

<https://blog.techbridge.cc/2018/12/08/javascript-closure/>



○ 所有的函式都是閉包：談 JS 中的作用域與 Closure



[script](#) [#closure](#)

Posted by huli on 2018-12-08

讚 8,531 分享

在正文開始前先幫自己小小工商一下，前陣子把自己以前寫過的文章都放到了 [GitHub](#) 上面，那邊比較方便整理文章以及回應，如果有想討論的可以在[這篇文章的 GitHub 版本](#)下面留言，想收到新文章通知的也可以按個 watch，感謝。

前言

請先原諒我用了一個比較聳動的標題，因為實在是想不到還有什麼標題好下，最後選擇了一個可能比較有爭議的標題，但能利用這樣的標題激起討論也是滿有趣的，何況我說這話也是有根據的。

在觀看此篇文章之前請先看過上一篇：[我知道你懂 hoisting，可是你了解到多深？](#)，因為文章內容有部分相關，所以必須先有 Execution Context 以及 Variable Object 的觀念以後，才能夠吸收這篇文章的東西。

如果你只對文章標題的那句：「所有的函式都是閉包」有興趣，那可以直接往下拉，因為要講閉包就必須先從作用域開始講起，所以這篇文章按照慣例不會太短，前面也會有一定程度的鋪陳。

好，讓我們從作用域開始吧。

作用域 (Scope)

什麼是作用域（或也有人翻做：範疇）？

我自己最喜歡的解釋是：「作用域就是一個變數的生存範圍，一旦出了這個範圍，就無法存取到這個變數」。

來看一個簡單的例子：

```
function test(){
  var a = 10
}
console.log(a) // Uncaught ReferenceError: a is not defined
```

- 0 6 以前，唯一產生作用域的方法就是 function，每一個 function 都有自己的作用域，
作用域外面你就存取不到這個 function 內部所定義的變數。然而 ES6 的時候引入了 let 跟
const，多了 block 的作用域，但那不是本文的重點所在，所以我就先這樣帶過了。

除了這種 function 的作用域以外，還有一種叫做作用域是 global 的，其實就是我們常在說的「全域」，或者是「全域變數」，任何地方都能夠存取到，如下範例：

```
var I_am_global = 123
function test() {
  console.log(I_am_global) // 123
}
test()
```

從上面的範例中你可以發現一件有趣的事情，那就是你在 function 裡面可以存取外面的變數，可是你從外面卻進不去 function 裡面，這邊我要引用之前看到的一個很有趣的解釋方法，這篇文章把作用域比喻成明星，把函式比喻成地區。

全域變數就是國際巨星，例如說湯姆克魯斯，無論到哪裡大家都認識這個人，因為實在是太紅了。而 function 裡面的變數就像是你那個很會唱歌的鄰居，整個社區都知道它的存在，但一但出了這個社區（超過了這個 function），就沒人認識他是誰。

所以 function 一層層的結構就像是地區那樣，最外層是地球、再來五大洲、亞洲、臺灣、台北市、大安區、大安森林公園，在大安森林公園運動的人知道那邊常在慢跑的朋友，也知道台北市內的名人，可是居住在台北市的人卻不一定知道大安區的區長是誰，因為那超出了它的範圍。

把上面的說法轉成程式碼就會變成這樣：

```
function taiwan() {
  var taiwan_star = 'taiwan_star'
  function taipei() {
    function daan() {
      var daan_star = 'daan_star'
      console.log(taiwan_star) // taiwan_star
    }
    daan()
    console.log(daan_star) // Uncaught ReferenceError: daan_star is not
                           defined
  }
  taipei()
}
taiwan()
```

所以你現在對作用域這個詞應該比較能夠理解了，就是一個變數的生存範圍，一但超過那個範圍就存取不到，而這個範圍就是 function 本身及其內部，所以你在 function 裡面宣告了一個變數，function 外是沒辦法存取的。

外面存取不到裡面的，但「內層」可以存取到「外層」的東西：

```
0  function test() {
1    var a = 100
2    function inner() {
3      console.log(a) // 100
4    }
5    inner()
6  }
7  test()
```

對於 `inner` 這個 function 來說，`a` 並不是它自己的變數，而這種不在自己作用域中，也不是被當成參數傳進來的變數，就可以稱作 free variable，可以翻做自由變數（聽起來滿酷的）。

對 `inner` 來說，`a` 就是一個自由變數。

那 `a` 的值會是什麼？

因為在 `inner` 這個作用域裡面找不到 `a`，就會去上一層 `test` 的作用域裡面尋找，如果還是找不到，就會再往上一層直到找到為止，所以你可以發現這樣會構成一個「作用域鏈」，`inner function scope -> test function scope -> global scope`，不斷在這條鏈往上找，如果最後還是找不到就拋出錯誤。

