

JavaScript 開發實戰：核心概念篇

全新 ES6 改版



多奇數位創意有限公司

技術總監 黃保翕 (Will 保哥)

部落格：<http://blog.miniasp.com/>

課程大綱

- 物件、變數與型別
- JavaScript 物件概念
- 深入理解 JavaScript 函式物件
- 更多 ES2015 / ES6 全新語言特性

JavaScript 物件可以指派給一個變數並會在執行時期擁有型別

物件、變數與型別



何謂「物件」

- 關於物件的定義
 - In computer science, an object is a value in memory which is possibly referenced by an identifier.
- 名詞解釋
 - computer science (電腦科學領域)
 - value in memory (在記憶體中的資料)
 - referenced by an identifier (被一個**識別符號**所參考)
 - JS 合法的**識別符號**必須是 英文字母、數字、金額符號 (\$)、底線 (_)
 - JS 合法的**識別符號**不能以 數字 開頭

JavaScript 是個物件導向程式語言

- 所有物件都是物件型別 (Object Type) , 除了以下 6 種 :

- number (數值)

- string (字串)

- boolean (布林)

- null (空值)

- undefined (未定義)

- symbol (符號) **ES6+**

原始型別 (Primitive Type)

物件型別的基本特性 (擁有屬性)

- 屬性 (Property)
 - 任何一個 JavaScript 物件都只會有一**種成分**，那就是**屬性**！
 - **屬性**的特性跟**變數**很像，你可以指派**任意資料**給**任意屬性**！

- 建立物件

```
var car = {}; // 物件 (Object)
```

- 擴增屬性

```
car.name = "Tesla"; // 屬性 (Property)  
car.start = function () { // 函式 (Function)  
    return "OK";  
};
```

JavaScript 取得物件屬性資料的方式

- 建立一個物件

```
var car = {  
  name: 'Tesla',  
  miles: 20000,  
  start: function() {  
    return 'OK';  
  },  
  '001': 'LogEntry#1'  
};
```

- 取得 name 屬性的方式
 - car.name
 - car['name']
- 執行 start 函式的方式
 - car.start()
 - car['start']();
- 取得 001 屬性的方式
 - car.001 *SyntaxError*
 - car['001']

取得網頁上第一個表單的 DOM 物件

- `window.document.forms[0]`
- 以上這行程式你還能想到幾種寫法？
 - <http://utf-8.jp/public/aaencode.html>
 - <http://utf-8.jp/public/jjencode.html>
 - <http://utf-8.jp/public/jsfuck.html>
 - <http://www.jsfuck.com/>
 - <http://patriciopalladino.com/files/hieroglyphy/>

JavaScript 是個動態型別語言

- 我們可以使用 `var` / `let` / `const` 宣告變數

```
var x;           // 沒有給值的變數預設為 undefined
```

```
x = 5;
```

```
var x;
```

```
typeof(x)        // 請問這行的 x 型別是什麼？
```

```
x = "Will"
```

- JavaScript 的動態型別特性
 - 無型別 (Untyped) 無法在**開發時期**宣告型別
 - 弱型別 (Weak-typed) 只能在**執行時期**檢查型別

物件、變數與型別之間的關係

- 物件 (Object)
 - 僅存在於**執行時期**
 - 這裡的物件代表的是一種存在於記憶體中的**資料**
- 變數 (Variable)
 - 只能在**開發時期**進行宣告 (使用 `var/let/const` 關鍵字)
 - 在**執行時期**只會用來儲存物件的**記憶體位址** (類似指標)
- 型別 (Type)
 - 僅存在於**執行時期**，並用來標示物件的種類 (類型)
 - 不同型別之間可能會有不同的預設**屬性與方法**

物件、變數與型別之間的關係 (範例)

- 以下 4 行程式碼在執行的過程中，請問：

1. 曾經在**記憶體中**建立過幾個**變數**？
2. 曾經在**記憶體中**出現過幾種**型別**？
3. 曾經在**記憶體中**出現過多少**物件**？

```
var a;
```

```
a = 1;
```

```
a = "a";
```

```
a = "a" + a;
```

練習 1：物件、變數與型別之間的關係

變數與屬性之間的關係

屬性	變數
以 執行時期 為主	以 開發時期 為主
擁有 指標 特性 (可以指向任意資料)	擁有 指標 特性 (可以指向任意資料)
屬性只會存在於 特定物件 下 屬性可以透過 <u>delete</u> 刪除	只有用 var / let / const 宣告的才能算變數 變數只存在於 當前範圍 下，而且 不能刪除

當前範圍 (Scoping) (又稱作用域範圍)

- 對瀏覽器來說
 - 在「**全域範圍**」宣告的變數，通稱為「**全域變數**」(Global Variables)
 - 所有不在 `function` 內執行的程式碼，都屬於「**全域範圍**」的程式碼
 - 所有在 `function` 內宣告的變數，通稱為「**區域變數**」(Local Variables)
- 對 Node.js 來說
 - 每個 JS 檔案就是一個 **模組** (module)
 - 每個 **模組** 都會擁有一個 **變數** 的 **作用域範圍**
 - 每個 **函式** 也會擁有一個 **變數** 的 **作用域範圍**
 - 嚴格來說 Node.js 沒有「**全域變數**」的概念
 - 但 Node.js 有 `global` 可用來設定「**類全域變數**」

執行時期的變數特性 (1)

- 變數被限定在當前範圍下，同時存於當前範圍的物件容器中

```
a = 1;           // 不用 var/let/const 宣告的都不能算變數  
console.assert(window.a == 1);
```

```
var a = 1;       // 全域範圍使用 window 物件容器  
console.assert(window.a == 1);
```

```
function test() {  
    var a = 1; // 區域範圍使用無法存取的 AO 物件容器  
    console.assert(window.a == 1);  
}  
test();
```

執行時期的變數特性 (2)

- 變數宣告之後就絕對無法刪除

```
a = 1;  
delete a;  
console.assert(window.a == 1);
```

```
var a = 1;  
delete a;  
console.assert(window.a == 1);
```

```
window.a = 1;  
delete a;  
console.assert(window.a == 1);
```

隨堂測驗：變數與屬性之間的關係 (1)

- 變數

- `var a = 1;`
- `var b = a;` // 請問這兩行在記憶體中建立過幾個物件？

- 屬性

- `var a = b;` // 請問執行完這行會得到什麼？
- `var a = window.b;`
- `b = 1;`
- `window.b = 1;`
- `delete b;`
- `var a = b;` // 請問執行完這行會得到什麼？

隨堂測驗：變數與屬性之間的關係 (2)

- 以下程式碼片段，請問輸出為何？

```
var a = 1;  
window['a'] = 2;  
delete window.a;  
console.log(a);
```

