

JavaScript

六個重要的特性

The Six Essential Features of JavaScript

ES6

Eddy Chang

前言

本書主要收集與整理 JavaScript 語言的幾個重要特性，有以下幾個章節內容：

- Callback(回調)
- Closure(閉包)
- this
- Event(事件處理)
- 異步程式設計與 Eventloop(事件迴圈)
- Prototype(原型) 基礎物件導向

以上的內容都是以 ES6 標準作為基礎，但重點仍然是放在 JavaScript 原本的設計上，這些特性存在於 JavaScript 非常的久了，幾乎是從一開始這個語言誕生就有了，經過一段時間之後，有些樣式或語法，在現今的開發方式上，都有各自不同的變化，有些舊式的作法可能被淘汰，使用新的作法來取代它，而有些太怪異的設計，可能在未來的標準中會被更動或修正。不論如何，這些特性現在仍然存在於 JavaScript 之中，它會是每個初學者想要更進階地學習 JavaScript 時，必經的學習路徑。

本書大部份是取材自我之前寫的一本免費的電子書 - [從 ES6 開始的 JavaScript 學習生活](#)，這本書可以很容易在網路上找到，它的內容相當豐富，也詳盡地說明了 ES6 的重要特性，以及 JavaScript 語言的重要內容。因為本書的篇幅有限，所以在閱讀過本書後，你可以再看這本電子書裡面相關的章節，作為一個補充與進階的學習。

在本書中希望用較為白話的方式，介紹該如何來學習這個 JavaScript 程式語言，每一個建議與意見，背後都有它的意涵，也有很多是實際中得到的寶貴經驗。如果你有發現任何的問題，歡迎 email 紿我。

~ Eddy Chang (eddy@joomla.com.tw) 2018.10(第 1 版)

Callback(回調)

Callback(回調)是什麼？

中文翻譯字詞會用"回呼""、"回叫"、"回調"，都是聽起來很怪異的講法，Callback 在英文是"call back"兩個單字的合體，你應該有聽過"Call me back"的英文，實際情況大概是有客戶打來電話給你，可是你正在電話中，客戶會留話說請你等會有空時再"回電"給它，這裡的說法是指在電信公司裡的 callback 的涵意。而在程式開發上，callback 的使用情境也是類似的地方。

註："回調"是簡體中文的翻譯，它會對應到繁體中文中的"函式呼叫或執行"，在簡體中文會被譯為"函數調用"，但"回調"在網路上的說法比較常見，"回呼"反而很少見。

CPS 風格與直接風格

延續傳遞風格(Continuation-passing style, CPS)，它的對比是"直接風格(Direct style)"，這兩種都是程式開發時所使用的風格。CPS 早在 1970 年代就已經被提出來，CPS 用的是明確地移轉控制權到下一個函式中，也就是使用"延續函式"的方式，一般稱它為"回調函式"或"回調(Callback)"。回調是一個可以作為傳入參數的函式，用於在目前的函式呼叫執行最後移交控制權，而不是使用函式回傳值的方式。相反的，直接風格的控制權移交是不明確的，它是用回傳值的方式，然後進行到下一行程式碼或呼叫接下來其他函式，下面以範例來說會比較容易。

直接風格的範例如下，就是一般函式呼叫的方式，或是使用回傳的方式：

```
//直接風格
function func(x) {
    return x
}
```

CPS 風格就不是這樣，它會用另一個函式作為函式中的傳入參數的樣式來撰寫程式，然後將本來應該要回傳的值(不限定只有一個)，傳給下一個延續函式，繼續下個函式的執行：

```
//CPS風格
function func(x, cb) {
    cb(x)
}
```

以明確的程式流程的例子來說，假設現在要從資料庫獲取某個會員的資料，然後把裡面的大頭照片輸出。

用直接風格的寫法：

```
function getAvatar(user) {
    //...一些程式碼
    return user
}
```

```
function display(avatar) {
    console.log(avatar)
}

const avatar = getAvatar('eddy')
display(avatar)
```

用 CPS 風格的寫法:

```
function getAvatar(user, cb) {
    //...一些程式碼
    cb(user)
}

function display(avatar) {
    console.log(avatar)
}

getAvatar('eddy', display)
```

長久以來在程式語言開發界，直接風格是最常被使用的，因為它容易被學習與理解，一個步驟接著一個步驟，也容易將函式的功能獨立拆分，在一般學校所教授的程式語言課程，大部份也是用這種風格來教學，在個人電腦上的，或是在伺服器端的程式語言設計通常也是這樣。而在個人電腦端或是伺服器端，通常使用多執行緒或改進底層運作的執行方式，來解決多工或並行的問題。CPS 風格很少被使用，也沒有明顯的誘因讓程式開發者一定要用 CPS 風格，另一個重要的原因是，不是所有的程式語言都能使用 CPS 風格，要有一些特殊的特性的程式語言才可以使用，所以在過去 CPS 風格並不算是主流的程式開發寫作風格。

CPS 風格相較於直接風格還有一些很明顯的缺點:

- 在愈複雜的應用情況時，程式碼愈不易撰寫與組織，維護與閱讀性也很低
- 在錯誤處理上較為困難

現今大概用最多的 CPS 風格的程式語言，就只有 JavaScript 這個程式語言而已。JavaScript 中會使用 CPS 風格，除了它本身的語言特性可以使用這種風格外，其實是有另它重要的原因:

- JavaScript 執行上的設計只有單執行緒，在瀏覽器端只有一個使用者，但事件或網路要求(AJAX)要求不能阻塞其他程序的進行，但這也僅限在這些特殊的情況。不過在伺服器端的執行情況都很嚴峻，要能同時讓多人連線使用，必需要達到不能阻塞 I/O，才能與以多執行緒執行的伺服器一樣的執行效益。
- JavaScript 一開始就是以 CPS 風格來設計事件異步處理的模型，用於配合異步回調函式的執行使用。

註：基本上一個程式語言要具有高階函式(High Order Function)的特性才能使用 CPS 風格，也就是可以把某個函式當作另一函式的傳入參數，也可以回傳函式。除了 JavaScript 語言外，具有高階函式特性的程式語言常見的有 Python、Java、Ruby、Swift 等等。

異步回調函式

並非所有的使用 callbacks(回調)函式的 API 都是異步執行的，但 CPS 的確是一種可以確保異步回調執行流程的風格。在 JavaScript 中，除了 DOM 事件處理中的回調函式大部份都是異步執行的之外，內建 API 中使用的回調函式並不一定是異步執行的，也有同步執行的。要讓開發者自訂的 callbacks(回調)的執行轉變為異步，有以下幾種方式：

- 使用計時器(timer)函式: `setTimeout`, `setInterval`
- 特殊的函式: `nextTick`, `setImmediate`
- 執行 I/O: 監聽網路、資料庫查詢或讀寫外部資源
- 訂閱事件

針對 callbacks(回調)函式來說，異步與同步的執行到底是差在那裡？下面用個簡單的例子來說明：

```
function aFunc(value, callback) {
    callback(value)
}

function bFunc(value, callback) {
    setTimeout(callback, 0, value)
}

function cb1(value) {
    console.log(value)
}
function cb2(value) {
    console.log(value)
}
function cb3(value) {
    console.log(value)
}
function cb4(value) {
    console.log(value)
}

aFunc(1, cb1)
bFunc(2, cb2)
aFunc(3, cb3)
bFunc(4, cb4)
```

`aFunc`是一個簡單的回調結構，`callback`回調函式被傳入後最後以`value`作為傳入參數執行。

`bFunc`函式則是包裹了一個`setTimeout`內建方法，它可以在一定時間內(第二個參數)執行第一個參數，也就是`setTimeout`會執行的回調函式，第三個參數是要加入到回調函式的傳入參數值。

`aFunc`中使用了一般的回調函式，只是傳入到函式中當作參數，然後最後執行而已，這種是同步執行的回調函式，只是用了 CPS 風格的寫法。

`bFunc`中使用了計時器 API `setTimeout`會把傳入的回調函式進行異步執行，也就是先移到工作佇列中，等執行主執行緒的呼叫堆疊空了，在某個時間回到主執行緒再執行。所以即使它的時間設定為 0 秒，裡面的回調函式並不是立即執行，而是會暫緩(延時)執行的一種回調函式，一般稱為異步回調函式。

最後的執行結果是1 → 3 → 2 → 4，也就是說，所有的同步回調函式都執行完成了，才會開始依順序執行異步的回調函式。如果你在瀏覽器上測試這個程式，應該會明顯感受到，2 與 4 的輸出時，會有點延遲的現象，這並不是你的瀏覽器或電腦的問題，這是因為不論你設定的`setTimeout`為 0，它要回到主執行緒上執行，仍然需要按照內部事件迴圈所設定的時間差，在某個時間點才會回來執行。

這個程式執行的流程，可以看[這個在 loupe 網站的流程模擬](#)，輸出一樣在瀏覽器的主控台中可以看到。

由這個範例中，可以看到異步回調函式執行比同步回調函式會更慢，"異步回調函式"還有另一個名稱是"延時回調"(defer callback)，是用延時執行特性來命名。這只是一種因應特別情況所採用的函式執行方式，例如需要與外部資源存取(I/O)、DOM 事件處理或是計時器的情況。等待的時間則是在 Web API 中，等有外部資源有回應了(或超時)才會加到佇列中，佇列裡並不會執行函式中的程式碼，只是個準備排隊進入主執行緒的機制，函式一律在主執行緒中執行。

註：關於函式的異步執行與事件迴圈一些原理的說明，請再參考[異步執行與事件迴圈]的章節裡的內容。

回調函式的複雜性

`callback`(回調)運用在瀏覽器端似乎並沒有想像中複雜，一個事件的處理範例大概會像下面這樣：

```
const el = document.getElementById('myButton')

el.addEventListener(
  'click',
  function() {
    console.log('hello!')
  },
  false
)
```

你也可以把 `callback`(回調) 寫成另一個函式定義，看起來會更清楚：

```
function callback() {
  console.log('hello!')
}

const el = document.getElementById('myButton')

el.addEventListener('click', callback, false)
```

AJAX 是另一個常使用的情況，內建的`XMLHttpRequest`物件的行為類似於事件處理，而且是已經打包好的。實際上`onreadystatechange`這個屬性，就是`XMLHttpRequest`物件在處理事件用的 `callback`(回調)函式。以下為一個簡單的範例：

```
var xhr = new XMLHttpRequest()

xhr.onreadystatechange = function() {
```

```

if (xhr.readyState == XMLHttpRequest.DONE) {
  if (xhr.status == 200) {
    document.getElementById('myDiv').innerHTML = xhr.responseText
  } else if (xhr.status == 400) {
    console.log('There was an error 400')
  } else {
    console.log('something else other than 200 was returned')
  }
}

xhr.open('GET', 'ajax_info.txt', true)
xhr.send()

```

"匿名函式"、"函式定義"與"函式呼叫"的混合

callback(回調)函式會讓開發者覺得複雜的主要原因，是來自"匿名函式"、"函式定義"與"函式呼叫"的混合寫法，很多不同的特性和語法混在一起，會讓程式碼的閱讀性很差，本章以一個簡單的例子開始來說明。

```

function func(x, cb) {
  cb(x)
}

func(123456, function(value) {
  console.log(value)
})

```

這例子很簡單，要分作幾個部份來看：

這是一個完整的"函式呼叫"，也就是說它是一個被執行的語句結構：

```

func(123456, function(value) {
  console.log(value)
})

```

但在其中的這一段程式碼是什麼？這個是一個"函式定義"的宣告，而且還是個"匿名函式"的定義宣告，它很顯然地是一個 callback(回調)函式的定義，它代表了func函式執行完後要作的下一件事，這個定義是在func函式中的程式碼的最後一句被呼叫執行：

```

function(value){
  console.log(value)
}

```

所以整個語法是代表"在函式呼叫時，要寫出下一個要執行的函式定義"，這就是常見回調函式的語法樣式。當然，你可以另外用一個函式來寫得更清楚：

```

function func(x, cb) {
  cb(x)
}

function callback(value) {
  console.log(value)
}

func(123456, callback)

```

不過，你可以發現幾件事情：

- `callback`(回調)的函式名稱，可以用匿名函式取代。(實際上 `callback` 的名稱在除錯時很有用，可以在錯誤的堆疊上指示出來)
- `callback`(回調)因為是函式的定義，所以傳入參數`value`的識別名稱叫什麼其實都可以
- `callback`(回調)其實有 Closure(閉包)結構的特性，可以獲取到 `func` 中的傳入參數，以及裡面的定義的值。(註：實際上 JavaScript 中只要函式建立就會有閉包產生)

那麼要說到 `callback`(回調)的最大優點，就是它給了程式開發者很大的彈性，允許開發者可以自訂下一個要執行函式的內容，也就是可以提高函式的擴充性與重覆使用性

回調地獄(Callback Hell)

更複雜的情況會出現在 CPS 風格使用 `callback`(回調)來移往下一個函式執行，當你開始撰寫一個接著一個執行的流程，也就是一個特定工作的函式呼叫後要接下一個特定工作的函式時，就會出現所謂的"回調地獄"的結構，像下面這樣的例子：

```

step1(x, function(value1) {
  //do something...
  step2(y, function(value2) {
    //do something...
    step3(z, function(value3) {
      //do something...
    })
  })
})

```

它的執行順序應該是`step1 -> step2 -> step3`沒錯，這三個都可能是已經寫好要作某件特定工作的函式。所以真正是這樣的流程嗎？你可能忘了匿名函式(`callback`)也是一個函式，所以執行的步驟是像下面這樣才對：

1. `step1`執行後，"value1"已經有值，移往`function(value1)`執行
2. `function(value1)`執行到`step2`，`step2`執行到最後，"value2"已經有值，移往`function(value2)`執行
3. `function(value2)`執行到`step3`，`step3`執行到最後，"value3"已經有值，移往`function(value3)`執行
4. `function(value3)`執行完成

寫成流程大概是像下面這樣的順序，一共有 6 個函式要執行的流程，其中的這三個匿名回調函式的主要工作，是負責準備接續下一個要執行特定工作的函式：

```
step1 -> function(value1) -> step2 -> function(value2) -> step3 -> function(
```

那你可能會想問，為何為不使用直接風格？而一定要用這麼難理解的程式流程結構？

上面已經有提到 JavaScript 中大量的使用 CPS 的原因：

因為有些 I/O 或事件類的函式，用直接風格會造成阻塞，所以要寫成異步的回調函式，也就是一定要用 CPS

你可能會認為阻塞有這麼容易發生嗎？

是的，在 JavaScript 中要"阻塞"太容易了，它是單執行緒執行的設計，一個比較長時間的程序執行就會造成阻塞，下面的for迴圈就會讓你的按鈕按下去沒反應，而且幾個訊息都要一段時間執行完才會顯示出來：

```
const el = document.getElementById('myButton')

el.addEventListener(
  'click',
  function() {
    alert('hello!')
  },
  false
)

const aArray = []
for (let i = 0; i < 1000000000; i++) {
  aArray[i] = i + 10
}

console.log('aArray done!')

const bArray = []
for (let i = 0; i < 1000000000; i++) {
  bArray[i] = i * 10
}

console.log('bArray done!')
```

或許你會認為在瀏覽器上讓使用者等個幾秒鐘不會怎麼樣，但如果在要求能讓多個使用者同時使用的伺服器上，每個使用者都來阻塞主執行緒幾秒，這個伺服器程式就可以廢了。不過，以異步執行的異步回調函式並不代表就不會阻塞，也有可能從佇列回到主緒行緒後，因為需要 CPU 密集型的運算，仍然會阻塞到緒行緒的進行。異步回調函式，只是暫時先移到佇列中放著，讓它先不干擾目前的主執行緒的執行而已。這是 JavaScript 為了在只有單執行緒的情況，用來達成並行(concurrency)模型的設計方式。

如果要配合 JavaScript 的異步處理流程，也就是非阻塞的 I/O 處理，只有 CPS 可以這樣作

在伺服端的 Node.js 一開始就使用了 CPS 作為主要的 I/O 處理方式，說實在是一個不得已的選擇，當時沒有太多的選擇，而且這原本就是 JavaScript 中對異步回調函式的設計。Node.js 使用的是 error-first(以錯誤為主)的 CPS 風格，因為考慮到 callback(回調)要處理錯誤不容易，所以優先處理錯誤，它的主要原則如下：

- callback 的第一個參數保留給 Error(錯誤)，當錯誤發生時，它將會以第一個參數回傳
- callback 的第二個參數保留給成功回應的資料。當沒有錯誤發生時，error(即第一個參數)會設定為 null，然後將成功回應的資料傳入第二個參數

一個典型的 Node.js 的回調語法範例如下：

```
var fs = require('fs')

fs.readFile('foo.txt', 'utf8', function(err, data) {
  if (err) {
    console.log('Unknown Error')
    return
  }
  console.log(data)
})
```

Node.js 使用 CPS 風格在複雜的流程時，很容易出現回調地獄的問題，這是因為在伺服器端的各種 I/O 處理會相當頻繁而且複雜。我會認為這是一種舊時代的程式碼組織方式的弊病，或是當時不得已的解決方案，當可以採用更好的語法結構來取代它時，這種語法未來大概只會出現在教科書中，現在這種寫法都已經不建議使用，你有很多其它的新的作法來處理這類的工作，例如 Promise、generator、async/await 之類的語法結構，或是專門處理用的一些函式庫等，來改善或重構原本的程式碼結構，在維護程式碼上會比較容易，這些才是你現在應該就要學習的方式。