講到這邊基本的概念應該都有了，再來我要出一個問題把你的認知重新打亂並混淆你：

```
var a = 100
function echo() {
  console.log(a) // 100 or 200?
}

function test() {
  var a = 200
  echo()
}

test()
```

請問最後 `log` 出來的 `a` 應該會是 100 還是 200？

我知道！是 100，因為全域變數裡面的 `a` 是 100...等等，可是我在 `test` 裡面的時候又宣告了一個叫 `a` 的變數並設為 200，`echo` 裡面的這個 `a` 好像也可能是 200...好混亂。

答案是 100，你只要掌握我們之前說的那些原則就行了，`echo` 裡面的那個 `a` 就是 global 的那個 `a`，跟 `test` 裡面的 `a` 一點關係都沒有。

但你會被混淆也是非常合理的事情，因為在某些程式語言裡面，`a` 的確會是 200 喔！`a` 最後出來的值（或換句話說，如何決定自由變數的值）跟程式語言如何決定「作用域」這件事情有關係。

其實我們一開始介紹的這種方法，叫做靜態作用域（static scope），為什麼叫做靜態？就作用域跟這個 function 在哪裡被「呼叫」一點關係都沒有，你用肉眼看程式碼的結構可以看出來它的作用域是什麼，而且是不會變的。

0

來說，上面那個範例印出來的 `a` 就會是 global 的 `a`，儘管我在 `test` 裡面宣告了另外一個呼叫 `echo` 這個 function，但這跟作用域一點關係都沒有，靜態作用域是在 `function` 被「宣告」的時候就決定了，而不是 `function` 被「執行」的時候。

相對地，有靜態作用域就有動態作用域（dynamic scope），如果這個程式語言是採用動態作用域，那最後 `log` 出來的值就會是 200 而不是 100；換句話說，`echo` 這個 function 裡面的 `a` 的值是在程式執行時期才被動態決定的，你只看程式碼的結構沒辦法決定 `a` 到底是什麼值。

而 JavaScript 的作用域是採用前者，靜態作用域，所以你分析程式碼的結構就可以知道作用域的長相。這邊順帶一提的是 JavaScript 裡面最難解的問題之一：`this`，其實原理跟動態作用域有異曲同工之妙，那就是 `this` 的值也是程式執行時才被動態決定的，這也是為什麼一大堆人會搞不清楚它的值是什麼，因為會變來變去的。

靜態作用域其實更學術的名詞叫做 lexical scope，我有看過有人翻成語彙範疇，有人則是翻成詞法作用域。

要理解什麼是 lexical，你必須先知道一點 compiler 的運作原理。在編譯的時候有幾個步驟是用程式去 parse 你的程式碼並且解析，而其中一個步驟就叫做 Lexical Analysis（詞法分析或語彙分析），其實就是去正確分析出程式碼裡面的每一個詞。

我舉一個例子好了，例如說 `a = 13 + 2` 這一個句子，經過詞法分析之後可能就會變成：`a`、`=`、`13`、`+`、`2`，這樣子的分組，就先理解到這邊就好，想知道更多編譯器的細節請自行參考相關書籍或文章，或等我有一天把這個基礎補足之後再來用白話文跟大家分享。

所以會叫做 lexical scope 的原因就是在編譯的時候其實就能決定作用域是什麼，才有了這樣子的一個名稱。

跟作用域相關的內容就到這裡了，幫大家複習幾個關鍵字：

- 作用域鏈 scope chain 往外找
- 自由變數 free variable global
- 靜態作用域 static scope (lexical scope) 不是執行時決定，而是宣告時決定
- 動態作用域 dynamic scope this, 實行時決定

閉包（Closure）

再來終於要進入到閉包的相關內容了，在這之前我先介紹一下大家印象中的閉包大概是什麼樣子，然後又具備了什麼樣的特性。

請看以下範例程式碼：

```
function test() {
  var a = 10
  function inner() {
    console.log(a) // 10
  }
  inner()
}
```

0

很特別的，就只是執行一個內部的 function 而已。但如果我們現在不要直接執行 inner，而是把這個 function 回傳呢？

```
function test() {
  var a = 10
  function inner() {
    console.log(a) // 還是 10
  }
  return inner
}

var inner = test()  test已經執行結束了
inner()
```

神奇的事情發生了，那就是程式碼依舊輸出了 10。

神奇在哪裡？神奇在一個 function 執行完成以後本來會把所有相關的資源釋放掉，可是我test 已經執行結束了，照理來說變數 a 的記憶體空間也被釋放，但我呼叫 inner 的時候居然還存取得到 a！

