函式與作用域

函式(function)

函式(function/訪遜/)是JavaScript的非常重要的特性。函式用於程式碼的重覆使用、資訊的隱藏與複合(composition)。我們經常會把一整組的功能程式碼，寫成一個函式，之後可以重覆再使用，JavaScript在執行時的呼叫堆疊也是以函式作為單位。

> 注意: 依據ECMAScript標準的定義，函式的 `typeof` 回傳值是'function'，而不是'object'。由此可見在標準中定義的'object'類型，只是針對"單純"的物件定義而言，但具有函式呼叫(call)實作的物件，將會歸類為'function'，因為它們的內部都有建構函式的特性，例如Date、String、Object這些內建物件，它們的 `typeof` 回傳值都是'function'。 `typeof` 的回傳值只能作為參考用，有很多複雜的應用下並沒有辦法辦別得出是什麼樣的物件。

> 註: 那麼要如何精確又有效的檢查一個變數/常數是否為函式類型？請參考這篇How can I check if a javascript variable is function type?的問答。

函式定義

函式的基本語法結構如下:

```
//匿名函式
function() {}

//有名稱的函式
function foo() {}
```

函式的名稱也是一個識別符，命名方式如同變數/常數的命名規則。

而匿名函式並沒有函式的名稱，通常用來當作一個指定值，指定給一個變數/常數，被指定後這個變數/常數名稱，就成了這個函式的名稱。實際上，匿名函式也有其他的用法，例如拿來當作其他函式的傳入參數值，或是進行一次性執行。

函式使用 `return` 作為最後的回傳值輸出，函式通常會有回傳值，但並非每種函式都需要回傳值，也有可能利用輸出的方式來輸出結果。以下兩種方式對於函式都是可以使用的宣告(定義)方式，使用帶有名稱的函式稱為"函式定義"的方式，而另一種用變數/常數指定匿名函式的稱為"函式表達式"的方式:

```
//函式定義 - 使用有名稱的函式
function sum(a, b){
    return a+b
}

//函式表達式 - 常數指定為匿名函式
const sum = function(a, b) {
    return a+b
}
```

函式的呼叫是使用函式名稱加上括號(())，以及在括號中傳入對應的參數值，即可呼叫這個函式執行。例如:

```
const newValue = sum(100,0)
console.log(sum(99, 1))
```

ES6中有一種新式的函式語法，稱為"箭頭函式(Arrow Function)"，使用肥箭頭符號(Flat Arrow)(=>)，它是一種匿名函式的縮短寫法，下面這個寫法相當於上面的 `sum` 函式定義:

```
const sum = (a, b) =>  a + b
```

箭頭函式(Arrow Function)因為語法簡單，而且可以綁定 `this` 變數，所以算得上最受歡迎的ES6新功能，現在在很多程式碼中被大量使用。我們在特性篇中裡會專門有一章的內容來說明箭頭函式(Arrow Function)。

> 註: `this` 變數與物件有關，在物件的章節會說明。

> 註: 箭頭函式在"特性篇"有一篇專門的教學文章

傳入參數

函式的傳入參數是需要討論的，它是函式與外部環境溝通的管道，也就是輸入資料的部份。

不過，你可能會在函式的"傳入參數"常會看到兩個不同的英文字詞，一個是parameter(或簡寫為param)，另一個是argument，這常常會造成混淆，它們的差異在於:

- parameters: 指的是在函式的那些傳入參數名稱的定義。我們在文章中會以"傳入參數定義名稱"來說明。
- arguments: 指的是當函式被呼叫時，傳入到函式中真正的那些值。我們在文章中會以"實際傳入參數值"來說明。

傳入參數預設值

關於函式的傳入參數預設值，在未指定實際的傳入值時，一定是 `undefined` ，這與變數宣告後但沒有指定值很相似。而且在函式定義時，我們並沒有辦法直接限定傳入參數的資料類型，所以在函式內的語句部份，一開始都會進行實際傳入參數值的資料類型檢查。

有幾種方式可以用來在函式內的語句中，進行預設值的設定，例如用 `typeof` 的回傳值來判斷是否為 `undefined` ，或是用邏輯或(||)運算符的預設值設定方式。用來指定預設值的範例:

```
//用邏輯或(||)
const link = function (point, url) {
    let point = point || 10
    let url = url || 'http://google.com'
    ...
}

//另一種設定的方式，typeof是回傳類型的字串值
const link = function (point, url) {
    let point = typeof point !== 'undefined' ? point : 10
    let url = typeof url !== 'undefined' ? url : 'http://google.com'
    ...
}
```