[練習 2：變數與屬性之間的關係](#)

```
b = 2;  
delete b;  
console.log(b);
```

ES6 變數與常數

- 使用 var 區域變數
 - 屬於 function scope 且容易遭遇 Hoisting 的問題
- 使用 let 區域變數
 - 屬於 block scope 且**同一範圍**不允許重複宣告變數
 - 用 var 宣告過的變數，不能再使用 let 宣告一次
 - 用 let 宣告變數之後才能開始使用 (變數特性與 C# 非常類似)
- 使用 const 區域變數
 - 宣告一個**唯讀的變數** (變數無法再指向其他物件)
 - 宣告完變數後必須立刻初始化變數 (給予變數預設值)
 - 變數作用域範圍與 let 完全相同 (block scope)

觀念驗證 1

- 請問執行以下程式是否會發生錯誤？

```
const a = {};
```

```
a.name = 'Will';
```

觀念驗證 2

- 請問執行以下程式是否會發生錯誤？

```
function f(type) {  
    var a = 1;  
    if (type == 1) {  
        let a = 2;  
    }  
}
```

```
f(2);
```

觀念驗證 3

- 請問執行以下程式的回傳值為何？

```
(function () {  
    let a = 1;  
    var test = function() {  
        console.log(a);  
        let a = 2;  
        console.log(a);  
    }  
    return test();  
})();
```

觀念驗證 4

- 請問執行以下程式的回傳值為何？ (Shadowing)

```
(function () {  
    let matrix = [ [1,2,3], [1,2,3], [1,2,3] ];  
    let sum = 0;  
    for (let i = 0; i < matrix.length; i++) {  
        var currentRow = matrix[i];  
        for (let i = 0; i < currentRow.length; i++) {  
            sum += currentRow[i];  
        }  
    }  
    console.log(sum);  
})();
```

觀念驗證 5

- 請問執行以下程式的回傳值為何？

```
(function () {  
    let getCity;  
    let city = "Taiwan";  
    if (true) {  
        let city = "Seattle";  
        getCity = function() {  
            return city;  
        }  
    }  
    return getCity();  
})();
```

觀念驗證 6

- 請問以下兩段程式分別的執行結果為何？

```
(function () {  
  for (let i = 0; i < 10 ; i++) {  
    setTimeout(function() { console.log(i); }, 100 * i);  
  }  
})();
```

```
(function () {  
  for (var i = 0; i < 10 ; i++) {  
    setTimeout(function() { console.log(i); }, 100 * i);  
  }  
})();
```


了解不同型別之間的特性

型別系統



JavaScript 型別有兩大分類

- 原始型別 (Primitive Data Types)

- number
- string
- boolean
- null
- undefined
- symbol

無法「自由擴增屬性」

- 物件型別 (Object Data Types)

- 原生物件
- 宿主物件

可以「自由擴增屬性」

原始型別的基本特性 (不允許擁有屬性)

- 宣告變數並指派原始型別資料

```
var a = 1;
```

- 擴增屬性

```
a.name = 'Will';
```

- 刪除屬性

```
delete a.name;
```

- 判斷屬性是否存在

```
if ('c' in obj) { }
```

物件型別資料下的屬性操作

- 宣告變數並指派物件型別資料

```
var obj = { 'a': 1, 'b': 2 };
```

- 擴增屬性

```
obj.c = 3;
```

- 刪除屬性

```
delete obj.c;
```

- 判斷屬性是否存在

```
if ( 'c' in obj ) { }
```

```
typeof(obj.c) == 'undefined'
```

➔ 這行有可能是 Bug

原始型別資料下的屬性操作

- 宣告變數並指派原始型別資料

```
var a = 'Will';
```

- 讀取屬性

```
var b = a[0]; // 'W'
```

- 置換屬性

```
a[0] = 'J';
```

- 判斷屬性值

```
a[0]
```

```
a
```

原始型別包裹物件

- 主要用途
 - 由於**原始型別**無法自由擴增屬性
 - 透過包裹物件後，可透過**物件型別**的特性，自由擴增**屬性**與**方法**
 - String, Number, Boolean, Object
- 共通方法
 - valueOf()
 - 取得**物件**內部的**原始值** (Primitive Value)
 - 推薦文章：[前端工程研究：關於 JavaScript 中物件的 valueOf 方法](#)
 - toString()
 - 取得**物件**內部的**原始值** (Primitive Value) 並轉換成**字串**型別

Primitive Data Types

原始資料型別



JavaScript 原始型別：數值 (number)

- 使用方法

```
var a = 100;    var a = 10e5;
```

```
var a = 0100;   var a = 0xFFFF;   var a = 0b010101101;
```

```
var a = new Number(100);
```

```
var a = Number('100');    ➔ 轉型
```

```
var a = Number('100a'); ➔ 轉型失敗 ➔ NaN
```

Not a Number is a Number!



```
var a = parseInt('100 aa', 10); ➔ 解析字串為數值型態
```

```
var a = Number.NaN
```

- 原始型別包裹物件

- Number 內建屬性與方法說明

- [數值 - JavaScript | MDN](#) / [Number - JavaScript | MDN](#)

數值型別常見的使用技巧

- 使用 `parseInt` / `parseFloat` 一律加上第 2 個參數
`parseInt('070');` *// IE8 以下會變成 8 進制*
`parseInt('070', 10)` *// 這個才是建議的寫法！*
- 使用 `+` 引發型別自動轉換 (自動轉成 `number` 型別)
`var a = +'7';`
`var b = +(a);`
- 使用 `(N).toString(baseN)` 轉換進制
`(0xAF).toString(10)`
`65535..toString(16)`
- 判斷 `NaN` 的唯一寫法
`Number.isNaN(NaN)`

數值型別的整數運算特性

- 最大安全整數 = 2^{53} (判斷**最大安全整數**非常重要)

`Number.MAX_SAFE_INTEGER == 2**53-1`

`Number.MAX_SAFE_INTEGER+1`

`Number.MAX_SAFE_INTEGER+2`

- 浮點數系統可保存極大數值 (但有精準度問題)

`Number.MAX_VALUE == 1.7976931348623157e+308`

`Number.MIN_VALUE == 5e-324`

- 判斷無限大數值的表示方法

`var a = 1 / 0;`

`Number.POSITIVE_INFINITY == Infinity`

`Number.NEGATIVE_INFINITY == -Infinity`

關於 **BigInt** 型別

- Chrome 67+ / Opera 54+ 開始支援 [BigInt](#) 型別 ([相容性](#))

- 基本用法

```
var a = 39837212195743250943287503298475432n
```

```
var a = BigInt('39837212195743250943287503298475432')
```

- 錯誤用法

```
BigInt(39837212195743250943287503298475432)
```

```
BigInt(1.5)           // RangeError
```

```
BigInt('1.5')         // SyntaxError
```

```
1 + 1n                // TypeError: 不能混用型別
```

```
new BigInt(123)       // TypeError: 不能用 new 建立物件
```