換句話說，a 這個變數被「關在」 inner 這個 function 裡面了，所以只要 inner 還存在的一天，a 就永無安寧，只能一直被關在裡面。

而事情的主因就是我在 function 裡面回傳了一個 function，才能造成這種明明執行完畢卻還有東西被關住的現象，而這種情形就是一般人所熟知的閉包，Closure。

那閉包的好處有什麼？優點之一就是能把變數隱藏在裡面讓外部存取不到，舉例來說我有個紀錄餘額的變數跟一個扣款的 function，但我有設置了一個上限，那就是最高只能扣 10 塊：

```
var my_balance = 999
function deduct(n) {
  my_balance -= (n > 10 ? 10 : n) // 超過 10 塊只扣 10 塊
}

deduct(13) // 只被扣 10 塊
my_balance // 還是被扣了 999 塊
任何人：其他 developer
```

儘管我們寫了 deduct 這個 function 來操作，但變數還是暴露在外部，任何人都可以直接來改這個變數。這時如果我們利用閉包來改寫，世界就不一樣了：

```
function getWallet() {
  var my_balance = 999
  return {
    deduct: function(n) {
      my_balance -= (n > 10 ? 10 : n) // 超過 10 塊只扣 10 塊
    }
}
```



0 · wallet = getWallet()
 └── deduct(13) // 只被扣 10 塊
 └── balance == 999 // *Uncaught ReferenceError: my_balance is not defined*

因為我把餘額這個變數給藏在 function 裡面，所外部是存取不到的，你想要修改只能夠利用我暴露出去的 deduct 這個函式，這樣子就達到了隱藏資訊的目的，確保這個變數不會隨意地被改到。

但比起這個閉包的用法，我相信有很多人應該都是從底下這個慘痛的經驗才知道有閉包這個東西：

```
var btn = document.querySelectorAll('button')
for(var i=0; i<=4; i++) {
  btn[i].addEventListener('click', function() {
    alert(i)
  })
}
```

假設頁面上有五個按鈕，我想要第一個按下去時彈出 0，第二個按下去時彈出 1，以此類推，於是寫了上面的程式碼，看起來十分合理。

誰知道我一點下去按鈕，靠腰勒為什麼每一個按鈕都彈出 5，都彈出一樣的數字就夠詭異了，5 到底是從哪來的啊？

包括我自己也是有類似的經驗才意識到自己對作用域以及閉包不太熟悉，現在有了經驗之後再回頭來看上面這段程式碼就能夠完全理解了。

首先，上面的迴圈你以為是這樣子：

```
btn[0].addEventListener('click', function() {
  alert(0)
})

btn[1].addEventListener('click', function() {
  alert(1)
})

...
```

但其實是這樣子：

```
不是function
btn[0].addEventListener('click', function() {
  alert(i)
})
  ↙ reference
只有定義，未執行

btn[1].addEventListener('click', function() {
  alert(i)
})
```



0



想想你會發現下面比較合理，我本來就是幫它加一個 function 是按下去的時候會跳出 i 而已，我又沒有直接執行這個 function。

所以當使用者按按鈕的時候，畫面就會跳出 i，那這個 i 的值會是什麼？因為你按按鈕的時候迴圈已經跑完了，所以 i 早已變成 5（迴圈的最後一圈，i 加一變成 5，判斷不符合 $i \leq 4$ 這個條件所以跳出迴圈），畫面也就跳出數字 5 了。

我加上的這幾個 function，本身都沒有 i 這個變數，所以往作用域的外層去尋找，就找到上面迴圈的那個變數 i 了，因此這幾個 function 所指涉到的 i 是同一個 i。

那應該怎麼解決這個問題呢？加上 function！

```
定義
function getAlert(num) {
  return function() {
    alert(num)
  }
}
for(var i=0; i<=4; i++) {
  btn[i].addEventListener('click', getAlert(i))
}
```

value
執行

這邊要注意的是 `getAlert(i)` 會「回傳」一個跳出 i 的 function，因此我額外產生了五個新的 function，每一個 function 裡面都有自己該跳出的值。

或是你要耍帥的話就這樣寫：

```
for(var i=0; i<=4; i++) {
  (function(num) {
    btn[i].addEventListener('click', function() {
      alert(num)
    })
  })(i)
}
```

定義
reference
reference
定義
執行

利用 IIFE (Immediately Invoked Function Expression) 把一個 function 包起來並傳入 i 立即執行，所以迴圈每跑一圈其實就會立刻呼叫一個新的 function，因此就產生了新的作用域。