反樣式(anti-pattern)

回傳 callback

callback(回調)有個很常見的反樣式，它會出現在如果回調除了要進行下一步之外，還要負責處理函式在執行中途的錯誤情況，例如：

```
//這是錯誤的寫法，最後的callback()依然會執行
function foo(err, callback) {
  if (err) {
    callback(err)
  }
  callback()
}
```

為了讓最後的 callback()不執行，可以正確的作錯誤處理，有可能會寫成這樣：

```
// 相當不好的寫法
function foo(err, callback) {
```

```
if (err) {  
    return callback(err)  
}  
callback()  
}
```

但是return用在 callback 上是個不對的樣式，正確的寫法應該是要用下面的寫法：

```
// 正確的寫法  
function foo(err, callback) {  
    if (err) {  
        callback(err)  
        return  
    }  
    callback()  
}
```

或是用if...else寫清楚整個情況，但這個樣式也不是太理想，CPS 風格寫法的最後一行應該就是個回調函式：

```
// 不是很理想的寫法  
function foo(err, callback) {  
    if (err) {  
        callback(err)  
    } else {  
        callback()  
    }  
}
```

參考資料

- [By example: Continuation-passing style in JavaScript](#)
- [Asynchronous programming and continuation-passing style in JavaScript](#)
- [Enforce Return After Callback](#)

Closure(閉包)

閉包定義

Closure 這個字詞是由 Close 與字尾-ure 所構成，-ure 有"動作"、"進行"或"結果"的意思，如果 close 是關閉的意思，closure 就是關閉的結果或動作，它可以作為名詞或動詞使用，中文有"封閉"、"終止"或"結束"的意思。

由於 JavaScript 語言中的函式是頭等函式(first-class function)的設計，代表函式在語言中的應用上享有與一般原始資料類型的值同等地位，函式可以傳入其他函式作為傳入參數，可以當作另一個函式的回傳值，也可以指定為一個變數的值，或是儲存在資料結構中(例如陣列或物件)，在語言中甚至是都有自己獨有的資料類型(typeof一個函式回傳值是'function')。

簡單地來說，閉包是一種資料結構，包含函式以及記住函式被建立時當下環境。

由於"閉包"這個字詞有多層意義，你可以說它是一種技術，或是一種資料結構，或是這種有記憶環境值的函式。

在 JavaScript 中每當函式被建立時，一個閉包就會被產生，閉包是一個函式建立時的就有的自然就有的特性。雖然我們經常使用函式中的函式，也就是巢狀(nested)函式(或內部函式)的語法結構作為範例來說明閉包，它也是一種最常見的、可被重覆利用閉包的語法樣式，但並不是只有巢狀函式才能產生閉包。以下是一個簡單的巢狀函式的範例：

```
function aFunc(x) {
  function bFunc() {
    console.log(x++)
  }
  return bFunc
}

const newFunc = aFunc(1)
newFunc()
newFunc()
```

bFunc 可以不需要名稱，直接用 return 匿名函式的語法更簡潔：

```
function aFunc(x) {
  return function() {
    console.log(x++)
  }
}
```

用箭頭函式更是簡潔，已經快要可以寫成一行了：

```
function aFunc(x) {
  return () => console.log(x++)
```

}

再簡化下去是兩道箭頭函式的寫法，這個寫法現在已經是很普遍了，請務必要理解這個語法是什麼意思與怎麼改寫來的：

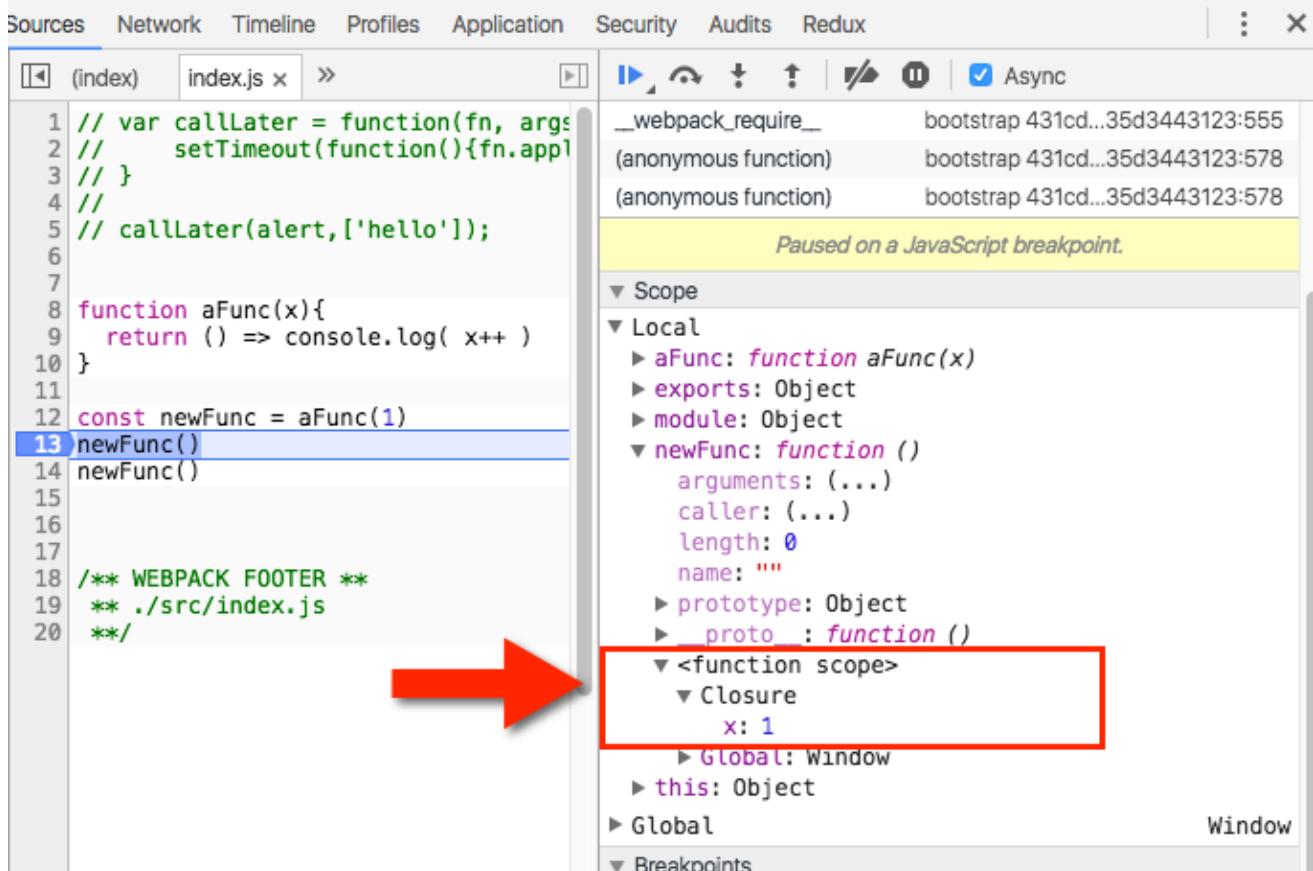
```
const aFunc = x => () => console.log(x++)
```

執行這個範例後，你會發現x值會在aFunc呼叫後陰魂不散的還留在新的newFunc函式裡，每當執行一次newFunc函式，x值就會再+1。

不過，用這個閉包範例可能會產生誤解的地方，在於以下幾點：

- "函式呼叫"與"函式建立"實際上是兩件事情，在aFunc"函式呼叫"過後，newFunc才是"函式建立"，這時候newFunc中的閉包結構才會產生。
- 匿名函式的使用。閉包並不是只會在匿名函式才會產生，純粹為了簡化語法使用匿名函式。
- 閉包不是只會產生在巢狀(內部)函式的回傳時。所有函式在建立時都會產生閉包。

要觀察閉包中所記錄的環境變數值，可以從瀏覽器的除錯器中看到，像上面的範例如果用瀏覽器除錯器在第一個newFunc函式加入中斷點時，執行後應該可以看到像下面的圖：



圖中可以看到作用域(Scope)會出現一種名稱為"Closure(閉包)"的變數值，這是觀察閉包中的環境變數值的方式。

註：在閉包中所記憶的變數與值，通常稱為自由(free)變數或獨立(independent)變數，這些變數是在函式中使用，但被封入作用域之中。

典型範例圖解

一個典型的 Closure(閉包)會長這個樣子：

```
const varGlobal = 'x'

function outer(paramOuter) {
  const varOuter = 'y'

  function inner(paramInner) {
    const varInner = 'z'

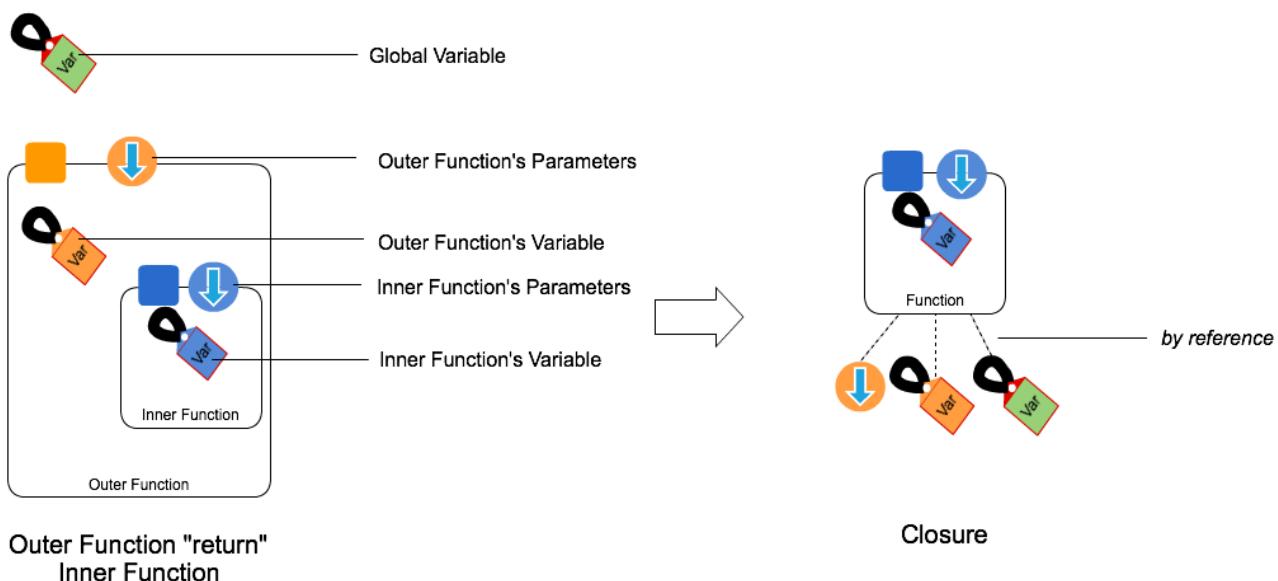
    //print
    console.log(varGlobal)
    console.log(varOuter)
    console.log(varInner)
    console.log(paramOuter)
    console.log(paramInner)
  }

  return inner
}

const func = outer('a')
func('b')
```

圖解

用簡單的圖來表示上面的例子，內部(Inner)函式被回傳後，除了自己本身的程式碼外，也會捕抓到了環境的變數值，記住了執行當時的環境：



為什麼可以這麼作？

要深究為什麼可以這麼作的原因，其實可以從下面兩個方面來看，當然複雜的底層實作就先不討論了，基本上這是一個 JavaScript 語言的特性就是：

- 函式在 JavaScript 中的設計：函式可以像一般的數值使用，可以在變數、物件或陣列中儲存，也可以傳入到另外的函式裡當參數，也可以當回傳值回傳。
- 函式作用域連鎖規則：內部函式可以看到(或存取得到)外部函式，而形成一個 Scope Chain(作用域連鎖)，內部函式可以有三個作用域：
 - 自己本身的
 - 外部函式的
 - 全域的

閉包的記憶環境

閉包的最大特點就是它會記憶住函式建立時的環境，也就是內部函式所能存取得到的作用域連鎖中的所有變數當下的值。

那麼這裡會產生一個問題，閉包是完全拷貝(copy)了這些值？還是只是參照(refer)指向這些值而已？

答案是"參照(refer)而非複製"。

最常用來解說的這個概念的是下面這個典型範例，因為它用了異步的回調函式，所以你可能要對異步回調有點概念才能理解：

```
//錯誤示範
function counter() {
    let i = 0
    for (i = 0; i < 5; i++) {
        setTimeout(function() {
            console.log('counter is ' + i)
        }, 1000)
    }
}

counter()
```

這個counter是我們希望能利用閉包結構，記憶其中的變數i，最後希望的結果是0,1,2,3,4這樣，不過上面這個範例是個錯誤的示範，並不會產生這個結果，最後的結果是5,5,5,5,5。

會造成這個結果的原因在這一節的最前面就說完了，因為"閉包結構中所記憶的環境值是用參照指向的"，setTimeout中的異步回調會先移到工作佇列中準備延時執行，等它回來主執行緒執行時，i老早就跳出迴圈執行，而且還變成了5，所以接下來執行的這些異步回調函式，能獲取到的i值全部都是5。要達到需求的話，可以用好幾種解法，這些都會考驗你的概念與基礎知識。

第一種是解除原先counter中閉包的結構，也就是說你已經知道這樣會造成閉包，而且會記憶(參照)到環境值，乾脆讓這個情況解除，像下面的寫法：

```

function counter(x) {
  setTimeout(function() {
    console.log('counter is ' + x)
  }, 1000)
}

for (let i = 0; i < 5; i++) {
  counter(i)
}

```

新的寫法中，counter函式中確保不會再帶有函式裡面的變數值，而是用一個傳入參數值讓setTimeout中的回調函式去作對應的輸出，因為每次傳入counter函式的x值都不同，也就能控制回調函式所能存取得到的環境值，所以能確保異步回調的輸出每次都是不同的。

第二種是想辦法鎖住setTimeout中回調函式的環境值，這個解法的有點像武俠小說中的慕容家族的"以彼之道，還之彼身"的感覺，用閉包來解決閉包的問題：

```

function print(i) {
  return function() {
    console.log('counter is ' + i)
  }
}

function counter() {
  let i = 0
  for (i = 0; i < 5; i++) {
    setTimeout(print(i), 1000)
  }
}

counter()

```

setTimeout中的print(i)一但被執行，會回傳一個帶有閉包的函式，也就是會鎖住當下傳入的值，作為setTimeout的回調函式。

第三種解決是要用 IIFE(立即呼叫函式表達式)的樣式，這個樣式已經是很特殊的用法了，IIFE 也有儲存閉包的環境狀態的功用：

```

for (let i = 0; i < 5; i++) {
;(function(value) {
  setTimeout(function() {
    console.log('counter is ' + value)
  }, 1000)
})(i)
}

```

註：在上面 for 迴圈內語法稱為"Immediately-Invoked Function Expression"（立即被呼叫的函式語樣）或簡稱為"IIFE"

閉包的記憶環境例外變數

閉包只有以下兩個外部函式的環境變數是不會自動記憶的，相信你如果很仔細的看過上面的範例，就已經有發現了，這其實是內部函式的特性，這兩個本來就不算是作用域鏈的成員之一。除非你先用另外的變數指定這兩個值，不然內部函式是獲取不到的，由於內部函式自己也是個函式，它也有自己的this與arguments值。這兩個例外值是：

- this: 執行外部函式時的this值
- arguments: 函式執行時的一個隱藏偽陣列物件

相關樣式

利用閉包這種特性，可以發展出很多適合各種情況應用的樣式，有許多樣式或許你已經有看過，或是有使用到的。要注意的是，以下的樣式有一些是很舊的語法了，並不符合新時代的程式開發使用，只是提出一些可能還會看得到的樣式，供學習者參考而已。

柯里化(Currying)與部份應用(Partial application)

柯里化是一種源自數學中求值的技術，它與部份應用(Partial application)經常被一起討論，這些都是在程式設計上稱為"部份求值"或"惰性(延時)求值"的一種技巧。JavaScript 語言中可以使用閉包結構很容易地實現這個技術。這個技術可以應用到不同的複雜情況，這裡只是篇簡介而已。

要理解這個技術先理解其中幾個專用術語的不同定義：

- 應用(Application): 代表傳入一個函式所需的傳入值，然後最後得到回傳結果。
- 部份應用: 代表一個函式其中有部份的傳入值(一個或多個)被傳入，然後回傳一個已經有部份傳入值的函式。
- 柯里: 一個具有多個傳入參數的函式，轉變為一個一次只傳入一個參數，只會回傳一個只有一個傳入參數的函式。也就是說把原本多個傳入參數的函式，轉變為一次只傳入一個參數的函式，可以連續呼叫使用。

柯里化與部份應用雖然都是套件部份的傳入參數值，但它們不太相同，也有下面幾個很明顯的差異：

- 部份應用: 一次套用一個或多個傳入參數，回傳函式有可能與原來函式結構的不同。
- 柯里: 一次只套用一個傳入參數，回傳另一個函式，回傳的函式與原來的結構相同，直到所有傳入參數都被套用才會回傳值。

