# Reference

1. 從ES6開始的JavaScrip學習生活 (<https://eyesofkids.gitbooks.io/javascript-start-from-es6/content/>)

# 7/24

1. JaveScrip特殊的原始資料類型
   1. null
   2. undefined
   3. Symbol

null會用來當作一種運算後的特殊情況回傳值。而未定義(undefined)則常使用的是加上typeof運算符的判斷方式。

1. JavaScript採用了與其他程式語言不同的邏輯運算的回傳值設計，我們把這兩個運算符稱為"短路求值(Short-circuit)"的運算符。實際上JavaScript中，在經過邏輯與(&&)與邏輯或(||)的運算後，它的回傳值是 - 最後的值(Last value)

邏輯或(Logical OR)(||)運算符在運算時，如果當第1個運算子為"falsy"時，則回傳第2個運算子。否則，將會回傳第1個運算子。

console.log('foo' || 'bar') // 'foo'

console.log(false || 'bar') // 'bar'

console.log( 0 || '' || 5 || 'bar') //5

console.log(false || null || '' || 0 || NaN || 'Hello' || undefined)

1. 二個等號(==)時稱為相等(equal)，它是比較運算符號，用於比較兩個變數or常數的其中的值是否相等。它的反義符號是(!=)。

三個等號(===)時稱為(identity)，它也是比較運算符號，用於比較兩個變數or常數是否完全相等(包含類型與值)。它的反義符號是(!==)。

1. switch有一種講法是相當於多個if...else的組合語句，不過這個理解是有些問題。實際上，它最常被使用的是在完全一致相等值(===)比較的情況下。

# 7/25

1. for...of語句 (類似foreach)

for...of語句是新的ES6語句，可以用於可迭代物件上，取出其中的值，可迭代物件包含陣列、字串、Map物件、Set物件等等。簡單的範例如下:

let aArray = [1, 2, 3]

for (let value of aArray) {

console.log(value)

}

let aString = "abcd";

for (let value of aString) {

console.log(value);

}

1. Javascrip的函式可分為兩種  
   1. 匿名函式 function() {}

2. 具名稱函式 function foo() {}

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。

//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

}

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

1. 在javascrip函式可做為傳入參數

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。

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，以此作為辦別。

const addOne = function(value){

return value + 1

}

const addOneAndTwo = function(value, fn){

return fn(value) + 2

}

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

1. 不固定傳入參數(Variadic)與其餘(rest)參數

下列範例中，函式sum函定義了三個傳入參數，但如果真正在呼叫函式時傳入的參數值(arguments)語定義的參數數量不同，會發生什麼情況？

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。

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

1. 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

1. 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

1. 提升(Hoisting) (let, const 好像沒有hoisting的特性)

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

1. 提升(Hoisting)特性整理 (let, const 好像沒有hoisting的特性)
   1. 所有的定義(var, let, const, function, function\*, class)都會被提升
   2. 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
   3. 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。
2. 陣列的宣告方式

const aArray = []

const bArray = [1, 2, 3]

console.log(aArray.length) //0

aArray[0] = 1

aArray[1] = 2

aArray[2] = 3

aArray[2] = 5

console.log(typeof aArray) // object

console.log(aArray) // [1,2,5]

console.log(aArray.length) //3

console.log(aArray[3]) //undefined

1. 陣列易混淆的宣告方式

//這是錯誤示範

const aArray = [10]

實際上這相當於:

const aArray = []

aArray[0] = 10

1. 複製陣列的特性

感覺JS的陣列類似指標，看下面的範例:

const aArray = [1, 2, 3]

const bArray = aArray

aArray[0] = 100

console.log(bArray) //[100, 2, 3]

bArray與aArray是共享同一陣列中的值，就像是不同名字的連體嬰。

1. 為了只取得原陣列的數值(shallow copy)，有下列四種方式可使用
2. 展開(spread)運算符

ES6後的新運算符，語法相當簡單，現在很常被使用:

const aArray = [1, 2, 3]

const copyArray = [...aArray]

它也可以用來組合陣列

const aArray = [1, 2, 3]

const bArray = [5, 6, ...aArray, 8, 9]

註: 展開(spread)運算符目前用babel轉換為ES5相容語法時，是使用concat方法

1. slice

slice(分割)原本是用在分割陣列為子陣列用的，當用0當參數或不加參數，相當於淺拷貝，這方式是目前效率較好的方式:

const newArray = oldArray.slice(0)

const newArray = oldArray.slice()