以上如果你都覺得太麻煩不想用，恭喜，在 ES6 裡面有了 block scope 以後，你只要簡單地把迴圈裡面用的 var 改成 let 就行了：

```
for(let i=0; i<=4; i++) {
  btn[i].addEventListener('click', function() {
    alert(i)
  })
}
```

- 愛 **let** 的特性，所以其實迴圈每跑一圈都會產生一個新的作用域，因此 `alert` 出來的值就是你想要的那個值。如果你還是覺得有點疑惑，你可以把迴圈看成這樣：

```
// 塊級作用域
let i=0 reference
btn[i].addEventListener('click', function() {
  alert(i)
})
// 塊級作用域
let i=1 reference
btn[i].addEventListener('click', function() {
  alert(i)
})
...
...
```

說到這邊我們對閉包有了初步的理解，但對於「什麼是閉包」這個問題似乎還沒有一個明確的定義，「閉包就是可以把值關在裡面的 function」聽起來怪怪的，如果你去找維基百科，他會跟你說：

在電腦科學中，閉包（英語：*Closure*），又稱詞法閉包（*Lexical Closure*）或函式閉包（*function closures*），是參照了自由變數的函式。這個被參照的自由變數將和這個函式一同存在，即使已經離開了創造它的環境也不例外。所以，有另一種說法認為閉包是由函式和與其相關的參照環境組合而成的實體。

如果去找英文的維基百科，可以看到它寫著：

Operationally, a closure is a record storing a function together with an environment

好，看起來還是有點霧煞煞，但總之對於閉包的定義先在此打住，大家心中有個模糊的概念就好，我們晚點再回來處理。

在這個段落我們知道了閉包可以實際應用在哪裡，也理解了閉包可以存取到應該被釋放的值（但卻因為閉包的存在無法被釋放），再來我們來看看 ECMAScript 是如何講述作用域的。

ECMAScript 中的作用域

在開始之前，如果你忘記我們之前講的運作模型，請回去我知道你懂 hoisting，可是你了解到多深？複習一下，因為我們等等會用到。

在這邊我一樣用篇幅較少的 ES3 來當範例，要注意的是 ES6 以後很多名詞變得不一樣了，但原理大致上是相通的。

- ♥ 我們在 [10.1.3 Variable Instantiation](#) 的章節看到了 hoisting 相關的東西，這次我
- 看的則是下一個段落而已，也就是 [10.1.4 Scope Chain and Identifier](#)
- [.ution](#)。

Every execution context has associated with it a scope chain. A scope chain is a list of objects that are searched when evaluating an Identifier. When control enters an execution context, a scope chain is created and populated with an initial set of objects, depending on the type of code.

每個 EC 都有自己的 scope chain，當進入 EC 的時候 scope chain 會被建立。

接著我們來看 [10.2 Entering An Execution Context](#) 底下的 [10.2.3 Function Code](#)：

The scope chain is initialised to contain the activation object followed by the objects in the scope chain stored in the [[Scope]] property of the Function object.

這一段描述了 scope chain 的內容到底是什麼，它講說當進入 EC 的時候，scope chain 會被初始化為 activation object 並加上 function 的 [\[\[Scope\]\]](#) 這個屬性。

以上段落其實要講的事情只有一個，就是在進入 EC 的時候會做下面這件事：

```
scope chain = activation object + [[Scope]]
```

接著要處理的是兩個問題：什麼是 activation object（以下簡稱 AO），什麼又是 [\[\[Scope\]\]](#)？

在 [10.1.6 Activation Object](#) 可以找到 AO 的解釋：

When control enters an execution context for function code, an object called the activation object is created and associated with the execution context.

- ↪ activation object is initialised with a property with name arguments and attributes { DontDelete }
- 0 ↪ activation object is then used as the variable object for the purposes of variable instantiation.

[reference](#)

這邊提到 **When control enters an execution context for function code**，意思就是只有在進入「函式」的時候會產生這個 AO，而之後 AO 便被當作 VO 拿去使用。

所以什麼是 AO？你可以把它直接當作 VO 的另外一種特別的型態，只在 function 的 EC 中出現，所以在 global 的時候我們有 VO，在 function 內的時候我們有 AO，但是做的事情都是一樣的，那就是會把一些相關的資訊放在裡面。

差別在哪裡？差別在於 AO 裡面會有一個 **arguments**，畢竟是給 function 用的嘛，一定要存這個，其餘地方都是差不多的。如果你偷懶把 VO 跟 AO 這兩個詞混在一起使用，我覺得也是可以接受的，因為差別真的太細微了。

[variable object](#)

[active object](#)

解決了 AO 的問題之後，那什麼是 **[[Scope]]**？在 **13.2 Creating Function Objects** 的部分可以看到更詳細的解釋：