更多 **BigInt** 型別的使用範例

- 比較運算

```
42n == 42           // true
42n === 42          // false
123 < 124n          // true
42n === BigInt(42)  // true
typeof 42n === 'bigint' // true
```

- 數學運算

```
const num = BigInt(Number.MAX_SAFE_INTEGER);
```

<pre>var a = num + 1n</pre>	<pre>var a = num * 1n</pre>
<pre>var a = num - 1n</pre>	<pre>var a = num / 10n (只取整數)</pre>
<pre>var a = num ** 2n</pre>	<pre>var a = num % 10n</pre>

數值型別的浮點數運算特性

- `0.1 + 0.2 != 0.3`
- `0.2 + 0.4 != 0.6`
- `0.3 + 0.6 != 0.9`
- `0.4 + 0.8 != 1.2`
- `0.123123123123123123123123 == 0.123123123123123123123124`
- `Math.ceil(0.1*0.2*100)`
- `Math.ceil(0.1*0.2*1000)`
- `Math.ceil(0.1*0.2*10000)`
- `parseInt(0.000001)`
- `parseInt(0.0000001)`

數值型別處理浮點數的注意事項

- 自動取整數
 - `parseInt(2.99)`
 - `~~2.99`
- 自動取小數幾位
 - `parseInt(2.99 * 10) / 10`
 - `+(2.999).toFixed(3)`
- 四捨五入 (負數超過 0.5 才會進位)
 - `Math.round(2.5) == 3`
 - `Math.round(-2.5) == -2`
 - `Math.round(-2.51) == -3`
- 無條件捨去 (小於自己的最大整數)
 - `Math.floor(2.99) == 2`
- 無條件進位 (大於自己的最小整數)
 - `Math.ceil(2.11) == 3`
- 推薦學習
 - [Math - JavaScript | MDN](#)
- 推薦數值相關 JS 套件
 - [Numeral.js](#) ([GitHub](#))
`numeral('10,000.12').value()`
`numeral('$10,000.00').value()`
 - [decimal.js](#) ([GitHub](#))
`+Decimal(0.1).add(Decimal(0.2))`
 - [bignumber.js](#) ([GitHub](#))
 - [big.js](#) ([GitHub](#))
 - [What is the difference between big.js, bignumber.js and decimal.js?](#)

JavaScript 原始型別：字串 (string)

- ES5 字串表示法

```
var a = 'Will';
```

```
var b = "Will";
```

- 字串長度與字元

```
a.length
```

```
a[0] === 'W'
```

- ES6 字串表示法

```
var name = `Will`;
```

```
$('#text').html(`  
  <p>
```

```
    How are you?
```

```
  </p>
```

```
`);
```

```
var str = `Mr. ${name}`;
```

```
`\` === ``
```

字串物件與型別轉換

- 字串物件

```
var a = new String('Will');  
    String { 0: "W", 1: "i", 2: "l", 3: "l", length: 4 }  
a.length // 取得字串長度  
a[0]     // 取得第一個字元
```

- 將任意型別轉為字串型別

```
var a = String(100);
```

- 原始型別包裹物件

- String 內建屬性與方法說明

- [字串 - JavaScript | MDN](#) / [String - JavaScript | MDN](#)

JavaScript 原始型別：布林 (boolean)

- 使用方法

```
var a = true;  
var a = false;  
var a = new Boolean(false);  
var a = Boolean('false');  
var a = Boolean('0');  
var a = Boolean('');  
var a = Boolean(0);
```

- 原始型別包裹物件

- Boolean 內建屬性與方法說明

- [布林 - JavaScript | MDN](#) / [Boolean - JavaScript | MDN](#)

認識 Truthy 與 Falsy 值

- `false` , `0` , `""`

`(false == 0) === (false == "") === (0 == "") === true`

- `null` , `undefined`

※ `null` 只能跟 `null` 與 `undefined` 比較

```
var d = (null == false);           // false
var e = (null == true);            // false
var d = (undefined == false);      // false
var e = (undefined == true);       // false
var f = (null == null);            // true
var g = (undefined == undefined);  // true
var h = (undefined == null);       // true
```

- `NaN`

※ `NaN` 跟任何物件比較都是 `false`

```
var i = (NaN == null);             // false
var j = (NaN == NaN);              // false
```

布林型別使用技巧

- 隱含比對 vs. 明確比對 ([JS Comparison Table](#))
 - `==` 或 `!=` (隱含比對) (會引發自動轉型)
 - `===` 或 `!==` (明確比對) (連同型別一起判斷)
- 一律使用 `===` 或 `!==` 比對，可避免**自動轉型**造成的**邏輯錯誤**！

- 使用 `!!` 強迫引發自動轉型

```
var a = !!({});  
var b = !!("0");
```

- 判斷物件是否有初始值

```
if (myVar) { }
```

- 給予變數預設值

```
var arr = arr || [];  
var num = num || 99;  
var str = str || "";  
var bool = bool || true;  
var obj = obj || {};
```

JavaScript 原始型別：空值 (null)

- 使用方法

- `var a = null;`

- 重點觀念

- `null` 跟 `NaN` 有點類似

- `NaN` 不是一個數值，但型別卻是個數值 | `typeof(NaN) == "number"`

- `null` 不是一個物件，但型別卻是個物件 | `typeof(null) == "object"`

- 最佳實務

- 盡量不要用 `null`

- 判斷物件是否為 `null` 的正確方法

- ```
var a = null;
```

- ```
if (a === null) { }
```

JavaScript 原始型別：未定義 (undefined)

- 使用方法

```
var a;      // 變數 a 尚未定義 ( a === undefined )  
a = undefined;
```

- 重點觀念

- undefined (型別) 是一個內建型別 (原始型別)
- undefined (物件) 在執行時期的記憶體中有個物件存在
- undefined (變數/屬性) 是一個全域變數 (也是個屬性)
 - window.undefined === undefined
 - 在 IE8 以下 (含) undefined 可以被重新指派成其他物件！
- 無論 null 或是 undefined 都會隱含轉型成 false
 - undefined == null // true

JavaScript 原始型別：符號 (symbol)

- 主要目的
 - 用來作為物件**屬性名稱**的唯一識別 (unique and immutable)
- 關於物件的屬性名稱
 - ES6 之前：物件的屬性名稱只能是「字串」型別
 - ES6 以後：物件的屬性名稱可以使用 **Symbol** 型別
- 使用方法

```
var key = Symbol('name');  
var obj = { [key]: 'Tesla' };  
var name = obj[key];
```