部份應用或柯里，常會用在固定某些已確定的參數值使用，在許多工具性函式庫或框架中經常被使用，以此提高函式的重覆使用性。基本上部份應用或柯里都是從左至右傳入參數值的。如果你要從右至左傳入參數值，類似像外部函式庫[lodash](#)中有從右至左的 curryRight 與 partialRight 函式，可能要再改寫或用這類的工具函式庫。另外也有一套知名的工具函式庫[Ramda](#)，它是完全以部份應用與柯里的特性為核心的函式庫。

註: Partial在專業術語中，中文也常會翻譯為"局部"、"偏"，例如"partial differential equation, PDE"是偏微分方程式的意思。

註: Curry的英文字詞是"咖哩"的意思，不過這裡是指這個技術以"Haskell Curry(柯里)"數學家的名字來命名。

部份應用(Partial application)

部份應用按照定義並沒有那麼嚴格，就只要能套用部份的傳入參數值就行了，有很多種寫法也不一定要用閉包結構。以下是要改寫的原本函式：

```
//原本的函式
function add(x, y, z) {
    return x + y + z
}
```

第一種寫法是用另一個函式來套用部份的值即可：

```
function addXY(z) {
    return add(1, 2, z)
}

addXY(3)
```

第二種寫法是用函式物件中的bind方法，它可以回傳一個套用部份參數值的新函式(第一個參數值是context，也就是this值，這裡不需要)：

```
const addXY = add.bind(null, 1, 2)

addXY(3)
```

第三種寫法是用閉包結構，不過要把原來的函式改寫才行：

```
//改寫
function add(x, y, z) {
    return function(z) {
        return x + y + z
    }
}

const addXY = add(1, 2)
addXY(3)
```

柯里化

柯里化會比較麻煩些，只能使用閉包結構來改寫原本的函式，例如下面的原本函式與柯里化後的樣子比較：

```
//原本的函式
add(x, y, z)

//柯里化後
add(x)(y)(z)
```

因為 JavaScript 中有閉包的特性，所以要改寫是容易的，需要改寫一下原先的函式：

```
//原本的函式
function add(x, y, z) {
    return x + y + z
}
```

```
//柯里化
function add(x, y, z) {
    return function(y) {
        return function(z) {
            return x + y + z
        }
    }
}
```

```
add(1)(2)(3)
```

第一個傳入值x會變為閉包中的變數被記憶，然後是第二個傳入值y，最後的加總是由閉包結構中的x與y與傳入參數z一起加總。這個範例中用了三個傳入參數，如果你沒辦法一下子看清楚，可以用二個傳入參數的情況來練習看看。

IIFE(立即呼叫函式表達式)

IIFE 是一種運用閉包與匿名函式立即執行的樣式，它的常見基本語法是下面這種：

```
; (function() {
    /* code */
})()
```

註：因為這裡的範例使用的是"不使用分號作為語句結尾"，像 IIFE 這種語句，如果最前面沒有指定運算時(以圓括號為開頭時)，會先使用分號();作開頭，這是一種保護性的語法。

IIFE 是一種會在建立時就會立即執行的匿名函式，經常用於封閉住一個作用域，避免與全域作用域污染。一個簡單的 IIFE 範例如下，counter是一個會鎖住函式裡面的變數值的閉包，這個樣式通常會用來模擬靜態變數。

```
const counter = (function() {
    let i = 1

    return function() {
        console.log(i++)
    }
})()
```

```
counter() //1  
counter() //2
```

物件封裝/物件工廠

使用閉包來產生物件，而不是用 Prototype 與 new 運算符。程式碼來自[Why use "closure"?](#)：

```
// 宣告一個工廠  
function newPerson(name, age) {  
    // 在閉包中儲存訊息  
    const message = name + ', who is ' + age + ' years old, says hi!'  
  
    return {  
        // 定義同步執行的函式  
        greet: function greet() {  
            console.log(message)  
        },  
  
        // 定義異步執行的函式  
        slowGreet: function slowGreet() {  
            setTimeout(function() {  
                console.log(message)  
            }, 1000)  
        },  
    }  
  
    const tim = newPerson('Tim', 28)  
    tim.greet()
```

模組(Module)樣式

模組樣式是用來模擬物件中的私有成員上(私有屬性與方法)與公開成員，JavaScript 語言中的物件導向本身並沒有這種設計，這個樣式使用了 IIFE。以下程式碼來自[JavaScript Closures and the Module Pattern](#)

```
var Module = (function() {  
    // 以下的方法是私有的，但是可以被公開的函式存取  
    function privateFunc() { ... }  
  
    // 回傳要指定給模組的物件  
    return {  
        publicFunc: function() {  
            privateFunc() // publicFunc 可以直接存取 privateFunc  
        }  
    }  
}())
```

模組樣式有另一種變型，稱之為"暴露的模組樣式(Revealing Module Pattern)"，語法會比原來的模組樣式容易理解得多。以下為一個範例：

```
var Module = (function() {
    // 所有函式現在可以互相存取
    var privateFunc = function() {
        publicFunc1()
    }

    var publicFunc1 = function() {
        publicFunc2()
    }

    var publicFunc2 = function() {
        privateFunc()
    }

    // 回傳要指定給模組的物件
    return {
        publicFunc1: publicFunc1,
        publicFunc2: publicFunc2,
    }
})()
```

閉包使用時注意事項

閉包結構的各種運用是一種高消費的語法，主因在於它需要尋遍整個作用域連鎖與記憶環境。這些都是需要時間來完成的動作。閉包結構也會鎖住額外的記憶體，所以要小心運用在記憶體高使用的語法上，例如迴圈或定時執行的函式。一般情況下，我們不需要擔心記憶體回收的問題，JavaScript 引擎有很好的 GC 機制來回收不需要使用的記體體。

結語

總結一下所有本章節的內容，讓你對閉包的理解有比較清楚的思維。

- 所有函式在建立時都會產生閉包。
- 閉包不是只會產生在巢狀(內部)函式的回傳時，巢狀(內部)函式是一種最常利用閉包結構的樣式，因為它可以重覆使用閉包中的記憶環境。
- 閉包所記憶的環境，其原理是來自作用域連鎖的設計，內部函式可以看(獲取)到外部函式的變數值與傳入參數值。
- 函式的this值與arguments值並不屬於作用域連鎖，所以不包含在閉包記憶的環境中。
- 閉包的運用是一種高消費的語法。

參考資料

- [How do JavaScript closures work?](#)
- [Master the JavaScript Interview: What is a Closure?](#)
- [Understanding JavaScript Closures](#)

- Really Understanding Javascript Closures

this

在其他以類別為基礎的程式語言中，this指的是目前使用類別進行實體化的物件。而在 JavaScript 語言中，因為在設計上並不是以類別為基礎的物件導向，設計上並不一樣，它的this的指向的是目前呼叫函式或方法的擁有者(owner)物件，也就是說它與函式如何被呼叫或調用有關，雖然是同一函式的呼叫，因為不同的物件呼叫，也有可能是不同的this值。

函式的呼叫

我們在"函式與作用域"、"物件"與"原型物件導向"的章節中，都有看到函式的一些說明內容，也有看到用函式作為建構式來作物件實體化的工作，這時候會看到以this的一些說明，那麼在不是用來當作建構式的函式中，就是我們所認知的一般函式，裡面也有this嗎？有的，不論是在物件中的方法，或是一般的函式，每個函式中都有this值。以下是一個很簡單的範例，一個是我們所認知的普通函式，一個是在物件中的方法：

```
function func(param1, param2) {  
    console.log(this)  
}  
  
const objA = {  
    test() {  
        console.log(this)  
    },  
}  
  
func() // undefined  
objA.test() // Object(objA)
```

func函式在呼叫時的this值是undefined，原本它應該會回傳全域物件，在瀏覽器中就是window物件，這是因為像 babel 這種編譯工具，預設會開啟 strict mode(嚴格模式)，為了安全性的理由，把原本的全域物件變成了undefined，以下的內容也是用這樣的作法。

objA.test方法在呼叫時的this值就是objA物件本身，這一點不難理解。用以下的範例可以檢查this和objA是不是相同。

```
const objA = {  
    test() {  
        console.log(this)  
        console.log(this === objA) //true  
        console.log(objA)  
    },  
}  
  
objA.test()
```

不過這裡有個小地方會讓你覺得很不可思議的是，objA物件中的方法竟然可以讀取到objA物件？

是的，實際上它不止物件中可以讀取到物件本身，物件中的方法還可以執行自己本身的方法，下面的程式碼還可以讓你的瀏覽器無止盡的執行，然後當掉：

```
//注意：瀏覽器有可能會當掉
const objA = {
  test() {
    objA.test()
  },
}
```

在函式中也可以這樣作，也是會讓你的瀏覽器最後當掉，這是都是錯誤的示範：

```
//注意：瀏覽器有可能會當掉
function func(param1, param2) {
  func()
}

func()
```

這原因也是 JavaScript 原本的設計造成的，因為內部的設計涉及太過底層，在此就不多加說明。

深入函式中

所有的函式在呼叫時，其實都有一個擁有者物件來進行呼叫。所以你可以說，其實所有函式都是物件中的"方法"。所有的函式執行都是以 Object.method() 方式呼叫。

關於這一點，下面的範例就可以說明一切了，有個全域物件(在瀏覽器中是 window 物件)是所有函式在呼叫時預設物件，下面三種函式呼叫都是同樣的作用：

```
function func(param) {
  console.log(this)
}

window.func() // 只能在不是strict mode下執行
this.func() // 只能在不是strict mode下執行
func()
```

所以對 this 值來說，它根本不關心函式是在哪裡定義或是怎麼定義的，它只關心是誰呼叫了它。

在 JavaScript 中函式是一個很神奇的東西，它的確是一個物件類型，又不太像是一般的物件，以typeof 的回傳值來說，它回傳的是function，代表擁有獨立的回傳類型值。在函式物件的 API 定義中，它比一般物件多了幾個特別的屬性與方法，其中最特別的是以下這三個，我把它們的定義寫出來：

- call(呼叫): 以個別提供的this值與傳入參數值來呼叫函式。
- bind(綁定): 建立一個新的函式，這個新函式在呼叫時，會以提供的 this 值與一連串的傳入參數值來進行呼叫。
- apply(應用): 與 call 方法功能一樣，只是除了 this 值傳入外，另一個傳入參數值使用陣列。

那麼，這個call方法與直接使用一般的程式呼叫方式來執行函式，例如func()有何不同？

基本上完全一樣，除了它在參數裡可以傳入一個物件，讓你可以轉換函式原本的上下文(context)到新的物件。(註: Context 的說明在下面)

call方法可以把函式的定義與呼叫拆成兩件事來作，定義是定義，呼叫是呼叫。以下為一個範例：

```
function func(param1) {  
    console.log('func', this)  
}  
  
const objA = {  
    methodA() {  
        console.log('objA methodA', this)  
    },  
}  
  
const objB = { a: 1, b: 2 }  
  
func.call(objB) //func Object {a: 1, b: 2}  
objA.methodA.call(objB) //objA methodA Object {a: 1, b: 2}
```

這種現實讓你對函式的印象崩壞，不論是一般的func呼叫，或是位於物件objA中的方法methodA，使用了call方法後，竟然this值就會變成call中的第一個傳入參數值，也就是物件objB。

bind方法更是厲害，它會從原有的函式或方法定義，產生一個新的方法。為了展示它的厲害之處，函式加了兩個傳入參數，下面是函式部份：

```
function funcA(param1, param2) {  
    console.log(this, param1, param2)  
}  
  
const objB = { a: 1, b: 2 }  
  
funcA() //undefined undefined undefined  
  
const funcB = funcA.bind(objB, objB.a)  
  
funcB() //Object {a: 1, b: 2} 1 undefined  
funcB(objB.b) //Object {a: 1, b: 2} 1 2
```

這是物件中的方法定義的範例，這和上面沒什麼兩樣，只是函式定義在物件objA之中而已：

```
const objA = {  
    methodA(param1, param2) {  
        console.log('objA methodA', this, param1, param2)  
    },  
}
```

```

const objB = { a: 1, b: 2 }

objA.methodA()

const methodB = objA.methodA.bind(objB, objB.a)
methodB()
methodB(objB.b)

```

不過因為用了物件，應該要讓方法可以直接使用物件中的屬性才是妥善利用，另一種範例如下：

```

const objA = {
  a: 8,
  b: 7,
  methodA() {
    console.log(this, this.a, this.b)
  },
}

const objB = { a: 1, b: 2 }

objA.methodA() //Object {a: 8, b: 7} 8 7

const methodB = objA.methodA.bind(objB, objB.a)

methodB() //Object {a: 1, b: 2} 1 2
methodB(objB.b) //Object {a: 1, b: 2} 1 2

```

從上面的例子中，可以看到這個bind方法可以用原有的函式，產生一個稱為部份套用(Partially applied)的新函式，也就是對原有的函式的傳入參數值固定住部份傳入參數的值(從左邊開始算)。這是一種很特別的特性，有一些應用情況會用到它。

最後用下面這個例子來總結，什麼叫作"函式定義是定義，呼叫是呼叫"，實際上在物件定義的所謂方法，你可以把它當作只是讓程式設計師方便集中管理的函式定義而已。

```

const objA = { a: 1 }

const objB = {
  a: 10,
  methodB() {
    console.log(this)
  },
}

const funcA = objB.methodB

objB.methodB() //objB
funcA() //undefined，也就是全域物件window
objB.methodB.call(objA) //objA

```

註: function call 與 function invoke(invocation)是同意義字詞

this 值是何時產生的？

函式呼叫執行時產生。

當函式被呼叫(call/invoke)時，有個新物件會被建立，裡面會包含一些資訊，例如傳入的參數值是什麼、函式是如何被呼叫的、函式是被誰呼叫的等等。這個物件裡面有個主要的屬性 this 參照值，指向呼叫這個函式的物件。不同的函式被呼叫時，this 值就會不同。

this 值的產生規則是什麼？

this 值會遵守[ECMAScript 標準](#)中所定義的一些基本規則，我把它摘要如下，函式中的this值按順序一一檢視，只會符合其一種結果(if...else 語句):

1. 當使用 strict code(嚴格模式程式碼)時，直接設定為 call 方法裡面的 thisArg(this 參數值)。
2. 當 thisArg(this 參數值)是 null 或 undefined 時，會綁定為全域(global)物件。
3. 當 thisArg(this 參數值)的類型不是物件類型時，會綁定為轉為物件類型的值。
4. 都不是以上的情況時，綁定為 thisArg(this 參數值)。

第 1 點就明確的說明了，為什麼使用 strict mode(嚴格模式)後，在全域的函式呼叫執行，this 值一定都是 undefined，因為在 call 中根本沒傳入 thisArg 值。除非關閉 strict mode(嚴格模式)才會變為第 2 點的全域 window 物件。

Context(上下文) 是什麼？

Context這個字詞是不易理解的，在英文裡有上下文、環境的意思，什麼叫作"上下文"？這中文翻譯也是有看沒有懂。還記得在國高中英文課的時候，英文老師有說過，有些英文字詞的意思需要用"上下文"來推敲才知道它的意思，為什麼要這樣作？老師一定沒有把原因說得很清楚，第一個原因是英文單字你學得不夠多，很多時候考試試題中的英文單字通常你都沒讀到，所以只好猜猜看。第二個原因是，英文字詞很多時候同一個字詞有很多種意思，有時候用於動詞與名詞是兩碼子事，舉個例子來說，"book"這個英文單字，你用腳底板不需經過大腦，第一時間就會說它是"書"的意思，幼稚園就學過了，但是你忘了那是用於當作名詞的情況，用於動詞是"預訂"的意思。

在程式語言中的Context指的是物件的環境之中，也就是處於物件所能提供的資料組合中，這個 Context是由this值來提供。

再用白話一點的講法，來看函式與this的關係，this是魔法師的角色，而函式是要施展的魔法，魔法的強度或破壞力，會依施法者的資質與能力有所不同，魔法在施展中會運用到施法者的本身的資質(智慧、MP、熟練度...等等)的這整體的素質特性，這就是所謂的Context了。

註: Context與常會看到的另一個名詞Execution Context(執行上下文)的意義是不同的。

四種函式呼叫樣式(invocation pattern)

函式的呼叫樣式共有四種，在本書中已經看到過這四種了，這裡只是集中整理而已:

- 一般的函式呼叫(Function Invocation Pattern)
- 物件中的方法呼叫(Method Invocation Pattern)
- 建構函式呼叫(Constructor Invocation Pattern)
- 使用 apply, call, bind 方法呼叫(Apply invocation pattern 或 Indirect Invocation Pattern)

其中的建構函式呼叫，就是使用 new 運算符來進行物件實體化的一種函式呼叫樣式，請參考"物件"與"原型物件導向"的章節內容。

Scope(作用域) vs Context(上下文)

Scope(作用域, 作用範圍)指的是在函式中變數(常數)的可使用範圍，JavaScript 使用的是靜態或詞法的(lexical)作用域，意思是說作用域在函式定義時就已經決定了。JavaScript 中原本只有兩種的 Scope(作用域)，全域(global)與函式(function)，在 ES6 之後加入了區塊(block)作用域。