Given an optional parameter list specified by FormalParameterList, a body specified by FunctionBody, and a scope chain specified by Scope, a Function object is constructed as follows

(中間省略)

7. Set the [[Scope]] property of F to a new scope chain (10.1.4) that contains the same objects as Scope.

就是說你在建立 function 的時候會給一個 Scope，而這一個 Scope 會被設定到 **[[Scope]]** 去。

那在建立 function 時給的 Scope 是什麼？還能有什麼，當然就是當前 EC 的 Scope。

這樣一段段看完之後其實我們可以整理出這樣的一個流程：

1. 當 function A 建立時，設置 **A. [[Scope]] = scope chain of current EC**
2. 當進入一個 function A 時，產生一個新的 EC，並設置 **EC.scope_chain = A0 + A. [[Scope]]**

想要完全搞清楚，就讓我們實際再來跑一遍這整個流程就行了，我們用下面這個非常簡單的程式碼作為範例：



```

var v1 = 10
function test() {
  var vTest = 20
  function inner() {
    console.log(v1, vTest) //10 20
  }
  return inner
}
inner = test()
outer()

```

第一步：進入 Global EC

現在進入 Global EC 並且初始化 VO 以及 scope chain，前面有講過 `scope chain = activation object + [[Scope]]`，但因為這不是一個 function 所以沒有 `[[Scope]]`，而沒有 AO 就直接拿 VO 來用。總之，最後 Global EC 會是這樣：

```

globalEC = {
  VO: {
    v1: undefined,
    inner: undefined,
    test: function
  },
  scopeChain: globalEC.VO
}

```

VO 的部分就按照之前講過的初始化，現在唯一多的步驟是多出了 scopeChain 這個屬性，而按照定義，scope chain 就是 globalEC 自己的 VO/AO。

這邊別忘了還有最後一步，那就是要設置 function 的 `[[Scope]]`，所以 test 這個 function 的 `[[Scope]]` 就會是 `globalEC.scopeChain` 也就是 `globalEC.VO`。

第二步：執行程式碼

再來第二步執行程式碼，跑了 `var v1 = 10` 之後碰到 `var inner = test()`，這邊要準備進入到 test 的 EC 了，在進入前我們現在的資訊長這樣：

```

globalEC = {
  VO: {
    v1: 10,
    inner: undefined,
    test: function
  },
  scopeChain: globalEC.VO
}

test.[[Scope]] = globalEC.scopeChain

```

第三步：進入 test EC

按照慣例，進入的時候先把 test EC 跟 AO 建立起來，然後記得 `scope chain = activation object + [[Scope]]`

```

testEC = {
  A0: {
    arguments,
    vTest: undefined,
    inner: function
  },
  scopeChain:
  [testEC.A0, test.[[Scope]]]
  : [testEC.A0, globalEC.scopeChain]
  0 : [testEC.A0, globalEC.V0]

  globalEC = {
    V0: {
      v1: 10,
      inner: undefined,
      test: function
    },
    scopeChain: globalEC.V0
  }

  test.[[Scope]] = globalEC.scopeChain
}

```

可以看到的是 testEC 的 scope chain 就是自己的 AO 加上之前設置過的 `[[Scope]]`，然後說穿了，其實 scope chain 就是上層的 EC 的 VO 嘛！只是我們用了比較複雜的程序去設置這件事情，但本質上其實就是 VO/AO 的組合。

最後別忘記設置 inner 的 scope，`inner.[[Scope]] = testEC.scopeChain`。

第四步：執行 test 中的程式碼

其實也就只跑了 `var vTest = 20` 跟 `return inner`，執行完以後變成這樣：

```

testEC = {
  A0: {
    arguments,
    vTest: 20,
    inner: function
  },
  scopeChain: [testEC.A0, globalEC.V0]
}

globalEC = {
  V0: {
    v1: 10,
    inner: undefined,
    test: function
  },
  scopeChain: globalEC.V0
}

inner.[[Scope]] = testEC.scopeChain = [testEC.A0, globalEC.V0]

```

接著把 inner 回傳回去，而 `test` 這個 function 就結束了，照理來說資源應該要被釋放才對。

可是！你有沒有發現現在 `inner.[[Scope]]` 記著 `testEC.A0`？因為有人還需要它，所以它沒辦法就這樣被釋放，僅管 `test` 結束了，`testEC.A0` 還是存在於記憶體裡面。

