[1]

何謂範疇？

幾乎所有的程式語言（programming languages）都有一個最根本的典範，也就是在變數（variables）中儲存值（values），並在之後取回或修改那些值的能力。事實上，儲存值並從變數取出值的能力就是賦予一個程式**狀態**（*state*）的東西。

如果沒有這種概念，程式還是可以執行某些任務，但它們將會非常受限而且不怎麼有趣。

但引進了變數之後，就會產生我們現在要處理的有趣問題：那些變數**存在**於何處呢？（where do those variables *live*?）。換句話說，它們被儲存在哪裡呢？還有最重要的，我們的程式如何在需要時找到它們呢？

這些問題意味著我們必須以一組定義良好的規則來將變數儲存在某些位置，以便在之後找回那些變數。我們會將這組規則（set of rules）稱作是：**範疇**（*scope*）。

不過這些**範疇**規則是在何處以何種方式設定的呢？

編譯器理論

這或許不言而喻，又或許令人驚訝，得要視你與各種語言互動的程度而定，不過儘管 JavaScript 一般被歸類為「動態的（dynamic）」或「直譯式（interpreted）」語言，但它實際上是一種編譯式語言（compiled language）。跟許多傳統編譯式語言不同，它**並非**早在事先就編譯好，編譯出來的結果，也無法在各種分散式系統之間移植。

但儘管如此，JavaScript 引擎（engine）也會進行許多與傳統語言編譯器相同的步驟，雖然是以比我們普遍認為的還要複雜精密得多的方式進行。

碁峰 www.gotop.com.tw

在傳統的編譯式語言處理過程中，你程式原始碼（source code）中的某一塊，在執行之前，通常會經歷三個步驟，大略稱為「編譯」：

Tokenizing（語法基本單元化）與 *Lexing*（語彙分析）

將一串字元（a string of characters）拆解成（對該語言來說）有意義的組塊（meaningful chunks），這些組塊就叫做語法基本單元（tokens，或稱「語彙單元」）。舉例來說，考慮 `var a = 2;` 這個程式。這個程式很有可能被拆解為下列這些語法基本單元：`var`、`a`、`=`、`2` 及 `;`。空白（whitespace）可能會也可能不會被視為語法基本單元，要視它是否有意義而定。

tokenizing（語法基本單元化）與 lexing（語彙分析）之間的差異很細微，而且比較學術，但它的要點在於，那些語法基本單元（tokens）是以無狀態（*stateless*）的方式，或是以有狀態（*stateful*）的方式來識別的。簡單地說，如果 tokenizer（語法基本單元產生器）得調用有狀態的剖析（parsing，或稱「語法分析」）規則才能斷定 a 是一個獨立的語法單元，或只是其他語法單元的一部分，那就會是 *lexing*。

Parsing（剖析，或「語法分析」）

接受由語法基本單元（tokens）所構成的串流（stream，或「陣列」，array），並將之轉為一種元素內嵌的樹狀結構（a tree of nested elements），集體代表了程式的文法結構（grammatical structure）。這種樹狀結構就稱為「AST」（*a*bstract *s*yntax *t*ree，抽象語法樹）。

代表 `var a = 2;` 的樹狀結構可能會以一個叫做 `VariableDeclaration` 的頂層節點（top-level node）為始，並帶有一個叫做 `Identifier` 的子節點（child node，其值為 `a`），而另外一個子節點則叫做 `AssignmentExpression`，它本身帶有一個叫做 `NumericLiteral` 的子節點（其值為 `2`）。

Code-Generation（產生目的程式碼）

接受一個 AST 並將之轉為可執行程式碼（executable code）的過程。這部分會隨著語言、其目標平台等因素的不同而大幅變化。

所以，與其深陷細節之中，我們只會簡單帶過，就說存在一種方式可以把前面描述過的 `var a = 2;` 的 AST 轉為一組機器指令（a set of

碁峰 www.gotop.com.tw

machine instructions），以實際建立出一個叫做 a 的變數（包括了保留記憶體等步驟），然後將一個值儲存到 a 中。

引擎管理系統資源（system resources）的方式，其細節比我們會探討的還要深入，所以我們只會理所當然地假設引擎能在必要時建立並儲存變數。

JavaScript 引擎所進行的工作遠比那三個單純的步驟複雜得多，就跟其他大多數的語言編譯器一樣。舉例來說，在剖析與程式碼產生（code-generation）的過程中，當然會有最佳化（optimize）執行效能的步驟，包括了消除多餘元素等。