Context(上下文)指的是函式在被呼叫執行時，所處的物件環境。上面已經有很詳細的解說了。這兩個東西雖然都與函式有關，但是是不一樣概念的東西。

Scope 通常被稱為 Variable Scope(變數作用域)，意思是"作用域代表變數存取的範圍"。

Context 通常被稱為 this Context，意思是"由 this 值所代表的上下文"。

this 的分界

當函式被呼叫執行時，this 值隨之產生，那如果是函式中的函式呢？像下面這樣的巢狀或內部函式的結構：

```
const obj = { a: 1 }

function outer() {
  function inner() {
    console.log(this)
  }

  inner()
}

outer.call(obj) //undefined
```

結果是undefined，內部的inner函式不知道this值是什麼呢，為什麼？因為執行上下文(EC)是以函式呼叫作為區分，所以this值在不同的函式呼叫時，預設上就會不同。這稱之為this或Context的分界。

解決方式是要利用作用域鏈(Scope Chain)的設計，也就是說，雖然 inner 函式與外面的 outer 分屬不同函式，但 inner 函式具有存取得到 outer 函式的作用域的能力，所以可以用這樣的解決方法：

```
const obj = { a: 1 }

function outer() {
  //暫存outer的this值
  const that = this

  function inner() {
    console.log(that) //用作用域鏈讀取outer中的that值
  }

  inner()
}
```

```
outer.call(obj) //Object {a: 1}
```

`that`是一個隨你高興的變數(常數)名稱，這並不是什麼特殊的關鍵字或保留字，也有人喜歡取`self`或`_that`。它只是為了暫時保存在 `outer` 函式被呼叫時的`this`值用的，讓`this`可以傳遞到 `inner` 函式之中。

第二種寫法其實也是同樣的概念，只不過用了 `call` 來呼叫，`outer` 函式在呼叫時，它裡面是有`this`值的，因此可以當作 `call` 的傳入參數值，這範例與上面相同，也是有同樣作用：

```
const obj = { a: 1 }

function outer() {
  function inner() {
    console.log(this)
  }

  inner.call(this) //用outer中的this值來呼叫內部函式的inner
}

outer.call(obj) //Object {a: 1}
```

第三種寫法是用`bind`方法，不過因為`bind`方法會回傳新的函式，函式宣告要變成用函式表達式(FE)的方法才行：

```
const obj = { a: 1 }

function outer() {
  const inner = function() {
    console.log(this)
  }.bind(this)

  inner()
}

outer.call(obj) //Object {a: 1}
```

那麼在 `callback`(回調)的情況下又是如何？`this`能順利傳到 `callback`(回調)函式之中嗎？像下面的範例這樣，結果當然是不行：

```
const obj = { a: 1 }

function funcCb(x, cb) {
  cb(x)
}

const callback = function(x) {
  console.log(this)
```

```
}

funcCb.call(obj, 1, callback) //undefined
```

傳入參數值這個東西，如果是個"函式"的話，都是位於全域物件之下的，所以當這 callback(回調)在呼叫時的this值就是全域物件。這可以用 call 或 bind 方法來解決這個問題：

```
const obj = { a: 1 }

function funcCb(x, cb) {
  cb.call(this, x)
}

const callback = function() {
  console.log(this)
}

funcCb.call(obj, 1, callback) //Object {a: 1}
```

更進階的一種情況，使用例如像setTimeout方法，裡面帶有 callback(回調)函式的傳入參數，像下面這樣的程式碼：

```
const obj = { a: 1 }

function func() {
  setTimeout(function() {
    console.log(this)
  }, 2000)
}

func.call(obj) //window物件
```

這也是運用上面類似的幾種作法，其一，用一個函式內的變數(常數)來傳遞this值：

```
const obj = { a: 1 }

function func() {
  const that = this

  setTimeout(function() {
    console.log(that)
  }, 2000)
}

func.call(obj) //Object {a: 1}
```

其二，直接用 bind 方法(因為這裡不適合使用 call 方法)

```

const obj = { a: 1 }

function func() {
  setTimeout(
    function() {
      console.log(this)
    }.bind(this),
    2000
  )
}

func.call(obj)

```

或是寫得更清楚點，把其中的 callback 函式獨立出來：

```

const obj = { a: 1 }

function func() {
  function cb() {
    console.log(this)
  }

  setTimeout(cb.bind(this), 2000)
}

func.call(obj)

```

下面愈寫愈長，其實沒有那麼必要，只是說明也可以用另一個已經 bind 好的函式，傳入當作新的 callback(回調)函式：

```

function func() {
  function cb() {
    console.log(this)
  }

  const cb2 = cb.bind(this)

  setTimeout(cb2, 2000)
}

func.call(obj)

```

箭頭函式綁定 this

箭頭函式(Arrow Function)除了作為以函式表達式(FE)的方式來宣告函式之外，它還有一個特別的作用，即是可以作綁定(bind)this 值的功能。這特性在某些特定的應用情況下非常有用，同時也可以讓程式碼有更高

的可閱讀性，例如以上一節的例子中，有使用到 bind 方法的，都可以用箭頭函式來取代，以下為改寫過的範例，你可以比對一下：

```
const obj = { a: 1 }

function func() {
  setTimeout(() => {
    console.log(this)
  }, 2000)
}

func.call(obj)
```

在物件與陣列中函式的this

按照上面的說明，在一般情況，也就是在全作用域中定義中的函式，它的 this 值預設是在瀏覽器中是 window 物件，在 Node.js 中是 global 物件，也就是全域(全局)的物件，但要注意這只能適用在非嚴格模式(non-strict mode)的情況。

那如果是一般使用建構函式(或類別定義)所建立出來的物件實體，因為這是使用了new關鍵字進行實體化，符合了建構函式呼叫樣式(Constructor Invocation Pattern)，其中的成員的this值將自動指向新建立的物件實體。這在類別或物件的章節中都可以看到其中的內容，就不再多作描述。

但存放在陣列中的函式，實際上它也是像這樣的物件實體，因為在設計上陣列是個物件的類型。所以像下面這樣的例子：

```
const a = []
const func = function() {
  console.log(this)
}

a.push(func)
a[0]() //指向a
```

你會發現如果對在陣列中的成員函式呼叫，它的this值將指回陣列本身，也就是a。這是因為陣列的 const a = [] 的宣告，相當於 const a = new Array()，陣列是個物件類型，正確來說是陣列是以原始的物件類型為基礎，再發展延伸出來的特殊物件類型。

這將對應不論是使用物件字面方式來定義的物件方式，或是使用new Object()來實體化物件的定義方式，兩種獲得的結果都是相同的。如下面的例子：

```
const e = {
  func: function() {
    console.log(this)
  },
}

e.func() //指向e
```

```
const f = new Object()

f.func = function() {
    console.log(this)
}

f.func() //指向f
```

有一個特別的實例是函式中的隱藏物件 - `arguments`物件，它實際上也被隱藏了實體化的過程，由 JavaScript 自動實體化這個物件。因此，對應上面的陣列與物件的設計，預設裡面的成員如果是個函式也是自動指向自己本身。如下面的例子：

```
function func(fn) {
    arguments[0]() //指向arguments本身

    fn() //指向window(全域物件)
}

func(function() {
    console.log(this)
})
```

不過使用`arguments`物件容易產生誤解，這也是其中一個常見的陷阱，在許多實際使用的情況都容易造成誤用，例如上面的例子中，直接呼叫傳入的函式與使用 `arguments` 物件來呼叫，結果是不同的，`this`值是不同的。我會建議你不要使用`arguments`物件，它是一個有很多問題的設計。

特別注意：在 JavaScript 中，不要使用`new Array()`或`new Object()`來建立陣列與物件的實體（實例），原因主要是不需要而且容易被誤用。取而代之的是應該使用`[]`或`{}`的定義方式。

結語

本章說明了很多 JavaScript 底層實作的技術，不過只是簡單的介紹而已。`this`的概念因為涉及很多底層的設計，所以造成很多初學者非常容易搞混與誤解，相信在讀過這章的內容後，你可以對 JavaScript 語言中，`this`所扮演的角色，有更清楚或更深入的理解。在往後的開發日子中，可以更正確而且靈活的操控 `this`的各種用法。在以下的參考資料中有更多深入的內容等待你進一步去研究。

參考資料

- [What is the Execution Context & Stack in JavaScript?](#)
- [ECMA-262-3 in detail. Chapter 1. Execution Contexts](#)
- [How to access the correct this / context inside a callback?](#)
- [How does the “this” keyword work?](#)
- [JavaScript function invocation and this \(with examples\)](#)

Event(事件處理)

JavaScript 是一個以事件驅動(Event-driven)的程式語言。事件驅動程式設計的主要流程，是由圖形化使用者操作介面(UI)的互動事件為主要核心，藉由事件的觸發動作(滑鼠點按、鍵盤輸入等等)或是感應器的訊息，來啟動整體的程式流程。

依據"異步程式設計與事件迴圈"一章的內容中的說明，在 JavaScript 中有一個不斷偵測事件發生的事件迴圈(Event Loop)，所有在網頁上的 DOM 元素註冊的事件，都會進入一個佇列(queue)中，等待被觸發事件，然後作出相對的回應送還給使用者。負責實作事件迴圈的是瀏覽器環境，佇列有可能不只一個，一般認為至少會依照 W3C 所定義的各種不同會發生佇列情況，也就是有 5 種佇列，包含事件、回調、使用外部資源等等。

我們在這裡所稱的事件(Event)，通常我們把它稱為瀏覽器事件(browser events)，這些是網頁中的 DOM(Document Object Model)元素的事件，標準制定者是 W3C 組織，然後由 ECMAScript 負責支援的角色。也有一些特定的事件是由瀏覽器品牌自行制定，這不在我們討論的範圍之中。

近年來實現了伺服器端的 JavaScript 語言執行環境的 Node.js，它並沒有這一系列直接可以使用的事件，內建的事件模型是 Node.js 自行設計的，不過它採用了相當類似的樣式，使用一個EventEmitter 類別的物件實體來作事件處理，在底層也使用事件迴圈(Event Loop)的設計來達成。不過，你如果要進入伺服器端的事件處理領域，建議先從熟悉瀏覽器端的事件先著手，你會發覺它們有很多相似的設計與樣式。

DOM 事件

DOM(Document Object Model, 文件物件模型)是由 W3C 組織所制定的跨平台與跨程式語言的應用程式介面，它是把 HTML、XHTML、XML 文件的視為樹狀的結構，每個節點都是代表文件的一個物件。DOM 的標準大戰可以從第一次瀏覽器大戰，由 Netscape 與微軟 IE 開始算起，到 2015 年已經是第 4 級(Level 4)的標準。

DOM 事件則是可以註冊各種事件處理器/監聽器(event handlers/listeners)在 DOM 的節點元素上，在 DOM 標準的第 2 級(Level 2)時，W3C 標準化了 DOM 中的事件模型，統一不同瀏覽器中的事件模型差異。這個標準也就是我們現在在瀏覽器上使用的事件模型基準。

JavaScript 程式語言一直以來對瀏覽器中的物件模型(object model)與其事件模型，有非常高的支援性，雖然 DOM 標準從來就不是只專門制定給 JavaScript 單一種語言使用的，不過因為 JavaScript 的使用群太高了，現在也差不多就是制定給它用的。

那麼，什麼樣的 DOM 元素中會對應什麼樣的 DOM 事件，或是什麼樣的操作方式會觸發怎麼樣的事件？

這都由 DOM 中的標準來制定，例如像這個[DOM 事件列表](#)非常的長，從滑鼠、鍵盤輸入到 HTML 表單... 等等，近年來由於觸碰式的行動裝置很流行，W3C 也開始制定[觸碰事件\(Touch Events\)](#)的標準。不過，有幾個瀏覽器品牌自己實作了不少非標準的事件，有些事件是只用於該瀏覽器的特殊用途。

註: [DOM events \(維基百科\)](#)

Event 物件(介面)

JavaScript 中定義了 Event 物件，其中包含了事件通用的屬性與方法，例如事件的對象元素等等，事件的來源是來自 DOM，所以這個物件通常稱之為 Event 介面。W3C 制定了標準的 Event 介面規格。其中有幾

個常用的屬性與方法分列如下。

Event 物件屬性(以下屬性都是只能讀不能寫):

- currentTarget: 目前的事件對象
- target: 分派事件的原始對象
- type: 事件的類型，共有數十種
- bubbles: 冒泡狀態，布林值，true 代表會在 DOM 中往上冒泡
- cancelable: 事件是否為可取消的，布林值

註: 目前的事件對象(currentTarget)與分派事件的原始對象(target)，在冒泡與捕捉階段，其中的值會不同。

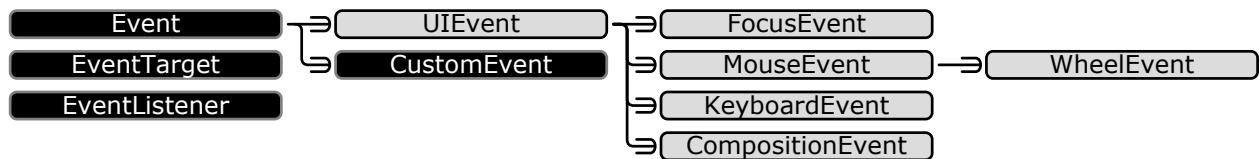
Event 物件方法:

- preventDefault(): 取消事件的行為(需為可取消的事件)，但無法阻止事件的傳播(propagation)
- stopPropagation(): 停止事件的傳播行為

註:事件的傳播行為在下面的章節有再加說明。

Event 介面只是個基礎物件，從它擴充了使用於特定情況的事件，包含對特定事件的資訊，詳見以下的階層圖(出自[這個網站](#)):

UI Events Interface Inheritance



依照事件階層圖中，**UIEvent** 與 **CustomEvent**(自訂事件)繼承自 **Event** 物件，從 **UIEvent** 中又擴充出各種不同的對應事件，例如針對滑鼠與鍵盤的，**FocusEvent** 是設計給 Focus(鎖定、聚焦)事件使用的，**CompositionEvent** 是針對輸入文字事件使用的，這是因為輸入文字並不一定單純使用鍵盤(用輸入法輸入或用語音輸入等等)，它與鍵盤事件也可以相互輔助。而 **WheelEvent** 是有滾輪的設備使用的。

註: 許多外部函式庫例如 **jQuery**，對於 **Event** 物件會以 W3C 的標準進行擴充。

EventTarget 物件(介面)

EventTarget 物件則是 JavaScript 所設計的一種當作介面的物件，它可以接收事件，以及讓監聽者註冊到上面。DOM 元素、document、window 物件，是最常見的 **EventTarget** 物件，另外也有其他的物件可作為 **EventTarget**。**EventTarget** 物件中有三個方法:

- **addEventListener**: 在事件對象上加入事件監聽者
- **removeEventListener**: 從事件對象移除事件監聽者
- **dispatchEvent**: 送出事件給所有有訂閱的監聽者

另外需要提到的一點，W3C 標準中對於**EventListener**也有定義它是一個介面，作為事件監聽者之用，不過 JavaScript 語言在所有的函式中都有實作這個介面，所以事件監聽者在呼叫**handleEvent**(處理事件)方法時，相當於呼叫函式。**EventListener**(事件監聽者)或稱為事件處理函式，可以自動得到事件傳入參數值，以此可以存取得到事件的屬性與方法，例如以下的範例:

```

const me = document.getElementById('me')

me.addEventListener(
  'click',
  function(e) {
    console.log(e.currentTarget)
    console.log(e.target)
    console.log(e.type)
    console.log(e.bubbles)
    console.log(e.cancelable)
    e.stopPropagation()
  },
  false
)

```

事件處理模型

現行的事件處理模型通常會使用"監聽"(listen)的方式，作為事件處理的標準樣式，因為 DOM 標準制定歷史版本不同，實際上有好幾種不同的方法可以作事件處理。以下分述這幾級的差異，其中第一種與第二種是舊的方式，不建議使用。

內聯模型

這種方式是最簡單的，也稱為內聯模型(Inline model)。它直接在 HTML 裡 DOM 元素中標記中使用，每個元素都會實作對應的可使用事件，名稱都會是像"onxxxxx"這樣的全小寫字詞，例如按鈕會實作 onclick 的事件屬性(attribute)，就在這裡面寫上事件處理的程式碼:

```
<button onclick="console.log('hello!');"> Say Hello! </button>
```

JavaScript 引擎中會產生一個對應的匿名函式，包含在 onclick 中的語句。這個方式也是最不建議的方式，它完全不像個用 JavaScript 語言寫的應用程式，也因為需要直接寫在 HTML 中，完全沒有彈性可言。

傳統模型

傳統模型(Traditional model)方式，提供了分離 HTML 與 JavaScript 程式碼的語法，它比之前內聯模型的方式好得多了。不過它依然有個大問題，就是它只能在一個元素上使用一個事件，所以也不建議使用。

```
document.getElementById('myButton').onclick = function() {
  console.log('hello!')
}
```

DOM Level 2

這個方式又被稱為 W3C 方式(W3C Way)模型，這個方式才能說它是用事件監聽(Event Listen)的方式，使用 callback(回調)函式，作為事件的監聽者(或稱之為事件處理函式)。

```
const el = document.getElementById('myButton')

el.addEventListener(
  'click',
  function() {
    console.log('hello!')
  },
  false
)
```