第五步：進入 inner EC

♥ 尤不贅述了，就按照同樣的原則去做初始化：

0

```

innerEC = {
  A0: {
    arguments
  },
  scopeChain:
    [innerEC.A0, inner.[[Scope]]]
  = [innerEC.A0, testEC.scopeChain]
  = [innerEC.A0, testEC.A0, globalEC.V0]
}

testEC = {
  A0: {
    arguments,
    vTest: 20,
    inner: function
  },
  scopeChain: [testEC.A0, globalEC.V0]
}

globalEC = {
  V0: {
    v1: 10,
    inner: undefined,
    test: function
  },
  scopeChain: globalEC.V0
}

inner.[[Scope]] = testEC.scopeChain = [testEC.A0, globalEC.V0]

```

有沒有發現就跟我剛才講的一樣，其實 scope chain 說穿了就是 VO/AO 的組合而已。

第六步：執行 inner

從 scope chain 裡面尋找 `v1` 跟 `vTest` 這兩個變數，在自己的 AO 裡面找不到所以往上找，找到 `testEC.A0` 並尋獲 `vTest`，但 `v1` 還是沒找到所以又往上一層去看 `globalEC.V0`，最後找到 `v1`，成功獲得這兩個變數的值並印出。

結束。

上面的流程講得較為詳細，可以自己再開個小視窗在旁邊搭配著程式碼一起看，一步步來看相信會比較容易理解。其實在上次討論 hoisting 的時候就已經講過這個模型了，而今天只是補充上次沒講到的部分，那就是 scope chain，加上去之後這個模型就完整許多，不但能解釋 hoisting，也能解釋為什麼 function 執行結束以後還可以存取的到那些變數。

因為那些變數被留在 `innerEC` 的 scope chain 裡面，所以不會也不能被 GC 回收掉，才會發生這種現象。

而理解了 scope chain 其實只是 VO/AO 的組合以後，也能很輕易地就知道我們開頭所說的「在 scope chain 往上找」是什麼意思，就是往上一層去看有沒有這個變數嘛，因為有的話一定會存在 VO/AO 裡面。

最後，上面這個模型還有一件事情要注意，那就是無論我有沒有把內部的 function 紹回傳（上面這個例子就是 inner），都不影響這個機制的運行。



0 是說儘管我的程式碼長這樣：



```
var v1 = 10
function test() {
  var vTest = 20
  function inner() {
    console.log(v1, vTest) //10 20
  }
  inner() // 不回傳直接執行
}
test()
```

他最後出來的模型跟剛剛的程式碼是一模一樣的，inner 都有一樣的 scope chain，並且一樣存著 test 跟 global EC 的 VO/AO。

你有注意到我們正一步步邁向我們的標題嗎？

所有的函式都是閉包

我們再回來看 wiki 上面對閉包的定義：

在電腦科學中，閉包（英語：*Closure*），又稱詞法閉包（*Lexical Closure*）或函式閉包（*function closures*），是參照了自由變數的函式。這個被參照的自由變數將和這個函式一同存在，即使已經離開了創造它的環境也不例外。所以，有另一種說法認為閉包是由函式和與其相關的參照環境組合而成的實體。

如果說你認為閉包一定要：「離開創造它的環境」，那顯然「所有的函式都是閉包」這句話就不成立；但如果你認同閉包的定義是：「由函式和與其相關的參照環境組合而成的實體」，那就代表在 JavaScript 裡面，所有的函式都是閉包。

為什麼？因為這就是 JavaScript 的運行機制，你每個宣告的 function 都會儲存著 **[[Scope]]**，而這個資訊裡面就是參照的環境。

而這個說法也不是我自創的，在解釋 ECMAScript 最經典的系列文章中的其中一篇：[ECMA-262-3 in detail. Chapter 6. Closures.](#)，裡面是這樣說的：

Let's make a note again, that all functions, independently from their type: anonymous, named, function expression or function declaration, because of the scope chain mechanism, are closures.

- ♥ *in the theoretical viewpoint: all functions, since all they save at creation*
- *variables of a parent context. Even a simple global function, referencing a global variable refers a free variable and therefore, the general scope chain mechanism is used;*

所以從理論上來說，JavaScript 裡面的所有 function 都是閉包。

from the practical viewpoint: those functions are interesting which:

continue to exist after their parent context is finished, e.g. inner functions returned from a parent function;

use free variables.

