Function 故事

# 6/17 function-1

開頭:

故事是這樣的，小明是個對寫coding有那麼點興趣的小小工程師，目前正在練習寫javasciprt，某天他寫著寫著寫出了下面這些東西:

CODE

小明寫了兩張cashCard，而且呢!都有給了個充值的方法，只要餘額(money)是大於等於0的就符合條件給充值，但小明很快了的就發現了個問題:「重複的程式碼太多了，且做了相同的事情」，重複對於程式碼來說是個不好聞的味道，小明就去山里的瀑布底下，誠心思考了七天，突然靈光乍現!「只要把相同的地方包裝起來，需要的時候再呼叫不叫好了嗎?」回家後把程式碼改成下面這個樣子:

CODE

之後如果有卡片需要充值，只要呼叫這個方法並把需要充值的卡片丟入引數內就可以拉!! 這樣程式碼就更簡便了許多!

Function 宣告

………

Function 引數:

……

* 引數 v.s 參數

後來小明覺得這樣還挺不方便的，充值的錢這樣都是固定的，所以小明就想到了再傳入一個引數是我們希望充值的錢不就得了?!於是改成了下面

Code

用著用著有次小明不小心少傳入了一個引數，也就是要充值多少錢，小明心想「完了，這肯定出問題」於是檢查了一下充值過後的cashCard，發現值變成了NaN，到底是發生了什麼事情呢?經過一翻查找資料後發現:

Function arguments

…..

　　又過了幾天，小明又發覺了一個問題，就是這個新的function看了賊煩，cashCard這個名稱太長了，想著能不能夠簡短點，於是又到了瀑布底下誠心思考，果不其然的又想出了個妙法，以後呢我們不寫上cashCard了，通通把cashCard改成this，也不傳入引數了，那這樣要怎麼把我們需要操作的值或是物件丟進去呢?沒問題，我們額外再使用call()方法，把我們需要操作的值或是物件丟進去就行，這樣function就又能夠更簡短一些:

Code

Function ‘this’

* Call()
* Apply()

# 6/18 function-2

開頭

今天呢!我們有個新主角「小華」，他是小明的親梅竹馬，也看見小明這在學習寫程式，自己也打算一起來試試看，畢竟有個人一起練功，是多麼棒的事情!!下面我們就開始今天的故事吧!!

預告:

小華是小明的親梅竹馬，也因為小明在寫code的關係自己也有了興趣，想要嘗試看看!某天小華正想著要寫些什麼來玩玩，突然看到自家養的小狗與小貓，心理出現個念頭!「我來寫個小狗與小貓的計數器好了!!」於是呢就寫出:

Code

現在這個樣子很正常的在運作了!小華很高興也繼續往下寫小貓的計數器:

Code

很快的小華發現了問題，心中的如意算盤破滅，原本心裏想的count並不如心中所想，兩者的count用到了同一樣，以至於無法分開去計數，遇到瓶頸的小華決定去找小明解決方法!!

果不其然，小明之前也碰過這樣的問題，於是開始跟小華分享之前的解決之道:

小明開始畫圖，共畫了兩個方形，一個上頭寫了x，一個上頭寫了y，小明:「現在我這了畫了兩個箱子，最外層的箱子裏頭放了一個x,而裡面的第二個箱子呢放了y，這些箱子有很特別的地方，就是呢外面的箱子看不到裡面的箱子，就如同隔著牆壁一般，但是裡面的箱子看的到外面的箱子，如隔著一層玻璃一般那樣。」

兩個箱子圖(用css做)\_

小華這就不明白了!詢問小明為什麼要跟我這些呢?!

小明繼續解釋，其實這些箱子在程式裡頭我們可以稱為叫做Scope(區塊)，簡單來看就是兩個大括號括住的地方就是一個Scope，而箱子內的環境我們可以稱為Execution context(執行環境)，拿你寫的計數器來說明，你的dog()函式用兩個大括號括住的地方就是一個Scope，而在其中的執行環境我們可以稱為Function Execution context，這樣還是很抽象，我們來舉個例子吧!

比如說現在有:

Code Scope

現在可以看到有outer箱子裡面放了一個inner箱子，剛剛說過，裡面的箱子是看的到外面的箱子的，也就是裡面可以存取到外面的變數，自然可以console出x，但是外面的箱子看不到裡面，所以outer要console出y變數的時候就會出錯了!!這就是Scope，(注意var let const)

那假如說自己箱子裡面本來就有x變數呢?