> 注意: 邏輯或(||)運算符設定預設值，雖然語法簡單，但有可能不精準。它會在實際傳入參數值只要是"falsy"，就直接指定為預設值。

在ES6中加入了函式傳入參數的預設值指定語法，現在可以直接在傳入參數時就定義這些參數的預設值，這個作法是建議的用法:

```
const link = function (point = 10, url = 'http://google.com') {
    ...
}
```

> 註: 只有 `undefined` 的情況下才會觸發預設值的指定值。

> 註: 有預設值的參數在習慣上都是擺在傳入參數列表的"後面"，雖然這並不是個強制的作法只是個習慣。

以函式作為傳入參數

前面有說明函式可以作為變數/常數的指定值。不僅如此，在JavaScript中函式也可以當作實際傳入參數的值，將一個函式傳入到另一個函式中作為參數值，而且在函式的最後也可以回傳函式。這種函式的結構稱之為"高階函式(Higher-order function)"，是一種JavaScript程式語言的獨特特性，高階函式可以讓在函式的定義與使用上能有更多的彈性，它也延申出很多不同的應用結構。你可能常聽到JavaScript的callback(回呼、回調)結構，它就是高階函式的應用。

習慣上，因為函式在定義時，它的傳入參數並沒辦法限定資料類型，所以當要定義傳入參數將會是個函式時，通常會用fn或func作為傳入參數名稱，以此作為辨別。不過，當你在撰寫一個函式時，最好是要加上傳入值的，以及回傳值的註解說明。以下為一個簡單的範例:

```
const addOne = function(value){
    return value + 1
}

const addOneAndTwo = function(value, fn){
    return fn(value) + 2
}
```

```
console.log(addOneAndTwo(10, addOne)) //13
```

無名的傳入參數(unnamed arguments)

這是函式的一種隱藏機制，實際上對於傳入的參數值是有一個隱藏在背後的物件，名稱為 `arguments` ，它會對傳入參數實際值進行保存，可以直接在函式內的語句中直接取用。 `arguments` 雖是一個物件資料類型，但它有"陣列"的一些基本特性，不過缺少大部份陣列中的方法，所以被稱作"pseudo-array"(偽陣列)。以下為一個簡單的範例:

```
function sum() {
  return arguments[0]+arguments[1]
}

console.log(sum(1, 100))
```

不過，如果你在函式的傳入參數定義名稱中，使用了 `arguments` 這個參數名稱，或是在函式中的語句裡，定義了一個名稱為 `arguments` 的自訂變數/常數名稱，這個隱藏的物件就會被覆蓋掉。總之它的行為相當怪異，有一說是說它一開始設計時就錯了，隱藏的 `arguments` 物件對開發者來說，並不像隱藏版的密技，而是比較像是隱藏版的陷阱。

> 註: 關於arguments的詳細介紹，可以參考The JavaScript arguments object…and beyond

不固定傳入參數(Variadic)與其餘(rest)參數

像下面這個範例中，原先sum函式中，定義了要有三個傳入參數，但如果真正在呼叫函式時傳入的參數值(arguments)並沒有的情況下，或是多出來的時候，會發生什麼情況？

前面有說到，沒有預設值的時候會視為 `undefined` 值，而多出來的情況，是會被直接略過。有的時候需要一種能夠"不固定傳入參數"的機制，在各種函式應用時，才能比較方便。

```
function sum(x, y, z) {
  return x+y+z
}

console.log(sum(1, 2, 3))  //6
console.log(sum(1, 2))    //NaN
console.log(sum(1, 2, 3, 4))  //6
console.log(sum('1', '2', '3'))  //123
console.log(sum('1', '2')) //12undefined
console.log(sum('1', '2', '3', '4'))  //123
```

雖然上一節有說過的，有個隱藏的 `arguments` 物件，它可以獲取到所有傳入的參數值，然後用類似陣列的方式來使用，但它的設計相當怪異(有一說是設計錯誤)，使用時要注意很多例外的情況，加上使用前根本也不需要定義，很容易造成程式碼閱讀上的困難。另外，在一些測試報告中，使用 `arguments` 物件本身比有直接使用具有名稱傳入參數慢了數倍。所以結論是， `arguments` 物件的是不建議使用它的。