但如果你只從「實作」上的觀點來關心閉包的話，我們會說閉包必須要用到自由變數，也必須在離開了建立的 context 以後還能夠存在，這樣才是我們真正所關心的那個閉包。

所以閉包到底是什麼，端看你從哪個角度去看他，但無庸置疑的，從理論上的角度來看 JavaScript 裡所有的 function 都是閉包，如果你還是不信的話，那最後我帶你來看看 V8 是怎麼想的。

再探 V8

我們一樣寫一段簡單的程式碼，看看最後會 compile 出什麼東西：

```
var a = 23
function yoyoyo(){
}
yoyoyo()
```

這邊放一個 23 是因為方便我們在 byte code 中定位到這段程式，只有 function 會有名稱可以識別，這種寫在 global 裡面的東西比較難找。

產生出的結果是這樣：

```
[generating bytecode for function: ]
Parameter count 6
Frame size 16
  0x3e0ed5f6a9da @ 0 : 6e 00 00 02      CreateClosure [0],
[0], #2
  0x3e0ed5f6a9de @ 4 : 1e fa              Star r1
  10 E> 0x3e0ed5f6a9e0 @ 6 : 91            StackCheck
  70 S> 0x3e0ed5f6a9e1 @ 7 : 03 17        LdaSmi [23]
  0x3e0ed5f6a9e3 @ 9 : 1e fb              Star r0
  0 95 S> 0x3e0ed5f6a9e5 @ 11 : 4f fa 01
  .lUndefinedReceiver0 r1, [1]
  0x3e0ed5f6a9e8 @ 14 : 04              LdaUndefined
  107 S> 0x3e0ed5f6a9e9 @ 15 : 95          Return

[generating bytecode for function: yoyoyo]
Parameter count 1
Frame size 0
  88 E> 0x3e0ed5f6b022 @ 0 : 91          StackCheck
  0x3e0ed5f6b023 @ 1 : 04              LdaUndefined
  93 S> 0x3e0ed5f6b024 @ 2 : 95          Return
```

你只要看關鍵字就好了，你有沒有看到建立 function 的那邊是什麼？是 `CreateClosure`，我們只是很簡單的創建一個 function 並且呼叫他而已，V8 依然是用 `CreateClosure` 這個指令。

那如果今天是在 function 裡面建立一個新的 function 呢？

```
function yoyoyo(){
  function inner(){}
}
yoyoyo()
```

結果：

```
[generating bytecode for function: yoyoyo]
Parameter count 1
Frame size 8
  0x2c9f0836b0fa @ 0 : 6e 00 00 02      CreateClosure [0],
[0], #2
  0x2c9f0836b0fe @ 4 : 1e fb              Star r0
  77 E> 0x2c9f0836b100 @ 6 : 91            StackCheck
  0x2c9f0836b101 @ 7 : 04              LdaUndefined
  106 S> 0x2c9f0836b102 @ 8 : 95          Return
```

一樣還是呼叫了 `CreateClosure`。最後讓我們來試試看我們所熟知的那種，也就是要回傳建立的 function：

```
function yoyoyo(){
  function inner(){}
  return inner
}
yoyoyo()
```

結果：

```
[generating bytecode for function: yoyoyo]
Parameter count 1
Frame size 8
    0x3f4bde3eb0fa @ 0 : 6e 00 00 02      CreateClosure [0],
[0], #2
    0x3f4bde3eb0fe @ 4 : 1e fb            Star r0
    77 E> 0x3f4bde3eb100 @ 6 : 91          StackCheck
    16 S> 0x3f4bde3eb101 @ 7 : 95          Return
```

0

王哪裡？只差在前者在回傳前多了一個 `LdaUndefined` 載入 `undefined`，後者沒加所以直接出來的 `function` 傳回去。可是在建立 `function` 的指令上面，是一模一樣的，都叫做 `CreateClosure`。

只看 `compile` 出來的程式碼或許有失公允，如果能看到 V8 內部怎麼講就再好不過了。

以前曾經試圖想找但是 V8 太大了，這次我碰巧在找資料的時候看見這篇文章：[Analyze implementation of closures in V8](#)，雖然是九年前的文章，但裡面有稍微提到一些關鍵字，我循著關鍵字去找，找到了幾個很有趣的地方。

第一個是 [src/interpreter/interpreter-generator.cc](#) 這隻檔案，裡面記錄著所有 byte code 的指令，對 `CreateClosure` 它是這麼描述的：

```
// CreateClosure <index> <slot> <tenured>
//
// Creates a new closure for SharedFunctionInfo at position |index| in
// the
// constant pool and with the PretenureFlag <tenured>.
```

這檔案對以後看 byte code 很有幫助，所以要特地 po 在這邊記起來。

第二個是 [src/context.h](#)，這邊紀錄的資訊十分豐富，你可以看到這一段註解：

```
// JSFunctions are pairs (context, function code), sometimes also
// called
// closures. A Context object is used to represent function contexts
// and
// dynamically pushed 'with' contexts (or 'scopes' in ECMA-262 speak).
//
// At runtime, the contexts build a stack in parallel to the execution
// stack, with the top-most context being the current context. All
// contexts
// have the following slots:
//
// [ scope_info ] This is the scope info describing the current
// context. It
// contains the names of statically allocated
// context slots,
// needed for
// dynamic lookups in the presence of 'with' or
// 'eval', and
// for the debugger.
```

除了我們最想知道的 Closure，它也提到了 context 跟 scope info，都是我們上面所討論的東西，概念類似只是名詞有點不太一樣而已。

但最重要的是這句：

JSFunctions are pairs (context, function code), sometimes also called closures.