事件監聽的方式可以對一個元件附加多個事件處理函式，而且以標準來說基本有定義三種方法可使用：

- `addEventListener`: 在事件對象上加入事件監聽者
- `removeEventListener`: 從事件對象移除事件監聽者
- `dispatchEvent`: 送出事件給所有有訂閱的監聽者

註：微軟 IE 瀏覽器在舊版本中使用自己定義的事件處理方法，所以要與舊版本相容時需要特別注意。IE9 之後就能使用上述的事件監聽方式。

事件觸發的順序

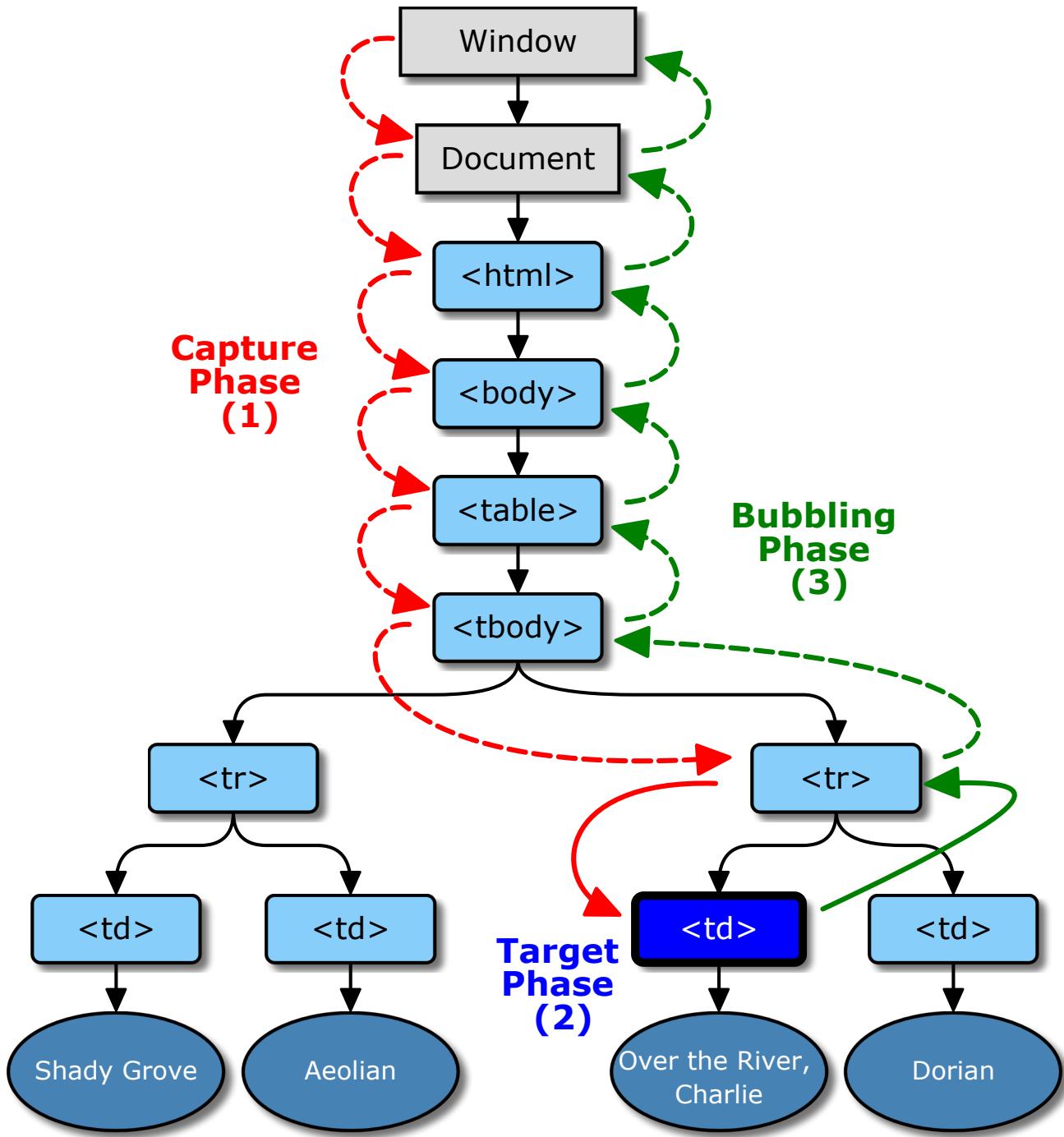
在同一個元素上註冊多個不同的事件監聽者(事件處理函式)，它的在事件觸發時的執行順序是怎麼樣子的？是按照程式碼所寫的順序嗎？

由於 W3C 的 DOM 第 2 代標準中並沒有明確的說明順序這件事，所以當如果有多个事件處理函式在同一元素中註冊時，它們的執行順序將會由瀏覽器來決定，有些舊的瀏覽器並不是按照程式碼中的順序執行。

在 W3C 的 DOM 第 3 代標準也已經明定順序由註冊到事件目標的監聽者順序決定，而且現在瀏覽器各種不同品牌與新版本，都會依照程式碼中的順序為執行的順序。

事件的冒泡、捕捉

事件的冒泡(往上冒泡)與捕捉(往下捕捉)是兩種事件在 DOM 中的傳播(propagation)的方式。這是由於 DOM 元素的樹狀結構，它是有父母-子女關係(parent-child)，這兩種傳播會在事件監聽時，形成一種特別的事件傳播模型，影響事件監聽者在不同父母-子女關係(parent-child)時的執行順序。例如下面的圖解(出自[這個網站](#))：



那為什麼會出現兩種相反的事件傳播方式？這是因為在第一次瀏覽器大戰時，Netscape 採用了事件捕捉(capturing)，而微軟採用了事件冒泡(bubbling)，現行的 W3C 標準則一併使用了兩者，目前的瀏覽器品牌中都有支援兩者，而微軟從 IE9 之後也支援兩者。事件捕捉(capturing)與事件冒泡(bubbling)並沒有說哪一種就比較好，這純粹是看程式設計師的需求而定，不過要特別注意的是，有些幾個事件並沒有支援事件的傳播，例如 `onfocus` 或 `onblur`。

那麼要如何控制使用哪一種？一般都是使用事件監聽的方法，以傳入參數值作控制，也就是 `addEventListener` 方法的最後一個參數 `phase`(階段)來決定。`addEventListener` 的語法如下：

| `addEventListener(type, handler, phase)`

`phase`(階段)是一個布林值，如果是 `false` 就用事件冒泡(bubbling)，如果是 `true` 就使用事件捕捉(capturing)。預設沒寫的話，就是 `false`，也就是預設使用事件冒泡(bubbling)機制。

| 註：記法有很多種，自行想像發揮力。例如 `fb(false = bubbling)`。`true` 與 `capture` 都有 "t"。抓兔子(捕捉=`true`)，廢砲(`false`=冒泡)，浴室才會有泡泡(預設使用泡泡)。

事件冒泡(bubbling)的情況時，當最內部的元素被觸發事件時，會先執行自己本身的事件處理函式，然後才會執行上層父元素的事件處理函式。以下為範例：

```
const parent = document.getElementById('parent')
const child = document.getElementById('child')

parent.addEventListener(
  'click',
  function() {
    console.log('parent clicked')
  },
  false
)

child.addEventListener(
  'click',
  function() {
    console.log('child clicked')
  },
  false
)
```

HTML 中的程式碼如下：

```
<div id="parent">
  click me(parent)
  <div id="child">
    click me(child)
  </div>
</div>
```

事件捕捉(capturing)則是倒過來的情況，當最內部的元素被觸發事件時，會先從最外圍的事件處理函式執行，依序到最後才是執行自己本身的處理函式。以下為範例：

```
const parent = document.getElementById('parent')
const child = document.getElementById('child')

parent.addEventListener(
  'click',
  function() {
    console.log('parent clicked')
  },
  true
)

child.addEventListener(
  'click',
  function() {
    console.log('child clicked')
  }
)
```

```
},
true
)
```

HTML 中的程式碼如下：

```
<div id="parent">
  click me(parent)
  <div id="child">
    click me(child)
  </div>
</div>
```

不論事件的傳播方式為何種，Event 物件提供了`stopPropagation()`方法，可以阻止事件的傳播，這可以在某些應用情況中使用。不過這個`stopPropagation`方法在事件冒泡(bubbling)與事件捕捉(capturing)兩種不同情況下使用時要特別小心，以免連元素自己的事件處理函式都被擋住。例如像下面的例子：

```
const taiwan = document.getElementById('taiwan')
const taipei = document.getElementById('taipei')
const me = document.getElementById('me')

taiwan.addEventListener(
  'click',
  function() {
    console.log('taiwan clicked')
  },
  false
)

taipei.addEventListener(
  'click',
  function() {
    console.log('taipei clicked')
  },
  false
)

me.addEventListener(
  'click',
  function() {
    console.log('me clicked')
  },
  false
)
```

index.html 上的 HTML 程式碼如下，為了明顯標出每個區域，加了顏色與大小的樣式：

```
<div id="taiwan" style="border:1px solid green; height:300px; width:300px">
    Click Taiwan
    <div id="taipei" style="border:1px solid blue; height:200px; width:200px">
        Click Taipei
        <div id="me" style="border:1px solid red; height:100px; width:100px">C
    </div>
</div>
```

taiwan是最外圍的 DOM 元素，然後裡面含有taipei層，最裡面是me這一層。以下是點按 me 層元素的結果：

```
//事件冒泡(bubbling)，全部false參數值的結果是
me -> taipei -> taiwan
```

```
//事件捕捉(capturing)，全部為true參數值的結果是
taiwan -> taipei -> me
```

假使在第 2 層中，也就是 taipei 層加上stopPropagation方法，阻止事件傳播，如以下的程式碼：

```
taipei.addEventListener(
    'click',
    function(e) {
        console.log('taipei clicked')
        e.stopPropagation()
    },
    false
)
```

你可以再比對一次點按 me 層元素的結果如下：

```
//事件冒泡(bubbling)，全部false參數值的結果是
me -> taipei (stop!)
```

```
//事件捕捉(capturing)，全部為true參數值的結果是
taiwan -> taipei (stop!)
```

也就是說只要經過有阻止傳播的層，就會不再有事件處理函式被觸發，不論這個事件處理函式是不是已經被註冊為該層的事件處理函式，這種特性需要特別小心使用。

另外，混用事件冒泡(bubbling)與事件捕捉(capturing)兩種在程式碼中，絕對是個不智之舉，在簡單的 DOM 結構時，可能可以推測出來事件處理函式的順序，但在真實情況，DOM 的結構的複雜程度是超過你所能想像的，千萬不要這麼作。如果你不確定該使用哪一種方式，使用預設的事件冒泡(bubbling)方式，也就是使用false值就行了。

註: 事件捕捉(capturing)也有人稱它為事件滴流(trickling)，滴流是向下流動的意思。總之就是向上冒泡的反義詞。

this與event.target、event.currentTarget

在 W3C 標準的定義中，this會相等於event.currentTarget，也就是說this永遠會指向目前的事件目標對象，就像地震的傳播一樣，地震中心點先震完了，開始傳播到別的地區，event.currentTarget會跟著改變。事件監聽者(事件處理函式)是一個 callback(回調)函式，但這個行為與一般的 callback(回調)函式不同，一般的 callback(回調)函式的this值會總是指向全域的 window 物件。

但不管是事件冒泡(bubbling)或事件捕捉(capturing)，event.target永遠指向觸發事件的那個元素，也就是地震的發生源。以下有一張事件冒泡(bubbling)的解說圖片，你可以看一下event.target與this(也就是event.currentTarget)的比較。(來自[這個網站](#)):



自訂事件

自訂事件是由開發者自己建立所需要的事件，再來是附加到某個 DOM 元素上監聽，最後也是要由開發者自行觸發。會這樣作的原因是，DOM 元素上的事件早就被定義固定了，按鈕有按扭可用的事件，表單中的元素有自己的事件。那如果在開發者自己作的應用程式中，想要依照程式中的某個執行功能，定義自訂的事件要怎麼作？例如會員登入時想要用個會員登入事件，聊天室程式常見的好友上線時的好友上線事件，這就是自訂事件的功用了。

首先我們會先使用 CustomEvent 這個介面(物件)來定義自訂的事件物件，它只需要設定一些屬性，這些屬性與 Event 中無異，唯一的差異是在於它可以使用detail屬性，來額外新加入自訂事件的屬性值，下面是範例:

```
const myEvent = new CustomEvent('userLogin', {
  detail: {
    message: 'Hello World!',
    time: new Date(),
  },
  bubbles: true,
  cancelable: true,
})
```

'userLogin'是這個自訂事件的名稱，detail物件中的屬性是額外的屬性。bubbles與cancelable是可以定義也可以不需要定義的屬性，沒定義會使用預設值false，CustomEvent 介面(物件)的屬性是繼承自Event(介面)物件而來，所以會有這兩個屬性。bubbles值代表可否使用冒泡機制，cancelable則是代表可否使用stopPropagation()方法。

接下來在你想要監聽這個事件的元素上，加入這個事件與對應的事件處理函式:

```
const myButton = document.getElementById('myButton')

myButton.addEventListener('userLogin', function(e) {
  console.log('Event is: ', e)
```

```
        console.log('Custom data is: ', e.detail)
    })
}
```

要觸發這個自訂事件，你只能手動地在程式碼中觸發，使用下面的程式碼：

```
myButton.dispatchEvent(myEvent)
```

自訂事件在 IE 系統的瀏覽器沒辦法直接使用，有相容性的問題，在其他的瀏覽器上則不會有。IE9 之後的瀏覽器可以使用這裡的[Polyfill\(填充\)](#)程式碼或是外部函式庫(如 jQuery)來擴充這個功能。

參考資料

- [UI Events\(W3C\)](#)
- [Document Object Model \(DOM\) Level 2 Events Specification\(W3C\)](#)
- [HTML5 - 6.1.5 Events\(W3C\)](#)
- [Event developer guide\(MDN\)](#)
- [Advanced event registration models](#)
- [Event order](#)
- [Bubbling and capturing](#)
- [A crash course in how DOM events work](#)
- [The Order of Multiple Event Listeners](#)
- [Unable to understand useCapture attribute in addEventListener](#)
- [How to Create Custom Events in JavaScript](#)

異步程式設計與 Eventloop(事件迴圈)

JavaScript 程式語言在設計時，需考量異步、單執行緒與非阻塞 I/O 等等的問題

JavaScript 程式執行的確都在單一個執行緒(Single Thread)中的。聽起來有點不可思議，現在的電腦硬體不都是多核心(多執行緒)而且資源豐富嗎？這個設計對於程式執行不會有問題嗎？

我們用另外的角度來思考，為何 JavaScript 會這樣設計，仍然可以符合程式執行的需求的幾個原因：

- JavaScript 從一開始就是這樣設計，它執行的主要環境是在瀏覽器上，只有一個使用者，而且是在資源受限的環境中執行，這是一個很合理的設計。
- JavaScript 程式執行雖然是單執行緒，但伺服器或瀏覽器執行環境並不是：表面上看起來是只有一個執行緒在執行 JavaScript 程式，但實際上在背後有數個的其他在執行環境中的執行緒，在輔助程式碼的執行。
- 外部資料的執行時間，大部份都是等待時間：像連接資料庫、執行資料庫查詢等等的執行語句，真正執行是在資料庫裡，程式只是傳送對應的查詢語句而已，並不是在 JavaScript 程式中執行查詢，這種情況對 JavaScript 程式而言，大部份的執行時間中都是在等待查詢結果而已。讀寫檔案、網路連線要求與回應、傳送資料等等，都有類似的情況。大量而且複雜的運算，的確是在語言的程式中執行，不過 JavaScript 原本就不是設計用來作複雜運算用的，或許要在特殊的執行環境中才有辦法作這件事。

要理解 JavaScript 是如何執行程式，首先要先理解同步(Synchronous, sync)與異步(Asynchronous, async)程式執行的差異。

由於 JavaScript 中的執行區分是以執行上下文(EC)作為一個單位，也就是以函式區分，所以一般要討論異步或同步執行，也通常會用異步執行的函式或同步執行的函式來區分。程式碼中的每個語句一定是同步執行的。而異步執行函式必定會使用 CPS 風格的異步回調函式，這是 JavaScript 中的設計，關於異步回調函式，可以參考特性篇的"Callback 回調"章節的內容。