那麼，有沒有其他的機制可以讓程式設計師能處理不固定的傳入參數？

在ES6中加入了其餘參數(rest parameters)的新作法，它使用省略符號(ellipsis)(...)加在傳入參數名稱前面，其餘參數的傳入值是一個標準的陣列值，以下是一個範例:

```
function sum(...value) {
    let total = 0
    for (let i = 0 ; i< value.length; i++){
        total += value[i]
    }
    return total
}

console.log(sum(1, 2, 3))  //6
console.log(sum(1, 2))    //3
console.log(sum(1, 2, 3, 4))  //10
console.log(sum('1', '2', '3'))  //123
console.log(sum('1', '2')) //12
console.log(sum('1', '2', '3', '4'))  //1234
```

如果要寫得更漂亮、簡潔的的語法，直接使用Array(陣列)本身的好用方法，像下面這樣把原本的範例重寫一下:

```
function sum(...value) {
    return value.reduce((prev, curr) => prev + curr )
}
```

> 註: reduce(歸納)是陣列的方法之一，它可以用來作"累加"

其餘參數只是扮演好參數的角色，代表不確定的其他參數名稱，所以如果一個函式中的參數值有其他的確定傳入參數名稱，其餘參數名稱應該要寫在最後一個位子，而且一個函式只能有一個其餘參數名稱:

```
function(a, b, ...theArgs) {
  // ...
}
```

其餘參數與 `arguments` 物件的幾個簡單比較:

- 其餘參數只是代表其餘的傳入參數值，而 `arguments` 物件是代表所有傳入的參數值
- 其餘參數的傳入值是一個標準陣列，可以使用所有的陣列方法。而 `arguments` 物件是"偽"陣列的物件類型，不能使用陣列的大部份內建方法
- 其餘參數需要定義才能使用， `arguments` 物件不需要定義即可使用，它是隱藏機制

內部(巢狀)函式

函式中的語句中，還可以包含其他的函式，這稱為內部函式(inner)，或是巢狀函式(nested)的結構。以下為一個簡單的範例:

```
function addOuter(a, b) {

    function addInner() {
        return a + b
    }

    return addInner()
}

addOuter(1, 2) //3
```

這樣的程式碼有點類似下面的寫法，不過你可以仔細的比較一下這兩個程式碼之間，傳入參數值的差異:

```
function addOuter(a, b) {
    return addInner(a, b)
}

function addInner(a, b) {
    return a + b
}

addOuter(1, 2) //3
```

內部函式可以獲取到外部函式所包含的環境值，例如外部函式的傳入參數、宣告變數等等。而內部函式又可以成為外部函式的回傳值，所以當內部函式接收到外部函式的環境值，又被回傳出去，內部函式間接變成一種可以讓函式對外曝露包含在函式內部環境值的溝通管道，這種結構稱之為"閉包(closure)"。

內部函式在JavaScript中被廣泛的使用，因為它可以形成所謂"閉包"(closure)的結構。

> 註: "閉包"(closure)在特性篇中有一個獨立的章節來說明。

作用範圍(scope)

"作用範圍(scope)"或稱之為"作用域"，指的是"變數或常數的定義與語句的可見(被存取得到)的範圍"，作用範圍可簡單區分為本地端的(local)與全域的(global)。

JavaScript程式語言的作用範圍，基本上是使用"函式作用範圍(function scope)"的，或稱之以函式為基礎(function-based)的作用範圍。也就是說只有使用函式才能劃出一個本地端的作用範圍，其他的區塊像if、for、switch、while等等，雖然有使用區塊語句({...})，但卻是無法界定出作用範圍。

因此，當你在所有函式的外面宣告變數或常數，這個變數或常數就會變成"全域的"作用範圍的一員，稱為"全域變數或常數"，也就是說在程式碼裡的任何地方都可以被存取得到。

> 註: 在JavaScript語言中，你應該要把"函式"也當作一種"值"來看待，它可以被指定到一個變數/常數，也可以作為函式回傳值。而在作用範圍中的行為也和"值"類似。

在ES6後的新作法，加入"區塊作用範圍(block scope)"概念，也就是使用具有區塊的語句，例如上述的if、for、switch、while等等，都可以劃分出作用範圍，這是一個很棒的改進。那要怎麼作呢？就是以 `let` 與 `const` 取代原本 `var` 的宣告變數的方式， `var` 可以完全捨棄不使用。