關於 ES6 內建的 Symbol 物件

- Iteration symbols
 - [Symbol.iterator](#)
 - Symbol.asyncIterator
- Regular expression symbols
 - [Symbol.match](#)
 - Symbol.matchAll
 - [Symbol.replace](#)
 - [Symbol.search](#)
 - [Symbol.split](#)
- Other symbols
 - [Symbol.hasInstance](#)
 - [Symbol.isConcatSpreadable](#)
 - [Symbol.unscopables](#)
 - [Symbol.species](#)
 - [Symbol.toPrimitive](#)
 - [Symbol.toStringTag](#)
- Methods
 - [Symbol.key\(key\)](#)
 - [Symbol.keyFor\(sym\)](#)

關於 Symbol 重要特性

- 基本語法
 - `Symbol([符號名稱])`
 - `const symbol1 = Symbol();`
 - `const symbol2 = Symbol(42);`
 - `const symbol3 = Symbol('foo');`
- 唯一 (Unique)、不可變(Immutable)
 - `Symbol("name") !== Symbol("name")`
- 全域符號資料庫 (Global Symbol Registry) (GSR)
 - 內建的 Symbols 不會存在於 GSR
 - 透過 `Symbol.for(key)` 會建立 Symbol 物件並註冊到 GSR
 - 透過 `Symbol.keyFor(sym)` 會回傳註冊在 GSR 的 Symbol key 值 (字串)

Object Data Types

物件資料型別



JavaScript 物件型別

- 原生物件 (Native Objects)
 - 於 [ECMAScript](#) 標準中定義的物件
 - [Array](#), [Date](#), [Math](#), [RegExp](#)
 - [Number](#), [String](#), [Boolean](#), [Object](#), [Function](#)
 - [Typed Arrays](#), [Map](#), [Set](#), [WeakMap](#), [WeakSet](#), [JSON](#)
- 宿主物件 (Host Objects)
 - 由 JavaScript 執行環境 額外提供的物件
 - 瀏覽器 : [Window](#) , 所有 [DOM](#) 物件
 - Node.js : [Global](#) , [Process](#) , [OS](#) , [Net](#) , [Path](#)

JavaScript 原生物件：使用者定義物件

- 如何建立一個物件？

```
var obj = new Object();  
obj.name = 'Will';  
obj.company = '多奇數位創意有限公司';
```

- 透過 **物件實字語法** (Object Literal Syntax) 建立物件

```
var obj = {};  
obj.name = 'Will';  
obj.company = '多奇數位創意有限公司';
```

```
var obj = { 0: 'test', name: 'Will', 'tel': 'xxx' };
```

ES2015 新增的物件實字語法

```
var obj = {  
  // 直接指定上層物件 ( __proto__ )  
  __proto__: Lesson,  
  // 簡化設定屬性的語法 ( handler: handler )  
  handler,  
  // 直接宣告或覆寫屬性方法  
  toString() {  
    // 可直接呼叫上層物件的方法  
    return 'd ' + super.toString();  
  },  
  // 可在建立物件時動態計算屬性名稱  
  [ 'prop_' + test() ]: 42  
};
```

JavaScript 原生物件：Array

- 使用方法 1
 - `var mycars = new Array();`
 - `mycars[0] = "Will";`
 - `mycars[1] = "Web";`
 - `mycars.push("The");`

- 使用方法 2 (物件實字表示法)

- `var mycars = [];`
 - `mycars[0] = "Will";`
 - `mycars[1] = "Web";`
 - `mycars.push('The');`
- Array 內建屬性與方法說明

- [陣列 - JavaScript | MDN](#) / [Array - JavaScript | MDN](#)

Elements	Resources	Network	Elements	Resources	Network
> var a = [5];			> var a = [5,10];		
undefined			undefined		
> var b = new Array(5);			> var b = new Array(5,10);		
undefined			undefined		
> a			> a		
[5]			[5, 10]		
> b			> b		
[undefined × 5]			[5, 10]		
> a.length			> a.length		
1			2		
> b.length			> b.length		
5			2		
>			>		

JavaScript 原生物件：Date

- 使用方法

```
var d = new Date();
```

```
var d = new Date(milliseconds);
```

- 這是 GMT 時區的 Timestamp

```
var d = new Date(dateString);
```

```
var d = new Date(year, month, day, hours, minutes,  
                  seconds, milliseconds);
```

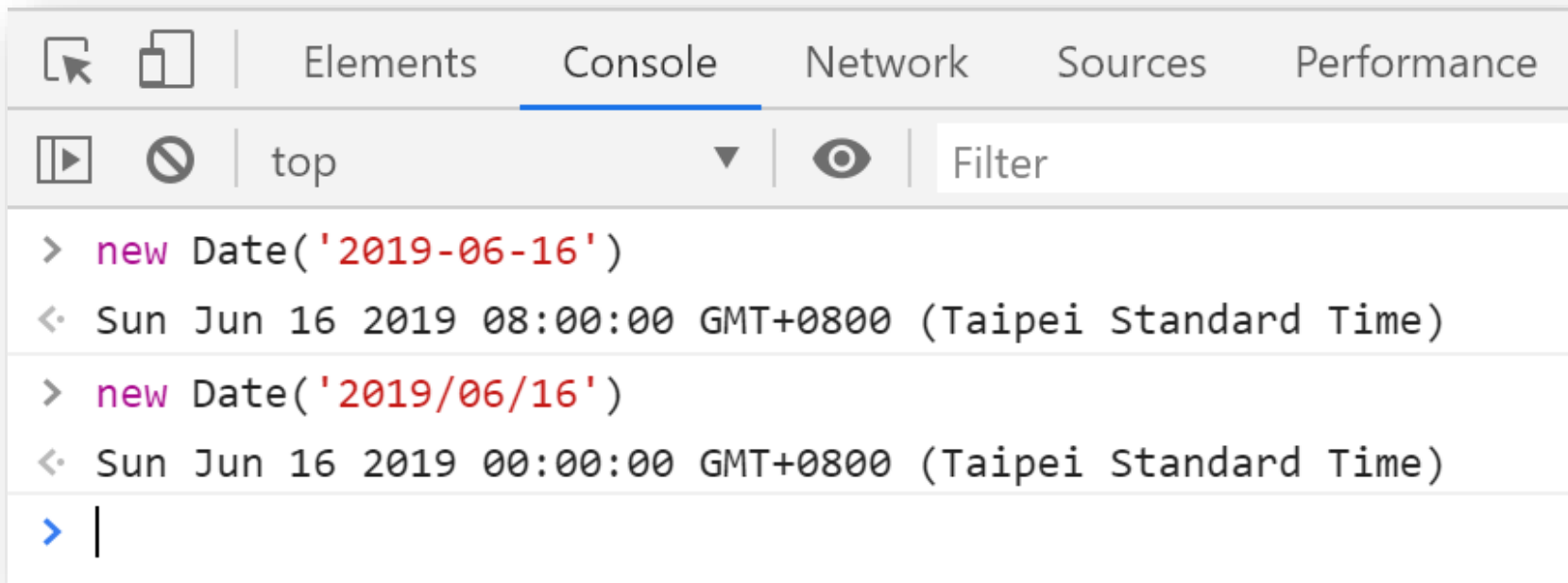
- 請注意 month 的數字範圍是從 0 ~ 11

- Date 內建屬性與方法說明

- [日期 - JavaScript | MDN](#) / [Date - JavaScript | MDN](#) || [ExtDate](#)
- [前端工程研究：關於 JavaScript 中 Date 型別的常見地雷與建議作法](#)

