異步程式設計與事件迴圈

> 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。

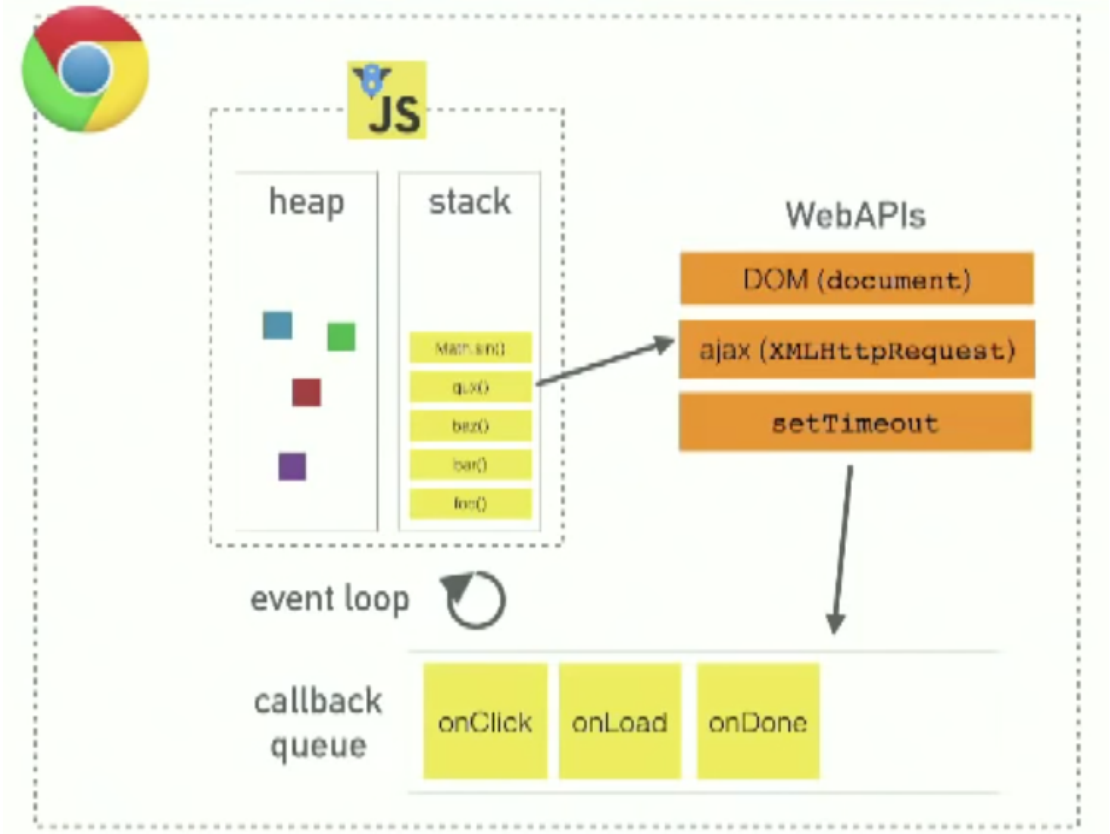
> 由於JavaScript中的執行區分是以執行上下文(EC)作為一個單位，也就是以函式區分，所以一般要討論異步或同步執行，也通常會用異步執行的函式或同步執行的函式來區分。程式碼中的每個語句一定是同步執行的。而異步執行函式必定會使用CPS風格的異步回調函式，這是JavaScript中的設計，關於異步回調函式，可以參考特性篇的"Callback 回調"章節的內容。

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。



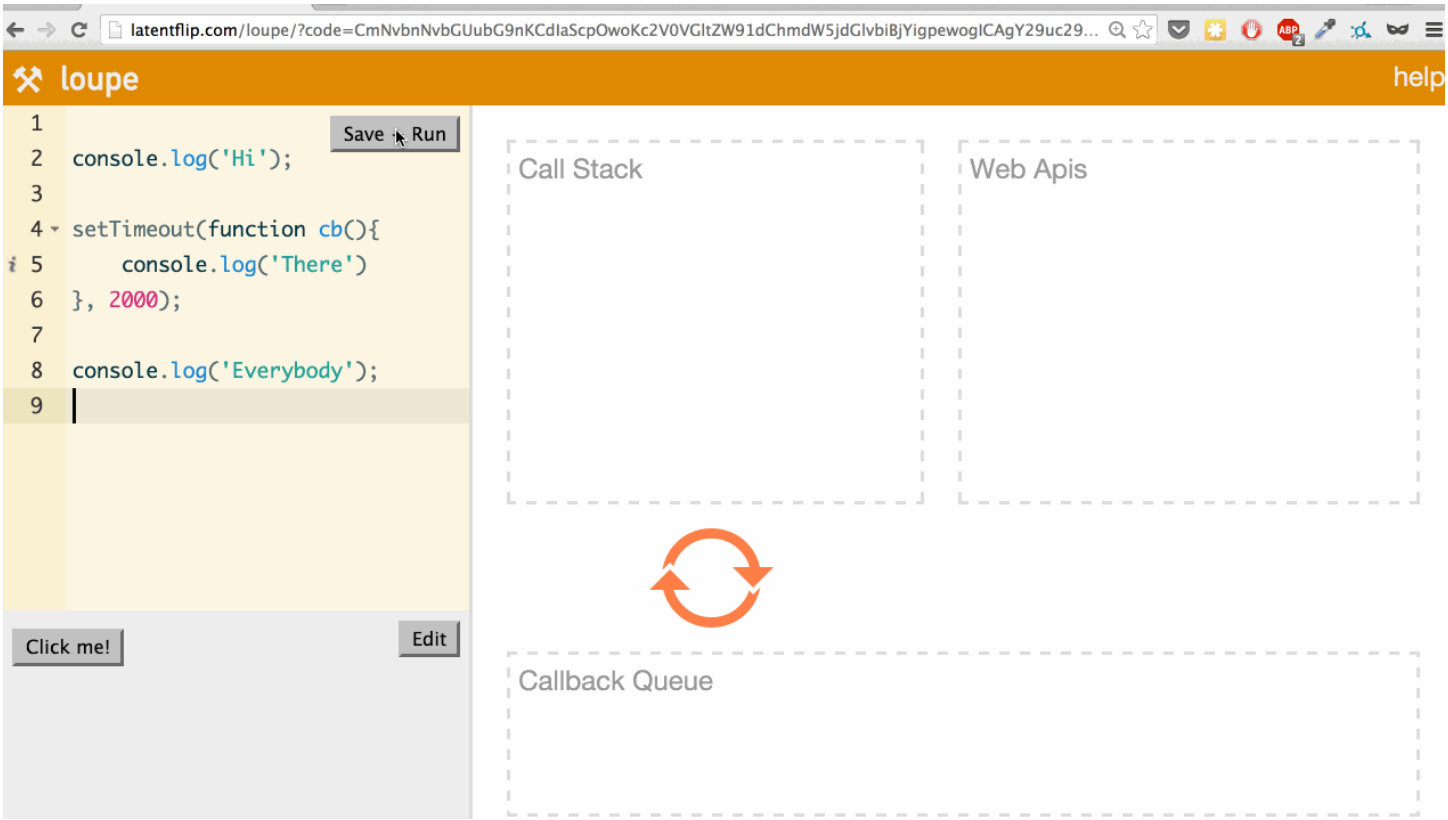
最常使用的例子是用 `setTimeout` 這個內建的方法，它會在某個設定的時間的執行其中的callback(回調)函式傳入值一次:

```
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(FILO)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(FILO)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、asyc/await或外部相關函式庫的組織方式。單純這樣用是沒辦法的。

小結

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