如果是使用 `var` 來定義變數，程式碼中的變數x並不是在函式中定義的，所以會變為"全域變數"：

```
if (true) {
  var x = 5
}

console.log(x) //5
```

對比使用 `let` 來宣告變數，程式碼中的y位於區塊中，無法在外部環境獲取得到：

```
if (true) {
  let y = 5
}

console.log(y) //y is not defined
```

其他函式特性

回調(callback)

回調(callback)是一種特別的函式結構，也因為JavaScript具有"高階函式(Higher-order function)"的特性，意思是說在函式中可以用另一個函式當作傳入參數值，最後也可以回傳函式。

一般而言，函式使用回傳值(return)作為最後的執行語句。但回調並不是，回調結構首先會定義一個函式類型的傳入參數，在此函式的最後執行語句，即是呼叫這個函式傳入參數，這個函式傳入參數，通常我們稱它為回調(callback)函式。回調函式經常使用匿名函式的語法，直接寫在函式的傳入參數中。

```
function showMessage(greeting, name, callback) {
    console.log('you call showMessage')
    callback(greeting, name)
}

showMessage('Hello!', 'Eddy', function(param1, param2) {
    console.log(param1 + ' ' + param2)
})
```

由於回調函式是一個函式類型，通常會在使用它的函式中，作一個基本檢查，以免造成程式錯誤，本章節的最前面有說明過了，函式的 `typeof` 回傳值是'function'，以下為改寫過的程式碼，改寫過後不論是不是有傳入回調函式，都可以正常運作，也就是回調函式變成是一個選項:

```
function showMessage(greeting, name, callback) {
     console.log('you call showMessage')

    if (callback && typeof(callback) === 'function') {
        callback(greeting, name)
    }
}
```

回調(callback)提供了使用此函式的開發者一種彈性的機制，讓程式開發者可以自行定義在此函式的最後完成時，要如何進行下一步，這通常是在具有執行流程的數個函式的組合情況。實際上，回調函式實現了JavaScript中非同步(asynchronously)的執行流程，這使得原本只能從頭到底(top-to-bottom)的程式碼，可以在同時間執行多次與多種不同的程式碼。在實際應用情況時，回調結構在JavaScript程式中大量的被使用，它也變成一種很明顯的特色，例如以下的應用中很常見:

- HTML中的DOM事件
- AJAX
- 動畫
- Node.js

提升(Hoisting)

簡單的來說，提升是JavaScript語言中的一種執行階段時的特性，也是一種隱性機制。不過，沒先定義與指定值就使用，這絕對是個壞習慣是吧？變數/常數沒指定好就使用，結果一定是不是你要的。

`var` 、 `let` 和 `const` 會被提升其定義，但指定的值不會一併提升上去，像下面這樣的程式碼:

```
console.log(x) //undefined
var x = 5

console.log(y) //undefined
let y = 5
```

最後的結果出乎意料，竟然只是沒指定值的 `undefined` ，而不是程式錯誤。實際上這程式碼裡的變數被提升(Hoisting)了，相當於:

```
var x
console.log(x)
x = 5

let y
console.log(y)
y = 5
```

函式定義也會被提升，而且它變成可以先呼叫再定義，也就是整個函式定義內容都會被提升到程式碼最前面。不過這對程式設計師來說是合理的，在很多程式語言中都可以這樣作:

```
foo() //可執行

function foo(){
  console.log('Hello')
}
```

不過使用匿名函式的指定值方式(函式表達式, FE)，就不會有整個函式定義都被提升的情況，只有變數名稱被提升，這與上面的變數宣告方式的結果一致:

```
foo() //錯誤: foo is not a function

let foo = function(){
  console.log('Hello')
}
```

結論如下:

- 所有的定義(var, let, const, function, function*, class)都會被提升
- 使用函式定義時，在函式區塊中的這些定義也會被提升到該區塊的最前面
- 當函式與變數/常數同名稱而提升時，函式的優先程度高於變數/常數。
- 遵守好的風格習慣可以避免掉變數提升的問題

全域作用範圍污染

全域作用範圍污染(global scope pollution)，整體來說，也是壞的程式特性+壞的程式寫作習慣，所造成的不良後果。例如沒有經過 var 宣告的變數，會自動變為全域作用範圍，或是是在把變數宣告在全域作用範圍中。過多的變數常常會造成記憶體無法回收，或是全域變數與函式中的變數常常互相衝突。