1. concat

concat(串聯)是用於合併多個陣列用的，把一個空的陣列和原先陣列合併，相當於拷貝的概念。

const newArray = [].concat(oldArray)

1. for/while迴圈語句

迴圈語句也可以作為淺拷貝，寫起來不難也直覺，只是要多打很多字，通常不會單純只用來作淺拷貝。以下為範例程式:

const newArray = []

for (let i = 0, len = oldArray.length ; i < len ; i++){

newArray[i] = oldArray[i]

}

const newArray = []

let i = oldArray.length

while (i--){

newArray[i] = oldArray[i]

}

# 7/26

1. Array的屬性與方法 – length

* length用來回傳陣列的長度(成員個數)
* length的整數值竟然是可以更動的，它並不是只能讀不能寫的屬性:

const bArray = [1, 2, 3]

console.log(bArray.length)

bArray.length = 4

console.log(bArray.length) //4

console.log(bArray) //1,2,3,undefined

bArray.length = 2

console.log(bArray.length) //2

console.log(bArray) // 1,2

* 使用bArray.length = 0，與清空陣列同效果

1. 陣列與字串的轉換 join與split

* Join :把陣列中的成員(值)組合為一個字串。

const aArray = ['Hello', 'Hi', 'Hey']

const aString = aArray.join() //Hello,Hi,Hey

const bString = aArray.join(', ') //Hello, Hi, Hey

const cString = aArray.join('') //HelloHiHey

* Split : 把字串拆解成陣列的成員，拆解時需要給定一個拆解基準的符號(或空白)，通常是用逗號(,)分隔。

const monthString = 'Jan,Feb,Mar,Apr,May'

const monthArray1 = monthString.split(',') //["Jan", "Feb", "Mar", "Apr", "May"]

//以下為不好的示範

const monthArray2 = monthString.split() //["Jan,Feb,Mar,Apr,May"]

const monthArray3 = monthString.split('') //["J", "a", "n", ",", "F", "e", "b", ",", "M", "a", "r",

1. 物件(Object)

用於資料描述的物件定義，使用{}作為區塊宣告，其中加入"鍵-值"成對值，屬性的值可以包含陣列、函式或其他物件。

const bObject = {

firstKey: 'foo',

secondKey: 'bar'

}

bObject.thirdKey = 'yes'

console.log(bArray[2]) //yes

console.log(bObject.thirdKey) //yes

上面的程式碼雖然在thirdKey不存在時，會自動進行擴充，但通常物件的定義是在使用前就會定義好的，總是要處於可預測情況下是比較好的作法。

1. 類別(Class)

類別(Class)是先裡面定義好物件的整體結構藍圖，類別在定義時並不會直接產生出物件，要經過實體化的過程(new運算符)，才會產生真正的物件實體。

class MyClass{

constructor(val){

this.menber = val

this.menber2 = '你好'

}

print(){

console.log("print by myclass")

}

}

let classs = new MyClass(99)

classs.men = 10

console.log(class.men) //10

console.log(classs.menber2) //'你好'

classs.print() //"print by myclass"

1. getter與setter
2. Getter and Setter (getter不會有傳入參數，setter只會有一個傳入參數)

在類別定義中可以使用get與set關鍵字，作為類別方法的修飾字。只有私有屬性或特殊值，才需要用這兩種方法來作取得或設定。

class Option {

constructor(key, value, autoLoad = false) {

if (typeof key != 'undefined') {

this['\_' + key] = value;

}

this.autoLoad = autoLoad;

}

get color() {

if (this.\_color !== undefined) {

return this.\_color

} else {

return 'no color prop'

}

}

set color(value) {

this.\_color = value

}

}

const op1 = new Option('color', 'red')

op1.color = 'yellow'

const op2 = new Option('action', 'run')

op2.color = 'yellow' //no color prop

1. Static

JavaScrip提供靜態方法，沒有靜態屬性，使用的是static作為方法的修飾字詞。

class CStudent{

constructor(){

CStudent.countStudent()}

static countStudent(){

if(this.counter === undefined){this.counter = 1}

else{this.counter++}}

static get getCounter(){return this.counter}

}

let test1 = new CStudent()

let test2 = new CStudent()

let test3 = new CStudent()

console.log(CStudent.getCounter) //3

1. 繼承

用extends關鍵字可以作類別的繼承，而在建構式中會多呼叫一個super()方法，用於執行上層父母類別的建構式之用。super也可以用於指向上層父母類別，呼叫其中的方法或存取屬性。