關於 Date 字串格式與時區的關係

```
new Date('2019-06-16')    new Date('2019-06-16 00:00:00')  
new Date('2019/06/16')    new Date('2019/06/16 00:00:00')
```



JavaScript 原生物件：Math

- 使用方法

```
var a = Math.PI;           // PI 常數
var a = Math.sqrt(16);     // 開根號(4)
var a = Math.round(2.5);   // 四捨五入(3)
var a = Math.floor(1.8);   // 無條件捨去(1)
var a = Math.ceil(1.1);    // 無條件進位(2)
```

- 靜態物件

- Math 本身是一個完全沒有屬性的物件，但可以擴充額外屬性！

- Math 內建屬性與方法說明

- [Math - JavaScript | MDN](#)

JavaScript 原生物件：RegExp

- 使用方法 1

```
var re = new RegExp(pattern, modifiers);
```

```
var re = new RegExp("[A-Z][12]\\d{8}", "igm");
```

- 使用方法 2 (物件實字表示法)

```
var re = /pattern/igm;
```

```
var re = /[A-Z][12]\\d{8}/igm;
```

- RegExp 內建屬性與方法說明

- [RegExp - JavaScript | MDN](#)

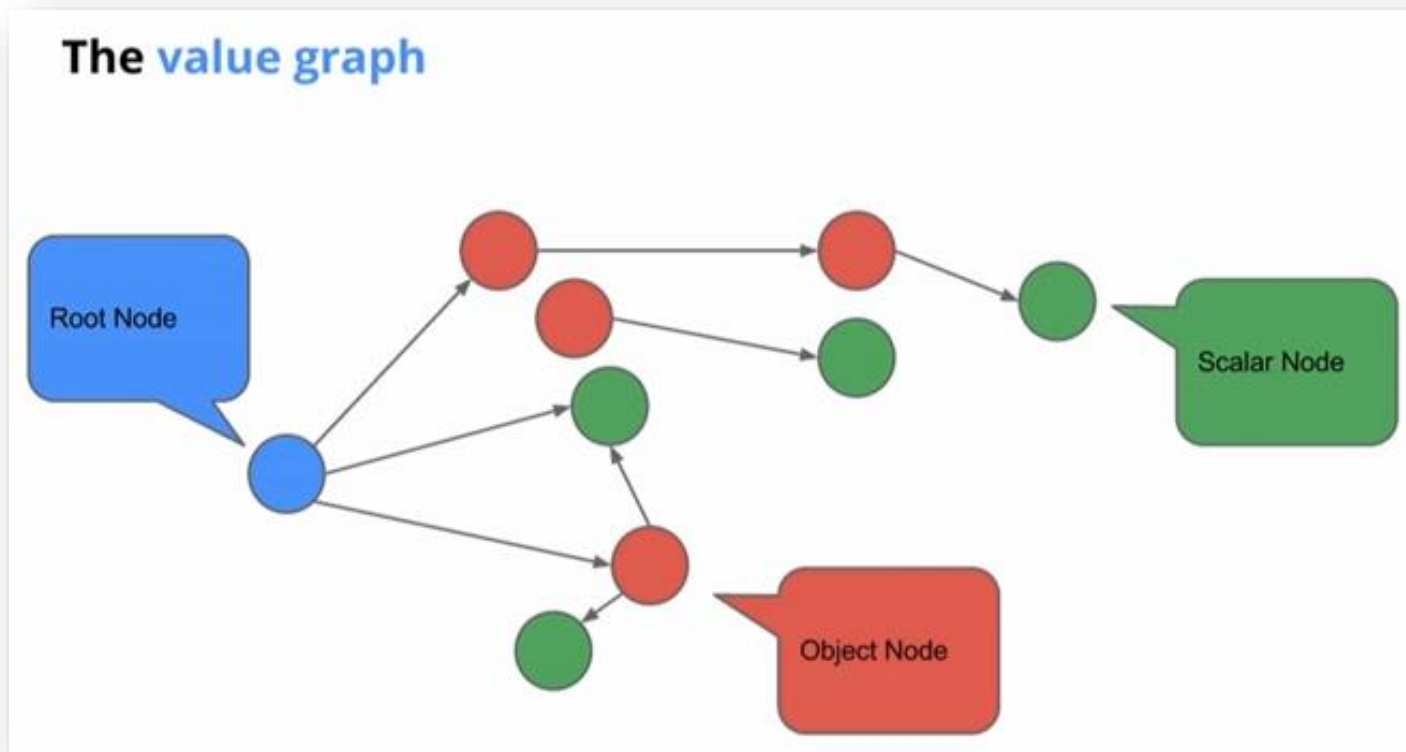
你所不知道的 JavaScript 程式語言特性

JavaScript 物件概念



JavaScript 物件資料結構

- 所有物件資料都從**根物件**開始**連結**(chain)



探討變數與物件之間的連結 (1)

- 物件

```
window.document.myobj = { 'num': 1 };
```

- 變數 / 型別

```
var o = window.document.myobj;  
typeof(o.num)
```

- 重新指派變數值

```
o.num = 2;
```

- 請問以下陳述式是否為真？

```
o == window.document.myobj
```

探討變數與物件之間的連結 (2)

- 物件

`window.document.forms[0]`

- 變數 / 型別

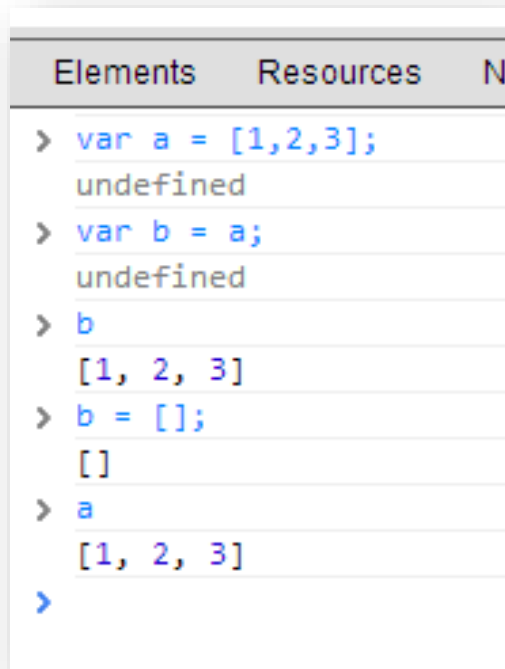
`var o = window.document.forms[0];`
`typeof(o)`