全域作用範圍污染在JavaScript中，一直是一個長久以來經常會發生的問題，尤其是在程式愈來愈龐大，整體的組織與結構沒有一開始就預作規劃時。ES6中針對作用範圍作了很多的標準上的改進，但不論程式特性如何進步，維護一個良好的寫作習慣，可能比程式本身的功能還重要幾百倍。

所以，好的程式設計師，在撰寫程式時應遵守一些建議的風格習慣，好好地理解作用範圍的概念，這樣就可以避免全域作用範圍污染情況發生。

匿名函式與IIFE

匿名函式還有另一個會被使用的情況，就是實現只執行一次的函式，也就是IIFE結構。IIFE是Immediately-invoked function expressions的縮寫，中文稱之為"立即呼叫的函式表達式"，IIFE可以說是JavaScript中獨特的一種設計模式，它是被某些聰明的程式設計師研究出來的一種結構，它與表達式的強制執行有關，有兩種語法長得很像但功能一樣，這兩種都有人在使用:

```
(function () { … })()
(function () { … }())
```

IIFE在執行環境一讀取到定義時，就會立即執行，而不像一般的函式需要呼叫才會執行，這也是它的名稱的由來 - 立即呼叫，唯一的例外當然是它如果在一個函式的內部中，那只有呼叫到那個函式才會執行。

```
(function(){
    console.log('IIFE test1')
}())

function test2(){
    (function(){
        console.log('IIFE test2')
    }())
}

test2()
```

IIFE的主要用途，例如分隔作用範圍，避免全域作用範圍的污染，避免在區塊中的變數提升(Hoisting)。IIFE也可以再進而形成一種Module Pattern(模組模式)，用來封裝物件的公用與私有成員。許多知名的函式庫例如jQuery、Underscore、Backbone一開始發展時，都使用模組模式來作為擴充結構。

不過，模組模式的結構存在於JavaScript已有很久一段時間，算是前一代主要的設計模式與程式碼組織方式，現在網路上看到的教學文，大概都是很有歷史的了。而現今的JavaScript已經都改為另一種更具彈性的、更全面化的稱為Module System(模組系統)的作法，例如AMD、CommonJS與Harmony(ES6標準的代號)，這在特性篇會有一個獨立的章節再作介紹。

純粹函式與副作用

在表達式的章節中，我們有講到在表達式中副作用(Side Effect)的分別，函式也有這種區分方式，不過對於函式來說，具有副作用代表著可能會更動到外部環境，或是更動到傳入的參數值。函式的區分是以 純粹(pure)函式 與 不純粹(impure)函式 兩者來區分，這不光只有無副作用的差異，還有其他的條件。

純粹函式(pure function)即滿足以下定義的函式:

- 給定相同的輸入(傳入值)，一定會回傳相同輸出值結果(回傳值)
- 不會產生副作用
- 不依賴任何外部的狀態

一個典型的純粹函式的例子如下:

```
const sum = function(value1, value2) {
  return value1 + value2;
}
```

套用上面說的定義，你可以用下面這樣理解它是不是一個純粹函式:

- 只要每次給定相同的輸入值(例如1與2)，就一定會得到相同的輸出值(例如3)
- 不會改變原始輸入參數，或是外部的環境，所以沒有副作用
- 不依賴其他外部的狀態(變數之類的)

那什麼又是一個不純粹的函式？看以下的範例就是，它需要依賴外部的狀態值(變數值):

```
let count = 1;

let increaseAge = function(value) {
  return count += value;
}
```

在JavaScript中不純粹函式很常見，像我們一直用來作為輸出的 `console.log` 函式，或是你可能會在很多範例程式看到的 `alert` 函式，都是不純粹函式，這類函式因為沒有回傳值，都是用來作某件事而已。像 `console.log` 會更動瀏覽器的主控台(外部環境)的輸出，也算是一種副作用。

純粹函式具有以下的優點:

- 程式碼可以簡單化，閱讀性提高
- 較為封閉與固定，可重覆使用性高
- 易於單元測試(Unit Test)、除錯

不過有許多內建的或常用的函式都是免不了有副作用的，例如這些應用:

