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。而在程式開發上上，callback它的使用情境其實也是有類似的地方。

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了。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，而不使用函式回傳值的方式。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。下面以範例來說明會比較容易。

直接風格的範例如下，其實就是一般函式呼叫的方式，或是用回傳的方式:

```
//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，像下面這樣:

```
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，直接風格的程式碼是最常被使用的，因為它容易被學習與理解，一個步驟接著一個步驟，學校的程式語言課程大部份也是用這種風格來教學，不論在個人電腦上的、在伺服器端的程式語言設計通常也是這樣。主要是因為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風格反而很少被使用，並沒有明顯的誘因讓程式設計師一定要用CPS風格，而且也不是所有的程式語言都能使用CPS風格，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，維護性與閱讀性也很低
- 在錯誤處理上較為困難

JavaScript中會大量使用CPS風格，除了它本身可以使用這種風格外，其實是有原因:

- 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式的進行，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。
- 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。

> 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9成9都是異步執行的，語言內建API中使用的回調函式不一定是異步執行的，也有同步執行的例如 `Array.forEach` ，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？你可能會產生疑惑。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。

> 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，而且都打包好好的。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

我會認為回調函式會複雜的原因是主要是來自"匿名函式"、"函式定義"與"函式呼叫"的混合寫法。所以當在看程式碼時，你的腦袋很容易打結。

```
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)
})
```

但其中的這一段是什麼，這個是一個"函式定義"，而且還是個"匿名函式"定義，它是一個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。(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，等於說它可以提高函式的擴充性與重覆使用性。

回調地獄(Callback Hell)

複雜的情況是在於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，就會看到所謂的"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(value3)
```

那為何為不使用直接風格？而一定要用這麼不易理解的程式流程結構。上面已經有講為什麼JavaScript中會大量的使用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< 100000000;i++){
  aArray[i] = i+10
}

console.log('aArray done!')

const bArray = []
for(let i=0; i< 100000000;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，老實說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，所以要優先處理錯誤，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2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。像下面這個資料庫連接與查詢的範例，出自Node.js MongoDB Driver API:

```
// A simple query using the find method on the collection.

var MongoClient = require('mongodb').MongoClient,
  test = require('assert');
MongoClient.connect('mongodb://localhost:27017/test', function(err, db) {

  // Create a collection we want to drop later
  var collection = db.collection('simple_query');

  // Insert a bunch of documents for the testing
  collection.insertMany([{a:1}, {a:2}, {a:3}], {w:1}, function(err, result) {
    test.equal(null, err);

    // Peform a simple find and return all the documents
    collection.find().toArray(function(err, docs) {
      test.equal(null, err);
      test.equal(3, docs.length);

      db.close();
    });
  });
});
```

上面這個範例還算簡單，但裡面的回調函式有3個，函式呼叫了11個，再複雜的話就會更不好維護，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。現在這種寫法都是一般都已經不建議使用。

現在已經有很多協助處理的方式，回調地獄可以用例如Promise、generator、async/await之類的語法結構，或是Async、co外部函式庫等，來改善或重構原本的程式碼結構，在往後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。

此外，隨著技術不斷的進步，現在的JavaScript也已經有可以讓它使用其他執行緒的技術，有Web Worker，或是專門給Node.js使用的child_process模組與cluster(叢集)模組。

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:

```
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:

```
function foo(err, callback) {
    if (err) {
        callback(err);
    } else {
        callback();
    }
}
```

參考資源

- By example: Continuation-passing style in JavaScript
- Asynchronous programming and continuation-passing style in JavaScript
- Enforce Return After Callback