- 重新指派變數

`o = "123";`

- 請問以下陳述式是否為真？

`o == window.document.forms[0]`



Elements	Resources	N
>	<code>var a = [1,2,3];</code>	undefined
>	<code>var b = a;</code>	undefined
>	<code>b</code>	[1, 2, 3]
>	<code>b = [];</code>	[]
>	<code>a</code>	[1, 2, 3]
>		

任何 JavaScript 執行環境都有個根物件

- 瀏覽器
 - [window](#)
- Node.js
 - [global](#)
- AngularJS 1.x (非 JavaScript 執行環境，但沿用其概念)
 - [\\$rootScope](#)
- VueJS 2.x (非 JavaScript 執行環境，但沿用其概念)
 - [vm.\\$root](#)

瀏覽器的 根物件 (window) 有什麼？

- 屬性

- name
- self
- opener
- parent
- navigator
- NaN
- Infinity
- undefined

- 方法

- isNaN()
- decodeURI()
- decodeURIComponent()
- encodeURI()
- encodeURIComponent()
- eval()

- 其他根物件下的屬性

- [Standard built-in objects - JavaScript | MDN](#)

Understanding Function Objects

深入理解 JavaScript 函数物件



函式物件 (function)

- 函式物件就是一般物件外加可以被呼叫的能力

```
var a = function () { }
```

```
typeof (a)
```

```
a.x = 1;
```

```
a.x
```

- 函式物件的兩大特色
 - 一級物件 (First-class object)
 - 可以被動態建立、可以指定給變數、可以複製給其他變數
 - 可以擁有自己的屬性或方法 (一般物件特性)
 - 提供了變數的作用域 (Scope)
 - 使用 **var** 宣告變數時不以區塊 { } 建立作用域

函式的表示法

	函式表示式 (function expression)	函式宣告式 (function declaration)
具名函式	<pre>var add = function add(a, b) { return a + b; }; add.name</pre>	<pre>function add(a, b) { return a + b; } add.name</pre>
匿名函式	<pre>var add = function (a, b) { return a + b; }; add.name // 匿名函式無法取得函式名稱</pre>	<pre>function (a, b) { return a + b; } Uncaught SyntaxError: Unexpected token (</pre>

立即函式 (Immediate Function)

- IIFE = Immediately-invoked function expression
- 函式表達式 + 匿名函式

```
(function (a, b) {  
    var c = 10;  
    return a + b + c;  
})(10, 20);
```

- 主要用途
 - 限制變數存在的作用域！
 - 讓變數不輕易成為「全域變數」的方法！

回呼模式 (Callback Pattern)

- 將函式當成參數傳遞給其他函式

```
function get(url, handler) {  
    $.get(url, function(data) {  
        handler(data);  
    });  
}
```

- 以具名函式方式傳入 (也可以透過匿名函式傳入)

```
function callback(data) {  
    $('<div>.result</div>').html(data);  
}  
get('ajax/test.html', callback);
```

全域變數 vs. 區域變數

- 全域變數
 - 在全域範圍宣告的變數，通稱為「**全域變數**」(Global Variables)
- 區域變數
 - 在有限範圍內宣告的變數，通稱為「**區域變數**」(Local Variables)
- 不是變數
 - 不用 `var` / `let` / `const` 宣告的變數，通通都不能算是變數
- 全域屬性
 - 所有置於「**根物件**」下的屬性，通稱為「**全域屬性**」
- 區域屬性
 - 沒有這個玩意

忘記使用 `var` / `let` / `const` 宣告變數

- 思考以下程式碼的執行過程，感受程式碼的壞味道！

```
function test(a, b) {  
    c = a + b;  
    return c;  
}
```

```
var d = test(1, 2);
```

分散 var 變數宣告的問題: **Hoisting** (提升)

- 思考以下程式碼的執行過程，感受程式碼的壞味道！

```
var tmp = 'React';  
function Choose() {  
    console.log(tmp);  
    var tmp = 'Angular';  
    console.log(tmp);  
}
```

- JS 允許在任何位置使用 var 宣告變數，且允許重複宣告
- 執行時期所有用 var 宣告的變數，都會自動提升至範圍第一行

分散 `let` 變數宣告的問題: **Hoisting** (提升)

- 思考以下程式碼的執行過程，感受程式碼的壞味道！

```
var tmp = 'React';  
function Choose() {  
    console.log(tmp);  
    let tmp = 'Angular';  
    console.log(tmp);  
}
```

- JS 不允許在用 `let` 宣告變數前使用該變數，且不允許重複宣告
- 執行時期所有用 `let` 宣告的變數，都會自動提升至範圍第一行

閉包 (Closure)

- 觀念回顧：Function 的兩大特色
 - JS 的一級物件 (First-class object)
 - 提供了變數的作用域 (scope)
- 閉包的特性
 - 在 內層函式 中存取 外層函式 的 變數 (不同的範圍)
 - 在 外層函式 宣告的變數可以在 內層函式 中取用
 - 重點觀念：作用域鏈結 (Scope Chain)
- 閉包的程式碼外觀
 - 運用在巢狀函式的定義中

ECMAScript 2015 Functions

ES6 新增的函式特性



箭頭函式 (Arrow Functions) (=>)

- 無參數、有回傳值

傳統函式	箭頭函式
<code>var f = function() { return 42; }</code>	<code>var f = () => 42;</code>
<code>var f = function() { return 42; }</code>	<code>var f = () => { return 42; }</code>

- 無參數、無回傳值

傳統函式	箭頭函式
<code>var f = function() { run(); }</code>	<code>var f = () => void run();</code>
<code>var f = function() { run(); }</code>	<code>var f = () => { run(); }</code>

箭頭函式 (Arrow Functions) (=>)

- 有參數、有回傳值

傳統函式	箭頭函式
<code>var f = function(p) { return p; }</code>	<code>var f = p => p;</code>
<code>var f = function(p) { return p; }</code>	<code>var f = (p) => p;</code>
<code>var f = function(p) { return p; }</code>	<code>var f = (p) => { return p; }</code>
<code>var f = function(a, b) { return a + b; }</code>	<code>var f = (a, b) => a + b;</code>
<code>[1, 2, 3].map(function (v) { return v * v; });</code>	<code>[1, 2, 3].map(v => v * v);</code>
<code>[1, 2, 3].filter(function(v) { return v % 2 == 1; });</code>	<code>[1, 2, 3].filter(v => v % 2 == 0)</code>

箭頭函式 (Arrow Functions) (=>)