繼承時還有有幾個注意的事項:

* 繼承的子類別中的建構式，super()需要放在第一行，這是標準的呼叫方式。
* 繼承的子類別中的屬性與方法，都會覆蓋掉原有的在父母類別中的同名稱屬性或方法，要區為不同的屬性或方法要用super關鍵字來存取父母類別中的屬性或方法

class Point {

constructor(x, y) {

this.x = x

this.y = y

}

toString() {

return '(' + this.x + ', ' + this.y + ')';

}

}

class ColorPoint extends Point {

constructor(x, y, color) {

super(x, y)

this.color = color

}

toString() {

return super.toString() + ' in ' + this.color

}

}

# 7/27

1. 在Javascript中，物件(Object)與類別(Class)有指標的特性，所以需以特別的方式進行淺拷貝(shallow copy)

Object.assign()

Object.assign()的用法很直覺，它除了拷貝之外，也可以作物件的合併，合併時成員有重覆名稱以愈後面的物件中的成員為主進行覆蓋:

//物件拷貝

const aObj = { a: 1, b: 'Test' }

const copy = Object.assign({}, aObj)

console.log(copy) // {a: 1, b: "Test"}

//物件合併

const bObj = { a: 2, c: 'boo' }

const newObj = Object.assign(aObj, bObj)

console.log(newObj) //{a: 2, b: "Test", c: "boo"}

1. 檢查物件(Object)、類別(Class)內屬性或方式的方法

* Undefined判對方式 (快，但當這個屬性本身就是undefined時，這個判斷方法就失效了)

//判斷鍵是否存在

typeof obj.key !== 'undefined'

//判斷值是否存在

obj.key !== undefined

obj['key'] !== undefined

* in 與 hasOwnProperty

obj.hasOwnProperty('key')

'key' in obj

hasOwnProperty方法只會檢查這個物件中有的屬性鍵，用類別定義時的方法是沒辦法檢測到，由原型繼承的方法也沒辦法檢測到

，以下為範例:

const obj ={}

obj.prop = 'exists'

console.log(obj.hasOwnProperty('prop') ) //ture

console.log(obj.hasOwnProperty('toString')) // false

console.log(obj.hasOwnProperty('hasOwnProperty')) // false

console.log('prop' in obj) //ture

console.log('toString' in obj) //ture

console.log('hasOwnProperty' in obj) //ture

1. 物件的遍歷(traverse)

for...in語句

for...in語句是用來在物件中以鍵(key)值進行迭代，因為是無序的，所以有可能每次運算的結果會不同。它通常會用來配合hasOwnProperty作判斷，主要原因是in運算符和前面在判斷時一樣，它會對所有原型鏈(prototype chain)都整個掃過一遍，hasOwnProperty可以限定在物件本身的屬性。

for(let key in obj){ //此key的type為string

if (obj.hasOwnProperty(key)) {

console.log(obj[key]);

