能會擲出一個錯誤。省略那個 var 會防止這種拉升的宣告。

對全域變數進行這種檢查，但不使用 typeof 安全防護功能的另一個方式是觀察到「所有的全域變數都是全域物件（global object）的特性」這個事實，而在瀏覽器中，這個全域物件基本上就是 window 物件。所以，上面的檢查可以（相當安全地）改為這樣做：

```
if (window.DEBUG) {
    // ..
}

if (!window.atob) {
    // ..
}
```

不同於參考未宣告的變數，你試著存取一個不存在的物件特性（即使是在 window 全域物件上的）時，不會有 ReferenceError 被擲出。

另一方面，藉由一個 window 參考（reference）手動地參考全域變數，是某些開發人員會選擇避免的事情，特別是在你的程式碼需要在多個 JS 環境中執行的情況下（例如不只在瀏覽器中，還有在伺服端的 node.js 中），因為屆時全域物件就不一定叫做 window 了。

嚴格來說，即使你沒有使用全域變數，typeof 上的這個安全防護也有用處，雖然這些情況較為少見，而有些開發人員可能會發現這種設計途徑不是他們所要的。想像有一個你希望其他人複製貼上到他們程式或模組中的工具函式，在其中你想要檢查看看包含該函式的程式是否有定義某個特定的變數（如此你才能使用它）：

```
function doSomethingCool() {
    var helper =
        (typeof FeatureXYZ !== "undefined") ?
        FeatureXYZ :
        function() { /*.. 預設功能 ..*/ };

    var val = helper();
    // ..
}
```

doSomethingCool() 會測試一個叫做 FeatureXYZ 的變數，如果有找到，就用它；如果沒有，就使用它自己的版本。現在，如果有人將這個工具函式引入他們的模組或程式，它會安全地檢查他們是否定義有 FeatureXYZ：

```
// 一個 IIFE（參閱本系列中《範疇與 Closures》一書對
// 「Immediately Invoked Function Expressions」的討論）
(function(){
    function FeatureXYZ() { /*.. my XYZ feature ..*/ }

    // 引入 `doSomethingCool(..)`
    function doSomethingCool() {
        var helper =
            (typeof FeatureXYZ !== "undefined") ?
            FeatureXYZ :
            function() { /*.. 預設功能 ..*/ };

        var val = helper();
        // ..
    }

    doSomethingCool();
})();
```

在此，FeatureXYZ 不是一個全域變數，但我們仍然使用 typeof 的安全防護來讓檢查變得安全。重要的是，這裡我們**沒有**物件可以用來進行檢查（就像我們透過 window.___ 檢查全域變數那樣），所以 typeof 相當有幫助。

其他的開發人員可能會偏好一種叫做「依存性注入（dependency injection）」的設計模式，在這種模式中，doSomethingCool() 不會隱含地查驗 FeatureXYZ 是否有定義，而是要求這個依存性明確地被傳入，像這樣：

```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*.. 預設功能 ..*/ };

    var val = helper();
    // ..
}
```

設計這樣的功能時有很多的選擇可用。在此沒有任何一個模式是「正確的」或「錯誤的」，每種做法都有需要取捨的地方。但整體而言，有 typeof undeclared 這樣的安全防護提供我們更多選擇，是很不錯的。

複習

JavaScript 有七個內建型別（*types*）：null、undefined、boolean、number、string、object，以及 symbol。我們可用 typeof 運算子來識別它們。

變數（variables）沒有型別，但在它們其中的值（values）會有型別。這些型別定義那些值固有的行為。

許多開發人員會認為「undefined」（未定義）與「undeclared」（未宣告）大略是相同的事情，但在 JavaScript 中，它們相當的不同。undefined 是一個已宣告的變數能夠持有的值。「Undeclared」則代表一個變數從未被宣告。

很遺憾地，JavaScript 將這兩個詞混為一談了，不只是在它的錯誤訊息中（「ReferenceError: a is not defined」），也在 typeof 的回傳值中，對於這兩者皆回傳 "undefined"。

然而，在某些情況中，typeof 上的安全防護（避免錯誤）用於未宣告的變數（undeclared variable）時，是有用處的。