所以，這裡我只是很粗略地帶過。不過我想稍後你就會了解為何我們有涵蓋的這些細節，即使從較高階的觀點來看，都是很重要的。

一方面 JavaScript 引擎並沒有很多時間上的餘裕來進行最佳化（跟其他語言編譯器不同），因為 JavaScript 的編譯並不是在建置（build）步驟中事先處理的，不像其他的語言。

對 JavaScript 來說，在許多情況下，編譯的過程發生在程式碼被執行之前僅以微秒（microseconds）來計算（或更少）的時間中。要確保最高的效能，JS 引擎使用了各種技巧（像是會進行 lazy compile 或甚至 hot recompile 的 JIT 等），這些技巧遠遠超出了我們這裡討論的「範疇（scope）」。

為了簡單起見，讓我們單純假設，任何的 JavaScript 程式碼片段都必須在它被執行之前（通常是剛好在執行之前！）編譯完成。所以，JS 編譯器會接受 `var a = 2;` 這個程式，並且先編譯它，然後準備好執行它，通常是即刻進行。

了解範疇

我們用來學習範疇（scope）的方式是把整個過程想成是一種對話（conversation）。不過是誰在進行對話呢？

碁峰 www.gotop.com.tw

卡司

讓我們見見會彼此互動以處理程式 `var a = 2;` 的角色陣容吧，如此我們才能理解稍後我們會聽到的它們的對話內容：

Engine（引擎）

負責從開始到結束的編譯程序，並執行我們的 JavaScript 程式。

Compiler（編譯器）

Engine 的朋友之一，處理剖析（parsing）與程式碼產生（code-generation）的所有苦工（請參閱前一節）。

Scope（範疇）

Engine 的另外一位朋友，負責收集並維護由所有已宣告的識別字（declared identifiers，即「變數」，variables）所構成的一個查找清單（look-up list），並強制施加一組嚴格的規則，來規範這些變數對於目前正在執行的程式碼而言，是否可以取用。

為了要**完整地理解** JavaScript 的運作方式，你得先學著從 Engine（及其朋友們）的角度來**思考**，詢問它們會問的問題，並如同它們般回答那些問題。

前後往返（Back and Forth）

當你看到 `var a = 2;` 這個程式時，你最有可能把它想成是一個述句（statement）。但我們的新朋友 Engine 並不是如此看待它的。事實上，Engine 看到了兩個不同的述句：Compiler 會在編譯過程中處理的一個述句，以及 Engine 會在執行過程中處理的另一個述句。

所以，讓我們來分析 Engine 及其朋友會如何處理 `var a = 2;` 這個程式。

Compiler 會對這個程式做的第一件事情，是進行 lexing（語彙分析）動作，將它拆解成語法基本單元（tokens），然後它會再將之剖析為一個樹狀結構。但等到 Compiler 到達程式碼產生（code generation）的階段時，它對待此程式的方式會與我們可能假設的有些不同。

一個合理的假設可能會是，Compiler 會產生可以由這段虛擬程式碼總結的程式碼：「Allocate memory for a variable, label it `a`, then stick the value

碁峰 www.gotop.com.tw

2 into that variable.（為一個變數配置記憶體，將它標示為 a，然後將 2 這個值放到那個變數中。）」遺憾的是，那並不是很準確。

Compiler 會進行的處理如下：

1. 遇到 var a 時，Compiler 會詢問 Scope，看看 a 這個變數是否已經存在於那個特定的範疇集合（scope collection）中。若是如此，Compiler 就會忽略這個宣告（declaration），繼續前進。否則的話，Compiler 會要求 Scope 為那個範疇集合宣告一個叫做 a 的新變數。
2. Compiler 接著會產生之後要讓 Engine 去執行的程式碼，以處理 a = 2 指定式（assignment）。Engine 執行的程式碼會先詢問 Scope 目前的範疇集合中是否有一個叫做 a 的變數可以取用。如果有，Engine 就會使用那個變數。如果沒有，Engine 就會往他處找（參閱後面章節的「巢狀範疇」）。

如果 Engine 最後有找到一個變數，就會將 2 這個值指定給它。如果沒有，Engine 就會舉起手來，大叫出錯了！

總結就是，一個變數指定（variable assignment）觸發了兩個不同的動作：首先，Compiler 在目前的 Scope 中宣告了一個變數（如果之前沒有宣告的話），第二，執行時 Engine 會在 Scope 中查找該變數，然後如果有找到的話，就指定值給它。