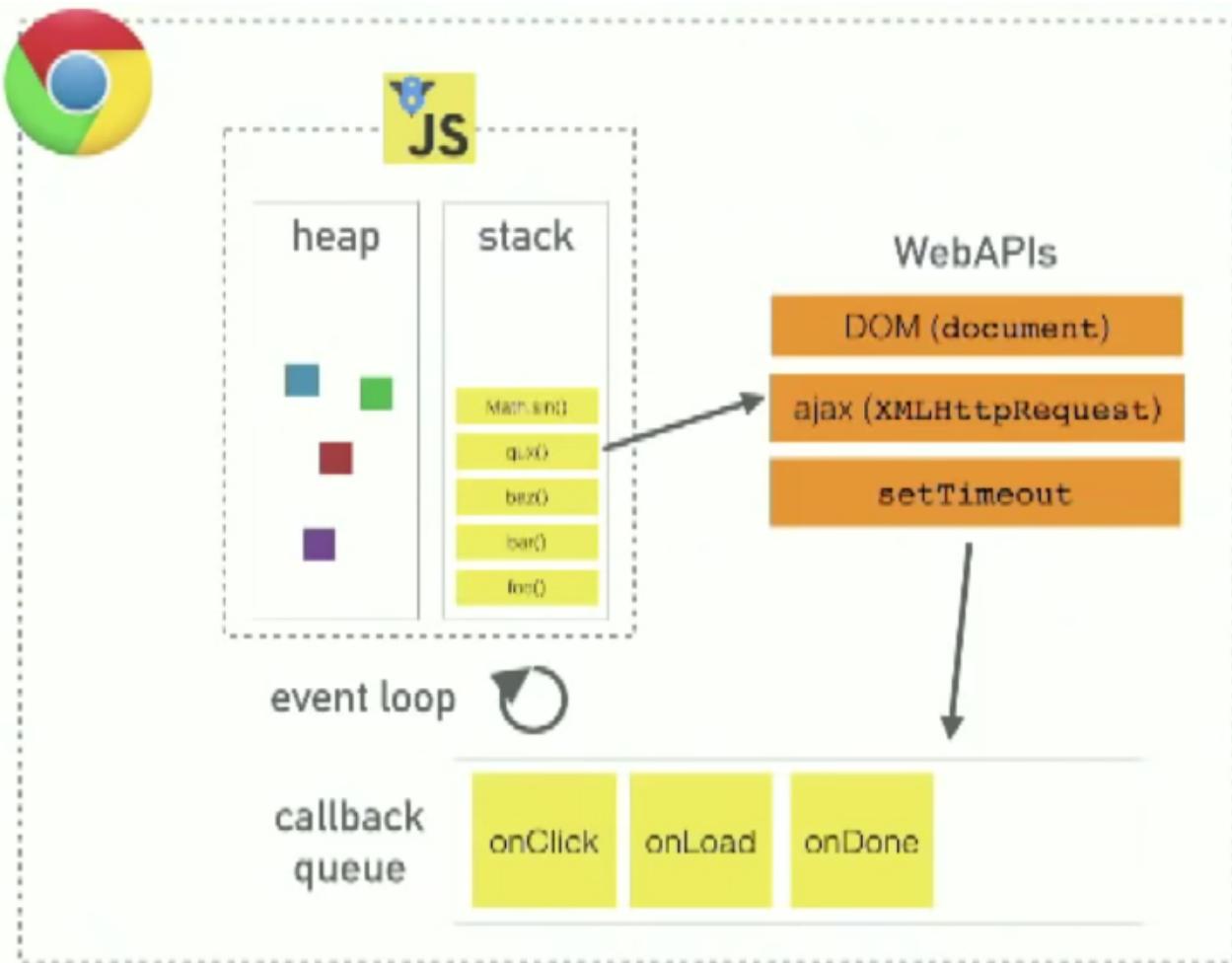
同步程序執行是指程式碼的執行順序，都是由上往下依順序執行，一個執行程序完成後才會再接著下一個，一般的程式語言都是按照這樣的流程來執行，JavaScript 語言也不例外。例如像連接資料庫存取資料的程式，應該會遵守下列的步驟進行：

1. 連接資料庫(給定帳號、密碼、主機、資料庫名)
2. 執行資料庫查詢語句
3. 取得資料，格式化資料

這對於"從資料庫查詢資料"的這種程式本身並沒有太特別的地方，一般都是這樣執行沒錯。但對於 JavaScript 這種只有單執行緒的程式語言，這樣作會造成阻塞(blocking)，也就是說當這個資料庫查詢的執行程序，需要很長的一段時間才能結束時，在這期間其他的操作都會停擺，像是滑鼠要點按按鈕之類的功能，就完全沒有作用。因此，我們需要用另一種不同的方式，來進行這類會阻塞其他程式的執行，也就是異步程式執行的方式。

異步程式執行的作法，是使用異步 callback(回調)函式的語法，讓會造成阻塞的程式組成一個異步回調函式，先丟往一個任務佇列(task queue)先丟，在之後的某個時間再回傳它的值與狀態回來。這些函式裡的程式碼多半都是與外部資源存取的 I/O 有關，如果需要等待的話，是在實作的 API 裡(外部模組)，並不是在佇列或語言執行緒中，超時或有回應後再加入到佇列中，事件迴圈在主執行緒完全沒有其他的 EC 時，再加

回到主執行緒中執行。下面這張圖是出自影片[Philip Roberts: What the heck is the event loop anyway?](#) | JSConf EU 2014。



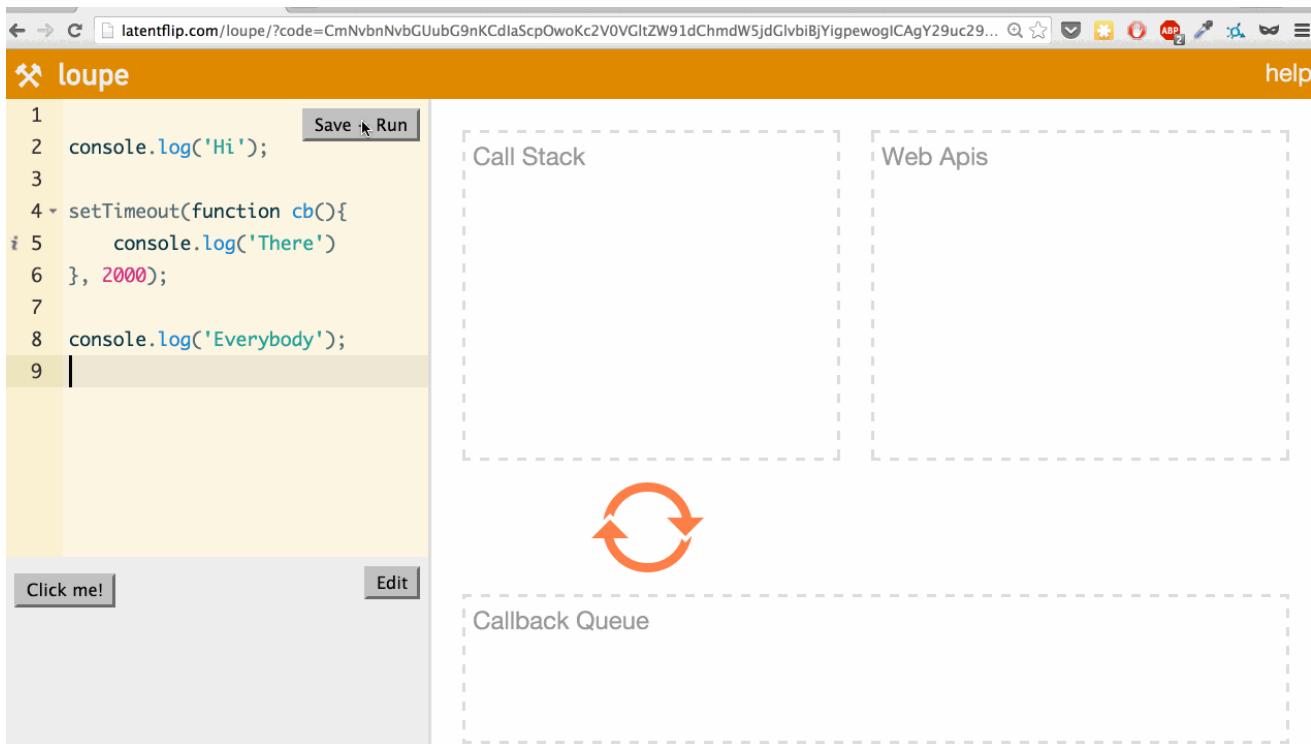
最常使用的例子是用`setTimeout`這個內建的方法，它會在某個設定的時間的執行其中的 callback(回調)函式傳入值一次：

```
console.log('a')

setTimeout(function cb() {
  console.log('b')
}, 1000)

console.log('c')
```

按照同步程式的執行順序，應該是`a -> b -> c`這個結果，但真正的結果是`a -> c -> b`，也就是`cb`這個在`setTimeout`中的 callback(回調)函式，在程式執行到這一行時，先被移出主執行緒外面，先到任務佇列去了，最後等主執行緒空了，在某個時間才會再回到主執行緒中，執行其中的輸出值的動作。下面是類似的程式的執行模擬圖解，來自[JavaScript's Call Stack, Callback Queue, and Event Loop](#)。如果你覺得圖解不夠，你可以直接到這個[loupe](#)網站來執行看看。



註: 請不要誤解了，並不是所有的 callback(回調)函式都是會丟到任務佇列(task queue)之中執行。只有經過特殊設計過的異步 callback(回調)才會這樣作。

另一個最常見的例子是 AJAX 技術的實作，AJAX 的全名是"Asynchronous JavaScript and XML"，它在名稱上就有異步的字詞，是運用 XMLHttpRequest 物件與網站伺服器溝通的一種技術，我們在特性篇有一篇專文介紹它。一個簡單範例如下：

```

const xhr = new XMLHttpRequest()

xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    console.log(xhr.responseText)
  }
}

xhr.open('GET', 'test.txt', true)
xhr.send()

```

AJAX 技術可以不需要刷新瀏覽器的頁面，它是一種在瀏覽器背後模擬與伺服器要求與回應溝通的機制，因為是與外部環境作溝通，有可能會因為網路連線或伺服器的狀況造成等待時間，所以一開始就被設計為異步的 API，也就是說當 AJAX 執行時，onreadystatechange 屬性中的這個回調函式，也會先往一個任務佇列(task queue)丟去，之後等主執行緒清空後，在某個時間點再回來回傳回應的值。

註: 任務佇列(task queue)也有其他名稱的講法，例如消息佇列(message queue)、事件佇列(event queue)、回調佇列(callback queue)

異步程式設計中，JavaScript 使用 Event Loop(事件迴圈)的設計來協助達成異步函式的執行，它是一種並行Concurrency的模型，事件迴圈可以想成是一個內部迴圈功能，它會不斷地每一段時間就檢查佇列與執行程序，然後決定是否要把佇列中的任務程序，移回目前 JavaScript 程式的主要執行緒中(呼叫堆疊)執行，原則其實簡單，"當只有在呼叫堆疊空空如也時，才會把佇列中任務移回呼叫堆疊"。

你或許會認為 JavaScript 因為只有單一條執行緒的並行(Concurrency)模型，根本就不是"同時執行"的，同一時間還是只能作一件事的確是個事實，不過單執行緒那也只能這樣設計。單只有一條執行緒的並行模型，並非完全沒有任何的優點，相較於多執行緒的設計複雜，它會比較簡單，而且以作同樣的事情(例如服務同樣多的使用者連線)時，消耗的資源會比較少。

Event Loop(事件迴圈)直接由名稱上理解，主要是為了事件(Events)所設計的，存放有被進行分派的事件函式程序到任務佇列中，內部的迴圈功能會不斷的重覆檢查，目前現在瀏覽器上的各種 HTML 元素是不是有被觸發事件，當有事件被觸發時，就立即把函式，先移到 JavaScript 的呼叫堆疊中來執行，當然它也是一種異步的回調函式，所以也要先移往任務佇列中，等其他在呼叫堆疊中的程式都執行完了，才會回到主執行緒來執行。

那麼，什麼樣的執行程序(函式)會被移到任務佇列之中？它的順序又是如何的？

依照[W3C 的定義](#)，Event Loop(事件迴圈)中可能會有一個以上的任務佇列(task queue)，一般認為就是用以下幾種分出不同的佇列，但實作部份要視瀏覽器實作決定，其中的順序是依照 FIFO(先進先出)，以下是幾種會包含的任務：

- Events(事件): EventTarget 物件異步分派到對應的 Events 物件
- Parsing(解析): HTML parser
- Callbacks(回調): 呼叫異步回調函式
- 使用外部資源: 資料庫、檔案、Web I/O
- DOM 處理的反應: 回應 DOM 處理時的元素對應事件

註: 對照 Call Stack(呼叫堆疊)是 FILO(先進後出)或 LIFO(後進先出)的順序。

在瀏覽器端的 JavaScript 程式語言中，除了一般的事件分派外，還有少數幾個內建的 API 與相關物件有類似的異步機制，有一些簡單的樣式可以利用它們模擬出異步的執行程式：

- setTimeout
- setInterval
- XMLHttpRequest
- requestAnimationFrame
- WebSocket
- Worker
- 某些 HTML5 API，例如 File API、Web Database API
- 有使用 onload 的 API

而在伺服器端(Node.js)的 JavaScript 程式，大部份的 API 都會考量到異步的問題，尤其是與 I/O 相關的，是半點都不能夠有阻塞的情況發生，這稱為非阻塞 I/O(Non-blocking I/O)的設計方式，都有對應的異步呼叫方式。

註: 有個說法是說"JavaScript 是有非阻塞 I/O 特性的程式語言"，比較好的理解應該是"JavaScript 是個沒辦法阻塞住 I/O 的程式語言"，畢竟只有一條執行緒，一但塞住了就會無法正常運作。因此"非阻塞 I/O(Non-blocking I/O)"才會成為它的一種特性。

不過，對於異步程序(函式)也有一些問題要考量：

- 異步程序(函式)間沒辦法保証執行的時間順序：在複雜的多個的異步程序(函式)，可能有很多存在於任務佇列的等待被執行的程序，也可能有一個以上的任務佇列，它們的執行時間與順序的沒有辦法保証。

- Run-to-completion(一執行就要執行到完成): 每個任務程序(函式)中的程式語句都會只要一執行就會到完成，所以基本上任務程序(函式)中都是同步執行的程序，一個完成才會接著下一個任務。這個特性有可能會導致過長時間的任務，阻塞到 Event Loop(事件迴圈)的進行，建議是把任務切割成更小的任務。

因此，異步執行程式並非只有單純的幾個函式呼叫這麼簡單，有很多情況是需要整個程式的執行流程一併考慮的，例如上面的資料庫查詢的例子，如果後面還有一些要把查詢到的資料進行其他運算的程式碼，就會需要進行執行流程上的分離或合併(例如異步合併同步、同步中的異步、異步中的同步等等)。

異步的程式流程的組織方式，現在也有好幾種作法:

- Promise 語法結構(ES6)
- Generators (ES6)
- 使用工具函式庫，例如[Async](#)
- Async 函式(ES7)

常見問答

異步函式執行比同步函式執行快？

沒有。一定比較慢。

事件迴圈在呼叫堆疊與任務佇列切換要加入的異步函式，是需要一定時間的。你可以把異步執行的函式，視為一種"暫緩執行"的函式，既然是暫緩執行，一定不會比直接在呼叫堆疊中的執行的函式快。

JavaScript 沒有辦法使用多執行緒之類的作法嗎？

現在有一些新的技術，可以使用到其他的執行緒，例如:

- Web Workers
- 伺服器端(Node.js)用的 child_processes 模組與 cluster 模組

異步執行函式裡面如果還有異步執行的其他函式，這樣的執行有順序規則可言嗎？

有。不過執行順序還是要視情況決定。

主執行緒的執行規則是同步規則，所以是一行接一行，先放到呼叫堆疊中。呼叫堆疊在執行時，是先進後出(FIFO)的規則。

如果呼叫堆疊中看到異步的函式，會先移到任務佇列中，任務佇列中等事件迴圈看到呼叫堆疊沒有其他函式 EC(執行上下文後)後才會把佇列中的 EC 移回主執行緒中。這個移回去的順序是依照先進先出(FIFO)的規則。

異步中的異步，上面的流程會再重新作一遍，不斷循環直到所有程式碼都執行完畢。所以以幾個情況來說，如果假設都是相同延時(例如都是 1000ms 延時執行)的異步函式。

1. 相同的異步執行函式，看誰在全域 EC 中(也就是程式碼中)先被執行，誰就先完成執行
2. 異步只要差一層，在全域 EC 中(也就是程式碼中的)執行順序就會無關，愈多層的一定比較慢完成

下面的範例的輸出結果必定是a->b，因為setTimeout中的時間代表加入到任務佇列的時間，只要加入佇列的時間一樣，就會依程式碼從上到下執行的順序為順序。

```

function aFunc(value, cb) {
  setTimeout(cb, 1000, value)
}

function bFunc(value, cb) {
  setTimeout(cb, 1000, value)
}

aFunc('a', function cbA(value) {
  console.log(value)
})
bFunc('b', function cbB(value) {
  console.log(value)
})

```

但上面都是同樣的異步執行函式的規則，如果是有一點點的時間差，結果會變為b->a，像下面的範例：

```

function aFunc(value, cb) {
  setTimeout(cb, 1000, value)
}

function bFunc(value, cb) {
  setTimeout(cb, 900, value)
}

function cbA(value) {
  console.log(value)
}

function cbB(value) {
  console.log(value)
}

aFunc('a', cbA)
bFunc('b', cbB)

```

時間差不只對同樣層的異步執行函式有用，對多層的情況也會影響，因為時間差代表的是異步回調函式加到佇列的時間，如果這個時間晚於前一個多層異步函式加入又移回執行，然後又加入移回執行，那就只能比之前的慢。下面的範例說明了這點。

```

function aFunc(value, cb) {
  setTimeout(cb, 1000, value)
}

function bFunc(value, cb) {
  setTimeout(cb, 0, value)
}

function inCbB(value) {

```

```

        console.log(value)
    }

    function cbB(value) {
        setTimeout(inCbB, 0, value)
    }

    function cbA(value) {
        console.log(value)
    }

aFunc('a', cbA)
bFunc('b', cbB)

```

雖然aFunc中的異步執行函式cbA只有 1 秒的時間差才加到任務佇列中，但bFunc中的cbB會搶先加到任務佇列，然後回到呼叫堆疊中執行，inCbA再搶先加到任務佇列中，先執行完成，對我們來說短短一秒，實際上在電腦世界裡是可能有幾個小時的感覺差異。這個結果也是b->a

要確保異步回調函式的執行順序，這已經涉及到異步程序的執行流程的問題，只能使用新式的 Promise、Generators、async/await 或外部相關函式庫的組織方式

因不同瀏覽器品牌與版本的 JavaScript 執行引擎的設計不同，異步中的異步這種執行程序的順序，以上面的例子來說，是沒有辦法完全保証執行順序的，這只是一個簡單的說明用例子而已。

結語

以下列出常見的對 JavaScript 語言"誤解"的講法:

- JavaScript 中的 callback(回調)函式都是異步執行的。
- JavaScript 執行是多執行緒執行的。
- 異步執行的函式通常比同步的還快。
- JavaScript 中的程式碼每段都是異步執行的。
- JavaScript 會把異步的函式先移到佇列中去執行，執行後再把結果返回到呼叫堆疊中。
- 大部份的事件處理函式是直接執行的函式，是同步執行的而不是異步執行的。

上面全部都是"錯誤"的。如果你有一點點懷疑，請再看一遍這篇的內容吧。

參考資料

- [JavaScript's Call Stack, Callback Queue, and Event Loop](#)
- [loupe](#)
- [Philip Roberts: What the heck is the event loop anyway? | JSConf EU 2014](#)
- [Concurrency model and Event Loop\(MDN\)](#)
- [Evolution of Asynchronous JavaScript](#)
- [Node Hero - Understanding Async Programming in Node.js](#)
- [The JavaScript Event Loop: Explained](#)

Prototype(原型) 基礎物件導向

If you don't understand prototypes, you don't understand JavaScript.