○ 因 JS 的 function 都記錄著 context 的資訊，再次印證了我們先前所講的機制。



...固也是最後一個，我意外地發現了 V8 裡面處理 scope 的地方，在 [src/ast/scopes.cc](#)，點連結之後連到的地方 [LookupRecursive](#) 就是在講述尋找變數的過程，先在 scope 裡面找，沒有的話再往上面找，還是找不到的話就在 global 告知一個。

熟知這段過程這麼久，第一次看見 V8 的實現長怎樣，實在是很有趣。雖然 C++ 看不太懂，但幸好文章裡面有大量的註解，所以看著註解就可以理解五六成的程式碼。

結論

有一個小地方要先說明，在我這篇以及上一篇裡面，我都刻意不提及 `eval` 跟 `with` 這兩個東西，因為這兩個東西會讓作用域變得複雜許多所以我才故意不帶到，在我看 V8 程式碼的時候也看到大量程式碼是在處理這兩個的操作，如果你對這兩個操作有興趣，可以自行去找相關的文章來看。

在上一次徹底理解 hoisting 的過程中我們有了最重要的底層機制的運行概念，也稍微看到了 V8 的 byte code，在這一次則是把上次的模型補充得更完整，只要按照那個模型去解釋程式的運行，什麼 hoisting 什麼 closure 的都能夠輕鬆理解。

這次也更深入了 V8，直接看到處理 scope 以及 context 相關的程式碼，但 V8 畢竟還是一個很大的專案，光幾個檔案我就看不完了，根本不能談上理解，因此只是以一種好玩的角度想去看一下而已。

這篇的目標跟上一篇一樣，對於本來就對這個主題不熟的人，希望能夠讓你們理解這個主題；對於已經熟悉的人也希望能帶來一些新的想法，畢竟我左看右看上看下看都沒看到什麼人直接跑去 V8 找相關的程式碼段落出來。

最後，再次幫自己工商一下，我前陣子把自己以前寫過的文章都放到了 [GitHub 上面](#)，如果有想討論的可以在[這篇文章的 GitHub 版本](#)下面留言，想收到新文章通知的也可以按個 watch 跟 star，感謝。

參考資料：

1. [从static/dynamic scope来谈JS的作用域](#)
2. [MDN](#)
3. [深入淺出瞭解 JavaScript 閉包 \(closure\)](#)
4. [你不可不知的 JavaScript 二三事#Day5：湯姆克魯斯與唐家霸王槍——變數的作用域\(Scope\)](#)
 - (1).
5. [關於 JS 作用域裡面的解釋](#)
6. [ECMA-262-3 in detail. Chapter 6. Closures.](#)
7. [Grokking V8 closures for fun \(and profit?\)](#)
8. [Understanding JavaScript Closures](#)

9. <https://javascript.info/closure>

10. [Analyze implementation of closures in V8](#)

關於作者：

@huli 野生工程師，相信分享與交流能讓世界變得更美好



0

[script](#)

#closure



huli [Follow](#)

野生工程師，相信分享與交流能讓世界變得更美好



0



0



0

讚 27

分享



TechBridge Weekly 技術...

說這專頁讚 8,531 likes

Related Posts



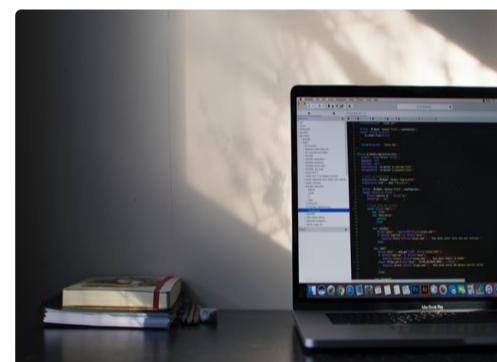
[\[JavaScript\] ES6 : Template Literals 樣板字面值](#)

[Nicolakacha](#)



[Command Line note](#)

[Sansan](#)



[MTR04_0915](#)

[cwc329](#)

Sponsored

Newsletter

First Name

Last Name

Email Address

Subscribe

Comments

[Sign In](#) to join in the discussion.



0



Copyright © TechBridge 技術共筆部落格 2021 | Powered by [CoderBridge](#) ❤