}

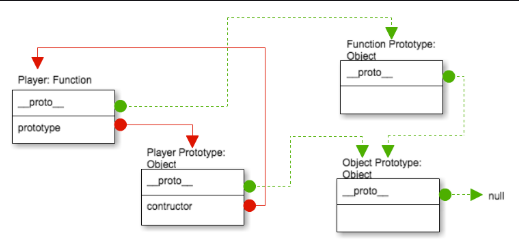
}

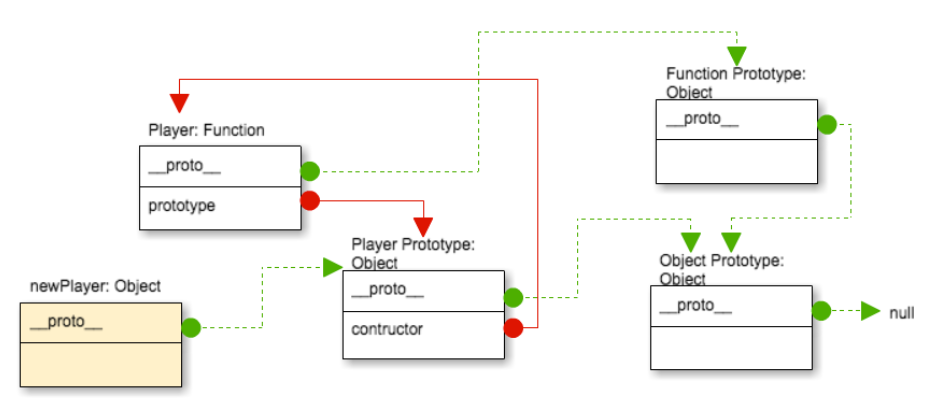
1. 日期與時間整理

Date在JavaScript中是一個內建的特殊物件，不只包含所謂的日期 - 年月日而已，它也包含時間中的時、分、秒與微秒。

* Date物件一定只能使用new運算符來建立
  + const date = new Date()
  + const date = new Date(milliseconds)
  + const date = new Date(dateString) //只有這種方式是傳入字串類型，其他都不是。(建議使用通用的格式 例 “2018/07/27 06:30:28”)
  + const date = new Date(year, month, day, hours, minutes, seconds, milliseconds)
* 輸出時間字串方法
  + const datetime = new Date()
  + console.log(datetime) //Tue Jul 12 2016 11:35:48 GMT+0800 (CST)
  + console.log(datetime.toString()) //Tue Jul 12 2016 11:35:48 GMT+0800 (CST)
  + console.log(datetime.toUTCString()) //Tue, 12 Jul 2016 03:35:48 GMT
  + console.log(datetime.toTimeString()) //11:35:48 GMT+0800 (CST)
  + console.log(datetime.toLocaleString()) //2016/9/14 下午4:33:41
  + console.log(datetime.toLocaleDateString()) //2016/9/14
  + console.log(datetime.toLocaleTimeString()) //下午4:33:41
* 取得或設定日期時間其中某個值
  + const dateObject = new Date()
  + dateObject.setFullYear(2015) //設定
  + dateObject.setMonth(0)
  + dateObject.setDate(24)
  + dateObject.setHours(2)
  + dateObject.setMinutes(20)
  + dateObject.setSeconds(60)
  + dateObject.setMilliseconds(150)
  + const date = dateObject.getDate() //取值
  + const day = dateObject.getDay()
  + const month = dateObject.getMonth()
  + const year = dateObject.getFullYear()
* Date時間屬性的範圍
  + Date: 日期的值，範圍為1-31
  + FullYear: 年(西元年)，為4位數數字
  + Day: 星期幾的值，範例為0-6
  + Month: 月份的值，範例為0-11

# 7/31

1. 原型物件導向整理
2. 利用圓型鏈(prototype chain)串連物件，以建立原型繼承的連接結構，即原型物件導向(自己亂解釋的….)
3. 以函式說明圓型物件導向
   1. 所有物件皆繼承Object prototype
   2. 每個物件皆有\_\_proto\_\_(鏈結到此物件的原型)與constructor
   3. 每個函式都有prototype(鏈結到此函式的constructor)
   4. 若定義一變數為let newPlayer = new Player()，其鏈結方式如下



1. 用IIFE模擬出靜態變數的結構:

var Employee = (function() {

var sharedVariable = 0;

function Employee() {

++sharedVariable;

console.log("sharedVariable = " + sharedVariable);

}

return Employee;

})();

new Employee(); //1

new Employee(); //2

new Employee(); //3

new Employee(); //4

1. call 與 bind

* call (以個別提供的this值與傳入參數值來呼叫函式)

function func(param1){

console.log('func', this)

}

const objA = {

methodA(){

console.log('objA methodA', this)

}

}

const objB = { a:1, b:2 }

func.call(objB) //func Object {a: 1, b: 2}

objA.methodA.call(objB) //objA methodA Object {a: 1, b: 2}

* bind (以傳入參數設定為函式的輸入值)

function funcA(param1, param2){

console.log(this, param1, param2)

}

const objB = { a: 1, b: 2 }

funcA() //undefined undefined undefined

const funcB = funcA.bind(objB, objB.a)

funcB() //Object {a: 1, b: 2} 1 undefined

funcB(objB.b) //Object {a: 1, b: 2} 1 2

1. 異步回調函式 (可避免阻塞的現象)

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)

異步回調函式會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。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。

1. 閉包(closure)小整理

* 所有函式在建立時都會產生閉包。
* 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。

1. 柯里化(Currying)與部份應用(Partial application)

* 兩者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。

**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。

* 皆下來以下列函式作介紹

function add(x, y, z){

return x+y+z

}

* Partial application - 1

function addXY(z){

return add(1, 2, z)

}

addXY(3)

* Partial application - 2

//改寫

function add(x, y, z){

return function(z){

return x+y+z

}

}

const addXY = add(1, 2)

addXY(3)

* Currying

function add(x, y, z){

return function(y){

return function(z){

return x + y + z

}

}

}

add(1)(2)(3)