如果你沒搞懂原型，你不算真的懂 JavaScript

JavaScript 本身就是原型為基礎的物件導向設計，至 ES6 標準制定後仍沒變動過。在物件的章節中所介紹的類別定義方式，只是原型物件導向語法的語法糖，骨子裡還是原型，並不是真正的以類型為基礎的物件導向設計。理解 JavaScript 的原型是很重要的，只是混亂得讓初學者難以理解。

註: 語法糖(Syntactic sugar)指的是在程式語言中添加的某些語法，這些語法對語言本身的功能並沒有影響，但是能更方便使用，可以讓程式碼更加簡潔，有更高可讀性。另外類似的術語還有"語法糖精"與"語法鹽"。

函式 - 原型的起手式

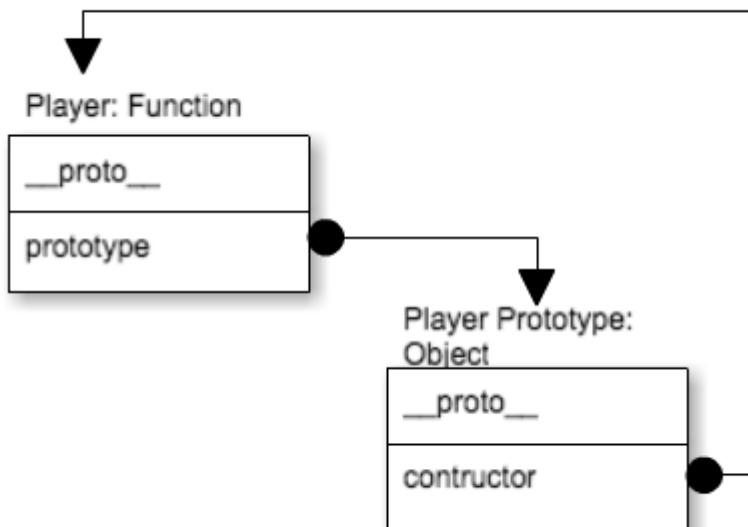
所有 JavaScript 中的函式都有一個內建的prototype屬性，指向一個特殊的 prototype 物件，prototype 物件中也有一個constructor屬性，指向原來的函式，互相指來指去會讓你覺得有點怪異，但設計就是如此。

以下的程式碼可以看出這個關係:

```
function Player() {}

console.log(Player)
console.log(Player.prototype)
console.log(Player.prototype.constructor)
console.log(Player.prototype.constructor === Player) //true
```

為了更容易理解，以下是一個簡單的關係圖:



再來是`__proto__`這個內部屬性，它是一個存取器(accessor)屬性，意思是用getter和setter函式合成出來的屬性，我們可以用它來更加深入理解整個原型的樣貌。`__proto__`是每一個 JavaScript 中物件都有的內部屬性，代表該物件繼承而來的源頭，也就是指向該物件的原型(prototype)，它會用來連接出原型鏈，或可以理解為原型的繼承結構。

對於一個函式而言，它本身也是一個物件，它的原型就是最上層的`Function.prototype`，你可以說這是所有函式的發源地。所以`Player`函式本身的`__proto__`指向`Function Prototype`，這應該可以很容易理解。

那麼，`Player.prototype`的`__proto__`指向哪裡？`Player.prototype`本身也是個物件，它指向的就是所有 JavaScript 中最上層的物件起源，也就是`Object.prototype`。由此也可推知，`Function.prototype`也同樣指向`Object.prototype`。以下面的程式就可以看到這個結果：

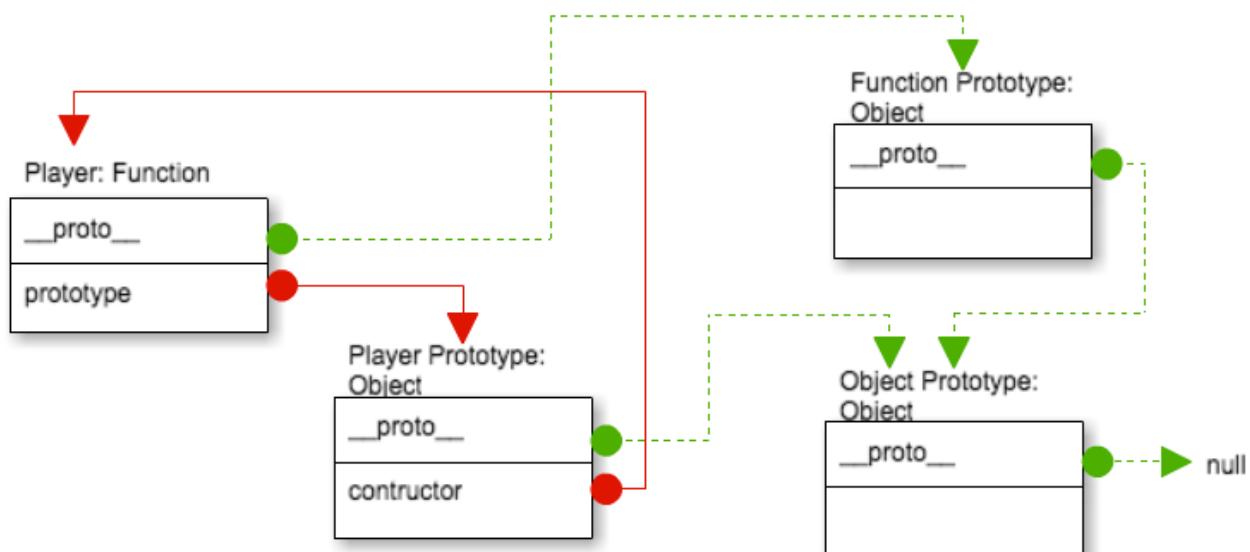
```
function Player() {}

console.log(Player.__proto__)
console.log(Player.prototype.__proto__)

//最上層的Object.prototype的__proto__是null值，它是一個特例
console.log(Object.prototype.__proto__) //null

console.log(Player.__proto__ === Function.prototype) //true
console.log(Player.prototype.__proto__ === Object.prototype) //true
console.log(Function.prototype.__proto__ === Object.prototype) //true
```

為了更容易理解，以下是一個簡單的關係圖，在圖片中綠色的虛線即是`__proto__`的指向，原本的`prototype`為紅色的實線：



註：`__proto__`注意是前後各有兩條下底線(_)，不是只有一條而已。

註：`__proto__`在一些舊的瀏覽器品牌(例如 IE)中不能使用。雖然在 ES6 中已經正式納入標準之中，它是個危險的內部屬性，也不要用在真正的應用程式上。

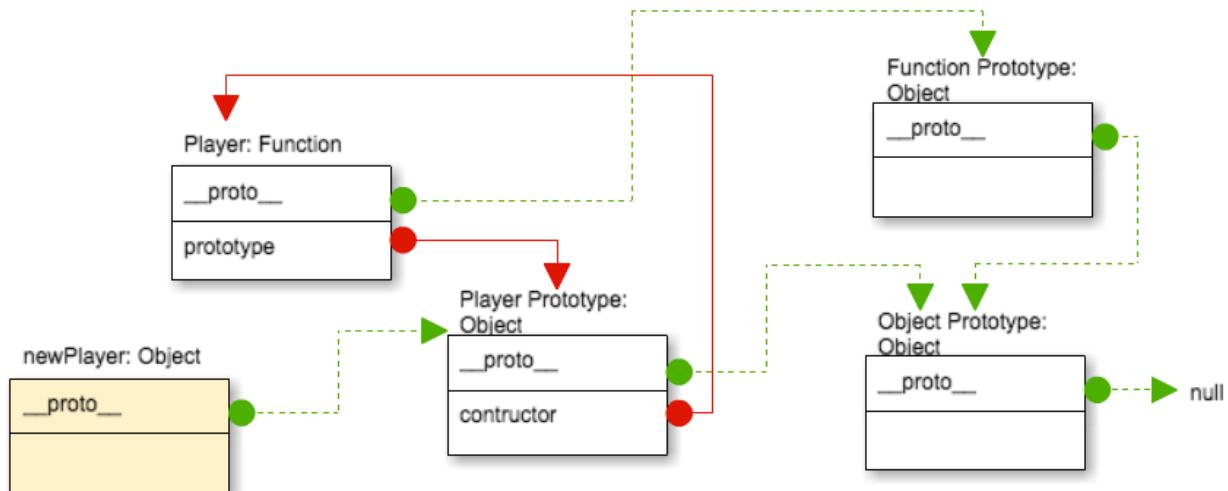
當進一步使用 Player 函式作為建構函式，產生物件實體時，也就是使用new運算符的語句。像下面這樣簡單的例子，在建構函式中會用this.name的方式來指定傳入參數，this按照之前在物件篇的內容所說明，指向的是new運算符中指定的物件實體。

```
function Player(name) {  
    this.name = name  
}  
  
const newPlayer = new Player('Inori')
```

此時在newPlayer物件中的prototype與__proto__又是如何？由於newPlayer是一個物件，並不是函式，它不會有prototype這個屬性。newPlayer的__proto__則是指向Player.prototype，把物件的原型鏈整個串接起來。

```
//不是函式，不會有prototype  
console.log(newPlayer.prototype) // undefined  
  
console.log(newPlayer.__proto__)  
console.log(newPlayer.__proto__ === Player.prototype) //true
```

以下是一個簡單的關係圖，黃色的代表剛剛實體化的newPlayer物件，在圖片中綠色的虛線即是__proto__的指向：



最後總結以下的幾個摘要，讓這章節的內容更加清楚：

- 每個函式中都會有prototype屬性，指向一個prototype物件。例如 MyFunc 函式的prototype 屬性，會指向對應的 MyFunc.prototype 物件。
- 每個函式的prototype物件，會有一個constructor屬性，指回到這個函式。例如 MyFunc.prototype 物件的constructor屬性，會指向 MyFunc 函式。
- 每個物件都有一個__proto__內部屬性，指向它的繼承而來的原型prototype物件
- 由__proto__指向連接起來的結構，稱之為原型鏈(prototype chain)，也就是原型繼承的整個連接結構

原型真相 - 由問答理解

原型到底是什麼，從上面看到原型鏈、new 運算符、建構式等等的概念，你可能會有一些疑惑或誤解。以下是幾個重點，我們用問答的方式來說明。

原型是什麼？

一種讓別的物件繼承其中的屬性的物件。

任何物件都有其原型？

是的。唯一的例外是 Object.prototype(Object 的原型)沒有原型，它是所有物件的最上層的源頭。

任何物件可以拿來作為原型？

是的。

那為什麼上面的例子中，newPlayer物件沒有 prototype 屬性

用Object.getPrototypeOf方法或__proto__屬性，可以找到這個物件真正的原型是什麼，而這個prototype 屬性是給建構函式用的。事實上，每個物件都一定有 constructor(建構式)屬性，constructor(建構式)屬性會指向建立這個物件的函式，建構函式的屬性其中就有 prototype 屬性。

```
const newPlayer = new Player('Inori')

console.log(Object.getPrototypeOf(newPlayer))
console.log(newPlayer.__proto__)

//只能用於不是使用Object.create建立的物件
console.log(newPlayer.constructor.prototype)
```

為什麼函式中一定會有 prototype 屬性，那建構函式與函式的區分又是什麼？

設計上的原則而已，JavaScript 並沒有對你所建立的函式，區分建構函式與一般的函式，所以只要是函式，就一定會有 prototype 屬性(除了語言內建的函式不會有這個屬性)。而且，函式以外的類型，不會有這個屬性。

擴充

原型可以很容易的擴充屬性與方法，而且是動態的，可以在物件實體後繼續擴充其中的成員，這也是 JavaScript 中常用來擴充內建物件的方式。當使用Player.prototype來進行擴充時，這些擴充出來的屬性與方法，是所有物件共享的，見以下的範例。

```
function Player(name) {
  this.name = name
}

const newPlayer = new Player('Inori')
```

```

console.log(newPlayer.age)

Player.prototype.age = 11
console.log(newPlayer.age)

Player.prototype.toString = function() {
    return 'Name: ' + this.name
}

console.log(newPlayer.toString())
console.log(newPlayer) //觀察物件的內容

```

註: 如果以類別為基礎的物件導向，在物件實體化後，物件或類別都是無法擴充其中的成員的。這一點是原型物件導向的彈性之處。

繼承

繼承是什麼？我們回歸本質上來思考"繼承的目的是什麼"，在程式開發時的目的通常是為了擴充原有的物件定義不足之處。繼承基本上可以依照不同程式語言的物件導向特性區分為：

- 類別的繼承(Classical inheritance)
- 原型為基礎的繼承(Prototype-based inheritance)

原型繼承是什麼，其實就是原型的擴充語法，上一節有說明過了。至於類別的繼承方式，在 JavaScript 可以用Object.create方法模擬出來，不過複雜多了。

```

//superclass
function Player(name) {
    this.name = name
}

// 1. 呼叫上層的建構式
function VipPlayer(name, level) {
    Player.call(this, name)
    this.level = level
}

// 2. 使用Object.create建立prototype物件。
//     VipPlayer建構式的prototype.constructor為自己。
VipPlayer.prototype = Object.create(Player.prototype)
VipPlayer.prototype.constructor = VipPlayer

var inori = new VipPlayer('inori', 5)

console.log(inori instanceof Player) //true
console.log(inori instanceof VipPlayer) //true

```

相當於 ES6 中的類別定義方法，使用 extends 關鍵字來作類別的繼承：

```

class Player {
  constructor(name) {
    this.name = name
  }
}

class VipPlayer extends Player {
  constructor(name, level) {
    super(name)
    this.level = level
  }
}

```

總而言之，在很多真實的應用情況下，“以合成(或擴充)代替繼承(composition over inheritance)”才是正解，在 JavaScript 中合成比繼承容易得多了，彈性高應用也很廣，也比較符合語言本身的特性。思考的重點不同，才能撰寫出符合應用情況的程式碼。

私有/公開成員

早在十多年前 Douglas Crockford 大師的這篇[Private Members in JavaScript](#)就有提出關於私有成員的樣式，使用的是建構式樣式來模擬私有、公有與特權方法。網路上大部份的文章都是使用這個樣式來說明，或是進一步改良。

不過，私有或公開成員完全不是 JavaScript 語言中原本就有的設計，用這種方法會限制住只能使用“建構式樣式”的語法來作物件實體化，而且也與原型物件導向概念相去甚遠。原型物件導向對於封裝的概念，基本上是根本沒有。

目前比較簡單常見的區分方式，就是在私有成員(或方法)的名稱前面，加上下底線符號(_)前綴字，用於區分這是私有的(private)成員，這只是由程式開發者撰寫上的區分差別，與語言本身特性無關，對 JavaScript 語言來說，成員名稱前有沒有下底線符號(_)的，都是視為一樣的變數，至於要如何保護這些私有成員，就靠程式設計者自己了。

靜態屬性/靜態方法

JavaScript 語言中也沒這概念。不過，原型物件另外定義的屬性與方法，是所有以此原型物件實體化的物件共享的就是了：

```

function Employee() {}
Employee.prototype.count = 0

var e = new Employee()
console.log(e.hasOwnProperty('count')) // logs "false"
e.count += 1
console.log(e.hasOwnProperty('count')) // logs "true"

console.log(e.count) // logs "1"
console.log(Employee.prototype.count) // logs "0"
var e = new Employee()

```

```

console.log(e.count) // logs "0"
++Employee.prototype.count
console.log(e.count) // logs "1"

var e = new Employee()
console.log(e.count) // Logs "0", because it's using `Employee.prototype.c
++e.count // Now `e` has its own `count` property
console.log(e.count) // Logs "1", `e`'s own `count`
delete e.count // Now `e` doesn't have a `count` property anymore
console.log(e.count) // Logs "0", we're back to using `Employee.prototype.