- 有參數、無回傳值

傳統函式	箭頭函式
<code>var f = function(p) { run(); }</code>	<code>var f = p => void run();</code>
<code>var f = function(p) { run(); }</code>	<code>var f = p => { run(); };</code>

- 回傳物件實字

箭頭函式
<code>var f = () => ({ a: 1, b: 2 });</code>
<code>var f = () => { return { a: 1, b: 2 } };</code>
<code>var f = () => [1,2,3];</code>
<code>var f = () => /^[A-Z][12]\d{8}\$/i;</code>

箭頭函式重點觀念提醒

- 關於箭頭函式的重要觀念
 - 沒有**函式宣告式**，只有**函式表達式** (arrow function expression)
 - 箭頭函式沒有自己的 [this](#), [arguments](#), [super](#) 或 [new.target](#) 可用

簡單來說：箭頭函式並**不是**一個完整的函式物件！

- 使用箭頭函式有以下限制
 - 不能當成 **建構式** 來用、也沒有 **prototype** 屬性
 - 不能當成 **物件** 或 **類別** 中的 **方法(methods)** 使用
 - 不能使用 **arguments** 變數
 - 不能使用 **yield** 命令 (不能當作 [Generator](#) 函式使用)
 - 函式內的 **this** 是包含箭頭函式的那個函式所建立的物件實體

預設參數 (Default Parameters)

- 傳統 ES5 函式寫法 (1)

```
function f(a, b) {  
    if (b === undefined) { b = 0.7; }  
    return +(a * b).toFixed(2);  
}
```

- 傳統 ES5 函式寫法 (2)

```
function f(a, b) {  
    b = b || 0.7;  
    return +(a * b).toFixed(2);  
}
```

預設參數 (Default Parameters)

- 現代 ES6+ 函式寫法 (1)

```
function f(a, b = 0.7) {  
    return +(a * b).toFixed(2);  
}
```

- 現代 ES6+ 函式寫法 (2)

```
var f = (a, b = 0.7) => +(a * b).toFixed(2);
```

- 預設參數值可設定表達式，呼叫時才會執行

```
function f(a, b = (x + 0.7)) {  
    return +(a * b).toFixed(2);  
}
```


其餘參數 (Rest parameters) (...)

- 傳統 ES5 函式寫法 (這裡的 arguments 並非陣列型別)

```
function f(a, b) {  
    var args = Array.prototype.slice.call(arguments, f.length);  
    return args;  
}  
f(1,2,3,4,5,6); // [3, 4, 5, 6]
```

- 現代 ES6+ 函式寫法

```
function f(a, b, ...args){  
    return args;  
}  
f(1,2,3,4,5,6); // [3, 4, 5, 6]
```

其餘參數的應用技巧

- 不限參數數量的函式

```
function sum(...args) {  
    return args.reduce((previous, current) => {  
        return previous + current;  
    });  
}
```

- 重要觀念提醒

- 透過 **其餘參數** (...) 取得的物件是一個**陣列物件**型別 (Array)
- 傳統函式的 `arguments` 變數並不是一個真的陣列 (不能用陣列API)

ES6 簡化物件中方法的宣告方式

早期 ES5 在物件中宣告方法的方式

```
var obj = {  
  foo: function() {  
    /* code */  
  },  
  bar: function() {  
    /* code */  
  }  
};
```

現在 ES6 在物件中宣告方法的方式

```
var obj = {  
  foo() {  
    /* code */  
  },  
  bar() {  
    /* code */  
  }  
};
```

More ES2015 New Language Features

更多 ES2015 / ES6 全新語言特性



展開運算子 (Spread operator)

- 主要用途
 - 讓一個 **可迭代** (iterable) 的 **類陣列** (array-like) 物件，能夠展開成為一個或多個**參數列**或**陣列項目**。
- 應用情境
 - 簡化函式呼叫語法 (當有多個不固定參數時相當方便)
 - 簡化陣列串接語法
 - 開發 Immutable Array 時相當實用
 - 此技巧在 Vue, React, Angular 等 SPA 框架中常用

展開運算子情境一

- 宣告一個函式

```
function myFunction(x, y, z) { }
```

- 宣告一個陣列

```
var args = [0, 1, 2];
```

- 如何將 `args` 陣列元素傳入 `myFunction` 的 `x`, `y`, `z` 參數？

- 傳統 ES5 解法

```
myFunction.apply(null, args);
```

- 使用 ES6 展開運算子

- `myFunction(...args);`

展開運算子情境二

- 宣告兩個陣列

```
var arr1 = [0, 1, 2];
```

```
var arr2 = [3, 4, 5];
```

- 請問該如何把這兩個陣列合併成一個？

- 傳統 ES5 解法

```
Array.prototype.concat.apply(arr1, arr2);
```

- 使用 ES6 展開運算子

```
[...arr1, ...arr2]
```

```
arr1.push(...arr2);
```

展開運算子情境三

- 宣告一個陣列

```
var arr1 = [0, 1, 2];
```

- 透過 Array 的 push 方法新增元素

```
arr1.push(3)
```

- 透過 Immutable 的方式新增陣列元素

```
arr1 = [...arr1, 3];
```


更多展開運算子範例

- 更彈性的函式參數傳入

```
function myFunction(v, w, x, y, z) { }  
var args = [0, 1];  
myFunction(-1, ...args, 2, ...[3]); // "展開陣列"
```

- 更強大的陣列表示法

```
var parts = ['shoulders', 'knees'];  
var lyrics = ['head', ...parts, 'and', 'toes'];  
// ["head", "shoulders", "knees", "and", "toes"]
```

展開運算子的重要觀念

- 只有實作 **迭代器** (Iterators) 的物件，才能使用展開運算子！
 - 預設所有**類陣列物件**都可以使用 (String, Array, Map, Set, arguments)
 - 預設所有一般物件都**不能用**，除非有實作**迭代器屬性**！
- 實作迭代器屬性的方式 ([Iterators and generators](#))

```
window[Symbol.iterator] = function*() {  
    for (let i in window) {  
        yield window[i];  
    }  
};  
[...window]
```

實作**迭代器**就能用 ES6 的 for..of 跑迴圈

```
let list = [4, 5, 6];
```

```
for (let i in list) {  
    console.log(i); // "0", "1", "2"  
}
```

```
for (let i of list) {  
    console.log(i); // "4", "5", "6"  
}
```

陣列的解構賦值 (Destructuring assignment)

- 陣列的解構賦值

```
var a, b, rest; [a, b] = [10, 20];
```

- 陣列外加展開運算子的解構賦值

```
[a, b, ...rest] = [10, 20, 30, 40, 50];
```

- 使用展開運算子與忽略某些解構賦值

```
[a, , b, ...rest] = [10, 20, 30, 40, 50];
```

- 變數交換

```
var a = 1; var b = 3; [a, b] = [b, a];
```

函式搭配陣列的解構賦值

- 解析自函式回傳的陣列

```
function f() {  
    return [1, 2];  
}  
var a, b; [a, b] = f();
```