Compiler 說話了

我們需要稍微多一點的編譯器術語才能了解接下來的內容。

當 Engine 執行 Compiler 為第 2 步驟所產生的程式碼，它得查找（look up）變數 a 來看看它是否已被宣告，而這個查找動作就是去請教 Scope。不過 Engine 所進行的查找動作種類（type of look-up）會影響到最後的結果。

在我們的例子中，我們說 Engine 會為變數 a 進行一種 LHS 查找動作。另外一種的查找動作叫做 RHS。

我打賭你猜得到「L」與「R」所代表的意義。這些詞代表的是 lefthand side（左手邊）和 righthand side（右手邊）。

左右邊，什麼東西的？**一個指定作業**（*an assignment operation*）的。

碁峰 www.gotop.com.tw

換句話說，當一個變數出現在一個指定作業的左手邊，進行的就是 LHS 查找動作，而 RHS 查找動作會在一個變數出現在一個指定作業的右手邊時進行。

說實在的，讓我們更精確一點。就我們的目的而言，RHS 查找動作與單純查找某個變數的值沒什麼兩樣，而 LHS 查找是試著要找出那個變數容器（variable container）本身，如此才能指定東西給它。在這種思維下，RHS 並不是**真的**代表字面上的「一個指定的右手邊」，更準確地說，它所代表的只是「不是左手邊」。

稍微放寬標準一下，你可以把 RHS 的意思想成是「retrieve his/her source (value)」（取回他或她的來源（值））」，這暗示著 RHS 就代表「取得…的值（go get the value of ...）」。

讓我們深入一點。

當我說：

```
console.log( a );
```

對 a 的參考（reference）就是一個 RHS 參考，因為這裡沒有指定任何東西給 a，而是要取回 a 的值，讓那個值可以被傳入給 console.log(..)。

相較之下：

```
a = 2;
```

在此 a 的參考則是一個 LHS 參考，因為我們實際上並不在意目前的值是什麼，我們只是想要找到那個變數，作為 = 2 這個指定作業的一個目標。

代表「left/righthand side of an assignment（一個指定的左右手邊）」的 LHS 與 RHS 並不一定代表「left/right side of the = assignment operator（= 指定運算子的左右手邊）」。指定（signments）也可能以其他幾種方式發生，所以在概念上把它們想成是「誰是指定的目標（LHS）？」和「誰是指定的來源（RHS）？」會比較好。

請考慮這個程式，LHS 及 RHS 參考在其中都有出現：

```
function foo(a) {
    console.log( a ); // 2
```


碁峰 www.gotop.com.tw

```
}

foo( 2 );
```

將 foo(..) 作為一個函式呼叫調用的最後一行程式碼需要對 foo 的一個 RHS 參考，這意味著「去查找 foo 的值，並把它交給我」。此外，(..) 代表 foo 的值應該被執行，所以它實質上最好是一個函式！

這裡有一個難以察覺但重要的指定發生。

你可能會忽略這段程式碼中隱含的 a = 2。它在 2 這個值作為一個引數（argument）被傳入 foo(..) 函式時發生，在那種情況下，2 這個值會被**指定**（*assigned*）給參數 a。要（隱含地）指定值給參數 a，就會進行 LHS 查找動作。

a 的值還有另外的一個 RHS 參考，而那所產生的值會被傳入給 console.log(..)。console.log(..) 需要一個參考以執行。它是 console 物件的一次 RHS 查找動作，然後就會發生一個特性解析（property resolution）動作，看看它是否具有一個叫做 log 的方法。

最後，我們可以形成這樣的概念：把 2 這個值（藉由變數 a 的 RHS 查找動作）傳入給 log(..) 時會有一次 LHS/RHS 的交換。在 log(..) 的原生實作（native implementation）內部，我們可以假設它會有參數，而其中第一個（或許叫做 arg1）在指定 2 給它之前，會有一次 LHS 參考查找動作。

你可能很容易會產生這樣的概念：把函式宣告 function foo(a) {… 想成是一個正常的變數宣告，加上一個指定，例如 var foo 及 foo = function(a){…。若是如此，你也很容易會認為這個函式宣告涉及了一次 LHS 查找動作。

然而，細微但重要的差異在於，Compiler 會在程式碼產生階段處理這種宣告及值的定義，如此一來，Engine 執行程式碼的過程中，就不需要去處理將一個函式值「指定」給 foo 的工作。因此，以我們在此討論的那種方式，將一個函式宣告視為會進行 LSH 查找的指定，並不是真的很恰當。