```

另一種作法是用 IIFE 模擬出靜態變數的結構:

```

var Employee = (function() {
  var sharedVariable = 0

  function Employee() {
    ++sharedVariable
    console.log('sharedVariable = ' + sharedVariable)
  }

  return Employee
})()

new Employee() //1
new Employee() //2
new Employee() //3
new Employee() //4

```

原型鏈(prototype chain)

原型鏈是 JavaScript 中以原型為基礎的物件導向特性，指的是使用原型來作物件實體化，會產生"原型鏈"的結構。原型鏈的觀念如此重要，在於有很多物件的行為都是與它相關，在物件篇已經有介紹過物件中的一些方法，都是會遍歷整個物件的原型鏈，而不僅是物件本身而已。這與原型的物件實體化設計有關，因為物件的實體化過程，就是原型的繼承過程，也就是物件的實體化是由繼承其他物件而來的。

先看一下物件屬性的存取這件事，在下面的例子中，我們並沒有在player物件中定義toString方法，但它的確是存在的，這個toString方法是來自原型鏈上層的物件中，也就是繼承得來的。

```

const player = {}
console.log(player.toString())

```

instanceof

instanceof是一個運算符，主要是用來判斷一個物件在原型鏈中是否有存在某個建構式，如果存在回傳 true，要不然就是false值。它的前面的運算子是物件，後面則是建構式，語法是像下面這樣:

object instanceof constructor

`instanceof`常會拿來和存取物件實體的`constructor`屬性的方式作比較，由於`constructor`只會指向最接近的建構式，而`instanceof`會找遍整個原型鏈，當然結果會有所有不同，以下為範例：

```
function Animal() {}

function Cat() {}
Cat.prototype = new Animal()
Cat.prototype.constructor = Cat

const kitty = new Cat()

console.log(kitty instanceof Cat) // true
console.log(kitty instanceof Animal) // true
console.log(kitty.constructor === Cat) // true
console.log(kitty.constructor === Animal) // false
```

`instanceof`有一些例外情況，它對於原始資料類型(數字、字串、布林、null、undefined)是無法判斷的，必定是回傳`false`，這和用`constructor`屬性判斷是不同的結果，例如以下的範例：

```
console.log(3 instanceof Number) // false
console.log(true instanceof Boolean) // false
console.log('abc' instanceof Boolean) // false
console.log((3).constructor === Number) // true
console.log(true.constructor === Boolean) // true
console.log('abc'.constructor === String) // true
```

另外對在`iframe`、`frame`或著另開視窗中所產生的物件進行判斷時，它也會失效。

在不使用建構式的物件建立的情況，它也無法判斷，而且會產生錯誤，所以它並不能使用於像`Object.create`方法，或直接回傳物件的工廠樣式，只能用於建構式樣式。

in 與 for...in

`in`運算符是用來判斷某個屬性是否存在於某個物件中，存在的話會回傳`true`，否則會回傳`false`。`in`一樣會尋遍整個原型鏈，對比`hasOwnProperty`則是只會在這物件當中，並不會往原型鏈尋找，`in`通常會搭配`for`使用`for...in`語句。以下這個語法是最常見到的：

```
for (let key in obj) {
  if (obj.hasOwnProperty(key)) {
    console.log(obj[key])
  }
}
```

鴨子類型(Duck typing)

那麼我們要如何正確的判斷一個物件，就是我們需要的物件？首先，`typeof`運算符所能判斷的情況過少，只能用於判斷資料類型，在資料類型那個章節就有提及它的內容。對於物件、陣列、`null`來說，它都會回傳'object'。

`instanceof`只能用於以`new`實體化的物件，也就是有建構式的情況，而且它另外有一些失效的情況，`instanceof`有時會直接產生錯誤中斷執行，而不是回傳`false`。`instanceof`並不是不能使用，常見的使用情況是用於判斷語言內建的幾個物件，例如以下幾個：

```
[1, 2, 3] instanceof Array // true  
/abc/ instanceof RegExp // true  
({}) instanceof Object // true  
(function(){}) instanceof Function // true
```

那如果是在一個函式的傳入參數，要如何判斷這個傳入物件實體是我們要的？

答案就是使用"鴨子類型(Duck typing)"，也就是使用該物件的屬性或行為複合地來判斷它，類似像下面的說明：

當看到一隻鳥走起來像鴨子、游泳起來像鴨子、叫起來也像鴨子，那麼這隻鳥就可以被稱為鴨子。

所以當我們要判斷一個自訂的物件，可以用其中應該包含的屬性與方法來判斷，例如以下的判斷函式：

```
function isPlayer(object) {  
    return (  
        object != null &&  
        typeof object === 'object' &&  
        'name' in object &&  
        'age' in object &&  
        'toString' in object  
    )  
}
```

new 有害說與物件實體化

JavaScript 長期以來就有反對使用 `new` 運算符用於實體化物件的言論，建議開發者不要使用它來作物件的實體化，主要的理由是語言本身設計上的有一個很明顯的缺陷：

開發者忘了在物件實體化時加上 `new` 運算符：函式並無明確區分建立物件用的建構函式與一般的函式，如果你是要用來實體化物件，而忘了加上 `new` 關鍵字，雖然不會產生任何錯誤，但結果會很嚴重。

以下用幾個方向來討論如何正確的其他作法，以及如何避免其中可能的問題。

防止錯誤的語法樣式

這個樣式可以防止程式設計師少加了`new`運算子。一般的 JavaScript 函式庫並不鼓勵使用它的開發者使用 `new`，反而會希望使用函式的方式來建立物件實體，原因除了防止漏寫的錯誤外，也可以隱藏對於物件實體的複雜的生成過程。以下為範例：

```

function Player(name, age) {
  if (this instanceof Player) {
    this.name = name
    this.age = age
  } else {
    return new Player(name, age)
  }
}

const aPlayer = Player('Inori', 16)
const bPlayer = new Player('Gi', 16)

```

Object.create

`Object.create`方法可以使用物件的原型來建立新物件，這是一個 ES5 後加入的新方法。最早在十年前在這篇文章[Prototypal Inheritance in JavaScript](#)提出的想法與實作，文章中提及`new`本身就是一個為了要讓 JavaScript 中的物件實體化，用起來像是類別為基礎的程式語言，才會設計的一個語法，但因此模糊了原型繼承的真正作法。

`Object.create`並不只是`new`運算符的取代方法這麼簡單，它提供了更多的彈性，把物件導向的語法結構變成原本的原型導向，它的基本語法如下：

Object.create(proto[, propertiesObject])

必要的傳入參數是物件的原型，在下面的範例中可以看到。不過要先說明的是為何它把整個語法結構都轉變了。按照`new`運算符來作實體物件的工作，原本的步驟是像下面這樣的：

1. 先撰寫建構函式，定義好裡面的屬性與方法
2. 然後用`new`來建立物件實體

`new`會作的工作已經說明過很多次了，指向`this`到新建立的物件實體、執行建構函式，最後回傳物件實體。如果你比較一下 ES6 中的類別語法，這個流程與以類別為基礎的物件導向幾乎無異，差異只是在撰寫類別定義與撰寫建構函式定義上，類別定義現在只是個語法糖，還記得嗎？所以當然是一樣的。

那麼原型物件導向原來的流程應該是如何的？原型物件導向有幾個核心的概念：

- 物件繼承自其他物件：並不是先有物件的藍圖(類別)，然後實體化它。而是先有要作為原型的物件，然後用由這個物件產生新的物件。
- 以合成(或擴充)代替繼承：合成(或擴充)的方式才是重點，由原型物件開始實體一個新物件，可以很容易的合成與擴充。

`Object.create`方法相較於`new`運算符，在物件實體化過程中有一個非常明顯的差異：

Object.create 不會執行建構函式

`Object.create`的使用流程會是這樣的，這是簡化過的版本，實際上可能不只這樣：

1. 定義一個物件當作原型物件
2. 從這個原型物件建立另一個新的物件(過程可以加入其他的屬性)

以一個最簡單的範例來說，你可以看到Object.create完全不使用建構函式來設定新物件實體的屬性值，只是從原型物件產生一個新物件而已。

```
const PlayerPrototype = {
  name: '',
  toString() {
    return 'Name: ' + this.name
  },
}

const inori = Object.create(PlayerPrototype)
inori.name = 'iNori'
console.log(inori.toString())
```

那麼我們如果要對物件實體進行初始化要怎麼作，就是呼叫原型物件裡一個自訂的初始化用的方法就行了，通常會用 init 來當這個方法的名稱：

```
const PlayerPrototype = {
  init(name, age) {
    this.name = name
    this.age = age
  },
  toString() {
    return 'Name: ' + this.name + ' Age:' + this.age
  },
}

const inori = Object.create(PlayerPrototype)
inori.init('iNori', 16)
console.log(inori.toString())
```

註：這個語法樣式，有個專有名稱叫作 OLOO(objects linked to other objects)樣式

註：因為沒有了建構式，所以有一些屬性(例如 constructor)與方法(例如 instanceof)不能使用，會有錯誤的情況。可以用isPrototypeOf方法來判斷原型的關係。

Object.create方法的第二個傳入參數，可以使用一種特殊的屬性物件(properties Object)，提供更多在物件建立時的彈性運用，例如以下的範例：

```
const inori = Object.create(PlayerPrototype, {
  hairColor: {
    value: 'pink',
    writable: true,
  },
})
inori.init('iNori', 16)

console.log(inori)
```

註: 屬性物件是一種用來定義屬性的特殊物件，請參考[Object.defineProperties\(\)](#)

建立物件的語法比較

我們目前為止已經看到物件的建立有以下這三種方式:

```
//物件字面定義，相等於new Object()  
const newObject = {}  
  
//使用Object.create方法  
const newObject = Object.create(proto)  
  
//ES6類別定義，或是建構函式。通常稱為建構式樣式。  
const newObject = new ConstructorFunc()  
const newObject = new ClassName()
```

在進入建立物件的主題前，我想先說明一下，在 JavaScript 中關於建立物件這件事，是有需要那麼常用到的嗎？或是真的會在單一個應用程式中，建立大量的物件的情況？

你應該了解，JavaScript 的應用程式都是執行在瀏覽器的環境中，這是一個有受到限制的執行環境。而且是當一個使用者連到網頁時，JavaScript 的應用程式才會透過網站傳遞到使用者電腦中的瀏覽器，然後才執行，這與一般的桌面或手機上的應用程式，先安裝後才執行完全不同。

一般常見程式設計師會在 JavaScript 應用程式裡，建立不重覆的自訂物件資料的應用情況有可能是以下幾種:

- 網頁上的 UI 小元件、行事曆、對話盒等等: 一個網頁上頂多是 10-20 個物件。
- 用來描述資料模型的物件: 用來作為最終的資料交換使用，描述資料的物件，頂多 5~10 個物件。
- 用於應用程式的物件: 例如一個遊戲中，對於怪物、玩家角色、NPC 的這些物件，大概就是 20-50 個。

以效能來說，物件的建立這件事，不太可能像在網路上測試報告的情況，一次建立幾十萬個或百萬個物件。物件的建立與各種運算，本身就是高消費的，所以在物件的建立，反而效率並不是太重要的課題，而是它在撰寫時的高閱讀性、易於維護與擴充、或是使用上的彈性等等。

以上面的三種物件建立的方式來說，效率最佳的是物件字面定義(花括號{})定義物件)，其次為建構函式加上 new 運算符這種，通常稱之為"建構式樣式(Constructor Pattern)"，最差的則是Object.create。

但物件字面定義語法有一些問題，它只會有一個物件實體。它沒辦法直接複製出其他的物件實體，所以如果是要指"可重覆建立多個物件"的語法，這個並不是可以這樣用的，它需要寫成一個像下面這樣的函式，才能達到需求，這通常被稱為"工廠樣式(Factory Pattern)"的語法:

```
function PlayerFactory(name, age) {  
    return {  
        name: name,  
        age: age,  
        toString: function() {  
            return 'Name: ' + this.name + ' Age: ' + this.age  
        },  
    }  
}
```

```

}

const inori = PlayerFactory('Inori', 16)
console.log(inori.toString())

const ayase = PlayerFactory('ayase', 17)
console.log(ayase.toString())

```

單純使用物件字面定義的工廠樣式在效率上是吊車尾的，而且由於所有的物件實體都類似於物件字面所定義出來的，它們的原型都是`Object.prototype`。工廠模式也可以用的`Object.create`方法來建立物件實體，搭配物件字面定義出來的物件，建立新的物件實體，上面已經有範例，不過要用哪一種，都是要視應用的情況而決定的較為合適的語法。

必要情況(內建物件)

有些 JavaScript 語言中的內建物件，在使用時一定要用 `new` 運算符進行實體化，例如以下幾個：

```

new Date()
new XMLHttpRequest()
new Error()
new RegExp('ab+c')

```

除了這些比較特別的物件之外，JavaScript 語言中內建的包裝物件幾乎都不使用 `new` 作物件實體化，可以用但不建議使用。

建構式樣式 vs 工廠樣式

`new` 在真實的應用情況也很少會用到，不過我認為主因應該是語法樣式，而非單純只有`new`本身的問題，重點應該放在，對於"建構式樣式"與"工廠樣式"的比較。至少到目前為止的所看到的，"建構式樣式"教得人多但用得人很少，"工廠樣式"的使用頻率遠遠勝過"建構式樣式"。

"建構式樣式"是以建構函式為主的語法樣式，使用 `new` 作為物件實體化的唯一方式，最後回傳物件實體。而只能把物件的定義內容寫在建構式之中，只會回傳物件實體，限制住很多能使用的情況。建構函式原本就是一個 JavaScript 中十分怪異的設計，除了初學者一定會很容易與一般的函式搞混，它有一些隱含的機制也很奇特。

"工廠樣式"的語法則提供了更多的彈性，相較於"建構式樣式"只能回傳物件，"工廠樣式"最後直接回傳物件實體，但"工廠樣式"也可以多了很多彈性，可以視情況提供各種物件實體的應對程式碼，最後可以回傳以物件字面定義的物件、使用 `new` 或 `Object.create` 方法。此外，工廠樣式可以對物件資料進行更好的封裝(`encapsulation`)與資料隱藏(`data hiding`)，這一點在建構式樣式中完全是個無法比得上的。

更多的樣式

原型鏈共享的工廠樣式

工廠樣式提供了更多的彈性，因為`Object.create`直接由一個單純的物件來建立物件，失去了原型鏈的擴充彈性，你可以用下面的樣式來調整：

```

function Player() {}

Player.prototype.toString = function() {
    return this.name
}

function PlayerFactory(name) {
    const obj = Object.create(Player.prototype)
    obj.name = name

    return obj
}

```

實際上 Player 函式與 PlayerFactory 函式兩者可以合併，像下面這樣：

```

function PlayerFactory(name) {
    const obj = Object.create(PlayerFactory.prototype)
    obj.name = name

    return obj
}

PlayerFactory.prototype.toString = function() {
    return this.name
}

const inori = PlayerFactory('Inori')
console.log(inori.toString())
console.log(inori)

```

多重繼承(複合)樣式

這個樣式可以建立繼承自多個物件的新物件，用的是Object.assign加上Object.create方法的語法，這方式可以一次增加多個新物件的屬性與方法，此外Object.assign並沒有限定第二個參數之後只能加一個物件進來合併，所以可以加很多物件來合併成為一個新的物件，類似多重繼承的結果。以下為範例：

```

let player = {
    name: 'player',
    toString() {
        return this.name
    },
}

function PlayerFactory(name, age) {
    return Object.assign(Object.create(player), {
        name: name,
        age: age,
        toString() {

```

```

        return 'Name: ' + this.name + ' Age:' + this.age
    },
})
}

const inori = PlayerFactory('inori', 16)
console.log(inori.toString())
console.log(inori)

```

extend(擴充)樣式

某些時候對於簡單的物件，像是設定值之類的物件，如果都要用到物件實體化或各種語法樣式，實在太過沉重。這個時候用 extend(擴充)的方式是快速簡便的，並不一定要額外進行物件實體化，extend(擴充)樣式是一種 Mixins(混合)樣式，它並不是繼承或物件實體化的樣式。簡單的 extend 函式就只是個迴圈語句而已，範例如下(出自[SweetAlert](#)):

```

var extend = function extend(a, b) {
  for (var key in b) {
    if (b.hasOwnProperty(key)) {
      a[key] = b[key]
    }
  }
  return a
}

```

註: 許多 JavaScript 函式庫例如 jQuery、underscore、lodash 都有提供 extend(擴充)的 API，其他也有像 clone(複製)、merge(合併)、assign(指定)之類的用於物件的 API。

結語

JavaScript 中原型物件導向設計其實並不難理解，難的是它裡面混雜的太多奇怪與不合常理的設計，而且又常常要與類別為基礎的物件導向設計相互比較。本章除了提供原型鏈的基礎知識說明外，也加入了很多你可能會在實際使用時遇到的樣式，我會認為工廠樣式與建構式樣式，這兩個基本的樣式你應該要先熟悉。