- 會改變傳入參數變數(物件、陣列)的函式
- 時間性質的函式
- I/O相關
- 資料庫相關
- AJAX

例如而每次輸出值都不同的不純粹函式一類，最典型的就是 `Math.random` ，這是產生隨機值的內建函式，既然是隨機值當然每次執行的回傳值都不一樣。

總之，並不是說有副作用的函式就不要使用，而且要理解這個概念，然後儘可能在你自己的寫的函式上使用純粹函式，以及讓必要有副作用的函式得到管理與控制。現在已經有一些新式的函式庫或框架，會特別要求在某些地方只能使用純粹函式，而具有副作用的不純粹函式只能在特定的情況下才能使用，這就需要先有這樣的概念與特別注意了。

風格指引

- (Airbnb 7.3、Google) 永遠不要在非函式區塊(if, while等等)裡面，定義一個函式。而是把函式指定給一個變數(註: 函式表達式)
- (Airbnb 18.3) 在控制語句(if, while等等)的圓括號()開頭加上一個空白字元。函式在呼叫或定義時，函式名稱與傳入參數列則不需要空白。
- (idiomatic.js 7.B) 提前回傳可以增加程式碼可閱讀性，對於效率沒有明顯差異。

```
// 不好的寫法:
function returnLate( foo ) {
  var ret;

  if ( foo ) {
    ret = "foo";
  } else {
    ret = "quux";
  }
  return ret;
}

// 好的寫法:

function returnEarly( foo ) {
```

```
  if ( foo ) {
    return "foo";
  }
  return "quux";
}
```

常見問答

函式要用那種定義方式比較好？

由上面的內容來說，有三種定義函式的方式，一種是傳統的函式定義(FD)，另一種是用常數或變數指定匿名函式的方式(函式表達式, FE)，最後一種是新式的箭頭函式，這三種的範例如下:

```
//函式定義
function sum(a, b){
  return a+b
}

//函式表達式
const sum = function(a, b) {
  return a+b
}

//箭頭函式
const sum = (a, b) => a+b
```

那麼，到底那一種是比較建議的方式？

首先，由於第二種方式(函式表達式)完全可以被箭頭函式取代，箭頭函式又有另外的好處(綁定 `this`)，所以它幾乎可以不用了。

而第一種方式(函式定義)有一些特點，所以它會被用以下的情況:

- 全域作用範圍
- 模組作用範圍
- Object.prototype的屬性值

函式定義的優點:

- 函式定義名稱可以加入到執行期間的呼叫堆疊(call stack)中，除錯方便
- 函式定義可以被提升，也就是可以在定義前呼叫(請參考上面的說明內容)

除此之外，都使用第三種方式，也就是箭頭函式。

> 請參考When should I use Arrow functions in ECMAScript 6?

arguments物件還要用嗎？

當然是不要用。這東西是有設計缺陷的，除非你對它很了解，不然用了也可能會出問題，不過話說回來，如果你對它很了解的話，就不會想用它了。改用"其餘參數"就可以了。

IIFE語法結構還要用嗎？

看情況而定。如果是有一些函式庫例如jQuery中的擴充方式會用到，這當然避免不了。如果是其他的已經採用新式的模組系統，可能是根本不需要。

英文解說

Scope/史溝波/ 中文有"視野"、"導彈範圍"的意思。也就是相當於程式語言中，看不看得到(能不能存取得到)的意思。一般中文會翻譯成"作用域"或"作用範圍"，有點難懂的文言文。與作用範圍(scope)相關的還有一個Namespace(命名空間)，這是一種語法結構或組織方法，讓程式設計師可以把不同的識別符與程式碼敘述，放到不同的"空間"之中，以免造成衝突或混亂。不過，JavaScript語言中並沒有內建的命名空間(Namespace)的特性。

Context/康鐵斯/ ，中文有"上下文"、"環境"的意思。在程式語言中指的是程式碼的執行上下文內容，它與作用範圍相關，但不相等於作用範圍。這個名詞有時會和Scope一起拿出來比較，在JavaScript語言中它也是很重要的一個概念。我們在物件中將會再說明它的意思。以下是基本的比較:

- Scope是屬於以函式為基礎的(function-based)。而Context則是以物件為基礎的(object-based)。
- Scope指的是在程式碼函式中的變數的可使用範圍。而Context指的是 `this` ，也就是指向擁有(或執行)目前執行的程式碼的物件。

家庭作業