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，clousure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)，在語言中甚至是有自己獨有的資料類型(`typeof` 一個函式回傳值是'function')。

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。

```
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:

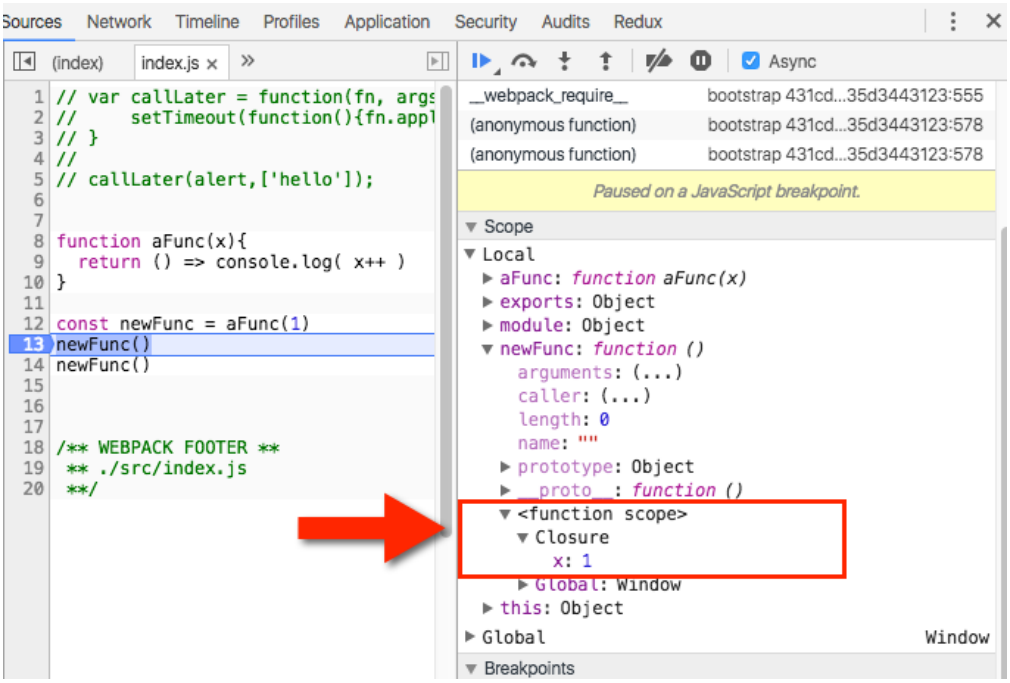
```
function aFunc(x){
  return () => console.log( x++ )
}
```

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，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"Closoure(閉包)"的變數值，這是可以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：
  - 自已本身的
  - 外部函式的
  - 全域的

閉包的記憶環境

閉包的最大特點(賣點)就是它會記憶函式建立時的環境，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。

那麼一個問題會由然而生，閉包是完全複製(copy)了這些值？還是參照(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看得懂:

```
//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` 。這個問題可以用好幾種解法，其實都是要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for迴圈中的語法稱為"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，實際上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，有許多常見的樣式或許你已經有看過，或是一直有使用到的。

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本身就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(實際有很多種寫法)有下面這兩種:

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

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

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。

```
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。

小結

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