Code

會看到console出來的是自己本身Scope中的變數!其實，查找時會會先在自己的context裡面找，如果沒有的話就會往外層context繼續找，直到找到全域context為止，而這樣的查找動作就形成像鍊子般，所以我們稱為Scope chain。

※全域context??就是在指在函式function以外的環境。在let還沒出現之前都是使用var，var除了在function內無法被外部存取之外，在其他任何地方都可以存取。

Function Clourse

小華邊聽邊理解，也做了點重點整理:

* 裡面看的到外面，外面看不到裡面;裡面存取的到外面的變數，外面存取不到裡面的變數
* 查找變數的時候會先從目前的context尋找，若找不到就往外層context尋找，直到全域context為止。

小華有些聽明白了，但是還是不懂為何跟自己碰上的問題有關，小明提醒道，我們遇到的問題就是count共用到了對吧?那如果我們能把狗與貓的count分開來，不就解決了嗎?

小華頓時醒悟，奔回家中開始修改:

Code

這個就是所謂的閉包，為了不讓變數洩漏出去給其他的函式，我們就在創造另外一個環境，讓變數只屬於這個環境內，再利用剛剛所學到的原理，就能夠輕鬆解決了!!

Function constructor

小明解決的小華的問題後繼續練習自己所寫的cashcard，現在整體看來是這個樣子:

Code

第一個遇到的問題就是每個cashcard都有addmoney方法太站資源了，所以可以改成:

Code

這樣子做的好處是節省資源，原本每個cashcard各有自己的addMoney方法，有100張cashcard的話就會有100個addMoney function但是改成上方那樣子之後，每個cashcard的addMoney方法都指向同一個，不管有幾張卡片，都會只有一個addMoney function。

解下來小明碰到的問題，就是每次要像目前方式要建立新的cashcard真的太費時費力了所以小明想到可以用function來創立物件，於是就改成：

Code

這樣CashCard的函示我們就稱為建構式Constructor，如果學過Java就肯定不陌生!!(像我就是學Java出生的)，不過不是Java那樣的物件導向語言就是了。當我們有需求需要創建物件的時候，就可以使用建構式。

而其實使用建構式的時候，有一部份的動作是會先建立一個空的物件，然後把呼叫建構式並回傳物件本身，看起來會很像是:

Code

上面只是建構時的一部份動作，還有關prototype特性與更多的東西我們就留到下次再說吧!!

# 6/19 function-3

一改完想了一下，「恩~這樣每個new出來的物件都有toString方法，也太佔資源了吧!這個方法只要一個就好了」於是又改成:

Code

Function toString() {…};

Function CashCard(ID,Money){ };

Function 原型鍊

但其實呢!了解原型練的話，也可以;

CashCard\_pro.prototype.toString = function()

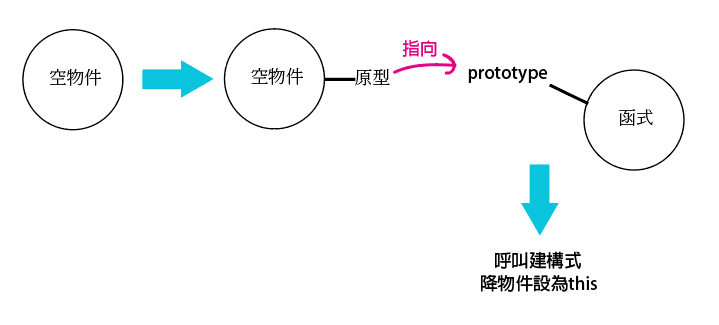
{…}

事實上，用new建立物件的時候，物件上會有個隱藏的特性，”prototype”，而protoype則是物件了原型(等等會說明得更詳細)，會指向使用函式prototype屬性所指向的物件(但不代表function的原型物件，而是建立函式的時候會自動生成prototype屬性，而這個屬性指向一個物件，之後我們就稱為prototype物件)，這個物件會有一個預設的特性=>Constructor，會指向自己(function)，當我們用new創立物件的時候會先創立一個空物件，再來空物件的prototype特性

指向函式的prototype物件，最後在呼叫建構式，看起來很是下面這個樣子:

Code

圖片



\_\_proto\_\_是在ECMAScript 6 規範，被加入了附錄的，可以用來取得實例被建立的被設定的原型，不過在ES5增加了一個Object.getprototypeof()可以取得物件的原型

Code -> Object.getprototypeof

Function 什麼是原型

要談原型呢，這就要來談談繼承了，繼承是什麼意思?就是自表面上的意思，也可以用繼承遺產的「繼承」了解意思，簡單打個比方:今天我有一件武器，每當我斬殺一個敵人，武器都能夠繼承敵人的能力，「他人所擁有的，我也都會有」。

舉例說今天我們寫了一個Person物件，Person有名字跟年齡，所以我們加上了name與age:

Code

但很快的我們發現這需求不夠，世界上有那麼多皮膚顏色不一樣的人，所以我們又寫了WhitePerson與BlackPerson:

CODE

於是我們又發現了一個問題，這Person、whitePerson、BlackPerson每個物件上都有name與age，只不過皮膚顏色物一樣而已，這樣子寫很耗時也很耗資源，而且不管是黑人還是白人都是人類，要怎麼做才能夠簡短又不失去邏輯呢?

如果是用Java來寫的話，我們只需要寫一個Person class類別，並讓其他兩個繼承Person類別(但不加上name與age)，這樣子黑人與白人類別就會繼承了Person類別所擁有的屬性與方法了(不論隱私性)，但我們今天寫的是javascirpt，javascript是怎麼設計繼承這件事情的呢?

答案就是prototype，用建構式出來的實例，或是幾乎所有的物件上都會有prototype屬性，用上面的例子來說，我們可以這樣做:

CODE

這樣子就算WhietePerson上沒有寫上name或是age屬性也都可以找到了，那怎麼能夠找到呢?原理是這樣的，當查找屬性的時候會在自己身上找，當自己身上找不到的時候就會去prototype物件上面找，如果一直到最後都找不到的話，就會回傳undefined，而這樣的查找過程像是一條一條的鍊子把物件串在一起，所以我們就稱為「原型鍊」了。

也可以找到我們創建陣列的prototype喔!包括String物件或是Boolean物件等等都有:

CODE

回歸剛剛的function主題，在用建構式的方式的時候過程中就會把物件的原型指向建構物件函式的prototype囉!所以可以在創建的物件上找到prototype，可以拿來比對是不是跟function的一樣。

CODE

Function \_\_proto\_\_

介紹原型的時候我們曾經提過，在 JavaScript 每一個物件都會有它的原型物件 [[prototype]]。

在過去，雖然 JavaScript 沒有提供標準方法讓我們直接對原型物件 [[prototype]] 來進行存取，不過幸運的是，大多數的瀏覽器 (精準一點說，大多數的 JavaScript 引擎) 都有提供一種叫做 \_\_proto\_\_ 的特殊屬性，來讓我們取得某個物件的原型物件。 要注意的是，並非所有 JavaScript 的執行環境都支援 \_\_proto\_\_，而瀏覽器之間對 \_\_proto\_\_ 的支援也並非完全相容。

所以簡單來說，不管是 \_\_proto\_\_ 這個特殊屬性或者是 Object.getPrototypeOf( ) 其實都是取得某個物件的原型物件 [[prototype]] 的方式。

這次就先說到這裡，之後有相關的知識壓或是其他資訊，會在做補充，接下來我們會想到怎麼樣去模擬封裝與類別! 像是Java那樣子!之後就開始說明DOM囉!!

以Java來說，比如我寫了一個Person類別，類別上有name屬性與age屬性，但一樣米養百樣人，今天有各種膚色的人我可能會去寫很多種其他類別，每個類別(人)上都有name跟age，不同的只是膚色而已，但不管是黃種人還是白人都是人類，且大家都有name跟age,所以我可以把”白皮膚類別”繼承”Person類別，”白皮膚類別”也不寫上name與age了，雖然本身不寫上name與age，但是因為繼承的Person類別的關係，就會擁有Person類別上所有的屬性與方法了(先不論有隱私性)，「Person類別有的，白皮膚類別也都有」這就是繼承。

但是javascript上沒有class(ES6有新增，但實質上也不是Java的類別)，那要怎麼做到繼承這件事情呢?

答案是javascript使用「原型」這樣的功夫，