碁峰 www.gotop.com.tw

Engine 與 Scope 的對話

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

讓我們將前面的交換（處理這段程式碼的）想像成一種對話。這段對話會有點類似這樣：

Engine：嘿 Scope，我有 foo 的一個 RHS 參考。有聽過它嗎？

Scope：喔！是的，我有。Compiler 不久前剛宣告了它。它是個函式。來，拿去吧。

Engine：太好了，謝囉！很好，我正在執行 foo。

Engine：嘿 Scope，我拿到 a 的一個 LHS 參考，有聽過它嗎？

Scope：喔！是的，我有。Compiler 最近才把它宣告為 foo 的一個形式參數（formal parameter）。拿去吧。

Engine：一如以往樂於助人啊 Scope。再次感謝。現在該是時候把 2 指定給 a 了。

Engine：嘿 Scope，抱歉再次打擾。我需要進行 console 的 RHS 查找。有聽過它嗎？

Scope：別在意，Engine，我整天做的工作就是這個啊。有的，我找到 console 了。它是內建的。拿去吧。

Engine：太好了。查找 log(..)。好，很好，它是一個函式。

Engine：唷！ Scope。你能幫我處理一個對 a 的 RHS 參考嗎？我想我記得它，只是想要再確認一下。

Scope：Engine 你是對的，它是同一個變數，沒變過。拿去吧。

Engine：酷！將 a 的值，也就是 2 傳入給 log(..)。

…

小測驗

我們來檢查一下你到目前為止的理解程度如何。確保你有扮演過 Engine 的部分，並與 Scope 有過「對話」：

碁峰 www.gotop.com.tw

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 識別出所有的 LHS 查找（有 3 個！）。
2. 識別出所有的 RHS 查找（有 4 個！）。

小測驗的答案請參閱本章的複習！

巢狀範疇

我們說過，Scope（範疇）是藉由其識別字名稱（identifier name）來查找變數的一組規則。然而，通常要考慮的範疇都不只一個。

就像一個區塊（block）或函式可以內嵌（nested）在其他區塊或函式內，範疇也能夠嵌入到其他範疇內。所以，如果在最接近的範疇中找不到一個變數，Engine 就會向下一個外層的包含範疇（outer containing scope）諮詢，如此持續下去，直到找到了，或是到達最外層（outermost，即「全域」）範疇為止。

請考慮下列這段程式碼：

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

`b` 的 RHS 參考無法在函式 `foo` 之中解析（resolved）完成，但它可以在包圍它的範疇（在此即為全域範疇）中解析出來。

所以，再次回到 Engine 與 Scope 之間的對話，我們就會聽到：

> Engine：嘿，`foo` 的 Scope，有聽過 `b` 嗎？我需要它的一個 RHS 參考。
>
> Scope：沒有，從沒聽過它。再問問吧。

www.gotop.com.tw

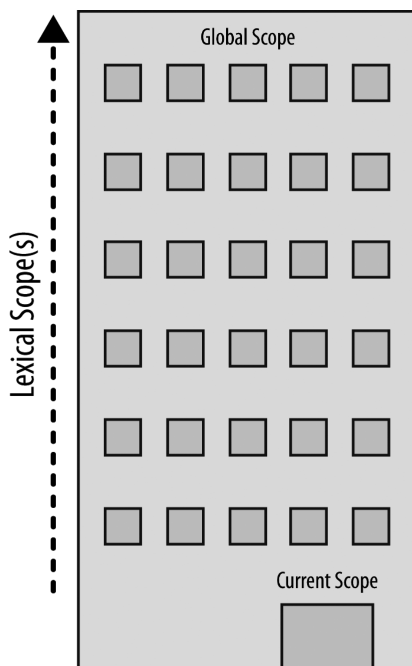
Engine：嘿，foo 外面的 Scope，喔，原來是全域範疇（global scope）你啊，那也很好，有聽過 b 嗎？我需要它的一個 RHS 參考。

Scope：當然，我這邊有。拿去吧。

巡訪（traversing）巢狀範疇（nested scope）的簡單規則：Engine 會從目前正在執行的範疇（currently executing scope）開始，在那裡尋找目標變數，如果沒找到，就會持續往上一層找，依此類推。如果到達最外層的全域範疇，搜尋動作就會停止，不管是否有找到那個變數。