- 解構賦值同時宣告變數 + 忽略某些回傳值

```
function f() {  
    return [1, 2, 3];  
}  
var [a, , b] = f();
```

物件的解構賦值

- 物件的解構賦值

```
var o = { p: 42, q: true };  
var { p, q } = o;
```

- 無宣告指派 (前後必須加上小括號)

```
var a, b;  
( { a, b } = { a: 1, b: 2 } );
```

- 預設值

```
var { a = 10, b = 5 } = { a: 3 };
```

解構賦值同時宣告新變數

- 指派到新的變數

```
var o = { p: 42, q: true };
```

```
var { p: foo, q: bar } = o;
```

- 指派到新的變數並設定預設值

```
let o = { p: 42, q: true };
```

```
let { p: foo = 1, q: bar = 2 } = o;
```

函式搭配物件的解構賦值

// 在參數列使用解構賦值最好可以給一個空物件當預設值（避免例外）

```
function f({size = 'big', cords = {x: 0, y: 0}} = {}) {  
    console.log(size, cords);  
}
```

// 從函式使用者的角度來看，使用物件解構賦值的可讀性極高

```
f({  
    cords: {x: 18, y: 30}  
});
```


函式 (function) 與 建構式 (constructor)

- JavaScript 在 ES2015 之前都沒有 class
 - 代表你不用事先建立藍圖，就能建立物件！
- JavaScript 透過「建構式」建立物件藍圖
 - **建構式**就是**函式**，又稱**建構式函式**

```
var Car = function (name) {  
    this.name = name;  
    this.slogan = function () {  
        return 'Driven by technology. ' + this.name + '.';  
    }  
}  
  
var o = new Car('Tesla');
```

傳統 ES5 之前的物件繼承的寫法

// 宣告上層物件藍圖

```
function Mammals() { }
```

```
Mammals.prototype.getName = function() { return this.name; }
```

// 宣告下層物件藍圖

```
function Cat(name = 'kitty') {
```

```
    this.name = name;
```

```
}
```

// 建立物件繼承關係

```
Cat.prototype = new Mammals();
```

// 鏈結上層物件

```
Cat.prototype.constructor = Cat;
```

// 重設建構式為自己

// 建立物件

```
var obj = new Cat('Cookie');
```

現代 ES6 的物件繼承的寫法 (使用類別語法)

// 宣告上層物件藍圖

```
class Mammals {  
    constructor() { }  
    getName() { return this.name; }  
}
```

// 宣告下層物件藍圖

```
class Cat extends Mammals {  
    constructor(name = 'kitty') {  
        super();  
        this.name = name;  
    }  
}  
  
var obj = new Cat('Cookie');
```

類別 (Classes)

- 重要觀念
 - JavaScript 是一種**以原形為主** (Prototype-based) 的物件導向**架構**
 - 從 ES2015 開始新增 **以類別為主的** (Class-based) 的物件導向**語法**
- 主要目的
 - 提供一套 **語法糖** (syntactical sugar) 簡化物件導向程式設計方式

關於類別的基本語法結構

```
class Greeter {  
    constructor(name, title = 'Mr.') {  
        this.name = name;  
        this.title = title;  
    }  
    greet() {  
        return `Hello, ${this.title} ${this.name}`;  
    }  
}  
  
let greeter = new Greeter("world");
```

類別屬性的 get 與 set 存取子 (Accessors)

```
class Rectangle {  
    constructor(height, width) {                // Constructor  
        this.height = height;    this.width = width;  
    }  
    get area() {                                // Getter  
        return this.width * this.height;  
    }  
    set area(height, width) {                  // Setter  
        this.height = height;    this.width = width;  
    }  
}  
  
const square = new Rectangle(10, 10);  
console.log(square.area); // 100
```

物件屬性的 get 與 set 存取子 (Accessors)

```
var square = {  
  _width: 10,  
  
  get area() {                                // Getter  
    return this._width ** 2;  
  },  
  
  set width(value) {                          // Setter  
    if(value < 0) { throw new Error('ERR'); }  
    this._width = value;  
  }  
};
```

認識 ES2015 模組化技術

- ES2015 標準定義的模組技術簡稱 ESM (ES Modules)
- 重要觀念
 - 每個 JavaScript 檔案都是一個「模組」(module)
 - 每個「模組」都會自動形成一個「作用域範圍」(module scope)
- 主要優點
 - 可將變數限制在一個獨立的 JavaScript 檔案內 (模組內)
 - 變數的可見範圍可透過 [export](#) 語法「匯出」給其他模組使用
 - 不同的模組之間可透過 [import](#) 語法將模組內的變數「匯入」使用
- [網站如何開始使用 ES6 / ES2015 模組化技術進行前端開發](#)

產生「變數」的時間點

- 使用 **var** 宣告變數的時候
`var a = 1;`
- 使用 **let** 宣告變數的時候
`let a = 1;`
- 使用 **const** 宣告變數的時候
`const a = 1;`
- 使用 **function** 宣告函式的時候(如同 **var** 變數)
`function f() { }`
`var f = function () { }`
- 使用 **class** 宣告類別的時候(如同 **let** 變數)
`class Rect { }`
`let Rect = class { }`

ESM 範例專案

- VanillaJS + Parcel
 - <https://github.com/duotify/todomvc-vanilla-es6-parcel>
- Angular 7 (webpack)
`npm install -g @angular/cli`
`ng new demo1 --routing --style css`
- Vue 3
`npm install -g @vue/cli`
`vue create hello-world -d`

匯出模組變數

- 具名匯出

```
function f() { }  
export { f };  
export var foo = Math.sqrt(2);  
export let name = 'Will H';  
export const PI = Math.PI;
```

- 預設匯出 (一個模組只能有一個預設匯出)

```
export default function() {}  
export default class {}  
export { name as default }
```

匯入模組變數

- 基本語法

```
import { 變數名稱 } from '模組名稱或路徑';
```

宣告在目前模組使用的變數名稱

- 更多範例

```
import varname from 'module-name'; // 取得預設匯出物件
import * as varname from 'module-name'; // 取得所有匯出
import { exportName1, exportName2 } from 'module-name';
import { exportName as varname } from 'module-name';
import varname, { exportName } from 'module-name';
import 'module-name'; // side effects only (ref)
```

聯絡資訊

- The Will Will Web

記載著 Will 在網路世界的學習心得與技術分享

– <http://blog.miniasp.com/>

- Will 保哥的技術交流中心 (臉書粉絲專頁)

– <http://www.facebook.com/will.fans>

- Will 保哥的推特

– https://twitter.com/Will_Huang