以隱喻為基礎

為了視覺化巢狀範疇的變數解析過程，我希望你想像這個高聳的建築物：



這個建築物代表了我們程式的巢狀範疇規則組合。建築物的第一樓代表你目前正在執行的範疇，不管你在哪裡都是一樣。建築物的頂層就是全域範疇（global scope）。

碁峰 www.gotop.com.tw

你會查看你目前的樓層以解析 LHS 與 RHS 參考，如果你沒有找到，就搭電梯到上一樓，在那裡找找看，然後是再上一層，依此類推。一旦你到達頂樓（全域範疇），你要不是找到了想找的東西，就是沒找到，但無論如何你都得停止了。

錯誤

為什麼區分 LHS 和 RHS 是很重要的事情？

因為在目標變數尚未被宣告（沒有在任何諮詢過的範疇中找到）的情形之下，這兩種類型的查找在行為上會有所不同。

請考慮：

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

當 RHS 查找動作為了 b 而初次發生時，我們找不到它。它被稱為一個「未宣告（undeclared）」的變數，因為範疇中找不到它。

如果 RHS 查找動作無法在巢狀的那些 scopes 中找到一個變數，結果就是引擎會擲出一個 ReferenceError。很重要的一點是要注意這種錯誤的種類是 ReferenceError。

相較之下，如果引擎進行的是 LHS 查找動作，而它抵達了頂樓（全域範疇）卻沒有找到目標變數，若程式不是在「Strict Mode（嚴格模式）」[1] 中執行，那麼全域範疇就會在全域範疇中創建以那個名稱命名的新變數，然後將它交回給 Engine。

「沒有，之前並沒有那個變數，但我能幫上忙，為你建立了一個。」

在 ES5 中新增的「嚴格模式（Strict Mode）」跟一般／寬鬆／懶惰模式比起來，有幾個行為上的不同之處。其中之一就是它不允許這種自動／隱含的全域變數創建動作。在那種情況下，不會有可以從 LHS 查找動作

1　參閱 MDN 對 Strict Mode（嚴格模式）的分析（*http://mzl.la/1epvZNQ*）

碁峰 www.gotop.com.tw

交回的全域變數，而 Engine 會擲出一個 ReferenceError，就類似 RHS 的情況那樣。

現在，如果某次的 RHS 查找動作有找到一個變數，但是你試圖對它的值做的事情是不可能成功的，例如試著將一個非函式值當成一個函式來執行，或是在 null 或 undefined 值之上參考一個特性，那麼 Engine 就會擲出不同種類的錯誤，叫做 TypeError。

ReferenceError 與範疇解析（scope resolution）動作的失敗有關，而 TypeError 則暗示範疇的解析動作成功了，但你試圖對結果進行非法或不可能的動作。

複習

Scope（範疇）是一組規則，用來決定一個變數（識別字）要在何處查找以及如何查找。這種查找動作的目的可能是為了指定值給變數，也就是 LHS（lefthand-side，左手邊）參考，又或者是為了取回其值，即 RHS（righthand-side）參考。

LHS 參考源自於指定作業（assignment operations）。範疇相關的指定可能藉由 = 運算子或是在傳入引數（指定）給函式參數時發生。

JavaScript 引擎會在執行前先編譯程式碼，而過程中它會把像是 var a = 2; 這樣的述句拆成兩個分開的步驟：

1. 首先，var a 在那個範疇中宣告它。這是在一開始時進行，在程式碼執行之前。
2. 接著，a = 2 會去查找那個變數（LHS 參考），並在找到時指定值給它。

LHS 及 RHS 參考的查找都會從目前正在執行的範疇（currently executing scope）開始，而如果需要（也就是在那裡找不到它們要找的），它們就會隨著巢狀範疇（nested scope）往上層找，一次搜尋一個範疇（一層樓），找尋目標識別字（identifier），直到它們到達全域範疇（頂樓）為止，然後就會停下來，無論有沒有找到。

解析失敗的 RHS 參考會使得 ReferenceError 被擲出。解析失敗的 LHS 參考會產生帶有目標名稱的一個自動隱含建立的全域變數（如果不是在 Strict Mode 中執行的話），或是產生 ReferenceError（在 Strict Mode 中）。

碁峯 www.gotop.com.tw

小測驗的解答

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 識別出所有的 LHS 查找（有 3 個！）。

   c = ..;、*a = 2*（隱含的參數指定）及 *b = ..*

2. 識別出所有的 RHS 查找（有 4 個！）。

   foo(2..、*= a;*、*a ..* 及 *.. b*

碁峰 www.gotop.com.tw